This lecture contains basic information about three different and often used classes of algorithms: Divide and Conquer, Greedy algorithms and Dynamic programming.

1 Divide and Conquer

Divide and conquer algorithms can be generalized in three steps.

1. Divide the main problem into two or more subproblems.
2. Solve each subproblem recursively.
3. Combine the solutions for the subproblems to build a solution for the main problem.

Algorithm 1: Merge Sort

Input: List $L = [u_1, u_2, \ldots, u_n]$

Output: The elements in $L$ in sorted order.

```
MERGESORT(L)
(1) if $|U| = 1$
(2) return $L$
(3) $L_l \leftarrow$ MERGESORT($[u_1, \ldots, u_{n/2}]$)
(4) $L_r \leftarrow$ MERGESORT($[u_{n/2}+1, \ldots, u_n]$)
(5) return MERGE($L_l, L_r$)
```

2 Greedy algorithms

2.1 Disjoint intervals

Example: Activities in lecture hall

Problem: Activities are to be held in a lecture hall. The problem is that all activities do not fit in the same hall. Each activity has a starting and an ending time. The task is to fit in as many activities as possible into the schedule.

Formally: Given a set of time intervals $[s_1, f_1), \ldots, [s_n, f_n]$, choose the maximum number of mutually disjoint intervals.
Solution: Choose the activity which ends first and does not conflict with earlier chosen activities.

**Algorithm 2:** Maximum number of mutually disjoint intervals, 

**Input:** The intervals \([s_1, f_1], \ldots, [s_n, f_n]\).

**Output:** The largest subset of mutual disjoint intervals.

\[\text{DISJOINTINTERVAL}([s_1, f_1], \ldots, [s_n, f_n])\]

1. Sort the intervals so that \(f_1 \leq f_2 \leq \ldots \leq f_n\).
2. \(S \leftarrow \{1\}, j \leftarrow 1\)
3. for \(i = 2\) to \(n\)
4. \(\text{if } s_i \geq f_j\)
5. \(S \leftarrow S \cup \{i\}\)
6. \(j \leftarrow i\)
7. return \(S\)

**Greedy choice:** The interval \(I_{\text{min}}\) which contains the earliest ending time is a part of an optimal solution.

**Optimal substructure:** The optimal solution to \(I_1, \ldots, I_n - I_{\text{min}}\) combined with \(I_{\text{min}}\) is the optimal solution to \(I_1, \ldots, I_n\).

If we take the interval \(I_{\text{min}}\) which has the earliest ending time then we can change the first interval in any optimal solution to \(I_{\text{min}}\) and still have an optimal solution. This can then be repeated for \(I - I_{\text{min}}\) and so on. Therefore greedy choice always finds an optimal solution.

### 2.2 Interval covering

**Example:** Lighting an old car tunnel

Problem: In an old car tunnel there are some lamps located unevenly throughout the tunnel. Each lamp has a different lighting range. The task is to light up the whole tunnel while using as few lamps as possible.

Formally: Given an interval \([s, f]\), and a set of intervals \([s_1, f_1], \ldots, [s_n, f_n]\). Choose a minimal subset \(S \subset \{1, \ldots, n\}\) such that \([s, f] \subset \bigcup_{i \in S} [s_i, f_i]\).

Solution: Choose that lamp which covers the entrance of the tunnel and has the longest lighting range into the tunnel. Continuously choose the lamp which overlap the lighted part and has the longest range further into the tunnel - until you reach the exit.

**Greedy Choice:** The interval \(I_{\text{max}}\) which contains \(s\) and has the largest ending point \(f_{\text{max}}\) is a part of an optimal solution.
Greedy algorithms and dynamic programming

Algorithm 3: Interval covering.

**Input:** Goal interval \([s, f]\) and the intervals \([s_1, f_1], \ldots, [s_n, f_n]\) such that \([s, f] \subset \bigcup [s_i, f_i]\).

**Output:** A smallest subset \(S \subset \{1, \ldots, n\}\) such that \([s, f] \subset \bigcup_{i \in S} [s_i, f_i]\).

\[
\text{IntervalCover}([s_1, f_1], \ldots, [s_n, f_n])
\]

\[
(1) \quad S \leftarrow \emptyset, t \leftarrow s \\
(2) \quad \text{while } t < f \\
(3) \quad f_i \leftarrow \max \{f_j | s_j \leq t \leq f_j\} \\
(4) \quad S \leftarrow S \cup \{i\} \\
(5) \quad t \leftarrow f_i \\
(6) \quad \text{return } S
\]

**Optimal Substructure.** The optimal solution to \([f_{max}, f], \{I_1, \ldots, I_n\}\) combined with \(I_{max}\) is the optimal solution for \(\{I_1, \ldots, I_n\}\).

