<table>
<thead>
<tr>
<th>1 Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Assignment and Knapsack problem</td>
</tr>
<tr>
<td>2.1 Assignment problem: Greedy heuristic, Complete Enumeration, Branch-And-Bound</td>
</tr>
<tr>
<td>2.2 Knapsack problem: Greedy heuristic, Complete Enumeration, Branch-And-Bound</td>
</tr>
<tr>
<td>2.3 Algorithmic complexity and NP-hard problems: Combinatorial explosion, NP-hard problems, NP-Completeness, Optimization vs. Decision problems, P, NP and reducibility, Approximation algorithms, Heuristics</td>
</tr>
<tr>
<td>2.4 Dynamic Programming and Backtracking</td>
</tr>
<tr>
<td>3 Bin Packing</td>
</tr>
<tr>
<td>4 Scheduling</td>
</tr>
<tr>
<td>5 Graph algorithms</td>
</tr>
<tr>
<td>6 Linear Programming</td>
</tr>
<tr>
<td>7 Heuristics</td>
</tr>
</tbody>
</table>
# Content

<table>
<thead>
<tr>
<th></th>
<th>Algorithms</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Assignment and Knapsack problem</td>
<td>2.1 Assignment problem: Greedy heuristic, Complete Enumeration, Branch-And-Bound</td>
</tr>
<tr>
<td></td>
<td>2.2 Knapsack problem: Greedy heuristic, Complete Enumeration, Branch-And-Bound</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.3 Algorithmic complexity and NP-hard problems: Combinatorial explosion, NP-hard problems, NP-Completeness, Optimization vs. Decision problems, P, NP and reducibility, Approximation algorithms, Heuristics</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.4 Dynamic Programming and Backtracking</td>
<td></td>
</tr>
</tbody>
</table>

| 3 | Bin Packing |
| 4 | Scheduling |
| 5 | Graph algorithms |
| 6 | Linear Programming |
| 7 | Heuristics |
Assignment Problem
Assignment problem

Four students (Julia, Marie, Daniel, Niclas) have to accomplish four tasks (A, B, C, D). Each student must be assigned to exactly one of the tasks.

The students need different amounts of time to accomplish these tasks.

**Question:** What task should be assigned to which student such that the total time needed to complete all four tasks is as small as possible?

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

$n!$ possible solutions
1. **Greedy Heuristic** (greedy = choose the locally best alternative)
   
   1. Look for the smallest number in the table and assign that person to that task.
   
   2. Delete the corresponding row and column from the table and repeat Step 1.

### Minimierungsproblem - Best-First-Search

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>
1. Greedy Heuristic (greedy = choose the locally best alternative)
   1. Look for the smallest number in the table and assign that person to that task.
   2. Delete the corresponding row and column from the table and repeat Step 1.

   Daniel  →  C  →  1
1. **Greedy Heuristic** *(greedy = choose the locally best alternative)*
   1. Look for the smallest number in the table and assign that person to that task.
   2. Delete the corresponding row and column from the table and repeat Step 1.

   Daniel → C  1
   Julia   → B  2

---

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
1. **Greedy Heuristic** (greedy = choose the locally best alternative)
   1. Look for the smallest number in the table and assign that person to that task.
   2. Delete the corresponding row and column from the table and repeat Step 1.

Daniel  → C  1
Julia  → B  2
Niclas  → D  4
1. Greedy Heuristic (greedy = choose the locally best alternative)
   1. Look for the smallest number in the table and assign that person to that task.
   2. Delete the corresponding row and column from the table and repeat Step 1.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>6</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   Daniel → C 1
   Julia → B 2
   Niclas → D 4
   Marie → A 6

   This means that we have an **Upper Bound** of 13.
1. Greedy Heuristic (greedy = choose the locally best alternative)
   1. Look for the smallest number in the table and assign that person to that task.
   2. Delete the corresponding row and column from the table and repeat Step 1.

2. Find a problem instance where the heuristic fails.

---

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>
Minimierungsproblem - Best-First-Search

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

1. Greedy Heuristic (greedy = choose the locally best alternative)
   1. Look for the smallest number in the table and assign that person to that task.
   2. Delete the corresponding row and column from the table and repeat Step 1.

2. Find a problem instance where the heuristic fails.

3. Complete Enumeration.
   How many leaves are constructed? How many vertices in total?
Complete Enumeration

$\text{n! possible solutions}$
## Branch-And-Bound: Row-Min

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

Upper Bound: 13

**Upper Bound** = actual value (e.g. achieved by a heuristic: „a definitely possible result“)
### Branch-And-Bound: Row-Min

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

- **Upper Bound**: 13
- **Lower Bound** = smallest, possibly achievable total sum ("it doesn't go any better than that")
- **Row-Min** = "if each student could do what he/she can do best"

**Row Min**:

- **Julia → A**
- \[9 + 3 + 1 + 4 = 17\]

**Upper Bound** = actual value (e.g. achieved by a heuristic: "a definitely possible result")

- **Upper Bound** = actual value (e.g. achieved by a heuristic: "a definitely possible result")
- **Lower Bound** = smallest, possibly achievable total sum ("it doesn't go any better than that")
- **Row-Min** = "if each student could do what he/she can do best"
### Branch-And-Bound: Row-Min

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Julia</strong></td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td><strong>Marie</strong></td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td><strong>Daniel</strong></td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td><strong>Niclas</strong></td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

**Upper Bound:** 13

**Upper Bound** = actual value (e.g. achieved by a heuristic: „a definitely possible result“)

**Lower Bound** = smallest, possibly achievable total sum („it doesn‘t go any better than that“)

Row-Min „if each student could do what he/she can do best“

Column-Min „if each task could be done in minimal possible time“

→ Choose either Row-Min or Column-Min or the maximum of those two values as Lower Bound.

