Imperial College London

CO202 – Software Engineering – Algorithms

Greedy Algorithms

Ben Glocker
Huxley Building, Room 377
b.glocker@imperial.ac.uk

Algorithm Design

How to design an (efficient) algorithm?

Structure the problem!*

Make use of **algorithmic schemes/design paradigms**

- Incremental Approach
- Divide and Conquer (previous week)
- Dynamic Programming (last week)
- **Greedy Algorithms** (this week)

*Take some time and think!

Greedy Strategy

- Applied to optimisation problems (similar to DP)
- When there is a choice to make, always make the one that looks best at the moment: be **greedy***
- This strategy does not always yield optimal solutions, but for many problems it does and often greedy is efficient
- Wide range of applications
  - Dijsktra's shortest path algorithm from single source
  - Minimum spanning tree algorithm
  - Approximate solutions (e.g. traveling salesman problem)

*without looking ahead!

Example: Activity-Selection Problem

Task: Schedule several competing activities that require exclusive use of a common resource

Goal: Select a maximum-size set of mutually compatible activities

Example: Activity-Selection Problem (cont’d)

Suppose we have a set of n proposed **activities**

$$S = \{a_1, a_2, \dots, a_n\}$$

- Each activity a_i has a **start time** s_i and a **finish time** f_i where $0 \leq s_i < f_i < \infty$
- All activities wish to use the same resource
- Activities a_i and a_j are **compatible** if $s_i \geq f_j$ or $s_j \geq f_i$

Assume that activities are sorted in increasing order of finish time:

$$f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n$$

Example: Activity-Selection Problem (cont'd)

Consider the following set

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

What about the following subset:

$$\{a_3, a_9, a_{11}\}$$

- Mutually compatible?
- Largest possible subset?

Example: Activity-Selection Problem (cont'd)

Consider the following set

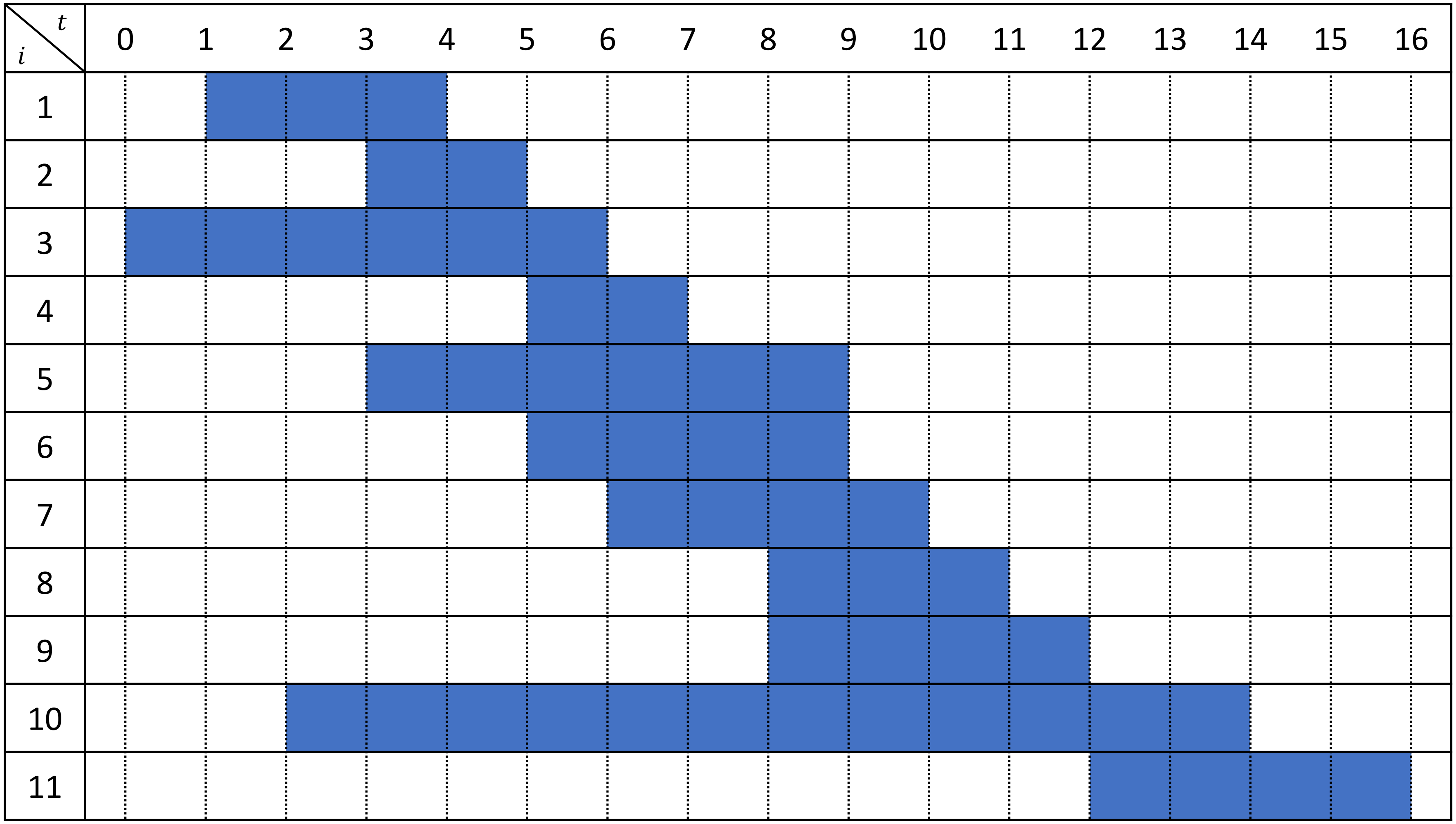
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

