CSE3081 Design and Analysis of Algorithms

Dept. of Computer Engineering,
Sogang University

Chapter 15. Greedy Algorithms

Overview

- For some optimization problems, even dynamic programming is an overkill.
- A simpler and more efficient algorithm can be applied.

- An optimization problem typically goes through a sequence of steps, with a set of choices at each step.
- A greedy algorithm always makes the choice that looks best at the moment.
- In other words, it makes a locally optimal choice in the hope that this choice leads to a globally optimal solution.

- Depending on the problem, a greedy algorithm may not yield the optimal solution. Therefore, you should make sure that for your problem, a greedy algorithm can produce the globally optimal solution.
- That said, the greedy method is quite powerful and works well for a wide range of problems (e.g. minimum spanning tree algorithms, Dijkstra's shortest-path algorithms)

15.1 An Activity-Selection Problem

Introduction

- We consider the problem of scheduling several competing activities that require exclusive use of a common resource, with the goal of selecting a maximum-size set of mutually compatible activities.

- We are presented with a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to reserve the conference room.
  - The conference room can serve only one activity at a time.

- Each activity a_i has a start time s_i and a finish time f_i.
  - $0 \leq s_i \leq f_i < \infty$

- If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$.

- Activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, they are compatible if $s_i \geq f_j$ or $s_j \geq f_i$.

The Problem

- In the activity-selection problem, our goal is to select a maximum-size subset of mutually compatible activities.
- In the input, assume the activities are sorted in monotonically increasing order of finish time.

- Example

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

- $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities.
- $\{a_1, a_4, a_8, a_{11}\}$ also consists of mutually compatible activities.
- $\{a_2, a_4, a_9, a_{11}\}$ also consists of mutually compatible activities.
- There is no subset of size ≥ 5 that has mutually compatible activities.

- The maximum-size is 4, and there are multiple solutions with this size.

The Optimal Substructure

- This problem looks like we can establish an optimal substructure and apply dynamic programming to solve it.

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

- We want to find a maximum set of mutually compatible activities in S_{ij}.
- Suppose that such a maximum set is A_{ij}, which includes some activity a_k.

- By including a_k in an optimal solution, we are left with two subproblems.
  - Finding mutually compatible activities in the set S_{ik}, and
  - Finding mutually compatible activities in the set S_{kj}.

The Optimal Substructure

- Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$
  - A_{ik} contains the activities in A_{ij} that finish before a_k starts.
  - A_{kj} contains the activities in A_{ij} that start after a_k finishes.

- Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$.

- The maximum size set A_{ij} of mutually compatible activities in S_{ij} consist of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.

- An optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}.

The Optimal Substructure

- Let us denote the size of an optimal solution for the set S_{ij} by $c[i,j]$.

- Then, the recurrence relation will be:
- $c[i,j] = c[i,k] + c[k,j] + 1$

- Since we do not know that an optimal solution for the set S_{ij} includes activity a_k, we must examine all activities in S_{ij} to find which one to choose.

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

- Now we can use dynamic programming to solve this problem.

- But is this the best we can do?

Making the Greedy Choice

- Do we really need to solve all the subproblems in order to choose an activity to add to an optimal solution?

- In the activity-selection problem, we need to consider only one choice: the greedy choice.

- Let's approach this problem intuitively. In order to construct the optimal solution, we want to select one activity from the given list.

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

- Since our goal is to find the maximum number of activities, intuitively we select an activity that leaves the resource available for as many other activities as possible. → Choose an activity with the earliest finish time?

Making the Greedy Choice

- There could by many "greedy" choices, but let us follow this method and choose the activity with the earliest finish time.

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

- Since the input is sorted according to the finish times, we choose a_1.

- Once we make a greedy choice, we are left with one remaining subproblem: finding activities that start after a_1 finishes.
- Let $S_k = \{a_i \in S: s_i \geq f_k\}$ be the set of activities that start after a_k finishes.
- The optimal substructure says that if a_1 belongs to an optimal solution, then an optimal solution to the original problem consists of activity a_1 and the activities in an optimal solution to the subproblem S_1.

Making the Greedy Choice

- One big question remains: Is the greedy choice always part of some optimal solution?

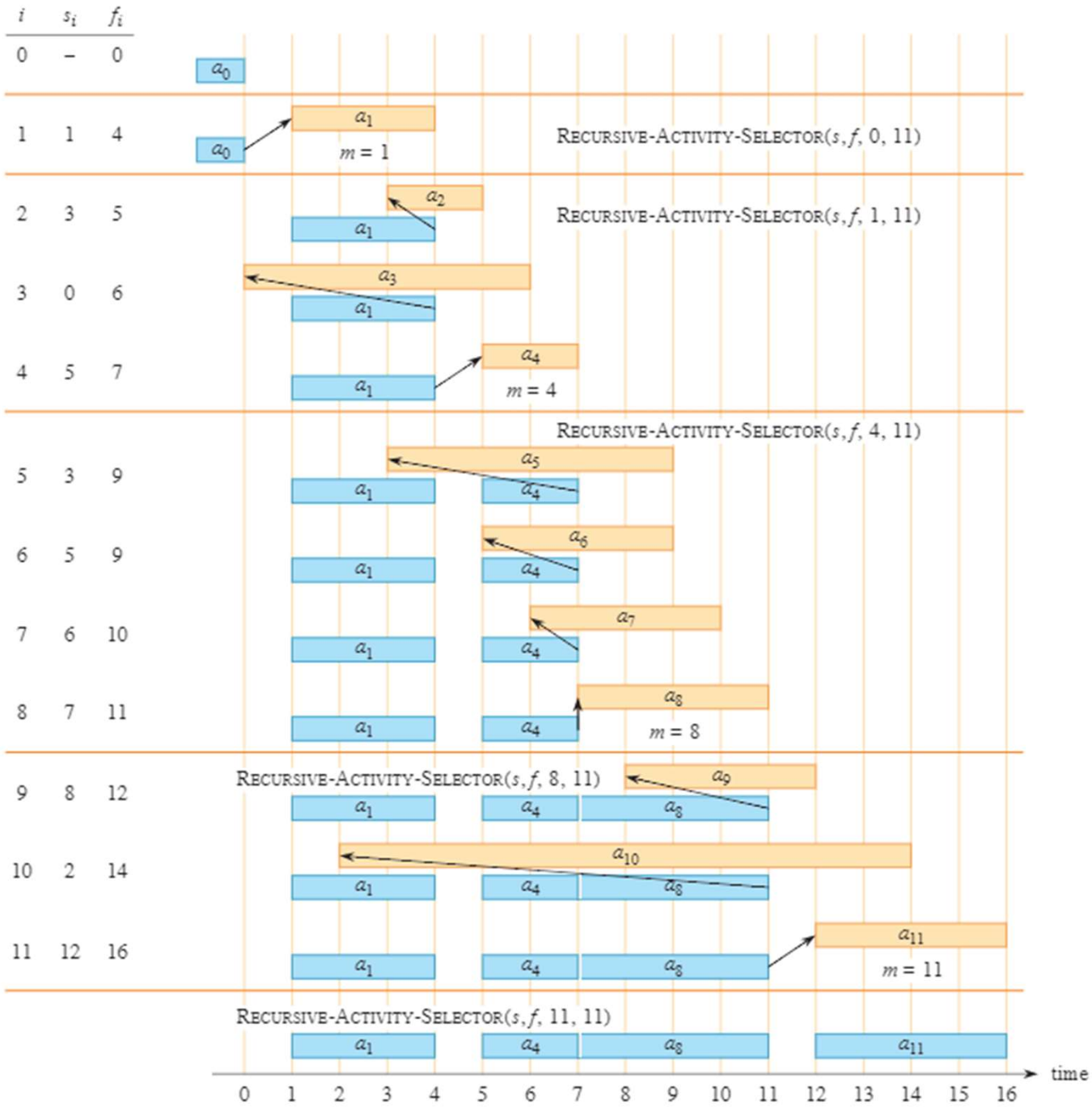
- Theorem 15.1
  - Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then, a_m is included in some maximum-size subset of mutually compatible activities of S_k.
- Proof
  - Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time.
  - If $a_j = a_m$, we are done.
  - If $a_j \neq a_m$, let the set $A'_k = \left(A_k - \{a_j\}\right) \cup \{a_m\}$ be A_k but substituting a_m for a_j.
  - The activities in A'_k are compatible, because the activities in A'_k are compatible, a_j is the first activity in A_k to finish, 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.

