Algorithms and data structures I

TIN060

Ondřej Čepek

Sylabus

1. Asymptotic notation
2. Graph algorithms
3. Extremal paths in graphs
4. Minimum spanning trees
5. Tree data structures
6. „Divide and conquer“ algorithms
7. Sorting
8. Hashing
9. Linear algebra algorithms

How to compare algorithms (doing the same task)?

Time complexity of algorithms
Space complexity of algorithms
} both depend on the "size" of input data

How to measure the "size" of input data?

rigorously: number of bits necessary to encode the input data

---

Example: input consists of (natural) numbers $a_1, a_2, \dots, a_n$ which are to be sorted

the size of data D coded in binary is $|D| = d = \lceil \log_2 a_1 \rceil + \dots + \lceil \log_2 a_n \rceil$

Time complexity = function f(d) specifying the number of "steps" the algorithm takes depending on the size of the input data

intuitively: the exact description of function f (constants, coefficients) is not so much important, what matters is to which "class" f belongs (linear, quadratic, cubic, exponential, …)

Example:	$f(d) = ad + b$	linear algorithm
	$f(\|D\|) = ad^2 + bd + c$	quadratic algorithm
	$f(\|D\|) = k2^d$	exponential algorithm

What is a "step" of an algorithm?

rigorously: an operation of a given abstract machine (typically a Turing machine)

simplification (we shall use): step = operation executable in constant time, i.e. in time independent of the size of input data

- arithmetic operations (addition, subtraction, multiplication, … for numbers stored in some standard numerical data types – integer, longint, real, float)
- comparison of (two) values (numbers, characters)
- assignment (only for simple data types, not for arrays …)

$\Rightarrow$ this also simplifies the way the size of input data is measured (all numbers have a fixed size in memory)

Example: sort numbers $a_1, a_2, \ldots, a_n$ $\Rightarrow$ the size of the input data is $|D| = n$

This simplification is OK when comparing the quality of different algorithms, however it can lead to an error when deciding to which complexity class the given problem belongs (numbers bounded by a constant versus unbounded numbers).

Why is time complexity of algorithms important?

(many people – even some programmers – think that current computers are so fast they solve everything in a few seconds anyway so why bother…)

Time it takes to execute f(n) operations (= time needed for the run of the algorithm) for input data of size n based on the assumption that the hardware used is capable of executing 1 million operations per second

	n						
f(n)	20	40	60	80	100	500	1000
n	20μs	40μs	60μs	80μs	0.1ms	0.5ms	1ms
n log n	86μs	0.2ms	0.35ms	0.5ms	0.7ms	4.5ms	10ms
n^2	0.4ms	1.6ms	3.6ms	6.4ms	10ms	0.25s	1s
n^3	8ms	64ms	0.22s	0.5s	1s	125s	17min
2^n	1s	12days	36000y				
n!	77000y						

The growth of a “manageable” size of input data based on hardware speed-up (assuming that manageable size of input data for “current” hardware is x)

Manageable size of input data = maximum size for which the algorithm terminates in some specified time t, where t = maximum time the user is willing to wait for results

	Hardware speed-up			
f(n)	original	10 times	100 times	1000 times
n	x	10x	100x	1000x
n log n	x	7.02x	53.56x	431.5x
n^2	x	3.16x	10x	31.62x
n^3	x	2.15x	4.64x	10x
2^n	x	x+3	x+6	x+9

Asymptotic notation

Intuitively: allows to disregard the behavior on “small” data and to disregard additive and multiplicative constants (coefficients) in the precise complexity functions, allows to capture the important features needed to properly classify complexity functions

Rigorously:

$f(n)$ is asymptotically less or equal than $g(n)$, denoted by $f(n) \in O(g(n))$, if

$$\exists c>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq f(n) \leq c\, g(n)$$

$f(n)$ is asymptotically greater or equal than $g(n)$, denoted by $f(n) \in \Omega(g(n))$, if

$$\exists c>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq c\, g(n) \leq f(n)$$

$f(n)$ is asymptotically same as $g(n)$, denoted by $f(n) \in \Theta(g(n))$, if

$$\exists c>0\ \exists d>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq c\, g(n) \leq f(n) \leq d\, g(n)$$

$f(n)$ is asymptotically strictly smaller than $g(n)$, denoted by $f(n) \in o(g(n))$, if

$$\forall c>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq f(n) \leq c\, g(n)$$

$f(n)$ is asymptotically strictly greater than $g(n)$, denoted by $f(n) \in \omega(g(n))$, if

$$\forall c>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq c\, g(n) \leq f(n)$$

Elementary graph algorithms

Notation:graph G=(V,E), V vertices (nodes), |V|=n, E edges (arcs), |E|=m

undirected graph: edge = unordered pair of vertices

directed graph : arc = ordered pair of nodes

Graph representations: adjacency matrix $\Theta(n^2)$

neighbor lists $\Theta(n+m)$

Search on graphs

BFS – breadth first search

```
BFS(G,s)
for each u∈V do begin color[u]:=white; d[u]:=Maxint; p[u]:=NIL end;
color[s]:=grey; d[s]:=0; Queue:={s};
while Queue nonempty do
begin
  u:=first in Queue;
  for each v ∈ Neighbor(u) do if color[v]=white then
          begin color[v]:=grey; d[v]:=d[u]+1; p[v]:=u; insert v at the end of Queue end;
  color[u]:=black; delete u from Queue
end
```

BFS remarks:

1. Searches the graph level by level, where each level is defined as a set of vertices with the same distance (number of edges on the shortest path) form vertex s
2. Subsequently visits all vertices reachable from s and creates a shortest path tree
3. Is a basis for more sophisticated algorithms such as Dijkstra's algorithm (single source shortest paths in a graph with nonnegative weights on all edges) or Prim's algorithm (minimum spanning tree)
4. Works also on undirected graphs (no change is necessary)
5. When the input graph is given by neighbor lists, BFS runs in $\Theta(n+m)$ time

BFS use (example): connectivity testing for an undirected graph

- select a vertex at random and run BFS from it
- graph is not connected $\Leftrightarrow$ some vertex remains white when BFS terminates
- counting the number of connected components: repeated runs of BFS from a randomly selected white vertex (while a white vertex exists)
- again runs in $\Theta(n+m)$ time

DFS – depth first search

• undirected version – the main difference compared to BFS is that the active (grey) vertices are not stored in a queue but on a stack (either explicitly created by DFS or implicitly created by recursion – if DFS is a recursive routine)

• directed version – we shall study it in detail, let us assume that the input graph is represented by neighbor lists

```
DFS(G)
begin    for i:=1 to n do color[i]:=white;
         time:=0;
         for i:=1 to n do if color[i]=white then VISIT(i)
end;

VISIT(i) {simple version}
begin    color[i]:=grey;
         time:=time+1;
         d[i]:=time;
         for each j ∈ Neighbor(i) do if color[j]=white then VISIT(j);
         color[i]:=black;
         time:=time+1;
         f[i]:=time
end;
```

Edge classification for DFS on undirected graphs - (i,j) is a:

tree edge	j is discovered from i	j is **white** when (i,j) is scanned
back edge	j is a predecessor of i in a DFS tree	j is **grey** when (i,j) is scanned
forward edge	i is a predecessor of j in a DFS tree (but not a direct parent)	j is **black** when (i,j) is scanned and moreover $d(i) < d(j)$
cross edge	otherwise (none of the above three cases has occurred)	j is **black** when (i,j) is scanned and moreover $d(i) > d(j)$

