Theoretical Algorithms in C++

By Kevin De Keyser – 2015 – version 0.1a

Feel free to improve any of these algorithms and mail them to:
sup@teegabel.com

Algorithms

0. Introduction / Explanation
1. Basics
    a. Arithmetics
        i. C++ Operators
        ii. Boolean arithmetic's
        iii. Bit tricks.
    b. Runtime analysis
        i. Bachmann-Landau notations
    c. Algorithm design paradigms
        i. Iteration & Recursion
        ii. Monte Carlo & Las Vegas
        iii. Divide & Conquer
        iv. Functional programming
2. Data Management
    a. String matching algorithms
        i. Knuth-Morris-Pratt search (KMP)
        ii. Rabin-Karp algorithm
        iii. Z-algorithm
    b. Median algorithm
    c. Sorting algorithms
        i. Selection sort
        ii. Bubble sort
        iii. Insertion sort
        iv. Bogo sort
        v. Quick sort
        vi. Merge sort
        vii. Radix sort
3. Graph Theory
    a. Graph data structures
        i. Adjacency matrix
        ii. Adjacency list
        iii. Adjacency set
        iv. Edge list
        v. Sorted edge list

b. Traverses
    i. BFS
    ii. DFS
    iii. Backtracking
    iv. Brute Force
    v. Heuristic DFS Traversals
c. Connected components
    i. Strongly connected components
d. Euler trail
    i. Hierholzer algorithm
e. Topological sorting
f. Articulation points
    i. Bridges
g. 2-coloring maximum bipartite matching
    i. bipartite graph
    ii. general graph
h. Stable marriage problem
i. Union Find
j. Minimum Spanning Tree
    i. Kruskal
    ii. Reverse-delete
    iii. Boruvka
    iv. Prim
    v. Tarjans MST Coloring Rules
    vi. Arborescence
k. Shortest-Path problem
    i. Dijkstra algorithm
    ii. All-pairs shortest path problem
l. Maximum Network Flow
m. NP-Hard Problems
    i. Clique
    ii. Hamilton path
    iii. Minimum vertex cover
    iv. Graph isomorphism
    v. Traveler salesman problem
    vi. K-coloring problem

vii. Boolean satisfiability
viii. NP-Complete problems without heuristics

4. Heuristic algorithms
    a. Polynomial-time approximation scheme
    b. Greedy algorithms
    c. TSP (example)
        i. Nearest neighbor algorithm
        ii. Christofides' algorithm
        iii. Ant colony optimization algorithm
    d. Supervised machine learning
    e. Neuronal networks
        i. Perceptron neurons
        ii. Sigmoid neurons
        iii. Recurrent neuronal network
        iv. Matrix representation
        v. Backpropagation algorithm
        vi. Layered NN implementation
        vii. Curse of dimensionality
5. Data Structures
    a. Arrays
        i. Dynamic arrays
    b. Linked Lists
        i. Stack
        ii. Queue (Singly linked list)
        iii. Doubly linked list
    c. Hash maps
    d. Trees
        i. Segment tree
        ii. Quadtree & Octree
        iii. Binary search tree
        iv. Red-black tree
        v. Lowest common ancestor
6. Cryptography
    a. XOR-Cryptography
    b. Diffie-Hellman key exchange
    c. RSA

7. Mathematical Algorithms
    a. Basic
        i. Absolute value function
        ii. Signum function
        iii. Max / min
        iv. Floor / ceil
    b. Binary arithmetics
        i. Addition
        ii. Subtraction
        iii. Multiplication
        iv. Karatsuba multiplication
        v. Division / Modulo
        vi. Conversions
        vii. Log N Higer Order arithmetic's algorithm
        viii. Exponentiation
    c. Arithmetics
        i. N-th root algorithm
        ii. Arithmetic sequence
        iii. Geometric sequence
        iv. Factorial algorithm
        v. Binomial coefficient
        vi. Trigonometric functions
        vii. Sequences
        viii. Stein's Binary GCD algorithm
        ix. Modulo arithmetic algorithms
        x. Prime sieve
        xi. Miller-Rabin prime test
        xii. Prime factorization
        xiii. Derivative
        xiv. Integral
        xv. Newton's method
        xvi. Root algorithm using newton's method
    d. Matrices
        i. Addition & Subtraction
        ii. Scalar multiplication & division
        iii. Transpose

iv. Matrix multiplication
v. Strassen's matrix multiplication
vi. Gaussian elimination
vii. Gaussian elimination using pivoting
viii. General gauss elimination
ix. Gauss-Seidel algorithm
x. Trace
xi. Determinant with La-Place method
xii. Determinatn with Gaussian elimination
xiii. Identity matrix
xiv. Inverse using Gauss
xv. Normalizing
xvi. Lower-upper decomposition
xvii. Lower-upper decomposition with pivoting
xviii. Matrix exponentiation
xix. Matrix division
xx. Vectorial transformations
xxi. Affine transformations
xxii. Eigenvalues

e. CORDIC
   i. Atangent
   ii. Sine, Cosine & Tangent
   iii. Matrix representation
   iv. Hyperbolic functions

f. Complex numbers
   i. Imaginary numbers
   ii. I/O streams
   iii. Addition / subtraction
   iv. Scalar multiplication / division
   v. Complex multiplication / division
   vi. Scalar exponentiation
   vii. Complex norm
   viii. Complex argument
   ix. Euler's formula
   x. Euler's identity
   xi. De Moivre's identity

        xii. Phasor
        xiii. Complex Exponentiation
        xiv. Complex root
    g. Curve fitting
        i. Least squares regression
        ii. Interpolation
        iii. Spline Interpolation
        iv. Least square regression with sinusoids
        v. Continuous Fourier approximation
        vi. Fourier transformation
        vii. Inverse Fourier transformation
        viii. Fast Fourier transformation (Radix-2)
        ix. Chirp-Z transformation
    h. Lindenmayer-system
        i. Koch snowflake
8. Dynamic Programming
    a. Optimal substructure
    b. Shortest path in DAG
    c. Longest increasing subsequence
    d. Generalized subsequence search
    e. Minimum edit distance
    f. Wagner-fisher algorithm
    g. Longest common substring
    h. Longest common subsequence
    i. Matrix chain multiplication
    j. Maximum empty rectangle
    k. Unbound Knapsack
9. Geometry in Informatics
    a. Point / Line representations
    b. Intersections
    c. Determinant
    d. Area of a polygon
    e. Point in polygon (Ray-casting)
    f. Convex hull
        i. Grahams scanline algorithm
        ii. Jarvis March gift wrapping algorithm

g. Circle from 3 points

10. Graphics algorithms

a. Bresenham algorithm

b. Edge detection (Image filtering)

c. Transforming a point

No sources were harmed in the making of this book.

Introduction

These are some C++ code snippets I wrote back when I was 17. I don’t claim correctness of any of these. Thank you.

The Programming Language C++
I created this book for my own reference.
C++ is especially suitable, since it contains the STL-Library:

STL Structure	Java Equivalent	Data Structure
pair<**int**, **int**>	**int** first; **int** second;	**Tuple**
vector<**type**>	ArrayList<**Type**>	**Dynamic Array**
list<**type**>	LinkedList<**Type**>	**Doubly Linked List**
slist<**type**>	-	**Singly Linked List**
queue<**type**>	Queue<**Type**>	**Queue**
stack<**type**>	Stack<**Type**>	**Stack**
priority_queue<**type**>	PriorityQueue<**Type**>	**Priority Queue**
set<**type**>	SortedSet<**Type**>	**Sorted Set (usually with TreeSet log n)**
map<**type, type**>	TreeMap<**Type, Type**>	**Map with log n insertion**
unordered_map<**type, type, hash function**>	HashMap<**Type, Type**>	**Map with constant time insertion**

Layout

Most algorithms are cramped into one page. Some of them have the name of the creator and the run-time complexity written under the title. A few of them also have the memory usage noted.

Collatz Algorithm
Inventor: Lothar Collatz (1937)
Complexity: Unkown

The Collatz algorithm is a recursive algorithm, which returns the list of the Collatz conjecture. The Collatz conjecture says if you recursively input a given positive number to the following algorithm: If n is dividable by 2 without remainder, divide it by 2. Elsewise calculate n := 3 * n + 1. Now if you recalculate the output the conjecture proclaims it will end in a loop with 1.

Implementation note: This is C++ code. Lorem ipsum.

```
int collatzSequence(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else if (n % 2 == 0) return collatzSequence(n / 2);
    else return collatzSequence(3*n + 1);
}
```

Basics

Operators

Instead of functions, programming languages also use operators. Operators act like functions, but usually compact the code even more, though with the disadvantage of having an order of operations.
Usually there are 3 types of operators, the suffix
The following operations are listed by their precedence in C++.
A and b stand for expression. (You can think of it like a number or Boolean).

Operators	*Operators as functions*	*Operators by type*
a++ a--	++(a) --(a)	Suffix (After expression)
++a --a +a –a !a ~a	++(a) --(a) +(a) –(a) !(a) ~(a)	Prefix (Before expression, also: unary)
a*b a/b a%b	*(a,b) /(a,b) %(a,b)	Infix (multiplicative)
a+b a-b	+(a,b) –(a,b)	Infix (additive)
a<<b a>>b a>>>b	>>(a,b) <<(a,b) >>>(a,b)	Infix (shifts)
a<b a<=b a>b a>=b	<(a,b) <=(a,b) >(a,b) >=(a,b)	Infix (relational)
a==b a!=b	==(a,b) !=(a,b)	Infix (equality)
a&b	&(a,b)	Infix (bitwise AND)
a^b	^(a,b)	Infix (bitwise XOR)
a|b	|(a,b)	Infix (bitwise OR)
a&&b	&&(a,b)	Infix (logical AND)
a||b	||(a,b)	Infix (logical OR)
a ? b : c	If(a) {b} else {c}	Ternary Operators
a=b a+=b -= *= /= %= &= ^= |= <<= >>= >>>=	a=b a=+(a,b) .. etc.	Assignment Operators

Bit Operators

In most programming languages the Integers (whole numbers) are a set of bytes ordered in the power of 2.

- / +	2^{31}	…	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0 if + 1 if -	2147483648	…	64	32	16	8	4	2	1

These operators can only be reliantly performed on variables with fixed length: boolean, int, long, char → it won't work on floats, texts, other abstract data structures.
~ invert-operator. This operator inverts every single bit inside a data type.
& = and–operator: The and operator is an infix. With integer it uses the and operator on 2 integers to create a new one.
| = or – operator:
^= xor – operator:
<< = shift operator. This shifts the bits by n places to the left.
>> = shift operator. This shifts the bits by n places to the right.

Arithmetic Operators

Operators	Commutative a+b = b+a	Associative (a+b)+c = a+(b+c)	Identity Element
+ Addition	yes	yes	0
- Subtraction	no	no	0
*Multiplication	yes	yes	1
/ Division	no	no	1
% Modulo	no	no	∞

A ° B == B ° A Commutativity : a + b = b + a
(A ° B) ° C == A ° (B ° C) Associativity : a * (b * c) = (a * b) *c
A ° (B ° C) == (A°B) ° (A°C) Left-Distributivity : a*(b+c)=(a*b)+(a*c)
(A ° B) ° C == (A°C) ° (B°C) Right-Distributivity: (a+b)*c=(a*c)+(b*x)

Bittricks

These tricks will work on all data types which have a set length of bits: boolean, int, short, char, but not with: float, string, bigint

Check if the n^{th} place (counted from the left side) in a bit sequence is a 1 or a zero.

```
if ((x & (1 << n)) != 0 ) //The nth bit is a 1.
```

Check if the integers have the same prefix (- or +)

```
if(x ^ y < 0) //Both x and y share the same prefix
```

Turn off the bit at the n^{th} location.

```
x = x & ~(1 << n)
```

`(1 << n) == pow(2, n) ==` 2^n (Which we regard as O(1))

`(x >> n) == x / pow(2, n)` (Which we also regard as O(1))

`Integer.MIN_VALUE = -Integer.MAX_VALUE - 1`

`-n == ~n + 1` (Change prefix)

Swapping 2 Variables without using a temporary variable! This works for all set sized data structures.

```
x = x ^ y;
y = y ^ x;
x = y ^ x;
```

If you use BigIntegers or floats you can use this method instead:

```
x = x + y;
y = x - y;
x = x - y;
```

Or even:

```
x = x * y;
y = x / y;
x = x / y;
```

In fact you can do this with pretty much every pair function:

x = x^y

y = $\sqrt[y]{x}$

x = $\sqrt[y]{x}$

Boolean rules

A Boolean is basically a one bit data type (either true or false).

Double Negation rule

!!a == a

!!!a == !a

Commutativity

a & b == b & a

a | b == b | a

Associativity

a & (b & c) == (a & b) & c

a | (b | c) == (a | b) | c

De Morgan's law

!(a & b) == (!a) | (!b)

!(a | b) == (!a) & (!b)

Distributivity

a | (b & c) == (a | b) & (a | c)

a & (b | c) == (a & b) | (a & c)

Absorption

a & (a | b) == a

a | (a & b) == a

Tautologies & Contradictions

a & a = Tautology

a | a = Tautology

a & !a = Contradiction

a | !a = Tautology

Every gate can be written with any other gate plus the not gate.

a & b == !(!a | !b) (AND)

a | b == !(!a & !b) (OR)

a ^ b == (a | b) & !(a & b) == (a & b) | !(a & b) (XOR)

When doing comparisons with Booleans make sure to always use && or || instead of & or |, because the program will stop if the first condition creates a tautology or contradiction:

if(false && true) or if(true || false): both only need to check the first comparison value.

&& == &

|| == |

= != == → Don't get confused here. = is used to set a value != and == are comparisons.

Big-O notation (Bachmann-Landau notation)
To understand the Big-O notation we should use the amortized analysis to get the limiting factor inside an algorithm.

For example we want to calculate:
$f(n) = 40 * n^3 + 20 * n^2 + 17$

N =	0	1	2	10	100	10000
F(n)	17	77	417	42017	40200017	40002000000017
$40 * n^3 + 20*n^2$	0	60	400	42000	40200000	40002000000000
$40 * n^3$	0	40	320	40000	40000000	40000000000000
n^3	0	1	8	1000	1000000	1000000000000

The bigger n gets, the more we see the insignificance of the other bits.
The Big-O notation describes the worst-case scenario, which is why you can remove all smaller parameters of an addition/subtraction and can remove constant factors.
$O(40 * N^3 + 20 * N^2 + 17) = O(N^3)$

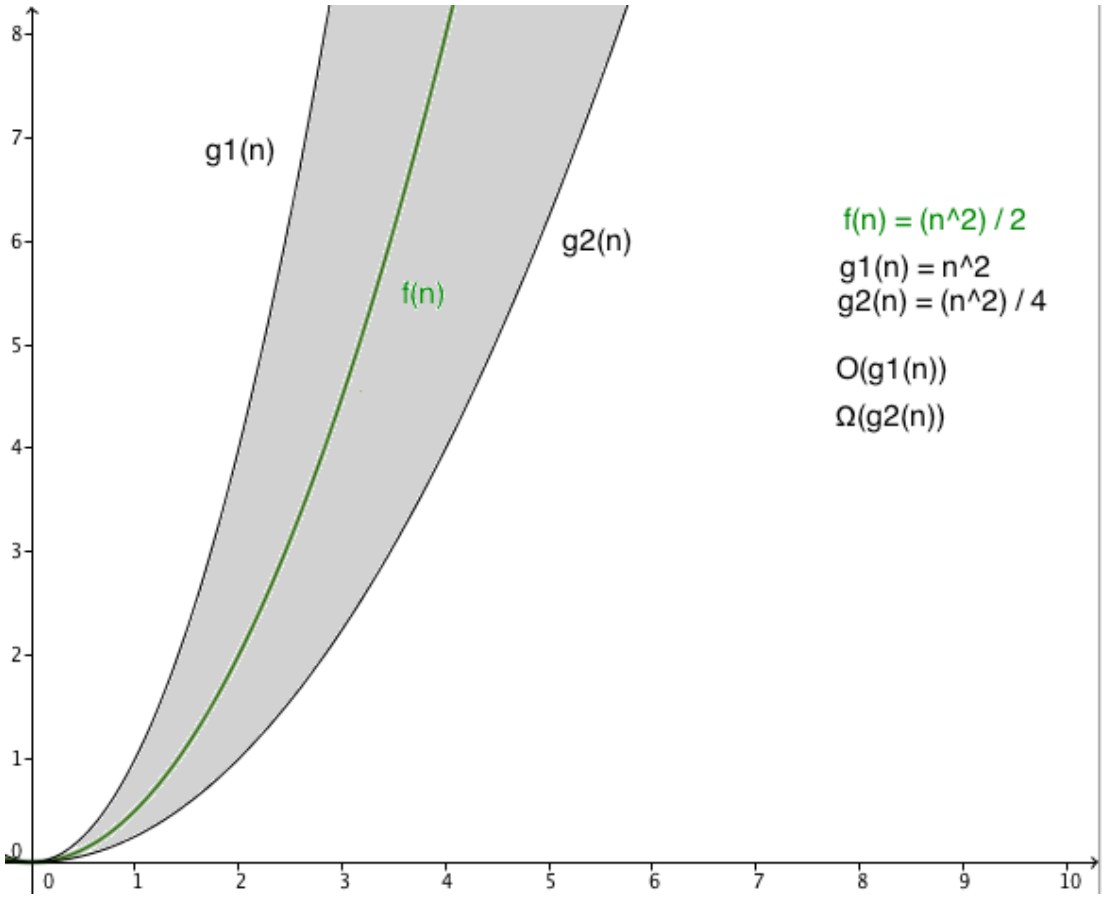
Occurrences:

Big-O Notation of N	Term	Example
O(1)	Constant time	Accessing memory in an array.
O(log N)	Logarithmic time	Binary search
O(N)	Linear time	KMP-Search
O(N log N) = O(log N!)	Linearithmic time	Merge sort
$O(N^2)$	Quardratic time	Basic multiplication
$O(N^k)$	Polynomial time	All of the above
$O(k^N)$	Exponential time	Hamilton path
$O(N!) < O(N^N)$	Factorial time	Traveling salesman

Example:
Now let's say we have a sentence where the words get sorted with bubble sort and then using linear search trying to find a word. After the analysis it turns out we have the following complexity: $n^2 + n$

If the program should be faster we can optimize the linear search to binary search as many times as we want, but the Big-O notation still stays at $O(n^2)$. If we instead replace the bubble sort to merge sort we are getting a new complexity: n * log(n) + n, which is Big-O notation of:
O(n * log(n))

Big-O notation & Big-Ω notation



Graph: 1.01 – Big O Notation O(f(n))

Correct Definitions:
These notations are correct if there exist positive constants c_1, c_2, n_0 such that:

$f(n) \in O(g(n)) : 0 \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$

$f(n) \in \Omega(g(n)) : 0 \leq c_1 * g(n) \leq f(n)$ for all $n \geq n_0$

$f(n) \in \Theta(g(n)) : 0 \leq c_1 * g(n) \leq f(n) \leqslant c_2 * g(n)$ for all $n \geq n_0$

Theorem:
$f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

A few other notations:
$f(n) \in o(n) : 0 \leq f(n) < c_2 * g(n)$ for all $n \geq n_0$
$f(n) \in \omega(n) : 0 \leq c_1 * g(n) < f(n)$ for all $n \geq n_0$

$\Theta(f) \subseteq O(f)$ $\Theta(f) \subseteq \Omega(f)$ $o(f) \subseteq O(f)$ $\omega(f) \subseteq \Omega(f)$
There also exist soft-notations: $\tilde{O}(n)$, $\tilde{o}(n)$, etc. which are the previous notations defined without logarithmic factors, example:
$f(n) \in \tilde{O}(g(n)) : 0 \leq f(n) \leq c_2 * g(n) * \lg^k(n)$ for all $n \geq n_0$

Iterative & Recursive

The decision to use either the iterative or recursive programming paradigm is a classical problem in Divide & Conquer and its effects on amortized running-time is discussed in detail in the Divide & Conquer chapter.

Most mathematical functions are defined recursively, because they usually compact the information more tightly then iterative functions. For example:

$fibonacci(N) = fibonacci(N-1) + fibonacci(N-2)$

$factorial(N) = N * factorial(N-1)$

For both if N = 0 or 1 return 1.

$$ackermann(m,n)\begin{cases} n+1 & if\ m=0 \\ ack(m-1,1) & if\ m>0\ and\ n=0 \\ ack(m-1,ack(m,n-1)) & f\ m>0\ and\ n>0 \end{cases}$$

or the previously discussed collatz function.

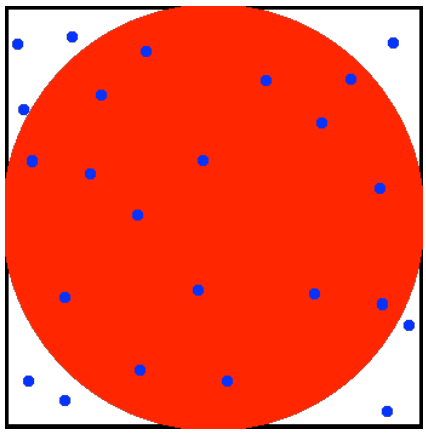
However recursive programming can lead to an increase in amortized running-time. Also most programming languages have a stack limit, which is a limit on how many times you can call a function, within a function. Especially in DFS-searches this limit is quickly reached. Every Recursive function can however be rewritten as an iterative function, usually with a stack data structure.

```
int collatzRecursive(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else if (n % 2 == 0) return collatzRecursive(n / 2);
    else return collatzRecursive(3*n + 1);
}
void collatzIterative(int n) {
    stack<int> recursion;
    recursion.push(n);
    while (!recursion.empty()) {
        int element = recursion.top();
        recursion.pop();
        cout << element << endl;
        //recursive code
        if(element == 1) break;
        else if (element % 2 == 0) recursion.push(element / 2);
        else recursion.push(3 * element + 1);
    }
}
```

Monte Carlo & Las Vegas Algorithms

Las Vegas algorithms are our typical algorithms, algorithms that will return the result to the problem with a 100% guarantee. Monte Carlo algorithms return a result with a certain percentage of it being the true answer. Using more computing power will increase this percentage usually.

To show the difference this book will introduce 2 ways how to calculate Pi.
The Monte Carlo Pi is an algorithm, which shoots different points onto a square upon which a circle lies. Now we can check if the radius is inside the square by placing the circle in the centre of the coordinate system and then calculating the distance of the blue dots and check if they are smaller then the radius (1).



$$\frac{r^2 * \pi}{(2 * r)^2} = \frac{points\ which\ hit\ the\ circle}{total\ used\ points}$$

Another way how to calculate Pi with an increasing precision is the Gregory-Leibnitz series.

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} \ldots.$$

A Las Vegas approach to these algorithms is for e.g. a spiggot algorithm, which is relatively slower then heuristic iterative algorithms, but can calculate the n-th digit of Pi (in binary).

Bellards Formula for the n-th base 2 digit of pi:

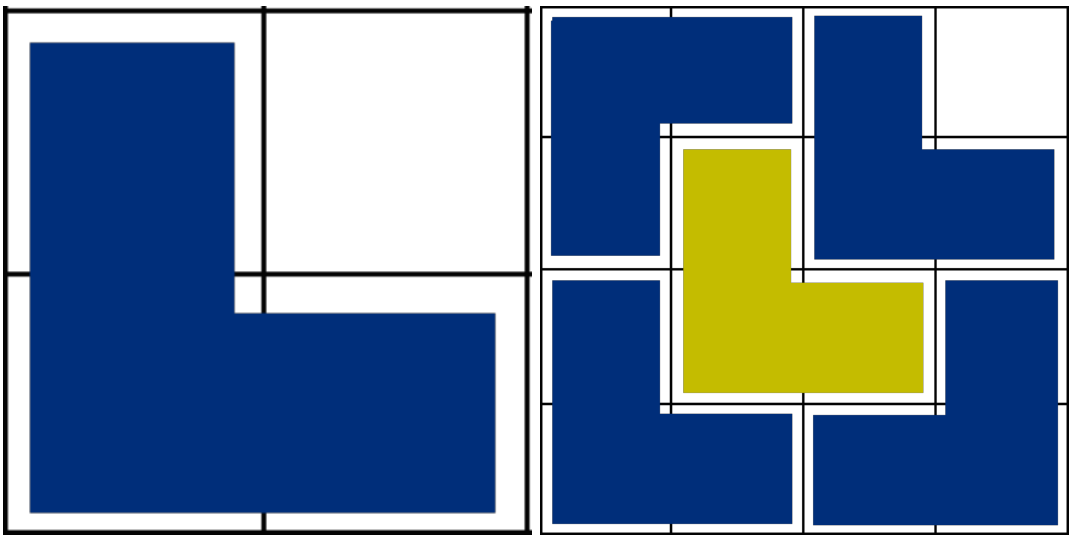
$$\pi = \frac{1}{2^6} * \frac{(-1)^n}{2^{10*n}} * \left(-\frac{2^5}{4*n+1} - \frac{1}{4*n+3} + \frac{2^8}{10*n+1} - \frac{2^6}{10*n+3} - \frac{2^2}{10*n+5} - \frac{2^2}{10*n+7} + \frac{1}{10*n+1} \right)$$

```
float pi()
{
  float precision = 10000000;
  float hitMark = 0;
  for (number i = 0; i < precision; i++)
  {
    float pX = random(-1, 1);
    float pY = random(-1, 1);
    if (sqrt(pX * pX + pY * pY) <= 1)
    {
      hitMark++;
    }
  }
  float pi = (hitMark / precision) * 2 * 2;
  return pi;
}
```

Example 1.01: Monte Carlo Pi

Divide & Conquer

Divide & Conquer is an approach to problems in informatics, which divides a bigger problem into smaller sub-problems. If you had to fill a $2^N * 2^N$ field with L-shaped pieces and you are only allowed to have one free space. Divide & Conquer first tries to solve this problem as easy as possible, with a 2x2 field:



The next step to go from a 2x2 field into a 4x4 field is to use the previous solution again in some way or the other. In this case, all sub 2x2 fields are rotated versions of the original. However you have to come up with a new connected form, which again has the same properties as the previous field, in this case a hole in the top right corner.

Another typical example of Divide & Conquer is the optimal solution to the towers of Hanoi. Divide & Conquer algorithms are often implemented recursively.

Solving the towers with 10 pieces requires the call:

```
hanoi(10, 0, 1, 2);
```

```
void hanoi(int n, int from, int to, int other)
{
   if(n >= 1)
   {
      hanoi(n - 1, from, other, to);
      cout << from << " " << other << endl;
      hanoi(n - 1, other, to, from);
   }
}
```

Universal Turing-Machine

A universal Turing machine is a hypothetical device, which manipulates symbols on an infinitely large strip of tape according to an algorithm. The programming language Brainf*ck very closely resembles the idea of a Turing machine and is Turing complete. A programming language is Turing complete if and only if every computable function can be written within it. Therefor proving that a programming language can emulate a Turing machine makes it Turing complete. Another approach to prove if a language is Turing complete is by proving that every possible program within another Turing complete programming language is creatable within the programming language to be proven.

Halting problem

The halting problem proofs that computer cannot compute everything. Consider a Turing machine A with an input and an output. Its inputs are blueprints to other Turing machines and their inputs. It only has one output, which will decide if the blueprinted Turing machine will continue to run forever or if it will receive a result. Consider another Turing machine X, which gets stuck when the result of A is not stuck and doesn't get stuck when the result of A is stuck. Now input 2 blueprints of machine X into machine X itself. If the submachine A decides X gets stuck, it won't get stuck. If the submachine A decides X doesn't get stuck, it will get stuck. Therefor machine X can't compute everything.

Functional programming
Functional programming is based on the concepts of lambda calculus and not on imperative programming.
Lambda calculus is a programming language with a very simplistic syntax and is Turing complete. It only has one expression and builds up on the other expressions:
<expr> can be a <constant>
<expr> can be a <variable>
<expr> can be a list of 2 expressions (<expr> <expr>)
<expr> can be a lambda expression: (λ <variable> . <expr>)

Or in Backus-Naur notation:
<exp> ::= <constant / variable>
| (<exp> <exp>)
| (λ <var> . <exp>)
The lambda expression can be used as a function.
$f(x) = y$ in lambda calculus is $(\lambda \mathrm{x} \,.\, \mathrm{y})$ For example:
$f(5) = x^2$ in lambda calculus is $(\lambda \mathrm{x} \,.\, \mathrm{x}^2)\ 5$

α-equivalence
Alpha equivalence states, that every variable can be renamed into any other variable, as long as it doesn't yet exist in the equation.
This means: $(\lambda \mathrm{x} \,.\, \mathrm{x}) = (\lambda \mathrm{y} \,.\, \mathrm{y}) = (\lambda \$ \,.\, \$)$

Free variables are defined outside the scope of a lambda body, where *bound variables* are defined inside the scope of a lambda body. X is free: $(\lambda \mathrm{y} \,.\, \mathrm{y})\ x$, X is bound: $(\lambda \mathrm{x} \,.\, \mathrm{x})\ y$

Beta-reduction
$(\lambda \mathrm{x} \,.\, \mathrm{M})\ N \rightsquigarrow M[x := N], so\ \big(\lambda \mathrm{x} \,.\, (\mathrm{M}\ \mathrm{x})\big)\ N = (M\ N)$
Beta reduction allows any bound variable to be replaced by its input variable. Beta reduction makes mathematics possible.

Eta-reduction
$(\lambda \mathrm{x} \,.\, \mathrm{M}) N \rightsquigarrow M\ if\ M\ does\ not\ contain\ x.$
This is almost equivalent to the beta-reduction, but allows M to be freed without having an input expression.

Shorthand notation:
$(\mathrm{a\ b\ c}) = ((a\ b)\ c)$, because of left-associativity.
$\lambda \mathrm{abc.\,f} = \lambda \mathrm{a.}\, \lambda \mathrm{b.}\, \lambda \mathrm{c} \,.\, \mathrm{f} = \lambda \mathrm{a} \,.\, (\lambda \mathrm{b} \,.\, (\lambda \mathrm{c} \,.\, \mathrm{f}))$

Commonly used Combinators=Lambda expressions, functions, which don't require variables to be stored, only evaluated as such.

Name of function	Lambda expression	Classical expression
Constant combinator	$(\lambda x\,.\,c)$	$f(x) = c$
Identity combinator	$(\lambda x\,.\,x)$	$f(x) = x$
Flip combinator	$\lambda x\,.\ (\lambda y\,.\ (y\ x))$	$f(x,y) = (y,x)$
Composition combinator	$\lambda x.\,\lambda y.\,\lambda z\,.\,f\ (g\ x)$	$f((x,y),z) = (x,(y,z))$
Duplication combinator	$\lambda x.\ (x\ x)$	$f(x) = (x,x)$
Infinite Recursion	$(\lambda x\,.\,(x\ x))\ (\lambda x\,.\,(x\ x))$	$\text{while}(f = f)\ \{\ f = f\ \}$
Y-Combinator (Curry's fixed-point combinator)	$\lambda f.\ (\lambda x.\,f\ (x\ x))\ (\lambda x\,.\,f\ (x\ x))$	

In order to remove a suffix in front of a function use $\lambda r.\ (\lambda x.\ (\lambda))$

Fixed-Point

A fixed point is an input x, such that f(x) = x. For example:
$f(x) = x * x = x;\ x = 0\ or\ 1$
Given any expression f as input, the Y-Combinator will find its fixed point: $\lambda f.\ (\lambda x.\,f\ (x\ x))\ (\lambda x\,.\,f\ (x\ x))$

Church encoding

Church numerals are a unary representation of natural numbers within lambda calculus. The variable names don't matter due to α-equivalence, however z is the zero variable and f is the successor variable in this book.

Value	Syntax inbounds	Full lambda expression
0	z	$\lambda f.\,\lambda z.\ (z)$
1	$f\ z$	$\lambda f.\,\lambda z.\ (f\ z)$
2	$f\ (f\ z)$	$\lambda f.\,\lambda z.\,f\ (f\ z)$
3	$f(f(f\ z))$	$\lambda f.\,\lambda z.\,f(f(f\ z))$
N	$f^N\ z$	$\lambda f.\,\lambda z.\ (f^N\ z)$

Church arithmetics:
$q = (n\ f\ z) = f^N\ z$

Operation	Classical	Simplified	Lambda expression
Increment (successor)	Incr($f^N\ z$) = $f^{N+1}\ z$	$\lambda q\ .\ (f\ q)$	$\lambda n.\ \lambda f.\ \lambda z.\ f\ (n\ f\ z)$ $= \lambda nf.\ \lambda z.\ n\ f\ (f\ z)$
Addition	Add($f^M\ z, f^N\ z$) = $f^{N+M}\ z$	$\lambda mf.\ \lambda nf.\ \lambda z.$ $(mf\ (nf\ z))$	$\lambda m.\ \lambda f.\ \lambda z.\ \lambda n.\ \lambda f.\ \lambda z.$ $m\ f\ (n\ f\ z)$
Multiplication	Mul($f^M\ z, f^N\ z$) = $f^{N*M}\ z$	$n\ (add\ m)\ z$	$\lambda m.\ \lambda f.\ \lambda z.\ \lambda n.\ \lambda f.\ \lambda z.$ $(m\ (n\ f)\ z)$
Exponentiation	Pow($f^M\ z, f^N\ z$) = $f^{N^M}\ z$	$n\ (mul\ m)\ f\ z$	$\lambda m.\ \lambda f.\ \lambda z.\ \lambda n.\ \lambda f.\ \lambda z.$ $(n\ m)\ f\ z$
Predecessor	Incr($f^N\ z$) = $f^{\max(0,N-1)}\ z$	$\lambda n.\ \lambda f.\ \lambda z.\ \ n\ \big(\lambda p.\ \lambda q.\ q\ (p\ f)\big)(\lambda\_.\ c)(\lambda x.\ x)$	
Subtraction	Sub($f^M\ z, f^N\ z$) = $\max(0, f^{M-N} z)$	$(n\ pred)\ m$	

The constant combinator is important, because it doesn't matter which input it has, the result is always x. Instead of defining 0 as z it can also be defined as $\lambda z\ .\ (\text{flip const})$

Church Booleans
Church encoding can also implement Boolean arithmetic.

TRUE	$\lambda x.\ \lambda y.\ x$
FALSE	$\lambda x.\ \lambda y.\ y$
NOT - NORMAL	$\lambda p.\ p\ (\lambda a.\ \lambda b.\ b)(\lambda a.\ \lambda b.\ a) = \lambda p.\ p\ \text{false true}$
NOT - APPLICATIVE	$\lambda p.\ \lambda a.\ \lambda b.\ (p\ b\ a)$
AND	$\lambda p.\ \lambda q.\ (p\ q\ p)$
OR	$\lambda p.\ \lambda q.\ (p\ p\ q)$
IS ZERO	$\lambda n.\ (n\ (\lambda x.\ \text{false})\ true)$

SKI-Combinator calculus

SKI is a turing complete sub-language based on lambda calculus. All possible parameters have 2 fields, which makes SKI implementable as a binary tree instead of brackets form.

Its Backus-Naur notation is simply:

<exp> ::= S, K or I
| (exp, exp)

I is the identity function, K is the constant function and S is the substitution function.

The shorthand notation: SKSISS = S(K(S(I(S S))))

S	$\lambda x.\ (\lambda y.\ (\lambda z.\ (x\ z\ (y\ z))))$
K	$\lambda x.\ (\lambda y.\ x)$
I	$\lambda x\ .\ x$

Iota (Turing tarpet)

A Turing tarpet is a language aimed to be as minimal as possible, but still be Turing complete. Iota is a good candidate, because it again is a binary search tree, with contains only one character. In the Backus-Naur notation:

<exp> ::= U
| (exp, exp)

U = $\lambda f.\ ((f\ S)\ K)$

The SKI-combinators can be re-deduced from the Iota combinator:

S	$(U(U(U(U\ U))))$
K	$(U(U(U\ U)))$
I	$(U\ U)$

Chaitin's constant

In order to simplify the Iota language even more, simply enumerate every possible Iota program:

0	$\mathrm{U\ U}$
1	$\mathrm{U\ (U\ U)}$
2	$\mathrm{(U\ U)\ U}$
3	$\mathrm{U(U(U\ U))}$
4	$\mathrm{U((U\ U)U)}$
5	$\mathrm{\big(U(U\ U)\big)\ U}$
6	$\mathrm{\big((U\ U)U\big)U}$
7	$\mathrm{((U\ U)\ (U\ U))}$

Chaitin's constant is the halting probability of any legal program within a programming language. This of course differs from programming language to programming language, but it gets interesting if you theoretically had found the simplest Turing tarpet. Because there are infinitely many programs, the halting probability is basically incomputable, because it needn't to diverge. The halting problem itself is also not decidable: See halting problem.

Data Management

Searches

This page only discusses array searching. Graph searching is a separate subject.

Linear Search

Complexity: O(N)

An unsorted array has to be searched-through naively.

```
int linearSearch(vector<int>& list, int toFind) {
    for(int i = 0; i < list.size(); ++i) {
        if(list[i] == toFind) return i;
    }
    return -1;
}
```

Binary Search

Complexity: O(log N)

Binary search has the task of finding an element in a sorted list. It is similar to phone book searching, where you start in the middle of the sequence and eliminate one half, then go into the next sub-half, etc.

