Design and Analysis of Algorithms
Part III: Greedy Algorithms

Lecture 9: Fraction Knapsack, Huffman Coding and Activity Selection Problems

Yongxin Tong (童咏昕)

School of CSE, Beihang University

yxtong@buaa.edu.cn

Outline

- Introduction to Part III
- The Fractional Knapsack Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
- The Huffman Coding Problem
  - Problem Definition
  - A Greedy Algorithm
- The Activity Selection Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
  - Extended: Weighted Activity Selection

Introduction to Greedy Algorithm

- A greedy algorithm for an optimization problem always makes the choice that looks best at the moment and adds it to the current subsolution.

Introduction to Greedy Algorithm

- A greedy algorithm for an optimization problem always makes the choice that looks best at the moment and adds it to the current subsolution.

- Greedy algorithms don't always yield optimal solutions but, when they do, they're usually the simplest and most efficient algorithms available.

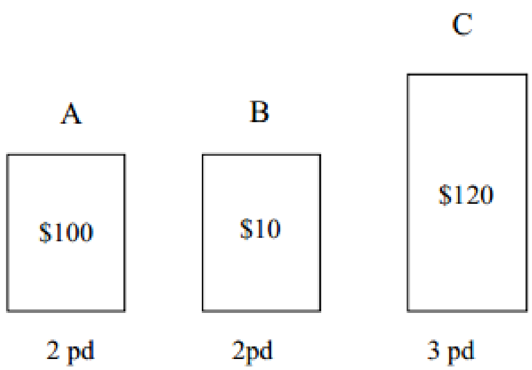
Introduction to Part III

- In Part III, we will illustrate greedy strategies using several examples:
  - Fractional Knapsack Problem (部分背包问题)
  - Huffman Coding Problem (赫夫曼编码问题)
  - Activity Selection Problem (活动选择问题)

Outline

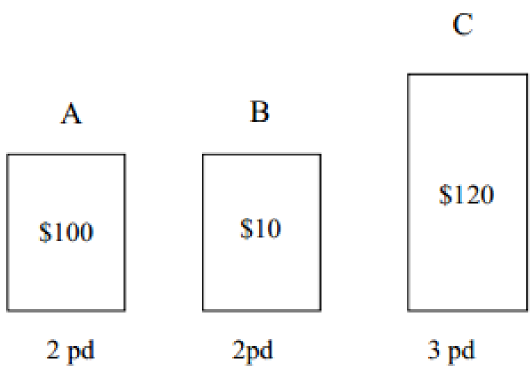
- Introduction to Part III
- Fractional Knapsack Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
- Huffman Coding Problem
  - Problem Definition
  - A Greedy Algorithm
- Activity Selection Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
  - Extended: Weighted Activity Selection

The Knapsack Problem...



Capacity of knapsack: $K = 4$

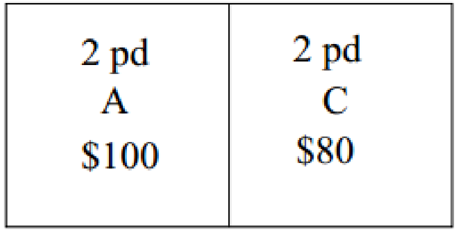
The Knapsack Problem...



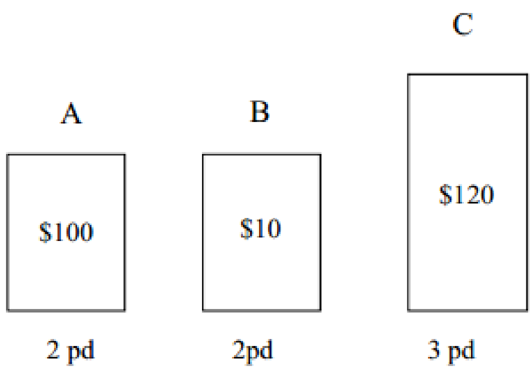
Capacity of knapsack: $K = 4$

Fractional Knapsack Problem:
Can take a fraction of an item.

Solution:



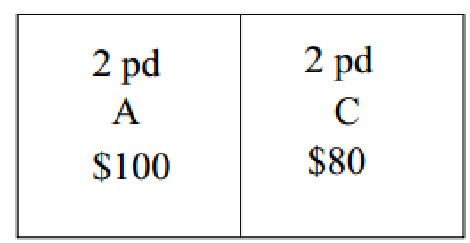
The Knapsack Problem...



Capacity of knapsack: $K = 4$

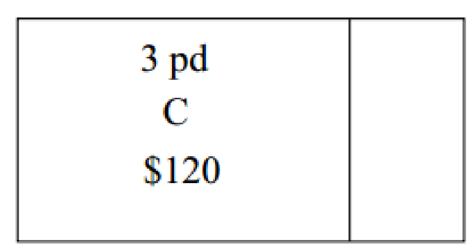
Fractional Knapsack Problem:
Can take a fraction of an item.

Solution:



0-1 Knapsack Problem:
Can only take or leave item. You can't take a fraction.

Solution:



The Fractional Knapsack Problem: Formal Definition

- Given K and a set of n items:

weight	w_1	w_2	$\ldots$	w_n
value	v_1	v_2	$\ldots$	v_n

- Find: $0 \le x_i \le 1$, $i = 1,2,\ldots, n$ such that

$$\sum_{i=1}^{n} x_i w_i \le K$$

and the following is maximized:

$$\sum_{i=1}^{n} x_i v_i$$

Outline

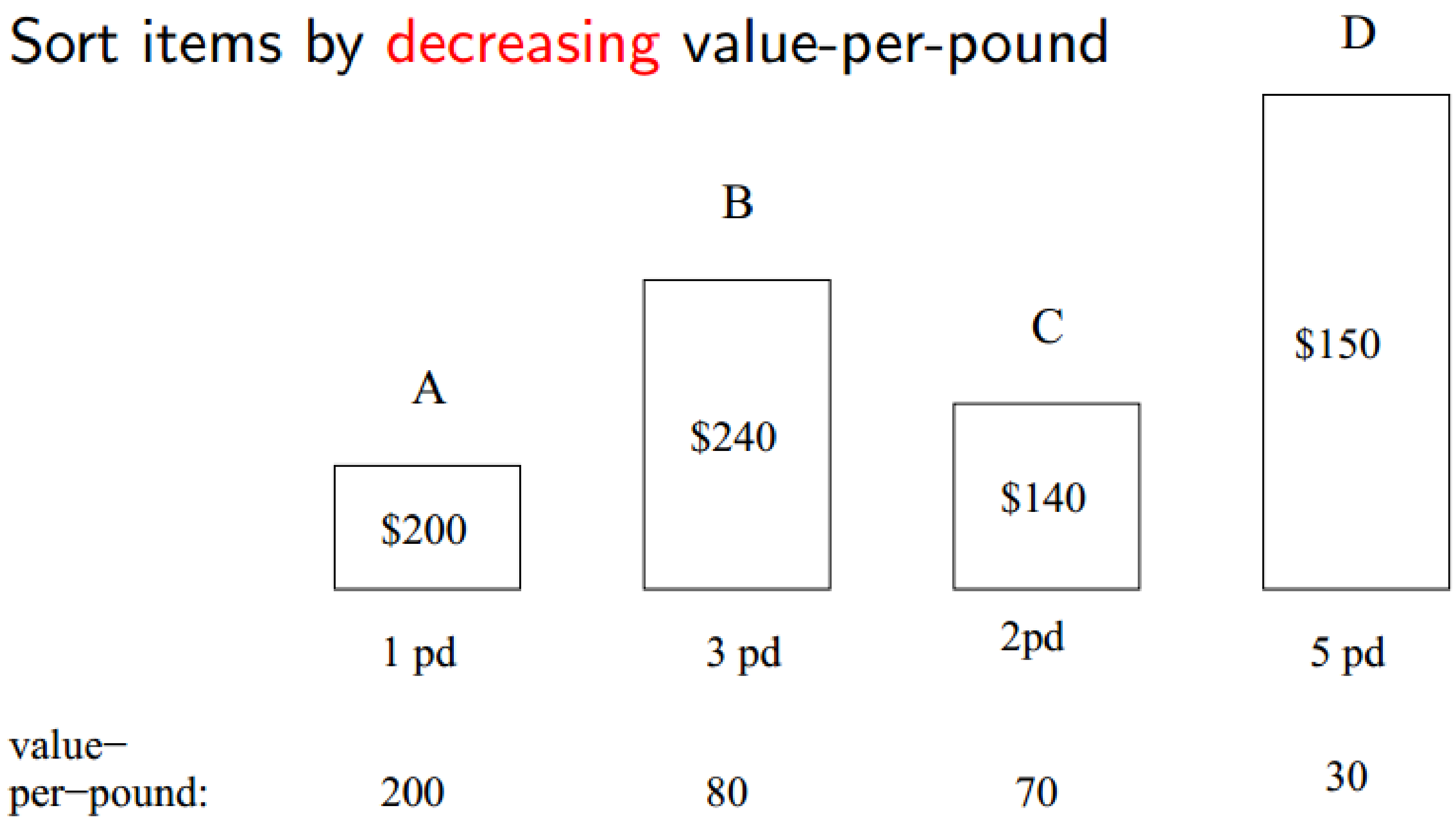
- Introduction to Part III
- Fractional Knapsack Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
- Huffman Coding Problem
  - Problem Definition
  - A Greedy Algorithm
- Activity Selection Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
  - Extended: Weighted Activity Selection

Greedy Solution for Fractional Knapsack

Sort items by decreasing value-per-pound

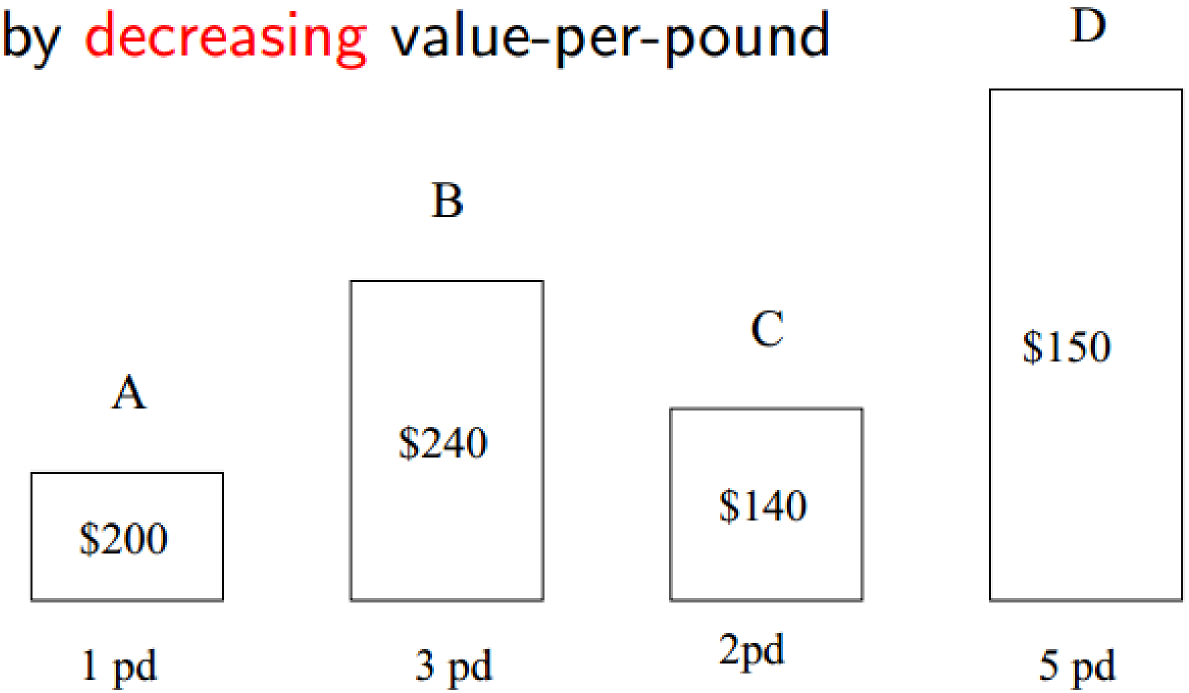
Greedy Solution for Fractional Knapsack

Sort items by decreasing value-per-pound



Greedy Solution for Fractional Knapsack

Sort items by decreasing value-per-pound



value–per–pound:	200	80	70	30

If knapsack holds $K = 5$ pd, solution is:

1	pd	A
3	pd	B
1	pd	C

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, ..., n.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the head of the unselected list.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the head of the unselected list.
  - If $k \geq w_i$, set $x_i = 1$ (we take item i), and reduce $k = k - w_i$, then consider the next unselected item.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the head of the unselected list.
  - If $k \geq w_i$, set $x_i = 1$ (we take item i), and reduce $k = k - w_i$, then consider the next unselected item.
  - If $k < w_i$, set $x_i = k/w_i$ (we take a fraction k/w_i of item i), Then the algorithm terminates.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the head of the unselected list.
  - If $k \geq w_i$, set $x_i = 1$ (we take item i), and reduce $k = k - w_i$, then consider the next unselected item.
  - If $k < w_i$, set $x_i = k/w_i$ (we take a fraction k/w_i of item i), Then the algorithm terminates.

Running time: O(n log n).

Pseudocode

Fraction-Knapsack(n,v,w,K)

```
Input: Value array v and weight array w of n items, capacity of
       knapsack K.
Output: Solution of maximum value.
Let r[1..n],x[1..n] be two new arrays;
for i ← 1 to n do
|   r[i] ← v[i]/w[i];
|   x[i] ← 0;
end
Sort the items in decreasing order of their ratios r, rename the items if
 necessary so that the sorted order of items is ⟨1, 2, ..., n⟩;
j ← 0;
while K > 0 and j ≤ n do
|   j ← j + 1;
|   if K > w[j] then
|   |   x[j] ← 1;
|   |   K ← K − w[j];
|   end
|   else
|   |   x[j] ← k/w[j];
|   |   break;
|   end
end
return x;
```

Example of Optimal Solution Construction

	1	**2**	**3**	**4**
v_i	60	75	100	120
w_i	10	25	20	30

K = 50

Example of Optimal Solution Construction

	1	**2**	**3**	**4**
v_i	60	75	100	120
w_i	10	25	20	30
r_i	6	3	5	4

K = 50

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6	5	4	3

K = 50

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6			3
x_i	0	0	0	0

K = 50

$w_i < K$

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6	5	4	3
x_i	1	0	0	0

K = 40

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6	5	4	3
x_i	1	1	0	0

K = 20

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6	5	4	
x_i	1	1	0	0

$w_i \geq K$

K = 20

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6	5	4	3
x_i	1	1	2/3	0

K = 20

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6	5	4	3
x_i	1	1	2/3	0

Result

Correctness

- We are given a set of n items {1, 2, …, n}.

Correctness

- We are given a set of n items {1, 2, …, n}.
  - Assume that the items have been sorted by their per-pound values, i.e., $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$.

Correctness

- We are given a set of n items {1, 2, …, n}.
  - Assume that the items have been sorted by their per-pound values, i.e., $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$.
- Let the greedy solution be $G = \langle x_1, x_2, \ldots, x_n \rangle$

Correctness

- We are given a set of n items {1, 2, …, n}.
  - Assume that the items have been sorted by their per-pound values, i.e., $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$.
- Let the greedy solution be $G = \langle x_1, x_2, \ldots, x_n \rangle$
  - x_i indicates the fraction of item i taken.
  - $x_1 \ldots x_{i-1}$ must be equal to 1.

Correctness

- We are given a set of n items {1, 2, …, n}.
  - Assume that the items have been sorted by their per-pound values, i.e., $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$.
- Let the greedy solution be $G = \langle x_1, x_2, \ldots, x_n \rangle$
  - x_i indicates the fraction of item i taken.
  - $x_1 \ldots x_{i-1}$ must be equal to 1.
- Consider any optimal solution $O = \langle y_1, y_2, \ldots, y_n \rangle$

Correctness

- We are given a set of n items {1, 2, …, n}.
  - Assume that the items have been sorted by their per-pound values, i.e., $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$.
- Let the greedy solution be G = <x_1, x_2, …, x_n>
  - x_i indicates the fraction of item i taken.
  - $x_1 \ldots x_{i-1}$ must be equal to 1.
- Consider any optimal solution O = <y_1, y_2, …, y_n>
  - y_i indicates the fraction of item i taken in the optimal solution.

Correctness

- We are given a set of n items {1, 2, …, n}.
  - Assume that the items have been sorted by their per-pound values, i.e., $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$.
- Let the greedy solution be G = <x_1, x_2, …, x_n>
  - x_i indicates the fraction of item i taken.
  - $x_1 \ldots x_{i-1}$ must be equal to 1.
- Consider any optimal solution O = <y_1, y_2, …, y_n>
  - y_i indicates the fraction of item i taken in the optimal solution.
  - Note that $0 \leq y_i \leq 1$, $1 \leq i \leq n$, and the knapsack must be full in both G and O: $\sum_{i=1}^{n} x_i w_i = \sum_{i=1}^{n} y_i w_i = K$.

Correctness

- We are given a set of n items {1, 2, …, n}.
  - Assume that the items have been sorted by their per-pound values, i.e., $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$.
- Let the greedy solution be G = $<x_1, x_2, \ldots, x_n>$
  - x_i indicates the fraction of item i taken.
  - $x_1 \ldots x_{i-1}$ must be equal to 1.
- Consider any optimal solution O = $<y_1, y_2, \ldots, y_n>$
  - y_i indicates the fraction of item i taken in the optimal solution.
  - Note that $0 \leq y_i \leq 1$, $1 \leq i \leq n$, and the knapsack must be full in both G and O: $\sum_{i=1}^{n} x_i w_i = \sum_{i=1}^{n} y_i w_i = K$.
- Consider the first item i where the two selections differ.

Correctness

- We are given a set of n items {1, 2, …, n}.
  - Assume that the items have been sorted by their per-pound values, i.e., $\rho_1 \geq \rho_2 \geq \ldots \geq \rho_n$.
- Let the greedy solution be G = $\langle x_1, x_2, \ldots, x_n \rangle$
  - x_i indicates the fraction of item i taken.
  - $x_1 \ldots x_{i-1}$ must be equal to 1.
- Consider any optimal solution O = $\langle y_1, y_2, \ldots, y_n \rangle$
  - y_i indicates the fraction of item i taken in the optimal solution.
  - Note that $0 \leq y_i \leq 1$, $1 \leq i \leq n$, and the knapsack must be full in both G and O: $\sum_{i=1}^{n} x_i w_i = \sum_{i=1}^{n} y_i w_i = K$.
- Consider the first item i where the two selections differ.
  - By definition, solution G takes a greater amount of item i than solution O (because the greedy solution always takes as much as it can).

Correctness…

- Consider the new solution O’ constructed from solution O as follows:

Correctness…

- Consider the new solution O' constructed from solution O as follows:
  - Set y_i equal to x_i and remove part of any item i + 1 to item n of total weight $(x_i - y_i)\ w_i$. This is always doable because in O, the total weight of items i to n is the same as that in G, which is at least $x_i w_i$.

Correctness…

- Consider the new solution O' constructed from solution O as follows:
  - Set y_i equal to x_i and remove part of any item i + 1 to item n of total weight $(x_i - y_i)\ w_i$. This is always doable because in O, the total weight of items i to n is the same as that in G, which is at least $x_i w_i$.
  - The total value of solution O' is more than or equal to the total value of solution O (since all the subsequent items have lesser or equal ρ).

Correctness...

- Consider the new solution O' constructed from solution O as follows:
  - Set y_i equal to x_i and remove part of any item i + 1 to item n of total weight $(x_i - y_i)\ w_i$. This is always doable because in O, the total weight of items i to n is the same as that in G, which is at least $x_i w_i$.
  - The total value of solution O' is more than or equal to the total value of solution O (since all the subsequent items have lesser or equal ρ).
  - Since O is an optimal solution, the value of O' cannot be better than that of O, so O and O' must have the same value.

Correctness…

- Consider the new solution O’ constructed from solution O as follows:
  - Set y_i equal to x_i and remove part of any item i + 1 to item n of total weight $(x_i - y_i)\ w_i$. This is always doable because in O, the total weight of items i to n is the same as that in G, which is at least $x_i w_i$.
  - The total value of solution O’ is more than or equal to the total value of solution O (since all the subsequent items have lesser or equal ρ).
  - Since O is an optimal solution, the value of O’ cannot be better than that of O, so O and O’ must have the same value.
  - Thus solution O’ is also optimal.

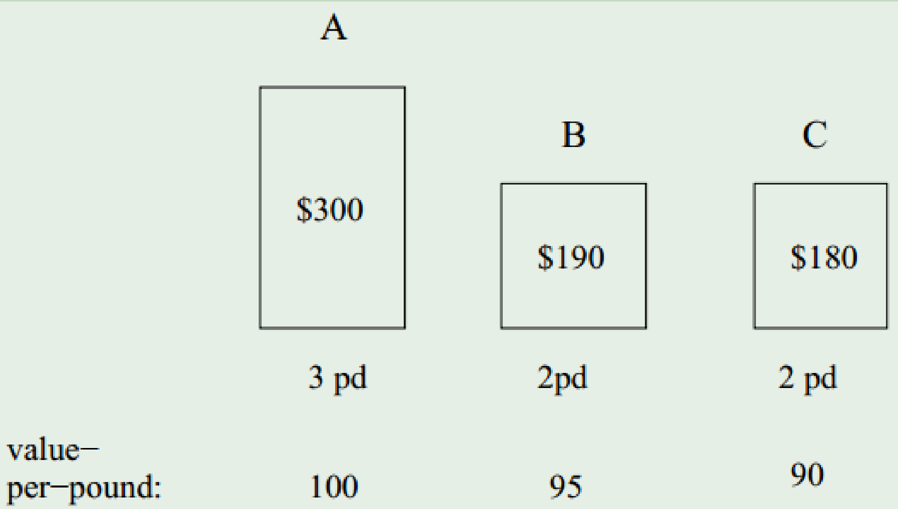
Correctness…

- Consider the new solution O' constructed from solution O as follows:
  - Set y_i equal to x_i and remove part of any item i + 1 to item n of total weight $(x_i - y_i)\, w_i$. This is always doable because in O, the total weight of items i to n is the same as that in G, which is at least $x_i w_i$.
  - The total value of solution O' is more than or equal to the total value of solution O (since all the subsequent items have lesser or equal ρ).
  - Since O is an optimal solution, the value of O' cannot be better than that of O, so O and O' must have the same value.
  - Thus solution O' is also optimal.
- By repeating this process, we will eventually convert O into G, without changing the total value of the selection. Therefore G is also optimal.

Greedy solution for 0-1 Knapsack Problem?