What about the following subset:

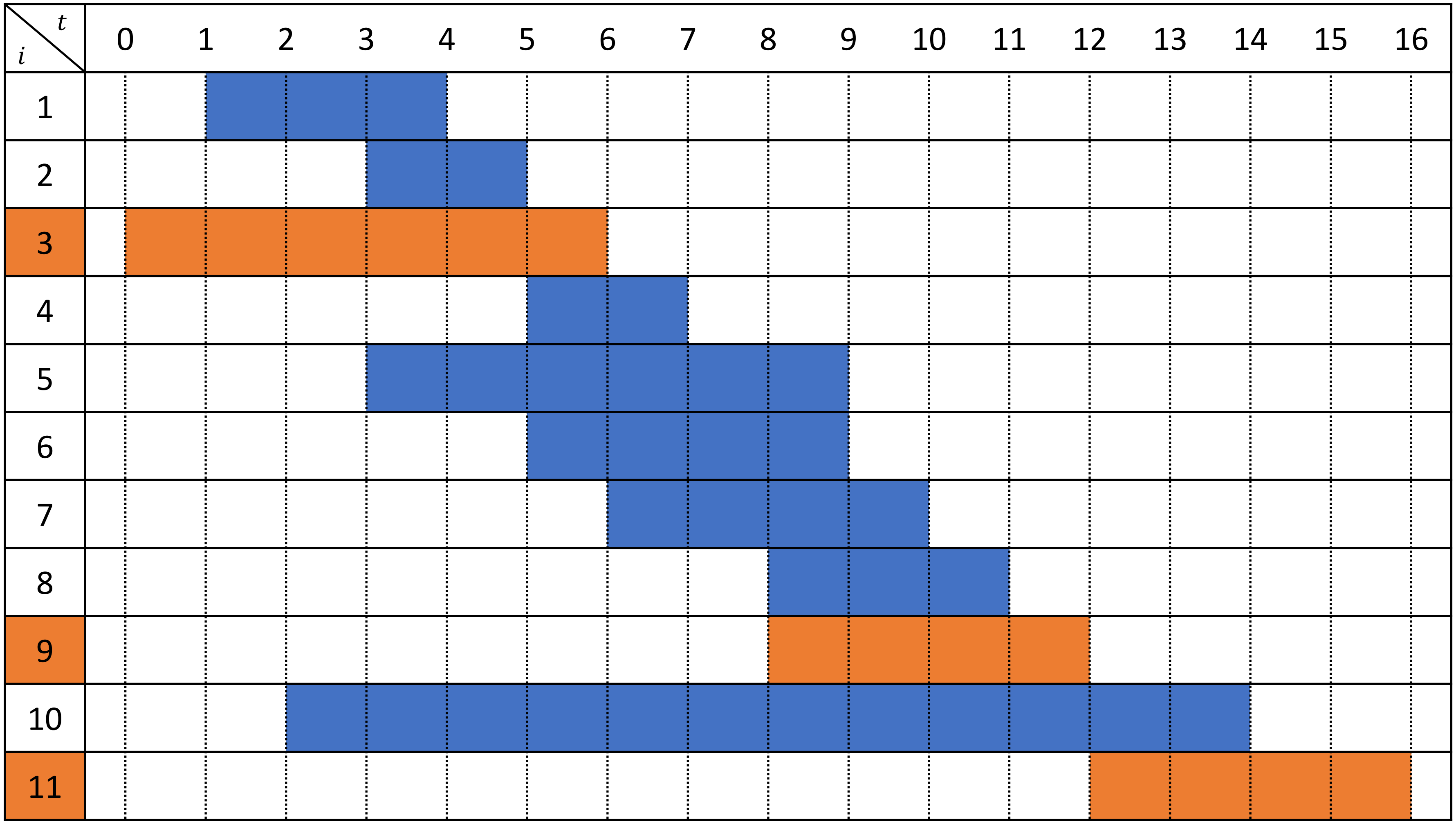
$$\{a_3, a_9, a_{11}\}$$

- Mutually compatible? Yes!
- Largest possible subset? No! For example $\{a_2, a_4, a_9, a_{11}\}$ is larger

