CSE 221: Algorithms

Greedy algorithms

Mumit Khan
Fatema Tuz Zohora

Computer Science and Engineering
BRAC University

References

1. Jon Kleinberg and Éva Tardos, *Algorithm Design*. Pearson Education, 2006.
2. Michael T. Goodrich and Roberto Tamassia, *Data Structures and Algorithms in Java, Fourth Edition*. John Wiley & Sons, 2006.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

Last modified: November 13, 2012

Contents

Contents

Greedy design strategy

Greed . . . is good. Greed is right. Greed works.

Greedy design strategy

Greed . . . is good. Greed is right. Greed works.

Gordon Gekko (played by Michael Douglas), in the 1987 movie *Wall Street*.

Greedy design strategy

Greed . . . is good. Greed is right. Greed works.

Gordon Gekko (played by Michael Douglas), in the 1987 movie *Wall Street*.

Basic idea:

- Solves the problem step by step.

Greedy design strategy

Greed . . . is good. Greed is right. Greed works.

Gordon Gekko (played by Michael Douglas), in the 1987 movie *Wall Street*.

Basic idea:

- Solves the problem step by step.
- At each step, pick the choice which looks best **based on some criteria**(i.e. *greedily*) at that moment given the information currently available.

Greedy design strategy

Greed . . . is good. Greed is right. Greed works.

Gordon Gekko (played by Michael Douglas), in the 1987 movie *Wall Street*.

Basic idea:

- Solves the problem step by step.
- At each step, pick the choice which looks best **based on some criteria**(i.e. *greedily*) at that moment given the information currently available.
- Main challenge is to determine the criteria on which the next item will be selected

Greedy design strategy

Greed . . . is good. Greed is right. Greed works.

Gordon Gekko (played by Michael Douglas), in the 1987 movie *Wall Street*.

Basic idea:

- Solves the problem step by step.
- At each step, pick the choice which looks best **based on some criteria**(i.e. *greedily*) at that moment given the information currently available.
- Main challenge is to determine the criteria on which the next item will be selected
- Often leads to very efficient solutions to optimization problems.

Greedy design strategy

Greed . . . is good. Greed is right. Greed works.

Gordon Gekko (played by Michael Douglas), in the 1987 movie *Wall Street*.

Basic idea:

- Solves the problem step by step.
- At each step, pick the choice which looks best **based on some criteria**(i.e. *greedily*) at that moment given the information currently available.
- Main challenge is to determine the criteria on which the next item will be selected
- Often leads to very efficient solutions to optimization problems.
- However, not all problems have greedy solutions.

Optimal Solution

What is an Optimal Solution?

- Given a problem, more than one solution exist
- One of the solution is the best based on some given constraints, that solution is called the optimal solution

What is Global Optimal Solution?

- Optimal Solution to the main problem

What is local Optimal Solution?

- Optimal Solution to the subproblems

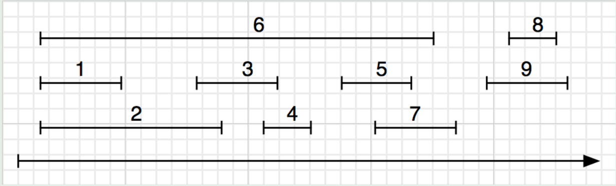
Contents

Designing a greedy algorithm

Definition (Interval scheduling problem)

Given a set of schedules $I = \{I_i\}$, find the largest set $A \subseteq I$ such that the members of A are non-conflicting.

Example

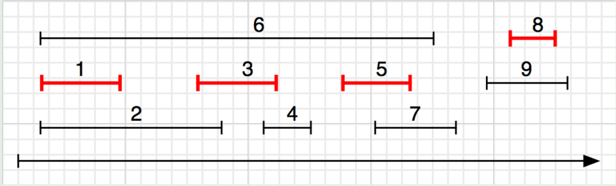


Designing a greedy algorithm

Definition (Interval scheduling problem)

Given a set of schedules $I = \{I_i\}$, find the largest set $A \subseteq I$ such that the members of A are non-conflicting.

Example



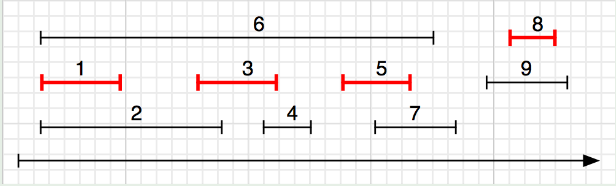
$A = \{1, 3, 5, 8\}, \quad |A| = 4.$

Designing a greedy algorithm

Definition (Interval scheduling problem)

Given a set of schedules $I = \{I_i\}$, find the largest set $A \subseteq I$ such that the members of A are non-conflicting.

Example



$A = \{1, 3, 5, 8\}, \quad |A| = 4.$

Question

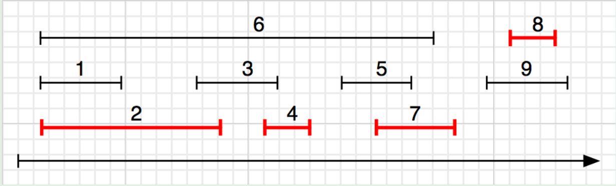
Is this the only "correct" answer?

Designing a greedy algorithm

Definition (Interval scheduling problem)

Given a set of schedules $I = \{I_i\}$, find the largest set $A \subseteq I$ such that the members of A are non-conflicting.

Example



$A = \{1, 3, 5, 8\}, \quad |A| = 4.$

Question

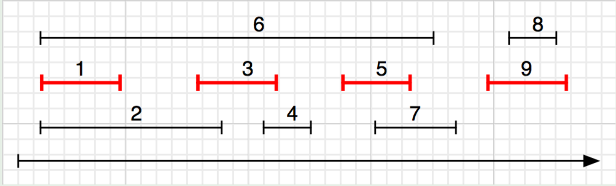
How about $\{2, 4, 7, 8\}$?

Designing a greedy algorithm

Definition (Interval scheduling problem)

Given a set of schedules $I = \{I_i\}$, find the largest set $A \subseteq I$ such that the members of A are non-conflicting.

Example



$A = \{1, 3, 5, 8\}, \quad |A| = 4.$

Question

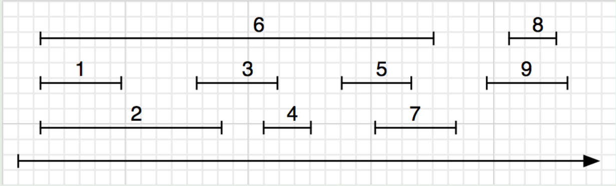
How about $\{2, 4, 7, 8\}$? $\{1, 3, 5, 9\}$?

Designing a greedy algorithm

Definition (Interval scheduling problem)

Given a set of schedules $I = \{I_i\}$, find $A \subseteq I$ such that the members of A are non-conflicting *and* $|A|$ is maximized.

Example



$A = \{1, 3, 5, 8\}, \quad |A| = 4.$

Question

$\{1, 3, 5, 8\}$? $\{2, 4, 7, 8\}$? $\{1, 3, 5, 9\}$? ... **How many?**

Brute force solution

A solution must be one of the subsets of the set of intervals.

Brute force solution

A solution must be one of the subsets of the set of intervals.

1. Enumerate all possible *configurations* (i.e., all possible subsets of the intervals).

Brute force solution

A solution must be one of the subsets of the set of intervals.

1. Enumerate all possible *configurations* (i.e., all possible subsets of the intervals).
2. Go through the set of subsets and remove the ones that have one or more conflicting schedules.

Brute force solution

A solution must be one of the subsets of the set of intervals.

1. Enumerate all possible *configurations* (i.e., all possible subsets of the intervals).
2. Go through the set of subsets and remove the ones that have one or more conflicting schedules.
3. Pick (any one of) the largest subset from the ones that survive.

Brute force solution

A solution must be one of the subsets of the set of intervals.

1. Enumerate all possible *configurations* (i.e., all possible subsets of the intervals).
2. Go through the set of subsets and remove the ones that have one or more conflicting schedules.
3. Pick (any one of) the largest subset from the ones that survive.

Complexity

- There are $2^n - 1$ non-empty subsets, one or more of which may be a feasible solution.

Brute force solution

A solution must be one of the subsets of the set of intervals.

1. Enumerate all possible *configurations* (i.e., all possible subsets of the intervals).
2. Go through the set of subsets and remove the ones that have one or more conflicting schedules.
3. Pick (any one of) the largest subset from the ones that survive.

Complexity

- There are $2^n - 1$ non-empty subsets, one or more of which may be a feasible solution.
- Each feasible solution must be scanned for conflict, which takes $O(n)$ time.

Brute force solution

A solution must be one of the subsets of the set of intervals.

1. Enumerate all possible *configurations* (i.e., all possible subsets of the intervals).
2. Go through the set of subsets and remove the ones that have one or more conflicting schedules.
3. Pick (any one of) the largest subset from the ones that survive.

Complexity

- There are $2^n - 1$ non-empty subsets, one or more of which may be a feasible solution.
- Each feasible solution must be scanned for conflict, which takes $O(n)$ time.
- The algorithm runs in $\Theta(n2^n)$ time

Brute force solution

A solution must be one of the subsets of the set of intervals.

1. Enumerate all possible *configurations* (i.e., all possible subsets of the intervals).
2. Go through the set of subsets and remove the ones that have one or more conflicting schedules.
3. Pick (any one of) the largest subset from the ones that survive.

Complexity

- There are $2^n - 1$ non-empty subsets, one or more of which may be a feasible solution.
- Each feasible solution must be scanned for conflict, which takes $O(n)$ time.
- The algorithm runs in $\Theta(n2^n)$ time $\Rightarrow$ an exponential time algorithm!

