Algorithms and Data Structures 2

TIN061

Jan Hric

Lectures, part 1, v. 1.12.2019/f
Syllabus ADS2

- A string search: alg. Aho-Corasick, …
- Flow networks
- Fast (discrete) Fourier transform
- Gate networks, sorting networks, ...
- Problem classes P, NP, NPC, reducibility
- Approximation algorithms
- Cryptographic protocols
- Probabilistic algorithms, primality testing
- Algorithms in plane, convex hull
- (Dynamic programming)
  ~ Algorithms in a wider sense
A string search: Aho-Corasick alg.

• A search of multiple patterns in a text
• An alphabet $\Sigma$, finite words $\Sigma^*$, length, concatenation, empty word $\varepsilon$ (or $\lambda$)
• A problem: Given an alphabet $\Sigma$, a word $x = x_1x_2...x_n$, searched patterns $K = \{y_1, ..., y_k\}$
• Output: All instances of patterns from $K$ in $x$, i.e. $[i, p]$, $y_p$ is a suffix of $x_1...x_i$ (tricky: only pointers)
• Parameter $l = |K| = \sum_{i=1}^{k} \text{length}(y_i)$
Naive alg.

- For all patterns $p$, for all valid positions $i$:
  
  match a pattern $p$ from the beginning with a text at a position $i$

  if the whole pattern successfully matches, then Report($i$, $p$)

- Complexity: in the worst case $O(l.n)$
  
  - Without counting of an output writing
    
    - It is the same for all (correct) algorithms
    - It depends on input data
An idea of AC alg.

• We construct an algorithm dependent on patterns (∼ Finite Deterministic Automaton) in time $O(l)$, which finds patterns in a text in $O(n)$.

• Alg. 1 – an interpret of a searching machine

• Alg. 2 – a compilation of patterns, a creation of a forward function

• Alg. 3 – a compilation of patterns, a creation of a backward func.
Wider context

• A compiler, a generation of a machine and a code

• DSL: Domain Specific Languages

• Different views on a search machine (interpretation)
  - An abstract machine: data structure or bytecode
  - Source code or executable code
    • Use of a runtime library for specific operations
Search AC machine

- The machine (over $\Sigma$) is a tuple $(Q, g, f, \text{out})$
  - $Q = \{0..q\}$ is a set of states
  - $g: Q \times \Sigma \rightarrow Q \cup \{\perp\}$ ; a (forward) goto function
    - $g(0,c) \in Q$, a step from state 0 is defined for all letters
  - $f: Q \rightarrow Q$ ; a backward fail function
    - $f(0) = 0$
    - $f$ is used, when $g$ returns $\perp$
  - $\text{out}: Q \rightarrow P(K)$ ; an output function
    - As multiple patterns can be finished on the same place, we must return a subset of patterns
Properties of a search: \( g \)

- A graph of the function \( g \), excluding a loop in 0, is a tree
  - State 0 is the root of the tree
  - Each path from the root is valuated by some prefix of a pattern
  - Each prefix of each pattern describes a path from the root to a (single) state \( s \); a prefix \( u \) represents a state \( s \). Particularly, the word \( \varepsilon \) represents the state 0
  - Each step using \( g \) goes one level deeper in the tree.
Properties of a search: f and out

• The backward function f:
  – For each state s represented by a word u, the value f(s) is represented by the longest proper suffix of u, which is also a prefix of a pattern from K
    • f(s) is defined for all states, because an empty suffix \(\varepsilon\) is a possible value
  
• The output function out
  – If u represents s and \(y \in K\), then \(y \in \text{out}(s)\) whenever y is a suffix of u.
AC: alg. 1

- Input: $x = x_1 x_2 ... x_n \in \Sigma^*$, $M = (Q,g,f,out)$
- Output: pairs $(i,y) \ldots$ (a position $i$, a pattern $y$)

1. $\text{state} := 0$
2. $\text{for } i := 1 \text{ to } n \text{ do }$ ; through letters
   3. \hspace{1em} $\text{while } g(\text{state},x[i]) = \bot \text{ do}$
   4. \hspace{2em} $\text{state} := f(\text{state})$
   5. \hspace{2em} $\text{state} := g(\text{state},x[i])$
   6. \hspace{2em} $\text{forall } y \in \text{out}(\text{state}) \text{ do}$
   7. \hspace{3em} Report( (i,y) ).
Notes on searching

- A pattern is reported on a final position
- A pattern can be a suffix of another pattern → the function `out` reports a set of patterns
- Patterns are reported only after g-step (line 6)
- Conditions on f and g are „boundary conditions“
- The function g creates a data structure for search: TRIE

- Ex: SLICE, SLICES, ICE, SCENE
Correctness

• Invariant (declaratively): The algorithm visits states each representing the longest suffix of a processed part of the text, which is also a prefix from K.
  – Proof: using property of f

• The algorithm returns all patterns found
  – Proof: using property of out
Complexity of interpretation

• A hard part: the number of f-steps (lines 3,4)
  – A separate count gives too loose approx. O(n.l)
  – → we must count f-steps globally (to reach O(n))

• A potential method:
  – A depth of a current state is a potential. A g-step increases a potential, an f-step decreases a potential.
  – We want to show that globally a count of f-steps is O(n).

• Note: this is an example of an amortized complexity. (It counts complexity of sequences of ops.)
Complexity 2

- **Th:** The count of f-steps is less than n.

- **Pr:** $n = \text{a count of } g\text{-steps } \{g \text{ is increased by at most 1}\}$
  
  $\geq \text{a cumulative increase of potential } \{\text{alg. starts at 0}\}$

  $= \text{cumulative decrease of potential + final depth}$

  $\geq \text{cumulative decrease of potential}$

  $\{f \text{ is decreased by at least 1}\}$

  $\geq \text{a cumulative count of } f\text{-steps}$

- Therefore globally a complexity of the search is $O(n)$
Algorithm 2 for \( g \) and \( o \)

In: patterns \( K \); Out: states \( Q, g, o: Q \rightarrow P(K) \)

1. procedure Enter(\( c[1]..c[m] \)) ; adds the pattern \( y[p] \)
2. \( \text{state}:=0; \ j:=1 \)
3. while \( j<m \) and \( g(\text{state}, \ c[j]) \neq \bot \) do
4. \( \text{state} := g(\text{state}, \ c[j]) \) ; repeated chars
5. \( j++ \)
6. for \( p:= j \) to \( m \) do ; new branch
7. \( q++; \ Q := Q \cup \{q\} \) ; new state
8. forall \( x \) in \( \Sigma \) do \( g(q, x) := \bot \) ; undef. implicitly
9. \( g(\text{state}, c[p]) := q \) ; adding a character
10. \( \text{state} := q \) ; shift to a new state
11. \( o(\text{state}) = y[p] \) ; a preliminary output
Alg 2, main

12 \( Q := \{0\}; \ q := 0 \) ; init of states count
13 forall \( x \) in \( \Sigma \) do ; for all letters
14 \( g(0,x) := \_ \) ;
15 for \( i := 1 \) to \( k \) do ; through all patterns
16 \( \text{Enter}(y[k]) \) ; add pattern to a trie
17 forall \( x \) in \( \Sigma \) do
18 if \( g(0,x) = \_ \) then \( g(0,x) := 0 \) ; a boundary cond.
Alg. 3 for \( f \) and \( \text{out} \)

In: \( Q = \{0..q\}, \ g: Q \times \Sigma \rightarrow Q \cup \{\bot\} \), \( o: Q \rightarrow \mathcal{P}(K) \)

Out: \( f: Q \rightarrow Q \), \( \text{out}: Q \rightarrow \mathcal{P}(K) \)

- Using queue for unprocessed states

01 \( \text{queue} := \text{empty} \) \quad ; \text{init}

02 \( f(0) := 0; \ \text{out}(0) := \emptyset; \)

03 \textit{forall} \ x \ \text{in} \ \Sigma \ \text{do}

04 \textit{if} \ (s := g(0,x)) \neq \bot \ \text{then} \quad ; \text{nodes below root}

05 \quad f(s) := 0; \ \text{out}(s) := o(s) \quad ; \text{trivial init}

06 \text{queue} := \text{queue} \cup \{s\} \quad ; \text{a new state to the end}
Alg. 3 cont'd

07 while queue is not empty do
08   r:= take the first element of queue (and delete)
09   forall x in Σ do
10     if g(r,x) ≠ then ;process descendants of r
11     s:=g(r,x); t:=f(r)
12     while g(t,x) ≠ do t:=f(t) ;through suffixes
13     f(s) := g(t,x) ; a valid node (~prefix) found
14     out(s):=o(s)∪ out(f(s)) ; out() from suffixes
15     insert s to queue.
Alg 3: comments

- We must use a queue in the alg. 3
  - We may need an arbitrary f(t) for a lower depth state
- The line 12 stops because g(0,.) is defined
- A value of f(s) can be the state 0, as $\varepsilon$ is a valid prefix of any pattern.
Properties: Correctness

• The output of alg. 3 is a correct AC search machine
  – Used $f$ is defined
    • Due to a queue and a lower depth
  – $f$ points to the longest possible suffix
  – $out$ includes shorter patterns
    • A patterns $p$ can be embedded in a longer pattern $r$, so a machine can visit only states of $r$, but must report also $p$. 
Complexity

• It is nontrivial to count f-steps on line 12
  – We can have $O(l)$ patterns with max. length $O(l)$ giving naively $O(l^2)$.
    • (Practically, a correctly implemented machine is quick also without a proof – $O(l)$, but ...)
  
• For each pattern $p$, a cumulative count of f-steps on prefixes of $p$ is bounded by the length of $p$.

• So globally we have $O(l)$ f-steps. If $|\Sigma|$ is not taken as a constant, then $O(l \cdot |\Sigma|)$ steps.
Implementation

• We can use a sparse (or implicit) representation of $\bot$ and $g(0,.) = 0$: values are not in memory and need not be initialised.
  
  – A sparse representation needs $O(l)$ cells
    
    • It does not have $O(1)$ access, but $O(\log |\Sigma|)$.
  