If an unsorted list expects many search queries, it would be wise to sort the data, in order to use binary search.

```
int binarySearch(vector<int>& list, int toFind) {
    int pos = 1;
    int half = list.size() / 2;
    if(toFind < list[0]) return -1;
    if(toFind == list[0]) return 0;
    if(toFind > list[list.size()-1]) return -1;
    while(!(list[pos-1] <= toFind && list[pos] >= toFind)) {
        if(list[pos] < toFind) pos += half;
        else pos -= half;
        half /= 2;
        if(half == 0) half = 1;
    }
    if(list[pos] == toFind) return pos;
    else return -1;
}
```

Pattern Matching Algorithms
Knuth-Morris-Pratt Search
Inventor: D. Knuth, J. Morris, V. Pratt (1977)
Worst-case complexity: O(N + M), where N/M are the length of the strings.
The Knuth-Morris-Pratt search computes a deterministic finite automaton, which finds all loops within the pattern and creates a pattern map. First it creates a DFA of only the pattern, storing the places you have to fall back in case two chars don't match during a matchup. After finding this DFA, it uses almost the same algorithm on the sequence by applying the DFA and not modifying it.

```
vector<int> KMP(string sequence, string pattern) {
   vector<int> DFA(pattern.size() + 1, -1);
   vector<int> matches;

   for(int i = 1; i <= pattern.size(); i++)
   {
      int pos = DFA[i - 1];
      while(pos != -1 && pattern.at(pos) != pattern.at(i-1))pos=DFA[pos];
      DFA[i] = pos + 1;
   }

   int sp = 0; //sequence pointer
   int pp = 0; //pattern pointer
   while(sp < sequence.size())
   {
      while(pp != -1 && (pp == pattern.size() || pattern.at(pp) !=
sequence.at(sp))) pp = DFA[pp];
      pp++;
      sp++;
      if(pp == pattern.size()) matches.push_back(sp - pattern.size());
   }

   return matches;
}
```

Rabin-Karp Algorithm
Inventor: R. Karp, M. Rabin (1987)
Average-case complexity: O(N+M)
Worst-case complexity: O(N*M)
Rabin-Karp algorithm also matches a pattern P on a huge string S the same way the naïve algorithm does by always shifting P by one element further. Rabin-Karp can use any rolling hash function, which has the following property at any point i:
hash(sequence[i+1 .. i+m])= rehash(sequence [i+m], hash(sequence[i .. i+m-1])
Rabin-Karp proposed the following O(1) hash function, where d is the size of the alphabet of the set (256 for ASCII), m the size of pattern P and $h = d^{m-1} \% q$:
hash(seq[i+1 .. i+m]) = d (hash(seq[i .. i+m-1]) – seq[i]*h) + seq[i+m] %q
Computing the hash value of the pattern therefor can be expressed as: $patternHash = \sum_{i=0}^{M-1} 10^{M-i} * pattern[i] = P[M-1] + 10 * (P[M-2] + 10 * (\ldots + 10 * P[1]))$
Using modulo allows to store values in a reasonable range, however collisions are possible and this is why every hit might be true and has to be checked in O(M) for spurious hits.

```
vector<int> rabinKarp(string sequence, string pattern) {
   vector<int> output;
   int q = 1000000007; //preferebly a random prime number
   int d = 256; //Size of alphabet
   int h = expModulo(d, pattern.size() - 1, q);
   int patternHash = 0, seqHash = 0;
   for(int i = 0; i < pattern.size(); ++i) {
      patternHash = (d * patternHash + pattern[i]) % q;
      seqHash = (d * seqHash + sequence[i]) % q;
   }
   for (int i = 0; i <= sequence.size() - pattern.size(); ++i) {
      if (patternHash == seqHash) {
         if(sequence.substr(i, pattern.size()) == pattern) {
            output.push_back(i);
         }
      }
      if (i < sequence.size() - pattern.size()) {
         seqHash = (d * (seqHash - sequence[i]*h + q) +
sequence[i+pattern.size()]) % q;
         if(seqHash < 0) seqHash += q;
      }
   }
   return output;}
```

Z-Algorithm

Worst-case complexity: O(n+m)

The Z-algorithm uses a Z-array. Given a string S Z[i] is the length of the longest substring starting from S[i], which is also a prefix of S. For the matching one only has to concat the pattern in front of the text and add a separator.

```
vector<int> zArray(string str) {
    vector<int> z (str.size(), 0);
    int l = 0, r = 0;
    for (int i = 1; i < str.size(); ++i) {
        if (r < i) { //r < i, therefor we have to calculate the matches the
naive way
            l = r = i;
            while (r < str.size() && str[r-i] == str[r]) ++r;
            z[i] = r - i; //size of match
            --r; //make r inclusive
        }
        else {
            //z[i] has to be equal to z[i-l], whenever z[i-l]+i doesn't exceed r.
            if (z[i-l] + i <= r) z[i] = z[i-l];
            else {
                l = i; //because the match is greater than the bounding box
[l,r], r can stay where it is and be extended.
                while (r < str.size() && str[r-i] == str[r]) ++r;
                z[i] = r - i; //size of match
                --r; //make r inclusive
            }
        }
    }
    return z;
}

vector<int> stringMatching(string str, string pattern) {
    vector<int> matches;
    string seperator = "&&&"; //the seperator is a string that does not
appear in the pattern!
    vector<int> arr = zArray(pattern + seperator + str);
    for(int i = 0; i < str.size(); ++i) {
        if(arr[pattern.size() + seperator.size() + i] == pattern.size())
matches.push_back(i);
    }
    return matches;
}
```

Median (Quickselect)
Inventor: Tony Hoare (1961)
Average-case complexity: O(N)
Worst-case complexity: $O(N^2)$
The median is the element which seperates the higher elements from the lower elements in an array exactly half way. For example if you want to know the average salary of any person, calculating the mean wouldn't result in the salary of the average person, because money is not equally distributed (Gaussian bell curve). There is a pivot algorithm, which has a complexity of O(N). More generally quickselect finds the k-th smallest element in O(N). Quicksort is based upon this approach.

```
int partition(vector<float>& input, int l, int r) {
    float pivot = input[r];
    while ( l < r ) {
        while(input[l] < pivot) ++l;
        while(input[r] > pivot) --r;
        if (input[l] == input[r]) ++l;
        else if (l < r) {
            int tmp = input[l];
            input[l] = input[r];
            input[r] = tmp;
        }
    }
    return r;
}

float quickSelect(vector<float>& input, int k, int l, int r)  {
    if (l == r) return input[l];
    int newR = partition(input, l, r);
    int len = newR - l + 1;
    if (len == k) return input[newR];
    else if (k < len) return quickSelect(input, l, newR - 1, k);
    else return quickSelect(input, k - len, newR + 1, r);
}
```

Sorting
Sorting Algorithms need a bit of explanation:

First there are 2 types of sorting algorithms:
- *Comparison sort systems*, which works by using whatever comparison operator needed (this can be any logical gate comparison between a and b).
- *Specific sorting systems,* which have specific uses: Sorting an array of integers or a lexicographical sorting system.

The advantage of *comparison sort systems* is that you can use its comparison part for different kinds of comparisions: For e.g. up and down and the other way around. You could also instead of : if(a < b) do something like: if(abs(a) < b && b != a)
These kind of comparisons are impossible in other systems. The comparison part has been commented on all examples.
Specific sorting systems can have O(n) complexity which is a very fast speed improvement, but it lacks the modifiability.

A sorting algorithm can also be *stable* or *unstable*.
For example you have a list of numbers, which you want to order by increasing order. A *stable* algorithm would not change the order of numbers, which have the same value. This can be useful if you want to sort for example the birthdays of your friends by days per year. Then the previous order would not change, only the day order. So the 1987 birthday of your mom would turn up after the 1990 birthday of hers. An *unstable* algorithm can't guarantee this.

Selection Sort
Complexity: $O(N^2)$
Memory usage: 1
Stable: Yes

```
void selectionSort(vector<int>& toSort) {
    for(int i = 0; i < toSort.size(); ++i) {
        for(int j = i + 1; j < toSort.size(); ++j) {
            if(toSort[j] < toSort[i]) swap(toSort[i], toSort[j]);
        }
    }
}
```

Bubble Sort
Complexity: $O(N^2)$
Memory usage: 1
Stable: Yes

```
void bubbleSort(vector<int>& toSort) {
    for(int i = 0; i < toSort.size() - 1; ++i) {
        for(int j = 0; j < toSort.size() - 1; ++j) {
            if(toSort[j+1] < toSort[j]) swap(toSort[j], toSort[j+1]);
        }
    }
}
```

Insertion Sort
Complexity: $O(N^2)$
Memory usage: 1
Stable: Yes

```
void insertionSort(vector<int>& toSort) {
    for(int i = 0; i < toSort.size(); ++i) {
        for(int j = i - 1; j >= 0; --j) {
            if(toSort[j+1] < toSort[j]) swap(toSort[j], toSort[j+1]);
        }
    }
}
```

A cool thing about insertion sort is that the insertion can be done using binary search, however the shifting (implemented as swapping here), still requires O(N) steps, leading to $O(N^2)$.

Bogo Sort
Average Complexity: O(N * N!)
Worst Complexity: O(∞)
Memory usage: 1
Stable: No

```
bool checkIfSorted(vector<int>& toSort) {
    int previousElement = toSort[0]; //If toSort != null
    for(int i = 1; i < toSort.size(); ++i) {
        if(toSort[i] < previousElement) return false;
        previousElement = toSort[i];
    }
    return true;
}

void bogoSort(vector<int>& toSort) {
    while(checkIfSorted(toSort) == false) {
        for(int i = 0; i < toSort.size(); ++i) {
            int bogo = randInt(i, toSort.size() - 1);
            swap(toSort[i], toSort[bogo]);
        }
    }
}
```

Quicksort
Inventor: Tony Hoare (1961)
Average Complexity: O(N * log N)
Worst Complexity: $O(N^2)$
Stable: No
Memory usage: 1

```
void quickSort(vector<int>& toSort, int first, int last) {
    int left = first;
    int right = last;
    int pivot = toSort[first];
    while (left <= right) {
        while (toSort[left] < pivot) left++;
        while (toSort[right] > pivot) right--;

        if (left <= right) {
            swap(toSort[left], toSort[right]);
            left++; right--;
        }
    }
    if(first < left - 1) quickSort(toSort, 0, left - 1);
    if(last > left) quickSort(toSort, left, last);
}

void quickSort(vector<int>& toSort) {
    quickSort(toSort, 0, toSort.size() - 1);
}
```

Merge Sort
Inventor: John von Neumann (1945)
Worst Complexity: O(N * log N)
Stable: Yes
Memory usage: N

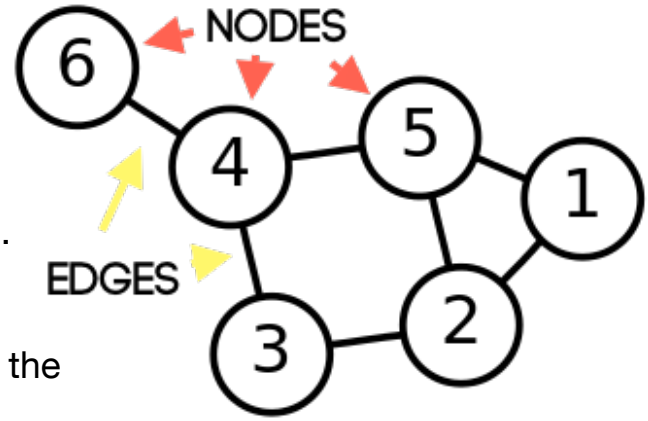
```
void mergeSort(vector<int>& toSort, vector<int>& tempMergArr, int
first, int last) {
    if (first < last) {
        int middle = first + (last - first) / 2;
        mergeSort(toSort, tempMergArr, first, middle);
        mergeSort(toSort, tempMergArr, middle + 1, last);

        for (int i = first; i <= last; i++) tempMergArr[i] = toSort[i];
        int i = first;
        int j = middle + 1;
        int k = first;
        while (i <= middle && j <= last) {
            if (tempMergArr[i] <= tempMergArr[j]) {
                toSort[k] = tempMergArr[i];
                i++;
            } else {
                toSort[k] = tempMergArr[j];
                j++;
            }
            k++;
        }
        while (i <= middle) {
            toSort[k] = tempMergArr[i];
            k++;
            i++;
        }
    }
}


void mergeSort(vector<int>& toSort) {
    vector<int> tempMergArr(toSort.size(), 0);
    mergeSort(toSort, tempMergArr, 0, toSort.size() - 1);
}
```

Radix Sort
Inventor: Herman Hollerith (1887)
Worst Complexity: O(N * R); R = Number of digits of largest number.
Stable: Yes
Memory usage: N
Specific sorting system: Only works for lexicographical ordering.

```
void radixSort(vector<int>& toSort) {
    int radix = 10; //For numbers use 10 in a decimal system. For ASCII
make this 128.

    vector<vector<int> > bucket(radix);
    bool stop = false;
    int tmp = -1;
    int placement = 1;
    while (stop == false) {
        stop = true;
        for (int i = 0; i < toSort.size(); ++i) {
            tmp = toSort[i] / placement; //In binary you could also use a
binary shift here.
            bucket[tmp % radix].push_back(toSort[i]);
            if (stop == true && tmp > 0) stop = false;
        }

        int counter = 0;
        for (int i = 0; i < radix; ++i) {
            for (int j = 0; j < bucket[i].size(); ++j) {
                toSort[counter] = bucket[i][j];
                counter++;
            }
            bucket[i].clear();
        }
        placement *= radix;
    }
}
```

Graph Theory

Graph Theory is the study of *graphs*.
Graphs are structures to show the relation between *nodes* (or vertices). These relations are called *edges*.



The most basic graph only shows if the nodes are connected to each other.

Terms & Notions

Trail: Is an instruction how to walk down a graph. A trial can be 4 -> 3 -> 5 ->4 -> 6
Path: A path is a trail where a node can only be visited once.
Circuit: A closed trail, where beginning and ending node are the same..
Loops: A loop is when an edge connects with the same node twice.
Neighbor: All the nodes, which are directly connected with edges, are neighbors.
Degree: The amount of neighbors an edge has.
Simple Graph: A graph which has at most one edge between two nodes and doesn't have loops.
Connected Graph: Each node can be visited by another node by using any paths you like.
Cycled Graph: Is a graph that has at least 2 paths to go from one node to another, meaning you can take multiple paths. It is the only way to get lost going from one node to another, without back-tracking.
Tree: A tree is a connected graph, where exactly one path exists to go from one node to another. It therefor has no cycles.
Complete Graph: Each node has all of the other edges as neighbors.
Directed Graph: A graph where the edges can point one or the other way. If you want both directions you need to edges, each facing another direction.
Mixed Graph: In this graph some edges are directed and some aren't.
Quiver: A directed graph which can have multiple direct paths between edges.

Graphs which have extra properties:
Weighted Graph: The edges hold values. This can be the distance for a GPS-Street, the cost to build a brige between 2 nodes, etc.
Colored Graph: The nodes hold values. This can be used to store diplomacy between nodes. For example 0 can be allies, 1 can be neutral and 2 can be enemies. Typical question: Go from node A to node B by avoiding enemies.
Directed acyclic graph (DAG): Is a graph which may have cycles, however there may not be a path inside the directed graph which builds a cycle.

Geometric Graphs:
These are a special kind of colored graphs, which hold the node values of their coordinates: x and y (and z) pos. Their weighted edges can be simply calculated by doing:
$Dist\ between\ a\ and\ b = \sqrt{(ax - bx)^2 + (ay - by)^2}$
Dist between a and b (3D)
$= \sqrt{(ax - bx)^2 + (ay - by)^2 + (az - bz)^2}$

Graph Data Structures
Adjacency Matrix
Advantages:
You can simply check if the graph has loops or not.
Easy to implement / use for many applications.
You can use the matrix to either store the amount of edges between the nodes, the weight between the nodes in a weighted graph or both.
The adjacency matrix needn't to be parallel and can have directed graphs stored inside them. It is easy to check if the graph is directed or not.
O(1) to check if an edge exists or not.
O(1) insertion / deletion of an edge.
Disadvantages:
Always uses n^2 memory, even if you don't have n^2 edges.
O(n^2) insertion / removal of a node.

Example	A	B	C	D	E
A	0	0	2	1	1
B	0	1	1	0	0
C	2	1	0	1	2
D	1	0	1	0	1
E	1	0	2	1	0

Adjacency List

Advantages:

Only uses m space, where m is the amount of edges.

The adjacency matrix can be used to store the amount of edges between the nodes or the weight between the nodes in a weighted graph.

O(1) insertion of an edge / $O(n^2)$ to remove a node “if” directed, else O(n)

Supports directed graphs.

Disadvantages:

O(n) to check if an edge exists or not.

Fairly easy to implement.

O(n) to check for undirected graph and to check for loops.

A	D (1)		E(1)	C(2)
B	B(1)		C(1)	
C	A(2)	E(2)	D(1)	B(1)
D	C(1)		A(1)	E(1)
E	C(2)		A(1)	D(1)

Sorted Adjacency List (or Adjacency Set)

Advantages:

Only uses m space, where m is the amount of edges.

log(n) to check if an edge exists or not.

Supports directed graphs.

Disadvantages:

log(n) insertion / removal of an edge O(1) / $O(n^2)$ to remove a node "if" directed, else O(n).

The adjacency matrix can be used to store the amount of edges between the node or the weight of the node. However if you want both you can only sort for one of these values. → log(n) insertion still, but maybe O(n) access.

Hard to implement.

Sorted by edge:

A	C (2)	D(1)	E(1)	
B	B(1)	C(1)		
C	A(2)	B(1)	D(1)	E(2)
D	A(1)	C(1)	E(1)	
E	A(1)	C(2)	D(1)	

Or sorted by weight:

A	C (2)	D(1)	E(1)	
B	B(1)	C(1)		
C	A(2)	E(2)	B(1)	D(1)
D	A(1)	C(1)	E(1)	
E	C(2)	A(1)	D(1)	

Edge List
This data structure is simply a dynamic array, which contains a pair: One for the first value and one for the second. To store a directed edge we simply only store the connection once, connection B being the direction the arrow faces. If we have an edge which goes both directions we store it twice. Loops are also stored twice, however directed loops are only stored once.

Connection A	Connection B	Weight
A	C	2
B	B	1
A	D	1
B	C	1
A	E	1
C	E	2
C	A	2
E	C	2
D	A	1
C	B	1
D	E	1
E	D	1
E	A	1
D	C	1
C	D	1
B	B	1

In C++ you can simply use the following data structure if you don't use a weight:
```
std::vector<pair<int, int>>
```

Insertion: O(1)
Find: O(n)
Deletion: O(n)

Sorted Edge List
This data structure is exactly the same as the edge list with the exception that the bridges are lexicographically sorted, first by their connection A and then by their connection B (and then by their weight).

Connection A	Connection B	Weight
A	C	2
A	D	1
A	E	1
B	B	1
B	B	1
B	C	1
C	A	2
C	B	1
C	D	1
C	E	2
D	A	1
D	C	1
D	E	1
E	A	1
E	C	2
E	D	1

In C++ you can simply use the following data structure if you don't use a weight:
```
std::multiset<pair<int, int>>
```

Insertion: O(log n)
Find: O(log n)
Deletion: O(log n)

Traverses

Different problems require different traverses inside a graph. Programming language like C++ have a recursion limit of calling a function over and over again without heading back. This can lead to a problem and is why the following examples also have data structure implementations.

BFS (Breadth – First Search)

BFS visits all children of the starting node to check if they have the solution. If not it continues with all the grand – children.

```
Node* BFS(Node find) {
    queue<Node*> nodeList;
    nodeList.push(rootNode);
    while(!nodeList.empty()) {
        Node* currentNode = nodeList.back();
        nodeList.pop();
        if(currentNode == find) return currentNode;
        for(Node* child : currentNode->children) {
            nodeList.push(child);
        }
    }
    return null;
}
```

DFS (Depth – First Search): Mostly used in trees or in calculations where you don't need the optimal solution or when there are many optimal solutions. In DFS you try to traverse the tree as deep as possible until the visited node has no more children. Then you are going back to visit the other generations. Note that in an infinite tree (with infinite children), whenever you can use DFS you can use BFS, but not the other way around. BFS is doomed to visit all nodes that DFS has traversed, but DFS doesn't return in an infinite graph and can miss a few solutions.

Recursive implementation:

```
Node* DFS (Node* node, Node& find) {
    if(node == find) return node;
    else {
        for(Node* child : node->children) {
            answer = DFS(child, find);
            if(answer != null) return answer;
        }
        return null;
    }
}
```

Stack Implemenation (Array Implementation):

```
Node* DFS(Node find) {
    stack<Node*> nodeList;
    nodeList.push(rootNode);

    while(!nodeList.empty()) {
        Node* currentNode = nodeList.top();
        nodeList.pop();
        if(currentNode == find) return currentNode;
        for(Node* child : currentNode->children) {
            nodeList.push(child);
        }
    }
    return null;
}
```

Backtracking is some sort of DFS, which can be used outside of trees. If you have cycles inside the graph, you store the values to not endlessly backtrack them. To solve a Sudoku one has to try out one possible possibility on the first empty space. Then move to the next empty field and try a possibility there until we land in a contradiction: A situation where a field can't be stored anymore. Then we back trace to the field causing the problem and try a different number there and continue.

Brute Force: Any form of trying out every possible solution is can be called Brute Force. DFS, BFS & Backtracking are all Brute Force methods. Brute Force doesn't need to be a trace in a graph, but nearly all brute force problems can be reduced to some sort of traversal.

Heuristic DFS Traversals

Now imagine a tree with each 4 possible subtrees. These 4 sub-trees are the movement to the west, to the east, to the north and to the south. If you have to find a quick way to go from point A to point B and you know that point B is to the south-west of point A, you can do a DFS search, where you prefer to go to the left and to the south, instead of going to the right and north. Since we are moving, point A also changes and therefor the heuristic changes. This is very commonly used in Dijkstra on an infinite field. If you know that the goal is 4 blocks away, then we will not need to search a thousand blocks away on the other side. This is very commonly used in path finding systems on huge data bases, such as a navigation tool for an entire country and / or video games. Another approach is to "cheat" and instead of going one node at a time, we are moving N-nodes at the time in the same direction. If we now arrive at the goal, we will have smaller sub problems to go from these splitted points. Distance / N to be accurate. These can be done the same way leading to a divide and conquer algorithm. If you now have a tiny wall in between 2 paths, you have to backtrack again and use a smaller N, keeping the wall in mind. These heuristic traversals usually have a star behind their names: A*, B*, D*, IDA*, SMA*.

Connected Components

The Connected Components of a graph are all sub-graphs, which are connected in one way or the other. To get all sub-graphs inside an adjacency list or matrix, we have to simply do a DFS search, starting from nodes, which are not yet inside a component. The implementation of this algorithm should be straightforward. The function components() now returns a list of nodes and their group they belong to.
Straight-forward implementation with an adjacencyMatrix. Same principle with an adjacencyList, but realize that an adjacency matrix requires $O(n^2)$ comparisons, while an adjacency list only requires O(n + m) comparisons (n being the vertices and m being the amount of bridges).

DFS – Implementation with DP:

```
vector<vector<int> > adjacencyMatrix;
vector<int> hasComponent;
int nextComponent = 0;

void addComponent(int i) {
    if(hasComponent[i] == -1) {
        hasComponent[i] = nextComponent;
        nextComponent++;
        for(int j = 0; j < adjacencyMatrix[i].size(); j++)
        {
            if(adjacencyMatrix[i][j] != 0)
addComponent(adjacencyMatrix[i][j]);
        }
    }
}

vector<int> components(vector<vector<int> >& inputMatrix) {
    adjacencyMatrix = inputMatrix;
    hasComponent.resize(inputMatrix.size(), -1);

    for(int i = 0; i < hasComponent.size(); i++) addComponent(i);
    return hasComponent;
}
```

Strongly Connected Components
Strongly connected components appear in a directed graph. A strongly connected component is a sub graph, in which every node can reach the other node. Realize that even if they are connected, they mustn't all be strongly connected components: A -> B <-> C <- D for example has 3 strongly connected components: {A}, {B, C}, {D}
An interesting fact about strongly connected components: When you invert the direction of the edges, the strongly connected components will stay the same. Kosaraju-Sharir uses this idea.

Kosaraju Algorithm
Inventor: S. Rao Kosaraju (1978)
Worst-complexity: O(N + M)

The Kosaraju-Sharir algorithm uses DFS from every start point, as long as this start point has not yet been traversed in any way. If any node inside the traversal doesn't have anywhere to go, where it hasn't been yet it adds the current node to a stack and backtracks within the traversal. Now the algorithm inverts the directions of the input graph and does the same DFS, starting from the stack top to bottom this time. Now every traversal will be a strongly connected component.

```
vector< vector<int> > adjacencyList;
vector<bool> didGraph; //0 = not yet visited, 1 = visited
stack<int> inverseOrder;
vector<int> tempResult;

bool isInverse;
```

```
void DFS(int node) {
    if(didGraph[node] == false)
    {
        didGraph[node] = true;

        for(int i = 0; i < adjacencyList[node].size(); i++)
        {
            DFS(adjacencyList[node][i]);
        }
        if(isInverse == false) inverseOrder.push(node);
        else tempResult.push_back(node);
    }
}

vector<vector<int>> kosaraju (vector< vector<int> > graph)
{
    adjacencyList = graph;
    for(int i = 0; i < adjacencyList.size(); i++) didGraph.push_back(false);
    isInverse = false;
    for(int i = 0; i < adjacencyList.size(); i++) DFS(i);
    //Invert Adjacency List
    adjacencyList.clear();
    vector<int> emptyVector;
    for(int i = 0; i < graph.size(); i++)
adjacencyList.push_back(emptyVector);
    for(int i = 0; i < graph.size(); i++)
    {
        for(int j = 0; j < graph[i].size(); j++)
        {
            adjacencyList[graph[i][j]].push_back(i);
        }
    }
    for(int i = 0; i < adjacencyList.size(); i++) didGraph[i] = false;
    isInverse = true;
    vector< vector<int> > result;
    while(inverseOrder.size() != 0)
    {
        tempResult.clear();
        DFS(inverseOrder.top());
        if(tempResult.size() > 0) result.push_back(tempResult);
        inverseOrder.pop();
    }
    return result;
}
```

Euler trail
Euler trail is a trail, which visits every edge exactly once.
A closed Euler trail (= Eulerian tour) exists if the degree of every node is even. An open Euler tour exists if the degree of every node is even with the exception of 2 nodes, the starting and the end point.
Algorithm: Hierholzer Algorithm
Inventor: Carl Hierholzer (1873)
Worst-case complexity: O(n + m)

Hierholzer-Algorithm
Pseudo code concept:
euler-trail(vertex v) {
foreach vertex u in succ(v) do { remove edge(v,u) from graph;
euler-trail(u); push(edge(v,u));
} }

Using a sorted adjacency list as input for fast (non-optimal log n) deletion:

```
vector<multiset<int>> input;
vector<pair<int, int>> trail;
void eulerTrail(int node)
{
    while(input[node].size() != 0)
    {
        auto it = input[node].begin();
        int temp = *it;
        input[node].erase(it);
        //Remove other direction. Only add this line if you have edges which go to both directions.
        input[temp].erase( input[temp].find(node) );
        eulerTrail(temp);
        trail.push_back({temp, node}); //Push the path in the other way around.
    }
}
```

If the Euler cycle is closed you have to call the eulerTrail function with a node of an uneven degree, elsewise you can choose any node. **Fleury's** algorithm is the divide & conquer approach to the same problem. It simply finds a path from A to B using for example DFS and adds the remaining cycles afterwards.

Topological Sorting
Worst-case complexity: O(N + M)
Topological sorting is an order in a DAG, in which every node is removed at a time, so that the node removed doesn't have any parent nodes (No edge pointing towards the node).

```
vector<int> topological_sorting(vector< vector<int> > graph) {
    vector<int> indegree (graph.size(), 0);
    queue<int> q;
    vector<int> solution;
    for(int i = 0; i < graph.size(); i++) {
        for(int j = 0; j < graph[i].size(); j++) {
            indegree[ graph[i][j] ]++;
        }
    }
    //enqueue all nodes with indegree 0
    for(int i = 0; i < graph.size(); i++) {
        if(indegree[i] == 0) {
            q.push(i);
        }
    }
    //remove one node after the other
    while(q.size() > 0) {
        int currentNode = q.front();
        q.pop();
        solution.push_back(currentNode);
        for(int j = 0; j < graph[currentNode].size(); j++) { //remove all edges
            int newNode = graph[currentNode][j];
            indegree[newNode]--;
            if(indegree[newNode] == 0) { //target node has now no more
incoming edges
                q.push(newNode);
            }
        }
    }
    if(solution.size() < graph.size()) cerr << "Graph contains cycles.";
    return solution;
}
```

Articulation points (Hopcroft & Tarjan algorithm)
Inventor: J. Hopcroft, R. Tarjan (1973)
Worst-case complexity: O(N + M)
Articulation points are similar to the connected components, however the exercise is not to find out the components, but finding the nodes, which may not be removed, so that the given graph won't disconnect from itself, if removed.

The concept of the algorithm is to start a DFS search from a root node (without looping). Then for each vertex we store the lowest depth of either its traversal or its neighbors + 1. The root node has depth 0. So the nodes called after the neighbors are called with depth 1. It can however happen that a node gets accessed with a lower depth then its neighbors, so we check for the lowest depth of its neighbors adding 1.
If the distance to the root is *smaller or equal* to the minimum depth of the other neighbors, said vertex is an articulation point.

Tarjan & Vishkin proposed a log(n) algorithm with O(n) space on a PRAM machine. DOI: 10.1137/0214061

```
vector< vector<int> > graph; //Identity Linked List graph
vector<int> depth; //depth of discovery of each node
vector<int> lowpoint; //lowest depth of all vertices reachable
vector<int> parent; //parental node in DFS discovery tree
vector<int> solution;

bool articulationPoint(int cVertex, int cDepth) //start (0,0)
{
   depth[cVertex] = cDepth;
   lowpoint[cVertex] = cDepth;

   bool is_articulation_point = false;
   for(int i = 0; i < graph[cVertex].size(); i++) {
      int neighbor = graph[cVertex][i];
      if(depth[neighbor] == -1) //if undiscovered vertex
      {
         parent[neighbor] = cVertex;
         articulationPoint(neighbor, cDepth + 1); //recursion
      }
      //minimize lowpoint, if the study of the neighbor has reveiled a
new possible root for the component
      if(neighbor != parent[cVertex]) lowpoint[cVertex] =
min(lowpoint[cVertex], lowpoint[neighbor]);
}
   int childCount = 0;
   for(int i = 0; i < graph[cVertex].size(); i++) {
      int neighbor = graph[cVertex][i];
      if(parent[neighbor] == cVertex) { //if child
         if(lowpoint[neighbor] >= cDepth && cDepth != 0) { //means that
there is no other way for the child to get to the root than this edge.
cDepth != 0 means not root node
            is_articulation_point = true;
         }
         childCount++;
      }
      if(cDepth == 0) {
         if(childCount > 1) is_articulation_point = true;
      }
   }

   if(is_articulation_point) solution.push_back(cVertex);
   return is_articulation_point;
}
```

```
vector<int> traverseArticulationPoints(vector< vector<int> > input)
{
    graph = input;
    int n = input.size();
    depth.resize(n+1);
    parent.resize(n+1);
    lowpoint.resize(n+1);
    for(int i = 0; i <= n; ++i) { //set to un-discovered
        parent[i] = 0;
        lowpoint[i] = 0;
        depth[i] = -1;
    }
    solution.clear();

    vector<int> output;
    articulationPoint(0, 0);
    output = solution;
    return output;
}
```

Bridges

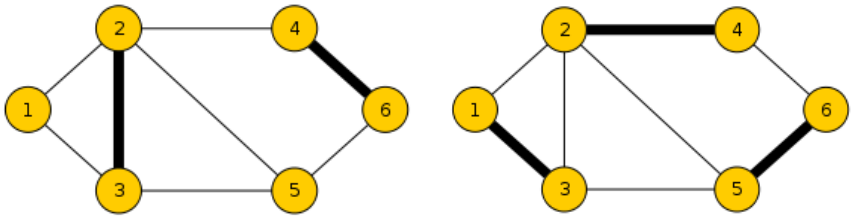
Worst-case complexity: O(N + M)

Bridges is nearly identical to the articulation point problem, however this time bridges are removed and we need to check that if we remove a given bridge, that it won't disconnect the graph into multiple components.

The algorithm works the same way with the exception that a given edge is a bridge only if the distance to the root edge is *smaller* then the minimum depth of the other connected edges.

```
vector< vector<int> > graph; //Identity Linked List graph
vector<int> depth; //depth of discovery of each node
vector<int> lowpoint; //lowest depth of all vertices reachable
vector<int> parent; //parental node in DFS discovery tree
vector<pair<int, int>> solution; //solution

void bridge(int cVertex, int cDepth) { //start (0,0)
    depth[cVertex] = cDepth;
    lowpoint[cVertex] = cDepth;

    for(int i = 0; i < graph[cVertex].size(); i++) {
        int neighbor = graph[cVertex][i];
        if(depth[neighbor] == -1) {
            parent[neighbor] = cVertex;
            bridge(neighbor, cDepth + 1);
        }
        if(neighbor != parent[cVertex]) lowpoint[cVertex] =
min(lowpoint[cVertex], lowpoint[neighbor]);
    }

    for(int i = 0; i < graph[cVertex].size(); i++) {
        int neighbor = graph[cVertex][i];
        if(parent[neighbor] == cVertex) {
            if(lowpoint[neighbor] > cDepth) {
                solution.push_back(make_pair(cVertex, neighbor));
            }
        }
    }
}
```

Bipartite Matching
Bipartite Matching is when you are given two types of nodes. Those nodes are colored in either on or off (white or black). Edges can only connect a white vertex with a black vertex. Given any graph, try to delete as few edges as possible to end up with a bi-partite graph. On the images below the thin lines were deleted. Realize that there can be multiple solutions.



The bipartite matching problem can't be solved using greedy algorithms.

2-color maximum bipartite matching on bipartite graph
Algorithm: Hopcroft & Karp Algorithm
Worst-case complexity: O(√N * M)

Hopcroft & Karp algorithm first chooses a bipartite graph, where every node is only connected once. In a maximum matched graph, every node is matched. We now need to take the nodes, which don't have an edge connected to them yet and choose any of its neighbors. If this new edge has a connection, you must follow the connection, if not we can simply continue by choosing another random node (do a BFS search). If any node, doesn't have any remaining edges, continue the search as if you would do BFS. If the resulting maximum path doesn't have every node in the graph, continue until you finally found a bipartite maximum matching.
Of course there can be more then 2 sets of vertices (colors) and then the above algorithm won't work anymore. If you do multiple BFS searches, from every empty node possible, it is guaranteed that the new path will be bigger then the previous one. This can be exploited to obtain O(√N * M) complexity. The general graph coloring algorithm is NP-complete and can therefor only be solved in polynomial running time.

