Design and Analysis of Algorithms

Michael Gelfond

Texas Tech University

September, 2017

Introduction

Two ideas are gleaming on the jeweler's velvet.

First is the *calculus*, the second, the *algorithm*.

The calculus made modern science possible; but it has been the algorithm that has made possible the modern world.

David Berlinski, The Advent of the Algorithm.

Introduction

The idea of the algorithm is at least 300 years old, so its understanding requires much work and effort.

The main goal of this class is to deepen our thinking about algorithms.

This is necessary to excel in your profession, but, perhaps more importantly, it can substantially improve your thinking skills.

BASIC TERMINOLOGY

Basic computation - systematic manipulation of input data.

Description of such a manipulation is called *algorithm*.

An algorithm presupposes a fixed collection of elementary data types and operations which can be directly executed by a *computer* (a person or a machine).

Computation is normally used to obtain output data which satisfy some desired properties (often referred to as *specification*).

Problem 1: Stable Marriage Problem

Given: N men and N women. Each man ranks all the women (no equal rank is allowed). Similarly for each women.

Problem: Find an engagement between men and women such that

- Engagement is between one man and one women.
- Everyone must be engaged.
- Engagement is stable, i.e. there are no pairs $\langle m_1, w_1 \rangle$ and $\langle m_2, w_2 \rangle$ such that
    - m_1 prefers w_2 to w_1 and
    - w_2 prefers m_1 to m_2.

Modeling the problem

It is often useful to replace original problem by its mathematical version, often referred to as *model* of the problem.
In our case the model is simple: we have two sets of points and the corresponding preference relations.

Need to find one-to-one correspondence between the sets which satisfy stable condition on preferences.

Recall, that function $F : A \to B$ is one-to-one correspondence if

- **Different elements of A are mapped into different elements of B.**
- **For every element Y of B there is an element X in A such that $Y = F(X)$.**

Stable Marriage Algorithm

We use variable m for men and w for women.

```
function match
while $\exists\, \mathrm{m}\ (\mathrm{free\_man}(\mathrm{m}) \wedge \exists\, w\ \neg \mathrm{proposed}(\mathrm{m}, w))$ do
  1. Select a free man $\mathrm{m}$.
  2. let $\mathrm{m}$ propose to highest-ranked woman $w$ he has
     not yet proposed.
  3. if $w$ is free let her accept, otherwise
  4. if $w$ prefers $\mathrm{m}$ to her current fiance let her break
     the engagement and accept $\mathrm{m}$'s proposal.
return the set of all engaged pairs.
```

Correctness Proof

1. *The algorithm terminates.* At every step a man proposes to a woman he has not proposed yet. Therefore, the maximum number of steps is $N \times N$.

2. *The algorithm defines one-to-one correspondence between* men *and* women.

- *A man is engaged to at most one woman.* A man proposes (and hence can get engaged) only if he is free.
- *Every man is engaged.* Suppose m is not engaged, i.e. he was rejected by every woman. A woman rejects a proposal only if she is engaged, and once engaged she stays engaged. Thus, at the end every woman is engaged to some man. Since a man cannot be engaged to two women (see above), N men are engaged. Contradicts our assumption.

Correctness Proof

- *Different men are engaged to different women.*
  A woman remains engaged from the point at which she receives her first proposal, and, since ties are not allowed, every change increases the quality of her partner. Hence she cannot be engaged to two different men.

This completes the proof of (2).

Correctness proof

3. *The final engagement S is stable.* Suppose it is not, i.e. S contains pairs $\langle m_1, w_1 \rangle$ and $\langle m_2, w_2 \rangle$ such that

- m_1 prefers w_2 to w_1 and
- w_2 prefers m_1 to m_2

To get engaged to w_1, m_1 had to propose to her at some iteration. Since m_1 prefers w_2 he had to propose to w_2 at earlier iteration. There are two possible outcomes:

(a) m_1 and w_2 became engaged. To see that this is impossible let us notice that *every change of engagement for a women increases quality of her partner* so she would never get engaged to m_2. Contradiction.

(b) w_2 was engaged to someone better than m_1. By the same argument she would never get engaged to m_2. Contradiction. m_1 could not have been engaged to w_1!

History

First considered by David Gale and Lloyd Shapley (two mathematical economists), in "College Admission and Stability of Marriage", 1962:

Given a set of preferences among employers and applicants, can we assign applicants to employers so that for every employer E, and every applicant A who is not scheduled to work for E, at least one of the following holds:

1. E prefers everyone of its accepted applicants to A; or
2. A prefers his current situation over working for E.

In 2012 Shapley received a Nobel Prize in Economics: "for the theory of stable allocations and the practice of market design".

Do we get all the stable matchings?

First there could be more than one stable matching.

Example:
$\mathfrak{m}$ **prefers** w **to** w'
$\mathfrak{m}'$ **prefers** w' **to** w
w **prefers** $\mathfrak{m}'$ **to** $\mathfrak{m}$
w' **prefers** $\mathfrak{m}$ **to** $\mathfrak{m}'$

Our algorithm returns $\langle \mathfrak{m}, w \rangle$ **and** $\langle \mathfrak{m}', w' \rangle$

But, $\langle \mathfrak{m}, w' \rangle$ **and** $\langle \mathfrak{m}', w \rangle$ **is also a stable matching!**

It will not be returned by our algorithm no matter which free man is selected first.

Does the outcome depend on the choice of free man?

It does not! First, some definitions:

w is a *valid partner* of m if $\langle \mathrm{m}, w\rangle$ belongs to some stable matching.

w is the *best valid partner* of m ($\mathrm{best}(\mathrm{m})$) if w is a valid partner of m and no woman ranked higher than w is a valid partner of m.

Let $S^* = \{\langle \mathrm{m}, \mathrm{best}(\mathrm{m})\rangle\}$

Theorem: Every execution of our algorithm returns S^*

1. Suppose this is not the case, i,e. there is a run I of the algorithm which returns $S_{\mathrm{I}} \neq S^*$.

2. Then there is a man who, in run I, proposed to his valid partner and was (immediately or eventually) rejected by her.

3. Let m be the *first* such man, and denote the woman he proposed to by w.

4. Since everyone is engaged, and a woman rejects a proposal only if she gets a better partner, S_{I} contains $\langle \mathrm{m}', w \rangle$ where w prefers m' to m.

Theorem: Every execution of our algorithm returns S^*

5. Since w is a valid partner of m, there is stable matching S such that

$$\langle m, w \rangle \in S.$$

In S, m' is engaged to some woman. Let us denote her by w', i.e.

$$\langle m', w' \rangle \in S$$

We have already shown that w prefers m' to m. But who does m' prefer, w or w'?

6. Since m is the *first* man rejected by his valid partner in run I, m' has not been rejected by any valid partner when he got engaged to w, i.e. he prefers w to w'.

7. Therefore, S is unstable. Contradiction!

Review

Let α be an algorithm defined on a set D of strings, and $T(n)$ be the worst-case running time of α on input of size n.

