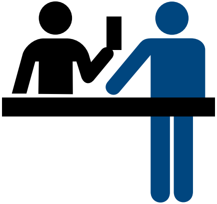
Algorithms for Programming Contests - Week 07

Prof. Dr. Javier Esparza,
Vincent Fischer,
Jakob Schulz,
conpra@model.cit.tum.de

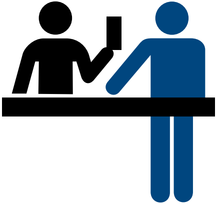
December 9, 2025

Change making



How to give change back
with the minimum number of coins?

Change making



How to give change back
with the minimum number of coins?

Be **greedy**: go for the largest coins first!

Change making

2$

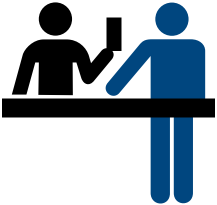
1$

25¢

10¢

5¢

1¢



$$\begin{aligned}
5.82\$ - 2 \times 2\$ &= 1.82\$ \\
1.82\$ - 1 \times 1\$ &= 0.82\$ \\
0.82\$ - 3 \times 25¢ &= 0.07\$ \\
0.07\$ - 1 \times 5¢ &= 0.02\$ \\
0.02\$ - 2 \times 1¢ &= 0.00\$
\end{aligned}$$

Change making


2$

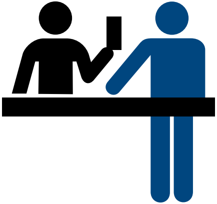

1$


25¢


10¢


5¢


1¢



Approach still works if we introduce 20¢ ?

Change making

2$

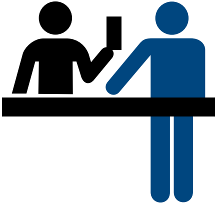
1$

25¢

10¢

5¢

1¢



No, for 40¢ it returns 25¢ + 10¢ + 5¢
instead of 2 × 20¢

Change making: greedy approach

```
procedure GREEDY-CHANGE-MAKING($c_1, \ldots, c_n, m$)
    sort $c_1, \ldots, c_n$ in descending order
    $S \leftarrow [\,]$
    $i \leftarrow 1$, $rem \leftarrow m$
    while $i \leq n$ and $rem > 0$ do
        if $c_i \leq rem$ then
            $rem \leftarrow rem - c_i$
            add $c_i$ to $S$
        else
            $i \leftarrow i + 1$
    if $rem = 0$ then return $S$
    else return impossible
```

Change making: greedy approach

```
procedure GREEDY-CHANGE-MAKING($c_1, \ldots, c_n, m$)
    sort $c_1, \ldots, c_n$ in descending order
    $S \leftarrow [\,]$
    $i \leftarrow 1$, $rem \leftarrow m$
    while $i \leq n$ and $rem > 0$ do
        if $c_i \leq rem$ then
            $rem \leftarrow rem - c_i$
            add $c_i$ to $S$
        else
            $i \leftarrow i + 1$
    if $rem = 0$ then return $S$
    else return impossible
```

GREEDY-CHANGE-MAKING is optimal for $ (CAD) and € (EUR)

Change making: greedy approach

```
procedure GREEDY-CHANGE-MAKING($c_1, \ldots, c_n, m$)
    sort $c_1, \ldots, c_n$ in descending order
    $S \leftarrow [\,]$
    $i \leftarrow 1$, $rem \leftarrow m$
    while $i \leq n$ and $rem > 0$ do
        if $c_i \leq rem$ then
            $rem \leftarrow rem - c_i$
            add $c_i$ to $S$
        else
            $i \leftarrow i + 1$
    if $rem = 0$ then return $S$
    else return impossible
```

The solution of GREEDY-CHANGE-MAKING can be arbitrarily bad

Let $n > 2$. On input $(c_1 = n + 2, c_2 = n + 1, c_3 = n, c_4 = 1, m = 2n + 1)$, GREEDY-CHANGE-MAKING returns n coins instead of 2 coins.

Change making: greedy approach

```
procedure GREEDY-CHANGE-MAKING($c_1, \ldots, c_n, m$)
    sort $c_1, \ldots, c_n$ in descending order
    $S \leftarrow [\,]$
    $i \leftarrow 1$, $rem \leftarrow m$
    while $i \leq n$ and $rem > 0$ do
        if $c_i \leq rem$ then
            $rem \leftarrow rem - c_i$
            add $c_i$ to $S$
        else
            $i \leftarrow i + 1$
    if $rem = 0$ then return $S$
    else return impossible
```

The solution of GREEDY-CHANGE-MAKING can be arbitrarily bad

On input ($c_1 = 50, c_2 = 20, m = 60$), GREEDY-CHANGE-MAKING returns "impossible" instead of 3 coins.

Change making: greedy approach

```
procedure GREEDY-CHANGE-MAKING($c_1, \ldots, c_n, m$)
    sort $c_1, \ldots, c_n$ in descending order
    $S \leftarrow [\,]$
    $i \leftarrow 1$, $rem \leftarrow m$
    while $i \leq n$ and $rem > 0$ do
        if $c_i \leq rem$ then
            $rem \leftarrow rem - c_i$
            add $c_i$ to $S$
        else
            $i \leftarrow i + 1$
    if $rem = 0$ then return $S$
    else return impossible
```

Finding an optimal solution is NP-hard for arbitrary currencies

Change making: Remarks

- In practice, instead of returning a list of the used coins, one returns a vector $v = (v_1, \dots, v_n) \in \mathbb{N}^n$ s.t. $v \cdot c = m$
    - Algorithm then faster by not adding one coin each iteration, but $\lfloor rem/c_i \rfloor$ many

Change making: Remarks

- In practice, instead of returning a list of the used coins, one returns a vector $v = (v_1, \ldots, v_n) \in \mathbb{N}^n$ s.t. $v \cdot c = m$
    - Algorithm then faster by not adding one coin each iteration, but $\lfloor rem/c_i \rfloor$ many
- Still active research, e.g.:
    - Call a coin system $c = (c_1, \ldots, c_n)$ *canonical* if greedy is optimal for its change-making problem
    - Suzuki, Miyashiro (2023): Characterization of canonical coin systems with 6 coins
    - Van Cott, Zhang (2025): Canonical coin systems $(c_1, \ldots, c_n)$ with n arbitrary large s.t. $(c_1, \ldots, c_i)$ is not canonical for $3 < i < n$

Change making: Remarks

- In practice, instead of returning a list of the used coins, one returns a vector $v = (v_1, \ldots, v_n) \in \mathbb{N}^n$ s.t. $v \cdot c = m$
    - Algorithm then faster by not adding one coin each iteration, but $\lfloor rem/c_i \rfloor$ many
- Still active research, e.g.:
    - Call a coin system $c = (c_1, \ldots, c_n)$ *canonical* if greedy is optimal for its change-making problem
    - Suzuki, Miyashiro (2023): Characterization of canonical coin systems with 6 coins
    - Van Cott, Zhang (2025): Canonical coin systems $(c_1, \ldots, c_n)$ with n arbitrary large s.t. $(c_1, \ldots, c_i)$ is not canonical for $3 < i < n$
- Related:
    - There exists an $m_0 \in \mathbb{N}$ s.t. all $m \geq m_0$ can be expressed as such $v \cdot c$ iff $m \in \gcd(c_1, \ldots, c_n)\mathbb{Z}$

Change making: Remarks

- In practice, instead of returning a list of the used coins, one returns a vector $v = (v_1, \dots, v_n) \in \mathbb{N}^n$ s.t. $v \cdot c = m$
    - Algorithm then faster by not adding one coin each iteration, but $\lfloor rem/c_i \rfloor$ many
- Still active research, e.g.:
    - Call a coin system $c = (c_1, \dots, c_n)$ *canonical* if greedy is optimal for its change-making problem
    - Suzuki, Miyashiro (2023): Characterization of canonical coin systems with 6 coins
    - Van Cott, Zhang (2025): Canonical coin systems $(c_1, \dots, c_n)$ with n arbitrary large s.t. $(c_1, \dots, c_i)$ is not canonical for $3 < i < n$
- Related:
    - There exists an $m_0 \in \mathbb{N}$ s.t. all $m \geq m_0$ can be expressed as such $v \cdot c$ iff $m \in \gcd(c_1, \dots, c_n)\mathbb{Z}$
    - Finding the smallest such m_0 is called the Frobenius coin problem

Change making: Canonical coin systems

Characterization of canonical coin systems up to $n = 6$ ($c_1 = 1$ usually assumed so that each m is representable):

Here, $g(c, m)$ is the number of coins used by the greedy algorithm.

