Algorithms and data structures II

TIN061

Ondřej Čepek

Syllabus

1. String matching
2. Network flows
3. Algebraic algorithms
4. Parallel arithmetic algorithms
5. Basic geometric algorithms in the plane
6. Problem reducibility, time complexity classes
7. Approximation algorithms
8. Probabilistic algorithms and cryptography

String matching

Σ finite alphabet (symbol set)

Σ^* set of words over Σ (finite symbol sequences)

word length : $u = x_1x_2 \dots x_m \in \Sigma^* \quad \Rightarrow \quad length(u) = m$ (number of symbols)

word concatenation :

$u = x_1x_2 \dots x_m$, $v = y_1y_2 \dots y_n \quad \Rightarrow \quad uv = x_1x_2 \dots x_my_1y_2 \dots y_n$

(and of course $length(uv) = length(u) + length(v)$)

empty word ε ($\forall\ u \in \Sigma^*$ we have $u\varepsilon = \varepsilon u = u$)

prefix : $u \in \Sigma^*$ is a prefix of $v \in \Sigma^*$ if $\exists\ w \in \Sigma^* : uw = v$

suffix : $u \in \Sigma^*$ is a suffix of $v \in \Sigma^*$ if $\exists\ w \in \Sigma^* : wu = v$

if $w \neq \varepsilon$ than we talk about a proper prefix (suffix)

String matching task:

input: finite alphabet Σ, searched text $x = x_1x_2 \dots x_n \in \Sigma^*$ and samples $K = \{y_1, y_2, \dots, y_k\}$, where $y_p = y_{p,1} \dots y_{p,length(p)} \in \Sigma^*$ for $p = 1, \dots, k$

output: all appearances of the samples in x, i.e. all pairs $[i,p]$ such that y_p is a suffix of the word $x_1x_2 \dots x_i$ ($1 \le i \le n$ and $1 \le p \le k$)

Naive algorithm

```
for p := 1 to k do
  for i := 1 to (n – length(p) + 1) do
  begin  j := 0;
         while (j < length(p)) and (x_{i+j} = y_{p,1+j}) do j := j + 1;
         if (j = length(p)) then Report(i,p)
  end
```

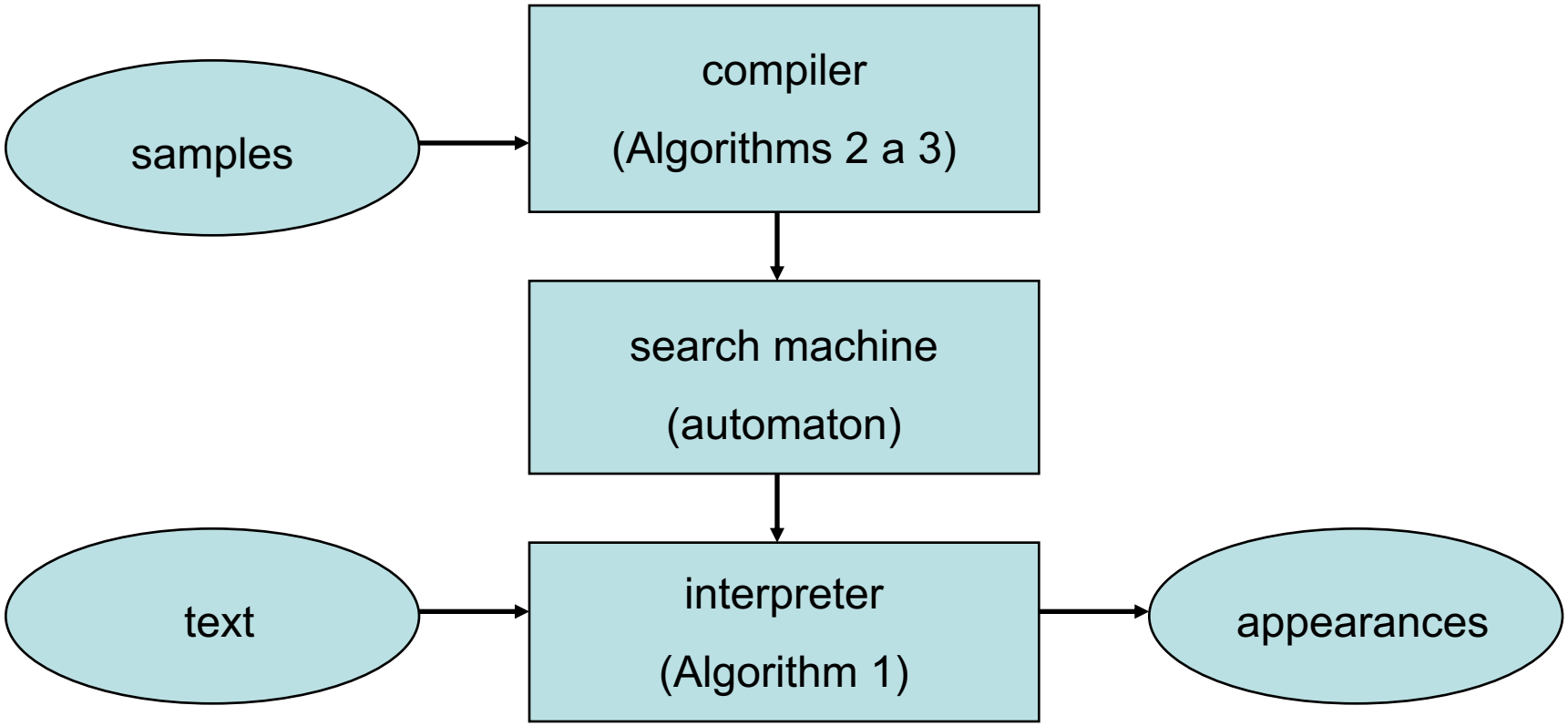
Custom made algorithm

```
c := 0;
for i := 1 to n do
begin
  if (x_i = a) then c := c + 1
          else begin
                 if (c ≥ h – 1) then Report(i,1) ; c := 0
               end
end
```

We shall show, that a custom made algorithm (= finite automaton) can be made for an arbitrary sample or a set of samples in a way, that:

- the automaton (search machine) is built in $\Theta(h \cdot |\Sigma|)$
- the built automaton searches the text in $\Theta(n)$
- total time complexity of the algorithm is $\Theta(n + h \cdot |\Sigma|)$

Algorithm Aho-Corasick (1975)



Search machine for a finite alphabet Σ and a set K of samples is a quadruple

$M = (Q, g, f, out)$, where

1. $Q = \{0,1, \ldots ,q\}$ is a set of states
2. $g : Q \times \Sigma \rightarrow Q \cup \{\perp\}$ is a transition function, for which we have $\forall x \in \Sigma: g(0,x) \in Q$
   (symbol $\perp$ means „undefined“, the transition from state 0 is defined $\forall x \in \Sigma$)
3. $f : Q \rightarrow Q$ is a return function, for which we have $f(0) = 0$ (used when g gives $\perp$)
4. $out : Q \rightarrow P(K)$ is an output function (returns a subset of samples for the given state)

Algorithm 1 (interpreter of the search machine)

input: $x = x_1 \ldots x_n \in \Sigma^*$, $K = \{y_1 \ldots y_k\}$, $M = (Q, g, f, out)$

state := 0;

for i := 1 to n do

begin

(1) while ($g(state,x_i) = \perp$) do state := f(state);

(2) state := $g(state,x_i)$;

(3) for all $y_p \in out(state)$ do Report (i,p)

end

Key properties of the search machine (finite automaton):

1. transition function g

   graph of g (for defined pairs without the loop at state 0) is a valued tree, for which

   - state 0 is the root of the tree
   - each path from the root is valued by some prefix of some sample from K
   - each prefix u of each sample from K values a path from the root to some (exactly one) state s $\Rightarrow$ we say that the prefix (word) u represents state s (in particular the empty word ε represents state 0)
   - the depth of state s represented by word u is defined as d(s) = length(u) and function g satisfies on each tree edge the equality $d(g(s,x_i)) = d(s) + 1$

2. return function f

   for each state s represented by word u we have: state f(s) is represented by the longest proper suffix of u, which is at the same time a prefix of some sample from K

3. output function out

   for each state s represented by word u and for each sample $y_p \in K$ we have:

   $y_p \in out(s)$ if and only if y is a suffix of u

Algorithm 2 (construction of the search machine – 1.phase)

```
input:    K = {y_1 … y_k}                 {set of samples}
output:   Q = {0, … ,q}                   {set of states of the search machine}
          g : Q x Σ → Q ∪ {⊥}              {transition function fulfilling Property 1}
          o : Q → P(K)                    {„half finished“ output function out}
procedure Enter(y_p,1 … y_p,m);           {adding word y_p to the graph of function g}
begin     state := 0; i:= 1;
          while (i <= m) and (g(state,y_p,i) <> ⊥) do
            begin   state := g(state,y_p,i);        {moving along an existing branch}
                    i := i + 1                      {shift in y_p}
            end;
          while (i <= m) do
            begin   Q := Q ∪ {q+1}; q := q+1        {creation of a new state}
                    for all x ∈ Σ do g(q,x) := ⊥;
                    g(state,y_p,i) := q;            {extending a branch}
                    state := q;                     {moving into the new state}
                    i := i + 1                      {shift in y_p}
            end;
          o(state) := {y_p}
end;      {of Enter}
begin     Q := {0}; for all x ∈ Σ do g(0,x) := ⊥;        {main program}
          for p := 1 to k do Enter(y_p);
          for all x ∈ Σ do if g(0,x) = ⊥ then g(0,x) = 0
end.
```

Algorithm 3 (construction of the search machine – 2.phase)

```
input:   Q = {0, … ,q}                    {set of states of the search machine}
         g : Q x Σ → Q ∪ {⊥}               {transition function fulfilling Property 1}
         o : Q → P(K)                      {„half finished“ output function out}
output:  f : Q → Q                         {return function fulfilling Property 2}
         out : Q → P(K)                    {output function fulfilling Property 3}
create an empty queue of states;
f(0) := 0; out(0) := ∅;
for all x ∈ Σ do begin                     {processing the descendants of the root}
                    s := g(0,x);
                    if s <> 0 then
                    begin    f(s) := 0; out(s) := o(s);
                             insert s at the end of the queue
                    end
                 end;
while queue not empty do
begin    r := first state from the queue (and delete r from the queue);
         for all x ∈ Σ do if g(r,x) <> ⊥ then    {processing the descendants of node r}
         begin    s := g(r,x); t := f(r);
                  while g(t,x) = ⊥ then t := f(t);
                  f(s) := g(t,x); out(s) := o(s) ∪ out(f(s));
                  insert s at the end of the queue
         end
end
```

