Algorithms and Data Structure Lecture

Computational complexity - how fast the algorithm accomplishment time grows, if the volume of data grows to infinity

Θ (theta) notation

~~Assume that $T(n) = \Theta(g(n))$~~

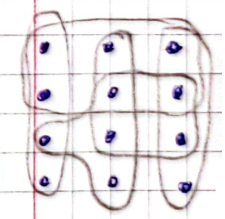
O notation (asymptotic upper bound)

Ω notation (asymptotic lower bound

Θ theta	=
O ~~omega~~	<=
Omega	>=
o	<
omega	>

Dynamic programming (DP) - use to solve complex problem by breaking it down into collection of simpler subproblems, solving each of those just once, and storing their solutions.

Set cover - any family of sets over a given universe, such that their union covers whole universe



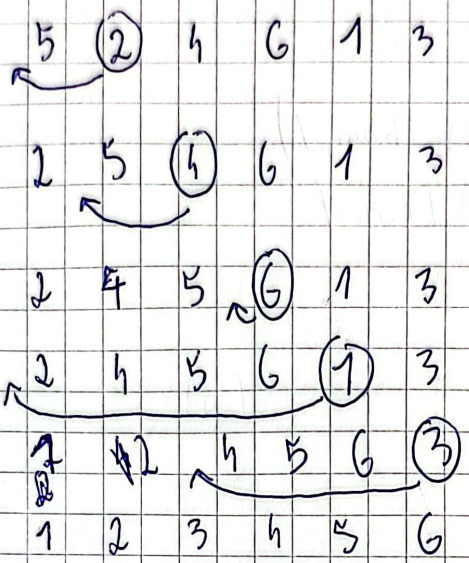
Bubble sort - compares adjacent elements and switch them if necessary

compares: $(n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2} \approx \frac{n^2}{2} = \Theta(n^2)$

Insertion sort - works similar to the way you sort cards in your hand. The array is virtually split into a sorted and unsorted part. Values from the unsorted part are picked and placed at the correct position in sorted part.

$\Theta(n^2)$ - comparisons

ex.



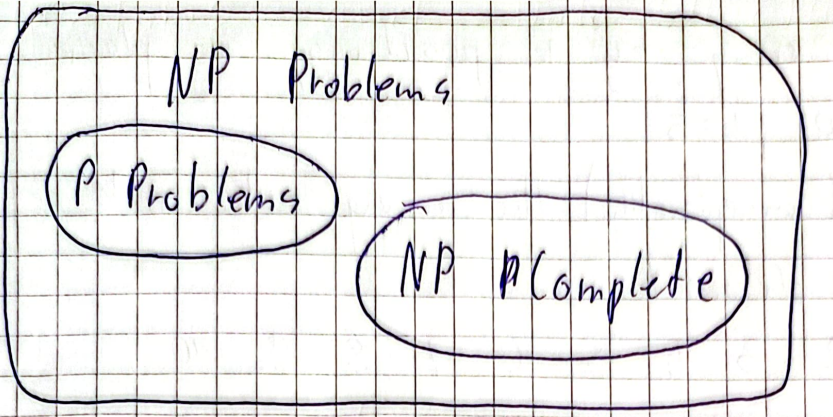
Selection sort - find the smallest element and put it in 1st place, find the next smallest element and put is 2nd and so on.

$\Theta(n^2)$ - comparisons

Stable sorting it is a sorting preserving orginal relative order of itkems having the same valu

Features of any algorithm:

- is finite
- is well-defined (each step must be defined precisely)
- input data
- output data
- is defined effectively



NP Problems - Problems for which the solution can be verified in polinomial time. It means that if someone already have a solution we can check if it is correct in polinomial time.

P Problems - Problems for which we can find solution in polinomial time

NP compleate - Most difficult problems in NP. If we could solve one of those problems in polinomial time, we could solve all NP problems.

Moore majority voting

It is an efficient algorithm that finds a majority element in a list (appears more than half time). It maintains majority element and add to count if find it or subtract if find other. If count reaches zero maj. element is changed to current and count = 1. After this alg. goes over the sequence once more counting occurences of majority el. if it appears more than half times it is a majority el. It goes over the array only 2 times so it is $\Theta(n)$ and space is $O(1)$

Rotate an array by k positions, in place

i.e. $n = 10$, $k = 3$

1. Divide (conceptually) the array into n-k left and k right elements:

1 2 3 4 5 6 7 | 8 9 10

2. Reverse left and right part

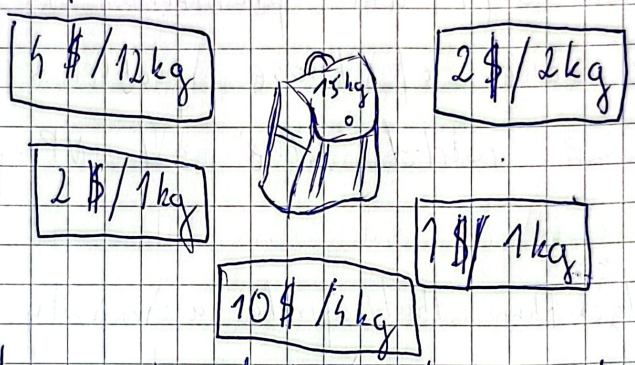
7 6 5 4 3 2 1 | 10 9 8

3. Reverse whole array

8 9 10 1 2 3 4 5 6 7

Time $O(n)$ and in place - space $O(1)$

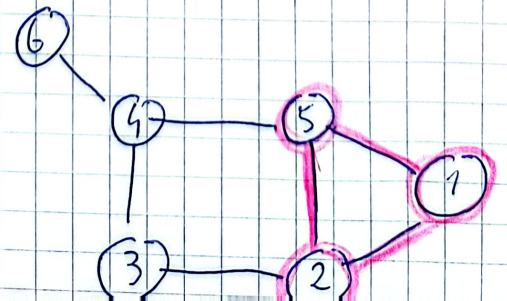
Knapsack problem



Which items should be chosen to maximize total value of items and not exceed its weight limit?