  – A dense representation (e.g. using arrays) needs $O(l \cdot |\Sigma|)$ cells. It is a standard representation for a finite automaton (from another lecture Automata and grammars).
Alg. Knuth -Morris-Pratt

• An alg. for a search of 1 pattern.
• In our context, it is a simplified AC alg.
• A graph of $g$ is not a tree but a string. So a state corresponds to a count of characters being read (including 0) and we can use $g$ implicitly.
• An asymptotic complexity is $\Theta(n+l)$ instead of $\Theta(n+l \cdot |\Sigma|)$
• We use the prefix function $\pi$ instead of $f$: $\pi(s)$ is the length of the longest proper suffix of the state represented by $s$, which is also a prefix of the pattern.
Alg. Rabin-Karp

• Idea: take a pattern of a length \( l \) as \( l \)-digit number with a base \( a = |\Sigma| \)

• We compute a signature of a pattern as well as a signature of a section from the text of the same length (called a window) modulo a (prime) number \( q \).
  – It is a hash function, but not for a table search

• If a signature \( v \) of \( p \) doesn't match a signature at a position \( i \), then \( p \) is definitely not at a pos. \( i \).
  – The signature at a pos. \( i \) is denoted by \( t_i \)
Implementation

• We compute $v$ and $t_1$ using Horner schema
  
  $$v = ((...(\tau_1 \cdot a + \tau_2) \cdot a + ... ) \cdot a + \tau_{l-1}) \cdot a + \tau_l$$

• A time complexity $O(l)$, where $l$ is the length of $p$

• A shift of the window
  
  $$t_{i+1} = a \cdot (t_i - a^{l-1} \cdot \sigma_i) + \sigma_{i+l}$$

• $\sigma_i$ is the first deleted digit and $\sigma_{i+l}$ is a newly appended digit.

• ! if we use exact numbers (without modulo), then their length is $O(l)$ bits. :-(
Implementation 2

- A choice of \( q \): such a prime number that \( a \cdot q \) can be computed in a register
  \[ t_{i+1} = (a \cdot (t_i - h \cdot \sigma_i) + \sigma_{i+1}) \mod q \]
  \( \rightarrow \) arithmetic operations in time \( O(1) \) instead of \( O(l) \)

- We used \( h = a^{l-1} \mod q \) precomputed in \( O(l) \)

- But: Equality of signatures modulo \( q \) causes a false hit when a pattern \( p \) doesn't equal a relevant text window
Time complexity

- The worst case: $\Theta((n-l+1) \cdot l)$
- Expected complexity:
  
  $O(n) + O(l \cdot OK) + O(l \cdot F)$

  - OK is a count of found positions (we must verify it)
  - F is a count of false hits: (supposing a uniform distribution of $t_i$) $F = O(n/q)$
    
    $\rightarrow O(n) + O(l \cdot (1+ n/q))$

- HW: more patterns of the same length, of a different length
Flow networks

- A flow network is $S = (G, c, s, t)$ where
  - $G=(V,E)$ is a directed graph (if $(u,v) \in E \rightarrow (v,u) \in E$)
  - $c : E \rightarrow R_0^+$ represents a capacity of edges
  - $s \in V$: the *source* vertex
  - $t \in V, s \neq t$: the *sink* vertex ($t$ as a „target“)

- Notation: $|V|=n$, $|E|=m$ ; $c(h)=c(u,v)$ for $h=(u,v)$...

- Without loss of generality
  1. Single source and single sink
  2. Capacity only for edges, not for vertices

HW: using transformation/reduction (and the same sw)
Flow

• A flow $f$ in the network $S = (G, c, s, t)$ is a function $f: V \times V \rightarrow \mathbb{R}$, such that

1. Symmetry: $f(u, v) = -f(v, u)$ for all $u, v$
2. Capacity: $f(u, v) \leq c(u, v)$ for all $u, v$
3. Flow conservation: $d(f, u) = 0$ for $\forall u \in V \setminus \{s, t\}$
   where $d(f, u) = \sum_{v \in V} f(u, v)$ (a divergence of $f$ in $u$)

• $Df$: An edge $e$ is saturated iff $c(e) = f(e)$

• $Df$: A flow size of $f$ is $d(f, s)$ for a source $s$; a notation $|f|$
Maximum flow problem

- A problem: To find a flow of a maximal size in a given network, i.e. a maximal flow $f^*$.
  - We denote $f^*$, it is not unique, but its size is unique
- Df: a cut in a graph is a disjunctive pair of sets, s.t. $X \cup Y = V$, $s \in X$, $t \in Y$
- Df: A capacity of a cut: $c(X, Y) = \sum_{u \in X, v \in Y} c(u, v)$
- Df. A flow over a cut: $f(X, Y) = \sum_{u \in X, v \in Y} f(u, v)$
- Df: A minimal cut is a cut with a minimal capacity
Flows and cuts

• Lemma 1: It is valid for each flow $f$ and each cut $(X,Y)$, that a flow over a cut $(X,Y)$ is equal to $|f|$
  – Proof: By induction over $|X|$ with a base $X=\{s\}$

• Corollary: As $f(X,Y) \leq c(X,Y)$ for each cut $(X,Y)$, the size of a max. flow is at most the capacity of a minimal cut. → We show equality.

• Definition: A residual capacity of $f$ is a function $r: V \times V \to \mathbb{R}$ defined $r(u,v) = c(u,v) - f(u,v)$
Residual net

- Df: A residual net $R$ for a net $S$ and a flow $f$ is $R = (G', r, s, t)$, where $(u, v)$ is in $G'$ whenever $r(u, v) > 0$.
  - The value $r(u, v)$ is an edge capacity in a residual graph
  - (We want only potentially usable edges in the residual graph)

- Df: An augmenting path $P$ is a path from $s$ to $t$ in $R$.

- Df: A residual capacity of $P$ is $r(P) = \min\{r(u, v), (u, v) \in P\}$
  - A size of a flow can be increased by $r(P)$ on edges of the augmenting (improving) path $P$
Max-flow min-cut theorem

- The following conditions are equivalent:
  1. A flow $f$ is maximal
  2. There is no augmenting path for $f$
  3. $|f|=c(X,Y)$ for some cut $(X,Y)$

- Pr: $1 \rightarrow 2$: by contradiction: If $f$ is maximal, but an augmenting path $P$ exists, then $|f|$ increases after improving. A contradiction.
2 → 3: We suppose that no augmenting path exists in G from s to t. Define $X = \{v | \text{an augmenting path from s to v exists}\}$ and $Y = V \setminus X$. A division $(X, Y)$ is a cut because s and t are in different parts (by construction)

Each edge from X to Y is saturated, otherwise we can extend X. Using lemma 1:

$|f| = f(X, Y) = c(X, Y)$, second eq. from saturation
Cont. 3

3 → 1: We have $|f| \leq c(X,Y)$ for all cuts $(X,Y)$ by corollary.

So the condition $|f| = c(X,Y)$ implies that $|f|$ is maximal
Ford-Fulkerson method

- Also known as: an augmenting path method
  - It is a generic algorithm with a strategy for a path finding (line 2)

1. Initialize a flow $f$ to 0
2. while an augmenting path exists do ; found by a strat.
3. improve $f$ on edges of $P$ by $r(P)$
4. return $f$
Properties

1. We can construct a minimal cut in $O(m)$ based on a maximal flow. (using Theorem, more cuts)

2. If capacities are irrational numbers, then implementation can diverge. The size of a flow converges but possibly to a suboptimal flow
   - Informally: a strategy is not fair: a path is not selected

3. Rational capacities can be transformed to integer capacities

4. Each augmenting path improves a flow at least by 1 for integer capacities. So $|f^*|$ steps are enough. The $f^*$ has integer values on edges.
Properties 2

5. The F-F alg. is generic. An augmenting path can be found by any algorithm for a graph search.
   - It is an advantage for a proof of the correctness and a disadvantage for proving a complexity bound.

• Ex: a graph with long computation

• Th: The constructed function $f$ is a flow.
   - Pr: by induction on cycle iterations. A zero flow is a flow. Changing a flow along the whole path does not change a flow conservation except $s, t$

The new flow is allowed using $r(P)$
Properties 3

- A time complexity of the F-F alg. with integer capacities is $O(|f^*| \cdot m)$ → Alg. finishes
  - Note: Time is not polynomial wrt. a binary size of an input

- Partial correctness: If the F-F alg. finishes, it has not found an augmenting path and so the found flow is a maximal flow, by Theorem.

- Best complexity: A max. flow can be constructed by $m$ augmenting paths. (HW)
Strategies for path choosing

- (A maximal augmenting path)
  - A variant of the Dijkstra alg. for a minimal path finding
- A shortest augmenting path
  - Based on a breadth-first search
  - $O(n \cdot m^2)$ globally: $n$ phases, $m$ edges in a phase to be saturated, $O(n+m)$ for finding an augmenting path
- An improvement: All shortest paths in „a batch“
- HW: to find a time complexity bound for a graph with capacities 1
Dinic alg: A level graph

• Idea: Based on a level graph and a blocking flow

• Df: A level graph has a finite number of levels and directed edges are only between adjacent levels.

A level of a vertex is the length of a shortest path from s. The first level is \{s\} and the last one is \{t\}

  – A level graph is usually *pruned*: each edge and vertex are on some shortest path (this simplifies a complexity analysis)
  – Let \(d(u,v)\) denote the shortest path from \(u\) to \(v\). It is true that \(d(s,v) + d(v,t) = d(s,t)\)

• Df: A blocking flow has a saturated edge on each shortest path

  – There can be augmenting paths but they must be longer.
Blocking flow

- We look (only) for a blocking flow in a level graph
  - Longer augmenting paths are processed and saturated in next iterations with new level graphs
Dinic alg.

- **In**: A network $G = ((V,E), c, s, t)$
- **Out**: a maximal flow $f$ from $s$ to $t$

1. Initialize $f(e) = 0$ for all edges
2. Construct level graph $G_l$ of a residual graph
3. If $dist(t) = \infty$ then stop and output $f$
4. Find a blocking flow $f'$ in $G_l$
5. Improve $f$ by $f'$ and continue at 2
Properties of Dinic Alg.

- L: A distance $d(s,t)$ increases during alg.
  - Idea: New paths have some new edge in an opposite direction
  
  => We have $n$ phases, so complexity is $O(n \cdot h(n,m))$, where $h(n,m)$ is a time necessary to find a blocking flow.

- We have a pruned network: we can use any edge (greedy) for prolongation of any partial aug. path
  - As backtracking is not needed, we have $O(n)$ for a single path, so a phase takes $O(n \cdot m)$
  - Globally: $O(n^2 \cdot m)$
Implementation 1

1. A creation of a level graph
   - To get $d(s,x)$ and $d(x,t)$ for all vertices $x$, in $O(n+m)$
   - A vertex $u$ stays in $R$: $d(s,u)+d(u,t)=d(s,t)$
   - An edge $(u,v)$ stays in $R$: $d(s,u)+1+d(v,t)=d(s,t)$
     - An invariant of a pruned net: each vertex and edge are on some minimal path $\Rightarrow$ any edge can be used for a path

2. Pruning (after augmenting vs. during a search)
   1. A net is pruned after each augmenting $\Rightarrow$ invariant
   2. Backtracking: unsuccessful vertices and edges are deleted once (from a level graph per phase)
Implementation 2

- A network pruning: we need a good implementation. We need only constant time per an edge and a vertex

- A possible technique: a cascade pruning
  - Store a count of in- and out-degrees of vertices
  - Decrement counts for all saturated edges. If any count is zero, propagate through vertices and edges

- Note: A selection of a vertex with a minimal inflow and a change propagation from it by levels gives $O(n^3)$ globally
Goldberg alg., a preflow-push alg.