Properties of DFS

1. Tree edges form a directed forest (DFS forest = set of DFS trees)
2. Vertex j is a successor of vertex i in a DFS tree $\Leftrightarrow$ There exists a path from i to j formed only from white vertices at time d(i)
3. Intervals [d(i), f(i)] form a "legal parenthesis structure", i.e. for each pair $i \neq j$:
   - either $[d(j), f(j)] \cap [d(i), f(i)] = \varnothing$
   - or $[d(i), f(i)] \subset [d(j), f(j)]$ and i is a successor of j in a DFS tree
   - or $[d(j), f(j)] \subset [d(i), f(i)]$ and j is a successor of i in a DFS tree

   Corollary: j is a successor of i in a DFS tree $\Leftrightarrow$ $[d(j), f(j)] \subset [d(i), f(i)]$

```
VISIT(i) {full version}
begin     color[i]:=grey;
          time:=time+1;
          d[i]:=time;
          for each j ∈ Neighbor(i) do
          if color[j]=white
                    then      begin     VISIT(j);
                                        label (i,j) as a tree edge
                              end
                    else if color[j]=grey
                              then      begin     report cycle detection;
                                                  label (i,j) as a back edge
                                        end
                              else if d(i) < d(j)
                                        then label (i,j) as a forward edge
                                        else label (i,j) as a cross edge
          color[i]:=black;
          time:=time+1;
          f[i]:=time
end;
```

Time complexity: still linear in the size of input data (neighbor lists), i.e. $\Theta(n+m)$

Topological sort

Definition: Function $t : V \to \{1,2, \ldots ,n\}$ is a topological numbering of set V of vertices if and only if $t(i) < t(j)$ holds for every edge $(i,j) \in E$.

Observation: topological numbering exists only for acyclic graphs

Naïve algorithm:

1. Find a vertex with no outgoing edge and label it by the highest available number
2. Delete the numbered vertex from the graph, and if the graph is nonempty go to 1.

Time complexity: $\Theta(n(n+m))$

Sophisticated algorithm: a slight modification of DFS, runs in $\Theta(n+m)$ time

Lemma: G contains a cycle $\Leftrightarrow$ DFS(G) discovers a back edge

Theorem: The numbering (ordering) of vertices of an acyclic graph $G = (V,E)$ according to **decreasing** finish times $f(i)$, $j \in V$, is topological.

Transitive closure of a directed graph

Definition: Directed graph $G'=(V,E')$ is a transitive closure of a directed graph $G=(V,E)$ if and only if every pair of vertices $i,j \in V$ such that $i \neq j$ satisfies the following condition: there is a directed path from i to j in $G \Rightarrow (i,j) \in E'$

Transitive closure G' represented by an adjacency matrix = reachability matrix of G

Reachability matrix of G can be computed in $\Theta(n(n+m))$ time by n runs of DFS

Strongly connected components of a directed graph

Definition: Let $G=(V,E)$ be a directed graph. Set of vertices $K \subseteq V$ is called a strongly connected component of G if and only if

1. A directed path from i to j and a directed path from j to i exists in G for every pair of vertices $i,j \in K$ such that $i \neq j$.
2. There is no set of vertices L which is a strict superset of K and fulfils property 1.

Naïve algorithm: compute a transitive closure (reachability matrix) and then "read out" the strongly connected components from the matrix in $\Theta(n^2)$ time

Sophisticated algorithm:

Input: directed graph $G=(V,E)$ represented by neighbor lists

Phase1: DFS(G) supplemented by a construction of a linked list of vertices ordered by decreasing finish times

Phase2: construction of the transposed graph G^T

Phase3: $DFS(G^T)$ modified in such a way, that the vertices are selected in the main loop according to the order given by the linked list from Phase1 (instead of an order given by vertex numbers)

Output: DFS trees from Phase3 = strongly connected components of graph G

Definition: Let $G=(V,E)$ be a directed graph. Then graph $G^T=(V,E^T)$, where

$$(i,j) \in E^T \Leftrightarrow (j,i) \in E$$

is called the transposed graph of graph G.

Remark: Transposed graph can be constructed in $\Theta(n+m)$ time and thus the whole algorithm runs in $\Theta(n+m)$ time.

Lemma: Let $G=(V,E)$ be a directed graph and K be SCC in G. After the run of DFS(G):

1. Set K is a subset of vertices of a single DFS tree T
2. Set K constitutes a subtree of the given tree T

Extremal paths in (directed) graphs

extremal path = shortest (longest) path (depends on context)

unweighted graph: path length = number of edges on the path (shortest paths by BFS)

weighted graph: denote

$G = (V,E)$ directed graph

$w : E \rightarrow R$ weight function

if $p = (v_0, v_1, \ldots, v_k)$ is a directed path (vertex repetition is allowed), then

$$w(p) = w(v_0,v_1) + w(v_1,v_2) + \ldots + w(v_{k-1},v_k)$$

Definition (weight of the shortest path from u to v)

$$\delta(u,v) = \begin{cases} \min \{ w(p) \mid p \text{ is a path from } u \text{ to } v \} & \text{if } \exists \text{ path from } u \text{ to } v \\ \infty & \text{else} \end{cases}$$

Definition (shortest path from u to v)

A shortest path from u to v is an arbitrary path from u to v for which $w(p) = \delta(u,v)$

Negative cycles: negative cycle = a directed cycle with a total negative weight

- Graph w/o negative cycles: $\delta(u,v)$ defined for all pairs of vertices u and v, and for each pair at least one shortest path is simple (i.e. w/o any cycles)
- Graph with negative cycles : if $\exists$ a path from u to v containing a negative cycle, then we set $\delta(u,v) = -\infty$

Single source shortest paths

Task: for a fixed vertex $s \in V$ (the source) we want to compute $\delta(s,v)$ for all $v \in V \setminus \{s\}$

What we shall cover:

1. acyclic graph (and arbitrary weights) $\rightarrow$ DAG (critical path) algorithm
2. nonnegative weights (and any graph) $\rightarrow$ Dijkstra's algorithm
3. no restriction (any graph and weights) $\rightarrow$ Bellman-Ford algorithm

Trivial observations

Property 1 If $p=(v_0,v_1, \dots ,v_k)$ is a shortest path from v_0 to v_k, then $\forall i,j : 0 \le i \le j \le k$ holds, that the (sub)path $p_{ij}=(v_i, \dots ,v_j)$ is a shortest path from v_i to v_j.

Property 2 If p is a shortest path from s to v and the last edge on p is $(u,v) \in E$, then $\delta(s,v) = \delta(s,u) + w(u,v)$

Property 3 If $(u,v) \in E$ is an edge, then $\delta(s,v) \le \delta(s,u) + w(u,v)$.

Upper bounds for shortest paths

For every $v \in V$ we shall maintain a value $d(v)$, for which an invariant $d(v) \ge \delta(s,v)$ will hold throughout the run of each of the algorithms.

```
Initialize (G,s);
  for each v∈V(G) do
            begin     d(v) := ∞ ;
                      p(v) := NIL          {predecessor on the output shortest path}
            end;
  d(s) := 0.
```

After the initialization the algorithms repeatedly (in some order) perform relaxation (possible lowering) of the upper bounds:

```
Relax (u,v,w);
  if d(v) > d(u) + w(u,v) then
  begin   d(v) := d(u) + w(u,v);
          p(v) := u
  end.
```

Property 4 If $(u,v) \in E$ is an edge, then immediately after executing Relax (u,v,w) we have $d(v) \leq d(u) + w(u,v)$.

Property 5 If Initialize (G,s) was executed, then $\forall\ v \in V$ we have $d(v) \geq \delta(s,v)$ and this invariant is maintained over any sequence of relaxation steps on the edges of G. Moreover, once d(v) achieves its lower bound $\delta(s,v)$, it never changes.

Property 6 If there is no directed path from s to v, then after Initialize (G,s) we have $d(v) = \delta(s,v) = \infty$ and this equality is maintained over any sequence of relaxation steps.