**Julia → A**

9 + 3 + 1 + 4 = 17
Branch-And-Bound: Row-Min

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

Upper Bound: 13

Julia → A
9 + 3 + 1 + 4 = 17

Upper Bound = actual value (e.g. achieved by a heuristic: „a definitely possible result“)
Lower Bound = smallest, possibly achievable total sum („it doesn‘t go any better than that“)

Row-Min „if each student could do what he/she can do best“
Column-Min „if each task could be done in minimal possible time“
Choose either Row-Min or Column-Min or the maximum of those two values as Lower Bound.

When bound? Whenever Lower Bound („it doesn‘t go any better than that“) ≥ Upper Bound („a definitely possible result“).
### Branch-And-Bound

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

**Upper Bound: 13**

- **Julia \rightarrow A**
  \[9 + 3 + 1 + 4 = 17\]

- **Julia \rightarrow B**
  \[2 + 3 + 1 + 4 = 10\]
## Branch-And-Bound

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Julia</strong></td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td><strong>Marie</strong></td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td><strong>Daniel</strong></td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td><strong>Niclas</strong></td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

**Upper Bound: 13**

- **Julia → A**: 9 + 3 + 1 + 4 = 17
- **Julia → B**: 2 + 3 + 1 + 4 = 10
- **Julia → C**: 7 + 4 + 5 + 4 = 20

Red crosses indicate solutions that are not valid for the current branch.
### Branch-And-Bound

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

- **Upper Bound**: 13

#### Branches:

- **Julia → A**: 9 + 3 + 1 + 4 = 17
- **Julia → B**: 2 + 3 + 1 + 4 = 10
- **Julia → C**: 7 + 4 + 5 + 4 = 20
- **Julia → D**: 8 + 3 + 1 + 6 = 18
# Branch-And-Bound

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

Upper Bound: 13

- Julia → A: 9 + 3 + 1 + 4 = 17
- Julia → B: 2 + 3 + 1 + 4 = 10
- Julia → C: 7 + 4 + 5 + 4 = 20
- Julia → D: 8 + 3 + 1 + 6 = 18
### Branch-And-Bound

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Julia</strong></td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td><strong>Marie</strong></td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td><strong>Daniel</strong></td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td><strong>Niclas</strong></td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

Upper Bound: 13

- **Julia → A**: 9+3+1+4 = 17
- **Julia → B**: 2+3+1+4 = 10
- **Julia → C**: 7+4+5+4 = 20
- **Julia → D**: 8+3+1+6 = 18
- **Marie → A**: 2+6+1+4 = 13

[Diagram showing Branch-And-Bound process with decision nodes and evaluation of tasks assigned to each person]
### Branch-And-Bound

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td><img src="#" alt="Circle" /></td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td><img src="#" alt="Circle" /></td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td><img src="#" alt="Circle" /></td>
<td>8</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td><img src="#" alt="Circle" /></td>
</tr>
</tbody>
</table>

**Upper Bound:** 13

- **Julia → A**: 9 + 3 + 1 + 4 = 17
- **Julia → B**: 2 + 3 + 1 + 4 = 10
- **Julia → C**: 7 + 4 + 5 + 4 = 20
- **Julia → D**: 8 + 3 + 1 + 6 = 18

- **Marie → A**: 2 + 6 + 1 + 4 = 13
- **Marie → C**: 2 + 3 + 5 + 4 = 14

[Branch-and-Bound diagram]

---

Branch-And-Bound

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

Upper Bound: 13
Branch-And-Bound

- all leaves are crossed out
- the greedy solution is in this case the optimal solution

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Marie</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Daniel</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Niclas</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>
Additional question:

How many vertices are constructed with Complete Enumeration?

\[ n=2 \quad 2 \]
\[ n=3 \quad 9 \]
\[ n=4 \quad 40 \quad \text{(note that we had to construct only 7 with Branch-And-Bound in our example!)} \]
\[ n=5 \quad 205 \]

General case?

\[
\sum_{i=1}^{n-1} \frac{n!}{i!} = n! \cdot \sum_{i=1}^{n-1} \frac{1}{i!} = n! \cdot \sum_{i=1}^{n} \frac{1}{i!} - 1
\]
\[
= (e - 1) \cdot n! - 1 \approx 1.71828182845905 \cdot n! - 1
\]
\[
= O(n!)
\]
**Assignment problem**

**Remark:**
There is actually a polynomial time algorithm to solve the Assignment problem optimally:

http:www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf

H. W. Kuhn (1955):
The Hungarian method for the assignment problem,
*Naval Research Logistics Quarterly* 2, 83–97

J. Munkres (1957):
Algorithms for the Assignment and Transportation Problems,
*Journal of the Society of Industrial and Applied Mathematics* 5, 32–38
Problem

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

a) Compute a feasible solution with the „Smallest-Number-Heuristic“.

b) Solve the problem with *Branch-And-Bound*:

1. Use the solution value of a) as the first upper bound UB1.
2. Use the „Row-min-strategy“ and always continue at that vertex with the smallest lower bound.
3. Number the vertices in the order they are constructed.
4. In the end, you should have computed two Upper Bounds UB1, UB2 and seven vertices.
5. You can cross out a vertex when all assignments are done (that comes out of the Upper Bound - computation) or if a Lower Bound is \( \geq \) to an Upper Bound, i.e. UB1, UB2 for this problem instance.

c) How many vertices do you not have to compute when you use *Branch-And-Bound* instead of *Complete Enumeration* (for this problem instance)?
## Branch-And-Bound