• Idea of an alg.: it uses a preflow and a height $f$.

• Df: A preflow is a function: $V \times V \rightarrow R$, that fulfills conditions of a capacity and a symmetry, but it is allowed an excess: $V \rightarrow R$ for all vertices except a source $s$.

• $\text{excess}(v) \geq 0$, $\text{excess}(v) = \sum_{w \in V} f(w, v)$

• A vertex (except $s$ and $t$) is active, if $\text{excess}(v) > 0$
Height function

- Df: Let \( f \) be a preflow and \( R \) is a residual graph for \( f \). A function \( h: V \rightarrow \mathbb{N} \) is a height function if:
  1. \( h(s) = |V| \)
  2. \( h(t) = 0 \)
  3. \( \forall (u, v) \in E_R: h(u) \leq h(v) + 1 \)

- An edge \((u,v)\) is available if equality holds in 3.

- Idea: we construct a preflow, not a flow along a whole augmenting path. We shift an excess along an unsaturated edge – if an edge goes „down“ and a height difference is exactly 1.
Goldberg alg. - generic;

01 \( h(s) = n \); \( h(v) = 0 \) for other vertices except \( s \)
02 \( f(s,v) = c(s,v) \) for all edges \((s,v)\) //\( f \) from \( s \) is satur.
03 \( f(e) = 0 \) for other edges
04 while an vertex \( v \neq s \) with positive excess exists do
05 \( \text{if ex. } e = (v,w) \text{ with positive reserve and } h(v) > h(w) \text{ then} \)
06 \( \text{choose } (v,w) \text{ as an edge from } v \)
07 \( d = \min(\text{excess}(v), r(v,w)) \)
08 \( \text{we shift an excess of size } d \text{ from } v \text{ to } w \)
09 \( \text{else } h(v) := h(v) + 1 \) // increasing a height of \( v \)
10 end

- A choice of a vertex \( v \) (line 4) and an edge \( e \) (l. 5,6) is given by a strategy
Steps of an alg.

- A main loop execution (dependent on an order):
  1. A saturated shift of an excess ($d=r(v,w)$)
      → an edge changes to saturated
  2. An unsaturated shift of an excess ($d<r(v,w)$)
      → an excess of $v$ changes to zero
  3. Increasing a height of $v$ (line 9)

- Note: A vertex can get higher than a source height $n = h(s)$, so it can return an excess to the source
Partial correctness 1

• L1: After an initialisation, there is no edge \((v,w)\) s.t. \(h(v) > h(w)+1\) and its edge reserve is positive

• Pr: A condition holds after an initialisation, as all edges with a height difference start in a source and all edges from source have zero reserve

• A main loop does not create such edge, because:
  - Increase of \(v\): If \(v\) has an excess (by choice in an alg.) and an edge has a positive reserve (a precondition), then \(v\) is not increased (a contradiction), but an excess is shifted. So edges have zero reserve.
  - A shift of an excess along an opposite edge \((w,v)\) means \(h(w) > h(v)\)
Partial correctness 2

• Th: (a partial correctness) If Goldberg alg. finishes, then it has found a maximal flow.

• Pr: if a while cycle finishes, then all vertices except s have zero excess and a preflow is a flow as well.

• It remains to prove: a found flow f is maximal ← there is no augmenting path ← each path (from s to t) has a saturated edge

• Any path from s to t starts at height n=h(s), ends at 0=h(t) and it has n-1 edges. So an edge with a height difference 2 exists. We proved in Lemma 1 that this edge has zero reserve.
Time complexity: idea

• We give upper bounds on 3 operations:
  1. The maximal height of a vertex → number of a height increasing
  2. Number of saturated shifts
  3. Number of unsaturated shifts

• It is a generic algorithm and a generic proof (of worst-case complexity). A particular strategy can have a better time complexity.
Height count - preparation

- **L2:** If vertex $v$ has a positive excess after an initialisation, then there exists a directed path from $v$ to $s$, such that all edges on a path have a positive reserve.

- **Pr:** Let $v$ to have a positive excess. Let $A$ be a set of vertices, which have a directed path from $v$ consisting of edges with a positive reserve.

  $\rightarrow$ an inflow to $A$ is zero $\rightarrow$ an excess of $A$ is nonpositive $\rightarrow$ as $s$ is the only vertex with a nonpositive excess, it belongs to $A$. QED
Height count

• L3: The height of any vertex is bounded by $2n$.

• Pr: Suppose we want to lift a vertex over $2n$. Then it is in a height $2n$ and has a positive excess. Using Lemma 2, we have a path from unsaturated edges from $v$ to $s$. Similarly as before: a path starts in a height $2n$, it finishes in a height $n$, and it has at most $n-1$ edges. So some edge has a height difference at least 2 and it has no reserve. A contradiction.

• L4: A count of lifts globally in the alg. is $O(n^2)$
Saturated shifts

• L5: Number of saturated shifts is globally n.m
• Pr: Let e=(u,v) be an edge. Sum of h(u) and h(v) is between 0 and 4n. A reserve of (u,v) is 0 and h(u) = h(v)+1 after a saturated shift.

• A reserve must increase before next saturated shift on the same edge. It is possible only if a shift along an opposite edge (v,u) occurs. So h(v) increases by at least 2 (a shift along an opposite edge), then h(u) increases by at least 2. A sum h(u) + h(v) increases by at least 4 between any saturated shifts.

→ A count of saturated shifts per an edge is at most n and globally n.m, so we have O(n.m)
Unsaturated shifts

- **L6**: Number of unsaturated shifts is globally at most $2n^2 + 2n^2 \cdot m$

- **Pr**: (using a potential method):

- Let $S$ be a sum of vertex heights with a positive excess, except $s$ and $t$.

- **Boundary conditions for $S$**:
  - After initialization: $S=0$, as only $s$ has a nonzero height
  - At the end: $S=0$, as no internal vertex has an excess
Unsaturated shift, cont'd

- Operations:
- A lift of a vertex increases $S$ by 1.
- A saturated shift along $(u,v)$ increases $S$ by at most $h(v) \leq 2n$, if $v$ did not have an excess and $u$ remains with an excess.
  - A cumulative increase of $S$: $2n^2 + 2n \cdot nm$ (A)
- An unsaturated shift along $(u,v)$ decreases $S$ by at least 1. Heights are the same, a summand $h(u)$ disappears and $h(v)$ is possibly added, if it was not present before. As $h(u) = h(v) + 1$, the value $S$ decreases. Globally, (A) gives a bound for a step count.
Time complexity

• Th2: A time complexity of a Goldberg algorithm is $O(n^2 \cdot m)$

• Pr: from Lemmas 4,5,6

• A strategy for a vertex selection: the highest vertex with an excess $\rightarrow$ # of unsaturated shifts is $\leq 8n^2 \cdot \sqrt{m}$
  
  – Idea: Lower vertices wait for many shifts and then they propagate at once and maybe using a saturated shift

• Best alg.: Goldberg, Tarjan 1996: $O(nm \log(n^2/m))$
Fast (Discrete) Fourier transform

- Motivation: a fast multiplication of polynomials

\[ A(x) = \sum_{j=0}^{n-1} a_j x^j \]
\[ B(x) = \sum_{j=0}^{n-1} b_j x^j \]
\[ C(x) = A(x) \cdot B(x) = \sum_{j=0}^{2n-2} c_j x^j, \quad \text{with} \quad c_j = \sum_{k=0}^{j} a_k b_{j-k} \]

- A multiplication in a lower part: using a point representation in O(n) (for carefully chosen points) vs. a multiplication in an upper part: O(n.n)
Motivation 2

- Df: A vector of coefficients \( c = (c_0, c_1, \ldots, c_{2n-2}) \) is a convolution of vectors \( a = (a_0, a_1, \ldots, a_{n-1}) \) and \( b = (b_0, b_1, \ldots, b_{n-1}) \).

- Evaluation of a polynomial in a given point \( x_0 \) using Horner method:
  \[
  A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \ldots + x_0(a_{n-2} + x_0.a_{n-1})\ldots))
  \]
  - Direct approach: Time complexity per point: \( O(n) \); for 2n points cummulatively \( O(n.n) \)
  - For comparison: Polynomial multiplication using Divide et impera: \( O(n^{\log_2 3}) \)
  - FFT (and IFT): \( O(n \log n) \)
Complex numbers

- Points chosen for evaluation: complex roots of 1
- We use Divide et impera method (so \( n=2^l \))
- Arithmetic of complex numbers …

  ex: \( \omega_8 = \sqrt[8]{1}, \omega^8 = 1; \omega_4 = \sqrt[4]{1} = i, \omega_2 = -1 \)
  
  - Complex n-th roots: roots of a polynomial \( x^n - 1 \)
  
  - A number of roots: \( n \), values \( \omega_n = e^{2\pi i k/n} \) for \( k=0..n-1 \)
    and \( e^{iu} = \cos(u) + i \sin(u) \)

- A primitive n-th root of 1 generates all other roots as its powers. We will use \( \omega_n = e^{2\pi i/n} \). FFT can use any primitive root.
About roots

- Equalities:

\[ \omega_{dn}^{dk} = \omega_n^k : \text{LS} = (e^{2\pi \frac{i}{dn}})^{dk} = (e^{2\pi \frac{i}{n}})^k = \text{PS} \]
\[ \omega_n^{n/2} = \omega_2 = -1 \]
\[ (\omega_n^{k+n/2})^2 = (\omega_n^k)^2 : \text{LS} = \omega_n^{2k+n} = \omega_n^{2k} \cdot \omega_n^n = \omega_n^{2k} = (\omega_n^k)^2 = \text{PS} \]

→ Squares of all n-th roots are only n/2 different n/2-th roots of 1 → recursive calls are evaluated in half of points; in two polynomials (but each result is used 2 times – it is an application of dynamic programming.)
About roots

• For $n \geq 1$ and $k \geq 0$, if $n \mod k \neq 0$ (not $k|n$):
  \[
  \sum_{j=0}^{n-1} (\omega_n^k)^j = 0, \quad LS = \frac{(\omega_n^k)^{n-1}}{\omega_n^k - 1} = \frac{(\omega_n^n)^{k-1}}{\omega_n^k - 1} = \frac{k-1}{\omega_n^k - 1} = 0 = PS
  \]

• ..., if $k|n$:
  \[
  \sum_{j=0}^{n-1} (\omega_n^k)^j = \sum_{j=0}^{n-1} 1 = n
  \]
  
  – A sum of a geometric sequence.

