Algorithm design patterns

3.1 "Brute force"

Straightforward "brute force" is often beaten by more advanced design methods
but sometimes sufficient,
and with some hard problems essentially the best we can do

Example Brute-force computation of \( a^n \) \((n \in \mathbb{N})\) takes \( n \) multiplications;

"Decrease-by-half" leads to exponentially fewer multiplications

Example Naive string pattern matching

Input: Strings \( P = p_1 \ldots p_m \) (pattern, hahmao) and \( T = t_1 \ldots t_n \) (target, kohde)

(often \( m \ll n \))

\( \text{find}(P, T) = [\min\{i \mid t_1 \ldots t_{i+m-1} = p_1 \ldots p_m\} \]

Applications: editors, text searching (grep), bio-sequence (DNA, protein) databases

procedure naiveSearch(\( P[1..m], T[1..n] \) of char)

for \( i := 1 \) to \( n - m + 1 \) do

\( l := 0; \)

while \( l < m \) and \( T[i+l] = P[l+1] \) do

\( l := l + 1; \)

if \( l = m \) then return/write \( i; \)

Example Naive search of \( abaab \) on target \( ababbaabba \):

\[
T: \begin{array}{cccccccc}
  & a & b & a & b & a & a & b & a \\
  a & b & a & a & b & a \\
  a & b & a \\
  a & b & a & a & b \\
  
\end{array}
\]

In the worst-case, the entire pattern is compared against each substring of \( m \) chars
\( \sim T_{\text{max}} = \Theta(mn) = \Theta(n^2) \),
but \( T_{\text{avg}} = O(n) \) only, on random texts

Sufficient for text processing, but more efficient search is needed, for example, in sequence databases

Problem was solved in optimal time \( O(m + n) \) first time with the method of
Knuth, Morris & Pratt (KMP)

KMP beats naive search, in practice, on binary alphabets only

Example Exhaustive search (kattava haku) examines each alternative

Problem is a minimal Hamiltonian cycle in a graph with \( n \) vertices

\( n! \) candidate solutions

Search space can be pruned:
(Why?)

Fix the initial vertex, say, \( a \)

If the graph is undirected, restrict to paths that visit selected vertices \( b \) and \( c \) in this order
Example (Weighted knapsack; painotettu reppu)

Maximize the value of a load that fits in volume $T$, selecting from items whose volumes and values are $((t_1,a_1),\ldots, (t_n,a_n))$

Exhaustive search generates all $2^n$ subsets, and chooses a maximally valuable load that fits in the "knapsack"

Example: $T = 10$

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>value</td>
<td>42</td>
<td>12</td>
<td>40</td>
<td>25</td>
</tr>
</tbody>
</table>

Supersets of too large loads can be ignored (Why?)

<table>
<thead>
<tr>
<th>load</th>
<th>volume</th>
<th>value</th>
<th>load</th>
<th>vol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>0</td>
<td>0</td>
<td>{1,3}</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>{1}</td>
<td>7</td>
<td>42</td>
<td>{1,4}</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>{2}</td>
<td>3</td>
<td>12</td>
<td>{2,3}</td>
<td>7</td>
<td>52</td>
</tr>
<tr>
<td>{3}</td>
<td>4</td>
<td>40</td>
<td>{2,4}</td>
<td>8</td>
<td>37</td>
</tr>
<tr>
<td>{4}</td>
<td>5</td>
<td>25</td>
<td>{3,4}</td>
<td>9</td>
<td>65</td>
</tr>
<tr>
<td>{1,2}</td>
<td>10</td>
<td>54</td>
<td>{2,3,4}</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

The number of alternatives, $(n-1)!/2$, grows explosively;

For example, $n = 11 \approx 1.8 \times 10^6$

TSP and Knapsack are examples of so-called NP-hard (NP-vaikaa) problems

We’ll discuss later more about their relationship with other hard problems, and about their heuristic and approximate solutions

3.2 Divide and conquer

"Hajota ja hallitse"

1. Divide the task into sub-tasks

2. Solve the sub-tasks

3. Combine sub-solutions into full solution

$T(n)$ often by Theorems 2.3 and 2.4
Example (Merge sort; limityslajittelu)