This is an example of NP complete problem. Most efficient way ie. DP let us solve this problem relatively fast for small number of data

Clique - graph's subset where each vertex is connected with every other vertex in this subset

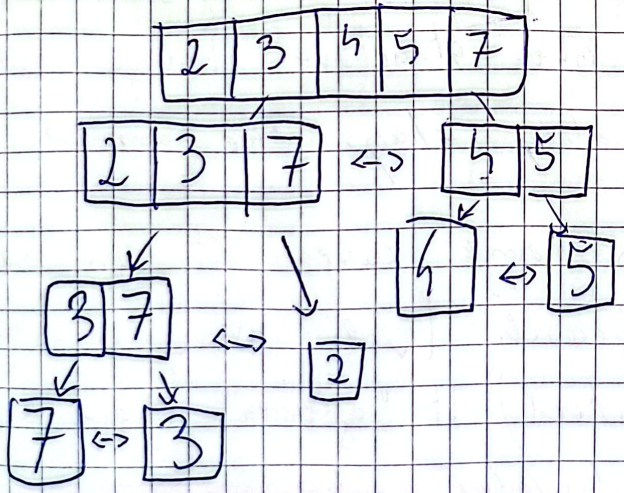


How to handle NP-complete problems in practice?
- change problem
- approximate algorithms
- heuristics; e.g. generic alg., neural network

Comparison based sorting is an operation with $\Omega(n \log n)$ worst case time complexity

Tail recursion is a technique where function calls itself as the last operation. It optimizes memory usage and avoids stack overflow errors.

Merge sort - example of "divide and conquer" alg.
Divide: n-element seq. is divided into 2 subsequences of $\frac{n}{2}$ length
Conquer: sort independently those subsequences
Combine: create one sorted seq. from 2 prev. subseq.



$\Theta(n \log n)$ even in worst case
stable
extra $\Theta(n)$ space needed

Introspection sort (introsort) - starts with quick sort and switch to heap sort if QS gets in trouble (recursion depth over some threshold)
Heap sort: $O(n \log n)$ - worst-case, in place

Quick sort: - $\Theta(n \log n)$ avg. case
- fast in practice
- $\Theta(n^2)$ worst case
- not stable

Adaptive sort - when the sorting alg. takes advantage of the existing order in the input. Sorting time is fun. of both n and disorder in seq.

Abstract data types (ADTs)

ADTs are like user defined data types which defin operations on values using functions without specifying what is inside the function and how the operation is performed. ADT are like a black box which hides the inner structure and design of data type from user. There are multiple ways to implement ADT.

i.e. stack ADT can be implemented using arrays or linked lists, however inner structure change, allow operations and functionality stays the same.

Interpolation search - in some cases it can be faster than binary search (when array is sorted and uniformly distributed). It "guesses" where a valu might be based on calculated probe.

avg. case $O(\log(\log(n)))$, worst case $O(n)$ (not uni dist.)

We can combine Interpolation and binary search to achieve avg. $O(\log(\log(n)))$ and worst of $O(\log n)$

Counting sort it counts the occurrences of each element and uses this info to determine the correct poition of each element. It works well when the range of values is small. $O(n)$ time complexity but requires additional memory.

Radix sort uses counting sort as subroutine, it works by grouping elements by each significant digit or char. and repeatedly sorting them until entire array is sorted. Useful when range of values is small, sorts in $O(n)$

Simple uniform hashing - any given item is equally likely to hash into any of the m slots, independently to any other item has hashed to

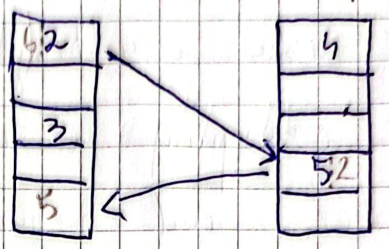
Good hash function should (approx.) satisfy the simple uniform hashing condition

Hash tables are DSs that store key-value pairs, allowing efficient search, insertion and deletion in $O(1)$. They use a hash functions to map keys to specific elem.

Universal Hashing is a technique to minimalize probability of collisions in hash tables. It involves using a family of hash functions and selecting 1 at runtime.

Open Addressing is collision resolution method used in hash tables. When an collision occures, open addressing seeks an alternative empty slot within the hash table instead of putting multiple elements in single slot.

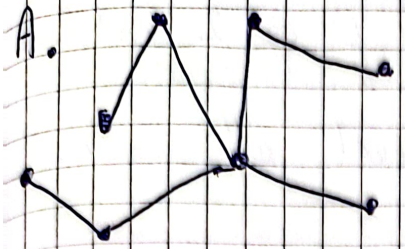
Cuckoo hashing resolves collisions in hash tables by using multiple hash tables and hash functions (1 for each table). If collision appears during insertion, the algorithm "kicks" the existing key and puts current "kicked" in another table, it repeats until no collision occures.



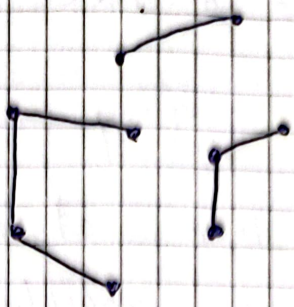
Bloom filter is a probabilistic DS used to check if an element is possibly in a set. It uses multiple hash functions and a bit array. When inserting an el the hash functions determine which positions in array will be To check if elem. is in set, hash is calculated and the same possitions in array are checked. If all are on set it means that elem. is probably in a set but if some is not set, the elem. for sure isn't in se

MinHash is a ~~function~~ technique used for estimating similarities bet sets. It involves hashing the elems of sets and comparing resulting hash values. The idea is that if 2 sets have high similarity, their hashed values will have high probability of matching.