The Greedy Algorithm: Recursive Version

- The procedure assumes that the n input activities are already ordered by monotonically increasing finish time.
  - If not, we need $O(n \log n)$ time to first sort the activities.

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

- Assuming that the inputs are already sorted, the procedure runs in $\Theta(n)$.
  - Over all recursive calls, each activity is examined exactly once in line 2.


i s_i f_i
0 – 0
1 1 4
2 3 5
3 0 6
4 5 7
5 3 9
6 5 9
7 6 10
8 7 11
9 8 12
10 2 14
11 12 16
a0
a1
a2
a3
a4
a5
a6
a7
a8
a9
a10
a11
m = 1
m = 4
m = 8
m = 11
RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 1, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)
time
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

The Greedy Algorithm: Iterative Version

- This procedure also assumes that the inputs are already sorted.

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
1  A = {a_1}
2  k = 1
3  for m = 2 to n
4       if s[m] ≥ f[k]          // is a_m in S_k?
5            A = A ∪ {a_m}      // yes, so choose it
6            k = m              // and continue from there
7  return A
```

- Assuming that the activities are already sorted, GREEDY-ACTIVITY-SELECTOR runs in $\Theta(n)$, same as the recursive version.

15.2 Elements of the Greedy Strategy

Greedy Algorithms

- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices.

- At each decision point, the algorithm makes the choice that seems best at the moment.

- This heuristic strategy does not always produce an optimal solution, but as in the activity-selection problem, sometimes it does.

Greedy-Choice Property

- For certain problems, you can assemble a globally optimal solution by making locally optimal (greedy) choices.
  - These problems are said to have a greedy-choice property.
  - When you are considering which choice to make, you make the choice that looks best in the current problem, without considering results from subproblems.
  - You make whatever choice seems best at the moment and then solve the subproblems that remains.

- The case for dynamic programming problems
  - You make a choice at each step, but the choice usually depends on the solutions to subproblems.
  - Consequently, you typically solve dynamic programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems.

Exchange Argument

- In order to apply a greedy algorithm, we need to prove that the greedy choices actually lead to a globally optimal solution.

- We typically use what is called the "exchange argument".
  - We assume an optimal solution to a subproblem.
  - Then, we substitute the greedy choice for some other choice in the optimal solution.
  - Finally, we prove that the greedy choice yields the result at least as good the other choice.

- Example: proof used in Theorem 15.1
  - Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time.
  - If $a_j = a_m$, we are done.
  - If $a_j \neq a_m$, let the set $A'_k = \left(A_k - \{a_j\}\right) \cup \{a_m\}$ be A_k but substituting a_m for a_j.

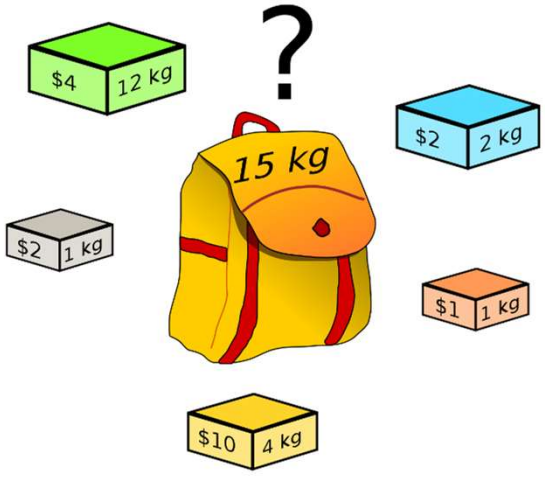
Dynamic Programming vs. Greedy Method

- Because both the greedy and dynamic programming strategies exploit optimal substructure, we are tempted to generate a dynamic programming solution to a problem when a greedy solution suffices, or, conversely, we may mistakenly think a greedy solution works when in fact a dynamic programming solution is required.

The Knapsack Problems

- 0-1 Knapsack Problem
  - A thief robbing a store wants to take the most valuable load that can be carried in a knapsack capable of carrying at most W pounds of loot.
  - The thief can choose to take any subset of n items in the store.
  - The ith item is worth v_i dollars and weights w_i pounds, where v_i and w_i are integers.
  - Which items should the thief take?

- Fractional Knapsack Problem
  - The setup is the same as the 0-1 knapsack problem, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item.



The Knapsack Problems: Optimal Substructure

- For the 0-1 problem, if the most valuable load weighing at most W pounds includes item j, then the remaining load must be the most valuable load weighing at most $W - w_j$ pounds that the thief can take from the $n - 1$ original items excluding item j.

- For the comparable fractional problem, if the most valuable load weighing at most W pounds includes weights w of item j, then the remaining load must be the most valuable load weighing at most $W - w$ pounds that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j.

Applying Greedy Strategy to Fractional Knapsack

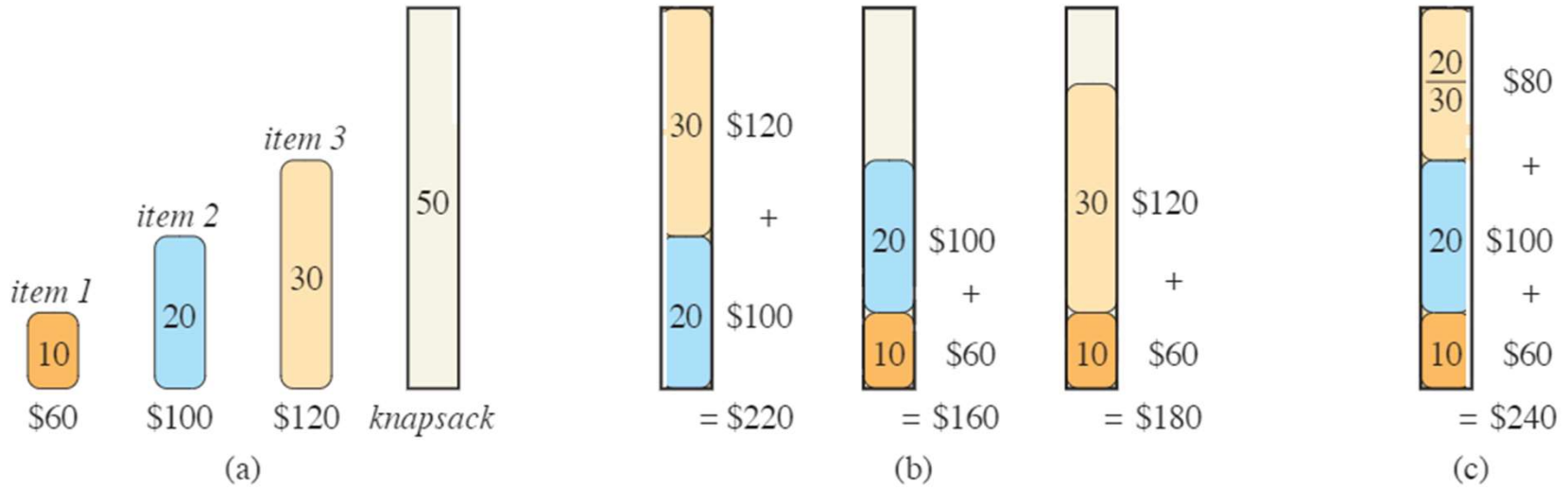
- The greedy strategy works for the fractional knapsack problem.

- First, compute the value per pound v_i/w_i for each item.
- Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound.
- If the supply of that item is exhausted and the thief can still carry more, then the thief takes as much as possible of the item with the next greatest value per pound, and so forth, until reaching the weight limit W.

- By sorting the items by value per pound, the greedy algorithm runs in $O(n \log n)$ time.

Applying Greedy Strategy to 0-1 Knapsack

- However, the greedy strategy does not work for the 0-1 knapsack problem.
  - The empty space wasted lowers the value per pound of the selected items.



input 0-1 Knapsack Fractional

The 0-1 Knapsack Problem

- Problem

  Given two sets of positive integers $\{w_1, w_2, \cdots, w_n\}$ and $\{p_1, p_2, \cdots, p_n\}$ of size n and a positive integer W, find a subset A of $\{1, 2, \cdots, n\}$ that maximizes $\sum_{i \in A} p_i$ subject to $\sum_{i \in A} w_i \leq W$.

- Example

  $\{w_1, w_2, \cdots, w_5\} = \{6, 5, 10, 3, 4\}$, $\{p_1, p_2, \cdots, p_5\} = \{9, 7, 11, 6, 8\}$, $W = 15$
  $\longrightarrow \{1, 2, 5\}$

- An intuitive interpretation
  - There are *n* items in a store.
  - The *i* th item weighs w_i kilograms and is worth p_i wons, where w_i and p_i are positive integers.
  - A thief has a knapsack that can carry at most *W* kilograms, where *W* is a positive integer.
  - What items should the thief take to maximize his "profit"?

The 0-1 Knapsack Problem

✓ There are 2^n subsets of {1, 2, …, n}!

- The optimal substructure
  - Let $P(i, w)$ be the maximized profit obtained when choosing items only from the first i items under the restriction that the total weight cannot exceed w.
  - If we let A^* be an optimal subset of {1, 2, …, n},

$$\begin{aligned} &1.\ n \in A^* :\ P(n, W) = p_n + P(n-1, W - w_n) \\ &2.\ n \notin A^* :\ P(n, W) = P(n-1, W) \end{aligned}$$

$$P(i,w) = \begin{cases} 0, & \text{if } i = 0 \text{ or } w = 0 \\ P(i-1, w), & \text{if } w_i > w \\ \max\{\, P(i-1, w),\ p_i + P(i-1, w - w_i)\,\}, & \text{if } i > 0 \text{ and } w_i \le w \end{cases}$$

The 0-1 Knapsack Problem