Algorithm Knuth-Morris-Pratt

• simplified version of Aho-Corasick when searching for a single sample ($K = \{y\}$)

• shorter and easier to understand description

• (slightly) better asymptotic time complexity ($\Theta(n + h)$ instead of $\Theta(n + h \cdot |\Sigma|)$)

• the graph of the transition function g is not a tree but a chain (path), which allows that g is not used explicitly at all (this is where the saving in the complexity of preprocessing comes from because g has $h \cdot |\Sigma|$ transitions), function g is used only implicitly

• the return function f is in this case called the prefix function - because for a single sample y the state number corresponds to the length of the prefix (of y) which is represented by the given state, function f has a simpler definition:

$f(s)$ is the length of the longest proper suffix of the word represented by state s (this word is the prefix of y of length s), which is at the same time a prefix of y

• the output function is trivial, it reports y in state h (and nothing in all remaining states)

procedure Prefix (replaces Algorithms 2 a 3)

vstup: $K = \{y\}$ {single sample}
výstup: $f : Q \rightarrow Q$ {prefix function}

```
f(1) := 0;
t := 0;
for q := 2 to h do
begin     while (t > 0) and (y_{t+1} <> y_q) do t := f(t);
          if (y_{t+1} = y_q) then t := t +1;
          f(q) := t
end
```

Algorithm KMP (replaces Algorithm 1)

input: $x = x_1 \ldots x_n \in \Sigma^*$, $K = \{y\}$, prefix function f

```
state := 0;
for i := 1 to n do
begin
          (1)  while (state > 0) and (y_{state+1} <> x_i) do state := f(state);
          (2)  if y_{state+1} = x_i then state := state + 1;
          (3)  if (state = h) then begin Report (i);
                                         state := f(state)
                                   end
end
```

Network flows

Network: directed graph $G = (V,E)$ with two selected vertices s,t (source and sink) and a positive capacity $c(u,v)$ on each edge $(u,v) \in E$. The capacity is defined also for all other pairs of vertices: $c(u,v) = 0$ if $(u,v) \notin E$.

Flow: is a function $f : V \times V \rightarrow R$ that satisfies the following three properties:

1. (Skew symmetry) $\forall\ u,v \in V :\ f(u,v) = -f(v,u)$
2. (Capacity constraint) $\forall\ u,v \in V :\ f(u,v) \leq c(u,v)$
3. (Flow conservation) $\forall\ u \in V - \{s,t\} :\ \Sigma_{v \in V} f(u,v) = 0$ (similar to Kirkhoff's Current Law)

If $f(u,v) > 0$ we say that there is a (positive) flow from u to v. If $f(u,v) = c(u,v)$ then we say that the edge (u,v) is saturated. The value of flow f is denoted by $|f|$ and it is the total net flow from the source, i.e. $|f| = \Sigma_{v \in V} f(s,v)$.

Problem: find a maximum flow, i.e. a flow of a maximum possible value.

Cut: in the context of flows a cut is a partition X,Y of V such that $s \in X$, $t \in Y$. A capacity of cut X,Y is the sum of capacities of all edges crossing the cut from X to Y, i.e. $c(X,Y) = \Sigma_{u \in X, v \in Y} c(u,v)$. A minimum cut is a cut with a minimum capacity. A flow over cut X,Y is the sum of flows on all edges crossing the cut, i.e. $f(X,Y) = \Sigma_{u \in X, v \in Y} f(u,v)$.

Lemma 1: Let f be a flow and X,Y a cut. Then the flow over cut X,Y is equal to the value of the flow f, i.e. $f(X,Y) = |f|$.

Due to Lemma 1 and a trivial fact that $f(X,Y) \leq c(X,Y)$ for every cut X,Y (due to the capacity constraint) it must hold that the value of a maximum flow is less or equal to the capacity of a minimum cut. We shall show, that in fact an equality always holds.

Residual capacity of flow f is a function $r : V \times V \rightarrow R$ defined by

$$r(u,v) = c(u,v) - f(u,v)$$

Residual graph for f: directed graph $R_f = (V,E')$, where $(u,v) \in E'$ if and only if $r(u,v) > 0$ and the value $r(u,v)$ is then called the capacity of edge (u,v) in the residual graph R_f.

Augmenting path for f: an arbitrary path p from s to t in R_f. A residual capacity $r(p)$ of an augmenting path p is equal to the minimum $r(u,v)$ of all edges (u,v) on path p. The value of f can be increased by $r(p)$ by increasing the flow by $r(p)$ on all edges of p.

Remark: when increasing $f(u,v)$ the value $f(v,u)$ must be decreased accordingly to preserve skew symmetry.

Theorem (max flow min cut theorem): The following conditions are equivalent

1. flow f is a maximum flow
2. There exists no augmenting path for f
3. $|f| = c(X,Y)$ for some cut X,Y

The max flow min cut theorem immediately gives a method how to construct a maximum flow by gradual augmentation.

Augmenting path method: Start with a zero flow and repeat the following step, until a flow with no augmenting path is reached.

Augmenting step: Find an augmenting path p for the current flow f and increase the value of the flow by increasing the flow by $r(p)$ along the path p.

Simplest implementation (Ford and Fulkerson): Find some augmenting path for f by searching the graph R_f (using an arbitrary search algorithm for directed graphs).

Remarks:

• if we know a maximum flow we can construct a minimum cut in $0(m)$ time (as in the proof of the max flow min cut theorem)

• if we allow the capacities to be irrational numbers, the simple implementation of the augmenting path method may not terminate after a finite number of steps, the value of the flow always converges but may not converge to the value of the maximum flow

• if all capacities are rational numbers, the problem may be transformed to an equivalent problem where all capacities are integers

• if all capacities are integers, then every augmenting step increases the value of the flow by at least one, and hence the method terminates after at most $|f^*|$ augmenting steps – moreover the constructed maximum flow is integral on every edge

„Ideal version“ of the augmenting path method:

Lemma 2: There always exists a sequence of at most m augmenting steps producing a maximal flow (starting with a zero flow).

Implementation with maximum augmentation (Edmonds and Karp): At each step find an augmenting path with the maximum residual capacity (among all augmenting paths).

Lemma 3: Let f be an arbitrary flow and let f* be a maximum flow on G. Then the value of a maximum flow on the residual graph R_f is $|f^*| - |f|$.

Theorem: The number of augmenting steps when using the implementation with maximum augmentation is $0(m \log c)$ where c is a maximum capacity of an edge.

Remarks:

• this is polynomial (number of augmenting steps) with respect to the size of the data

• the bound is true only for integral capacities (and the method in this case of course converges to a maximum flow)

• an augmenting path with the maximum residual capacity can be found in polynomial time using a modified Dijkstra's algorithm (for a so called bottleneck problem), where the length of a path is not equal to the sum of edge lengths but to the length of the shortest edge (we omit details here as subsequent max flow algorithms we shall cover are better)

• our next goal is an implementation with polynomial time complexity depending only on n and m

Implementation with shortest augmentation (Edmonds and Karp): At each step find a shortest augmenting path, i.e. an augmenting path with the least number of edges.

Theorem: The number of augmenting steps when using the implementation with shortest augmentation is $0(nm)$ and hence the augmenting path method runs in $0(nm^2)$.

Definition: Let f be a flow and R_f its residual graph. Level $u(x)$ of vertex x in R_f is the length of a shortest path from s to x in R_f. Level graph U_f for flow f is a subgraph of R_f, which contains only vertices reachable from s and edges (x,y) for which $u(y) = u(x) + 1$.

Observation: Graph U_f can be constructed in $O(m)$ time using BFS and if there exists an augmenting path then U_f contains all shortest augmenting paths.

Further improvement can be achieved by augmenting the flow on all shortest paths at once instead of augmenting the flow along shortest paths one by one.

Definition: Flow g on graph U_f is called a blocking flow if each path from s to t in U_f contains a saturated edge.

Algorithm (Dinic): Start with a zero flow and repeat the following (blocking) step as long as an augmenting path exist, i.e. as long as t is reachable in the current level graph:

(Blocking) step: Find a blocking flow g on U_f and replace f by $f + g$ defined by

$$(f + g)(x,y) = f(x,y) + g(x,y)$$

Lemma 4: Dinic's algorithm stops after at most $n - 1$ blocking steps.

How to find a blocking flow: Several methods exist, we shall describe the original one proposed by Dinic which is the simplest.

Idea: Find a path from s to t in U_f (using DFS), increase the flow f along this path so that some edge on it becomes saturated. Remove all newly saturated edges from U_f and repeat this procedure as long as t is reachable from s in U_f.

Formal description: Start with a zero flow and go to Initialize. Currently searched vertex will be denoted by x and p will be an augmenting path from s to x.

Initialize: Define p = [s] and x = s. Go to Forward.

Forward: If there is no edge leading from x in U_f go to Backward. Otherwise take an arbitrary edge (x,y), extend p by vertex y and assign y into x (shift the current vertex). If $y \neq t$ then repeat Forward, if $y = t$ go to Augment.

Backward: If $x = s$ then stop (no augmenting path to t exists). If $x \neq s$ then let (y,x) be the last edge on p. Shorten p by deleting x and remove edge (y,x) from U_f. Assign y into x (shift the current vertex) and go to Forward.

Augment: Let $d = \min\{c(x,y) - f(x,y) \mid (x,y)$ is an edge in p$\}$. Add d to the flow on all edges in p, remove all newly saturated edges from U_f and go to Initialize.

Theorem: Dinic's algorithm finds a blocking flow in the level graph U_f in $0(nm)$ time and a maximum flow in the input graph G in $0(n^2m)$ time.

