Greedy Algorithms

Heejin Park
Division of Computer Science and Engineering
Hanyang University

Contents

Introduction

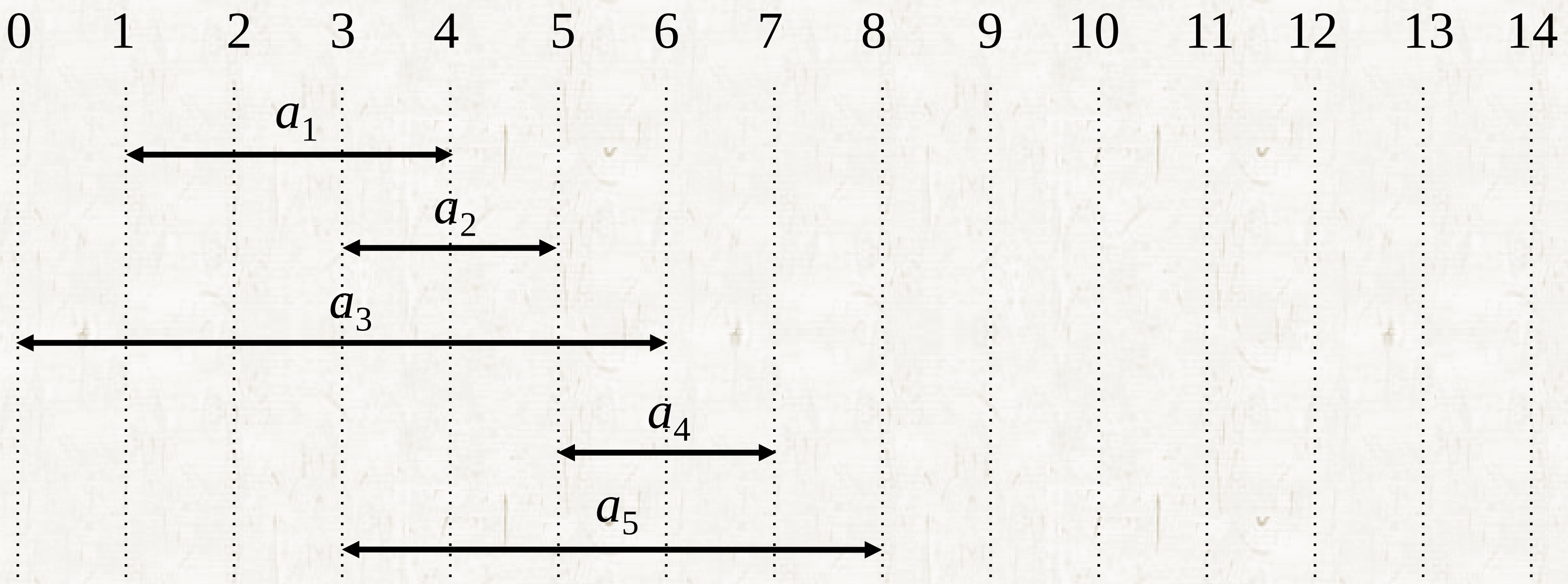
- A ***greedy algorithm*** always makes the choice that looks best at the moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- It makes the choice ***before*** the subproblems are solved.

An activity selection problem

- **An activity selection problem**
  - To select a maximum-size subset of mutually compatible activities.
  - For example
    - Given *n* classes and 1 lecture room,
    - to select the maximum number of classes