$T(n)$ is order $f(n)$ ($T(n)$ is $O(f(n))$) if there are constants C and x_0 such that for every $x > x_0$, $T(x) \leq C \times f(x)$.

We also say that f is an asymptotic upper bound of T. Asymptotic lower bound, g, of T is defined similarly and denoted by $\Omega(T)$.

Show that $T(n) = n^2 + 5n$ is in $O(n^2)$.

A deterministic algorithm is *polynomial* if it has an asymptotic polynomial upper bound.

Problem 2: Interval Scheduling Problem

There is one resource which can be used by at most one person at a time.

Multiple people request the use of this resource. A request is of the form "Can I use the resource starting at time s, until time f?"

The goal of a scheduler is to maximize the number of requests accepted.

Modeling the Problem

Given: A finite set $R = \{i_1 \ldots, i_n\}$ of names. Elements of R are called ***requests***. Each request i is associated with an interval $[s(i), f(i))$.

A ***schedule*** of R is a subset of R containing no overlapping request.

Find: A ***best (optimal) schedule***. i.e. schedule A of R such that no schedule of R contains more requests than A.

Solution

```
function schedule($R_0$ : set of requests) : set of requests
% returns a best schedule for $R_0$.
var $R$,$A$ : set of requests

$R := R_0$,        $A := \emptyset$.

while $R \neq \emptyset$ do
    1. select a request $i \in R$ with earliest finish time,
    2. add $i$ to $A$,
    3. remove from $R$ all requests which overlap with $i$
       (including $i$).
return $A$.
```

Greedy Algorithms

In our scheduling algorithm we have:

- **A *candidate set*, R from which a solution A is created.**
- **A *selection function* (earliest finish time) which, at each step, chooses the candidate to be added to the potential solution.**

A greedy algorithm follows its selection function, never reverses its coices, and hopes for the best.

It is your job to guarantee that the hope is justified and a truly best solution is found.

Correctness

Lemma 1. The algorithm terminates.

Termination follows since during each step R decreases in size.

In what follows, by $A(m)$ and $R(m)$ we denote the values of A and R after the execution of the m'th step of the loop.

Correctness

Lemma 2. The algorithm returns a schedule.

Show by induction on k that for every step k:

(a) Intervals in $A(k)$ are pairwise disjoint.

(b) No interval in $R(k)$ overlaps with an interval in $A(k)$.

Base. After the first execution $A(1) = \{i_1\}$ and, due to step (3), no interval in $R(1)$ overlaps with i_1 (step 3).

Inductive step: Suppose (a) and (b) hold for $k = m$.

$A(m+1) = A(m) \cup \{i_{m+1}\}$ where $\{i_{m+1}\}$ is selected from R_m. By inductive hypothesis intervals in A_m are pairwise disjoint and no interval in $R(m)$ overlaps with an interval in $A(m)$. Hence, intervals in $A(m+1)$ are pairwise disjoint, and, by step (3) of the algorithm, (b) holds for $k = m+1$.

Correctness

Lemma 3. Let $O = \{j_1, \dots, j_n\}$ be an optimal solution of the problem and A be returned by our algorithm. Then,

- $|A| \leq |O|$;
- **For every step k of the algorithm**

$$(*) \quad f(i_k) \leq f(j_k).$$

The first condition follows immediately since O is optimal and A is a solution (Lemma 2).
Let us prove the second condition by induction on k.
Base: $k = 1$ – is true by construction.
Inductive Hypothesis: $(*)$ holds for every $k \leq m$.
Show that $(*)$ holds for $k = m + 1$.

Correctness

Since O is a schedule we have that

$$(1) \quad f(j_m) \leq s(j_{m+1}).$$

By inductive hypothesis,

$$(2) \quad f(i_m) \leq f(j_m).$$

(1) and (2) imply

$$(3) \quad f(i_m) \leq s(j_{m+1})$$

This means that at the moment of selection of i_{m+1} interval j_{m+1} does not overlap with intervals in A, and therefore belongs to R. Since the algorithm selects interval from R with the earliest finish time, we have

$$(4) \quad f(i_{m+1}) \leq f(j_{m+1})$$

and $(*)$ follows by induction.

Correctness

Lemma 4. A **is optimal.**

Suppose it is not and there is optimal $O = \{j_1, \dots, j_n\}$, **i.e.** O **is a schedule and** $|O| > |A|$.
Let i_m **be the interval added to** A **during the last iteration. By Lemma 3,**

$$(1) \quad f(i_m) \leq f(j_m).$$

Since $|O| > |A|$, **there is** $j_{m+1} \in O$. **Since** O **is a schedule,**

$$(2) \quad f(j_m) \leq s(j_{m+1}).$$

and, by (1) and (2)

$$(3) \quad f(i_m) \leq s(j_{m+1}).$$

i.e. after the last iteration of the algorithm $j_{m+1} \in R$.
But, at this time, R **must be empty. Contradiction.**

Lemmas 2, 4 imply algorithm's correcteness.

Time Complexity

Our scheduling algorithm traverses R twice (in steps (1) and (3)) and hence its time complexity is n^2.

Can we refine the algorithm to make its time complexity be $O(n * \log n)$?

Step one can be reduced to sorting R and traversing it from left to right which is $O(n * \log n)$.

Instead of removing from R all intervals overlapping with the just scheduled interval i we can simply traverse R in search of intervals to be added to A.

This can be done as follows:

Time Complexity

Let $R[1..n]$, $A[1..n]$ **be arrays of requests where** $R[k]$ **is a record containing** k**'th starting and finishing times.**

```
1. Sort $R$ in order of finishing time. [$O(n * \log n)$ time]
2. $k, i := 0$
3. $f := -1$ % finishing time of the last scheduled request
4. while $k < n$ do [Linear Time]
5.       $k := k + 1$
6.       if $R[k].s \geq f$ then
              $i := i + 1$
              $A[i] := R[k]$
              $f := R[k].f$
7. return $A[1..i]$.
```

Problem 3: Interval Partitioning

***GIVEN*: A large set of identical resources and a set $R = \{i_1 \ldots, i_n\}$ of requests.**

Each request i is associated with an interval $[s(i), f(i))$.

A request i asks for the use of one resource during the interval $[s(i), f(i))$.

***DO*: Schedule *all* requests using as *few* resources as possible.**

***Examples*: Schedule classes using as few classrooms as possible, allocate printing jobs using as few printers as possible, etc.**

Modeling the Problem:

The set R of requests (also called *jobs*) is defined as in Problem 2. Resources are numbers from 1 to n.

To define a schedule we need the following notation: Let

$$\alpha : R \rightarrow [1..n]$$

be an assgnment of jobs to resources and

$$R_k = \{j \in R : \alpha(j) = k\}$$

be the set of all jobs served by a resource k.

Modeling the Problem:

Mapping $\alpha : R \to [1..n]$ is called a *schedule* if

- For some $0 < d \leq n$, $R_1, \dots, R_d$ is a partition of R, i.e.
    - $R = R_1 \cup \dots \cup R_d$ and
    - if $m \neq k$ then $R_m \cap R_k = \emptyset$.
- For every $0 < m \leq d$, intervals requested by jobs from R_m do not overlap.