The „preflow-push“ method

Definition: Preflow is a function $f : V \times V \rightarrow R$ which fulfils the symmetry and capacity constraints on each edge and which relaxes the flow conservation condition at each vertex u (except of s) by $e(u) = \Sigma_{v \in V} f(u,v) \geq 0$, where e(u) is the excess in u. Vertex u different from both s and t is called active if it has a positive excess ($e(u) > 0$).

Definition: Let f be a preflow and R_f a residual graph for f. Then we call a function $h : V \rightarrow N$ a height function with respect to f if

- $h(s) = |V|$
- $h(t) = 0$
- $\forall (u,v) \in R_f : h(u) \leq h(v) + 1$

If $h(u) = h(v) + 1$ holds for edge $(u,v) \in R_f$ we call (u,v) to be feasible.

Initialization (for a generic preflow-push algorithm):

1. Set all h(u), e(u) and f(u,v) to zero.
2. Assign $h(s) := |V|$.
3. Assign for each neighbor u of source s
   - $f(s,u) := c(s,u)$ a $f(u,s) := - c(s,u)$
   - $e(u) := c(s,u)$

After initialization: f is a preflow and h is a height function with respect to f.

The algorithm uses two basic actions:

PUSH(u)

- USAGE : if u is active (e(u) > 0) and there exists a feasible edge (u,v) in R_f, i.e. an edge satisfying r(u,v) > 0 and h(u) = h(v) + 1.
- ACTION : send d(u,v) = min {e(u), r(u,v)} units of flow from u to v and update f(u,v), f(v,u), e(u) and e(v) accordingly.

Definition: PUSH(u) is saturating, if edge (u,v) over which the flow is sent becomes saturated (which happens if d(u,v) = r(u,v)) and non-saturating in the opposite case (i.e. when d(u,v) = e(u) < r(u,v)).

LIFT(u)

- USAGE : if u is active (e(u) > 0) and there exists no feasible edge (u,v) in R_f.
- ACTION : h(u) := 1 + min {h(v) | (u,v) v R}

Algorithm (generic preflow-push algorithm):

- Initialization
- As long as an active vertex exists, select one (let it be u) and perform either PUSH(u) or LIFT(u).

Lemma 1: Each vertex u satisfies during the entire run of the algorithm the following:

- $h(u)$ never decreases
- each LIFT(u) increases $h(u)$ by at least one

Lemma 2: Function h is a height function with respect to the current (pre)flow f during the entire run of the algorithm.

Lemma 3: There is no path from s to t in R_f during the entire run of the algorithm.

Theorem (correctness of the algorithm): When the algorithm terminates f represents a maximum flow.

Time complexity: The algorithm performs during its entire run:

- $O(|V|^2)$ lifts
- $O(|V||E|)$ saturating pushes
- $O(|V|^2|E|)$ non-saturating pushes

which, under an implementation requiring $O(|V|)$ time for each lift and $O(1)$ time for each push, gives an overall $O(|V|^2|E|)$ complexity of the algorithm.

Remark: The above time complexity can be reduced to $O(|V||E| \log(|V|^2 / |E|))$ by a „smart“ selection of active vertices [Goldberg, Tarjan 88], which is still an asymptotically fastest known algorithm.

Maximum matching in bipartite graphs

For a graph $G=(V,E)$ a matching is a subset of edges $M \subseteq E$ such that no two edges from M share a vertex. The size of a matching is the number of edges in it.

A graph $G = (V,E)$ is bipartite if $V = V_1 \cup V_2$ and $\forall\ (u,v) \in E$ we have $u \in V_1$ and $v \in V_2$.

Finding a maximum size matching in a bipartite graph has many practical applications when we want to pair two types of objects (e.g. people and available sandwiches). This problem can be solved by finding a maximum flow in an extended graph:

- Direct all edges from V_1 to V_2 (assume V_1 has size n and V_2 at most n)
- Add a source vertex s and an edge from s to every vertex in V_1
- Add a sink vertex t and an edge from every vertex in V_2 to t
- Set all edge capacities to 1

Since all capacities are integers, also the maximum flow will be integral on every edge. A maximum size matching corresponds to edges from V_1 to V_2 with flow 1.

Moreover, even the simplest max-flow algorithm (generic Ford-Fulkerson) works in $O(nm)$ time in this special case

- Finding an augmenting path takes $O(m)$
- Each augmenting path increases the flow by one unit
- The algorithm performs at most n augmentations

Multiplication of polynomials

Two ways to represent polynomials:

1. Vector of coefficients (in our case a vector of complex numbers)

$$A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1x^1 + a_0x^0$$

A(x) is a polynomial with a degree bound n, coefficients a_0 … a_{n-1}, and A(x) has degree k if a_k is the highest nonzero coefficient

- Addition of two polynomials: $\Theta(n)$
- Computation of a value in a given point: $\Theta(n)$ (using Horner's schema)
- Multiplication of two polynomials : $\Theta(n^2)$ (convolution of two coefficient vectors)

2. Function values in given points (set of pairs)

$$\{ (x_0,y_0), (x_1,y_1), \ldots , (x_{n-1},y_{n-1}) \}$$

Any polynomial A(x) with a degree bound n can be uniquely represented in this way by any such n-tuple of pairs, where x_i's are pairwise disjoint and $y_i = A(x_i)$ for each i.

- Addition of two polynomials: $\Theta(n)$
- Computation of a value in a given point: except of the given points impossible to compute without a conversion to the representation by a vector of coefficients
- Multiplication of two polynomials : $\Theta(n)$ BUT we need 2n points

Conversions between both representations

1. Coefficients → Pairs (evaluation): $\Theta(n^2)$ using Horner's schema (also for $2n$ points), possible in $\Theta(n \log n)$ if points for the computation of function values are selected in a „clever" way
2. Pairs → Coefficients (interpolation): in general $\Theta(n^3)$ using Gaussian elimination or $\Theta(n^2)$ by Lagrange's formula (will not cover), possible in $\Theta(n \log n)$ when the function values for the „cleverly" selected points are known

Multiplication of polynomials in $\Theta(n \log n)$

The selected points are $2n$-th complex roots of unity (number 1), denoted by

$$w_{2n}^0, w_{2n}^1, \ldots, w_{2n}^{2n-1}$$

Multiplication for two vectors of coefficients $a_0 \ldots a_{n-1}$ and $b_0 \ldots b_{n-1}$ (both of degree bound n) then proceeds as follows:

1. Set $a_n = \ldots = a_{2n-1} = b_n \ldots b_{2n-1} = 0$ (new degree bound is $2n-1$)
2. Evaluation: compute function values $A(x)$ and $B(x)$ in all $2n$ roots of number 1
3. Multiplication: $C(x) = A(x)B(x)$ in all $2n$ roots of number 1
4. Interpolation: compute the vector of coefficients of the polynomial $C(x)$ which is given by $2n$ function values (in all $2n$ roots of number 1)

Steps 1 and 3 take $\Theta(n)$, steps 2 and 4 take $\Theta(n \log n)$ if implemented by FFT

Properties of complex roots of unity

Cancelation lemma: If $n \geq 0$, $k \geq 0$ and $d > 0$ then $w_{dn}^{dk} = w_n^k$.

Corollary: For any even integer $n > 0$ we get $w_n^{n/2} = w_2^1 = -1$.

Halving lemma: If $n > 0$ is even, then the squares of the n complex n-th roots of unity are equal to the $n/2$ complex $n/2$-th roots of unity (each occurring twice).

Summation lemma: If $n \geq 0$ and $k > 0$ such that k is not divisible by n then $\Sigma_{j=0}^{n-1} (w_n^k)^j = 0$.

Discrete Fourier Transform (DFT)

We want to represent a polynomial $A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1x^1 + a_0x^0$ with degree bound n given by the coefficient vector by function values in n complex n-th roots of unity, i.e. we want to compute the values

$$y_k = A(w_n^k) \text{ pro } k = 0, 1, \ldots, n-1.$$

Vector y (of function values) is the DFT of vector a (of coefficients), we write $y = DFT_n(a)$.

The reverse transform is called the inverse DFT and we write $a = DFT_n^{-1}(y)$.

Remark: if n is the degree bound of both input polynomials (which we want to multiply), then we in fact work with $n' = 2n$, however, to keep the notation simple we shall use n (moreover the asymptotic complexity is the same in terms of both n and n').

Fast Fourier Transform (FFT)

An algorithm using the „divide and conquer“ strategy for computing $DFT_n(a)$, by dividing $A(x)$ into two polynomials of degree bound $n/2$ (separating odd and even degrees):

$$A^s(x) = a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{n/2-1}$$
$$A^l(x) = a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{n/2-1}$$

Now clearly $A(x) = A^s(x^2) + x\,A^l(x^2)$ (denote this equation by (¤)), so the task to compute the function values of $A(x)$ at $w_n^0, w_n^1, \dots, w_n^{n-1}$ is transformed into the tasks to

• compute the function values of polynomials $A^s(x)$ a $A^l(x)$ (with degree bound $n/2$) at $(w_n^0)^2, (w_n^1)^2, \dots, (w_n^{n-1})^2$ (which is only $n/2$ different points due to the halving lemma), i.e. instead of solving the original task solve twice the same task on data of half the size

• combine the results using the equation (¤) which takes $\Theta(n)$ time

Total time $T(n)$ for FFT of length n fulfils $T(n) = 2T(n/2) + \Theta(n)$ and so $T(n) = \Theta(n \log n)$.

Lemma Let V_n be the Vandermonde matrix for $w_n^0, w_n^1, \dots, w_n^{n-1}$. Then the entry at position (j,k) in the inverse matrix V_n^{-1} is equal to w_n^{-kj} / n.