Property 7 Let p be a shortest path from s to v and let (u,v) be the last edge on p. Suppose Initialize (G,s) is performed and then a sequence of relaxation steps that includes the call Relax (u,v,w) is executed. If $d(u) = \delta(s,u)$ at any time prior to the call of Relax (u,v,w), then $d(v) = \delta(s,v)$ at all times after the call.

DAG (directed acyclic graph) algorithm = critical path algorithm

```
DAG (G,w,s);
  topologically sort the vertices of graph G;
  Initialize (G,s);
  for each (u ∈ V(G) in topological order) do
        for each (v ∈ V(G) such that (u,v) ∈ E(G)) do Relax (u,v,w)
```

Theorem: Let $G=(V,E)$ be an acyclic weighted directed graph and $s \in V(G)$ an arbitrary vertex. Then at the termination of DAG (G,w,s) we have $d(v) = \delta(s,v)$ for each $v \in V(G)$.

Time complexity: DAG (G,w,s) runs in $\Theta(n+m)$ because

- topological sort using DFS takes $\Theta(n+m)$
- the body of the algorithm takes $\Theta(1)$ per vertex and $\Theta(1)$ per edge, e.g. $\Theta(n+m)$ in total

Application: Acyclic graph, where edges = activities and weights = activity durations. The graph expresses dependencies among activities – each directed path represents activities which must be performed sequentially. The task is to find a “critical path”, i.e. the longest path from the source to the sink in the graph. Any delay on such a path delays the whole project.

Solution: DAG can be modified by either

- reversing the signs of all weights or
- replacing ∞ by $-\infty$ in Initialize (G,s) and reversing the inequality in Relax (u,v,w)

Dijkstra's algorithm

- assumption: all edge weights are nonnegative ($\forall\ (u,v) \in E : w(u,v) \geq 0$)
- the set of vertices is split into two subsets during the run of the algorithm:

a) vertex v is in set S if its shortest distance from the source s was already computed, and therefore $d(v) = \delta(s,v)$ – after executing Initialize (G,s)) we have $S = \emptyset$

b) otherwise v belongs to set $Q = V \setminus S$ where Q is implemented as a data structure which supports efficient search and delete for vertex v with minimal $d(v)$

```
Dijkstra (G,w,s);
 Initialize(G,s);
  S := ∅; Q := V(G);
  while (Q ≠ ∅) do
     u := Extract-Min (Q);
     S := S ∪ {u};
     for each (v ∈ V(G) such that (u,v) ∈E(G)) do Relax (u,v,w)
```

Theorem: Let $G=(V,E)$ be a weighted directed graph with nonnegative edge weights and let $s \in V(G)$ be an arbitrary vertex. Then at the termination of Dijkstra (G,w,s) we have $d(v) = \delta(s,v)$ for each $v \in V(G)$.

Time complexity: $\Theta(n^2)$ if Q is implemented as an array

$\Theta((n+m)\log n)$ if Q is implemented as a binary heap

The Bellman-Ford algorithm

slower than Dijkstra but more general (can handle also negative edge weights)

```
Bellman-Ford (G,w,s);
 Initialize(G,s);
  for i:=1 to |V(G)|-1 do
    for each ((u,v) ∈E(G)) do Relax (u,v,w);
    {n-1 iterations, all edges are relaxed in an arbitrary order in every iteration}
  for each ((u,v) ∈E(G)) do if d(v) > d(u) + w(u,v) then return FALSE;
  {this signals an existence of a negative cycle reachable from the source}
  return TRUE
```

Time complexity: $\Theta(nm)$ (each iteration takes $\Theta(m)$)

Lemma: Let $G=(V,E)$ be a weighted directed graph with a weights function w and a source vertex s. If G contains no negative cycles reachable from s then at the termination of the algorithm we have $d(v) = \delta(s,v)$ for every vertex v reachable from s.

Theorem: Let $G=(V,E)$ be a weighted directed graph with a weights function w and a source vertex s. Then at the termination of Bellman-Ford (G,w,s) we have:

- if G contains a negative cycle reachable from s, then the algorithm returned FALSE
- if G contains no negative cycle reachable from s, then the algorithm returned TRUE and we have $d(v) = \delta(s,v)$ for every vertex $v \in V(G)$.

All pairs shortest paths

Task: compute $\delta(u,v)$ for every (ordered) pair u,v of vertices

Assumption: graph G is represented by an adjacency matrix W^G (if not, it can be generated from the neighbor lists in $\Theta(n^2)$ time), which is defined by

$$w_{uv} = \begin{cases} 0 & \text{if } u=v \\ w(u,v) & \text{if } u\neq v \text{ and } (u,v) \in E \\ \infty & \text{if } u\neq v \text{ and } (u,v) \notin E \end{cases}$$

Simplification: assume that G has no negative cycles (some edges may have negative weights as long as there is no negative cycle).

Aim: Generate the matrix D^G for which $d_{uv} = \delta(u,v)$.

Solution (using single-source algorithms): run a single source algorithm n times (each run produces one row of the matrix D^G)

- G acyclic: n times DAG $\rightarrow$ time complexity $\Theta(n(n+m))$
- nonnegative weights: n times Dijkstra $\rightarrow$ time complexity $\Theta(n^3)$ (array), $\Theta(n(n+m)\log n)$ (heap)
- no restricion: n times Bellman-Ford $\rightarrow$ time complexity $\Theta(n^2m)$ ($\Theta(n^4)$ for dense graphs)

We will try to improve the last value (the „no restriction“ case).

„Matrix multiplication“ algorithms

We will proceed by induction on the number of edges on the shortest path. Define

$d_{uv}(k)$ = minimal weight of a path from u to v, which contains at most k edges

1. $k = 1$: $d_{uv}(1) = w_{uv}$ when organized into a matrix, we have $D^G(1) = W^G$
2. $k - 1 \rightarrow k$: $d_{uv}(k) = \min\{d_{uv}(k-1), \min_{1 \le i \le n}\{d_{ui}(k-1) + w_{iv}\}\} = \min_{1 \le i \le n}\{d_{ui}(k-1) + w_{iv}\}$

The last equality follows from the fact that for $i = v$, we have $w_{v\,v} = 0$ and hence

$$D^G(k+1) = D^G(k) \otimes W^G$$

where the matrix multiplication $\otimes$ uses a „special“ scalar product in which:

- multiplication is replaced by addition and
- addition is replaced by minimum operator.

G contains no negative cycles $\rightarrow$ every pair has a simple shortest path (w/o cycles)

$\rightarrow$ each shortest path has at most $n - 1$ edges $\rightarrow$ $D^G(n-1) = D^G(n) = D^G(n+1) = \ldots = D^G$

Slow algorithm: compute $D^G(1), D^G(2), \ldots, D^G(n-1)$ by sequential multiplication

Time complexity: $n-2$ matrix multiplications of order $n \rightarrow (n-2)$ times $\Theta(n^3) \rightarrow \Theta(n^4)$

Faster algorithm: compute only powers of 2 using the associative property of $\otimes$.

Time complexity: $\log_2 n$ matrix multiplications $\rightarrow \log_2 n$ times $\Theta(n^3) \rightarrow \Theta(n^3 \log_2 n)$

The Floyd-Warshall algorithm

- similar „dynamic programming“ idea as matrix multiplication (building the solution iteratively from „partial solutions“)
- main difference: induction on the set of „allowed intermediate vertices“ on shortest paths, rather than on the number of edges on shortest paths

$d_{uv}(k)$ = minimum weight of a path from u to v with intermediate vertices only from the set of vertices $\{1, \dots, k\}$

1. k=0: $d_{uv}(0) = w_{uv}$ and therefore $D^G(0) = W^G$
2. k -1 $\rightarrow$ k: $d_{uv}(k) = \min\{d_{uv}(k-1), d_{uk}(k-1) + d_{kv}(k-1)\}$

Source of improvement: computing $d_{uv}(k)$ takes $\Theta(1)$ rather than $\Theta(n)$ for scalar product in the matrix multiplication algorithm

```
Floyd-Warshall (G,w);
D^G(0) := W^G;
for k:=1 to n do
  for u:=1 to n do
    for v:=1 to n do d_uv(k) = min{d_uv(k-1), d_uk(k-1) + d_kv(k-1)};
return D^G(n)
```

Time complexity: $\Theta(n^3)$

Minimum spanning trees

Input: Connected undirected graph $G=(V,E)$ with a weight function $w : E \rightarrow R$.

