Overview

- I Introduction.
- II Inductive Design, Divide-and-Conquer, Dynamic Programming.
- III Algorithsms for Lists (Sorting and Searching).
- IV Algorithms for graphs.
- V Algebraic algorithms.
- VI Reduction to other problems. Outlook: The classes P and NP.

Literature

- M Manber: *Introduction to Algorithms*, 1989.
- K Knuth: *The Art of Computer Programming*, Vol 1-3, 1970-73/ 1997-1998.
- AHU Aho, Hopcroft, Ullman: *The Design and Analysis of Computer Algorithms*, 1974.
- B Bentley, Programming Pearls, 1986.
- Sh Shen: *Algorithms and Programming*, Birkhäuser, 1997.
- Sed Sedgewick: *Algorithms*, 1988.

I. Introduction

What is "Algorithm Design and Analysis"?

- How to build algorithms.
- Prove that algorithms do what they claim.
- Determine the (relative) Performance of Algorithms.
- Important "standard" algorithms.

Not considered:

- Hardware-dependence
- Programming Language (will use pseudocode).
- Input/Output (unless this is the algorithms task)

Inductive building

Take a list L of n integers. We want to find the largest element of L.

Suppose we know how to solve the problem for $n-1$ elements. Then:

$a := L[n]$;
$newL := L\{[1..n-1]\}$;
$m := \max(newL)$;
$m := \max(m, a)$;

We can use the same idea to compute the maximum of $newL$, getting to smaller and smaller problems.

Of course eventually we need a "seed": The maximum of a 1-element list is its element.

Many problems come with a measure of "size" (the n) and there are natural ways to define subproblems of smaller size. One can try to use a similar approach for them.

This process has close relation to induction and we can use induction techniques to prove results for the algorithm.

I.1 Induction

Induction is a technique to prove statements that depend on a parameter n.

Assume that P is such a statement, $P(i)$ means that P holds for the value i of the parameter. Then we prove:

1. $P(1)$ holds.
2. If $P(i-1)$ holds then also $P(i)$ holds.

If we have proven these two statements, then $P(i)$ holds for every positive i. (But not for "infinity"!)

A common variation is a slightly stronger assumption:

2. If $P(j)$ holds for all $j < i$ then also $P(i)$ holds.

(This version sometimes requires *several* initial statements $P(1), P(2), \ldots$.)

For example:

$$P(n) : 1 + 2 + \cdots + (n-1) + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

or:

$$P(n) : \text{If } 1 + x > 0 \text{ then } \quad (1+x)^n \geq 1 + nx$$

Geometric Series

A series

$$a, aq, aq^2, aq^3, \ldots$$

is called an *geometric series*. We have

Theorem: For $q \neq 1$ we have

$$a + aq + aq^2 + \cdots + aq^{n-1} = \sum_{i=0}^{n-1} aq^i = a\frac{q^n - 1}{q - 1} = a\frac{1 - q^n}{1 - q}$$

in particular, for $q < 1$ we have

$$\lim_{n \to \infty} \sum_{i=0}^{n} aq^i = a\frac{1}{1-q}$$

Graph Terminology

A *graph* is a formal way to describe a "net" (or, say: a road pattern; People knowing each other; Dependencies; ...)

A graph consistes of a set V of *vertices* and a set $E \subset V \times V$ of *edges* between vertices (directed or undirected). Two vertices u and v are called *adjacent* if there is an edge between them.

The *degree* of a vertex is the number of edges leaving the vertex ($\deg(v) = |\{w \mid (v, w) \quad \text{Edge}\}|$).

A *path* from vertex u to vertex v is a sequence of edges leading from u to v.

Two paths are *edge-disjoint* if they have no common edge.

If U is a set of vertices, the *subgraph induced by* U is the graph with vertex set U and edges $\{(v, w) \in E \mid v, w \in U\}$.

A set of vertices is called an *independent set* if there are no adjacencies between its elements.

Example: Independent Set

Theorem: (see [M, p. 19])
Let $G = (V, E)$ be a directed graph. There is an independent set $S \subset V$ such that every vertex in G is connected to a vertex in S by a path of length at most 2.

begin
  Let v a Vertex;
  Let $N(v) := \{v\} \cup \{w \in V \mid (v, w) \in E\}$;
  Call recursively for induced graph on $V - N(v)$. Let T be the result.
  if $T \cup \{v\}$ is independent **then**
    return $T \cup \{v\}$;
  else
    return T;
  fi;
end

Example: Strengthening the hypothesis

Theorem: (see [M, p. 23])
Let $G = (V, E)$ be a connected, undirected graph. Let $O \subset V$ be the set of vertices with odd degree. We can divide O in pairs such that the vertices of each pair are connected by a path and all these paths are edge-disjoint.

Loop invariants

Induction is frequently used to prove that loops in an algorithm are correct. Here we use induction over the number of times a loop is execute.

The induction hypothesis is about the value of a variable depending of the index of the iteration. Such an induction hypothesis is called a *loop invariant*.

Input: m (a positive integer).
Output: b (bit array, binary representation of m).

```
begin
  t := m; {preserve the original value}
  k := 0;
  while t > 0 do
    k := k + 1;
    b[k] := t mod 2;
    t := t div 2;
  od;
end
```

Theorem: When this algorithm terminates, b contains the binary representation of m.

We use the following induction hypothesis:

$P(k)$: If m is represented by the binary number in $b\{[1..k]\}$, then $n = t \cdot 2^k + m$.

1. The hypothesis holds at the beginning of the loop ($k = 0$).
2. If it holds for k, it holds for $k + 1$.
3. When the loop terminates, $P(k)$ implies the theorem's claim.

I.2 Algorithm analysis

For many problems a variaty of algorithms are possible. We want to compare them for *speed* (and sometimes *storage requirements*).

We do not simply test these by running examples on machines (implementation independence!) but by estimating the number of required computation steps if the input gets “bigger”.

We associate to each input an integer, the *size* n.

We want to estimate the maximum runtime of an algorithm with input of size n, the *worst case time complexity* as a function of n. Assumption: n large.

Similar: *average time complexity* and *space complexities*.

Computation model

We assume a machine of the following type:

Assumptions

We assume there is an infinite number of memory cells, each of which can hold an integer of arbitrary size.

(Real world: both limited. For the algorithms considered here this is irrelevant.)

The program cannot modify itself.

For details of such an machine see [AHU, 1.2] or [K, 1.3].

We usually assume each step (loading/storing of variable, single arithmetic operation, single comparison) to take one unit of time. (Refinements: Multiplication more expensive than addition.)

Sometimes we count only certain expensive operations (comparisons when sorting).

Three simple algorithms

1) Vector Addition

```
for i in [1..n] do
   c[i] := a[i] + b[i];
od;
```

Runtime:
$T_1(n) = A + Bn$
A: Setup
B: Addition

2) Scalar Product

```
s := 0;
for i in [1..n] do
   s := s + a[i] * b[i];
od;
```

Runtime:
$T_2(n) = A + Cn$
A: Setup
C: Multiplication and Addition.
(So $2B \sim C$.)

3) Multiplication Table

```
for i in [1..n] do
   for j in [1..n] do
      c[i, j] := a[i] * b[j];
   od;
od;
```

Runtime:
$T_3(n) = A + Xn + Dn^2$
A, X: Setup
D: Multiplication

We measure the runtime depending on the size of the input ($2n$ numbers in each case).

The O notation

We want to consider the times for large n and to ignore constant factors between algorithms.

Definition: Let $f(n)$, $g(n)$ be functions of the natural (or real) numbers. We say that

$$f(n) = O\left(g(n)\right)$$

if there exist constants A, N such that

$$|f(n)| \leq |A \cdot g(n)| \quad \text{for all} \quad n \geq N.$$

For example

$$\begin{aligned} 1 + n &= O\left(n\right) \\ 9n^2 + 7n - 28 &= O\left(n^2\right) \\ 31415926 &= O\left(1\right) \end{aligned}$$

We may also write $f(n) = g(n) + O\left(h(n)\right)$, meaning that $f(n) - g(n) = O(h(n))$.

In the examples we have that $T_1(n) = O(n)$, $T_2(n) = O(n)$, $T_3(n) = O\left(n^2\right)$.

We show easily: $O\left(n^a\right) = O\left(n^{a+1}\right)$.

For example, remember that
$1 + 2 + \cdots + n = n(n+1)/2 = \frac{1}{2}n^2 + \frac{1}{2}n$. So

$$\begin{aligned} 1 + 2 + \cdots + n &= O(n^3) \\ 1 + 2 + \cdots + n &= O(n^2) \\ 1 + 2 + \cdots + n &= \frac{1}{2}n^2 + O(n) \end{aligned}$$

Lemma: Let $f(n) = O\left(r(n)\right)$ and $g(n) = O\left(r(n)\right)$.
Then:

$$\begin{aligned} f(n) + g(n) &= O\left(r(n) + s(n)\right) \\ f(n) \cdot g(n) &= O\left(r(n) \cdot s(n)\right) \\ f(n) + r(n) &= O\left(r(n)\right) \end{aligned}$$

Other useful rules:

$$f(n) = O\left(f(n)\right)$$

$$
\begin{aligned}
c \cdot O(f(n)) &= O(f(n)) \\
O(f(n)) + O(f(n)) &= O(f(n)) \\
O(O(f(n))) &= O(f(n)) \\
O(f(n))\,O(g(n)) &= O(f(n)g(n)) \\
O(f(n)g(n)) &= f(n)O(g(n))
\end{aligned}
$$

but:

$$
\begin{aligned}
O(f(n)) - O(g(n)) &\neq O(f(n) - g(n)) \\
O(f(n))/O(g(n)) &\neq O(f(n)/g(n))
\end{aligned}
$$

Theorem: Let f be a *monotonically growing* function (i.e. $f(n_1) \geq f(n_2)$ whenever $n_1 \geq n_2$). Then for every $c > 0$ and every $a > 1$ we have that

$$(f(n))^c = O\left(a^{f(n)}\right)$$

In particular $n^c = O(a^n)$.

Similarly $\log n = O\left(n^d\right)$ for all $d > 0$.