Thus we can compute also the inverse DFT using FFT, because computing the coefficients of the polynomial given by function values $y_0, y_1, \dots, y_{n-1}$ is the same as computing function values of the polynomial $Y(x)/n$ with coefficients $y_0/n, y_1/n, \dots, y_{n-1}/n$ at points $w_n^0, w_n^{-1}, \dots, w_n^{-(n-1)}$

FFT implementations

1. Recursive implementation (direct transcription of the algorithm)

```
REC-FFT(a);
n := length(a);                    {n is a power of two}
if n=1 then return a;              {bottom of the recursion}
p := e^(2πi / n);                  {principal root w_n^1 = generator of all other roots}
w := 1;                            {currently processed root, starting with w = w_n^0}
a^s := (a_0, a_2, … , a_(n-2));
a^l := (a_1, a_3, … , a_(n-1));    {preparation of the input data for recursion}
y^s := REC-FFT(a^s);
y^l := REC-FFT(a^l);               {recursion}
for k := 0 to (n/2 – 1) do
        y_k := y_k^s + w y_k^l;        {computing A(w) in the current root w = w_n^k}
        y_(k+n/2) := y_k^s – w y_k^l;  {computing A(w) in the opposite root w_n^(k+n/2)}
        w := w p;                      {moving to the next root w_n^(k+1)}
return y
```

Time complexity: obviously $T(n) = 2T(n/2) + \Theta(n)$ and so $T(n) = \Theta(n \log n)$, because the work outside of the recursion is clearly $\Theta(n)$.

2. Iterative implementation (main idea)

originates from the recursive implementation by the following adaptations

- the body of the for cycle is replaced by the „butterfly“ operation
- the tree of recursion is transversed level by level from the bottom
- the initial order of leaves for starting the algorithm is obtained by bit inversion

Time complexity: exactly $n/2$ butterfly operations is called on each level of the tree, so the total work on each level is $\Theta(n)$, which together with the fact that the depth of the tree is exactly $\log n$ gives the $\Theta(n \log n)$ time complexity.

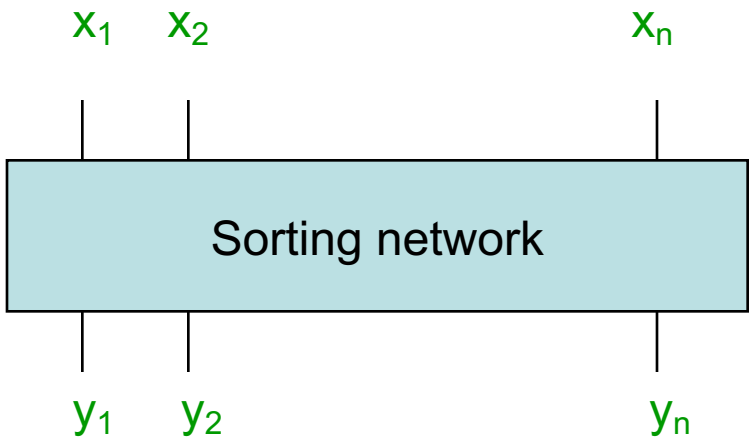
3. Parallel implementation (main idea)

- all butterfly operations on the same level of the tree are performed in parallel
- the algorithm can be hardwired by an appropriate chaining of butterfly circuits into a network with „width“ n and „depth“ $\log n$

Time complexity: the work on each level no w takes $O(1)$ (using $n/2$ „processors“), and so the total time complexity is $\Theta(\log n)$.

Sorting networks

Sorting network is a circuit with n inputs with values from some linearly ordered type (typically numbers) and n outputs, which output the input values sorted (regardless of the order in which these values appeared on the input).



This circuit contains only one type of a gate, namely a comparator, which is a gate with two inputs x_1 and x_2 and two outputs y_1 and y_2, where $y_1=\min\{x_1,x_2\}$ and $y_2=\max\{x_1,x_2\}$.

Formal definition of a sorting network:

- $K = \{K_1, K_2, \ldots, K_s\}$ is the set of comparators, s is the size of the network
- $O = \{ (k,i) \mid 1 \le k \le s, 1 \le i \le 2 \}$ is the set of outputs (k is the comparator number, i the output number)
- $I = \{ (k,i) \mid 1 \le k \le s, 1 \le i \le 2 \}$ is the set of inputs
- $C = (K, f)$ is a sorting network, where $f : O \rightarrow I$ is a partial injective mapping

Network acyclicity condition:

We require that the directed graph $G = (K,E)$, where $(K_u,K_v) \in E$ if there exist i and j such that $f(u,i) = (v,j)$, is acyclic.

Dividing comparators into levels:

- Define $L_1 = \{ K_i \mid K_i$ has indegree zero in $G\}$ (L_1 is nonempty due to the acyclicity of G)
- Let $L_1, L_2, \ldots, L_h$ be defined, where $L = L_1 \cup L_2 \cup \ldots \cup L_h \not\subset K$. Then we define $L_{h+1} = \{ K_i \mid K_i$ has indegree zero in $G \setminus L\}$ (L_{h+1} is nonempty due to the acyclicity of G)
- The number of levels is denoted by d and called the depth of the network

How the network functions:

- time 0 : network inputs defined (all comparators in L_1 have defined inputs)

  comparators in L_1 working
- time 1 : all comparators in L_2 have defined inputs

  comparators in L_2 working

  …
- time d-1 : all comparators in L_d have defined inputs

  comparators in L_d working
- time d : network outputs defined

Observation: time complexity of sorting corresponds to the network depth

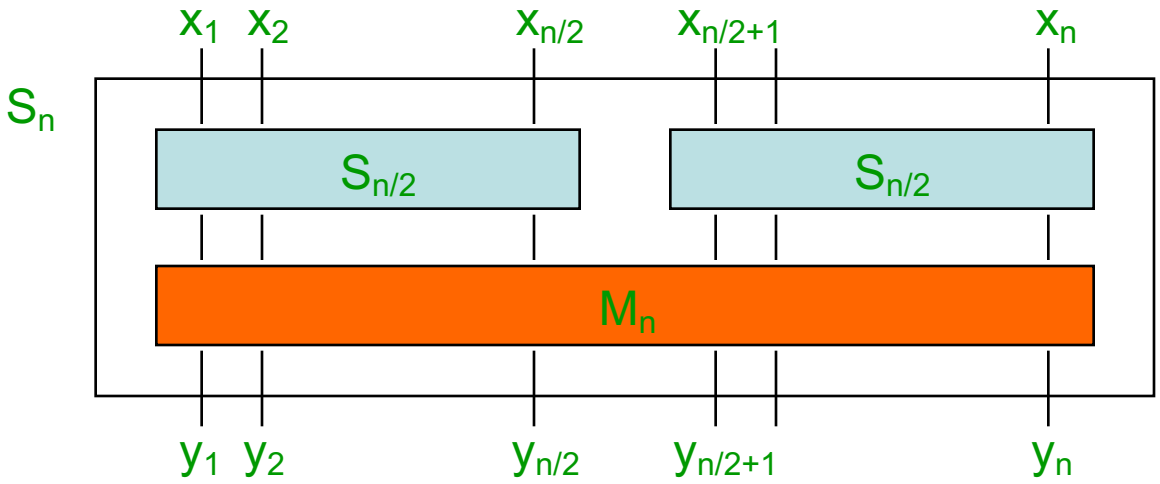
Topologically different network representation:

- each „wire“ from input x_i into output y_i is drawn as a straight line
- individual comparators are „stretched“ between the corresponding „wires“
- every sorting network can be redrawn in this way
- the number of inputs/outputs (wires) is called the width of the network

Merge-Sort implemented by a sorting network

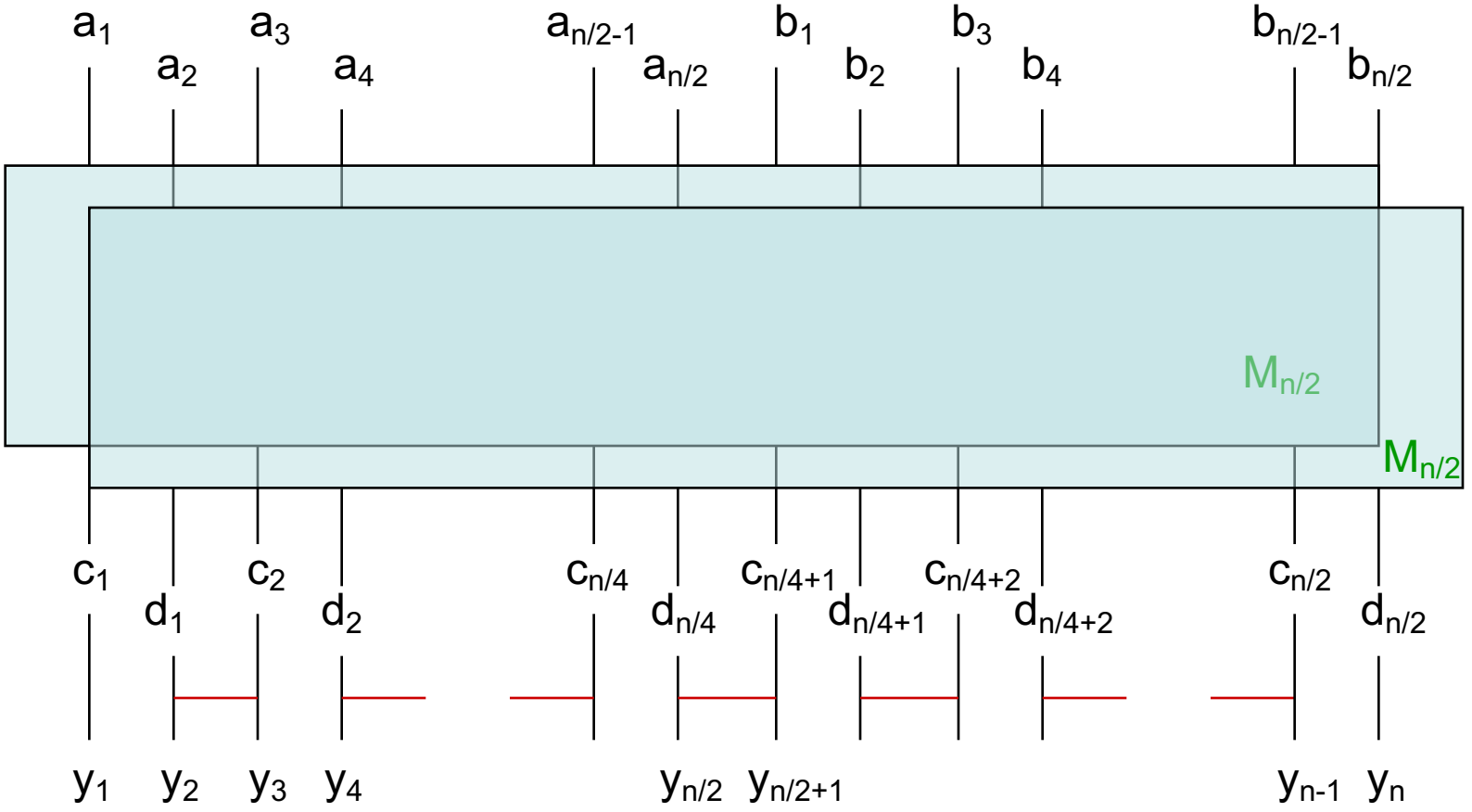
We want to sort $x_1, x_2, \ldots, x_n$ (and assume that n is a power of two)