Schedule which uses as few resources as possible is called *optimal*.

Discussion

$\text{depth}(R)$ - max number of overlapping requests in R.

Can we partition R into $R_1, \ldots, R_d$ where $d = \text{depth}(R)$ and requests in any R_i do not overlap?

If the answer is positive then this allocation of resources and the subsequent scheduling of jobs is optimal – clearly, the number of resources needed is at least d.

IDEA:

1. Schedule jobs according to their starting times.

2. Divide the set of available resources into those which are in use (*occupied*), had been used but are now available (*released*), and not being used at all. If possible, select next resource from *released*.

Pseudocode

```
function IntervalPartitioning($R$ : set of jobs)
% Returns an optimal schedule $(R_1, \dots, R_m)$ for $R$.
var $released, occupied : resources;$   $m : resource$
$m := 0;$      $released, occupied := \emptyset;$
1. Sort $R$ w.r.t. starting times of its jobs.
2. for every $I \in R$ do
    3. Move all resources in $occupied$ which finished
       before the start of $I$ into $released$.
    4. if $released \neq \emptyset$ then
           $m := select(released);$
           Move $m$ from $released$ to $occupied$.
    5. else $m := m + 1;$ % select new resource.
              create $R_m$, set it to $\emptyset$, add $m$ to $occupied$.
    6. Add $I$ to $R_m$.
Let jobs in $R_i$ be served by resource $i$.
```

Proof of Correctness

- **Algorithm obviously terminates after all jobs are scheduled.**
- **Algorithm returns a schedule.**

1. Since every $I \in R$ is moved into some R_i (step 6)

$R = R_1 \cup \dots \cup R_m$.

No job is served by two resources. $R_1 \cup \dots \cup R_m$ is a partition.

2. No two jobs in R_i overlap. [Induction on number of iterations]
Note that when resource m is allocated to serve I every job served by resources in *released* finishes before I starts.

Proof of Correctness

To show that the schedule $R_1, \dots R_m$ is optimal, it suffices to show that $m \leq d$.

Suppose (3) is not the case.

Then, before the allocation, every resource from 1 to d is in *occupied*, i.e. every such resource is in use. But this means that there are at least $d + 1$ overlapping requests in R which contradicts the definition of d.

Since any solution needs at least d resources, $m = d$, and $R_1, \dots, R_m$ created by the algorithm is an optimal solution.

Scheduling to minimize maximum lateness

GIVEN: **Set R of requests, where each $i \in R$ has duration, t_i, and deadline, d_i. A single, but unlimited, resource, e.g. a processor.**

Schedule **maps a request $i \in R$ into interval $[s(i), f(i))$ of length t_i such that intervals for different requests are disjoint.**

A request i is *late* if $f(i) > d_i$. A request's *lateness*, L_i is $f(i) - d_i$ if i is late and 0 otherwise.

DO: **Find a schedule which**

- **starts at a given point s;**
- **minimizes maximum lateness, $L = max_i(L_i)$.**

Basic Idea: schedule request with earliest deadline first!

Scheduling to minimize maximum lateness

```
min-max-lateness($R$ : set of requests, $s$ : time) : schedule
var f : time % finishing time of the last scheduled job.

Sort requests in order of their deadlines: $d_1 \leq \cdots \leq d_n$.
$f := s$;
For every job $i$ from $1$ to $n$ do
    $s(i) := f$
    $f(i) := s(i) + t_i$
    $f := f + t_i$
return $[s(i), f(i))$ for every $i$.
```

Correctness (exchange argument)

idle time - time between jobs.

inversion - assignment of times to two jobs in which the job with later deadline is scheduled first, i.e.

- $s(i) < s(j)$,
- $d_j < d_i$

Obviously, the schedule produced by our algorithm has no idle time and no inversion. Optimality follows from the following Lemmas:

Lemma 1. All schedules with no idle time and no inversion have the same maximum lateness.

Lemma 2. There is an optimal schedule with no idle time and no inversion.

Proof of Lemma 1

Let us compute the maximum lateness.

First notice, that two different schedules with neither inversion nor idle time only differ in order in which jobs with the same deadline are scheduled.

Clearly, all rearrangement of jobs with the same deadline start and finish at the same time, say s_d and f_d. Hence the maximum lateness for every rearrangment is the same $- f_d - d$.

Proof of Lemma 2

Let O be an optimal schedule. Clearly it has no idle time.

Suppose it has an inversion. Then there are two consecutively scheduled jobs, i and j such that

$$(1)\ d_j < d_i.$$

Let us swap i and j and show that the maximum lateness, $\bar{L}$ of the new schedule, $\bar{O}$ does not exceed the maximum lateness, L of O, i.e. $\bar{O}$ is still optimal.

The optimal schedule without inversion and idle time can be, therefore, obtained by repeating swapping until all inverse pairs are eliminated.

Proof of Lemma 2

To show that $\bar{L} \leq L$ consider a job r in O scheduled for interval $[s(r), f(r))$ with lateness L_r.
The corresponding quantities in $\bar{O}$ will be denoted by $[\bar{s}(r), \bar{f}(r))$, and $\bar{L}_r$.
The only possible increase in lateness $\bar{L}$ in $\bar{O}$ could have occur because of increase in lateness of i. Let us show that this is impossible. First, by definition, $\bar{f}(i) = f(j)$ and hence

$$(2)\ \bar{L}_i = \bar{f}(i) - d_i = f(j) - d_i.$$

From (1) and (2) we have

$$(3)\ \bar{L}_i = f(j) - d_i \ < \ f(j) - d_j = L_j.$$

Therefore,

$$\bar{L}_i < L_j \leq L$$

and hence the swap does not increase L.

Spanning Trees

Given: Graph $G = \langle V, E \rangle$ with positive cost c_e associated with every edge e.

Definitions:

- An undirected graph is a *tree* if it is connected and has no cycles (recall that a cycle must include at least two different links).
- *Spanning tree* of G is a set T of edges such that $\langle V, T \rangle$ is a tree.
- *Cost* of T is the sum of costs of its edges.
- *Minimum spanning tree* of G is a spanning tree of G with minimal cost.

Problem: Find a minimum spanning tree of G.

Number of spanning trees in a graph is $O(n^n)$.
Exhaustive search is impossible.

Kruskal's Algorithm

```
kruscal($G = \langle V, E \rangle$ : weighted graph) : set of edges
% Assumption: all costs are different
% returns a minimal spanning tree of $G$.

var T : set of edges

$T := \emptyset$
Sort $E$ in increasing order of costs.
for every $e \in E$ do
    if $T \cup \{e\}$ has no circles then
        $T := T \cup \{e\}$
return(T).
```

Kruskal's Algorithm: Cut Property

Cut Property. If S is a non-empty set of nodes different from V and $e = \langle v, w \rangle$ is a minimum-cost edge with one end in S and another in $V \setminus S$ then every minimum spanning tree contains e.

Proof. Let e be as above. Suppose there is a minimum spanning tree T not containing e.

