

CSE 301

Combinatorial Optimization

Greedy Algorithms

Greedy Algorithm

- Solves an optimization problem
 - For many optimization problems, greedy algorithm can be used. (not always)
- Greedy algorithm for optimization problems typically go through a sequence of steps, with a set of choices at each step.
- Current choice does not depend on evaluating potential future choices or pre-solving repeatedly occurring subproblems (a.k.a., *overlapping subproblems*).
- With each step, the original problem is reduced to a smaller problem.
- Greedy algorithm always makes the choice that looks best at the moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

When can we use Greedy algorithms?

We can use a greedy algorithm when the following are true:

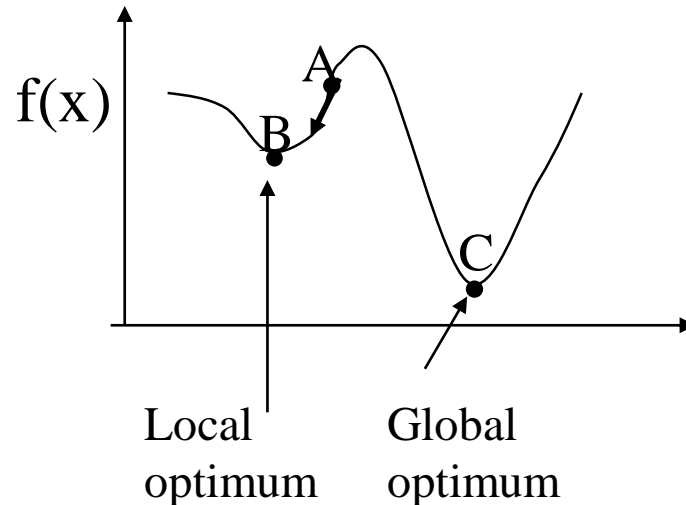
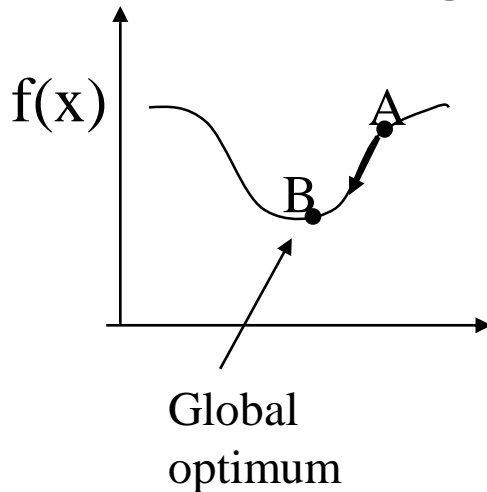
- 1) **The greedy choice property:** A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- 2) **The optimal substructure property:** The optimal solution contains within **its optimal solutions to subproblems**.

Designing Greedy Algorithms

1. Cast the optimization problem as one for which:
 - we make a choice and are left with only one subproblem to solve
2. Prove the **GREEDY CHOICE**
 - that there is always an optimal solution to the original problem that makes the greedy choice
3. Prove the **OPTIMAL SUBSTRUCTURE**:
 - the greedy choice + an optimal solution to the resulting subproblem leads to an optimal solution

Finding the global minima of a function

- start from an arbitrary value of x
- If $f(x+1) < f(x)$ then set $x = x+1$
 - Otherwise if $f(x-1) < f(x)$ then set $x = x-1$
- Continue until changing x doesn't decrease $f(x)$



If we start at A and move in the direction of descent, we will end up at the local optimum, B.

On the left graph, B is also at the global optimum.

On the right graph, the global optimum is elsewhere, at C.

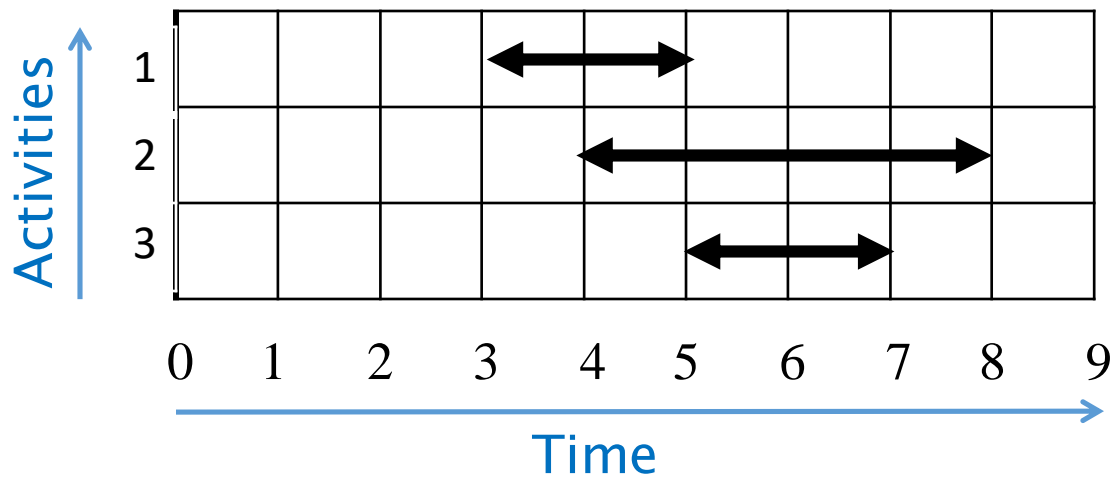
Activity Selection *aka.* Interval Scheduling Problem

Definition: Scheduling a resource among several competing activities.

Elaboration: Suppose we have a set $S = \{1, 2, \dots, n\}$ of n proposed activities that wish to allocate a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity i has a **start time** s_i and **finish time** f_i where $s_i \leq f_i$.

Compatibility: Activities i and j are compatible if the interval $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (i.e. $s_i \geq f_j$ or $s_j \geq f_i$). E.g. in the example below, activities 1 & 2 (as well as activities 2 & 3) are incompatible, and activities 1 & 3 are compatible.