Change making: Canonical coin systems

Characterization of canonical coin systems up to $n = 6$ ($c_1 = 1$ usually assumed so that each m is representable):

- $n \in \{1, 2\}$: always canonical.

Here, $g(c, m)$ is the number of coins used by the greedy algorithm.

Change making: Canonical coin systems

Characterization of canonical coin systems up to $n = 6$ ($c_1 = 1$ usually assumed so that each m is representable):

- $n \in \{1, 2\}$: always canonical.
- $n = 3$: $(1, c_2, c_3)$ canonical iff $r = 0$ or $c_2 - q \leq 0$, where $c_3 = q \cdot c_2 + r$ (with $r \in \{0, \ldots, c_2 - 1\}$)

Here, $g(c, m)$ is the number of coins used by the greedy algorithm.

Change making: Canonical coin systems

Characterization of canonical coin systems up to $n = 6$ ($c_1 = 1$ usually assumed so that each m is representable):

- $n \in \{1, 2\}$: always canonical.
- $n = 3$: $(1, c_2, c_3)$ canonical iff $r = 0$ or $c_2 - q \leq 0$, where $c_3 = q \cdot c_2 + r$ (with $r \in \{0, \ldots, c_2 - 1\}$)
- $n = 4$: $(1, c_2, c_3, c_4)$ canonical iff $(1, c_2, c_3)$ canonical and $g(c, m \cdot c_3) \leq m$, where $m = \lceil c_4 / c_3 \rceil$

Here, $g(c, m)$ is the number of coins used by the greedy algorithm.

Change making: Canonical coin systems

Characterization of canonical coin systems up to $n = 6$ ($c_1 = 1$ usually assumed so that each m is representable):

- $n \in \{1, 2\}$: always canonical.
- $n = 3$: $(1, c_2, c_3)$ canonical iff $r = 0$ or $c_2 - q \leq 0$, where $c_3 = q \cdot c_2 + r$ (with $r \in \{0, \dots, c_2 - 1\}$)
- $n = 4$: $(1, c_2, c_3, c_4)$ canonical iff $(1, c_2, c_3)$ canonical and $g(c, m \cdot c_3) \leq m$, where $m = \lceil c_4/c_3 \rceil$
- $n = 5$: $(1, c_2, c_3, c_4, c_5)$ canonical iff one of the following holds:
    - $(1, c_2, c_3, c_4)$ canonical and $g(c, m \cdot c_4) \leq m$, where $m = \lceil c_5/c_4 \rceil$

Here, $g(c, m)$ is the number of coins used by the greedy algorithm.

Change making: Canonical coin systems

Characterization of canonical coin systems up to $n = 6$ ($c_1 = 1$ usually assumed so that each m is representable):

- $n \in \{1, 2\}$: always canonical.
- $n = 3$: $(1, c_2, c_3)$ canonical iff $r = 0$ or $c_2 - q \leq 0$, where $c_3 = q \cdot c_2 + r$ (with $r \in \{0, \ldots, c_2 - 1\}$)
- $n = 4$: $(1, c_2, c_3, c_4)$ canonical iff $(1, c_2, c_3)$ canonical and $g(c, m \cdot c_3) \leq m$, where $m = \lceil c_4 / c_3 \rceil$
- $n = 5$: $(1, c_2, c_3, c_4, c_5)$ canonical iff one of the following holds:
  - $(1, c_2, c_3, c_4)$ canonical and $g(c, m \cdot c_4) \leq m$, where $m = \lceil c_5 / c_4 \rceil$
  - $c_2 = 2, c_4 = c_3 + 1, c_5 = 2c_3$ and $c_3 \geq 4$

Here, $g(c, m)$ is the number of coins used by the greedy algorithm.

Change making: Canonical coin systems

Characterization of canonical coin systems up to $n = 6$ ($c_1 = 1$ usually assumed so that each m is representable):

- $n \in \{1, 2\}$: always canonical.
- $n = 3$: $(1, c_2, c_3)$ canonical iff $r = 0$ or $c_2 - q \leq 0$, where $c_3 = q \cdot c_2 + r$ (with $r \in \{0, \dots, c_2 - 1\}$)
- $n = 4$: $(1, c_2, c_3, c_4)$ canonical iff $(1, c_2, c_3)$ canonical and $g(c, m \cdot c_3) \leq m$, where $m = \lceil c_4 / c_3 \rceil$
- $n = 5$: $(1, c_2, c_3, c_4, c_5)$ canonical iff one of the following holds:
  - $(1, c_2, c_3, c_4)$ canonical and $g(c, m \cdot c_4) \leq m$, where $m = \lceil c_5 / c_4 \rceil$
  - $c_2 = 2, c_4 = c_3 + 1, c_5 = 2c_3$ and $c_3 \geq 4$
- $n = 6$: $c = (1, c_2, c_3, c_4, c_5, c_6)$ canonical iff one of the following two holds:
  - $(1, c_2, c_3, c_4, c_5)$ canonical and $g(c, m \cdot c_5) \leq m$, where $m = \lceil c_6 / c_5 \rceil$
  - $(1, c_2, c_3, c_4, c_5)$ is not canonical and, for $l = \lceil c_5 / c_3 \rceil$, one of the following holds:
    - $c = (1, 2, 3, c_4, c_4 + 1, 2c_4)$ and $c_4 \geq 5$
    - $c = (1, c_2, 2c_2 - 1, c_4, c_2 + c_4 - 1, 2c_4 - 1)$, $c_4 \geq 3c_2 - 1$ and $g(c, l \cdot c_3) \leq l$
    - $c = (1, c_2, 2c_2, c_4, c_2 + c_4, 2c_4)$, $c_4 \geq 3c_2 - 1$, $c_4 \neq 3c_2$ and $g(c, l \cdot c_3) \leq l$

Here, $g(c, m)$ is the number of coins used by the greedy algorithm.

Greedy algorithms

- Paradigm for solving optimization problems
- Make local choices, never global
- Do not reconsider choices

– Often non optimal

\+ Can be good heuristic

\+ Can be good approximation

\+ Simple

\+ Fast

Greedy algorithms: general template

```
procedure GREEDY(candidates)
    $S \leftarrow [\,]$
    while $|candidates| > 0$ and $\neg$solution($S$) do
        $c \leftarrow$ select($candidates$)
        remove $c$ from $candidates$
        if feasible($S, c$) then
            add $c$ to $S$
    if solution($S$) then
        return $S$
    else
        return impossible
```

Greedy algorithms: general template

```
procedure GREEDY($candidates$)
    $S \leftarrow [\,]$
    while $|candidates| > 0$ and $\neg$solution($S$) do
        $c \leftarrow$ select($candidates$)
        remove $c$ from $candidates$
        if feasible($S, c$) then
            add $c$ to $S$
    if solution($S$) then
        return $S$
    else
        return impossible
```

Kruskal's algorithm

candidates:	edges
select:	smallest edge
feasible:	connects two connected components?
solution:	contains $|V| - 1$ edges?

Greedy algorithms: general template

```
procedure GREEDY($candidates$)
    $S \leftarrow [\,]$
    while $|candidates| > 0$ and $\neg$ solution($S$) do
        $c \leftarrow$ select($candidates$)
        remove $c$ from $candidates$
        if feasible($S, c$) then
            add $c$ to $S$
    if solution($S$) then
        return $S$
    else
        return impossible
```

Prim's algorithm

candidates: edges
select: smallest edge with an endpoint in explored nodes
feasible: has no cycle?
solution: covers every node?