Since $\langle V, T \rangle$ is a tree, v and w are connected by a path $P = v, \dots, v_1, w_1, \dots w$ in G where $e_1 = \langle v_1, w_1 \rangle$ is the first edge in P with one end in S and another in $V \setminus S$.

Let T_1 be obtained from T by replacing e_1 by e.

We will show that T_1 is a spanning tree of G. Since cost of T_1 is smaller than that of T this will contradict the minimality of T.

Kruskal's Algorithm: Cut Property

To show that T_1 is a spanning tree we need to show that

(a) Graph $\langle V, T_1 \rangle$ is connected.

Take two nodes x_1 and x_2 of V. Since T is a spanning tree, $\langle V, T \rangle$ is connected, i.e. contains a path Q connecting these nodes. If Q does not contain e_1 it is also a path of $\langle V, T_1 \rangle$ (which connects x_1 and x_2). If Q contains $e_1 = \langle v_1, w_1 \rangle$, i.e. $Q = \langle x_1, \dots, v_1, w_1, \dots x_2 \rangle$ then path $R = \langle x_1, \dots, v_1, \dots, v, w, \dots w_1, \dots x_2 \rangle$ in T_1 connects x_1 and x_2.

(b) $\langle V, T_1 \rangle$ has no cycles.

The only cycle in $\langle V, T_1 \cup \{e_1\} \rangle$ is the one composed of e and P. But this cycle can not be in $\langle V, T_1 \rangle$ since e_1 has been removed.

Kruskal's Algorithm: Correctness

By the Cut Property every edge added to T in the Kruskal's algorithm belongs to every minimum spanning tree of G.
So, if T is a spanning tree then it is also minimum spanning tree.
Show that it is a spanning tree, i.e. $\langle V, T \rangle$ is a tree.
Clearly, T has no cycles. Suppose it is not connected i.e. there is set S different from $\emptyset$ and V such that nodes from S and $V \setminus S$ are not connected by edges from T.
But there is an edge from S to $V \setminus S$ in G. Cheapest such edge would have been added to T by the algorithm.
Contradiction.

Implementing Kruscal's Algorithm

Naive implementation of the algorithm may require $|E| \times |V|$ steps. Can we do better?

Challenge: **How to check that $T \cup \{e\}$ has no cycles in time better than $|V|$?**

Idea: Represent T as a collection $T_{i_1}, \dots, T_{i_k}$ of disjoint sets. Design function $\mathit{find}(u)$ which, for every $u \in S$ returns i_m such that $u \in T_{i_m}$.

Let $\{e = \langle v, w \rangle\}$, $m = \mathit{find}(v)$ and $n = \mathit{find}(w)$, and $|T_{i_m}| \geq |T_{i_n}|$.
$T \cup \{e\}$ has no cycles iff $m \neq n$. In this case replace T_{i_m}, T_{i_n} by $T_{i_m} = \mathit{unite}(T_{i_m}, T_{i_n})$.

Good data structure allow $\log(|V|)$ find and constant unite.

History

One of the early applications of graph theory to computing. Even though the graphs in some form appeared in math starting with Euler, the first textbook on graph theory was published in 1935.

Similar algorithm was first published by a Czech mathematician Boruvka (1926) in a paper "Contribution to the solution of a problem of economical construction of electrical networks (Czech)".

In 1954 graph theory was sufficiently known so Kruscal could publish his work in a mathematical journal.

Now the algorithm forms the basis for solutions of many optimization problems.

Shortest Path in the Graph

GIVEN: directed graph $G = \langle V, E \rangle$ and the length $l_e \geq 0$ for each edge e; starting node s.

FIND shortest path from s to each other node.

Length, $l(P)$ of path P is the sum of the lengths of its edges.

fringe(S) - set of all $X \in V \setminus S$ connected to some node of S by an edge.

Simplification: (a) determine the *length* of the shortest path from s to each other node. (b) Assume that there is a path from s to any other node in G.

Discussion

Our goal is to "serve", i.e., compute minimal distance from s, for each vertice of G. To do that

1. Maintain $S \subseteq V$ such that for each $u \in S$ the shortest distance, $d(u)$ from s to u had been already computed, and is less or equal to $d(v)$ for every $v \notin S$.

2. Consider set $E = \{\langle u, v\rangle : u \in S, v \notin S\}$ and function α defined on E such that

$$\alpha(\langle u, v\rangle) = d(u) + l(\langle u, v\rangle),$$

select $\langle u_0, v_0\rangle$ which minimizes α, add v_0 to S, and set $d(v_0)$ to $\alpha(\langle u_0, v_0\rangle)$.

Clearly, new S still satisfies properties from (1).

Pseudocode

```
Dijkstra(G : directed graph, l - length, s : node)
% returns function $d(u)$ whose value is the shortest
% distance between $s$ and $u$.

var $S$ : set of explored nodes; function $d(u)$

$S := \{s\}$        $d(s) := 0$
while $S \neq V$ do
   1. Select $v_0 \in Fringe(S)$ for which
         $d'(v) = \min_{\{u \,:\, u \in S, \langle u,v \rangle \in G\}} d(u) + l(\langle u, v \rangle)$
      is as small as possible.
   2. $d(v_0) := d'(v_0)$
   3. $S := S \cup \{v_0\}$
Return d
```

Second Refinement

How to represent G**,** S**, and** $\mathsf{Fringe}(S)$**?**

G **- adjacency list** $O(max\_degree(G))$**.**

S **- a tree of nodes**

An array, $nodes$**, where** $nodes(v)$ **contains a pointer to a position of** v **in** S **(or** nil **if** v **is not there yet).**

$F = \mathsf{Fringe}(F)$ **– a priority queue with entries** $\langle v, key(v) \rangle$ **where at each point of the execution of the algorithm** $key(v)$ **is the length of a shortest path from** s **to** v **found so far.**

Second Refinement

$S := \{s\} \qquad d(s) := 0$

$F := \{\langle y, key(y)\rangle\}$

where y **is adjacent to** s **and** $key(y)$ **is the length of shortest edge from** s **to** y**.**

```
while $F \neq \emptyset$ do
    1. $m := extract\_minimum(F)$
    2. $S := S \cup \{m\}$
    3. $d(m) := key(m)$
    4. For every edge $\langle m, v\rangle$ with $v \in V \setminus S$ do
        4a. $Temp := d(m) + l_{\langle m,v\rangle}$
        4b. if $v \notin F$ then $insert(F, v, Temp)$
        4c. else if $Temp < key(v)$ then
                 $change\_key(F, v, Temp)$
return d
```

Divide and Conquer

A type of algorithm which breaks the input into parts, solves each part recursively, and then combines the solutions of these subproblems into an overal solution.

Example: computing the value of an arithmetic expression represented by a tree, merge sort, etc.

Counting Inversions

Problem: Given a person's preferences (for books, movies, etc.) match them with preferences of other people on the Web with similar interests to provide a suggestion.

Preferences are often defined by rankings, i.e., labeling the objects from 1 to n. So the problem is to *define and compute the distance between the rankings*.