The following list is in "ascending O-order":

$$1 \qquad \log n \qquad \sqrt{n} \qquad n \qquad n \log n \qquad n^2$$

$$n^3 \quad \cdots \quad 1.5^n \quad 2^n \quad \cdots \quad n^n$$

Ω and Θ

There is a similar notation for giving lower bounds. We say that

$$f(n) = \Omega(g(n))$$

if there exist constants A, N such that

$$|f(n)| \geq |A \cdot g(n)| \quad \text{for all} \quad n \geq N.$$

We say that $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

For example we have that:

$$\begin{aligned} n^2 &= \Omega(n) \\ n &= \Omega(n^{0.9}) \\ n^2 &= \Theta(n^2 + 1000) \\ 2^n &= \Omega(n^{9999}) \end{aligned}$$

For polynomials, the O,Ω and Θ class is determined by the

largest exponent:

$$\sum_{i=0}^{n} a_i x^i = \Theta(x^n)$$

Summations and Iterations

for i in $[1..n]$ **do**
Statements;
od;

has the time cost $\sum_{i=1}^{n} S(i)$ where $S(i)$ is the time for statements. Assume we know that $S(i) = O\left(U(i)\right)$, $(U(i) > 0$ because time is positive!) we want to know $O\left(\sum_{i=1}^{n} U(i)\right)$.

Lemma: If $S(i) = O\left(i^k\right)$ then $\sum_{i=1}^{n} S(i) = O\left(n^{k+1}\right)$.

A technique which is often useful for this is to shift terms:

Let $S_2(n) = \sum_{i=1}^{n} i^2$ and $S_3(n) = \sum_{i=1}^{n} i^3$. We want to give an explicit formula for $S_2(n)$.

Recursion and Recurrence

```
procedure Job(n)
  if n = 0 then
    special treatment of case 0;
  else
    Job(n − 1);
    extend to n;
  fi;
end
```

A call Job(n) takes time $T(n)$. We have:

$$
\begin{aligned}
T(0) &= A \\
T(n) &= B + T(n-1) \qquad (n > 0)
\end{aligned}
$$

For example:

```
procedure hanoi(n,start,finish,intermediate)
  if n > 0 then
    hanoi(n − 1,start,intermediate,finish);
    move disc from start to finish;
    hanoi(n − 1,intermediate,finish,start);
  fi;
end
```

Recurrence relation

A *Recurrence relation* is a way of defining a function by an expression involving the same function. For example:

$$F(n) = F(n-1) + F(n-2), \quad F(1) = F(2) = 1$$

Recurrence relations appear frequently in the analysis of algorithms. We will sometimes need to express their values by an explicit (or closed-form) expression or at least an O-estimate.

For this one usually takes an "(educated) guess" about the behaviour of the recurrence. A few expansions of the recurrence can help the guess [M, 3.5.1].

(An alternative way (for those who know linear algebra) is to express the recurrence by a matrix A and the n-th value by a vector $v(n) = (F(n), F(n-1), \ldots, F(n-m))$. Then $v(i) = v(0) \cdot A^i$ and we can find a solution by diagonalizing A using eigenvalues.)

Divide-and-conquer style algorithm

```
procedure search(L,obj,from,to)
  if from = to then
    return L[from] = obj;
  else
    m := (from + to) div 2;
    if from ≤ m then search(L,obj,from,m);
    else search(L,obj,m + 1,to); fi;
  fi;
end
```

Here we have $T(n) = T(\frac{n}{2}) + A$.

The master reccurence solution

Theorem: [M., 3.5.2] Suppose that $T(n) = aT(\frac{n}{b}) + cn^k$ ($n \geq 1$, a, b, c, k constants). Then

$$T(n) = \begin{cases} O\left(n^{\log_b a}\right) & \text{if } a > b^k \\ O\left(n^k \log_b n\right) & \text{if } a = b^k \\ O\left(n^k\right) & \text{if } a < b^k \end{cases}$$

II. Design by Induction

Problem:[M,5.2] Given numbers $a_n, a_{n-1}, \ldots, a_0$ and x, evaluate the polynomial

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Induction Hypothesis: We know how to evaluate $P_{n-1}(x) = a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$.

$$P_n(x) = P_{n-1}(x) + a_n x^n$$

$Mult(n) = Mult(n-1) + n + 1 = \frac{n(n+1)}{2}$,
$Add(n) = Add(n-1) + 1 = n$.

Induction Hypothesis 2: We know how to evaluate $P_{n-1}(x) = a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ and how to compute x^{n-1}.

$Mult(n) = Mult(n-1) + 2 = 2n$.

Induction Hypothesis 3: We know how to evaluate $\widetilde{P}_{n-1}(x) = a_n x^{n-1} + a_{n-1} x^{n-2} \cdots + a_2 x + a_1$.

$$P_n(x) = x \cdot \widetilde{P}_{n-1}(x) + a_0.$$

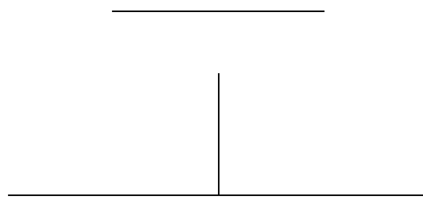
$Mult(n) = Mult(n-1) + 1 = n$.
$Add(n) = Add(n-1) + 1 = n$.

$$\begin{array}{ccc} a_n x^n & + & a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = \\ ((& \cdots & (a_n x + a_{n-1}) + a_{n-2}) \cdots) x + a_1) x + a_0 \end{array}$$

This is sometimes called **Horner's rule** (W.G. Horner, 1786-1837, see `http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Horner.html`).

Maximal Induced Subgraph

Given an undirected graph $G = (V, E)$ and $k \in \mathbb{N}$, find an induced subgraph $H = (U, F)$ of G of maximum size such that all vertices of H have degree $\geq k$ (in H) if such a graph exist.[M 5.3]



Try to *prove that such an algorithm exists*:

Induction hypothesis: If the number of vertices is $< n$ we can find a maximum induced subgraph with vertex degree $\geq k$ each.

Input: Graph $G = (V, E)$, $k \in \mathbb{N}$

```
function maxinduced(V,E,k)
begin
  n := |V|;
  if n ≤ k then
    return false;
  elif n = k + 1 then
    for v ∈ V do
      if deg(v) < k then return false; fi;
    od;
    return V;
  else {n > k + 1}
    lv :=false;{lv will be a vertex of degree < k}
    for v ∈ V do
      if deg(v) < k then
        lv := v; break;
      fi;
    od;
    if lv =false then
      return V;
    else
      F := {e ∈ E | lv ∉ e};
      return maxinduced(V − {lv},F,k);
    fi;
  fi;
end
```

Finding One-To-One Mappings

Given a finite set A and $f\colon A \rightarrow A$, find $S \subset A$ as large as possible such that $f(S) \subseteq S$ and f is a bijection on S.[M 5.4]

Induction hypothesis: We know how to solve the problem for $|A| = n - 1$.

Input: f (array from 1 to n)
Output: $S \subset [1..n]$.

```
begin
  S := [1..n];
  for j ∈ [1..n] do count[j] := 0; od;
  for j ∈ [1..n] do increment count[f[j]]; od;
  Queue := [];
  for j ∈ [1..n] do
    if count[j] = 0 then Add(Queue, j); fi;
  od;
  while Queue not empty do
    remove i from Queue;
    S := S \ {i};
    decrement count[f[j]];
    if count[f[j]] = 0 then Add(Queue, f[j]); fi;
  od;
end
```

The Celebrity Problem

A *celebrity* is somebody whom everybody knows but who does not know anyone. We are given a group of n persons and want to find out whether it contains a celebrity.[M 5.5]

For this we may ask a a whether she knows b and so forth. There are $n(n-1)/2$ pairs of persons, so we pontentially would have to ask $n(n-1)$ questions.

Graph theoretic formulation: Directed graph. Vertices=People, Edge from a to b if a knows b. A celebrity has in-degree $n-1$ and out-degree 0. (There is at most one celebrity!)

The input is a $n \times n$ matrix $Know$, $Know[a,b]=1$ if a knows b.

Problem: Is there an i such that $Know[j,i]=1$ $(j \neq i)$ and $Know[i,j]=0$ $(j \neq i)$?

Again, we unravel the recursion:

Input: $Know$ (a $n \times n$ boolean matrix).
Output: celebrity (if exists)

```
begin
  i := 1; j := 2; next := 3; {eliminate all but one}
  while next ≤ n + 1 do
    {eliminate the one who knows}
    if Know[i, j] then i := next;
    else j := next; fi;
    next := next + 1;
  od;
  if i = n + 1 then cand := j;
  else cand := i; fi;
  flag := false; k := 1; {check}
  while not flag and k ≤ n do
    if cand ≠ k then
      if Know[cand, k] then flag := true; fi;
      if not Know[k, cand] then flag := true; fi;
    fi;
    k := k + 1;
  od;
  if not flag then return cand;
  else return "no celebrity"; fi;
end
```

Divide-and-Conquer

So far we have reduced a problem of size n to a problem of size $n-1$. In some situations however we then still have to relate the n-th element to the $n-1$ other elements, so we would get algorithms of complexity $n(n-1) = O\left(n^2\right)$ or worse.

A frequently used approach here is to "balance the execution tree" by recurring to problems of size $\frac{n}{a}$. Every recursion step then covers a problem whose size is a *factor* a larger than the one before, so after b steps, we can cover problems of size a^b. This may reduce the $n-1$ recursion steps to $\log_a(n)$ steps.

Such an approach is called "Divide-and-Conquer".

As a simple example lets try to find the largest and heaviest of a list l of people:

$large := l[1]$; $heavy := l[1]$;
for $i \in [2..n]$ **do**
   if larger($l[i], large$) **then** $large := l[i]$; **fi**;
   if heavier($l[i], heavy$) **then** $heavy := l[i]$; **fi**;
od;

This algorithm uses $2n-2$ comparisons.

Alternative "Divide-and-Conquer" algorithm:

```
function compare(l)
  if |l| = 2 then return [largest(l), heaviest(l)]; fi;
  divide l in two halfs, l1 and l2.
  c1 := compare(l1);
  c2 := compare(l2);
  return [largest([c1[1], c2[1]]), heaviest([c1[1], c2[1]])];
```

Number of comparisons:

$$T(n) = \begin{cases} 1, & n = 2, \\ 2T(n/2) + 2, & n > 2. \end{cases}$$

By induction we prove: $T(n) = \frac{3}{2}n - 2$.