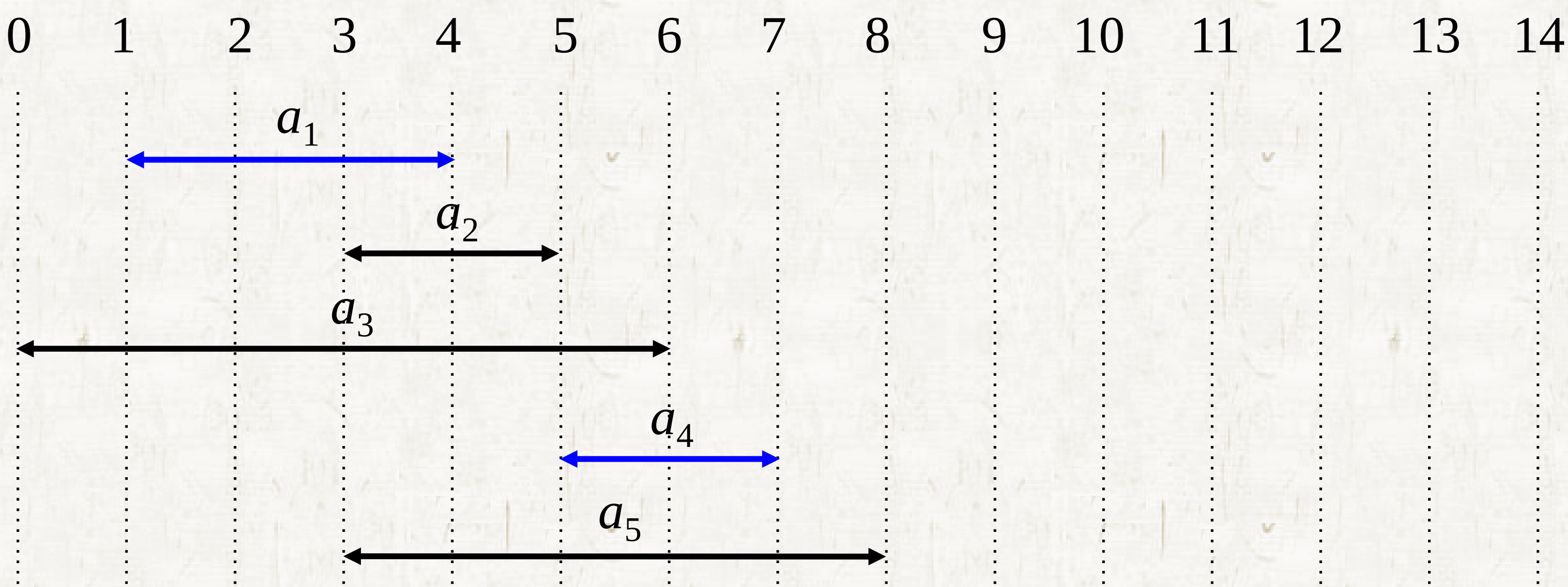
An activity selection problem

- A set of ***activities***: $S = \{a_1, a_2, \ldots, a_n\}$
- Each activity a_i has its ***start time*** $\boldsymbol{s_i}$ and ***finish time*** $\boldsymbol{f_i}$.
  - $0 \leq s_i < f_i < \infty$

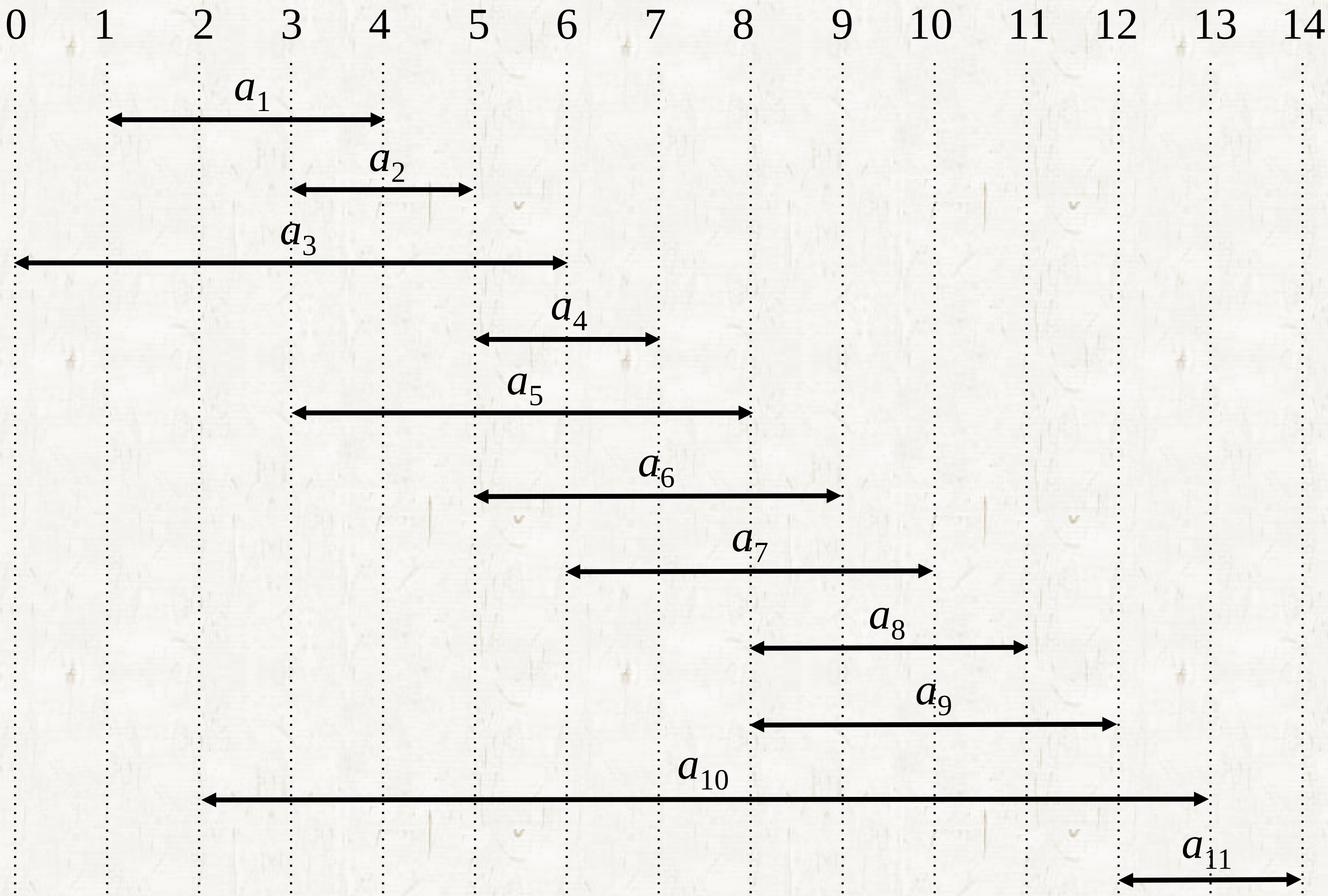


An activity selection problem

- Activity a_i takes place during $[s_i, f_i)$
- Activities a_i and a_j are ***compatible*** if the intervals $[s_i, f_i)$ and $[s_j\ f_j)$ do not overlap.