Greedy algorithms: general template

```
procedure GREEDY($candidates$)
    $S \leftarrow [\,]$
    while $|candidates| > 0$ and $\neg$solution($S$) do
        $c \leftarrow$ select($candidates$)
        remove $c$ from $candidates$
        if feasible($S, c$) then
            add $c$ to $S$
    if solution($S$) then
        return $S$
    else
        return impossible
```

Change making

candidates: coins

select: largest coin smaller or equal to remaining amount

feasible: —

solution: sums up to the amount?

Greedy algorithms: Remarks

There exists a characterization of when Greedy is optimal:

Definition

Let E be a finite set and $\mathcal{F} \subseteq 2^E$. $(E, \mathcal{F})$ is called *independence system* iff

1. $\emptyset \in \mathcal{F}$
2. for all $Y \in \mathcal{F}$: if $X \subseteq Y$, then $X \in \mathcal{F}$

An independence system is called *matroid* iff additionally

1. for all $X, Y \in \mathcal{F}$ with $|X| > |Y|$, there is an $x \in X \setminus Y$ s.t. $Y \cup \{x\} \in \mathcal{F}$

Theorem (Edmonds-Rado Theorem)

Let $(E, \mathcal{F})$ be an independence system, and $c : E \to \mathbb{R}_+$. Then, starting with $\emptyset$ and greedily adding $e \in E$ based on $c(e)$, yields an optimal solution for all $c : E \to \mathbb{R}_+$ iff $(E, \mathcal{F})$ is a matroid.

Approximation algorithms

Definition

Assume we have an optimization problem P with nonnegative values. A *k-factor approximation algorithm* is a polynomial time algorithm A for P s.t. for all inputs I,

$$\frac{1}{k} \cdot OPT(I) \leq A(I) \leq k \cdot OPT(I)$$

where $OPT(I)$ is the optimal value for instance I.

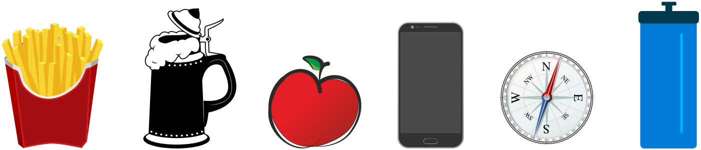
- Way to circumvent NP-hardness
- Sometimes, greedy algorithms are efficient approximation algorithms

0/1 Knapsack problem

Given:
- backpack of capacity $W \in \mathbb{N}_{>0}$
- n objects of value $v_1, \ldots, v_n \in \mathbb{N}$ and weight $w_1, \ldots, w_n \in [1, W]$

Compute: subset of objects of weight at most W maximizing value (among all such subsets)

0/1 Knapsack problem

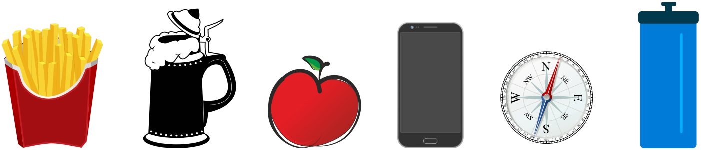


Value:	10	15	5	50	7	20
Weight:	150g	540g	100g	200g	70g	700g

Capacity: 900g

What to bring in the backpack?

0/1 Knapsack problem

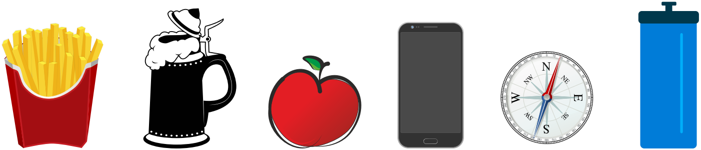


Value:	10	15	5	50	7	20
Weight:	150g	540g	100g	200g	70g	700g

Value: 32 (870g)

Capacity: 900g

0/1 Knapsack problem

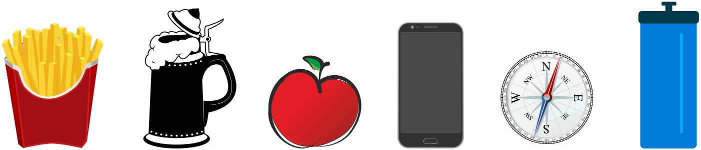


Value:	10	15	5	50	7	20
Weight:	150g	540g	100g	200g	70g	700g

Value: 70 (900g)

Capacity: 900g

0/1 Knapsack problem

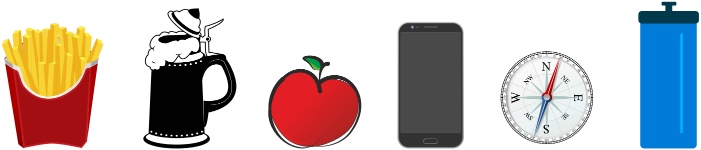


Value:	10	15	5	50	7	20
Weight:	150g	540g	100g	200g	70g	700g

Value: 75 (890g)

Capacity: 900g

0/1 Knapsack problem

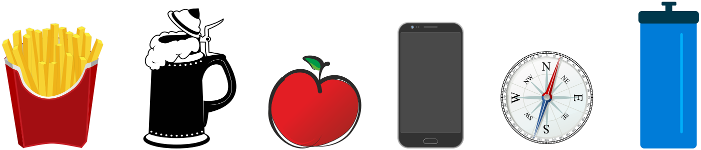


Value:	10	15	5	50	7	20
Weight:	150g	540g	100g	200g	70g	700g

Capacity: 900g

Greedy way to obtain solution?

0/1 Knapsack problem

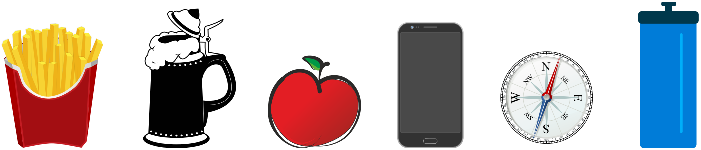


Value:	10	15	5	50	7	20
Weight:	150g	540g	100g	200g	70g	700g

Capacity: 900g

Sort in desc. order w.r.t. v_i / w_i...

0/1 Knapsack problem

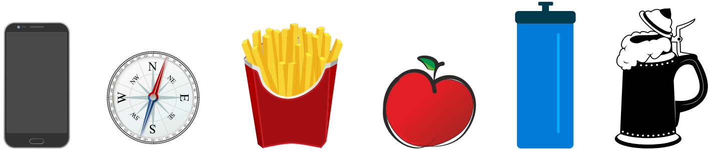


Value:	10	15	5	50	7	20
Weight:	150g	540g	100g	200g	70g	700g
Ratio:	1/15	1/36	1/20	1/4	1/10	1/35

Capacity: 900g

Sort in desc. order w.r.t. v_i / w_i...

0/1 Knapsack problem

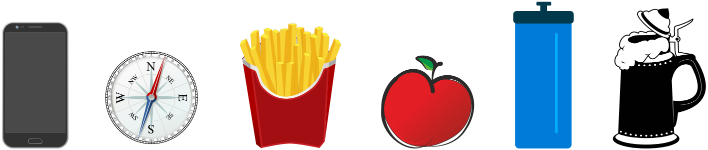


Value:	50	7	10	5	20	15
Weight:	200g	70g	150g	100g	700g	540g
Ratio:	1/4	1/10	1/15	1/20	1/35	1/36

Capacity: 900g

Sort in desc. order w.r.t. v_i/w_i...

0/1 Knapsack problem

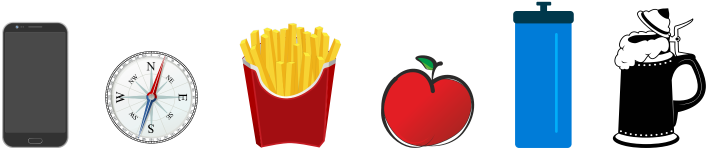


Value:	50	7	10	5	20	15
Weight:	200g	70g	150g	100g	700g	540g
Ratio:	1/4	1/10	1/15	1/20	1/35	1/36

Value: 72 (520g)

Capacity: 900g

0/1 Knapsack problem



Value:	50	7	10	5	20	15
Weight:	200g	70g	150g	100g	700g	540g
Ratio:	$1/4$	$1/10$	$1/15$	$1/20$	$1/35$	$1/36$

Capacity: 900g

Not optimal, but by how much?