```
int zero_one_knapsack(int *p, int *w, int n, int W) {
  int i, ww, tmp;
    …
  for (ww = 0; ww <= W; ww++) P[0][ww] = 0;
  for (i = 1; i <= n; i++) {
    P[i][0] = 0;
    for (ww = 1; ww <= W; ww++) {
      if (w[i] <= ww) {
        if ((tmp = p[i] + P[i-1][ww-w[i]]) > P[i-1][ww])
          P[i][ww] = tmp;
        else
          P[i][ww] = P[i-1][ww];
      }
      else P[i][ww] = P[i-1][ww];
    }
  }
  return P[n][W];
}
```

→ $O(nW)$ time

$$P(i,w) = \begin{cases} 0, & \text{if } i = 0 \text{ or } w = 0 \\ P(i-1,w), & \text{if } w_i > w \\ \max\{\, P(i-1,w),\ p_i + P(i-1,w-w_i)\,\}, & \text{if } i > 0 \text{ and } w_i \le w \end{cases}$$

The 0-1 Knapsack Problem: Analysis

- Is the time-complexity $O(nW)$ an efficient one?
  - This is not a linear-time algorithm!
    - A problem is that W is not bounded with respect to n.
    - What if $n = 20$ and $W = 20$!? → $O(n*n!)$
    - When W is extremely large in comparison with n, this algorithm is worse than the brute-force algorithm that simply considers all subsets.
  - This algorithm can be improved so that the worst-case number of entries computed is $O(2^n)$.
  - What if we use the divide-and-conquer strategy to solve this problem? → would get an $O(2^n)$ algorithm.
  - ✓ No one has ever found an algorithm for the 0-1 Knapsack problem whose worst-case time complexity is better than exponential, yet no one has proven that such an algorithm is not possible!

A Variation of the 0-1 Knapsack Problem

- Problem

  Given n items of length $l_1, l_2, \cdots, l_n$, is there a subset of these items with total length exactly L?

A Variation of the 0-1 Knapsack Problem: Divide-and-Conquer

- Let *fill*(*i*, *j*) return TRUE if and only if there is a subset of the first *i* items that has total length *j*.
- When *fill*(*i*, *j*) returns TRUE,
  - ① If the *i*th item is used, *fill*(*i* - 1, *j* - l_i) must return TRUE.
  - ② If the *i*th item is not used, *fill*(*i* - 1, *j*) must return TRUE.

```
int fill(int i, int j) {
  // l[i]: global
  if (i == 0) {
    if(j == 0) return TRUE;
    else return FALSE;
  }
  if (fill(i-1, j))
    return TRUE;
  else if (l[i] <= j)
    return fill(i-1, j-l[i]);
}
```

$$T(n) \leq \begin{cases} c, & \text{if } n = 0 \\ 2T(n-1) + d, & \text{if } n > 0 \end{cases}$$

$$\Downarrow$$

$$T(n) = \Theta(2^n)$$

A Variation of the 0-1 Knapsack Problem: Dynamic Programming

- The optimal substructure

$$F(i,j) = \begin{cases} \text{FALSE}, & \text{if } i = 0 \text{ and } j \neq 0 \\ \text{TRUE}, & \text{if } i = 0 \text{ and } j = 0 \\ F(i-1,j) \ \textbf{or} \ ((l_i \leq j) \ \textbf{and} \ F(i-1, j-l_i)), & \text{if } i > 0 \end{cases}$$

- *O*(*nL*)-time implementation