• We evaluate a polynom $A(x)$ of degree $n-1$ with coefs $a_0, a_1, a_2, ..., a_{n-1}$ in points $\omega_n^0, \omega_n^1, \omega_n^2, ..., \omega_n^{n-1}$
  
  – It is a linear transformation, in a matrix form using Vandermonde matrix $F_n$ of order $nxn$ (next slide)
    
    • A note about linearity: higher powers of roots are precomputed
Vandermonde matrix

\[
\begin{pmatrix}
1 & 1 & 1 \\
1 & \omega & \omega^2 \\
1 & \omega^2 & \omega^4 \\
\vdots & \vdots & \vdots \\
1 & \omega^{n-1} & \omega^{2n-2} \\
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
\vdots \\
a_{n-1} \\
\end{pmatrix}
= 
\begin{pmatrix}
A(\omega^0) \\
A(\omega^1) \\
A(\omega^2) \\
\vdots \\
A(\omega^{n-1}) \\
\end{pmatrix}
\]

- \( F_n(i, j) = (\omega_n^i)^j \) - i-th root in a power j
  - Different rows contain different roots
  - The **linear** transformation from \((a_i) \rightarrow A(\omega^i)\) is a Discrete Fourier Transformation (DFT)
  - A DFT is computed in O(n.n) using a definition
Vandermonde matrix, example

- Vandermonde matrix, n=4, for FT (and IFT w/o \( \frac{1}{4} \))

  - \( \omega_4 = i \) \( \lor \) \( \omega_4 = -i \); two possible primitive roots

    1  1  1  1             1  1  1  1
    1  i  -1  -i             1  i  -1  -i
    1 -1  1 -1             1 -1  1 -1
    1 -i  -1  i             1 i  -1  -i

- Lower rows represent higher frequencies
Inverse DFT

- An inversion matrix $F_n^{-1}$ by a guess (no insight, no motivation, but with a check)
  - $(F_n^{-1})_{ij} = \frac{\omega^{-ij}}{n}$, an inv. matrix has the same form (up to a factor $1/n$), but from primitive root $\omega^{-1} = \omega^{n-1}$ (a complex conjugate to the root $\omega$)
- Th: $F_n$ and $F_n^{-1}$ are inverse.
  $$(F_n \cdot F_n^{-1})_{ij} = \sum_{k=0}^{n-1} \omega^{ik} \cdot \frac{\omega^{-kj}}{n} = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{k(i-j)} =$$
  $= 1$, if $i=j$, and 0 otherwise (as in a unit matrix).
  - Corollary: Time complexity of IFT is as FFT.
Algorithm: **Fast FT**

- We create new polynomials $B(x)$ and $C(x)$ for an input polynomial $A(x)$.

\[
A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}
\]

\[
B(x) = a_0 + a_2x + a_4x^2 + \ldots + a_{n-2}x^{n/2-1}
\]

\[
C(x) = a_1 + a_3x + a_5x^2 + \ldots + a_{n-1}x^{n/2-1}
\]

(even coefs)  

(odd coefs)

- It holds:  

\[
A(x) = B(x^2) + x \cdot C(x^2) \tag{1}
\]

so evaluation of $A(x)$ in $n$ points reduces to

1. Evaluation of $B(x)$ and $C(x)$ in $n/2$ points each
2. Evaluation of $A(x)$ from $B(x)$, $C(x)$ according to (1)
Algorithm FFT

recursive_FFT(a)

1 n:=length(a)
2 if n=1 then return(a)
3 \( wn := \exp(2\pi i/n) \); \( w:=1 \); a primitive root+actual
4 \( b:=(a[0],a[2]...a[n-2]) \); \( b:=(a_0,a_2,...,a_{n-2}) \)
5 \( c:=(a[1],a[3]...a[n-1]) \)
6 \( u:=\text{recursive}_\text{FFT}(b) \)
7 \( v:=\text{recursive}_\text{FFT}(c) \)
8 for \( k:=0 \) to \( n/2-1 \) do
9 \( y[k] := u[k]+w*v[k] \); first half of a result
10 \( y[k+n/2]:=u[k]-w*v[k] \); common results u,v
11 \( w := w * wn \); w is an actual root
12 return(y)
Correctness

- **Base case:** $y_0 = a_0$
  - $y_0 = a_0 \cdot \omega_0^1 = a_0 \cdot 1 = a_0$

- **Recursive case:** for $k = 0, 1, \ldots, n/2 - 1$
  
  \begin{align*}
  u_k &= B \left( \omega_n^{k/2} \right) = B \left( \omega_n^{2k} \right) \\
  v_k &= C \left( \omega_n^{k/2} \right) = C \left( \omega_n^{2k} \right)
  \end{align*}

- **Result for $k=0, 1, \ldots, n/2 - 1$**
  
  $y_k = u_k + \omega_n^k v_k = B(\omega_n^{2k}) + \omega_n^k C(\omega_n^{2k}) = A(\omega_n^k)$

- **Result $y_{k+n/2}$ for $k=0, 1, \ldots, n/2 - 1$**
  
  $y_{k+n/2} = u_k - \omega_n^k v_k = u_k + \omega_n^{k+n/2} v_k = B(\omega_n^{2k}) + \omega_n^{k+n/2} C(\omega_n^{2k}) = A(\omega_n^{k+n/2})$

  - Using $-\omega_n^k = \omega_n^{k+n/2}$, $\omega_n^n = 1$
Complexity

• Overhead $\Theta(n)$ in each recursive call ($n$ is an actual size of data)

• Using Master theorem:

$$T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = O(n \log n).$$
Ex. FFT and IFT

\[
\begin{bmatrix}
1 & 0 \\
1 & 3 \\
1 & 3
\end{bmatrix}
\begin{bmatrix}
3 & 2 \\
0 & 2 \\
0 & 2
\end{bmatrix}
\]

\[
\begin{bmatrix}
4 & -2 \\
4+2 & -2-2i \\
6 & -2-2i
\end{bmatrix}
\begin{bmatrix}
2 & -2 \\
4-2 & -2+2i \\
2 & -2+2i
\end{bmatrix}
\]

/ \cdot (1 \ i \ | \ -1 \ -i)

\[
\begin{bmatrix}
6 & 2 \\
6 & 2 \\
8 & 4
\end{bmatrix}
\begin{bmatrix}
-2-2i & -2+2i \\
-2-2i & -2+2i \\
-4 & -4i
\end{bmatrix}
\]

/ \cdot (1 \ -i \ | \ -1 \ i)

\[
\begin{bmatrix}
8-4 & 4+(-i)(-4i) \\
8-4 & 4+(-i)(-4i) \\
8 & 4
\end{bmatrix}
\begin{bmatrix}
12 & 8 \\
12 & 8 \\
12 & 8
\end{bmatrix}
\]

/ \cdot 1/4 = 1/n

OK
Notes

- Row vectors of Vandermonde matrix are independent (as vectors in $\mathbb{C}^n$)
- There are other transformations: a cosine transform (in $\mathbb{R}^n$, JPEG), a wavelet transform
- FFT can be done in finite fields (and weaker struct.)
  - Ex. in $\mathbb{Z}_{17}$: $2^4 \equiv 16 \equiv -1 \pmod{17}$, so $\omega_8 \equiv 2$ in $\mathbb{Z}_{17}$
  - It needs an inverse element to $n$
  - No round-off errors
- (HW:) FFT for $n=8$, $x=(abcdadcb)$, $a, b, c, d \in \mathbb{R}$;
  1. $x=(abcdefgh)$, all numbers are real
  2. $x=(abcdadcdb)$, $a, b, c, d \in \mathbb{C}$
• FFT is (also) a dynamic programming algorithm →
• A transformation of recursion on an iteration
  + a lower (time) overhead
  + (sometimes) a lower memory consumption, compared to a tabulation
    - a more complex and longer program
      • (Q: what is an usual complexity measure in practice?)
• In FFT: reordering of coefs, by a reverse bit notation

\[
\begin{align*}
(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7) \\
(a_0, a_2, a_4, a_6) & (a_1, a_3, a_5, a_7) \\
(a_0, a_4) & (a_2, a_6) & (a_1, a_5) & (a_3, a_7) \\
\end{align*}
\]

\[
\begin{align*}
a_0 & a_4 \\
a_2 & a_6 \\
a_1 & a_5 \\
a_3 & a_7 \\
\end{align*}
\]

• FFT in hardware: A butterfly operation
Butterfly operation

- Inputs (left): $u_k, v_k$, with $\omega_n^k$
- Outputs (right): $y_k = u_k + \omega_n^k \cdot v_k$, $y_{k+n/2} = u_k - \omega_n^k \cdot v_k$
Applications of FT

• A convolution of polynomials
• A signal analysis, a spectral analysis
  • In a function space: each continuous (complex) function can be expressed in a basis of \( \cos(nx) \) and \( \sin(nx) \)
    - Image, video, and audio processing
• A long numbers multiplication
Dynamic programming 1/3

• It is a method for problem solving
  – Usually optimisation problems, an instance decomposition gives common subproblems

• Classical problems:
  – Fibonacci numbers (in 1D)
  – The best matrix multiplication
  – The longest increasing subsequence
  – The longest peak subseq. (increasing and decreasing)
  – The longest common subseq. of two seqs (in 2D)
    • The best „match“ of two/n seqs (DTW: dynamic time warping)
  – The shortest triangulation of a polygon
Dynamic programming 2/3

- Other problems
  - Floyd-Warshall alg. for all minimal paths
  - Fast Fourier transform
  - Bitonic paths in TSP
  - Optimal search tree
  - Optimal print (min. sum of squares of line errors)
  - Optimal coding (in QR codes)
  - Two-player games with perfect information
  - Existence of a derivation in context-free grammars
  - Subset sum (both existence and approx. sol; nonpolynomial)
  - Viterbi alg. (~ the most probable path in DAG)
  - Search of an optimal strategy in a discrete optimisation
Dynamic programming 3/3

• Necessary properties, DP is usable only for some problems
  1. Optimal substructure
  2. Overlapping subproblems
• Bellman principle of optimality: An optimal solution consists only of optimal subsolutions.
  – A possibility to reconstruct a solution; to remember or to recompute an optimal solution using Bellman equation
  – No reconstruction, if only a value of opt. is needed (we can prune subresults)
• Tabelation vs. Bottom-up vs. Top-down computation
  – Tabelation (memoization) for direct use of a rec. alg.
    • A bottom-up approach saves space for some problems
  – Incomplete tabelation
• A various space of subproblems, lazy evaluation
• Ex: A longest path in a graph (DP in nonpolynomial time)
Gate networks