0/1 Knapsack problem: greedy approach

```
procedure GREEDY-KNAPSACK-NAIVE($W, (v_1, w_1), \ldots, (v_n, w_n)$)
    sort $(v_1, w_1), \ldots, (v_n, w_n)$ in descending order w.r.t. $v_i / w_i$
    $value, weight \leftarrow 0$
    $i \leftarrow 1$
    while $weight + w_i \leq W$ and $i \leq n$ do
        $value \leftarrow value + v_i$
        $weight \leftarrow weight + w_i$
        $i \leftarrow i + 1$
    return $value$
```

0/1 Knapsack problem: greedy approach

```
procedure GREEDY-KNAPSACK-NAIVE($W, (v_1, w_1), \dots, (v_n, w_n)$)
    sort $(v_1, w_1), \dots, (v_n, w_n)$ in descending order w.r.t. $v_i/w_i$
    $value, weight \leftarrow 0$
    $i \leftarrow 1$
    while $weight + w_i \le W$ and $i \le n$ do
        $value \leftarrow value + v_i$
        $weight \leftarrow weight + w_i$
        $i \leftarrow i + 1$
    return $value$
```

The solution of GREEDY-KNAPSACK-NAIVE can be arbitrarily bad

0/1 Knapsack problem: greedy approach

```
procedure GREEDY-KNAPSACK-NAIVE($W, (v_1, w_1), \ldots, (v_n, w_n)$)
    sort $(v_1, w_1), \ldots, (v_n, w_n)$ in descending order w.r.t. $v_i/w_i$
    $value, weight \leftarrow 0$
    $i \leftarrow 1$
    while $weight + w_i \leq W$ and $i \leq n$ do
        $value \leftarrow value + v_i$
        $weight \leftarrow weight + w_i$
        $i \leftarrow i + 1$
    return $value$
```

The solution of GREEDY-KNAPSACK-NAIVE can be arbitrarily bad

Let $W > 2$. On input $(v_1 = 2, w_1 = 1), (v_2 = W, w_2 = W)$, GREEDY-KNAPSACK-NAIVE returns 2 while the optimal value is W ☐

0/1 Knapsack problem: greedy approach

```
procedure GREEDY-KNAPSACK-FRAC($W, (v_1, w_1), \ldots, (v_n, w_n)$)
    sort $(v_1, w_1), \ldots, (v_n, w_n)$ in descending order w.r.t. $v_i / w_i$
    $value, weight \leftarrow 0$
    $i \leftarrow 1$
    while $weight + w_i \leq W$ and $i \leq n$ do
        $value \leftarrow value + v_i$
        $weight \leftarrow weight + w_i$
        $i \leftarrow i + 1$
    return $value + \frac{(W - weight)}{w_i} \cdot v_i$
```

0/1 Knapsack problem: greedy approach

```
procedure GREEDY-KNAPSACK-FRAC($W, (v_1, w_1), \dots, (v_n, w_n)$)
    sort $(v_1, w_1), \dots, (v_n, w_n)$ in descending order w.r.t. $v_i/w_i$
    $value, weight \leftarrow 0$
    $i \leftarrow 1$
    while $weight + w_i \leq W$ and $i \leq n$ do
        $value \leftarrow value + v_i$
        $weight \leftarrow weight + w_i$
        $i \leftarrow i + 1$
    return $value + \frac{(W - weight)}{w_i} \cdot v_i$
```

GREEDY-KNAPSACK-FRAC is optimal if objects can be taken partially by a factor $0 \leq \alpha \leq 1$

0/1 Knapsack problem: greedy approach

```
procedure GREEDY-KNAPSACK-FRAC($W, (v_1, w_1), \ldots, (v_n, w_n)$)
    sort $(v_1, w_1), \ldots, (v_n, w_n)$ in descending order w.r.t. $v_i/w_i$
    $value, weight \leftarrow 0$
    $i \leftarrow 1$
    while $weight + w_i \leq W$ and $i \leq n$ do
        $value \leftarrow value + v_i$
        $weight \leftarrow weight + w_i$
        $i \leftarrow i + 1$
    return $value + \frac{(W - weight)}{w_i} \cdot v_i$
```

GREEDY-KNAPSACK-FRAC is optimal if objects can be taken partially by a factor $0 \leq \alpha \leq 1$

Relatively straightforward proof

0/1 Knapsack problem: greedy approach

```
procedure GREEDY-KNAPSACK($W, (v_1, w_1), \ldots, (v_n, w_n)$)
    sort $(v_1, w_1), \ldots, (v_n, w_n)$ in descending order w.r.t. $v_i/w_i$
    $value, weight \leftarrow 0$
    $i \leftarrow 1$
    while $weight + w_i \leq W$ and $i \leq n$ do
        $value \leftarrow value + v_i$
        $weight \leftarrow weight + w_i$
        $i \leftarrow i + 1$
    if $i \leq n$ then return $\max(value, v_i)$
    else return $value$
```

0/1 Knapsack problem: greedy approach

```
procedure GREEDY-KNAPSACK($W, (v_1, w_1), \ldots, (v_n, w_n)$)
    sort $(v_1, w_1), \ldots, (v_n, w_n)$ in descending order w.r.t. $v_i/w_i$
    $value, weight \leftarrow 0$
    $i \leftarrow 1$
    while $weight + w_i \leq W$ and $i \leq n$ do
        $value \leftarrow value + v_i$
        $weight \leftarrow weight + w_i$
        $i \leftarrow i + 1$
    if $i \leq n$ then return $\max(value, v_i)$
    else return $value$
```

The solution of GREEDY-KNAPSACK is at least $\frac{1}{2}$ of the optimal solution

0/1 Knapsack problem: greedy approach

```
procedure GREEDY-KNAPSACK($W, (v_1, w_1), \ldots, (v_n, w_n)$)
    sort $(v_1, w_1), \ldots, (v_n, w_n)$ in descending order w.r.t. $v_i/w_i$
    $value, weight \leftarrow 0$
    $i \leftarrow 1$
    while $weight + w_i \leq W$ and $i \leq n$ do
        $value \leftarrow value + v_i$
        $weight \leftarrow weight + w_i$
        $i \leftarrow i + 1$
    if $i \leq n$ then return $\max(value, v_i)$
    else return $value$
```

The solution of GREEDY-KNAPSACK is at least $\frac{1}{2}$ of the optimal solution

$$\underbrace{(v_1 + \ldots + v_{i-1})}_{value} + v_i \geq opt_{\text{frac}} \geq opt \implies \max(value, v_i) \geq opt/2 \quad \square$$

0/1 Knapsack problem: greedy approach

```
procedure GREEDY-KNAPSACK($W, (v_1, w_1), \ldots, (v_n, w_n)$)
    sort $(v_1, w_1), \ldots, (v_n, w_n)$ in descending order w.r.t. $v_i/w_i$
    $value, weight \leftarrow 0$
    $i \leftarrow 1$
    while $weight + w_i \leq W$ and $i \leq n$ do
        $value \leftarrow value + v_i$
        $weight \leftarrow weight + w_i$
        $i \leftarrow i + 1$
    if $i \leq n$ then return $\max(value, v_i)$
    else return $value$
```

Worst-case time complexity:

by sorting:	$O(n \cdot \log n)$
using recursion and linear time median:	$O(n)$

0/1 Knapsack problem: greedy approach

```
procedure GREEDY-KNAPSACK($W, (v_1, w_1), \ldots, (v_n, w_n)$)
    sort $(v_1, w_1), \ldots, (v_n, w_n)$ in descending order w.r.t. $v_i/w_i$
    $value, weight \leftarrow 0$
    $i \leftarrow 1$
    while $weight + w_i \leq W$ and $i \leq n$ do
        $value \leftarrow value + v_i$
        $weight \leftarrow weight + w_i$
        $i \leftarrow i + 1$
    if $i \leq n$ then return $\max(value, v_i)$
    else return $value$
```