```
…
F[0][0] = TRUE;
for (ll = 1; ll <= L; ll++) F[0][ll] =FALSE;
for (i = 1; i <= n; i++) {
  for (ll = 0; ll <= L; ll++) {
    F[i][ll] = F[i-1][ll];
    if (ll - l[i] >= 0)
      F[i][ll] = F[i][ll] || F[i-1][ll-l[i]];
  }
}
return (F[n][L]);
```

A Variation of the 0-1 Knapsack Problem

- Example
  - $L = 15$, $(l_1, l_2, l_3, l_4, l_5, l_6, l_7) = (1, 2, 2, 4, 5, 2, 4)$

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

The Fractional Knapsack Problem

- Problem

  Given two sets of positive integers $\{w_1, w_2, \cdots, w_n\}$ and $\{p_1, p_2, \cdots, p_n\}$ of size n and a positive integer W, find a set of ratios $\{r_1, r_2, \cdots, r_n\}$ $(0 \leq r_i \leq 1)$ that maximizes $\sum_i r_i \cdot p_i$ subject to $\sum_i r_i \cdot w_i \leq W$.

- Analogy
  - In the supermarket, there are n items of different prices. In an event day, the supermarket gives you a bag where you can put portion of any items for free. Naturally, your goal is to put the items so that the total price is the maximum. What is your solution?

The Fractional Knapsack Problem

- A greedy approach
  - Sort the items in non-increasing order by profits per unit weight pi / wi.
  - Choose the items, possibly partially, one by one until the knapsack is full.

- Example Problem
  - $\{w_1, w_2, w_3\} = \{5, 10, 20\}$, $\{p_1, p_2, p_3\} = \{50, 60, 140\}$, $W = 30$

- Solution
  - Since $\frac{p_1}{w_1} = 10, \frac{p_2}{w_2} = 6, \frac{p_3}{w_3} = 7$
  - Choose all of the 1st item: (5, 50)
  - Choose all of the 3rd item: (20, 140)
  - Choose half of the 2nd item: (10/2, 60/2)

Time Complexity: O(nlogn)

- Does the greedy algorithm always find an optimal solution to the fractional knapsack problem? What about the 0-1 knapsack problem?

0-1 Knapsack Problem Example: n = 6, W = 10

	1	2	3	4	5	6
p_i	6	4	5	3	9	7
w_i	4	2	3	1	6	4

$$P(i,w) = \begin{cases} 0, & \text{if } i = 0 \text{ or } w = 0 \\ P(i-1,w), & \text{if } w_i > w \\ \max\{\, P(i-1,w),\ p_i + P(i-1,w-w_i)\,\}, & \text{if } i > 0 \text{ and } w_i \le w \end{cases}$$

P	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	6	6	6	6	6	6	6
2	0	0	4	4	6	6	10	10	10	10	10
3	0	0	4	5	6	9	10	11	11	15	15
4	0	3	4	7	8	9	12	13	14	15	18
5	0	3	4	7	8	9	12	13	14	16	18
6	0	3	4	7	8	10	12	14	15	16	**19**

Q	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	**0**	0	0	0	1	1	1	1	1	1	1
2	0	0	**1**	1	0	0	1	1	1	1	1
3	0	0	0	1	0	**1**	0	1	1	1	1
4	0	1	0	1	1	0	**1**	1	1	0	1
5	0	0	0	0	0	0	**0**	0	0	1	0
6	0	0	0	0	0	1	0	1	1	0	**1**

?	0	1	2	3	4	5	6	7	8	9	10
0											
1	**0**										
2			**2**								
3						**5**					
4							**6**				
5							**6**				
6											**10**

Time Complexity: O(nW)

Selected items: i = 2, 3, 4, 6
Obtained profit: 19

0-1 Knapsack Problem Example: n = 6, W = 10

- **0-1 knapsack (dynamic programming)**

	1	2	**3**	**4**	5	6
pi	4	5	**12**	**3**	4	3
wi	4	2	**9**	**1**	6	2

Selected items: i = 3, 4
Obtained profit: 15
Time Complexity: O(nW)

- **Fractional knapsack (greedy)**

	4	**2**	**6**	**3**	1	5
pi	**3**	**5**	**3**	**12**	4	4
wi	**1**	**2**	**2**	**9**	4	6
pi/wi	**3.000**	**2.500**	**1.500**	**1.333**	1.000	0.667

Selected items: i = 4, 2, 6, 3(5)
Obtained profit: 17.67
Time Complexity: O(n log n)

- **0-1 knapsack (greedy)**

	4	**2**	**6**	3	1	5
pi	**3**	**5**	**3**	12	4	4
wi	**1**	**2**	**2**	9	4	6
pi/wi	**3.000**	**2.500**	**1.500**	1.333	1.000	0.667

Selected items: i = 4, 2, 6
Obtained profit: 11
Time Complexity: O(n log n)

15.3 Huffman Codes

Huffman Codes

- Huffman codes are used to compress data.
- The (raw) data arrive as a sequence of characters .
- Huffman's greedy algorithm uses a table giving how often each character occurs (its frequency) to build up an optimal way of representing each character as a binary string.

- Suppose we have a 100,000-character data file that you wish to store compactly and you know that the 6 distinct characters in the file occur with the following frequencies.

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

- If a character takes up a byte, this file will need 100,000 bytes of disk space.

Huffman Codes

- Instead of storing the characters as they are, we want to assign a binary code (called codeword) for each characters.
- Then, we convert a raw file into a compressed file, by encoding each character with the binary code.
- The binary codes should be assigned so that the compressed file can be converted back to the raw file.

Huffman Codes: Codewords

- Fixed-length codes
  - We encode every character with a codeword with a fixed number of bits.
  - If we have n characters, we need $\lceil \log n \rceil$ bits to represent the characters.
  - For 6 characters, we need 3 bits. For example,
  - a = 000, b = 001, c = 010, d = 011, e = 100, and f = 101.
  - Using this method, we need 300,000 bits to store the file (compared to 800,000 bits to store the raw file.)
- Variable-length codes
  - We encode characters using codewords with different lengths.
  - We give frequent characters short codewords and infrequent characters long codewords.
  - a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
  - Using this method, we need 224,000 bits to store the file.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Huffman Codes: Finding Optimal Codes

- Given a file, our goal is to find a codeword set which can represent the file in the least number of bits.

- When using a variable-length code, we need to use a prefix-free code in order to be able to restore the compressed file back into the raw file.

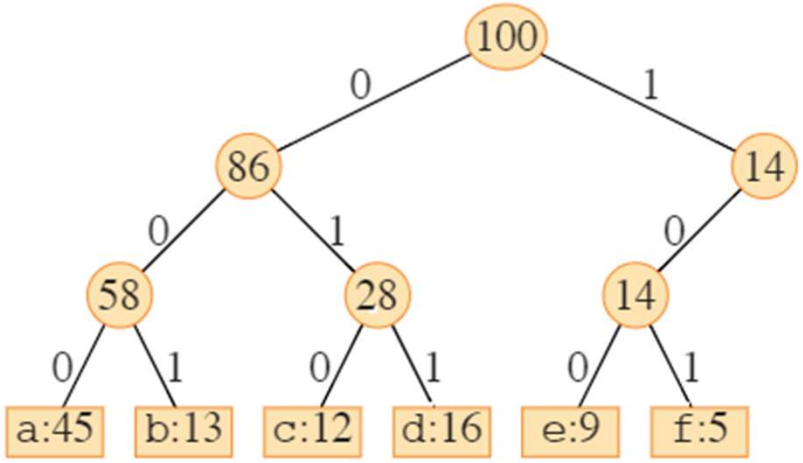
- A prefix-free code: no codeword is a prefix of another codeword.
  - We do not need to use a delimiter if we use a prefix-free code for compression.

- A non-prefix-free code example
  - Suppose we assign a = 0, b = 1, and c = 01.
  - The raw file contains characters "abc", so we encode the file as "0101".
  - When converting back to the raw file, we do not know if the raw file is "abc", "abab", or "cc".
  - If we want to use a non-prefix-free code, we need explicit delimiters that separates each character → not desirable.

Huffman Codes: Binary Tree Representation

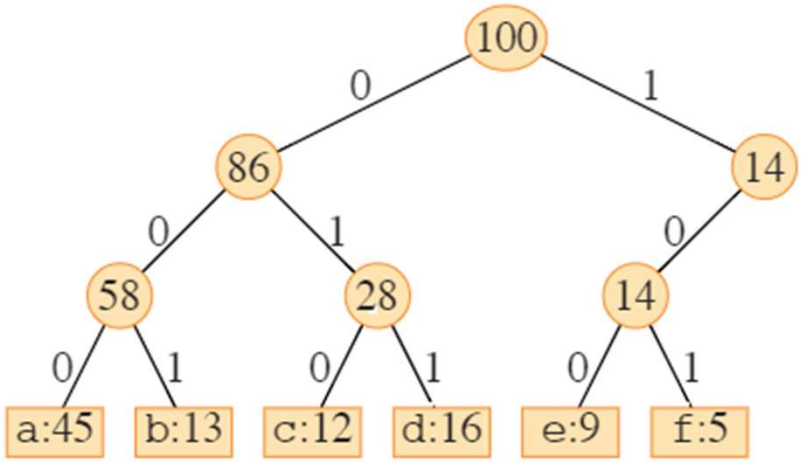
- The decoding process needs a convenient representation for the prefix-free code. A binary tree whose leaves are the given characters provides on such representation.

- Binary representation for the fixed-length code
  - The leaves represent the characters.
  - The numbers in the node indicates cumulative frequency of characters in the subtree.
  - To decode a codeword, we start from the root and follow the paths until we reach the leaf.



Huffman Codes: Binary Tree Representation

- An optimal code for a file is always represented by a full binary tree.
  - Every nonleaf node has two children.
- Proof
  - Let T be a nonfull binary tree. Then T contains an internal node u with only one child, v. Replace edge (u, v) by edges from u to the child or children of v, obtaining a tree T'. Tree T' represents a better coding than T.



Huffman Codes: Binary Tree Representation

- Since we can restrict our attention to full binary trees, we can say this:
- If C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix-free code has exactly $|C|$ leaves, and exactly $|C| - 1$ internal nodes.

- Given a tree T corresponding to a prefix-free code, we can compute the number of bits required to encode a file.
- For each character in the alphabet C, let the attribute $c.freq$ denote the frequency of c in the file and let $d_T(c)$ is also the length of the codeword for character c.
- The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

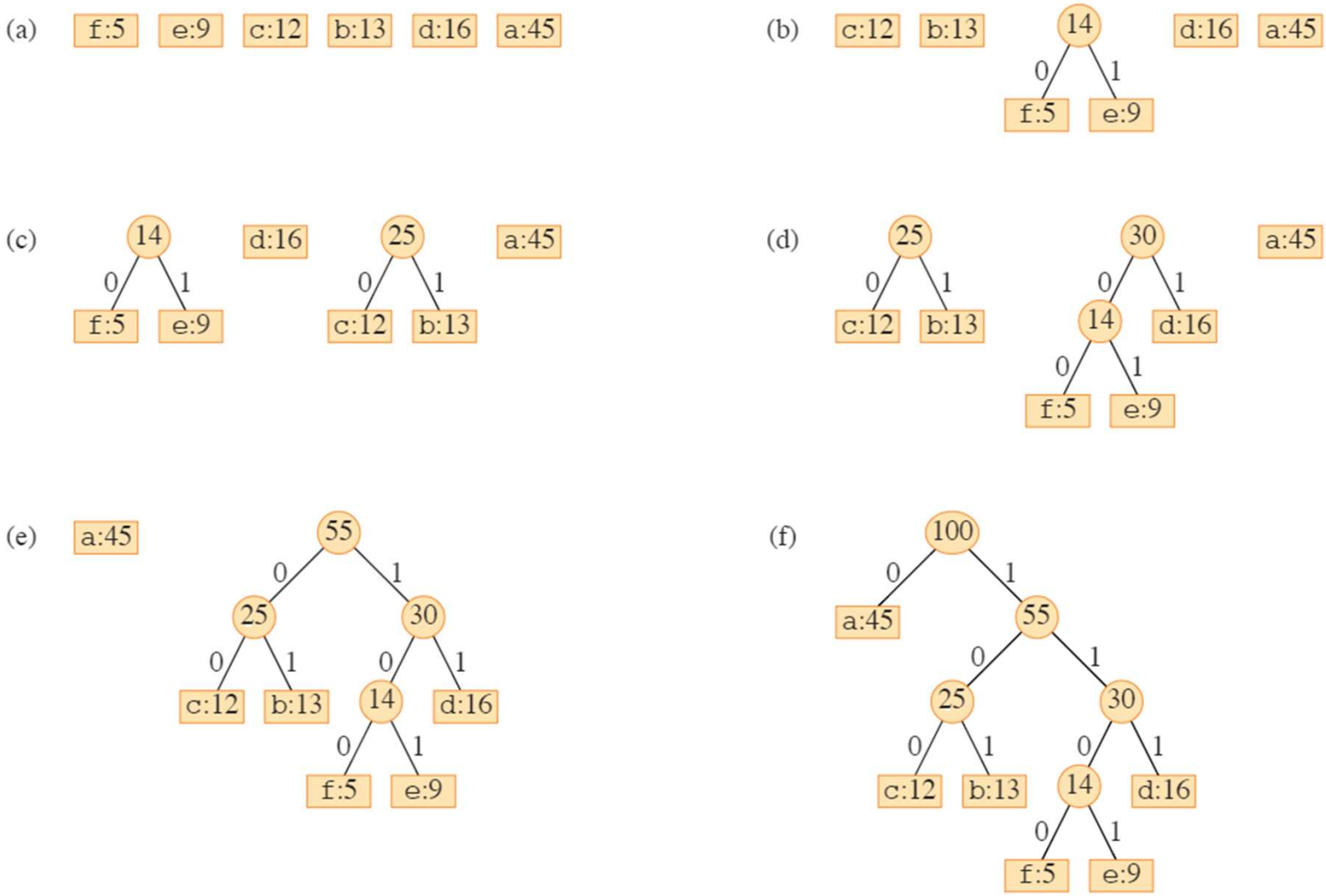
  - which we define as the cost of the tree T.

Constructing a Huffman Code

- Huffman code is an optimal prefix-free code named after David A. Huffman who invented a greedy algorithm to construct the code.
- The procedure for constructing a Huffman code tree