Goal: To select a maximum- size set of mutually compatible activities.



Activity Selection a.k.a. Interval Scheduling Problem

Here are a set of start and finish times

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

What is the maximum number of activities that can be completed?

- {3,9,11} can be completed (not an optimal solution)
- But so can {1,4,8,11} which is a larger set (an optimal solution)
- Solution is not unique, consider {2,4,9,11} (another optimal solution)

The Activity Selection Problem

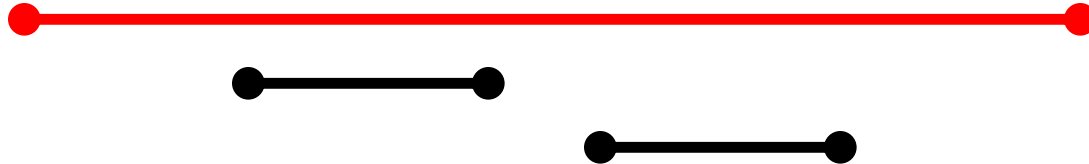
Algorithm 1:

1. sort the activities by the starting time
2. pick the first activity ***a***
3. remove all activities conflicting with ***a***
4. repeat

The Activity Selection Problem

Algorithm 1:

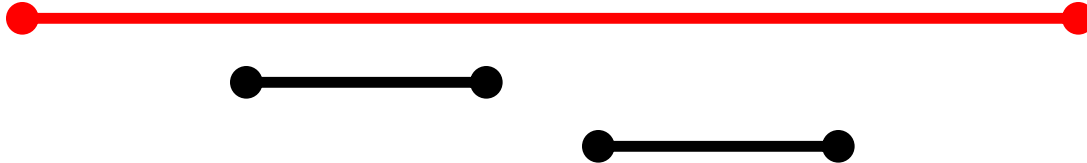
1. sort the activities by the starting time
2. pick the first activity *a*
3. remove all activities conflicting with *a*
4. repeat



The Activity Selection Problem

Algorithm 1:

1. sort the activities by the starting time
2. pick the first activity *a*
3. remove all activities conflicting with *a*
4. repeat



The Activity Selection Problem

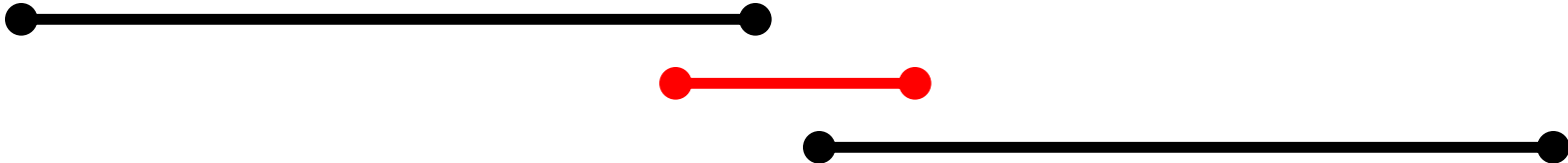
Algorithm 2:

1. **sort** the activities **by length**
2. pick the **shortest activity** ***a***
3. remove all activities conflicting with ***a***
4. repeat

The Activity Selection Problem

Algorithm 2:

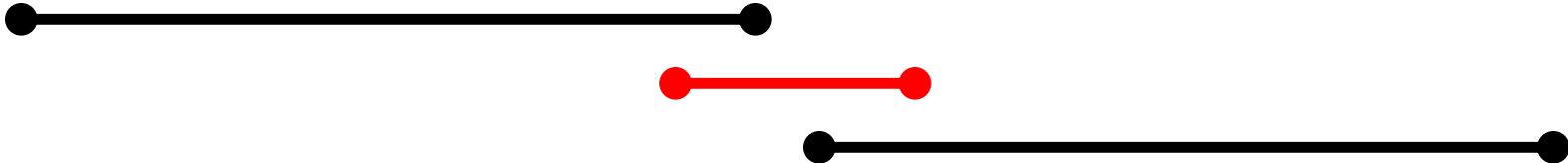
1. **sort** the activities **by length**
2. pick the **shortest activity** *a*
3. remove all activities conflicting with *a*
4. repeat



The Activity Selection Problem

Algorithm 2:

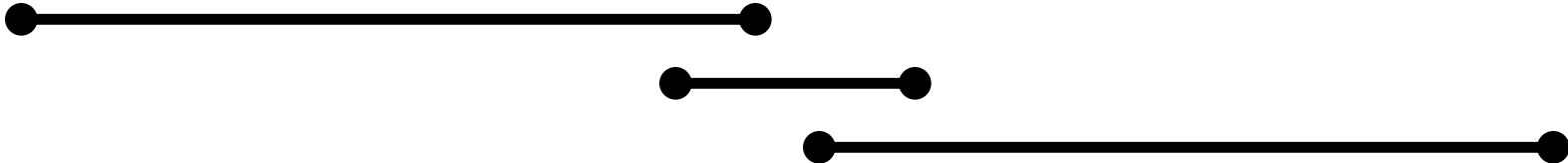
1. **sort** the activities **by length**
2. pick the **shortest activity** ***a***
3. remove all activities conflicting with ***a***
4. repeat



The Activity Selection Problem

Algorithm 3:

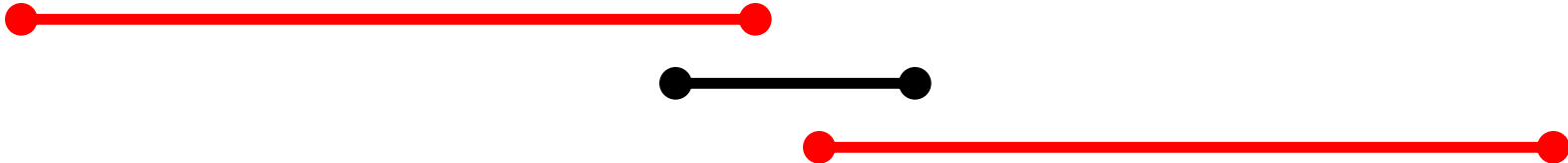
1. **sort** the activities by **ending time**
2. **pick** the activity ***a*** which **ends first**
3. remove all activities conflicting with ***a***
4. repeat