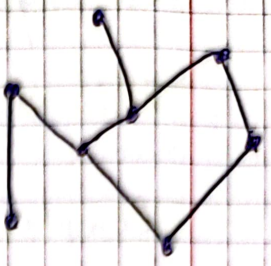
A.



B.



C.



A (free) tree [A.] - connected acyclic (without cycles) undirected graph.

If an undirected graph is acyclic but possibly disconnected, we call it a forest [B.]

[C] is neither a tree or forest because it has cycle

parent (or father), child (or son), leaf (or external node), non-leaf (or internal node), subtree rooted at node x

Vertices = Nodes

Ancestor is a node located higher up in hierarchy and is connected to particular node through series of parent-child relations.

Descendant is a node located lower down in hierarchy and is connected to particular node through series of parent-child relations

Siblings - 2 or more nodes having the same parent

Degree of x - the num. of its children

Depth of x - the length of path from root to x

Height of x - max depth in x

Binary tree - a tree of degree 2

Full binary tree - every node has 0 or 2 children

Complete binary tree - at deepest level all nodes as far left as possible

A full complete (perfect) binary tree - has 2^{h-1} internal nodes, all leaves at the same depth

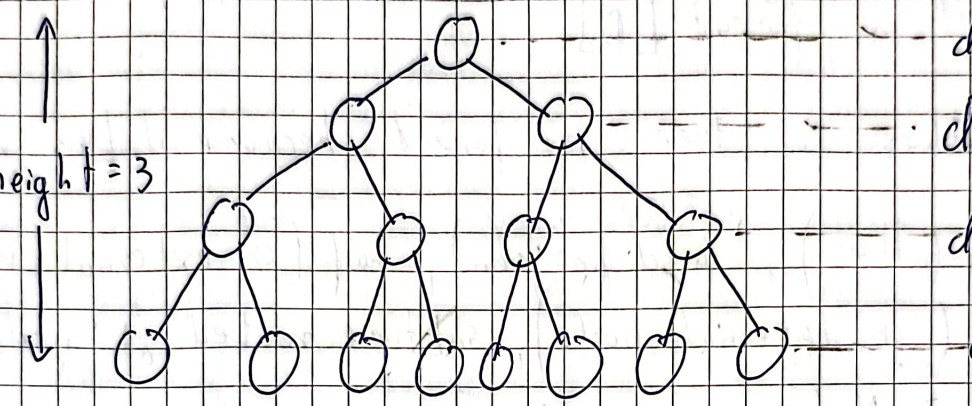


Fig. Complete binary tree, height = 3, leaves = 8, internal nodes = 7

Binary search tree (BST)

common operations: minimum, maximum, predecessor, successor, insert, delete

BST support those operations in $O(\log n)$ avg time and $O(n)$ worst case

Each node in BST has at least 4 fields:

key, right, left, and parent

In the left subtree of x everything is smaller than x and in right everything is greater than x

BST has recursive nature - every internal node is a BST too

Traversing a tree

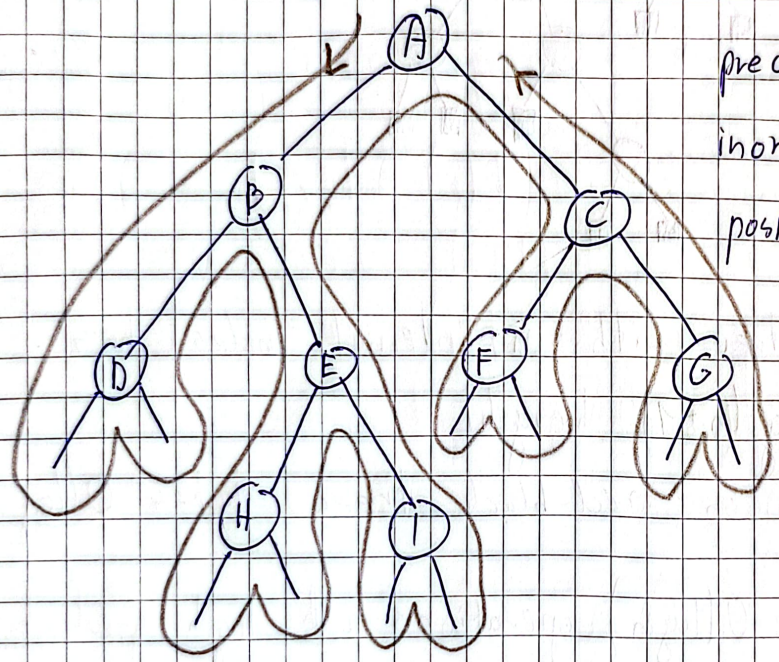
Inorder - visits nodes in order left subtree, current node, right subtree. It produces a sorted output when applied to BST. $O(n)$ time

Preorder - in order: left subtree, right subtree, ~~current node~~,
Used to create copy of tree

Postorder - in order: left subtree, right subtree, current node
Used to delete nodes

Euler tour traversal

We walk around the tree and visit each node 3 times:
- on the left we add to preorder (1st time)
- ~~on~~ from below we add to inorder (2nd time)
- on the right we add to postorder (3rd time)



preorder: A, B, D, E, H, I, C, F, G
inorder: D, B, H, E, I, A, F, C, G
postorder: D, H, I, E, F, G, C, A

Find min/max in BST - min - left as far
max - right as far
$O(h)$ $h \geq \log n$ (at least)
$O(\log n)$ best case
$O(n)$ worst case