Designing a greedy algorithm (continued)

Basic steps

To compute the maximal set of intervals that can be scheduled, the basic idea is to:

Designing a greedy algorithm (continued)

Basic steps

To compute the maximal set of intervals that can be scheduled, the basic idea is to:

1. Use a “simple” rule (or strategy) to select the first interval i_1 to be accepted.

Designing a greedy algorithm (continued)

Basic steps

To compute the maximal set of intervals that can be scheduled, the basic idea is to:

1. Use a “simple” rule (or strategy) to select the first interval i_1 to be accepted.
2. Once i_1 is accepted, remove from consideration all intervals the conflict with i_1.

Designing a greedy algorithm (continued)

Basic steps

To compute the maximal set of intervals that can be scheduled, the basic idea is to:

1. Use a “simple” rule (or strategy) to select the first interval i_1 to be accepted.
2. Once i_1 is accepted, remove from consideration all intervals the conflict with i_1.
3. Select the second interval i_2 to be accepted, and remove all the intervals that conflict with i_2.

Designing a greedy algorithm (continued)

Basic steps

To compute the maximal set of intervals that can be scheduled, the basic idea is to:

1. Use a “simple” rule (or strategy) to select the first interval i_1 to be accepted.
2. Once i_1 is accepted, remove from consideration all intervals the conflict with i_1.
3. Select the second interval i_2 to be accepted, and remove all the intervals that conflict with i_2.
4. And so on until there are no more requests remain.

Designing a greedy algorithm (continued)

Basic steps

To compute the maximal set of intervals that can be scheduled, the basic idea is to:

1. Use a "simple" rule (or strategy) to select the first interval i_1 to be accepted.
2. Once i_1 is accepted, remove from consideration all intervals the conflict with i_1.
3. Select the second interval i_2 to be accepted, and remove all the intervals that conflict with i_2.
4. And so on until there are no more requests remain.

Key challenge

How to choose the "simple" rule to select the next interval that leads to an optimal solution?

Designing a greedy algorithm (continued)

Strategy 1. *Earliest First*

The idea is to start using the resource as early as possible.

1. Sort the intervals by starting time, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

Example

$|A| = ???.$

Designing a greedy algorithm (continued)

Strategy 1. *Earliest First*

The idea is to start using the resource as early as possible.

1. Sort the intervals by starting time, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

Example (using ***Earliest First*** **strategy)**

$$|A| = 1.$$

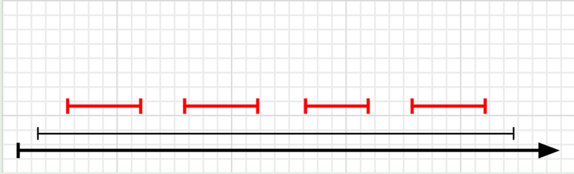
Designing a greedy algorithm (continued)

Strategy 1. *Earliest First*

The idea is to start using the resource as early as possible.

1. Sort the intervals by starting time, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

Example (using an optimal strategy)



$|A| = 4.$

Designing a greedy algorithm (continued)

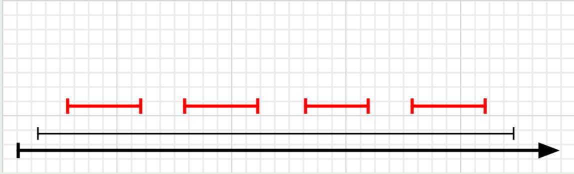
Strategy 1. *Earliest First*

The idea is to start using the resource as early as possible.

1. Sort the intervals by starting time, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

This strategy does not lead to an optimal solution.

Example (using an optimal strategy)



$$|A| = 4.$$

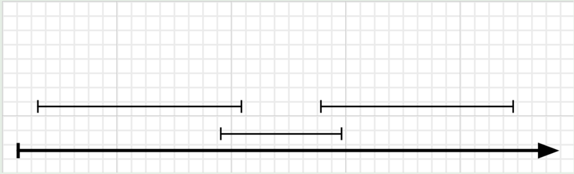
Designing a greedy algorithm (continued)

Strategy 2. *Shortest First*

The *Earliest First* strategy failed perhaps because it missed the shorter intervals, which would accommodate more intervals.

1. Sort the intervals by length, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

Example



$|A| = ???.$

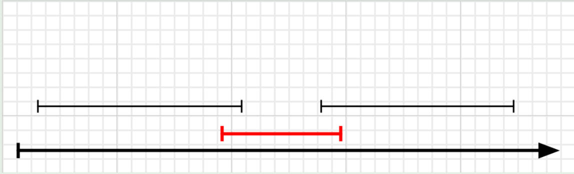
Designing a greedy algorithm (continued)

Strategy 2. *Shortest First*

The *Earliest First* strategy failed perhaps because it missed the shorter intervals, which would accommodate more intervals.

1. Sort the intervals by length, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

Example (using *Shortest First* strategy)



$|A| = 1.$

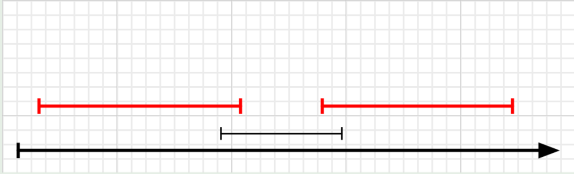
Designing a greedy algorithm (continued)

Strategy 2. *Shortest First*

The *Earliest First* strategy failed perhaps because it missed the shorter intervals, which would accommodate more intervals.

1. Sort the intervals by length, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

Example (using an optimal strategy)



$|A| = 2.$

Designing a greedy algorithm (continued)

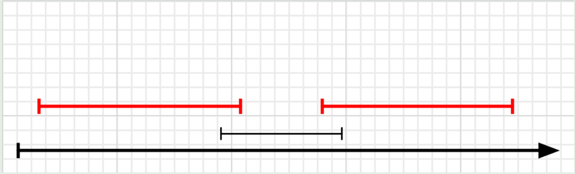
Strategy 2. *Shortest First*

The *Earliest First* strategy failed perhaps because it missed the shorter intervals, which would accommodate more intervals.

1. Sort the intervals by length, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

This strategy does not lead to an optimal solution.

Example (using an optimal strategy)



$|A| = 2.$

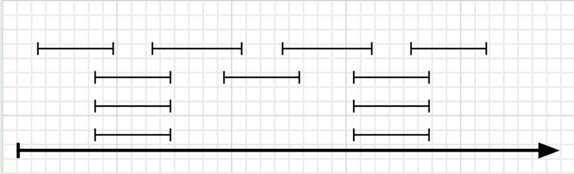
Designing a greedy algorithm (continued)

Strategy 3. *Least-conflict First*

The *Shortest First* strategy failed perhaps because the shorter ones had more conflicts, and ruled out too many intervals in the process.

1. Sort the intervals by the number of other intervals which conflict with it.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

Example



$|A| = ???.$

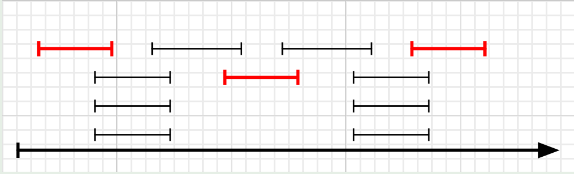
Designing a greedy algorithm (continued)

Strategy 3. *Least-conflict First*

The *Shortest First* strategy failed perhaps because the shorter ones had more conflicts, and ruled out too many intervals in the process.

1. Sort the intervals by the number of other intervals which conflict with it.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

Example (using *Shortest First* strategy)



$$|A| = 3.$$

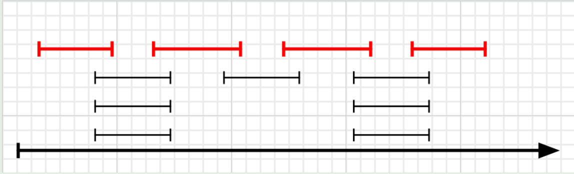
Designing a greedy algorithm (continued)

Strategy 3. *Least-conflict First*

The *Shortest First* strategy failed perhaps because the shorter ones had more conflicts, and ruled out too many intervals in the process.

1. Sort the intervals by the number of other intervals which conflict with it.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

Example (using an optimal strategy)



$|A| = 4.$

Designing a greedy algorithm (continued)

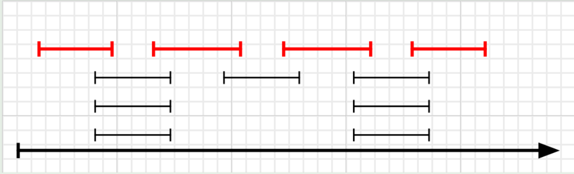
Strategy 3. *Least-conflict First*

The *Shortest First* strategy failed perhaps because the shorter ones had more conflicts, and ruled out too many intervals in the process.

1. Sort the intervals by the number of other intervals which conflict with it.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

This strategy does not lead to an optimal solution.

Example (using an optimal strategy)



$|A| = 4.$

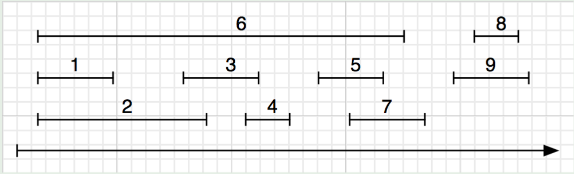
Designing a greedy algorithm (continued)

Strategy 4. *Finish First*

The idea is to free up the resource as early as possible.

1. Sort the intervals by the finishing time, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

Example



$|A| =???$.

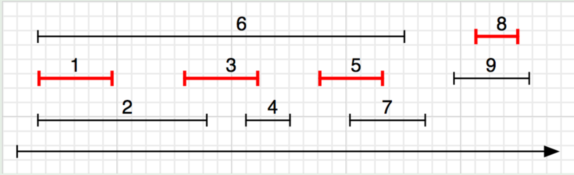
Designing a greedy algorithm (continued)

Strategy 4. *Finish First*

The idea is to free up the resource as early as possible.