Mathematical formulation

Given: a sequence $S = \langle a_1, \ldots, a_n \rangle$ of distinct numbers.

Def: a pair $i < j$ of indices form an *inversion* if $a_i > a_j$.

Find: The number of inversions in S.

Sequence $\langle 1, 2, 3 \rangle$ has no inversions, sequence $\langle 2, 1, 3 \rangle$ has one, formed by indices 1 and 2, sequence $\langle 3, 2, 1 \rangle$ has three.

Counting Inversions

***Basic Idea*: Sort sequences together with counting inversions to facilitate merging.**

```
CountInversions
Given: Sequence $S = \langle a_1, \ldots, a_n \rangle$
Return: Number of inversions in $S$ and sorted $S$.
if $n = 1$ then return(0,S)
Divide $S$ into two halves, $A$ and $B$.
    $(r_A, A) := \mathrm{CountInversion}(A)$
    $(r_B, B) := \mathrm{CountInversion}(B)$
    $(r, S) := \mathrm{merge\_and\_count}(A, B)$
return($r_A + r_B + r, S$).
```

Counting Inversions

Merge_and_count
Given: Sorted sequences A,B.
Return: The number of pairs X, Y such that $X \in A$, $Y \in B$ and $X > Y$, and sorted sequence C consisting of elements of A and B.

```
var i,j : ponter, C : sequence, Count : integer
% $i$ and $j$ point to element $a_i$ of $A$ and element $b_j$ of $B$
$i, j := 1$     $C := \emptyset$     $Count := 0$
*while* both lists are non-empty *do*
    Move the smaller of two elements $a_i$, $b_j$ to $C$.
    if $b_j < a_i$ then $Count := Count + |A|$.
    Advance pointer to the list from which the smaller
    element was selected.
Append the remainder of the non-empty list to $C$.
*return* (Count,C).
```

The Algorithm Complexity

Clearly, $\mathsf{Merge\_and\_count}$ **requires** $O(n)$ **where** $n = |A| + |B|$.

Let $T(n)$ **denote the worst-case running time on input of size** n. **Let** $n = 2^k$. **To get the input down to 2 we need** $k = \log_2 n$ **levels of recursion.**

First level has one merge which requires $c \times n$ **steps.**

On the i'th level the number of subproblems has doubled i **times, so their number is** 2^i. **Each has input size** $n/2^i$ **and takes at most** $c \times n \ / \ 2^i$ **steps. Overall time spent on level** i **is** $2^i \times (c \times n \ / \ 2^i) = c \times n$.

Executing k **levels requires** $c \times n \times \log_2 n$, **i.e. the algorithm time is** $O(n \times \log_2 n)$.

Dynamic Programming

1. Divide the problem into subproblems such that
   - The number of subproblems is polynomial.
   - The solution to the original problem can be easily computed from solutions to the subproblems.
   - Subproblems can be ordered from “smallest” to “largest” and solutions of larger are connected to that of smaller by a recurrence relation.
2. Get rid of recursion by storying solutions of problems from smaller to larger in an array.

Weighted Interval Scheduling

Given: n requests, each request i associated with an interval $[s_i, f_i)$ and has the weight, v_i. One server.

Find: A subset of mutually compatible requests with maximum summary weight.

This time ordering intervals by the earliest finishing time does not work. Later interval not compatible with the selected one may have much higher weight.

First solve a simpler problem – find maximum summary weight.

Notation: $f_1 \leq f_2 \cdots \leq f_n$; $p(j)$ is the last interval in the sequence which finishes before j starts, 0 if no such interval exists.

Recursive Algorithm

Divide the problem into finding maximum weight of a schedule containing j and that not containing j.

```
opt(j : requests) : weight
% Requests from 1 to j are sorted by finishing time.
% Returns weight of the optimal solution of the
% scheduling problem.
% Recurrence relation:
```

$$\mathrm{opt}(j) = \max(v_j + \mathrm{opt}(p(j)), \mathrm{opt}(j-1))$$

```
if $j = 0$ return(0).
return($\max(v_j + \mathrm{opt}(p(j)), \mathrm{opt}(j-1))$).
% Returns max of the best solution containing j
% and that not containing j.
```

Unfortunately has exponential number of calls. Can we reduce this number?

Memoization [Getting rid of recursion]

The first time $\mathrm{opt}(k)$ **is computed store the value in an array, say** $M[0\ldots n]$**.**

```
$\mathrm{opt}(n : \mathrm{requests}) : \mathrm{array}$
% Requests $1\ldots n$ sorted by finish time
% Returns $M[0..n]$ where $M[j]$ is the weight of
% optimal scheduling of first $j$ requests.

var $M[0..n]$.

$M[0] := 0$
for j from 1 to n do
    $M[j] := \max(v_j + M[p(j)], M[j-1])$
return($M$)
```

Finding Optimal Solution

```
$Find\_Solution(M$ : array, $j$ : index) : sequence of requests.
% For $j > 0$, $M[j]$ contains the weight of optimal solution
% for scheduling first $j$ requests.
% Returns optimal solution for scheduling first $j$
% requests.

if $j = 0$ return([ ])
if $v_j + M[p(j)] > M[j-1])$ then
    return($Find\_Solution(M, p(j)) \circ j$)
return($Find\_Solution(M, j-1)$)
```

The function is in $O(n)$.

Approximating Set of Points by a Segment

Given: A set $S = \{(x_1, y_1) \dots (x_n, y_n)\}$ of points on the coordinate plane
Find a segment of ***best fit***.

How to define ***best fit***, i.e., how to measure an error – distance between S and function $f(x, a, b) = ax + b$ is an interesting theoretical question.

A popular ***least squares*** method finds a and b which minimize

$$R^2 = \sum_{i=1}^{n} (y_i - f(x_i, a, b))^2$$

There are good algorithms for such a minimization, and we assume that they are given.

Segmented Least Square Problem

What to do if S can only be reasonably approximated by more than one segment?

1. Partition S into a number of segments and approximate each segment separately.

2. Balance the number of segments in the approximation and the quality of the fit.

In other words, *fit points well using as few segments as possible.*

Formulating Segmented Least Square Problem

We order partitions by assigning *penalties* – the sum of the number of segments in the partition (times some constant C) AND the error value of the optimal line through each segment.

The first value keeps the number of segments small. The second ensures that each component of the partition is accurately approximated by a segment.

Segmented Least Square Problem: *Given* C*, find a partition of minimum penalty.*

Recurrence Relation

Sort points $p_1, \ldots, p_n$ **with respect to X-coordinate.**

$opt(n)$ **penalty of optimal approximation of** $p_1, \ldots, p_n$ $(0$ **if** $i = 0)$.

e_{ij} **- minimum error of any line approximating** $p_i, \ldots, p_j$.