```
HUFFMAN(C)
 1  n = |C|
 2  Q = C
 3  for i = 1 to n − 1
 4      allocate a new node z
 5      x = EXTRACT-MIN(Q)
 6      y = EXTRACT-MIN(Q)
 7      z.left = x
 8      z.right = y
 9      z.freq = x.freq + y.freq
10      INSERT(Q, z)
11  return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

Constructing a Huffman Code



Correctness of Huffman's Algorithm

- Lemma 15.2 (Optimal prefix-free codes have greedy-choice property)
  - Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then, there exists an optimal prefix-free code for C in which the codewords for x and y have the same length and differ only in the last bit.

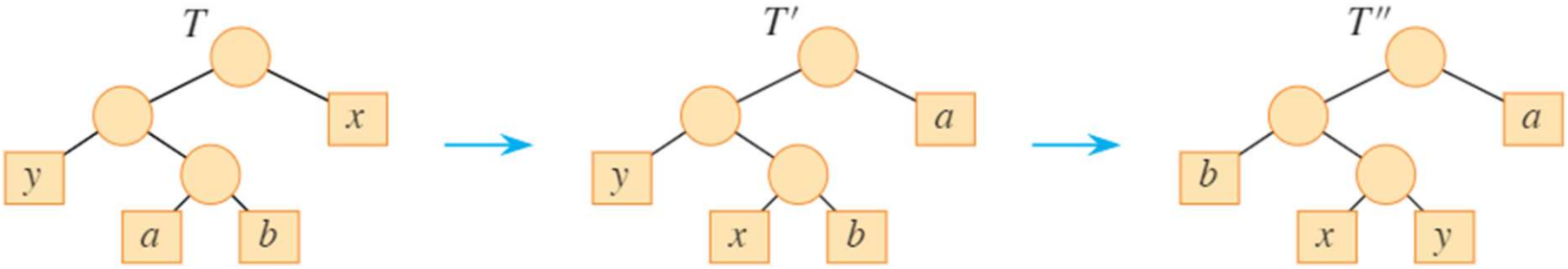
- Idea of the Proof
  - Suppose we have a tree T representing an arbitrary optimal prefix-free code.
  - We modify T to make a tree representing another optimal prefix-free code such that the characters x and y appear as sibling leaves of maximum depth in the new tree.
  - In such a tree, the codewords for x and y have the same length and differ only in the last bit.

Correctness of Huffman's Algorithm

- Proof
  - Let a and b be any two characters that are sibling leaves of maximum depth in T.
  - Without loss of generality, assume that $a.freq \leq b.freq$ and $x.freq \leq y.freq$.
  - Since $x.freq$ and $y.freq$ are the two lowest leaf frequencies, in order, and $a.freq$ and $b.freq$ are two arbitrary frequencies, in order, we have $x.freq \leq a.freq$ and $y.freq \leq b.freq$.
  - If $x.freq = b.freq$, it implies $a.freq = b.freq = x.freq = y.freq$. In this case, the lemma will be trivially true. Thus, we consider the case where $x.freq \neq b.freq$, which means $x \neq b$.

Correctness of Huffman's Algorithm

- Proof (cont.)
  - Imagine exchanging the positions in T of a and x to produce a tree T', and then exchanging the positions in T' of b and y to produce a tree T'' in which x and y are sibling leaves of maximum depth.



  - The difference in cost between T and T' is:

$$
\begin{aligned}
& B(T) - B(T') \\
&\quad = \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&\quad = (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\quad \geq 0 \,,
\end{aligned}
$$

Correctness of Huffman's Algorithm

- Proof (cont.)
  - Similarly, exchanging y and b does not increase the cost, and so $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T') \leq B(T)$.
  - Since T is optimal, we can say $B(T'') = B(T') = B(T)$.
  - Thus, T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows.