- Gate networks in a wider context: algorithms in a hardware implementation
  - Usually represented by DAG (without loops)
    - Arithmetic expressions: a term/tree structure vs. DAG
  - Operations in parallel
    - Architecture of computation does not depend on input data
    - Gates can have more outputs, which can be used repeatedly
- Particular types of gates
  - Comparator; And, Or, Xor; Plus, Minus, ...
- A nonuniform representation of algorithms
  - Different networks for a various input size
  - (Networks are generated/compiled from an abstract description)
Comparison and Sorting networks

- Comparison network: n inputs, n outputs over some linearly ordered type
  - C.N. uses only one type of a gate: comparator
    - 2 inputs, 2 outputs
  - Sorting network: Outputs are sorted after computation
  - Ex.: an insertion sort, 2-way bubble sort
  - (A counting sort is not implementable)
Sorting networks

• A formal representation:
  – $C = \{C_1, C_2, ..., C_s\}$ is a set of comparators
  – $O = \{(k, i), 1 \leq k \leq s, 1 \leq i \leq 2\}$ is a set of outputs
  – $I = \{(k, i), 1 \leq k \leq s, 1 \leq i \leq 2\}$ is a set of inputs
  – $S = (C, f), f : O \rightarrow I$, N is a sorting network, f is a partial mapping, $f(u,i) \neq f(v,j)$

• A network is acyclic; it has a size $s(S)$ (~sequential time) and a depth $d(S)$ (~parallel time)
  – A comparator is in a depth $d$, if it can run in a step $d$
1. Wires go from an input to an output

2. Comparators connect two wires
   - Each sorting network can be represented in this fashion
   - A network S is a set of comparators. A comparator is a triple \((j,p,q)\), \(1 \leq j \leq d\), \(1 \leq p < q \leq m\), where \(d\) and \(m\) are a depth and a width of a network, respectively

   \[ S_4 = \{(1,1,2), (1,3,4), (2,1,3), (2,2,4), (3,2,3)\} \]
Mergesort

• A sorting network $S_n$ of a width $n$ is recursively defined using two sorting networks $S_{n/2}$ and a merging network $M_{n/2}$ of a width $n$; $n = 2^l$

• Recursion ends for $n=2$. $S_1$ is an empty net.

\[
S_n = S_{n/2} \cup \{(j, \frac{n}{2} + p_1, \frac{n}{2} + p_2) | (j, p_1, p_2) \in S_{n/2}\} \cup \{(k + j, p_1, p_2) | (j, p_1, p_2) \in M_{n/2}\}
\]

for $k = d(S_{n/2})$
Picture: Sorting network
Merging network

- A merging network \( M_n \) of a width 2n merges two sorted sequences of a length n to a single sorted sequence. A construction uses recursion and a base case is for n=1.

- \( M_1 \) is a single comparator \( \{(1,1,2)\} \)

\[
M_n = \{(j, 2p_1 - 1, 2p_2 - 1) | (j, p_1, p_2) \in M_{n/2}\} \\
\cup \{(j, 2p_1, 2p_2) | (j, p_1, p_2) \in M_{n/2}\} \\
\cup \{(k + 1, 2p, 2p + 1) | 1 \leq p \leq \frac{n}{2} - 1\}
\]

for \( k = d(M_{n/2}) \)
Picture: Merging network
Merging network

• Odd elements of both sequences are an input of the first copy of $M_{n/2}$ with outputs $c_i$ and even elements are an input of the second copy of $M_{n/2}$ with outputs $d_i$.

• Outputs of both copies are connected by a single comparator layer, $y_{2i} = d_i$ with $y_{2i+1} = c_{i+1}$.
Correctness proof: a preparation

• L: Let $f$ be a (nonstrictly) increasing function. If sorting network sorts a sequence $a_1, a_2, ... a_n$, then it sorts a sequence $f(a_1), f(a_2), ... f(a_n)$.

• Pr: By induction on $\#$ comparators. If a comparator has inputs $u$ and $v$, then it returns $\min(u, v)$ and $\max(u, v)$ on its output wires. For an increasing function, a comparator returns $\min(f(u), f(v))$ and $\max(f(u), f(v))$, so the ordering is the same.
Zero-one principle

- T: If a sorting network sorts correctly all possible inputs of zeros and ones, then it sorts correctly all inputs.
- Idea: A threshold between any two elements of an input gives a zero-one sequence.
- If an arbitrary sequence is not sorted, then for some \( u \) and \( v \), \( u < v \), the element \( v \) is before \( u \).
- We construct \( f \):
  - \( f(x) = 0 \) if \( x \leq u \) and
  - \( f(x) = 1 \) if \( x > u \)
- The corresponding 0-1 sequence after transformation by \( f \) is not sorted: a contradiction.
Correctness of merging network

• Recall a construction of a merging network.
• For 0-1 input sequences: There are 4 cases depending on a parity of a count of 0 in a's and b's. We show configuration of c's and d's from last zeros in both c's and d's. (if any)
• In all cases the output is sorted or the last level of comparators sorts it. Comparators are shown as „-“.
  1. Even zeros in a's and b's: output „0 0-1 1“
  2. Even zeros in a's and odd zeros in b's: 0 0-0 1-1 1
  3. Odd zeros in a's and even zeros in b's: the same
  4. Odd zeros in both: 0 0-0 1-0 1-1 1

→ a merging network sorts correctly
Size and depth of networks

- A merging network $M_n$ of a width $2n$:
  - A depth from recursion: $d(M_n) = d(M_{n/2}) + 1, d(M_1) = 1$
  - A depth explicitly: $d(M_n) = \log_2 n + 1$
  - A size from recursion: $s(M_n) = 2s(M_{n/2}) + n - 1, s(M_1) = 1$
  - A size explicitly: $s(M_n) = n \log_2 n + 1$

- A sorting network $S_n$ of a width $n$:
  - A depth from recursion: $d(S_n) = d(S_{n/2}) + d(M_{n/2}), d(S_1) = 0$
  - A depth explicitly: $d(S_n) = 1/2 \log_2 n (\log_2 n + 1)$
  - A size from recursion: $s(S_n) = 2s(S_{n/2}) + s(M_{n/2}), s(S_1) = 0$
  - A size explicitly: $s(S_n) = n/4 \log_2 n (\log_2 n - 1) + n - 1$

- Proofs by induction. A size of the sorting network is suboptimal.
A lower bound for a sorting network

• Each sorting network is a comparison network

• L1: Each comparison network returns a permutation of its input values
  – Pr: By induction on a count of comparators. Each comparator swaps or does not swap its inputs

• L2: For a sorting network, all n! permutations are accessible.
  – Pr: We can input an inverse permutation of a chosen permutation and a correct sorting network must sort it.
Sorting networks: a size

- Let $C$ be a sorting network with a width $n$ and let $p$ be a count of accessible permutations in $C$. Then $n! \leq 2^{s(C)}$

- Corollary: $s(C) \in \Omega(n \log n)$ and $d(C) \in \Omega(\log n)$
• HW: Can you restrict a sorting network to less wires? E.g. \( n=5 \).

• Can you add a comparator arbitrarily to a sorting network such that it remains a sorting network?
Arithmetic networks

• An implementation of arithmetic operations using boolean gates (And, Or, Not, Xor, Nand..).
• We show an adder for n-bit numbers
• A single-bit adder: an input x,y,z; an output s,c (sum, carry)
  \[ s = x \text{xor} y \text{xor} z \]
  \[ c = \text{majority}(x, y, z) = (x \land y) \lor (x \land z) \lor (y \land z) \]
• HW: To find a bigger number from two given numbers using also a „<“ gate. (2-way or 3-way)
Adder

• An adder with carry:
  - An input: \( u = \sum_{i=0}^{n-1} u_i 2^i \) and \( v = \sum_{i=0}^{n-1} v_i 2^i \)
    - \( u_i, v_i \in \{0, 1\} \)
  - An output: \( s = u + v = \sum_{i=0}^{n} s_i 2^i, s_i \in \{0, 1\} \)

\[
\begin{align*}
  s_i &= u_i \oplus v_i \oplus c_{i-1} \\
  s_n &= c_{n-1} \\
  c_{-1} &= 0 \\
  c_i &= \text{majority}(u_i, v_i, c_{i-1})
\end{align*}
\]

• A depth of a network (corresponding to a parallel time) is \( \Theta(n) \) and a size is also \( \Theta(n) \).
Carry-lookahead alg.

• We don't have a carry bit quickly enough
• We create a tree structure instead of a linear one:
  – A trick (usable in programming, in theory):
  – Computing with functions (~a f. represents all possible computations) using composition
  – We use 3 functions: Generate, Propagate, Kill
    • Bigger segments are created using a composition of fnc's
• If we have carry bits, then we can compute $s_i$ in a constant depth
Composition of functions

- 3 possible functions of the type: bit → bit
  1. Generate (G): it sets an output bit; \( g_i = u_i \land v_i \)
  2. Propagate (P): it returns input bit; \( p_i = u_i \oplus v_i \)
  3. Kill (K): it returns 0 everytime; \( k_i = \neg (g_i \lor p_i) \)

- A composition \((f_1 \circ f_2)(x) = f_1(f_2(x))\)
  - A single composition enables to double a dependency length.
  - Initial dependencies \( g_i, p_i \)
    are computed from \( u \) and \( v \).

- A representation of a fnc: using two bits: \( g, p \)
  
  \[
  (g_1, p_1) \circ (g_2, p_2) = (g_1 \lor (p_1 \land g_2), p_1 \land p_2)
  \]
Computing all carry bits

- A direct approach of computing $n$ carry bits independently needs $\omega(n)$ gates
  → Computing in two phases to get an $O(n)$ size

1. Computing segments of a length $2^i$ ending on positions $k \cdot 2^i$, for an increasing length

2. Computing remaining segments ending at $k \cdot 2^i$ and starting at 0, for decreasing $i$-th powers

- Because an initial carry bit is 0, the Generate function returns 1 and other functions return 0
Computation of carry bits

- The expression i-j means that the function f, \( c_i = f(c_j) \) was computed.
- 7-0 5-0 3-0 last level
- 6-0
- 8-0
- 8-4 4-0
- 8-6 6-4 4-2 2-0
- 8-7 7-6 6-5 5-4 4-3 3-2 2-1 1-0 first level

- A geometric sequence in both phases: size is \( O(n) \) ← less than 4n gates
(Multiplication)

• HW: Show that a sum of three numbers can be reduced to a sum of two numbers in a constant depth!