If $p_i, \ldots, p_n$ **is the best last segment then**

$$opt(n) = C + e_{in} + opt(i)$$

The recurrence relation

$$opt(j) = \ min_{1 \leq i < j}(C + e_{ij} + opt(i))$$

Memoization

```
$opt(p_1, \dots, p_n : points) : array$
% Points are sorted by X coordinate.
% Returns $M[0..n]$ where $M[j]$ is the minimum penalty
% for approximation of $p_1, \dots, p_j$.

var $M[0..n]$, $e[0..n, 0..n]$

$M[0] := 0$
for every $i \leq j \leq n$ do
    compute minimal penalty $e_{ij}$ for approximation of
    $p_i, \dots, p_j$ by one line.
for $j = 1$ to $n$ do

        $M[j] := \min_{1 \leq i < j}(C + e_{ij} + M[i])$

return($M$)
```

Subset Sum

Given: Single machine to process n job requests. Machine is available in time interval from 0 to W. A request i requires time w_i to process.

Goal: find a schedule which keeps the machine as busy as possible.

Equivalent formulation: Given a set of items with non-negative weights w_i and integer W find a subset S of items such that

- $$\sum_{i \in S} w_i \leq W$$
- subject to restriction above the sum is as large as possible.

Subset Sum

$\mathrm{opt}(i, w)$ **- solution of the problem with items** $1 \dots i$ **and allowed weight** w**.**

Recurrence relation:

$$\mathrm{opt}(i, w) = \max(\mathrm{opt}(i-1, w), w_i + \mathrm{opt}(i-1, w-w_i))$$

Subset Sum

```
function MaxWeight
input: items $1 \dots n$, weights $w_1, \dots w_n$, max weight $W$
output: $M[i,w] = \max(\sum_{k \in S} w_k \le w$ where $S \subseteq \{1,\dots,i\})$

var : array $M[0\dots n, 0\dots W]$

$M := 0$
For every $i$ do
    For every $w$ do
      if $w < w_i$ then (ith item does not fit)

                    $M[i,w] := M[i-1,w]$

      else

          $M[i,w] := \max(M[i-1,w], w_i + M[i-1, w-w_i])$

return($M$)
```

Subset Sum

To complete the solution of the original problem compute an array M, $M := MaxWeight(P)$ **and call function** $\mathrm{print}(n, W)$ **defined as follows:**

```
function $\mathrm{print}(i, w)$
if $i > 0$ then
    if $M[i, w] \neq M[i-1, w]$ then
        $\mathrm{display}(i)$
        $\mathrm{print}(i-1, w-w_i)$
    else $\mathrm{print}(i-1, w)$
```

Knapsack

```
knapsack(n : item, W : capacity) : set of items
% items 1..n, each item i has weight $w_i$ and value $v_i$
% Returns subset of items which maximizes the sum of
% their values subject to the capacity restriction.
```

```
function MaxVal(n : items, W : knapsack capacity) : int
Recursive algorithm to compute maximim value of
items which can be packed in a knapsack of capacity $W$.

if $n = 0$ return $0$
if $W < w_n$ then $return(MaxVal(n-1, W))$
$return(max(MaxVal(n-1, W), v_n + MaxVal(n-1, W-w_n)))$
```

Knapsack (memoization)

```
MaxValues(n : items, W : knapsack capacity) : array
%Create array $M$ such that $M[i, w] = MaxVal(i, w)$.

Set the entries of $M$ to $0$.

For every $i = 1$ to $n$ do
    For every $w = 1$ to $W$ do
        if $w < w_i$ then (ith item does not fit)
            $M[i, w] = M[i - 1, w]$
        else
            $M[i, w] = max(M[i - 1, w], v_i + M[i - 1, w - w_i])$
return($M$)
```

Knapsack (Original Problem)

To complete the solution of the original problem compute an array M, $M := MaxValues(n, W)$ **and call function** $print(n, W)$ **defined as follows:**

```
function print(i, w)
if i > 0 then
    if $w < w_i$ then print(i − 1, w)
    else
        if $M[i-1, w] \geq v_i + M[i-1, w-w_i]$ then
            print(i − 1, w)
        else display(i)
            $print(i-1, w-w_i)$
```

Number of Combinations

$C(n, k)$ - number of subsets (combinations) of k-elements from an n-element set.
Standard formulas:

$$C(n, k) = n!/(k!(n - k)!)$$

$$C(n, k) = n \times n - 1 \times \cdots \times (n - k + 1)/k!$$

and

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k) \text{ for } 0 < k < n$$

$$C(n, 0) = C(n, n) = 1$$

Number of Combinations

```
**function** comb($n, k$)
**input**: $n \geq k \geq 0$
**output**: $C(n, k)$

var matrix $M[n, k]$ of integers

for $i$ from $0$ to $n$ do
    for $j$ from $0$ to $\min(i, k)$ do
        if $j = 0$ or $j = i$ then $M[i, j] := 1$
        else $M[i, j] := M[i-1, j-1] + M[i-1, j]$
return($M$)
```

NP problems

A problem belongs to class NP if it can be solved in polynomial time by a non-deterministic Turing machine.

To solve a problem such a machine non-deterministically selects the problem's candidate solution and checks, if it is indeed a solution, in polynomial time.

Every problem belonging to a class P of polynomially solvable problems also belongs to class NP. It is not known if the reverse is true.

Figuring out if $P = NP$ is one of the most important problems of CS.

Polynomial Reductions and NP-complete problems

Decision problem: Given a set L of string over alphabet Σ (often called a language) check if a string $x \in \Sigma^*$ belongs to L.
Let $L_1, L_2 \subseteq \Sigma^*$.

A polynomial function $\tau : \Sigma^* \to \Sigma^*$ is a *polynomial reduction* of L_1 to L_2 if for every $x \in \Sigma^*$, $x \in L_1$ iff $\tau(x) \in L_2$.

A decision problem for L is NP-complete if

- L belongs to NP.
- Every NP problem is polynomially reducible to L.

It is easy to see that $P = NP$ iff there is a polynomial solution of at least one NP-complete problem.

Boolean Satisfiability (SAT)

A *literal* is a boolean variable or its negation.

Clause is a set of literals. A clause C is *satisfied* by an assignment of truth values to variables if this assignment makes at least one elements of C true.

Formula is a set of clauses. Formula F is satisfied by a truth assignment I if all of its clauses are satisfied by I. Assignment I satisfying F is called a *model* of F.

Problem: Check if a given formula is satisfiable, i.e. has a model. (Here our language L consists of all satisfiable formulas).

SAT

It is easy to prove that SAT is an NP problem.

The corresponding non-deterministic Turing Machine will non-deterministically select an assignment $I = \{X_1 = v_1, \ldots, X_n = v_n\}$ of the truth values to the variables of F and checks if I is a model of F.

The latter can be done in polynomial time as follows:

```
repeat
    Select $X_i = v_i$.
    Remove all clauses containing a literal $l$ formed by $X_i$
    which is true under this assignment.
    Remove from the clauses of $F$ all remaining
    occurrences of such literals.
until $F = \emptyset \lor \{\ \} \in F$.
if $F = \emptyset$ return $true$ else return $false$.
```

SAT

In the early 70s of the last century Cook and Levin independently established that SAT is an NP- complete problem.

Despite the apparent absence of a polynomial algorithm for solving SAT its instances are frequently solved in many practical applications (including planning, diagnostics, decision support systems, etc.) by programs called SAT-solvers.