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

Example

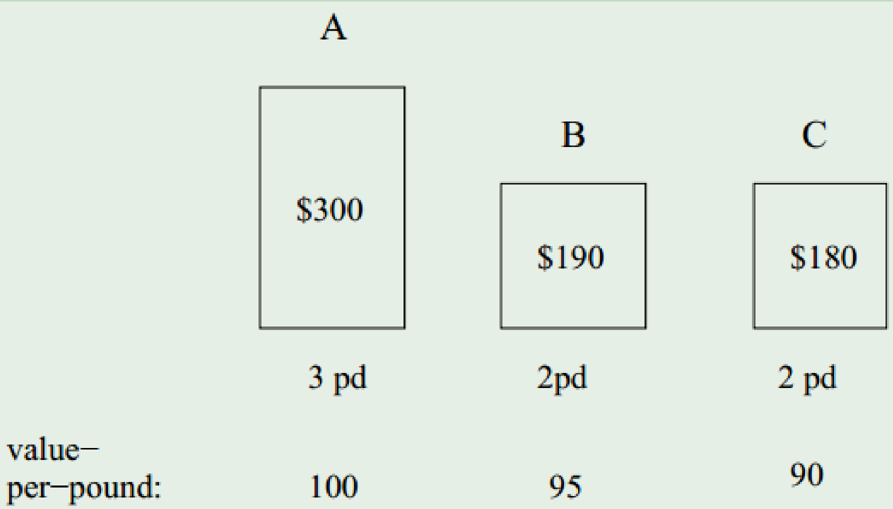


$K = 4$. Solution is item B + item C

Greedy solution for 0-1 Knapsack Problem?

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

Example



$K = 4$. Solution is item B + item C

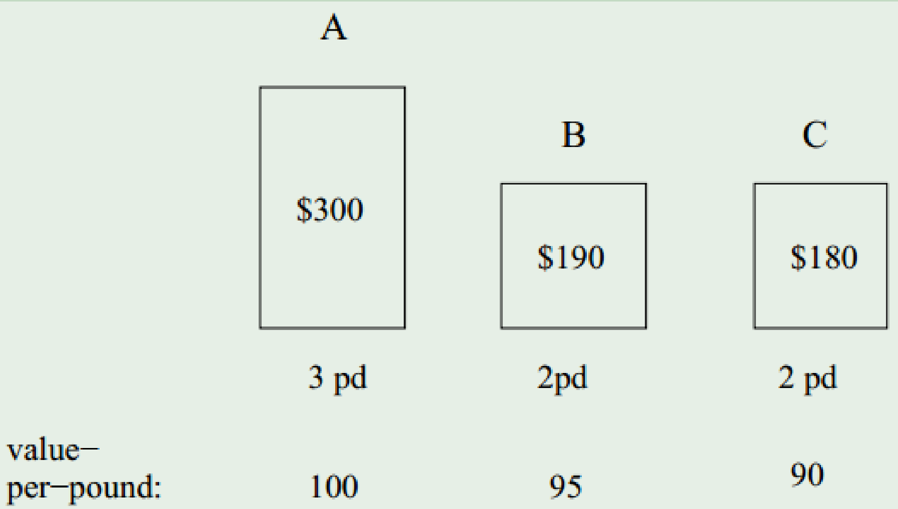
Question

Suppose we try to prove the greedy algorithm for 0-1 knapsack problem is correct. We follow exactly the same lines of arguments as fractional knapsack problem.

Greedy solution for 0-1 Knapsack Problem?

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

Example



$K = 4$. Solution is item B + item C

Question

Suppose we try to prove the greedy algorithm for 0-1 knapsack problem is correct. We follow exactly the same lines of arguments as fractional knapsack problem. Of course, it must fail. Where is the problem in the proof?

Outline

- Introduction to Part III
- Fractional Knapsack Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
- Huffman Coding Problem
  - Problem Definition
  - A Greedy Algorithm
- Activity Selection Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
  - Extended: Weighted Activity Selection

Encoding

Example

Suppose that we have a $100,000$ character data file that we wish to store. The file contains only 6 characters, appearing with the following frequencies:

	a	b	c	d	e	f
Frequency in '000s	45	13	12	16	9	5

Encoding

Example

Suppose that we have a $100,000$ character data file that we wish to store. The file contains only 6 characters, appearing with the following frequencies:

	a	b	c	d	e	f
Frequency in '000s	45	13	12	16	9	5

- A binary code encodes each character as a binary string or codeword.

Encoding

Example

Suppose that we have a $100,000$ character data file that we wish to store. The file contains only 6 characters, appearing with the following frequencies:

	a	b	c	d	e	f
Frequency in '000s	45	13	12	16	9	5

- A binary code encodes each character as a binary string or codeword.
  - a code is a set of codewords

Encoding

Example

Suppose that we have a $100,000$ character data file that we wish to store. The file contains only 6 characters, appearing with the following frequencies:

	a	b	c	d	e	f
Frequency in '000s	45	13	12	16	9	5

- A binary code encodes each character as a binary string or codeword.
  - a code is a set of codewords
  - e.g., {000, 001, 010, 011, 100, 101} and {0, 101, 100, 111, 1101, 1100}

Encoding

Given a code (corresponding to some alphabet Σ) and a message it is easy to encode the message. Just replace the characters by the codewords.

Encoding

Given a code (corresponding to some alphabet Σ) and a message it is easy to encode the message. Just replace the characters by the codewords.

Example

$\Sigma = \{a, b, c, d\}$
If the code is

$$C_1 = \{a = 00,\ b = 01,\ c = 10,\ d = 11\}$$

Encoding

Given a code (corresponding to some alphabet Σ) and a message it is easy to encode the message. Just replace the characters by the codewords.

Example

$\Sigma = \{a, b, c, d\}$

If the code is

$$C_1 = \{a = 00,\ b = 01,\ c = 10,\ d = 11\}$$

then bad is encoded into

Encoding

Given a code (corresponding to some alphabet Σ) and a message it is easy to encode the message. Just replace the characters by the codewords.

Example

$\Sigma = \{a, b, c, d\}$

If the code is

$$C_1 = \{a = 00,\ b = 01,\ c = 10,\ d = 11\}$$

then bad is encoded into 01

Encoding

Given a code (corresponding to some alphabet Σ) and a message it is easy to encode the message. Just replace the characters by the codewords.

Example

$\Sigma = \{a, b, c, d\}$

If the code is

$$C_1 = \{a = 00,\ b = 01,\ c = 10,\ d = 11\}$$

then bad is encoded into 01 00

Encoding

Given a code (corresponding to some alphabet Σ) and a message it is easy to encode the message. Just replace the characters by the codewords.

Example

$\Sigma = \{a, b, c, d\}$
If the code is

$$C_1 = \{a = 00,\ b = 01,\ c = 10,\ d = 11\}$$

then bad is encoded into 01 00 11

Encoding

Given a code (corresponding to some alphabet Σ) and a message it is easy to encode the message. Just replace the characters by the codewords.

Example

$\Sigma = \{a, b, c, d\}$
If the code is

$$C_1 = \{a = 00,\ b = 01,\ c = 10,\ d = 11\}$$

then bad is encoded into 01 00 11
If the code is

$$C_2 = \{a = 0,\ b = 110,\ c = 10,\ d = 111\}$$

Encoding

Given a code (corresponding to some alphabet Σ) and a message it is easy to encode the message. Just replace the characters by the codewords.

Example

$\Sigma = \{a, b, c, d\}$
If the code is

$$C_1 = \{a = 00,\ b = 01,\ c = 10,\ d = 11\}$$

then bad is encoded into 01 00 11
If the code is

$$C_2 = \{a = 0,\ b = 110,\ c = 10,\ d = 111\}$$

then bad is encoded into

Encoding

Given a code (corresponding to some alphabet Σ) and a message it is easy to encode the message. Just replace the characters by the codewords.

Example

$\Sigma = \{a, b, c, d\}$
If the code is

$$C_1 = \{a = 00,\ b = 01,\ c = 10,\ d = 11\}$$

then bad is encoded into 01 00 11
If the code is

$$C_2 = \{a = 0,\ b = 110,\ c = 10,\ d = 111\}$$

then bad is encoded into 110

Encoding

Given a code (corresponding to some alphabet Σ) and a message it is easy to encode the message. Just replace the characters by the codewords.

Example

$\Sigma = \{a, b, c, d\}$
If the code is

$$C_1 = \{a = 00,\ b = 01,\ c = 10,\ d = 11\}$$

then bad is encoded into 01 00 11
If the code is

$$C_2 = \{a = 0,\ b = 110,\ c = 10,\ d = 111\}$$

then bad is encoded into 110 0

Encoding

Given a code (corresponding to some alphabet Σ) and a message it is easy to encode the message. Just replace the characters by the codewords.

Example

$\Sigma = \{a, b, c, d\}$

If the code is

$$C_1 = \{a = 00,\ b = 01,\ c = 10,\ d = 11\}$$

then bad is encoded into 01 00 11

If the code is

$$C_2 = \{a = 0,\ b = 110,\ c = 10,\ d = 111\}$$

then bad is encoded into 110 0 111

Decoding

$$C_1 = \{a = 00, b = 01, c = 10, d = 11\}.$$
$$C_2 = \{a = 0, b = 110, c = 10, d = 111\}.$$
$$C_3 = \{a = 1, b = 110, c = 10, d = 111\}.$$

Decoding

$$C_1 = \{a = 00, b = 01, c = 10, d = 11\}.$$
$$C_2 = \{a = 0, b = 110, c = 10, d = 111\}.$$
$$C_3 = \{a = 1, b = 110, c = 10, d = 111\}.$$

Given an encoded message, decoding is the process of turning it back into the original message.

Decoding

$$C_1 = \{a = 00, b = 01, c = 10, d = 11\}.$$
$$C_2 = \{a = 0, b = 110, c = 10, d = 111\}.$$
$$C_3 = \{a = 1, b = 110, c = 10, d = 111\}.$$

Given an encoded message, decoding is the process of turning it back into the original message.

A message is uniquely decodable if it can only be decoded in one way.

Decoding

$$C_1 = \{a = 00, b = 01, c = 10, d = 11\}.$$
$$C_2 = \{a = 0, b = 110, c = 10, d = 111\}.$$
$$C_3 = \{a = 1, b = 110, c = 10, d = 111\}.$$

Given an encoded message, decoding is the process of turning it back into the original message.

A message is uniquely decodable if it can only be decoded in one way.

Example

Relative to C_1, 010011 is uniquely decodable to bad.

Decoding

$$C_1 = \{a = 00, b = 01, c = 10, d = 11\}.$$
$$C_2 = \{a = 0, b = 110, c = 10, d = 111\}.$$
$$C_3 = \{a = 1, b = 110, c = 10, d = 111\}.$$

Given an encoded message, decoding is the process of turning it back into the original message.

A message is uniquely decodable if it can only be decoded in one way.

Example

Relative to C_1, 010011 is uniquely decodable to bad.

Relative to C_2, 1100111 is uniquely decodable to bad.

Decoding

$$C_1 = \{a = 00, b = 01, c = 10, d = 11\}.$$
$$C_2 = \{a = 0, b = 110, c = 10, d = 111\}.$$
$$C_3 = \{a = 1, b = 110, c = 10, d = 111\}.$$

Given an encoded message, decoding is the process of turning it back into the original message.

A message is uniquely decodable if it can only be decoded in one way.

Example

Relative to C_1, 010011 is uniquely decodable to bad.
Relative to C_2, 1100111 is uniquely decodable to bad. But, relative to C_3, 1101111 is not uniquely decipherable since it could have encoded either bad or acad.

Decoding

$$C_1 = \{a = 00, b = 01, c = 10, d = 11\}.$$
$$C_2 = \{a = 0, b = 110, c = 10, d = 111\}.$$
$$C_3 = \{a = 1, b = 110, c = 10, d = 111\}.$$

Given an encoded message, decoding is the process of turning it back into the original message.

A message is uniquely decodable if it can only be decoded in one way.

Example

Relative to C_1, 010011 is uniquely decodable to bad.
Relative to C_2, 1100111 is uniquely decodable to bad. But, relative to C_3, 1101111 is not uniquely decipherable since it could have encoded either bad or acad.

In fact, one can show that every message encoded using C_1 or C_2 is uniquely decipherable.

Decoding

$$C_1 = \{a = 00, b = 01, c = 10, d = 11\}.$$
$$C_2 = \{a = 0, b = 110, c = 10, d = 111\}.$$
$$C_3 = \{a = 1, b = 110, c = 10, d = 111\}.$$

Given an encoded message, decoding is the process of turning it back into the original message.

A message is uniquely decodable if it can only be decoded in one way.

Example

Relative to C_1, 010011 is uniquely decodable to bad.
Relative to C_2, 1100111 is uniquely decodable to bad. But, relative to C_3, 1101111 is not uniquely decipherable since it could have encoded either bad or acad.

In fact, one can show that every message encoded using C_1 or C_2 is uniquely decipherable. The unique decipherability property is needed in order for a code to be useful.

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
fixed-len code	000	001	010	011	100	101

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
fixed-len code	000	001	010	011	100	101
variable-len code	0	101	100	111	1101	1100

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
fixed-len code	000	001	010	011	100	101
variable-len code	0	101	100	111	1101	1100

(note that, since there are 6 characters, a fixed-length code must have at least 3 bits per codeword).

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
fixed-len code	000	001	010	011	100	101
variable-len code	0	101	100	111	1101	1100

(note that, since there are 6 characters, a fixed-length code must have at least 3 bits per codeword).

- The fixed-length code requires $300,000$ bits to store the file.

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
fixed-len code	000	001	010	011	100	101
variable-len code	0	101	100	111	1101	1100

(note that, since there are 6 characters, a fixed-length code must have at least 3 bits per codeword).

- The fixed-length code requires $300,000$ bits to store the file.
- The variable-length code uses only

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
fixed-len code	000	001	010	011	100	101
variable-len code	0	101	100	111	1101	1100

(note that, since there are 6 characters, a fixed-length code must have at least 3 bits per codeword).

- The fixed-length code requires $300,000$ bits to store the file.
- The variable-length code uses only
  $(45 \cdot 1 +$

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
fixed-len code	000	001	010	011	100	101
variable-len code	0	101	100	111	1101	1100

(note that, since there are 6 characters, a fixed-length code must have at least 3 bits per codeword).

- The fixed-length code requires $300,000$ bits to store the file.
- The variable-length code uses only
  $(45 \cdot 1 + 13 \cdot 3 +$

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
fixed-len code	000	001	010	011	100	101
variable-len code	0	101	100	111	1101	1100

(note that, since there are 6 characters, a fixed-length code must have at least 3 bits per codeword).

- The fixed-length code requires $300,000$ bits to store the file.
- The variable-length code uses only $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 =$

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
fixed-len code	000	001	010	011	100	101
variable-len code	0	101	100	111	1101	1100

(note that, since there are 6 characters, a fixed-length code must have at least 3 bits per codeword).

- The fixed-length code requires $300,000$ bits to store the file.
- The variable-length code uses only $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224,000\text{bits}$,

Fixed-Length vs Variable-Length

- In a fixed-length code each codeword has the same length.
- In a variable-length code codewords may have different lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
fixed-len code	000	001	010	011	100	101
variable-len code	0	101	100	111	1101	1100

(note that, since there are 6 characters, a fixed-length code must have at least 3 bits per codeword).

- The fixed-length code requires $300,000$ bits to store the file.
- The variable-length code uses only $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224,000$bits, saving a lot of space!

Prefix-Codes

Fixed-length codes are always uniquely decipherable

Prefix-Codes

Fixed-length codes are always uniquely decipherable

Question

Why?

Prefix-Codes

Fixed-length codes are always uniquely decipherable

Question

Why?

These do not always give the best compression so we prefer to use variable length codes.

Prefix-Codes

Fixed-length codes are always uniquely decipherable

Question

Why?

These do not always give the best compression so we prefer to use variable length codes.

Definition

A code is called a prefix (free) code if no codeword is a prefix of another one.

Prefix-Codes

Fixed-length codes are always uniquely decipherable

Question

Why?

These do not always give the best compression so we prefer to use variable length codes.

Definition

A code is called a prefix (free) code if no codeword is a prefix of another one.

Example

$\{a = 0,\ b = 110,\ c = 10,\ d = 111\}$ is a prefix code.

Prefix-Codes…

Important Fact: Every message encoded by a prefix free code is uniquely decipherable.

Prefix-Codes…

Important Fact: Every message encoded by a prefix free code is uniquely decipherable.

- Since no codeword is a prefix of any other, we can always find the first codeword in a message, peel it off, and continue decoding.

Prefix-Codes...

Important Fact: Every message encoded by a prefix free code is uniquely decipherable.

- Since no codeword is a prefix of any other, we can always find the first codeword in a message, peel it off, and continue decoding.

Example

code: $\{a = 0,\ b = 110,\ c = 10,\ d = 111\}$.

$$01101100 =$$

Prefix-Codes...

Important Fact: Every message encoded by a prefix free code is uniquely decipherable.

- Since no codeword is a prefix of any other, we can always find the first codeword in a message, peel it off, and continue decoding.

Example

code: $\{a = 0,\ b = 110,\ c = 10,\ d = 111\}$.

$$01101100 = 0$$

Prefix-Codes…

Important Fact: Every message encoded by a prefix free code is uniquely decipherable.

- Since no codeword is a prefix of any other, we can always find the first codeword in a message, peel it off, and continue decoding.

Example

code: $\{a = 0,\ b = 110,\ c = 10,\ d = 111\}$.

$$01101100 = 0110$$

Prefix-Codes...

Important Fact: Every message encoded by a prefix free code is uniquely decipherable.

- Since no codeword is a prefix of any other, we can always find the first codeword in a message, peel it off, and continue decoding.

Example

code: $\{a = 0,\ b = 110,\ c = 10,\ d = 111\}$.

$$01101100 = 0110110$$

Prefix-Codes…

Important Fact: Every message encoded by a prefix free code is uniquely decipherable.

- Since no codeword is a prefix of any other, we can always find the first codeword in a message, peel it off, and continue decoding.

Example

code: $\{a = 0,\ b = 110,\ c = 10,\ d = 111\}$.

$$01101100 = 01101100$$

Prefix-Codes…

Important Fact: Every message encoded by a prefix free code is uniquely decipherable.

- Since no codeword is a prefix of any other, we can always find the first codeword in a message, peel it off, and continue decoding.

Example

code: $\{a = 0,\ b = 110,\ c = 10,\ d = 111\}.$

$$01101100 = 01101100 = abba$$

Prefix-Codes...

Important Fact: Every message encoded by a prefix free code is uniquely decipherable.

- Since no codeword is a prefix of any other, we can always find the first codeword in a message, peel it off, and continue decoding.

Example

code: $\{a = 0,\ b = 110,\ c = 10,\ d = 111\}$.

$$01101100 = 01101100 = abba$$

We are therefore interested in finding good (best compression) prefix-free codes.

The Optimal Source Coding Problem

Problem

Given an alphabet $A = \{a_1, \ldots, a_n\}$ with frequency distribution $f(a_i)$, find a binary prefix code C for A that minimizes the number of bits

$$B(C) = \sum_{i=1}^{n} f(a_i) L(c(a_i))$$

needed to encode a message of $\sum_{i=1}^{n} f(a_i)$ characters, where

- $c(a_i)$ is the codeword for encoding a_i, and
- $L(c(a_i))$ is the length of the codeword $c(a_i)$.

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively.

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code				

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code				

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store 100 of these characters,

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store 100 of these characters,
(1) the fixed-length code requires $100 \times 2 = 200$ bits,

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store 100 of these characters,
(1) the fixed-length code requires $100 \times 2 = 200$ bits,
(2) the prefix code uses only

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store 100 of these characters,
(1) the fixed-length code requires $100 \times 2 = 200$ bits,
(2) the prefix code uses only

$$60 \times 1 +$$

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store 100 of these characters,
(1) the fixed-length code requires $100 \times 2 = 200$ bits,
(2) the prefix code uses only

$$60 \times 1 + 5 \times 3 +$$

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store 100 of these characters,
(1) the fixed-length code requires $100 \times 2 = 200$ bits,
(2) the prefix code uses only

$$60 \times 1 + 5 \times 3 + 30 \times 2 + 5 \times 3$$

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store 100 of these characters,
(1) the fixed-length code requires $100 \times 2 = 200$ bits,
(2) the prefix code uses only

$$60 \times 1 + 5 \times 3 + 30 \times 2 + 5 \times 3 = 150$$

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store 100 of these characters,
(1) the fixed-length code requires $100 \times 2 = 200$ bits,
(2) the prefix code uses only

$$60 \times 1 + 5 \times 3 + 30 \times 2 + 5 \times 3 = 150$$

a 25% saving.

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store 100 of these characters,
(1) the fixed-length code requires $100 \times 2 = 200$ bits,
(2) the prefix code uses only

$$60 \times 1 + 5 \times 3 + 30 \times 2 + 5 \times 3 = 150$$

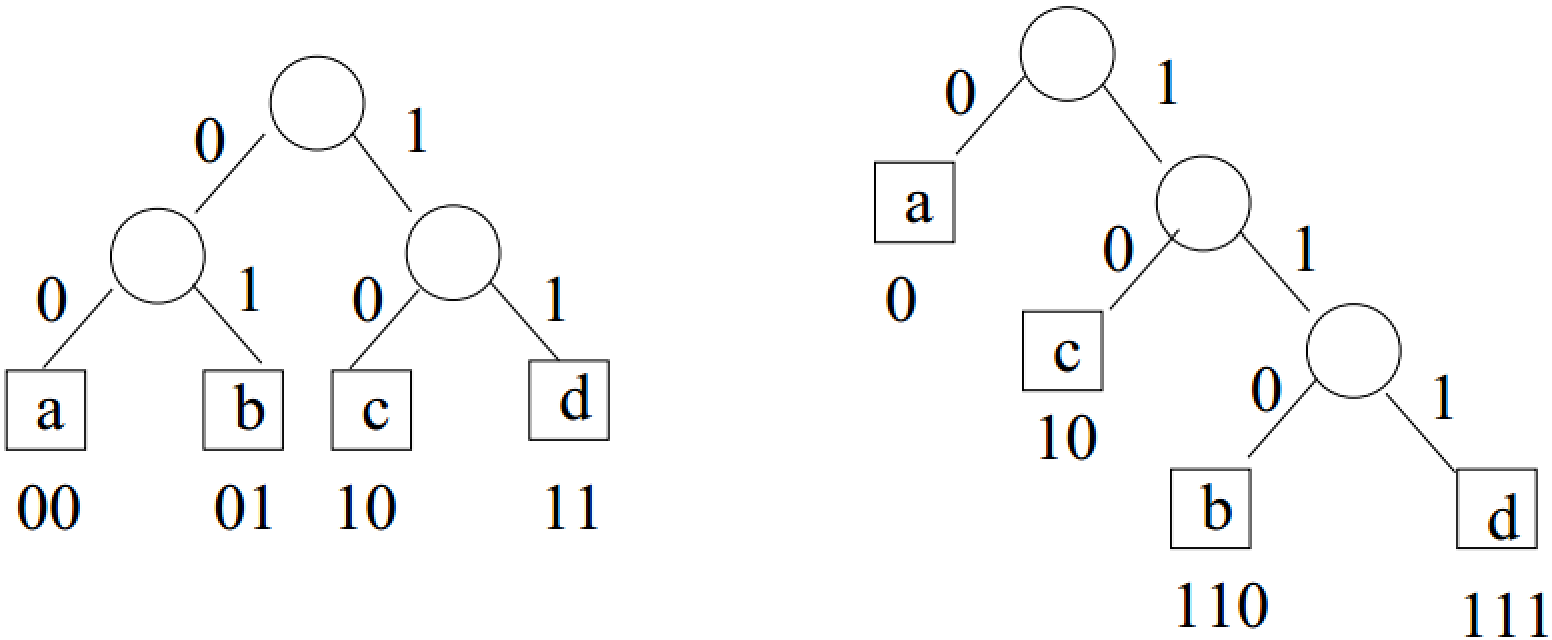
a 25% saving.
Remark: We will see later that this is the optimum (lowest cost) prefix code.