If we take the interval \(I_{max}\) which contains the starting point \(a\) and has the largest ending point then we can change the first interval in any optimal solution to \(I_{max}\) and still have an optimal solution. This can then be repeated for \(I - I_{max}\) and so on. Therefore greedy choice always finds the optimal solution.

2.3 Continuous knapsack problem

Recall the usual knapsack problem: We have \(n\) items and item \(i\) has value \(v_i\) and weight \(w_i\). And a bag which can carry a maximum weight of \(c\). The task is to maximize the sum of the items’ value in the bag. However, this problem is NP-complete and we will not try to solve it. We will instead solve a modified version.

Problem: We now have \(n\) piles of different metal grain. Each pile contains a specific weight of metal grain, which has a specific value per weight. A thief carrying a bag with a weight capacity of \(c\) wants to maximize the value of his loot. The task is to figure out how much of each pile he should steal.

Formally: Given a knapsack with capacity \(c\) and a set of value/weight-pairs \([\langle v_1, w_1 \rangle, \ldots, \langle v_n, w_n \rangle]\). Choose \([a_1, \ldots, a_n]\) such that \(a_i \leq w_i\) and \(\sum_{i=1}^n a_i \cdot v_i / w_i\) is maximized.

Solution: Choose the grain with the highest value per weight and take as much as you can and continue until the bag is full or you run out of grain.

2.4 Huffman coding

Problem: Given a table of how common each letter in an alphabet is, find the prefix-free bit string which gives the shortest expected string length. Prefix-free means that the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol.

Formally: Given a set of letter/weight-pairs \(\{(a_1, f_1), \ldots, (a_n, f_n)\}\) for some alphabet \(A = \{a_1, \ldots, a_n\}\). Find a prefix-free code \(\tau : A \mapsto \{0, 1\}^*\) that minimizes \(\sum_{i=1}^n f_i \cdot |\tau(a_i)|\), where \(|\tau(a_i)|\) is the length of the bit string \(\tau(a_i)\).
Algorithm 4: Continuous knapsack.

**Input:** The capacity $W$ and value/weight-pairs $[(v_1, w_1), \ldots, (v_n, w_n)]$.

**Output:** $[a_1, \ldots, a_n]$, $a_i \leq w_i$, such that $\sum_{i=1}^{n} a_i \frac{w_i}{w_i}$ is maximized.

$\text{ContKnapsack}(W; [(v_1, w_1), \ldots, (v_n, w_n)])$

1. Sort the value/weight-pairs such that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$.
2. for $i = 1$ to $n$
3. $a_i \leftarrow \min\{W, w_i\}$
4. $W \leftarrow W \setminus a_i$
5. return $[a_1, \ldots, a_n]$

![Huffman tree](image)

Figure 1: Example of Huffman coding.

Solution: We identify a prefix-free code $\tau$ with a 0/1-labeled tree. The bit string is represented by the path down the tree. How do we build the tree in the best way? We start at the leaves. We repeatedly choose the two letters with the least weight and join them to form a node of the tree.

In the algorithm we use the notation $a_i \circ a_j$, which denotes concatenation and is simply a way to invent names for internal nodes.

### 3 Dynamic programming

#### 3.1 Binomial coefficients and Pascals triangle

$\binom{n}{k}$ is pronounced “n choose k” and is the number of $k$-element subsets from an $n$-element set.

The definition of binomial coefficients is

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$
Algorithm 5: Huffman coding.

**Input:** Symbol/weight-paris \{\(a_1, f_1\), \ldots, \(a_n, f_n\)\}.

**Output:** 0/1-branch marked tree such that \(\sum_{i=1}^{n} f_i \cdot |\tau(a_i)|\) is minimized.

**HUFFMAN(\{(a_1, f_1), \ldots, (a_n, f_n)\})**

1. \(S \leftarrow \{(a_1, f_1, a_1), \ldots, (a_n, f_n, a_n)\}\)
2. **while** \(|S| \geq 2\)
3. \(\text{Find } (b_i, f_i, t_i), (b_j, f_j, t_j) \in S \text{ with } f_i, f_j \text{ as small as possible.}\)
4. \(S \leftarrow S \setminus \{(b_i, f_i, t_i), (b_j, f_j, t_j)\}\)
5. \(S \leftarrow S \cup \{(a_i \circ a_j, f_i + f_j, \text{TreeNode}((t_i, t_j)))\}\)
6. **return** \(S\).