1. Sort the intervals by the finishing time, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

Example (using optimal *Finish First* strategy)



$$|A| = 4.$$

Designing a greedy algorithm (continued)

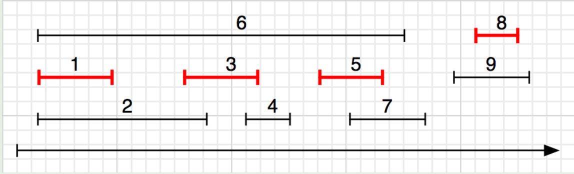
Strategy 4. *Finish First*

The idea is to free up the resource as early as possible.

1. Sort the intervals by the finishing time, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

This strategy is the one that works.

Example (using optimal *Finish First* strategy)



$|A| = 4.$

Designing a greedy algorithm (continued)

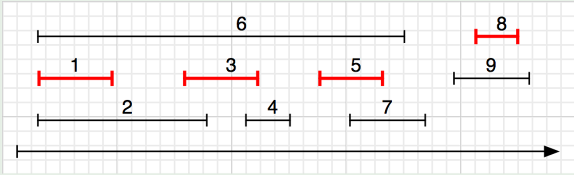
Strategy 4. *Finish First*

The idea is to free up the resource as early as possible.

1. Sort the intervals by the finishing time, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the intervals that conflict with it.
3. Repeat Step 2, until the list is empty.

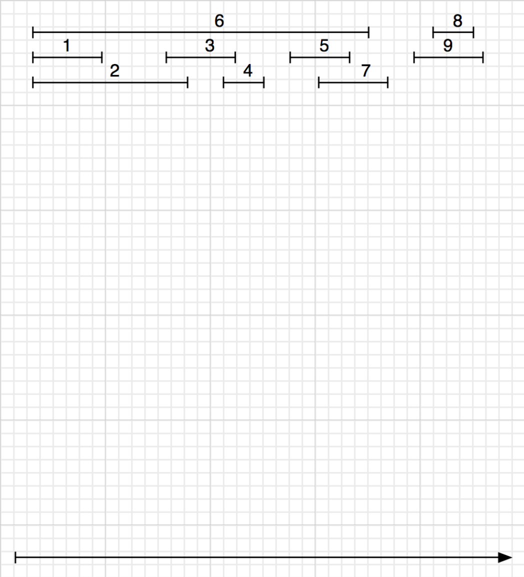
This strategy is the one that works. But can you prove that it works?

Example (using optimal *Finish First* strategy)

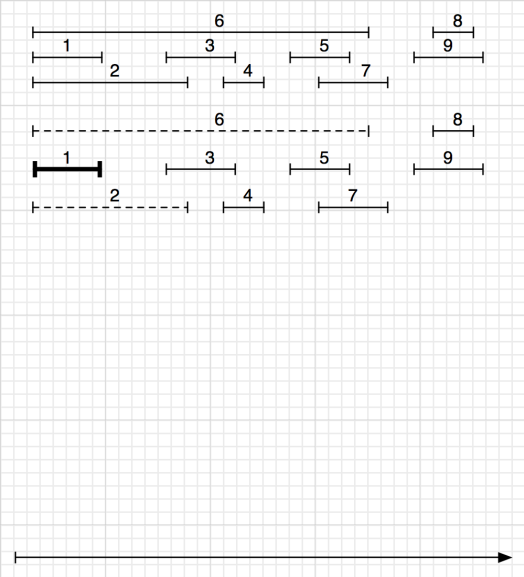


$|A| = 4.$

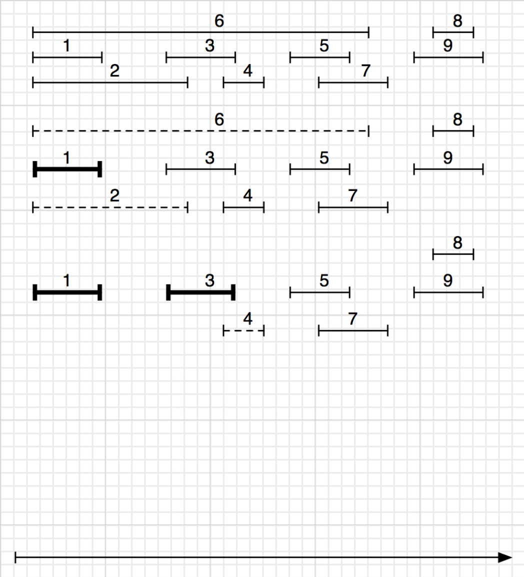
Interval scheduling in action



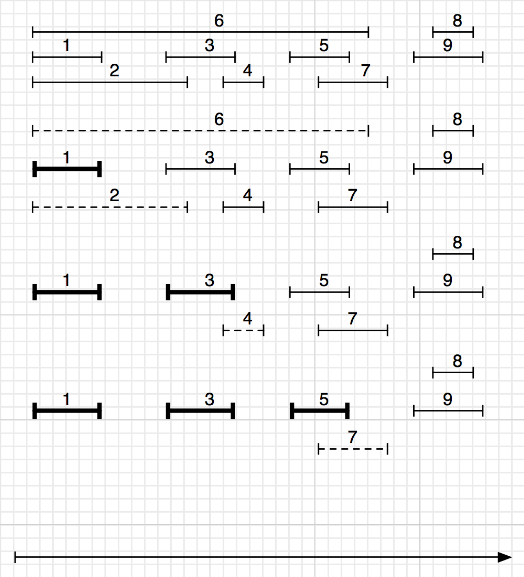
Interval scheduling in action



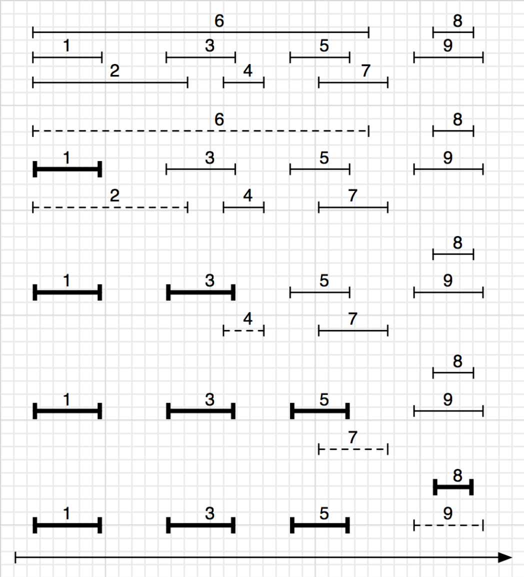
Interval scheduling in action



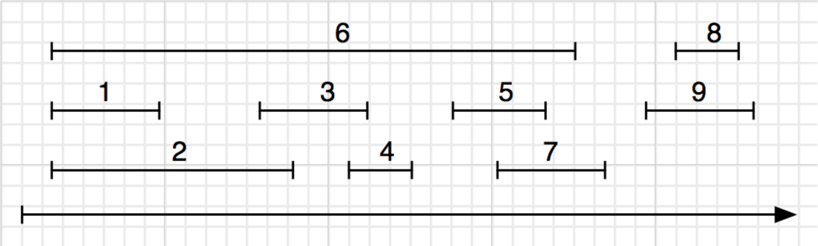
Interval scheduling in action



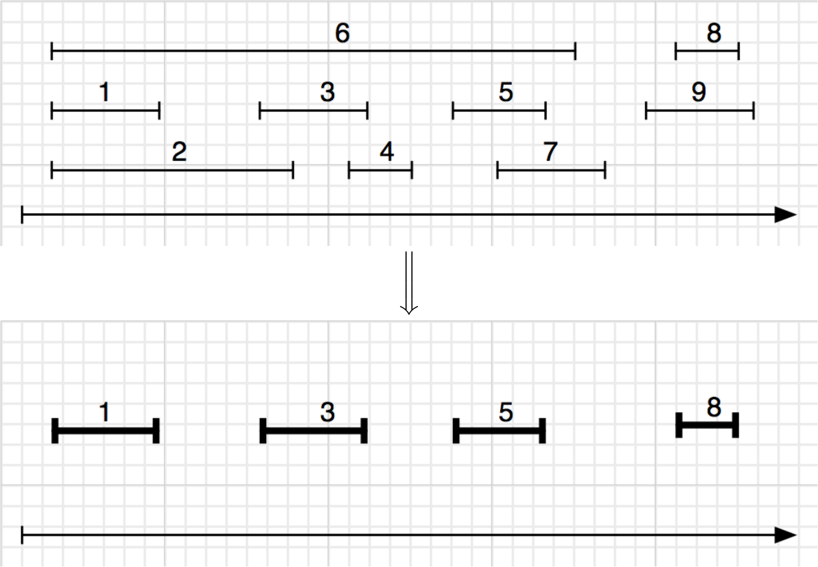
Interval scheduling in action



Interval scheduling in action (continued)



Interval scheduling in action (continued)



An $O(n \lg n)$ greedy algorithm for interval scheduling

SCHEDULE-INTERVALS(I) $\quad \triangleright\ I = \{I_i\}, I_i = (s_i, f_i)$

R = Sorted requests in order of finishing times such that $f_i \le f_j$ when $i < j$.
Create an array $S[1 \,..\, n]$ with starting times such that $S[i]$ contains s_i.
$A = \{R_1\}$ $\quad \triangleright$ select first interval from sorted list
$f = f_1$
while there are more intervals in S to look at
    do j = first interval for which $s_j \ge f$
        $A \leftarrow A \cup \{j\}$
        $f \leftarrow f_j$
return A

An $O(n \lg n)$ greedy algorithm for interval scheduling

SCHEDULE-INTERVALS(I) $\triangleright$ $I = \{I_i\}, I_i = (s_i, f_i)$

R = Sorted requests in order of finishing times such that $f_i \leq f_j$ when $i < j$.
Create an array $S[1 \,..\, n]$ with starting times such that $S[i]$ contains s_i.
$A = \{R_1\}$ $\triangleright$ select first interval from sorted list
$f = f_1$
while there are more intervals in S to look at
  do j = first interval for which $s_j \geq f$
    $A \leftarrow A \cup \{j\}$
    $f \leftarrow f_j$
return A

Analysis

- The sorting step in takes $O(n \lg n)$ time.