Implementation note: The bottom implementation is for bipartite graphs only. For general graphs look at the next algorithm.

```
int lNodes, rNodes, NIL, MAX_INT = (1<<28);
vector<int> match, dist;
vector<vector<int> > graph;
bool bfs() {
    queue< int > query;
    for(int i = 0; i < lNodes; ++i) {
        if(match[i] == NIL) {
            dist[i] = 0;
            query.push(i);
        }
        else dist[i] = MAX_INT;
    }
    dist[NIL] = MAX_INT;
    while(!query.empty()) {
        int u = query.front();
        query.pop();
        if(u != NIL) {
            for(int i = 0; i < graph[u].size(); ++i) {
                int v = graph[u][i];
                if(dist[match[v]] == MAX_INT) {
                    dist[match[v]] = dist[u] + 1;
                    query.push(match[v]);
                }
            }
        }
    }
    return (dist[NIL] != MAX_INT);
}
```

```
bool dfs(int u) {
    if(u != NIL) {
        for(int i = 0; i < graph[u].size(); ++i) {
            int v = graph[u][i];
            if(dist[match[v]] == dist[u]+1) {
                if(dfs(match[v])) {
                    match[v] = u;
                    match[u] = v;
                    return true;
                }
            }
        }
        dist[u] = MAX_INT;
        return false;
    }
    return true;
}

int hopcroft_karp(int _lNodes, int _rNodes, vector< pair<int, int> >&
bridges) {
    lNodes = _lNodes; rNodes = _rNodes;
    graph.resize(lNodes+rNodes);
    for(int i = 0; i < bridges.size(); ++i) {
        int u = bridges[i].first;
        int v = bridges[i].second + lNodes;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }
    NIL = lNodes + rNodes + 1;
    match.resize(lNodes+rNodes, NIL);
    dist.resize(NIL, NIL);

    int matching = 0;
    while(bfs()) {
        for(int i = 0; i < lNodes; ++i) {
            if(match[i] == NIL && dfs(i)) ++matching;
        }
    }
    return matching;
}
```

2-color Maximum Bipartite Matching on general graph
Instead of the Edmonds algorithm, which runs in $O(V^2 * E)$ we implement the more efficient (easier to implement IMO), randomized algorithm by Mucha and Sankowski in $O(V^3)$, which can be reduced a bit more using the Stassen's matrix multiplication in the mathematical section.
Memory usage: $O(V^2)$

```
int PRIME = 32887; //bigger prime number = more undeterministic result


int powMod(int a, int b) {
    int res = 1;
    for (; b > 0; b >>= 1) {
        if ((b & 1) != 0)
            res = res * a % PRIME;
        a = a * a % PRIME;
    }
    return res;
}
```

```
int bipartiteMatching(vector<vector<bool> >& adjacencyMatrix) {
    int n = adjacencyMatrix.size();
    vector<vector<int> > m(n, vector<int>(n));
    srand (time(NULL));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            if (adjacencyMatrix[i][j]) {
                m[i][j] = rand() % (PRIME - 1) + 1;
                m[j][i] = PRIME - m[i][j];
            }
        }
    }

    int r = 0;
    for (int j = 0; j < n; ++j) {
        int k = r;
        while(k < n && m[k][j] == 0) ++k;
        if (k != n) {
            swap(m[k], m[r]);

            int inv = powMod(m[r][j], PRIME - 2);
            for (int i = j; i < n; ++i) m[r][i] = m[r][i] * inv % PRIME;

            for (int u = r + 1; u < n; ++u) {
                for (int v = j + 1; v < n; ++v) {
                    m[u][v] = (m[u][v] - m[r][v] * m[u][j] % PRIME + PRIME) %
PRIME;
                }
            }
            ++r;
        }
    }
    return r / 2;
}
```

Stable Marriage Problem (Gale-Shapley algorithm)
Inventor: D. Gale, L. Shapley (1962)
Worst-complexity: $O(N^2)$
The stable marriage problem is not strictly a graph theoretic problem, but it was the only algorithm not fitting in any of the other categories.
Definition of the problem: Given *n* men and *n* women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. When there are no such pairs of people, the set of marriages is deemed stable.

```
vector<int> stableMarriageProblem(vector<vector<int> > men,
vector<vector<int> > women) {
   vector<int> marriedWomen(men.size(), -1);
   stack<int> unmarriedMen;
   for(int i = 0; i < men.size(); ++i) {
      unmarriedMen.push(i);
   }
   while(!unmarriedMen.empty()) {
      int pos = unmarriedMen.top();
      for(int j = 0; j < men[pos].size(); ++j) {
         int partner = men[pos][j];
         if(marriedWomen[ partner ] == -1) {
            unmarriedMen.pop();
            marriedWomen[ partner ] = pos;
            break;
         }
         else {
            int currentMalePartner = marriedWomen[partner];
            int ranking = 0, partnerRanking = 0;

            for(int i = 0; i < women[partner].size(); ++i) {
               if(women[partner][i] == pos) ranking = i;
              if(women[partner][i]==currentMalePartner)partnerRanking=i;
            }
            if(ranking < partnerRanking) {
               unmarriedMen.pop();
               unmarriedMen.push(currentMalePartner);
               marriedWomen[ partner ] = pos;
               break;
            }
         }}} return marriedWomen; }
```

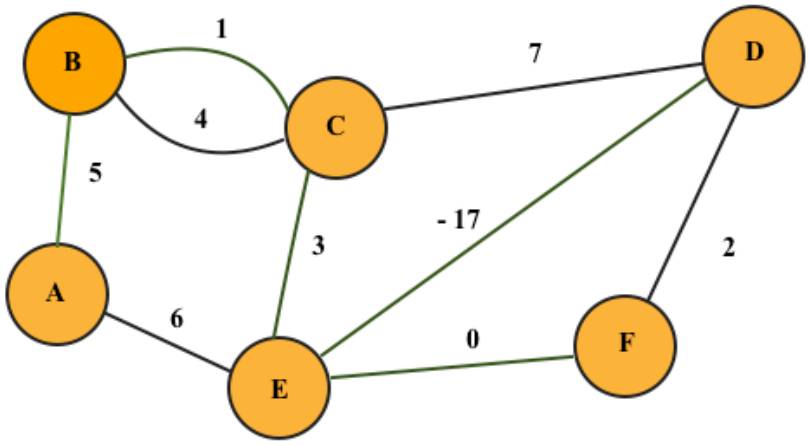
Union – Find (Disjoint Set Problem)
The disjoint set data structure must be able to tell if an edge within a graph is connected to another edge. Every node is first assigned to a leader, at the beginning itself. Whenever two nodes get united, the leader of one node becomes the leader of the leader of the other node. To increase the average running time, make sure to swap the parameters in the union function.

Quick-union implementation
findLeader: O(log(n))
union(): O(log(n))

Implementation note: The keyword union is already used by the C++ language, so instead unite is used.

```
vector<int> p;
int findLeader(int x) {
    if(p[x] != x) p[x] = findLeader(p[x]);
    return p[x];
}

void unite(int x, int y){
    p[findLeader(x)] = findLeader(y);
}

void init(int SIZE) {
    p.resize(SIZE);
    for(int i = 0; i < SIZE; ++i) p[i] = i;
}
```

Minimum Spanning Tree (MST)
Minimum Spanning tree is the smallest framework you can have to connect all edges in a weighted graph.



Sample of a solved Minimum Spanning Tree (green edges).

The optimal solution is always a tree, hence if it had a cycle we could still remove another edge. If it had negative edges, then it wouldn't be considered a tree anymore, hence the optimal solution would be a graph.
The optimal solution for this problem is O (n log n), all of which are greedy algorithms.

Approach 1: Kruskal

Worst-case complexity: O(m log m)

The Kruskal algorithm requires the knowledge of the union-find data structure, which is discussed in the previous sub chapter.

Kruskal algorithm:

1. Sort all edges by cost.
2. Now we use the union function to add the edges if the nodes are not in the same union. Repeat this step until all unions are added.

```
vector<int> p;
int findLeader(int x) {
    if(p[x] != x) p[x] = findLeader(p[x]);
    return p[x];
}
void unite(int x, int y){
    p[findLeader(x)] = findLeader(y);
}
void init(int SIZE) {
    p.resize(SIZE);
    for(int i = 0; i < SIZE; ++i) p[i] = i;
}
struct edge {
    int first, second, cost;
    edge(int _first, int _second, int _cost) : first(_first), second(_second),
cost(_cost) {}
};
bool operator<(const edge& lhs, const edge& rhs) {
    return lhs.cost < rhs.cost;
}
vector<edge> kruskalMST (vector<edge> edges, int SIZE) {
    init(SIZE); //count of nodes
    sort(edges.begin(), edges.end());
    vector<edge> output;
    for(auto& lowestCost : edges) {
        if(findLeader(lowestCost.first) != findLeader(lowestCost.second)) {
                unite(lowestCost.first, lowestCost.second);
                output.push_back(lowestCost);
        }
    }
    return output;
}
```

Approach 2: Reverse-delete Algorithm
Reverse-delete is the exact opposite of the Kruskal algorithm. With reverse-delete you start sorting the edges by highest cost and then start removing the highest cost. It only removes the edge if a MST is still possible.

The straight forward implementation requires O(n * n), hence we can't use quick-union, but only the reverse version of quick-find → which is basically a linear search to check if 2 components are connected with each other. Straight forward implementation: $O(n^2)$

Approach 3: Boruvka's Algorithm O(m log m)
Boruvka's algorithm is a divide and conquer approach of the problem. It again starts by making every node its own union. Every union remembers its smallest edge. Then every edge is getting a union function, which unites the different unions we already have. Then all of the inner edges are getting deleted. Then we repeat this step over and over until all of the nodes are in the same union.

Approach 4: Prim's Algorithm O(n log n + m)
Prims algorithm doesn't require sorting for the edges at the start. We start at a random node and add it to the MST tree (it doesn't have any edges yet). Then we check for the smallest edge, which connects to a node inside the MST – tree and a node outside of it. This edge now gets added to the MST – tree with its new node. Repeat this until all the nodes are inside the MST – tree. This is basically Dijkstra with a slight modification.

Tarjans MST Coloring Rules
1. Choose a slice of a graph, which has no green colored edge and at least one uncolored edge. Now color the smallest uncolored edge in the slice green.
2. Choose any cycle, which has no red colored edge and at least one uncolored edge. Now color the largest uncolored edge in the cycle red.
When all edges are colored the green edges form the MST. It is guaranteed that any of the 2 rules can be executed greedily.
The other algorithms basically describe when to use which of these rules, if any. Prim's Algorithm only uses the green rule for example.

MST in a directed graph?
Technically there are 2 problems in a directed graph that is similar to the MST problem. Arborescence creates the minimum cost tree at a root vertex to any other vertex. The **minimum spanning strong sub(di)graph (MSSS)** is the minimum cost subgraph inside a strongly connected component, which connects every vertex with every other vertex. It has been shown to be NP-complete.

Arborescence
In graph theory arborescence is an MST in a directed graph, starting from a predefined vertex, with a given root r. The solution always is a directed acyclic graph. This can be solved efficiently using Chu-Liu/Edmonds algorithm $O(n^2)$. Prim's algorithm can also be applied for a faster run-time complexity.

Other MST problems
Capacitated Minimum Spanning Tree (CMST) *See NP-Hard problems*
Steiner tree problem (STP): *NP - complete*
Minimum bottleneck spanning tree: Camerini algorithm

Shortest-Path Problem
The shortest path problem is the exercise to find the smallest cost path from node a to node b in a weighted graph.
Sometimes a graph can contain negative weighted cycles inside the graph. The Bellmann-Ford algorithm can detect them easily. There obviously is no solution to a graph with negative weighted cycles.
This problem can be reduced to a dynamic programming problem. Every node C within the shortest path between A and B is guaranteed to be a shortest path between C and B & C and A. Dijkstra and Bellmann-Ford use this in order to calculate the solution.

Dijkstra Algorithm
Worst-case complexity: O(N * log(N) + M) or O(M * log(N) + N)

Dijkstra is an algorithm to find the shortest-path inside a directed graph, which doesn't have any negative weighted cycles. However if any negative edges emerge (without having negative cycles), we can simply add the smallest negative digit to all edges.
→ Make all edges positive, by adding any number N, greater or equal to the absolute value of the smallest negative edge.
Dijkstra can be implemented in a complexity of N log N with the help of a priority queue. A priority queue can be implemented using a Fibonacci heap or a normal binary heap; these are explained in the data structure section. We are using the priority_queue from C++ in the example. Dijkstra is starting at the starting point and pushes all edges coming from the starting point into the minimum priority queue. Now if we take the shortest edge from the priority queue, it is guaranteed to also be the shortest path from the starting point to the connected new point. Now we insert all edges from this new point and add the distance, which we already used to get to this point. We also mark all edges as crossed, when we visit them, hence the first time we visit them, it is guaranteed for it to be the shortest path.
The algorithm is also used for *single-destination shortest-paths*, where the start point is regarded as the destination. The implementation below outputs an array where every node gets the shortest path returned. If the result is -1, there is no connection from the start point to B.

```
//Implementation in O (M * log(N) )
struct Edge {
    int from; //Node A (not required)
    int to; //Node B
    int price; //Price to go from A to B
    Edge(int _from, int _to, int _price) : from(_from), to(_to), price(_price)
        {}
};

inline bool operator< (const Edge& lhs, const Edge& rhs) {
    return lhs.price > rhs.price; //Sort it from smallest to largest
}

vector<int> Dijkstra(vector< vector<Edge> > edges, int startPoint) {
    priority_queue<Edge> sortedEdges; //minimum priority queue.
    vector<int> shortestPath;
    for(int i = 0; i < edges.size(); i++) shortestPath.push_back(-1);
    shortestPath[startPoint] = 0;
    for(int i = 0; i < edges[startPoint].size(); i++)
sortedEdges.push(edges[startPoint][i]);
    while(sortedEdges.size() != 0)
    {
        auto currentEdge = sortedEdges.top();
        sortedEdges.pop();
        if(shortestPath[currentEdge.to] == -1)
        {
            shortestPath[currentEdge.to] = currentEdge.price; //The
shortest path has been found to current.second
            for(int i = 0; i < edges[currentEdge.to].size(); i++)
            {
                Edge newBridge = edges[currentEdge.to][i];
                newBridge.price += currentEdge.price;
                sortedEdges.push(newBridge);
            }
        }
    }
    return shortestPath; //shortestPath[endPoint] = result; If -1 no path.
}
```

In order to solve the **all-pairs shortest path** problem, we can simply use Dijkstra n-times per node with a resulting complexity of $O(n^2 * \log n + m)$ or $O(n * m * \log(n))$. Hence it does not cost more complexity to run the Bellman Ford algorithm first to find negative cycles, you can still check for negative cycles with the same complexity.

A slower well-known algorithm for this problem is the **Floyd-Warshall** algorithm with an $O(N^3)$ complexity.

Maximum Network Flow (Dinic algorithm)
Inventor: Y. Dinitz (1970)
Worst-case complexity: $O(N^2+M)$
Also: Maximum network flow = minimum cut in a flow network
The maximum network flow is the best possible capacity in a flow network from node start to destination. Every node has a maximum capacity, which may not be surpassed. Also assuming a message c gets sent from start to destination, it may be split up over multiple networks and get joined together at the destination node. The maximum network flow finds out the biggest message c, which can possibly be sent.

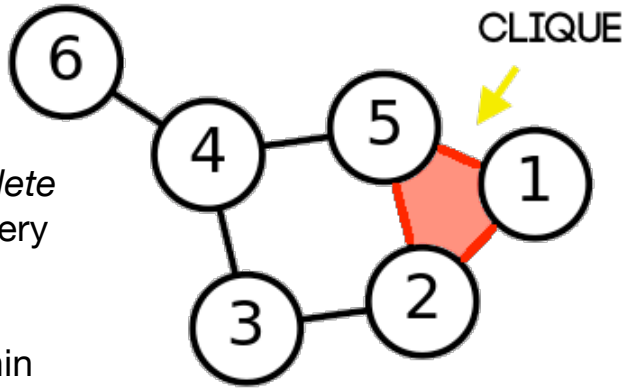
Implementation note: The otherID assumes that the network flow graph is doubly linked.

```
struct Edge {
    int to, otherID;
    int flow, capacity;
};
vector<vector<Edge> > graph;
vector<int> dist, q, work;
int N, start, dest;

bool dinicBFS() {
    for(int i = 0; i < N; ++i) dist[i] = -1;
    dist[start] = 0;
    q[0] = start;
    int left = 0, right = 1;
    while (left < right) {
        int from = q[left];
        for (int j = 0; j < graph[from].size(); ++j) {
            Edge &e = graph[from][j];
            int to = e.to;
            if (dist[to] < 0 && e.flow < e.capacity) {
                dist[to] = dist[from] + 1;
                q[right] = to;
                ++right;
            }
        }
        ++left;
    }
    return dist[dest] >= 0;
}
```

```
int dinicDFS(int from, int maxFlow) {
    if (from == dest) return maxFlow;
    for (int &i = work[from]; i < graph[from].size(); ++i) {
        Edge &e = graph[from][i];
        if (e.capacity <= e.flow) continue;
        int to = e.to;
        if (dist[to] == dist[from] + 1) {
            int tempFlow = dinicDFS(to, min(e.capacity - e.flow, maxFlow));
            if (tempFlow > 0) {
                e.flow += tempFlow;
                graph[to][e.otherID].flow -= tempFlow;
                return tempFlow;
            }
        }
    }
    return 0;
}

int dinicMaxFlow(vector<vector<Edge> >& _graph, int _start, int _dest)
{
    graph = _graph; start = _start; dest = _dest;
    N = graph.size();
    dist.resize(N, 0);
    work.resize(N, 0);
    q.resize(N, 0);
    int result = 0;
    while (dinicBFS()) {
        for(int i = 0; i < N; ++i) work[i] = 0;
        while (int dfs = dinicDFS(start, INT_MAX)) {
            result += dfs;
        }
    }
    return result;
}
```

NP- Hard Problems

Cliques



A clique is a part of an *undirected graph*, which is *complete* (every node is connected with every other node).

A clique is a complete graph within another graph. The exercise of finding the biggest clique or checking if a clique of size n exists is called the clique-problem. The generalization of the problem is to check if a sub-graph exists inside another graph. Both problems are proven to be NP-Complete.

Hamilton path

The Hamilton path asks a similar question as the euler trail, but questions: Is there a path which visits every vertex (node) exactly once? This problem has been proven to be NP – Complete and the naive brute force implementation requires O(n!). However there are algorithms that can solve this problems, the current record holder being $O(1.657^n)$: A Monte-Carlo algorithm by Andreas Björklund. Because Hamilton-path is NP-Complete, we can prove that TSP is NP hard.

Minimum Vertex cover

Minimum Vertex cover is another NP-Complete problem, which can be shown through the NP-Completeness of the Minimum Clique problem (or k-clique problem). The vertex cover problem searches for all (or the maximum number of) vertices, which can be removed, so that every edge still is connected to the graph. Note: When the degree of a vertex >= 2, we can simply invert the edges to vertices and the vertices to edges and then solve the spanning tree. Other solutions are not allowed.

Graph isomorphism

2 Graphs are isomorph, if you can re-label one graph into the other graph. The adjacency matrix must look the same after re-labeling. (Works for normal graphs, directed, weighted, etc.)

The only known method is brute-forcing which requires O(N!) permutations. However checking if 2 graphs are not isomorph can be found very quickly with certain properties. Properties which both graphs must have the same:
Count of degrees
Count of connected components
Cycles

Traveler salesman problem
Given a graph G, what's the shortest path to visit every node once and return back to the starting node? It is an extension of the closed Hamilton path problem, where not only you need to find a closed Hamilton path, but the shortest possible closed Hamilton path.

K-coloring problem
It has been proven in 1976 in one of the first computer-assisted proofs, that a planar graph can always be colored in 4 colors, when no edge crossings occur. It is basically the proof that utilizing 4 colors one can draw every possible Mandala.
The k-color problem is the problem of coloring every vertex in k-colors, such that no two adjacent vertices share the same color. So disproving that a graph can be colored in 2 colors requires a brute-force NP approach. It is very similar to the clique problem, because if a graph has a maximum clique of size N, then it can only be colored with at least N colors. The argument is valid vice versa.

Boolean satisfiability
Every NP-complete problem can be reduced into a Boolean satisfiability problem. Given any logical equation the Boolean satisfiability algorithm has to find out whether the equation can ultimately be true or not. For example: A or B is satisfiable, A = 1, B = 0. The solution can be checked in polynomial time, however this problem can currently only be solved in NP time by trying out every possibility.
A and not A is the classical example of a non-satisfiable equation. More specific the 3-satisfiability problem can't be solved either and is a group of 2 Boolean operators (and & or) and 3 variables, which can be negated or not. For example:

$$F = (!\,x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee !\,x_3 \vee x_4)$$

NP-complete problems without heuristics
There is a thin line between P and NP-complete problems and many problems are only NP-hard in the general case.
2-Sat is in P, while 3-Sat in NP.
Vertex coloring is in NP for colors greater than 2.
Lowest common subsequence is in NP for sequences greater than 2.
There are graphs for which a Hamiltonian path can be found in P-complete time.

The best way to prove a problem is in NP is by showing that you can solve every 3-SAT problem in polynomial time if your problem could be solved in NP.

If you still want to solve an NP-complete problem optimally and fast, you can still use some techniques like pruning and precomputing.
Pruning is an algorithm that reduces the input size in polynomial (or even exponential) time to decrease the time required for the actual NP algorithm. Pruning approaches also include **search trees**. For 3-vertex coloring problem you can start at vertex 0 and color it either in white, green or black. Now for each of these possibilities you can then color vertex 1 in either white, green or black again and so on, resulting in a total complexity of $O(3^N)$. But you can for example detect if vertex 1 is a neighbor of vertex 0 and if that is the case, there are only 2 possible coloring situations and now the algorithm takes O(3 * 2) for the first 2 steps, instead of O(3 * 3). Depending on your pruning techniques you can get running time complexities of $O(1.1^N)$ or $O(2^{sqrt(N)})$, which all still fall under NP-hard. Using search trees for
Pre-computing can for example be used for general SAT problems, by storing tables, tautologies and contradictions. Now you might have a huge library of tautologies for example, but you can directly replace a few statements for example A and not A with satisfiable. For vertex coloring you can merge all vertices that have only one neighbor with their neighbor, where you simply give that vertex a different color.

Heuristic Algorithms

Heuristic algorithms solve a problem non-optimally. They can give an answer out of multiple possible answers or can even give wrong answers with a certain percentage of being correct (Monte Carlo). Heuristic algorithms are usually used to solve NP-complete problems. Heuristic algorithms tend to do many little micro optimizations and can quickly grow in order of complexity.

Polynomial-time approximation scheme (PTAS)

PTAS algorithms are heuristic algorithms, which give you an answer with a certain approximation to the optimal solution. It allows a measurement of the worst-case performance of an heuristical algorithm. This is expressed by a parameter $1 >= \varepsilon >= 0$. $\varepsilon * 100$ gives you the accuracy of the algorithm compared to the optimal solution:
$(1 + \varepsilon)$ for minimization problems and $(1 - \varepsilon)$ for maximization problems.

Greedy algorithms

Greedy algorithms are algorithms that always do the best move first without thinking steps ahead. Greedy algorithms always find a *local maximum* of a given problem as a solution, however this needn't to be the best solution, therefor the global maximum. Greedy algorithms don't *think* about the consequences of their moves and ignore any errors. For example in chess the best possible move is to kill the king. Greedy algorithms don't always have a clearly defined implementation, however every greedy algorithm works with a certain ruleset. A chess greedy algorithm might have a heuristic *decision table* like this:

Figure	Progress
King	Infinite (win)
Queen	9
Rook	5
Bishop	3
Knight	3
Pawn	1

They can even fall in a trap and never find a solution (the same way like DFS in an infinite graph).

If there is only one local maximum, any greedy algorithm will always find the right answer. If there only is one function to climb, you simply climb the function up and up, until you are at the top. With 3 Values you have 3 dimensions and therefor only have one hill, if there only is one local maximum. Again the greedy algorithm will find the top. Greedy algorithms therefor always find a local maximum, but can't find the global maximum. If you for example could take out the queen with the king and a bishop with a pawn a greedy algorithm would choose killing the queen with the king. Now in the next move however the king will be directly killed making it the globally worst move, but the best move in that one situation. The better you grind the decision table, the better the local maximum would be. You could for example add negative progress for putting your figures into killable positions. On complex systems it is usually easier to use machine-learning algorithms instead of greedy algorithms.

Traveler Salesmen Problem
The problem is explained in the graph theory chapter under the section NP-complete problems.

Nearest neighbor algorithm (Greedy)
The nearest neighbor algorithm is the greedy O(N + M) implementation of the traveler salesman problem and usually doesn't come up with the optimal route. In fact it can end up with very bad solutions, so it is probably better using another algorithm, although it is very straightforward.

```
vector<int> nearestNeighbour (vector<vector<int> > graph, int
startPos) {
   vector<bool> done(graph.size(), false);
   vector<int> path;
   int pos = startPos;
   bool allDone = false;
   while(!allDone) {
      done[pos] = true;
      path.push_back(pos);

      int minDist = MAX_INTEGER;
      int dest = 0;
      for(int j = 0; j < graph[pos].size(); ++j) {
         if(done[j] == false && graph[pos][j] < minDist) {
            minDist = graph[pos][j];
            dest = j;
         }
      }
      allDone = true;
      for(int j = 0; j < graph.size(); ++j) {
         if(done[j] == false) allDone = false;
      }
   }
   return path;
}
```

It is much more interesting when we extend the algorithm into a Divide & conquer nearest neighbor algorithm. First we create a set of nearest pairs, which don't already belong to a previously calculated set. Then we connect these pairs and form sets of four, by again finding the smallest distance between the set of pairs, if they are not yet included. This can go on until we have one big set with connected lines. Now we can follow that route.

Christofides' algorithm

The Christodifdes' algorithm has $\varepsilon = 0.5$.
It is a combination of various other algorithms previously discussed in this book.

1. Find a minimum spanning tree for the problem.
2. Create a bipartite matching for the problem with the set of cities of odd order.
3. Find an Eulerian tour for this graph.
4. Convert to TSP by using shortcuts (not necessary)

Ant colony optimization algorithm (ACO)

Ant colony optimization is an example of a swarm intelligence algorithm. The single entity itself, cannot achieve anything, while a growing number of entities form a more favorable result.
The shortest path problem can be solved by the ant colony algorithm. Send out N ants from your starting position in the graph. Each ant can take a random path over the edges of the graph. Every ant leaves a pheromone trail behind, effectively increasing the edge weight of its path. The higher the pheromone rate on an edge is, the higher is the chance, that an ant will follow its path. This behavior results in finding the shortest path. In order to find the TSP, the pheromone trail may only be updated, once a tour is complete. It is also possible to disallow ants to move to revisit nodes, making the algorithm slightly more reliable.

Supervised Machine Learning
A computer program *is said* to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E. – Tom Mitchell

Given some input data and a task we have to design an algorithm which is able to solve this task in a sufficient state. After we sifted out the non-important data and focused on a predictor we can start creating a mathematical description of the problem. A typical task of machine learning is for example to create two sets of groups. Now our **predictor** wants for example a line, which parts 2 groups of points. The predictor is given in supervised learning.
Our predictor:
f(x) = a0 * x + a1
Our program has to find argument a0, the altitude of the line and a1, the position of the line.
Now in order to help the system we could for example already define an approximate a0 and a1, from our human intellect.
For this easy example, we could of course solve the problem mathematically, however if we have 50-dimensions and therefor 50 parameters, math becomes progressively more difficult. This is where machine learning is usefull. In order to find the best line, we have to do predictions. These predictions aim to be the most suitable parameters for each function. So how do we minimize wrongness in guessing now? We implement a **cost function** (or loss function), which takes the arguments of the predictor as parameters:
Cost(p0, p1);

There are multiple ways to create artificial intelligence, however only neuronal networks are described in this book.

Neuronal Network with Perceptrons

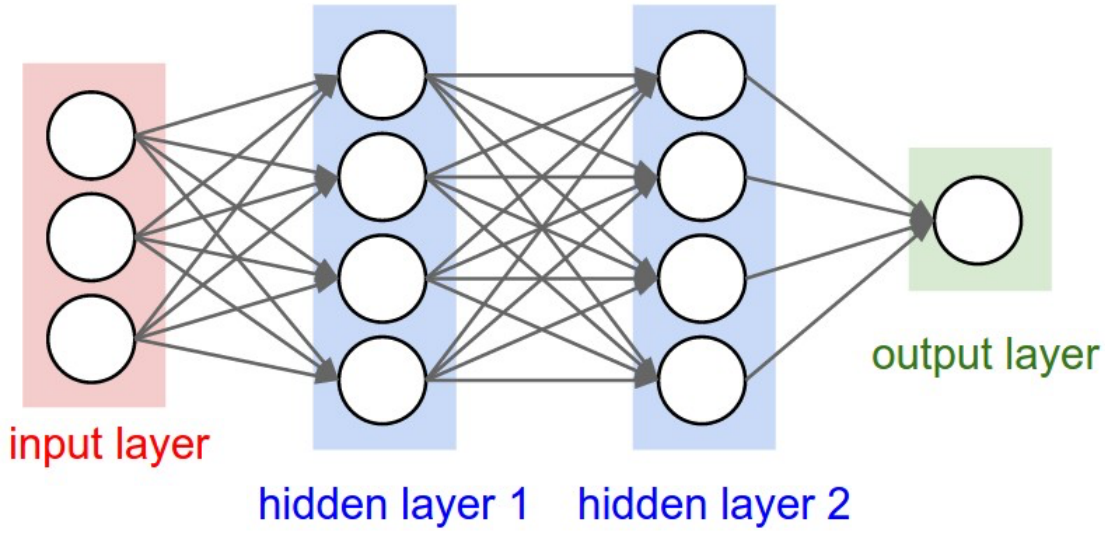
When using a neuronal network to learn something, sigmoid neurons are preferred. A neuronal network can be represented as a weighted directed acyclic graph (W-DAG), where every perceptron is placed in a layer N. If the task where to detect handwriting, in the input layer perceptrons could be linked up to the pixel display. Whenever a pixel goes black, the perceptron is in the ON (1) state or elsewise in the OFF (0) state. Whenever a perceptron gets switched on, its edges get weighted. Also every perceptron (node in the DAG) has a threshold value (with the exception of the input perceptrons). If the sum of the input edges is greater then the threshold, the perceptron either turns the edges on or off (x=0). Each edge also has a weight w, so the edge gets weighed as w * x. For each node i, the following equation holds true, relative to their neighbour nodes j.

$$x_i = \begin{cases} 0 \ \ if \ \sum_j w_j * x_j \ \leq threshold \\ 1 \ \ if \ \sum_j w_j * x_j \ > threshold \end{cases}$$

The threshold of each node is also called the bias b and is defined as b = -threshold. Also $\sum_j w_j * x_j = dotProduct(w, x)$. Therefor the above equation can be rewritten as:

$$x_i = \begin{cases} 0 \ \ if \ w * x + b \leq 0 \\ 1 \ \ if \ w * x + b > 0 \end{cases}$$

The neuronal network has 3 main types of layers, the input layer, the output layer and hidden layers inside a multilayered perceptron network.



```
float getInputState(int ID) {}
void sendOutputState(int ID, bool output) {}

struct edge {
    float weight;
    bool on = false;
    edge(float _weight) : weight(_weight) {}
};

struct perceptron {
    vector<edge*> inputEdges;
    vector<edge*> outputEdges;
    float bias;

    perceptron() {}
    perceptron(float _bias) : bias(_bias) {}
    void update() {
        float sum = 0;
        for(edge* inputEdge : inputEdges) {
            if(inputEdge->on) {
                sum += inputEdge->weight;
            }
        }
        bool newOn = true;
        if(sum + bias <= 0) newOn = false;
        for(edge* outputEdge : outputEdges) {
            outputEdge->on = newOn;
        }
    }
};
struct inputPerceptron : perceptron {
    int ID;
    inputPerceptron(float _bias, int _ID) {
        bias = _bias;
        ID = _ID;
    }
    void update() {
        bool inputOn = getInputState(ID);
        for(edge* outputEdge : outputEdges) {
            outputEdge->on = inputOn;
        }
    }
};
```

```
struct outputPerceptron : perceptron {
    int ID;
    outputPerceptron(float _bias, int _ID) {
        bias = _bias;
        ID = _ID;
    }
    void update() {
        float sum = 0;
        for(edge* inputEdge : inputEdges) {
            if(inputEdge->on) {
                sum += inputEdge->weight;
            }
        }
        bool outputOn = true;
        if(sum + bias <= 0) outputOn = false;
        sendOutputState(ID, outputOn);
    }
};

vector<vector<perceptron> > layers;
void runPerceptronNetwork() {
    for(int i = 0; i < layers.size(); ++i) {
        for(int j = 0; j < layers[i].size(); ++j) {
            layers[i][j].update();
        }
    }
}
```

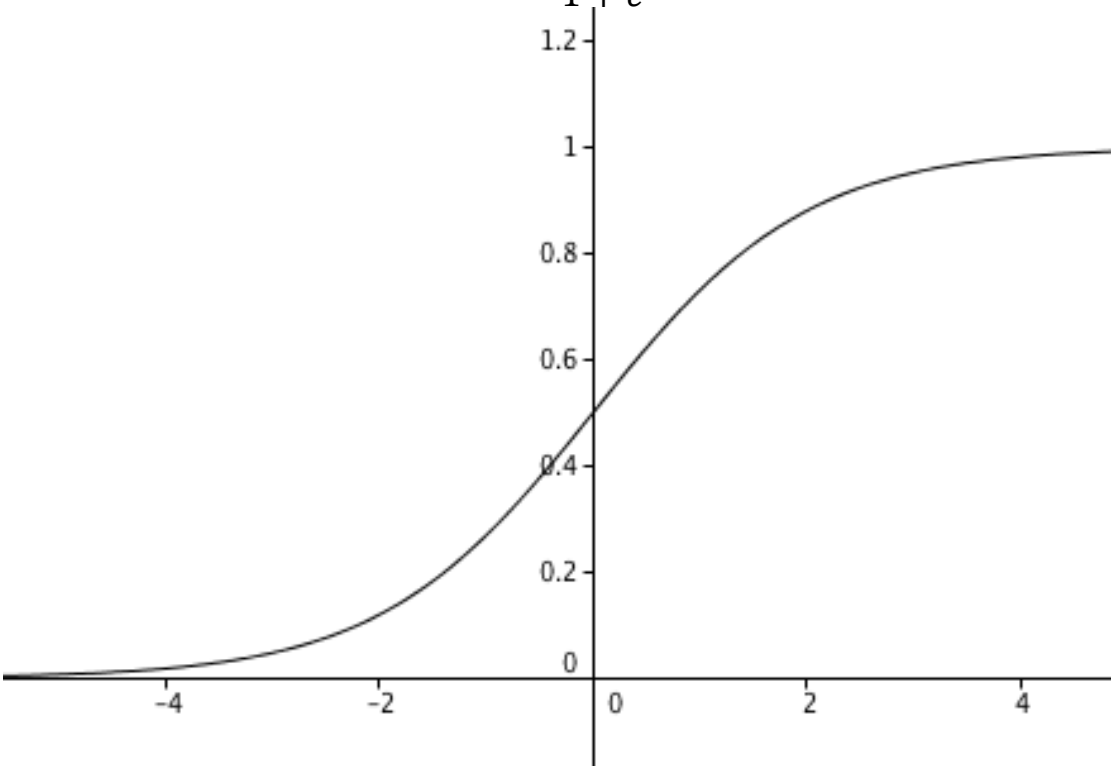
Bias Neuron

In order to create every possible output from every possible input a neuronal network might need a bias node, which always has the same value, usually 1. Just add an inputNeuron, which is always true.

Neuronal Network with Sigmoid Neurons

Optimizing a neuronal network with perceptrons, turns out to be more difficult, because if you change a weight, you will trigger an ON state somewhere else, because their threshold has been triggered. The current threshold function is a step function, where the sigmoid function is a smoothed out step function:

$$f(x) = \frac{1}{1 + e^{-x}}$$



Instead of x being either 0 or 1 (the step function), we can modify the preceptor equation as follows:

$$x_i = \frac{1}{1 + \exp\left(-\sum_j w_j * x_j - b\right)}$$

Feed forward loops don't need an update query, because they are layered.

```
float getInputState(int ID) {}
void sendOutputState(int ID) {}

struct edge {
    float weight;
    float x = 0.0;
    edge(float _weight) : weight(_weight) {}
};

struct neuron {
    vector<edge*> inputEdges;
    vector<edge*> outputEdges;
    float bias;

    neuron() {}
    neuron(float _bias) : bias(_bias) {}
    void update() {
        float sum = 0;
        for(edge* inputEdge : inputEdges) {
            sum += inputEdge->weight * inputEdge->x;
        }
        sum = 1.0 / (1.0 + exp(-sum - bias));
        for(edge* outputEdge : outputEdges) {
            outputEdge->x = sum;
        }
    }
};

struct inputNode : neuron {
    int ID;
    inputNode(float _bias, int _ID) {
        bias = _bias;
        ID = _ID;
    }
    void update() {
        float inputX = getInputState(ID);
        for(edge* outputEdge : outputEdges) {
            outputEdge->x = inputX;
        }
    }
};
```

```
struct outputNode : neuron {
    int ID;
    outputNode(float _bias, int _ID) {
        bias = _bias;
        ID = _ID;
    }
    void update() {
        update();
        sendOutputState(ID);
    }
};

vector<vector<neuron> > layers;
void runNeuronalNetwork() {
    for(int i = 0; i < layers.size(); ++i) {
        for(int j = 0; j < layers[i].size(); ++j) {
            layers[i][j].update();
        }
    }
}
```

Recurrent Neuronal Network (RNN)

Instead of having multilayered neuronal networks, the RNN can have cycles inside its network. When simulating a feed forward network, one only has to simulate the nodes in the chosen layer. For Recurrent Neuronal Networks you have to update every node at step i, if any of their input edges j changed at step i – 1. This can be done, by using a set, which removes duplicates, but has a log N insertion time.

Change edge code to add output location:

```
set<struct neuron*> updateList;
struct edge {
    float weight;
    float x = 0;
    struct neuron* to;
    edge(float _weight, struct neuron* _to) : weight(_weight), to(_to) {}
};
```

Change all occurrences to:

```
for(edge* outputEdge : outputEdges) {
    outputEdge->x = sum;
    updateList.insert(outputEdge->to);
}
```

And add the update loop for a set amount of iterations:

```
void runRecursiveNeuronalNetwork(int steps) {
   for(int i = 0; i < steps; ++i) {
      set<neuron*> currentUpdateList = updateList;
      updateList.clear();
      for(neuron* neuron : currentUpdateList) {
         neuron->update();
      }
   }
}
```

Matrix representation of Neuronal Networks

A few definitions before starting:

b_j^l = j-th bias in the l-th layer

x_j^l =j-th action potential in the l-th layer

w_{jk}^l = the weight of the edge, which connects the k-th node in layer (l-1), with the j-th node in layer l.

$x_j^l = \sigma\left(\sum_k w_{jk}^l * x_k^{l-1} + b_j^l\right)$

w^l = weight matrix of the weight connecting layer (l-1) with layer l.

x^l = activation vector at layer l.

x^l = bias matrix at layer l.