A better example is given by long integer multiplication. Here the traditional "pencil and paper" algorithm requires $O\left(n^2\right)$ operations.

Let x and y be two numbers. We represent these in "radix form" with respect to a base B, using n "digits" from 0 to $B - 1$:

$$x = \sum_{i=0}^{n} x_i B^i$$

For simplicity assume that n is a power of two and that we are calculating in base $B = 2$. We partition x and y in two $\frac{n}{2}$-parts

each:

$$x = \boxed{a \mid b} \qquad x = a2^{n/2} + b$$

$$y = \boxed{c \mid d} \qquad y = c2^{n/2} + d$$

Then we have:

$$\begin{aligned} xy &= (a2^{n/2} + b)(c2^{n/2} + d) \\ &= ac2^n + (ad + bc)2^{n/2} + bd \end{aligned}$$

Furthermore, we can evaluate the products in a different way:

$u := (a + b)(c + d)$;
$v := ac$;
$w := bd$;
$z := v2^n + (u - v - w)2^{n/2} + w$;

Then $z = xy$ and we get the product expressed in radix form again.

Ignore for the moment that we might have carries from the multiplications, then the evaluation of the product xy requires

3 multiplications of $n/2$-bit numbers.

6 additions of $n/2$-bit numbers (which take $n/2$ steps each).
some shifts (which are easy to do)

So the time for a n-bit multiplication satisfies:

$$T(n) = 3T(n/2) + 6n/2$$

and the master recurrence solution gives (as $3 > 2$)

$$T_n = O\left(n^{\log_2(3)}\right) \sim O\left(n^{1.59}\right)$$

(This estimate still holds if we consider carries.)
The clever evaluation of the product
$(a2^{n/2} + b)(c2^{n/2} + d)$

uses only 3 multiplications instead of 4. This 25% saving is compounded and thus accounts for the improved complexity.

Another example is given in [M 5.6].

Strengthening the Hypothesis

Problem: [M 5.8] Given a sequence $X = (x_1, x_2, \ldots, x_n)$ of (not necessarily positive) real numbers, find a subsequence $x_i, x_{i+1}, \ldots, x_j$ of consecutive elements such that the sum over its elements is the maximal sum possible.

We want to design an algorithm to solve this problem

Induction hypothesis: We know how to find the maximum subsequence in a sequence of length $< n$.

Problem: what if the maximum subsequence of $x_1, \ldots, x_{n-1}$ is $x_i, \ldots, x_j$ ($j < n-1$) but the maximum subsequence of $x_1, \ldots, x_n$ involves x_n (i.e. x_n is large)?

Stronger Induction hypothesis: We know how to find the maximum subsequence in a sequence of length $< n$ and the maximum subsequence that is a suffix.

(For simplicity the following algorithm only computes the *sum* of the subsequence:)

Input: X (array of length n)
Output: $MaxVal$ (The sum over a maximum subsequence).
begin
$MaxVal := 0$;$MaxSuff := 0$;

```
    for i ∈ [1..n] do
        if x[i] + MaxSuff > MaxVal then
            MaxSuff := MaxSuff + x[i];
            MaxVal := MaxSuff;
        elif x[i] + MaxSuff > 0 then
            MaxSuff := MaxSuff + x[i];
        else
            MaxSuff := 0;
        fi;
    od;
end
```

The technique we have used – *strengthening the induction hypothesis* – is used often, when instead of $P(< n) \Rightarrow P(n)$ it is easier (or only possible) to prove:
$P(< n)$ and $Q(< n) \Rightarrow P(n)$.

However then $Q(n)$ must become part of the induction hypothesis, we must in fact prove:

$P(< n)$ and $Q(< n) \Rightarrow P(n)$ and $Q(n)$.

Dynamic Programming

Divide-and-Conquer is a useful technique if we can split a problem in proportionally smaller parts (i.e. in a problem of size $\frac{n}{a}$). Sometimes however the only possible reduction is by a constant, (i.e. we can reduce a problem of size n to several problems of size $n-a$). Take for example the fibonacci numbers:

function fibonacci(n)
begin
  if $n < 3$ **then return** 1; **fi**;
  return fibonacci($n-1$)+fibonacci($n-2$);
end

A simple recursive approach for this type of problems will have a runtime satisfying $T(n) = aT(n-1) + b$, such a recurrence relation has a solution with $T(n) = O(a^n)$. (See the time estimate for the "Towers of Hanoi" problem.)

The reason for the exponential growth is, that we do the same subtasks again and again (the problems of size $n-1$ might *all* call the same problem of size $n-2$ recursively).

We can therefore save time if we remember all solutions of subproblems we have computed so far in a table. This technique is called "Dynamical Programming".

```
l := [];
function fibonacci(n)
begin
  if IsBound(l[n]) then return l[n];
  elif n < 3 then return 1; fi;
  l[n] :=fibonacci(n − 1)+fibonacci(n − 2);
  return l[n];
end
```

Now the call to fibonacci($n-2$) only takes constant time (because the value has been computed before and stored in the call to fibonacci($n-1$)). Therefore we have that $T(n) = T(n-1) + a$, solving this recurrence yields $T(n) = O(n)$.

Eventually we trade space (the table of subresults) against time. Therefore this approach only works if the number of subproblems is not too large.

The Knapsack Problem

Given an integer K and n integers $1 \leq s_i \leq K$, find a subset of $S = \{s_i\}$ such that the sum over these s_i is exactly K (or determine that no such set exists).

Induction hypothesis: We know how to solve the problem for $n - 1$ items.

If there is a solution for the first $n - 1$ numbers we take it. If not we have to try with using the n-th number. This however will reduce the remaining free space.

We write $P(i, k)$ for a problem with fitting the first i numbers and a knapsack of size k.

Induction hypothesis (second attempt): We know how to solve the problem $P(n - 1, k)$ for all $0 \leq k \leq K$.

We implement the algorithm using dynamical programming:

Input: S (the items) and K (the knapsack)
Output: A two dimensional array P with $P[i,k].exist$ =true if there exists a solution with the first i items for a knapsack of size k and $P[i,k].belong$ =true

```
begin
  P[0,0].exist := true;
  for k ∈ [1..K] do
    P[0,k].exist := true; {P[i,0] will be computed and
    thus does not need to be initialized}
  od;
  for i ∈ [1..n] do
    for k ∈ [0..K] do
      P[i,k].exist := true;{default}
      if P[i-1,k].exist then
        P[i,k].exist := true;{Can do with subset}
        P[i,k].belong := false;
      elif k - S[i] ≥ 0 then
        if P[i-1,k-S[i]].exist then
          P[i,k].exist := true;{extend solution}
          P[i,k].belong := true;
        fi;
      fi;
    od;
  od;
end
```

Summary

- Use the idea behind induction to solve problems. Reduce the problem instead of immediate solving.
- The simplest reduction is to remove one element (or merge two elements). Sometimes one has to look for a suitable element to remove.
- There may be different ways of reduction. It might pay off to use some initial work to find the best way for reduction.
- A straightforward reduction might not give sufficient information to extend the result, in this case one can try to strengthen the induction hypothesis.
- A particular efficient way of reduction is to reduce the problem to subproblems of equal size (so each subproblem will be much smaller).
- A reduction to several subproblems might eventually compute the same subproblems again and again. In this case try to store these intermediate results if memory permits.
- If a problem cannot be solved directly, try to find an easier versions of the problem you can solve and then try to strengthen or extend this solution.

III. Sorting and Searching

- Often done. (60's estimate: 25 percent of runtime.)
- Examples of design paradigms.
- Used to build other algorithms.

Binary Search With a comparisons, we can distinguish 2^a elements. Binary search in a list of size n uses $\log_2(n)$ comparisons.

Variations:

- Binary search in list of unknown size: Double the search space if bigger. To find an element in the i-th position we need $\log_2(i)$ comparisons. May also be used if we suspect an entry to be at the beginning of a long list.
- For large list of uniformly distributes entries, *interpolation search* can be used. This takes in average (over all sequences) $O(\log \log n)$ comparisons. This however is worth only if n is *VERY* large for the access time. [M. 6.3]

Binary search as building block

Stuttering Subsequence For a sequence B define $B^{[i]}$ to be the sequence in which each letter occurs i times: $B = xyzzx$, $B^{[3]} = xxxyyyzzzzzzxxx$.

Given two sequences A and B, $|A| = n$, $|B| = m$, find the maximal value of i such that $B^{[i]}$ can be embedded as a subsequence (leaving holes) in A.

We can check whether B is a subsequence in $O(n+m)$ (linear scanning both sequences).

The range of i to consider is $1, \ldots, \lfloor n/m \rfloor$.

We can use binary search to determine the maximal i, this takes $\lceil \log_2(n/m) \rceil$ test runs. So we can solve the problem in time $O((n+m)\log(n/m)) = O(n \log(n/m))$.

The same technique can be used in other cases to find a maximal parameter value. However it does not necessarily yield the optimal solution.

Solving Equations Solve $f(x) = a$ for monotonic function f (bisection, BOLZANO method).

Sorting

Problem: Given n numbers $X = [x_1, \ldots, x_n]$, arrange them in increasing order (using as few comparisons as possible).

The algorithm is called *in-place* if no additional work space is needed for intermediate results.

It is called *stable* if equal elements remain in their original order.

Assumptions: We can keep all data in main memory, access to every x_i takes the same time. (Not true for large n: Cache, RAM, Disk, Tape. See [K, 5.4] for techniques.)

When sorting larger objects it is worth to sort only pointers (and not to move the objects) (*list sorting*).

The most obvious way may be "Bubble Sort":

for $i \in [1, \ldots, n]$ **do**
  for $j \in [i + 1, \ldots, n]$ **do**
    if $X[j] < X[i]$ **then** swap $X[i]$ with $X[j]$; **fi**;
  od;
od;

This algorithm is not very good, it takes
$n(n-1)/2 = O\left(n^2\right)$ comparisons.