The Activity Selection Problem

Algorithm 3:

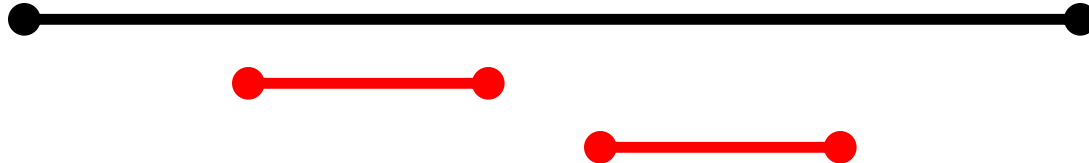
1. **sort** the activities by **ending time**
2. **pick** the activity ***a*** which **ends first**
3. remove all activities conflicting with ***a***
4. repeat



The Activity Selection Problem

Algorithm 3:

1. **sort** the activities by **ending time**
2. **pick** the activity ***a*** which **ends first**
3. remove all activities conflicting with ***a***
4. repeat



Iterative Greedy Algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

//s: array containing start times of input activities

//f: array containing finishing times of input activities

1. $n = s.length$
2. $A = \{a_1\}$
3. $k = 1$ *//k: previously chosen activity*
4. **for** $m = 2$ **to** n
5. **if** $s[m] \geq f[k]$ *//find earliest-finishing activity m which is compatible with k*
6. $A = A \cup \{a_m\}$
7. $k = m$ *//f[k] = max{f[k]: k \in A}, since activities are sorted*
8. **return** A

Running time is $\Theta(n)$

GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

Recursive Greedy Algorithm

Initial call: RECURSIVE-ACTIVITY-SELECTOR($s, f, 0$)

RECURSIVE-ACTIVITY-SELECTOR(s, f, i)

*//Solves the subproblem $S_{i+1,n} = \{a_{i+1}, a_{i+2}, \dots, a_n\}$ of the problem $S_{i,n} = \{a_i, a_{i+1}, \dots, a_n\}$, i.e.,
//computes mutually compatible activities in the set $S_{i+1,n}$ given that $a_i \in A_{i,n}$: solution set of $S_{i,n}$*

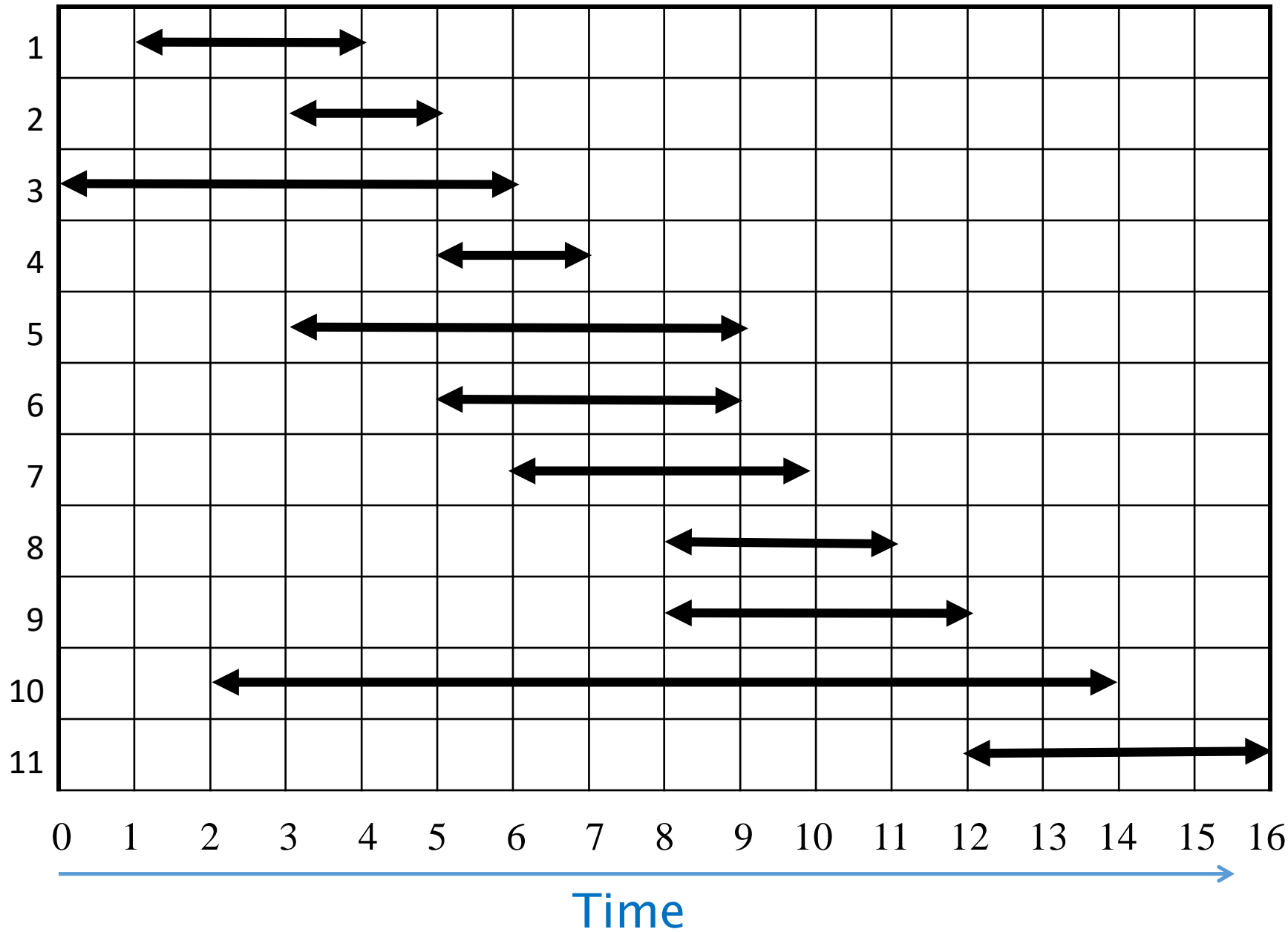
1. $n = s.length$
2. $m = i + 1$
3. **while** $m \leq n$ **and** $s[m] < f[i]$ *//skip all activities which are incompatible with i*
4. $m = m + 1$
5. **if** $m \leq n$ *//if an activity is found in $S_{i+1,n}$ which is compatible with i*
6. $\text{return } \{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, m)$ *//add a_m to solution set*
6. **else** $\text{return } \emptyset$