<table>
<thead>
<tr>
<th></th>
<th>MAX (Knapsack)</th>
<th>MIN (Assignment)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lower Bounds</strong></td>
<td>Largest actually possible value („a definitely possible result“)</td>
<td>Smallest possibly achievable value („it doesn‘t go any better than that“)</td>
</tr>
<tr>
<td><strong>Upper Bounds</strong></td>
<td>Largest possibly achievable value („it doesn‘t go any better than that“)</td>
<td>Smallest already achieved value („a definitely possible result“)</td>
</tr>
<tr>
<td><strong>Bound</strong></td>
<td>Upper Bound ( \leq ) Largest Lower Bound</td>
<td>Lower Bound ( \geq ) Smallest Upper Bound</td>
</tr>
<tr>
<td></td>
<td>Knapsack full</td>
<td>Assignment done</td>
</tr>
</tbody>
</table>
# Content

<table>
<thead>
<tr>
<th></th>
<th>Algorithms</th>
<th>Assignment and Knapsack problem</th>
<th>Bin Packing</th>
<th>Scheduling</th>
<th>Graph algorithms</th>
<th>Linear Programming</th>
<th>Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithms</td>
<td>2.1 Assignment problem: Greedy heuristic, Complete Enumeration, Branch-And-Bound</td>
<td>2.3 Algorithmic complexity and NP-hard problems: Combinatorial explosion, NP-hard problems, NP-Completeness, Optimization vs. Decision problems, P, NP and reducibility, Approximation algorithms, Heuristics</td>
<td>2.4 Dynamic Programming and Backtracking</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Assignment and Knapsack problem</td>
<td>2.2 Knapsack problem: Greedy heuristic, Complete Enumeration, Branch-And-Bound</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Bin Packing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Scheduling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Graph algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Linear Programming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Heuristics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Knapsack Problem
Knapsack problem

Capacity: 24

Knapsack:

Objects:

<table>
<thead>
<tr>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>21</td>
<td>14</td>
</tr>
</tbody>
</table>

Values

Weights

16
3
10
3
14
Knapsack problem

Capacity: 24

Knapsack:

Objects:

Values

Weights
Knapsack problem

Capacity: 24

Knapsack:

Objects:

<table>
<thead>
<tr>
<th>Values</th>
<th>Weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>14</td>
</tr>
</tbody>
</table>

Weights

Values
Knapsack problem

Capacity: 24

Knapsack:

Total Value: 40

Capacity: 24

Total Value: 41

Objects:

Values

23 5 20 12 21

Weights

23 5 20 12 21

16

10

14

16

10

3

3

3

3

23              5 20 12 21

Capacity:

24

Knapsack:

Values

23 5 20 12 21

Weights

23 5 20 12 21

16

10

14

16

10

3

3

3

3
Knapsack:

Capacity: 24

Objects:

23 5 20 12 21

Values

16 10 14

Weights

3 3

Find a heuristic to solve this problem. Describe your algorithm as precisely as possible.
1. Greedy Heuristic „Largest Value“
   1. Look for the largest value, pack that object in the knapsack and delete it from the list.
   If this object does not fit in the knapsack, try the next most valuable object.
   Stop if there are no objects left to choose from.
   2. Repeat Step 1 until no objects are left.

2. Find a problem instance where the largest-value-heuristic fails.

3. Another heuristic is to use the largest relative value (value per unit) instead of the largest value.

4. Find a problem instance where the largest-relative-value-heuristic fails.
Example

Capacity of the knapsack: 5

<table>
<thead>
<tr>
<th>Objects</th>
<th>Values</th>
<th>Weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (Cash)</td>
<td>$v_1 = 6\text{ Mio }$</td>
<td>$w_1 = 1$</td>
</tr>
<tr>
<td>B (Jewellery)</td>
<td>$v_2 = 10\text{ Mio }$</td>
<td>$w_2 = 2$</td>
</tr>
<tr>
<td>C (Paintings)</td>
<td>$v_3 = 12\text{ Mio }$</td>
<td>$w_3 = 3$</td>
</tr>
</tbody>
</table>

You could pack only object A or objects A and B, and so on.
How many solutions are there to this problem
a) for $n=3$ objects?
b) for $n$ objects?

Complete Enumeration:

a) How many leaves are constructed?
b) How many vertices in total?
Example

Capacity of the knapsack: 5

<table>
<thead>
<tr>
<th>Objects</th>
<th>Values</th>
<th>Weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (Cash)</td>
<td>( v_1 = 6 \text{ Mio } $ )</td>
<td>( w_1 = 1 )</td>
</tr>
<tr>
<td>B (Jewellery)</td>
<td>( v_2 = 10 \text{ Mio } $ )</td>
<td>( w_2 = 2 )</td>
</tr>
<tr>
<td>C (Paintings)</td>
<td>( v_3 = 12 \text{ Mio } $ )</td>
<td>( w_3 = 3 )</td>
</tr>
</tbody>
</table>

You could pack only object A or objects A and B, and so on.
How many solutions are there to this problem
a) for \( n=3 \) objects? \( 2^3 = 8 \)
b) for \( n \) objects? \( 2^n \)

Complete Enumeration:
a) How many leaves are constructed? \( 2^n = 2^n \) possible solutions
b) How many vertices in total? \( 2+2^2+2^3+\ldots+2^n = 2^{n+1}-2 \)
Complete Enumeration