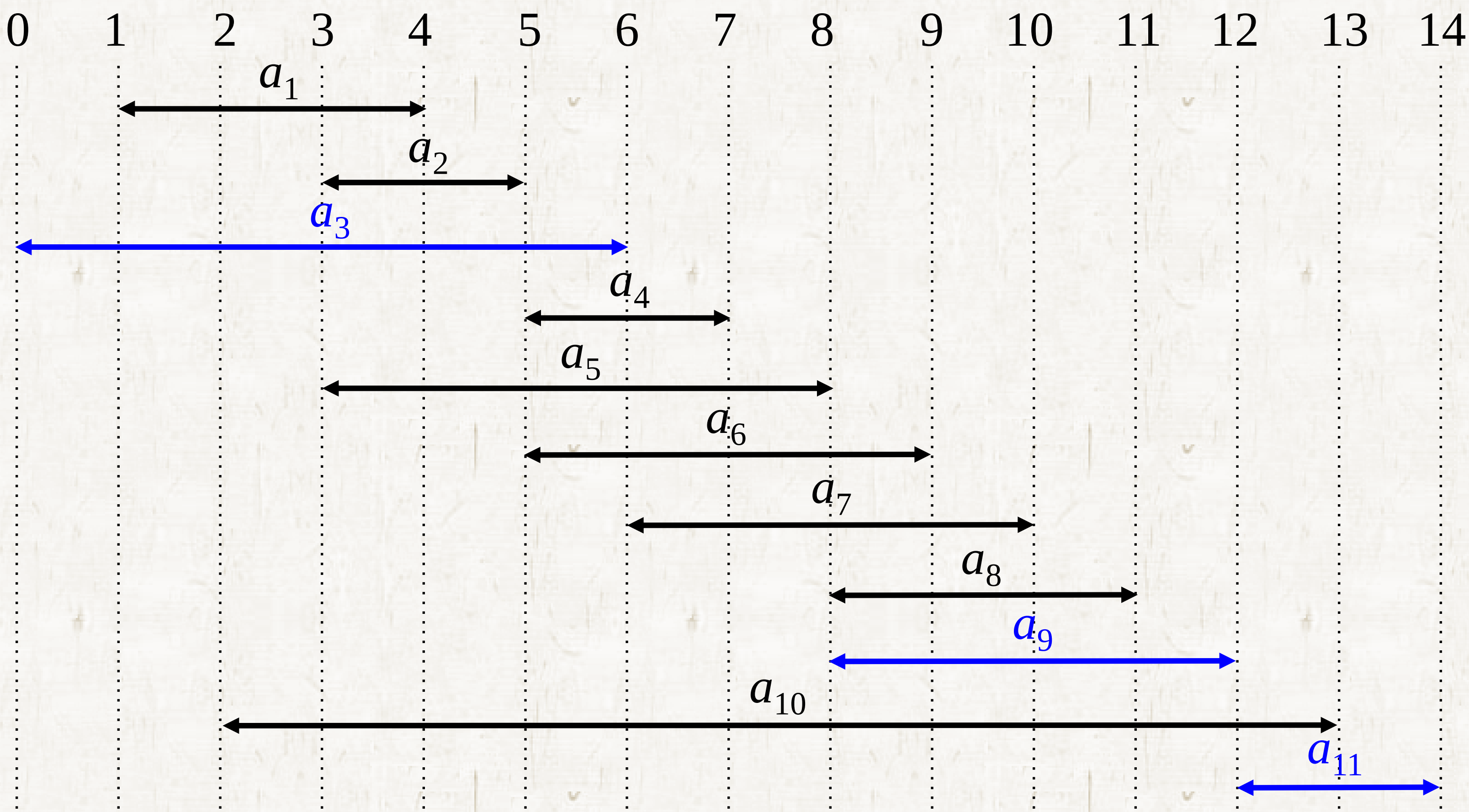


An activity selection problem



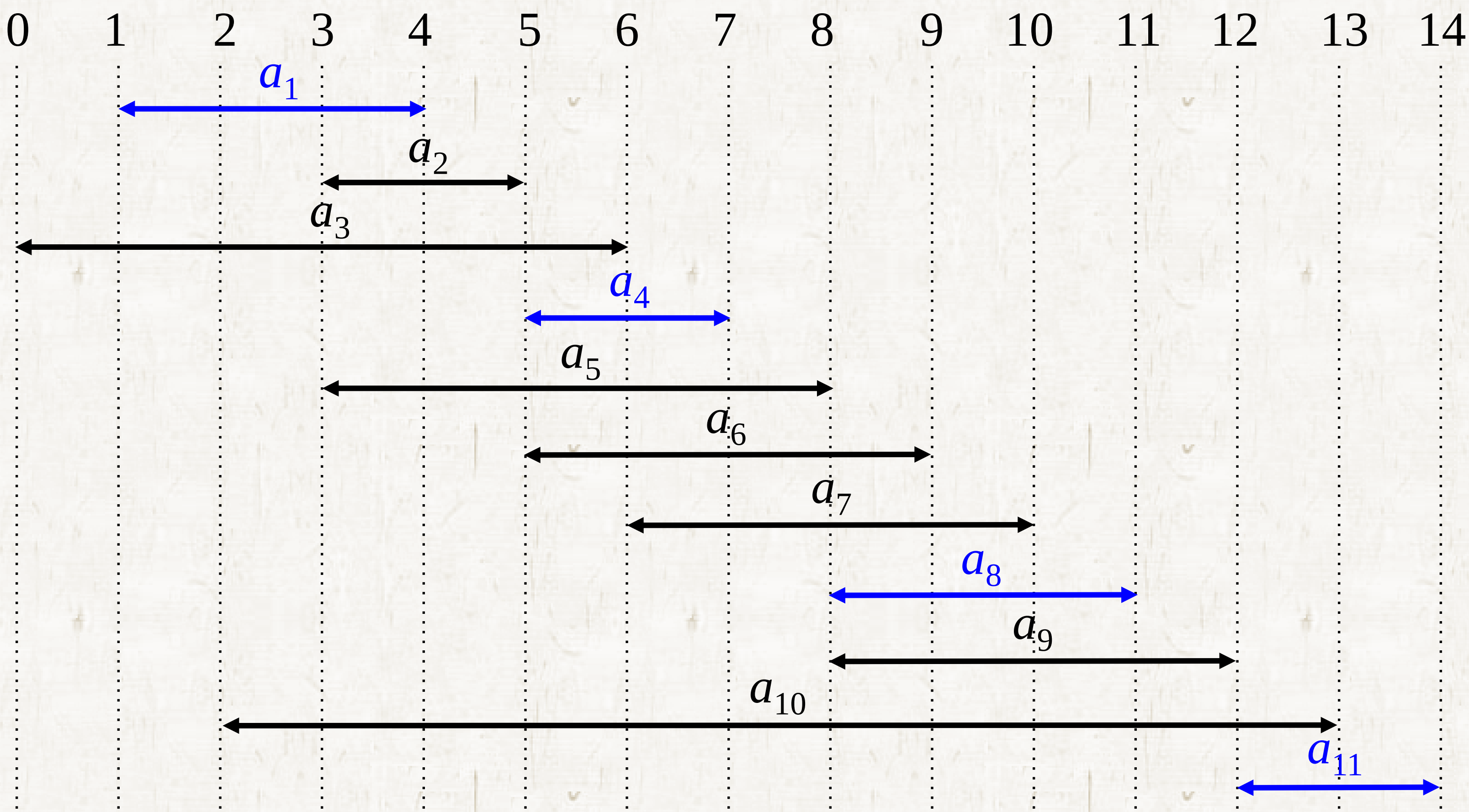
An activity selection problem

- $\{a_3, a_9, a_{11}\}$: mutually compatible activities, not a largest set



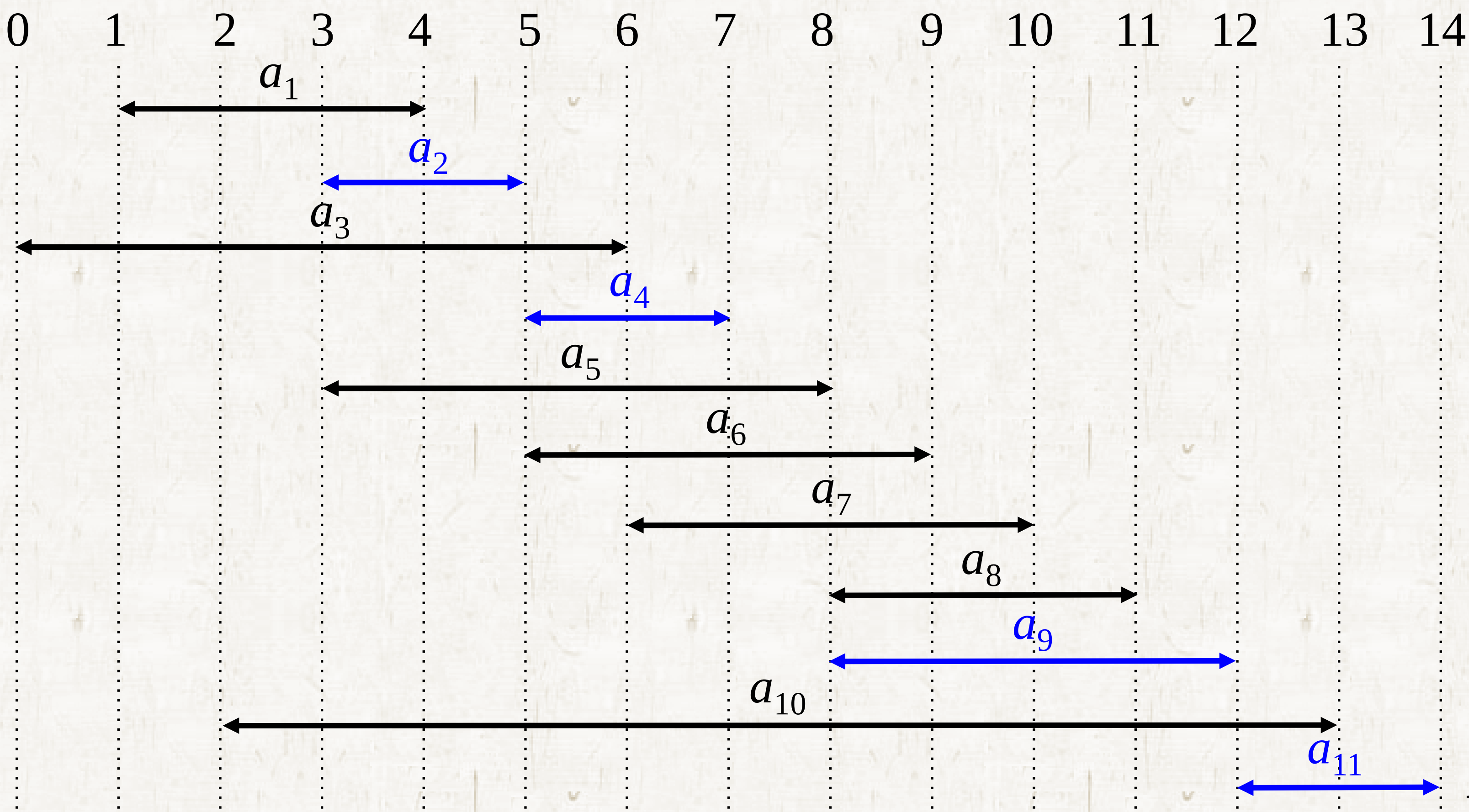
An activity selection problem

- $\{a_1, a_4, a_8, a_{11}\}$: A largest set of mutually compatible activities



An activity selection problem

- $\{a_2, a_4, a_9, a_{11}\}$: Another largest subset



An activity selection problem

- **Optimal substructure**
  - S_{ij} denote the set of activities between a_i and a_j and compatible with a_i and a_j.
    - Activities start after a_i finishes and finish before a_j starts.