Bucket Sort and Radix Sort

Mailroom: One box for each recipient. For n items in the range $1, \ldots, m$ we need m containers. Sorting takes $O(n+m)$ steps (arrange elements and collect containers). Very efficient if m small, Storage problems when m large.

Extension: Radix Sort. Sort *lexicographically* by iterated bucket sort. If the data structure to hold the buckets is ordered, sorting *from right to left* needs only one pass for each stage.

Induction hypothesis We can sort elements with $< k$ digits.

Assume we have a list of elements sorted according to their last $k-1$ digits. Scan all elements again and sort them according to the k-th digit in d buckets. Then collect all buckets in order (*straight-radix sort*). Then the elements are sorted according to their last k digits.

For intermediate storage the implementation uses a queue. We then run k times over the queue of n elements, so the runtime is $O(nk)$.

Insertion and Selection Sort

We now try to reduce to problems with a smaller n. The base case $n = 1$ is always trivial.

Suppose we can sort $n - 1$ elements. Then we can scan these numbers to find the right place for the n-th element ("*Insertion Sort*"). It is probably the simplest algorithm and fairly effective for small n, taking $\sum_{i=1}^{n-1} i = n(n-1)/2 = O\left(n^2\right)$ comparisons.

We can improve on this using binary search. So in total we have $O\left(n \log n\right)$ comparisons.

However we also need to make space to insert the n-th element. In the worst case we have to move $n(n-1)/2$ elements, thus the algorithm remains quadratic.

Alternatively we can try to remove a special element, the smallest one. We can put it immediately in the right place. This needs at worst $n - 1$ data movements, however we need $O\left(n^2\right)$ comparisons.

Further improvements are possible if we use balanced trees.

Mergesort

So far we have considered scanning and placing for each number separately. While we scan for the right place for one number, however we may place *several* numbers. This suggests the following divide-and-conquer approach: Sort two subsets and then merge them.

Merging two sorted sequences of size m and n can be done with $O(m+n)$ operations (if storage for the result is available).

This suggests a divide-and-conquer approach: Split the list in two parts, sort both, then merge the sorted results to get an overall sorted list.

(See [M, p.132] or [K, 5.2.4] for a more explicit algorithm.)

procedure mergesort(a,b)
begin
  if $a = b$ **then**
    if $X[j] < X[i]$ **then** swap $X[i]$ with $X[j]$; **fi**;
  else
    $c := \lfloor \frac{a+b}{2} \rfloor$;
    mergesort(a,c);
    mergesort(c+1,b);
    merge $X[1], \ldots, X[c]$ with $X[c+1], \ldots, X[b]$ into a new list L.
    for $i \in \{1, \ldots, n\}$ **do** $X[i] := L[i]$; **od**;
  fi;
end

Assume for simplicity that n is a power of 2. Then we have that

$$T(n) = 2T(n/2) + c \cdot n, \qquad T(n) = O(n \log n)$$

Which is asymptotically better than $O\left(n^2\right)$. However the merging needs extra space. (One can improve this to constant extra space.)

The sorting is stable and can be easily parallelized.

Quicksort

A problem of mergesort is the extra memory as we cannot predict where elements will end up. We therefore try to make the division in a more intelligent way.

Suppose we know a number x (a *pivot*) such that half the elements are larger and half are smaller than x. If we partition the elements accordingly, both halfs are already in the right place. We then recurse to both halfs.

We can achieve such a partition by using two pointers L and R to the list which initially point to te right and left end of the list:

Induction hypothesis: At step k of the algorithm, $x \geq x_i$ $(i < L)$ and $x < x_j$ $(j > R)$.

In step $k+1$ we move either L or R towards each other without invalidating the hypothesis: The only situation in which we cannot move either is if $x_L > x$ and $x_R < x$. We then swap x_L and x_R.

Eventually the pointers will meet, this is the place where the pivot must be placed.

procedure partition(X,$Left$,$Right$);
begin

```
  assign pivot; swap it in position Left.
  L := Left; R := Right;
  while L < R do
    while X[L] ≤ x and L ≤ Right do L := L + 1; od;
    while X[R] > x and R ≤ Left do R := R − 1; od;
    if L < R then swap X[L] with X[R]; fi;
  od;{If no entries are equal the pointers now have crossed each
  other. As the pivot still resides in the left half, we have to put
  it in the place pointed to by R.}
  Middle := R;
  swap X[Left] with X[Middle];
end
procedure quicksort(X,Left,Right);
begin
  if Left < Right then
    partition(X,Left,Right);
    quicksort(X,Left,Middle − 1);
    quicksort(X,Middle + 1,Right);
  fi;
end
```

The runtime depends crucially on the selection of the pivot. If it always splits the sequence in half, we have that
$T(n) = 2T(n/2) + O(n)$.

Finding the median would take too long. In practice one therefore picks a fixed index element – say the first – (or a

random element) to be the pivot.

Suppose however the sequence is already sorted (and we pick the first element). Then the runtime becomes $T(n) = T(n-1) + n = O\left(n^2\right)$. *If we suspect that the input may be in nonrandom order, quicksort must pick a random pivot to minimize the chance of this!*

Assume that each x_i has the same probability $(1/n)$ to be picked as a pivot. The runtime if the i-th smallest element is the pivot is

$$T(n) = n - 1 + T(i-1) + T(n-i)$$

So the average runtime satisfies $T(n) = O(n \log n)$.

The bookkeeping of quicksort becomes too expensive when n is small. It is therefore worth to choose the base case of the induction differently and to switch for small n (say $\sim 10 - 20$) to another sort (for example insertion sort). (improves constant)

The runtime depends crucially on the selection of the pivot. If it always splits the sequence in half, we have that $T(n) = 2T(n/2) + O(n)$.

Finding the median would take too long. In practice one therefore picks a fixed index element – say the first – (or a random element) to be the pivot.

Suppose however the sequence is already sorted (and we pick the first element). Then the runtime becomes $T(n) = T(n-1) + n = O\left(n^2\right)$. *If we suspect that the input may be in nonrandom order, quicksort must pick a random pivot to minimize the chance of this!*

Assume that each x_i has the same probability $(1/n)$ to be picked as a pivot. The runtime if the i-th smallest element is the pivot is

$$T(n) = n - 1 + T(i-1) + T(n-i)$$

So the average runtime satisfies $T(n) = O(n \log n)$.

The bookkeeping of quicksort becomes too expensive when n is small. It is therefore worth to choose the base case of the induction differently and to switch for small n (say $\sim 10 - 20$) to another sort (for example insertion sort). (improves constant)

Heapsort

Suppose we conduct a knock-out tournament to determine the largest element of a set. We then remove the largest element x from the set and conduct another knock-out tournament to determine the second largest element (and so forth). For this however we do not need to conduct a whole tournament again, but only redo those decisions in which x was involved.

A *priority queue* is an abstract data structure that supports the operations:

Insert(x): insert an entry x in the data structure
Remove(): remove the largest entry from the data structure.

We will implement them using binary trees.

Definition: A *heap* is a binary tree in which the key of each node is greater or equal than the keys of its children.

When balancing the tree we can store the tree in an array A: $A[1]$ contains the root and if $A[i]$ contains the key of a node, $A[2i]$ and $A[2i+1]$ contain the keys of its children. (This corresponds to enumerating the nodes level by level from left to right.)

Using this data structure the heapsort algorithm looks like this

Input: : A (array of length n)
Output: : A (the array in sorted order)
begin
  Build heap in A;
  for $i \in [n, n-1, \ldots, 2]$ **do**
    remove x from heap A; {The heap now consists of the entries 1 to $i - 1$}
    $A[i] := x$; {next element in its position}
  od;
end

This algorithm is clearly in-place. We now concentrate on the subtasks:

Remove: By the heap property the largest key is in $A[1]$. When we remove it $A[2..n]$ contains two heaps. Now place $x := A[n]$ in position 1 and decrement n. To restore the heap property propagate x down the tree until is is in an acceptable position, then the heap property is fulfilled again. The inductive step is to compare the root node with the values of its children and – if it is not maximal – swapping it with the largest one (the other subtree remains unchanged). We then recurse on the subtree whose root we have replaced. The maximal number of comparisons is $2\lceil \log_2 n \rceil$.

In heapsort we remove every element once. Thus the runtime of the loop in heapsort is $O(n \log n)$.

Insert: The algorithm is similar: Suppose the heap contains $n-1$ elements. We insert the new element x in the new leaf n, then compare the leaf with its parent and swap if necessary. The new entry x thus raises to the correct position. The maximum number of comparisons needed is $\lfloor \log_2(n+1) \rfloor$.

Building a heap: There are two natural ways of building a heap, top-down an bottom-up. They correspond to scanning the array A from left to right or from right to left.

Hypothesis (↓): The array $A[1, \ldots, i]$ is a heap.

The base case is trivial, incorporating the next entry in the heap is one **Insert** operation, it takes in the worst case $\lfloor \log_2 i \rfloor < \lfloor \log_2 n \rfloor$ comparisons. This happens n times, so we have $O(n \log n)$ comparisons in total.

In the bottom-up approach the tail $A[i..n]$ represents several heaps.

Hypothesis (↑): All the trees represented by the array $A[i+1, \ldots, n]$ are heaps.

Again the base case is trivial. However the whole array $A[\lfloor n/2 \rfloor + 1..n]$ represents leaves, which are heaps of size 1. Thus the induction needs to start only at $\lfloor n/2 \rfloor$. Half the work is trivial.

Now we prepend $A[i]$. It has at most two children, $A[2i+1]$ and $A[2i]$ which are roots of heaps. We now let $A[i]$ sink (like in the **Remove** routine). The height of $A[i]$ is $\lfloor \log_2(n/i) \rfloor$, the number of comparisons is at worst twice the height.

Let $H(i)$ be the sum of heigths in a tree of height i. Then we have:

$$H(i) = 2H(i-1) + i, \qquad H(0) = 0$$

A solution (guess the result and prove by induction!) is $H(i) = 2^{i+1} - (i+2)$. In contrast the number of nodes in a tree of height i is $2^{i+1} - 1$. Therefore the bottom-up approach uses only $O(n)$ comparisons.