Running time is $\Theta(n)$

Assuming that the activities have already been sorted by finish times,
the running time of the call RECURSIVE-ACTIVITY-SELECTOR is $\Theta(n)$

Because, over all recursive calls, each activity is examined exactly once in the while loop test of line 3.

Activities

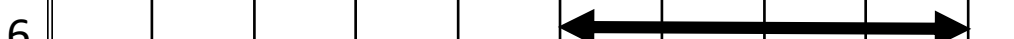


Activities

1
2
3
4
5
6
7
8
9
10
11

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Time



Activities

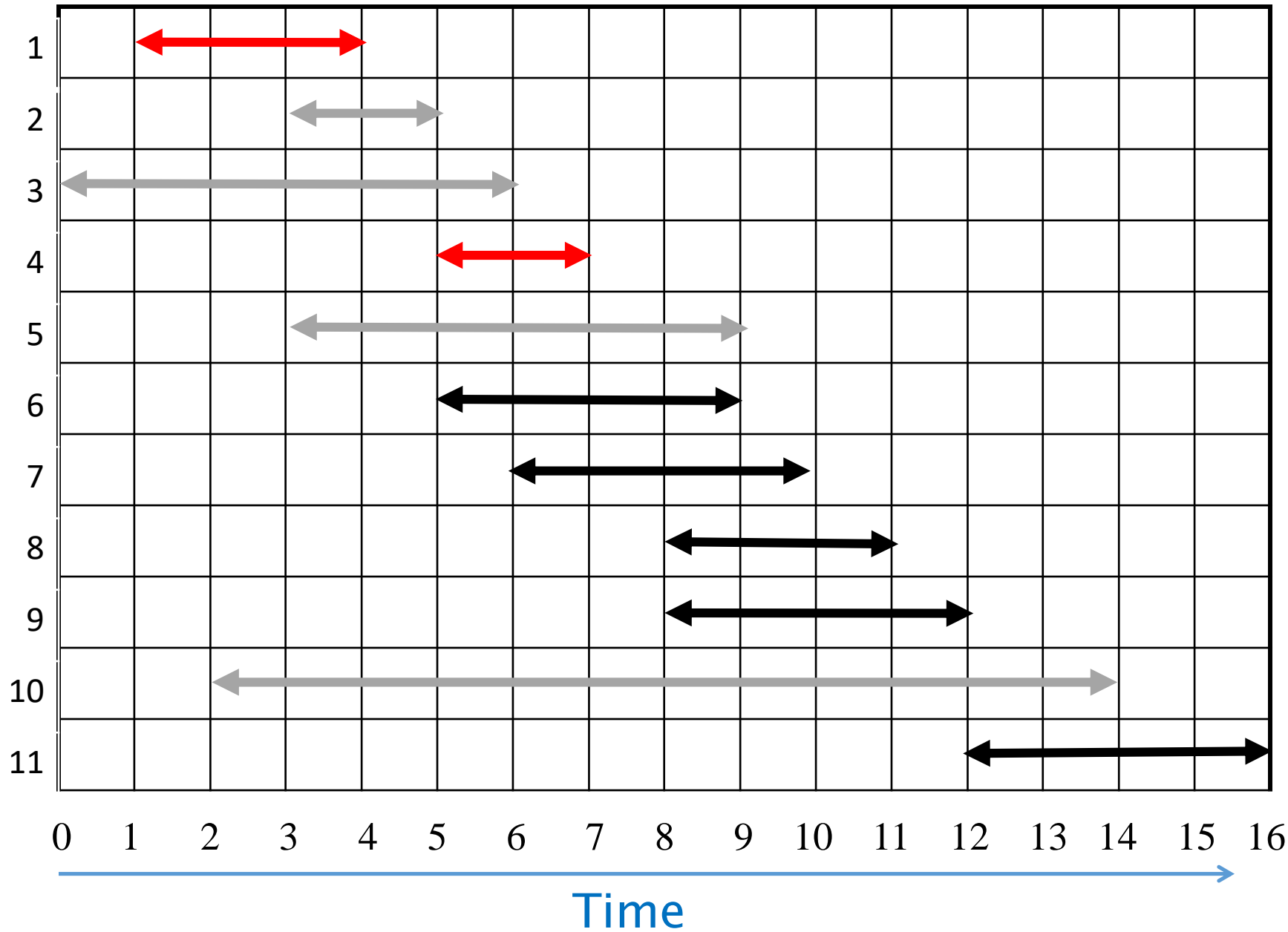
1
2
3
4
5
6
7
8
9
10
11

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Time



Activities



Activities

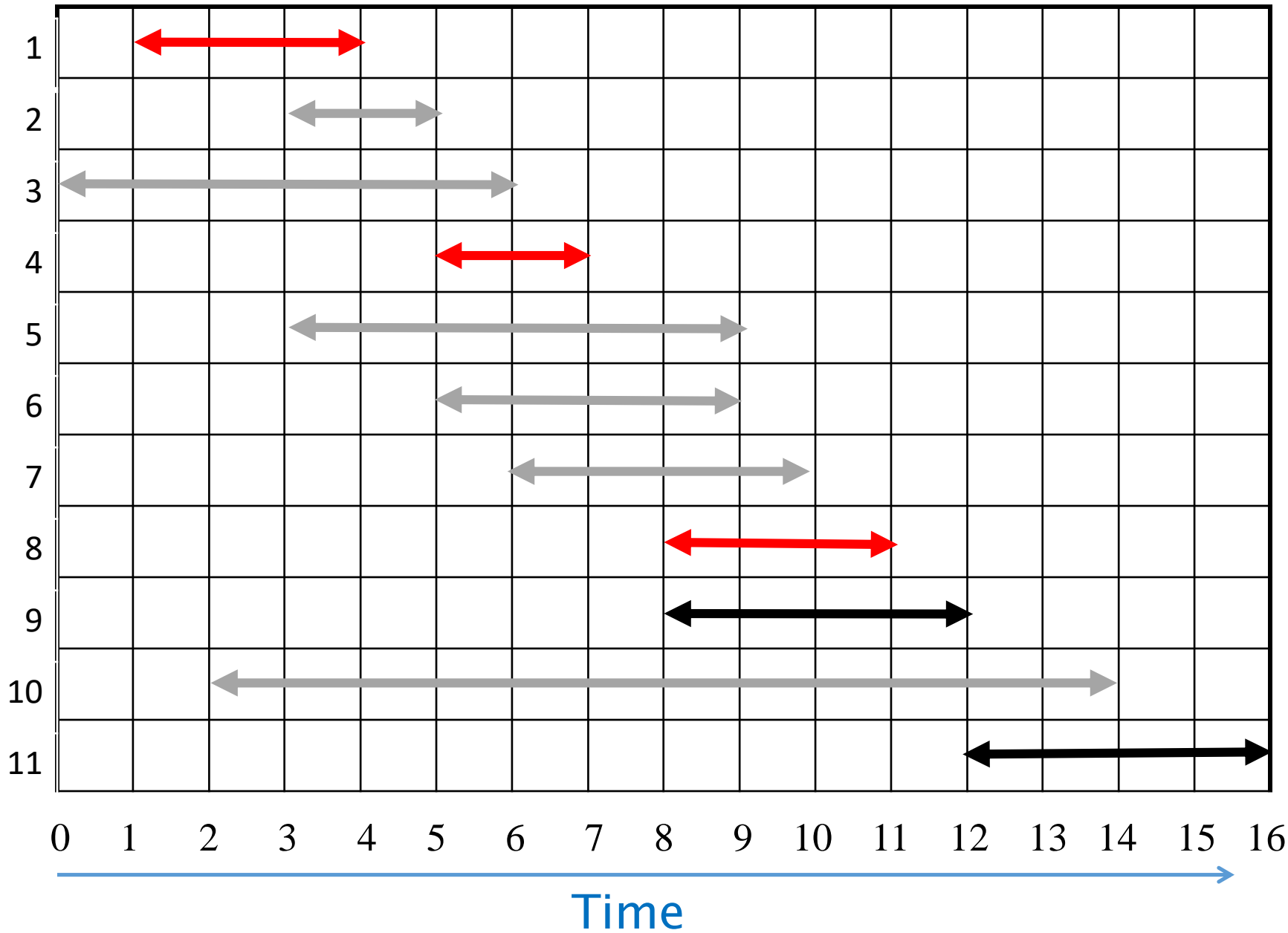
1
2
3
4
5
6
7
8
9
10
11

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Time



Activities



Activities

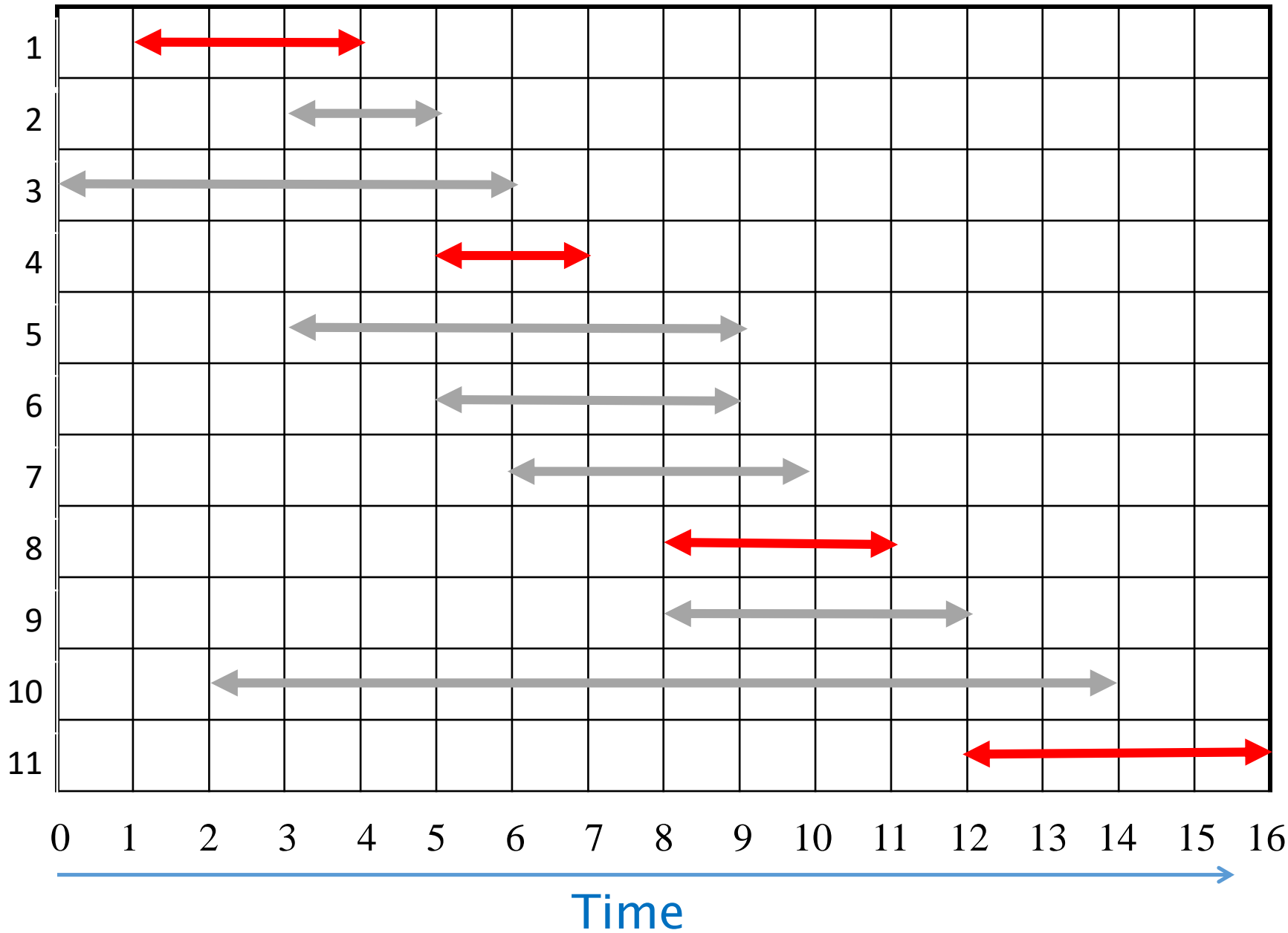
1
2
3
4
5
6
7
8
9
10
11

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Time



Activities



Properties of Greedy Problems

How can one tell if a greedy algorithm will be able to solve an optimization problem i.e., whether a problem is a “Greedy problem”?

There is no way in general. But there are 2 ingredients exhibited by most Greedy Problems (problems solvable via a greedy algorithm):

- 1. Greedy Choice Property:** there is an optimal solution that contains the first greedy choice
 - It implies that a globally optimal solution can be arrived at by making a locally optimal (Greedy) choice.
 - So we can make whatever choice seems best at this moment (i.e., greedy choice) and then solve the subproblem that results after the choice is made.