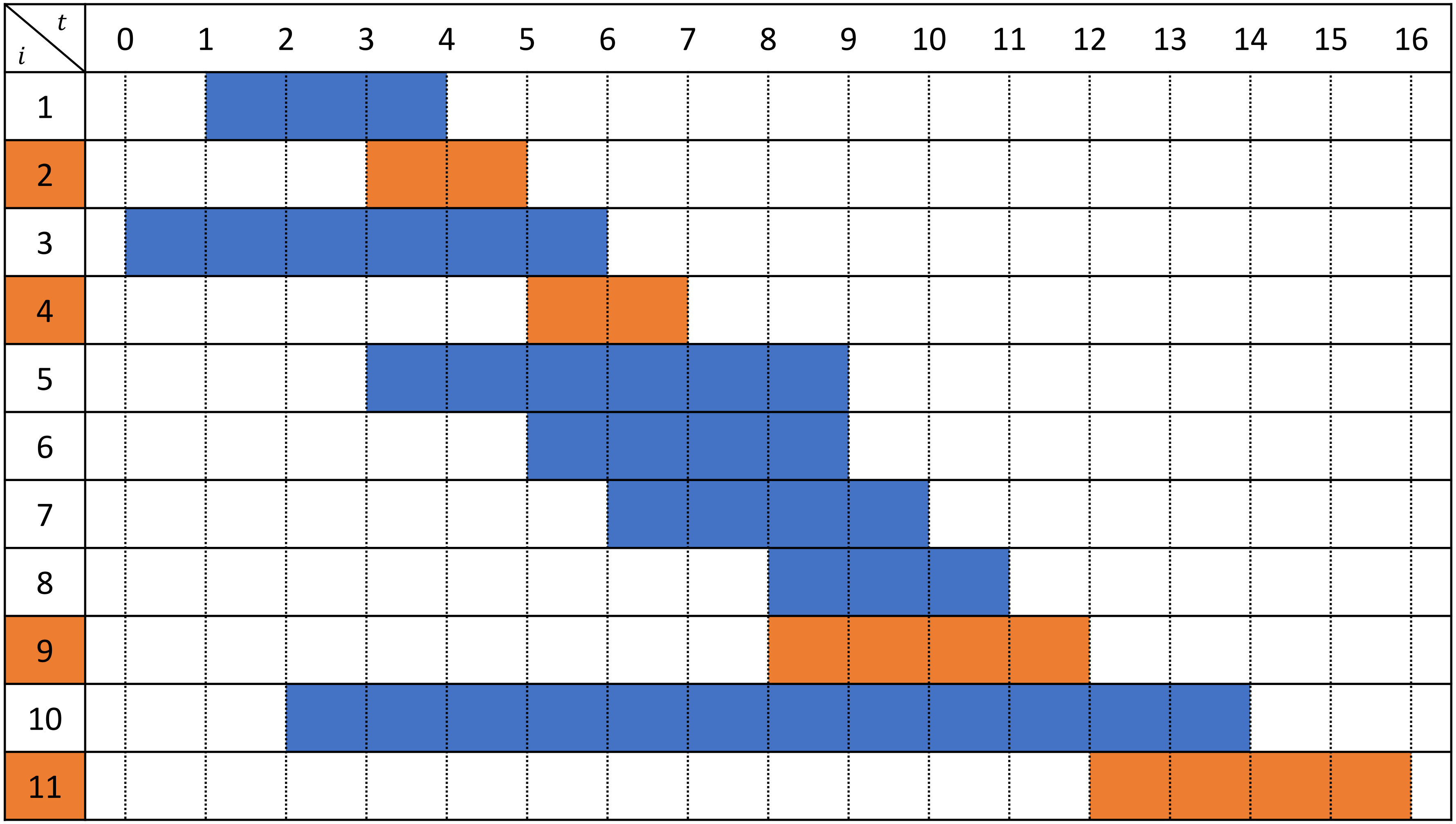
Example: Activity-Selection Problem (cont'd)