$z^l = w^l * x^{l-1} + b^l$

$x^l = \sigma(w^l * x^{l-1} + b^l)$

```
float getInput(int ID) {return;}
void sendOutput(int ID, float value) {}
struct neuronalNetwork {
   vector<vector<float> > bias;
   vector<vector<float > > x;
   vector<vector<vector<float> > > weights;

   void updateLayer(int l) {
      for(int j = 0; j < x[l-1].size(); ++j) {
         float z = 0;
         for(int k = 0; k < weights[l-1][j].size(); ++k) {
            z += weights[l-1][j][k] * x[l-1][k];
         }
         z += bias[l][j];
         x[l][j] = 1 / (1 + exp(z));
      }
   }
```

```
    void runNeuronalNetwork() {
        for(int i = 0; i < x[0].size(); ++i) {
            x[0][i] = getInput(i);
        }
        for(int i = 1; i < x.size(); ++i) {
            updateLayer(i);
        }
        for(int i = 0; i < x.back().size(); ++i) {
            sendOutput(i, x.back()[i]);
        }
    }
};
```

Backpropagation algorithm (Cost function)

$$C = \frac{1}{n} * \sum_x C_x$$

$$C_x = \frac{1}{2} * \left(\sum_j \left(expected_j - x_j^L \right) \right)^2 ; where\ L = size\ of\ X$$

The above implementation shows training with batch gradient descent for a single training example. The cost function has to be minimized, not maximized.

Layered Neuronal Network with backpropagation

```
float randomFloat(float LO, float HI) {
    return LO + static_cast <float> (rand()) /( static_cast <float>
(RAND_MAX/(HI-LO)));
}

struct layer {
    vector<float> output;
    vector<float> input;
    vector<vector<float> > weights;
    vector<vector<float> > dweights;
    bool isSigmoid = true;
    layer() {}
    layer(int inputSize, int outputSize) {
        output.resize(outputSize);
        //Add bias node
        input.resize(inputSize + 1);
        weights.resize(output.size(), vector<float>(input.size()));
        dweights.resize(output.size(), vector<float>(input.size()));
        for(int i = 0; i < output.size(); ++i) {
            for(int j = 0; j < input.size(); ++j) {
                weights[i][j] = randomFloat(-1.0, 1.0);
            }
        }
    }
    vector<float> updateLayer(vector<float> newInput) {
        for(int i = 0; i < newInput.size(); ++i) input[i] = newInput[i];
        input[input.size() - 1] = 1.0; //bias
        for(int i = 0; i < output.size(); ++i) output[i] = 0;

        for (int i = 0; i < output.size(); ++i) {
            for (int j = 0; j < input.size(); ++j) {
                output[i] += weights[i][j] * input[j];
            }
            if(isSigmoid) output[i] = (1.0 / (1.0 + exp(-output[i])));
        }
        return output;
    }
    vector<float> train(vector<float> error, float learningRate, float
momentum) {
        vector<float> nextError(input.size());
        for (int i = 0; i < output.size(); ++i) {
            float d = error[i];
```

```
            if(isSigmoid) d *= output[i] * (1 - output[i]);
            for (int j = 0; j < input.size(); ++j) {
                nextError[j] += weights[i][j] * d;
                float dw = input[j] * d * learningRate;
                weights[i][j] += dweights[i][j] * momentum + dw;
                dweights[i][j] = dw;
            }
        }
        return nextError;
    }
};

struct neuronalNetwork {
    vector<layer> layers;
    neuronalNetwork(int inputSize, vector<int> layerSizes) {
        layers.resize(layerSizes.size());

        for (int i = 0; i < layerSizes.size(); ++i) {
            int previousLayerSize = inputSize;
            if(i != 0) previousLayerSize = layerSizes[i - 1];
            layers[i] = layer(previousLayerSize, layerSizes[i]);
        }
    }

    vector<float> feedforward(vector<float> input) {
        vector<float> previousInput = input;
        for (int i = 0; i < layers.size(); ++i) {
            previousInput = layers[i].updateLayer(previousInput);
        }
        return previousInput;
    }

    vector<float> train(vector<float> input, vector<float> solution, float
learningRate, float momentum) {
        vector<float> calculatedOutput = feedforward(input);
        vector<float> error(calculatedOutput.size());
        for (int i = 0; i < error.size(); ++i) {
            error[i] = solution[i] - calculatedOutput[i]; // negative error
        }
        for (int i = layers.size() - 1; i >= 0; --i) {
            error = layers[i].train(error, learningRate, momentum);
        }
        return calculatedOutput;
    }};
```

The following function makes it easier to handle training input and output data.

```
neuronalNetwork createNeuronalNetwork(
vector<vector<float> > trainInput, vector<vector<float> > trainOutput,
 vector<int> layerSizes, int iterations, float learningRate,
float momentum) {
    srand (time(NULL));
    layerSizes.push_back(trainOutput[0].size());
    neuronalNetwork nn(trainInput[0].size(), layerSizes);
    nn.layers[nn.layers.size() - 1].isSigmoid = false;
    for(int i = 0; i < iterations; ++i) {
        int random = rand() % trainInput.size();
        nn.train(trainInput[random], trainOutput[random], learningRate,
momentum);
    }
    return nn;
}
```

Examples:

XOR:

```
vector<vector<float> > train = {{0,0},{0,1},{1,0},{1,1}};
vector<vector<float> > res = {{0},{1},{1},{0}};
createNeuronalNetwork(train, res, {4,4}, 10000, 0.3, 0.6);
```

Unary to Hexadecimal:

```
vector<vector<float> > train;
for(int i = 0; i < 16; ++i) {
    vector<float> temp;
    for(int j = 0; j < 16; ++j) {
        if(i == j) temp.push_back(1);
        else temp.push_back(0);
    }
    train.push_back(temp);
}
vector<vector<float> > res;
for(int i = 0; i < 16; ++i) {
    std::string s = std::bitset< 4 >( i ).to_string(); // string conversion
    vector<float> temp;
    for(int j = 0; j < 4; ++j) {
        if(s[j] == '1') temp.push_back(1);
        else temp.push_back(0);
    }
    res.push_back(temp);
}
```

Sine function (or any mathematical function):

```
neuronalNetwork sineCalculator(int iterations, float learningRate, float
momentum) {
    srand (time(NULL));
    vector<int> layerSizes = {4,4,1};
    neuronalNetwork nn(1, layerSizes);
    nn.layers[nn.layers.size() - 1].isSigmoid = false;
    for(int i = 0; i < iterations; ++i) {
        float random = randomFloat(-1, 1);
        vector<float> trainInput;
        vector<float> trainOutput;
        trainInput.push_back(random);
        trainOutput.push_back(sinh(random));

        vector<float> calculatedOutput = nn.train(trainInput, trainOutput,
learningRate, momentum);
        cout << "Expected: " << trainOutput[0] << " Received: " <<
calculatedOutput[0] << endl;
    }
    return nn;
}
```

Curse of dimensionality

Depending on the complexity of the pattern and the number of input nodes, the program might not achieve results in a reasonable time. Usually the available data is sparse compared to the amount of information (dimensions) per data set. For example if you gave an algorithm multiple poems of authors and expect the machine to create similarly good poems, the machine has to understand the entire concept of language, which either needs a huge amount of poems or different data sets, in order to prepare the neuronal network for its real task. This does apply to all machine learning algorithms.

Data Structures

	Array	Dynamic Array	Linked List	Balanced trees	Hash Map
Indexing	O(1)	O(1)	O(n)	O(log n)	O(1)
Insert / delete at beginning	Not available	O(n)	O(1)	O(log n)	-
Insert / delete at end	Not available	O(1)	O(1)	O(log n)	-
Insert / delete anywhere	Not available	O(n)	O(n)	O(log n)	O(1)

Graph data structures are in the graph theory section.

Array
A usual array implementation can hold n elements of any given data type, which has a fixed size. Arrays have a fixed size from the beginning and extending their size requires creating a new array and copying every element from the old array into the new with amortized time O(n). There usually is no reason in using the array over the dynamic array. However they don't waste space.
The array is implemented in most languages by the compiler. Languages like batch simply create variables, with array names: var0, var1, var2 …

Dynamic Array
The dynamic array works exactly as the array, with the exception that it reserves some unused memory, usually double the amount of memory already stored in the dynamic array. Whenever one tries to insert a value into a dynamic array, which doesn't have any reserved space anymore, the dynamic array doubles its size, which results in storing exactly the same amount as reserving memory, which leads to O(1) insertion cost. Alternatively you can simply make an array with a size it won't ever exceed. Dynamic Arrays can also be made to allow insertion at the front and at the end, again by doubling the memory amount if needed in the front. In C++ the deque can use this.

```
struct dynamicArray {
    int firstPlace = 0;
    int lastPlace = 0;
    int fullSize = 4;
    int *array = new int[4];

    dynamicArray() {}

    void push_back(int value) {
        ++lastPlace;
        while(fullSize <= lastPlace) {
            int* newArr = new int[fullSize * 2];
            memcpy( newArr, array, fullSize * sizeof(int) );
            fullSize += fullSize;
            delete [] array;
            array = newArr;
        }
        array[lastPlace] = value;
    }

    void push_front(int value) {
        --firstPlace;
        while(firstPlace < 0) {
            int* newArr = new int[fullSize * 2];
            memcpy( &newArr[fullSize - 1], array, fullSize * sizeof(int) );
            firstPlace += fullSize;
            lastPlace += fullSize;
            fullSize += fullSize;
            delete [] array;
            array = newArr;
        }
        array[firstPlace] = value;
    }

    int& operator[](size_t pos) {
        return array[pos+firstPlace];
    };
};
```

Linked List
Linked List are a collection of nodes which point to each other. In most linked lists, the node either points towards his child node and/or towards his parent. It can also be used as a **queue**, which is a FIFO (First in, last out) data structure.

Stack (LIFO = Last in, first out)
A stack is a linked list with first in, first out principle. It therefor works like a dynamic array, with the commands push_back(Node) = add a Node and pop_back(), remove the top node.

Doubly Linked List
Doubly linked have all 4 commands: push_back(), pop_back(), push_front(), pop_front(). All of these data structures can also be represented by a dynamic array, which extends memory both to the left and to the right hand side.

Circular Linked List
A circular linked list simply links the last element to the first element, by changing its link.

C++ Implementation

```
struct Node {
    int value;
    Node* link = NULL;
    Node (int _value) : value(_value) {}
};

struct Stack {
    Node* top;
    int peek() {
        return top->value;
    }
    void pop_back() {
        if(top != NULL) {
            top = top->link;
        }
    }
    void push_back(int value) {
        Node *newNode = new Node(value);
        newNode->link = top;
        top = newNode;
    }
};

struct Queue {
    Node* first = NULL;
    Node* last = NULL;
    int peek() {
        return first->value;
    }
    void push_back(int value) {
        Node* newNode = new Node(value);
        if(first == NULL) {
            first = newNode;
            last = first;
        } else {
            last->link = newNode;
            last = newNode;
        }
    }
    void pop_front(){
        first = first->link;
    }
};
```

Java Implementations differ quite a bit from the C++ implementation:

```
class Node {
  int value;
  Node link;
  Node (int _value) {value = _value;}
}

class Stack {
  Node top;
  Node pop_back(){
    if(top != null) {
      Node temp = top;
      top = top.link;
      return temp;
    }
  }

  void push_back(Node newNode){
      newNode.link = top;
      top = newNode;
  }
}

class Queue{
  Node first, last;
 void push_back(Node newNode){
    if(first == null){
      first = newNode;
      last = first;
    }else{
      last.next = newNode;
      last = newNode;
    }
  }

  void pop_front(){
    first = first.link;
  }
}
```

Hash Maps

Hash Maps are generally abstract arrays, that don't require integers as an index, but it can be anything as long as a hash function is provided. The hash function down below is used for strings. The greater SIZE is, the more likely the lookup complexity will be O(1). The implementation simply adds the chars up inside the string and takes it modulo the size. There are various other hash functions (for a safe hash function there is RSA, for Robin-Karp algorithm there is a rolling hash).

```
template<class T> class hashMap {
private:
    int SIZE;
    vector<T> valueMap;
    vector<string> keyMap;
    int getHash(string input) {
        int pos = 0;
        for(int i = 0; i < input.size(); ++i) {
            pos += input[i];
            pos %= keyMap.size();
        }
        while(keyMap[pos] != "" && keyMap[pos] != input) ++pos;
        return pos;
    }
public:
    hashMap(int size) {
        SIZE = size;
        valueMap.resize(SIZE);
        keyMap.resize(SIZE, "");
    }
    void push(T value, string hash) {
        int pos = getHash(hash);
        valueMap[pos] = value;
        keyMap[pos] = hash;
    }

    int getValue(string hash) {
        return valueMap[getHash(hash)];
    }
};
```

Trees
The most trivial tree is a singly linked list. In this example every node is followed by another node.
A -> B -> C -> D, etc.

A doubly linked list again is a doubly linked tree, still very trivial.
A <> B <> C <> D, etc.

Trees have a lookup time of O(d), d being the depth of the tree. Every search starts from the root node and traverses its children and grandchildren until it found its node. If a tree is well-balanced the depth of a tree is d = log (n). Depending on the structure of the tree, every node requires d insertion, deletion, lookup. Concepts like lazy propagation decreases the modifying time of a set of given nodes, if it has been built under a certain rule.

Binary Segment tree
The following implementation is a segment tree with lazy propagation. In this example it is used to get the max element of an array. In order to add a sum to the query you want to call:
update_tree(**1**, 0, MAX, from, to, value);

Initialize the tree by size MAX by calling:
init_tree(**MAX**);

C++ implementation

```
vector<int> arr;
vector<int> tree;
vector<int> lazy;

void build_tree(int node, int bottom, int top) {
    if(bottom > top) return; // Nonsense

    if(bottom == top) { // Leaf node
        tree[node] = arr[bottom]; // Init value
        return;
    }

    int mid = (bottom + top) / 2;
    build_tree(node*2, bottom, mid); // Init left child
    build_tree(node*2 + 1, mid + 1, top); // Init right child

    tree[node] = max(tree[node*2], tree[node*2+1]); // Init root value
}

void init_tree(int size) {
    arr.resize(size+1, 0);
    int realSize = 1;
    while(realSize < (size * 2)) realSize <<= 1;
    realSize += 1; //parent node is 1 not 0
    tree.resize(realSize, 0);
    lazy.resize(realSize, 0);
    build_tree(1, 0, size);
}
```

```
void update_tree(int node, int bottom, int top, int from, int to, int
value) {

    if(lazy[node] != 0) {
        tree[node] += lazy[node];

        if(bottom != top) {
            lazy[node*2] += lazy[node];
            lazy[node*2+1] += lazy[node];
        }

        lazy[node] = 0; // Reset it
    }

    if(bottom > top || bottom > to || top < from) return;
    if(bottom >= from && top <= to) { // In range
        tree[node] += value;

        if(bottom != top) { // Not leaf node
            lazy[node*2] += value;
            lazy[node*2+1] += value;
        }
        return;
    }

    int mid = (bottom + top) / 2;
    update_tree(node*2, bottom, mid, from, to, value); // Left child
    update_tree(1+node*2, mid + 1, top, from, to, value); // Right child

    tree[node] = max(tree[node*2], tree[node*2+1]); // Updating root
}
```

```
int query_tree(int node, int bottom, int top, int from, int to) {

    if(bottom > top || bottom > to || top < from) return -INF; // NaN

    if(lazy[node] != 0) {
        tree[node] += lazy[node]; // Update

        if(bottom != top) {
            lazy[node*2] += lazy[node]; // Mark left child as lazy
            lazy[node*2+1] += lazy[node]; // Mark right child as lazy
        }

        lazy[node] = 0; // Reset it
    }

    if(bottom >= from && top <= to) return tree[node];

    int mid = (bottom + top) / 2;
    int q1 = query_tree(node*2, bottom, mid, from, to); // Query left child
    int q2 = query_tree(1+node*2, mid + 1, top, from, to);//Query right ch

    int res = max(q1, q2);
    return res;
}
```

Quadtree & Octree
Quadtrees are used to store 2D objects in a sorted order, the same way as a binary index tree stores 1D objects. Octrees are used to store 3D objects.

C++ Implementation

```
struct Point {
    float x, y;
    Point(float _x, float _y) : x(_x), y(_y) {}
};

int maxCapacity = 16;

struct QuadTree {
    vector<Point> container;
    QuadTree * ul = NULL;
    QuadTree * ur = NULL;
    QuadTree * dl = NULL;
    QuadTree * dr = NULL;

    float x, y, width, height;

    QuadTree(float _x, float _y, float _width, float _height) {
        x = _x;
        y = _y;
        width = _width;
        height = _height;
    }

    void split()
    {
        float halfwidth = width / 2;
        float halfheight = height / 2;
        ul = new QuadTree(x, y, halfwidth, halfheight);
        ur = new QuadTree(x + halfwidth, y, halfwidth, halfheight);
        dl = new QuadTree(x, y + halfheight, halfwidth, halfheight);
        dr = new QuadTree(x+halfwidt, y+halfheight, halfwidth, halfheight);
        for(int i = 0; i < container.size(); i++) {
            ul->addPoint(container[i]);
            ur->addPoint(container[i]);
            dl->addPoint(container[i]);
            dr->addPoint(container[i]);
        }
    }
```

```
    void addPoint(Point &a)
    {
        cout << x << endl;
        if(a.x >= x && a.y >= y && a.x < x + width && a.y < y + height) {
            if(ul == NULL) {
                container.push_back(a);
                if(container.size() > maxCapacity) split();
            }
            else {
                ul->addPoint(a);
                ur->addPoint(a);
                dl->addPoint(a);
                dr->addPoint(a);
            }
        }
    }
};
```

Binary Search Tree (BST)

A binary search tree is a binary index tree, which has its content sorted when traversing. It needn't to be balanced, for a balanced BST, look at the Red-Black tree.

```
struct Node {
    int data;
    Node *leftChild = nullptr, *rightChild = nullptr, *parent = nullptr;
    Node(int n) : data(n) {}
};

void bstTraverse(Node* node) {
    if (node == nullptr) return;

    bstTraverse(node->leftChild);
    cout << node->data << endl;
    bstTraverse(node->rightChild);
}

void bstInsert(Node*& node, Node* newNode) {
    static Node* previous;
    if (node == nullptr) {
        node = newNode;
        newNode->parent = previous;
    }
    else {
        previous = node;
        //Remove duplicates with else if instead of else
        if (node->data > newNode->data) bstInsert(node->leftChild,
newNode);
        else if (node->data < newNode->data) bstInsert(node->rightChild,
newNode);
    }
}

struct BST {
    Node* root = nullptr;
    void insert(int value) {
        Node* newNode = new Node(value);
        bstInsert(root, newNode);
    }
    void traverse() {
        bstTraverse(root);
    }
};
```

Red-Black Tree
Space: O(N)
Search / Insertion / Deletion: worst-case O(log N) = balanced BIT

```
enum COLOR {RED, BLACK};
struct Node {
    int data;
    COLOR color = RED;
    Node *leftChild = nullptr, *rightChild = nullptr, *parent = nullptr;
    Node(int n) : data(n) {}
};

void bstTraverse(Node* node) {
    if (node == nullptr) return;
    bstTraverse(node->leftChild);
    cout << node->data << endl;
    bstTraverse(node->rightChild);
}
void bstInsert(Node*& node, Node* newNode) {
    static Node* previous = nullptr;
    if (node == nullptr) {
        node = newNode;
        newNode->parent = previous;
    }
    else {
        previous = node;
        //Remove duplicates with else if instead of else
        if (node->data > newNode->data) bstInsert(node->leftChild,
newNode);
        else if (node->data < newNode->data) bstInsert(node->rightChild,
newNode);
    }
}

struct RBTree {
    Node* root = nullptr;

    void rotateLeft(Node *&, Node *&);
    void rotateRight(Node *&, Node *&);
    void fixViolation(Node *&, Node *&);

    RBTree() {}
    void insert(int);
    void traverse();
};
```

```
void RBTree::rotateLeft(Node*& node, Node*& newNode) {
    Node* oppositeChild = newNode->rightChild;
    newNode->rightChild = oppositeChild->leftChild;

    if (newNode->rightChild != nullptr) newNode->rightChild->parent =
newNode;
    oppositeChild->parent = newNode->parent;

    if (newNode->parent == nullptr) node = oppositeChild;
    else if (newNode == newNode->parent->leftChild) newNode-
>parent->leftChild = oppositeChild;
    else newNode->parent->rightChild = oppositeChild;

    oppositeChild->leftChild = newNode;
    newNode->parent = oppositeChild;
}
void RBTree::rotateRight(Node*& node, Node*& newNode) {
    Node *oppositeChild = newNode->leftChild;
    newNode->leftChild = oppositeChild->rightChild;

    if (newNode->leftChild != nullptr) newNode->leftChild->parent =
newNode;
    oppositeChild->parent = newNode->parent;

    if (newNode->parent == nullptr) node = oppositeChild;
    else if (newNode == newNode->parent->leftChild) newNode-
>parent->leftChild = oppositeChild;
    else newNode->parent->rightChild = oppositeChild;

    oppositeChild->rightChild = newNode;
    newNode->parent = oppositeChild;
}

void RBTree::insert(int value) {
    Node* newNode = new Node(value);

    bstInsert(root, newNode);
    fixViolation(root, newNode);
}

void RBTree::traverse() {
    bstTraverse(root);
}
```

```
void RBTree::fixViolation(Node*& root, Node*& newNode) {
    while ((newNode != root) && (newNode->color != BLACK) &&
(newNode->parent->color == RED)) {
        Node *father = newNode->parent, *grandfather = father->parent;
        if (father == grandfather->leftChild) {
            Node* uncle = grandfather->rightChild;
            if (uncle != nullptr && uncle->color == RED) {
                grandfather->color = RED;
                father->color = BLACK;
                uncle->color = BLACK;
                newNode = grandfather;
            }
            else {
                if (newNode == father->rightChild) {
                    rotateLeft(root, father);
                    newNode = father;
                    father = newNode->parent;
                }
                rotateRight(root, grandfather);
                swap(father->color, grandfather->color);
                newNode = father;
            }
        }
        else {
            Node* uncle = grandfather->leftChild;
            if (uncle != nullptr && uncle->color == RED) {
                grandfather->color = RED;
                father->color = BLACK;
                uncle->color = BLACK;
                newNode = grandfather;
            }
            else {
                if (newNode == father->leftChild) {
                    rotateRight(root, father);
                    newNode = father;
                    father = newNode->parent;
                }
                rotateLeft(root, grandfather);
                swap(father->color, grandfather->color);
                newNode = father;
            }
        }
    }
    root->color = BLACK;}
```

AVL Tree
The AVL-Tree is almost identical to the Red-Black tree and is a balanced BST as well. Every node has a value, which is calculated by subtracting the height of its right sub tree by the height of its left sub tree. Whenever a node value gets above 1 or under -1, the tree has to be rebalanced, again using the 4 possible double-rotations (LL, LR, RL, RR).

Splay Tree
The splay tree is also a BST. It stores nodes, which are visited more often, closer to the root node, then nodes which are not. Its worst-case complexity can be linear, because it is not self-balanced. On the find, insertion & removal operations the selected nodes must first be splayed. Even if a node is not found in a removal, its parent has to be splayed. LL & RR operations are called the Zig-Zig shift, the LR & RL operations are called the Zig-Zag shift. These shifting methods are not described in this book.

Lowest Common Ancestor (LCA)

The lowest common ancestor problem is a general algorithm, which is used to find the lowest node in a tree, which has the two selected nodes as child or sub-child.

For a binary search tree this task can easily be implemented in O(H), either from a bottom up or top to bottom approach. H is the height of the binary search tree, which if balanced is equivalent to O(log N).

```
Node*& bstLCA(Node*& root, Node*& a, Node*& b) {
   if(root->data > a->data && root->data < b->data){
      return root;
   } else if(root->data > a->data && root->data > b->data) {
      return bstLCA(root->leftChild, a, b);
   } else if(root->data < a->data && root->data < b->data){
      return bstLCA(root->rightChild, a, b);
   }
   return root;
}
```

In order to get a O(H) bottom up algorithm on any binary tree, every node should store the amount of parents it has. Whenever a node gets updated, its child must be informed as well to change their values. Now wander up with one node (the one which has more parents), so both nodes a & b have the same amount of parents. Now every time you move both nodes up by one, check if they are on the same node. If so return this parent.

```
Node*& bstBottomUp(Node*& a, Node*& b) {
   if(b->parentCount < a->parentCount) swap(a, b);
   while(true) {
      if(a == b) return a;
      a = a->parent;
      if(b->parentCount < a->parentCount) b = b->parent;
   }
}
```

There are faster solutions out there with pre-computing O(1) and faster solutions without pre-computing O(log H).

Cryptography

XOR-Cryptography

This cryptography is safe as long as the key is longer then the message and the key is not used multiple times. Basically you don't want the same key repeat inside the message at any time. It's self-explanatory:

```
int encryptDecrypt(int message, int key) {
    message ^= key;
    return message;
}
```

If anyone knows part of the message, the message can't get deciphered, unless the key gets used multiple times. Most cryptography problems focus on reducing the size of the key, but if that isn't an issue, this system is perfect.

Diffie-Hellman key exchange

In order for Alice and Bob exchanging a random key both agree on 2 public numbers. One must be a prime number p and the other can be any random number g.
Alice and Bob each for themselves also invent a random number a & b, which are private.
Alice and Bob now calculate the following messages privately:
$messageToBob = g^a \ mod \ p$
$messageToAlice = g^b \ mod \ p$

Now that Alice and Bob received their messages, they can now calculate the secret code:
$keyFromAlicesPerspective = me\square sageToAlice^a \ mod \ p$
$keyFromBobsPerspective = messageToBob^b \ mod \ p$

This protocol possesses the required property:
$keyFromAlicesPerspective = keyFromBobsPerspective$

Public Key Cryptography

2 Advantages:

1. You don't need to exchange keys: Messages from Person A to Person B are safe.
2. Person A has guaranteed that the message of Person B is safe.

How it works:

Every person chooses a public and a private key. Everyone knows the public key of everyone else. In order to send a message from Alice to Bob, Alice has to encrypt her message with Bobs public key. Now only Bob can open Alice's message, hence you can only decrypt the message with Bob's private key.

However any troll can send Bob a message with his public key and can write as if the message were from Alice. To prevent this scenario Alice can also encrypt the message with her private key. Now Bob, who knows the message comes from Alice, has to decrypt the message both with Alice's public key and Bob's private key.

A broadcast station can encrypt their messages with their private key. Now everyone can read the message from the broadcast station, with their public key, however it is certain, that the message came from the broadcast station and not from a troll.

RSA

For RSA the user needs 2 random prime numbers. Those have to be prime numbers due to the property of euler's totient function (phi function) The phi function computes the number of integers in the range of 1 – n, which have the attribute gcd(n, number chosen between 1-n) = 1
Luckily for us, the phi of any prime is itself minus 1:
Phi (Prime) = Prime – 1

Generating the keys:
First we have to calculate:
N = p * q
Next the user chooses 2 random prime numbers (p, q). Next we calculate L = phi(p * q) = phi(p) * phi(q) = (p - 1) * (q - 1)

The next thing we need to calculate is the exponent *e*. The exponent e has to be chosen, so gcd(N, e) = 1. This is usually best done by randomly trying out numbers. Since you only have to do this step once to create the key, we are safe from there.

N and e are our public keys. Now we need to calculate our private decryption key *d.*
$d = e^{-1} \; mod \; L$

To encrypt a message *m* into an encrypted message *c* we simply have to calculate:
$c = m^e \; mod \; N$

Decrypting the message is even easier:
$m = \; c^d$

In order to crack the code, you'd have to do prime factorization on the number, which is perfectly possible, however very hard to achieve, since its complexity is $O(2^N)$ and therefore NP-Complete. On a quantum computer however factorization, using Shor's algorithm, can be solved in polynomial time.

Mathematical Algorithms

Absolute Value Function (ABS)

Note: Make sure to make a separate abs for floats & doubles to not loose precision!

```
int abs(int val) {
  if(val < 0) val = -val;
  return val;
}
```

Or using fancy ternary operators:

```
float fabs(float a) { return (a < 0) ? -a : a;}
```

Signum Function (SGN)

```
int sgn(int val) {
        if(val < 0) return  -1;
        else if(val == 0) return 0;
        else return 1;
}
```

Signum functions for floating point numbers should never be defined by an equal operator.

```
float fsgn(float val) {
   if(fabs(val) < 1e-7) return 0;
   else if(val < 0) return  -1;
   else return 1;
}
```

Max / Min

Pro tip: You can do many things wrong when implementing a min/max function. Seriously? Yes. Make sure to always only use one comparison operator for min and max and if you can't use preprocessor macros or operator overloading create a min/max function for each data type, elsewise you can have rounding/underflow/overflow errors. Same thing for abs or signum functions.

```
#define MAX(x, y) (((y) < (x)) ? (x) : (y))
#define MIN(x, y) (((x) < (y)) ? (x) : (y))
float minFloat(float a, float b) {
   if(a < b) return a;
   else return b;
}
float maxFloat(float a, float b) {
   if(b < a) return a;
   else return b;
}
```

Floor & Ceil

Floor and ceil functions shouldn't be implemented by hand, but if you really want an implementation using float modulo, please:

```
float ceil(float val) {
  if(val < 0) return floor(x);
  if(val % 1 == 0) return val;
  return val + (1 - (val % 1));
}
float floor(float val) {
  if(val < 0) return ceil(x);
  return val - (val % 1);
}
float round(float val) {return floor(val + 0.5);}
```

Interesting properties:

```
ceil(floor(val)) == floor(val)
floor(ceil(val)) == ceil(val)
floor(-val) == - ceil(val)
ceil(-val) == - floor(val)
round(val) ==  floor(val + 0.5)
```

Ceil and floor should always return a float!

Binary Addition

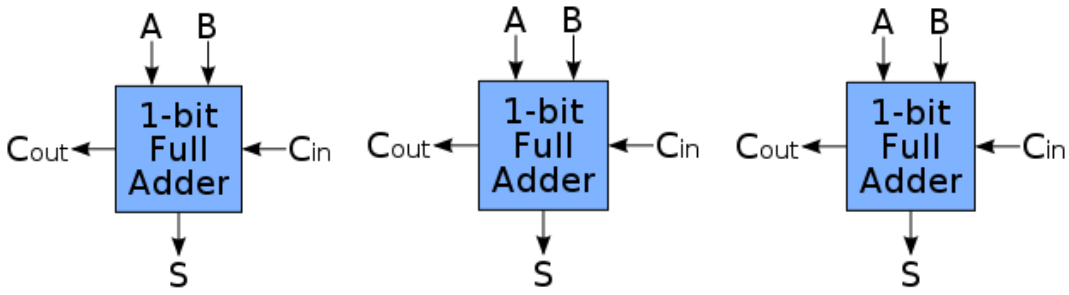
```
  1
+ 1
-----
1 0
```

In computers full binary adders perform addition. Binary representation of digits within a computer:
01110 = A = 14
11101 = B = 29
101011 = S = 43

We can create any number by only using log(N) space, by accounting its memory size and order of bits.
A_1 = 0
A_2 = 1
A_4 = 1
A_8 = 0
A_16 = 0

This is a full binary adder:
s = (a ^ b) ^ c_in
c_out = (a & b) | (c_in & (a ^ b))

Now for every digit of the calculation we need a full adder.
The initial setup is:
c_in = 0
a = A_1
b = B_1



1. First step:
(0 ^ 1) ^ 0 = S_1 = 1
(0 & 1) | (0 & (0 ^ 1)) = c_out = 0

For the next steps we do:
c_in = c_out
a = A_next (A_2 in step 2)
b = B_next (B_2 in step 2)

2. (1 ^ 0) ^ 0 = S_2 = 1
   (1 & 0) | (0 & (0 ^ 1)) = c_out = 0
3. (1 ^ 1) ^ 0 = S_4 = 0
   (1 & 1) | (0 & (0 ^ 1)) = c_out = 1
4. (1 ^ 1) ^ 1 = S_8 = 1
   (1 & 1) | (1 & (1 ^ 1)) = c_out = 1
5. (0 ^ 1) ^ 1 = S_16 = 0
   (0 & 1) | (1 & (0 ^ 1)) = c_out = 1
6. (0 ^ 0) ^1 = S_32 = 1
   (0 & 0) | (1 & (0 ^ 0)) = c_out = 0

From the fifth to the sixth step, the only remaining input was c_in, which built S_32.

Now you know how an ALU can do arithmetic using only logic gates.

Addition with negative 3-bit numbers:
3 + (- 1)
Positive number – negative number problem:
2^n - x
2^(3bit) – 1 = 8 – 1 = 7
BUT:
3 = 0 1 1
7 = 1 1 1
---
1 0 1 0
The same thing in a (8bit) system:
11 + (-9)
2^(8bit) – 1 = 256 – 9 = 247
(signed) 11 = 00001011
(unsigned) 247 = 11110111

1000000010 (cropped) = 0000|0010

Note: Using a deque allows push_front later at the multiplication.

```
deque<bool> binaryStringToArray(string str) {
    deque<bool> output;
    for(int i = str.size() - 1; i >= 0; -- i) {
        if(str[i] == '1') output.push_back(1);
        else output.push_back(0);
    }
    return output;
}

string arrayToBinaryString(deque<bool> arr) {
    string output = "";
    for(int i = arr.size() - 1; i >= 0; --i) output += to_string(arr[i]);
    return output;
}

deque<bool> addition(deque<bool> a, deque<bool> b) {
    deque<bool> output;
    bool carry = 0;
    for(int i = 0; i < max(a.size(), b.size()) || carry != 0; ++i) {
        bool q = a[i] ^ b[i];
        output.push_back( q ^ carry );
        carry = (a[i] & b[i]) | (carry & q);
    }
    return output;
}
```

Binary Subtraction

This is a full binary subtractor:

```
s = (a ^ b) ^ c_in
c_out = (!a[i] & (b[i] ^ c_in)) | (b[i] & c_in)

deque<bool> subtraction(deque<bool> a, deque<bool> b) {
    deque<bool> output;
    bool carry = 0;
    for(int i = 0; i < max(a.size(), b.size()) || carry != 0; ++i) {
        bool q = a[i] ^ b[i];
        output.push_back( q ^ carry );
        carry = (!a[i] & (b[i] ^ carry)) | (b[i] & carry);
    }
    return output;
}
```

Binary Multiplication

The basic school method is the most easy to implement and has a complexity of $O(N^2)$

```
deque<bool> multiply(deque<bool> a, deque<bool> b) {
    deque<bool> output(1, false);
    for(int i = 0; i < a.size(); ++i) {
        if(a[i] == 1) output = addition(output, b);
        b.push_front(0);
    }
    return output;
}
```

Another easy implementation of binary multiplication dates back to Al Khwarizmi and is especially easy to implement in binary. It requires $O(N^2)$ steps to compute and is based on the following idea:

$$\boldsymbol{a} * \boldsymbol{b} = \begin{cases} if\ b\ is\ even: \mathbf{2}(\boldsymbol{a} * floor(\frac{\boldsymbol{b}}{\mathbf{2}})) \\ if\ b\ is\ odd: \mathbf{2}\left(\boldsymbol{a} * floor(\frac{\boldsymbol{b}}{\mathbf{2}})\right) + \boldsymbol{a} \end{cases}$$

The floor division by 2 is a simple binary shift, removing the remainder.
Adding a zero to the number allows multiplication by 2.
And in order to check if a number is even or odd, one has to simply check the first digit.

Karatsuba Multiplication

Karatsuba multiplication reduces the general complexity of multiplication to $O(n^{\log(3)})$, which might not seem much, but can be a huge improvement towards the standard $O(n^2)$ algorithm. It also offers a divide and conquer approach to multiplication. The bottom implementation has also been done by binary deque's. Multiplying any 2 strings, which contain numbers in base B, it's product can be calculated using the algorithm of Karatusba:

$$X * Y = B^{2*ceil(\frac{n}{2})} * X1 * Y1 + B^{ceil(\frac{n}{2})} * [(X1 + X2) * (Y1 + Y2) - X1 * Y1 - X2 * Y2] + X2 * Y2$$

X1 is the number's left side (size: floor(n/2)). Same thing for Y1.
X2 is the number's right side (size: ceil(n/2)). Same thing for Y2.