Heapsort is *guaranteed* to run in time $O(n \log n)$ and does not need extra memory. In average its time is $23.08n \log n$ in comparison to quicksort's average $11.67n \log n$ [K 5.2.3].

A lower Bound for Sorting

All the sort algorithms seen so far which are based on comparisons use $O(n \log n)$ comparisons. Can we do better?

To investigate this we examine the amount of information a sort algorithm produces from an input:

Lemma: A list of n elements can be arranged in

$$n! = n(n-1)(n-2)\cdots 2 \cdot 1$$

different ways. ($n!$ is called "n-factorial".)

Proof: n choices to place the first element, $n-1$ remaining choices to place the second and so forth.

Lemma: A sort algorithm identifies one object in a set of size $n!$.

Proof: Take the set of all arrangements of the list we want to sort. The algorithm finds the arrangement in which the list is sorted.

Lemma: A set of a "yes/no" questions can identify single elements in a set of size at most 2^a.

Proof: Let S be a set of size m. A "yes/no" question abouts the elements of S splits S in two parts, the "yes" part and the "no" part. We then continue to split one of the parts with the next question.

At least one of the parts of S is of size at least $\lceil m/2 \rceil$, so after a questions we have identified a subset of size at least $\lceil m/2^a \rceil$. To be able to identify single elements, this subset must be of size 1.

Lemma: Comparison of elements is a "yes/no" question.

Proof: The question is: "Is $a > b$"?

Theorem: If we can sort a set of n objects $x_1, \ldots, x_n$ using k comparisons then

$$n! \leq 2^k$$

Proof: There are $n!$ possible orderings. The k comparisons allow to identify one of them uniquely.

Lemma: (Stirling estimate)

i) $n^n \geq n! \geq (n/2)^{(n-2)/2}$

ii) If $N \geq 3$ then $2^{(N2^N)} \geq 2^N! \geq 2^{(N-1)2^{N-2}}$

Now suppose we sort n objects with k comparisons and let $2^N \geq n \geq 2^{N-1}$. Then

$$2^k \geq n! \geq 2^{N-1}! \geq 2^{(N-2)2^{N-3}}$$

and thus (taking logarithms):

$$k \geq (N-2)2^{N-3} \geq N2^{N-4} = N2^N/16$$

Theorem: Any method of sorting n numbers requires at least $n \log_2(n)/16$ comparisons. In particular every sort algorithm based on comparisons requires $\Omega(n \log n)$ comparisons!

(Taken from T. Körner, The Pleasures of Counting, CUP, p.287f.)

This bound only affects sorting algorithms that are purely based on comparisons of elements and when the input can be completely arbitrary (so a partial result allows no prediction whatsoever about the placements of the remaining elements). If the elements to be sorted are from a predefined, restricted range, other sort methods (for example radix sort) can be quicker.

String Matching

Take two strings A and B which we consider as a list of characters: $A =$
$B = b_1 b_2 \cdots b_m$, $m \leq n$. We try to find the first occurrence of B as a
that is the smallest k such that for all $1 \leq i \leq m$ we have $a_{i+k-1} = b$
to denote the first i letters of B.

A naïve algorithm has runtime $O(mn)$. Take for example:

$x \quad y \quad x \quad x \quad y \quad x \quad y \quad x \quad y \quad y \quad x \quad y \quad x \quad y \quad x \quad y \quad y$

$x \quad y \quad x \quad y \quad y \quad x \quad y \quad x \quad y \quad x \quad x$

Every time we find a mismatch we check B again (now displaced) versus
bad case happens in $A = yyyyyyyyyyyyx$, $B = yyyyyx$.) If we ge
however we have already compared a part of B with a part of A. We wo
this information to increment the index as far as possible.

Suppose we get a (partial) match of $B(i)$ to A at position k which fails
$(i+1)$-th letter of B. So the next test will check at a position $k' > k$.
is still in the range of the i letters from k. Then $B(i)$ (which we know t
position k) must overlap the new attempt of matching B:

A	1	2	$\dots$	k		$\dots$		$k+i-1$	$k+i$
B				b_1	$\dots$	b_j	$\dots$	b_{i-1}	b_i
									$\uparrow$
B						b_1	$\dots$	b_{i-j}	b_{i-j+1}
						$\uparrow$		$\uparrow$	

For this new attempt to be successful we must get a match at the marke
There A already agrees with B. In other words: The tail of $B(i)$ must
head of B.

In such a situation however we only need to check if the remaining tail of B matches.

Assuming that $m \ll n$ we can pre-process B to compute head-tail overlap values. We set the maximum head-tail overlap for a mismatch at position i to be:

$$next(i) = \begin{cases} \max & \{0 < j < i-1 \mid b_{i-1} \cdots b_{i-j} = B(j)\} \\ 0 & \text{if no such } j \text{ exists} \\ -1 & i = 1 \end{cases}$$

In our example we have:

i	1	2	3	4	5	6	7	8	9	10	11
B	x	y	x	y	y	x	y	x	y	x	x
$next$	-1	0	0	1	2	0	1	2	3	4	3

If the match for b_i fails, we try to match the same letter of A against $b_{next(i)+1}$. (This also works if the first letter fails, as we set $next(1) = -1$.)

This approach gives the following algorithm ("KMP", Knuth, Morris, Pratt 1977):

Input: A, B (arrays of lengths n and m), $next$.
Output: s (smallest substring index or 0).

```
begin
  j := 1;i := 1;
  s := 0;
  while s = 0 and i ≤ n do
    if B[j] = A[i] then
      i := i + 1;j := j + 1;
    else
      j := next[j] + 1;
      if j = 0 then
        j := 1;i := i + 1;
      fi;
    fi;
    if j = m + 1 then s := i − m; fi;
  od;
end
```

If we mismatch a_i against b_j, we try to match a_i against $b_{next(j)+1}$ until find a match or arrive at index 0 (and increment i). Each of this steps decrements the index in B, so it can happen at most j times. However to have reached j we must have gone forward in A j times *without* mismatches: for every step in A we do 2 letter comparisons. Thus the algorithm runs in $O(n)$ (independent of m !).

Computing $next$

Here we use induction again. Assume that the values of $next$ for $1, \ldots, i-1$ are known and that we want to compute $next(i)$.

At best, $next(i) = next(i-1) + 1$, this happens if $b_{i-1} = b_{next(i-1)+1}$ and we extend a head.

Otherwise we need to find a new tail that is equal to a head. This is a string comparison problem.

We know already that $b_1 \cdots b_{next(i-1)}$ matches the tail of $B(i-2)$.

But $b_{i-1} \neq b_{next(i-1)+1}$ is simply a mismatch comparing B against itself.

We now can proceed like in the comparison algorithm and use the values of $next$ computed so far:

If we mismatch at $next(i-1)+1$, we go to $x = next(next(i-1)+1)$, trying to match b_{i-1} against b_{x+1}.

If they match we set $next(i) = x + 1$, otherwise we continue in the same fashion. If we arrive at index 0 we have to start from scratch at the next position.

This gives rise to the following short algorithm:

Input: B of length m
Output: $next$.
begin
  $next(1) := -1$; $next(2) := 0$;
  for $i \in [3..m]$ **do**
    $j := next(i-1)+1$;
    while $b_{i-1} \neq b_j$ and $j > 0$ **do**
      $j := next(j)+1$;
    od;
    $next(i) := j$;
  od;
end

Improvements are possible if only two letters occur: A mismatch automatically matches the other letter.

Another algorithm for the ame problem was suggested by Boyer and Moore. We match from the right (starting with b_m). If the corresponding a_i does not match we can increment the start index as far so that we find the letter a_i in B. (Becomes more difficult with partial matches.) If there are many letters this algorithm is likely (though not guaranteed) to perform *less* steps than n.

Sequence Comparison

We again take two strings $A = a_1a_2 \cdots a_n$ and $B = b_1b_2 \cdots b_m$. This time we want to find how different A and B are. We allow three *edit steps* to change A to B:

a) Insert a character
b) Delete a character
c) Replace a character with another one

We assume each step carries the same price, we want to minimize the number of steps.

This problem has applications in file comparisons (UNIX "diff"), revision maintainance or biology).

To permit an inductive approach, we examine how the best edit sequence affects the last letters of A and B.

Denote by $A(i)$ $(B(j))$ the substrings of the first i (j) letters, let $C(i, j)$ be the minimum number of edit steps to change $A(i)$ to $B(j)$.

We are eventually interested in finding $C(n, m)$ and thus try to relate $C(i, j)$ to $C(i', j')$ for $i' < i$ or $j' < j$.

We have four possible scenarios on what happens to a_n or b_m in the optimal edit sequence:

match: If $a_n = b_m$ then $C(n, m) = C(n-1, m-1)$.

replace: If $a_n \neq b_m$ and the minimum change replaces a_n by b_m then $C(n, m) = C(n-1, m-1) + 1$.

delete: If a_n is deleted, we can as well change $A(n-1)$ to $B(m)$ and delete a_n afterwards. Thus $C(n, m) = C(n-1, m) + 1$.

insert: If b_m is inserted we can as well change $A(n)$ to $B(m-1)$ and insert b_m at the end, $C(n, m) = C(n-1, m) + 1$.

Denoting $c(i, j) = \begin{cases} 0 & \text{if } a_i = b_j \\ 1 & \text{if } a_i \neq b_j \end{cases}$ we get:

$$C(n, m) = \min \begin{cases} C(n-1, m) + 1 & \text{delete } a_n, \\ C(n, m-1) + 1 & \text{insert } b_m, \\ C(n-1, m-1) & \\ \qquad\qquad +c(n, m) & \text{replace/match}, \\ n & m = 0, \\ m & n = 0 \end{cases}$$

and we can write a recursive algorithm.

However...

This algorithm would recurse three times, leading to an exponential runtime.

The total number of subproblems however is only nm. This is again a case for *dynamic programming*.

Input: A, B (arrays of lengths n and m)
Output: C (cost matrix)

```
begin
  for i ∈ [0..n] do C[i, 0] := i; od;
  for j ∈ [1..m] do C[0, j] := j; od;
  for i ∈ [1..n] do
    for j ∈ [1..m] do
      del := C[i − 1, j] + 1;
      ins := C[i, j − 1] + 1;
      if a_i = b_j then
        repl := C[i − 1, j − 1];
      else
        repl := C[i − 1, j − 1] + 1;
      fi;
      C[i, j] := min(del, ins, repl);
    od;
  od;
end
```

IV. Graph Algorithms
Bipartite Matching

A graph $G = (V, E)$ is called *bipartite* if we can separate its vertices V in two sets X and Y such that there is no edge between elements of X or between elements of Y.