In the last 20 years SAT-solvers moved from solving problems with 100 variables and 200 clauses to $1,000,000^+$ variables and $5,000,000^+$ clauses.

SAT-solvers – Basic Algorithm

Partial interpretation of a formula F is a mapping I of a subset of its variables to the truth values.

function *SAT*
input: Formula F_0 and partial interpretation I_0.
output: $\langle I, true\rangle$ where I is a model of F_0 which extends I_0.
$\langle I_0, false\rangle$ if no such model exists.

Uses function $Cons(F, I)$ which expands I by the assignment of variables which must be true to satisfy F and simplifies F accordingly. If there are no expansions of I returns $false$.

Basic Algorithm (continued)

```
var $F$ : formula; $I$ : partial interpretation; $X$ : boolean.

$F := F_0$;          $I := I_0$;
if $Cons(F, I) = false$ then
    return $\langle I_0, false\rangle$.
$\langle F, I\rangle := Cons(F, I)$.
if $F = \emptyset$ return $\langle I, true\rangle$.
select a boolean variable $p$ from $F$ undefined in $I$.
$\langle I, X\rangle := SAT(F \cup \{\{p\}\}, I)$.
if $X$=true return $\langle I, X\rangle$.
return $SAT(F \cup \{\{\neg p\}\}, I)$.
```

Computing Consequences

function $Cons(F, I)$
Returns $\langle F', I' \rangle$ such that a model of F contains I iff it is a model of F' which contains I'.
Return *false* if no such F', I' exist.

```
while F contains a clause of the form $\{l\}$ do
    Remove from F all clauses containing $l$.
    Remove from F all occurrences of $\bar{l}$ where $\overline{p} = \neg p$, $\overline{\neg p} = p$.
    $I := I \cup \{l\}$
if $\emptyset \in F$ return false
return $\langle F, I \rangle$.
```

Tracing SAT

Let $I = \emptyset$ and $F = \{\{X_1\}, \{\neg X_1, X_2, X_3\}, \{\neg X_1, X_4\}\}$.

$Cons(F, I)$ returns $F = \{\{X_2, X_3\}\}$ and $I = \{X_1, X_4\}$

Suppose the algorithm non-deterministically selects X_2 and calls $SAT(\{\{X_2, X_3\}, \{X_2\}\}, \{X_1, X_4\})$.

The new call to SAT returns $I = \{X_1, X_4, X_2\}$ and $F = \emptyset$.

The second termination condition is satisfied and SAT returns $\langle\{X_1, X_4, X_2\}, true\rangle$.

Comments on Actual Implementations

Developers of SAT-solvers found many ways to improve solvers' efficiency, including

- Smart heuristics which allow selection of a good p.
- Learning clauses containing information about previous failures and adding it to the formula. This leads to a smart backtracking.
- Occasional random restarting of search to avoid being stuck in a wrong path.
- Smart data structurs.
- Automatic tuning of heuristics for particular instances.

3-SAT is NP-complete.

SAT restricted to formulas whose clauses contain at most three literal is called 3-*SAT*.

To show that it is NP-complete we'll define a polynomial reduction τ from *SAT* to 3-*SAT*.

For every clause $C = \{\lambda_1, \dots, \lambda_n\}$, $\tau(C)$ consists of clauses

$\{\lambda_1, \lambda_2, Y_1\}$,
$\{\neg Y_1, \lambda_3, Y_2\}$,
$\dots$
$\{\neg Y_{i-2}, \lambda_i, Y_{i-1}\}$,
$\dots$
$\{\neg Y_{n-3}, \lambda_{n-1}, \lambda_n\}\}$

where Ys are new variables.

3-SAT is NP-complete (continued)

Intuitively, Y_1 says that at least one λ_i where $i \geq 3$ must be true.

$\{\neg Y_{i-2}, \lambda_i, Y_{i-1}\}$ says that if Y_{i-2} is true (i.e. at least one λ_k with $k \geq i$ is true) then λ_i or some literal with index greater than i is true.

Finally,

$$\tau(F) =_{def} \{\tau(C) : C \in F\}$$

3-SAT is NP-complete (continued)

Clearly, τ is polynomial.

To complete the proof we need to show that

$$F \in SAT \text{ iff } \tau(F) \in 3\text{-}SAT$$

We divide the proof into two parts:

(A) If $F \in SAT$ then $\tau(F) \in 3\text{-}SAT$.

(B) If $\tau(F) \in 3\text{-}SAT$ then $F \in SAT$.

Part A

Suppose I is a model of F. Let C be an arbitrary clause of F. To define a satisfying assignment of $\tau(C)$ let us assume that λ_i is the first literal in C made true by I and expand I as follows:

- Set variables $Y_1, \ldots, Y_{i-2}$ to true.
- Set variables $Y_{i-1}, \ldots, Y_{n-3}$ to false.

Clearly, the result is a model of $\tau(C)$. Repeat the process for all C's from F. Since all the new variables in different clauses are different the resulting assignment is a model of $\tau(C)$.

Part B

Suppose I is a model of $\tau(F)$. It is not difficult to show that I is also a model of F.

Consider arbitrary $C \in F$ and show that at least one $\lambda_i \in C$ is satisfied by I.

Suppose it is not the case.

Then, to satisfy the first $n - 3$ clauses of $\tau(C)$, I must set $Y_1, \ldots, Y_{n-3}$ to true. But this would mean that the last clause is not satisfied. Contradiction.

The assumption is false, I satisfies C and, hence, F.

MaxSat is NP-complete

MaxSat: Given a set F of clauses and a natural number K check if there is a truth assignment which satisfies at least K clauses of F.

To show that MaxSat is NP-complete consider a mapping τ which maps a boolean formula F to a pair $\langle F, k \rangle$ where k is the number of clauses in F.

Clearly, τ is polynomial and $F \in SAT$ iff $\tau(F) \in MaxSat$.

τ is a polynomial reduction and hence $MaxSat$ is NP-complete.

Other NP-complete problems

Hamiltonian Cycle: Given a graph G is there a cycle which passes through each node of G exactly once?

Independent Set: Given an undirected graph $G = (V, E)$ and an integer $K \geq 2$, is there a subset C of V such that $|C| \geq K$ and no two edges of C are connected by an edge from E?

3-coloring: Given an undirected graph $G = (V, E)$ can we color the nodes in three different colors so that no nodes connected by an edge are of the same color?