For simplicity reasons, we simply make the size of the two numbers equal and a multiple of 2, by adding zeros.

```
deque<bool> karatsuba(deque<bool> a, deque<bool> b)
{
    deque<bool> output;
    if(a.size() == 1) {
        if(a[0] == 1) return b;
        else return a;
    }
    if(b.size() == 1) {
        if(b[0] == 1) return a;
        else return b;
    }

    if(a.size() % 2 != 0) a.push_back(0);
    if(b.size() % 2 != 0) b.push_back(0);
    while(a.size() < b.size()) a.push_back(0);
    while(b.size() < a.size()) b.push_back(0);

    deque<bool> x1, x2, y1, y2;
    for(int i = a.size() - 1; i >= a.size() / 2; --i) x1.push_front(a[i]);
    for(int i = (a.size() / 2) - 1; i >= 0; --i) x2.push_front(a[i]);
    for(int i = b.size() - 1; i >= b.size() / 2; --i) y1.push_front(b[i]);
    for(int i = (b.size() / 2) - 1; i >= 0; --i) y2.push_front(b[i]);

    int deg = x2.size();
    deque<bool> newA = karatsuba(x1, y1);
    deque<bool> newC = karatsuba(x2, y2);
    deque<bool> newB = karatsuba(addition(x1, x2), addition(y1, y2));
    newB = subtraction(subtraction(newB, newA), newC);
    for(int i = 0; i < 2 * deg; ++i) newA.push_front(0); //newA <<= 2 * deg;
    for(int i = 0; i < deg; ++i) newB.push_front(0); //newB <<= deg;
    output = addition(addition(newA, newB), newC);
    while(output.size() > 1 && output.back() == 0) output.pop_back();
    return output;
}
```

Binary Division & Modulo

Again we implement the school method, with the complexity of $O(N^2)$. The reminder of the school method has to be returned to calculate the modulus.

```
deque<bool> division(deque<bool> a, deque<bool> b) {
    if(b == deque<bool>(1,0)) return b;
    deque<bool> quotient(a.size(), 0);
    deque<bool> reminder;
    for(int i = a.size() - 1; i >= 0; --i) {
        reminder.push_front(a[i]);
        if(!(reminder < b)) {
            reminder = subtraction(reminder, b);
            quotient[i] = 1;
        }
    }
    return quotient;
}

deque<bool> modulo(deque<bool> a, deque<bool> b) {
    if(b == deque<bool>(1,0)) return b;
    deque<bool> quotient(a.size(), 0);
    deque<bool> reminder;
    for(int i = a.size() - 1; i >= 0; --i) {
        reminder.push_front(a[i]);
        if(!(reminder < b)) {
            reminder = subtraction(reminder, b);
            quotient[i] = 1;
        }
    }
    return reminder;
}
```

Comparisons

```
 bool isLess(deque<bool> a, deque<bool> b) {
    bool isLess = false;
    for(int i = 0; i < max(a.size(), b.size()); ++i) {
        if(a[i] != b[i]) {
            if(a[i] == 0) isLess = true;
            else isLess = false;
        }
    }
    return isLess;
}
```

Other Comparisons using the less than operator:
A > B is equivalent to B < A
A == B is equivalent to !(A < B) && !(B < A)
A >= B is equivalent to !(A < B)
A <= B is equivalent to !(B < A)

Binary to Hexadecimal

Converting a binary number into hexadecimal, is fortunately very easy, hence 16 is a multiple of 2. We can split any binary number into groups of 4. For example:
0110 1101 0110 0001 = 2801
Now every 4-bit can be written as a unique hexadecimal, easy right?

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

So 0110 1101 0110 0001, becomes: 6D61

Binary to Decimal Conversion (Horner's Method)
Converting binary into decimal sadly requires a number implementation in the decimal system. It however only needs to be able to add numbers together, in order to complete the conversion.

```
//C++ Implementation
unsigned long long binaryToDecimal(deque<bool> input) {
    unsigned long long number = 0;
    for(int i = input.size() - 1; i >= 0; --i) {
        number += number;
        number += input[i];
    }
    return number;
}
```

Decimal to Binary Conversion

```
deque<bool> decimalToArray(unsigned long long dec) {
    unsigned long long temp = 1;
    deque<bool> output;
    while(temp <= dec) temp *= 2;
    while(temp >= 1) {
        if(temp <= dec) {
            output.push_front(1);
            dec -= temp;
        }
        else output.push_front(0);
        temp /= 2;
    }
    return output;
}
```

Log N Higher Order Arithmetic's Algorithm

We previously looked at how addition is defined logically. Multiplication nowadays basically is hard-wired and can be calculated way faster then the following method. For example we want to calculate 811 * 812. This equals 811 + 811 + 811 … N – times. So in the most naïve implementation we had O(N) - additions. Whenever an operation holds the *associativity rule* we can reduce the complexity of the given algorithm into O(log N).

2 * 811 = 811 + 811 = 1622

4 * 811 = (2 * 811) + (2 * 811) = 811 + 811 + 811 + 811 = 1622 + 1622 = 3244.

In order to calculate 4 * 811 we don't require 4 additions but only 2. 811 + 811 = 1622. And 1622 + 1622 = 3244.

The amount of calculations therefor can be reduced to log N:

8 * 811 = 3244 + 3244 = 6488

This works because we can change the order of brackets inside the addition, when the associativity rule is valid.

8 * 811 = 811 + (811 + (811 + (811 + (811 + (811 + (811 + 811))))))

Reducing this form can lead to:

((811 + 811) + (811 + 811)) + ((811 + 811) + (811 + 811))

This works of course for exponentiation as well and for matrix exponentiation, due to it being associative.

In lambda calculus we could easily replace the following equation as follows:

Lambda ((+ 811) 811) (equation) . 1622

We can also define the addition as a log N version of incrementing by one (++), this is the approach of Churchwells encodings in lambda calculus.

Exponentiation implementation

Exponentiation in log E, where E is the value of the exponent, if multiplication is viewed as a constant time operation.

```
deque<bool> powA(deque<bool> a, deque<bool> b) {
    deque<bool> output(1, 1);
    for(int i = b.size() - 1; i >= 0; --i) {
        output = karatsuba(output, output);
        if(b[i] == 1) output = karatsuba(output, a);
    }
    return output;
}
```

Starting from this point on, the algorithms will not be implemented with deque<bool> (binary), but instead will get calculated with the normal data types. The conversion should be straight-forward from this point on.

N-th root algorithm

Calculating the nth root, can be easily done using division and exponentation:

$$\sqrt[n]{x} = x^{1/n}$$

If any logarithm of base b is implemented (usually b = e) it can also be calculated using the following equation:

$$\sqrt[n]{x} = b^{\frac{\log_b x}{n}}$$

Arithmetic Sequences

Defining an iterative function a, which linearly increases and fits into this form for every sequence, can be calculated in constant time:

$$d = a_n - a_{n-1}$$

$$a_n = a_1 + (n - 1) * d$$

$$s_n = n * a_1 + \frac{n * (n-1)}{2} * d$$

S_n is the sum of the entire sequence up to n.

So if we wanted to calculate $\sum_{i=0}^{100} i = 1 + 2 + 3.. + 99 + 100$, we only need to calculate the above formula in O(1), where $a_1 = 1$ and n = 100.

```
ull sumOfSequence(ull n) {
    ull a1 = 1;
    ull d = 1;
    return n * a1 + ((n*(n-1)) / 2) * d; }
```

Geometric Sequences

Geometric sequences are like arithmetic sequences but don't use summation, but instead use multiplication as a tool.
Instead of d, we use q now.

$$q = \frac{a_n}{a_{n-1}}$$

$$a_n = a_1 * q^{n-1}$$

$$s_n = \sum_{i=1}^{n} a_i = a_1 * \frac{1-q^n}{1-q} \; ; when\ q \neq 1$$

Factorial

N! = 1 * 2 * 3 * 4 * … N-1 * N
A standard recursive implementation could be implemented as:

```
ull factorial(ull n) {
    if(n <= 1) return 1;
    return factorial(n - 1) * n;
}
```

Stirlings approximation suggests that $n! \approx \sqrt{2\pi} * n^{n+0.5} * e^{-n}$
Before starting the explanation on a faster exact algorithm I want to point out, that most implementations simply store the factorials and don't worry about calculating them another time, hence the complexity O(1).
Instead of storing each step 1!, 2!, 3!, etc. we can also store the prime factorizations of the numbers we don't know yet.
1 = 1, 2 = 2, 3 = 3, 4 = 2 * 2, 5 = 5, 6 = 2 * 3, 7 = 7, 8 = 2 * 2 * 2, 9 = 3 * 3, 10 = 2 * 5, 11 = 11, 12 = 2 * 2 * 3.
Now instead of multiplying the numbers, we can also multiply their prime factors:
12! =
2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 *
3 * 3 * 3 * 3 * 3 *
5 * 5 *
7 *
11
or $2^{10} * 3^5 * 5^2 * 7 * 11$.
Now since calculating the exponentiation only takes O(log N), calculating the prime factors can increase your efficiency of calculating the factorial. However since prime factorization is $O(b^k)$, I'd go with the first method.

Binomial Coefficient

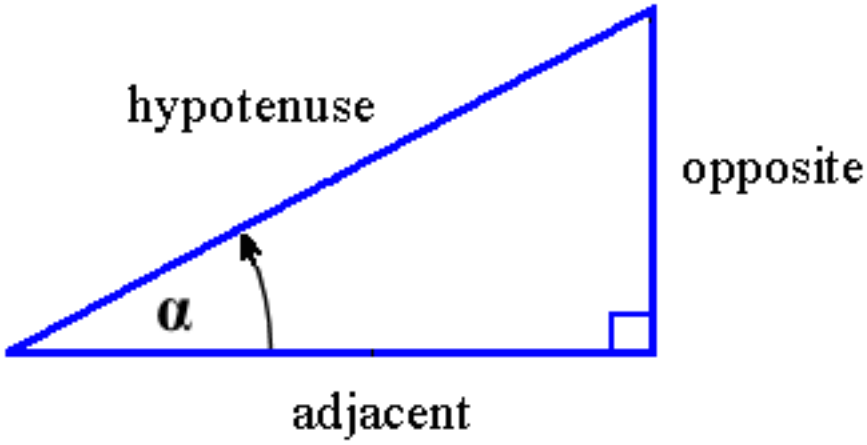
$$\binom{n}{k} = \prod_{i=1}^{n} \frac{n+1-j}{j} = \frac{n!}{k! * (n-k)!}$$

$$\binom{n}{k} = \frac{n!}{k!*(n-k)!} = \frac{n!}{(n-k)!*k!} = \frac{n!}{(n-k)!*(n-(n-k))!} = \binom{n}{n-k}$$

Using the above rules an algorithm with worst case complexity of O(K) (regardless of N), can be created.

```
int binomialCoefficient(int n, int k) {
    int output = 1;
    if (n - k < k) k = n - k;
    for (int i = 0; i < k; ++i) {
        output *= (n - i);
        output /= (i + 1);
    }
    return output;
}
```

Trigonemetric functions (classical)



$$\sin(\propto) = \frac{opposite}{hypotenuse}$$

$$\cos(\propto) = \frac{adjacent}{hypotenuse}$$

$$\tan(\propto) = \frac{opposite}{adjacent} = \frac{\sin(\propto)}{\cos(\propto)}$$

$$1 = (\sin(\propto))^2 + (\cos(\propto))^2$$

Any triangle with segments abc and opposite angles ABC:

Sine law: $\frac{a}{sin(A)} = \frac{b}{sin(B)} = \frac{c}{sin(C)}$

Cosine law: $a^2 = b^2 + c^2 - 2 * a * b * \cos(A)$

$$\sin(-\propto) = -\sin(\propto);\ \cos(-\propto) = -\cos(\propto);\ \tan(-\propto) = -\tan(\propto)$$

For CORDIC:

$$\cos(\propto) = \frac{1}{\sqrt{1 + (\tan(\propto))^2}}$$

$$\sin(\propto) = \frac{\tan(\propto)}{\sqrt{1 + (\tan(\propto))^2}}$$

$$\frac{1}{(cos(\propto))^2} = 1 + (\tan(\propto))^2$$

Sequences

Many mathematical functions are defined as sequences and sadly suboptimal (Remember the classical Fibonacci example).

$$e^z = \sum_{k=1}^{\infty} \frac{z^k}{k!} = \exp(z)$$

$$\cos(z) = \frac{e^{ix} + e^{-ix}}{2} = \sum_{k=0}^{\infty} \frac{(-1)^k z^{2k+1}}{(2k)!}$$

$$\cosh(z) = \sum_{k=0}^{\infty} \frac{z^{2k+1}}{(2k)!}$$

$$\frac{1 - z^{n+1}}{1 - z^n} = \sum_{k=0}^{n} z^k \text{ (geometric sequence)}$$

$$2^n = \sum_{k=0}^{n} \binom{n}{k}$$

$$\log(1 - x) = -\sum_{k=0}^{\infty} \frac{x^k}{k} \text{ ; for } |x| < 1$$

$$(1 + z)^x = \sum_{k=0}^{\infty} \binom{x}{k} * z^k \text{ ; for } |z| < 1 \text{ (binomial theorem)}$$

Taylor series:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!} * (x - x_0)^k$$

After the matrix section, CORDIC will be explained in detail, a modern algorithm to calculate trigonometric functions and other sequences.

Stein's Binary GCD Algorithm
More optimized for running on a computer would be stein's algorithm, which does many computations as binary operations.

```
#define ull unsigned long long int
ull gcd(ull a, ull b)
{
    if (a == b) return a;
    if (a == 0) return b;
    if (b == 0) return a;
    if (~a & 1)
    {
        if (b & 1) return gcd(a >> 1, b);
        else return gcd(a >> 1, b >> 1) << 1;
    }

    if (~b & 1) return gcd(a, b >> 1);
    if (a > b) return gcd((a - b) >> 1, b);
    return gcd((b - a) >> 1, a);
}
```

Least Common Multiple (LCM)
The lcm can be calculated out of the gcd by calculating:

```
ull lcm(ull a, ull b) {
    return (a / gcd(a, b)) * b;
}
```

Modulo rules
A % B = C
C is the remainder when A is divided by B.

Most important rules:

```
(a + b) % n == (a % n + b % n) % n
(a * b) % n == (a % n * b % n) % n
(a ^ b) % n == ((a % n) ^ b) % n
```

Since multiplying big integers can cause overflow errors, there are a few simple algorithms to avoid this problem.

(A * B) % N

```
#define ll long long
ll mulModulo(ll a, ll b, ll n) {
    ll x = 0, y = a % n;
    while(b > 0){
        if(b % 2 == 1) {
            x = (x+y) % n;
        }
        y = (y*2) % n;
        b /= 2;
    }
    return x % n;
}
```

A^B % N

```
#define ll long long
ll expModulo(ll a, ll b, ll n) {
    ll temp = 1;
    while(b > 0){
        if(b%2 == 1){
            temp = (temp*a) % n;
        }
        a = (a*a) % n; // squaring the base
        b /= 2;
    }
    return temp % n;
}
```

A^B % N in a fast binary way:

```
#define ll long long
ll expModulo(ll a, ll b, ll n) {
    ll temp = 1;
    for(temp = 1; b > 0; b >>= 1)
    {
        if (b & 1) temp = ((temp % n) * (a % n)) % n;
        a = ((a % n) * (a % n)) % n;
    }
    return temp;
}
```

Prime Number
Checking if a number is prime can be primitively calculated in O(N) by checking every previous number.

```
bool isPrime (unsigned long long input) {
    if(input <= 1) return false;
    bool isPrime = true;
    for(int i = 2; i < input; ++i) {
        if(input % i == 0) isPrime = false;
    }
    return isPrime;
}
```

Prime sieve
Listing all prime numbers until a final number still have a general complexity of $O(N^2)$, but you must only check every prime number. Therefor N is the amount of prime numbers between 2 and the point.

```
vector<ull> primeSieve (ull until) {
    vector<ull> primes;
    for(int i = 2; i <= until; ++i) {
        bool isPrime = true;
        for(prime : primes) {
            if(i % prime == 0) isPrime = false;
        }
        if(isPrime) primes.push_back(i);
    }
    return primes;
}
```

Miller-Rabin prime test

The Miller-Rabin prime test is based upon Fermat's little theorem.

$r \,\%\, p = 0$

$r^{p-1} \equiv 1 \,(mod\ p)$

Or in a more programmer like syntax:

$(r^{p-1} - 1) \,\%\, p = 0$

If the above statement is false, p is definetly not a prime number.
If the above statement is true, p is probably a prime number.
341 = 11 * 31 and therefor is not prime.
Yet (2^{340} – 1) % 341 = 0. While for prime numbers any random number creates a true response, some non-prime numbers might return a true response for certain numbers of r.
Therefor this algorithm is considered a Monte-Carlo algorithm. In order to achieve higher likelihood of it being correct you can increase the iteration to about 30 (almost 0% falsehood).

C++ Implementation
*Note: This implementation uses exponentiation modulo (a^b)%c and multiplication modulo (a*b)%c. These are explained a little further up on the subject.*

```
#define ll long long
bool Miller (ll prime, int iteration){
    if(prime < 2) return false;
    if(prime !=2 && prime % 2 == 0) return false;

    ll s = prime - 1;
    while(s%2==0){
        s/=2;
    }
    for(int i=0; i < iteration; ++i) {
        ll randomNum = rand()%(prime-1) + 1; //1 <= randomNum < prime
        ll temp = s;
        ll mod = expModulo(randomNum, temp, prime);
        while(temp != prime - 1 && mod != 1 && mod != prime-1){
            mod = mulModulo(mod,mod,prime);
            temp *= 2;
        }
        if(mod != prime-1 && temp%2 == 0){
            return false;
        }
    }
    return true; }
```

Prime factorization

Prime factorization has an exponential lower bound and has therefor a lower bound of $O(b^k)$. However there are
Trial division easily implements prime factorization.
First you need to have a list of prime numbers, which you can calculate through a prime sieve or through the Miller-Rabin test.
Then you want to try dividing your number by the first prime number. Let's take 12 for example. Can 12 be divided by 2? Yes, so add 2 to the list. 6 can still be divided by 2, add 2 another time. 3 can divide 3, we are done, the prime factors are:
2 * 2 * 3 = 12.

```
vector<ull> primeFactorization(ull n) {
    vector<ull> primes = primeSieve(n);
    vector<ull> primeFactors;
    for(int i = 0; i < primes.size(); ++i) {
        while(n % primes[i] == 0) {
            primeFactors.push_back(primes[i]);
            n /= primes[i];
        }
    }
    return primeFactors;
}
```

If you want to calculate multiple prime factors, you should keep track of the prime factors of previous steps. After factoring 12, you not only know the prime factors of 12, but also those of 6 and 3.

Derivative

The differential at a point is basically the derivative of a point at a function.

Axiom:

$$f'(x_0) = \lim_{x_0 \to x} \frac{f(x)-f(x_0)}{x-x_0} = \lim_{h \to 0} \frac{f(x_0+h)-f(x_0)}{h}$$

Other Abbreviations:

$$f'(K) = \frac{d}{dx} f(K) = \dot{K} = D\, f(K)$$

$$f''(K) = \frac{d^2}{dx^2} f(K) = \ddot{K} = D^2\, f(K)$$

$$Generally: \square^{(n)}(K) = f^{\prime\prime\prime\prime\prime\prime \ldots n-times} = \frac{d^n}{dx^n} f(K) = \overset{\ldots\ldots n-times}{K}$$
$$= D^n\, f(K)$$

Rules:

Multidimensional functions have multiple derivations.

Given a function f(x, y)

$$f'_x = \lim_{h \to 0} \frac{f(x_0 + h, y_0) - f(x_0, y_0)}{h}$$

$$f'_y = \lim_{h \to 0} \frac{f(x_0, y_0 + h) - f(x_0, y_0 + h)}{h}$$

Sum rule:

$$f(x) = u(x) + v(x)\,;\; f'(x) = u'(x) + v'(x)$$
$$f(x) = u(x) - v(x)\,;\; f'(x) = u'(x) - v'(x)$$
$$(u \pm v)' = u' \pm v'$$

Factor rule:

$$f(x) = c * u(x)\,; f'(x) = c * u'(x);\; c \in \mathbb{R}$$
$$(cu)' = cu'$$

Product rule:

$$f(x) = u(x) * v(x)\,;\; f'(x) = u'(x) * v(x) + u'(x) * v(x);$$
$$(uv)' = u'v + uv'$$

Quotient rule:

$$f(x) = \frac{u(x)}{v(x)}; f'(x) = \frac{u'(x) * v(x) - u(x) * v'(x)}{\left(v(\square)\right)^2}\,;$$

$$\left(\frac{u}{v}\right)' = \frac{u'v - uv'}{v^2}$$

Power rule:
$f(x) = x^n; f'(x) = n * x^{n-1}$

Chain rule:
$f(x) = u\big(v(x)\big); f'(x) = u'\big(v(x)\big) * v'(x)$
$(u \circ v)' = (u' \circ v)\, v'$

Inverse rule:
If f(g(x)) = x and g(f(y)) = y then:
$f'(x) = \dfrac{1}{g'(f(x))}$

Differentiate:
$f(x) = a^x; then: \lim_{h\to 0} \dfrac{a^{x+h} - a^x}{h} = \lim_{h\to 0} \dfrac{a^x * a^h - a^x}{h}$
$= \lim_{h\to 0} \dfrac{a^x * (a^h - 1)}{h} = a^x * \lim_{h\to 0} \dfrac{a^h - 1}{h} = a^x * f'(0)$
$1 = \lim_{h\to 0} \dfrac{e^h - 1}{h} \to e = \lim_{h\to \infty} \left(1 + \dfrac{1}{h}\right)^h$
$f(x) = e^x = f'(x) = f^{(n)}(x)$
$a^x = b;\ \ then\ x = \log_a b$
$e^{\ln(x)} = \ln(e^x) = x = f\big(g(x)\big) = g\big(f(x)\big)$
Now using the inverse function we can differentiate $\mathrm{f(x)} = \ln(x)$.
$f'(x) = \dfrac{1}{e^{\ln(x)}} = \dfrac{1}{x}$
If you recall correctly any logarithm can be calculated using the following method:
$f(x) = \log_a x = \dfrac{\ln(x)}{\ln(a)}$
Using the quotient rule the above equation can be derivated into:
$f'(x) = \dfrac{1}{x * \ln(a)}$
$f(x) = a^x = \left(e^{\ln(a)}\right)^{\square} = e^{x*\ln(a)}$

Using the chain rule:
$f(x) = e^{g(x)};\qquad f'(x) = e^{g(x)} * g'(x)$
$f(x) = e^{x*\ln(a)}$
$f'(x) = e^{x*\ln(a)} * \ln(a) = a^x * \ln(a)$

Trigonometric functions

Using the squeeze theorem it can be proven that:

$$\lim_{h\to 0}\frac{\sin(h)}{h}=1\,;and\ therefor\ \lim_{h\to 0}\frac{\cos(h)-1}{h}=0$$

$$\mathrm{f(x)}=\sin(x)$$

$$f'(x)=\lim_{i\to 0}\frac{\sin(x+h)-\sin(x)}{h}$$

$$\sin(x+h)=\sin(x)*\cos(h)+\sin(h)*\cos(x)$$

$$f'(x)=\lim_{i\to 0}\frac{\sin(x)*\cos(h)+\sin(h)*\cos(x)-\sin(x)}{h}$$

$$f'(x)=\lim_{i\to 0}\frac{\sin(x)*(\cos(h)-1)+\sin(h)*\cos(x)}{h}$$

$$f'(x)=\sin(x)*\lim_{i\to 0}\frac{\cos(h)-1}{h}+\cos(x)*\lim_{i\to 0}\frac{\sin(\mathrm{h})}{h}$$

$$f'(x)=\cos(\mathrm{x})\,;\ \mathrm{f(x)}=\sin(\mathrm{x})$$

$$f'(x)=-\sin(\mathrm{x})\,;\ \mathrm{f(x)}=\cos(\mathrm{x})$$

$$f'(x)=\tan(\mathrm{x})\,;\ \mathrm{f(x)}=\left(\frac{1}{\cos\,(x)}\right)^2$$

Euler-Cauchy iteration method: $x(t+\ \Delta t)\approx x(t)+\Delta t*x'(t)$

Integral

$$\int f'^{(x)}dx=f(x)+c$$

$$\int f(x)dx=F(x)+\ c$$

$$\int a*x^n\,dx=\ \frac{a}{n+1}*x^{n+1}+c$$

$$\int_s^t f(x)dx=F(x)\Big|_s^t=(F(t)+c)-(F(s)\ +c)=F(t)-F(s)$$

$$\int_s^u f(x)dx=\ \int_s^t f(x)dx+\ \int_t^u f(x)dx\ \,;when\ s\le t\le u$$

$$\int_s^t (f(x)+g(x))dx=\ \int_s^t f(x)dx+\ \int_s^t g(x)dx$$

$$\int_s^t c*f(x)\,dx=\ c*\int_s^t f(x)\,dx$$

$$\int f(g(x))=\frac{1}{g'(x)}*F(g(x))$$

$$\int_s^t f(x)*g(x)\,dx=(f(x)*G(x))\Big|_s^t-\int_s^t f'(x)*G(x)\,dx$$

$$\bar{f}_s^t=\ \frac{1}{t-s}*\int_s^t f(x)\,dx$$

Special cases:

$$\int a * e^{f(x)}\, dx = \frac{a}{f'(x)} * e^{f(x)} + c$$

$$\int x^{-1}\, dx = \ln(abs(x)) + c\ ; when\ x\ \neq 0$$

$$\int \sin(x) * \cos(x)\, dx = sin^2(x) + c - \int \cos(x) * \sin(x)\, dx$$

$$\left| + \int \cos(x) * \sin(x) \right.$$

$$\int \sin(x) * \cos(x)\, dx = \frac{sin^2(x)}{2} + c$$

Or via addition theorem:

$$2 * \sin(x) * \cos(x) = \sin(2x)$$

$$\int \sin(x) * \cos(x)\, dx = \int \frac{1}{2}\sin(2x)\, dx = \frac{1}{4}\cos(2x) + c$$

Solving Polynomials

You can solve up to degree 4 polynomials (quartic polynomials) in an algebraic way. Because the quartic polynomial is a huge equation, let's only write down the equation for degree 2.

$$a * x^2 + b * x + c = 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a * b}$$

Example: a=-2, b=5, c=3; x_1 = -0.5, x_2 = 3

A few equations have a negative value under the root and turn complex. An example would be:

a=2, b=4, c=4; x_1=-1+i, x_2=-1-i

Newton's Method

Solving a n-degree polynomial to be equal to zero can be done using Newtons approximation. $x_{n+1} = x_n * \frac{f(x_n)}{f'(x_n)}$

It however only returns one solution.

```
struct polynomial {
    float a, e;
    polynomial(float _a, float _e) : a(_a), e(_e) {}
    void derivate() {
        if(e == 0) a = 0;
        else {
            a *= e;
            e--;
        }
    }
    void integrate() {
        if(e == -1) polynomial = ln(x);
        else {
            ++e;
            a /= e;
        }
    }
};

float newtonApprox(vector<polynomial> equation, float x) {
    vector<polynomial> derivative = equation;
    for(int i = 0; i < derivative.size(); ++i) derivative[i].derivate();
    float y1 = 0, y2 = 0;
    for(polynomial& elm : equation) y1 += elm.a * pow(x, elm.e);
    for(polynomial& elm : derivative) y2 += elm.a * pow(x, elm.e);
    if(y2 == 0) return x - y1;
    return x - (y1 / y2);
}

float solveEquation (vector<polynomial> equation, float precision) {
    float approx = 1;
    float pastApprox = MAX_FLOAT;
    while(abs(approx - pastApprox) > precision) {
        pastApprox = approx;
        approx = newtonApprox(equation, approx);
    }
    return approx;
}
```

Root Algorithm using Newton's Method

Calculating a root can also be done using Newton's method.
Solving the equation $f(x) = x^2 - 5 = 0$ will result into $x = \sqrt{5}$
For a general n-th root simply calculate:
$f(x) = x^d - r = 0$ will result into $x = \sqrt[d]{r}$

```
float sqrt(float degree, float radicand, float precision) {
   vector<polynomial> equ = {polynomial(1, degree), polynomial(
-radicand, 0)};
   return solveEquation(equ, precision);
}
```

Other numerical householder algorithms exist:
Halley's method: $x_{n+1} = x_n - \frac{2*f(x_n)*f'(x_n)}{2*[f'(x_n)]^2 - f(x_n)*f''(x_n)}$
In order to get different results x_0 (approx) should be different every time. It is also useful to calculate other irrational numbers such as pi or the golden ratio:
$x = \pi;\ f(x) = 1 + cos(x) = 0\ and\ approx = 3$
$x = \varphi = \frac{\sqrt{5}+1}{2};\ f(x) = x^2 - x - 1 = 0\ and\ approx = 1$

Matrices

Matrices are 2 dimensional containers of data, which are used incredibly often in mathematics to compact formulas, hence mathematicians prefer single step equations.
They have various advantages, which will be explained later.
For the matrix implementation, we'll use the following class, where matrix.field is the matrix.

```
struct matrix {
    vector<vector<float> > field;
    matrix() {}
    matrix(vector<vector<float> > _field) {
        field = _field;
    }
};
```

Terms & Notions

Matrices are usually written in box brackets. This is a normal matrix:

$$\begin{bmatrix} 2 & 5 \\ 4 & 1 \end{bmatrix}$$

Element inside the matrix at i, j: $a_{i,j}$

Identity matrix = $I_n = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$

Vector = 1-D Matrix

Row vector = 1 x N = $[a_{1,1} \quad \dots \quad a_{1,N}]$

Column vector = N x 1 = $\begin{bmatrix} a_{1,1} \\ \vdots \\ a_{N,1} \end{bmatrix}$

Square matrix = N x N = $\begin{bmatrix} a_{1,1} & \cdots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{N,1} & \cdots & a_{N,N} \end{bmatrix}$

Defining a matrix = $\begin{bmatrix} a_{1,1} & \cdots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{M,1} & \cdots & a_{M,N} \end{bmatrix} \in \mathbb{R}^{MxN}; M, N \in \mathbb{N}$

All matrices belong to the set $\mathbb{M}$.

Square matrices:

Matrix A is invertible if there exists B, such that: $A * B = B * A$

Inverse Matrix A is A^{-1} if: $A * \square^{-1} = A^{-1} * A = I_n$

Transpose Matrix: A^T *(more later)*

Orthogonal Matrix: $A^T = A^{-1}$

Complex numbers can be represented in a 2x2 matrix:

$a + bi \leftrightarrow \begin{bmatrix} a & -b \\ b & a \end{bmatrix}$

A minor is a matrix, which got one column and one row removed. It is also called a cofactor matrix.

If a matrix A gets multiplied by a permutation matrix P, the rows of A get swapped. P always has the size of MxM. It is best visualized by example:

$$\begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} e_4 \\ e_2 \\ e_1 \\ e_5 \\ e_3 \end{bmatrix}$$

Matrix addition & subtraction

$A \pm B = R\ ;\ where\ A, B, R \in \mathbb{Z}$

$R_{i,j} = A_{i,j} + B_{i,j}$

$A + B = B + A$

$(A + B) + C = A + (B + C)$

$A - B \neq B + A$

$(A - B) - C \neq A - (B - C)$

Implementation note: Adding two matrices must require the dimensions to be the same size. However in this implementation, it will simply scale the smaller matrix by adding rows or columns of zeros.

```
matrix operator+(const matrix& lhs, const matrix& rhs) {
    matrix output;
    for(int i = 0; i < max(lhs.field.size(), rhs.field.size()); ++i) {
        vector<float> row;
        for(int j = 0; j < max(lhs.field[i].size(), rhs.field[i].size()); ++j) {
            row.push_back(lhs.field[i][j] + rhs.field[i][j]);
        }
        output.field.push_back(row);
    }
    return output;
}

matrix operator-(const matrix& lhs, const matrix& rhs) {
    matrix output;
    for(int i = 0; i < max(lhs.field.size(), rhs.field.size()); ++i) {
        vector<float> row;
        for(int j = 0; j < max(lhs.field[i].size(), rhs.field[i].size()); ++j) {
            row.push_back(lhs.field[i][j] - rhs.field[i][j]);
        }
        output.field.push_back(row);
    }
    return output;
}
```

Matrix scalar multiplication & division

$s * M = R; \quad where\ M, R \in \mathbb{M}, \quad s \in \mathbb{R}$

$R_{i,j} = s * M_{i,j}$

$\frac{M}{s} = \frac{1}{s} * M$

```
matrix operator*(float lhs, matrix rhs) {
    matrix output = rhs;
    for(int i = 0; i < output.field.size(); ++i) {
        for(int j = 0; j < output.field[i].size(); ++j) {
            output.field[i][j] *= lhs;
        }
    }
    return output;
}

matrix operator/(matrix lhs, float rhs) {
    return (1/rhs) * lhs;
}
```

Matrix Transpose

$A_{i,j} = A^T{}_{j,i}; \quad where\ A, A^T \in \mathbb{M}$

```
matrix transpose(const matrix& input) {
    matrix output;
    output.field.resize(input.field[0].size(),
vector<float>(input.field.size()));
    for(int i = 0; i < input.field.size(); ++i) {
        for(int j = 0; j < input.field[i].size(); ++j) {
            output.field[j][i] = input.field[i][j];
        }
    }
    return output;
}
```

Matrix multiplication

$A * B = R\,; \quad where\ M, R \in \mathbb{M}, \qquad s \in \mathbb{R}$

$$R_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$$

$A * B \neq B * A$

$(A * B) * C = A * (B * C)$ *: This allows fast chain multiplication.*

$\frac{A}{B} = A * B^{-1}$

In a 2-dimensional context:

$$\begin{bmatrix} 2 & 5 \\ 3 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 3 \\ 4 & 7 \end{bmatrix} = \begin{bmatrix} 22 & 41 \\ 7 & 16 \end{bmatrix}$$

1. Row of A multiplied with 1. Colon of B = a_{11} 2×1+5×4 = 22
1. Row of A multiplied with 2. Colon of B = a_{12} 2×3+5×7 = 41
2. Row of A multiplied with 1. Colon of B = a_{21} 3×1+1×4 = 7
2. Row of A multiplied with 2. Colon of B = a_{22} 3×3+1×7 = 16

The standard implementation requires $O(N^3)$ multiplications.

```
matrix operator*(const matrix& lhs, const matrix& rhs) {
   if(lhs.field[0].size() != rhs.field.size()) return error;

   matrix output;
   for(int li = 0; li < lhs.field.size(); ++li) {
      vector<float> row;
      for(int rj = 0; rj < rhs.field[0].size(); ++rj) {
         float sum = 0;
         for(int it = 0; it < rhs.field.size(); ++it) {
            sum += lhs.field[li][it] * rhs.field[it][rj];
         }
         row.push_back(sum);
      }
      output.field.push_back(row);
   }

   return output;
}
```

Strassen's matrix multiplication
Strassen's matrix multiplication algorithm is like Karatsuba's multiplication algorithm, as it is a divide & conquer algorithm, which saves one multiplication per iteration. In order to calculate a NxN matrix, the algorithm calculates 7 N/2 x N/2 matrices and adds them together.

For simplicity sake, the algorithm implemented here calculates matrices of multiple of 2's. So other matrices get columns and rows of zeros added to them, so they become a matrix multiple of 2. Afterwards it crops the zeros.

```
matrix strassen(const matrix &a, const matrix &b) {
    int aYSize = a.field.size(), aXSize = a.field[0].size();
    int bYSize = b.field.size(), bXSize = b.field[0].size();

    if(aYSize == 1 && aXSize == 1 && bYSize == 1 && bXSize == 1) {
        matrix r( {a.field[0][0] * b.field[0][0]} );
        return r;
    }

    int xSize = 1, ySize = 1;
    while(xSize < aXSize || xSize < bXSize) xSize *= 2;
    while(ySize < aYSize || ySize < bYSize) ySize *= 2;
    while(xSize < ySize) xSize *= 2;
    while(ySize < xSize) ySize *= 2;
    xSize /= 2; ySize /= 2;
    vector<vector<float> > init(ySize, vector<float>(xSize, 0));
    matrix a11(init), a12(init), a21(init), a22(init), b11(init), b12(init),
b21(init), b22(init);

    for(int i = 0; i < aYSize; ++i) {
        for(int j = 0; j < aXSize; ++j) {
            if(i < ySize && j < xSize)   a11.field[i][j] = a.field[i][j];
            if(i < ySize && j >= xSize)  a12.field[i][j-xSize] = a.field[i][j];
            if(i >= ySize && j < xSize)  a21.field[i-ySize][j] = a.field[i][j];
            if(i >= ySize && j >= xSize) a22.field[i-ySize][j-xSize] = a.field[i][j];
        }
    }
```

```
    for(int i = 0; i < bYSize; ++i) {
        for(int j = 0; j < bXSize; ++j) {
            if(i < ySize && j < xSize)   b11.field[i][j] = b.field[i][j];
            if(i < ySize && j >= xSize)  b12.field[i][j-xSize] = b.field[i][j];
            if(i >= ySize && j < xSize)  b21.field[i-ySize][j] = b.field[i][j];
            if(i >= ySize && j >= xSize) b22.field[i-ySize][j-xSize] = b.field[i][j];
        }
    }
    matrix p1 = strassen((a11 + a22), (b11 + b22));
    matrix p2 = strassen((a21 + a22), b11);
    matrix p3 = strassen(a11, (b12 - b22));
    matrix p4 = strassen(a22, (b21 - b11));
    matrix p5 = strassen((a11 + a12), b22);
    matrix p6 = strassen((a21 - a11), (b11 + b12));
    matrix p7 = strassen((a12 - a22), (b21 + b22));

    matrix r(vector<vector<float> >(aYSize, vector<float>(bXSize)));
    matrix r11 = p1 + p4 - p5 + p7;
    matrix r12 = p3 + p5;
    matrix r21 = p2 + p4;
    matrix r22 = p1 - p2 + p3 + p6;

    for(int i = 0; i < aYSize; ++i) {
        for(int j = 0; j < bXSize; ++j) {
            if(i < ySize && j < xSize)   r.field[i][j] = r11.field[i][j];
            if(i < ySize && j >= xSize)  r.field[i][j] = r12.field[i][j-xSize];
            if(i >= ySize && j < xSize)  r.field[i][j] = r21.field[i-ySize][j];
            if(i >= ySize && j >= xSize) r.field[i][j] = r22.field[i-ySize][j-xSize];
        }
    }

    return r;
}
```

Gaussian Elimination

One use of matrices in mathematics is to describe a linear equation. Gaussian Elimination has a complexity of $O(N^3)$.