For example take X people who are interested in buying Y houses, an edge says that person x is interested in house y.

A *matching* of G is a set of edges from E (so every edge connexts X with Y), such that two edges have no vertex in common.

Problem: Given a bipartite graph G, find a matching with the largest possible number of edges (a *maximum matching*).

The most simple approach is to match as long as we can (a so-called *greedy algorithm*.) This might produce very bad results.

A more careful attempt might try to match "problematic" vertices first, but this becomes very hard to analyze.

Instead we will try to improve a matching which cannot be enlarged but which is not optimal.

Take a vertex $x \in X$ which is not matched. Choose a vertex $y \in Y$ adjacent to x (which was matched with w before), dissolve its matching and match it with x. (So we are not worse than before.)

If w has an unmatched neighbour, we match and thus have extended the matching, otherwise we can iterate.

To build an algorithm we have to formalize this procedure, show that it terminates and have to show that it produces an optimal solution.

Definition: An *alternating path* P for a matching M is a path in G from a vertex $x \in X$ to a vertex $y \in Y$ (where x and y both are unmatched in M) such that the edges in P are alternatively in $E \setminus M$ and M. (I.e. ($\{x, z\} \notin M$, $\{z, w\} \in M$ etc.)

The number of edges in P must be odd (we change from X to Y) and P has one more edge in $E \setminus M$ than in M.

Thus we can swap the edges in P in and out of M and obtain a larger matching. The converse also holds:

Theorem: A matching is maximum if and only if it admits no alternating paths.

We will prove this theorem later in a more general context.

This suggests immediately an algorithm: start with a greedy algorithm to find one matching. Then look for alternating paths and modify the matching.

Each step increases the matching, there are at most $\lfloor \frac{n}{2} \rfloor$ edges in a matching, so we have at most $\lfloor \frac{n}{2} \rfloor$ iterations.

To find an alternating path consider the graph as an directed graph G' with edges in M going from X to Y, edges in $E \setminus M$ going from Y to X. An alternating path in G then is a directed path in G' from an unmatched vertex in Y to an unmatched vertex in X.

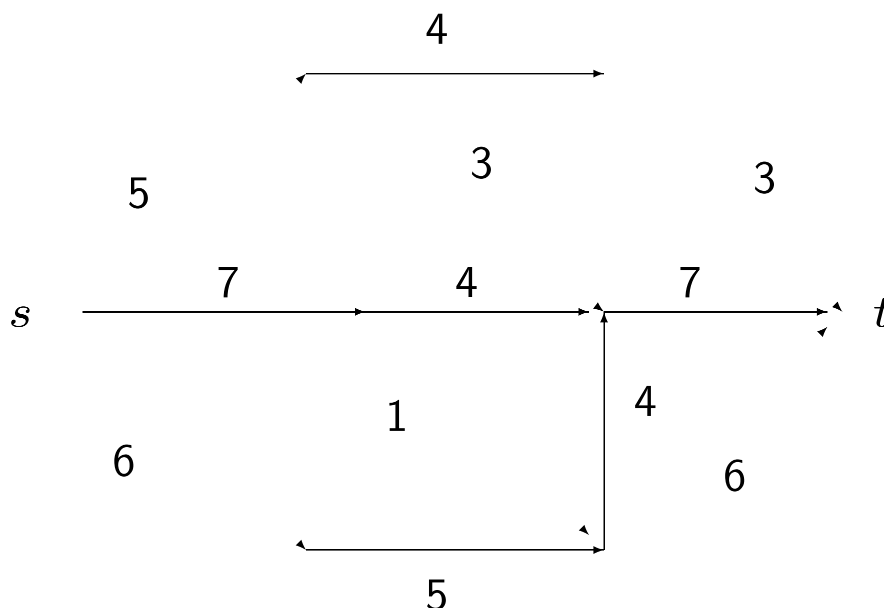
Such a path can be found by graph search algorithms in time $O\left(|V| + |E|\right)$, so the maximum matching algorithm runs in time $O\left(|V|\left(|V| + |E|\right)\right)$.

One can improve this to $O\left(\sqrt{|V|}(|V| + |E|)\right)$ by looking for several alternating paths at once.

Network Flow

Suppose we have a transport network (Railways, Roads, Pipelines, Phone or Data Networks) consisting of point-to-point connections. We use this network to transport from place s (source) to place t (sink), we want to find out how much we can transport (and how we have to route objects).

We model the network as a directed graph $G = (V, E)$ with two distinguished vertices s and t. We associate to each edge $e \in E$ a positive weight $c(e)$, the *capacity* of e to indicate the maximal possible flow through this edge. (Such a graph is called a *network*.)



A *flow* on G is a function f on the edges that satisfies:

a) For every $e \in E$ we have $0 \leq f(e) \leq c(e)$.

b) For all $v \in V$, $v \neq s, t$ we have
$\sum\limits_{u \in V} f(u, v) = \sum\limits_{w \in V} f(v, w)$ (*Kirchhoff's law*).

Again we want to identify paths in the graph that help to improve a given flow.

There are two potential ways to increase the total flow from from s to t:

1. Increase flow in the right direction.
2. Decrease flow in the wrong direction.

An *augmenting path* (with respect to a given flow f) is a directed path from s to t (not necessarily following the edges in their direction) such that for every edge (v, u) in this path holds either:

1. (v, u) is the same direction as in G (a *forward edge*) and $f(v, u) < c(v, u)$. (We call $c(v, u) - f(v, u)$ the *slack*.) Or:

2. (v, u) is the other direction as in G (a *backward edge*) and $f(v, u) > 0$.

If f *admits* an augmenting path, f is not maximum:

1. If all edges are forward edges, we can increase the total flow by the minimum slack.
2. If there are backward edges take m the minimum of the minimal slack and the minimal flow values of backward edges. We decrease the flow through all backward edges by m. At every start of a (series of) backward edges thus there is m extra flow, which we direct through a forward edge. Vice versa at the end of the (series of) backward edges, flow m is missing, which we supply via forward edges.

Theorem: A flow f is maximum if and only if it does not admit an augmenting path.

For a set of vertices $A \subset V$ with $s \in A, t \not\in A$, we define the *cut* induced by A to be the set of those edges $(v, w) \in E$ that connect a vertex in $v \in A$ with a vertex $w \not\in A$. The *cut value* of this cut is

$$\sum_{e \in \text{this cut}} c(e).$$

As everything going from s to t must pass a cut, no flow can have a higher value than the cut. (Reduce all edged in the cut to capacity 0, and nothing will flow.)

Take a flow f which does not admit an augmenting path. We will construct a cut whose capacity is that of f – so f must be a maximal possible flow.

$$A = \{v \in V \mid v = s \text{ or augmenting path from } s \text{ to } v \text{ exists}\}$$

Then $s \in A$, $t \notin A$, so A defines a cut.

For every edge (v, w) in that cut we have either

- The edge is forward and $f(v, w) = c(v, w)$ (otherwise extend augmenting path to w). Or
- The edge is backward and $f(w, v) = 0$ (otherwise reduce flow, this extends an augmenting path from v to w).

This proves also the following important theorem:

Max-Flow Min-Cut Theorem: (Ford and Fulkerson, 1956) The value of a maximum flow in a network is equal to the minimum capacity of a cut.

It also implies:

Integral Flow Theorem: If the capacity of all edges in the network are integers, there is a maximum flow whose value is an integer.

To compute a maximum flow we start with one flow and search for augmenting paths. To find augmenting paths, we define a new graph $R = (V, F)$ with $(v, w) \in F$ if either $(v, w) \in E$ and $f(v, w) < c(v, w)$ or $(w, v) \in E$ and $f(w, v) > 0$.

An augmenting path for f is a path in R from s to t, we can construct R in $|E|$ steps.

But: The extension process may be repeated very often (bounded only by the maximum weights).

If we take the path with the smallest number of edges, it can be shown that at most $(|V|^3 - |V|)/4$ augmentation steps are required (Edmonds, Karp, 1970). Further improvements are possible.

Reduction of bipartite matching

We finally show that one can consider the maximum matching problem of the previous section as a flow problem:

Given a bipartite graph $H = (W_1 \cup W_2, F)$ we add two new vertices s and t and connect s to all vertices in W_1 and all vertices in W_2 to t. We also direct the edges in F to go from W_1 to W_2. We call the resulting graph $G = (V, E)$ and assign capacity 1 to all edges in E.

Every matching M in H then gives a flow, the number of edges in the matching is the flow value. We want to show that this flow is a maximum flow if and only if M is a maximum matching.

If the maximum flow corresponds to a matching this matching is certainly maximal.

On the other hand an alternating path in G corresponds to an augmenting path in H and vice versa.

This proves the theorem about bipartite matchings.

V. The Fast Fourier Transform

Fact: A function that varies over time can be composed from sinus and cosinus functions.

The infinite sum becomes an integral, we have

$$(*) \qquad f(t) = \int_0^\infty A(\omega)\sin(\omega t)d\omega + \int_0^\infty B(\omega)\cos(\omega t)d\omega$$

The functions A and B indicate the weight of the various frequencies.

Interpretation over the complex numbers

Things become more uniform if we move to the complex numbers:

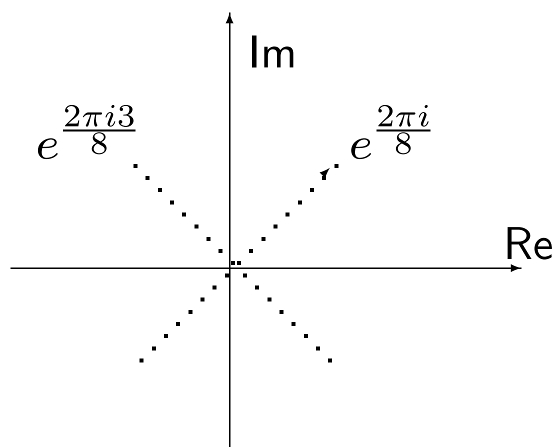
Fact: Sinus and Cosinus are closely related with the exponential function, we have (over the complex numbers): $\sin(x) = \frac{e^{ix}-e^{-ix}}{2i}$, $\cos(x) = \frac{e^{ix}+e^{-ix}}{2}$ and

$$e^{a+ib} = e^a \cdot (\cos b + i \sin b)$$

In particular for every natural n and $0 \leq m < n$ the numbers:

$$e^{\frac{2\pi i m}{n}}$$

lie equally distributed on the complex unit circle. We call these numbers *Roots of Unity*.