Correspondence between Binary Trees and Prefix Codes

1-1 correspondence between leaves and characters.

Correspondence between Binary Trees and Prefix Codes

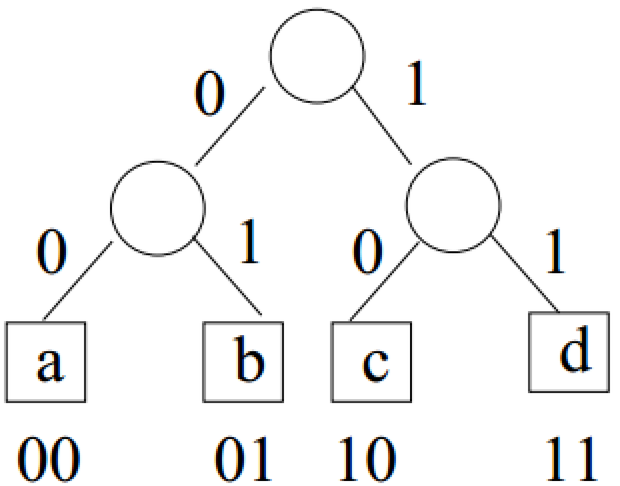
1-1 correspondence between leaves and characters.

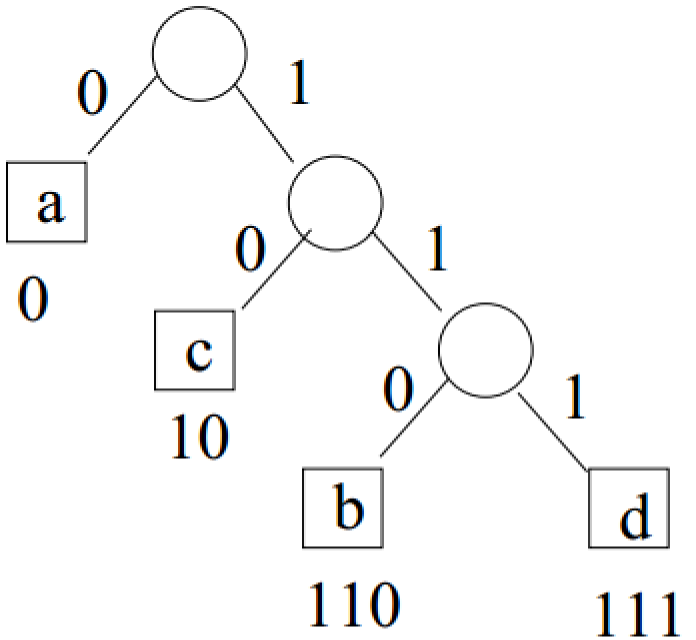


- Left edge is labeled 0; right edge is labeled 1

Correspondence between Binary Trees and Prefix Codes

1-1 correspondence between leaves and characters.





- Left edge is labeled 0; right edge is labeled 1
- The binary string on a path from the root to a leaf is the codeword associated with the character at the leaf.

Huffman Encoding Problem

- $d(a_i)$, the depth of leaf a_i, is equal to the length $L(c(a_i))$ of the codeword associated with that leaf.

Huffman Encoding Problem

- $d(a_i)$, the depth of leaf a_i, is equal to the length $L(c(a_i))$ of the codeword associated with that leaf.
- Weighted external pathlength B(T) of tree T

$$\sum_{i=1}^{n} f(a_i)d(a_i) = \sum_{i=1}^{n} f(a_i)L(c(a_i))$$

Huffman Encoding Problem

- $d(a_i)$, the depth of leaf a_i, is equal to the length $L(c(a_i))$ of the codeword associated with that leaf.
- Weighted external pathlength $B(T)$ of tree T

$$\sum_{i=1}^{n} f(a_i)d(a_i) = \sum_{i=1}^{n} f(a_i)L(c(a_i))$$

Definition (Minimum-Weight External Pathlength Problem)

Given weights $f(a_1), \ldots, f(a_n)$,

Huffman Encoding Problem

- $d(a_i)$, the depth of leaf a_i, is equal to the length $L(c(a_i))$ of the codeword associated with that leaf.
- Weighted external pathlength B(T) of tree T

$$\sum_{i=1}^{n} f(a_i)d(a_i) = \sum_{i=1}^{n} f(a_i)L(c(a_i))$$

Definition (Minimum-Weight External Pathlength Problem)

Given weights $f(a_1), \ldots, f(a_n)$, find a tree T with n leaves labeled $a_1, \ldots, a_n$ that has minimum weighted external path length.

Huffman Encoding Problem

- $d(a_i)$, the depth of leaf a_i, is equal to the length $L(c(a_i))$ of the codeword associated with that leaf.
- Weighted external pathlength B(T) of tree T

$$\sum_{i=1}^{n} f(a_i)d(a_i) = \sum_{i=1}^{n} f(a_i)L(c(a_i))$$

Definition (Minimum-Weight External Pathlength Problem)

Given weights $f(a_1), \ldots, f(a_n)$, find a tree T with n leaves labeled $a_1, \ldots, a_n$ that has minimum weighted external path length.

The Huffman encoding problem is equivalent to the minimum weighted external path length problem.

Outline

- Introduction to Part III
- Fractional Knapsack Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
- Huffman Coding Problem
  - Problem Definition
  - A Greedy Algorithm
- Activity Selection Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
  - Extended: Weighted Activity Selection

Huffman Coding

- (Greedy idea)
  - Pick two characters x, y from alphabet A with the smallest frequencies.

Huffman Coding

- (Greedy idea)
  - Pick two characters x, y from alphabet A with the smallest frequencies.
  - Create a subtree that has these two characters as leaves.

Huffman Coding

- (Greedy idea)
  - Pick two characters x, y from alphabet A with the smallest frequencies.
  - Create a subtree that has these two characters as leaves.
  - Label the root of this subtree as z.

Huffman Coding

- (Greedy idea)
  - Pick two characters x, y from alphabet A with the smallest frequencies.
  - Create a subtree that has these two characters as leaves.
  - Label the root of this subtree as z.
- Update
  - Set frequency f(z) = f(x) + f(y).

Huffman Coding

- (Greedy idea)
  - Pick two characters x, y from alphabet A with the smallest frequencies.
  - Create a subtree that has these two characters as leaves.
  - Label the root of this subtree as z.
- Update
  - Set frequency f(z) = f(x) + f(y).
  - Remove x; y and add z creating a new alphabet A'.

Huffman Coding

- (Greedy idea)
  - Pick two characters x, y from alphabet A with the smallest frequencies.
  - Create a subtree that has these two characters as leaves.
  - Label the root of this subtree as z.
- Update
  - Set frequency f(z) = f(x) + f(y).
  - Remove x; y and add z creating a new alphabet A'.
    - $A' = A \cup \{z\} - \{x,y\}$.

Huffman Coding

- (Greedy idea)
  - Pick two characters x, y from alphabet A with the smallest frequencies.
  - Create a subtree that has these two characters as leaves.
  - Label the root of this subtree as z.
- Update
  - Set frequency $f(z) = f(x) + f(y)$.
  - Remove x; y and add z creating a new alphabet A'.
    - $A' = A \cup \{z\} - \{x,y\}$.
    - Note that $|A'| = |A| - 1$.

Huffman Coding

- (Greedy idea)
  - Pick two characters x, y from alphabet A with the smallest frequencies.
  - Create a subtree that has these two characters as leaves.
  - Label the root of this subtree as z.
- Update
  - Set frequency f(z) = f(x) + f(y).
  - Remove x; y and add z creating a new alphabet A'.
    - $A' = A \cup \{z\} - \{x,y\}$.
    - Note that $|A'| = |A| - 1$.
- Repeat this procedure, called merge, with the new alphabet A' until an alphabet with only one symbol is left. The resulting tree is the Huffman code.

Huffman Coding

- (Greedy idea)
  - Pick two characters x, y from alphabet A with the smallest frequencies.
  - Create a subtree that has these two characters as leaves.
  - Label the root of this subtree as z.
- Update
  - Set frequency f(z) = f(x) + f(y).
  - Remove x; y and add z creating a new alphabet A'.
    - $A' = A \cup \{z\} - \{x,y\}$.
    - Note that $|A'| = |A| - 1$.
- Repeat this procedure, called merge, with the new alphabet A' until an alphabet with only one symbol is left. The resulting tree is the Huffman code.
- The optimum (minimum-cost) prefix code for the given frequency distribution.

Huffman Coding Algorithm

Given an alphabet A with frequency distribution $\{f(a_i):a_i \in A\}$. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with frequencies as keys.

Huffman Coding Algorithm

Given an alphabet A with frequency distribution $\{f(a_i):a_i \in A\}$. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with frequencies as keys.

Huffman(A)

Input: An alphabet $\boldsymbol{A}$ with frequency distribution.
Output: Huffman tree.
$n \leftarrow |A|;$

Huffman Coding Algorithm

Given an alphabet A with frequency distribution $\{f(a_i):a_i \in A\}$. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with frequencies as keys.

Huffman(A)

Input: An alphabet $\boldsymbol{A}$ with frequency distribution.
Output: Huffman tree.
$n \leftarrow |A|$;
$Q \leftarrow$ a new Priority Queue of A;

Huffman Coding Algorithm

Given an alphabet A with frequency distribution $\{f(a_i):a_i \in A\}$. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with frequencies as keys.

Huffman(A)

Input: An alphabet $\boldsymbol{A}$ with frequency distribution.
Output: Huffman tree.
$n \leftarrow |A|$;
$Q \leftarrow$ a new Priority Queue of A;
for $i \leftarrow 1$ *to* $n-1$ **do**
  // Why $n-1$?
  $z \leftarrow$ a new node;

Huffman Coding Algorithm

Given an alphabet A with frequency distribution $\{f(a_i):a_i \in A\}$. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with frequencies as keys.

Huffman(A)

Input: An alphabet $\boldsymbol{A}$ with frequency distribution.
Output: Huffman tree.
$n \leftarrow |A|$;
$Q \leftarrow$ a new Priority Queue of A;
for $i \leftarrow 1$ *to* $n-1$ **do**
  // Why $n-1$?
  $z \leftarrow$ a new node;
  $z.left \leftarrow$

Huffman Coding Algorithm

Given an alphabet A with frequency distribution $\{f(a_i):a_i \in A\}$. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with frequencies as keys.

Huffman(A)

Input: An alphabet $\boldsymbol{A}$ with frequency distribution.
Output: Huffman tree.

$n \leftarrow |A|$;
$Q \leftarrow$ a new Priority Queue of A;
for $i \leftarrow 1$ *to* $n-1$ **do**
    // Why $n-1$?
    $z \leftarrow$ a new node;
    $z.left \leftarrow$Extract-Min(Q);
    $z.right \leftarrow$

Huffman Coding Algorithm

Given an alphabet A with frequency distribution $\{f(a_i):a_i \in A\}$. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with frequencies as keys.

Huffman(A)

Input: An alphabet $\boldsymbol{A}$ with frequency distribution.
Output: Huffman tree.
$n \leftarrow |A|$;
$Q \leftarrow$ a new Priority Queue of A;
for $i \leftarrow 1$ *to* $n-1$ **do**
    // Why $n-1$?
    $z \leftarrow$ a new node;
    $z.left \leftarrow$Extract-Min(Q);
    $z.right \leftarrow$Extract-Min(Q);
    $z.freq \leftarrow$

Huffman Coding Algorithm

Given an alphabet A with frequency distribution $\{f(a_i):a_i \in A\}$. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with frequencies as keys.

Huffman(A)

Input: An alphabet $\boldsymbol{A}$ with frequency distribution.
Output: Huffman tree.
$n \leftarrow |A|$;
$Q \leftarrow$ a new Priority Queue of A;
for $i \leftarrow 1$ *to* $n-1$ **do**
  // Why $n-1$?
  $z \leftarrow$ a new node;
  $z.left \leftarrow$Extract-Min(Q);
  $z.right \leftarrow$Extract-Min(Q);
  $z.freq \leftarrow z.left.freq + z.right.freq$;

Huffman Coding Algorithm

Given an alphabet A with frequency distribution $\{f(a_i):a_i \in A\}$. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with frequencies as keys.

Huffman(A)

Input: An alphabet $\boldsymbol{A}$ with frequency distribution.
Output: Huffman tree.
$n \leftarrow |A|$;
$Q \leftarrow$ a new Priority Queue of A;
for $i \leftarrow 1$ *to* $n-1$ **do**
    // Why $n-1$?
    $z \leftarrow$ a new node;
    $z.left \leftarrow$Extract-Min(Q);
    $z.right \leftarrow$Extract-Min(Q);
    $z.freq \leftarrow z.left.freq + z.right.freq$;
    Insert(Q, z);
end
return *Extract-Min(Q)*;

Huffman Coding Algorithm

Given an alphabet A with frequency distribution {f(a):a∈A}. The binary Huffman tree is constructed using a priority queue, Q, of nodes, with frequencies as keys.

Huffman(A)

Input: An alphabet $\boldsymbol{A}$ with frequency distribution.
Output: Huffman tree.
$n \leftarrow |A|$;
$Q \leftarrow$ a new Priority Queue of A;
for $i \leftarrow 1$ *to* $n-1$ **do**
    // Why $n-1$?
    $z \leftarrow$ a new node;
    $z.left \leftarrow$Extract-Min(Q);
    $z.right \leftarrow$Extract-Min(Q);
    $z.freq \leftarrow z.left.freq + z.right.freq$;
    Insert(Q, z);
end
return *Extract-Min(Q)*;

Running time is O(n log n), as each priority queue operation takes time O(log n).

Example of Optimal Solution Construction

	a	b	c	d	e
$\boldsymbol{freq}$	18	16	5	15	45

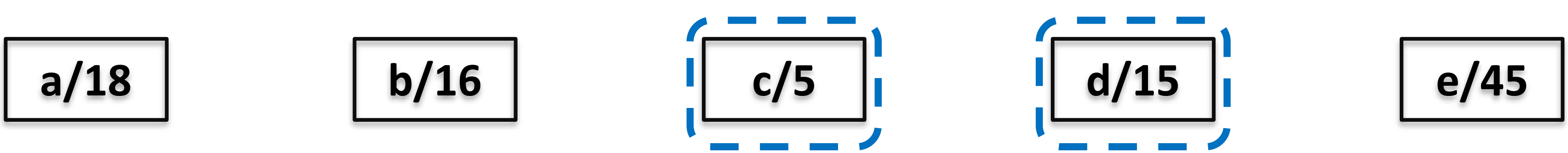
Example of Optimal Solution Construction

	a	**b**	**c**	**d**	**e**
$freq$	18	16	5	15	45

a/18 **b/16** **c/5** **d/15** **e/45**

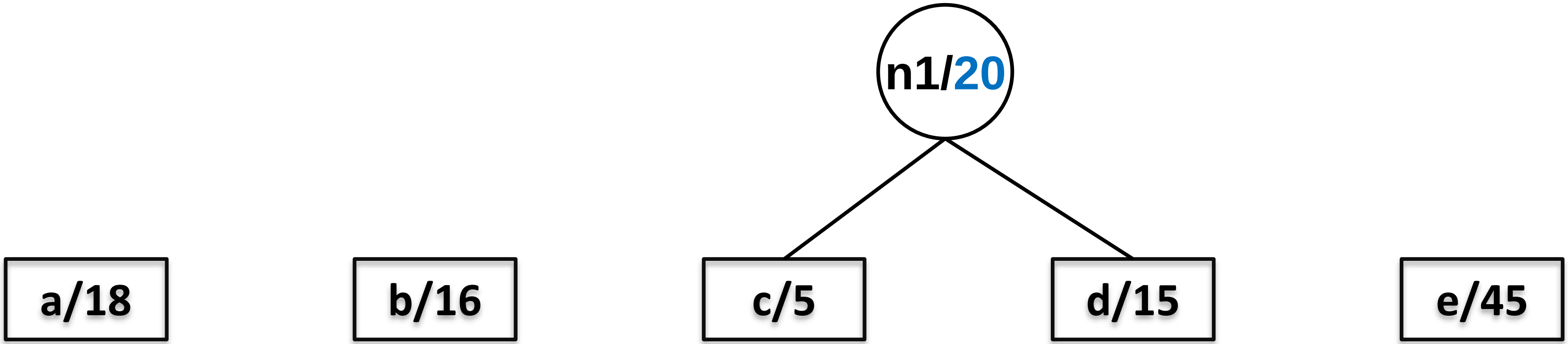
Example of Optimal Solution Construction

	a	b	c	d	e
$freq$	18	16	5	15	45



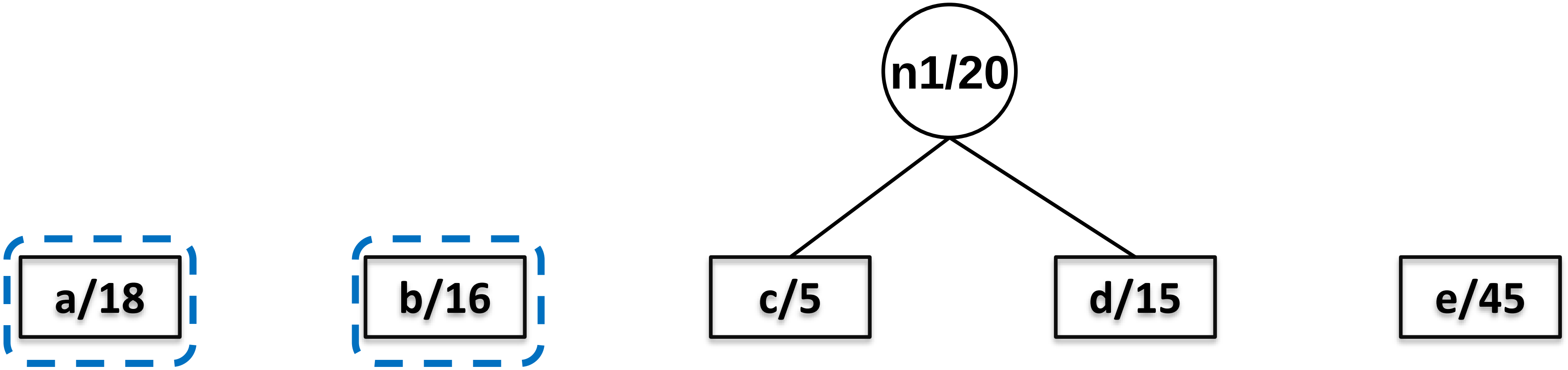
Example of Optimal Solution Construction

	a	b	c	d	e
$freq$	18	16	5	15	45



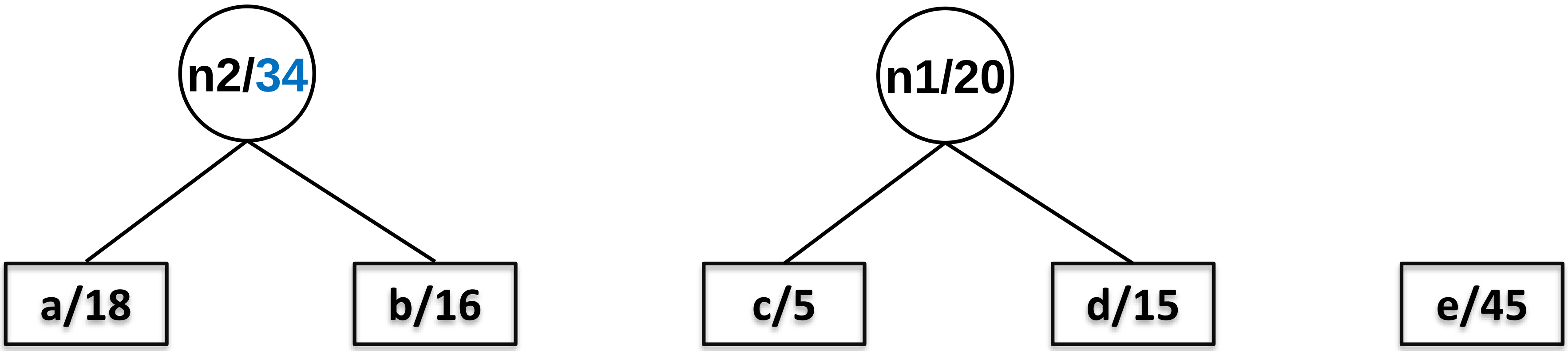
Example of Optimal Solution Construction

	a	b	c	d	e
$freq$	18	16	5	15	45



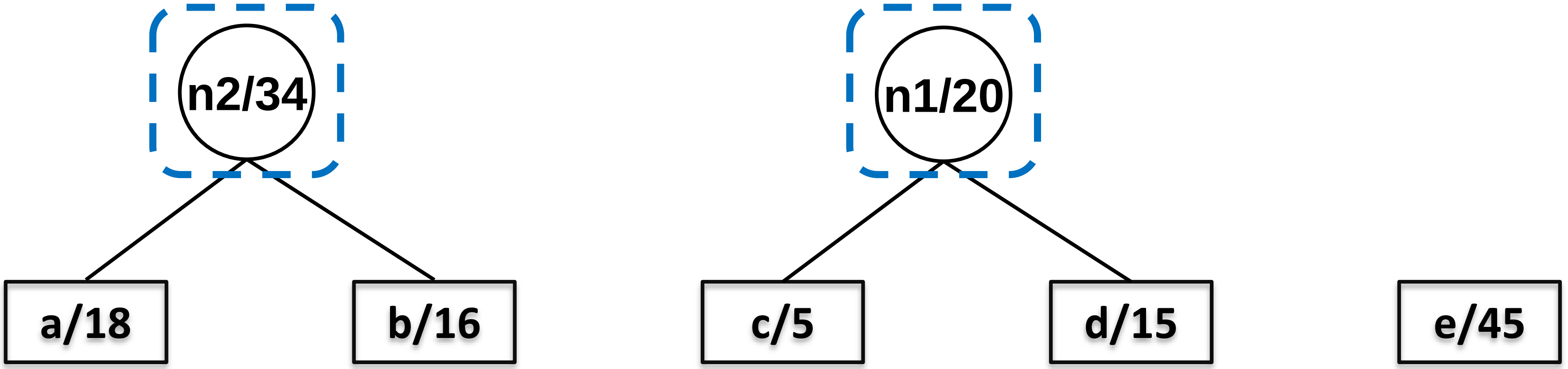
Example of Optimal Solution Construction