$$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$$

    - For example, $S_{18} = \{a_4\}$

An activity selection problem

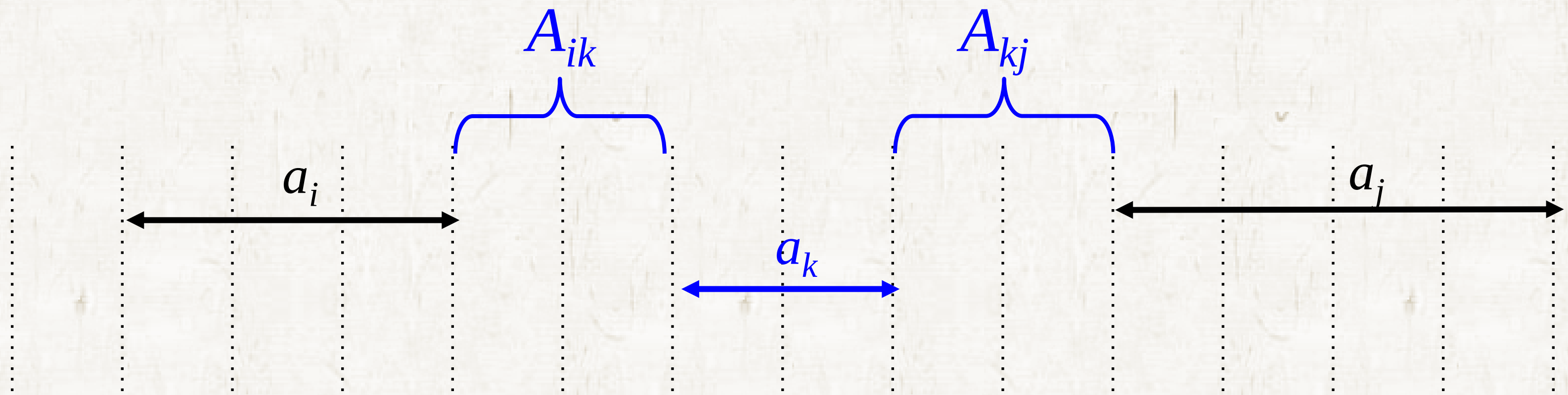
- **Optimal substructure**
  - Assume that activities are sorted in increasing order of finish time.

$$f_0 \le f_1 \le f_2 \le \cdots \le f_n < f_{n+1}$$

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	8	9	10	11	12	13	14

An activity selection problem

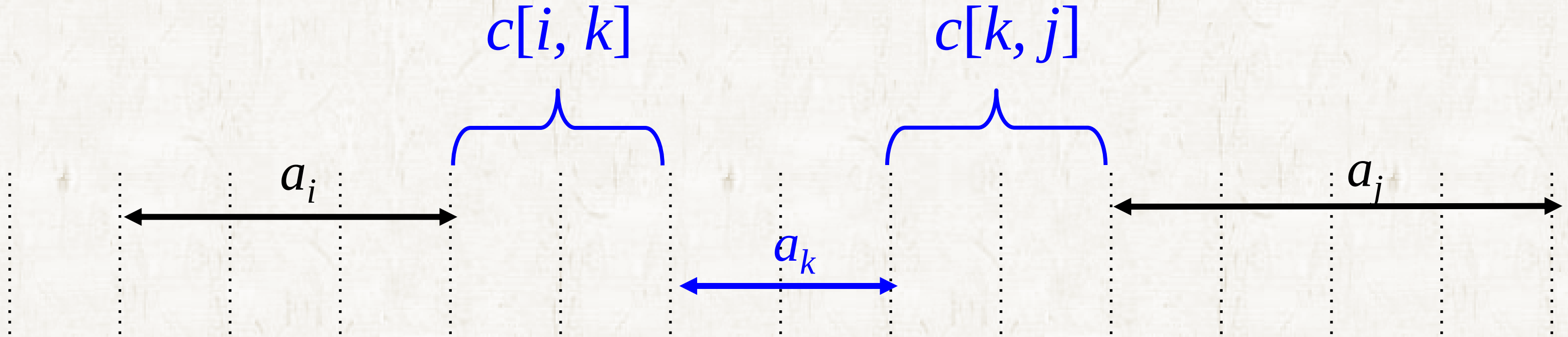
- **Optimal substructure**
  - A_{ij} denote an optimal solution to S_{ij} for $i \le j$.
  - If A_{ij} includes a_k, $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$



An activity selection problem

- **Optimal substructure**
  - $c[i, j]$: The number of activities in A_{ij}.

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



An activity selection problem

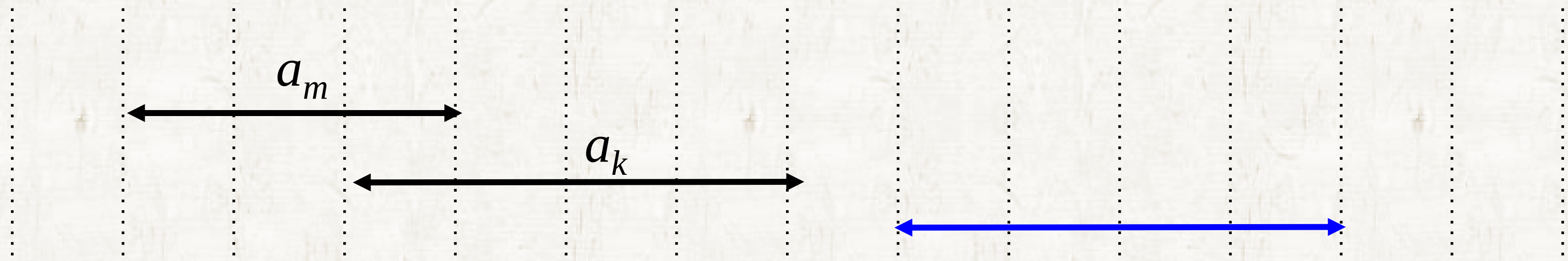
- **Greedy algorithm**
  - Consider any nonempty S_{ij}, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min \{f_k : a_k \in S_{ij}\}$.