In general, computing an optimal solution is NP-hard

0/1 Knapsack problem: greedy approach

```
procedure GREEDY-KNAPSACK($W, (v_1, w_1), \ldots, (v_n, w_n)$)
    sort $(v_1, w_1), \ldots, (v_n, w_n)$ in descending order w.r.t. $v_i/w_i$
    $value, weight \leftarrow 0$
    $i \leftarrow 1$
    while $weight + w_i \leq W$ and $i \leq n$ do
        $value \leftarrow value + v_i$
        $weight \leftarrow weight + w_i$
        $i \leftarrow i + 1$
    if $i \leq n$ then return $\max(value, v_i)$
    else return $value$
```

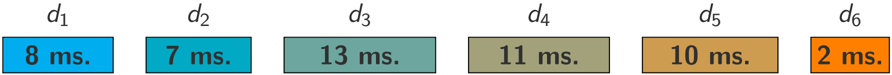
However, greedy approach is optimal when all weights are equal

Job scheduling

Given:
- n jobs of duration $d_1, \ldots, d_n \in \mathbb{N}_{>0}$
- m processors

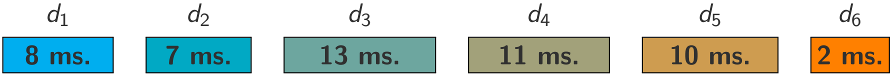
Compute: smallest amount of time to complete all jobs

Job scheduling



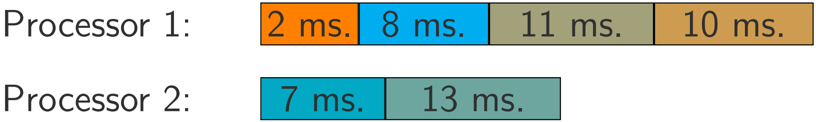
How to schedule the jobs on two processors?

Job scheduling



d_1	d_2	d_3	d_4	d_5	d_6
8 ms.	**7 ms.**	**13 ms.**	**11 ms.**	**10 ms.**	**2 ms.**

Time: 31 ms.



Processor 1: 2 ms. 8 ms. 11 ms. 10 ms.

Processor 2: 7 ms. 13 ms.

Job scheduling

d_1	d_2	d_3	d_4	d_5	d_6
8 ms.	**7 ms.**	**13 ms.**	**11 ms.**	**10 ms.**	**2 ms.**

Greedy way to obtain solution?

Job scheduling

d_1	d_2	d_3	d_4	d_5	d_6
8 ms.	**7 ms.**	**13 ms.**	**11 ms.**	**10 ms.**	**2 ms.**

Assign next job to less busy processor...

Job scheduling

d_1	d_2	d_3	d_4	d_5	d_6
8 ms.	**7 ms.**	**13 ms.**	**11 ms.**	**10 ms.**	**2 ms.**

Time: 29 ms.

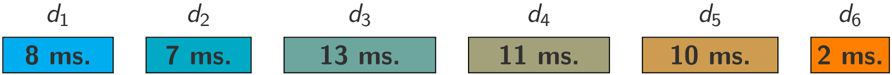
Processor 1:	8 ms.	11 ms.	10 ms.
Processor 2:	7 ms.	13 ms.	2 ms.

Job scheduling

d_1	d_2	d_3	d_4	d_5	d_6
8 ms.	**7 ms.**	**13 ms.**	**11 ms.**	**10 ms.**	**2 ms.**

Assign longest job to less busy processor...

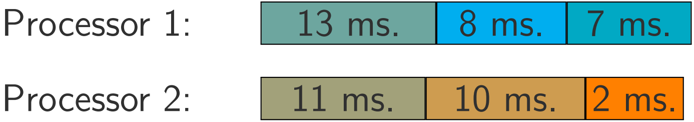
Job scheduling



d_1 d_2 d_3 d_4 d_5 d_6

8 ms. **7 ms.** **13 ms.** **11 ms.** **10 ms.** **2 ms.**

Time: 28 ms.



Processor 1: 13 ms. 8 ms. 7 ms.

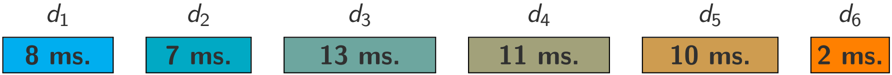
Processor 2: 11 ms. 10 ms. 2 ms.

Job scheduling

d_1	d_2	d_3	d_4	d_5	d_6
8 ms.	**7 ms.**	**13 ms.**	**11 ms.**	**10 ms.**	**2 ms.**

None are optimal!

Job scheduling



d_1 **8 ms.**

d_2 **7 ms.**

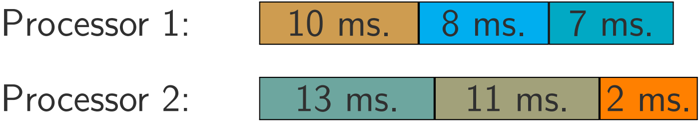
d_3 **13 ms.**

d_4 **11 ms.**

d_5 **10 ms.**

d_6 **2 ms.**

Time: 26 ms.



Processor 1: 10 ms. 8 ms. 7 ms.

Processor 2: 13 ms. 11 ms. 2 ms.

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY($d_1, \dots, d_n, m$)
    $P_1, \dots, P_m \leftarrow 0$
    $time \leftarrow 0$

    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$

    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

The solution of SCHEDULING-GREEDY is at most twice the optimal one

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$

    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

The solution of SCHEDULING-GREEDY is at most twice the optimal one

First observe that $opt \geq \max(d_1, \ldots, d_n)$ and $opt \geq \frac{1}{m}(d_1 + \ldots + d_n)$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$

    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $opt \geq \max(d_1, \ldots, d_n)$
- $opt \geq \frac{1}{m}(d_1 + \ldots + d_n)$

The solution of SCHEDULING-GREEDY is at most twice the optimal one

First observe that $opt \geq \max(d_1, \ldots, d_n)$ and $opt \geq \frac{1}{m}(d_1 + \ldots + d_n)$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$

    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $opt \geq \max(d_1, \ldots, d_n)$
- $opt \geq \frac{1}{m}(d_1 + \ldots + d_n)$

The solution of SCHEDULING-GREEDY is at most twice the optimal one

Let i^*, j^* be s.t. $P_{j^*} = time$ and i^* is the last job assigned to processor j^*

Let P'_k be the load of processor k just before job i^* is assigned

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$

    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $opt \geq \max(d_1, \ldots, d_n)$
- $opt \geq \frac{1}{m}(d_1 + \ldots + d_n)$

The solution of SCHEDULING-GREEDY is at most twice the optimal one

$$m \cdot P'_{j^*} \;\leq\; \sum_{1 \leq j \leq m} P'_j \;=\; \sum_{1 \leq i < i^*} d_i \;\leq\; \sum_{1 \leq i \leq n} d_i \;\leq\; m \cdot opt$$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$

    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $opt \geq \max(d_1, \ldots, d_n)$
- $opt \geq \frac{1}{m}(d_1 + \ldots + d_n)$
- $opt \geq P'_{j^*}$

The solution of SCHEDULING-GREEDY is at most twice the optimal one

$$m \cdot P'_{j^*} \;\leq\; \sum_{1 \leq j \leq m} P'_j \;=\; \sum_{1 \leq i < i^*} d_i \;\leq\; \sum_{1 \leq i \leq n} d_i \;\leq\; m \cdot opt$$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$

    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $opt \geq \max(d_1, \ldots, d_n)$
- $opt \geq \frac{1}{m}(d_1 + \ldots + d_n)$
- $opt \geq P'_{j^*}$

The solution of SCHEDULING-GREEDY is at most twice the optimal one

$$time \;=\; P_{j^*} \;=\; P'_{j^*} + d_{i^*} \;\leq\; opt + opt \;=\; 2 \cdot opt \quad \square$$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \ldots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \ldots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

The solution of SCHEDULING-GREEDY-ORD is at most $\frac{3}{2} \cdot opt$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \ldots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

The solution of SCHEDULING-GREEDY-ORD is at most $\frac{3}{2} \cdot opt$

Let i^*, j^* be s.t. $P_{j^*} = time$ and i^* is the last job assigned to processor j^*