	a	b	c	d	e
$freq$	18	16	5	15	45



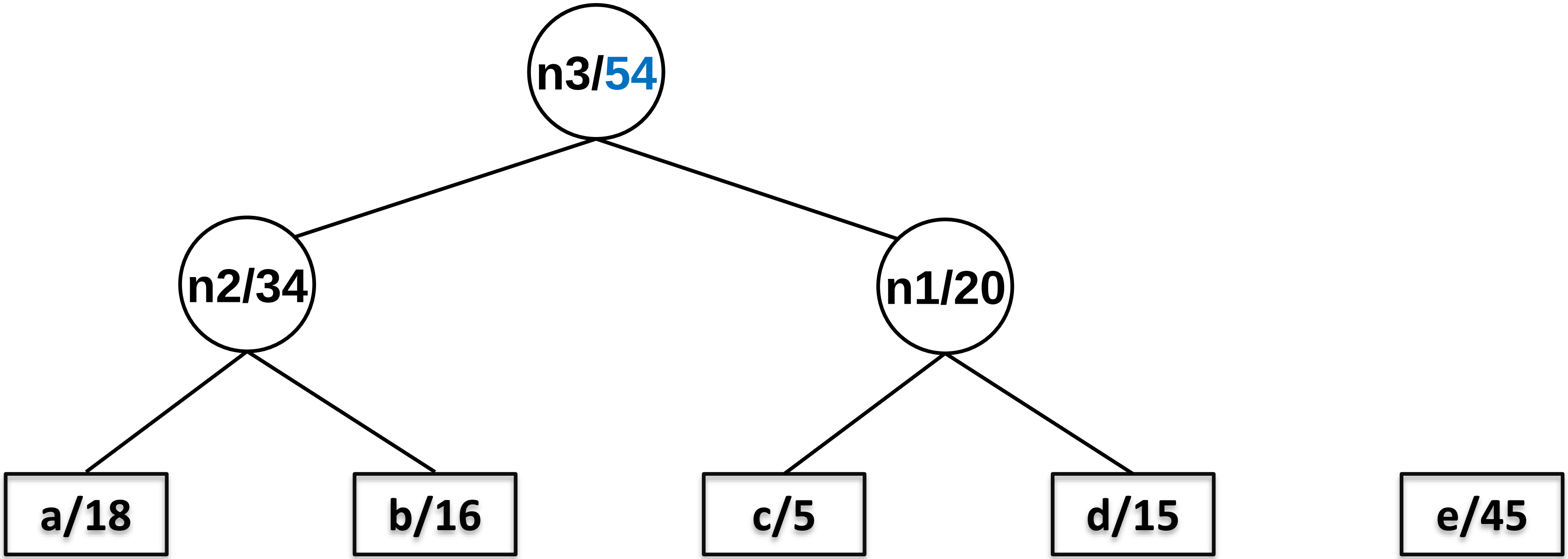
Example of Optimal Solution Construction

	a	b	c	d	e
$freq$	18	16	5	15	45



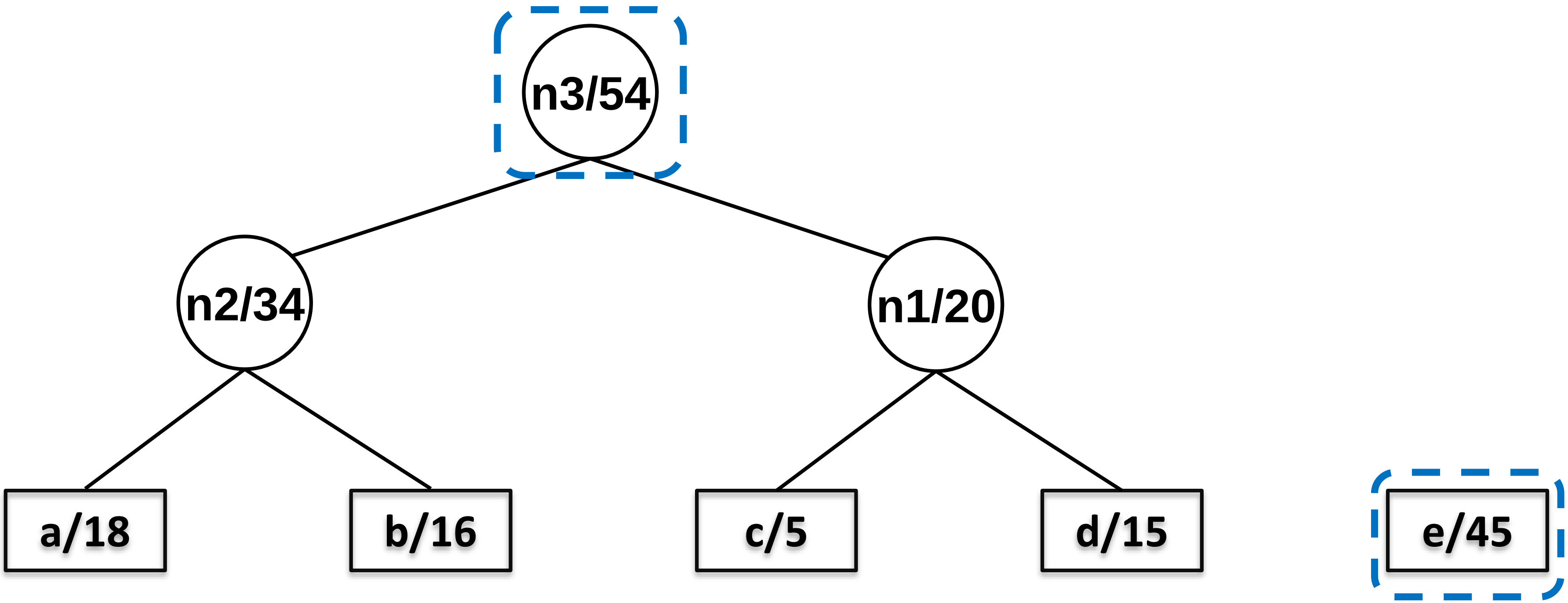
Example of Optimal Solution Construction

	a	b	c	d	e
$freq$	18	16	5	15	45



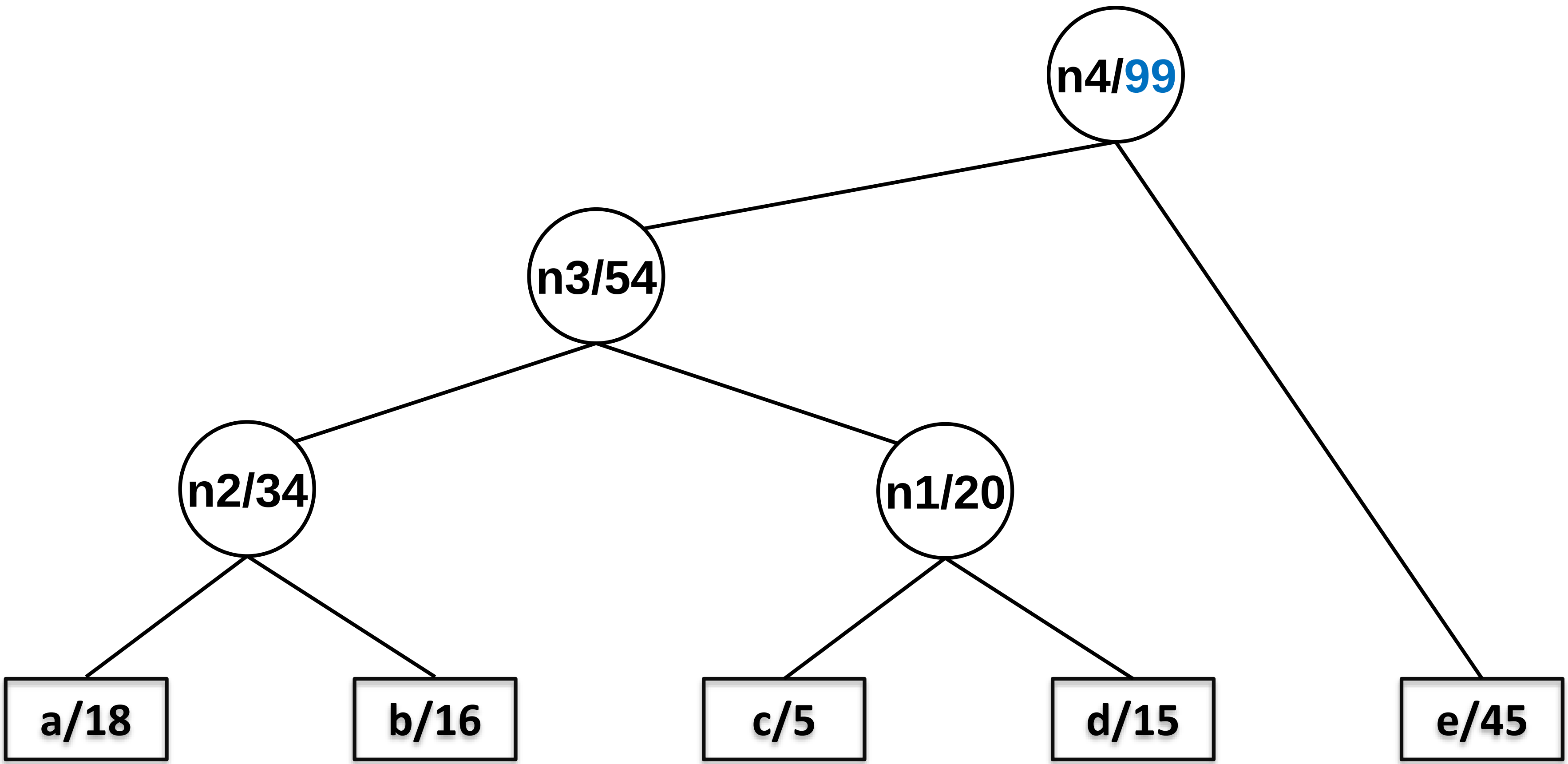
Example of Optimal Solution Construction

	a	b	c	d	e
$freq$	18	16	5	15	45



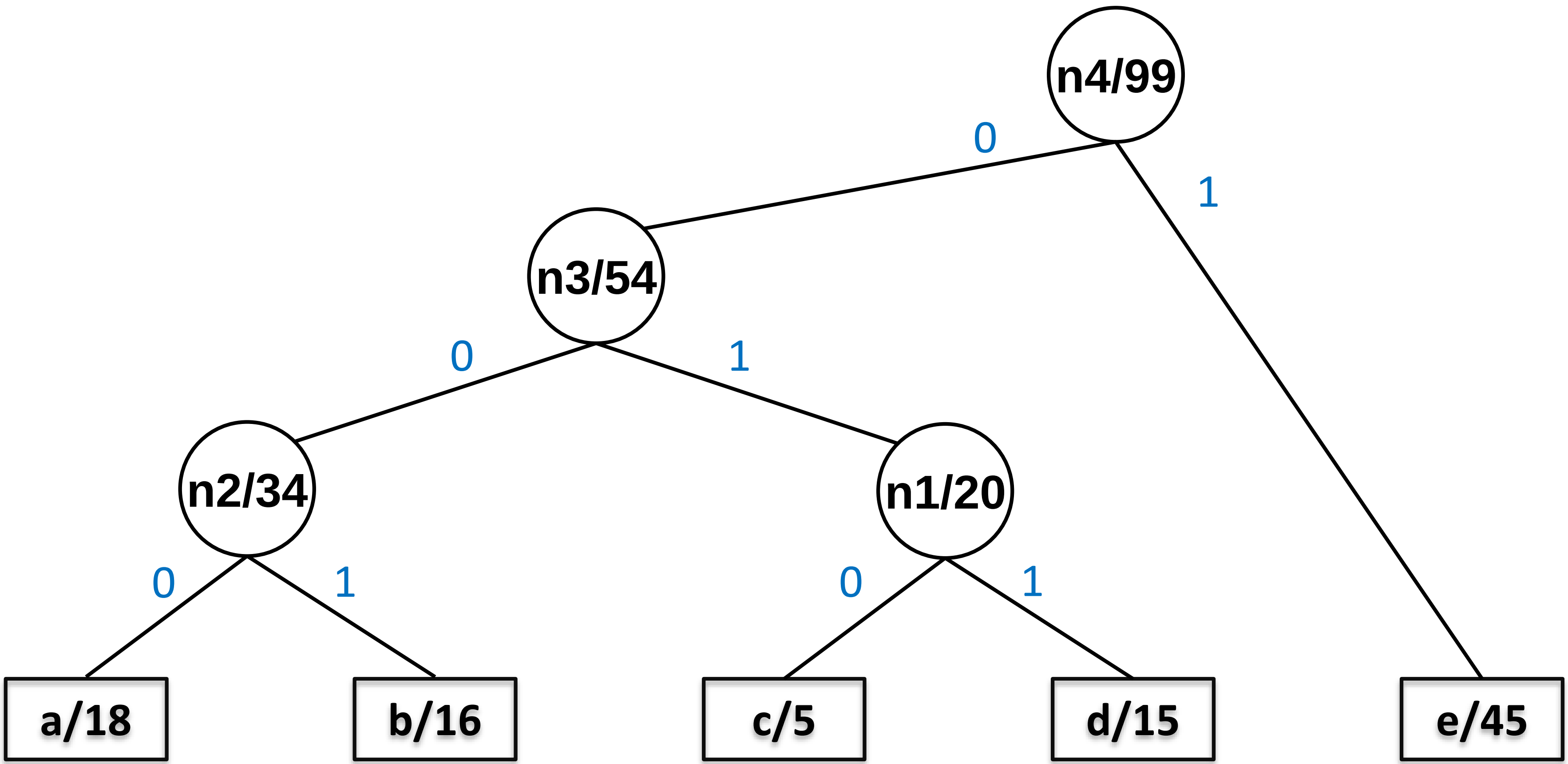
Example of Optimal Solution Construction

	a	b	c	d	e
$freq$	18	16	5	15	45



Example of Optimal Solution Construction

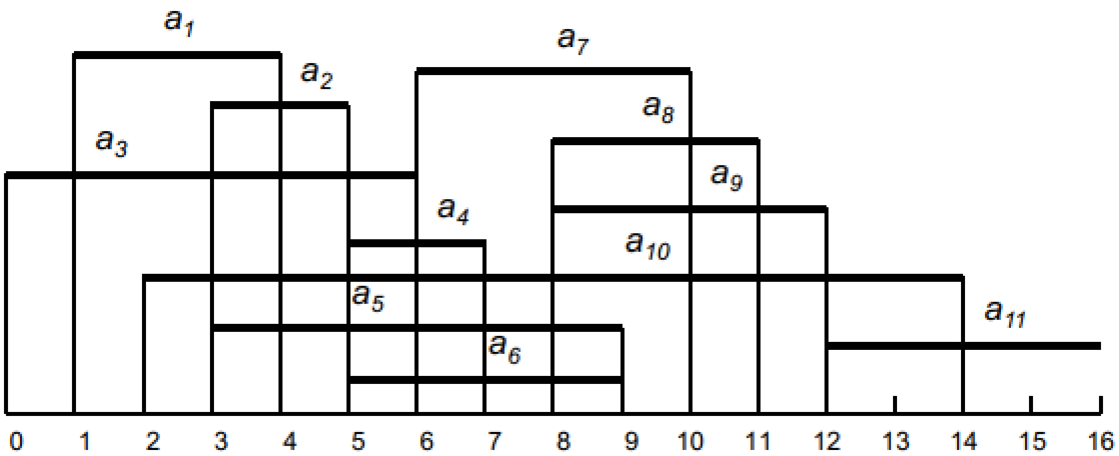
	a	b	c	d	e
$freq$	18	16	5	15	45
$code$	000	001	010	011	1



Outline

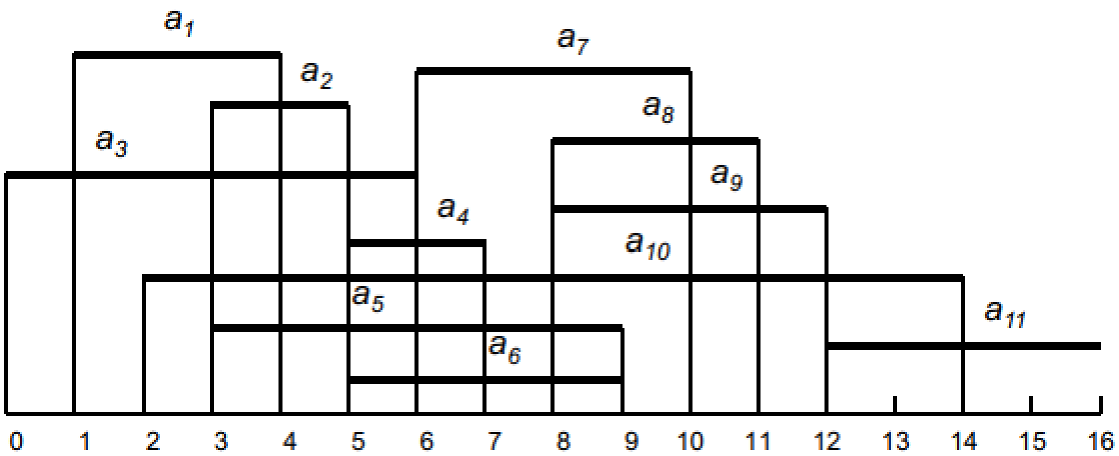
- Introduction to Part III
- Fractional Knapsack Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
- Huffman Coding Problem
  - Problem Definition
  - A Greedy Algorithm
- Activity Selection Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
  - Extended: Weighted Activity Selection

Activity Selection Problem



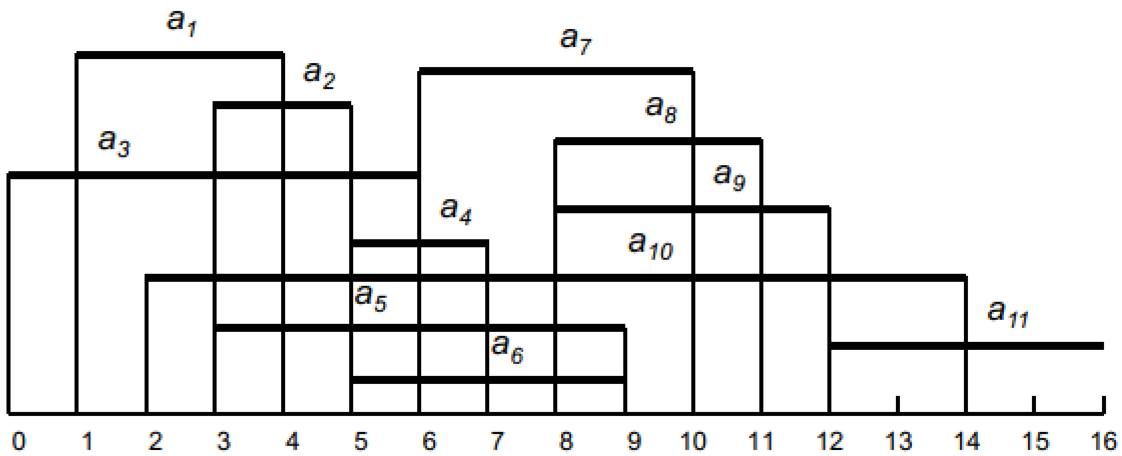
- Problem: get your money's worth out of a festival
  - Buy a wristband that lets you take part in any activity

Activity Selection Problem



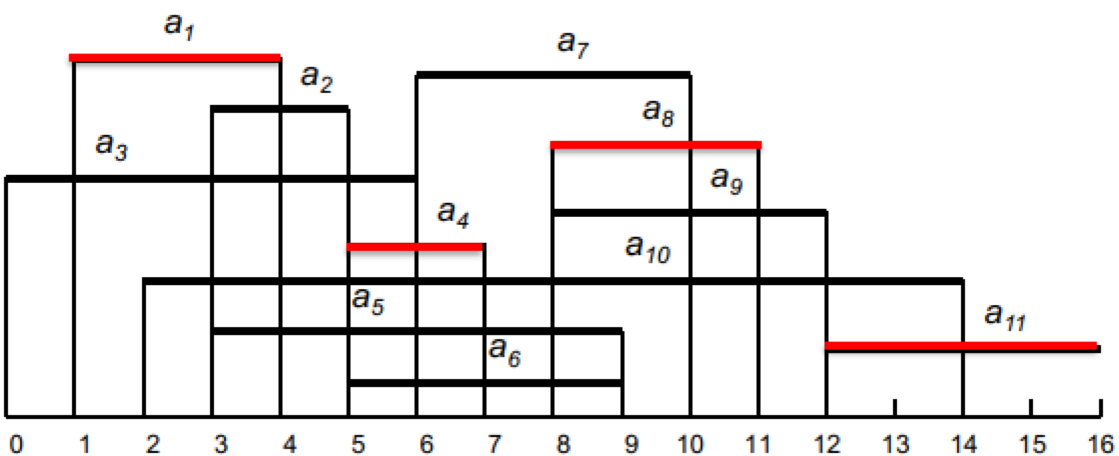
- Problem: get your money's worth out of a festival
  - Buy a wristband that lets you take part in any activity
  - Lots of activities, each starting and ending at different times

Activity Selection Problem



- Problem: get your money's worth out of a festival
  - Buy a wristband that lets you take part in any activity
  - Lots of activities, each starting and ending at different times
  - Your goal: take part in as many activities as possible

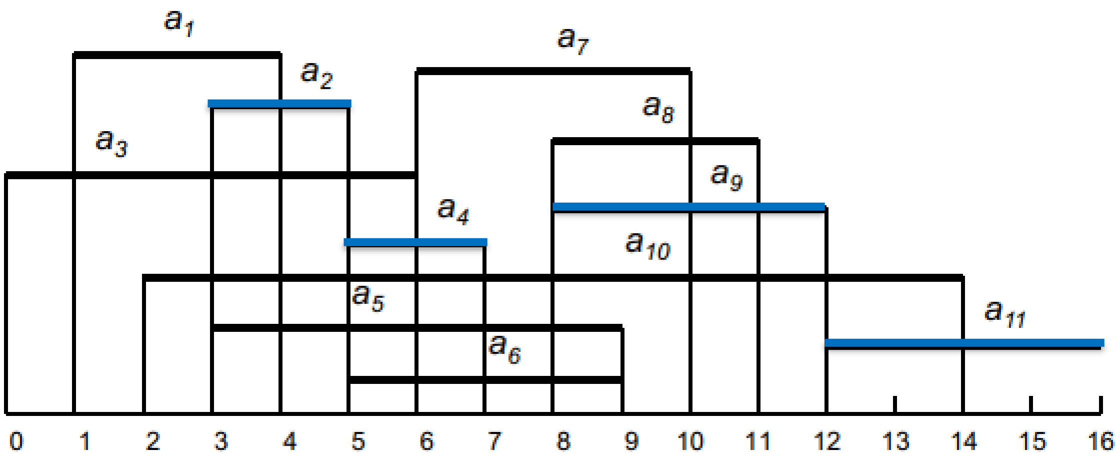
Activity Selection Problem



Some optimal solutions:
a_1, a_4, a_8, a_{11}

- Problem: get your money's worth out of a festival
  - Buy a wristband that lets you take part in any activity
  - Lots of activities, each starting and ending at different times
  - Your goal: take part in as many activities as possible

Activity Selection Problem



Some optimal solutions:
a_1, a_4, a_8, a_{11} or
a_2, a_4, a_9, a_{11}

- Problem: get your money's worth out of a festival
  - Buy a wristband that lets you take part in any activity
  - Lots of activities, each starting and ending at different times
  - Your goal: take part in as many activities as possible

Activity Selection Problem : Formal Definition

- Given a set A of activities, such that each activity a_i has a start time s_i and a finishing time f_i:

$\mathbf{a}_i$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
s_i	1	3	0	5	2	5	6	8	8	8	12
f_i	4	5	6	7	14	9	10	11	12	14	16

Activity Selection Problem : Formal Definition

- Given a set A of activities, such that each activity a_i has a start time s_i and a finishing time f_i:

$\mathbf{a_i}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
s_i	1	3	0	5	2	5	6	8	8	8	12
f_i	4	5	6	7	14	9	10	11	12	14	16

- Goal: we want to select the largest number of activities that do not overlap.