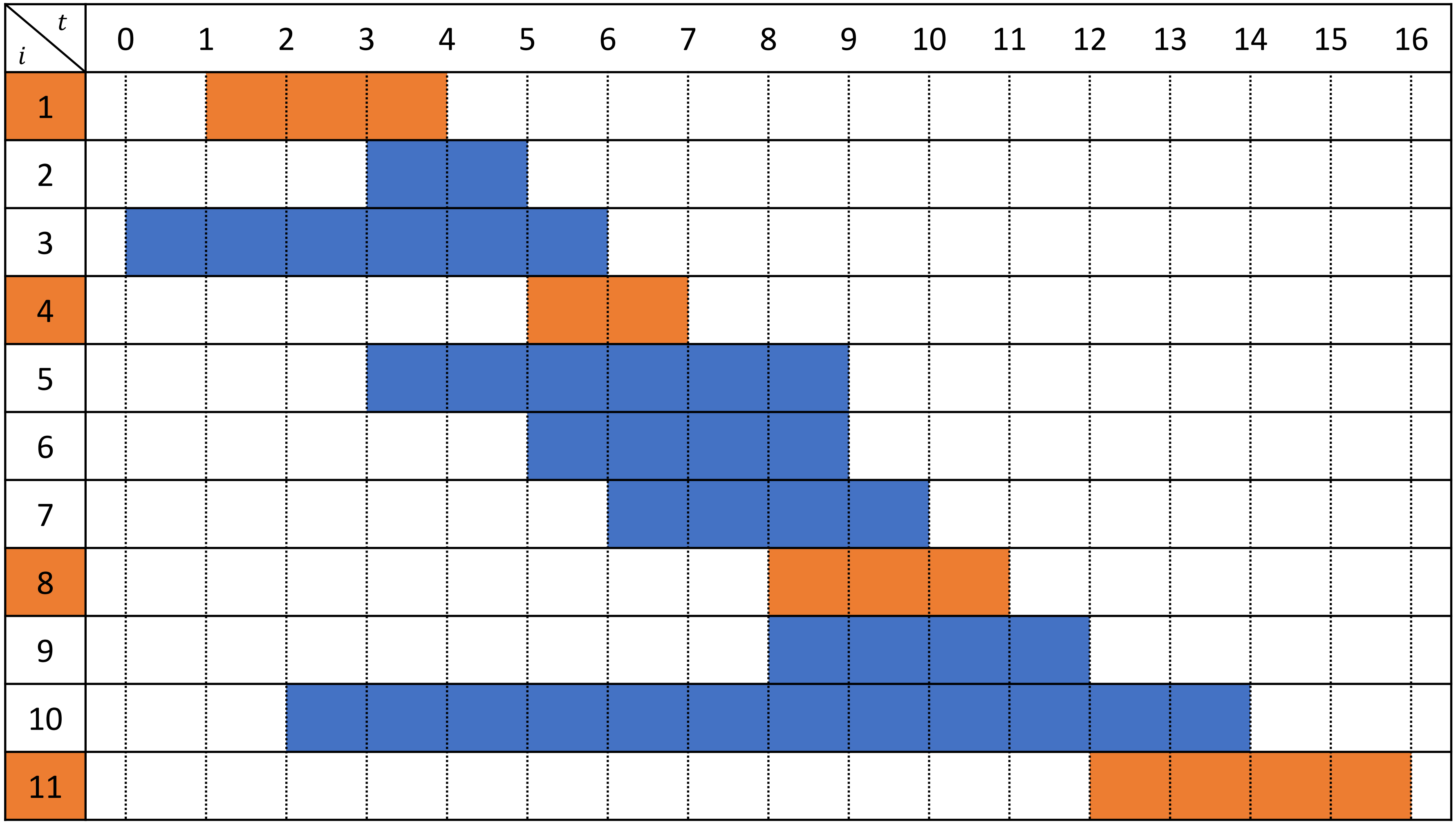
Example: Activity-Selection Problem (cont’d)



Example: Activity-Selection Problem (cont’d)



Example: Activity-Selection Problem (cont’d)



Optimal Substructure of Activity-Selection Problem

Let us denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts

$$S_{ij} = \{a_k \in S \colon f_i \leq s_k < f_k \leq s_j\}$$

- Assuming the maximum subset is A_{ij} including some activity a_k then there are **two subproblems** for sets S_{ik} and S_{kj}
- Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$ so we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
- The maximum size is then $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$
- Optimal solution A_{ij} includes optimal solutions to subproblems for S_{ik} and S_{kj}
- Entire space of activities is $S = S_{0(n+1)}$ with $f_0 = 0$ and $s_{n+1} = \infty$

Solve with Dynamic Programming?

DP for Activity-Selection Problem

Denote the size of an optimal solution for S_{ij} by $c[i,j]$ which yields the recurrence $c[i,j] = c[i,k] + c[k,j] + 1$

But we do not know activity a_k, so we need to examine all activities in S_{ij}:

$$c[i,j] = \begin{cases} 0, & \text{if } S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\}, & \text{if } S_{ij} \neq \emptyset \end{cases}$$

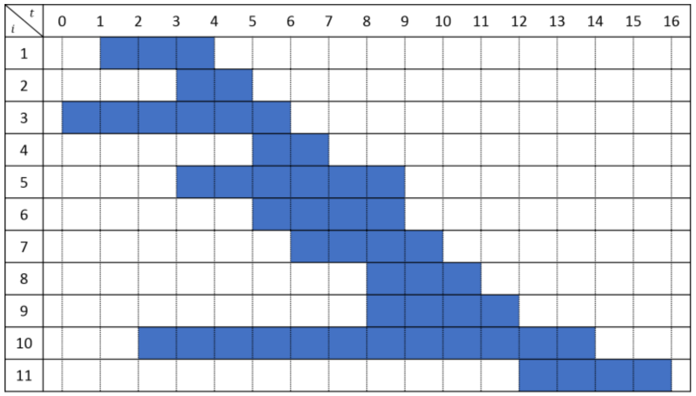
Implement either as
“top-down with memoization” or “bottom-up”!