We implement the sorting by a network S_n, which is defined recursively by the picture



where M_n is a merging network of width n (the recursion stops for $n=2$)

It remains to show how to construct merging network M_n (by a recursive construction):

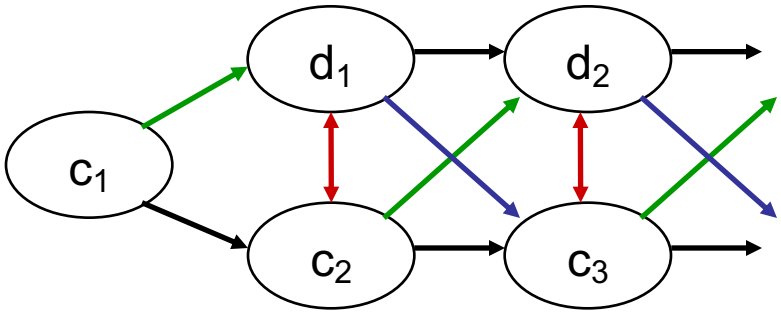


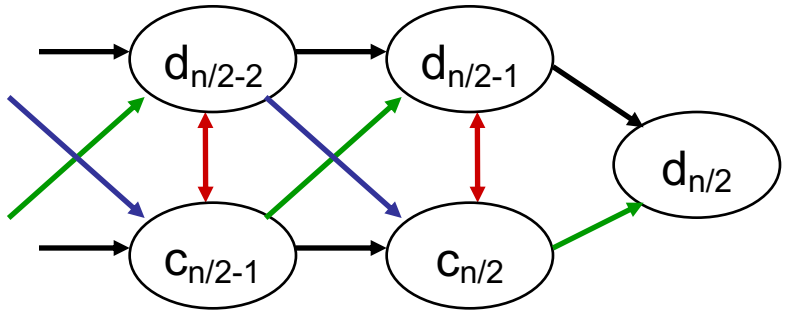
Odd members of both sorted sequences are the input of the first copy of $M_{n/2}$ and even members are the input of the second copy of $M_{n/2}$. Moreover, outputs of both networks are interconnected by one layer of comparators (marked red in the picture).

The input satisfies: $a_1 \le a_2 \le \ldots \le a_{n/2}$ and $b_1 \le b_2 \le \ldots \le b_{n/2}$

Induction hypothesis: $c_1 \le c_2 \le \ldots \le c_{n/2}$ and $d_1 \le d_2 \le \ldots \le d_{n/2}$

We shall prove that: $y_1 \le y_2 \le \ldots \le y_n$





Black inequalities (arrows) are valid, green inequalities and blue inequalities will be proved. Regardless of how the red comparators order their pairs of values, the resulting arrows will generate a linear order, which is the correct ordering of the input values.

Depth and size of a sorting network of width $n = 2^k$

1. Merging network M_n

   has depth (number of levels) $d(M_n) = \log_2 n$

   and size (number of comparators) $s(M_n) = n/2 \log_2(n/2) + 1$

2. Sorting network S_n

   has depth (number of levels) $d(S_n) = 1/2 \log_2 n (\log_2 n+1)$

   and size (number of comparators) $s(S_n) = 1/4\ n \log_2 n (\log_2 n–1) + (n–1)$

Lower bound for the complexity of sorting by comparator networks

Comparator network = network composed of comparators (arbitrarily placed)

$\Rightarrow$ every sorting network is a (specially designed) comparator network

Lemma 1 Every comparator network working on input $x_1, x_2, \ldots, x_n$ outputs some permutation Π of the input values, i.e. outputs $x_{\Pi(1)}, x_{\Pi(2)}, \ldots, x_{\Pi(n)}$.

Proof Trivial, the network cannot do anything else then permute the inputs.

Definition Let C be an arbitrary comparator network of width n. A permutation $\Pi : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, n\}$ is reachable for C, if there exists an input sequence $x_1, x_2, \ldots, x_n$ such that C outputs the sequence $x_{\Pi(1)}, x_{\Pi(2)}, \ldots, x_{\Pi(n)}$.

Lemma 2 Let C be an arbitrary sorting network of width n. Then all n! permutations are reachable for C.

Lemma 3 Let C be an arbitrary comparator network of width n and let p be the number of reachable permutations for C. Then

$$p \leq 2^{s(C)}.$$

Corollary If C is a sorting network of width n then

$$n! \leq 2^{s(C)}$$

and thus C has size $s(C) = \Omega(n \log n)$ and depth $d(C) = \Omega(\log n)$.

Arithmetic networks

• definition is similar to sorting networks, identically defined width (number of inputs or "input wires"), size (number of gates), and depth (number of levels) of the network

• time complexity of the computation is again proportional to the depth of the network

• "wires" carry values 0 and 1 (i.e. a single bit of information, unlike the wires in sorting networks which carry more bits each, e.g. n-bit numbers)

• gates are logical gates, typically a unary gate NOT and binary gates AND, OR, XOR

• a "wire" from a single gate output may split into several gate inputs, number of outputs may differ from the number of inputs, but network acyclicity must be preserved

Full adder

Full adder is a circuit with three inputs x,y,z and two outputs c,s. Its function can be explained as follows: x and y are two bits of the same order in two binary numbers, z is the carry-in from the lower order, s the corresponding bit of x+y, and c is the carry-out into the higher order. The 8-line table for computing s and c implies s = parity(x,y,z) and c = majority(x,y,z).

Adding two n-bit binary numbers

• n is assumed to be a power of two, inputs are $a = (a_{n-1}, \ldots, a_0)$ and $b = (b_{n-1}, \ldots, b_0)$

• we shall show two circuits: for classical addition and carry-look-ahead addition

Circuit for classical addition

• addition is realized by a cascade of n full adders, where adder of order i waits for the carry-in bit from adder of order $i-1$ and sends its carry-out bit to adder of order $i+1$

• the network has size $\Theta(n)$ and depth also $\Theta(n)$

Circuit for carry-look-ahead addition

The speed-up is based on the idea, that in some cases the carry-out bit from order i may be determined based on a_i and b_i only (these values are known from the start) and there is no need to wait for the carry-in bit (which may take a very long time to compute)

• in case $a_i = b_i = 0$ we get $c_{i+1} = 0$ (regardless of the value of c_i) → kill carry bit c_{i+1}

• in case $a_i = b_i = 1$ we get $c_{i+1} = 1$ (regardless of the value of c_i) → generate carry bit c_{i+1}

• in case $a_i \neq b_i$ we get $c_{i+1} = c_i$ → propagate carry-in bit c_i into carry-out bit c_{i+1}

This defines the carry status of each order $i \in \{0, \ldots, n\}$, denoted by variable x_i. Its value from the set $\{k,p,g\}$ can be easily computed from a_{i-1} and b_{i-1} using the above rules.

Consider two consecutive full adders as a single circuit with one carry-in bit and one carry-out bit: the carry status of the combined circuit can be computed from the (k,p,g) carry statuses of both participating full adders, and this can be extended to triples …

This defines a binary carry status operator $\otimes$ on the set $\{k,p,g\}$, which is associative (associativity can be proved by brute force by analyzing all 27 cases)

If we define $x_0 = y_0 = k$, then for every order $i \in \{1, \ldots, n\}$ we may define a variable

$$y_i = y_{i-1} \otimes x_i = x_0 \otimes x_1 \otimes \ldots \otimes x_i$$

which may be interpreted as the i-th prefix of the product $x_0 \otimes x_1 \otimes \ldots \otimes x_n$

Lemma For $i = 0, 1, \ldots, n$ we get

1. if $y_i = k$ then $c_i = 0$
2. if $y_i = g$ then $c_i = 1$
3. the case $y_i = p$ cannot happen

Corollary If all y_i are known then the sum can be computed in $O(1)$ using n full adders running in parallel (where $k = 0$ ang $g = 1$ on the input of the full adders, p cannot occur), so the task of adding two binary numbers is reduced to computing all prefixes.

The adder circuit consists of n KPG circuits and one prefix circuit. Each KPG circuit is used twice. During the first pass the i-th KPG circuit has a_i and b_i on the input and outputs x_i, which it sends into the prefix circuit. During the second pass the i-th KPG circuit works as a full adder (in fact only as its part computing the parity function) on the input a_i, b_i and y_i (the last input coming from the prefix circuit, where the values k and g are interpreted as 0 and 1) and outputs the relevant bit of the sum $s_i = \text{parity}(a_i, b_i, y_i)$.

Thus, all that remains to construct now is the parallel prefix circuit …

Parallel prefix circuit

The inputs are the carry statuses of all orders, i.e. $x_0, x_1, \ldots, x_n$ and the computed outputs are the values of all prefixes $y_0, y_1, \ldots, y_n$

Let us denote $[i, j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$ (and so $[i, i] = x_i$). By associativity we get for any j such that $i < j \leq k$ the equality $[i, k] = [i, j-1] \otimes [j,k]$.