Activity Selection Problem : Formal Definition

- Given a set A of activities, such that each activity a_i has a start time s_i and a finishing time f_i:

$\mathbf{a}_i$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
s_i	1	3	0	5	2	5	6	8	8	8	12
f_i	4	5	6	7	14	9	10	11	12	14	16

- Goal: we want to select the largest number of activities that do not overlap.

- That is, find a subset S of A, such that:

$$s_i \geq f_j \; or \; f_i \leq s_j \qquad \forall \mathrm{a}_i, \mathrm{a}_j \in S, i \neq j$$

Activity Selection Problem : Formal Definition

- Given a set A of activities, such that each activity a_i has a start time s_i and a finishing time f_i:

$\mathbf{a}_i$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
s_i	1	3	0	5	2	5	6	8	8	8	12
f_i	4	5	6	7	14	9	10	11	12	14	16

- Goal: we want to select the largest number of activities that do not overlap.

- That is, find a subset S of A, such that:

$$s_i \geq f_j \; or \; f_i \leq s_j \qquad \forall \mathrm{a}_i, \mathrm{a}_j \in S, i \neq j$$

and the cardinality of S, $|S|$, is maximized.

Outline

- Introduction to Part III
- Fractional Knapsack Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
- Huffman Coding Problem
  - Problem Definition
  - A Greedy Algorithm
- Activity Selection Problem
  - Problem Definition
  - A Greedy Algorithm and correctness
  - Extended: Weighted Activity Selection

Concept of Greedy Algorithms

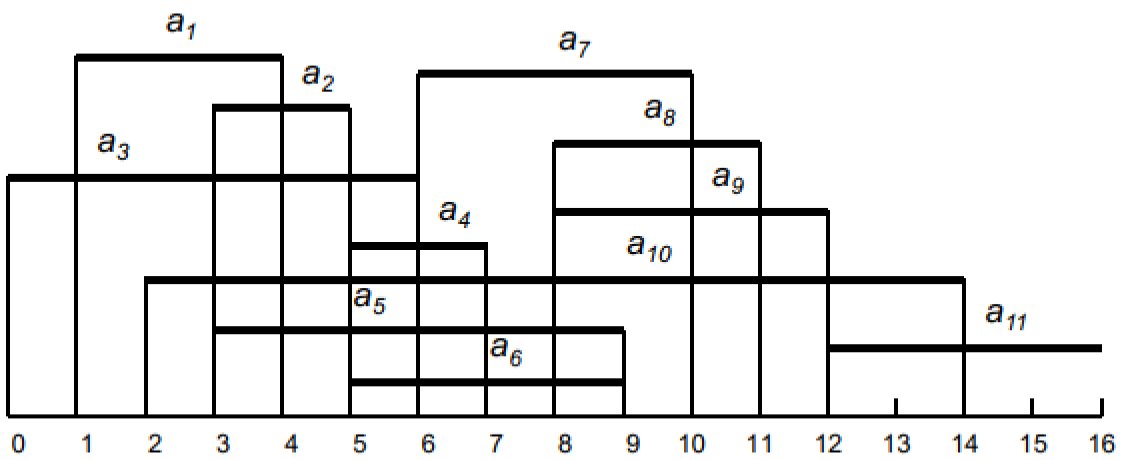
- Making a locally optimal choice can sometimes lead to a globally optimal solution.

Concept of Greedy Algorithms

- Making a locally optimal choice can sometimes lead to a globally optimal solution.
  - This choice depends on choices made so far, but not on choices to be made in the future.

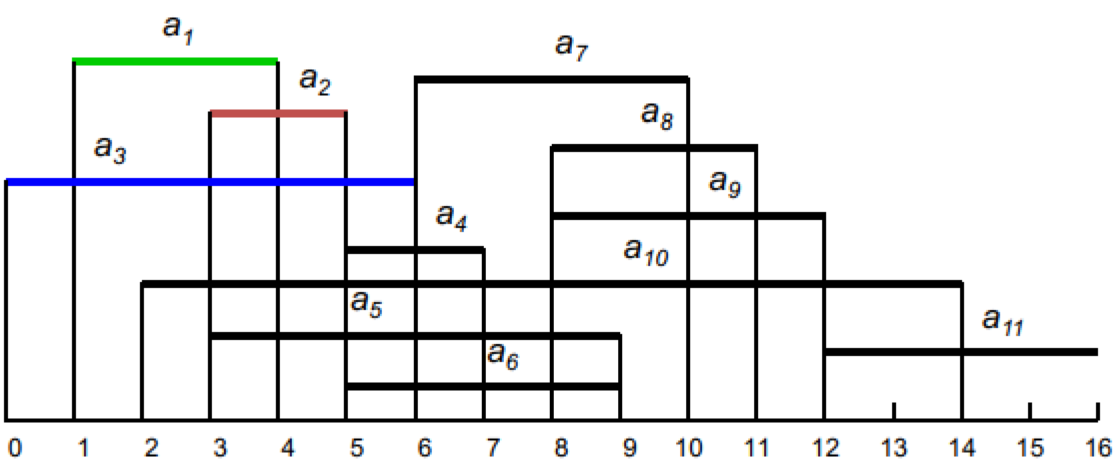
Concept of Greedy Algorithms

- Making a locally optimal choice can sometimes lead to a globally optimal solution.
  - This choice depends on choices made so far, but not on choices to be made in the future.
- In the activity selection problem, which activity would you select first?



Concept of Greedy Algorithms

- Making a locally optimal choice can sometimes lead to a globally optimal solution.
  - This choice depends on choices made so far, but not on choices to be made in the future.
- In the activity selection problem, which activity would you select first?
  - The shortest activity?
  - The one that starts at the earliest time?
  - The one that finishes at the earliest time?



Comparison of Different Greedy Strategies

- The shortest activity first?
  - Counterexample for shortest activity

This is optimal →

We will choose this →

- The activity with earliest start time first?
  - Counterexample for earliest start time

This is optimal →

We will choose this →

Greedy Activity Selection

- Answer to the previous question:
  - Select the activity that finishes at the earliest time
    - Intuition: it leaves the largest possible empty space for more activities

Greedy Activity Selection

- Answer to the previous question:
  - Select the activity that finishes at the earliest time
    - Intuition: it leaves the largest possible empty space for more activities
- We can propose a 3-step algorithm:

Greedy Activity Selection

- Answer to the previous question:
  - Select the activity that finishes at the earliest time
    - Intuition: it leaves the largest possible empty space for more activities
- We can propose a 3-step algorithm:
  - Select the activity that ends first (smallest finishing time)
    - Greedy Choice: Select the next best activity (Local Optimal)

Greedy Activity Selection

- Answer to the previous question:
  - Select the activity that finishes at the earliest time
    - Intuition: it leaves the largest possible empty space for more activities
- We can propose a 3-step algorithm:
  - Select the activity that ends first (smallest finishing time)
    - Greedy Choice: Select the next best activity (Local Optimal)
  - Once making the choice, delete all non-compatible activities
    - The activities that overlap our choice can not be selected

Greedy Activity Selection

- Answer to the previous question:
  - Select the activity that finishes at the earliest time
    - Intuition: it leaves the largest possible empty space for more activities
- We can propose a 3-step algorithm:
  - Select the activity that ends first (smallest finishing time)
    - Greedy Choice: Select the next best activity (Local Optimal)
  - Once making the choice, delete all non-compatible activities
    - The activities that overlap our choice can not be selected
  - Repeat the algorithm for the remaining activities
    - Either using iterations or recursion
    - We created one sub-problem to solve(Find the optimal schedule after the selected activity)

Pseudocode

Greedy Activity Selection(A)

Input: a set of activities $A = a_1, a_2 \ldots, a_n$
Output: the largest subset of A that do not overlap

Pseudocode

Greedy Activity Selection(A)

Input: a set of activities $A = a_1, a_2 \ldots, a_n$
Output: the largest subset of A that do not overlap
Sort activities in increasing order of finishing time;

Pseudocode

Greedy Activity Selection(A)

Input: a set of activities $A = a_1, a_2 \ldots, a_n$
Output: the largest subset of A that do not overlap
Sort activities in increasing order of finishing time;
$P = a_1$; `// insert the activity with earliest finishing time`

Pseudocode

Greedy Activity Selection(A)

Input: a set of activities $A = a_1, a_2 \ldots, a_n$
Output: the largest subset of A that do not overlap
Sort activities in increasing order of finishing time;
$P = a_1$; `// insert the activity with earliest finishing time`
$k = 1$; `// index to the last activity in A`

Pseudocode

Greedy Activity Selection(A)

Input: a set of activities $A = a_1, a_2 \ldots, a_n$
Output: the largest subset of A that do not overlap
Sort activities in increasing order of finishing time;
$P = a_1$; `// insert the activity with earliest finishing time`
$k = 1$; `// index to the last activity in A`
for $i \leftarrow 2$ ***to*** n **do**

end

Pseudocode

Greedy Activity Selection(A)

Input: a set of activities $A = a_1, a_2 \ldots, a_n$
Output: the largest subset of A that do not overlap
Sort activities in increasing order of finishing time;
$P = a_1$; `// insert the activity with earliest finishing time`
$k = 1$; `// index to the last activity in A`
for $i \leftarrow 2$ ***to*** n **do**
    if $s[i] \geq f[k]$ **then**
        `//` i `starts after` k `finishes - no overlap`
    end
end

Pseudocode

Greedy Activity Selection(A)

Input: a set of activities $A = a_1, a_2 \ldots, a_n$
Output: the largest subset of A that do not overlap
Sort activities in increasing order of finishing time;
$P = a_1$; `// insert the activity with earliest finishing time`
$k = 1$; `// index to the last activity in A`
for $i \leftarrow 2$ ***to*** n **do**
    if $s[i] \geq f[k]$ **then**
        `//` i `starts after` k `finishes - no overlap`
        $P \leftarrow P \cup a_i$;
        $k \leftarrow i$;
    end
end

Pseudocode

Greedy Activity Selection(A)

```
Input: a set of activities A = a_1, a_2 ..., a_n
Output: the largest subset of A that do not overlap
Sort activities in increasing order of finishing time;
P = a_1; // insert the activity with earliest finishing time
k = 1; // index to the last activity in A
for i ← 2 to n do
    if s[i] ≥ f[k] then
        // i starts after k finishes - no overlap
        P ← P ∪ a_i;
        k ← i;
    end
end
return P;
```

Pseudocode

Greedy Activity Selection(A)

```
Input: a set of activities A = a_1, a_2 ..., a_n
Output: the largest subset of A that do not overlap
Sort activities in increasing order of finishing time;  // cost: O(n log n)
P = a_1; // insert the activity with earliest finishing time
k = 1; // index to the last activity in A
for i ← 2 to n do
    if s[i] ≥ f[k] then
        // i starts after k finishes - no overlap
        P ← P ∪ a_i;
        k ← i;
    end
end
return P;
```

Pseudocode

Greedy Activity Selection(A)

Input: a set of activities $A = a_1, a_2 \ldots, a_n$
Output: the largest subset of A that do not overlap

```
Sort activities in increasing order of finishing time;   // cost: O(n log n)
P = a_1; // insert the activity with earliest finishing time
k = 1; // index to the last activity in A
for i ← 2 to n do                                           // cost: O(n)
    if s[i] ≥ f[k] then
        // i starts after k finishes - no overlap
        P ← P ∪ a_i;
        k ← i;
    end
end
return P;
```

Pseudocode

Greedy Activity Selection(A)

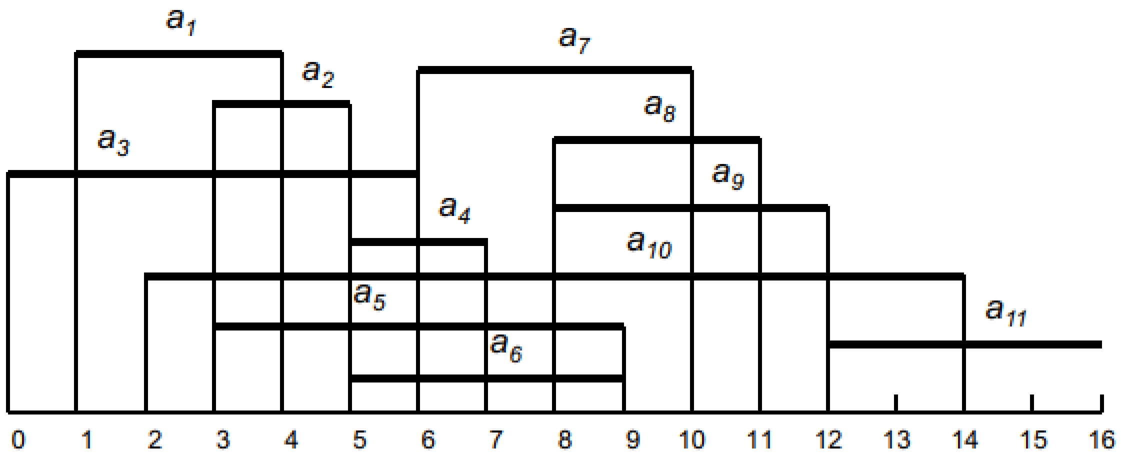
```
Input: a set of activities A = a_1, a_2 ..., a_n
Output: the largest subset of A that do not overlap
Sort activities in increasing order of finishing time;  // cost: O(n log n)
P = a_1; // insert the activity with earliest finishing time
k = 1; // index to the last activity in A
for i ← 2 to n do                                        // cost: O(n)
    if s[i] ≥ f[k] then
        // i starts after k finishes - no overlap
        P ← P ∪ a_i;
        k ← i;
    end
end
return P;
```

- Total Time Complexity: $O(n \log n)$

Example of Greedy Activity Selection Algorithm

- Sort activities in increasing order of finishing time(f_i)

a_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

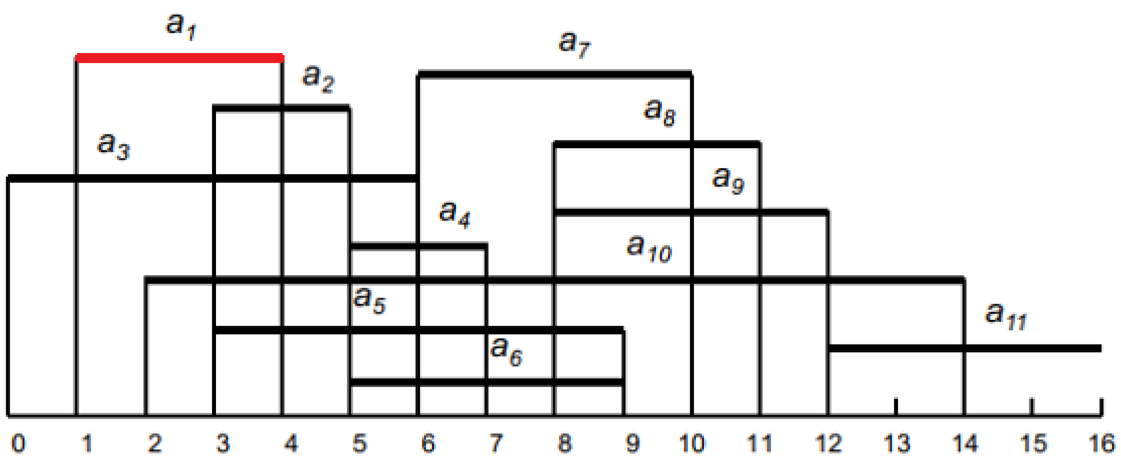


- $P = \{\}$

Example of Greedy Activity Selection Algorithm

- Insert the first activity(a_1) in the optimal solution

a_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

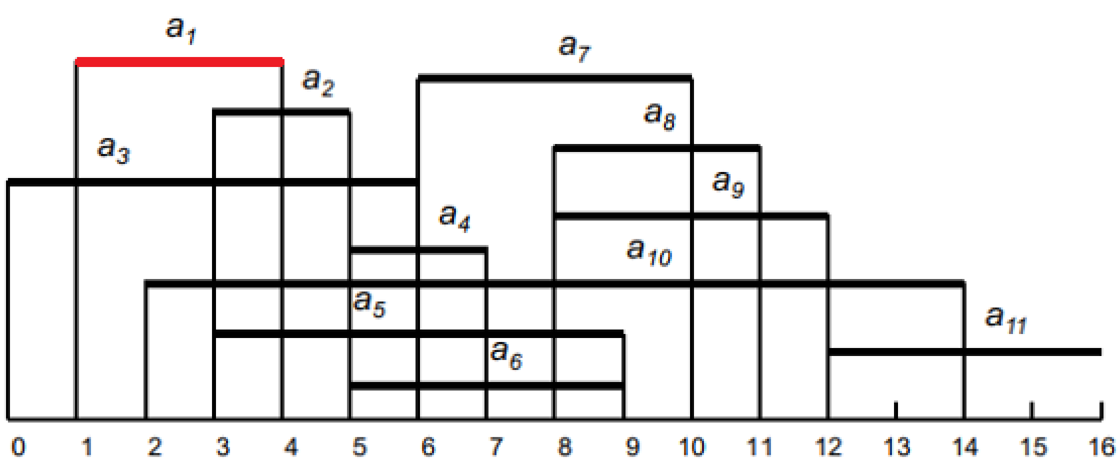


- $P = \{a_1\}$

Example of Greedy Activity Selection Algorithm

- $i = 2$, a_2 is non-compatible because $s_2 < f_1$, skip

a_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

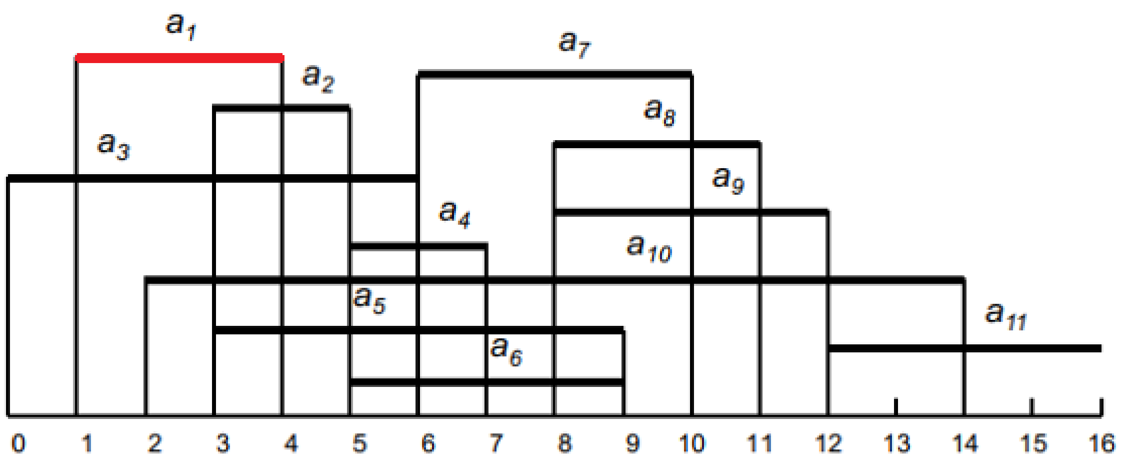


- $P = \{\mathrm{a}_1\}$

Example of Greedy Activity Selection Algorithm

- $i = 3, \mathrm{a}_3$ is also non-compatible because $s_3 < f_1$, skip

a_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

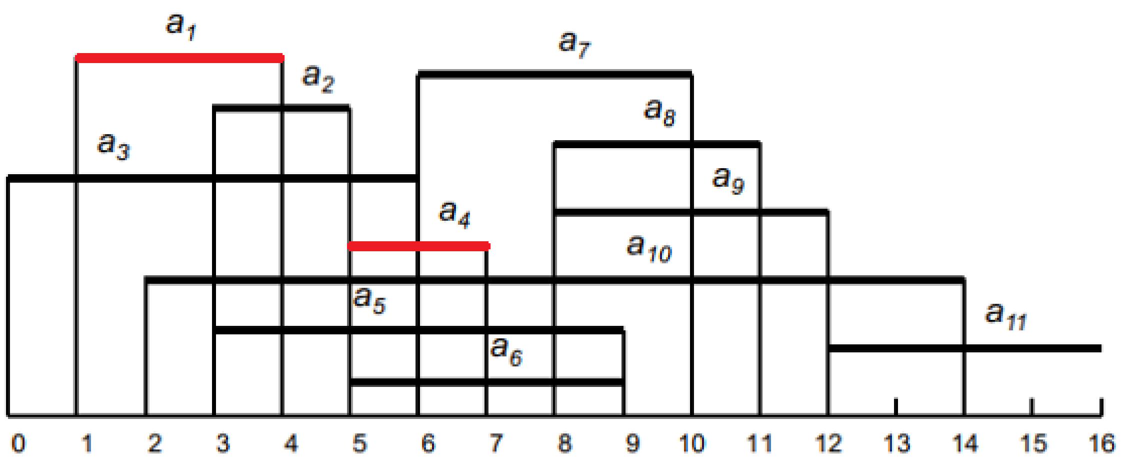


- $P = \{\mathrm{a}_1\}$

Example of Greedy Activity Selection Algorithm

- $i = 4, \mathrm{a}_4$ is a compatible choice because it satisfies $s_4 \geq f_1$, insert it in optimal solution