Task: Find a minimum spanning tree of G, i.e. a connected acyclic subgraph $G'=(V,T)$ where $T \subseteq E$ with a minimum possible total weight $w(T)$.

Fact: $|T| = |V| - 1$ and thus we may w.l.o.g. Assume that $\forall\ e \in E: w(e) \geq 0$

Idea: Sequentially add edges into a set A which fulfils an invariant that it is at every moment a subset of some minimal spanning tree.

Definition: Let a set of edges A be a subset of some minimal spanning tree. Edge $e \in E$ is called safe for A if also $A \cup \{e\}$ is a subset of some minimal spanning tree.

```
MST (G,w);
A := ∅;
for i := 1 to n -1 do
  find  (u,v) ∈ E which is safe for A;
  A := A ∪ {(u,v)};
return A
```

Definition: A partition $(S,V \setminus S)$ of the vertex into two disjoint subsets is called a cut. Edge $(u,v) \in E$ crosses a cut $(S,V \setminus S)$ if $|\{u,v\} \cap S| = 1$. A cut respects a set A of edges if no edge in A crosses the cut. An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the given cut.

Theorem: Let $G=(V,E)$ be a connected undirected graph with a weight function $w: E \to R$, let $A \subseteq E$ be a subset of some minimum spanning tree, and let $(S, V \setminus S)$ be an arbitrary cut which respects A. If $(u,v) \in E$ is light for $(S, V \setminus S)$, then it is safe for A.

Corollary: Let $G=(V,E)$ be a connected undirected graph with a weight function $w: E \to R$, let $A \subseteq E$ be a subset of some minimum spanning tree, and let C be a connected component (a tree) of the subgraph given by A. If $(u,v) \in E$ is a light edge connecting C to some other component of the subgraph given by A, then (u,v) is safe for A.

We shall describe two different strategies of selecting safe edges using the Corollary:

- algorithm Borůvka (1926) – Kruskal (1956)

  always selects an edge with the smallest weight among all edges connecting any two of the current components of the subgraph given by A

  in every iteration it merges two trees of A into a single tree

  at any moment, the edges in A form a forest

- algorithm Jarník (1930) – Prim (1957)

  at any moment, the edges in A form a single tree

  always selects an edge with the smallest weight among all edges connecting this single tree with the rest of the graph (which is a set of isolated vertices)

```
Kruskal (G,w); (similar to an older algorithm by Borůvka which works in stages)
  sort all edges in E into a non-decreasing sequence by their weights;
  A := ∅;
  for each v ∈ V do Make-Set (v) ;     {each vertex is in a single element set}
  for each (u,v) ∈ E in the sorted order do
    if Find-Set (u)  ≠ Find-Set (v) then{vertices in different sets}
            A := A ∪ {(u,v)};
            Union (u,v);                {merge both sets}
  return A
```

Time complexity: $\Theta(m \log m)$ if the sets are implemented by linked lists

```
Jarník-Prim (G,w,r);                    {r is a starting vertex, a root of the constructed tree}
  Q := V(G);
  for each v∈V(G) do key(v) := ∞ ;
  key(r) := 0; p(r) := NIL;
  while (Q ≠ ∅) do
            u := Extract-Min (Q);
            for each (v∈V(G) such that (u,v) ∈E(G)) do
               if (v ∈ Q) and  key(v) > w(u,v) then
                        key(v) := w(u,v);
                        p(v) := u
```

Time complexity: $\Theta(n^2)$ if Q is implemented as an array

$\Theta(m \log n)$ if Q is implemented as a binary heap

Dynamic sets

dynamic – they change in time (size, content, …)

dynamic set element: accessible via a pointer and contains

- key – typically from some linearly ordered set
- pointer (or several pointers) to the next element (or elements)
- optionally some other data

Dynamic set operations

Let S be a dynamic set of elements, k a key value and x a pointer to an element:

Find(S,k)	returns a pointer to an element with key k in set S or NIL
Insert(S,x)	inserts into S an element pointed to by x
Delete(S,x)	deletes from S an element pointed to by x
Min(S)	returns a pointer to an element of S with a minimum key
Max(S)	returns a pointer to an element of S with a maximum key
Succ(S,x)	returns a pointer to an element of S with a key, which is immediately succeeding a key of an element pointed to by x
Predec(S,x)	the same for an immediately preceding element

Binary search trees (BST)

dynamic data structure which supports all dynamic set operations

binary tree: every element of the dynamic set contains three pointers pointing to

- left child (left)
- right child (right)
- parent (parent)

binary search tree:

for every element x it must hold : all elements in the left subtree of x have smaller (or equal) key than x and all elements in the right subtree of x have greater key than x (WARNING – do NOT confuse a BST with a binary heap)

```
Find(x,k)              {x is a pointer to the root of a BST containing set S}
while (x<>NIL) and (k<>key(x)) do
   if (k< key(x)) then x := left(x)
                  else  x := right(x)
return x
```

Time complexity is O(h), where h is the height of the BST in question

```
Min(x)              {x is a pointer to the root of a BST containing set S}
while (left(x)<>NIL) do x := left(x)
return x

Max(x)              {x is a pointer to the root of a BST containing set S}
while (right(x)<>NIL) do x := right(x)
return x

Succ(x)             {a pointer to the root of a BST containing set S is not needed}
if (right(x)<> NIL)
then return Min(right(x))     {x has a right child - Succ(x) is min in the right subtree}
else  {x has no right child – climb up until a step from left child to parent is done}
begin     y := parent(x)
          while (y<>NIL) and (x=right(y)) do
          begin     x := y
                    y := parent(y)
          end
          return y
end

Predec(x)           {symmetric to  Succ(x)}
```

Time complexity of these search operations is again O(h)

Now we describe the modifying operations Insert and Delete :

```
Insert(x,z)          {x is a pointer to the root of a BST containing set S and
                      z is a pointer to the inserted element with left(z) = right(z) = NIL}
y := NIL
w := x
while (w<>NIL) do                    {going down the tree with a pair of pointers
begin     y := w                      w (first) and y (second), when w reaches NIL,
          if (key(z) < key(w))        then y points to an element, under which z
            then w := left(w)         should be placed}
            else w := right(w)
end
parent(z) := y
if (y<>NIL)
then      if (key(z) < key(y))
            then left(y) := z
            else right(y) := z
else      x := z                      {z becomes a new root, the tree was empty}
```

Time complexity is again O(h).

Delete has three cases, depending on the number of children of the deleted element: 0,1, or 2 children

```
Delete(x,z)                    {x is a pointer to the root of a BST containing set S and
                                z is a pointer to the deleted element}
if (left(z) = NIL) or (right(z) = NIL)
  then y := z
  else y := Succ(z)            {y now points to the element to be deleted}
if (left(y) <> NIL)
  then w := left(y)            {w points to the single child of y (if it exists) or to
  else w := right(y)           NIL (if y has no children) }
if (w <> NIL)  then parent(w) := parent(y)      {connecting the pointer upwards}
if (parent(y) = NIL)
  then x := w                                   {the root was deleted, w points to the new root}
  else if (y = left(parent(y)))                 {y is a left child of its parent}
          then left(parent(y)) := w             {connecting the pointer downwards}
          else right(parent(y)) := w
if (y <> z)  then key(z) := key(y)              {and also content(z) := content(y) e.g. copy
                                                all content of y into z
                                                (except of pointers to children and the parent)}
```

Time complexity is again O(h).