We want to compute $y_i = [0, i]$ for all $i \in \{0, 1, \ldots, n\}$ by a parallel prefix circuit consisting only of sub-circuits realizing the carry status operator $\otimes$. The circuit is topologically a complete binary tree (here we need n to be a power of 2):

- the leaves of the tree contain the inputs x_1 to x_n and the root contains the input x_0
- during the pass from the leaves to the root the value $[i, k] = [i, j-1] \otimes [j,k]$ is computed in the inner node representing the interval from i to k, based on the values in the left son representing the interval from i to $j-1$ (and thus having the value $[i, j-1]$) and in the right son representing the interval from j to k (and thus having the value $[j, k]$)
- during the pass from the root to the leaves the value $[0, i-1]$ comes from the parent to an inner node which represents the interval from i to k - this value is sent unchanged to the left son (which represents the interval from i to $j-1$) and value sent to the right son (which represents the interval from j to k) is computed as $[0, j-1] = [0, i-1] \otimes [i,j-1]$
- the outputs y_0 to y_{n-1} are computed in the leafs and the output y_n in the root of the tree
- the size of the circuit is $\Theta(n)$ and its depth is $\Theta(\log n)$

Basic geometric algorithms in the plane

Convex hull

Input: n points in the plane given by their coordinates $[x_1,y_1], \ldots ,[x_n,y_n]$
Output: sequence of points which define the convex hull = vertices of the smallest convex polygon that encircles all the input points

Plane sweeping algorithm

- Sort points by their x-coordinates (from smallest „on the left“ to largest „on the right“)
- Use a sweeping line that goes from left to right (from x_1 to x_n) in n moves
- After k moves of the line we hold the convex hull of the first k points
- The convex hull is composed of an upper envelope and a lower envelope

Sort points by their x-coordinates (assume general position) and label them $b_1, \ldots ,b_n$
Initialize upper and lower envelopes $U \leftarrow (b_1)$ and $L \leftarrow (b_1)$
for k from 2 to n do
    while $|U| > 2$ and $x_{j-1}x_jb_k$ is a left-turning angle (where $U = (x_1, \ldots , x_{j-1},x_j)$) do
        remove the last vertex from U
    add b_k to the end of U
    while $|L| > 2$ and $x_{j-1}x_jb_k$ is a right-turning angle (where $L = (x_1, \ldots , x_{j-1},x_j)$) do
        remove the last vertex from L
    add b_k to the end of L
Output the convex hull by traversing U and then L backwards

Time complexity: O(n logn) for sorting and O(n) the rest (each point removed at most 1x)

Intersections of line segments

Input: n line segments given by pairs of their endpoints (in general position)
Output: all intersections (which pairs of line segments intersect and where)
Can be easily done in $O(n^2)$ but we want $O((n+p)\log n)$ where p is the # of intersections

Plane sweeping algorithm which uses a calendar of events

- Events in the calendar C are all line segment endpoints and some intersections
- The line sweeps from top to bottom jumping from event to event in the calendar
- We keep a profile P = a sequence of line segments intersected by the sweep line
- C contains only intersections of neighboring line segments in P (n-1 such pairs)

Initialize P as an empty set (it will be ordered by x-coordinates of the intersections)
Insert all line segment endpoints sorted by decreasing y-coordinate into C
While C is not empty do (let x be the first event in C)
    if x is a beginning of a line segment s then insert s into P
    if x is an end of a line segment s then delete s from P
    if x is an intersection of s_1 and s_2 then output x and switch s_1 and s_2 in P
    recompute planned intersection events around the current updates in P
        (removes at most 2 events and adds at most 2 events)

Data structures: balanced BSTs for both C and P (supports all operations in O(logn))
For C (has at most 3n events at any time) we need: insert and delete
For P (contains links to line segments) we need: insert, delete, next, previous
Time complexity: O(logn) per event O((n+p)logn) in total

Classes P and NP, polynomial transformations, NP-completeness

Task: for a given input find an output structure with given properties

Examples:

• for a given input graph find a cycle in it

• for two given input matrices compute their product

Optimization task: for a given input find an optimal (typically minimum or maximum) output structure with given properties

Examples:

• for a given input (undirected) graph find the biggest complete subgraph (clique)

• for a given set of jobs find the shortest feasible schedule

Decision problem: for a given input answer YES/NO

Examples:

• does a given input graph contain a (Hamiltonian) cycle?

• is a given input square matrix regular?

From now on we shall restrict ourselves to decision problems only, which can be done (more or less) without loss of generality – in the sense that for (almost) every (optimization) task there exists a decision problem which is "equally hard to solve".

Definition (vague): Class P is a set of decision problems. A decision problem X is in P if there exists a deterministic sequential algorithm, which for any input instance y of X correctly answers YES/NO (solves y) in a **p**olynomial time (with respect to the size of y).

- Is a given input directed graph strongly connected?
- Does a given input undirected graph contain a triangle? (clique of size three)
- Is a given input matrix regular?

Nondeterministic algorithm = algorithm, which can choose in each of its steps from several options

Nondeterministic algorithm solves a given decision problem $\Leftrightarrow$ there exists a sequence of choices leading to a YES answer for every YES input instance of the problem and no sequence of choices leads to a YES answer for a NO input instance of the problem.

Definition (vague): Class NP is a set of decision problems. A decision problem X is in NP if there exists a **n**ondeterministic sequential algorithm, which for any input instance y of the problem X solves y in a **p**olynomial time (with respect to the size of y).

A different model of a nondeterministic algorithm: do the choices in advance (and record them in memory) and then run the original nondeterministic algorithm in a deterministic manner using the recorded advice (certificate).

Alternative definition (vague again): A decision problem X is in class NP, if there exists a (polynomially large) certificate for each YES instance y of the problem X, using which it can be deterministically checked in polynomial time, that the answer to y is YES.

Examples of decision problems from the class NP:

- CLIQUE (complete subgraph) input: An undirected graph G and a number k.

  Question: Does there exist in G a complete subgraph on at least k vertices?

- HC (Hamiltonian cycle) input: An undirected graph G.

  Question : Does there exist a Hamiltonian cycle in G?

- TSP (traveling salesman) input: A weighted complete undirected graph G, a number k.

  Question : Does there exist a Hamiltonian cycle in G of total weight at most k?

- SS (subset sum) input: Natural numbers $a_1, a_2, \ldots, a_n, b$.

  Question : Does there exists a subset of $a_1, a_2, \ldots, a_n$ which sums up to b?

- SPM (scheduling on parallel machines) input: # of jobs, their lengths, # of machines, k.

  Question : Does there exists a feasible schedule of length at most k?

- SAT (CNF satisfiability) input: A CNF F on variables $x_1, x_2, \ldots, x_n$.

  Question : Does there exists a satisfying assignment for F?

We shall show, that HC → TSP, SS→ SPM, and SAT → 3-SAT → CLIQUE → SAT, where A → B means, that if there exists a (deterministic) polynomial algorithm solving B, then there also exists a (deterministic) polynomial algorithm solving A, or in other words solving B is at least as “hard” (up to a polynomial) as solving A.

Transformations (reductions) among decision problems

Let A,B be two decision problems. We say that A is polynomially reducible to B, if there exists a mapping f from the set of input instances of problem A into the set of input instances of problem B with the following properties:

1. Let X be an input instance of problem A and Y an input instance of problem B such that Y = f(X). Then X is a YES instance of A if and only if Y is a YES instance of B.

2. Let X be an input instance of problem A. Then f(X) is an input instance of problem B which can be (deterministicly, sequentially) constructed from X in a polynomial time.

Remark: Property 2. implies, that size of f(X) is polynomial with respect to size of X.

NP-completeness

Definition: Problem B is NP-hard if and only if every problem A from NP is polynomially reducible to B.

Definition: Problem B je NP-complete if 1) belongs to NP and 2) is NP-hard.

Corollary 1: If A is NP-hard and polynomially reducible to B, then B is also NP-hard.

Corollary 2: If there exists a polynomial time algorithm for some NP-hard problem, then there exist polynomial time algorithms for all problems in the class NP.

Theorem (Cook-Levin 1971): There exists an NP-complete problem (proved for SAT).

Pseudopolynomial algorithms

Algorithm for SS: assume $a_1 \geq \ldots \geq a_n$ holds, and A is an array of length b.

```
for j := 1 to b do {A[j] := 0; a_0 := b+1};
for i := 1 to n do
        A[a_i] := 1;
        for j := b downto a_{i-1} do if (A[j] = 1) and (j+a_i ≤ b) then A[j+a_i] := 1;
SP := (A[b] = 1).
```

Fact: After i-th pass through the main loop the array A contains ones exactly at those indices, which correspond to sums of all nonempty subsets of the set $\{a_1, \ldots, a_i\}$, which are less or equal to b.

Time complexity: O(nb), which is

- exponential time complexity w.r.t. binary (but also ternary, decadic, …) coding of input data, but
- polynomial time complexity w.r.t. unary coding of input data.

Algorithms with these properties are called pseudopolynomial. Formal definition follows on the next slide.

Consider a decision problem Q and its instance X. Let us define:

code(X) = the lenght of the coding (number of bits) of X in binary (or „higher“) coding

max(X) = biggest number in X (the number NOT the length of its binary coding)

Definition: An algorithm solving Q is called pseudopolynomial, if its time complexity when run on input X is upper bounded by a polynomial in variables code(X) and max(X).

Remark: Every polynomial algorithm is of course also pseudopolynomial.

Observation: If Q is such that for each instance X of Q we have $\max(X) \leq p(code(X))$ for some (fixed) polynomial p, then there is no difference between pseudopolynomial and polynomial algorithms for Q. Decision problems where this does NOT happen are called number problems.

Definition: Decision problem Q is called a number problem, if there exists no polynomial p such that for every instance X of Q the inequality $\max(X) \leq p(code(X))$ holds.

Theorem: Let Q be an NP-hard problem, which is not a number problem. Then Q cannot be solved by a pseudopolynomial algorithm unless ***P = NP***.

Question: Is every number problem solvable by some pseudopolynomial algorithm?

Answer: NO (and typical examples of such problems are called strongly NP-hard)

Strongly NP-complete problems

Let Q be a decision problem and p a polynomal. We denote the set of instances of Q for which $\max(X) \leq p(\text{code}(X))$ holds (i.e. a subproblem of Q) by Q_p, that is

$$Q_p = \{X \in Q \mid \max(X) \leq p(\text{code}(X))\}$$

Theorem: Let A be a pseudopolynomial algorithm solving Q. Then A is a polynomial algorithm solving Q_p for every polynomal p.

Definition: Decision problem Q is called strongly NP-complete, if $Q \in$ ***NP*** and there exists a polynomial p such that the subproblem Q_p is NP-complete.