• → We need only a logarithmic depth to sum n numbers to two numbers.

• Then we can use an adder of two numbers with a logarithmic depth
(Time) Complexity of problems

• We analysed complexity of algorithms previously.
• We are interested in *complexity of problems* with respect to some classes of algorithms (e.g. sequential or parallel)
• Df: Complexity of a problem is complexity of the best algorithm which solves a given problem
  – An upper bound of a problem complexity is complexity of any algorithm which solves a problem
  – A lower bound is derived from some characteristics of a problem
Decision problems

• Df: A decision problem is a problem which returns an output YES/NO
  - Is there a colouring of a graph G using k colours?
  - Is there a clique of a size (at least) k in a graph G?
  - Is there a solution to Travelling Salesman Problem in a graph G smaller than a threshold t?
    • An optimisation problem reformulated as a decision problem.

• A particular input of a problem is called an instance

• Note: A problem is taken as a set of true instances and an algorithm computes its characteristic function
  - The multiplication problem \( c = a \cdot b \) is formulated as a decision problem \( \{(a, b, c) | a \cdot b = c\} \) w.l.o.g.
Nondeterministic algorithm

- We use nondeterministic algorithm only for decision problems
- A nondeterministic algorithm can use nondeterministic steps. If any branch returns YES, then the whole algorithm returns YES
  - Ex: CLIQUE: We have an instance (G,k). An algorithm chooses k different vertices nondeterministically and then it verifies (in time \(O(k.k)\)) that they create a clique.
    - But: this problem is solvable in polynomial time for fixed k
(Polynomial) Reducibility

- A decision problem $P$ is reducible to a problem $Q$ if we have a function $f$ such that every instance $L$ of $P$ gives the same result as an instance $f(L)$ of $Q$ ($f$ need not be "onto" and "one-to-one")
  - In general, we need functions $f: P_{\text{in}} \rightarrow Q_{\text{in}}$ and $g: Q_{\text{out}} \rightarrow P_{\text{out}}$ for transforming an input and output
  - We work with a polynomial reducibility: $P \leq Q$ (or $\leq_p$)
    A function $f$ (or $f$ and $g$) runs in polynomial time
  - Ex: A problem of finding of a spanning tree is reducible to a problem of a minimal spanning tree.
  - Ex, for general problems: A multiplication for decimal numbers is reducible to a binary multiplication.
Classes of problems

- The complexity class P (or PTIME) is a class of decision problems which are solvable by sequential deterministic algorithms in polynomial time.
- The class NP (or NPTIME) is a class of problems solvable by a nondeterministic sequential algorithm.
- A problem Q belongs to NPComplete if it is from NP and every problem from NP is reducible to Q.
  - Problems from NPC are the hardest problems from NP.
  - Problems form NPC are mutually reducible.
  - NPC ⊆ NP
  - P ⊆ NP, but it is unknown if P=NP.
- A YES solution of an NP-problem can be verified using a certificate deterministically and polynomially.
A class: NP Complete (NPC)

- Polynomial reducibility is transitive.
- How to find a first problem from the NPC class: using a definition. A construction depends on a particular computation model (...)
- Next NPC problems can be found using reducibility: if \( P \leq Q \) and \( P \) is NPC and \( Q \) is NP, then \( Q \) is NPC
  - If \( Q \) has a polynomial alg. then also \( P \) has a polynomial one
  - If there is no polynomial alg. for \( P \) then there is no one for \( Q \)
  - (Df: \( Q \) is \textit{NP hard} if \( P \leq Q \) for any \( P \) from NP)
  - (If \( Q \) is NP hard and from NP then \( Q \) is NPC)
NPC problems

- Warning: A size of an input is measured in bits
- COLOURING: (G,k)
- \(3\text{SAT} \leq \text{SAT}, \text{SAT} \leq 3\text{SAT}; \text{SAT}: a\) satisfiability of propositional formulas in CNF
- HAM: Does a Hamiltonian cycle in G exist?
- Independent Set \(\leq\) CLIQUE
- VertexCover
- SubsetSum \(\leq,\geq\) EqualSubsets; Backpack
- HW: HAM to SAT
- HW: polynomial solutions: 2COLOUR, 2SAT
Easy example

• HAM: a problem of a Hamiltonian cycle
  – An instance: G
  – A question: Does a cycle through all vertices in G (called a Hamiltonian cycle) exist?

• uvHAMP: a problem of a fixed Hamiltonian path
  – An instance: (G,u,v), a graph G and two vertices u,v
  – A question: Does a path through all vertices from u to v (called a Hamiltonian path) exist?

• We show: uvHAMP $\leq_p$ HAM
  – If we know that uvHAMP is NPC and we want to prove that HAM is NPC, then we must also show that HAM is NP.
Easy example: a reduction

- Let \((G,u,v)\) be an instance of uvHAMP ; \(G=(V,E)\)
- We construct \(G' = (V \cup \{x\}, E \cup \{(u,x), (x,v)\})\)
  - \(G'\) is an instance of HAM
1. A construction of \(G'\) is polynomial
2. A graph \(G\) has a Ham. path from \(u\) to \(v\), then \(G'\) has a Ham. cycle from \(u\) to \(v\) to \(x\) to \(u\)
3. A graph \(G'\) has a Ham. cycle. It must go through \(x\), so except \(x\) it must start in \(u\) then go through all vertices and visit \(v\) before it returns to \(x\).
4. (HAM is in NP: we describe a nondeterministic polynomial algorithm)
Reduction

• SAT ≤ CLIQUE

• SAT: An instance is a formula (in propositional logic) in conjunctive normal form. A question is if it exists a satisfying evaluation of variables

• CLIQUE: An instance is a graph G and a number k. A question is if a clique with k vertices exists in G

• Th: SAT in NP, CLIQUE in NP
  – Pr: Directly, we describe relevant algorithms

• Note: Finding a solution of SAT by brute force
Reduction 2

- Syntax of formulas (in CNF):
  - An atomic formula \( \sim \) a propositional variable \( x_i \)
  - A conjunction \( A \land B \)
  - A disjunction \( A + B \)
  - A negation \( \overline{A} \)

- Semantics of formulas:
  - An evaluation of variables \( v: \text{Vars} \rightarrow \{\text{True, False}\} \) generates an evaluation of formulas \( e: \text{Formulas} \rightarrow \{\text{True, False}\} \)
  - \( e(x) = v(x) \); \( e(A \land B) = e(A) \land e(B) \); \( e(A + B) = e(A) \lor e(b) \);
    \[ e(\overline{A}) = \neg e(A) \]

- A Conjunctive Normal Form: a negation has the highest priority, then a disjunction and then a conjunction. \( (x_1 + \overline{x}_2) \cdot (x_2 + \overline{x}_3) \cdot (x_3 + \overline{x}_1) \)
Reduction 3

- Let a formula $A$ be $A = F_1 \cdot F_2 \ldots F_p$, where $F_i = L_{i,1} + L_{i,2} + \ldots + L_{i,q_i}$ and $L_{i,j}$ is a variable or its negation.

- A construction: we create $V = \{(i, j); 1 \leq i \leq k, 1 \leq j \leq q_i\}$
  - Vertices correspond to literals.

- Edges $E$: $((i_1,j_1),(i_2,j_2))$ is an edge, iff $i_1 \neq i_2$ and corresponding literals are not a negation of each other (i.e. they can be both satisfied).

- A new instance of CLIQUE is $((V,E),p)$; a size of a clique is $p$. 
Reduction 4, a proof

• A reduction is polynomial.

• Th: The answers for original and new instances are the same: A formula A is satisfiable iff there exists a clique of a size p in a graph (V,E)

• “→“ : A valid evaluation has some valid literal in each factor. Then a corresponding vertex belongs to a clique, because each two selected vertices are connected by an edge and we selected p vertices.
Reduction 5, a proof 2

• "←": a p-clique fixes an evaluation for some variables. The evaluation is consistent, because possible multiple evaluations to a variable are the same. A formula is valid in this evaluation because a literal was selected and is true in each factor. Remaining variables can take any value. QED
Notes

• An instance belongs to a problem; a problem belongs to a complexity class

• If an instance I (written over an alphabet) is not syntactically correct for P, then \( I \notin P \). An example: 3SAT

• A problem is NPC if it has hard instances. Some instances can be easy (be careful in cryptography).
  
  – (Constraints. SAT solvers. A phase transition for 3SAT)

• Programming in CNF formulas: a (propositional) variable \( x_{O,V,T} \) represents „an object O has a value V in time T“ (e.g. HAM to SAT); an object \( \sim \) a domain var.

• Nonpolynomial \( O(2^{\sqrt{n}}) \) vs. exponential \( O((1+\epsilon)^n) \) algorithms
Approximation alg.

- Approximation algorithms vs. heuristics
  - And how to combine them.

- We want to get an approximate solution to NPC problems in a **polynomial** time.

- **Df:** Approximation ratio. Let \( C^* \) be an (unknown) optimal solution for an optimisation problem. An algorithm has an **approximation ratio** \( r(n) \), iff, for any input size, the cost \( C \) produced by an algorithm is within factor \( r(n) \) of the cost \( C^* \) of the optimal solution: \( \max(C^*/C, C/C^*) \leq r(n) \)
  - (The definition is usable both for min. and max. problems.)
Approximation scheme

- Some problems have an approximation algorithm with a fixed approximation ratio
  - Ex: Colouring of a graph with a ratio of 1,33 (~33% error). It enables a decision between 3 or 4 colours, but 3Colouring is NPC
- An approximation scheme for an optimisation problem is an approximation algorithm that takes an instance and a value $\varepsilon > 0$ and the scheme is an $(1+\varepsilon)$-approximation algorithm for any fixed $\varepsilon$.
  - A polynomial-time approximation scheme (PTAS) runs in time polynomial in the size of $n$
  - A fully PTAS runs in polynomial time in both the size $n$ and $1/\varepsilon$
Example

• A scheme of polynomial algorithms that solve a k-clique problems for a graph G for fixed k's.

• A scheme: We generate k nested cycles through vertices. We test in the innermost cycle that all vertices are different and they create k-clique. (An unoptimized alg.)

• An algorithm is polynomial ($O(n^k)$) for a fixed value k. But the Clique problem is NPC for k given as a parameter
Overview

• An approximation algorithm for Vertex Covering with an approximation ratio 2

• An approximation algorithm for a Travelling Salesman Problem (TSP) with a triangle inequality with an approximation ratio 2

• A nonexistence of an approximation algorithm for a general TSP, without a triangle inequality.

• (Full) PTAS for a Subset Sum problem
Vertex covering

• We give an approximation alg. for a vertex covering with the approximation ratio 2.