To sort \(a[i..j] \ (i \leq j)\):

- if \(i = j\), done; otherwise ...
- split into subtasks \(a[i..k]\) and \(a[k + 1..j]\)
- solve them recursively
- (combine): merge the sorted sub-arrays

**procedure** mergeSort(modifies \(a[i..j]\))

if \(i < j\) then

\[ k := \lfloor (i + j) / 2 \rfloor; \]
mergeSort \((a[i..k])\);
mergeSort \((a[k + 1..j])\);
merge \((a, i, k, j)\);

Merge \(a[i..k]\) and \(a[k + 1..j]\) using aux. space

Idea: \(a[i..k \mid k + 1..j]\)

I: \(b[i] \leftarrow \min \{a[i], a[m]\}\)
(or the first of the remaining)

II: \(a[i..j] \leftarrow b[i..j]\)

**procedure** merge(modifies \(a : array\); \(i, k, j: integer\))

\(l := i; \ m := k + 1; \ t := i;\)
while \(l \leq k\) or \(m \leq j\) do

if \(l > k\) then \(b_l := a_m; \ m := m + 1;\)
elseif \(m > j\) then \(b_j := a_l; \ l := l + 1;\)
elseif \(a_l \leq a_m\) then \(b_l := a_l; \ l := l + 1;\)
else \(b_l := a_m; \ m := m + 1;\)
\(t := t + 1;\)
for \(t := i\) to \(j\) do \(a_t := b_t;\)

We get the following recurrence for the time complexity of merge sort:

\[ T(n) = T(\lfloor n / 2 \rfloor) + T(\lfloor n / 2 \rfloor) + O(n) \]
\[ \leq 2 T(\lfloor n / 2 \rfloor) + O(n) \]

Theorem 2.4 applies
with \(b = 2, \ c = 2, \ d = 1\)

\(~ T(n) = O(n \log n)~\)

Merge sort is an asymptotically optimal sorting algorithm

Example Quicksort (pikalajittelu)

Divide-and-conquer sort, based on partitioning the sequence by values of its keys

Algorithm on a high level:

**procedure** quickSort(S: sequence of keys):

if \(|S| \leq 1\) then return \(S;\)
else

Choose a pivot \(s \in S;\)
Partition \(S\) into \(S_{<s}, \ S_{=s}\) and \(S_{>s};\)
return quickSort \((S_{<s}) \circ S_{=s} \circ \text{quickSort}(S_{>s})\);

Correctness by induction on the length \(|S|\)

Complexity?
Local work is $\Theta(n)$

Each $s \in S$ is used as the pivot at most once, and each pivot leads to two recursive calls $\sim T_{\text{max}} = O(n^2)$

The bound is tight:
Partition requires time $\geq bn$ (for some $b$).
In the worst case the pivot is the minimum or the maximum $\sim$ \[ T_{\text{max}}(n) \geq bn + T(n - 1) \]
\[ \geq bn + b(n - 1) + T(n - 2) \]
\[ \geq b(n + (n - 1) + \ldots + 2) \]
\[ = b \sum_{i=2}^{n} i = \Theta(n^2) \]

On the other hand, $T_{\text{avg}}(n) = O(n \log n)$:

Let $T(n) := T_{\text{avg}}(n)$.
Assume that we are sorting evenly distributed permutations of disjoint keys

$T(0) \approx T(1) \leq a$ for some constant $a$

When $n \geq 2$: Local work takes time $bn$ (for some $b$). The final position of the pivot may be any $i = 1, \ldots, n$ with the same probability $1/n$, and $|S_{<i}| = i - 1$ and $|S_{>i}| = n - i$. Thus

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

Show that $T(n) \leq cn \log n$
by induction on $n \geq 2$, where $c = 2(a + b)$:

$T(2) \leq 2b + 2a = 2(a + b)2 \ln 2$; OK

Write the upper bound (1) as

$bn + 2/n[T(0) + T(1) + \sum_{i=2}^{n-1} T(i)]$
\[ \leq bn + 4a/n + 2c/n \sum_{i=2}^{n-1} i \ln i \] (ind. ass.)
The sum $\sum_{i=2}^{n-1} i \ln i$ can be approximated upwards with $\int_{2}^{n} x \ln x \, dx$:

\[ \int_{2}^{n} x \ln x \, dx = \left[ \frac{n^2 \ln n}{2} - \frac{n^2}{4} - (4 \ln 2 - 1) \right] \]
\[ \leq \frac{n^2 \ln n}{2} - \frac{n^2}{4} \]

Therefore $T(n) \leq bn + 4a/n + cn \ln n - cn/2$

Now $bn + 4a/n \leq cn/2$, and so $T(n) \leq cn \ln n$

Merge sort wins quicksort in the worst case, but QS is faster on the average in practice, and thus popular
Generally it is beneficial to divide the problem instance to subproblems of roughly equal size

For example, *merge sort* using uneven partition $a[1\ldots j−1]$ and $a[j]$ would be essentially similar to *insertion sort*

\[
T(n) = \begin{cases} 
  a & \text{when } n = 1, \\
  T(n−1) + dn + e & \text{when } n > 1.
\end{cases}
\]

\[T(n) = \Theta(n^2) \quad (\text{\textit{Theorem 2.3}})\]

Example: Multiplication of $n$-digit numbers using the school method

\[
\begin{array}{c}
  3624 \\
  2345 \\
  \hline
  18120 \\
  14496 \\
  10872 \\
  7248 \\
  \hline
  8498280
\end{array}
\]

Number of basic operations is $\Theta(n^2)$.

Could we do with less?

(DC multiplication by Karatsuba & Ofman)

Let $x$ and $y$ be $n$-digit numbers ($n = 2^p$, with base $k$). Split them in two halves:

\[
\begin{align*}
x &= s \times k^{n/2} + t \\
y &= u \times k^{n/2} + v
\end{align*}
\]

Compute using the split:

\[
x \times y = (s \times k^{n/2} + t) \times (u \times k^{n/2} + v)
= su \times k^n + (sv + tu) \times k^{n/2} + tv
\]

Four $n/2$-digit multiplications + local work:

\[
T(n) = 4T(n/2) + O(n) = O(n^{\log_2 4})
\]

Three $n/2$-digit multiplications suffice:

\[
x \times y = su \times k^n \\
\quad + ((s + t)(u + v) - su - tv) \times k^{n/2} + tv
\]

\[
T(n) = 3T(n/2) + O(n) = O(n^{\log_2 3}) = O(n^{1.59})
\]

Refinement: $s + t$ and $u + v$ may have $n/2 + 1$ digits. If so, extract their most significant digits $s_1$ and $u_1$:

\[
s + t = s_1 k^{n/2} + t_1
u + v = u_1 k^{n/2} + v_1
\]

Then

\[
(s + t)(u + v) = (s_1 k^{n/2} + t_1)(u_1 k^{n/2} + v_1)
= s_1 u_1 k^n + (s_1 v_1 + u_1 t_1) k^{n/2} + t_1 v_1
\]

This can be done with one $n/2$-digit multiplication and $O(n)$ local basic ops

The school method is more efficient in practice, up to a few-hundred-digit numbers
Binary search trees

Tree structures naturally lead to Div&Conq

A binary search tree is a binary tree where

1. internal nodes contain one or two children, n.left and n.right;
   if v is a child of u, then u is v's parent

2. each node n carries a key n.key
   from an ordered type

3. each key k in the left (corresponding right) subtree of n satisfies
   k ≤ n.key (correspondence k > n.key)

Example:

```
    Maija
   /     /
Ilkka  Rajja
  /     /   
Antti Kaija  Tauno
       /     /
          Risto
```

3.3 "Decrease and Conquer"

Sometimes the problem reduces to a single smaller instance ("Div&Conq": several)

Insertion sort ("decrease-by-one"):
1. Sort (a₁, ..., aᵢ−1);
2. Insert aᵢ at correct position

Sometimes the instance can be decreased by a constant factor (like 1/2), or by an amount that depends on the instance at hand

Example Computing a^n with "decr-by-half"