The Greedy Choice

$$c[i,j] = \begin{cases} 0, & \text{if } S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\}, & \text{if } S_{ij} \neq \emptyset \end{cases}$$

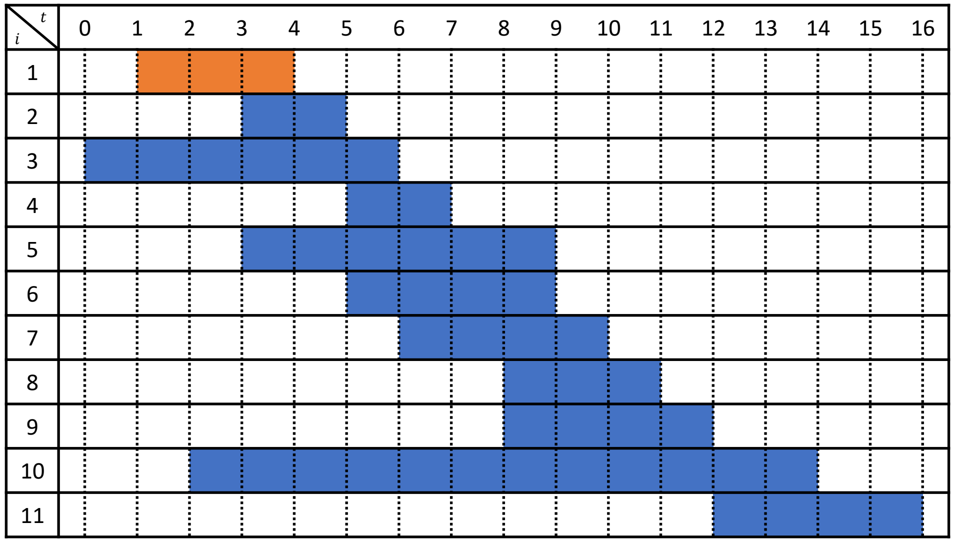
- Can we avoid to consider all possible choices?
- Can we just pick the activity that leaves the resource available for as many other activities as possible?

Which one should we pick?



The Greedy Choice (cont'd)

Choose the activity a_k with the earliest finish time

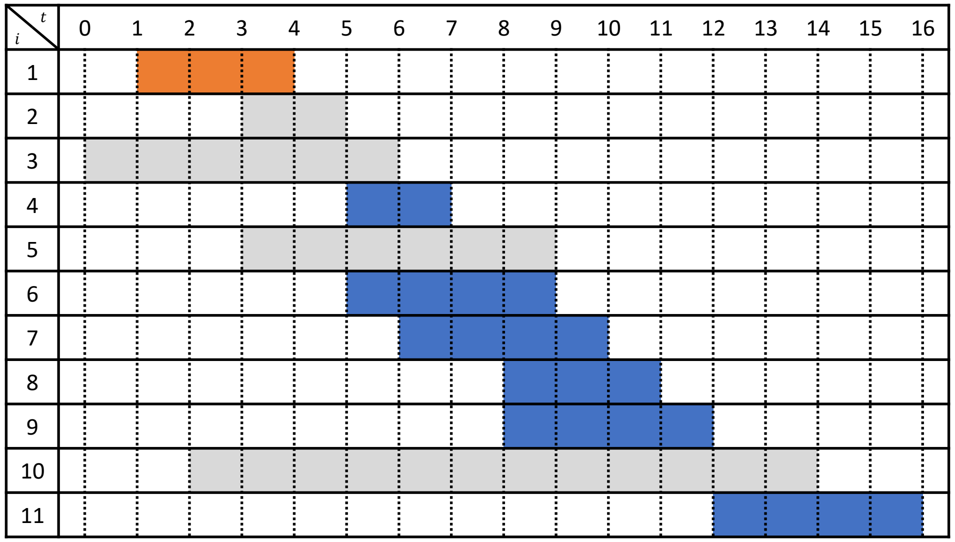


The Greedy Choice (cont'd)

Choose the activity a_k with the earliest finish time

- Only one subproblem to solve

$$S_k = \{a_i \in S: s_i \geq f_k\}$$



Is this strategy yielding optimal solutions?

Optimal Greedy Choice

Theorem:

Consider $S_k \neq \emptyset$, and let $a_m \in S_k$ with $f_m = \min \{f_k : a_k \in S_k\}$.

Then a_m is included in some maximum-size subset A_k of mutually compatible activities of S_k.

Proof:

Let a_j be the activity in A_k with earliest finish time.