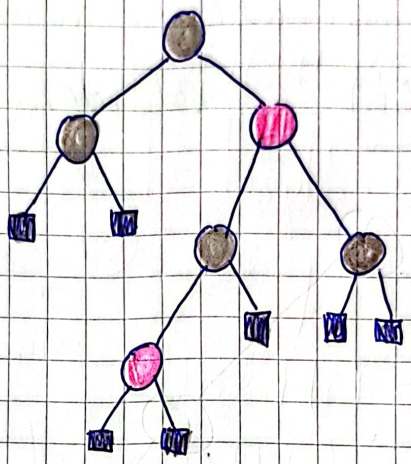
Searching key in BST $O(h)$ - worst case

Successor of x: either the min in the right subtree or min in subtree of parent (smallest key that is greater than current node)

Predecessor - node that comes immediately before our node

Red-black trees - BST, which also satisfies:

- every node is either red or black,
- every leaf (NIL) is black,
- if node is red, then both its children are black
- black-height of the tree - every path from a root to leaf contains the same num of black nodes
- root node is black



A red-black tree with n internal nodes has at most $2 \lg(n+1)$ height

All operations on red-black trees takes $O(\log n)$ in worst case

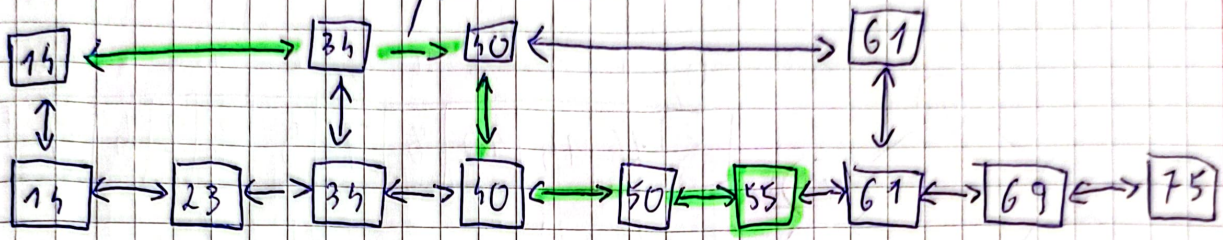
Treap - have $O(\log n)$ operations with high probability

A Treap is a binary tree where each node has a search key and priority (random)

Search keys fulfill BST property (smaller on left, larger on right)

Priority values fulfill heap property (parents $\leq$ its children or $\geq$)

Skip list - we have ie. 2 lists and each element appears in one or both lists. "Express" and "local subway lines" one of those is used to traverse whole collection fast (going from 1 express stop to another) and the other is used to find the exact position of searched key.



with 2 levels we have $2n^{\frac{1}{2}}$ steps i.e we look for 55
with k levels we have $kn^{\frac{1}{k}}$ steps
it works up to $\lg n$ levels $\Rightarrow \Theta(\log n)$

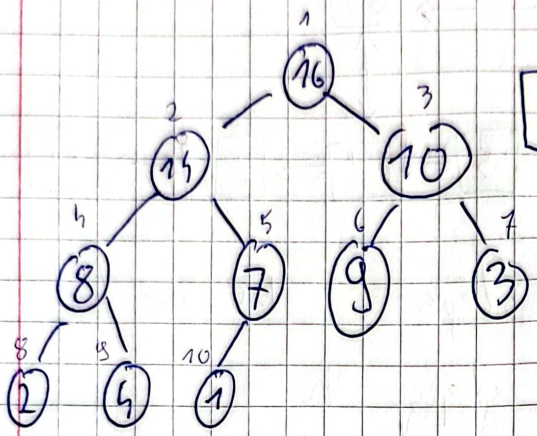
Inserting elements to skip list, we always add elem. to bottom list and for each upper layer we flip the coin. We also limit of growth of levels to up to 1 per added element

$O(\log n)$ avg search time / insertion, deletion also

Priority Queue - an ADT that stores elem. along with their priorities. It allows efficient access to elem. with highest (or smallest) priority.
Applications: maintaining k-best list, graph algorithms, event simulation, sorting, multi-way merging, Huffman coding
usually implemented with a heap.

Implicit data structures - DSs that need no extra space apart from the keys themselfs

Heap is a complete binary tree that satisfies the heap property - for every node, the value of that node is greater (or ~~equal~~ smaller) than the values of its children.



0	1	2	3	4	5	6	7	8	9	10
X	16	14	10	8	7	9	3	2	4	1

heap can be stored in an array because it fulfil those

i - array ind

$Parent(i) = \lfloor i/2 \rfloor$

$Left(i) = 2i$

$Right(i) = 2i + 1$

Heap sort - $O(n \lg n)$ in worst case

- in place
- relatively slow
- not stable

D-ary heaps - heaps of arbitrary arity.

D-ary heaps can also be represented in an plain array (no pointers). First the root, then its d children, then its d^2 grandchildren, and so on. Good practical decision is to choose d being a power of 2 because dividing by power of 2 is equivalent to bit shifting which is faster on most CPU's. D-ary heaps usually have faster insertion but slower extraction of min/max

Data compression is a set of techniques to transform data into more compact representation. The orginal data can be recovered in exact or approximate form.

require lossless compression:
- text
- programs
- databases
- medical images

allow lossy compression:
- video
- images
- sound

Criteria to compare compression algorithms:
- compression ratio,
- compression time,
- decompression time

Entropy is avg. info. in a symbol

Higher entropy = less compressible data

Redundancy of ~~data~~ code is the avg. excess (over entropy) per symbol. A redundant code is not optimal

Kraft-McMillan inequality is a rule that says the length of binary codes assigned to symbols can't be too long. It states that if we want to assign codes to symbols in a way providing efficient encoding and decoding, the sum of the probabilities of all symbols with the same code length cannot be greater than 1.