The Fourier transform

Using these relations, we can write $(*)$ in the form

$$f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega) e^{-i\omega t} d\omega$$

We call g the *Fourier transform* of f and write $g = \mathcal{F}(f)$. We call similarly $f = \overline{\mathcal{F}}(g)$ the *Inverse Fourier transform*.

Fact:

$$g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{i\omega t} dt$$

The Fourier transform has many applications in signal processing. We can use it for example to filter low frequencies:

Perceptual Coding

Another application is in audio recording.

The human ear cannot recognize frequencies with low amplitude, if another frequency "close by" has high amplitude.

When recording the audio signal, we can remove such "masked" frequencies, this data reduction cannot be spotted by the ear.

The same technique applies to pictures (with x/y-coordinates for time).

This principle is used for example in the MiniDisc and the MPEG standards.

Discretization

On a computer we cannot take infinitely many values but have to *discretize* an continuous process.

Let $\omega := e^{\frac{2\pi i}{n}}$. Then ω fulfills:

$$1. \quad \omega^n = 1, \qquad \omega^m \neq 1 (1 \leq m < n)$$

$$2. \qquad \sum_{j=0}^{n-1} \omega^{jp} = 0 \text{ for } 1 \leq p < n$$

A number ($\neq 1$) fulfilling both properties is called a *principal n-th root of unity*.

If ω is a principal n-th root of unity, then ω^{-1} is as well. (This is all we need to know about complex numbers.)

The integrals of the Fourier transform thus become sums in the *Discrete Fourier transform* (DFT). Let $\mathbf{a} = [a_0, \ldots, a_{n-1}]$. Then $\mathcal{F}(\mathbf{a}) = [u_0, \ldots, u_{n-1}]$ with:

$$u_j = \sum_{k=0}^{n-1} a_k \omega^{jk}$$

If we define a polynomial $p(x) = \sum_{k=0}^{n-1} a_k x^k$, we have $u_j = p(\omega^j)$, the Fourier transformation thus consists of multiple polynomial evaluations.

Similarly we define the inverse Fourier transform:

$$\overline{\mathcal{F}}(\mathbf{a}) = [v_0, \ldots, v_{n-1}] \text{ with } v_j = \frac{1}{n} \sum_{k=0}^{n-1} a_k \omega^{-jk}.$$

It is easily checked that $\mathcal{F}$ and $\overline{\mathcal{F}}$ are mutually inverse.

Furthermore the inverse can be computed by the same algorithm, replacing ω by ω^{-1}.

If $\mathbf{a} \in \mathbb{R}^n$, a naïve evaluation of the DFT takes $O\left(n^2\right)$ operations.

Fast Fourier Transformation

Our aim will be to develop an $O(n \log n)$ algorithm.

A particular efficient way to evaluate the discrete Fourier transform is the *Fast-Fourier transform* (FFT) algorithm of Cooley and Tuckey.

We will assume that $n = 2^l$ and set $m = 2^{l-1}$. Let $\mathbf{u} = \mathcal{F}(\mathbf{a})$. Then for every j we have:

$$\begin{aligned} u_j &= \sum_{k=0}^{n-1} a_k \omega^{jk} = \sum_{k \text{ even}} \cdots + \sum_{k \text{ odd}} \cdots \\ &= \sum_{k=0}^{m-1} a_{2k} \underbrace{\omega^{2jk}}_{=\theta^{jk}} + \omega^j \left(\sum_{k=0}^{m-1} a_{2k+1} \underbrace{\omega^{2jk}}_{=\theta^{jk}} \right) \end{aligned}$$

setting $\theta = \omega^2$.

Then θ is an primitive m-th root of unity and thus $u_j = c_j + \omega^j d_j$ with:

$$c_j = \sum_{k=0}^{m-1} a_{2k} \theta^{jk} \text{ and } d_j = \sum_{k=0}^{m-1} a_{2k+1} \theta^{jk},$$

which are the Fourier Transforms of
$\mathbf{a}_e := [a_0, a_2, \ldots, a_{n-2}] \in \mathbb{R}^m$ and
$\mathbf{a}_o := [a_1, \ldots, a_{n-2}] \in \mathbb{R}^m$ respectively.

As $\omega^m = -1$, $\theta^m = 1$, we have:

$$\begin{aligned} u_{j+m} &= \sum_{k=0}^{m-1} a_{2k}\theta^{(j+m)k} + \omega^{j+m}\left(\sum_{k=0}^{m-1} a_{2k+1}\theta^{(j+m)k}\right) \\ &= \sum_{k=0}^{m-1} a_{2k}\theta^{jk} - \omega^{j}\left(\sum_{k=0}^{m-1} a_{2k+1}\theta^{jk}\right) \\ &= c_j - \omega^j d_j \end{aligned}$$

This suggests a divide-and-conquer approach: To compute $\mathbf{u} = \mathcal{F}(\mathbf{a})$ we compute $\mathbf{c} = \mathcal{F}(\mathbf{a}_e)$ and $\mathbf{d} = \mathcal{F}(\mathbf{a}_o)$ and get the result as

$$\left.\begin{aligned} u_j &= c_j + \omega^j d_j \\ u_{j+m} &= c_j - \omega^j d_j \end{aligned}\right\} j \in \{0, \ldots, m-1\}$$

```
function FFT(n,a,ω);
Input: n a power of 2
Output: u = F(a).
begin
   u := [];
   if n = 1 then
      u[0] := a[0];
   else
      m := n/2;
      c :=FFT(m,a_e,ω^2);
      d :=FFT(m,a_o,ω^2);
      for j ∈ [0..m − 1] do
         u[j] := c[j] + ω^j d[j];
         u[j + m] := c[j] − ω^j d[j];
      od;
   fi;
   return u;
end
```

The runtime fulfills

$$T(n) = 2T(\frac{n}{2}) + An = O\left(n \log n\right).$$

The inverse transform uses the same algorithm with ω replaced by ω^{-1}.

Convolution

Take two polynomials $p(x) = \sum_{k=0}^{n-1} a_k x^k$ and $q(x) = \sum_{k=0}^{n-1} b_k x^k$. Then

$$(p \cdot q)(x) := p(x)q(x) = \sum_{j=0}^{2n-2} \left(\sum_{k=0}^{j} a_k b_{j-k} \right) x^j$$

We call the vector

$$\begin{aligned} \mathbf{a} \circledast \mathbf{b} &= \left[\sum_{k=0}^{0} a_k b_{0-k}, \sum_{k=0}^{1} a_k b_{1-k}, \ldots, \sum_{k=0}^{2n-2} a_k b_{2n-2-k} \right] \\ &= \left[\sum_{k=0}^{j} a_k b_{j-k} \right]_{j=0}^{2n-2} \end{aligned}$$

the *convolution* of $\mathbf{a}$ and $\mathbf{b}$ and have:

$$(p \cdot q)(x) = \sum_{j=0}^{2n-2} (a \circledast b)_j x^j.$$

Now assume we pad the vectors $\mathbf{a}$ and $\mathbf{b}$ with zeroes to length

$2n$ and take $\mathbf{u} = \mathcal{F}(\mathbf{a})$ and $\mathbf{v} = \mathcal{F}(\mathbf{b})$:

$$\begin{aligned} u_l v_l &= \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} a_j b_k \omega^{l(j+k)} = \sum_{k=0}^{2n-1} \sum_{j=0}^{2n-1} a_j b_k \omega^{l(j+k)} \\ &\overset{[k=m-j]}{=} \sum_{m=0}^{2n-1} \sum_{j=0}^{2n-1} a_j b_{m-j} \omega^{lm} = \sum_{m=0}^{2n-1} c_m \omega^{lm} = \mathcal{F}(\mathbf{c})_l \end{aligned}$$

with $\mathbf{c} = \mathbf{a} \circledast \mathbf{b}$. So we get:

Theorem: $\mathbf{a} \circledast \mathbf{b} = \overline{\mathcal{F}}(\mathcal{F}(\mathbf{a}) \cdot \mathcal{F}(\mathbf{b}))$.

The Fourier transform thus can be used to multiply polynomials. We can compute the product of two degree $n/2$ polynomials in time $2O(n \log n) + O(n) + O(n \log n) = O(n \log n)$.

SCHÖNHAGE and STRASSEN applied this to long integer multiplication. To multiply n-digit numbers they need $O(n \log n \log \log n)$ digit operations ($O(n \log n)$ if random memory access is possible).

This is *better* than the $O\left(n^{\log_2(3)}\right)$-algorithm given before, but only worth if n is gigantic.

Optical Fourier Transformation

VI. Reduction. $\mathcal{P}$ and $\mathcal{N}P$

So far we have always reduced a problem to the same type of problem of smaller size. Another type of reduction is to reduce a problem A to another problem B. If we can solve B, in turn we can solve A. In fact we don't need to know *how* to solve B but can take a solving algorithm as a "black box".

This approach can be used to:

- Make use of well-optimized algorithms for B.
- Show that B in general cannot be more simple than A.

It is not guaranteed to give the best possible solution.

Reducing a problem to a well-known other problem (like sorting or FFT) however often will give you a way to find a good practical solution.

We have seen an example already in the reduction of "Bipartite Matching" to 'Network Flow" and in some exercises.

Systems of Distinct Representatives

Take a collection $S_1, S_2, \ldots, S_k$ of sets. A *system of distinct representatives (SDR)* is a set $R = \{r_1, \ldots, r_k\}$ such that $r_i \in S_i$ and $r_i \neq r_j$ $(j \neq i)$.