- Implications of Lemma 15.2
  - Lemma 15.2 implies that the process of building up an optimal tree by mergers can begin with the greedy choice of merging together those two characters of lowest frequency.

Correctness of Huffman's Algorithm

- Lemma 15.3 (Optimal prefix-free code have the optimal substructure property)
  - Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be the two characters in C with minimum frequency.
  - Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = (C - \{x, y\}) \cup \{z\}$.
  - $z.freq = x.freq + y.freq$
  - Let T' be any tree representing an optimal prefix-free code for alphabet C'.
  - Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix-free code for the alphabet C.

Correctness of Huffman's Algorithm

- Proof
  - We first show how to express the cost $B(T)$ in terms of cost $B(T')$.
  - For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $c.freq.d_T(c) = c.freq.d_{T'}(c)$.
  - Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) , \end{aligned}$$

  - from which we conclude that
  - $B(T) = B(T') + x.freq + y.freq$

  - or, equivalently,
  - $B(T') = B(T) - x.freq - y.freq$

Correctness of Huffman's Algorithm

- Proof
  - Now we prove the lemma by contradiction.
  - Suppose that T does not represent an optimal prefix-free code for C.
  - Then there exists an optimal tree T'' such that $B(T'') < B(T)$.
  - According to Lemma 15.2, T'' has x and y as siblings.
  - Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $z.freq = x.freq + y.freq$. Then,

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') , \end{aligned}$$

  - yielding a contradiction to the assumption that T' represents an optimal prefix-free code for C'.
  - Thus, T must represent an optimal prefix-free code for the alphabet C.

Correctness of Huffman's Algorithm

- Theorem 15.4
  - Procedure HUFFMAN produces an optimal prefix-free code.

- Proof
  - Immediate from Lemmas 15.2 and 15.3.

Huffman's Algorithm: Implementation

- Implementation issues
  - How can you manage a dynamic set to which the following operations occur frequently:
    - Delete the elements with the highest priority from the list.
    - Insert an element with some priority into the list.
    - ✓ The answer is to use Priority Queue.
  - The priority queue can be implemented in many ways. Which one would you use?

Representation	Insertion	Deletion
Unordered array	$O(1)$	$O(n)$
Unordered linked list	$O(1)$	$O(n)$
Sorted array	$O(n)$	$O(1)$
Sorted linked list	$O(n)$	$O(1)$
Heap	$O(\log n)$	$O(\log n)$

  - ✓ The answer is to use the priority queue based on (min) heap.

Huffman's Algorithm: Implementation

- Time complexity
  - O(n log n)

```
typedef struct _node {
  char symbol;
  int freq;
  struct _node *left;
  struct _node *right;
} NODE;
NODE *u, *v, *w;
…
for (i = 1; i <= n; i++) {
  /* insert the n single-node trees */
}
for (i = 1; i <= n-1; i++) {
  u = PQ_delete();
  v = PQ_delete();
  w = make_a_new_node();
  w->left = u;
  w->right = v;
  w->freq = u->freq + v->freq;
  PQ_insert(w);
}
w = PQ_delete();
/* u points to the optimal tree. */
```

Extra: Scheduling with Greedy Methods

Scheduling: Minimizing Total Time in the System

- Problem
  - Consider a system in which a server is about to serve n clients. Let $T = \{t_1, t_2, \ldots, t_n\}$ be a set of positive numbers, where t_i is the estimated time-to-completion for the ith client. What is the optimal order of service where the total (wait+service) time in the system is **minimized**?
  - Other applications of scheduling: hair stylist with waiting clients, pending operations on a shared hard disk, etc.

- Example: $T = \{t_1, t_2, t_3\} = \{5, 10, 4\}$

Schedule	Total Time in the System
[1, 2, 3]	5 + (5 + 10) + (5 + 10 + 4) = 39
[1, 3, 2]	33
[2, 1, 3]	10 + (10 + 5) + (10 + 5 + 4) = 44
[2, 3, 1]	43
[3, 1, 2]	☞ 4 + (4 + 5) + (4 + 5 + 10) = 32
[3, 2, 1]	37

Scheduling: Minimizing Total Time in the System

- A naive approach
  - Enumerate all possible schedules of service, and select the optimal one.
  - Time complexity: O(n!)

- A greedy approach
  - Sort T in non-decreasing order to get the optimal schedule → O(n log n)

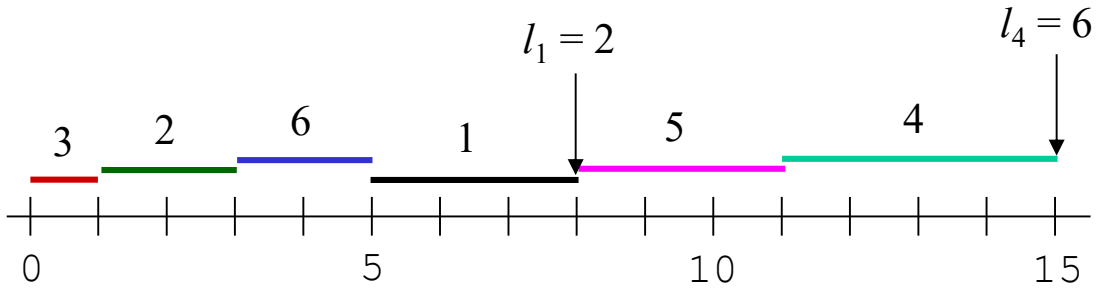
- Does the greedy approach always find a schedule that minimizes the total time in the system?
  - Let $S = [s_1, s_2, \ldots, s_n]$ be an optimal schedule. If they are not scheduled in nondecreasing order, then, for at least one i ($1 \le i \le n-1$), $s_i > s_{i+1}$.
  - Now consider the schedule $S' = [s_1, s_2, \ldots, s_{i+1}, s_i, \ldots, s_n]$ that is obtained by interchanging s_i and s_{i+1}. …

$$
\begin{aligned}
C &= s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \cdots + (s_1 + s_2 + \cdots + s_n) \\
&= n \cdot s_1 + (n-1) \cdot s_2 + \cdots + 2 \cdot s_{n-1} + 1 \cdot s_n \\
&= \sum_{i=1}^{n} (n+1-i) \cdot s_i = (n+1) \sum_{i=1}^{n} s_i - \sum_{i=1}^{n} i \cdot s_i
\end{aligned}
$$

Scheduling: Minimizing Lateness

- Problem
  - Let $J = \{1, 2, \ldots, n\}$ be a set of jobs to be served by a single processor.
  - The ith job takes t_i units of processing time, and is due at time d_i.
  - When the ith job starts at time s_i, its lateness $l_i = \max\{0, s_i + t_i - d_i\}$.
  - Goal: Find a schedule S so as to minimize the maximum lateness $L = \max\{l_i\}$.
- Example
  - $S = \{3, 2, 6, 1, 5, 4\}$ → maximum lateness = 6

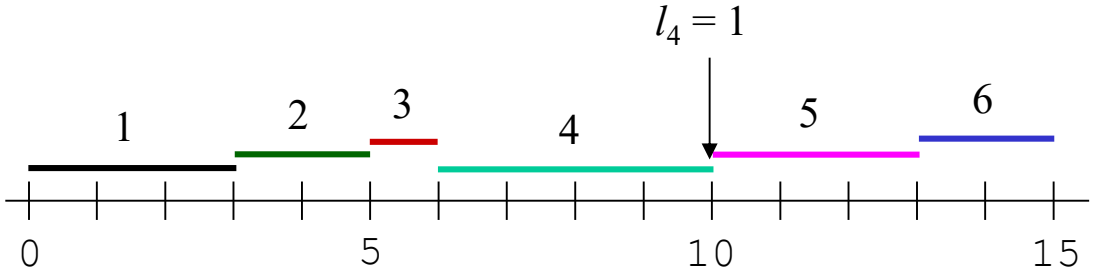
Job	t_i	d_i
1	3	6
2	2	8
3	1	9
4	4	9
5	3	14
6	2	15