$2x + y - z = 8$

$-3x - y + 2z = -11$ → becomes $\left(\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 0 & 2 & -6 \end{array}\right)$

$-2x + 0y + 2z = -6$

Of course the solution gets returned as a matrix aswell:

$x = 2, y = 3, z = -1, result = \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix}$

Gaussian elimination reduces every equation by one variable, by multiplying the equation so that one variable becomes 0 * x, until one variable has been found out and then reverses the process.

DON'T USE THIS IMPLEMENTATION. USE THE NEXT ONE.

```
matrix gaussianElimination(const matrix& input) {
    int ySize = input.field.size(), xSize = input.field[0].size();
    if(ySize + 1 != xSize) return error;

    vector<matrix> rows;
    for(int i = 0; i < input.field.size(); ++i) {
        matrix row( {input.field[i]} );
        rows.push_back(row);
    }

    for(int i = 0; i < rows.size(); ++i) {
        for(int j = i + 1; j < rows.size(); ++j) {
            float scale = rows[j].field[0][i] / rows[i].field[0][i];
            rows[j] = rows[j] - (scale * rows[i]);
        }
    }

    matrix solution(vector<vector<float> >(rows.size(), vector<float>(1,
0)));
    for(int i = rows.size() - 1; i >= 0; --i) {
        float sum = rows[i].field[0][rows.size()];
        for(int j = rows.size() - 1; j > i; --j) {
            sum -= solution.field[j][0] * rows[i].field[0][j];
        }
        solution.field[i][0] = sum / rows[i].field[0][i];
    }
    return solution;
}
```

Gaussian Elimination using Pivoting

Now the previous implementation only works, if there is no division by zero.

$\begin{bmatrix} 0 & 15 & 3 \\ 28 & 7 & 2 \\ 5 & 1 & 0 \end{bmatrix}$ This directly results in a division by zero error.

To solve this we have to understand what solving a Gaussian matrix really is. It can also be represented as a multiplication of 2 matrices A * X = solution, which we have to solve for X.

This section is only dedicated to solution and X as a vector.

The previous equation can therefor also be represented as:

$$\begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 0 & 2 \end{bmatrix} * X = \begin{bmatrix} 8 \\ -11 \\ -6 \end{bmatrix}, solution\ X\ is = \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix}$$

Now to solve the division by zero problem we use a permutation matrix. A * x = B, becomes P * A * x = P * B. The permutation matrix allows us to swap rows with each other (check notation section). Cleverly done using pivoting this can yield the following result:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 15 & 3 \\ 28 & 7 & 2 \\ 5 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 28 & 7 & 2 \\ 5 & 1 & 0 \\ 0 & 15 & 3 \end{bmatrix} = PA$$

Now it is possible to solve the Gaussian elimination and finally the swapped rows, must be reversed in the solution again.

BTW not all equations are solvable. If determinant(A) = 0 it isn't.

If you don't want to cram the equation into one matrix, you can use the following function to do so. A * X = S

```
matrix gaussianElimination(matrix A, matrix S) {
    if(S.field[0].size() != 1) return generalGauss(A, S);
    for(int i = 0; i < S.field.size(); ++i) {
        A.field[i].push_back(S.field[i][0]);
    }
    return gaussianElimination(A);
}
```

```
matrix gaussianElimination (const matrix& input) {
    int ySize = input.field.size(), xSize = input.field[0].size();
    if(ySize + 1 != xSize) return error;

    vector<matrix> rows;
    for(int i = 0; i < input.field.size(); ++i) {
        matrix row( {input.field[i]} );
        rows.push_back(row);
    }

    for(int i = 0; i < rows.size(); ++i) {
        //pivoting
        float pivot = -1;
        int pivotId = i;
        for(int j = i; j < rows.size(); ++j) {
            if(pivot < abs(rows[j].field[0][i]) ) {
                pivot = abs(rows[j].field[0][i]);
                pivotId = j;
            }
        }
        if(pivot == 0) return error; //matrix is singular
        swap(rows[i], rows[pivotId]);

        for(int j = i + 1; j < rows.size(); ++j) {
            float scale = rows[j].field[0][i] / rows[i].field[0][i];
            rows[j] = rows[j] - (scale * rows[i]);
        }
    }

    matrix solution(vector<vector<float> >(rows.size(), vector<float>(1,
0)));
    for(int i = rows.size() - 1; i >= 0; --i) {
        float sum = rows[i].field[0][rows.size()];
        for(int j = rows.size() - 1; j > i; --j) {
            sum -= solution.field[j][0] * rows[i].field[0][j];
        }
        solution.field[i][0] = sum / rows[i].field[0][i];
    }
    return solution;
}
```

General Gauss Elimination

Using a small trick we cannot only calculate the gauss elimination for vectors, but also for matrices of any order.
For example we have to solve the following system:

$$\begin{bmatrix} -5 & -4 & 4 \\ -1 & -1 & 1 \\ -4 & 2 & 2 \end{bmatrix} * X = \begin{bmatrix} 1 & 12 & 3 \\ -1 & 1 & 8 \\ 7 & 4 & 1 \end{bmatrix}; X = \begin{bmatrix} -5 & -8 & 29 \\ -0.25 & -3.5 & 10.75 \\ -6.25 & -10.5 & 47.75 \end{bmatrix}$$

Any matrix problem, can be reduced into a vector problem, by actually calculating every single number and make it an equation for itself. So you end up with an equation like:
$A_{11} * ?_1 + A_{12} * ?_2 + A_{13} * ?_3 + 0 * ?_4 \ldots = S_{11}$

The question marks then only have to turn back into a matrix, which has been done in the last 3 lines of the implementation.

```
matrix generalGauss(const matrix& A, const matrix& S) {
    matrix output, gaussMatrix, gaussVariables;
    output.field.resize(A.field.size(), vector<float>(S.field[0].size(),0));

    for(int li = 0; li < A.field.size(); ++li) {
        for(int rj = 0; rj < S.field[0].size(); ++rj) {
            vector<float> row(A.field.size() * A.field[0].size(), 0);
            for(int lj = 0; lj < A.field[0].size(); ++lj) {
                row[lj * S.field[0].size() + rj] = A.field[li][lj];
            }
            row.push_back(S.field[li][rj]);
            gaussMatrix.field.push_back(row);
        }
    }

    matrix solution = gaussianElimination(gaussMatrix);
    for(int i = 0; i < solution.field.size(); ++i) {
        output.field[i/S.field[0].size()][i%S.field[0].size()] = solution.field[i][0];
    }
    return output;
}
```

Gauss-Seidel / Jacobi algorithm

In order to calculate the Gaussian elimination iteratively, one can use the Jacobi equation:

$$x_i^{(m+1)} = \frac{1}{a_{i,i}} * \left(b_{i,1} - \sum_{j \neq i} a_{i,j} * x_j^m \right)$$

The Jacobi method always uses the values of the previous iteration, while the Gauss-Seidel method always uses the most recent value. Gauss-Seidel is easier to implement and is faster, but is harder to write down as a mathematical equation.

The bottom implementation solves a * x = b

$A * x = \vec{b}$; $wher\square A \in \mathbb{R}^{N,N}$ and $b \in \mathbb{R}^{M,1}$

```
matrix gaussSeidel(matrix &a, matrix &b, int iterations) {
    int ySize = a.field.size(), xSize = a.field[0].size();
    if(ySize != xSize) return error;

    matrix output = b;
    for(int it = 0; it < iterations; ++it) {
        for(int i = 0; i < output.field.size(); ++i) {
            float result = b.field[i][0];
            for(int j = 0; j < a.field[i].size(); ++j) {
                if(i != j) result -= a.field[i][j] * output.field[j][0];
            }
            output.field[i][0] = (result / a.field[i][i]);
        }
    }
    return output;
}
```

The same method from the general Gaussian elimination can be used to make the iterative algorithm general.

Trace

$trace(A);\quad where\ A \in \mathbb{R}^{N,N},\qquad trace(A) \in \mathbb{R};$

$$trace(A) = \sum_{i=1}^{N} a_{i,i} + a_{i,N-i+1}$$

$$trace(A * B) = trace(B * A)$$

```
float trace(const matrix& input) {
    if(input.field.size() != input.field[0].size()) return NULL;
    float sum = 0;
    for(int i = 0; i < input.field.size(); ++i) {
        sum += input.field[i][i] + input.field[i][input.field.size() - i - 1];
    }
    return sum;
}
```

Determinant with La-Place method

$determinant(A);\quad where\ A \in \mathbb{R}^{N,\square},\qquad determinant(A) \in \mathbb{R};$

$$determinant(A) = \sum_{i=1}^{N} a_{1,i} * determinant(minor_{1,i})$$

$$determinant(L * U) = determinant(L) * determinant(U);$$

NEVER use this method, because it takes O(N!) steps to complete.
Instead use the Gaussian elimination method.

```
float determinant(const matrix& input) {
    if(input.field.size() != input.field[0].size()) return NULL;
    if(input.field.size() == 1) return input.field[0][0];
    float sum = 0;
    for(int k = 0; k < input.field[0].size(); ++k) {
        matrix recursiveDet;
        for(int i = 1; i < input.field.size(); ++i) {
            vector<float> row;
            for(int j = 0; j < input.field[0].size(); ++j) {
                if (k != j) row.push_back(input.field[i][j]);
            }
            recursiveDet.field.push_back(row);
        }
        if(k % 2 == 0) sum += input.field[0][k] * determinant(recursiveDet);
        else sum -= input.field[0][k] * determinant(recursiveDet);
    }
    return sum;
}
```

Determinant with Gaussian Elimination

Calculating the determinant of a matrix can be done using Gaussian elimination with pivoting. For every time the row gets swapped, the end result has to be multiplied by -1. The end result is the last element in the matrix, on which the elimination has been performed on. It has the complexity of $O(N^3)$.

```
float determinant(const matrix& input) {
    int ySize = input.field.size(), xSize = input.field[0].size();
    if(ySize != xSize) return 0;

    vector<matrix> rows;
    for(int i = 0; i < input.field.size(); ++i) {
        matrix row( {input.field[i]} );
        rows.push_back(row);
    }

    int swapCounter = 1;
    for(int i = 0; i < rows.size(); ++i) {
        float pivot = -1;
        int pivotId = i;
        for(int j = i; j < rows.size(); ++j) {
            if(pivot < abs(rows[j].field[0][i]) ) {
                pivot = abs(rows[j].field[0][i]);
                pivotId = j;
            }
        }
        if(pivot == 0) return 0; //matrix is singular
        swap(rows[i], rows[pivotId]);
        if(i != pivotId) swapCounter *= -1;

        for(int j = i + 1; j < rows.size(); ++j) {
            float scale = rows[j].field[0][i] / rows[i].field[0][i];
            rows[j] = rows[j] - (scale * rows[i]);
        }
    }
    float result = 1;
    //multiply diagonal
    for(int i = 0; i < rows.size(); ++i) {
        result *= rows[i].field[0][i];
    }
    result *= swapCounter;
    return result;
}
```

Identity Matrix

This generates I_N , for example $I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

```
matrix identityMatrix(int N) {
    matrix output;
    output.field.resize(N, vector<float>(N, 0));
    for(int i = 0; i < N; ++i) {
        for(int j = 0; j < N; ++j) {
            if(i == j) output.field[i][j] = 0;
        }
    }
    return output;
}
```

Inverse using Gauss

$A * A^{-1} = A^{-1} * A = I_n; \quad where\ A, A^{-1}, I_n \in \mathbb{R}^{N,N}$

$$A^{-1} = \frac{adjugate(A)}{determinant(A)}$$

Calculating the inverse of a matrix can be done using the general Gaussian elimination. We only have to generate I_N to use it.

```
matrix inverse(const matrix& a) {
    return gaussianElimination(a, identityMatrix(a.field.size()));
}
```

Vector normalizing

$normalize(A); \quad where\ A \in \mathbb{R}^{1,N} or\ \mathbb{R}^{N,1}$

```
float normalize(matrix& a) {
    float norm = 0;
    for(int i = 0; i < a.field.size(); ++i) {
        for(int j = 0; j < a.field[i].size(); ++j) {
            norm += a.field[i][j] * a.field[i][j];
        }
    }
    return sqrt(norm);
}
```

Condition numbers

$condi\square ionNumber(A) = normalize(A) * normalize(A^{-1})$

Lower-Upper decomposition (LU)

$A = L * U\ ; \quad where\ A, L, U\ \in\ \mathbb{R}^{N,N}$

LU-decomposition has a simple task, which is to find 2 matrices L and U which hold: L * U = A, where A is the matrix to decompose. The Doolittle algorithm is simply a slight modification of the Gauss-Algorithm with pivoting. The bottom implementation doesn't have pivoting, for the pivoting approach see PLU decomposition.

```
struct LU {
    matrix L, U;
    LU(matrix _L, matrix _U) : L(_L), U(_U) {}
};
LU LUDecomposition (const matrix& input) {
    int ySize = input.field.size(), xSize = input.field[0].size();
    if(ySize != xSize) return error;
    vector<matrix> rows;
    for(int i = 0; i < input.field.size(); ++i) {
        matrix row( {input.field[i]} );
        rows.push_back(row);
    }

    vector<vector<float> > scaleList(ySize, vector<float>(xSize, 0));
    for(int i = 0; i < rows.size(); ++i) {
        for(int j = i + 1; j < rows.size(); ++j) {
            float scale = rows[j].field[0][i] / rows[i].field[0][i];
            scaleList[j][i] = scale;
            rows[j] = rows[j] - (scale * rows[i]);
        }
    }
    matrix L, U;
    L.field.resize(ySize, vector<float>(xSize, 0));
    for(int i = 0; i < rows.size(); ++i) {
        U.field.push_back(rows[i].field[0]);
        L.field[i][i] = 1;
    }
    for(int i = 0; i < ySize; ++i) {
        for(int j = 0; j < xSize; ++j) {
            if(scaleList[i][j] != 0) {
                L.field[i][j] = scaleList[i][j];
            }}}
     return LU(L,U);
}
```

Lower-upper decomposition with pivoting (PLU)

$P * A = L * U\ ; \quad where\ P, A, L, U\ \in\ \mathbb{R}^{N,N}$

```
struct PLU {
    matrix P, L, U;
    PLU(matrix _P, matrix _L, matrix _U) : P(_P), L(_L), U(_U) {}
};
PLU LUDecompositionwPivot (const matrix& input) {
    PLU erro(error,error,error);

    int ySize = input.field.size(), xSize = input.field[0].size();
    if(ySize != xSize) return error;

    vector<matrix> rows;
    for(int i = 0; i < input.field.size(); ++i) {
        matrix row( {input.field[i]} );
        rows.push_back(row);
    }

    matrix P(vector<vector<float> >(ySize, vector<float>(xSize, 0)));
    for(int i = 0; i < ySize; ++i) P.field[i][i] = 1;

    vector<vector<float> > scaleList(ySize, vector<float>(xSize, 0));

    for(int i = 0; i < rows.size(); ++i) {
        //pivoting
        float pivot = -1;
        int pivotId = i;
        for(int j = i; j < rows.size(); ++j) {
            if(pivot < abs(rows[j].field[0][i]) ) {
                pivot = abs(rows[j].field[0][i]);
                pivotId = j;
            }
        }
        if(pivot == 0) return error; //matrix is singular
        swap(rows[i], rows[pivotId]);
        swap(P.field[i], P.field[pivotId]);
        swap(scaleList[i], scaleList[pivotId]);

        for(int j = i + 1; j < rows.size(); ++j) {
            float scale = rows[j].field[0][i] / rows[i].field[0][i];
            scaleList[j][i] = scale;
            rows[j] = rows[j] - (scale * rows[i]);
        }
    }
```

```
    matrix L, U;
    L.field.resize(ySize, vector<float>(xSize, 0));
    for(int i = 0; i < rows.size(); ++i) {
        U.field.push_back(rows[i].field[0]);
        L.field[i][i] = 1;
    }

    for(int i = 0; i < ySize; ++i) {
        for(int j = 0; j < xSize; ++j) {
            if(scaleList[i][j] != 0) {
                L.field[i][j] = scaleList[i][j];
            }
        }
    }

    PLU output(P,L,U);
    return output;
}
```

Matrix exponentiation

Because matrix multiplication follows the rule of associativity, fast matrix exponentiation is possible in log(N), the same way as it was possible with scalar exponentiation. An even more interesting topic in this area is matrix chain multiplication (MCM).

```
matrix pow(matrix &a, int b) {
    vector<matrix> stored;
    stored.push_back(a);
    int it = 2;
    while(it < b) {
        stored.push_back(stored.back() * stored.back());
        it *= 2;
    }
    it /= 2;
    matrix solution = stored.back();
    b -= it;
    while(b != 0) {
        if(b >= it) {
            solution = solution * stored.back();
            b -= it;
        }
        stored.pop_back();
        it /= 2;
    }
    return solution;
}
```

Matrix Division

Matrix division is not defined in matrix mathematics, however many people use the following definition:

$$\frac{A}{B} = A * B^{-1}; \qquad where\ A, B \in \mathbb{R}^{NxN}$$

```
matrix operator/(matrix a, matrix b) {
    return a * inverse(b);
}
```

Vectorial Transformations

A vector can be transformed using for example scaling & rotation. This section tries to create a general rule for linear transformations. In the graphics section of this book examples can be seen of translating simple points. Vectors commonly have arrows on top of them, for example: $\vec{X}$. If $\vec{X}$ is the point, which has to get transformed it can be followed using a general equation:

$f(\vec{X}) = M * \vec{X} + \vec{V} - M * \vec{V}$

M: Transformation matrix

$\vec{V}$: Translation matrix

$\vec{X}$: The point vector to be moved.

A few transformation matrices:

$$Translation = \begin{bmatrix} transX \\ transY \end{bmatrix}$$

$$Scaling = \begin{bmatrix} k_x & 0 \\ 0 & k_y \end{bmatrix} where\ k_x\ is\ the\ scale\ factor\ in\ the\ x\ direction.$$

$$Rotation = \begin{bmatrix} \cos(\propto) & -\sin(\propto) \\ \sin(\propto) & \cos(\propto) \end{bmatrix} where\ \propto is\ the\ rotation\ angle.$$

$$Rotation_x(3D) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\propto) & -\sin(\propto) \\ 0 & \sin(\propto) & \cos(\propto) \end{bmatrix}$$

$$Rotation_y(3D) = \begin{bmatrix} \cos(\propto) & 0 & \sin(\propto) \\ 0 & 1 & 0 \\ -\sin(\propto) & 0 & \cos(\propto) \end{bmatrix}$$

$$Rotation_z(3D) = \begin{bmatrix} \cos(\propto) & -\sin(\propto) & 0 \\ \sin(\propto) & \cos(\propto) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If you wanted to rotate the point $\vec{X}$ = (4,3) by 90 degrees around point (3,1) the equation would look like this:

$$\begin{bmatrix} \cos(\pi/2) & -\sin(\pi/2) \\ \sin(\pi/2) & \cos(\pi/2) \end{bmatrix} * \begin{bmatrix} 4 \\ 3 \end{bmatrix} + \begin{bmatrix} 3 \\ 1 \end{bmatrix} - \begin{bmatrix} \cos(\pi/2) & -\sin(\pi/2) \\ \sin(\pi/2) & \cos(\pi/2) \end{bmatrix} * \begin{bmatrix} 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Any translation matrix can however be embedded within a $R^{1, N+1}$ matrix, where the coordinates get added to the transformation matrix.

The result is: $\begin{bmatrix} M_{1,1} & M_{1,2} & V_{1,1} \\ M_{2,1} & M_{2,2} & V_{2,1} \\ 0 & 0 & 1 \end{bmatrix}$

The old equation changes into:

$$\vec{X}' = \mathrm{f}(\vec{X}) = \begin{bmatrix} \vec{X}_{1,1} \\ \vec{X}_{2,1} \\ 1 \end{bmatrix} = \begin{bmatrix} M_{1,1} & M_{1,2} & V_{1,1} \\ M_{2,1} & M_{2,2} & V_{2,1} \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \vec{X}_{1,1} \\ \vec{X}_{2,1} \\ 1 \end{bmatrix}$$

And our example changes into:

$$\begin{bmatrix} \cos(\propto) & -\sin(\propto) & 3 \\ \sin(\propto) & \cos(\propto) & 1 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 4 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

Applying multiple transformation matrices to one point can be simplified. Let's say you want to scale your point on matrix M_1, rotate your point with matrix M_2, and scale your point again with a different matrix M_3.
The above method would create the following solution:

$$\vec{X}' = \mathrm{f}(\vec{X}) = M_3 * (M_2 * (M_1 * \vec{X})$$

Due to the rule of associativity this can also be changed into:

$$\vec{X}' = \mathrm{f}(\vec{X}) = \big((M_3 * M_2) * M_1\big) * \vec{X} = M_{3\circ 2\circ 1} * \vec{X}$$

So any vector which is transformed through M_1,M_2 and M_3 can simply be transformed using $M_{3\circ 2\circ 1}$

Affine Transformations

The main rule of affine transformation is that any triangle ABC can be uniquely transformed into any triangle A'B'C' using a unique affine map. The trick is to transform the identity triangle {{0,0},{1,0},{0,1}} unto the triangle ABC using transformation M. The next step is transforming the identity triangle unto the triangle A'B'C' using transformation N. Now to directly transform the triangle ABC to A'B'C', ABC has to be transformed by the inverse of M and the transformation N, so that we land back to A'B'C'. Using the above trick this gets reduced to a matrix of form: $N * M^{-1}$. Retrieving these transformations can be done by the following equation:

$$M = \begin{bmatrix} B_x - A_x & C_x - A_x & A_x \\ B_y - A_y & C_y - A_y & A_y \\ 0 & 0 & 1 \end{bmatrix} \quad N = \begin{bmatrix} B'_x - A'_x & C'_x - A'_x & A'_x \\ B'_y - A'_y & C'_y - A'_y & A'_y \\ 0 & 0 & 1 \end{bmatrix}$$

The special thing is it works on every other point transformed in the same way. So if you want to transform a house using matrices, you only have to calculate 3 points (for example the

housetop) in order to receive the matrix needed to transform all other points.

Eigenvalues
$M * \vec{v} = \lambda * \vec{v}$
M = Matrix
$\vec{v}$ = Eigenvector
λ = Eigenvalue (scalar)

In order to calculate the eigenvalues of a 3 dimensional vector, you have to simply solve the following equation:
$\lambda^2 - (M_{1,1} + M_{2,2})\lambda + \det(M) = 0$
Any higher order eigenvalue, would lead to a higher order equation, which cannot be calculated algebraically anymore and therefor needs an approximation like Newton's method.

There exist various algorithms to solve eigenvalues. The *power iteration* algorithm requires O(N^2) steps and finds the largest eigenvalue.
$$b_{i+1} = \frac{M * \vec{v}_i}{normalize(M * \vec{v}_i)}$$
Where the eigenvalue is $normalize(M * \vec{v}_i)$.

```
float eigenvalue(matrix &M, int iterations) {
    matrix eigenvector;
    float eigenvalue = 0;
    for(int i = 0; i < M.field.size(); ++i) eigenvector.field.push_back( 1 );
    for(int it = 0; it < iterations; ++it) {
        matrix temp = M * eigenvector;
        eigenvalue = normalize(temp);
        eigenvector = temp / eigenvalue;
    }
    return eigenvalue;
    //return eigenvector
}
```

In order to find the smallest eigenvalue you can simply calculate the eigenvalue of the inverse matrix, called the *inverse power iteration*, which is used for page ranking in search engines.

CORDIC

COordinte **R**otation **D**igital **C**omputer or Voler's algorithm is a binary-search algorithm to calculate hyperbolic and trigonometric functions.

In order to understand the algorithm, try calculating the arctangent function using only sine and cosine. You know that the angle = arctangent(y / x). The point x and y is given as input.

Turning a point clockwise around 0,0 will yield this formula:

$X_{rot} = X * \cos(angle) + Y * \sin(angle)$

$Y_{rot} = Y * \cos(angle) - X * \sin(angle)$

and counterclockwise:

$X_{rot} = X * \cos(angle) - Y * \sin(angle)$

$Y_{rot} = Y * \cos(angle) + X * \sin(angle)$

We only need to be able to calculate atan between the angles 0 and 90°. The rest will be explained later. By simply trying out different angles, we can approach the arctangent.

The difference between general binary search and CORDIC, is that we don't have to know every angle in order to find atan, but we only need to know a small section:

SinTable: sin(45°), sin(22°), sin(11°), sin(5°), sin(2°), sin(1°)

CosTable: cos(45°), cos(22°), cos(11°), cos(5°), cos(2°), cos(1°)

Now instead of using sine and cosine, we can simply use the tangent. Clockwise:

$X_{rot} = X - Y * \tan(angle)$

$Y_{rot} = Y + X * \tan(angle)$

Counter-Clockwise:

$X_{rot} = X + Y * \tan(angle)$

$Y_{rot} = Y - X * \tan(angle)$

It just so happens that tan(45°) = 1. However tan(22.5°) ≈ 0.41421… If we instead however take the tangent of 26.56505118..° we get 0.5. 14.03624347° happens to be 0.25. Binary shifting can easily half the number from 1 to 0.5 and from 0.5 to 0.25. This is used in order to calculate the arctangent, using tangent. It doesn't really matter that 26.5° isn't exactly half of 45°, it still allows binary-searching over the array, hence 45° < 26.5° < 22.5°

Note: angle = atan(y / x) = atan2(y, x). Also atan(b) = atan2(b, 1)

```
float shiftRight(float input, int it) {
    for(int i = 0; i < it; ++i) input /= 2;
    return input;
}

vector<float> tanTable = {45, 26.565, 14.036, 7.125, 3.576, 1.790,
0.895, 0.448};
float atan2(float y, float x) {
    float sumAngle = 0;
    for(int i = 0; i < tanTable.size(); ++i) {
        float xRot, yRot;
        if(y > 0) {
            xRot = x + shiftRight(y, i);
            yRot = y - shiftRight(x, i);
            sumAngle += tanTable[i];
        }
        if(y < 0) {
            xRot = x - shiftRight(y, i);
            yRot = y + shiftRight(x, i);
            sumAngle -= tanTable[i];
        }
        x = xRot;
        y = yRot;
    }
    float cordicGain = 0.60726;
    float hypotenuse = x * cordicGain;
    return sumAngle;
}
```

CORDIC Gain: cos(45°) * cos(26.565°) * cos(14.036°) * cos(7.125°) * cos(3.576°) * cos(1.790°) * cos(0.895°) * cos(0.488°) = 0.60726.

The CORDIC Gain value (scale factor K) depends on the size of the table and should be calculated as precise, as the table is. It is used to make functions more easy to understand. More in the CORDIC functions section.

CORDIC Sine & Cosine & Tangent

Calculating sine and cosine, requires the tangent table and its CORDIC gain. Again you only need to know a little part of every tangent, because binary search adds them together.

```
float shiftRight(float input, int it) {
   for(int i = 0; i < it; ++i) input /= 2;
   return input; }

vector<float> atanTable = {45, 26.565, 14.036, 7.125, 3.576, 1.790,
0.895, 0.448};
float sine(float desiredAngle) {
   desiredAngle %= 360;
   float sumAngle = 0;
   float cordicGain = 0.60726;
   float y = 0;
   float x = cordicGain;

   if(desiredAngle > 90)
      sumAngle = 180;
   if(desiredAngle > 270)
      sumAngle = 360;
   for(int i = 0; i < atanTable.size(); ++i) {
      float xRot, yRot;
      if(desiredAngle > sumAngle) {
         xRot = x - shiftRight(y, i);
         yRot = y + shiftRight(x, i);
         sumAngle += atanTable[i];
      }
      else if(desiredAngle < sumAngle)  {
         xRot = x + shiftRight(y, i);
         yRot = y - shiftRight(x, i);
         sumAngle -= atanTable[i];
      }
      x = xRot;
      y = yRot;
   }

   if (desiredAngle > 90 && desiredAngle < 270) {
      x = -x; y = -y;
   }
   float cos = x;
   float sin = y;
   float tan = sin / cos;
   return sin; //return cos for cosine, tan for tangent }
```

CORDIC Matrices

Transforming the atan2 function into matrix rotations can be done by introducing the variable σ_j, which is either 1 or -1.

Here $\sigma_i = -signu\square(\sigma_i) = \begin{cases} +1, y_i < 0 \\ -1, otherwise \end{cases}$

$$x[i+1] = x[i] - \sigma_i * 2^{-i} * y[i]$$
$$y[i+1] = y[i] + \sigma_i * 2^{-i} * x[i]$$
$$z[i+1] = z[i] - \sigma_i * \tan(2^{-i})$$

2^{-i} is the right shift, but the tangent table is indexed by i and not by 2^{-i}. Also z was previously called the sumAngle.

$$\begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} = \begin{pmatrix} 1 & -\sigma_i * 2^{-i} & 0 & 0 \\ \sigma_i * 2^{-i} & 1 & 0 & 0 \\ 0 & 0 & 1 & -\sigma_i * \tan(2^{-i}) \end{pmatrix} * \begin{pmatrix} x_{i-1} \\ y_{i-1} \\ z_{i-1} \\ 1 \end{pmatrix}$$

Or the more general case:

$$\begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} = k * \begin{pmatrix} 1 & -m * \sigma_i * \delta_i & 0 & 0 \\ \sigma_i * \delta_i & 1 & 0 & 0 \\ 0 & 0 & 1 & -\sigma_i * \mathrm{t}_i \end{pmatrix} * \begin{pmatrix} x_{i-1} \\ y_{i-1} \\ z_{i-1} \\ 1 \end{pmatrix}$$

Atan2 Matrix

Options: $k = 1; m = 1; \delta_i = 2^{-i}; t_i = \tan(\delta_i)$

Result: $z_\infty = z_0 + \tan({}^{y_0}/_{x_0})$ & $x_\infty = \sqrt{{x_0}^2 + {y_0}^2} \prod_{i=0}^{n} \sqrt{i + 2^{-2*i}}$

Linear Matrix (Multiplication/Addition)

Options: $k = 1; m = 0; \delta_i = 2^{-i};\ t_i = 2^{-i}$

Result: $y_\infty = y_0 + x_0 * z_0\ (if -2 < z_0 < 2)$

$z_\infty = z_0 + {}^{y_0}/_{x_0}\ (if -2 < {}^{y_0}/_{x_0} < 2)$

CORDIC Hyperbolic functions

Whenever we rotated x and y, until now, we never scaled the coordinates while rotating. Another factor of CORDIC is exactly this scaling with K (which is usually equal to the CORDIC gain). Because of K being smaller, the binary search function might get a flaw, because it is not guaranteed anymore, that atan(t) < t/2. Before atan(1) = 45°, atan(0.5) = 26.5°, atan(0.25) = 14°, etc. This is why a few iterations have to be repeated, namely of every sequence in:

3 * i + 1 iterations, starting from 4, then 13, 40, 121 …

Options: $k = 0.60726 \ldots; m = -1; \delta_i = 2^{-i-1} = 0.5^{i+1};\ t_i = \tanh(\delta_i)$

Result: $z_\infty = z_0 + \operatorname{atanh}\left({}^{y_0}/_{x_0}\right)\ \&\ x_\infty = k * ({x_0}^2 - {y_0}^2)$

Also $x_0 = a + b; y_0 = a - b; z_0 = 0; then\ x_\infty = 2\sqrt{a * b}; z_\infty = \frac{1}{2}\ln\left(\frac{a}{b}\right)$

```
vector<float> atanhTable = {0.0,0.54931, 0.25541, 0.12566, 0.06258,
0.03126, 0.01563, 0.00781};
float atanh(float y, float x) {
   float z = 0;
   int repeat = 4;
   for(int i = 1; i < atanhTable.size(); ++i) {
      float xT;
      if(y < 0) {
         xT = x + shiftRight(y, i);
         y = y + shiftRight(x, i);
         z = z - atanhTable[i];
      }
      else {
         xT = x - shiftRight(y, i);
         y = y - shiftRight(x, i);
         z = z + atanhTable[i];
      }
      x = xT;
      if(i == repeat) {
         repeat = (3 * i) + 1;
         --i;
      }
   }
   return z;
}
float ln(float x) {
   return 2 * atanh2(x - 1, x + 1);
   //return x in atanhTable() for sqrt
}
```

Complex numbers

Imaginary numbers

Complex numbers are basically a combination of 2 numbers, a real number and an imaginary number (a real and an imaginary part). The imaginary number i is defined as:

$$\boldsymbol{i^2 = -1}\ ;\ \ or\ i = \sqrt{-1}$$

The last definition is not wrong, but you have to be careful, see later.

The exponentiation of i is really the most obvious when written down:

$$i^0 = 1$$
$$i^1 = i$$
$$i^2 = -1$$
$$i^3 = i^2 * i = -1 * i = -i$$
$$i^4 = i^2 * i^2 = -1 * -1 = 1$$
$$i^5 = i^4 * i^1 = 1 * i = 1$$

The exponentiation cycles every 4 times, which makes it easy to implement later in the complex number library.

Of course you might now start to mix imaginary numbers with real numbers, for example:

$$? = 3 + \sqrt{-4}$$

The next few sections implement the complex numbers, where every complex number has an imaginary and a real part.

```
struct complex {
    float r, im; //real & imaginary part
    complex() {}
    complex(float _r, float _im) : r(_r), im(_im) {}
};
```

There is one rule of thumb when calculating with complex numbers, *always do the imaginary part first*!

Also I had to overload the equals operator, hence the implementation used floating point numbers.

```
bool operator==(const complex& lhs, const complex& rhs) {
    if(fabs(lhs.r - rhs.r) < 1e-7 && fabs(lhs.im - rhs.im)< 1e-7) return true;
    else return false;
}
```

I/O Stream

For my own reference, I wanted to overload I/O-streams somewhere in the book, because I always forget how, and it's handy to look it up here ☺.

```
ostream& operator<<(ostream &out, const complex& val) {
    out << val.im << "i+" << val.r;
    return out;
}
//e.g.: 5-3i or -5+3i
istream& operator>> (istream &in, complex& value) {
    string a;
    in >> a;
    string iPart="", rPart="";
    bool isImaginary = false;
    for(int i = 0; i < a.size(); ++i) {
        if(a[i] != 'i') {
            if (i != 0 && (a[i] == '-' || a[i] == '+')) {
                isImaginary = true;
            }
            if(isImaginary) iPart += a[i];
            else rPart += a[i];
        }
    }
    value.r = ::atof(rPart.c_str()); value.im = ::atof(iPart.c_str());
    return in;
}
```

Addition / Subtraction (Complex numbers)

Addition with complex numbers is straightforward. Add up / Subtract the real parts together and the imaginary parts.

```
complex operator+(const complex& lhs, const complex& rhs) {
    complex output;
    output.r = lhs.r + rhs.r;
    output.im = lhs.im + rhs.im;
    return output;
}
complex operator-(const complex& lhs, const complex& rhs) {
    complex output;
    output.r = lhs.r - rhs.r;
    output.im = lhs.im - rhs.im;
    return output;
}
```

Scalar Multiplication / Division (Complex numbers)

This is again straightforward.

```
complex operator*(float lhs, complex rhs) {
    rhs.r *= lhs;
    rhs.im *= lhs;
    return rhs;
}
complex operator*(complex lhs, float rhs) {return rhs * lhs;}
complex operator/(complex lhs, float rhs) {
    lhs.r /= rhs;
    lhs.im /= rhs;
    return lhs;
}
```

Multiplication / Division (Complex numbers)

$(a + bi) * (c + di) = a * c + a * di + bi * c + bi * di$

$= a * c + a * di + bi * c + b * d * i^2 = a * c + a * di + bi * c - b * d$

```
complex operator*(const complex& lhs, const complex& rhs) {
    complex output(0,0);
    output.r += lhs.r * rhs.r;
    output.r -= lhs.im * rhs.im;
    output.im += lhs.r * rhs.im;
    output.im += lhs.im * rhs.r;
    return output;
}
```

$$\frac{a + bi}{c + di} = \frac{(a + bi) * (c - di)}{(c + di) * (c - di)} = \frac{(a + bi) * (c - di)}{c * c + d * d}$$

```
complex operator/(complex lhs, complex rhs) {
    complex conjugate = rhs;
    conjugate.im = -rhs.im;
    lhs = lhs * conjugate;
    rhs = rhs * conjugate;
    lhs = lhs / rhs.r;
    return lhs;
}
```

Scalar Exponentiation (Complex numbers)
Complex number multiplication is associative, so we can use our trick as well to multiply it log(N)-times by a scalar.

```
complex pow(complex &a, int b) {
    vector<complex> stored;
    stored.push_back(a);
    int it = 2;
    while(it < b) {
        stored.push_back(stored.back() * stored.back());
        it *= 2;
    }
    it /= 2;
    complex solution = stored.back();
    b -= it;
    while(b != 0) {
        if(b >= it) {
            solution = solution * stored.back();
            b -= it;
        }
        stored.pop_back();
        it /= 2;
    }
    return solution;
}
```

Complex norm / Complex modulus / Complex absolute value

$norm(a + bi) = |a + bi| = \sqrt{a^2 + b^2} = z$

```
float norm(complex &a) {
    return sqrt(a.r*a.r + a.im*a.im);
}
```

Complex argument

$\phi = \tan^{-1}\frac{b}{a} = \arg(a + bi)$

```
float complexArgument(complex &a) {
    if(a.im == 0) return e;
    return atan2(a.im, a.r);
}
```

Euler's formula

$e^{i*x} = \cos(x) + \sin(x) * i = cis(x)$

Plotting the above equation, will yield the circle with radius 1 (or i).