Capacity of the knapsack: 5

Object A
Value: 6
Weight: 1

Object B
Value: 10
Weight: 2

Object C
Value: 12
Weight: 3

A
6 | 1

B
10 | 2

C
12 | 3

BC
22 | 5

AC
18 | 4

ABC
28 | 6

(0,0) (12,3) (10,2) (22,5)*** (6,1) (18,4) (16,3) (28,6)
### Branch-And-Bound

<table>
<thead>
<tr>
<th></th>
<th>MAX (Knapsack)</th>
<th>MIN (Assignment)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lower Bounds</strong></td>
<td>Largest actually possible value („a definitely possible result“)</td>
<td>Smallest possibly achievable value („it doesn‘t go any better than that“)</td>
</tr>
<tr>
<td><strong>Upper Bounds</strong></td>
<td>Largest possibly achievable value („it doesn‘t go any better than that“)</td>
<td>Smallest already achieved value („a definitely possible result“)</td>
</tr>
<tr>
<td><strong>Bound</strong></td>
<td>Upper Bound ≤ Largest Lower Bound</td>
<td>Lower Bound ≥ Smallest Upper Bound</td>
</tr>
<tr>
<td></td>
<td>Knapsack full</td>
<td>Assignment done</td>
</tr>
</tbody>
</table>
Branch-And-Bound for the Knapsack problem

A. Selection Rule: Take / Ban the object with the highest / lowest value, weight, relative value ...

B. Graph Traversal: Best First Search, Breadth First Search, Depth First Search

Compute a lower bound („a definitely possible result“) as follows:
Add all the values of the items that are already definitely in the knapsack. Take the remaining objects (that are not banned yet) in the order of their relative values (i.e. value / weight) and put as much of them in the virtual knapsack as possible until the first object does not fit anymore. Important: Then stop at this point. Do not try to pack any other objects that might fit.

Compute an upper bound („it doesn‘t go any better than that“) as follows:
Add to the lower bound the portion of the object that comes next in the order of relative values such that the knapsack would be 100% full.

When bound?
Always when the upper bound (i.e. the best possibly achievable value) is smaller than an existing (i.e. the so far largest) lower bound
OR
when the capacity restriction would be violated.
Example 1

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>90</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>60</td>
</tr>
<tr>
<td>weights</td>
<td>60</td>
<td>30</td>
<td>35</td>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td>Capacity:</td>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Selection Rule: Take at first that object with the largest value.

Graph Traversal: Best First Search (i.e. continue at that vertex with the largest upper bound).

Remarks:

1. There are other Selection Rules such as „Ban at first that object with the largest weight“ and so on. In this class, we only use „Take at first that object with the largest value“.

2. There are other Graph Traversal Strategies like Breath First Search or Depth First Search. In this class, we only use „Best First Search“. 
Example 1

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>90</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>60</td>
</tr>
<tr>
<td>weights</td>
<td>60</td>
<td>30</td>
<td>35</td>
<td>50</td>
<td>30</td>
</tr>
</tbody>
</table>

**Capacity:** 120

**Solution:**

1. **Step:** Find a feasible solution with the Largest-Relative-Value-Heuristic:

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>D</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>60</td>
<td>70</td>
<td>60</td>
<td>80</td>
<td>90</td>
</tr>
<tr>
<td>weights</td>
<td>30</td>
<td>35</td>
<td>30</td>
<td>50</td>
<td>60</td>
</tr>
</tbody>
</table>

60+70+60=190 (Lower Bound)

30+35+30=95, no space left for any other object.

**Remark:** Not necessarily only the first $k$ objects are chosen with this heuristic. Try to pack additional objects in the knapsack as long you can find an object that still fits in the knapsack.

→ Lower Bound is 190.
Example 1

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
<th>E</th>
<th>D</th>
<th>A</th>
<th>Capacity: 120</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>70</td>
<td>60</td>
<td>80</td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>values</th>
<th>weights</th>
<th>relative values</th>
</tr>
</thead>
<tbody>
<tr>
<td>60 70 60 80 90</td>
<td>30 35 30 50 60</td>
<td>2 2 2 1.6 1.5</td>
</tr>
</tbody>
</table>

Selection Rule: Take at first that object with the largest value.

Graph Traversal: Best First Search (best=largest upper bound)

„A in“

Lower Bound

<table>
<thead>
<tr>
<th>values</th>
<th>weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>90 60</td>
<td>60 30</td>
</tr>
</tbody>
</table>

90+60=150

60+30=90 (30 left, next object (C) does not fit in the knapsack anymore)

Upper Bound

Lower Bound + 30x2 (C) = 210

„A out“

Lower Bound

<table>
<thead>
<tr>
<th>values</th>
<th>weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>60 70 60</td>
<td>30 35 30</td>
</tr>
</tbody>
</table>

60+70+60=190

30+35+30=95 (25 left, next object (D) does not fit in the knapsack anymore)

Upper Bound

Lower Bound + 25x1.6 (D) = 230
## Example 2

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>100</td>
<td>162</td>
<td>280</td>
<td>300</td>
<td>320</td>
</tr>
<tr>
<td>weights</td>
<td>10</td>
<td>18</td>
<td>20</td>
<td>25</td>
<td>40</td>
</tr>
</tbody>
</table>

**Capacity:** 80  
**Selection Rule:** Ban at first that object with the largest weight.  
**Graph Traversal:** Breadth First Search
Example 2

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>100</td>
<td>162</td>
<td>280</td>
<td>300</td>
<td>320</td>
</tr>
<tr>
<td>weights</td>
<td>10</td>
<td>18</td>
<td>20</td>
<td>25</td>
<td>40</td>
</tr>
</tbody>
</table>

Capacity: 80
Selection Rule: Ban at first that object with the largest weight.
Graph Traversal: Breadth First Search

Solution:
1. Largest-Value-Heuristic:
   - values: 320 300 280 162 100
   - weights: 40 25 20 18 10
   - 320+300+100=720 (Lower Bound)
   - 40+25+10=75, no space left for THE NEXT object

2. Largest-Relative-Value-Heuristic:
   - values: 280 300 100 162 320
   - weights: 20 25 10 18 40
   - 280+300+100+162=842 (Lower Bound)
   - 20+25+10+18=73, no space left for THE NEXT object

→ Lower Bound is 842.