Red-Black Trees

Disadvantage of „ordinary“ BST – all operations are O(h) which is O(log n) on „balanced“ BST (with n nodes) but Ω(n) on „degenerated“ BST (the tree may degenerate to a linked list of length n)

Aim: we want to guarantee O(log n) for all operations in the worst case

Red-Black Tree is a BST, in which every node has two suplementary attributes:

• color, which is either a) red or b) black

• type, which is either a) internal or b) external

Internal nodes are all nodes in the BST with a key, external nodes are artificially added „new leaves“, e.g. each pointer to a descendant form an internal node which is NIL is replaced by a pointer to an external node. External nodes have neither a key nor a content, only color and a pointer to a parent.

Required properties of red-black trees (definition of red-black trees):

1. Every node is either red or black

2. Every external node is black

3. Both descendants of a red node (which must be internal due to 2.) are black

4. Every path from (an arbitrarily chosen but fixed) node x to the leaves in the subtree rooted at x contains **the same number** of black nodes

Observations:

• every internal node has exactly two descendants

• there are never two consecutive red nodes on any path (from a root to a leaf)(3.)

• every path (from a root to a leaf) has the same number of black nodes (4.)

• the longest path (from a root to a leaf) is at most twice as long as the shortest path – the tree is „balanced“

Definitions:

• node height: h(x) = number of nodes (excluding x) on the longest path from x to a leaf in the subtree rooted at x

• node black height : bh(x) = number of black nodes (excluding x) on any path from x to a leaf in the subtree rooted at x (the definition is consistent thanks to 4.)

Lemma 1: Let x be an arbitrary node. Then a subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes.

Lemma 2: Red-Black Tree with n internal nodes has a height at most $2 \log_2(n+1)$.

Corollary: Non-modifying operations (Find, Min, Max, Succ, Predec) on a BST have a guaranteed O(log n) complexity on RBT without any further effort (we can use the generic code for BST).

Rotation (left and right)

Auxiliary operations needed to implement Insert and Delete operations which fulfil:

• they preserve the BST property – for each node x, the keys in the left subtree of x are smaller than the key of x, keys in the right subtree are greater than the key of x

• they only redirect a constant number of pointers and thus run in O(1)

Inserting a node

Observation: if the root of a RBT is red, it can be re-colored to black without violating any of the RBT properties. Thus we may assume that prior to node insertion the root is black (and this property will be maintained in the sequel).

Preprocessing: the node is inserted using the standard insert operation for BST and it is colored red.

Which RBT property may be violated after preprocessing? Only property 3. if both the inserted node x and its parent node y are red. If y is red, it cannot be the root, so it must have a parent node z (which must be black).

Now there are three cases to consider:

1. **Sibling of node y (uncle of node x) is red.**

Action: Nodes y and the sibling of y are re-colored to black, z is re-colored to red. If node z has a black parent node, we do nothing, if z has a red parent node, then we have shifted the "fault" one level up and we iterate (there are three cases again). If node z has no parent node (it is a root), we re-color it to black.

2. **Sibling of node y is black and x is a right descendant of y and y is a left descendant of z, or vice versa.**

Action: If x is a right descendant of y and y is a left descendant of z, then perform LeftRotation(y), in the symmetric case perform RightRotation(y). This action transforms the problem to Case 3.

3. **Sibling of node y is black and x is the same descendant of y as y is of z.**

Action: If x is a left descendant of y and y is a left descendant of z, then perform RightRotation(z) and re-color y to black and z to red. This satisfies all RBT properties and we finish. The symmetric case is similar.

Time complexity of inserting a node is O(log n) :

- preprocessing (ordinary BST insert) is O(log n)
- action of case 1. is O(1) and is performed O(log n) times
- actions of cases 2. and 3. are both O(1) and each is performed at most once

Deleting a node

Preprocessing: the node is deleted using the standard delete operation for BST.

Observation: the actually deleted node (let it be y) has at most one (internal) descendant (let it be x) , if it has no internal descendants, then we mark as x one of the external descendants of y

If y is red, there is no need for any action, all RBT properties are valid after the delete of y as well. If y is black, then property 4. is violated (except when y is a root), since the paths previously leading through y lose one from their black height while all other paths maintain their black height unchanged.

If x is red, it suffices to re-color x to black and all RBT properties become valid. Thus the only interesting case is when x is black.

Main idea: node x is made “doubly black” (which fulfils property 4.) and this “extra black color” is shifted up the tree until it can be disposed of

Observation: if x is the root, then the extra black color can be deleted and the black height of **all** nodes stays the same, if x is not the root, then parent(x) must have one more internal descendant (let it be w), otherwise the paths to leaves violate property 4. (the external descendant of parent(x) would have a smaller black height than x)

We shall assume, that x is a left descendant of parent(x) (the other case is similar), and we shall distinguish four cases according to the color of w and its descendants:

1. **node w is red (and thus has two black descendants)**

Action: we exchange the colors of w and its parent (who is also the parent of x) and perform LeftRotation(parent(x)), which turns the situation into one of cases 2,3,4

2. **node w is black and has two black descendants**

Action: we delete one black color from x and re-color w to red. If the common parent is red we re-color it to black (and we are done), if it si black it gets the extra black color. Such moves of a doubly black node upwards stop in the root the latest.

Remark: if case 2 follows after case 1, the process ends (the common parent of x and w is red after case 1 and is re-colored to black in case 2)

3. **node w is black, its left descendant is red and its right descendant black**

Action: we exchange the colors of w and its left descendant and perform RightRotation(w), which changes the situation to case 4

4. **node w is black and its right descendant is red**

Action: we re-color the right descendant of w to black and remove the extra black color from x. If parent(x) is red, we re-color it to black and re-color w to red. Then we perform LeftRotation(parent(x)).

Time complexity of deleting a node is O(log n):

- preprocessing (ordinary BST delete) is O(log n)
- action of case 2 is O(1) and is performed O(log n) times
- actions of cases 1, 3, 4 are all O(1) and each is performed at most once

AVL trees

Definition (Adelson-Velskii, Landis) A BST is an AVL tree if and only if every node x in the tree fulfils the inequality

|(height of the left subtree of x) - (height of the right subtree of x)| ≤ 1

Theorem The height of an AVL tree with n nodes is O(log n).

Corollary Non-modifying operations (Find, Min, Max, Succ, Predec) on an AVL tree have a guaranteed O(log n) complexity (we can use the generic code for BST).

Modifying operations Insert and Delete are implemented similarly as on an ordinary BST with supplementary balancing using rotations (which are very similar to rotations on RBT).

Divide and conquer (Divide et impera) algorithms

- a design method for developing algorithms (not splitting up a problem in a set of more or less independent subproblems)
- a typical divide and conquer consists of 3 steps

1. DIVIDE the task into several subtasks of the same type but on smaller data
2. SOLVE subtasks, either:
   a) recursively by further divisions if the subtask is on big enough data
   b) directly for subtasks on small enough data (solution is often trivial)
3. UNIFY the solutions of subtasks into a solution of the original task

Examples: Merge-Sort, Binary-Search

Analyzing time complexity

$T(n)$ time to process a task of size n (assumption: if $n < c$ then $T(n) = \Theta(1)$)

$D(n)$ time to divide a task of size n into a subtasks of the same size n/c

$S(n)$ time to unify solutions of subtasks into a solution of the original task of size n

$\Rightarrow$ recursive equation: $T(n) = D(n) + aT(n/c) + S(n)$ for $n \geq c$

$T(n) = \Theta(1)$ for $n < c$

Methods to solve recursive equations

1. substitution method
2. master theorem (“cook-book“ solution)

In both cases the following simplification is used:

- the assumption $T(n) = \Theta(1)$ for small enough n is not explicitly written in the equation
- integrality is disregarded, e.g. $n/2$ is used instead of $\lceil n/2 \rceil$ or $\lfloor n/2 \rfloor$
- the solution is given asymptotically (exact constants are not considered) $\Rightarrow$ asymptotic notation is used already in the recursive equation

Example: Merge-Sort $T(n) = 2T(n/2) + \Theta(n)$

Substitution method

- not really a solution method, it is a verification (proof) method
- guess the asymptotically correct solution
- check the correctness of the estimate (separately for the upper and lower bound) by induction

Example: Merge-Sort again

Master theorem

Let $a \geq 1$, $c > 1$, $d \geq 0$ be real numbers and let $T : \mathbf{N} \rightarrow \mathbf{N}$ be a non-decreasing function such that for all n expressible as $n = c^k$ (where $k \in \mathbf{N}$) we can write

$$T(n) = aT(n/c) + F(n)$$

where the function $F : \mathbf{N} \rightarrow \mathbf{N}$ satisfies $F(n) = \Theta(n^d)$. Let us denote $x = \log_c a$. Then

a) if $x < d$, then $T(n) = \Theta(n^d)$,

b) if $x = d$, then $T(n) = \Theta(n^d \log n) = \Theta(n^x \log n)$,

c) if $x > d$, then $T(n) = \Theta(n^x)$.

Examples:

- Merge-Sort $T(n) = 2T(n/2) + \Theta(n)$
- Binary-Search $T(n) = T(n/2) + \Theta(1)$
- Equation $T(n) = 9T(n/3) + \Theta(n)$
- Equation $T(n) = 3T(n/4) + \Theta(n^2)$
- Equation $T(n) = 2T(n/2) + \Theta(n \log n)$

Square matrix multiplication

Input: matrices A and B of size n x n

Output: matrix C = A ⊗ B (also of size n x n)

Classical algorithm

```
begin for i:=1 to n do
    for j:=1 to n do
    begin C[i,j] := 0;
          for k:=1 to n do C[i,j] := C[i,j] + A[i,k] * B[k,j]
    end
end
```

Time complexity: $T(n) = \Theta(n^3)$ (n^2 scalar products of length n)

Now assume that n is a power of 2 ($n=2^k$), which allows a repeated division of A and B into 4 matrices of quarter the input size (all the way to matrices of size 1 x 1), and let us try "divide and conquer" (we shall get rid of the $n=2^k$ assumption later)

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$C_{11} = (A_{11} \otimes B_{11}) \oplus (A_{12} \otimes B_{21})$$

$$C_{12} = (A_{11} \otimes B_{12}) \oplus (A_{12} \otimes B_{22})$$

$$C_{21} = (A_{21} \otimes B_{11}) \oplus (A_{22} \otimes B_{21})$$

$$C_{22} = (A_{21} \otimes B_{12}) \oplus (A_{22} \otimes B_{22})$$

Each scalar product is “split” into two halves and “completed” by the matrix addition.

Number of matrix operations of order $n/2$: 8 multiplications $\otimes$ and 4 additions $\oplus$

Number of (real number) additions in matrix additions: $4(n/2)^2 = n^2$

Time complexity: $T(n) = 8T(n/2) + \Theta(n^2)$

Master theorem: $a=8, c=2, \log_c a=3, d=2$ $T(n) = \Theta(n^3)$

(asymptotically the same as classical algorithm, but higher hidden constants)

To lower the time complexity we need to lower $a=8$ and keep or slightly raise $d=2$.

Strassen's algorithm (1969)

Uses only 7 sub-matrix multiplications of order n/2 (instead of 8)

$$M_1 = (A_{12} \ominus A_{22}) \otimes (B_{21} \oplus B_{22})$$

$$M_2 = (A_{11} \oplus A_{22}) \otimes (B_{11} \oplus B_{22})$$

$$M_3 = (A_{11} \ominus A_{21}) \otimes (B_{11} \oplus B_{12})$$

$$M_4 = (A_{11} \oplus A_{12}) \otimes B_{22}$$

$$M_5 = A_{11} \otimes (B_{12} \ominus B_{22})$$

$$M_6 = A_{22} \otimes (B_{21} \ominus B_{11})$$

$$M_7 = (A_{21} \oplus A_{22}) \otimes B_{11}$$

Number of matrix operations of order n/2 : 7 multiplicat. $\otimes$, 10 addit. $\oplus$ and subtract. $\ominus$

$$C_{11} = M_1 \oplus M_2 \ominus M_4 \oplus M_6$$

$$C_{12} = M_4 \oplus M_5$$

$$C_{21} = M_6 \oplus M_7$$

$$C_{22} = M_2 \ominus M_3 \oplus M_5 \ominus M_7$$

Number of matrix operations of order n/2 : 8 additions $\oplus$ and subtractions $\ominus$

Time complexity: $T(n) = 7T(n/2) + \Theta(n^2)$

Master theorem: $a=7$, $c=2$, $\log_c a = \log_2 7 = x$, $d=2$ $\quad T(n) = \Theta(n^x) = \Theta(n^{2.81})$

Finding the k-th element out of n elements.

Input: unordered sequence of n (distinct) numbers

Output: k-th smallest number

Time complexity will be measured by the number of comparisons (like for sorting).

For k = 1 (minimum) and k = n (maximum) using n - 1 comparisons (trivially) suffices.

Pro k = n/2 (median) ?????? (we shall show how to solve the problem for general k)

First idea: sort the sequence, then choose k-th element $\Rightarrow$ time complexity $\Omega(n \log n)$

Can be done better? $\Rightarrow$ let's try "divide and conquer"

- select a pivot and divide the sequence into three parts, namely m elements smaller than the pivot, the selected pivot, and (n-m-1) elements greater than the pivot
- we need n-1 comparisons for this task
- if k>m+1 then throw away m+1 smallest elements and look for (k-m-1)-th element among (n-m-1) elements greater than the pivot
- if k=m+1 then the pivot is the sought element and we are done
- if k<m+1 then throw away n-m greatest elements and look for k-th element among m elements smaller than the pivot

This can end up disastrously … unless we make sure that the pivot is "well selected".

Algorithm (Blum et al. 1972)

1. Divide the sequence of length n into $\lceil n/5 \rceil$ 5-tuples (the last may be incomplete).
2. Find a median in each 5-tuple.
3. Recursively find a median of the selected set of $\lceil n/5 \rceil$ medians.
4. Use the median of medians as a pivot to divide the input sequence.
5. Unless the median of medians is the sought element, recursively search in the set of smaller elements (than the pivot) or in the set of greater elements.

How "good" is the division using the pivot found by the above algorithm?

Claim: The set of elements smaller than pivot and the set of elements greater than pivot contain at least 3n/10 elements each $\Rightarrow$ step 5 uses at most 7n/10 elements

Let: T(n) = worst case number of comparisons necessary to find k-th out of n elements

$$T(n) = 7n/5 + T(n/5) + (n-1) + T(7n/10)$$

medians of 5-tuples (1.+2.)

median of medians (3.)

division using the pivot (4.)

subproblem solution (5.)