 - Thus, a greedy strategy usually progresses in a **top-down** fashion, making one greedy choice after another, iteratively reducing each given problem instance to a smaller one.
- 2. Optimal Sub Structure Property:** Optimal solution of a problem contains within it optimal solution of its subproblem(s)
 - It implies that the optimal solution of subproblem(s) can be combined together to obtain the optimal solution of the problem itself

Greedy Choice Property Illustration

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

- An optimal solution which does not begin with activity 1 is: $A = \{2, 4, 9, 11\}$
- Let $B = A - \{2\} \cup \{1\} = \{1, 4, 9, 11\}$; B is an optimal solution, too. **Why?**
 - Activity 2 doesn't clash with 4, 9, or 11. Activity 1 finishes before 2 finishes and as such 1 cannot clash with 4, 9, or 11, either. Also, activities 4, 9, and 11 do not clash with each other (otherwise they wouldn't be members of A).
- Hence $B = \{1, 4, 9, 11\}$ is a valid solution. Since $|B| = |A|$, B is an optimal solution.

Optimal Substructure Property Illustration

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

- Take an optimal solution of $S = \{1,2,3,4,5,6,7,8,9,10,11\}$ which contains activity 1: $A = \{1,4,9,11\}$
- According to the optimal substructure property:
 - $A' = A - \{1\} = \{4,9,11\}$ is an optimal solution of $S' = \{4,6,7,8,9,11\}$ (*a.k.a.* a subproblem of the original problem S)

Implications of greedy choice and optimal substructure properties