- If $a_j = a_m$, we are done.
- If $a_j \neq a_m$, let $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k with substituting a_m for a_j. Activities in A'_k are disjoint, because activities in A_k are disjoint and a_j is the first one to finish in A_k, and $f_m \leq f_j$.
  Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m. ■

DP vs Greedy for Activity-Selection

- Comparing the number of subproblems and choices

	DP	Greedy
Number of subproblems	2 S_{ik} and S_{kj}	1 S_k
Number of choices	$j - i - 1$	1

Recursive Greedy Algorithm

```
RECURSIVE-ACTIVITY-SELECTOR(s,f,k,n)
 1: m = k+1
 2: # find the compatible activity in $S_k$ to finish first
 3: while m ≤ n and s[m] < f[k]
 4:     m = m+1
 5: if m ≤ n
 6:     return {$a_m$} ∪ RECURSIVE-ACTIVITY-SELECTOR(s,f,m,n)
 7: else
 8:     return ∅
```

We add the fictitious activity a_0 with $f_0 = 0$, so S_0 is the subproblem over the entire space of activities S.

The initial call is then `RECURSIVE-ACTIVITY-SELECTOR(s,f,0,n)`

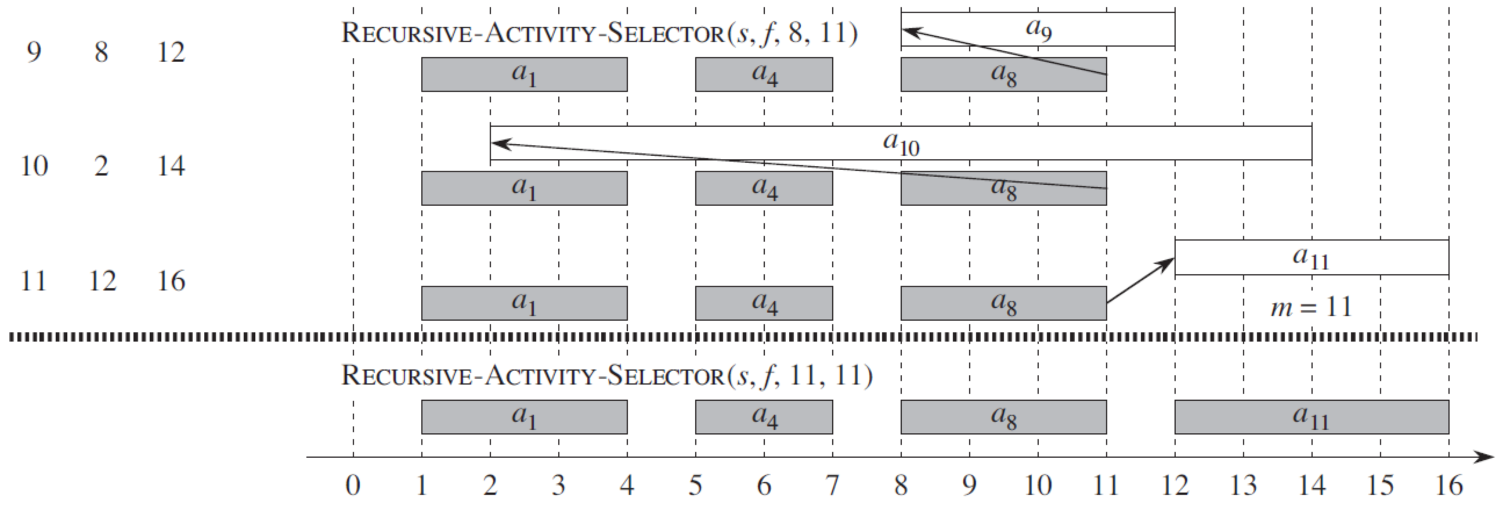
What is the running time?

Operations of the Recursive Algorithm

k	s_k	f_k
0	–	0

[Cormen] p.420

Operations of the Recursive Algorithm (cont'd)



[Cormen] p.420

Incremental Greedy Algorithm

The recursive algorithm is almost "tail recursive". This makes it easy to convert the recursive procedure to an incremental one.

```
GREEDY-ACTIVITY-SELECTOR(s,f)
n = s.length
A = {$a_1$}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {$a_m$}
        k = m
return A
```

Running time is $\Theta(n)$

Elements of the Greedy Strategy

The process to develop a greedy algorithm for the activity-selection problem involved the following steps:

1. Determine the optimal substructure of the problem
2. Develop a recursive solution
3. Show that with a greedy choice only one subproblem remains
4. Prove that it is always safe to make the greedy choice
5. Develop a recursive algorithm implementing the greedy strategy
6. Convert the recursive algorithm to an incremental one

Design of Greedy Algorithms

1. Cast the optimization problem as one in which making a choice leaves only one subproblem to solve
2. Prove the optimality of the solution when making greedy choices
3. Demonstrate optimal substructure

Note: Beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution.