Note: Here you continue trying to pack an object in the knapsack – you don’t stop at the first object that doesn’t fit (this is different from the B&B procedure as you stop there as soon as an object doesn’t fit anymore!)
## Example 2

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>E</th>
<th>Capacity:</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>values</strong></td>
<td>280</td>
<td>300</td>
<td>100</td>
<td>162</td>
<td>320</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>weights</strong></td>
<td>20</td>
<td>25</td>
<td>10</td>
<td>18</td>
<td>40</td>
<td>Selection Rule:</td>
<td>Ban at first that object with the largest weight.</td>
</tr>
<tr>
<td><strong>relative values</strong></td>
<td>14</td>
<td>12</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>Graph Traversal:</td>
<td>Breadth First Search</td>
</tr>
</tbody>
</table>

**„Ban E“**

**Lower Bound**

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>values</strong></td>
<td>280</td>
<td>300</td>
<td>100</td>
<td>162</td>
</tr>
<tr>
<td><strong>weights</strong></td>
<td>20</td>
<td>25</td>
<td>10</td>
<td>18</td>
</tr>
</tbody>
</table>

280+300+100+162=842 (Lower Bound)

(Next object (E) does not fit in the knapsack anymore)

**Upper Bound**

Lower Bound + 0 = 842 (Object E is banned)

**„Take E“**

**Lower Bound**

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>values</strong></td>
<td>320</td>
<td>280</td>
</tr>
<tr>
<td><strong>weights</strong></td>
<td>40</td>
<td>20</td>
</tr>
</tbody>
</table>

320+280=600 (Lower Bound)

(Next object (D) does not fit in the knapsack anymore)

**Upper Bound**

Lower Bound + 20 x 12 (D) = 840
Problem

Object A B C D E Capacity: 9
Value 12 15 10 8 6
Weight 4 3 5 2 1

a) Build a table with (from left to right) decreasing (non-increasing) relative values.
b) Determine a solution US1 with the Largest-Relative-Value-Heuristic.
c) Solve the problem with Branch-And-Bound.
   1. Start with LB1= that value that you computed in question b).
   2. Selection Rule: Take at first that object with the largest value.
   3. Graph Traversal: Best-First-Search (best=largest upper bound)
   4. Number the vertices in the order they are constructed.
   4. In the end, you should have computed three Lower Bounds LB1, LB2, LB3 and eight vertices.
   5. You can cross out a vertex when the capacity is reached (that comes out of the Lower Bound –
      computation) or if an Upper Bound is <= to a Lower Bound, i.e. LB1, LB2, LB3 for this problem
      instance.
d) How many vertices do you not have to compute when you use Branch-And-Bound instead of
   Complete Enumeration (for this problem instance)?
<table>
<thead>
<tr>
<th></th>
<th>Algorithms</th>
<th>Assignment and Knapsack problem</th>
<th>Bin Packing</th>
<th>Scheduling</th>
<th>Graph algorithms</th>
<th>Linear Programming</th>
<th>Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Assignment and Knapsack problem</td>
<td>2.1 Assignment problem: Greedy heuristic, Complete Enumeration, Branch-And-Bound</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.2 Knapsack problem: Greedy heuristic, Complete Enumeration, Branch-And-Bound</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.3 Algorithmic complexity and NP-hard problems: Combinatorial explosion, NP-hard problems, NP-Completeness, Optimization vs. Decision problems, P, NP and reducibility, Approximation algorithms, Heuristics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.4 Dynamic Programming and Backtracking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Bin Packing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Scheduling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Graph algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Linear Programming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Heuristics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algorithmic Complexity and NP-hard problems
**Time complexity of algorithms**

**Sunway TaihuLight: 93 Quadrillion \(10^{15}\) Operations per second (2016)**
The fastest supercomputer in the world Sunway TaihuLight performs 93 quadrillion operations per second. This corresponds to the processing power of more than 2,325,000 commercial PCs.

Even if we had a new supercomputer which would have 1000-fold computing power of Sunway TaihuLight, then we would still need the following time to compute \(2^n\) operations.

<table>
<thead>
<tr>
<th>(n)</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2^n)</td>
<td>360 Femtosec.</td>
<td>12 Microsec.</td>
<td>406 Seconds</td>
<td>432 Years</td>
</tr>
</tbody>
</table>

\[
x \times 1000
\]

Source: National Supercomputer centre in Wuxi (2016)
## Combinatorial Explosion

<table>
<thead>
<tr>
<th>$n^2$</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$6.7 \cdot 10^{-18}$ Seconds</td>
<td>$2.7 \cdot 10^{-17}$ Seconds</td>
<td>$6 \cdot 10^{-17}$ Seconds</td>
<td>$1 \cdot 10^{-16}$ Seconds</td>
</tr>
<tr>
<td>$n^{10}$</td>
<td>1 Microsec.</td>
<td>1 Millisec.</td>
<td>61 Millisec.</td>
<td>1 Second</td>
</tr>
<tr>
<td>$2^n$</td>
<td>360 Femtosec.</td>
<td>12 Microsec.</td>
<td>406 Seconds</td>
<td>432 Years</td>
</tr>
<tr>
<td>$n!$</td>
<td>46 Hours</td>
<td>$10 \cdot 10^{36}$ Years</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
NP-hard problems

**Problem**
Can you find an efficient algorithm?

- Yes: Continue with the next problem.
- No: Try again harder!