$$\sum_{i=0}^{s-1} 2^{-k_i} \leq 1$$

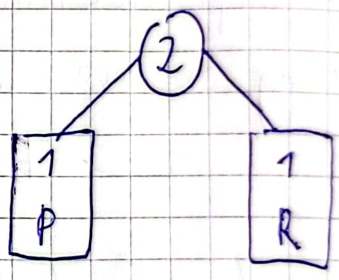
Huffman coding starts building the code tree from the bottom taking elements with the low probability (smallest occurences) making subtree from it which parent is the sum of children probabilities and then it is added again to priority q. It is done that way because elements with smallest probabilities should have longests codes to achieave the best compression.

occurences	3	3	2	1	1	1	1	1	
symbols	G	O	,	E	S	H	P	R	

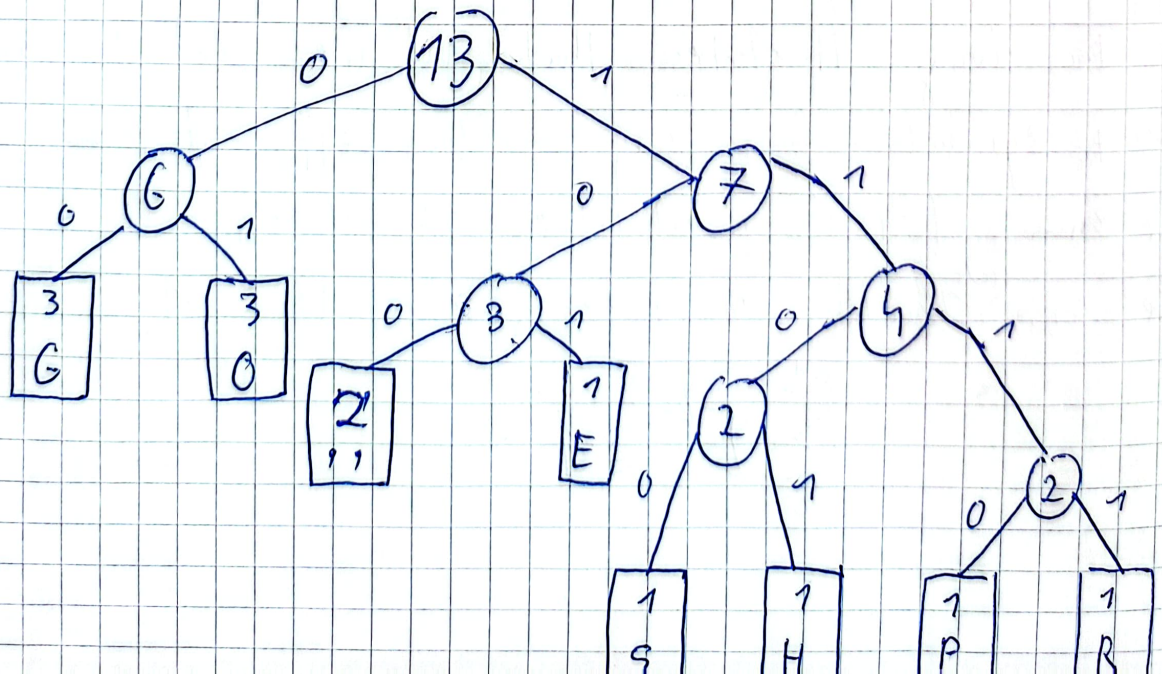
First we take 2 least occurences and merge them

Then we put it back to queue

3	3	2	2	1	1	1
G	O	,	P,R	E	S	H



We repeat this process until queue is empty



Now to encode some symbol word we add "0" each time we go left or add "1" when go right.
ie

E - 101
G - 00
H - 1101

Thanks to this more popular letters have shorter codes than less popular one.

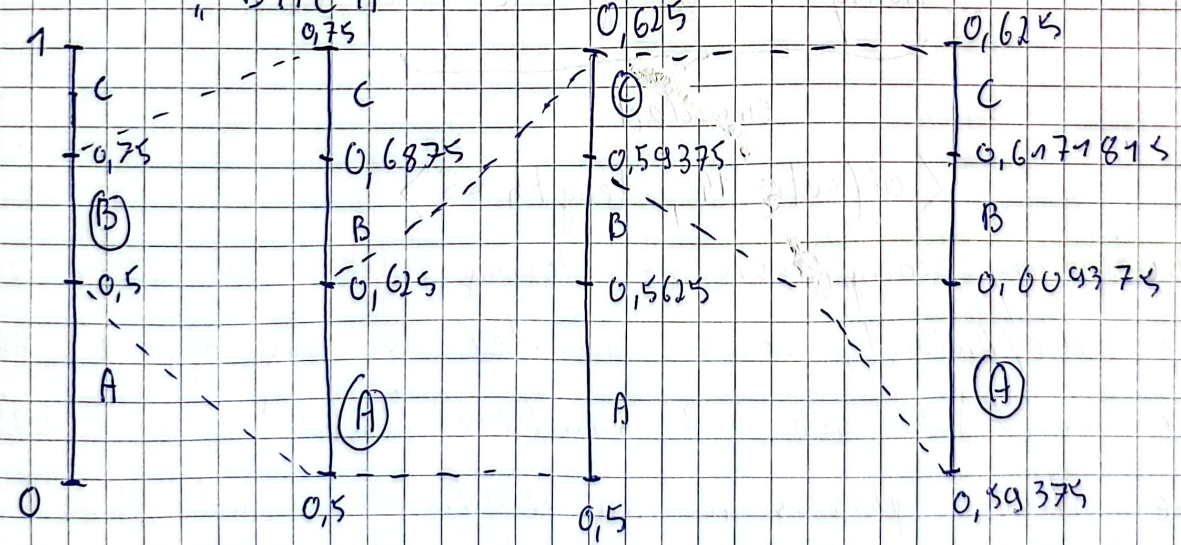
Huffman code is prefix-free which means that no code is a prefix to another code.