1. Activity a_m is in some A_{ij}.
2. The subproblem S_{im} is empty, so the subproblem S_{mj} is the only one to consider.

An activity selection problem

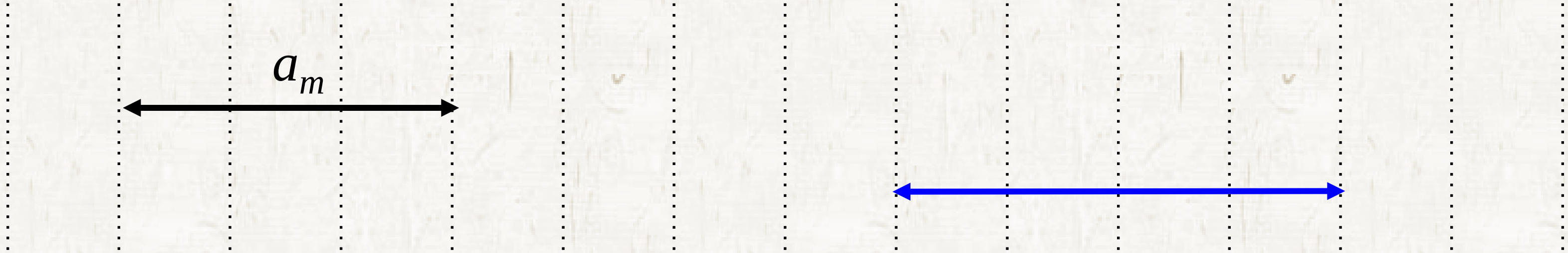
- Activity a_m is in some A_{ij}.
  - a_k: the first finishing activity in A_{ij}
  - If $a_k = a_m$, done.
  - If $a_k \neq a_m$, remove a_k from A_{ij} and add a_m to A_{ij} .

    The resulting A_{ij} is another optimal solution because $f_m \leq f_k$ and all other activities in A_{ij} start after a_k finishes.



An activity selection problem

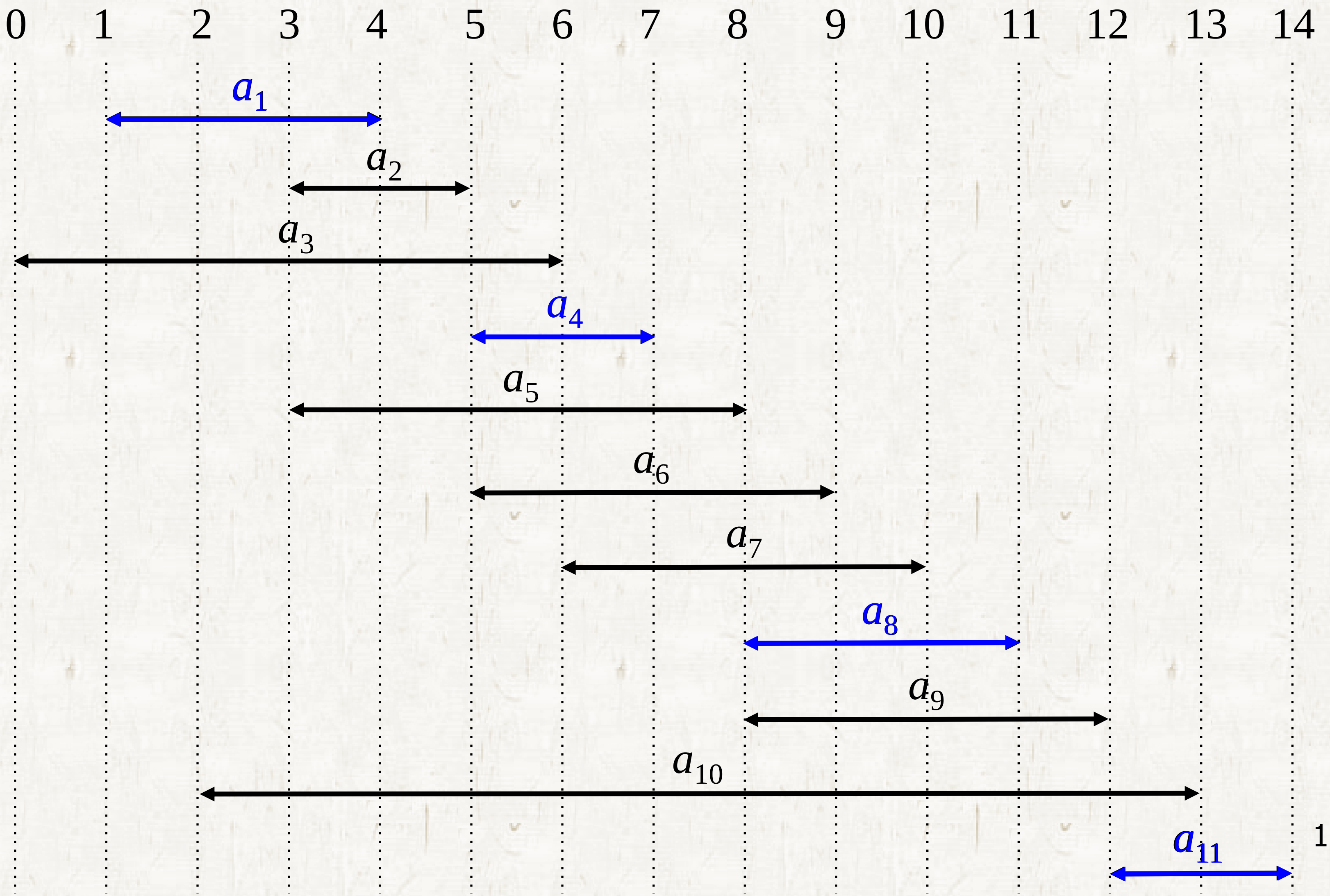
- The subproblem S_{im} is empty, so the subproblem S_{mj} is the only one to consider.
  - S_{im} is empty because a_m has the earlier finish time in S_{ij}.