**Still no?**
How do you convey the bad information to your boss?
Approach 1: Take the loser’s way out

„I can‘t find an efficient algorithm.
I guess, I‘m just too dumb.“

Drawback: Could seriously damage your position within the company ...
Approach 2: Prove that the problem is inherently intractable

„I can‘t find an efficient algorithm, because no such algorithm is possible!“

Drawback: Proving inherent intractability can be as hard as finding efficient algorithms. Even the best theoreticians have failed!
Approach 3: Prove that the problem is NP-complete

„I can‘t find an efficient algorithm, but neither can all these famous people.“

Advantage: This would inform your boss that it is no good to fire you and hire another expert on algorithms.
NP-Completeness

**NP-complete problems:**
Problems that are “just as hard” as a large number of other problems that are widely recognized as being difficult by algorithmic experts.

**Properties:**
- exponential running time (as what we know today)
- polynomial reducable

1 Million US$
for the solution of one of the Millenium problems.

Clay Mathematics Institute
of Cambridge,
Massachusetts (CMI)
## Optimization vs. Decision problems

<table>
<thead>
<tr>
<th>Optimization problem</th>
<th>Decision problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find a subset of all objects such that under the condition that the knapsack capacity is not exceeded, this subset has the maximum possible value.</td>
<td>The theory of NP-completeness applies only to decision problems, where the solution is either a “Yes” or a “No”. We associate with the optimization problem a decision problem that includes a numerical bound $B$ as an additional parameter and that asks whether there exists a solution having a value greater or equal to $B$.</td>
</tr>
</tbody>
</table>

**Answer:** optimal solution  
**NP-hard**  

**Answer:** yes or no  
**NP-complete**
NP-Completeness

NP-complete problems:
Problems that are “just as hard” as a large number of other problems that are widely recognized as being difficult by algorithmic experts.

Properties:
- exponential running time (as what we know today)
- polynomial reducable

1 Million US$ for the solution of one of the Millenium problems.
Clay Mathematics Institute of Cambridge, Massachusetts (CMI)
## Optimization vs. Decision problems

<table>
<thead>
<tr>
<th><strong>Optimization problem</strong></th>
<th><strong>Decision problem</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Find a subset of all objects such that under the condition that the knapsack capacity is not exceeded, this subset has the maximum possible value.</td>
<td>The theory of NP-completeness applies only to decision problems, where the solution is either a “Yes” or a “No”. We associate with the optimization problem a decision problem that includes a numerical bound $B$ as an additional parameter and that asks whether there exists a solution having a value greater or equal to $B$.</td>
</tr>
</tbody>
</table>

**Answer:** optimal solution

**Answer:** yes or no
Complexity Classes

- NP
- NP-Complete
- NP-Hard

P ≠ NP

P = NP = NP-Complete

P = NP

UCSD CSE
Computer Science and Engineering
Classes P and NP

**P = Polynomial:**
The class P is the class of all decision problems that, under reasonable encoding schemes, can be solved by polynomial-time **deterministic** algorithms.

**NP = Non-deterministic Polynomial:**
The class NP is the class of all decision problems that, under reasonable encoding schemes, can be solved by polynomial-time **non-deterministic** algorithms.

1. Guessing stage:
   – Given a problem instance $I$, some structure $S$ is “guessed”.
   – The algorithm can pursue an unbounded number of independent computational sequences in parallel.
2. Checking stage:
   – The correctness of $S$ is verified in a normal deterministic manner
**Classes NP-complete and NP-hard**

**NP–Complete**
The class NP-complete is the class of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time.

Intuitively this means that we can solve y quickly if we know how to solve x quickly. Precisely, y is reducible to x, if there is a polynomial time algorithm f to transform instances y of Y to instances x = f(y) of X in polynomial time, with the property that the answer to y is yes, if and only if

**NP–hard**
Intuitively, these are the problems that are *at least as hard as the NP-complete problems*. Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems.

The precise definition here is that a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time.

The *halting problem* is an NP-hard problem. This is the problem that given a program P and input I, will it halt? This is a decision problem but it is not in NP. It is clear that any NP-complete problem can be reduced to this one. As another example, any NP-complete problem is NP-hard.
Proving NP-completeness

Proving NP-completeness for a decision problem $\Pi$:

1. Show that $\Pi$ is in NP

2. Select a known NP-complete problem $\Pi'$

3. Construct a transformation $\propto$ from $\Pi'$ to $\Pi$

4. Prove that $\propto$ is a (polynomial) transformation
Let’s say we want to show that a (new) problem $\Pi$ that we can’t solve is NP-complete.

We do so by showing that a known NP-complete problem $\Pi'$ is reducible to problem $\Pi$.

A problem $\Pi'$ is reducible to problem $\Pi$ if for any instance of $\Pi'$ an instance of $\Pi$ can be constructed in polynomial time such that solving the instance of $\Pi$ will solve the instance of $\Pi'$ as well.

When $\Pi'$ is reducible to $\Pi$, we write $\Pi' \propto \Pi$.

A decision problem $\Pi$ is said to be NP-complete if $\Pi \in \text{NP}$ and, for all other decision problems $\Pi' \in \text{NP}$, $\Pi'$ polynomially reduces to $\Pi$. 
Reducibility

Let’s say we want to show that a (new) problem $\Pi$ that we can’t solve is NP-complete.