An $O(n \lg n)$ greedy algorithm for interval scheduling

SCHEDULE-INTERVALS(I) $\quad \triangleright\ I = \{I_i\}, I_i = (s_i, f_i)$

R = Sorted requests in order of finishing times such that $f_i \le f_j$ when $i < j$.
Create an array $S[1\,..\,n]$ with starting times such that $S[i]$ contains s_i.
$A = \{R_1\}$ $\quad \triangleright$ select first interval from sorted list
$f = f_1$
while there are more intervals in S to look at
    do j = first interval for which $s_j \ge f$
        $A \leftarrow A \cup \{j\}$
        $f \leftarrow f_j$
return A

Analysis

- The sorting step in takes $O(n \lg n)$ time.
- Creating the starting time array $S[1\,..\,n]$ takes $O(n)$ time.

An $O(n \lg n)$ greedy algorithm for interval scheduling

```
SCHEDULE-INTERVALS(I)    ▷ I = {I_i}, I_i = (s_i, f_i)
R = Sorted requests in order of finishing times such that f_i ≤ f_j when i < j.
Create an array S[1..n] with starting times such that S[i] contains s_i.
A = {R_1}                              ▷ select first interval from sorted list
f = f_1
while there are more intervals in S to look at
      do j = first interval for which s_j ≥ f
         A ← A ∪ {j}
         f ← f_j
return A
```

Analysis

- The sorting step in takes $O(n \lg n)$ time.
- Creating the starting time array $S[1\,..\,n]$ takes $O(n)$ time.
- The single pass through the array S takes $O(n)$ time

An $O(n \lg n)$ greedy algorithm for interval scheduling

SCHEDULE-INTERVALS(I) $\quad \triangleright\ I = \{I_i\}, I_i = (s_i, f_i)$

R = Sorted requests in order of finishing times such that $f_i \leq f_j$ when $i < j$.
Create an array $S[1\,..\,n]$ with starting times such that $S[i]$ contains s_i.
$A = \{R_1\}$ $\quad \triangleright$ select first interval from sorted list
$f = f_1$
while there are more intervals in S to look at
  do j = first interval for which $s_j \geq f$
    $A \leftarrow A \cup \{j\}$
    $f \leftarrow f_j$
return A

Analysis

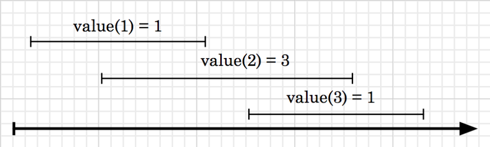
- The sorting step in takes $O(n \lg n)$ time.
- Creating the starting time array $S[1\,..\,n]$ takes $O(n)$ time.
- The single pass through the array S takes $O(n)$ time
- An $\boxed{O(n \lg n)}$ time algorithm for a problem with a natural search space of $O(n2^n)$.

Extension: weighted interval scheduling problem

Definition (Weighted interval scheduling problem)

Given a set of schedules $I = \{I_i\}$, with associated weights $W = \{w_i\}$, find $A \subseteq I$ such that the members of A are non-conflicting *and* the total weight $\sum_{i \in A} w_i$ is maximized.

Example



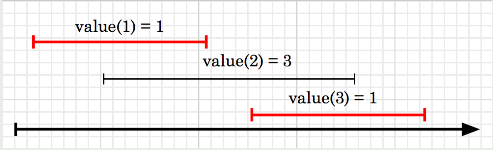
$$|A| = ???,\ \sum_{i \in A} w_i = ???.$$

Extension: weighted interval scheduling problem

Definition (Weighted interval scheduling problem)

Given a set of schedules $I = \{I_i\}$, with associated weights $W = \{w_i\}$, find $A \subseteq I$ such that the members of A are non-conflicting *and* the total weight $\sum_{i \in A} w_i$ is maximized.

Example (using our greedy strategy)



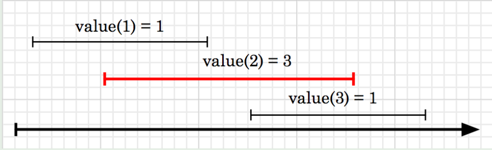
$$|A| = 2,\ \sum_{i \in A} w_i = 2.$$

Extension: weighted interval scheduling problem

Definition (Weighted interval scheduling problem)

Given a set of schedules $I = \{I_i\}$, with associated weights $W = \{w_i\}$, find $A \subseteq I$ such that the members of A are non-conflicting *and* the total weight $\sum_{i \in A} w_i$ is maximized.

Example (using an optimal strategy)



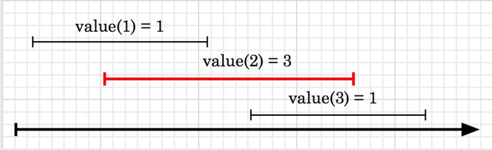
$$|A| = 1, \sum_{i \in A} w_i = 3.$$

Extension: weighted interval scheduling problem

Definition (Weighted interval scheduling problem)

Given a set of schedules $I = \{I_i\}$, with associated weights $W = \{w_i\}$, find $A \subseteq I$ such that the members of A are non-conflicting *and* the total weight $\sum_{i \in A} w_i$ is maximized.

Example (using an optimal strategy)



$$|A| = 1, \sum_{i \in A} w_i = 3.$$

Hmmm...

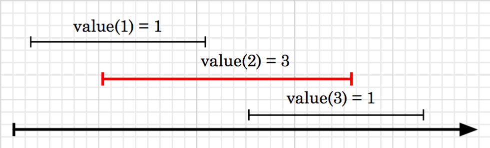
There is no greedy solution for the weighted interval scheduling problem!

Extension: weighted interval scheduling problem

Definition (Weighted interval scheduling problem)

Given a set of schedules $I = \{I_i\}$, with associated weights $W = \{w_i\}$, find $A \subseteq I$ such that the members of A are non-conflicting *and* the total weight $\sum_{i \in A} w_i$ is maximized.

Example (using an optimal strategy)



$$|A| = 1, \sum_{i \in A} w_i = 3.$$

Hmmm...

There is no greedy solution for the weighted interval scheduling problem! Why? (see Greedy Choice property later)

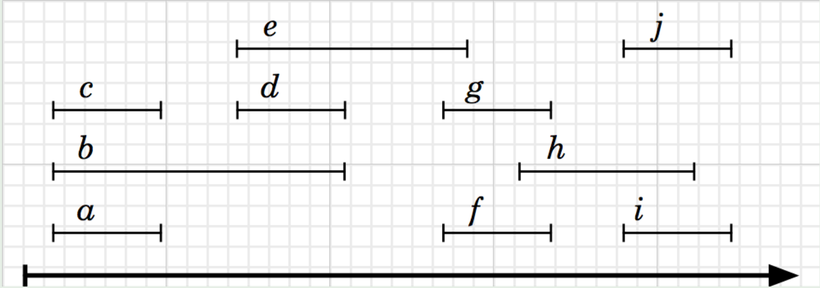
Contents

Scheduling all intervals greedy algorithm

Definition

Given a set of schedules $I = \{I_i\}$, find the minimum number of resources needed to schedule I such that the intervals on each resource are non-conflicting.

Example

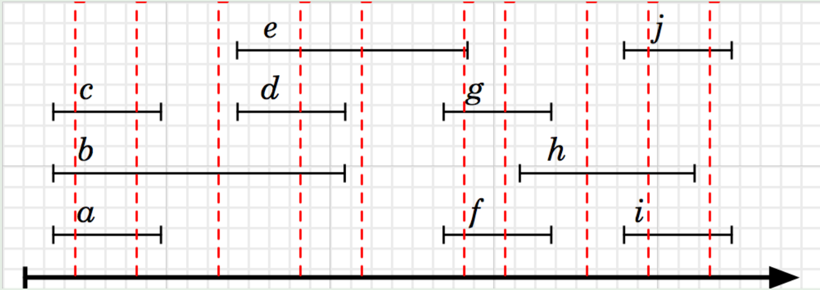


Scheduling all intervals greedy algorithm

Definition

Given a set of schedules $I = \{I_i\}$, find the minimum number of resources needed to schedule I such that the intervals on each resource are non-conflicting.

Example



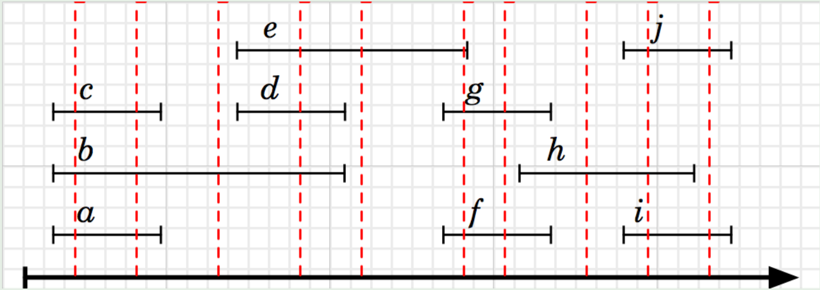
Depth = Maximum number of intervals at any point in time.

Scheduling all intervals greedy algorithm

Definition

Given a set of schedules $I = \{I_i\}$, find the minimum number of resources needed to schedule I such that the intervals on each resource are non-conflicting.

Example



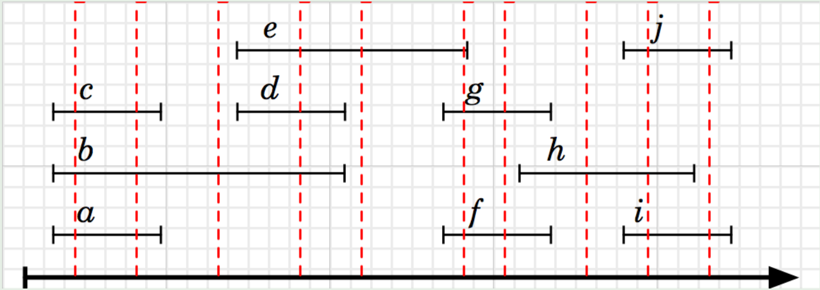
$\text{Depth} = 3$

Scheduling all intervals greedy algorithm

Definition

Given a set of schedules $I = \{I_i\}$, find the minimum number of resources needed to schedule I such that the intervals on each resource are non-conflicting.