Theorem: Let Q be a strongly NP-complete problem. Then Q cannot be solved by a pseudopolynomial algorithm unless ***P = NP***.

Examples of number problems which are strongly NP-complete:

Travelling salesman problem (TSP) :

- this is a number problem (edge weights may be arbitrarily large)
- strongly NP-complete (it stays NP-complete if the weights are bounded by a constant)

3-partition (3-P) – an example of a „pure“ number problem:

Input: $a_1, \ldots, a_{3m}, b \in N$, for which $\forall j : \frac{1}{4} b < a_j < \frac{1}{2} b$ and $\Sigma_{j=1}^{3m} a_j = mb$ holds.

Question: $\exists S_1, \ldots, S_m$ a disjunct partition of the set $\{1, \ldots, 3m\}$ such that $\forall i : \Sigma_{j \in S_i} a_j = b$?

Approximation algorithms

Approximation algorithms are typically used to solve "large" instances of NP-hard optimization problems, for which finding an optimal solution is "hopeless", i.e. too time consuming ("small" instances can be solved optimally in exponential time by "brute force"). An approximation algorithm has the following three properties:

1. Returns a suboptimal solution.

2. Gives an estimate of the quality of the obtained solution compared to the optimum

3. Runs in a polynomial time with respect to the size of the input.

Error estimation

Notation: OPT = optimal solution

APP = solution given by the approximation algorithm

f(Z) = value of solution Z (always assumed to be nonnegative)

Definition: Approximation algorithm A solves an optimization problem X with a ratio error r(n), if for every instance of size n of problem X we have

$$\max \{ f(APROX) / f(OPT) , f(OPT) / f(APROX) \} \leq r(n).$$

Definition: Approximation algorithm A solves an optimization problem X with a relative error e(n), if for every instance of size n of problem X we have

$$|f(APROX) - f(OPT)| / f(OPT) \leq e(n).$$

Example of a maximization problem (optimization version of CLIQUE):

For a given undirected graph find a biggest (in number of verices) clique in the graph.

The approximation algorithm should provide a guarantee of the type

$$f(\text{APROX}) \geq \tfrac{3}{4} f(\text{OPT}),$$

where f(X) is in this case the number of vertices (i.e. the clique size) in solution X.

Example of a minimization problem (optimization version of SPM):

For given jobs with given lengths and a given number of machines find a shortest feasible schedule.

The approximation algorithm should provide a guarantee of the type

$$f(\text{APROX}) \leq 2 f(\text{OPT}),$$

where f(X) is in this case the length of the schedule in solution X.

Examples of approximation algorithms

Scheduling on parallel machines (optimization) problem:

Input: Lengths of jobs $x_1, x_2, \ldots, x_n$, number of machines m.

Task: Find the shortest feasible schedule of the given jobs on the given number of machines.

Naive approximation algorithm QUEUE: takes jobs one by one according to their numbers and places each job on a machine which is free at the earliest time.

Notation: OPT = optimal schedule, Q = schedule produced by algortithm QUEUE, f(OPT) = o, f(Q) = q

Theorem: $q \leq ((2m - 1) / m)\, o$ and this bound is tight (cannot be improved).

Corollary: QUEUE has a ratio error $r(n) \leq 2$ (and it is tight for constant bounds).

Proof:

1. Tightness: For each m we shall construct an instance for which equality holds in the statement of the Theorem. Such instance is defined as follows:

$x_1 = x_2 = \ldots = x_{m-1} = m-1$ ($m-1$ jobs of length $m-1$)

$x_m = x_{m+1} = \ldots = x_{2m-2} = 1$ ($m-1$ jobs of length 1)

$x_{2m-1} = m$ (1 job of length m)

2. Correctness: Let j be a job which terminates last in schedule Q at time q and let t be the time when j starts in Q. Then no machine has any idle time before time t and moreover it is easy to see that $m\, q \leq (2m - 1)\, o$.

Less naive approximation algorithm SORTED QUEUE: works just like QUEUE, except that it starts by sorting jobs non-increasingly according to their lengths.

Notation: OPT = optimal schedule, SQ = schedule produced by SORTED QUEUE,
f(OPT) = o, f(SQ) = u

Theorem: $u \leq ((4m - 1) / 3m)o$ and this bound is tight (cannot be improved).

Corollary: SORTED QUEUE has a ratio error $r(n) \leq 4/3$ (again tight for constant bounds).

Tightness: For each m we shall construct an instance for which equality holds in the statement of the Theorem. Such instance is defined as follows:

$x_1 = x_2 = 2m-1$ (2 jobs of length $2m-1$)

$x_3 = x_4 = 2m-2$ (2 jobs of length $2m-2$)

$x_{2m-3} = x_{2m-2} = m+1$ (2 jobs of length $m+1$)

$x_{2m-1} = x_{2m} = x_{2m+1} = m$ (3 jobs of length m)

Lemma: If all job lengths satisfy the inequality $x_i \geq 1/3\ o$ then $u = o$.

Correctness: Let j be a job which terminates last in schedule SQ at time u. If $x_j > 1/3\ o$ then we use the above Lemma, in the other case we basically repeat the proof for algorithm QUEUE.

Vertex cover (optimization) problem:

Input: Undirected graph $G = (V,E)$.

Task: Find a vertex cover of a minimum size, i.e. $V' \subseteq V$ such that for every $(u,v) \in E$ there is $u \in V'$ or $v \in V'$ (or both), and moreover V' has a minimum possible cardinality.

Algorithm A: repeatedly select a vertex of the highest degree put it in the vertex cover and delete it form the graph together with all edges which it covers until no edge is left.

Does A have a constant ratio error?

Algorithm B: repeatedly select an arbitrary edge (u,v) put both u and v in the vertex cover and delete both u and v form the graph together with all edges which they cover until no edge is left.

Does B have a constant ratio error?

Traveling salesman problem:

Input: Complete weighted undirected graph $G = (V,E)$

with weights given by $c : E \rightarrow Z^+ \cup \{0\}$

Task: Find in G a Hamiltonian cycle of minimal total weight.

1. TSP with the triangular inequality

Is it even NP-hard??

Algorithm C:

a) Find a minimum spanning tree of the graph.

b) Select an arbitrary vertex and run a DFS on the tree numbering the vertices in the preorder manner

c) select the Hamiltonian cycle given by the permutation of vertices computed in b)

Theorem: Algorithm C has a ratio error $r(n) \leq 2$.

2. TSP without the triangular inequality

Theorem: Let $R \geq 1$ be an arbitrary constant. Then there exists no polynomial time approximation algorithm with ratio error R which solves general TSP, unless P = NP.

Corollary: There exist NP-hard optimization problems for which no polynomial time constant ratio error approximation algorithms can exists (unless P = NP).

Probabilistic (randomized) algorithms

Probabilistic algorithm makes (contrary to a deterministic one) random steps, e.g. uses values generated by a random number generator for some steps – thus two subsequent runs of the same probabilistic algorithm on the same data differ (with high probability).

There are many types of probabilistic algorithms, we shall mention only two of them, namely Las Vegas algorithms and Monte Carlo algorithms.

Las Vegas algorithms

The result is always correct, randomness influences only the run time, i.e. determines the path which the algorithm uses to arrive to the correct result.

Example: randomized QuickSort – the pivot is selected randomly from the current subsequence on every level of recursion, which brings the following advantages

• gives good average running time (i.e. O(n log n)) even in case that the input data are not random permutations – no input is a priori bad (of course there exist bad inputs for every deterministic rule for choosing the pivot)

• can be run in parallel in several copies on the same data, the result is obtained from the copy that finishes the computation first (this would make no sense for a deterministic version of QuickSort)

Monte Carlo algorithms

Randomness influences both running time and the correctness of the result: algorithm can make an error but the error can be only one-sided (we assume a YES/NO answer) and has a bounded probability

Example: Rabin-Miller algorithm for primality testing

Task: for a given number n (quickly) decide whether n is a prime number

A little bit of theory (Little) Fermat's theorem (without a proof):

Let p be a prime number. Then $\forall\ k \in \{1,2, \ldots ,p-1\}$ we get $k^{p-1} \equiv 1 \pmod{p}$.

Idea: If n is NOT a prime, we may try to (randomly) find a "witness" k, violating the rule $k^{n-1} \equiv 1 \pmod{n}$, which "witnesses" that n is a composite number (is not a prime)

Problem: For some composite numbers there are too few witnesses, and thus the probability that a witness will be selected at random is too small.

Definition: Let T be a set of pairs of natural numbers, where $(k,n) \in T$ if $0 < k < n$ and at least one of the following two conditions is satisfied:

1. the congruence $k^{n-1} \equiv 1 \pmod{n}$ is not valid,

2. exists i such that $m = (n-1) / 2^i$ is a natural number and $1 < LCD(k^{m-1}-1, n) < n$ holds

Theorem: Number n is composite if and only if there exists number k such that $(k,n) \in T$. Moreover, if n is composite then $(k,n) \in T$ holds for at least $(n-1)/2$ numbers k.

```
Rabin-Miller(n);
for i:=1 to number-of-trials do
   k_i := randomly selected number from the interval [1,n-1];
   if (k_i,n) ∈ T then Report (n is composite);
                       Abort;
Report (n is prime)
```

If Rabin-Miller(n) decides, that n is composite, then it is a correct result for sure (a witness has been found), if Rabin-Miller(n) decides, that n is prime, then it may be an error, but only in case that all selected k_i were „non-witnesses“ for a composite n, which may happen with a probability of at most

$$P(\text{error}) \leq (1/2)^{\text{number-of-trials}}$$

if the individual k_i are selected independently.