An activity selection problem

- **Greedy algorithm**
  - Select the earliest finishing activity one by one.

An activity selection problem



Contents

- *Introduction*
- *An activity selection problem*
- **Elements of the greedy strategy**
- **Huffman codes**

Elements of the greedy strategy

- **Greedy-choice property**
  - Make the choice ***before*** the subproblems are solved.
  - Only one subproblem is generated.

- **Dynamic programming**
  - Make a choice ***after*** the subproblems are solved.
  - Several subproblems may be generated.

Elements of the greedy strategy

- **Greedy vs. Dynamic programming**
  - **0-1 knapsack**
    - A thief robbing a store finds *n* items.
    - The *i*th item is worth v_i dollars and weighs w_i pounds.
    - He can carry at most W pounds in his knapsack.
    - The n, $v_{i,}$ $w_{i,}$ and W are integers.
    - Which items should he take?

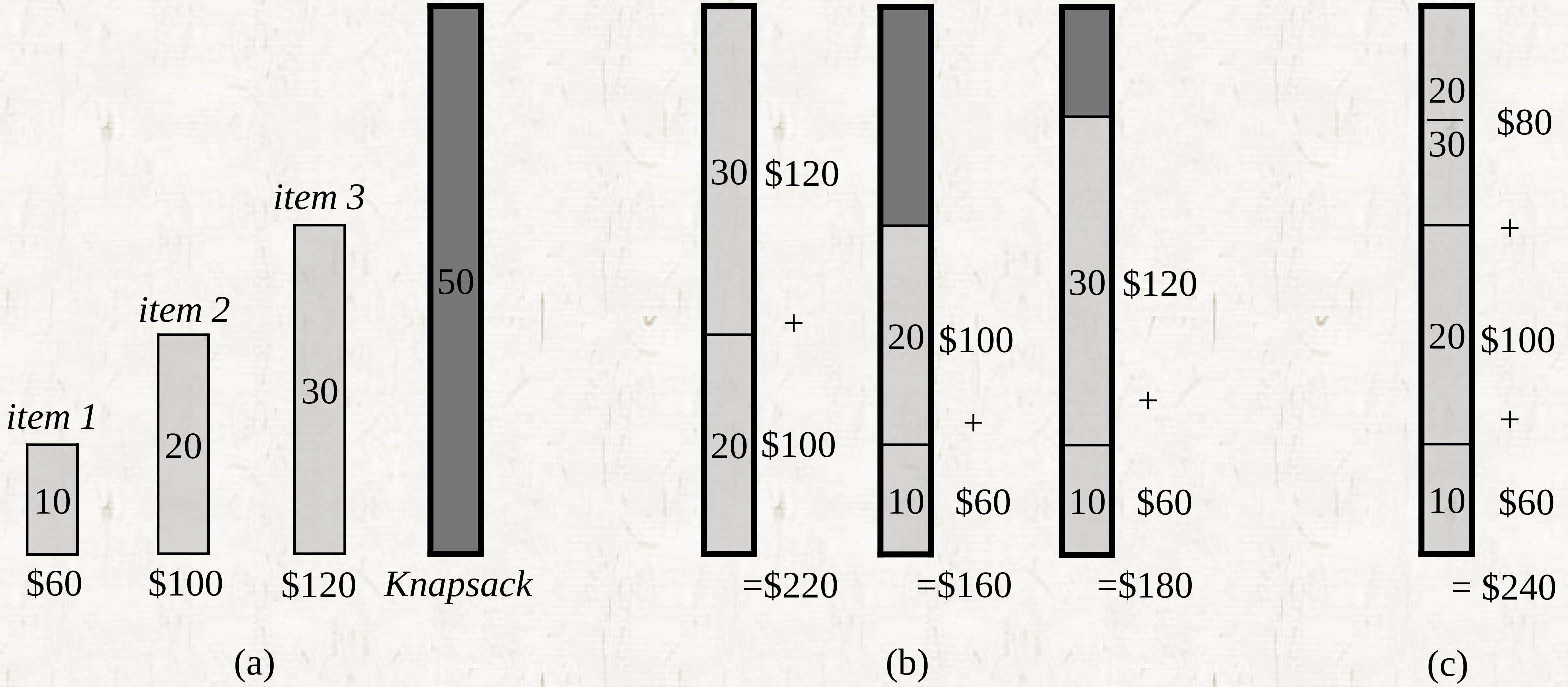
  - **Fractional knapsack**
    - In this case, the thief can take fractions of items.