Canonical Huffman coding have codewords asigned to symbols in such way that can be generated in predictible manner. This allows for storing only symbol order and codeword lengths instead of entire tree structure for decoding.

Arithmetic coding

P(A) = 0,5 , P(B) = 0,25 , P(C) = 0,25

"BACA"



codeword(BACA) = 0,59375

We construct nested intervals, one per each input char.

[0,59375, 0,609375)

Instead of outputing the interval, it is enough to output any number from it

~~Run-Length compression (RLC)~~

Run-Length encoding (RLE)

version with flags replace runs of characters with triplets (flag, char, run length) i.e.

A, A, A, A, B, A, A, C, A, C, C, C, C

⇓

#, A, 4, B, #, A, 2, C, A, #, C, 4

version without flags simply put before each char how many times it appears

A, A, A, A, B, A, A, C, A, C, C, C, C

⇓

4, A, 1, B, 2, A, 1, C, 1, A, 4, C

LZ77 - some sequences apear many times in a text, so instead of coding them again and again we can just tell how long ago this sequence occured and how long it was.

... Harry Potter got a plotter ...

to encode

encoded

⟨offset = 14, length = 5⟩

Dictionary compression - storing like in LZ77 can be ineffriont, we are limited to max offset and we need to use 2 numbers (64 bits). i.e. in LZW we store phrases in an dictionary and when them appear again we just put their index from dictionary

Adv. of dict. comp. over Huffman:

- better compression
- universal code: encode „on-the-fly", in one pass
- no need to store code table in compressed file

Looking for matches in LZ77 using brute force is extreamly slow we rather use hashing.

Context based compression

In statistical coding e.g Huffman compression can be poor because we ignore a lot of usefull information.
I.e. in english u appears relatively infrequent, but its appear is high in context q order 1 (eg. quite, queue, require, count. eg. traq)

Prediction by Partial matching (PPM) is a technique that predicts the next symbol in a sequence based on the context of previously observed symbols.
The idea behind PPM is to maintain a model that contains statistical dependencies bet. symbols in a seq.
The model is build by analyzing history of occurences of symbols and their frequencies.

Detecting errors

2 main error categories:
- erroneous (flipped) bits
- lost bits

Parity bits - 1 additional bit is added per block. It is added in such a way that the total number of 1s in the block is even. It will detect if an odd num of bits got flipped in block but will not detect even num.

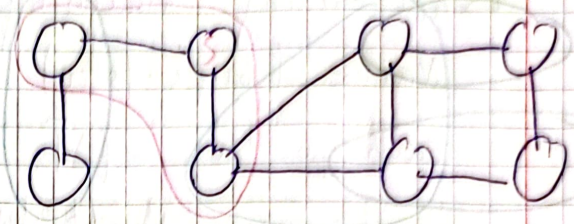
Checksums it is a hashing based technique, the hash func. is clever enough to change its value even if a(ny) single bit is modified.

Hamming codes are a type of error-correcting code used in digital communication to detect and correct errors in transmitted data. The main purpose of Hamming codes is to add parity bits to orginal data in certains possitions which allowes for detection and correction some data errors. The num of parity bits is determined by desired level of error detection and correction. They are designed to detect and correct single-bit errors and some multi-bit errors

Breadth-first search (BFS)

We visit all neighbours of a starting node and we mark it as visited (starting node). Then we visit every unvisited neighbour of neighbour, we continue doing it until we find our destination.

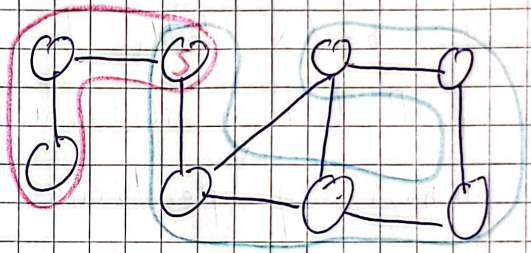
$O(V+E)$ worst time



Depth-first search (DFS)

It visits graph as „deep" as it can, if it can't go further it backtrack to last node at which it has a choice to go elsewere and go there.

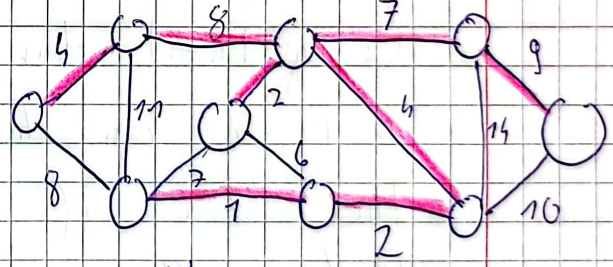
$O(V+E)$ worst time



Minimum Spanning Tree (MST)

MST connects all edges trying to minimize cost.

MST is not unique. It has $V-1$ edges.



Kruskal's algorithm

1. sort the edges by nondecreasing weight.
2. take the smallest weight unit
3. add it to a set if it doesn't create a cycle there.

   Repeat 2-3 until all nodes are connected and we obtain MST

Its time complexity is dependent on sorting which for ig. heap-sort is $O(E \log E)$. Total $O(V^2 + E \log E)$ in worst-case

Stringology basic notation

Text T, Pattern P, Alphabet Σ

$n = |T|$, $m = |P|$, $\sigma = |\Sigma|$ (alphabet size)

\$ - text terminator

Exact string matching

Naive (brute-force) tries to match P against each position of T. worst-case: $O(m \cdot n)$

Knuth-Morris-Pratt (KMP) alg. constructs prefix table also known as "longest proper prefix that is a suffix" to which determine the next possible match in case of mismatch. It first compares the P with T, char by char, when mismatch occurs, instead of shifting by 1, it uses the prefix table to determine the longest prefix of P that is also a suffix. This is used to bypass unnecessary positions. worst case $O(m+n)$

Boyer-Moore alg. - we compare P against T from right to left, if char of T aligned with the rightmost char in P does not appear anywhere in P, we can shift P by its whole length.