Theorem: T(n) = O(n)

Proof: by a substitution method (key fact: 1/5+7/10<1, not true for division into triples)

Average case complexity of QuickSort

1. select a pivot (e.g. the leftmost element in the current subsequence)
2. divide all elements into three sets by comparing them with the pivot
3. recursively sort the sets of smaller (than pivot) elements and greater elements

The time complexity is again measured by the number of comparisons

Best case: pivot is always exactly in the middle (pivot = median)

$$T(n) = 2T(n/2) + (n-1) \Rightarrow T(n) = \Theta(n \log n)$$

Worst case: pivot is always on the margin (pivot = max or pivot = min)

$$T(n) = T(n-1) + (n-1) \Rightarrow T(n) = \Theta(n^2)$$

Average case: ???? (try „intuition“ first)

What happens if the pivot is always selected in a way which splits the sorted sequence in a ratio of 99:1 or better?

$$T(n) = T(99n/100) + T(n/100) + (n-1) \leq T(99n/100) + T(n/100) + n$$

Solution: $T(n) = \Theta(n \log n)$ which can be verified by analyzing the tree of recursive calls

Remark: this will happen for every constant ratio, i.e. any ratio independent of n

Assumptions for a precise analysis:

1. the sorted sequence is {1,2, … ,n}, which can be assumed w.l.o.g.
2. each of the n! permutations has the same likelihood of appearing on the input
3. the pivot is always selected as the leftmost element in the current subsequence
4. each split of the sequence by the selected pivot preserves the randomness of the order of both created subsequences

pivot	likelihood	small	large
1	1/n	0	n-1
2	1/n	1	n-2
…	…	…	…
n-1	1/n	n-2	1
n	1/n	n-1	0

$$T(n) = 1/n \sum_{m+v=n-1}(T(m) + T(v)) + (n-1) = 1/n \sum_{i=0}^{n-1}(T(i) + T(n-1-i)) + (n-1)$$

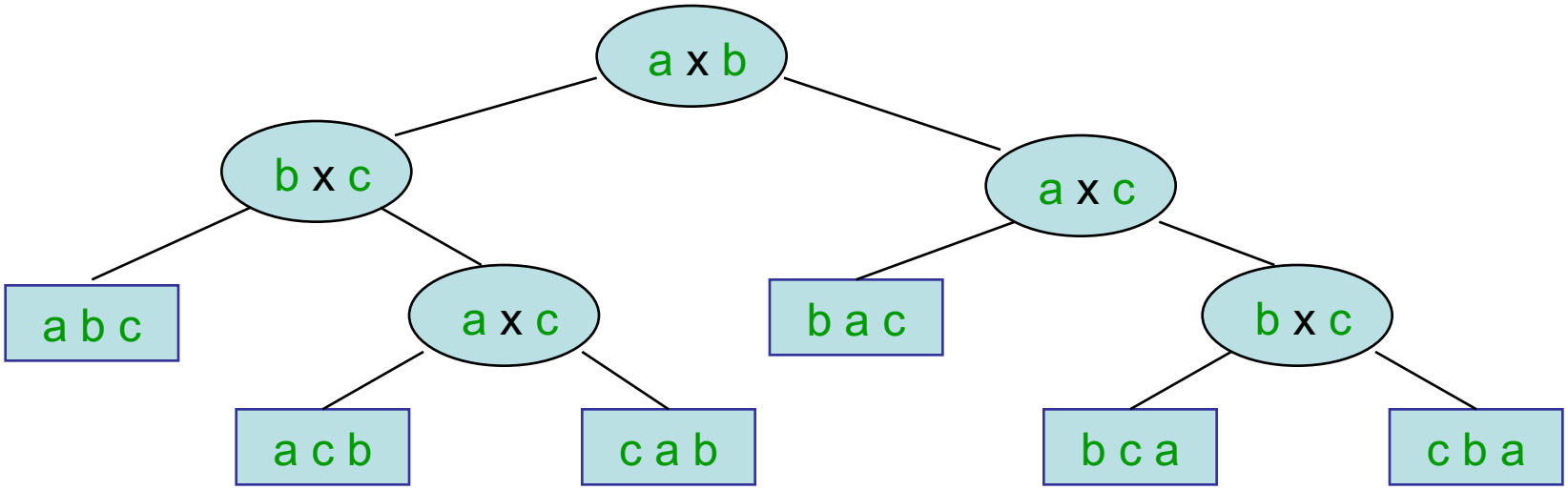
$$T(n) = 2/n \sum_{i=0}^{n-1} T(i) + (n-1)$$

with the initial condition $T(1) = 0$

Lower bound for sorting using pair-wise comparisons

Observation: every (deterministic) sorting algorithm based on pair-wise comparisons can be uniquely modeled by a decision tree, which is a binary tree with leafs corresponding to permutations of the input sequence, and inner nodes corresponding to comparisons.

Example: decision tree for Insertion-Sort and n=3 (let us denote the input by a,b,c)



The left branch always corresponds to "<" and the right branch to ">" (w.l.o.g. we assume that the inputs are pair-wise distinct to avoid ties)

The decision tree models a sound sorting algorithm $\Rightarrow$ it must contain leafs with all n! possible permutations of the input sequence.

Worst case number of comparisons = longest branch from root to leaf = tree height.

Theorem: A binary tree with at least n! leafs has height $\Omega(n \log n)$.

Sorting in linear time

- never compares pairs of elements (values, numbers) in the sorted sequence
- uses direct addressing (in an array) by the sorted values (keys)

Counting Sort

Input: n numbers from an interval 1 to k (and we shall assume $k \in O(n)$)

Data structures: A[1..n] input array

B[1..n] output array

C[1..k] auxiliary array

Algorithm:

```
for i := 1 to k do C[i] := 0;                    {initialization}
for j := 1 to n do C[A[j]] := C[A[j]] +1;        {each C[i] contains # of input numbers equal to i}
for i := 2 to k do C[i] := C[i] + C[i-1];        {each C[i] contains # of input numbers ≤ than i}
for j := n downto 1 do begin B[C[A[j]]] := A[j]; C[A[j]] := C[A[j]] – 1 end;
```

Time complexity: clearly $O(k+n)$ and thus $O(n)$ if $k \in O(n)$

Additional property: stability = equal values appear on the output in the same order in which they appeared on the input

Radix Sort

• historic usage: punch card sorting on a mechanical device

• task: sort n cards each having a d-digit number punched in the last d columns

• intuitive algorithm: sort into piles according to the *most* significant digit, then recursively sort the individual piles (very demanding for the sorting device operator)

• radix sort: sort into piles according to the *least* significant digit, put piles together (in the correct order), and sort similarly according to the second least significant digit etc.

• necessary condition: each sorting iteration (pass) is stable (two cards with the same digit in the currently processed column must go to the output in the same order in which they appeared on the input)

• contemporary usage (software versions of radix sort):

sorting data with multiple hierarchical keys (e.g. year, month, day)

sorting of alphanumerical keys (words)

counting sort can be used as a stable sort for the individual passes

• time complexity (when using counting sort as a subroutine)

$O(d(n+k)) = O(n)$ if $k \in O(n)$ and d is a constant

remark: numbers with different number of digits on the input: fill with zeros from the *left*

words of different length on the input: fill with spaces from the *right*

Hashing

Hash tables are suitable for representing dynamic sets, which need only Insert, Delete and Search operations

Direct address table = Array (trivial case of hash table)

- table size = number of **all** possible keys regardless of the number of **used** keys
- assumptions: distinct items always have distinct keys (= address in the table)
  and the set of all possible keys is sufficiently small
- the array stores: either directly the data (it sufficiently small) including the given key,
  or a pointer to the data including the given key (or NIL)
- Insert, Delete, and Search all have time complexity $\Theta(1)$

Hash table

- useful in case that the universe of all possible keys is too large, either in absolute terms (insatisfiable memory requirements), or relatively with respect to the number of stored items (inefficient representation, most cells in the array is not used).
- the address in the hash table is now **computed** from the key using a hash function

$h : U \rightarrow \{0,1, \dots , m-1\}$ where typically $|U| >> m$

Problem: two (or more) keys are hashed into the same value (table address) = **collision**

Observation: collisions are impossible to avoid if $|U| > m$

Methods for resolving collisions: item chaining

open addressing

Collision resolution by item chaining

- items hashed into the same slot (address) of the hash table are stored in a linked list
- each slot of the hash table contains either a pointer to the head of the list or NIL
- Insert(x) = compute h(key(x)) and insert x at the head of the corresponding list - $\Theta(1)$
- Delete(x) = $\Theta(1)$ if the linked lists are doubly linked and the deleted item x is pointed to by an outside pointer, otherwise the complexity is the same as for Search(x)

Analysis of Search(x) complexity with item chaining

Assumptions: 1) hash function value is computed in $\Theta(1)$
2) simple uniform hashing = a key is hashed with the same probability into each of m slots in the table, independently of any other key

Notation: Load factor $\alpha = n/m$
where m is the size of the table and n is the number of stored items

Theorem 1 In a hash table in which collisions are resolved by chaining, an unsuccessful search takes time $\Theta(1 + \alpha)$ on average under the assumption of simple uniform hashing.