- After each greedy choice is made, we are left with an smaller optimization problem (called a subproblem) of the same form as the original problem
- By induction, making the greedy choice for every subproblem produces an optimal solution of the whole problem

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

Current Problem	Greedy Choice for Current Problem	Solution to current problem
$S = \{1,2,3,4,5,6,7,8,9,10,11\}$	1	$A = \{1, \dots\}$
$S' = \{4,6,7,8,9,11\}$	4	$A' = \{4, \dots\}$
$S'' = \{6,7,8,9,11\}$	6	$A'' = \{6, \dots\}$
$S''' = ?$?	?

Practice Problem on Activity Selection

i	1	2	3	4	5	6	7	8	9	10	11
s_j	1	4	5	6	9	11	10	6	8	13	15
f_j	5	6	9	10	12	15	18	8	10	14	20

Current Problem	Greedy Choice for Current Problem	Solution to current problem

Greedy-Choice Property of Activity Selection Problem

Claim: There is an optimal solution that contains the first greedy choice (*i.e.*, activity 1, which has the earliest finish time)

Proof:

Let $S = \{1, 2, \dots, n\}$ be the set of input activities sorted by their finishing times, *i.e.*, $f_1 \leq f_2 \leq \dots \leq f_n$

An optimal solution will be a subset of S

Suppose $A \subseteq S$ is an optimal solution

Order the activities in A by finish time. Let k be the first activity in A in this order.

If $k = 1$, the solution A contains activity 1 and the proof completes.

If $k \neq 1$, we have to show that there is an optimal solution $B \subseteq S$ which contains activity 1

Greedy-Choice Property of Activity Selection Problem

Let $B = A - \{k\} \cup \{1\}$.