a_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

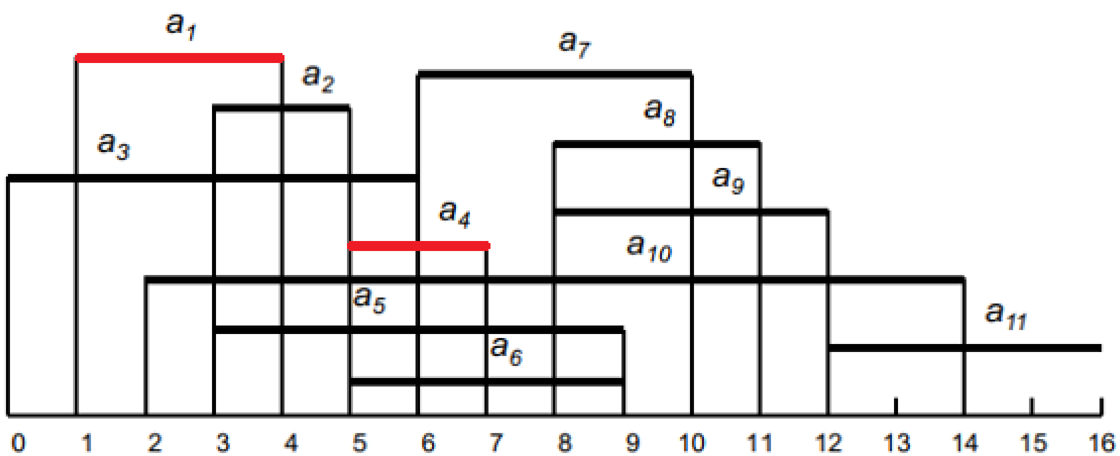


- $P = \{\mathrm{a}_1, \mathrm{a}_4\}$

Example of Greedy Activity Selection Algorithm

- $i = 5$, a_5 is non-compatible because $s_5 < f_4$, skip

a_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

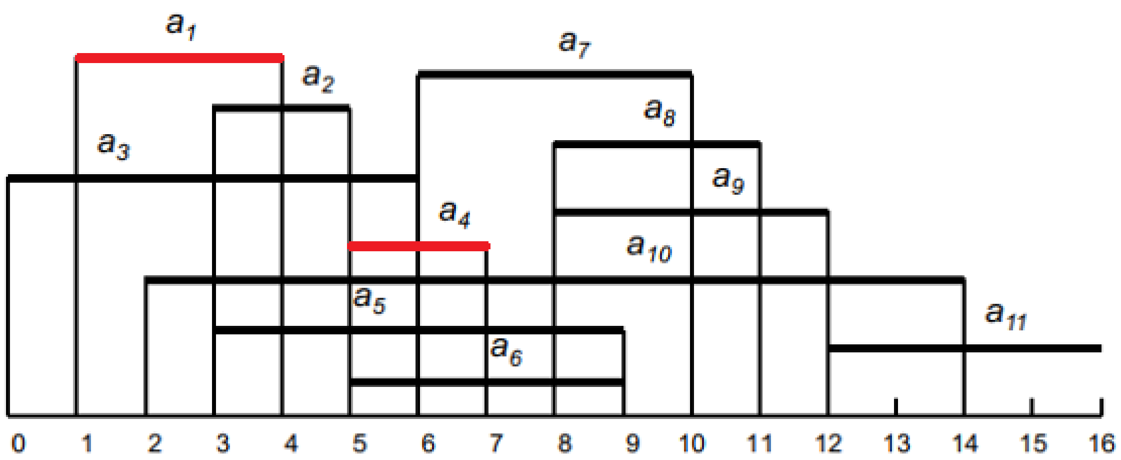


- $P = \{\mathrm{a}_1, \mathrm{a}_4\}$

Example of Greedy Activity Selection Algorithm

- $i = 6, \mathrm{a}_6$ is non-compatible because $s_6 < f_4$, skip

a_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

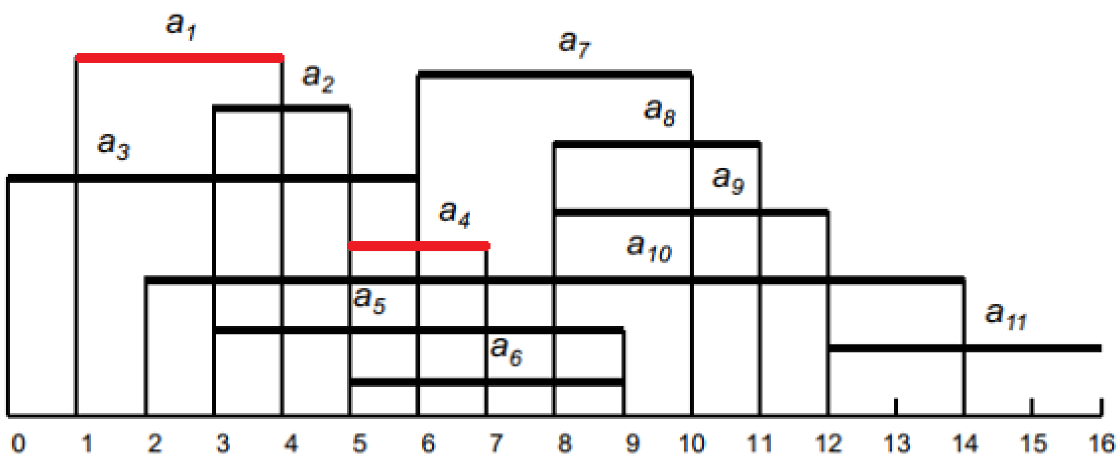


- $P = \{\mathrm{a}_1, \mathrm{a}_4\}$

Example of Greedy Activity Selection Algorithm

- $i = 7, \mathrm{a}_7$ is non-compatible because $s_7 < f_4$, skip

a_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

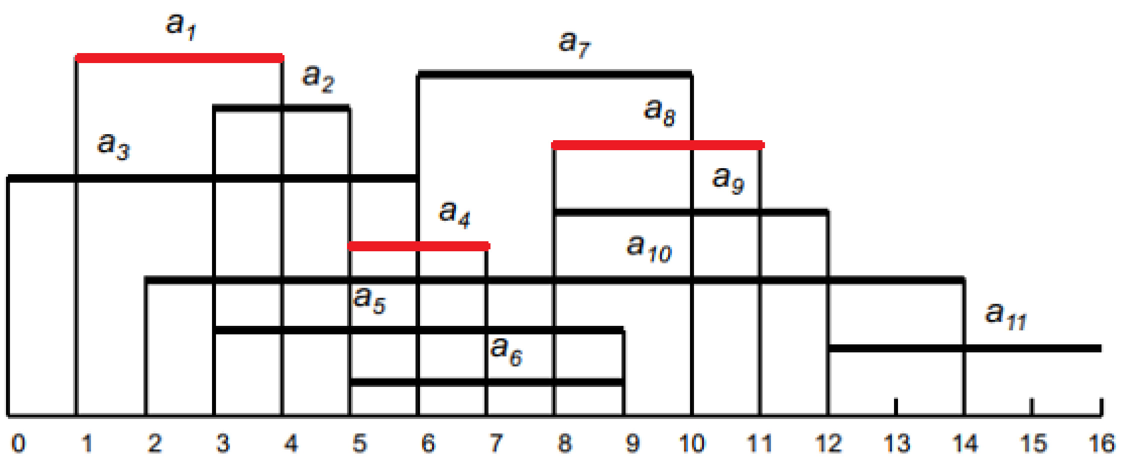


- $P = \{\mathrm{a}_1, \mathrm{a}_4\}$

Example of Greedy Activity Selection Algorithm

- $i = 8, \mathrm{a}_8$ is a compatible choice because it satisfies $s_8 \geq f_4$, insert it in optimal solution

a_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



- $P = \{\mathrm{a}_1, \mathrm{a}_4, \mathrm{a}_8\}$

Example of Greedy Activity Selection Algorithm

- $i = 9$, a_9 is non-compatible because $s_9 < f_8$, skip

a_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

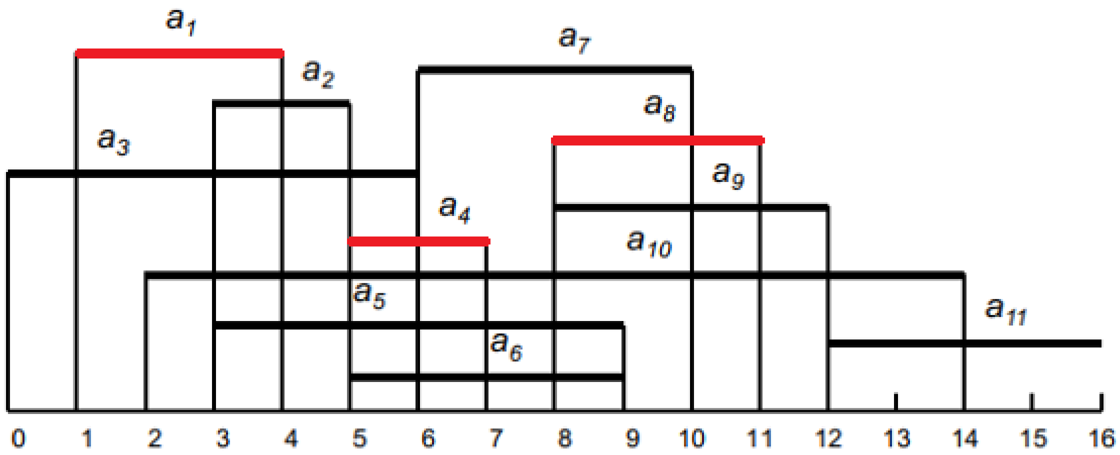
- $i = 9$, a_9 is non-compatible because $s_9 < f_8$, skip

- $P = \{\mathrm{a}_1, \mathrm{a}_4, \mathrm{a}_8\}$

Example of Greedy Activity Selection Algorithm

- $i = 10, \mathrm{a}_{10}$ is non-compatible because $s_{10} < f_8$, skip

a_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

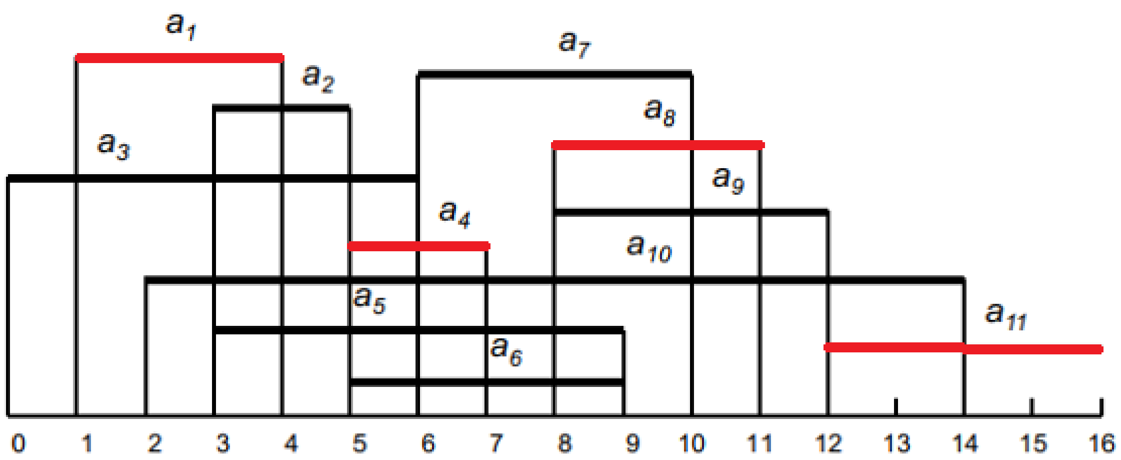


- $P = \{\mathrm{a}_1, \mathrm{a}_4, \mathrm{a}_8\}$

Example of Greedy Activity Selection Algorithm

- $i = 11, \mathrm{a}_{11}$ is a compatible choice because it satisfies $s_{11} \geq f_8$, insert it in optimal solution

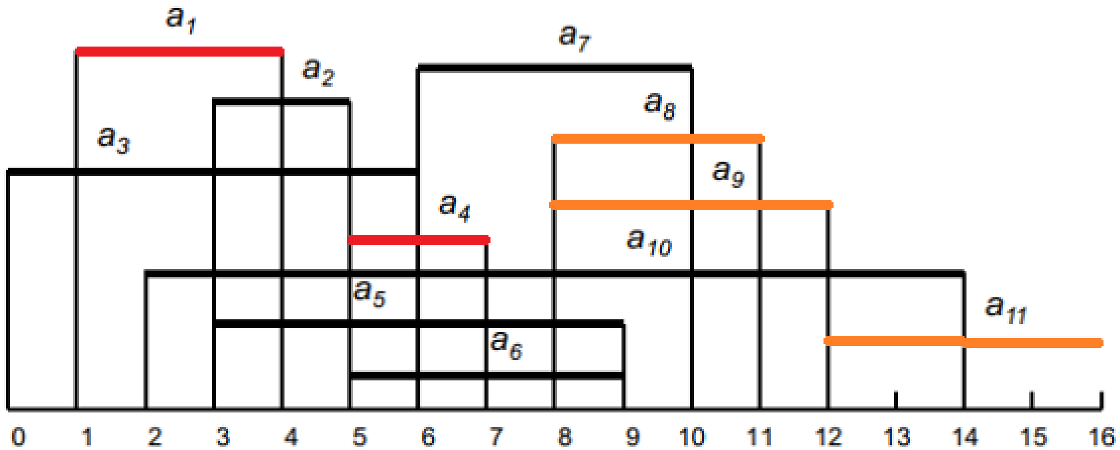
a_i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



- $P = \{\mathrm{a}_1, \mathrm{a}_4, \mathrm{a}_8, \mathrm{a}_{11}\}$

Correctness

- **Theorem**:
  - If a_k is a selected activity by Greedy Activity Selection algorithm, S_k (activities that start after a_k finishes) is nonempty and a_m has the earliest finishing time in S_k, then a_m is included in some optimal solution.



When we insert a_4 in optimal solution,

$S_4 = \{\mathrm{a}_8, \mathrm{a}_9, \mathrm{a}_{11}\}$

$\mathrm{a}_m = a_8$

Correctness

- How can we prove it?

Correctness

- How can we prove it?
  - We can convert any other optimal solution(P') to the greedy algorithm solution(P)

Correctness

- How can we prove it?
  - We can convert any other optimal solution(P') to the greedy algorithm solution(P)
- Idea:

Correctness

- How can we prove it?
  - We can convert any other optimal solution(P') to the greedy algorithm solution(P)

- Idea:
  - Compare the activities in P' and P from left to right

Correctness

- How can we prove it?
  - We can convert any other optimal solution(P') to the greedy algorithm solution(P)

- Idea:
  - Compare the activities in P' and P from left to right
  - If they match in the selected activity $\rightarrow$ skip

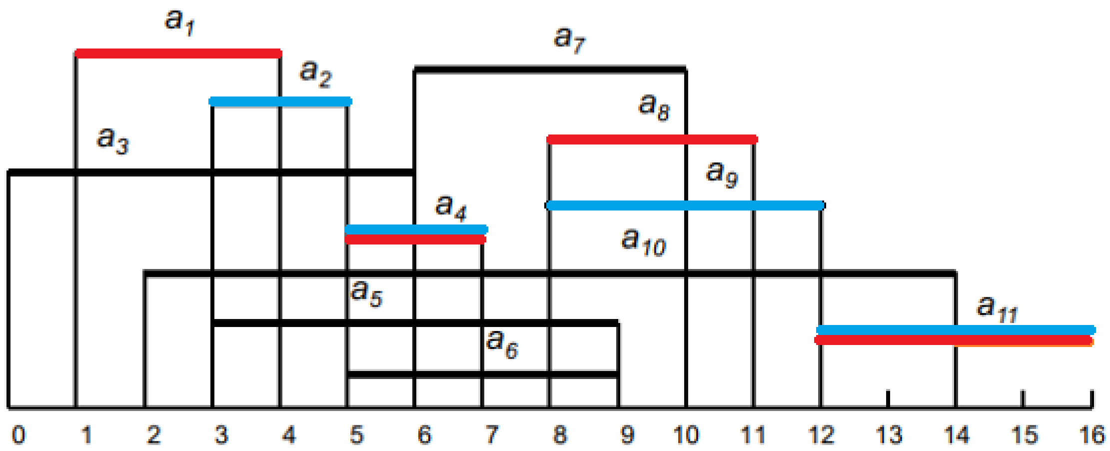
Correctness

- How can we prove it?
  - We can convert any other optimal solution(P') to the greedy algorithm solution(P)

- Idea:
  - Compare the activities in P' and P from left to right
  - If they match in the selected activity $\rightarrow$ skip
  - If they do not match, we can replace the activity in P' by that in P because the one in P finish first

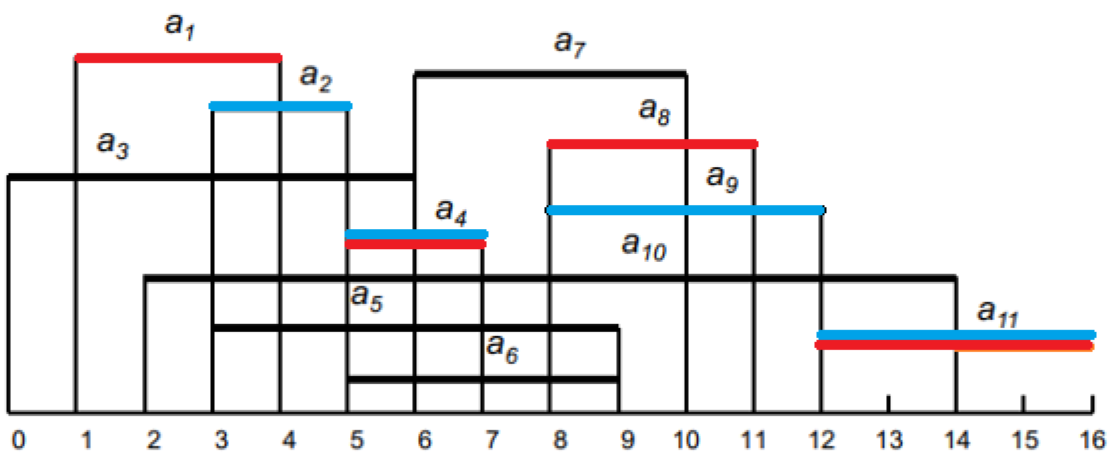
Correctness

- $P = \{\mathrm{a}_1, \mathrm{a}_4, \mathrm{a}_8, \mathrm{a}_{11}\}$
- $P' = \{\mathrm{a}_2, \mathrm{a}_4, \mathrm{a}_9, \mathrm{a}_{11}\}$



Correctness

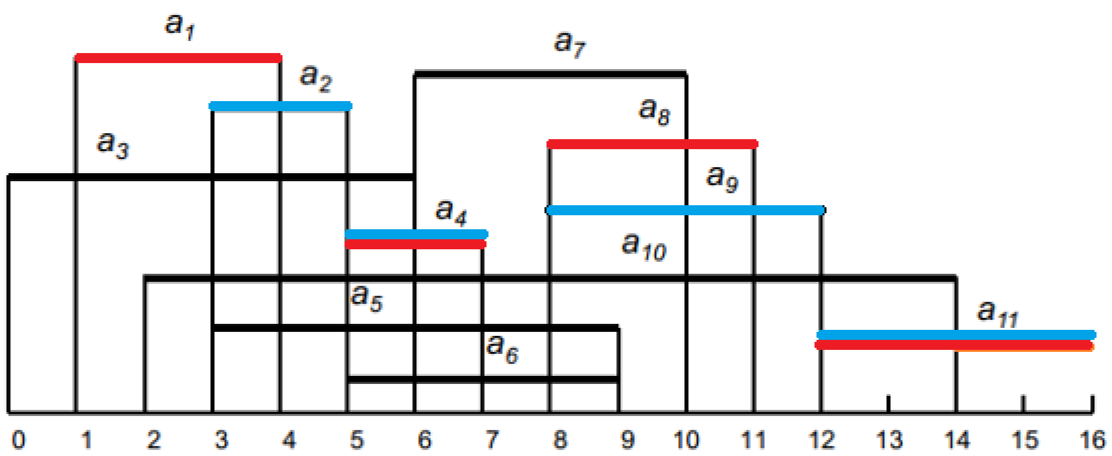
- $P = \{a_1, a_4, a_8, a_{11}\}$
- $P' = \{a_2, a_4, a_9, a_{11}\}$



- We can map P' to P

Correctness

- $P = \{a_1, a_4, a_8, a_{11}\}$
- $P' = \{a_2, a_4, a_9, a_{11}\}$



- We can map P' to P
- a_2, a_9 in P' can be replaced by a_1, a_8 from P (finishes earlier)

Correctness

- Proving a greedy solution is optimal:

Correctness

- Proving a greedy solution is optimal:
  - Remember: Not all problems have optimal greedy solution.

Correctness

- Proving a greedy solution is optimal:
  - Remember: Not all problems have optimal greedy solution.
  - If it does, you need to prove it.

Correctness

- Proving a greedy solution is optimal:
  - Remember: Not all problems have optimal greedy solution.
  - If it does, you need to prove it.
  - Usually the proof includes mapping or converting any other optimal solution to the greedy solution.

Outline

- Introduction to Part III
- Fractional Knapsack Problem
    - Problem Definition
    - A Greedy Algorithm and correctness
- Huffman Coding Problem
    - Problem Definition
    - A Greedy Algorithm
- Activity Selection Problem
    - Problem Definition
    - A Greedy Algorithm and correctness
    - Extended: Weighted Activity Selection

Weighted Activity Selection Problem

- Given a set A of activities, such that each activity a_i has a start time s_i, a finishing time f_i and a weight $w_i > 0$.

Weighted Activity Selection Problem

- Given a set A of activities, such that each activity a_i has a start time s_i, a finishing time f_i and a weight $w_i > 0$.

- Two activities are compatible if they do not overlap.

Weighted Activity Selection Problem

- Given a set A of activities, such that each activity a_i has a start time s_i, a finishing time f_i and a weight $w_i > 0$.
- Two activities are compatible if they do not overlap.
- Goal: find max-weighted subset of mutually compatible activities.

Weighted Activity Selection Problem

- Given a set A of activities, such that each activity a_i has a start time s_i, a finishing time f_i and a weight $w_i > 0$.

- Two activities are compatible if they do not overlap.

- Goal: find max-weighted subset of mutually compatible activities.