Scheduling: Minimizing Lateness

- Possible greedy approaches
  - Sort jobs in non-decreasing order of processing time t_i: shortest-job-first
  - Sort jobs in non-decreasing order of slack time $d_i - t_i$: smallest-slack-time-first
  - Sort jobs in non-decreasing order of deadline d_i: earliest-deadline-first
    - EDF scheduling: S = {1, 2, 3, 4, 5, 6} → maximum lateness = 1

Job	t_i	d_i
1	3	6
2	2	8
3	1	9
4	4	9
5	3	14
6	2	15



Earliest-Deadline-First algorithm

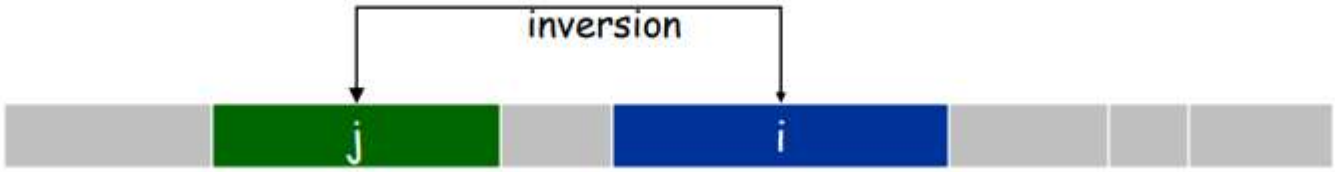
- Is the EDF the optimal algorithm for minimizing the maximum lateness?

- We can prove this by the “exchange argument”
  - If we exchange the two schedules...

Earliest-Deadline-First algorithm

- Preparing for the proof

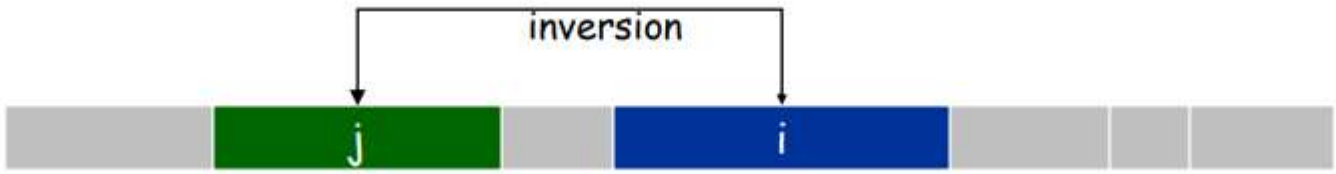
- Inversion
  - An inversion in schedule S is a pair of jobs i and j such that $d_i < d_j$ but j is scheduled before i.

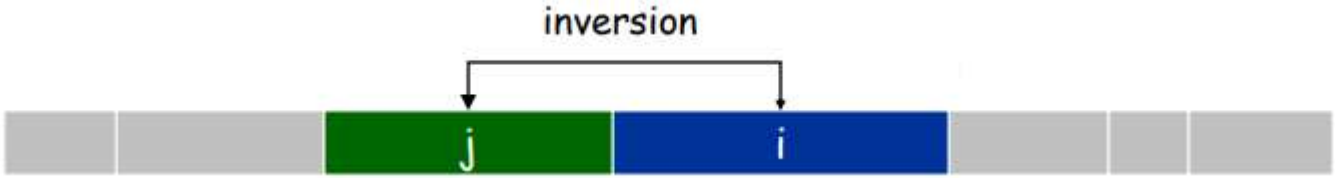


- Does EDF create an inversion?

Earliest-Deadline-First algorithm

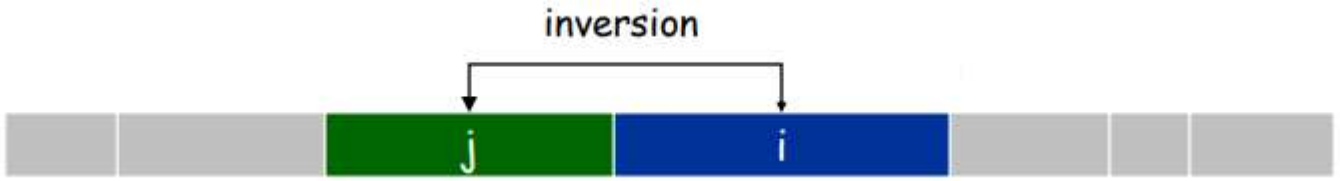
- Claim: If a schedule has an inversion, then there is a pair of inverted jobs scheduled consecutively.
  - If we walk through the schedule from j to i, eventually at some point the deadline decreases
  - Those two adjacent schedules are inverted.



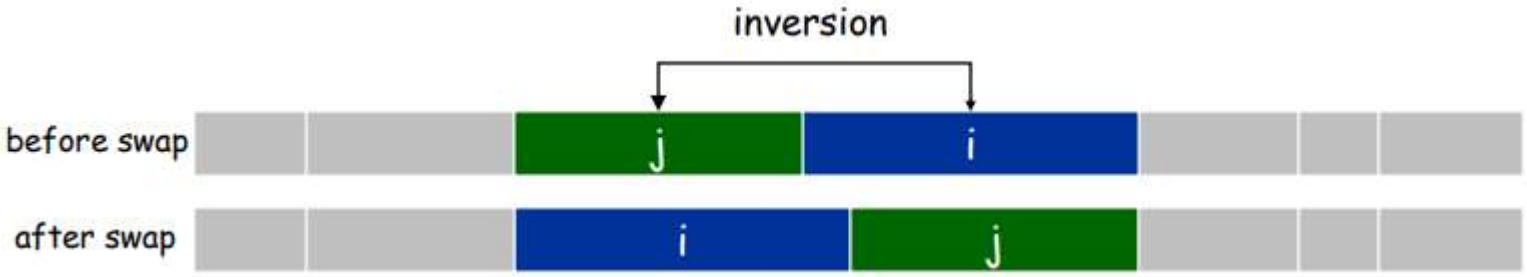


Earliest-Deadline-First algorithm

- Suppose we have a pair of adjacent schedules that are inverted.

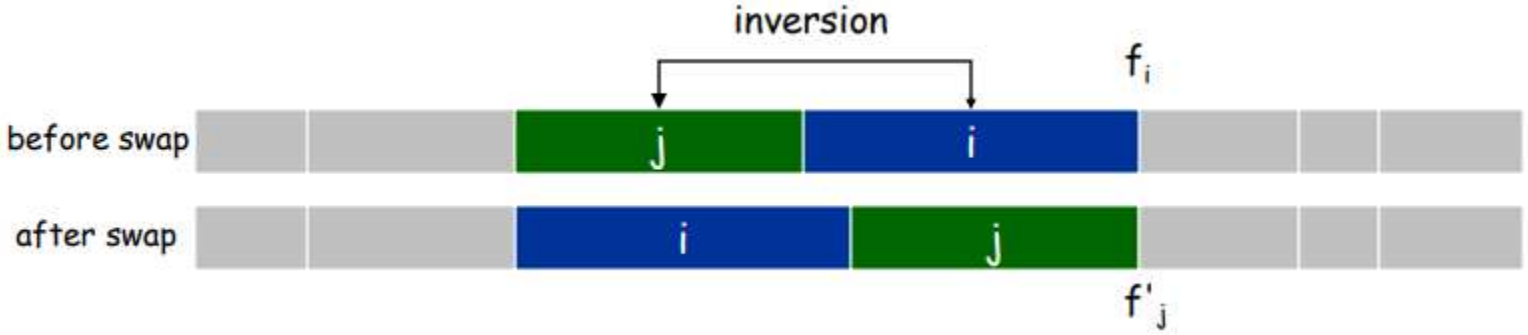


- What happens if we swap schedules of these two jobs?
  - what happens to the lateness of these two jobs?