Example



Depth $= 3$ $\implies$ Minimum # of resources needed $= 3$

A greedy algorithm for scheduling all intervals

SCHEDULE-INTERVALS(I) $\quad \triangleright\ I = \{I_i\}, I_i = (s_i, f_i)$

R = Sorted requests in order of starting times, breaking ties arbitrarily, such that $s_i \leq s_j$ when $i < j$.
$m \leftarrow 0$ $\triangleright$ the optimal number of resources needed to schedule R
while $R \neq \emptyset$
    do *req* = extract the next element in R
        if there is a resource j with no interval conflicting with *req*
            then schedule interval *req* on resource j
            else
                $m \leftarrow m + 1$ $\quad \triangleright$ allocate a new resource
                schedule interval *req* on resource m

A greedy algorithm for scheduling all intervals

SCHEDULE-INTERVALS(I) $\quad \triangleright\ I = \{I_i\}, I_i = (s_i, f_i)$

$R =$ Sorted requests in order of starting times, breaking ties arbitrarily, such that $s_i \leq s_j$ when $i < j$.
$m \leftarrow 0$ $\triangleright$ the optimal number of resources needed to schedule R
while $R \neq \emptyset$
    do *req* $=$ extract the next element in R
        if there is a resource j with no interval conflicting with *req*
            then schedule interval *req* on resource j
            else
                $m \leftarrow m + 1$ $\quad \triangleright$ allocate a new resource
                schedule interval *req* on resource m

Complexity

$T(n) = O(n \lg n)$.

A greedy algorithm for scheduling all intervals

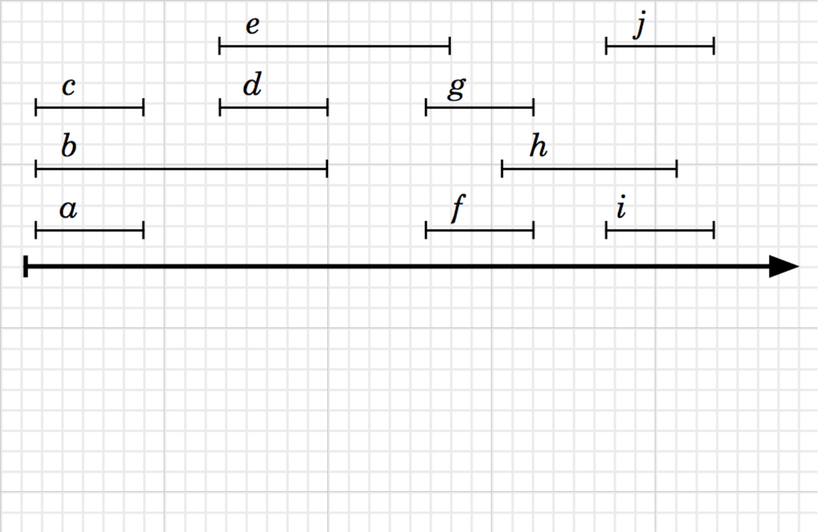
SCHEDULE-INTERVALS(I) $\quad \triangleright\ I = \{I_i\}, I_i = (s_i, f_i)$

R = Sorted requests in order of starting times, breaking ties arbitrarily, such that $s_i \leq s_j$ when $i < j$.
$m \leftarrow 0$ $\triangleright$ the optimal number of resources needed to schedule R
while $R \neq \emptyset$
  do *req* = extract the next element in R
    if there is a resource j with no interval conflicting with *req*
      then schedule interval *req* on resource j
      else
        $m \leftarrow m + 1$ $\quad \triangleright$ allocate a new resource
        schedule interval *req* on resource m

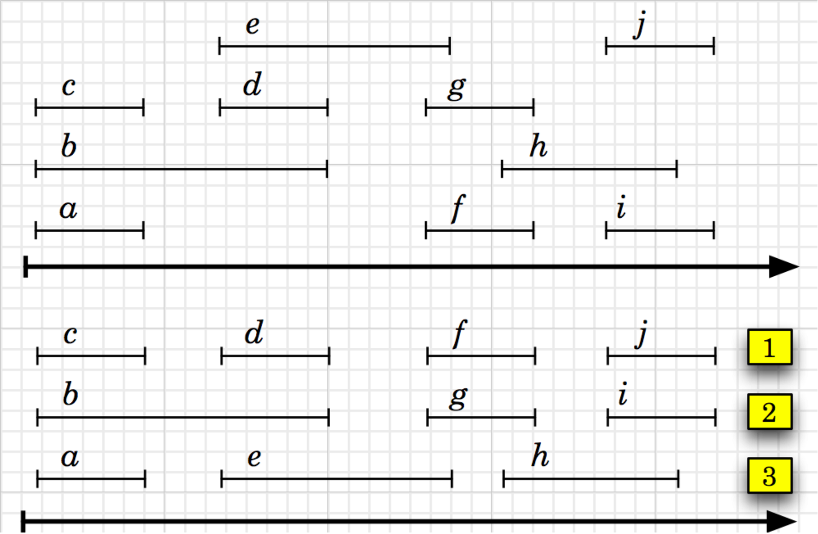
Complexity

$T(n) = O(n \lg n)$. Prove it.

Scheduling all intervals in action



Scheduling all intervals in action



Contents

Fractional knapsack problem

Definition (fractional knapsack problem)

Given a set S of n items, such that each item i has a positive benefit b_i and a positive weight w_i, the goal is to find the maximum-benefit subset that does not exceed a given weight W, allowing for fractional items.

Fractional knapsack problem

Definition (fractional knapsack problem)

Given a set S of n items, such that each item i has a positive benefit b_i and a positive weight w_i, the goal is to find the maximum-benefit subset that does not exceed a given weight W, allowing for fractional items.

- Taking fraction x_i of each item i, such that $0 \leq x_i \leq w_i$ for each $i \in S$, and $\sum_{i \in S} x_i \leq W$.

Fractional knapsack problem

Definition (fractional knapsack problem)

Given a set S of n items, such that each item i has a positive benefit b_i and a positive weight w_i, the goal is to find the maximum-benefit subset that does not exceed a given weight W, allowing for fractional items.

- Taking fraction x_i of each item i, such that $0 \leq x_i \leq w_i$ for each $i \in S$, and $\sum_{i \in S} x_i \leq W$.
- Benefit b_i is accumulated when whole object i is taken. So, for taking fraction x_i of item i is then $b_i(x_i/w_i)$

Fractional knapsack problem

Definition (fractional knapsack problem)

Given a set S of n items, such that each item i has a positive benefit b_i and a positive weight w_i, the goal is to find the maximum-benefit subset that does not exceed a given weight W, allowing for fractional items.

- Taking fraction x_i of each item i, such that $0 \leq x_i \leq w_i$ for each $i \in S$, and $\sum_{i \in S} x_i \leq W$.
- Benefit b_i is accumulated when whole object i is taken. So, for taking fraction x_i of item i is then $b_i(x_i/w_i)$
- Maximum-benefit subset is then maximizing $\sum_{i \in S} b_i(x_i/w_i)$.

Fractional knapsack problem

Key question

- What strategy to use to select the next item (and the amount of it)?

Fractional knapsack problem

Key question

- What strategy to use to select the next item (and the amount of it)?
- At least three different measures one can attempt to optimize when determining which object to select next

Fractional knapsack problem

Key question

- What strategy to use to select the next item (and the amount of it)?
- At least three different measures one can attempt to optimize when determining which object to select next
- Total Profit

Fractional knapsack problem

Key question

- What strategy to use to select the next item (and the amount of it)?
- At least three different measures one can attempt to optimize when determining which object to select next
- Total Profit
- Capacity Used

Fractional knapsack problem

Key question

- What strategy to use to select the next item (and the amount of it)?
- At least three different measures one can attempt to optimize when determining which object to select next
- Total Profit
- Capacity Used
- Ratio of accumulated profit to capacity used

Fractional knapsack problem

Key question

- What strategy to use to select the next item (and the amount of it)?
- At least three different measures one can attempt to optimize when determining which object to select next
- Total Profit
- Capacity Used
- Ratio of accumulated profit to capacity used

Let's see which measure to optimize...

Fractional knapsack problem

Item	Benefit	Weight
A	25	18 kg
B	15	10 kg
C	24	15 kg

Capacity W=20

Fractional knapsack problem

Item	Benefit	Weight
A	25	18 kg
B	15	10 kg
C	24	15 kg

Capacity W=20

Fractional knapsack problem

First Strategy: Greedy method using **Profit** as it's measure. At each step it will choose an object that increases the profit the **most**.

- Capacity $W = 20$

Fractional knapsack problem

First Strategy: Greedy method using **Profit** as it's measure. At each step it will choose an object that increases the profit the **most**.

- Capacity $W = 20$
- A has the maximum benefit 25. So select this. Remaining Capacity: $20 - 18 = 2$

Fractional knapsack problem

First Strategy: Greedy method using **Profit** as it's measure. At each step it will choose an object that increases the profit the **most**.

- Capacity $W = 20$
- A has the maximum benefit 25. So select this. Remaining Capacity: $20 - 18 = 2$
- Then select object C as it has the second maximum benefit.

Fractional knapsack problem

First Strategy: Greedy method using **Profit** as it's measure. At each step it will choose an object that increases the profit the **most**.

- Capacity $W = 20$
- A has the maximum benefit 25. So select this. Remaining Capacity: $20 - 18 = 2$
- Then select object C as it has the second maximum benefit.
- However, C's weight $15 >$ Current Capacity 2.

Fractional knapsack problem

First Strategy: Greedy method using **Profit** as it's measure. At each step it will choose an object that increases the profit the **most**.