- activity 1 has the earliest finish time in $S \Rightarrow f_1 \leq f_k \dots (1)$ and
- activity k finishes before other activities in A starts (otherwise, activity k would clash with other activities in set A) $\Rightarrow f_k \leq s_j \dots (2)$ for all $j \in A - \{k\}$

$(1) \& (2) \Rightarrow f_1 \leq s_j$ for all $j \in A - \{k\} \Rightarrow$ activity 1 doesn't clash with other activities in B

Also, activities in $A - \{k\}$ *do not clash with each other* (Since, A is a valid solution)

Hence activities in B are mutually compatible. Also, $|B| = |A|$

Thus, B is an optimal solution which contains activity 1 (the first greedy choice)

Optimal Substructure Property of Activity Selection Problem

Once the greedy choice of activity 1 is made, the problem reduces to finding an optimal solution for the activity-selection problem over those activities in S that are compatible with activity 1

Optimal Substructure property

If A is an optimal solution of input S which contains the first greedy choice (activity 1), then $A' = A - \{1\}$ is an optimal solution of input $S' = \{i \in S: s_i \geq f_1\}$

Proof:

If we could find an optimal solution, B' of S' where $|B'| > |A'|$, then $B = B' \cup \{1\}$ would be an optimal solution of S which contains more activities than A

(since $|B| = |B'| + 1 > |A'| + 1 = |A|$),

But this contradicts our premise that A is an optimal solution.






The Fractional Knapsack Problem

Given: A set S of n items, with each item i having

b_i - a positive benefit

w_i - a positive weight

Goal: Choose items with maximum total benefit but with weight at most W .

Items:					
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	\$12	\$32	\$40	\$30	\$50
Value: (\$ per ml)	3	4	20	5	50



"knapsack"

10 ml

Solution: P

- 1 ml of 5 50\$
- 2 ml of 3 40\$
- 6 ml of 4 30\$
- 1 ml of 2 4\$

•Total Profit: 124\$

The Fractional Knapsack Problem

Given: A set S of n items, with each item i having

b_i - a positive benefit

w_i - a positive weight

Goal: Choose items with maximum total benefit but with weight at most W .

If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.

In this case, we let x_i denote the amount we take of item i

Objective: maximize

Constraint:
$$\sum_{i \in S} b_i (x_i / w_i)$$

$$\sum_{i \in S} x_i \leq W, 0 \leq x_i \leq w_i$$

The Fractional Knapsack Algorithm

Greedy choice: Keep taking item with highest **value** (benefit to weight ratio)

Since

$$\sum_{i \in S} b_i (x_i / w_i) = \sum_{i \in S} (b_i / w_i) x_i$$

Algorithm *fractionalKnapsack*(S, W)

Input: set S of items w/ benefit b_i and weight w_i ; max. weight W

Output: amount x_i of each item i to maximize benefit w/ weight at most W

for *each item* i **in** S

$x_i \leftarrow 0$

$v_i \leftarrow b_i / w_i$ {value}

$w \leftarrow 0$ {total weight}

while $w < W$

remove item i *with highest* v_i

$x_i \leftarrow \min\{w_i, W - w\}$

$w \leftarrow w + x_i$

The Fractional Knapsack Algorithm

Running time:

Given a collection S of n items, such that each item i has a benefit b_i and weight w_i , we can construct a maximum-benefit subset of S , allowing for fractional amounts, that has a total weight W in $O(n \log n)$ time. (how?)

- Use heap-based priority queue to store S

- Removing the item with the highest value takes $O(\log n)$ time

- In the worst case, need to remove all items

Practice Problem on Fractional Knapsack

Item	1	2	3	4	5	6	7
Weight(Kg)	2	3	4	2	5	4	1
Price	7	6	9	6	12	10	3

Total capacity of Knapsack = 15Kg

Remaining capacity of knapsack	Item chosen	Per Kg Price of chosen item	Weight of chosen item	Total benefits/prices of the item chosen

Huffman Codes

Widely used technique for data compression

Assume the data to be a sequence of characters

Looking for an effective way of storing the data

Binary character code

Uniquely represents a character by a binary string

Fixed-Length Codes

E.g.: Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

3 bits needed

a = 000, b = 001, c = 010, d = 011, e = 100, f = 101

Requires: $100,000 \cdot 3 = 300,000$ bits

Compression ratio = $300000/800000 = 0.375$ using fixed length encoding.

Huffman Codes

Idea:

Use the frequencies of occurrence of characters to build a optimal way of representing each character

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

Variable-Length Codes

E.g.: Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

Assign short codewords to frequent characters and long codewords to infrequent characters

$a = 0$, $b = 101$, $c = 100$, $d = 111$, $e = 1101$, $f = 1100$

File size after encoding using these codewords =

$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000$

$= 224,000$ bits

Compression ratio using this encoding =

$224000/800000 = 0.28$

Prefix Codes

Prefix codes:

Codes for which no codeword is also a prefix of some other codeword

Better name would be “prefix-free codes”

We can achieve optimal data compression using prefix codes

We will restrict our attention to prefix codes

Encoding with Binary Character Codes

Encoding

Concatenate the codewords representing each character in the file

E.g.:

$a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100$

$abc = 0 \cdot 101 \cdot 100 = 0101100$

Decoding with Binary Character Codes

Prefix codes simplify decoding

No codeword is a prefix of another \Rightarrow the codeword that begins an encoded file is unambiguous

Approach

Identify the initial codeword

Translate it back to the original character

Repeat the process on the remainder of the file

E.g.:

a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

001011101 =

0 · 0 · 101 · 1101 = aabe

Constructing a Huffman Code

A greedy algorithm that constructs an optimal prefix code called a **Huffman code**

Assume that:

- \mathcal{C} is a set of n characters

- Each character has a frequency $f(c)$

- The tree T is built in a bottom up manner

Idea:

- Start with a set of $|\mathcal{C}|$ leaves

f: 5	e: 9	c: 12	b: 13	d: 16	a: 45
------	------	-------	-------	-------	-------

- At each step, merge the two least frequent objects: the frequency of the new node = sum of two frequencies

- Use a min-priority queue Q , keyed on f to identify the two least frequent objects