- That is, find a subset S of A, such that:

$$s_i \geq f_j \; or \; f_i \leq s_j \qquad \forall \mathrm{a}_i, \mathrm{a}_j \in S, i \neq j$$

  and the following is maximized:

$$\sum_{a_i \in S} w_i$$

Greedy Activity Selection Algorithm

- Can we use the Greedy Activity Selection algorithm to solve the Weighted Activity Selection Problem?

Greedy Activity Selection Algorithm

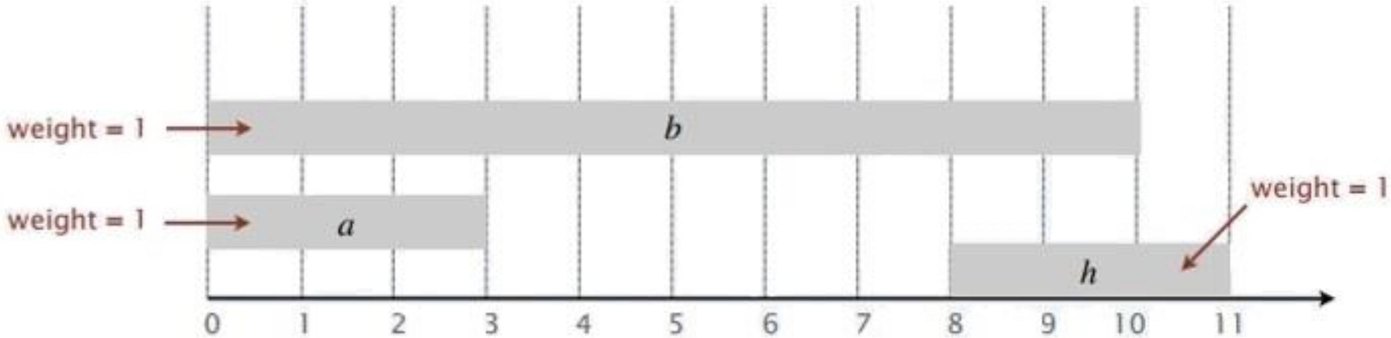
- Can we use the Greedy Activity Selection algorithm to solve the Weighted Activity Selection Problem?
- Main Idea.
  - Consider activities in ascending order of finishing time.

Greedy Activity Selection Algorithm

- Can we use the Greedy Activity Selection algorithm to solve the Weighted Activity Selection Problem?
- Main Idea.
  - Consider activities in ascending order of finishing time.
  - Add an activity to subset if it is compatible with previously chosen activities.

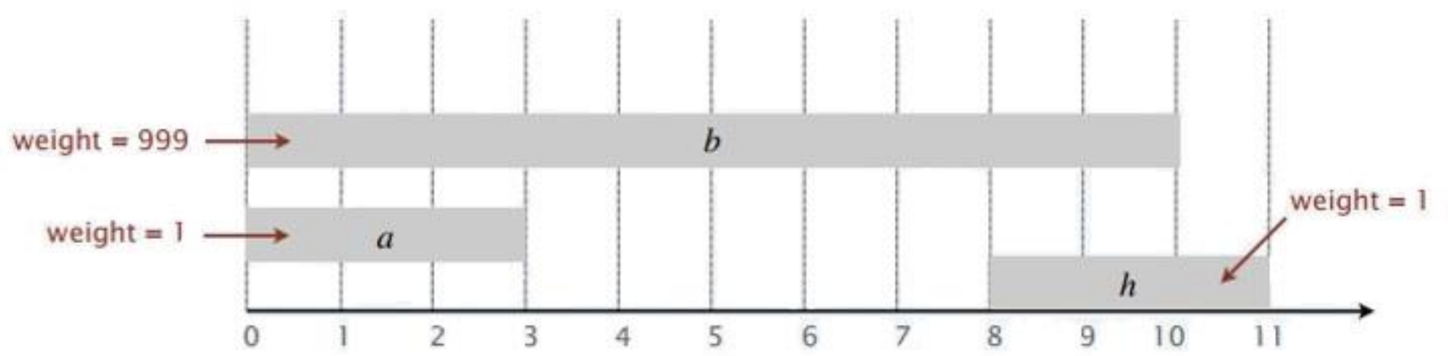
Greedy Activity Selection Algorithm

- Can we use the Greedy Activity Selection algorithm to solve the Weighted Activity Selection Problem?
- Main Idea.
  - Consider activities in ascending order of finishing time.
  - Add an activity to subset if it is compatible with previously chosen activities.
- Recall. Greedy algorithm is correct if all weights are 1.



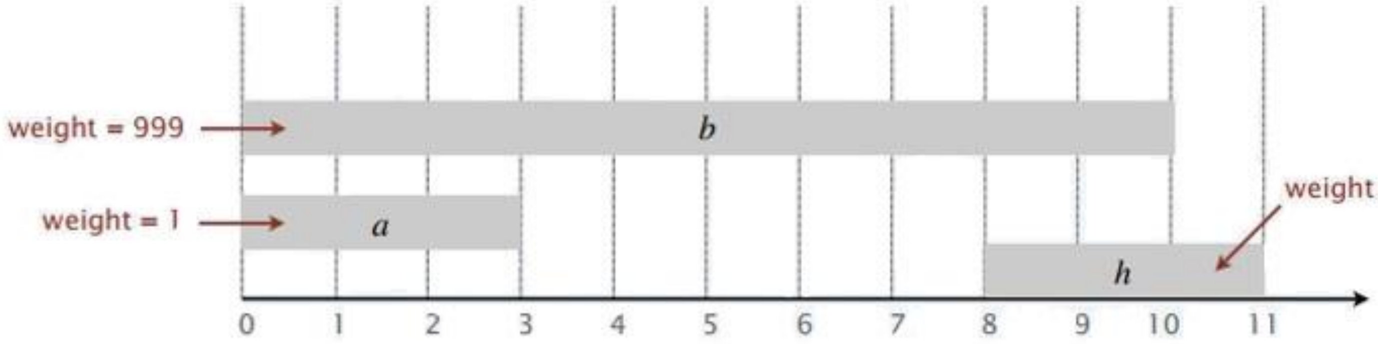
Greedy Activity Selection Algorithm

- Can we use the Greedy Activity Selection algorithm to solve the Weighted Activity Selection Problem?
- Main Idea.
  - Consider activities in ascending order of finishing time.
  - Add an activity to subset if it is compatible with previously chosen activities.
- Recall. Greedy algorithm is correct if all weights are 1.



Greedy Activity Selection Algorithm

- Can we use the Greedy Activity Selection algorithm to solve the Weighted Activity Selection Problem?
- Main Idea.
  - Consider activities in ascending order of finishing time.
  - Add an activity to subset if it is compatible with previously chosen activities.
- Recall. Greedy algorithm is correct if all weights are 1.
- Observation. Greedy algorithm fails spectacularly for weighted version.



A Dynamic Programming Algorithm

Step 1: Space of Subproblems (State)

- Convention. Activities are in ascending order of finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$

A Dynamic Programming Algorithm

Step 1: Space of Subproblems (State)

- Convention. Activities are in ascending order of finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$

- Definition. $OPT(j)$ is the max weight of any subset of mutually compatible activities for subproblem consisting only of activities $\mathrm{a}_1\mathrm{a}_2, \cdots, \mathrm{a}_j$

A Dynamic Programming Algorithm

Step 1: Space of Subproblems (State)

- Convention. Activities are in ascending order of finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$

- Definition. $OPT(j)$ is the max weight of any subset of mutually compatible activities for subproblem consisting only of activities $\mathrm{a}_1 \mathrm{a}_2, \cdots, \mathrm{a}_j$

- Goal. $OPT(n)$ is the max weight of any subset of mutually compatible activities.

A Dynamic Programming Algorithm

Step 1: Space of Subproblems (State)

- Convention. Activities are in ascending order of finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$

- Definition. $OPT(j)$ is the max weight of any subset of mutually compatible activities for subproblem consisting only of activities $\mathrm{a}_1 \mathrm{a}_2, \cdots, \mathrm{a}_j$

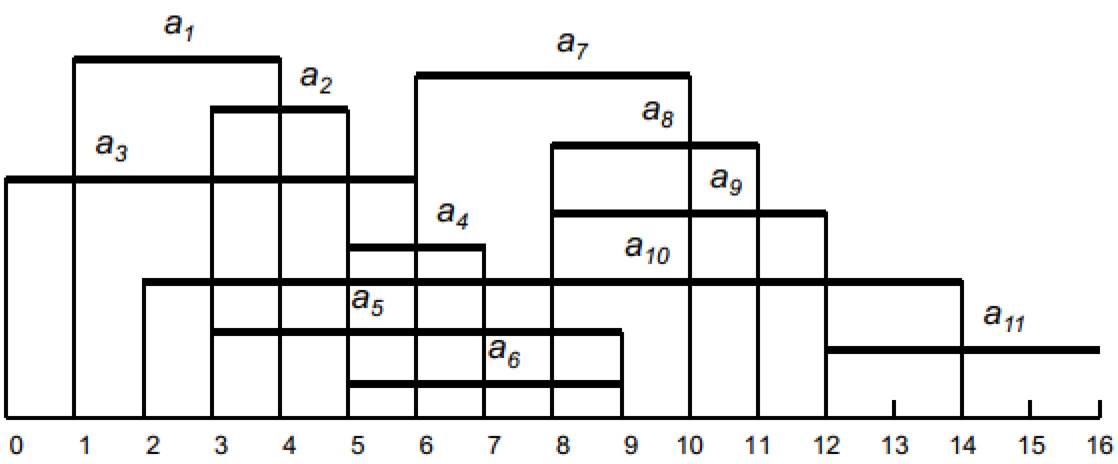
- Goal. $OPT(n)$ is the max weight of any subset of mutually compatible activities.

- Boundary case. $OPT(0) = 0$

A Dynamic Programming Algorithm

Step 2: Relating a problem to its subproblems

- Definition. $p(j)$ is the largest index $i < j$ such that activity a_i is compatible with j.
  - Ex. $p(7) = 3, p(8) = 4, p(11) = 9$
  - $p(j)$ can be easily computed via binary search



A Dynamic Programming Algorithm

Step 2: Relating a problem to its subproblems

- Case 1. $OPT(j)$ does not select activity a_j.
  - Must be an optimal solution to problem consisting of remaining activities $\mathrm{a}_1, \mathrm{a}_2, \cdots, \mathrm{a}_{j-1}$

A Dynamic Programming Algorithm

Step 2: Relating a problem to its subproblems

- Case 1. $OPT(j)$ does not select activity a_j.
  - Must be an optimal solution to problem consisting of remaining activities $\mathrm{a}_1, \mathrm{a}_2, \cdots, \mathrm{a}_{j-1}$
- Case 2. $OPT(j)$ selects activity a_j.
  - Collect profit w_j
  - Can't use incompatible activities $\{\mathrm{a}_{p(j)+1}, \mathrm{a}_{p(j)+2}, \cdots, \mathrm{a}_{j-1}\}$
  - Must include optimal solution to problem consisting of remaining compatible activities $\mathrm{a}_1, \mathrm{a}_2, \cdots, \mathrm{a}_{p(j)}$

A Dynamic Programming Algorithm

Step 2: Relating a problem to its subproblems

- Case 1. $OPT(j)$ does not select activity a_j.
  - Must be an optimal solution to problem consisting of remaining activities $\mathrm{a}_1, \mathrm{a}_2, \cdots, \mathrm{a}_{j-1}$
- Case 2. $OPT(j)$ selects activity a_j.
  - Collect profit w_j
  - Can't use incompatible activities $\{\mathrm{a}_{p(j)+1}, \mathrm{a}_{p(j)+2}, \cdots, \mathrm{a}_{j-1}\}$
  - Must include optimal solution to problem consisting of remaining compatible activities $\mathrm{a}_1, \mathrm{a}_2, \cdots, \mathrm{a}_{p(j)}$
- Dynamic Programming Equation:

$$OPT(j) = \begin{cases} 0 & if\ j = 0 \\ \max\{OPT(j-1), w_j + OPT(p(j))\} & if\ j > 0 \end{cases}$$

A Dynamic Programming Algorithm

Step 3: Bottom-up computation

- Dynamic Programming Equation:

$$OPT(j) = \begin{cases} 0 & if\ j = 0 \\ \max\{OPT(j-1), w_j + OPT(p(j))\} & if\ j > 0 \end{cases}$$

A Dynamic Programming Algorithm

Step 3: Bottom-up computation

- Dynamic Programming Equation:

$$OPT(j) = \begin{cases} 0 & if\ j = 0 \\ \max\{OPT(j-1), w_j + OPT(p(j))\} & if\ j > 0 \end{cases}$$

- We compute and save $OPT(j)$ in such an order that: When it is time to compute $OPT(j)$, the values of $OPT(j-1)$ and $OPT(p(j))$ are available

A Dynamic Programming Algorithm: Pseudocode

DP Activity Selection(A)

Input: a set of activities $A = a_1, a_2, \cdots, a_n$
Output: the max weight of any subset of mutually compatible activities

A Dynamic Programming Algorithm: Pseudocode

DP Activity Selection(A)

Input: a set of activities $A = a_1, a_2, \cdots, a_n$
Output: the max weight of any subset of mutually compatible activities
Sort activities by finishing time and renumber so that $f_1 \leq f_2 \leq \cdots \leq f_n$;

A Dynamic Programming Algorithm: Pseudocode

DP Activity Selection(A)

Input: a set of activities $A = a_1, a_2, \cdots, a_n$
Output: the max weight of any subset of mutually compatible activities
Sort activities by finishing time and renumber so that $f_1 \leq f_2 \leq \cdots \leq f_n$;
Compute $p[1], p[2], \cdots, p[n]$

A Dynamic Programming Algorithm: Pseudocode

DP Activity Selection(A)

Input: a set of activities $A = a_1, a_2, \cdots, a_n$
Output: the max weight of any subset of mutually compatible activities
Sort activities by finishing time and renumber so that $f_1 \leq f_2 \leq \cdots \leq f_n$;
Compute $p[1], p[2], \cdots, p[n]$
$OPT[0] \leftarrow 0$;
for $j = 1$ ***to*** n **do**
  $OPT[j] \leftarrow \max\{OPT[j-1], w_j + OPT[p[j]]\}$;
end
return $OPT[n]$;

A Dynamic Programming Algorithm: Pseudocode

DP Activity Selection(A)

Input: a set of activities $A = a_1, a_2, \cdots, a_n$
Output: the max weight of any subset of mutually compatible activities
Sort activities by finishing time and renumber so that $f_1 \leq f_2 \leq \cdots \leq f_n$;
Compute $p[1], p[2], \cdots, p[n]$
$OPT[0] \leftarrow 0$;
for $j = 1$ ***to*** n **do**
  $OPT[j] \leftarrow \max\{OPT[j-1], w_j + OPT[p[j]]\}$;
end
return $OPT[n]$;

Previously computed values

A Dynamic Programming Algorithm: Pseudocode

DP Activity Selection(A)

Input: a set of activities $A = a_1, a_2, \cdots, a_n$
Output: the max weight of any subset of mutually compatible activities
Sort activities by finishing time and renumber so that $f_1 \leq f_2 \leq \cdots \leq f_n$;
Compute $p[1], p[2], \cdots, p[n]$
$OPT[0] \leftarrow 0$;
for $j = 1$ ***to*** n **do**
  $OPT[j] \leftarrow \max\{OPT[j-1], w_j + OPT[p[j]]\}$;
end
return $OPT[n]$;

cost: $O(n \log n)$

A Dynamic Programming Algorithm: Pseudocode

DP Activity Selection(A)

Input: a set of activities $A = a_1, a_2, \cdots, a_n$
Output: the max weight of any subset of mutually compatible activities
Sort activities by finishing time and renumber so that $f_1 \le f_2 \le \cdots \le f_n$;
Compute $p[1], p[2], \cdots, p[n]$
$OPT[0] \leftarrow 0$;
for $j = 1$ ***to*** n **do** $\longrightarrow$ cost: $O(n)$
   $OPT[j] \leftarrow \max\{OPT[j-1], w_j + OPT[p[j]]\}$;
end
return $OPT[n]$;

A Dynamic Programming Algorithm: Pseudocode

DP Activity Selection(A)

Input: a set of activities $A = a_1, a_2, \cdots, a_n$
Output: the max weight of any subset of mutually compatible activities
Sort activities by finishing time and renumber so that $f_1 \le f_2 \le \cdots \le f_n$;
Compute $p[1], p[2], \cdots, p[n]$
$OPT[0] \leftarrow 0$;
for $j = 1$ ***to*** n **do**
  $OPT[j] \leftarrow \max\{OPT[j-1], w_j + OPT[p[j]]\}$;
end
return $OPT[n]$;

- Total Time Complexity: $O(n \log n)$

A Dynamic Programming Algorithm: Pseudocode

DP Activity Selection(A)

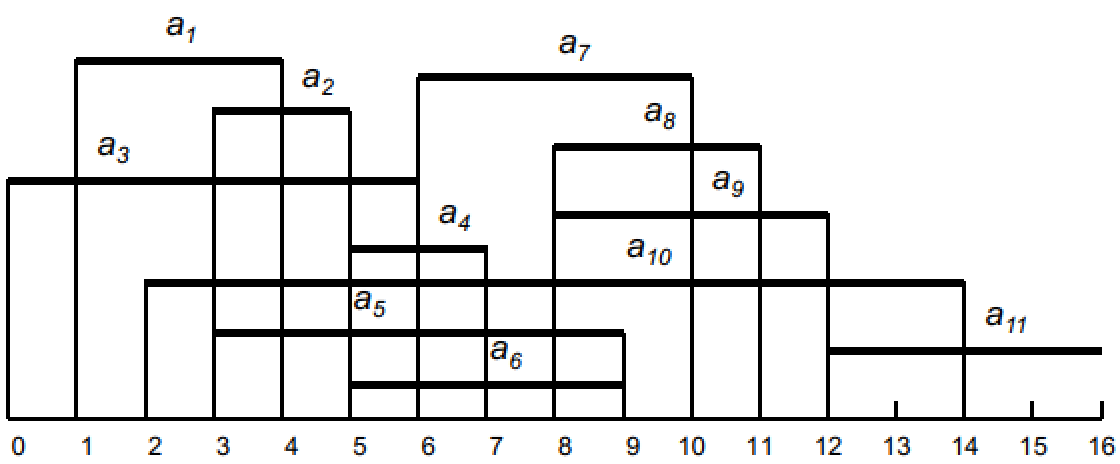
Input: a set of activities $A = a_1, a_2, \cdots, a_n$
Output: the max weight of any subset of mutually compatible activities
Sort activities by finishing time and renumber so that $f_1 \leq f_2 \leq \cdots \leq f_n$;
Compute $p[1], p[2], \cdots, p[n]$
$OPT[0] \leftarrow 0$;
for $j = 1$ ***to*** n **do**
    $OPT[j] \leftarrow \max\{OPT[j-1], w_j + OPT[p[j]]\}$;
end
return $OPT[n]$;

- Total Time Complexity: $O(n \log n)$
- How to construct the optimal solution (Step 4)?
    - Try by yourself

Example of Optimal Solution Construction

- Sort activities in increasing order of finishing time(f_i)

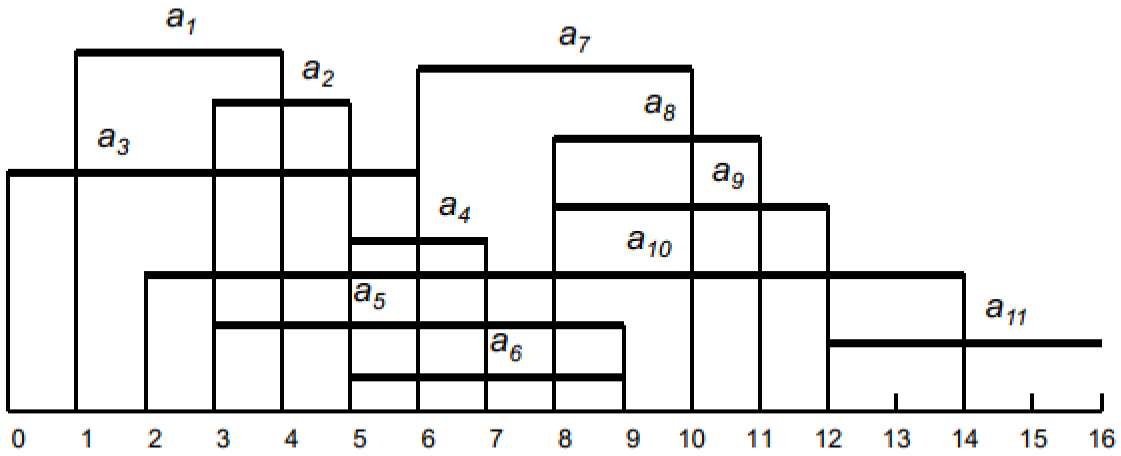
a_i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16
w_i	5	8	2	4	9	2	5	8	3	6	1



Example of Optimal Solution Construction

- Compute $p(1), p(2), \cdots, p(n)$ via binary search

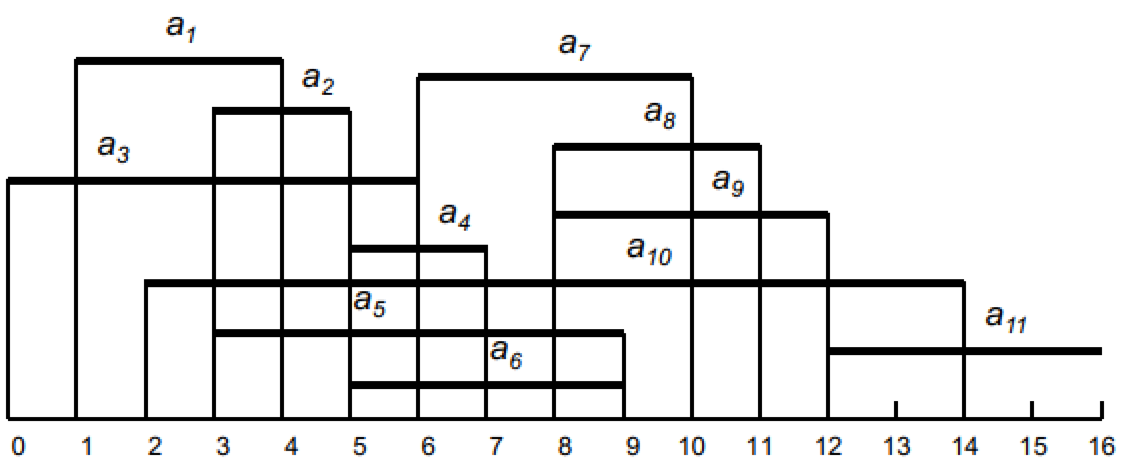
a_i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9



Example of Optimal Solution Construction

- Boundary case. $OPT(0) = 0$

a_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
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9