• Df: A vertex covering is a subset $V'$ of $V$, s.t. each edge has at least one vertex in $V'$

• Th: the problem of Vertex covering is NPC. (from Independent Set)

• Idea: we repeatedly choose an edge $e$, we add both its vertices to $C$, and we delete all edges incident with $e$.

• $C$ is vertex covering. $C$ has an approximation ratio 2, because no two edges of $C$ share a vertex and at least one vertex of a edge must be in an optimal covering $C^*$. 

• Note: A greedy algorithm selecting a vertex with max. degree does not have an approximation ratio 2. :-(
Travelling Salesman Problem, TSP

- Instance/input: a graph $G= (V,E)$, a length $l: E \rightarrow \mathbb{R}$; $l(e)$ are nonnegative values
- Question: we look for a shortest Hamiltonian cycle
- We give an approximation alg. for TSP with the ratio 2 for an undirected graph with the triangle inequality
  - The triangle inequality for a function $l$: for all $u,v,w$: $l(u,w) \leq l(u,v)+l(v,w)$
  - An alg.: we find a minimal spanning tree (MST) in $G$. We choose a vertex $r$, traverse the MST from $r$ by DFS, and remember a preorder list. A resulting list is an output Hamiltonian cycle $H$. 

Idea of a proof

- We use $|x|$ for a length of $x$:
- Some spanning tree $K$ is in $H^*$: $|K^*| \leq |K| \leq |H^*|$
- The full cycle $H^+$ which includes vertices for each visit is (exactly) 2 times longer than MST: $|H^+| \leq 2|K^*|$
- A deletion of vertices from $H^+$ decreases the length of a path because the triangle inequality holds: $|H| \leq |H^+|$
- Finally: $|H| \leq |H^+| \leq 2|K^*| \leq 2|H^*|$ QED
  - Note: a cycle $H$ can be optimized later locally or during construction (e.g. it can have a cross)
TSP without a triangle inequality

- A triangle inequality is important for TSP:
- Th: if P≠NP and r>1 then there is no polynomial approximation algorithm for TSP with an approximation ratio r.
- Proof: by a contradiction. We show that if exists an alg. A for the theorem then it can be used to solve a Hamiltonian cycle problem, which is NPC.
Transformation

- Let $G = (V, E)$ be an instance of HAM. We transform a graph $G$ to a TSP instance $G' = (V, E')$.
- $G'$ is a complete graph, $l(u,v) = 1$ if $(u,v) \in E$ and $l(u,v) = r.|V| + 1$ otherwise.
- A construction of $G'$ and $l$ is polynomial in $|V|$ and $|E|$.
- Analysis: Let $(G', l)$ be an instance of TSP. If $G$ has a Hamiltonian cycle $H$ then all edges of $H$ have a length 1 and $(G', l)$ has a cycle with the length $|V|$. 

Transformation 2

- If $G$ does not have a hamiltonian cycle then each cycle in $G'$ has an edge outside $E$ and the length of a cycle is at least $(r \cdot |V| + 1) + (|V| - 1) > r \cdot |V|$

  - Because edges outside $E$ are expensive, there is a big difference between a Hamiltonian cycle in $G$ (a length $|V|$) and any other cycle (a length at least $r.|V|$)

- An approximation algorithm must return a Hamiltonian cycle, if it exists, because it does not have any other possibility with a given error $r$.

- If a Hamiltonian cycle does not exist in $G$, then it returns a cycle with a length at least $r.|V| \rightarrow$ we solved HAM in a polynomial time, a contradiction
Subset Sum

- Instance: \((S, t)\), \(S\) is a set \(\{a_1, a_2, \ldots, a_n\}\) of positive integers and \(t\) is a positive integer.
- A decision problem: Does a subset \(S' \subset S\) exist s.t. \(\sum_{a_i \in S'} a_i = t\)?
- An optimisation problem: We look for a subset \(S'\) of \(S\), such that its sum is maximal, but not exceeding the value \(t\).
- Notation: \(S. + .x = \{s + x, s \in S\}\)
- An algorithm for a decision problem based on a dynamic programming in an array of size \(t\).
Alg.

- Alg: SubsetSum(S,t):

```
1 n:=|S|;
2 L[0]:= <0>; a sequence
3 for i:=1 to n do
4   L[i] := mergeList(L[i-1],L[i-1].+.a[i])
5   delete from L[i] all elements over t
6 return(maximum of L[n])
```

- A procedure mergeList merges sorted sequences to a sorted sequence
- A length of L[i] is up to $2^i$
- An approximation scheme: we cut the list L[i] based on a parameter $\delta$, $0<\delta<1$
An approximation scheme

- Each deleted element $y$ has an element $z \leq y$ in a shortened list $L$ such that $\frac{y-z}{y} \leq \delta$, that is $(1-\delta)y \leq z \leq y$. The $z$ represents $y$ with a "sufficiently small error".

- We need a smaller element as a representant, because a greater one can overflow the threshold $t$. 
Algorithm

- Alg: SubsetSumApprox(S,t, eps)

1 n:=|S|;
2 L[0]:= <0> ; a sequence
3 for i:=1 to n do
4   L[i] := mergeList(L[i-1],L[i-1].+.a[i])
5   L[i] := shorten(L[i],eps/n)
5   delete from L[i] all elements over t
6 return(z := maximum of L[n])
Description

- Elements of $L[i]$ are sums of subsets
- We want: $C^*(1-\epsilon) \leq C$ for $C^*$ an optimal solution and $C$ a found one
- We can have an error $\epsilon/n$ in each step. We can prove (using induction over $i$) that for each $y^* \leq t$ from a full version there is $z \in L[i]$ such that 

  \[(1 - \epsilon/n)^n y^* \leq z \leq y^*.\]

  Because $1 - \epsilon \leq (1 - \epsilon/n)^n$, we have $(1 - \epsilon) y^* \leq z$

- Th: A scheme is a fully polynomial-time approximation scheme
PTAS

- Idea: a relative error $\frac{\epsilon}{n}$ divides an interval $1..t$ to a polynomial count of sections and each section has $\leq 2$ representants.

- Another point of view: a computation with the given precision means that we must represent exactly some initial segments of bits of a number in $L[i]$. (We start with higher precision ($\frac{\epsilon}{n}$) because a cumulative error should be at most $\epsilon$). But a fixed count of bits allows only a polynomial count of different represented numbers.
Note

There are other approaches:

**Anytime algorithm**: An optimisation algorithm which can be stopped at any time (after some initial period) and which returns better results if it spends more time on a problem.

Heuristics, a combination of approx. alg. with heuristics, local optimisation as postprocessing.
Probabilistic algorithms

• ... postponed
Cryptography, RSA

- Algorithms are differentiated (in some context) as
  1. Parallel: synchronous, a known number of processors
  2. Distributed: asynchronous, heterogenous
     - Cryptography belongs to distributed algs. in a previous division. Partners compute each their own part of a (complex) algorithm.

- Cryptography: partners: Alice (A), Bob (B); Eve (E, an enemy/eavesdropper); Certification Authority (CA)
  - ... many different protocols and techniques
Motivation example - introduction

- Commuting ciphers. We use:
  - An encryption function $e()$: $\{0..K\} \rightarrow \{0..N\}$
  - A decryption function $d()$: $\{0..N\} \rightarrow \{0..K\}$
  - $d()$ is a left inversion of $e()$: $\forall m: d(e(m)) = m$

- Alice has (her own confidential) $e_A()$ and $d_A()$, Bob has $e_B()$ and $d_B()$.

- Ciphers are commuting: $e_A(e_B(m)) = e_B(e_A(m))$
Commuting ciphers, cont'd

- A protocol for a sending of a message \( m \):
  1. Alice encrypts \( m \) and sends it to Bob: \( e_A(m) \)
  2. Bob encrypts a message and sends \( e_B(e_A(m)) \) to Alice:
  3. Alice deciphers and sends:
     \[
     d_A(e_B(e_A(m))) = d_A(e_A(e_B(m))) = e_B(m)
     \]
  4. Bob deciphers: \( d_B(e_B(m)) = m \)

- A message was encrypted during each transmission with some key.

- Note: A message \( m \) can be a key for a (symmetric) communication, i.e. a session.
Public-key cryptosystems

- It is asymmetric cipher (e() and d() are different)
- It supports also a digital signature.
- Each participant X has a public key $P_X$ and a secret key $S_X$. A secret key is known by X only. A public key can be publicly known (in some list).
- Keys P and S specify functions on a set of all messages (a final sequences of bites) which are “one-to-one” and “onto“ (a permutation on D)
  - In practice: Block ciphers, for various functions f:
    \[ Cipher_i = f (Key, Plain_i, \{Cipher_{i-1}, Plain_{i-1}, i\}) \]
  - An advantage: The same plaintext is encrypted differently in different blocks i.
„Public key“, properties

• $\forall m \in D: P_X(S_X(m)) = m \land S_X(P_X(m)) = m$

• Functions $P()$ and $S()$ are practically evaluable with a knowledge of a key.

• A function $S_X()$ cannot be effectively evaluated with a knowledge of a key $P_X$ (and of a function $P_X()$)
  
  - This is a hard part of a design

  - Generally, algorithms for functions are known, only keys are kept secret (also it is supposed for a security analysis, vs. security by obscurity)
„Public key“, protocol

• Sending a message M from Alice to Bob
  1. Alice gets Bob's public key $P_B$ (from Bob, from „web“ or from a Certification Authority)
  2. Alice encrypts a plaintext M to a ciphertext $C = P_B(M)$
  3. Bob uses on C (from anybody) $S_B$ and gets $M = S_B(C)$

  – Because Eve does not have a key, she cannot compute M from C.

• Note: Alice needs to know that $P_B$ is Bob's key.

• Df/TT: A plaintext: a text to be encrypted

• Df/TT: A ciphertext: a text after an encryption
Public key, digital signature

• Sending a signed message $M'$ from $A$ to $B$
  1. Alice computed a digital signature $s = S_A(M')$
  2. Alice sends a message and a signature: $(M', s)$
     - The message $M'$ is not encrypted here
  3. Bob gets $P_A$ and checks $M' = P_A(s)$
     - If a decrypted message $M'$ is the same as the sent one $M'$, then Bob knows that a message is from Alice and was not altered
     - Practically, messages are also encrypted in step 2. Here we describe only a scheme of a communication.
Hybrid cipher

- Asymmetric ciphers are slow, symmetric ones are quicker. A symmetric cipher uses the same key K for encryption and decryption (i.e. AES and (unsecure) DES)
  - A key K is short, hundreds or thousands of bits
- Instead of a (slow) asymmetric encryption $C = P_A(M)$ Bob computes $C' = K(M)$, $K' = P_A(K)$ and sends $(C',K')$. Alice decrypts a key $K = S_A(K') = S_A(P_A(K))$ and then a message $M = K(C')$ (and checks a digital signature).
- A key K is one-shot generated for a message or for a session. There are also other protocols for a secure sending of a key, which can be combined in hybrid c.
Hybrid authentication

• It is slow to compute a digital signature of a whole message. Only a fingerprint is signed instead of a whole message.