Fake-Coin Problem

Among n identically looking coins one is fake, i.e. is lighter than a regular coin. Design an efficient algorithm to determine which coin is fake using a balance scale.

```
function fake(set $S$ of coins)
% $S$ contains one fake coin % return the fake coin
Divide $S$ into two halfs, $S_1$ and $S_2$ with possibly one coin, c left
on the table.
If $weight(S_1) = weight(S_2)$ then $return(c)$.
If $weight(S_1) < weight(S_2)$ then $return(fake(weight(S_1))$.
$return(fake(weight(S_2))$.
```

Fake-Coin Problem

$W(n)$ – number of weighs needed to find the fake coin in a set of n coins.

$$W(1) = 0$$

$$W(n) = W(\lfloor n/2 \rfloor) + 1 \text{ for } n > 1$$

Solve for $n = 2^k$.
$W(n) = W(2^k) = W(2^{k-1}) + 1 = W(2^{k-2}) + 1 + 1 = k + 1 = \log_2(n) + 1$
Check:
$W(2 \times 2^{n-1}) = \log_2(2) + \log_2(2^{n-1}) = W(\lfloor n/2 \rfloor) + 1$

Fake-Coin Problem

What if we divide into three parts, S_1, S_2, S_3?

1. No coin remains. Find the lightest, S (one weighing) and call $fake(S)$

2. One coin, c, remains.

```
If $weight(S_1) = weight(S_2)$ then
    select $c_1$ from $S_1$
    if $weight(c_1) > weight(c)$ then $return(c)$
    $return(fake(S_3))$
etc.
```

Reccurrence Relation:

$$W(n) = W(\lfloor n/3 \rfloor) + 3 \text{ for } n > 3$$

Since $\log_3 < \log_2$ this method is faster. (8 versus 13 for $n = 10000$)

Evaluating a polynomial

Given: $p(X) = a_n X^n + a_{n-1} X^{n-1} + \dots a_0$ and the value, x of X.

Efficiently compute: $p(x)$.

A good representation of $p(X)$ is obtained by successively taking X as a common factor from remaining polynomials of diminishing degrees (Horner Rule):

$p(X) = 2X^4 - X^3 + 3X^2 + X - 5 =$
$X(2X^3 - X^2 + 3X + 1) - 5 =$
$X(X(2X^2 - X + 3) + 1) - 5 =$
$X(X(X(2X - 1) + 3) + 1) - 5 =$

Substantially decreases the number of operations.

Evaluating a polynomial

```
$horner(P[0..n], x)$
$P[0..n] = (a_0, \ldots, a_n)$
$V := P[0]$
for $i$ from $n - 1$ to $0$ do
    $V := x \times V + P[i]$
```

Computing exponentiation

Let $n = b_k \dots b_0$ be a binary string representing a positive integer n and $p(x) = b_k x^k + \dots + b_i x^i + \dots + b_0$. Clearly, $n = p(2)$

Computing a^n

Horner rule for $p(2)$:

```
$V := 1$
for $i$ from $k - 1$ to $0$ do
    $V := 2V + b_i$
```

Computing $a^{p(2)}$:

```
$a^V := a^1$
for $i$ from $k - 1$ to $0$ do
    $a^V := a^{2V+b_i}$
```

Computing a^n

$a^{2V+b_i} = a^{2V} \times a^{b_i} = (a^V)^2 \times a^{b_i} =$
$(a^V)^2$ if $b_i = 0$; $(a^V)^2 \times a$ otherwise

```
$V := a$
for $i$ from $k - 1$ to $0$ do
$V := V \times V$
if $b_i = 1$ then $V := V \times a$
return$(V)$
```

Hospital Residency

We use variable h for hospitals and s for student.

```
*function* match
*while* $\exists$ h (open_slot(h) $\neq 0$) *do*
    1. Select a hospital h with an open slot.
    2. let h offer position to highest-ranked student s left
       on the list.
    3. *if* s is free let s accept and decrease open_slot(h)
       *otherwise*
    4. *if* s prefers h to h$'$ to which s is committed let her
       break the engagement and accept h's proposal;
       decrease open_slot(h) and increase open_slot(h$'$).
*return* the set of all engaged pairs.
```

Minimum Bottleneck Spanning Tree (Ch4, num 9)

Given: Spanning tree T with positive and distinct costs of edges.

***Bottleneck* of T is the edge with the greatest cost.**

Question 1: Is every minimum-bottleneck spanning tree of G also a minimum spanning tree of G?

NO. Consider graph with nodes a, b, c, and d. Every pair of nodes is connected by an edge. $c(a,b) = 8$, $c(a,c) = 6$, $c(b,d) = 10$, $c(c,d) = 12$, $c(c,b) = 2$, $c(a,d) = 11$. **Min-bottlneck** $= (c,a,b,d)$, **Min-spanning** $= (d,c,b,a)$.

Chapter 6, Problem 10

Given: n **minutes, two computers** a **and** b, a_i **- value of running machine** a **during minute** i**. Similarly for** b_i**.**

Find: Use the computers to run your job to achieve maximum value. Constraints: Job can run on one machine at any given minute. A minute is required to change the machine.

$opt(i : minutes)$ **- best value for job allocation during first** i **minutes.**
$opt(i : minutes, C : computer)$ **- best value for job allocation during first** i **minutes which ends in job being allocated to** C**.**
$opt(i) = max(opt(i, a), opt(i, b))$
$opt(i, a) = a_i + max(opt(i - 1, a), opt(i - 2, b))$

Chapter 6, Problem 10

```
$opt(n : min) : time$
$var\ M[0..n, a..b] : time$
$M[0, a], M[0, b] := 0$
$M[1, a] := a_1$
$M[1, b] := b_1$
$For\ i = 2\ to\ n\ do$
$M[i, a] := a_i + max(M[i-1, a], M[i-2, b])$
$M[i, b] := b_i + max(M[i-1, b], M[i-2, a])$
$return\ max(M[n, a], M[n, b])$
```

Chapter 6, Problem 10

```
$output\_plan(M)$
% outputs optimal allocation for $n$ minutes, $n \geq 0$
if $M[n, a] > M[n, b]$ then out(M,n,a)
else out(M,n,b)

$out(M, i, C)$
% outputs optimal allocation for $i$ minutes which ends in
% machine C; $\bar{C}$ stands for “the other machine”.
if $i > 0$ then $print(C, at, i)$
if $i > 1$ then
    if $M[i-1, C] > M[i-2, \bar{C}]$ then $out(M, i-1, C)$
    else $out(M, i-2, \bar{C})$
```

Chapter 6, Problem 11

Given: a sequence $\langle s_1, \ldots, s_n \rangle$ where i is a week and s_i is a number of pounds to be shiped in this week.

Do: For every i schedule company A or company B to ship s_i.

$\text{cost}(A, i) = r \times s_i$

$\text{cost}(B, i) = c$

Constraints: A contract with B must be made in blocks of four consequitive weeks. Ensure minimal cost.

Chapter 6, Problem 11

To find minimal cost divide the problem of scheduling shipments for i weeks into two. In one, the last shipment is done by A, in the other one – by B.

Recurrence relation:

if $i - 4 > 0$ then

$$opt(i) = \min(opt(i-1) + r \times s_i, opt(i-4) + 4 \times c)$$

else

$$opt(i) = r \times s_1 + \cdots + r \times s_i.$$

Chapter 6, Problem 11

```
function opt(i : index) : cost
var M[0..i].
M[0] := 0
for j from 1 to i do
if $j - 4 > 0$ then
    $M[j] := \min(M[j-1] + r \times s_j, M(j-4) + 4 \times c)$
etc
return(M)
```