We calculate these coefficients and make a table.

<table>
<thead>
<tr>
<th>(k)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

If we study this table we can see that the value in the \(n\)th row and the \(k\)th column is the sum of the two values in the row above, \(n-1\), in the \(k\)th column and the column the the left, \(k-1\). This will be even clearer if we write the table as Pascal's triangle. Hence, we can calculate the binomial coefficients with this recursive formula.

\[
\binom{n}{k} = \begin{cases} 1 & \text{if } n = k \text{ or } k = 0 \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{otherwise} \end{cases}
\]

**Pascal's triangle:**

<table>
<thead>
<tr>
<th>(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

3.1.1 Memoization

Dynamic programming divides the problem into smaller subproblems until we reach a simple case which can easily be calculated. These subproblems are then combined to find the solution. Dynamic programming with memoization works in the same way but saves the results of the calculation so that if we have to solve another problem these results can be retrieved instead of recalculated.

The difference can be illustrated in these two example algorithms.
Algorithm 6: Binomial coefficients with memoization.

Input: Integers \( n \) and \( k \), \( 0 \leq k \leq n \).

Output: \( \binom{n}{k} \)

\[
\begin{align*}
& \text{Bin}(n, k) \\
& \text{(1) if } T[n, k] = \perp \\
& \text{(2) if } n = k \lor k = 0 \\
& \text{(3) } T[n, k] = 1 \\
& \text{(4) else} \\
& \text{(5) } T[n, k] \leftarrow \text{Bin}(n - 1, k - 1) + \text{Bin}(n - 1, k) \\
& \text{(6) return } T[n, k]
\end{align*}
\]

Algorithm 7: Binomial coefficients without memoization.

Input: Integers \( n \) and \( k \), \( 0 \leq k \leq n \).

Output: \( \binom{n}{k} \)

\[
\begin{align*}
& \text{Bin}(n, k) \\
& \text{(1) for } i = 0 \text{ to } n \\
& \text{(2) } T[i, 0] \leftarrow T[i, i] \leftarrow 1 \\
& \text{(3) for } i = 2 \text{ to } n \\
& \text{(4) for } j = 1 \text{ to } \min\{i - 1\}{\{k\}} \\
& \text{(5) } T[i, j] \leftarrow T[i - 1, j - 1] + T[i - 1, j] \\
& \text{(6) return } T[n, k]
\end{align*}
\]

3.2 Maximal partial sum

Formally: Given a set of integers \( x_1, \ldots, x_n \), find \( \max_{s \leq f} \{ \sum_{i=s}^{f} x_i \} \)

Algorithm 8: Finding the maximal subarray.

Input: Integers \( x_1, \ldots, x_n \).

Output: \( \max_{s \leq f} \{ \sum_{i=s}^{f} x_i \} \).

\[
\begin{align*}
& \text{MaxSubarray}(x_1, \ldots, x_n) \\
& \text{(1) } S[0] \leftarrow 1 \\
& \text{(2) for } i = 1 \text{ to } n \\
& \text{(3) } S[i] \leftarrow \max\{0, S[i - 1]\} + x_i \\
& \text{(4) return } \max_i \{ S[i]\}
\end{align*}
\]

3.3 Longest increasing subsequence

Example: Circuit plugs

Two circuits has \( n \) plugs each on rows marked 1, 2, ..., \( n \) in some order. We want to connect the plugs pairwise without any connections crossing each other.

Formally: Given a \( n \)-permutation \( \pi \), find the size of the maximal subset \( S \subset \{1, 2, \ldots, n\} \) such that \( \pi(i) \leq \pi(j) \) for all \( i \leq j \) in the set \( S \).
Algorithm 9: Finding the longest increasing subsequence.

Input: Permutation \( \pi \).

Output: The size of the maximal subset \( S \subset \{1, 2, \ldots, n\} \) such that \( \pi(i) < \pi(j) \) for all \( i < j \) in \( S \).

\[
\text{LONGINCSEQ}(\pi)
\]
(1) \( T[0] \leftarrow \infty \)
(2) \( \text{for } i = 1 \text{ to } n \)
(3) \( T[i] \leftarrow \infty \)
(4) \( \text{for } i = 1 \text{ to } n \)
(5) \( \text{Find } j \text{ such that } T[j - 1] < \pi(i) < \pi(j). \)
(6) \( T[j] \leftarrow \pi(i) \)
(7) \( \text{return } \max\{j : T[j] < \infty\} \)