The idea of doing it fast comes to observation that usually $n >> m$ and $n >> \sigma$ so any preprocessing in $O(m+\sigma)$ time is almost free.

The BM preprocessing consists of σ-sized table containing info about the rightmost position of each alphabet symbol in P (0 if given symbol doesn't appear in P)

Thanks to this we can tell how far can we shift P after mismatch in $O(1)$ time.

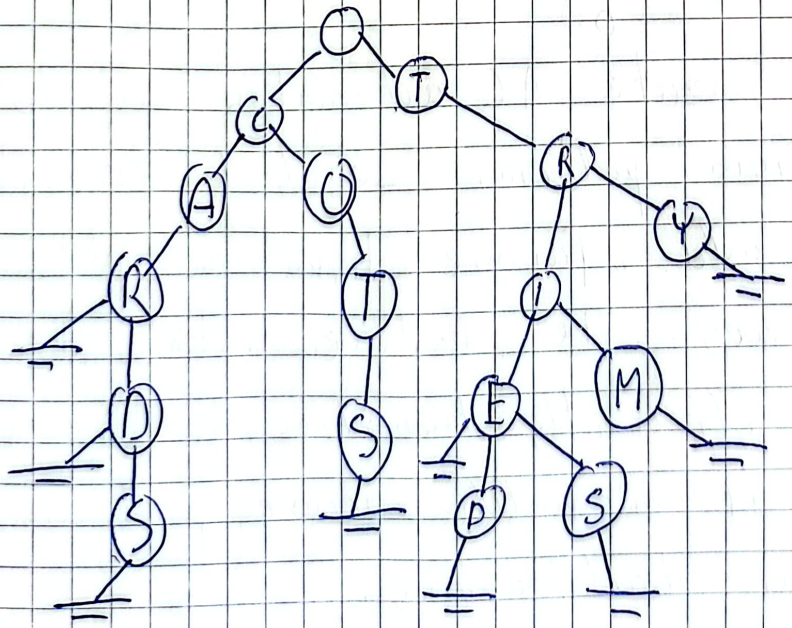
This variation is known as **Boyer-Moore-Horspool** because it only utalizes the bad character heuristic. The clasical BM alg. uses also good suffix but in many applications the cost of calculating it may surpress the skips.

avg. case $O(\frac{n}{\min(m, |\Sigma|)})$ and $O(mn)$ worst. both variants.

Multi string matching

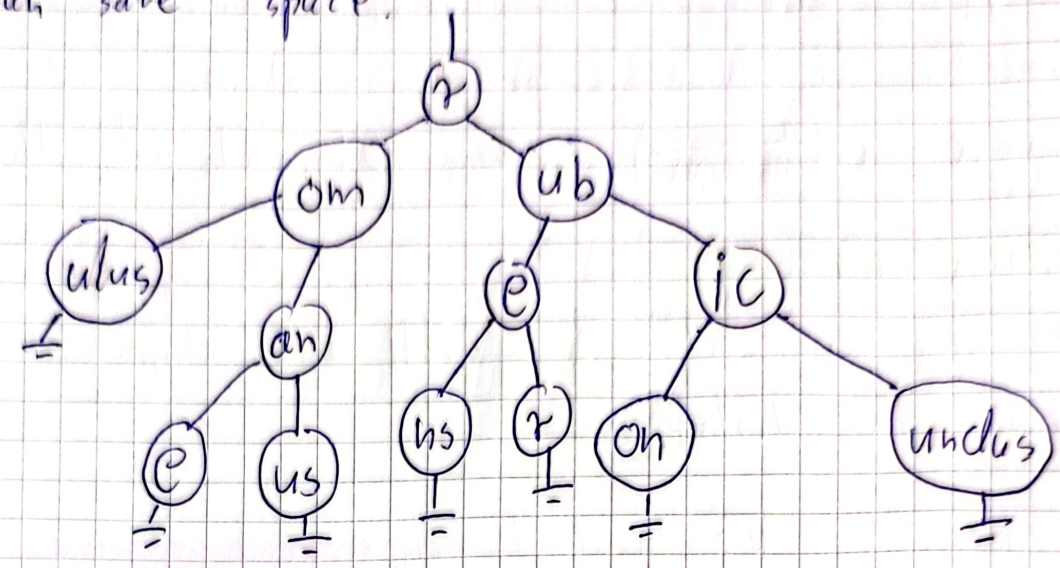
Here we can also use BMH, but skips will typically be shorter.

Trie - tree-like DS used to store and retrieve strings efficiently. Each node represent a character and edges leading from the node represent the possible characters that can follow that node. By traversing the edges, we can build a path from root to specific node, forming a string. Nodes that ends a proper string has a mark to know that is this is a proper word.

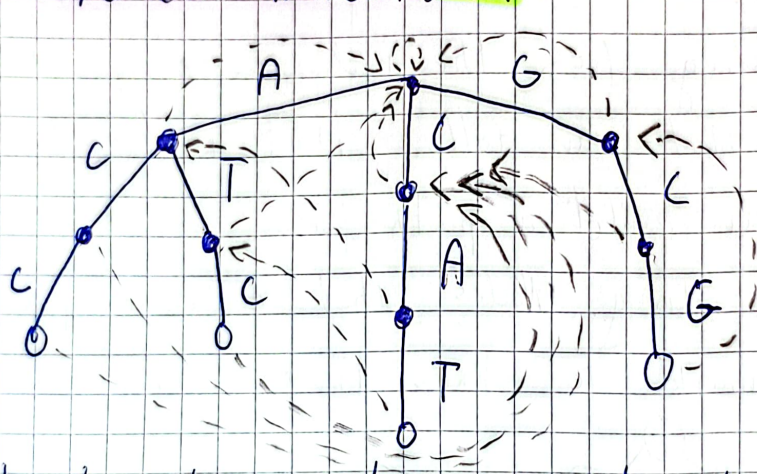


Pattern search takes either $O(m \log |\Sigma|)$ or $O(m)$ in worst-case

In most cases tries require a lot of space, one of the widely used improvements is Patricia trie. It combines every non branching nodes so we can save space.