• A fingerprint is computed using a (public one-way) hash function $h$ (SHA-2, MD5), with a typical length of an output 128-512 bits.
  
  – It is hard to find $M$ and $M'$ with $h(M) = h(M')$, i.e. a collision.
Combined protocol: A to B

1. Alice gets Bob's key $P_B$
2. Alice generates a symmetric key $K$, she computes $C = K(M)$ and encrypts $P_B(K)$
3. Alice computes a fingerprint $h(M)$ and its digital signature $s = S_A(h(M))$
4. She sends: (from : $A$, $C$, $P_B(K)$, $s$)
5. Bob reads: „from“: A. He gets $P_A$
6. Bob gets $K = S_B(P_B(K))$ and decrypts $M = K(C)$
7. Bob computes a fingerprint $h(M)$ and compares it with a deciphered $s$ signed by A: $P_A(s) = P_A(S_A(h(M))) = h(M)$
8. If a computed signature and a deciphered one are different then a signature is not from A or the message $M$ was altered.
Notes

• Ad 6: Only an owner of $S_B$ can decrypt the key K.
  − K should be selected from some big set to not enable a brute force search. (Do not select K from a subset)

• Ad 8: It is hard to forge a message M' because it is hard to find a (relevant) message with the same fingerprint as M.
  − Moreover, if M is structured or formatted, then a forged message M' must be structured as well.

• Ad 4: The first part „from:A“ can be encrypted using Bob's public key $P_B$, so Eve cannot recognise a sender.
Certification Authorities

• Bob needs to know that the key $P_A$ belongs to Alice (and is not a forgery)

• A basic solution, used in practice
  – There exists a Certification Authority Z and its public key is known (a key came with an installation or it can be verified on the web)
  – Alice gets (using a safe way) a signed certificate for $C=\text{“Alice's public key is } P_A \text{ “ from Z, i.e.} (C, S_Z(C))$
  – Alice appends this pair to any signed message, so Bob (and any owner of $P_Z$) can verify that C was issued by Z and the key $P_A$ (in C) belongs to Alice.
Extended Euclid Alg.

- **Df:** A greatest common divisor (GCD) of \(a\) and \(b\) is the smallest positive number from a set \(\{a \cdot x + b \cdot y | x, y \in \mathbb{N}\}\), we call it gcd(a,b)

- Extended EA allows a computation of an inverse element in a ring \(\mathbb{Z}_m\)

- **Input:** \(a \geq 0, b \geq 0\)

- **Output:** \(d=\text{gcd}(a,b), x, y: d=a \cdot x + b \cdot y\)
Extended Euclidean Alg.

1. ExtendedEuclidean(a, b)
2. if \( b = 0 \) then
3. return \((a, 1, 0)\)
4. \((d', x', y') := \text{ExtendedEuclidean}(b, \ a \mod \ b)\)
5. \((d, x, y) := (d', y', x'-(a \div b) \cdot y')\)
6. return \((d, x, y)\)

- **Correctness:** using induction through recursion:
  - The result from recursion: \( d' = b \cdot x' + (a \mod b) \cdot y' \)
  - We want \( x \) and \( y \), s.t. \( d = a \cdot x + b \cdot y \). We get using algebraic operations:

\[
\begin{align*}
d &= d' = bx' + (a \mod b)y' \\
&= bx' + (a - \lfloor a/b \rfloor \cdot b)y' \\
&= ay' + b(x' - \lfloor a/b \rfloor y')
\end{align*}
\]
Eucleid alg.

- The alg. needs for n-bit numbers $O(n^3)$ bit operations.
- Idea: The smallest numbers (that is the worst case) that need a given number of steps are Fibonacci numbers.
Rings: $\mathbb{Z}$ modulo $m$

- We define a congruence relation modulo $m$ for a fixed $m$:
  
  - $Df$: $a \equiv b \pmod{m} = m \mid (a-b)$.

- We can use representants $\{0..m-1\}$ of the factor set $\mathbb{Z}_m = \mathbb{Z}/\equiv$ instead of classes $\langle a \rangle_m$.

  - $\langle a \rangle_m + \langle b \rangle_m = \langle a+b \rangle_m$

  - $\langle a \rangle_m \cdot \langle b \rangle_m = \langle a \cdot b \rangle_m$

- A multiplicative inverse element $x$ to $a$ in $\mathbb{Z}_m$, denoted $x = \langle a \rangle_m^{-1}$, fulfills $a \cdot x \equiv 1 \pmod{m}$ and is defined if $a$ and $m$ are relatively prime.
Euler function

• Df: The Euler function $\varphi(n)$ is for $n>1$ a count of positive numbers up to $n$ which are relatively prime to $n$

• Th: If $n$ is a prime number, then $\varphi(n) = n-1$. If $n=p.q$ for different primes $p$ and $q$, then $\varphi(n) = (p-1)(q-1)$

• Th (Euler): For $a$ and $n$ relatively prime ($\gcd(a,n) = 1$) it holds $a^{\varphi(n)-1} \equiv 1 \mod n$

• Corollary: if $\gcd(a,n) = 1$ then $\langle a \rangle_n^{-1} = \langle a^{\varphi(n)-2} \rangle_n$
RSA

1. Choose two big prime numbers \( p \) and \( q \)
2. Compute \( n=pq \). Compute \( r=(p-1)(q-1) \)
3. Choose a (small) odd number \( e \) which is relatively prime to \( r \)
4. Compute a multiplicative inverse element \( d \) to \( e \) modulo \( r \) (using Extended EA).
5. Publish \( (e,n) \) as a public RSA key and remember \( (d,n) \) as a private RSA key. \( (p, q \) and \( r \) are kept secret as well)
Correctness of RSA

- **Th:** The functions $P(M) = M^e \pmod{n}$ and $S(M) = M^d \pmod{n}$ are a pair of mutually inverse functions.

- **Pr:** it holds for all $M < n$: $P(S(M)) = S(P(M)) = M^{ed} \pmod{n}$

- As $d$ and $e$ are mutually inverse elements modulo $r$, we get
  
  $M^{ed} \pmod{n} \equiv M^{1+c \cdot r} \pmod{n}$
  
  $\equiv M \cdot M^{c \cdot \varphi(n)} \pmod{n}$
  
  $\equiv M \cdot 1 \pmod{n}$
  
  $\equiv M \pmod{n}$. **QED**

- We used that $e \cdot d \equiv 1 \pmod{r}$ means $e \cdot d = 1 + c \cdot r$ for some $c$. 
Notes

- We use both \((\text{mod } r)\) and \((\text{mod } n)\) in the proof.
- How to find big prime numbers?
- We can prepare \(P\) and \(S\) ourself and let the CA sign only the \(P\) key. CA does not have the key \(S\).
- A long message is divided to several blocks of an allowed size, depending on a bit-length of \(n\).
RSA: Properties

- Why is the RSA method safe?
  - Nobody is (up to now) able to compute d effectively based on (e,n) without the knowledge of a decomposition n=p.q and also $\varphi(n)=(p-1)(q-1)$.
  - A factorisation of big numbers is a hard problem.
    - (Both checking primality and checking compositonality are polynomially verifiable)

- We can use a quick exponentiation algorithm with an included modulo operation.
- There are other usable hard problems which can be used in a public key cryptography.
Probabilistic algorithm

- Motivation: how we can get an effective alg. for hard decision problems.
  - We used approximation algorithms for optimisation problems.

- A probabilistic algorithm makes also random steps (compared to a deterministic alg.). It uses usually a random number generator (or a pseudorandom generator, to allow rerunning).
Types of probabilistic algorithms

- We describe
  - 1. Algorithms of Las Vegas type
     They return always a correct solution. Randomness affects only a running time.
     Ex: Randomisation of quicksort
  - 2. Algorithms of Monte Carlo type
     Randomness affects a running time as well as a correctness of results.
     Ex: Rabin-Miller test for primality
Randomisation of Quicksort

• A pivot is selected randomly and uniformly in each recursive call.
  – (Combination of methods: Median of three)

• Advantages:
  – An algorithm has good average time ($O(n \log n)$) for all inputs. No input is a priori bad, compared to a deterministic version. But for a particular input and particular random choices a running time can be $O(n^2)$
  – We can run more copies in parallel and take a result from the first finished copy.
Alg. Monte Carlo

• Randomness in an alg. affects correctness of a result. An alg. can make an error, usually only single-sided (for decision problems) and with a limited probability.

• For a comparison: Primality testing with a brute force takes on t-bits numbers $O(2^{t/2})$ steps
Primality testing

• Th (small Fermat): Let \( p \) be any prime number and \( c \) be a number relatively prime to \( p \), \( c < p \). Then \( c^{p-1} \equiv 1 \pmod{p} \)

  – Application: a test of a primality
  – If a conclusion of the Fermat theorem is not fulfilled for a number \( c \) then \( p \) is a composite number (definitely!) and \( c \) is a certificate of compositeness.
  – An implication in the opposite direction is sometimes valid but not always.

→ we need a better test
Witnesses for composite numbers

- Lets T be a set of tuples (k,n), k<n, such that some condition is fulfilled.
  
  1) $k^{n-1}$ is not congruent with 1 (mod n)
  
  2) There are i, s.t. $m=(n-1)/2^i$ is a natural number
     and $gcd(k^{m-1}-1, n)$ is between 1 and n

- Th 1.: A number n is a composite one if it exists k<n, s.t. (k,n) belongs to T

- Th 2.: Lets n be a composite number. Then there exists at least $(n-1)/2$ numbers k<n, s.t. (k,n) belongs to T
Primality test

• Rabin-Miller algorithm:
  – Choose m different probes $k[i]$ randomly from $(1, n-1)$
  – If $T(k[i], n)$ for any $k[i]$ then $n$ is a composite number
  – Otherwise $n$ is a prime number

• A probability of an error
  – If the alg. returns „$n$ is composite“, then it is true (some $k[i]$ is a witness)
  – If the alg. returns „$n$ is prime“, then it can be an error. But all $k[i]$ must be non-witnesses for $n$ in case of error. Then $P(\text{error}) \leq (1/2)^m$ for $m$ independent choices of $k[i]$
Convex hull

- ... skipped
$\mathbb{E} \in 2^{bh(x)} - 1 \cup$