Elegant Proof:

$f(t) = e^{-it} * (\cos(t) + i * \sin(t))$

$f'(t) = e^{-it} * (-\sin(t) + \cos(t) * i) - e^{-it} * i * (\cos(t) + \sin(t) * i)$

$f'(t) = 0$ (!)

If the derivative is zero, the function $f(t)$ must be a constant.

Since $f(0) = 1$, the following statement must be true: $f(t) = 1$.

$1 = e^{-it} * (\cos(t) + i * \sin(t))$

$\frac{1}{e^{-it}} = (\cos(t) + i * \sin(t)) = e^{it}$

Euler's identity

If you now set $x = \pi$, Euler's formula becomes Euler's identity:

$e^{i*\pi} = -1$

It also solves multiple problems in mathematics, for example:

$i^i = e^{\frac{-\pi}{2}}$

De Moivre's identity

$e^{i*(n*\theta)} = \left(e^{i*\theta}\right)^n$

$\cos(n * \theta) + \sin(n * \theta) * i = (\cos(\theta) + \sin(\theta) * i)^n$

Phasor

The phasor is a different way to express an imaginary number.

$a + bi = |a + bi| * e^{i*\phi} = |a + bi| * (\cos(\phi) + \sin(\phi) * i)$

Complex Exponentiation

The trick used by Euler for the derivative of an exponent can be used here as well:

$a^x = \left(e^{\ln(a)}\right)^x = e^{x*\ln(a)}$

For example $2^i = e^{i*\ln(2)} = \cos(\ln(2)) + \sin(\ln(2)) * i$

In order to solve the general equation $(a+bi)^{c+di}$, we have to use phasors:

$a + bi = |a+bi| * e^{i*\phi}; where\ a+bi = w\ and\ |a+bi| = r$

$w = r * e^{i*\phi}$

$\ln(w) = \ln\left(r * e^{i*\phi}\right) = \ln(r) + \ln\left(e^{i*\phi}\right) = \ln(r) + i*\phi*$ ~~$\ln(e)$~~

$w^z = e^{z*\log(w)} = e^{z*(\log(r)+i*\phi)}$

Now replace w with $a+bi$, z with $c+di$, where $r = |a+bi| = \sqrt{a^2+b^2}$ and $\phi = \arg(a+bi) = \tan^{-1}\frac{b}{a}$

$(a+bi)^{c+di} = e^{(c+di)*(\ln(|a+bi|)+\tan^{-1}\left(\frac{b}{a}\right)*i)}$

Thanks to a bit more mathematics it is possible to bring the equation to the below form:

$(a+bi)^{c+di} = (a^2+b^2)^{\frac{c+di}{2}} * e^{i*(c+di)*ar\square(a+bi)}$

And because of Euler's identity the entire equation can be expressed in real numbers:

$(a+bi)^{c+di} = x + yi$

$x = (a^2+b^2)^{c/2} * e^{-d*\arg(a+bi)}\left\{cos\left[c*\arg(a+bi) + \frac{1}{2}d*ln(a^2+b^2)\right]\right\}$

$y = (a^2+b^2)^{c/2} * e^{-d*\arg(a+bi)}\left\{sin\left[c*\arg(a+bi) + \frac{1}{2}d*ln(a^2+b^2)\right]\right\}$

```
complex pow(complex lhs, complex rhs) {
    complex output;
    float a = lhs.r;
    float b = lhs.im;
    float c = rhs.r;
    float d = rhs.im;
    float AB = a * a + b * b;
    float p1 = pow(AB, c/2);
    float p2 = exp(-d * complexArgument(lhs));
    float p3 = c * complexArgument(lhs) + 0.5 * d * ln(AB);
    output.r = p1 * p2 * cos(p3);
    output.im = p1 * p2 * sin(p3);
    return output;
}
```

Complex Root

After being able to calculate the complex exponential, the complex root is also calculable. By the way, for easy square functions: $\sqrt{-49} = \sqrt{-1} * \sqrt{49} = i * 49 = 0 + 49i$.

Another interesting rule of roots is that when the n-th root is given: $\sqrt[n]{-x} = -\sqrt[n]{x}; if\ n\ is\ uneven\ and\ x \geq 0$

Anyway in order to compute the $(c + di)$-th root, you can use the following equation: $\sqrt[c+di]{a + bi} = (a + bi)^{\frac{1}{c+di}}$

```
complex root(complex degree, complex radicand) {
    complex one(1,0);
    complex output = pow(radicand, one / degree);
    return output;
}
```

Curve Fitting

Least Squares (Polynomial Regression)

The standard deviation is the measurement of the error rate of the mean. To calculate the average, the mean of data sets, we simply add all of them together and divide them by the total number of points. In case overflow is an issue you can also divide each unit and then add them together.

$$Standard\ deviation\ (SD) = \ \sigma = \sqrt{\frac{\sum_{i=1}^{N} r_i^2}{df}}$$

$$Degrees\ of\ freedom\ (df) = N - \ \mu$$

$$Error\ from\ mean = r_i = x_i - \mu \ \ ; where\ x_i\ is\ the\ y - value\ at\ x_i.$$

$$Mean = \ \mu = \frac{\sum_{i=1}^{N} x_i}{N}$$

$$f(x) = \frac{1}{\sigma * \sqrt{2 * \pi}} * e^{-\frac{r_i^2}{2*\sigma^2}}$$

However a line can be written as:

$y: f(x) = m * x + b$

More generally this equation can be written as:

$y: f(x) = a_1 * x + a_0$

$$S_r = \sum_{i=1}^{N} {r_i}^2 = \sum_{i=1}^{N} (y_i - a_0 - a_1 * x_i)$$

$$\frac{d\ S_r}{d\ a_0} = 0\ (min) = -2 \sum_{i=1}^{N} (y_i - a_0 - a_1 * x_i)$$

$$\frac{d\ S_r}{d\ a_1} = 0\ (min) = 2 \sum_{i=1}^{N} [(y_i - a_0 - a_1 * x_i) * x_i]$$

Normal equation:

$$a_1 = \frac{n * \sum x_i * y_i - \sum x_i * y_i}{n * \sum x_i^2 - (\sum x_i)^2} \quad ; \quad a_0 = \bar{y} - a_1 * \bar{x}$$

The above equation can also be derived from matrix calculations, which is easier. The line can also be expressed as the following equation, which in return can be written as a matrix equation:

$$y_i = a_0 * x_i^0 + a_1 * x_i^1$$

$$A * x = \begin{bmatrix} x_1^0 & x_1^1 \\ \vdots & \vdots \\ x_N^0 & x_N^1 \end{bmatrix} * \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = b$$

Modifying x, so that A * x becomes as close as possible to b, we receive the normal equation.
Now multiplying both sides with the transpose of A : A^T

$$A^T * A * x = \begin{bmatrix} x_1^0 & \dots & x_N^0 \\ x_1^1 & \dots & x_N^1 \end{bmatrix} * \begin{bmatrix} x_1^0 & x_1^1 \\ \vdots & \vdots \\ x_N^0 & x_N^1 \end{bmatrix} * \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} x_1^0 & \dots & x_N^0 \\ x_1^1 & \dots & x_N^1 \end{bmatrix} * \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$$

$$= A^T * b$$

$$A^T * A = \begin{bmatrix} x_1^0 & \dots & x_N^0 \\ x_1^1 & \dots & x_N^1 \end{bmatrix} * \begin{bmatrix} x_1^0 & x_1^1 \\ \vdots & \vdots \\ x_N^0 & x_N^1 \end{bmatrix} = \begin{bmatrix} \sum_{i=1} (x_i^0)^2 & \sum_{i=1} x_i^0 * x_i^1 \\ \sum_{i=1} x_i^1 * x_i^0 & \sum_{i=1} (x_i^1)^2 \end{bmatrix}$$

$$A^T * b = \begin{bmatrix} x_1^0 & \dots & x_N^0 \\ x_1^1 & \dots & x_N^1 \end{bmatrix} * \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} \sum_{i=1} x_i^0 * y_i \\ \sum_{i=1} x_i^1 * y_i \end{bmatrix}$$

This allows compressing the entire information into a 2x2 matrix, which in return gives the normal equation:

$$a_1 = \frac{n * \sum x_i * y_i - \sum x_i * y_i}{n * \sum x_i^2 - (\sum x_i)^2} \quad ; \quad a_0 = \bar{y} - a_1 * \bar{x}$$

Inserting these values into the function should work:
$y: f(x) = a_1 * x + a_0$

This can be extended to any n-dimensional form of your liking.

$$r^2 = \frac{S_t - S_r}{S_t}; \; S_t = \sum_{i=1}^{N} (y_i - \bar{y})^2 \; ; \; S_r = \sum_{i=1}^{N} (y_i - a_0 - a_1 * x_i)$$

```
struct point {
    float x, y;
    point(float _x, float _y) : x(_x), y(_y) {}
};

vector<polynomial> polynomialRegression(vector<point> data, int
order) {
    matrix A, AT, Y;
    AT.field.resize(order, vector<float>(order, data.size()));
    for (int i = 0; i < order; ++i) {
        for(int j = 0; j < data.size(); ++j) {
            AT.field[i][j] = pow(data[j].x, order);
        }
    }
    Y.field.resize(order, vector<float>(1,0));
    for(int i = 0; i < data.size(); ++i) {
        Y.field[i][0] = data[i].y;
    }

    AT = transpose(AT);
    matrix result = gaussianElimination(AT * A, AT * Y);
    vector<polynomial> output;
    for(int i = 0; i < result.field.size(); ++i) {
        polynomial temp(result.field[i][0], i);
        output.push_back(temp);
    }
    return output;
}
```

Interpolation

Interpolation: $x_0 \leq x \leq x_1$

Extrapolation: $x < x_0 \; or \; x_1 < x$

Interpolation will formulate the linear equation for 2 points, the parable for 3 points and for every other point the next higher order function. The uniqueness of the function is what makes it different to regression.

If the data points are linear, the function is described as $f_1(x)$

For linear systems:

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f_1(x) - f(x_0)}{x - x_0}$$

Therefor

$$f_1(x) = \frac{f(x_1) - f(x_0)}{x_1 - x_0} * (x - x_0) + f(x_0)$$

For a general polynomial of order d:

Newton's divided difference interpolating polynomials

$$f_d(x) = b_0 + b_1 * (x - x_0) + b_2 * (x - x_0) * (x - x_1) + \cdots + b_d * (x - x_0) * (x - x_1) * \ldots * (x - x_{d-1})$$

$$f_d(x) = \sum_{i=0}^{d} (b_i * \prod_{j=0}^{i-1} (x - x_j))$$

$$b_i = f[x_0, x_1, \ldots, x_i] = \frac{f[x_1, x_2, \ldots, x_i] - f[x_0, x_1, \ldots, x_{i-1}]}{x_i - x_0}$$

Calculating b_i takes exponential complexity of $O(2^i)$. This is the same function in C++. Again DON'T USE THIS, but use the Lagrange interpolation instead.

```
//d=data, b=bottom, t=top
float f(vector<point>& d, int b, int t) {
   if(b == t) return d[b].y;
   return (f(d, b+1, t), f(d, b, t-1)) / (d[t].x - d[b].x);
}
```

Lagrange Interpolation

$$f_d(x) = \sum_{i=0}^{d} (y_i * (\prod_{\substack{j=0 \\ j \neq i}}^{d} \frac{x - x_j}{x_i - x_j}))$$

Spline Interpolation

Spline interpolation divides the input data in various functions, which each of them will be interpolated.

The easiest spline interpolation is linear spline interpolation, because it simply connects every data point to the next. In other words it creates a line from $x_i \; to \; x_{i+1}$, which can be denoted as:

$$if \; x_{i-1} \geq x \geq x_i, then \; use \; f(x) = f(x_i) + \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} (x - x_{i-1})$$

The main problem with linear spline interpolation is that we can't differentiate the function at a point (also called knot in the function).

Quadratic interpolation is a bit more complicated and needs a few modifications.

1) All function values are equal at knots.
2) First & last function pass through end points.
3) First / Last derivative equal at interior points.
4) The last and the first function have a second derivative of 0 or have $a_2 = 0$ in the equation: $f(x) = a_0 * x^0 + a_1 * x^1 + a_2 * x^2$

Cubic spine interpolation gets used most, because its first and second derivative can be calculated and most people care about these derivatives.

Cubic spine implementation:

```
struct spline {
   float x, y, derivative;
};
bool operator<(const spline& lhs, const spline& rhs) {
   return lhs.x < rhs.x;
}

float f(vector<spline>& d, float at) {
   sort(d.begin(), d.end());
   calculateDeriative(d);
```

```
    float i = 1;
    while(d[i].x < at) i++;
    float diffX = d[i].x - d[i-1].x;
    float diffY = d[i].y - d[i-1].y;
    float t = (at - d[i-1].x) / diffX;

    float a =  (d[i-1].derivative * diffX) - (diffY);
    float b = (-d[i].derivative * diffX) + (diffY);

    float q = (1-t) * d[i-1].y + t * d[i].y + t*(1-t)*(a*(1-t)+b*t);
    return q;
}

void calculateDeriative(vector<spline> &d) {
    sort(d.begin(), d.end());
    int n = d.size() - 1;
    matrix A;
    A.field.resize(n + 1, vector<float>(n + 2, 0));
    for(int i = 1; i < n; ++i) {
        float diffX = d[i].x - d[i-1].x;
        float diff2X = d[i+1].x - d[i].x;
        float diffY = d[i].y - d[i-1].y;
        float diff2Y = d[i+1].y - d[i].y;
        A.field[i][i-1] = 1 / diffX;
        A.field[i][i] = 2 * (1 / diffX) + (1 / diff2X);
        A.field[i][i+1] = 1/diff2X;
        A.field[i][n+1] = 3 * (diffY / (diffX*diffX) + diff2Y / (diff2X*diff2X));
    }
    //First spline
    float d0X = d[1].x - d[0].x, d0Y = d[1].y - d[0].y;
    A.field[0][0] = 2 / d0X;
    A.field[0][1] = 1 / d0X;
    A.field[0][n+1] = 3 * d0Y / (d0X * d0X);
    //Last spline
    float dnX = d[n].x - d[n-1].x, dnY = d[n].y - d[n-1].y;
    A.field[n][n-1] = 1 / dnX;
    A.field[n][n] = 2 / dnX;
    A.field[n][n+1] = 3 * dnY / (dnX * dnX);

    matrix derivatives = gaussianElimination(A);
    for(int i = 0; i < n+1; ++i) {
        d[i].derivative = derivatives.field[i][0];
    }
}
```

Least Square Regression with Sinusoids

Every sinusoid can be represented as:

$f(x) = a_0 + a_1 * \cos(w_0 * t + \theta)$

$f(x) = a_0 + a_1 * \cos(w_0 * t) + a_2 * \sin(w_0 * t)$

Polynomial Least Square Regression can be used for this problem when $t_i = x_i$.

Using the matrix method:

$$\begin{bmatrix} 1 & \cos(w_0 * t_1) & \sin(w_0 * t_1) \\ \vdots & \vdots & \vdots \\ 1 & \cos(w_0 * t_n) & \sin(w_0 * t_n) \end{bmatrix} * \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} f(t_1) \\ \vdots \\ f(t_n) \end{bmatrix}$$

Continuous Fourier Approximation

Not only can we calculate the least square regression of sinusoids, but also the least square regression of multiple sinusoids added together. Using the Continuous Fourier Series in the time domain:

$$F(t) = a_0 + \sum_{k=1}^{\infty} a_{2*k} * \cos(k * w_0 * t) + a_{2*k+1} * \sin(k * w_0 * t)$$

Instead of writing down the equation using the matrix method, instead Fourier used Integrals to his advantage. These are the Fourier coefficients:

$$a_0 = \frac{1}{T} * \int_0^T f(t)\, dt$$

$$a_{2*k} = \frac{2}{T} * \int_0^T f(t) * \cos(k * w_0 * t)\, dt$$

$$a_{2*k+1} = \frac{2}{T} * \int_0^T f(t) * \sin(k * w_0 * t)\, dt$$

Where: $T = \frac{2*\pi}{w_0} = duration\ of\ frequency\ (in\ time\ doma\square n)$

Because T is periodic:

$$\int_0^T q\, dt = \int_T^{2T} q\, dt$$

Fourier Transformation

Using Euler's formula in the complex number section $f(x)$ can be rewritten in the frequency domain as:

$$\cos(k * w_0 * t) = \frac{1}{2} * \left(e^{i*n*w_0*t} + e^{-i*n*w_0*t}\right)$$

$$\sin(k * w_0 * t) = \frac{1}{2} * \left(e^{i*n*w_0*t} - e^{-i*n*w_0*t}\right) * i$$

Quick proof:

$$\mathrm{e}^{i*w_0*t} = cos(w_0 * t) + \sin(w_0 * t) * i$$

$$\mathrm{e}^{-i*w_0*t} = cos(w_0 * t) - \sin(w_0 * t) * i$$

$$\mathrm{e}^{i*w_0*t} + \mathrm{e}^{-i*w_0*t} = 2 * \cos(w_0 * t)$$

$$\mathrm{e}^{i*w_0*t} - \mathrm{e}^{-i*w_0*t} = 2 * \sin(w_0 * t) * i$$

$$f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \frac{1}{2} * (a_{2*k} - a_{2*k+1} * i) * e^{i*k*w_0*t}$$

$$+ \sum_{k=1}^{\infty} \frac{1}{2} * (a_{2*k} + aa_{2*k+1} * i) * e^{-i*k*w_0*t}$$

$$= \sum_{k=-\infty}^{\infty} \frac{1}{2} * (a_{2*k} - a_{2*k+1} * i) * e^{i*k*w_0*t}$$

Where:

$$\frac{1}{2} * (a_{2*k} - a_{2*k+1} * i) = \frac{1}{T} * \int_0^T f(t) * e^{-i*k*w_0*t} \;\; ; \;\; for\ all\ k \in \mathbb{Z}$$

Continous:

$$F(i * w_0) = \int_{-\infty}^{\infty} f(t) * \mathrm{e}^{-i*w_0*t} dt$$

Discrete:

$$F_k = \sum_{n=0}^{N-1} f_n * e^{-i*k*w_0*n} \;\; ; \;\; for\ k = 1\ to\ N-1$$

Inverse Fourier Transformation

Continous:

$$f(t) = \frac{1}{2 * \pi} \int_{-\infty}^{\infty} F(i * w_0) * \mathrm{e}^{i*w_0*t} dw_0$$

Discrete:

$$f_n = \frac{1}{N} * \sum_{k=0}^{N-1} F_k * e^{-i*k*w_0*n} \;\; ; \;\; for\ n = 1\ to\ N-1$$

1D-DFT Implementation in $O(N^2)$

```
vector<complex> DFT1D(vector<complex>& f, bool inverse) {
    vector<complex> output;
    float w_0 = 2.0 * PI / f.size();
    if(inverse) w_0 *= -1;
    for(int k = 0; k < f.size(); ++k) {
        complex F_k(0,0);
        for(int n = 0; n < f.size(); ++n) {
            float angle = k * n * w_0;
            F_k.r += f[n].r * cos(angle) + f[n].im * sin(angle);
            F_k.im += -f[n].r * sin(angle) + f[n].im * cos(angle);
        }
        if(inverse) F_k = F_k * (1.0 / f.size());
        output.push_back(F_k);
    }
    return output;
}
```

Fast Fourier Transformation (Cooley-Tukey algorithm) (FFT)
In order to improve the speed of calculating the Fourier transformation into O(N log N), Cooley-Tukey proposed splitting up the Fourier Transformation into 2 Fourier Transformations, per calculation. This division is also called Danielson Lanczos Lemma.

$$F_k = \sum_{n=0}^{N-1} f_n * e^{-i*k*w_0*n} =$$

$$\sum_{n=0}^{(N/2)-1} f_n * e^{-i*k*w_0*n} + \sum_{n=(N/2)}^{N-1} f_n * e^{-i*k*w_0*n}$$

$$m = n - \left(\frac{N}{2}\right)$$

$$F_k = \sum_{n=0}^{(N/2)-1} f_n * e^{-i*k*w_0*n} + \sum_{m=0}^{(N/2)-1} f_{m+(N/2)} * e^{-i*k*w_0*(m+N/2)}$$

$$F_k = \sum_{n=0}^{(N/2)-1} f_n * e^{-i*k*w_0*n} + \sum_{m=0}^{N/2-1} f_{m+N/2} * e^{-i*k*w_0*m} * e^{-i*k*w_0*N/2}$$

$$e^{-i*k*w_0*N/2} = e^{-i*\pi*k} = (-1)^k$$

$$F_k = \sum_{n=0}^{(N/2)-1} f_n * e^{-i*k*w_0*n} + (-1)^k * \sum_{n=0}^{(N/2)-1} f_{n+N/2} * e^{-i*k*w_0*n}$$

You can also separate the odd sequences from the even ones:

$$F_{2*k} = \sum_{n=0}^{(N/2)-1} (f_n - f_{n+\frac{N}{2}}) * e^{-i2*k*w_0*n}$$

$$F_{2*k+1} = \sum_{n=0}^{(N/2)-1} (f_n - f_{n+\frac{N}{2}}) * e^{-i(2*k+1)*w_0*n} =$$

$$F_{2*k+1} = \sum_{n=0}^{(N/2)-1} (f_n - f_{n+\frac{N}{2}}) * e^{-i*2*k*w_0*n} * e^{-i*w_0*n}$$

Radix-2 implementation of the Fast Fourier transformation:

```
vector<complex> FFT_rec(vector<complex>& f, bool inverse) {
    int N = f.size();
    vector<complex> out(N);
    if (N == 1) {
        out[0] = f[0];
        return out;
    }
    vector<complex> o(N/2), e(N/2), even, odd;
    for(int k = 0; k < N/2; ++k) {
        e[k] = f[2*k];
        o[k] = f[2*k + 1];
    }
    even = FFT_rec(e, inverse);
    odd = FFT_rec(o, inverse);

    float w = 2.0 * PI / N;
    if(inverse) w = -w;
    for(int k = 0; k < N; ++k) {
        float c = cos(w*k);
        float s = sin(w*k);
        out[k].r = even[k%(N/2)].r + odd[k%(N/2)].r *c + odd[k%(N/2)].im*s;
        out[k].im=even[k%(N/2)].im + odd[k%(N/2)].im*c-odd[k%(N/2)].r*s;
    }
    return out;
}

vector<complex> FFT(vector<complex> f, bool inverse) {
    int pow2 = 1;
    while(pow2 < f.size()) pow2 *= 2;
    vector<complex> output;
    if(pow2 == f.size()) output = FFT_rec(f, inverse);
    else {
        output = chirpZTransform(f, inverse);
    }
    if(inverse) {
        for(int i = 0; i < output.size(); ++i) {
            output[i] = output[i] * (1.0 / f.size());
        }
    }
    return output;
}
```

Chirp-Z Transform (CSZ)

While Radix-2 is the simplest implementation of the FFT, it only works for multiples of 2. In order to calculate any FFT zero padding on the array won't work, instead there is something called chirp-z transformation. Chirp-Z Transformation is based on an idea of Bluestein:

$$e^{-i*2\pi*k*w_0*n} = W^{k*n}$$

$$k*n = \frac{k^2}{2} + \frac{n^2}{2} - \frac{(k-n)^2}{2}$$

Discrete:

$$F_k = W^{k^2/2} \sum_{n=0}^{N-1} f_n * W^{n^2/2} * W^{-(k-n)^2/2} \;\; ; \quad for\ k = 1\ to\ N-1$$

```
vector<complex> chirpZTransform(vector<complex> f, bool inverse) {
    int N = f.size(), M = 1;
    while(M < 2 * N) M *= 2;
    for(int i = N; i < M; ++i) {
        f.push_back(complex(0,0));
    }
    float PIN = PI/N;
    if(inverse) PIN = -PIN;
    vector<complex> scaled(M, complex(0,0));
    for(int i = 0; i < N; ++i) {
        scaled[i].r = cos(PIN * i * i);
        scaled[i].im = sin(PIN * i * i);
    }
    for(int i = N; i <= M - N; ++i) {
        scaled[i] = complex(0,0);
    }
    for(int i = 1 - N; i < 0; ++i) {
        scaled[M + i].r = cos(PIN * i * i);
        scaled[M + i].im = sin(PIN * i * i);
    }
    for(int i = 0; i < N; ++i) {
        complex W;
        W.r = cos(PIN * i * i);
        W.im = - sin(PIN * i * i);
        f[i] = f[i] * W;
    }
    f = FFT(f, false);
    scaled = FFT(scaled, false);

    for(int i = 0; i < M; ++i) {
        f[i] = f[i] * scaled[i];
    }
    f = FFT(f, true);
    for(int i = 0; i < M; ++i) {
        complex W;
        W.r = cos(PIN * i * i);
        W.im = - sin(PIN * i * i);
        f[i] = f[i] * W;
    }

    vector<complex> o(N);
    for(int i = 0; i < N; ++i) o[i] = f[i];
    return o;
}
```

Lindenmayer-System
A Lindenmayer system consists of 4 things L = (V, S, w, P).
V are variables.
S are constants. V & S form the alphabet of the L-System.
w axiomatic start state of the L-System.
P consists out of the underlying rule system of the L-system.

Lindenmayer-System can describe fractals, complicated recursive functions in a very reduced form. On the next page is an implementation of the Koch snowflake. Examples:

Cantor dust:
V={A, B} (where A&B can be printed per iteration to show the cantor dust.) S={}
w = {A}
P = {A -> ABA, B -> BBB}

Koch snowflake:
V={F} (where F moves forward by one unit)
S={+,-} (where – and + add/subtract 60° of view-angle).
w = {F--F--F}
P = {F -> F+F--F+F}

Sierpinski triangle:
V={A, B} (where A & B both move forward by one unit)
S={+,-} (where – and + add/subtract 60° of view-angle).
w = {A}
P = {A -> +B-A-B+, B -> -A+B+A-}

Dragon curve:
V={X, Y} (where X & Y do nothing)
S={F, +,-} (where – and + add/subtract 90° of view-angle & F moves forward by one unit).
w = {F,X}
P = {X -> X+YF+, Y -> -FX-Y}

Hilbert curve:
V={A, B} (where A & B do nothing}
S = {F,+,-} ={+,-} (-/+ 90° anlge, F = draw forward).
w = {A}
P = {A -> BF+AFA+FB–, + AF–BFB–FA+}

Koch snowflake

```
struct Koch {
    float x = 0, y = 0;
    string F = "F";
    string P = "F-F++F-F";
    string w = "F--F--F";
    float angle = 0;
    void F() {
        x += cos(angle * PI / 180.0);
        y += sin(angle * PI / 180.0);
    }
    void minus() {
        angle -= 60;
        angle = fmod(angle, 360);
    }
    void plus() {
        angle += 60;
        angle = fmod(angle, 360);
    }
    void executeF() {
        float pX = x;
        float pY = y;
        for(int i = 0; i < F.length(); ++i) {
            if(F[i] == 'F') F();
            else if(F[i] == '+') plus();
            else if(F[i] == '-') minus();
            line(x, y, pX, pY);
            pX = x; pY = y;
        }
    }
    void run() {
        //Update P
        string newF = "";
        for(int i = 0; i < P.length(); ++i) {
            if(P[i] == 'F') newF += F;
            else if(P[i] == '+') newF += '+';
            else if(P[i] == '-') newF += '-';
        }
        F = newF;
        for(int i = 0; i < w.length(); ++i) {
            if(w[i] == 'F') executeF();
            else if(w[i] == '+') plus();
            else if(w[i] == '-') minus();
        }}};
```

Dynamic Programming (DP)

“Those who don't know history are doomed to repeat it.”

- Edmund Burke

Dynamic Programming is the act of saving performance power by using more memory.

This is a generalization on how to approach DP exercises :

1. Characterize the substructure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

The last step is only required if we actually want a solution and not for example the size of a solution.

There are 2 types of dynamic programming algorithms:
Top-Bottom algorithms (Sometimes called Memoization)
Bottom-Up algorithms

Optimal Substructure
For any DP approach, we have to find a substructure (or a general mathematical) way how to describe the problem, in very much the way we did with Divide & Conquer.
The classical dynamic programming structures:
Factorial:

$$factorial\ (n) = \begin{cases} if(n == 0)\ return\ 1 \\ else\ return\ n * \ factorial(n-1) \end{cases}$$

Fibonacci:

$$fib\ (n) = \begin{cases} if(n == 0\ ||\ n == 1)\ return\ 1 \\ else\ return\ fib(n-1) + fib(n-2) \end{cases}$$

Rod Cutting:

$$cutRod(n) = \begin{cases} if(n == 0)\ \ return\ price[0] \\ else\ \max\big(price[i] + cutRod(n-i-1)\big)\ for\ all\ i\ in\ 0..n-1 \end{cases}$$

We can implement factorial directly the way we declared it recursively (ull = unsigned long long)

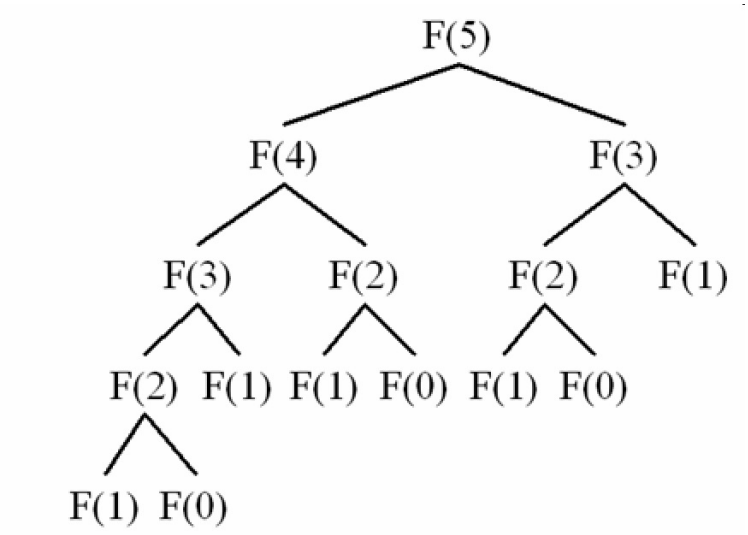
```
ull factorial(ull n)
{
   if(n == 0) return 1;
   else return n * factorial(n - 1);
}
```

The above has a perfect runtime of O(n) and is a top to bottom implementation.
However be careful when implementing top to bottom functions, because whenever you call the function twice with the same parameter you are loosing time, which you could've saved. When we implement Fibonacci this way, it comes quite clear why:

```
ull fib(ull n)
{
   if(n == 0 || n == 1) return 1;
   else return fib(n - 1) + fib(n - 2);
}
```

This program will only terminate for input values smaller then 50 on a modern computer. In fact whenever you increment n by one the program will take double the time. It has a running-time of $O(2^N)$, which is very slow.

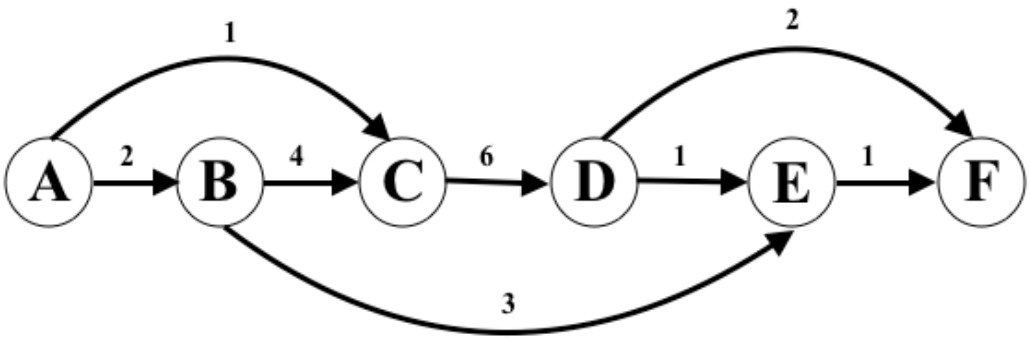
If we look at the stack-trace tree we see why this happens, for fib(5) = F(5) for example we get:



We recalculate all n – 1 values for every n twice and for all parameters n – 1 we calculate all previous values twice again, leading to such a horrible worst-case time.
In order to fix this recursive top to bottom approach, we need to store every access to the Fibonacci function, store it and make sure it only gets calculated once.

Shortest Path in DAG

Every dynamic programming algorithm has an underlying DAG (directed acyclic graph) substructure. The simplest example is the shortest path within a DAG.



The pseudo code is again quite easy:

$shortestPath(n)$: $\min_{1 \le i \le M} cost(path_i) + shortestPath(origin(path_i))$

```
struct path {
    int from, cost;
    path(int _from, int _cost) : from(_from), cost(_cost) {}
};

int shortestPath(vector<vector<path> > DAG) {
    vector<int> memoization(1, 0);
    for(int i = 0; i < DAG.size(); ++i) {
        int minCost = MAX_INTEGER;
        for(int j = 0; j < DAG[i].size(); ++j) {
            int cost = DAG[i][j].cost + memoization[DAG[i][j].from];
            if(cost < minCost) minCost = cost;
        }
        memoization.push_back(minCost);
    }
    return memoization[DAG.size() - 1];
}
```

Longest Increasing Subsequence (LIS)

The longest increasing subsequence is a sequence of numbers, which are monotonically increasing from left to right.

The sequence: {2, 1, 5, 3, 4, 0}, holds multiple longest increasing subsequences. The longest possible subsequences hold the length 3: {2,3,4} and {1,3,4} are both of maximum size. {1,5} for example doesn't have a maximum size.

In order to use memorization, we simply define all longest increasing subsequences of every subset of sequences:

{2}	has a longest subsequence of	{2}
{2,1}	has a longest subsequence of	{2}(or{1})
{2,1,5}	has a longest subsequence of	{2,5}
{2,1,5,3}	has a longest subsequence of	{2,5}
{2,1,5,3,4}	has a longest subsequence of	{2,3,4}
{2,1,5,3,4,0}	has a longest subsequence of	{2,3,4}

The sequenceList is the LIS up to the point i. Definition:

sequenceList[i] := MAX(sequenceList[j] | j < i, input[j] < input[i]) + input[i]

Suboptimal O(N^2) implementation (for O (N log N) see next page)

```
vector<int> longestIncreasingSubsequence(vector<int> input)
{
   vector<vector<int>> sequenceList(input.size());
   sequenceList[0].push_back(input[0]);
   for(int i = 1; i < input.size(); i++)
   {
      for(int j = 0; j < i; j++)
      {
         if(input[j] < input[i] && sequenceList[j].size() >
sequenceList[i].size()) {
            sequenceList[i] = sequenceList[j];
         }
      }
      sequenceList[i].push_back(input[i]);
   }
   int maxSize = 0, maxID = 0;
   for(int i = 0; i < sequenceList.size(); i++)
   {
      if(sequenceList[i].size() > maxSize) {
         maxSize = sequenceList[i].size();
         maxID = i;
      }
   }
   return sequenceList[maxID]; }
```

Generalized Subsequence search

Finding any subsequence can be implemented in N log N. In order to achieve this we simple sort all previously calculated subsequences (or put them directly in a set) and only always access the first viable input. This results in finding the required subsequence on average in log N. *Code for LIS in n log n:*

```
struct sequence
{
    vector<int> data;
    int id;
    sequence() {}
    sequence(vector<int> _data, int _id) : data(_data), id(_id) {}
};
struct cmpSequence {
    bool operator() (sequence const & lhs, sequence const & rhs)
const
    {
        if(lhs.data.size() == rhs.data.size())
        {
            return lhs.data.back() < rhs.data.back();
        }
        return lhs.data.size() > rhs.data.size();
    }
};

vector<int> longestIncreasingSubsequence(vector<int> input)
{
    set<sequence, cmpSequence> sequenceSet;
    for(int i = 0; i < input.size(); i++)
    {
        sequence currentSequence({}, i);
        for(auto &temp : sequenceSet)
        {
            //RULESET: Change this to any rule you want.
            if(input[temp.id] < input[i]) {
                currentSequence.data = temp.data;
                break; //!!!
            }
        }
        currentSequence.data.push_back(input[i]);
        sequenceSet.insert(currentSequence);
    }
    return (*sequenceSet.begin()) . data;
}
```

Minimum Edit Distance

Minimum Edit Distance is a perfect example of a dynamic programming table. The exercise consists of morphing one string into another with minimal cost to certain rules. The Levensthein Distance has the following operation costs:
Match = 0$
DeletionPrice = 1$
InsertPrice = 1$
ReplacePrice = 2$

M = Distance-Matrix

$$M\ i,j = \begin{cases} if(aj == bi) & return\ M\ i,j \\ else & min\begin{bmatrix} M\ i-1,j + deletionPrice \\ M\ i,j-1 + insertionPrice \\ M\ i-1,j-1 + \ replacePrice \end{bmatrix} \end{cases}$$

Another case that can be added is the Transposition, which swaps 2 adjacent values. In order to do that, we have to check if ai == b j–1 && ai-1 == bj.