We do so by showing that a known NP-complete problem $\Pi'$ is reducible to problem $\Pi$.

A problem $\Pi'$ is reducible to problem $\Pi$ if for any instance of $\Pi'$ an instance of $\Pi$ can be constructed in polynomial time such that solving the instance of $\Pi$ will solve the instance of $\Pi'$ as well.

When $\Pi'$ is reducible to $\Pi$, we write $\Pi' \preceq \Pi$.

A decision problem $\Pi$ is said to be NP-complete if $\Pi \in NP$ and, for all other decision problems $\Pi' \in NP$, $\Pi'$ polynomially reduces to $\Pi$. 
The question of whether or not NP-complete problems are intractable is considered to be one of the foremost open questions of contemporary computer science.
What I believe II (ft. Sarah Constantin and Stacey Jeffery)
August 13th, 2017

Unrelated Update: To everyone who keeps asking me about the “new” $P \neq NP$ proof: I’d again bet $200,000 that the paper won’t stand, except that the last time I tried that, it didn’t achieve its purpose, which was to get people to stop asking me about it. So: please stop asking, and if the thing hasn’t been refuted by the end of the week, you can come back and tell me I was a closed-minded fool.
What to do with NP-complete problems?

Qualified Heuristics = Approximation Algorithms

\[ A(I) \text{ approximate solution for a problem instance } I \]
\[ OPT(I) \text{ optimal solution for a problem instance } I \text{ of (here as an example) a MINIMIZATION problem} \]

Absolute performance: \[ A(I) \leq OPT(I) + c \]

Relative performance: \[ A(I) \leq \alpha \cdot OPT(I) \]

Combined: \[ A(I) \leq \alpha \cdot OPT(I) + c \]

Goal is to minimize \( \alpha \) and \( c \) such that the approximate solution for any problem instance \( I \) is as close as possible to the optimal solution \( \rightarrow \) Worst-case analysis
Examples for approximation results


\[ C_{\text{max}}^{LS} \leq \frac{4}{3} C^{*}_{\text{max}} + 1 \]

and the bound is tight.

B ≥ 3 resources:

List Scheduling vs. optimal non-preemptive scheduling is bounded from above by

\[ C_{\text{max}}^{LS} \leq \left( 2 - \frac{1}{B - 1} \right) C^{*}_{\text{max}} + \frac{B}{B - 1} \]

and the bound is tight.


\[ C_{\text{max}}^{LPT} \leq C^{*}_{\text{max}} \left( 2 - \frac{2}{B} \right) + 1 \]
What to do with NP-complete problems?

Unqualified Heuristics = Heuristics

Simulated Annealing – Probabilistically decide whether to move to a new solution or stay with the current one
Tabu Search – Make it hard to return to a previous solutions
Ant Algorithms – Iteratively move towards a more complete intermediate solution
Cuttlefish Algorithm – Mimics the mechanism of color changing behaviour of the cuttlefish to find global optimal solution
Compact Genetic Algorithm – Uses a vector to hold the probability of including each component in the next solution

Variable Neighborhood Search – Look in the Neighborhood of a solution and get out from it from time to time

<table>
<thead>
<tr>
<th></th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithms</td>
</tr>
<tr>
<td>2</td>
<td>Assignment and Knapsack problem</td>
</tr>
<tr>
<td>3</td>
<td>Bin Packing</td>
</tr>
<tr>
<td>3.1</td>
<td>Complete Enumeration, First Fit, First Fit Decreasing, Next Fit</td>
</tr>
<tr>
<td>3.2</td>
<td>Worst-case analysis of the approximation algorithms</td>
</tr>
<tr>
<td>4</td>
<td>Scheduling</td>
</tr>
<tr>
<td>5</td>
<td>Graph algorithms</td>
</tr>
<tr>
<td>6</td>
<td>Linear Programming</td>
</tr>
<tr>
<td>7</td>
<td>Heuristics</td>
</tr>
<tr>
<td></td>
<td>Content</td>
</tr>
<tr>
<td>---</td>
<td>---------</td>
</tr>
<tr>
<td>1</td>
<td>Algorithms</td>
</tr>
<tr>
<td>2</td>
<td>Assignment and Knapsack problem</td>
</tr>
<tr>
<td>3</td>
<td>Bin Packing</td>
</tr>
<tr>
<td></td>
<td>3.1 Complete Enumeration, First Fit, First Fit Decreasing, Next Fit</td>
</tr>
<tr>
<td></td>
<td>3.2 Worst-case analysis of the approximation algorithms</td>
</tr>
<tr>
<td>4</td>
<td>Scheduling</td>
</tr>
<tr>
<td>5</td>
<td>Graph algorithms</td>
</tr>
<tr>
<td>6</td>
<td>Linear Programming</td>
</tr>
<tr>
<td>7</td>
<td>Heuristics</td>
</tr>
</tbody>
</table>
Bin Packing

Minimizing the number of paper rolls when cutting single lanes out of it.

Minimizing the number of containers needed to transport a good.
Bin Packing

Here: 3 bins. Is there a better possibility? Better = „less bins“?
Bin Packing

Here: 3 bins. Is there a better possibility? Better = „less bins“?
Bin Packing

**Lower Bound:**
We need at least
\[
\left\lceil \frac{6+3+4+3+4}{10} \right\rceil = 2
\]
bins.
First Fit

• Nine groups of UCSD-people want to go to a Padres game.
• The groups consist of 4, 1, 2, 5, 3, 2, 3, 6, 3 persons.
• The buses have a maximum passenger capacity of 6 places.
• The groups want to stick together and are therefore not dividable.