- Capacity $W = 20$
- A has the maximum benefit 25. So select this. Remaining Capacity: $20 - 18 = 2$
- Then select object C as it has the second maximum benefit.
- However, C's weight $15 >$ Current Capacity 2.
- So select fraction of C: $x_C = 2$. Remaining Capacity: $2 - 2 = 0$

Fractional knapsack problem

First Strategy: Greedy method using **Profit** as it's measure. At each step it will choose an object that increases the profit the **most**.

- Capacity $W = 20$
- A has the maximum benefit 25. So select this. Remaining Capacity: $20 - 18 = 2$
- Then select object C as it has the second maximum benefit.
- However, C's weight $15 >$ Current Capacity 2.
- So select fraction of C: $x_C = 2$. Remaining Capacity: $2 - 2 = 0$
- Accumulated profit by adding $x_C = 2$ to the knapsack is: $24(2/15){=}3.2$

Fractional knapsack problem

First Strategy: Greedy method using **Profit** as it's measure. At each step it will choose an object that increases the profit the **most**.

- Capacity $W = 20$
- A has the maximum benefit 25. So select this. Remaining Capacity: $20 - 18 = 2$
- Then select object C as it has the second maximum benefit.
- However, C's weight $15 >$ Current Capacity 2.
- So select fraction of C: $x_C = 2$. Remaining Capacity: $2 - 2 = 0$
- Accumulated profit by adding $x_C = 2$ to the knapsack is: 24(2/15)=3.2
- So, finally, total benefit$= 25 + 3.2 = 28.2$

Fractional knapsack problem

- Note here, If we select C as first element and B as second element then,

Fractional knapsack problem

- Note here, If we select C as first element and B as second element then,
- Total benefit$= 24 + 15(5/10) = 31.5$
- So, previous solution is not the optimal one.
- Thus, First Strategy fails.

Fractional knapsack problem

Second Strategy: Greedy method using **Capacity** as it's measure will, at each step choose an object that increases the capacity the **least**.

- B has the least weight 10. So select this. Remaining Capacity: $20 - 10 = 10$

Fractional knapsack problem

Second Strategy: Greedy method using **Capacity** as it's measure will, at each step choose an object that increases the capacity the **least**.

- B has the least weight 10. So select this. Remaining Capacity: $20 - 10 = 10$
- Then select object C as it has the second least weight.

Fractional knapsack problem

Second Strategy: Greedy method using **Capacity** as it's measure will, at each step choose an object that increases the capacity the **least**.

- B has the least weight 10. So select this. Remaining Capacity: $20 - 10 = 10$
- Then select object C as it has the second least weight.
- However, C's weight $15 >$ Current Capacity 10.

Fractional knapsack problem

Second Strategy: Greedy method using **Capacity** as it's measure will, at each step choose an object that increases the capacity the **least**.

- B has the least weight 10. So select this. Remaining Capacity: $20 - 10 = 10$
- Then select object C as it has the second least weight.
- However, C's weight $15 >$ Current Capacity 10.
- So select fraction of C: $x_C = 10$. Remaining Capacity: $10 - 10 = 0$

Fractional knapsack problem

Second Strategy: Greedy method using **Capacity** as it's measure will, at each step choose an object that increases the capacity the **least**.

- B has the least weight 10. So select this. Remaining Capacity: $20 - 10 = 10$
- Then select object C as it has the second least weight.
- However, C's weight $15 >$ Current Capacity 10.
- So select fraction of C: $x_C = 10$. Remaining Capacity: $10 - 10 = 0$
- Accumulated profit by adding $x_C = 10$ to the knapsack is: 24(10/15)=16

Fractional knapsack problem

Second Strategy: Greedy method using **Capacity** as it's measure will, at each step choose an object that increases the capacity the **least**.

- B has the least weight 10. So select this. Remaining Capacity: $20 - 10 = 10$
- Then select object C as it has the second least weight.
- However, C's weight $15 >$ Current Capacity 10.
- So select fraction of C: $x_C = 10$. Remaining Capacity: $10 - 10 = 0$
- Accumulated profit by adding $x_C = 10$ to the knapsack is: 24(10/15)=16
- So, total benefit$= 15 + 16 = 31$.

Fractional knapsack problem

Second Strategy: Greedy method using **Capacity** as it's measure will, at each step choose an object that increases the capacity the **least**.

- B has the least weight 10. So select this. Remaining Capacity: $20 - 10 = 10$
- Then select object C as it has the second least weight.
- However, C's weight $15 >$ Current Capacity 10.
- So select fraction of C: $x_C = 10$. Remaining Capacity: $10 - 10 = 0$
- Accumulated profit by adding $x_C = 10$ to the knapsack is: 24(10/15)=16
- So, total benefit$= 15 + 16 = 31$. Again, it is not the optimal one.

Fractional knapsack problem

Second Strategy: Greedy method using **Capacity** as it's measure will, at each step choose an object that increases the capacity the **least**.

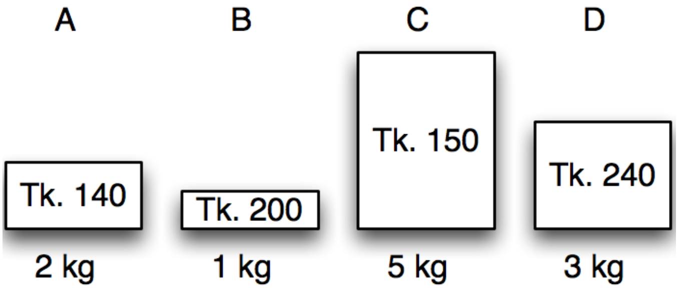
- B has the least weight 10. So select this. Remaining Capacity: $20 - 10 = 10$
- Then select object C as it has the second least weight.
- However, C's weight $15 >$ Current Capacity 10.
- So select fraction of C: $x_C = 10$. Remaining Capacity: $10 - 10 = 0$
- Accumulated profit by adding $x_C = 10$ to the knapsack is: $24(10/15)=16$
- So, total benefit$= 15 + 16 = 31$. Again, it is not the optimal one.
- Thus, second strategy fails.

Fractional knapsack problem

Third Strategy: Strives to achieve a balance between the rate at which profit increases and the rate at which capacity is used.

- At each step, include the object which has the maximum profit per unit of capacity used.
- That means, objects are considered in order of the ratio b_i/w_i.

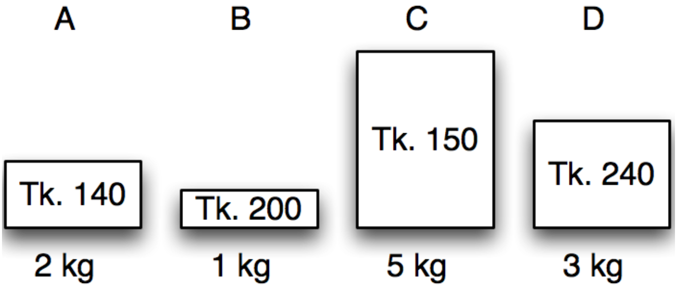
Fractional knapsack in action



Item	Benefit	Weight
A	140	2 kg
B	200	1 kg
C	150	5 kg
D	240	3 kg

Calculate benefit/kg – the value index.

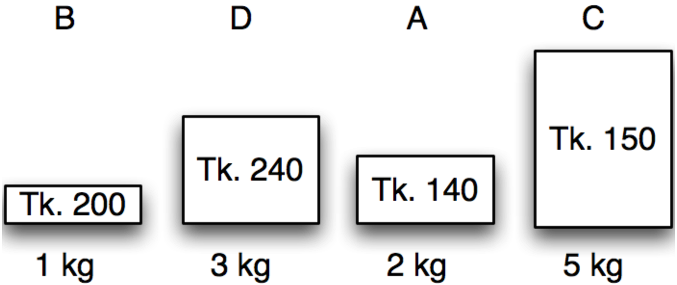
Fractional knapsack in action



Item	Benefit	Weight	Value index
A	140	2 kg	70
B	200	1 kg	200
C	150	5 kg	30
D	240	3 kg	80

Sort by non-increasing value index.

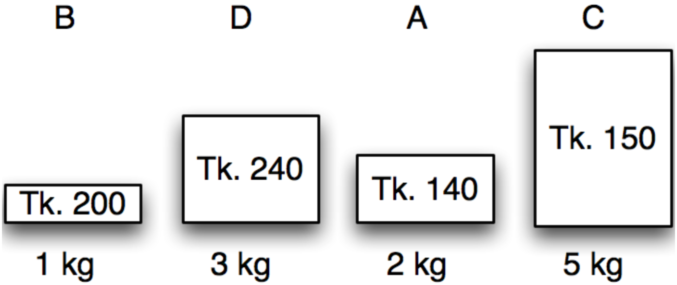
Fractional knapsack in action



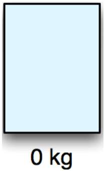
Item	Benefit	Weight	Value index
B	200	1 kg	200
D	240	3 kg	80
A	140	2 kg	70
C	150	5 kg	30

Maximum weight: 5 kg

Fractional knapsack in action

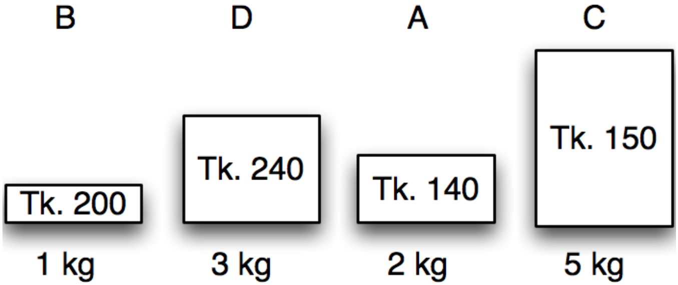


Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	0 kg
D	240	3 kg	80	0 kg
A	140	2 kg	70	0 kg
C	150	5 kg	30	0 kg