Key Ingredients for Greedy Algorithms

- Optimal substructure (as in DP)
  Optimal solutions contain optimal solutions to subproblems

- Greedy-choice property
  A globally optimal solution is obtained through locally optimal (greedy) choices

Dynamic Programming vs Greedy

Both strategies exploit optimal substructure, however…

- In DP the choices depend on the solutions to subproblems. We typically solve a problem bottom-up (or top-down with memoization)
- DP is solving subproblems before making any choices

- In a greedy algorithm the choice may depend on previous choices, but cannot depend on “future” choices or subproblems
- A greedy algorithm makes its first choice before solving any subproblems

Example: The Knapsack Problem

0-1 Knapsack Problem

A thief robbing a store finds n items:

- The ith item is worth v_i dollars and weights w_i pounds
- The thief's knapsack can carry at most K pounds
- v_i, w_i, and K are integers
- Items must be taken entirely or left behind

Fractional Knapsack Problem

The setup is the same, but the thief can take fractions of items, rather than having to make a binary decision.

Optimal Substructure of the Knapsack Problem

Consider the most valuable load that weighs at most K.

0-1 Problem:

If we remove item j from this load, the remaining load must be the most valuable load weighing at most $K - w_j$ that the thief can take from the $n - 1$ original items.

Fractional Problem:

If we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $K - w$ that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j.

Greedy Choice for the Fractional Knapsack Problem

Strategy 1: Pick the item with maximum value

Example

- Knapsack capacity: $K = 1$
- Two items:
  - $w_1 = 100$ $v_1 = 2$
  - $w_2 = 1$ $v_2 = 1$

Taking a fraction of the item with maximum value yields a total value of $V = 2/100$.

Taking the entire item 2 would have given $V = 1$.

Greedy Choice for the Fractional Knapsack Problem

Strategy 2: Pick the item with maximum value per pound

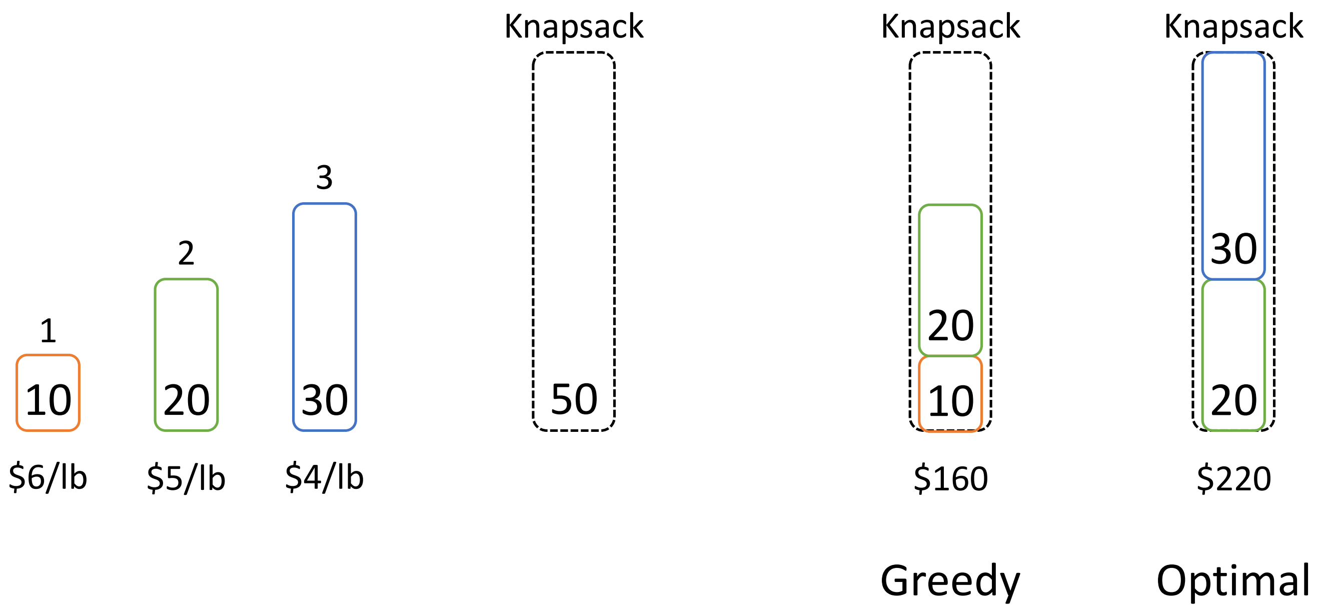
If the supply of that item is exhausted and there is capacity left, take as much as possible from the item with second best value per pound, and so on...