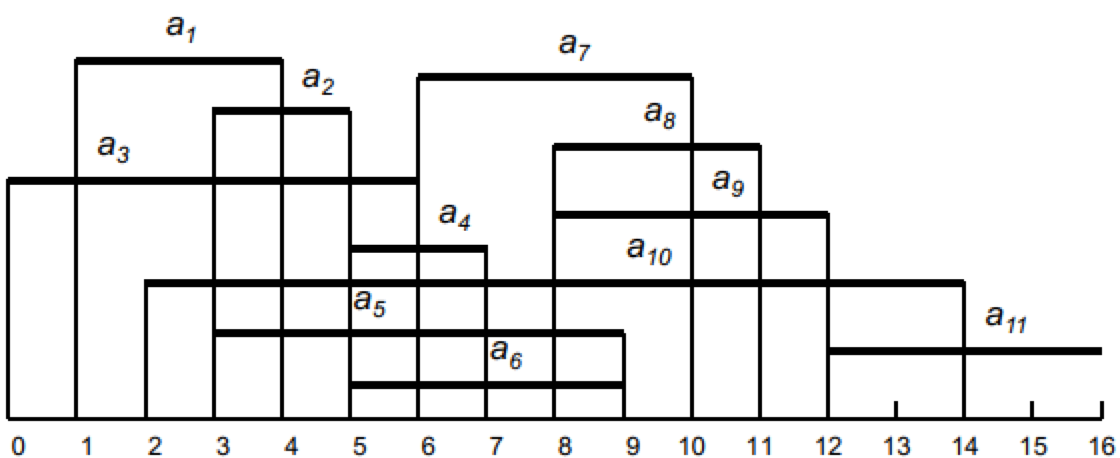


i	0	1	2	3	4	5	6	7	8	9	10	11
OPT	0											

Example of Optimal Solution Construction

- $OPT(1) = \max\big(OPT(0), w_1 + OPT(p(1))\big) = 5$

a_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
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9

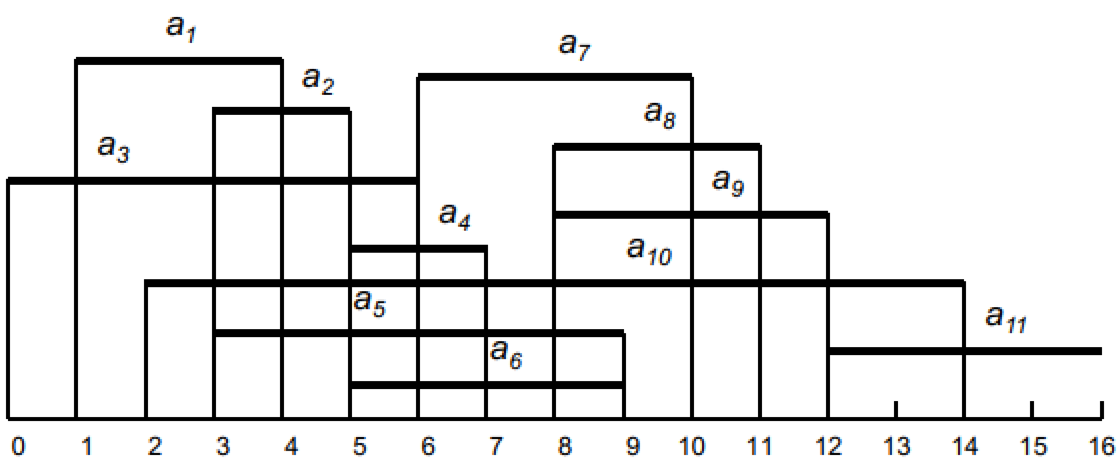


i	0	1	2	3	4	5	6	7	8	9	10	11
OPT	0	5										

Example of Optimal Solution Construction

- $OPT(2) = \max\big(OPT(1), w_2 + OPT(p(2))\big) = 8$

a_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
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9

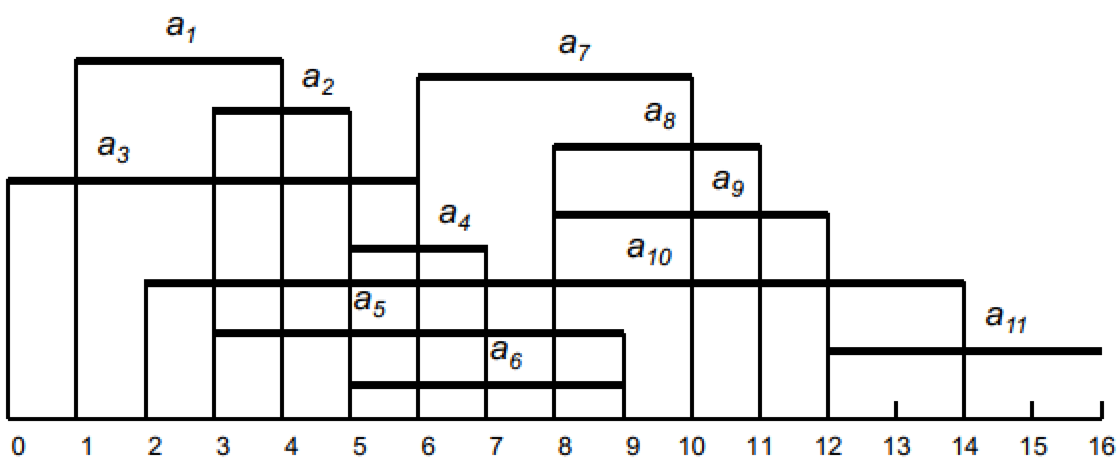


i	0	1	2	3	4	5	6	7	8	9	10	11
OPT	0	5	8									

Example of Optimal Solution Construction

- $OPT(3) = \max\big(OPT(2), w_3 + OPT(p(3))\big) = 8$

a_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
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9

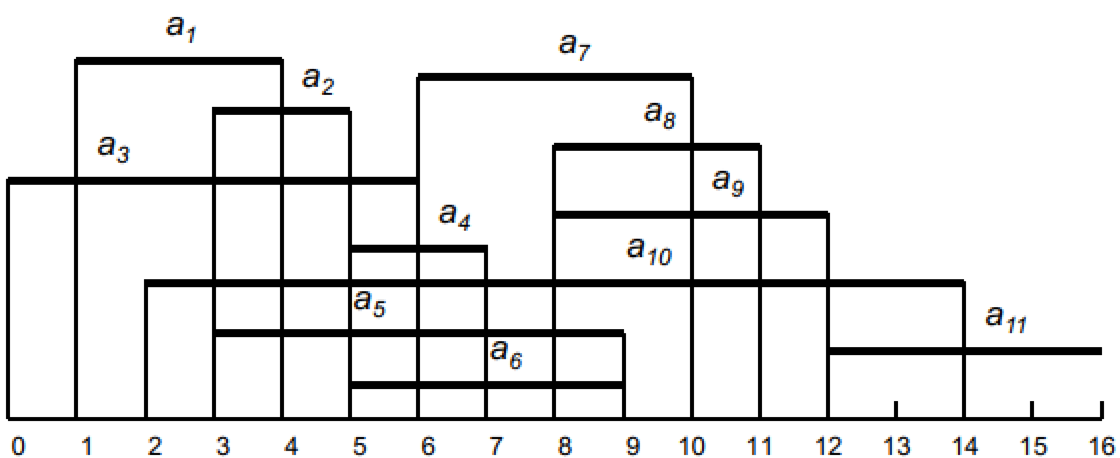


i	0	1	2	3	4	5	6	7	8	9	10	11
OPT	0	5	8	8								

Example of Optimal Solution Construction

- $OPT(4) = \max\left(OPT(3), w_4 + OPT(p(4))\right) = 12$

a_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
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9

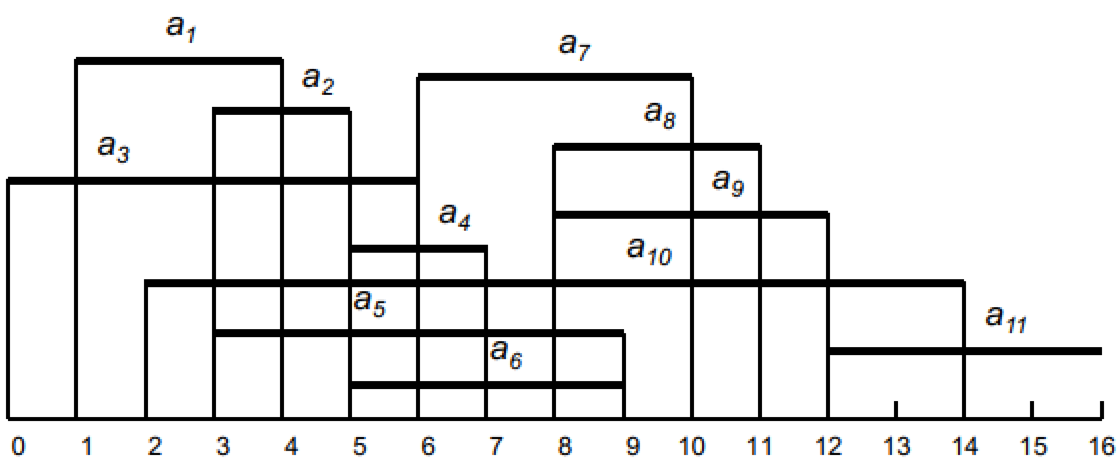


i	0	1	2	3	4	5	6	7	8	9	10	11
OPT	0	5	8	8	12							

Example of Optimal Solution Construction

- $OPT(5) = \max\big(OPT(4), w_5 + OPT(p(5))\big) = 12$

a_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
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9

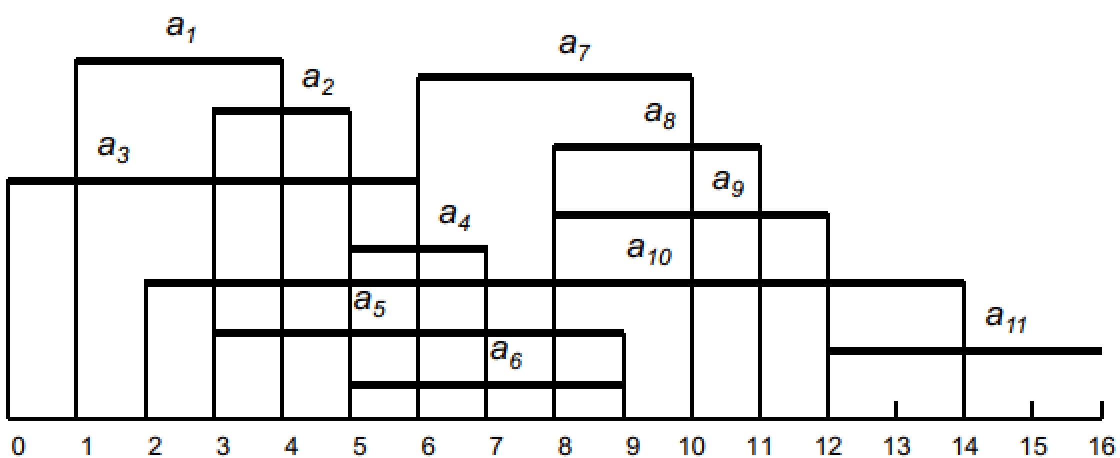


i	0	1	2	3	4	5	6	7	8	9	10	11
OPT	0	5	8	8	12	12						

Example of Optimal Solution Construction

- $OPT(6) = \max\big(OPT(5), w_6 + OPT(p(6))\big) = 12$

a_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
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9

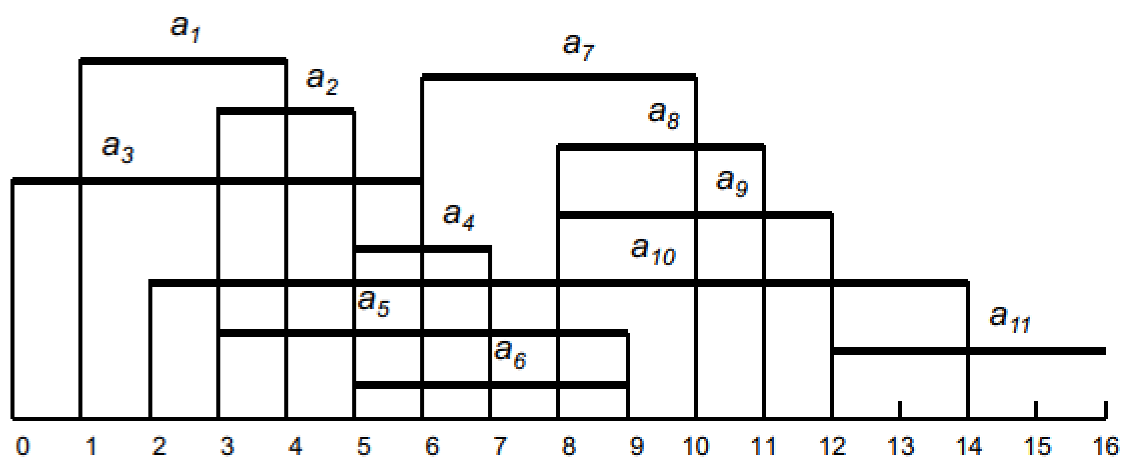


i	0	1	2	3	4	5	6	7	8	9	10	11
OPT	0	5	8	8	12	12	12					

Example of Optimal Solution Construction

- $OPT(7) = \max\big(OPT(6), w_7 + OPT(p(7))\big) = 13$

a_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
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9

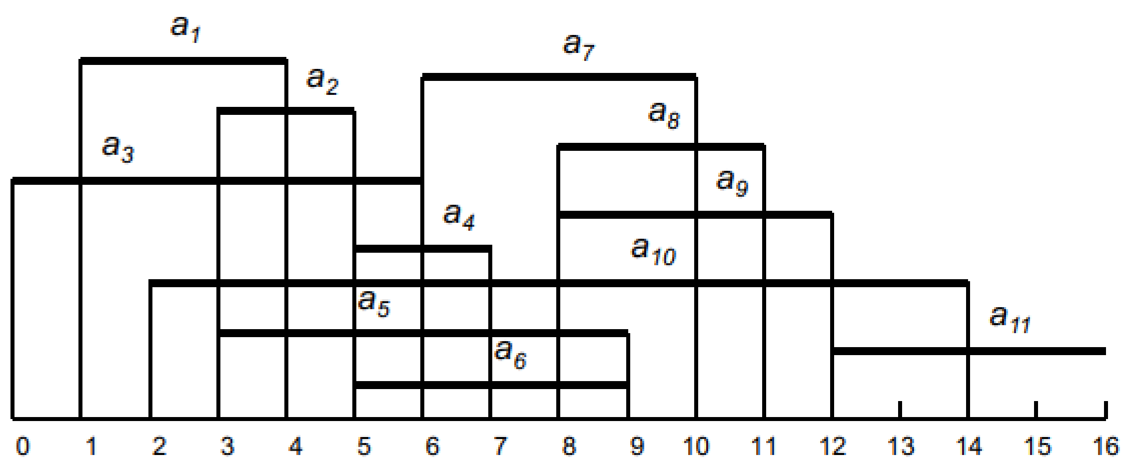


i	0	1	2	3	4	5	6	7	8	9	10	11
OPT	0	5	8	8	12	12	12	13				

Example of Optimal Solution Construction

- $OPT(8) = \max\big(OPT(7), w_8 + OPT(p(8))\big) = 20$

a_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
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9

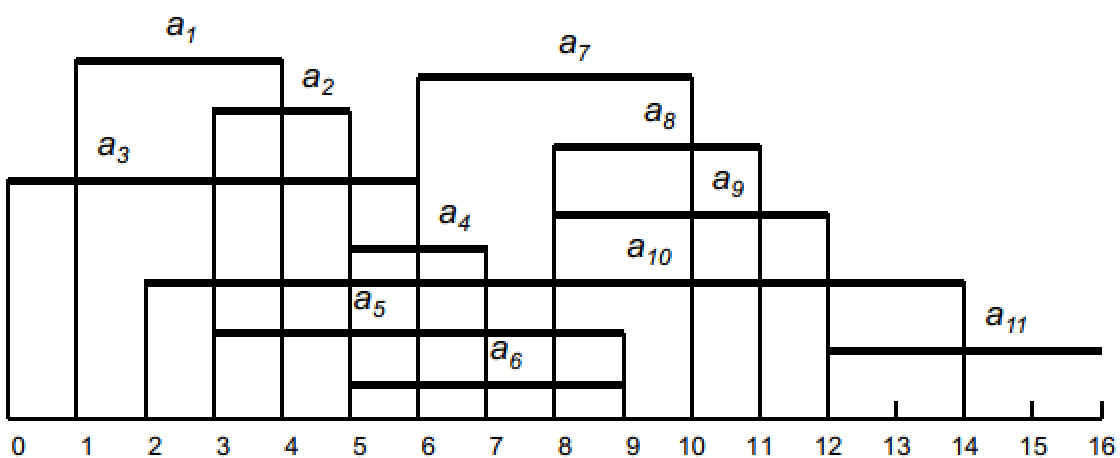


i	0	1	2	3	4	5	6	7	8	9	10	11
OPT	0	5	8	8	12	12	12	13	20			

Example of Optimal Solution Construction

- $OPT(9) = \max\big(OPT(8), w_9 + OPT(p(9))\big) = 20$

a_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
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9

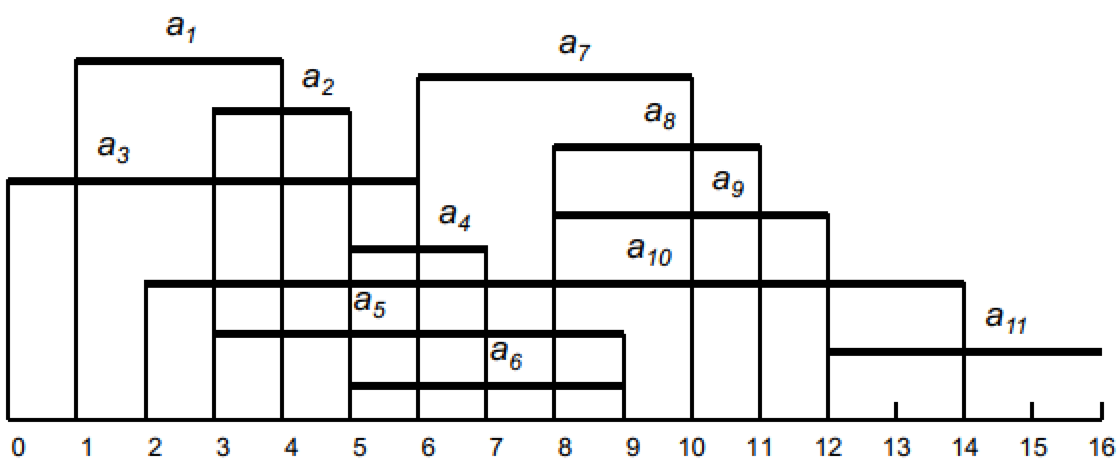


i	0	1	2	3	4	5	6	7	8	9	10	11
OPT	0	5	8	8	12	12	12	13	20	20		

Example of Optimal Solution Construction

- $OPT(10) = \max\big(OPT(9), w_{10} + OPT(p(10))\big) = 20$

a_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
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9

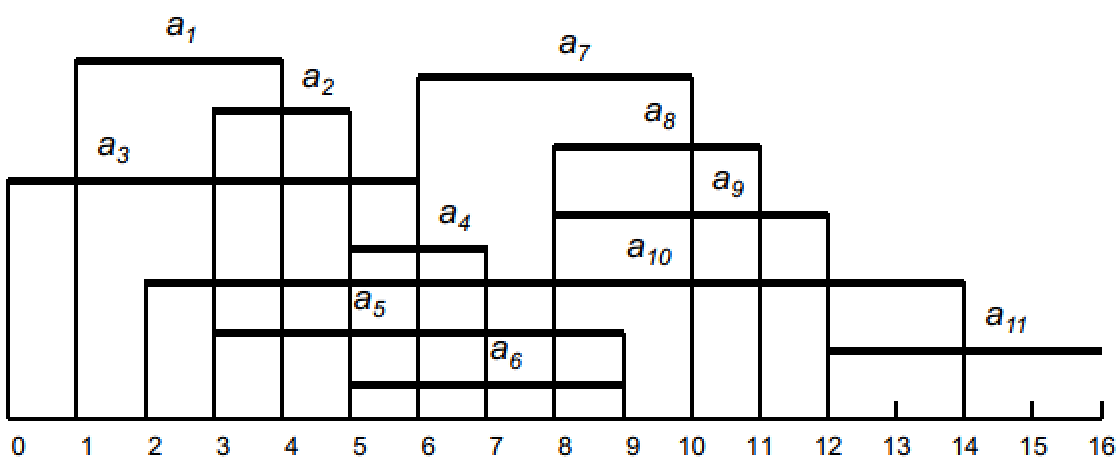


i	0	1	2	3	4	5	6	7	8	9	10	11
OPT	0	5	8	8	12	12	12	13	20	20	20	

Example of Optimal Solution Construction

- $OPT(11) = \max\big(OPT(10), w_{11} + OPT(p(11))\big) = 21$

a_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
w_i	5	8	2	4	9	2	5	8	3	6	1
$p(i)$	0	0	0	2	0	2	3	4	4	0	9



i	0	1	2	3	4	5	6	7	8	9	10	11
OPT	0	5	8	8	12	12	12	13	20	20	20	21

Greedy vs. Dynamic Programming

- Greedy
  - Build up a solution incrementally, myopically optimizing some local criterion

- Dynamic Programming
  - Break up a problem into a series of overlapping subproblems; combine solutions to smaller subproblems to form solution to large subproblem

Greedy	never reconsiders its choices	focus on the present
Dynamic Programming	based on previous decisions	record the history


thank you
danke
рахмат
謝謝
ngiyabonga
teşekkür ederim
Баярлалаа
спасибо
faafetai lava
mersi
kia ora
barka
welalin
tack
spas
dank je
misaotra
matondo
paldies
grazzi
mahalo
tapadh leat
gracias
vinaka
спасибі
blagodaram
kiitos
dankie
nanni
nandri
dhanyavad
mauruuru
bedankt
hvala
köszönöm
bayarlalaa
gràçie
enkosi
dziękuję
dankon
ačiū
akun
хвала
asante
manana
obrigada
tenki
murakoze
djiere dieuf
tau
mochchakkeram
дякую
mamnun
chokrane
chnorakaloutioun
gratias ago
gràcies
sulpáy
go raibh maith agat
sobodi
děkuji
obrigado
sagolun
sukriya
kop khun krap
taiku
arigatô
takk
dakujem
trugarez
mèsi
didi madloba
najis tuke
ありがとう
tanemirt
grazie
kam sah hamnida
rahmat
terima kasih
rahmet
diolch
dhanyavadagalu
shukriya
mercé
мерси
তোমাকে ধন্যবাদ
감사합니다
xièxie
ευχαριστώ
merci