Properties of the algorithm:

• by increasing the number of trials (the number of tested k_i) an arbitrarily small (preselected) probability of error can be achieved

• the individual trials (tests for different k_i) may be performed in parallel

Time complexity:

each trial takes only polynomial time with respect to the length of the encoding of n, the hard part is showing that testing $(k_i,n) \in T$ takes time polynomial in log n, which is not trivial at all (deeper knowledge of number theory is necessary)

Public key cryptography

- every participant X has his/her public key PX and secret (private) key SX
- SX is known only to X, public key PX can be given by X to all parties communicating with X, or it can be even published in a publicly available list of keys (e.g. on the web)
- both keys specify functions which may be applied to any message: if D is a set of finite bit strings (set of all possible messages), then both functions must be injective functions mapping D on D (i.e. both specify permutations of the set D)
- the function specified by the secret key SX is denoted SX(), the function specified by the public key PX is denoted PX(), and we assume that both functions are efficiently enumerable if we know the corresponding key
- functions SX() and PX() must form a mutually inverse pair of functions: for every message (finite bit string) M we must have SX(PX(M)) = M and PX(SX(M)) = M.
- the security of encryption is based on the fact that nobody except of X is capable of computing SX(M) for any message M in a „reasonable“ time, which means that
    1. participant X must hold secret key SX absolutely safe from any leakage
    2. function SX() must not be efficiently computable based on the knowledge of PX (and the ability to efficiently enumerate function PX()), what is the main difficulty when designing any public key cryptography method or system

Let us assume that we have 2 participants: A (Alice) a B (Bob) with keys SA, PA, SB, PB

Sending an encrypted message and its decrypting

Bob wants to send an encrypted message M to Alice:

• Bob gets Alice's public key PA (directly from Alice or from a public list of keys)

• Bob computes the encrypted text C = PA(M) and sends it to Alice

• Alice applies her secret key SA on C, i.e. computes SA(C) = SA(PA(M)) = M

• If C is read by someone else than Alice, he/she has no chance to obtain M, because he/she cannot efficiently compute SA(C).

Sending an authenticated and signed (non-encrypted) message

Alice wants to answer to Bob in a way, that Bob can be sure that the answer Q comes from Alice and that the text of the answer was not modified by someone else:

• Alice computes her digital signature q for message Q using her private key, i.e. computes q = SA(Q) and sends the pair (Q,q) to Bob – message Q is transmitted openly

• Bob computes PA(q) = PA(SA(Q)) = Q and compares the result with message Q

• If both messages match, Bob can be sure that the message comes form Alice and it was not modified along the way

• If both messages differ, than either the signature q is forged (was not created using function SA()) or the signature is correct but the open message Q was modified

Sending an authenticated and signed encrypted message

Alice wants to send message M to Bob, so that Bob is sure that M comes from Alice and that the text of M was not modified. Moreover, Alice wants that only Bob can read M.

• Alice computes her digital signature for M, i.e. computes m = SA(M)

• Alice encrypts the pair (M,m) using Bob's public key, i.e. computes encrypted text C = PB(M,m) and sends C to Bob

• Bob decrypts C using his secret key, i.e. computes SB(C) = SB(PB(M,m)) = (M,m)

• Bob verifies validity of Alice's signature and authenticity of M using Alice's public key, i.e. computes PA(m) and compares it with M – if they differ Bob knows, that either C was modified during transmission (deliberately or by an error) or C does not come from Alice.

Hybrid encryption

If message M which Bob wants to send to Alice is too long, and thus the computation of C = PA(M) and subsequently of M = SA(C) would be too time consuming, it is possible to combine public key cryptography with some symmetric cipher K, which works quickly:

• Bob computes C = K(M), which is again a long bit string, additionally computes PA(K), which is a short string (compared to M and PA(M)) and sends (C,PA(K)) to Alice

• Alice decrypts PA(K) using her secret key SA and thus obtains K, using which it decrypts C and obtains M

Hybrid authentication and signing

If M is too long, it is also very time consuming to compute digital signature m = SA(M). We may circumvent this difficulty by using a (publicly known) hash function h satisfying:

1. Even if M is very long h(M) can be computed quickly, typically h(M) is a very short (e.g. 128 bit) fingerprint of message M.

2. It is computationally difficult (impossible within a reasonable time) to find for M a different message Q such that h(M) = h(Q) holds

If Alice wants to sign a long message sent to Bob, she may proceed as follows:

• Alice computes the fingerprint h(M) of message M, turns it into the digital signature m = SA(h(M)), and sends the pair (M,m) to Bob

• Bob also computes the fingerprint h(M) of M, which he then compares with the decrypted Alice's signature PA(m) = PA(SA(h(M))). If M was modified along the way, the strings will not match, as it is impossible to modify M without changing its fingerprint (thanks to property 2). Nobody but Alice can compute m, so M is sure to come from her.

Certification authorities

If Bob obtains Alice's public key from a public list (or Alice sends it to Bob via internet), how can be Bob sure that the key is not forged? If the key is forged and the subsequent messages are modified or forged by the same person who forged the public key, it is then impossible to detect that something is wrong, because that person also has the corresponding secret key, and so all strings match as expected. Solution:

• There exist certification authority Z, whose public key PZ is installed on each participant's HW (including Bob's) – e.g. comes on the CD with the cryptographic SW

• Alice then has a certificate C (issued by authority Z) of the form „Alice's key is PB" signed by authority Z, i.e. a pair (C, SZ(C)) – Alice may have this from the cryptographic SW installation, or she may obtain this pair by some other secure method

• Alice attaches this pair (C, SZ(C)) to every signed message, and so Bob (and everybody else) can verify using public key PZ, that C was indeed issued by authority Z, and that therefore PB is really Alice's public key

RSA (Rivest, Shamir, Adelman) cipher

to explain RSA we need a number of concepts and results form number theory

Definition: The greatest common divisor of two natural numbers a,b is the largest natural number which divides both a and b. We shall denote it by GCD(a,b).

Theorem: Let a,b be natural numbers. Then GCD(a,b) is the smallest positive element in the set $L = \{ax + by \mid x,y \in \mathbf{Z}\}$.

Corollary: Let a,b be natural numbers. If d is a natural number which divides both a and b, then d divides also GCD(a,b).

Theorem: Let a,b be natural numbers, where b>0. Then GCD(a,b) = GCD(b, a mod b).

EUCLID(a,b)

if $b=0$ then Return(a)

else Return(EUCLID(b, a mod b))

Lemma: Let $a > b \geq 0$ such that EUCLID(a,b) performs $k \geq 1$ recursive calls. Then $a \geq F(k+2)$ and $b \geq F(k+1)$, where $F(i)$ is the i-th Fibonacci number.

Corollary (Lamé's theorem): Let $a > b \geq 0$ and $F(k) \leq b < F(k+1)$. Then EUCLID(a,b) performs at most $k - 1$ recursive calls.

Theorem (w/o proof – see AVL trees): $F(k) = \Theta(\varphi^k)$, where $\varphi = (1+\sqrt{5})/2$ („golden ratio").

Corollary: Let $a > b \geq 0$. Then EUCLID(a,b) performs $O(\log b)$ recursive calls.

Observation: If a,b are two binary numbers with at most t-bits, then EUCLID(a,b) performs $O(t)$ recursive calls and in each of them $O(1)$ arithmetic operations on (at most) t-bit numbers, i.e. $O(t^3)$ bit operations, if we assume that each arithmetic operation on (at most) t-bit numbers requires $O(t^2)$ bit operations (which can be shown easily). Thus EUCLID is a polynomial time algorithm with respect to the size of its input.

Euclid's algorithm can be easily extended so that it also computes the coefficients x,y, for which $GCD(a,b) = ax + by$.

```
EXTENDED-EUCLID(a,b)
if b=0 then Return(a,1,0)
        else (d',x',y') := EXTENDED-EUCLID(b, a mod b);
             (d,x,y) := (d',y',x' - ⌊a/b⌋ y');
             Return(d,x,y)
```

Theorem (w/o proof): Let $n > 1$ and $a < n$ be two relative prime natural numbers. Then the equation $ax \equiv 1 \pmod{n}$ has exactly one solution $0 < x < n$ (and if a,n are not relative primes it has no solution).

Definition: The solution x of the equation $ax \equiv 1 \pmod{n}$ is denoted by $(a^{-1} \bmod n)$ and it is called the multiplicative inverse of a modulo n (it exists only if a,n are relative primes).

Observation: $(a^{-1} \bmod n)$ can be easily obtained using the extended Euclid's algorithm.

Theorem (a special corollary of the "Chinese remainder theorem"– w/o proof):
Let a,b be relative prime natural numbers. Then for every pair of natural numbers x,y we have: $x \equiv y \pmod{ab}$ if and only if $x \equiv y \pmod{a}$ and $x \equiv y \pmod{b}$.

Theorem (Fermat's little):
Let p be a prime number. Then $\forall\ k \in \{1,2, \ldots ,p-1\}$ we get $k^{p-1} \equiv 1 \pmod{p}$.

Now we have all we need to define and explain RSA:

1. Pick two large primes p and q at random (e.g. consisting of 200 binary digits each).
2. Compute n = pq (in this case n consists of approx. 400 binary digits).
3. Select a small odd number e, which is relative prime to (p-1)(q-1).
4. Compute the multiplicative inverse d of e modulo (p-1)(q-1).
5. Publish (e,n) as the public RSA key and save (d,n) as the secret RSA key.

Theorem (RSA correctness): Functions $P(M) = M^e \bmod n$ and $S(M) = M^d \bmod n$ define a pair of inverse functions on the set of all messages, i.e. on the set of all numbers in $Z_n = \{0,1, \ldots ,n-1\}$.

Why is RSA safe?

Nobody is (currently) able to compute d based on the knowledge of (e,n) without knowing the factors n = pq and thus also the number (p-1)(q-1). Factorization of large natural numbers is currently a computationally intractable problem.

How fast is RSA?

The speed of computing P(M) and S(M) depends on how fast a remainder modulo n can be computed when taking powers of an integer, i.e. how fast we can compute $a^b \bmod n$.

POWER (a,b,n) {where the binary digits in b are $<b_k, \ldots, b_0>$ }

c := 1; d := a mod n;

for i := k-1 downto 0 do

  c := 2·c;

  d := (d·d) mod n;

  if b_i = 1 then c := c + 1;

                d := (d·a) mod n;

Return(d)

Time complexity: If a,b,n are binary numbers with at most t-bits, then POWER (a,b,n) performs O(t) arithmetic operations on (at most) t-bit numbers, i.e. $O(t^3)$ bit operations, if we assume that each arithmetic operation on (at most) t-bit numbers requires $O(t^2)$ bit operations (which can be shown easily).