"Decrease-by-one":

\[ a^n = a \cdot a^{n-1} = \cdots = a \cdot a \cdot \cdots \cdot a \]

\[ a^n = \begin{cases} 
   (a^n/2)^2 & \text{when } n \text{ is even and } n > 0, \\
   (a^{(n-1)/2})^2 \cdot a & \text{when } n \text{ is odd and } n > 0, \\
   1 & \text{when } n = 0.
\end{cases} \]

Number of multiplications is only \(O(\log n)\)

Example Binary search in an ordered array

```
procedure binSearch(A[1..n]; k: key)
returns index: // -1, if key not found
        d := 1; u := n;
while d ≤ u do
   m := [(d + u)/2];
   if k < A[m] then u := m − 1;
   else if k = A[m] then return m;
   else d := m + 1; // k > A[m]
endwhile;
return −1;
```

At each iteration, the range d...u is reduced by half (at least)
Therefore

\[ T(n) \leq T\left(\lfloor n/2 \rfloor \right) + O(1) \]
\[ \leq T\left(\lceil n/2 \rceil \right) + O(1) \]

Theorem 2.4 applies, with

\[ b = 1, c = 2, \text{ and } d = 0 \]

\[ \sim T(n) = \sim O(1) \]

Breadth-first search uses a queue instead:

**procedure** BFS\(v\)\(;\) vertex:

\( v.\text{marked} := \text{true}; \ Q := \text{new Queue}(v) ; \)

while \( Q \neq \emptyset \)

\( u := \text{Dequeue}(Q) ; \text{Process} \ u ; \)

for each successor \( s \) of \( u \)

\( \text{if not} \ s.\text{marked} \text{ then} \)

\( s.\text{marked} := \text{true}; \text{Enqueue}(Q, s) ; \)

end while ;

Example (The \( k \)th smallest item of a set)

By sorting: \( A[1..|S|] \leftarrow \text{Sort}(S) ; \text{return} \ A[k] ; \)

→ time \( O(n \log n) \)

"Decrease-and-conquer":

**proc** select\( (k; \) int, \( S; \) set) \text{returns} item:

**if** \( S = \{a\} \) \text{then return} \ a ;

**else** // \( |S| > 1 \)

Choose \( a \in S \); Partition \( S \) into \( S_{<a}, S_{=a}, S_{>a} \);

**if** \( |S_{<}| \geq k \) \text{then return} select\( (k, S_{<}) ; \)

**elsif** \( |S_{<}| + |S_{=}| \geq k \) \text{then return} \ a ;

**else return** select\( (k - |S_{<}| - |S_{=}|, S_{>}) ; \)

(Cf. quicksort)

\( T_{\text{max}}(n) = \Theta(n^2) \)

but \( T_{\text{avg}}(n) = O(n) \)
Ex. Search and insertion in binary trees

Search reduces to examining a smaller subtree determined by the key:

```plaintext
procedure search(a : key, v : node)    returns node: // or ∅ if not found
if v = ∅ then return ∅;
elsif a = v.key then
    return v;
elsif a < v.key then
    return search(a, v.left);
else // a > v.key
    return search(a, v.right);
```

Sub-instance size depends on tree structure

In a balanced tree the size is roughly halved, even in the worst case

\[ T_{max} = \Theta(\log n) \]

What about \( n \) insertions into a tree that remains balanced: \( \Theta(\log 1 + \cdots + \log n) \)?

\[
\sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n
\]

On the other hand

\[
\sum_{i=1}^{n} \log i = \log(1 \cdot 2 \cdots \lceil n/2 \rceil \cdot \lfloor n/2 \rfloor \cdots n)
\]

\[
\sum_{i=1}^{n} \log i \geq \log \lceil n/2 \rceil \lfloor n/2 \rfloor
\]

\[
\geq \log(n/2)^{(n/2)}
\]

(2) and (3) \( \Rightarrow \Theta(n \log n) \)

In a tree degraded to a chain, \( n \) insertions require time

\( \Theta(1 + 2 + \cdots + n) = \Theta(n^2) \)

Also an unbalanced tree can be efficient in practice: Inserting \( n \) disjoint keys succeeds in expected time \( O(n \log n) \)