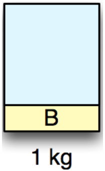


Maximum weight: 5 kg   Remaining: 5 kg   Benefit: 0 kg

Fractional knapsack in action

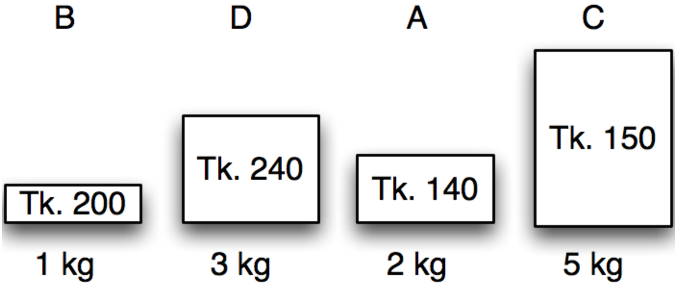


Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	0 kg
A	140	2 kg	70	0 kg
C	150	5 kg	30	0 kg

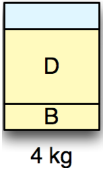


Maximum weight: 5 kg Remaining: 4 kg Benefit: 200 kg

Fractional knapsack in action

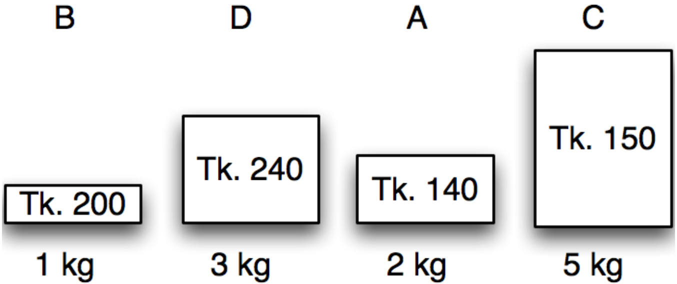


Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	3 kg
A	140	2 kg	70	0 kg
C	150	5 kg	30	0 kg

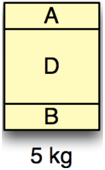


Maximum weight: 5 kg Remaining: 1 kg Benefit: 440 kg

Fractional knapsack in action

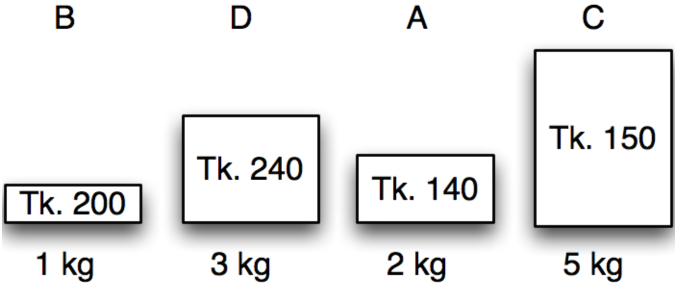


Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	3 kg
A	140	2 kg	70	1 kg
C	150	5 kg	30	0 kg

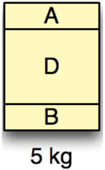


Maximum weight: 5 kg Remaining: 0 kg Benefit: 510 kg

Fractional knapsack in action



Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	3 kg
A	140	2 kg	70	1 kg
C	150	5 kg	30	0 kg



Maximum weight: 5 kg Remaining: 0 kg Benefit: 510 kg

Fractional knapsack greedy algorithm

FRACTIONAL-KNAPSACK(S, W) $\quad \triangleright\ S = \{(w_i, b_i)\}$

for each item $i \in S$
    do $x_i \leftarrow 0$ $\quad \triangleright$ amount of item i chosen $(0 \leq x \leq w_i)$
        $v_i \leftarrow b_i / w_i$ $\quad \triangleright$ compute *value index*
Sort the items in non-increasing order of the ratio b_i / w_i
$w \leftarrow 0$
while $w < W$
    do i = extract from S the item with highest value index
        $\triangleright$ greedy choice
        if $w + w_i \leq W$
            then $x_i = w_i$
            else $x_i = W - w$ $\triangleright$ fill up the remaining with i
        $w \leftarrow w + x_i$
return x $\quad \triangleright\ x_i$ contains amount of item i chosen

Fractional knapsack greedy algorithm

FRACTIONAL-KNAPSACK(S, W) $\quad \triangleright\ S = \{(w_i, b_i)\}$

for each item $i \in S$
    do $x_i \leftarrow 0$ $\quad \triangleright$ amount of item i chosen $(0 \leq x \leq w_i)$
        $v_i \leftarrow b_i / w_i$ $\quad \triangleright$ compute *value index*
Sort the items in non-increasing order of the ratio b_i / w_i
$w \leftarrow 0$
while $w < W$
    do $i =$ extract from S the item with highest value index
        $\triangleright$ greedy choice
        if $w + w_i \leq W$
            then $x_i = w_i$
            else $x_i = W - w$ $\triangleright$ fill up the remaining with i
        $w \leftarrow w + x_i$
return x $\quad \triangleright\ x_i$ contains amount of item i chosen

Complexity

$T(n) = O(n \lg n)$.

Fractional knapsack greedy algorithm

```
FRACTIONAL-KNAPSACK(S, W)    ▷ S = {(w_i, b_i)}
for each item i ∈ S
      do x_i ← 0        ▷ amount of item i chosen (0 ≤ x ≤ w_i)
         v_i ← b_i/w_i                 ▷ compute value index
Sort the items in non-increasing order of the ratio b_i/w_i
w ← 0
while w < W
      do i = extract from S the item with highest value index
            ▷ greedy choice
         if w + w_i ≤ W
           then x_i = w_i
           else  x_i = W − w ▷ fill up the remaining with i
         w ← w + x_i
return x      ▷ x_i contains amount of item i chosen
```

Complexity

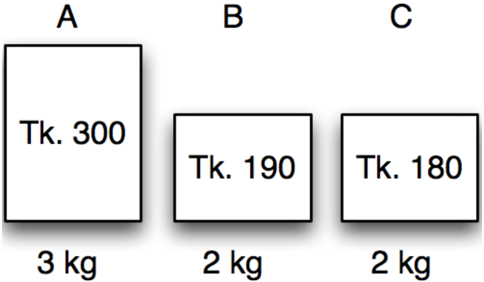
$T(n) = O(n \lg n)$. Prove it.

Extension: 0/1 knapsack problem

Exactly the same as the Fractional Knapsack Problem, except that fractional quantities are not allowed.

Extension: 0/1 knapsack problem

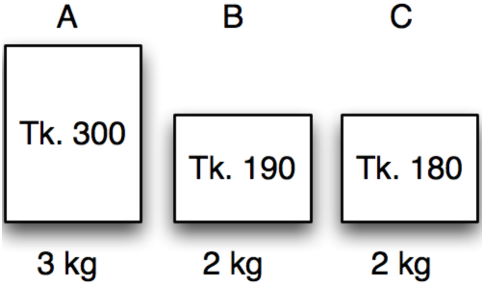
Exactly the same as the Fractional Knapsack Problem, except that fractional quantities are not allowed.



Maximum weight: 4 kg

Extension: 0/1 knapsack problem

Exactly the same as the Fractional Knapsack Problem, except that fractional quantities are not allowed.



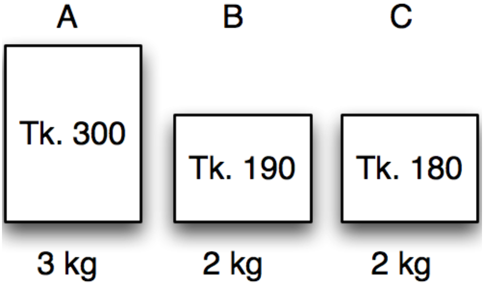
Maximum weight: 4 kg

Greedy solution: item A

Benefit: 300

Extension: 0/1 knapsack problem

Exactly the same as the Fractional Knapsack Problem, except that fractional quantities are not allowed.



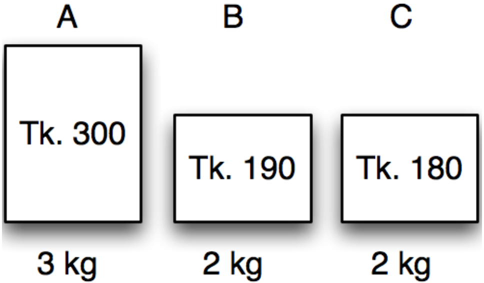
Maximum weight: $\boxed{4 \text{ kg}}$

Greedy solution: item A — Benefit: $\boxed{300}$

Optimal solution: items B and C — Benefit: $\boxed{370}$

Extension: 0/1 knapsack problem

Exactly the same as the Fractional Knapsack Problem, except that fractional quantities are not allowed.



Maximum weight: $\boxed{4 \text{ kg}}$

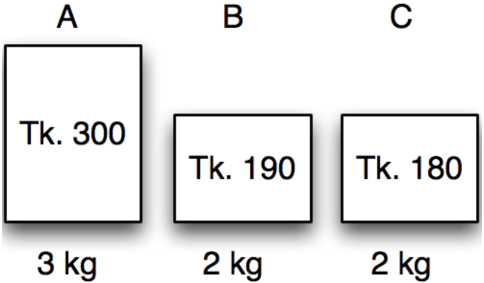
Greedy solution: item A — Benefit: $\boxed{300}$

Optimal solution: items B and C — Benefit: $\boxed{370}$

The 0/1 Knapsack Problem does not have a greedy solution!

Extension: 0/1 knapsack problem

Exactly the same as the Fractional Knapsack Problem, except that fractional quantities are not allowed.



Maximum weight: $\boxed{4 \text{ kg}}$

Greedy solution: item A Benefit: $\boxed{300}$

Optimal solution: items B and C Benefit: $\boxed{370}$

The 0/1 Knapsack Problem does not have a greedy solution! Why?

Contents

Coin changing problem

Definition

Given coin denominations in $\{C\}$, make change for a given amount A with the minimum number of coins.

Coin changing problem

Definition

Given coin denominations in $\{C\}$, make change for a given amount A with the minimum number of coins.