Theorem 2 In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1 + \alpha)$ on average under the assumption of simple uniform hashing.

Corollary If $n = O(m)$ then $\alpha = O(1)$ and thus Search(x) takes $\Theta(1)$ on average.

Hash functions

Three most common construction methods: the division method

the multiplication method

universal hashing

Remark 1: hash function should distribute the keys from the universe of keys into m slots in the table as evenly as possible

Remark 2: keys are assumed to be integers (so that common arithmetic operations can be applied to them), if the keys are not integers (e.g. strings of characters), then they have to be first converted to numerical by some adequate procedure.

The division method: $h(k) = k \bmod m$ (remainder after the division by m)
Bad choices of m: $m = 2^p$, 10^p, $2^p - 1$ (e.g. $k \bmod 2^p$ gives the last p bits in the binary representation of k, which may be not very random)
Good choices of m: primes not too close to exact powers of 2

The multiplication method:

$$h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$$

where $0<A<1$ is a suitably chosen number (Knuth recommends $A \approx (\sqrt{5} - 1)/2 = 0.618033\ldots$) and if m is selected as a power of 2, $h(k)$ can be computed quite easily

Universal hashing:

Problem: for each deterministic hash function h one can choose n keys in such a way, that h maps all of them into the same slot in the hash table (if $|U| > n^2$ and thus $|U|/n > n$)

$\Rightarrow$randomization

Idea: choose a hash function randomly and independently of the keys to be hashed (e.g. of the n keys, that will be used), from some suitably chosen set of functions

Advantage: $\Rightarrow$ no particular input (concrete n keys) is a priori bad for such hashing
$\Rightarrow$ hashing the same input repeatedly calls different hash functions

Definition: Let H be a finite set of hash functions from a universe U of keys into the set $\{0, \ldots, m-1\}$. Set H is called universal, if for every two distinct keys $x,y \in U$ the number of functions $h \in H$ satisfying $h(x) = h(y)$ is equal to $|H| / m$.

Observation: For a randomly chosen function $h \in H$ the probability of a collision of two randomly chosen keys $x \neq y$ equals $1 / m$, which is the same probability as if the values $h(x)$ and $h(y)$ are drawn randomly and independently from the set $\{0, \ldots, m-1\}$.

Theorem: Let h be randomly drawn from a universal set of hash functions and let it be used for hashing n keys into a table of size m, where $n \leq m$. Then the expected number of collisions affecting a randomly chosen but fixed key x is less than 1.

Remark: The assumption $n \leq m$ implies, that the average list length (of keys hashed into the same slot) is less than 1.

Does a universal set of hash functions exist? And if it does, how to construct it?

Construction: (one out of many possibilities): select a prime number m and divide each key x into $(r+1)$ parts (the value of r depends on the key lenght). Denote

$$x = \langle x_0, x_1, \ldots , x_r \rangle$$

and select r in such a way that the maximum value of each x_i is strictly less than m. Let

$$a = \langle a_0, a_1, \ldots , a_r \rangle$$

be a sequence of $(r+1)$ numbers randomly and independently drawn from $\{0, \ldots ,m-1\}$. Let

$$h_a(x) = \left(\sum_{i=0}^{r} a_i x_i\right) \bmod m \quad \text{a} \quad H = \bigcup_a \{h_a\}$$

We get: $|H| = m^{r+1}$ (the number of distinct vectors a)

Theorem: H is a universal set of hash functions.

Collision resolution by open addressing

• all items are stored directly in the hash table, thus the load factor must be at most one, i.e. $\alpha = n/m \leq 1$

• instead of following the pointers in the list of items hashed into the same slot (the list is typically stored outside of the table) the slot addresses are computed sequentially

• using the same memory size the hash table can be bigger than when resolving the collisions by item chaining (the space used for pointers is saved)

• the computed slot address depends on the hashed key and the probe number:

$$h : U \times \{0,1, \ldots ,m-1\} \rightarrow \{0,1, \ldots , m-1\}$$

• given key x, the slots in the hash table are probed in the following order:

$$(h(x,0), h(x,1), \ldots , h(x,m-1))$$

and this order must constitute a permutation of $\{0,1, \ldots , m-1\}$, i.e. the probe sequence for every key x subsequently probes all slots in the hash table

• open addressing supports Search and Insert operations well, but the implementation of the Delete operation is complicated (if Delete has to be implemented, then it is better to resolve collisions by item chaining)

```
Hash-Search(T,k)            {searching for key k in hash table T}
i := 0;
repeat    j := h(k,i);
          if T[j] = k then return j;       {and terminate}
          i := i+1
until (T[j]=NIL) or (i=m);
return NIL


Hash-Insert(T,k)            {inserting key k into hash table T}
i := 0;
repeat    j := h(k,i);
          if T[j] = NIL then   T[j] := k;
                               return j;  {and terminate}
                    else i:=i+1
until (i=m);
error  table overflow
```

When implementing hash function h it is desirable to get as close as possible to uniform hashing: the probe sequence of a randomly chosen key is with an equal probability any of the m! permutations of {0,1, … , m-1}

Most common methods for probe sequence construction are linear probing

quadratic probing

double hashing

(none of them achieves uniform hashing, but they are gradually closer to this goal).

Linear probing

uses „ordinary“ hash function $h' : U \to \{0,1, \dots , m-1\}$ using which it defines

$$h(x,i) = (h'(x) + i) \bmod m$$

Disadvantages:

- only m distinct probe sequences, each of them determined by its first slot $h'(x)$
- creates long primary clusters of occupied slots, which increases search time

Quadratic probing

Hash function $h(x,i) = (h'(x) + ci + di^2) \bmod m$ where $c \neq 0$ a $d \neq 0$ (and h' is as above).

Parameters c,d must be chosen carefully so that the probe sequence is a permutation of $\{0,1, \dots , m-1\}$. There are again only m distinct sequences, but no primary clusters are created, only secondary clusters of keys with the same starting slot $h'(x)$.

Double hashing

Hash function $h(x,i) = (h_1(x) + i\, h_2(x)) \bmod m$ where h_1 and h_2 are auxiliary hash functions $U \rightarrow \{0,1, \ldots , m-1\}$. Properties

- function h_2 must be chosen in such a way that $h_2(x)$ and m are relative primes (otherwise the probe sequence will not form a permutation)
- the number of distinct probe sequences is m^2
- this method is the best of the three methods and is the closest to uniform hashing
- examples of (common) choices:

1. $m = 2^p$ (power of two) and $h_2(x)$ gives an odd number (for every x), or
2. m is a prime and $0 \leq h_2(x) < m$

Analysis of hashing by open addressing

Theorem: The expected number of probes during an unsuccessful search in an open addressing hash table with a load factor $\alpha = n/m < 1$ is at most $1/(1-\alpha)$ (under the uniform hashing assumption).

Theorem : The expected number of probes during a successful search in an open addressing hash table with a load factor $\alpha = n/m < 1$ is at most $1/\alpha \ln(1/(1-\alpha)) + 1/\alpha$ (under the uniform hashing assumption and if every key is searched for with the same probability).

Euclid's Algorithm

Algorithm for computing the greatest common divisor of two natural numbers

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 least 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 operations 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.