Constructing a Huffman tree

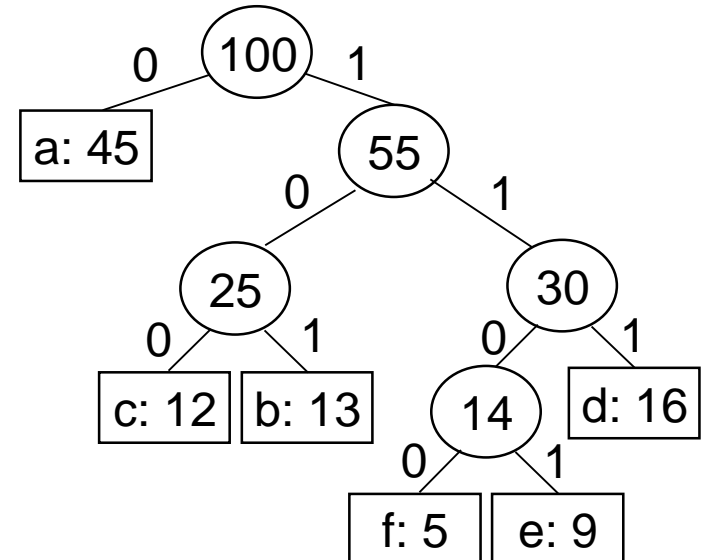
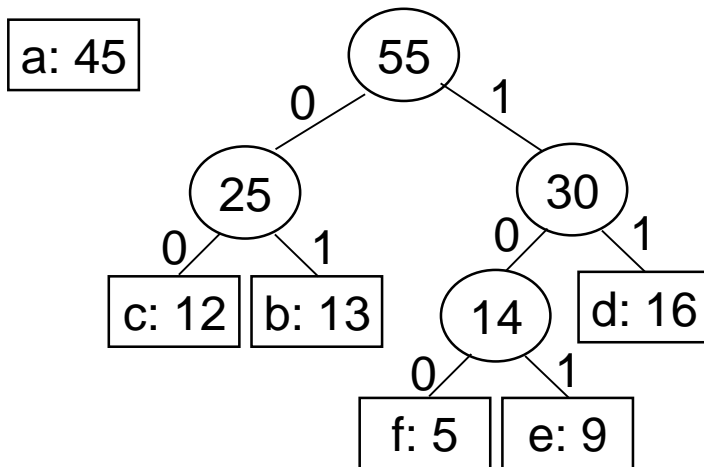
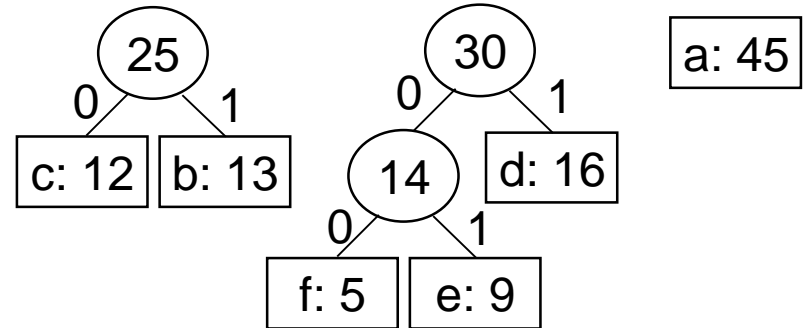
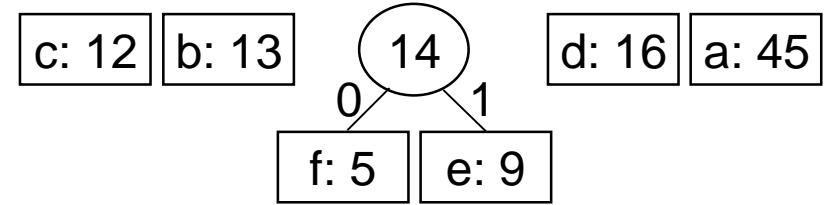
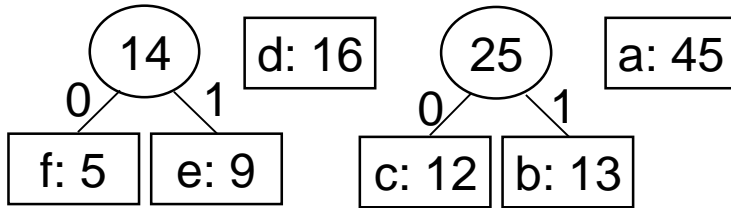
Alg.: HUFFMAN(C)

Running time: $O(n \lg n)$

1. $n \leftarrow |C|$
 2. $Q \leftarrow C$
 3. **for** $i \leftarrow 1$ **to** $n - 1$ $\longleftarrow O(n)$
 4. **do** allocate a new node z
 5. $\text{left}[z] \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$
 6. $\text{right}[z] \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$
 7. $f[z] \leftarrow f[x] + f[y]$
 8. $\text{INSERT}(Q, z)$
 9. **return** $\text{EXTRACT-MIN}(Q)$
- $\left. \begin{array}{l} \text{4.} \\ \text{5.} \\ \text{6.} \\ \text{7.} \\ \text{8.} \end{array} \right\} O(\lg n)$

Example

f: 5 e: 9 c: 12 b: 13 d: 16 a: 45



Huffman encoding/decoding using Huffman tree

Encoding:

- Construct Huffman tree using the previous algorithm
- Traverse the Huffman tree to assign a code to each leaf (representing an input character)
- Use these codes to encode the file

Decoding:

- Construct Huffman tree using the previous algorithm
- Traverse the Huffman tree starting from root node according to the bits you encounter until you reach a leaf node at which point you output the character represented by that leaf node & then go back to the root node.
- Continue in this fashion until all the bits in the file are read.

Huffman Coding Practice

A file has the following characters along with their frequencies:

a: 20, b: 30, c: 78, d: 12, e: 27, f: 67 g: 5, h: 80.

Draw Huffman coding tree and compute the optimal Huffman code of each character mentioned above. Also compute the compression ratio achieved via this encoding.