Earliest-Deadline-First algorithm

- If we swap the inverted schedules



  - Lateness of all other jobs are the same
  - Lateness of i is smaller $\ell'_i \le \ell_i$
  - Lateness of j is larger, but

$$
\begin{aligned}
l'_j &= f'_j - d_j && \text{(definition)} \\
&= f_i - d_j && (j \text{ finishes at time } f_i) \\
&\le f_i - d_i && (d_i < d_j) \\
&\le l_i && \text{(definition)}
\end{aligned}
$$

Earliest-Deadline-First algorithm

- Thus, if we swap the inverted schedules, we get a schedule where
  - Number of inversions is decreased by one.
  - The maximum lateness does not increase.

- The optimal schedule does not have inversions → EDF.

Scheduling with Deadlines

- Problem
  - Let $J = \{1, 2, \ldots, n\}$ be a set of jobs to be served.
  - Each job takes one unit of time to finish.
  - Each job has a deadline and a profit.
    - If the job starts before or at its deadline, the profit is obtained.
  - ➢ Schedule the jobs so as to **maximize** the total profit (not all jobs have to be scheduled).

- Example:

Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40

Schedule	Total Profit
[1, 3]	30 + 25 = 55
[2, 1]	35 + 30 = 65
[2, 3]	35 + 25 = 60
[3, 1]	25 + 30 = 55
[4, 1]	☞ 40 + 30 = 70
[4, 3]	40 + 25 = 65

Scheduling with Deadlines

- A greedy algorithm
  - Sort the jobs in non-increasing order by profit.
  - Scan each job in the sorted list, adding it to the schedule if possible.
- Example
  - S = EMPTY
  - Is S = {1} OK?
    - Yes: $S \leftarrow$ {1} ([1])
  - Is S = {1, 2} OK?
    - Yes: $S \leftarrow$ {1, 2} ([2, 1])
  - Is S = {1, 2, 3} OK?
    - No.
  - Is S = {1, 2, 4} OK?
    - Yes: $S \leftarrow$ {1, 2, 4} ([2, 1, 4] or [2, 4, 1])
  - Is S = {1, 2, 4, 5} OK?
    - No.
  - Is S = {1, 2, 4, 6} OK?
    - No.
  - Is S = {1, 2, 4, 7} OK?
    - No.

<After sorting by profit>

Job	Deadline	Profit
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

Scheduling with Deadlines

- Terminology
  - A sequence is called a feasible sequence if all the jobs in the sequence end by their deadlines.
  - A set of jobs is called a feasible set if there exists at least one feasible sequence for the jobs in the set.
  - A sequence is called an optimal sequence if it is a feasible sequence with the maximum total profit.
  - A set of jobs is called an optimal set of jobs if there exists at least one optimal sequence for the jobs in the set.

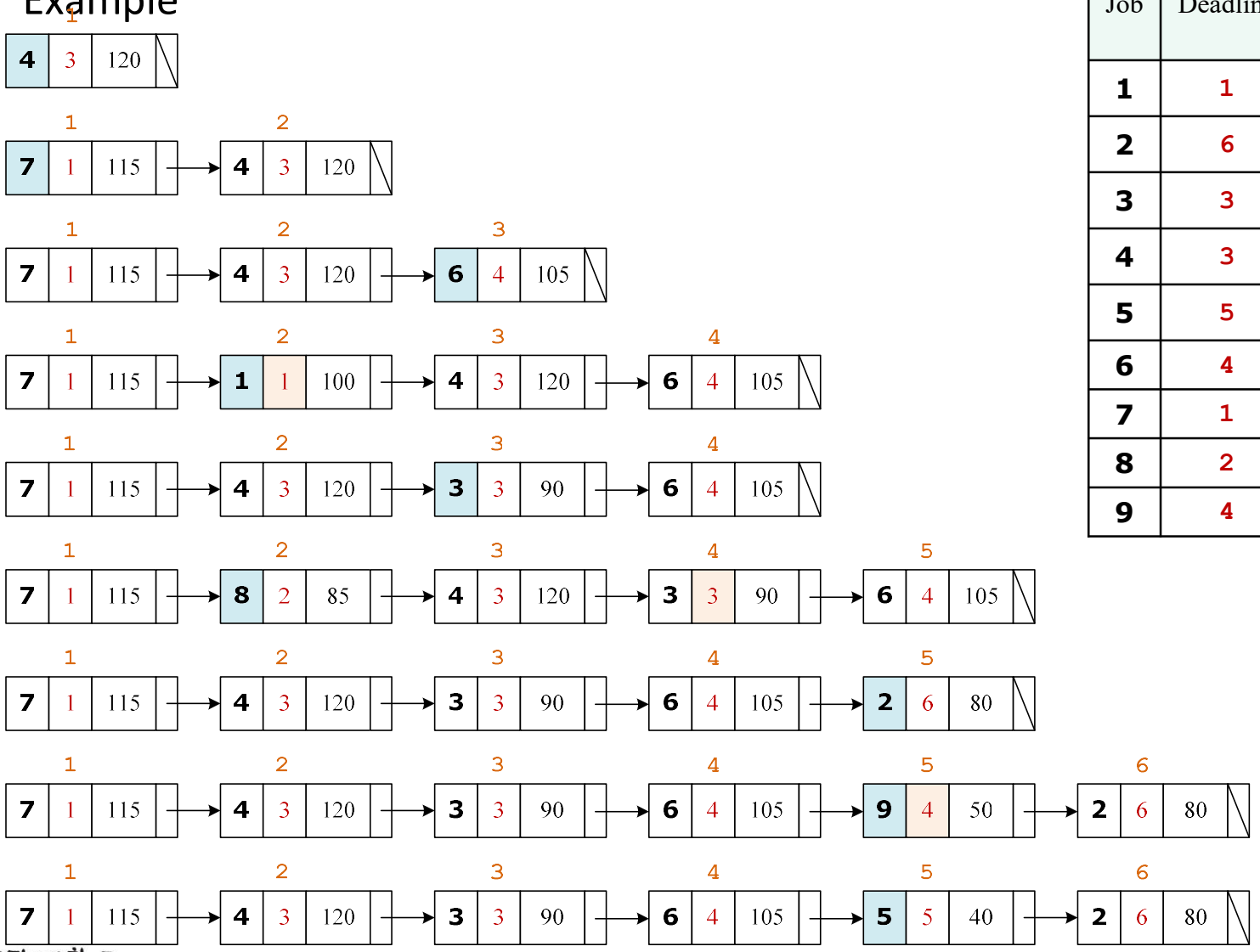
Scheduling with Deadlines: Implementation

- A key operation in the greedy approach
  - Determine if a set of jobs S is feasible.
  - Fact: S is feasible if an only if the sequence obtained by ordering the jobs in S according to non-decreasing deadlines is feasible.
  - Ex:
    - Is *S* = {1, 2, 4} OK? → [2(1), 1(3), 4(3)] → Yes!
    - Is *S* = {1, 2, 4, 7} OK? → [2(1), 7(2), 1(3), 4(3)] → No
- An $O(n^2)$ implementation
  - Sort the jobs in non-increasing order by profit.
  - For each job in the sorted order,
    - See if the current job can be scheduled together with the previously selected jobs, using a *linked list* data structure.
      - If yes, add it to the list of feasible sequence.
      - Otherwise, reject it.
  - Time complexity
    - When there are i-1 jobs in the sequence,
      - at most *i*-1 comparisons are needed to add a new job in the sequence, and
      - at most *i* comparisons are needed to check if the new sequence is feasible.

$$O(n \log n) + \sum_{i=2}^{n} \{ (i-1) + i \} = O(n^2)$$

Scheduling with Deadlines: Implementation

- Example



Job	Deadline	Profit
1	1	100
2	6	80
3	3	90
4	3	120
5	5	40
6	4	105
7	1	115
8	2	85
9	4	50

End of Class

Questions?

Instructor office: AS-1013

Email: jso1@sogang.ac.kr