Adding this new cost creates the Damerau-Levensthein distance, which detects around 80% of all mistakes in human typing. The described algorithm on the next page describes this example.

Other edit distances:
Hamming distance only allows replace price and requires the strings to be the same size.
Jaro-Winkler distance can be obtained by only using transposition.

This algorithm can be implemented in n + m memory, by actually only storing diagonal values, meaning only those values that will be accessed later from the memoization.

Wagner-Fisher Algorithm
Worst Complexity: O(N*M)
Memory Usage: M * N

```
int damerauLevenstheinDistance (string a, string b)
{
   vector<vector<int>> matrix (a.size() + 1, vector<int>(b.size() + 1, 0));

   for(int i = 0; i < a.size(); i++) matrix[i][0] = i; // Prepare Matrix
   for(int i = 0; i < b.size(); i++) matrix[0][i] = i;

   //DO FOR EACH MATRIX ELEMENT
   for(int i = 1; i < a.size() + 1; i++)
   {
      for(int j = 1; j < b.size() + 1; j++)
      {
            int matchingCost = INF;
            if(a.at(i - 1) == b.at(j - 1)) matchingCost = matrix[i-1][j-1] + 0;
            int replaceCost = matrix[i-1][j-1] + 2; //Replace
            int insertCost = matrix[i][j-1] + 1; //Insert
            int delCost = matrix[i-1][j] + 1; //Delete
            int transpositionCost = INF;
            if(i > 1 && j > 1 && a.at(i) == b.at(j - 1) && a.at(i - 1) == b.at(j))
transpositionCost = matrix[i-2][j-2] + 1;

            matrix[i][j] = min(matchingCost, min(replaceCost,
min(insertCost, min(delCost, transpositionCost))));
      }
   }
   return matrix[a.size()][b.size()];
}
```

Check out the GET MATRIX section in *lowest common subsequence* to get an actual result and not only a value.

Longest Common Substring

The longest common substring is the task in searching the largest substring in other substrings.

$$M\ i,j = \begin{cases} if\ (aj == bi) & return\ (M\ i-1, j-1) + 1 \\ else & return\ 0 \end{cases}$$

Algorithm which only outputs length:

```
string longestCommonSubstring (string a, string b)
{
   vector<vector<int>> matrix (a.size() + 1, vector<int>(b.size() + 1, 0));

   //DO FOR EACH MATRIX ELEMENT
   int maxCount = 0;
   for(int i = 1; i < a.size() + 1; i++)
   {
      for(int j = 1; j < b.size() + 1; j++)
      {
         if(a.at(i - 1) == b.at(j - 1)) matrix[i][j] = matrix[i - 1][j - 1] + 1;
      }
   }

   return maxCount;
}
```

Full algorithm, which returns one possible longest common substring:

```
string longestCommonSubstring (string a, string b)
{
   vector<vector<int>> matrix (a.size() + 1, vector<int>(b.size() + 1, 0));

   //DO FOR EACH MATRIX ELEMENT
   int maxCount = 0, maxI = 0, maxJ = 0;
   for(int i = 1; i < a.size() + 1; i++)
   {
      for(int j = 1; j < b.size() + 1; j++)
      {
         if(a.at(i - 1) == b.at(j - 1)) {
            matrix[i][j] = matrix[i - 1][j - 1] + 1;
            if(matrix[i][j] > maxCount) {
               maxCount = matrix[i][j];
               maxI = i;
               maxJ = j;
            }
         }
      }
   }

   string output = "";
   while(matrix[maxI][maxJ] != 0)
   {
      output += a.at(maxI - 1);
      maxI--; maxJ--;
   }
   return output;
}
```

Longest Common Subsequence

The longest common subsequence problem is similar to the longest common substring problem, with the exception that now it is allowed to have characters in-between the sequence itself. The problem is NP-complete for more than 2 sequences.

$$M\ i,j = \begin{cases} if\,(aj == bi) & return\ (M\ i-1, j-1) + 1 \\ else & return \max\,(M\ i-1, j\quad ,\ \ M\ i, j-1) \end{cases}$$

```
string longestCommonSubsequence (string a, string b)
{
   vector<vector<int>> matrix (a.size() + 1, vector<int>(b.size() + 1, 0));

   //DO FOR EACH MATRIX ELEMENT
   int maxCount = 0;
   for(int i = 1; i < a.size() + 1; i++)
   {
      for(int j = 1; j < b.size() + 1; j++)
      {
         if(a.at(i - 1) == b.at(j - 1)) matrix[i][j] = matrix[i - 1][j - 1] + 1;
         else
         {
            matrix[i][j] = max(matrix[i][j - 1]  ,  matrix[i - 1][j]);
         }
      }
   }

   //GET MATRIX
   string output = "";
   int it = a.size(), jt = b.size();
   while (it != 0 && jt != 0) {
      int previous = matrix[it][jt];
      if(it != 0 && matrix[it - 1][jt] == previous) it--;
      else if(jt != 0 && matrix[it][jt - 1] == previous) jt--;
      else {
         output += a.at(it - 1);
         it--; jt--;
      }
   }
   reverse(output.begin(), output.end());
   return output;
}
```

Matrix chain multiplication (MCM)

In order to multiply a queue of matrices together, we can use the associative property. First of all why is MCM a problem in informatics and not mathematics? Well we can increase the running time when multiplying a chain of matrices, depending on their order of multiplication. Since a matrix is not commutative and only associative, we can do this by using brackets. This problem doesn't care about the values of the matrix, only the size of the matrix.

Another property of matrix multiplication is that matrix A must have the same amount of columns as matrix B has rows.

$A_{columns} = B_{rows}$

If the initial order of the matrices is correct, we can place brackets wherever we want, hence the above rule still applies.

Due to the nature of matrix multiplication the size of the new matrix C is the number of rows of A multiplied by the number of columns on B. So $C_{rows} = A_{rows}$ and $C_{columns} = B_{columns}$

So in order to calculate the amount of multiplications required to multiply 2 matrices A * B we can use this formula:

Number of calculations = $A_{rows} * A_{columns} * B_{columns}$

Because either $A_{columns}$ or B_{rows} falls short in the above equation, the order of matrices matter in order to multiply 2 matrices.

Now let's imagine we multiply the following 3 matrices A * B * C:

$A_{rows} = 10$, $A_{columns} = 100$

$B_{rows} = A_{columns} = 100$, $B_{oolumns} = 5$

$C_{rows} = B_{oolumns} = 5$, $C_{columns} = 50$

Let's calculate the matrix using the following brackets:

(A * B) * C

D will be the sub-result of A * B:

$D_{rows} = A_{rows} = 10$

$D_{columns} = B_{columns} = 5$

Operations required to get to D = 10 * 100 * 5 = 5'000

D * C = R(esult)

$R_{rows} = D_{rows} = A_{rows} = 10$

$R_{columns} = C_{columns} = 50$

Operations required to get from D to R = 10 * 5 * 50 = 2'500

Total number of operations needed = 5'000 + 2'500 = **7'500**

Now let's calculate the matrix using the other brackets:
A * (B * C)
D will be the sub-result of B * C:
$D_{rows} = B_{rows} = 100$
$D_{columns} = C_{columns} = 50$
Operations required to get to D = 100 * 5 * 50 = 25'000
A * D = R(esult)
$R_{rows} = A_{rows} = 10$
$R_{columns} = D_{columns} = C_{columns} = 50$
Operations required to get from D to R = 10 * 100 * 50 = 50'000
Total number of operations needed = 25'000 + 50'000 = **75'000**.
After this example you can hopefully quite clearly see, that this is a problem of informatics. The second method involved 10 times less calculations than the first one. Multiplying huge chains of matrices together, this number can get way bigger then 10 times the difference and it was so important, that people came up with very efficient algorithms.
So how is this problem related to DP? Well this is a little harder to realize, especially when the optimal substructure is quite complicated. We will require N^2 space, where N are the amount of matrices (this is usually way less memory compared to the actual memory within the matrices).

$$OS[i,j] = \begin{cases} if(i == j)\ return\ 0 \\ if(i < j)\ return \min\left(OS[i+k,j] + p_i * p_{k+1} * p_{j+1}\right) \end{cases}$$

for all i <= k < j
P is the array of rows in every matrix. The first row however will get inserted in the ff.
So if we were to multiply A * B * C * D
A = (30 x 5), B = (5 x 10), C = (10 x 10), D = (10 x 30)
P would look like: P = {30, 5, 10, 10, 30}
To get a result and not only the amount of multiplication minimally possible, we also use a 2-dimensional recorder array. Also realize, that the matrix never accesses elements where i > j, making the array look like a pyramid.
The algorithm runs in $O(N^3)$.

Full Memoized implementation:

```
#define INF 9999999999
#define LL long long int
struct Matrix {
    vector<vector<LL>> data; //Unimportant for our task
    LL rows, columns;
    Matrix(vector<vector<LL>> _data) : data(_data) {
        rows = _data.size();
        columns = _data[0].size();
    }
};

Matrix operator*(const Matrix &lhs, const Matrix& rhs)
{
    vector<vector<LL>> newMatrix;
    LL same = lhs.columns; //= rhs.row;
    for(int i = 0; i < lhs.rows; i++)
    {
        for(int j = 0; j < rhs.columns; j++)
        {
            LL tempResult = 0;
            for(int k = 0; k < same; k++)
            {
                tempResult += lhs.data[i][k] * lhs.data[k][j];
            }
            newMatrix[i][j] = tempResult;
        }
    }
    Matrix result(newMatrix);
    return result;
}

vector<vector<LL>> recorder;
vector<Matrix> unorderedInput;
Matrix multiplyFromRecorder(int start, int end)
{
    if(start == end) return unorderedInput[start];
    else {
        int split = recorder[start][end];
        return multiplyFromRecorder(start, split) *
multiplyFromRecorder(split + 1, end);
    }
}
```

```
Matrix multiplyChain (vector<Matrix> &tempInput) //unorderedInput
{
    unorderedInput = tempInput;
    LL N = unorderedInput.size();
    vector<vector<LL>> OS (N, vector<LL> (N, 0));
    recorder = OS;

    vector<LL> P(N + 1, 0);
    for(int i = 0; i < N; i++) P[i] = unorderedInput[i].rows;
    P[N] = unorderedInput[0].rows;

    int i = -1, j = -1, jt = 0;
    do {
        ++i; ++j;
        if(j >= N) {
            ++jt;
            i = 0;
            j = jt;
        }
        if(i != j) OS[i][j] = INF;
        for(int k = i; k < j; k++)
        {
            LL tempResult = OS[i][k] + OS[k + 1][j] + P[i] * P[k + 1] * P[j + 1];
            if(OS[i][j] > tempResult) {
                OS[i][j] = tempResult;
                recorder[i][j] = k + 1;
            }
        }
    } while (i != 0 || j != N - 1);
    return multiplyFromRecorder(0, N - 1);
}
```

Full Recursive Implementation:

```
#define INF 9999999999
#define LL long long int
struct Matrix {
    vector<vector<LL>> data; //Unimportant for our task
    LL rows, columns;
    Matrix() {}
    Matrix(vector<vector<LL>> _data) : data(_data) {
        rows = _data.size();
        columns = _data[0].size();
    }
    Matrix(int _rows, int _columns) : rows(_rows), columns(_columns) {}
};

Matrix operator*(const Matrix &lhs, const Matrix& rhs)
{
    vector<vector<LL>> newMatrix;
    LL same = lhs.columns; //= rhs.row;
    for(int i = 0; i < lhs.rows; i++)
    {
        for(int j = 0; j < rhs.columns; j++)
        {
            LL tempResult = 0;
            for(int k = 0; k < same; k++)
            {
                tempResult += lhs.data[i][k] * lhs.data[k][j];
            }
            newMatrix[i][j] = tempResult;
        }
    }
    Matrix result(newMatrix);
    return result;
}



vector<vector<LL>> OS;
vector<vector<LL>> recorder;
vector<Matrix> unorderedInput;
vector<LL> P;
```

```
Matrix multiplyFromRecorder(int start, int end) {
    if(start == end) return unorderedInput[start];
    else {
        int split = recorder[start][end];
        return multiplyFromRecorder(start, split) *
multiplyFromRecorder(split + 1, end);
    }
}

void recursiveChain(int i, int j)
{
    if(OS[i][j] == INF)
    {
        if(i == j) OS[i][j] = 0;
        else
        {
            for(int k = i; k < j; k++)
            {
                recursiveChain(i, k);
                recursiveChain(k + 1, j);
                LL tempResult = OS[i][k] + OS[k + 1][j] + P[i] * P[k + 1] * P[j+1];
                if(OS[i][j] > tempResult) {
                    OS[i][j] = tempResult;
                    recorder[i][j] = k + 1;
                }
            }
        }
    }
}
Matrix multiplyChain (vector<Matrix> &tempInput) //unorderedInput
{
    unorderedInput = tempInput;
    LL N = unorderedInput.size();
    OS.clear();
    recorder.clear();
    P.clear();
    OS.resize(N, vector<LL> (N, INF));
    recorder.resize(N, vector<LL> (N, 0));
    P.resize(N + 1, 0);
    for(int i = 0; i < N; i++) P[i] = unorderedInput[i].rows;
    P[N] = unorderedInput[0].rows;
    recursiveChain(0, N - 1);
    return multiplyFromRecorder(0, N - 1);   }
```

Maximum Empty Rectangle (MER)
Worst-case complexity: O(X * Y)
The maximum empty rectangle is the problem of finding the largest possible rectangle in a 2-Dimensional boolean array. If the spot is empty the bool is true, else it is false.

```
int maximumEmptyRectangle(vector<vector<bool> > &field) {
    int n = field[0].size() + 1;
    int maxArea = 0;
    vector<int> cache(n, 0);

    for (int i = 0; i < field.size(); ++i) {
        stack<int> rects;
        for (int j = 0; j < n; ++j) {
            if (j < n - 1) {
                if (field[i][j]) cache[j] += 1;
                else cache[j] = 0;
            }
            while (!rects.empty() && cache[rects.top()] >= cache[j]) {
                int rectHeight = cache[rects.top()];
                rects.pop();
                int rectWidth = j;
                if(!rects.empty()) rectWidth = j - rects.top() - 1;
                if (rectHeight * rectWidth > maxArea)
maxArea = rectHeight * rectWidth;
            }
            rects.push(j);
        }
    }
    return maxArea;
}
```

Unbound Knapsack

Knapsack is an NP-Complete problem, although its time complexity O(n*W), can seem confusing. First let's explain the problem. You are a robber and can steal from a sortiment of items, which each have a price and weight. Your sack can only fit W-kilograms of material. What is the maximal value the can robber take with him?

Again the DAG lying underneath can be represented as value of w and is in fact the same problem as the shortest distance, with the difference that we need to find the maximal route.

PriceInWeight(w) = $\max_{1\leq 1\leq N} PriceInWeight(w - price_i) + price_i$

```
struct item {
    int price;
    int weight;
    item(int _price, int _weight, int _count) : price(_price),
weight(_weight) {}
};

int knapsack(vector<item> store, int maxWeight) {
    vector<int> priceInWeight;
    for(int i = 0; i <= maxWeight; ++i) {
        int maxPrice = 0;
        for(int j = 0; j < store.size(); ++i) {
            int price = 0;
            if(i - store[j].weight >= 0) price = priceInWeight[i - store[j].price]
+ store[j].price;
            else if(store[j].weight <= i) price = store[j].price;
            if(price > maxPrice) maxPrice = price;
        }
        priceInWeight.push_back(maxPrice);
    }
    return priceInWeight[maxWeight];
}
```

There also is the 2-dimensional variant of the knapsack problem, the 1/0 knapsack problem. Now instead of having every item infinitely available, every item can only be used once.

PiW(w, j) = $\max_{1\leq 1\leq N}\{PiW(w - p_i, j - 1) + p_i, PiW(w, j - 1)\}$

Geometry in Informatics

Points

Points need to store 2 coordinates: x and y. You could of course also add another axis z or k for higher order points.

```
struct point
{
    int x, y;
    point(int _x, int _y) : x(_x), y(_y) {}
};
```

Line Segments

Line segments are the connections between 2 points.

```
struct line
{
    point a, b;
    line(point _a, point _b) : a(_a), b(_b) {}
};
```

The mathematical form f(x) = m * x + q can always be calculated:

```
struct line
{
    point a, b;
    float m, q;
    line(point _a, point _b) : a(_a), b(_b) {
        m = (b.y - a.y) / (b.x - a.x);
        q = a.y - (m * a.x);
    }
};
```

Line Intersection Point

```
point intersection(line first, line second)
{
    point output;
    output.x = (second.q - first.q) / (first.m - second.m);
    output.y = first.m * output.x + first.q;
    return output;
}
```

Line Segment Intersection Point

```
point segmentIntersection(line f, line s)
{
   point iP = intersection(first, second);

   if ( //Check if intersection point is inside the line segment
       iP.x > max(min(f.a.x, f.b.x), min(s.a.x, s.b.x)) &&
       iP.x < min(max(f.a.x, f.b.x), max(s.a.x, s.b.x)) &&
       iP.y > max(min(f.a.y, f.b.y), min(s.a.y, s.b.y)) &&
       iP.y < min(max(f.a.y, f.b.y), max(s.a.y, s.b.y))
      ) return iP;
   else return NULL;
}
```

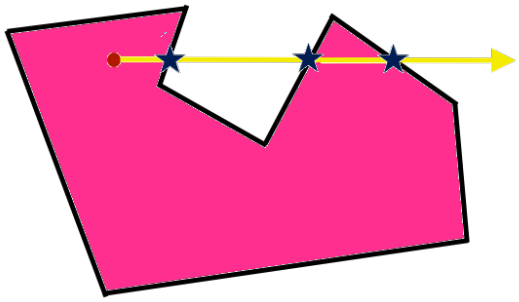
Determinant

```
float determinant(float a, float b, float c, float d)
{
   return a * d - b * c;
}
```

Area of a Polygon

```
float polygonArea(vector<line> polygon)
{
   float area = 0;
   int j = polygon.size() - 1;

   vector<float> X, Y;
   for(int i = 0; i < polygon.size(); i++)
   {
      X.push_back( abs(polygon[i].b.x - polygon[i].a.x) );
      Y.push_back( abs(polygon[i].b.y - polygon[i].a.y) );
   }
   for (int i = 0; i < polygon.size(); i++)
   {
      area = area +  (X[j]+X[i]) * (Y[j]-Y[i]);
      j = i;
   }
   return area/2;
}
```

Point in Polygon (PIP)
To check whether a point is inside or outside a polygon one can use a ray-casting algorithm. A line starting from the point P, will be drawn to the right of the polygon.



Every line has a multiple of 2 intersections with the polygon. If a segment line, which starts at the given point and ends in infinity, has an uneven amount of intersections with the polygon, the point is inside the polygon. Elsewise it can't be.
Algorithm in O(M).

```
bool pip(point included, vector<line> polygon)
{
    int count = 0;
    for(int i = 0; i < polygon.size(); i++)
    {
        if(polygon[i].a.y > polygon[i].b.y) swap(polygon[i].a, polygon[i].b);

        float diff=(included.y-polygon[i].a.y)/(polygon[i].b.y-polygon[i].a.y);
        if(diff >= 0 && diff <= 1)
        {
            float newX = (diff * (polygon[i].b.x - polygon[i].a.x)) +
polygon[i].a.x;
            if(newX >= included.x) count++;
        }
    }
    return (count % 2) == 1;
}
```

Convex Hull
Convex Hull is a typical informatics problem. It is basically the task of spanning a rubber band around all points and finding the corner points of that rubber band, of that hull. The *extreme points* are the points on top of the convex hull. The shown algorithms calculate the extreme points.

Before calculating the convex hull it is possible to quite easily eliminate points, which are definitely not in the convex hull. Consider using the simple quadrilateral: min(x + y), max(x + y), min(x - y), max(x - y)

Grahams Scanline Algorithm

This algorithm has a complexity of O(n log n), due to sorting, where n is the amount of points on the convex hull. This is the recommended algorithm.

```
struct point
{
    int x, y;
    point(int _x, int _y) : x(_x), y(_y) {}
};

bool operator<(const point &a, const point &b)
{
    if(a.x == b.x) return a.y < b.y;
    return a.x < b.x;
}

long long det(long long a, long long b, long long c, long long d)
{
    return (a * d) - (b * c);
}

int orientation(point &a, point &b, point &c)
{
    return sign( det(b.x - a.x, c.x - a.x, b.y - a.y, c.y - a.y) );
}

vector<point> result;
vector<point> input;
void recursiveScanline(int pos)
{
    if(result.size() < 2) result.push_back(input[pos]);
    else {
        int rotation = orientation(result[result.size() - 2], result.back(),
input[pos]);

        if( rotation <= 0 ) {
            result.pop_back();
            recursiveScanline(pos);
        }
        else {
            result.push_back(input[pos]);
        }
    }
}
```

```
vector<point> scanline(vector<point> &_input)
{
    input = _input;
    sort(input.begin(), input.end());
    vector<point> output;
    result.clear();
    for(int i = 0; i < input.size(); i++) recursiveScanline(i);
    for(int i = 0; i < result.size(); i++) output.push_back(result[i]);
    result.clear();
    for(int i = input.size() - 1; i >= 0; i--) recursiveScanline(i);
    for(int i = 1; i < result.size() - 1; i++) output.push_back(result[i]);
//FIRST AND LAST HAVE BEEN ADDED ABOVE
    return output;
}
```

Jarvis March Gift Wrapping Algorithm

This algorithm is *ouput-sensitive* and has a complexity of O(n h), where n is the amount of points on the convex hull and h is the amount of points, which will be on the hull at the end. Since measuring h requires to solve the problem, it has a worst-case complexity of $O(n^2)$, which is much slower then grahams scanline algorithm.

```
vector<point> wrap (vector<point> input) {
    vector<point> solution;
    int left = 0;
    for (int i = 1; i < input.size(); i++) {
        if (input[i] < input[left]) left = i;
    }

    int p = left;
    int next = (left + 1) % input.size();
    do {
        solution.push_back(input[p]);
        for (int i = 0; i < input.size(); i++) {
            if ( orientation(input[p], input[next], input[i]) < 0) next = i;
        }
        p = next;
        next = left;
    } while (p != left);
    return solution;
}
```

Circle from 3 points

Because the slope m can result in a division by zero, this algorithm won't work for all 3 points. Also when 3 points are on a line, the circle will be infinitely large, so sort out the corner cases.

```
point circleCenter(point A, point B, point C) {
    line l1 (A, B);
    line l2 (B, C);
    point output(0, 0);
    output.x = (l1.m * l2.m * (A.y - C.y) + l2.m * (A.x + B.x)
              - l1.m * (B.x+C.x) )/(2* (l2.m-l1.m) );
    output.y = -1*(output.x - (A.x+B.x)/2)/l1.m +  (A.y+B.y)/2;
    return output;
}
```

Graphics Algorithms

Line Drawing

Instead of computing f(x) = 2 x + 3; everytime, which requires a multiplication, we instead use Bresenham algorithm which only requires one multiplication (division) and not N multiplications.

Bresenham Algorithm

Implementation in 2D:

```
void bresenham-2D(int x1, int y1, int x2, int y2)
{
  //IMPORTANT FOR APPROXIMATION
  point(x1, y1);
  point(x2, y2);

  float diffX = x2 - x1;
  float diffY = y2 - y1;

  float biggest = max(abs(diffX), abs(diffY)); //IMPORTANT ABS (X) due to division sub-problem
  float dirX = 0;
  float dirY = 0;
  if (biggest != 0)
  {
    dirX = diffX / biggest;
    dirY = diffY / biggest;
  }

  float currentX = x1;
  float currentY = y1;
  do //DO-WHILE due to corner rounding.
  {
    currentX += dirX;
    currentY += dirY;
    point(currentX, currentY);
  }
  while (abs (currentX - x2) > 1 || abs(currentY - y2) > 1); //Only when all 2 positions are smaller then 1, which is guaranteed to happen, stop the loop.
}
```

Implementation in 3D:

```
void bresenham-3D(int x1, int y1, int z1, int x2, int y2, int z2)
{
  //IMPORTANT FOR APPROXIMATION
  point(x1, y1, z1);
  point(x2, y2, z2);

  float diffX = x2 - x1;
  float diffY = y2 - y1;
  float diffZ = z2 - z1;

  float biggest = max(max(abs(diffX), abs(diffY)), abs(diffZ));
//IMPORTANT ABS (X) due to division sub-problem
  float dirX = 0;
  float dirY = 0;
  float dirZ = 0;
  if (biggest != 0)
  {
    dirX = diffX / biggest;
    dirY = diffY / biggest;
    dirZ = diffZ / biggest;
 }

  float currentX = x1;
  float currentY = y1;
  float currentZ = z1;
  do //DO-WHILE due to corner rounding.
  {
    currentX += dirX;
    currentY += dirY;
    currentZ += dirZ;
    point(currentX, currentY, currentZ);
  }
  while (abs (currentX - x2) > 1 || abs(currentY - y2) > 1 || abs(currentZ -
z2) > 1); //Only when all 2 positions are smaller then 1, which is
guaranteed to happen, stop the loop.
}
```

Edge Detection

In a pixel grid M[x][y] is defined as:

$$M[x][y] = \begin{bmatrix} P[x-1][y-1] & P[x][y-1] & P[x+1][y-1] \\ P[x-1][y] & P[x][y] & P[x+1][y] \\ P[x-1][y+1] & P[x][y+1] & P[x+1][y+1] \end{bmatrix}$$

Where P[x][y] = gray value of pixels at x, y.

Sobel operator (Canny algorithm) is a filter, which is done twice, once for the x directions and once for the y directions; it is later summed up together and a threshold is defined in order to create various ways. For every pixel we multiply the grayscale value by the following convolution matrices. The resulting matrices get summed up.

Grayscale = 0.2126 R + 0.7152 G + 0.0722 B

$$tempX = sum\left(M[x][y] * \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \right)$$

$$tempY = sum\left(M[x][y] * \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \right)$$

The resulting matrices are then summed up together.
After the summation, we calculate the pixels size with:

$$P[x][y] = \sqrt{tempX^2 + tempY^2}$$

Laplace operator is another operator, which can be multiplied to your grid. It directly gives the result of P[x, y].

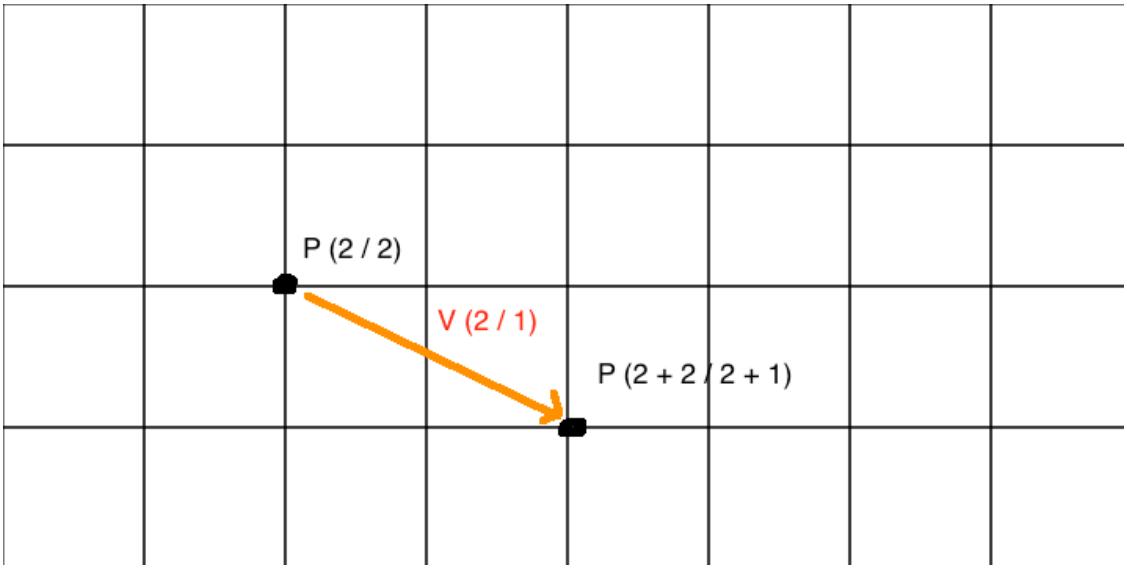
$$P[x][y] = sum\left(M[x][y] * \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \right) \text{ or}$$

$$P[x][y] = sum\left(M[x][y] * \begin{bmatrix} 0.5 & 1 & 0.5 \\ 1 & -6 & 1 \\ 0.5 & 1 & 0.5 \end{bmatrix} \right)$$

in order to include diagonals.

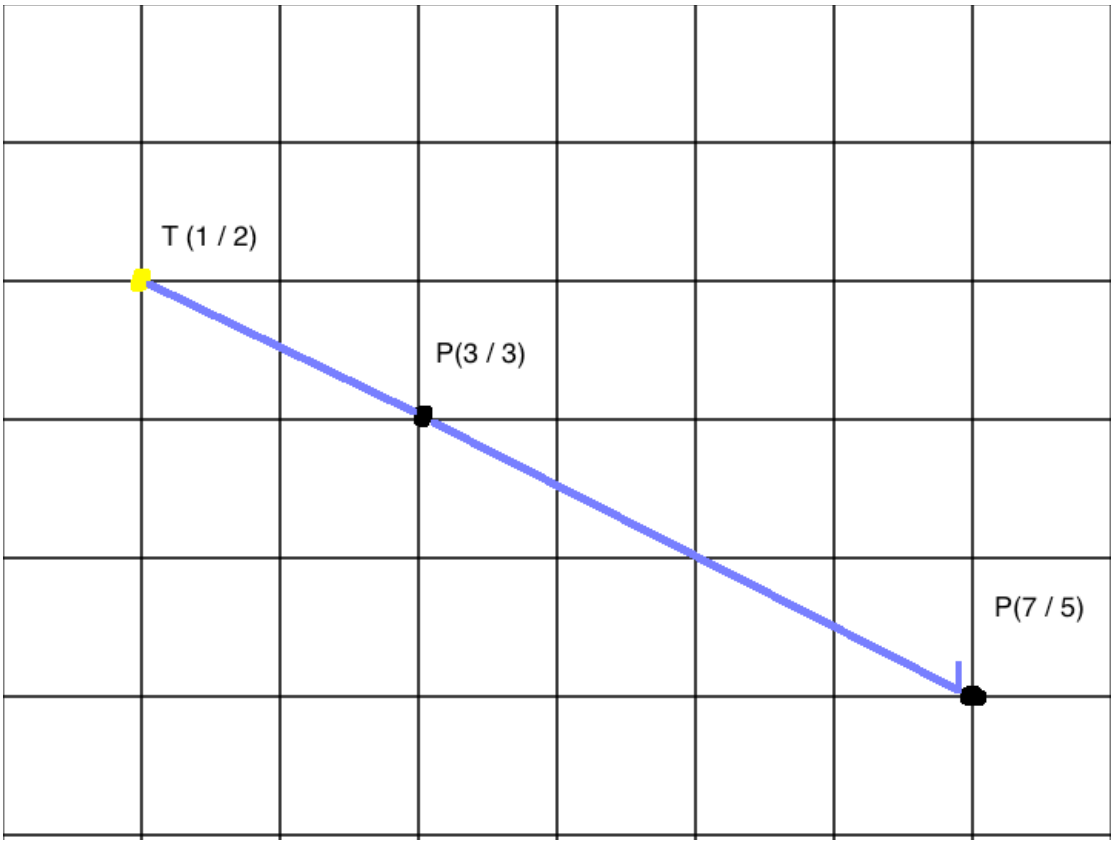
There are various other filters such as the Roberts operator, which only accesses diagonal values or Prewitt operator, which is basically sobels operator without using edges.

Translating a Point




```
point transPoint2D(point p, point trans) {
    point toReturn (p.x + trans.x, p.y + trans.y);
    return toReturn;
}


point transPoint3D(point p, point trans)
{
    point toReturn (p.x + trans.x, p.y + trans.y, p.z + trans.z);
    return toReturn;
}
```

Scaling a Point
T: Transformation Point
P (3 / 3): Point, which gets scaled.
P(7 / 5): Point, which got scaled.



Scaling at point (0,0) requires to simply multiply every coordinate by the scale factor:
newX = x * scale
newY = y * scale

Now we want to scale (x,y) at point (transX, transY).
newX = ((x - transX) * scale) + transX
newY = ((y - transY) * scale) + transY

```
point scalePoint2D(point p, point trans, float scale)
{
    float toReturnX = ((p.x - trans.x) * scale) + trans.x;
    float toReturnY = ((p.y - trans.y) * scale) + trans.y;
    point toReturn (toReturnX, toReturnY);
    return toReturn;
}

point scalePoint3D(point p, point trans, float scale)
{
    float toReturnX = ((p.x - trans.x) * scale) + trans.x;
    float toReturnY = ((p.y - trans.y) * scale) + trans.y;
    float toReturnZ = ((p.z - trans.z) * scale) + trans.z;
    point toReturn (toReturnX, toReturnY, toReturnZ);
    return toReturn;
}
```

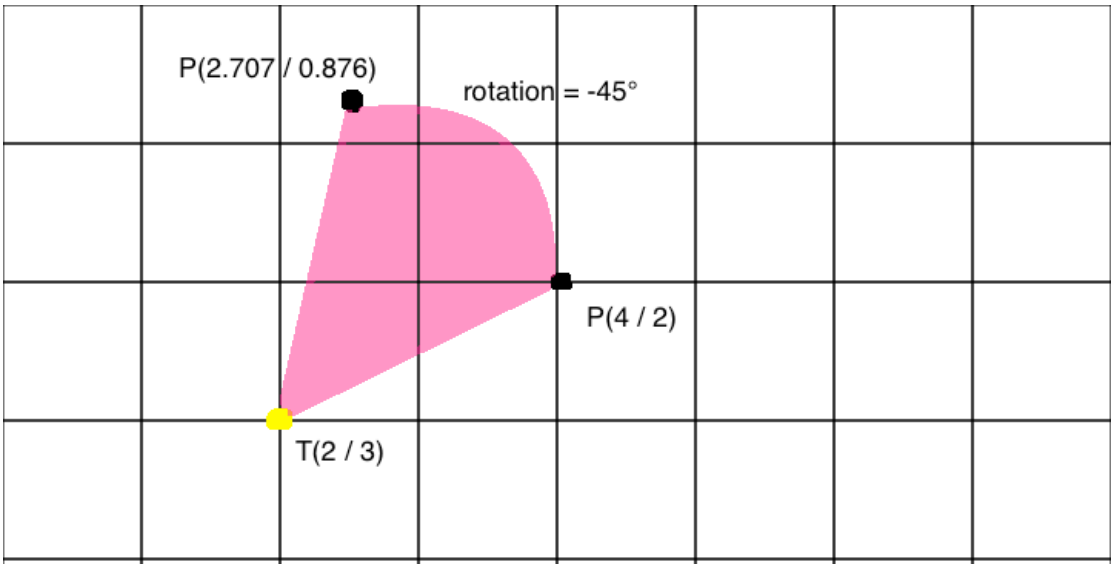
Realize that we can invert an image by the X and Y coordinate by scaling it to -1. To invert by either X or Y coordinate we simply need 2 scale (3D: 3 scale) directions.

```
point scalePoint2D(point p, point trans, point scale)
{
    float toReturnX = ((p.x - trans.x) * scale.x) + trans.x;
    float toReturnY = ((p.y - trans.y) * scale.y) + trans.y;
    point toReturn (toReturnX, toReturnY);
    return toReturn;
}
```

For 3D scaling add a scaleZ option (or look up the scaleZ matrix).

Rotating a Point

Scaling works by setting a transformation point and translating.
T: Transformation Point



First realize that we have an inverted coordinate systems, which means we rotate in a negative direction.
Also remember I calculate in radians!

To rotate around the center P (0 / 0) we need to calculate the following:
newX = cos(rotation) * x – sin(rotation) * y
newY = sin(rotation) * x + cos(rotation) * y

Now transform the transformation point:

```
point rotatePoint2D (point p, point trans, float rotation)
{
    float transRotX = trans.x - cos(rotation) * trans.x + sin(rotation) *
trans.x;
    float transRotY = trans.x - sin(rotation) * trans.x - cos(rotation) *
trans.x;
    float toReturnX = cos(rotation) * p.x - sin(rotation) * p.y + transRotX ;
    float toReturnY = sin(rotation) * p.x + cos(rotation) * p.y + transRotY;
    point toReturn (toReturnX, toReturnY);
    return toReturn;
}
```

While these translations work, for more complex transformations look up *vector transformation* in the matrix section of the book.