Example

Coin denominations, $C = \{25, 10, 5, 1\}$ Amount to change, $A = 73$

Coin changing problem

Definition

Given coin denominations in $\{C\}$, make change for a given amount A with the minimum number of coins.

Example

Coin denominations, $C = \{25, 10, 5, 1\}$ Amount to change, $A = 73$

1. Choose 2 25 coins, so remaining is $73 - 2 * 25 = 23$

Coin changing problem

Definition

Given coin denominations in $\{C\}$, make change for a given amount A with the minimum number of coins.

Example

Coin denominations, $C = \{25, 10, 5, 1\}$ Amount to change, $A = 73$

1. Choose 2 25 coins, so remaining is $73 - 2 * 25 = 23$
2. Choose 2 10 coins, so remaining is $23 - 2 * 10 = 3$

Coin changing problem

Definition

Given coin denominations in $\{C\}$, make change for a given amount A with the minimum number of coins.

Example

Coin denominations, $C = \{25, 10, 5, 1\}$ Amount to change, $A = 73$

1. Choose 2 25 coins, so remaining is $73 - 2 * 25 = 23$
2. Choose 2 10 coins, so remaining is $23 - 2 * 10 = 3$
3. Choose 0 5 coins, so remaining is 3

Coin changing problem

Definition

Given coin denominations in $\{C\}$, make change for a given amount A with the minimum number of coins.

Example

Coin denominations, $C = \{25, 10, 5, 1\}$ Amount to change, $A = 73$

1. Choose 2 25 coins, so remaining is $73 - 2 * 25 = 23$
2. Choose 2 10 coins, so remaining is $23 - 2 * 10 = 3$
3. Choose 0 5 coins, so remaining is 3
4. Choose 3 1 coins, so remaining is $3 - 1 * 3 = 0$

Coin changing problem

Definition

Given coin denominations in $\{C\}$, make change for a given amount A with the minimum number of coins.

Example

Coin denominations, $C = \{25, 10, 5, 1\}$ Amount to change, $A = 73$

1. Choose 2 25 coins, so remaining is $73 - 2 * 25 = 23$
2. Choose 2 10 coins, so remaining is $23 - 2 * 10 = 3$
3. Choose 0 5 coins, so remaining is 3
4. Choose 3 1 coins, so remaining is $3 - 1 * 3 = 0$

Solution (and it's optimal): $2 \times 25 + 2 \times 10 + 3 \times 1 = 7$ coins.

Coin changing problem

Definition

Given coin denominations in $\{C\}$, make change for a given amount A with the minimum number of coins.

Example

Coin denominations, $C = \{25, 10, 5, 1\}$ Amount to change, $A = 73$

1. Choose 2 25 coins, so remaining is $73 - 2 * 25 = 23$
2. Choose 2 10 coins, so remaining is $23 - 2 * 10 = 3$
3. Choose 0 5 coins, so remaining is 3
4. Choose 3 1 coins, so remaining is $3 - 1 * 3 = 0$

Solution (and it's optimal): $2 \times 25 + 2 \times 10 + 3 \times 1 = 7$ coins.

Key question

Does a greedy approach always produce the optimal solution?

Coin changing problem (continued)

Coin denominations, $C = \{12, 5, 1\}$ Amount to change, $A = 15$

Coin changing problem (continued)

Coin denominations, $C = \{12, 5, 1\}$ Amount to change, $A = 15$

Example (using greedy strategy)

Coin changing problem (continued)

Coin denominations, $C = \{12, 5, 1\}$ Amount to change, $A = 15$

Example (using greedy strategy)

1. Choose 1 12 coins, so remaining is $15 - 1 * 12 = 3$

Coin changing problem (continued)

Coin denominations, $C = \{12, 5, 1\}$ Amount to change, $A = 15$

Example (using greedy strategy)

1. Choose 1 12 coins, so remaining is $15 - 1 * 12 = 3$
2. Choose 3 1 coins, so remaining is $3 - 1 * 3 = 0$

Coin changing problem (continued)

Coin denominations, $C = \{12, 5, 1\}$ Amount to change, $A = 15$

Example (using greedy strategy)

1. Choose 1 12 coins, so remaining is $15 - 1 * 12 = 3$
2. Choose 3 1 coins, so remaining is $3 - 1 * 3 = 0$

Solution: 4 coins.

Coin changing problem (continued)

Coin denominations, $C = \{12, 5, 1\}$ Amount to change, $A = 15$

Example (using greedy strategy)

1. Choose 1 12 coins, so remaining is $15 - 1 * 12 = 3$
2. Choose 3 1 coins, so remaining is $3 - 1 * 3 = 0$

Solution: 4 coins.

Example (using optimal strategy)

Coin changing problem (continued)

Coin denominations, $C = \{12, 5, 1\}$ Amount to change, $A = 15$

Example (using greedy strategy)

1. Choose 1 12 coins, so remaining is $15 - 1 * 12 = 3$
2. Choose 3 1 coins, so remaining is $3 - 1 * 3 = 0$

Solution: 4 coins.

Example (using optimal strategy)

1. Choose 0 12 coins, so remaining is 15

Coin changing problem (continued)

Coin denominations, $C = \{12, 5, 1\}$ Amount to change, $A = 15$

Example (using greedy strategy)

1. Choose 1 12 coins, so remaining is $15 - 1 * 12 = 3$
2. Choose 3 1 coins, so remaining is $3 - 1 * 3 = 0$

Solution: 4 coins.

Example (using optimal strategy)

1. Choose 0 12 coins, so remaining is 15
2. Choose 3 5 coins, so remaining is $15 - 3 * 5 = 0$

Coin changing problem (continued)

Coin denominations, $C = \{12, 5, 1\}$ Amount to change, $A = 15$

Example (using greedy strategy)

1. Choose 1 12 coins, so remaining is $15 - 1 * 12 = 3$
2. Choose 3 1 coins, so remaining is $3 - 1 * 3 = 0$

Solution: 4 coins.

Example (using optimal strategy)

1. Choose 0 12 coins, so remaining is 15
2. Choose 3 5 coins, so remaining is $15 - 3 * 5 = 0$

Solution: 3 coins.

Coin changing problem (continued)

Coin denominations, $C = \{12, 5, 1\}$ Amount to change, $A = 15$

Example (using greedy strategy)

1. Choose 1 12 coins, so remaining is $15 - 1 * 12 = 3$
2. Choose 3 1 coins, so remaining is $3 - 1 * 3 = 0$

Solution: 4 coins.

Example (using optimal strategy)

1. Choose 0 12 coins, so remaining is 15
2. Choose 3 5 coins, so remaining is $15 - 3 * 5 = 0$

Solution: 3 coins.

Key observation

Correctness depends on the choice of coins, so greedy strategy does not provide a general solution to this problem!

Contents

Problem types solved by greedy algorithms

- There is no general of knowing whether a problem can be solved by a greedy algorithm.

Problem types solved by greedy algorithms

- There is no general of knowing whether a problem can be solved by a greedy algorithm.
- If a problem has the following properties, then it's likely to have a greedy solution.

Problem types solved by greedy algorithms

- There is no general of knowing whether a problem can be solved by a greedy algorithm.
- If a problem has the following properties, then it's likely to have a greedy solution.

  Greedy choice property: If the global optimal solution can be reached by making locally optimal choices, then it has the greedy choice property.

Problem types solved by greedy algorithms

- There is no general of knowing whether a problem can be solved by a greedy algorithm.
- If a problem has the following properties, then it's likely to have a greedy solution.

  Greedy choice property: If the global optimal solution can be reached by making locally optimal choices, then it has the greedy choice property.

  Subproblem optimality: If the optimal solution to the entire problem contain optimal solution to the subproblems, then it has the subproblem optimality property.

Conclusion

- Greedy algorithms often lead to polynomial time-solution for an exponential-time problem.

Conclusion

- Greedy algorithms often lead to polynomial time-solution for an exponential-time problem.
- However, not all optimization problems can be solved by the greedy strategy.

Conclusion

- Greedy algorithms often lead to polynomial time-solution for an exponential-time problem.
- However, not all optimization problems can be solved by the greedy strategy.
- The problem must have at least the following properties:
  1. Greedy choice property

Conclusion

- Greedy algorithms often lead to polynomial time-solution for an exponential-time problem.
- However, not all optimization problems can be solved by the greedy strategy.
- The problem must have at least the following properties:
  1. Greedy choice property
  2. Subproblem optimality

Conclusion

- Greedy algorithms often lead to polynomial time-solution for an exponential-time problem.
- However, not all optimization problems can be solved by the greedy strategy.
- The problem must have at least the following properties:
  1. Greedy choice property
  2. Subproblem optimality
- The algorithm must be rigorously proven to be correct!

Conclusion

- Greedy algorithms often lead to polynomial time-solution for an exponential-time problem.
- However, not all optimization problems can be solved by the greedy strategy.
- The problem must have at least the following properties:
  1. Greedy choice property
  2. Subproblem optimality
- The algorithm must be rigorously proven to be correct!
- Except for a few select problems, it is far better to use Dynamic Programming to solve such optimization problems.

Conclusion

- Greedy algorithms often lead to polynomial time-solution for an exponential-time problem.
- However, not all optimization problems can be solved by the greedy strategy.
- The problem must have at least the following properties:
  1. Greedy choice property
  2. Subproblem optimality
- The algorithm must be rigorously proven to be correct!
- Except for a few select problems, it is far better to use Dynamic Programming to solve such optimization problems.
- So why study greedy algorithms?

Conclusion

- Greedy algorithms often lead to polynomial time-solution for an exponential-time problem.
- However, not all optimization problems can be solved by the greedy strategy.
- The problem must have at least the following properties:
  1. Greedy choice property
  2. Subproblem optimality
- The algorithm must be rigorously proven to be correct!
- Except for a few select problems, it is far better to use Dynamic Programming to solve such optimization problems.
- So why study greedy algorithms? Because there are very efficient provably correct greedy algorithms for many common problems (wait till we study graph algorithms).