Consider sorting all items by their value per pound in decreasing order

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$$

Greedy Strategy for 0-1 Knapsack

The greedy choice from the fractional problem does not yield optimal solutions for the 0-1 problem



Dynamic Programming for 0-1 Knapsack

The solution is based on the optimal substructure observation:

If we remove the highest-numbered item from an optimal solution S for K pounds and items $1, \dots, n$. Then $S' = S - \{i\}$ must be optimal for $K - w_i$ pounds and items $1, \dots, i-1$, and the value $V = V' + v_i$

Define $c[i, w]$ to be the value of the solution for items $1, \dots, i$ and maximum weight w:

$$c[i, w] = \begin{cases} 0, & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1, w], & \text{if } w_i > w \\ \max(v_i + c[i-1, w - w_i], c[i-1, w]), & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$

Dynamic Programming for 0-1 Knapsack (cont'd)

```
DYNAMIC-0-1-KNAPSACK(v,w,K)
    n = v.length
    let c[0..n,0..K] be a new table
    for j = 0 to K
        c[0,j] = 0
    for i = 1 to n
        c[i,0] = 0
        for j = 1 to K
            if w[i] ≤ j
                c[i,j] = max(v[i] + c[i-1,j-w[i]], c[i-1,j])
            else
                c[i,j] = c[i-1,j]
    return c
```

Conclusions

- Greedy strategies can yield very efficient algorithms
- Whenever we encounter optimal substructure, it is worth thinking of a greedy strategy and check for the optimality of the solution
- Otherwise, we can always fall back on a dynamic programming solution

References

Books

- [Cormen] **Introduction to Algorithms**
  T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. MIT Press. 2009 (3rd Edition)
- [Sedgewick] **Algorithms**
  R. Sedgewick, K. Wayne. Addison-Wesley. 2011 (4th Edition)
- [Dasgupta] **Algorithms**
  S. Dasgupta, C. Papadimitriou, U. Vazirani. McGraw-Hill Higher Education. 2006

Online

- http://algs4.cs.princeton.edu/lectures/
- https://www.coursera.org/courses?query=algorithms

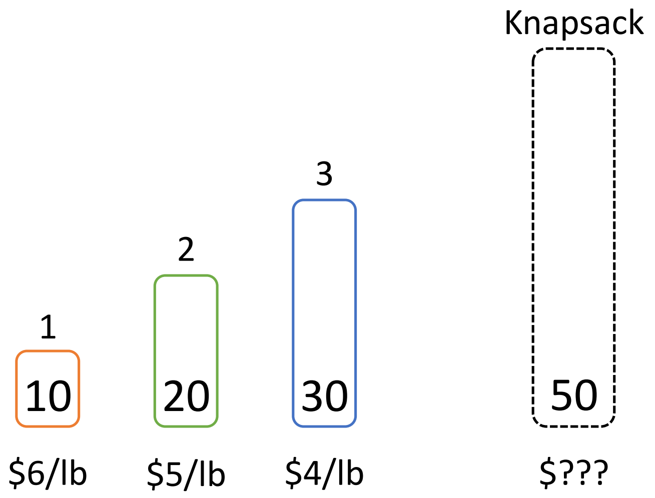
CO202 – Software Engineering – Algorithms

Greedy Algorithms - Exercises

Exercise 1: Implement Fractional Knapsack

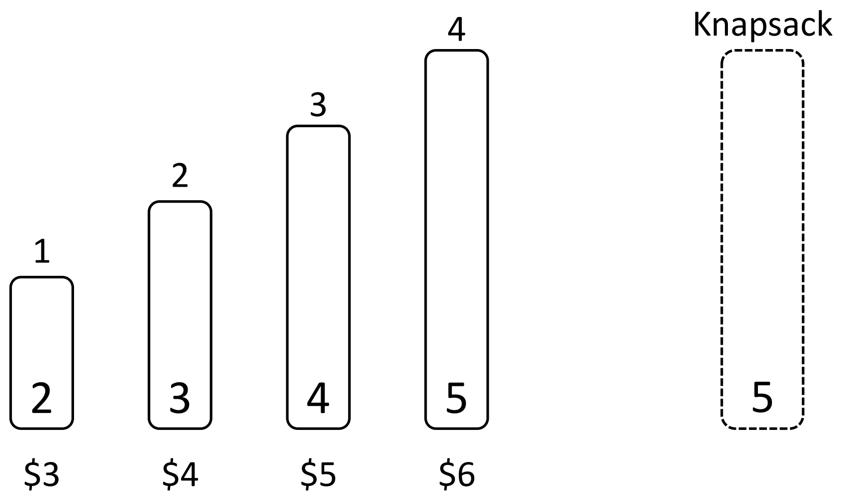
```
FRACTIONAL-KNAPSACK(v,w,K)
```

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4



Exercise 2: Solving 0-1 Knapsack

```
for i = 1 to n
    c[i,0] = 0
    for j = 1 to K
        if w[i] ≤ j
            c[i,j] = max(v[i] + c[i-1,j-w[i]], c[i-1,j])
        else
            c[i,j] = c[i-1,j]
```



i \ j	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

Exercise 3: Coin Change Problem

Prove that a greedy strategy of picking the highest valued coin which is less or equal than the remaining amount is not guaranteed to produce optimal results.

Exercise 4: Planning a Party

Invite as many people as possible from a set of n people, such that

1. Every person invited should know at least five other people that are invited
2. Every person invited should not know at least five other people that are invited

Hint: Maximizing the number of invitees is the same as minimizing the number of people that are not invited.