We interprete this problem as a bipartite matching problem. Take $U = \{S_1, \ldots, S_k\}$ the collection of sets and $W = \bigcup_{S \in U} S$ the union of all their elements. We form a bipartite graph $G = (V = U \cup W, E)$ with an edge $(S, r) \in E$ if $r \in S$.

A SDR then is a matching which matches every set in U. If it exists it must be a maximal matching for G.

We can thus use the algorithm to find matchings in bipartite graphs to solve the problem.

Linear Programming

A problem that can often be used to reduce to is the linear programming problem: This problem tries to maximize (or minimize) a linear function under (linear) inequalities as constraints.

A typical example is the animal-food factory that produces dog food (120 pence profit per packet) and cat food (80 pence profit) with different ingedients:

Ingredient	Available	Dog	Cat
Lamb	1400	4	4
Fish	1800	3	6
Beef	1800	6	2

We want to maximize the profit from producing animal food with these ingredients.

Producing x_1 units dog food and x_2 units cat food we earn:

$$120x_1 + 80x_2 \qquad\qquad \sum_{i=1}^{n} c_i x_i$$

However we need to satisfy:

$$\begin{aligned} 1400 &\leq 4x_1 + 4x_2 \\ 1800 &\leq 3x_1 + 6x_2 \\ 1800 &\leq 6x_1 + 2x_2 \\ b_j &\leq \sum_i a_i x_i \\ e_j &= \sum_i d_i x_i \end{aligned}$$

There exist efficient algorithms for solving this type of problems (Simplex-algorithm). We concentrate on reducing other problems to this:

Network Flow again

Let $x_i = f(i)$ be the flow over edge i. Then we want to maximize the flow leaving the source:

$$\sum_{i=(s,a)} x_i$$

fulfilling $x_i \leq c_i$ while Kirchhoff's law gives further equality constraints.

The Philantropist Problem

Suppose that n organizations are willing to give money to k good causes, organisation i has s_i money to spend and is willing to give at most $a_{i,j}$ to cause j and cause j can take at most t_j. We want to maximize the amount of money spent:

Let $x_{i,j}$ the money given by organization i to cause j. Then we want to maximize $\sum_{i,j} x_{i,j}$ with constraints

$$x_{i,j} \leq a_{i,j}, \qquad \sum_j x_{i,j} \leq s_i, \qquad \sum_i x_{i,j} \leq t_j$$

Integer Linear Programming

Normally the optimal solution for a linear programming problem requires rational numbers. A variation is the search for the best *integral* solution. In this variation the problem becomes suddenly much harder.

Knapsack with values

Suppose that every item has a value v_i. We are not required to fill the knapsack to the brim, but want to get the highest value in the knapsack. If we set $x_i = 1$ if we put item i in and $x_i = 0$ otherwise, then we want to maximise $\sum x_i v_i$ while $\sum x_i s_i \leq k$.

Nondeterministic Turing Machines

We now restrict ourselves to *decision problems* (problems which have an answer "yes" or "no" that can be decided now).

A turing machine is called an *acceptor* for a problem if it will stop and will give the right answer.

A *nondeterministic turing machine* has a (perfect) random number generator, a calculation may involve random numbers but is still guaranteed to stop. A result "yes" also guarantees that the answer is "yes", while an answer "no" guarantees nothing.

This models problems which might be difficult to solve, but for which a solution can be easily checked (typical puzzles):

There is a train connection from Leuchars to Penzance that takes less than 6 hours. Pick random train journeys and check whether they give a suitable journey.

The graph G contains a set of k vertices which induce a complete subgraph (a clique *of size k).* Select k vertices by random and check whether they form a complete subgraph.

but not: *What is the quickest train connection from Leuchars to Penzance?* (Not a "yes/no"-question.)

(However using binary search we can try to get as close to the optimum as we want.)

also not: *Is every even number the sum of two primes?* (We cannot check all numbers in finite time.)

The classes $\mathcal{P}$ and $\mathcal{N}P$

Definition: The class $\mathcal{P}$ consists of all problems that can be solved (on an ordinary turing machine) in polynomial time (in the size of the input).

Definition: The class $\mathcal{N}P$ consists of all problems for which there is a nondeterministic turing machine that will run in polynomial time and if the answer for a given input is "yes" there is a run of the machine that produces this answer.

A problem in $\mathcal{N}P$ is said to be $\mathcal{N}P$*-hard*.

It is easy to show that $\mathcal{P} \subset \mathcal{N}P$.

Most people believe that $\mathcal{P} \neq \mathcal{N}P$, but no proof (or counterexample) is known.

A typical problem in $\mathcal{N}P$ is *Travelling Salesman*: We are given a complete graph with weights on the edges (cities and their distances). We want to know whether there is a round path that goes past every vertex exactly once (a visit to all cities returning to the start again) whose total length is less than a constant C:

Select a random arrangement of the vertices. It only takes polynomial time to check whether there are edges that give a path with this arrangement and whether this path has length less that C.

(We cannot compute the *optimum* solution in polynomial time: For this we would have to consider all paths which is exponential in the number of vertices.)

Definition: A problem A is called $\mathcal{N}P$-complete, if every problem B in $\mathcal{N}P$ can be reduced in polynomial time to A. A is as difficult, as a problem in $\mathcal{N}P$ can be.

If we could show that an $\mathcal{N}P$-complete problem has a polynomial time solution, we would have shown that $\mathcal{P} = \mathcal{N}P$.

The SAT problem

A boolean expression is in *conjunctive normal form* (CNF) if it is an **or**$(+)$ of many **and**$(\cdot)$ expressions), for example:

$$(x + y + \bar{z}) \cdot (\bar{x} + y + z) \cdot (\bar{x} + \bar{y} + z)$$

Every boolean expression can be brought into this form.

The satisfiability (SAT) problem asks whether a given expression in CNF has a solution (there is an assignement to the variables such that the expression becomes true).

Testing a certain assignement of variables is cheap, so SAT is certainly in $\mathcal{N}P$.

Theorem: (Cook, 1971) The SAT problem is $\mathcal{N}P$-complete.

The proof (see [AHU] for details) consists of showing that any turing machine (even nondeterministic) can be described using an expressions in CNF.

The expression will encode all possible runs of the machine for a given input and will be quite complicated.

Yet it still is of polynomial size in the original input and can be constructed in polynomial time.

If we can reduce SAT in polynomial time to a problem $A \in \mathcal{N}P$, then A must be $\mathcal{N}P$-complete as well. This has been shown for many problems:

3-SAT We only permit 3 variables in each **or**-clause: $(x + y + z)$ is permitted, $(x + y + z + a)$ is not. (Note: 2-SAT is solvable in polynomial time!)

Knapsack with values

Travelling Salesman

Clique Determine whether a graph G contains a clique of size $\geq k$.

$\cdots$ Many more.

For some $\mathcal{N}P$-hard problems (notably: Graph isomorphism or primality testing) it is neither known whether they are in $\mathcal{P}$ nor whether they are $\mathcal{N}P$-complete.

What can we do?

In general, no better solution strategy than to test all possible combinations is known.

One can represent these combinations in form of a tree: The first level braches for all choices for the first “part”, the second level for the second “part” and so forth.

To run through all combinations one traverses the tree. This strategy is called *backtracking*.

Such an algorithm is almost guaranteed to be exponential.

To improve the practical performance one can try to stop backtracking not only at the leaves, but as soon as possible if we know that it cannot lead to a better solution than we have already (*branch-and-bound*).

Often one stops if a solution has been found that is good enough for practical purposes.

If $\mathcal{P} = \mathcal{N}P$

it would be possible to solve every $\mathcal{N}P$-hard problem in polynomial time.

No solution (or even idea for a solution) has been found so far.

If this was the case something very fundamental in algorithm design has not yet been discovered.

If $\mathcal{P} \neq \mathcal{N}P$

problems like travelling salesman are inherently hard or even intractable on a standard architecture computer.

Problems of such kind might then be used for purposes like encryption.

For many problems one can not even guarantee an approximated solution in polynomial time: If $\mathcal{P} \neq \mathcal{N}P$ for every polynomial time algorithm A to solve "travelling salesman" and every constant $M > 1$, there is an input such that the solution produced by A is a factor M worse than the optimum.

(If the distances fulfill the *triangle inequalty* – going from a to c over b cannot be shorter than the way from a to c – then we

can at least find a solution which is at most $3/2$ larger than the optimum.)

It has been shown that there would be infinitely many different complexity classes between $\mathcal{P}$ and $\mathcal{N}P$-complete.

However nobody has proven a single problem to be in $\mathcal{N}P$ but *not* in $\mathcal{P}$.

There's still a lot to be done.

Postscript versions of the slides, the exercises and an errata list can be found at: `http://www-gap.dcs.st-and.ac.uk/`
`~ahulpke/cs3001.html`

Errata CS3001 Slides

This list collects serious errors on the slides and handouts. (It does not correct orthographical, grammatical or errors which are obviously spotted and corrected.)

Slide

15 l3. add ($a \geq 1$).

15 Lemma add ($r(n), s(n) \geq 0$).

18 Before Lemma add ($U(i) \geq 0$).

21 Replace "**if** $from \leq m$" by "**if** $L[m] \geq obj$".

26 line -3, -4: replace j by i.

30 Number of comparisons:

$$C(n) = \begin{cases} 2, & n = 2, \\ 2C(n/2) + 2, & n > 2. \end{cases}$$

By induction we prove: $C(n) = \frac{3}{2}n - 1$.

38 The initializations must be $P[0, k].exist$:=false; and $P[i, k].exist$:=false;

46 Replace "**if** $X[j] < X[i]$ **then** swap $X[i]$ with $X[j]$;**fi**" by "**return**;"

52 (Insert algorithm:)The heap contains n elements and we insert a new element $n + 1$.

60 2nd par: parial match of $B(i-1)$, fails at i-th position.

61 (def. of next): $b_{i-j} \cdots b_{i-1}$.

66 in ‘insert’: $C(n,m) = C(n,m-1) - 1$.

70 line -1: Replace “at once” by “simultaneously”

71 The arrow heads are too small.

84 line 3: $\mathbf{a}_o := [a_1, \ldots, a_{n-1}]$.

91 All inequalities on this slide should be $\geq$.