Aho-Corasick Automaton



k
ACC
ATC
CAT
GCG
} m
↑ pattern

GCATCG - text

First step to create AC automaton is to make ~~wolfi~~ trie from patterns we look for. The next step is to add „failure links" to optimize the matching process. The „failure links" allow the automaton to efficiently transition to the next correct state if mismatch occures, eliminating the need for backtracking.

Time complexity: building the automaton $O(M)$ where M is sum of patterns lengths and searching time is $O(n+z)$ where z is total num of occurences of patterns in text.

Approximate string matching

Levenshtein distance also known as **edit distance**

it is a minimum number of elementary operations needed to convert string A to sting B (or vice versa)

Allowed operations:

- insert a single char,
- remove a single char,
- substitute a char.

Eg. edit(pile, spine) = 2 (insert s; relpace l with n)

		S	Z	Y	M	O	N
	0	1	2	3	4	5	6
A	1	1	2	3	4	5	6
D	2	2	2	3	4	5	6
A	3	3	3	3	4	5	6
M	4	4	4	4	3	4	(5)

← how many to transform '' into a prefix

← my answer

replace	insert
delete	

If both chars in row and col. are the same we take min of those 3 and if are diff. we take min and +1

Hamming distance is the number of positions of which S_1 and S_2 differ. lengths of S_1 and S_2 must be the same.

Longest Common Subsequence (LCS) is a problem that aims to find longest subsequence common to 2 or more input sequences. Subsequence is a sequence obtained by removing 0 or more elem. from a given sequence while maintaining the relative order of remaining elements. It means that the subseq. don't have to be contiguous.

		I	T	I	G	E	R
	0	0	0	0	0	0	0
I	0	1	1	1	1	1	1
R	0	1	1	1	1	1	2
I	0	1	2	2	2	2	2
G	0	1	2	3	3	3	3
G	0	1	2	3	3	3	3
E	0	1	2	3	4	4	4
R	0	1	2	3	4	5	5

1st row and column we match sim. of 1 text to empty string

← ANSWER

z	y
x	z

If letters in col. and row match we take z + 1 if don't we take max from x and y

Text indexing - if we expect many searches to be run over a text it is worth to sacrifice space and preprocessing time to build index over text.

Full-text index: match to any position in T is available to it.

Word based index: find only occurrences of P in T only at word boundaries.

Suffix tree is basically a Patricia trie containing all n suffixes of T. Space: $O(n \log n)$

Construction time: $O(n \log \Sigma)$

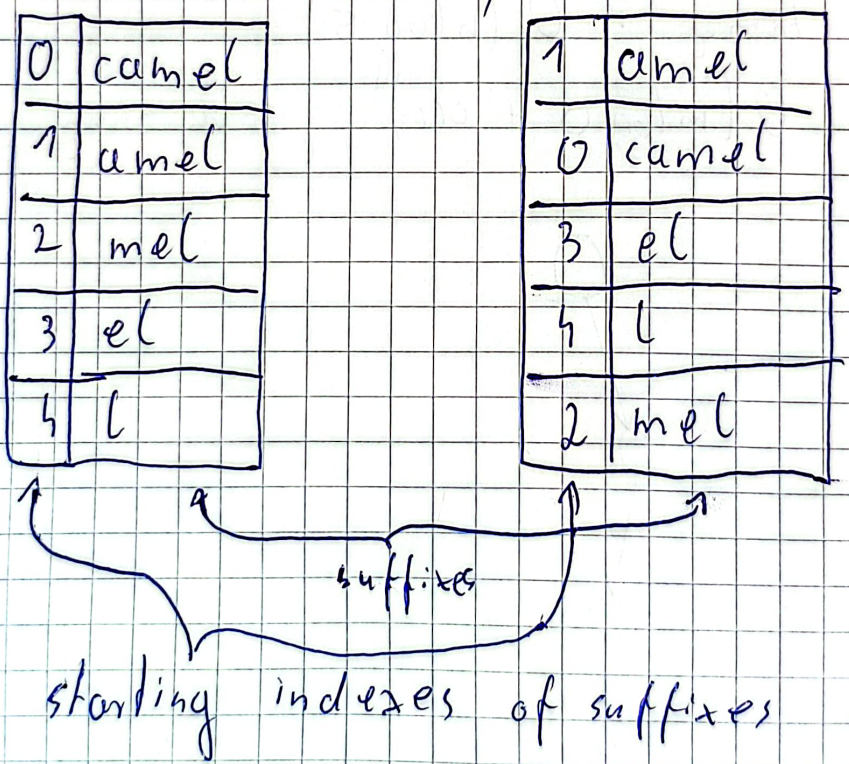
Search time $O(m \log \Sigma + occ)$ occ - num of occurrences of P in T

Suffix array

Sort all the suffixes of T, store their indexes in an array, keep T as well

(total space: $4n + 1n = 5n$ bytes which is much less than with suffix tree)

To search for a P in text, compare P against median suffix ~~in array~~ index, then refer to the orginal (T). If not found go left or right depending on result, each time halving the range of suffixes. It is binary search based.



storing indexes of suffixes in the text

We don't have to store entire suffixes in an array, because having the ~~index~~ its starting index in a text we can simply refer do it.