From previous proof: $P_{j^*} \leq opt + d_{i^*}$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \ldots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $P_{j^*} \leq opt + d_{i^*}$

The solution of SCHEDULING-GREEDY-ORD is at most $\frac{3}{2} \cdot opt$

Let i^*, j^* be s.t. $P_{j^*} = time$ and i^* is the last job assigned to processor j^*

From previous proof: $P_{j^*} \leq opt + d_{i^*}$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \ldots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $P_{j^*} \leq opt + d_{i^*}$

The solution of SCHEDULING-GREEDY-ORD is at most $\frac{3}{2} \cdot opt$

If $i^* \leq m$, then solution is optimal. Thus, assume

$$i^* > m$$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \dots, d_n, m$)
    $P_1, \dots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \dots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $P_{j^*} \leq opt + d_{i^*}$
- $i^* > m$

The solution of SCHEDULING-GREEDY-ORD is at most $\frac{3}{2} \cdot opt$

If $i^* \leq m$, then solution is optimal. Thus, assume

$$i^* > m$$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \dots, d_n, m$)
    $P_1, \dots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \dots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $P_{j^*} \leq opt + d_{i^*}$
- $i^* > m$

The solution of SCHEDULING-GREEDY-ORD is at most $\frac{3}{2} \cdot opt$

Since $i^* > m$ and jobs are scheduled in desc. order: $d_m \geq d_{m+1} \geq d_{i^*}$.

Thus, $d_{i^*} \leq (d_m + d_{m+1})/2$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \ldots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $P_{j^*} \leq opt + d_{i^*}$
- $i^* > m$
- $d_{i^*} \leq (d_m + d_{m+1})/2$

The solution of SCHEDULING-GREEDY-ORD is at most $\frac{3}{2} \cdot opt$

Since $i^* > m$ and jobs are scheduled in desc. order: $d_m \geq d_{m+1} \geq d_{i^*}$.

Thus, $d_{i^*} \leq (d_m + d_{m+1})/2$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \ldots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $P_{j^*} \leq opt + d_{i^*}$
- $i^* > m$
- $d_{i^*} \leq (d_m + d_{m+1})/2$

The solution of SCHEDULING-GREEDY-ORD is at most $\frac{3}{2} \cdot opt$

Since $n \geq i^* > m$, for every solution there are two jobs $k, k' \in [1, m+1]$ assigned to the same processor. Thus:

$$d_m + d_{m+1} \leq d_k + d_{k'} \leq opt$$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \ldots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $P_{j^*} \leq opt + d_{i^*}$
- $i^* > m$
- $d_{i^*} \leq (d_m + d_{m+1})/2$
- $d_m + d_{m+1} \leq opt$

The solution of SCHEDULING-GREEDY-ORD is at most $\frac{3}{2} \cdot opt$

Since $n \geq i^* > m$, for every solution there are two jobs $k, k' \in [1, m+1]$ assigned to the same processor. Thus:

$$d_m + d_{m+1} \leq d_k + d_{k'} \leq opt$$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \ldots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $P_{j^*} \leq opt + d_{i^*}$
- $i^* > m$
- $d_{i^*} \leq (d_m + d_{m+1})/2$
- $d_m + d_{m+1} \leq opt$

The solution of SCHEDULING-GREEDY-ORD is at most $\frac{3}{2} \cdot opt$

Therefore:

$$time = P_{j^*} \leq opt + d_{i^*} \leq opt + \frac{d_m + d_{m+1}}{2} \leq opt + \frac{opt}{2} = \frac{3}{2} \cdot opt \quad \square$$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \dots, d_n, m$)
    $P_1, \dots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \dots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $P_{j^*} \le opt + d_{i^*}$
- $i^* > m$
- $d_{i^*} \le (d_m + d_{m+1})/2$
- $d_m + d_{m+1} \le opt$

Worst-case time complexity when implemented with min-heap:

SCHEDULING-GREEDY-ORD: $O(m + n \cdot \log m + n \cdot \log n)$

SCHEDULING-GREEDY: $O(m + n \cdot \log m)$

Job scheduling: greedy approach

```
procedure SCHEDULING-GREEDY-ORD($d_1, \ldots, d_n, m$)
    $P_1, \ldots, P_m \leftarrow 0$
    $time \leftarrow 0$
    sort $d_1, \ldots, d_n$ in descending order
    for $i \leftarrow 1$ to $n$ do
        find $j$ such that $P_j$ is minimal
        $P_j \leftarrow P_j + d_i$
        $time \leftarrow \max(time, P_j)$
    return $time$
```

- $P_{j^*} \leq opt + d_{i^*}$
- $i^* > m$
- $d_{i^*} \leq (d_m + d_{m+1})/2$
- $d_m + d_{m+1} \leq opt$

Computing an optimal solution is NP-hard, even for two processors

Uncapacitated Facility location: problem

Given:

- m customers D
- n facilities F
- for each $i \in F$ and $j \in D$:
  - $f_i \in \mathbb{R}_+$: cost for opening facility i
  - $c_{i,j} \in \mathbb{R}_+$: service cost for serving customer j with facility i

Goal: open facilities $\emptyset \neq X \subseteq F$ s.t. costs are minimized, i.e. minimize

$$\sum_{i \in X} f_i + \sum_{j \in D} c_{\sigma_X(j),j}$$

where $\sigma_X(j) = \arg\min_{i \in X} c_{i,j}$ is the facility assigned to customer j.

Cost structure

- We assume: costs are metric, i.e. for all $i, i' \in F, j, j' \in D$:

$$c_{i,j} + c_{i',j} + c_{i',j'} \geq c_{i,j'}$$

- With this restriction, the problem is known as *Metric Uncapacitated Facility Location Problem*
- Metric version:

Cost structure

- We assume: costs are metric, i.e. for all $i, i' \in F, j, j' \in D$:

$$c_{i,j} + c_{i',j} + c_{i',j'} \geq c_{i,j'}$$

- With this restriction, the problem is known as *Metric Uncapacitated Facility Location Problem*
- Metric version:
    - Exact solution NP-hard

Cost structure

- We assume: costs are metric, i.e. for all $i, i' \in F, j, j' \in D$:

$$c_{i,j} + c_{i',j} + c_{i',j'} \geq c_{i,j'}$$

- With this restriction, the problem is known as *Metric Uncapacitated Facility Location Problem*
- Metric version:
    - Exact solution NP-hard
    - First constant-factor approximation discovered in 1997 (factor: 3.16)

Cost structure

- We assume: costs are metric, i.e. for all $i, i' \in F, j, j' \in D$:

$$c_{i,j} + c_{i',j} + c_{i',j'} \geq c_{i,j'}$$

- With this restriction, the problem is known as *Metric Uncapacitated Facility Location Problem*
- Metric version:
  - Exact solution NP-hard
  - First constant-factor approximation discovered in 1997 (factor: 3.16)
  - 1.488-factor approximation discovered in 2011

Cost structure

- We assume: costs are metric, i.e. for all $i, i' \in F, j, j' \in D$:

$$c_{i,j} + c_{i',j} + c_{i',j'} \geq c_{i,j'}$$

- With this restriction, the problem is known as *Metric Uncapacitated Facility Location Problem*
- Metric version:
  - Exact solution NP-hard
  - First constant-factor approximation discovered in 1997 (factor: 3.16)
  - 1.488-factor approximation discovered in 2011
  - No 1.463-factor approximation exists, unless P = NP

Cost structure

- We assume: costs are metric, i.e. for all $i, i' \in F, j, j' \in D$:

$$c_{i,j} + c_{i',j} + c_{i',j'} \geq c_{i,j'}$$

- With this restriction, the problem is known as *Metric Uncapacitated Facility Location Problem*
- Metric version:
  - Exact solution NP-hard
  - First constant-factor approximation discovered in 1997 (factor: 3.16)
  - 1.488-factor approximation discovered in 2011
  - No 1.463-factor approximation exists, unless P $=$ NP
  - Further restricting costs to distances in $\mathbb{R}^n$ allows for PTAS, i.e. $(1+\varepsilon)$-factor approximations for any $\varepsilon > 0$

Cost structure

- We assume: costs are metric, i.e. for all $i, i' \in F, j, j' \in D$:

$$c_{i,j} + c_{i',j} + c_{i',j'} \geq c_{i,j'}$$

- With this restriction, the problem is known as *Metric Uncapacitated Facility Location Problem*
- Metric version:
  - Exact solution NP-hard
  - First constant-factor approximation discovered in 1997 (factor: 3.16)
  - 1.488-factor approximation discovered in 2011
  - No 1.463-factor approximation exists, unless P = NP
  - Further restricting costs to distances in $\mathbb{R}^n$ allows for PTAS, i.e. $(1+\varepsilon)$-factor approximations for any $\varepsilon > 0$
- Non-metric case: No constant-factor approximation exists, unless P = NP.