• First Fit: As long as there are enough places in a bus, fill the bus up with the next group. Otherwise use a new bus. Start always with the first bus and proceed then to the next bus and so on.
First Fit

A B C D E F G

4 1 2 5 3 2 3 6

CSE 101
SS2, 2018
Prof. Dr. Oliver Braun
First Fit

A: 4

B: 2

C: 5

D: 3

E: 2

F: 3

G: 6

1 Algorithms | 2 Assignment/Knapsack | 3 Bin Packing | 4 Scheduling | 5 Graph algorithms | 6 Linear Programming | 7 Heuristics
First Fit

1. A
2. B
3. C
4. D
5. E
6. F
7. G

- A: 1
- B: 2
- C: 5
- D: 3
- E: 2
- F: 3
- G: 6
First Fit

AB C DE F G

A: 4
   - 2
   - 1
B: 2
C: D: E: F: G: 6
5 3 2 3 3
First Fit

A  4
   1

B  5
   2

C  5

D

E

F

G  6

Heuristics
First Fit

A: 4
B: 3
C: 5
D: 
E: 
F: 
G: 6

AB C DE F G
First Fit

A: 4
B: 2
C: 2
D: 2
E: 0
F: 0
G: 6

AB C DE F G

1 2 3 2 3 2

1 2 3 2 3

AB 2 C 3 D 6

1 Algorithms | 2 Assignment/Knapsack | 3 Bin Packing | 4 Scheduling | 5 Graph algorithms | 6 Linear Programming | 7 Heuristics
### First Fit

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Bin Packing

- **A**'s size: 3
- **B**'s size: 3
- **C**'s size: 3
- **D**'s size: 3
- **G**'s size: 6
First Fit

A   B   C   D   E   F   G
6   6   6   6
1   3   5   3   6
4   2   2   6

AB C DE FG
1 6 3 4 2 3 6
First Fit
First Fit Decreasing

- Nine groups of UCSD-people want to go to a Padres game.
- The groups consist of 4, 1, 2, 5, 3, 2, 3, 6, 3 persons.
- The buses have a maximum passenger capacity of 6 places.
- The groups want to stick together and are therefore not dividable.

- First Fit Decreasing: First sort the groups in non-increasing order. Then use First Fit.
First Fit Decreasing

A   B   C   D   E   F   G

AB C DE F G
First Fit Decreasing

AB C DE F G

A 5 B 4 C 3 D 3 E 3 F 2 G 1

1 Algorithms | 2 Assignment/Knapsack | 3 Bin Packing | 4 Scheduling | 5 Graph algorithms | 6 Linear Programming | 7 Heuristics
First Fit Decreasing

A: 6
B: 
C: 
D: 
E: 
F: 
G: 

5
4
3
3
3
2
2
1

AB C DE F G
First Fit Decreasing

A

B

C

D

E

F

G

4

4

6

5

4

3

3

3

2

2

1
First Fit Decreasing

A  B  C  D  E  F  G

3  3  3  3  5  4  6

AB C DE F G

First Fit Decreasing
First Fit Decreasing

A 3
B 5
C 4
D 2
E 3
F 2
G 1
First Fit Decreasing

A  B  C  D  E  F  G
6  5  4  3  3  -  -

2  2  2  3  -  -  -

1  1  -  -  -  -  -
First Fit Decreasing

A: 6
B: 5
C: 4
D: 3
E: 3
F: 
G: 

2 2 2 3 2
First Fit Decreasing

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AB C DE F G

1

6

5

4

3

2

3

2

1


First Fit Decreasing

A: 6
B: 5
C: 4
D: 3
E: 2
F: 
G: 

AB C DE F G
Next Fit

- Nine groups of UCSD-people want to visit a Padres game.
- The groups consist of 4, 1, 2, 5, 3, 2, 3, 6, 3 persons.
- The buses have a maximum passenger capacity of 6 places.
- The groups want to stick together and are therefore not dividable.

- Next Fit: Same as First Fit, but a bus leaves as soon as a group does not fit in it anymore.
Next Fit

1. Algorithms
2. Assignment/Knapsack
3. Bin Packing
4. Scheduling
5. Graph algorithms
6. Linear Programming
7. Heuristics
## Content

<table>
<thead>
<tr>
<th></th>
<th>Algorithms</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Assignment and Knapsack problem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Bin Packing</td>
<td>3.1</td>
<td>Complete Enumeration, First Fit, First Fit Decreasing, Next Fit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.2</td>
<td>Worst-case analysis of the approximation algorithms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Scheduling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Graph algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Linear Programming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Heuristics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bin Packing

a) How would you define the problem?
Remember: A problem is defined by
• input domain / problem instance
• output specification
  binsize (10), A (6), B (3), C (4), D (3), E (4)
  permutation of the jobs such that planning
  them in this order will lead to the minimum
  number of bins, here: A, C, B, D, E

b) How many possible solutions are there? (now you need Counting ...)

c) How would you develop a Complete Enumeration algorithm?
Next Fit – Worst-Case-Analysis

Next Fit requires up to about 100% bins more than the optimal solution does.

Find a problem instance where Next Fit behaves that bad in comparison to the optimal solution.
First Fit – Worst-Case-Analysis

First Fit requires up to about 70% bins more than the optimal solution does.

Find a problem instance where First Fit behaves that bad in comparison to the optimal solution.
First Fit Decreasing requires up to about 22% bins more than the optimal solution does.

Find a problem instance where First Fit Decreasing behaves that bad in comparison to the optimal solution.
Bin Packing

Develop your own Bin Packing algorithm.

Work out some problem instances that show how your algorithm works.

Analyze your algorithm’s worst-case behavior.