Facility Location: Greedy Approximation

Idea:

Facility Location: Greedy Approximation

Idea:

- Greedily open facilities

Facility Location: Greedy Approximation

Idea:

- Greedily open facilities
- Do not assign *all* customers to facilities *yet*

Facility Location: Greedy Approximation

Idea:

- Greedily open facilities
- Do not assign *all* customers to facilities *yet*
- Keep track of partial assignment $\sigma : A \to X$, where X is set of already opened facilities and A set of already assigned customers

Facility Location: Greedy Approximation

Idea:

- Greedily open facilities
- Do not assign *all* customers to facilities *yet*
- Keep track of partial assignment $\sigma : A \to X$, where X is set of already opened facilities and A set of already assigned customers
- Choose next facility i and new customers N so that average additional cost per customer is minimized

Facility Location: Greedy Approximation

Idea:

- Greedily open facilities
- Do not assign *all* customers to facilities *yet*
- Keep track of partial assignment $\sigma : A \to X$, where X is set of already opened facilities and A set of already assigned customers
- Choose next facility i and new customers N so that average additional cost per customer is minimized
  - Take potential decrease of cost for already assigned customers into account!

Facility Location: Greedy Approximation

Idea:

- Greedily open facilities
- Do not assign *all* customers to facilities *yet*
- Keep track of partial assignment $\sigma : A \to X$, where X is set of already opened facilities and A set of already assigned customers
- Choose next facility i and new customers N so that average additional cost per customer is minimized
  - Take potential decrease of cost for already assigned customers into account!
  - If assigning customers to an already open facility is more efficient: do that!

Preparations

Assume so far we have assigned clients $A \subseteq D$, opened facilities $X \subseteq F$, and consider assigning new clients $\emptyset \neq N \subseteq D \setminus A$ to a facility $i \in F$. Denote the additional cost by $c(i, N)$. We have:

Preparations

Assume so far we have assigned clients $A \subseteq D$, opened facilities $X \subseteq F$, and consider assigning new clients $\emptyset \neq N \subseteq D \setminus A$ to a facility $i \in F$. Denote the additional cost by $c(i, N)$. We have:

- i must be opened iff it is not open yet, contributing $f_i \cdot [i \notin X]$ (where $[\text{true}] = 1$ and $[\text{false}] = 0$).

Preparations

Assume so far we have assigned clients $A \subseteq D$, opened facilities $X \subseteq F$, and consider assigning new clients $\emptyset \neq N \subseteq D \setminus A$ to a facility $i \in F$. Denote the additional cost by $c(i, N)$. We have:

- i must be opened iff it is not open yet, contributing $f_i \cdot [i \notin X]$ (where $[\text{true}] = 1$ and $[\text{false}] = 0$).
- The new clients contribute $\sum_{j \in N} c_{i,j}$.

Preparations

Assume so far we have assigned clients $A \subseteq D$, opened facilities $X \subseteq F$, and consider assigning new clients $\emptyset \neq N \subseteq D \setminus A$ to a facility $i \in F$. Denote the additional cost by $c(i, N)$. We have:

- i must be opened iff it is not open yet, contributing $f_i \cdot [i \notin X]$ (where $[\text{true}] = 1$ and $[\text{false}] = 0$).
- The new clients contribute $\sum_{j \in N} c_{i,j}$.
- If a new facility is opened, we might be able to decrease our costs by reassigning some customers in A, contributing $-\sum_{j \in A_{\sigma,i}} (c_{\sigma(j),j} - c_{i,j})$, where

$$A_{\sigma,i} := \{ j \in A \mid c_{i,j} < c_{\sigma(j),j} \}.$$

Preparations

Assume so far we have assigned clients $A \subseteq D$, opened facilities $X \subseteq F$, and consider assigning new clients $\emptyset \neq N \subseteq D \setminus A$ to a facility $i \in F$. Denote the additional cost by $c(i, N)$. We have:

- i must be opened iff it is not open yet, contributing $f_i \cdot [i \notin X]$ (where $[\text{true}] = 1$ and $[\text{false}] = 0$).
- The new clients contribute $\sum_{j \in N} c_{i,j}$.
- If a new facility is opened, we might be able to decrease our costs by reassigning some customers in A, contributing $-\sum_{j \in A_{\sigma,i}} (c_{\sigma(j),j} - c_{i,j})$, where

$$A_{\sigma,i} := \{j \in A \mid c_{i,j} < c_{\sigma(j),j}\}.$$

We want to minimize the average cost increase per new customer, i.e.

$$\varphi(i, N) := \frac{c(i, N)}{|N|} = \frac{f_i \cdot [i \notin X] + \sum_{j \in N} c_{i,j} - \sum_{j \in A_{\sigma,i}} (c_{\sigma(j),j} - c_{i,j})}{|N|}$$

Facility Location: Greedy Approximation

Algorithm Greedy Approximation for Facility Location

Input: n facilities F, m customers $D, (f_i)_{i \in F}, (c_{i,j})_{i \in F, j \in D}$

Output: Approximation $\emptyset \neq X \subseteq F$ (and assignment $\sigma_X : D \to X$)

```
$X \leftarrow \emptyset$, $A \leftarrow \emptyset$, $\sigma \leftarrow \bot$ (empty map)
while $A \neq D$ do
    Choose $(i, N)$ with $i \in F$ and $\emptyset \neq N \subseteq D \setminus A$ minimizing $\varphi(i, N)$
    for all $j \in N \cup A_{\sigma,i}$ do
        $\sigma(j) \leftarrow i$
    $X \leftarrow X \cup \{i\}$
    $A \leftarrow A \cup N$
return $X$ (and $\sigma_X := \sigma$)
```

Performance guarantees

- This version is a 2-factor approximation.
- It can be implemented to run in polynomial time.
- Details are skipped, except:

Performance guarantees

- This version is a 2-factor approximation.
- It can be implemented to run in polynomial time.
- Details are skipped, except:
  - How to find the next (i, N) in polynomial time?

Finding (i, N)

Lemma

Let $X = \{x_1, \ldots, x_n\}$, *where* $x_1 < \ldots < x_n$. *Further, let* $c \in \mathbb{R}$. *Then*

$$\min_{\emptyset \neq Y \subseteq X} \frac{c + \sum Y}{|Y|} = \min_{i \in [n]} \frac{c + \sum X_i}{|X_i|}$$

where

$$X_i := \{x_1, \ldots, x_i\}.$$

Proof.

"$\leq$" is clear. For "$\geq$", let $\emptyset \neq Y \subseteq X$. Define $i := |Y|$. Then $\sum Y \geq \sum X_i$ and $|Y| = |X_i|$, and hence $\frac{c+\sum Y}{|Y|} \geq \frac{c+\sum X_i}{|X_i|}$.

Local search

Local search

- Guess some solution

Local search

- Guess some solution
- Improve it

Local search

- Guess some solution
- Improve it
- Repeat

Local search

- Guess some solution
- Improve it
- Repeat
- Success!

Local search

- Guess some solution
- Improve it
- Repeat
- Success! (?)

Local search: formal definition

Let $G = (V, E)$ be a directed graph, and $c : V \to \mathbb{R}$. A *local search* algorithm is an algorithm of the following form:

Algorithm Local search

```
Choose $v \in V$.
loop
    if $v$ is good enough, or time limit exceeded, or ... then
        return $v$
    Choose $v' \in vE$.
    $v \leftarrow v'$
end loop
```

Local search: Examples

Example (CNF-SAT)

Let $\varphi = C_1 \wedge \ldots \wedge C_n$ be a formula in CNF with variables $x_1, \ldots, x_m$. Choose $V = \{0,1\}^m$, and define

$$((v_1, \ldots, v_m), (w_1, \ldots, w_m)) \in E :\Longleftrightarrow \exists! j \in [m] : v_j \neq w_j$$

i.e. the neighbors of an assignment are the possible results of switching exactly one variable. Define

$$c : V \to \mathbb{R}, c(v) := |\{i \in [n] \mid v(C_i) = 1\}|$$

i.e. $c(v)$ is the number of clauses C_i that are satisfied by the assignment v. Note that φ is satisfiable iff $\max_{v \in V} c(v) = n$. Observations:

- Huge state space: 2^m
- Small neighborhoods: m
- (V, E) connected

Local search: Examples

Example (TSP)

In the Travelling Salesman Problem, we are given n cities, and pairwise distances $(d_{i,j})_{i,j \in [n]}$. The goal is to find a roundtrip of shortest length. Choose

- $V := S_n$ (i.e. states are permutations)
- Connect σ, τ in E iff there are $i < j$ s.t. σ can be obtained by "reversing" the images of $i, \ldots, j$ in τ, i.e. iff

$$\sigma_1 \cdots \sigma_n = \tau_1 \cdots \tau_{i-1} \tau_j \tau_{j-1} \ldots \tau_i \tau_{j+1} \cdots \tau_n$$

- $c : V \to \mathbb{R}, \sigma \mapsto \sum_{i \in [n]} d_{\sigma(i), \sigma(i+1)}$ (where $n+1$ is interpreted as 1)

Observations:

- Huge state space: $n!$
- Small neighborhoods: $\mathcal{O}(n^2)$
- (V, E) connected

Local search: Gradient ascent/descent

Common local search algorithm: Gradient ascent/descent

Local search: Gradient ascent/descent

Common local search algorithm: Gradient ascent/descent

- W.l.o.g., only consider gradient ascent (try to maximize c)

Local search: Gradient ascent/descent

Common local search algorithm: Gradient ascent/descent

- W.l.o.g., only consider gradient ascent (try to maximize c)
- Greedily choose $v' \in vE$ maximizing $c(v')$ (among neighborhood)

Local search: Gradient ascent/descent

Common local search algorithm: Gradient ascent/descent

- W.l.o.g., only consider gradient ascent (try to maximize c)
- Greedily choose $v' \in vE$ maximizing $c(v')$ (among neighborhood)
    - E.g. if vE is finite: iterate over vE to find $\arg\max_{v' \in vE} c(v')$

Local search: Gradient ascent/descent

Common local search algorithm: Gradient ascent/descent

- W.l.o.g., only consider gradient ascent (try to maximize c)
- Greedily choose $v' \in vE$ maximizing $c(v')$ (among neighborhood)
  - E.g. if vE is finite: iterate over vE to find $\arg\max_{v' \in vE} c(v')$
  - E.g. if $V \subseteq \mathbb{R}^n$ and f is cont. differentiable: choose $v' = v + \nabla f(v)h$ for some $h \in \mathbb{R}_+$

Local search: Gradient ascent/descent

Common local search algorithm: Gradient ascent/descent

- W.l.o.g., only consider gradient ascent (try to maximize c)
- Greedily choose $v' \in vE$ maximizing $c(v')$ (among neighborhood)
    - E.g. if vE is finite: iterate over vE to find $\arg\max_{v' \in vE} c(v')$
    - E.g. if $V \subseteq \mathbb{R}^n$ and f is cont. differentiable: choose $v' = v + \nabla f(v)h$ for some $h \in \mathbb{R}_+$
- return once no improvement possible

Gradient descent: benefits

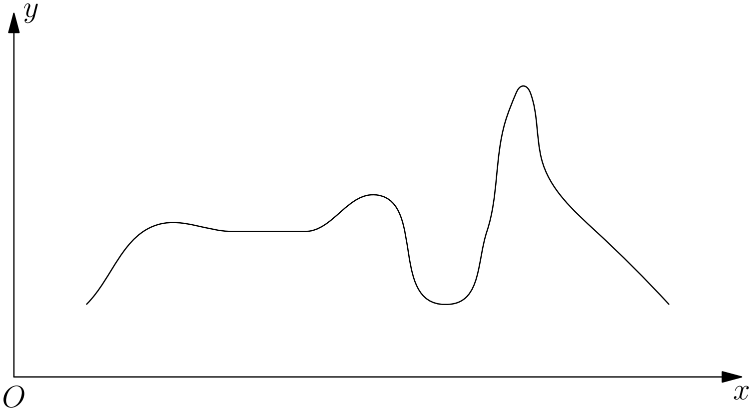
- Simple
- Things always improve!
- Can stop early and get. . . something

Gradient descent: drawbacks

- Cannot escape local optima
- Plateaus

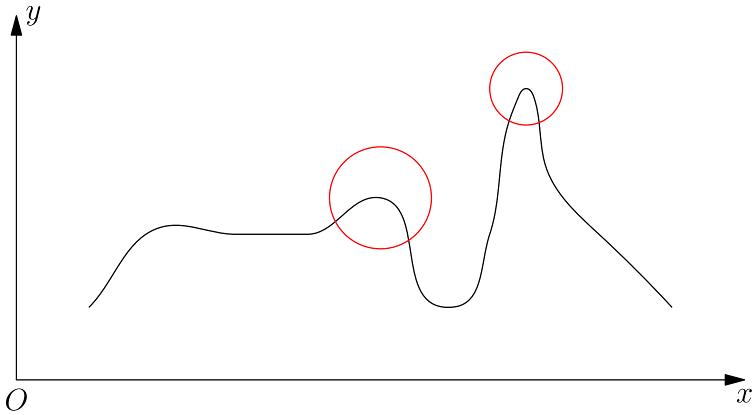
Gradient descent: drawbacks

- Cannot escape local optima
- Plateaus



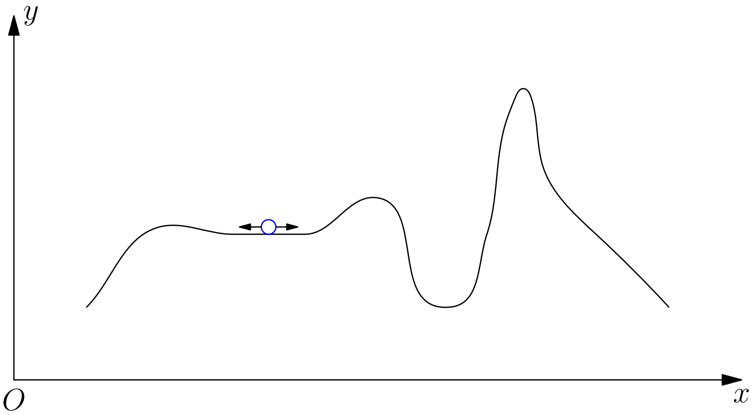
Gradient descent: drawbacks

- Cannot escape local optima
-



Gradient descent: drawbacks

-
- Plateaus



Local search: remarks

- There exist approaches to deal with these problems, e.g. Simulated Annealing:
    - Allow to go to "worse" neighbors with certain probability
    - Decrease this probability over time

Local search: remarks

- There exist approaches to deal with these problems, e.g. Simulated Annealing:
    - Allow to go to "worse" neighbors with certain probability
    - Decrease this probability over time
    - Let X_n be the random variable denoting the state in V after n steps. One can show: if done carefully, one can achieve that distribution of X_n converges to distribution with support in optimizers, i.e. procedure converges to optima.

Local search: remarks

- There exist approaches to deal with these problems, e.g. Simulated Annealing:
    - Allow to go to "worse" neighbors with certain probability
    - Decrease this probability over time
    - Let X_n be the random variable denoting the state in V after n steps. One can show: if done carefully, one can achieve that distribution of X_n converges to distribution with support in optimizers, i.e. procedure converges to optima.
- Also possible: heuristics, e.g. start in different positions (or split execution), take best result of any run