Elements of the greedy strategy

- **Fractional knapsack**
  - The greedy strategy works.
  - Compute the value per pound v_i/w_i for each item.
  - Take as much as possible of the item with the greatest value per pound.

Elements of the greedy strategy

- **0-1 knapsack**
  - The greedy strategy does not work.



Self-study

- **Exercise 16.2-1**
- **Exercise 16.2-2**
- **Exercise 16.2-5**
- **Exercise 16.2-7**

Contents

- *Introduction*
- *An activity selection problem*
- *Elements of the greedy strategy*
- **Huffman codes**

Huffman codes

- **Huffman Codes**
  - A widely used technique for compressing data.

- Consider representing 100,000 characters from {a, b, c, d, e, f}.
  - 3-bit ***fixed-length code*** is used in general.
  - It takes 300,000 bits in total

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

Huffman codes

- We can reduce the space if ***variable-length code*** is used.

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

- Shorter **codewords** for frequent characters.
- 224,000 bits in total
  - $(45 \cdot 1+13 \cdot 3+12 \cdot 3+16 \cdot 3+9 \cdot 4+5 \cdot 4) \cdot 1000$ bits

Huffman codes

- **Encoding and decoding of variable-length code**
  - Encoding abc : 0·101·100
  - Decoding 001011101
    - 0·0·101·1101: aabe

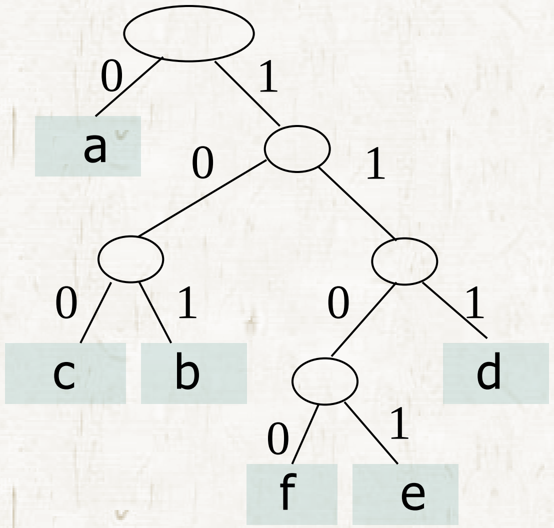
	a	b	c	d	e	f
Variable-length codeword	0	101	100	111	1101	1100

  - Decoding 001 when a: 0 b: 01 c: 1
    - 001: aac or ab
    - The codeword 0 for a is a prefix of the codeword 01 for b.

Huffman codes

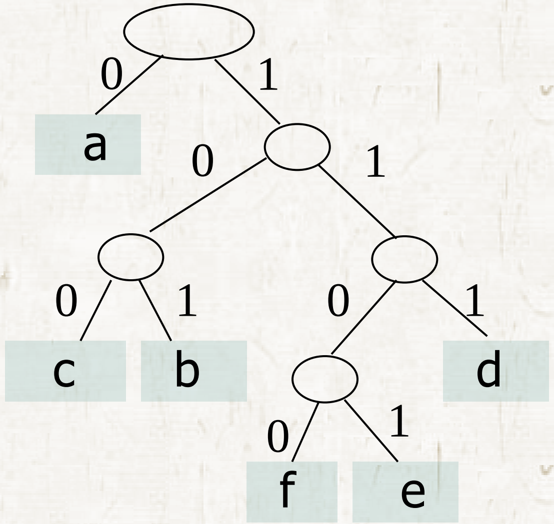
- **Prefix codes**
  - No codeword is a prefix of some other codeword.

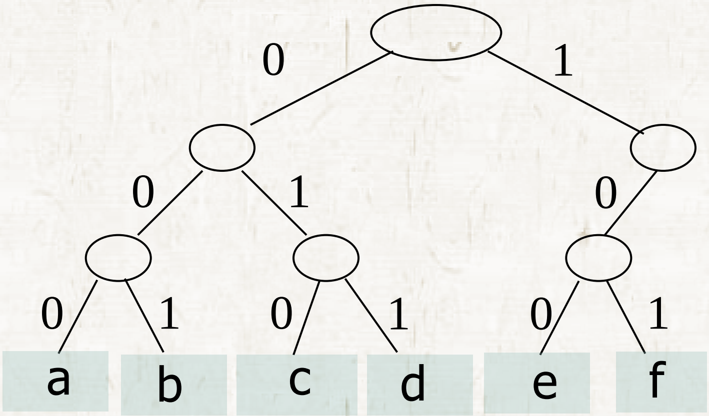
	a	b	c	d	e	f
Variable-length codeword	0	101	100	111	1101	1100



Huffman codes

- **Prefix codes**
  - 3-bit fixed-length code is also a prefix code.





  - The left tree is a ***full binary tree*** *while the right one is not.*
    - Every node is either leaf or has two children
    - A full binary tree for alphabet C has $|C|$ leaves and $|C|$-1 internal nodes.

Huffman codes

- **The cost of tree *T***
  - $f(c)$: frequency of a character c
  - $d_T(c)$: length of the codework for c

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

  - An optimal code is represented by a full binary tree.

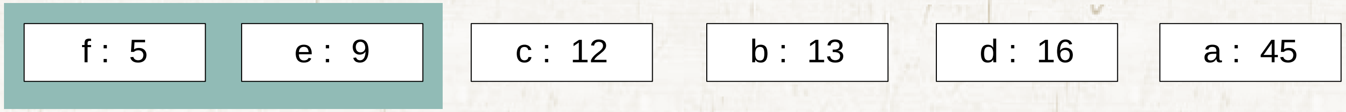
Huffman codes

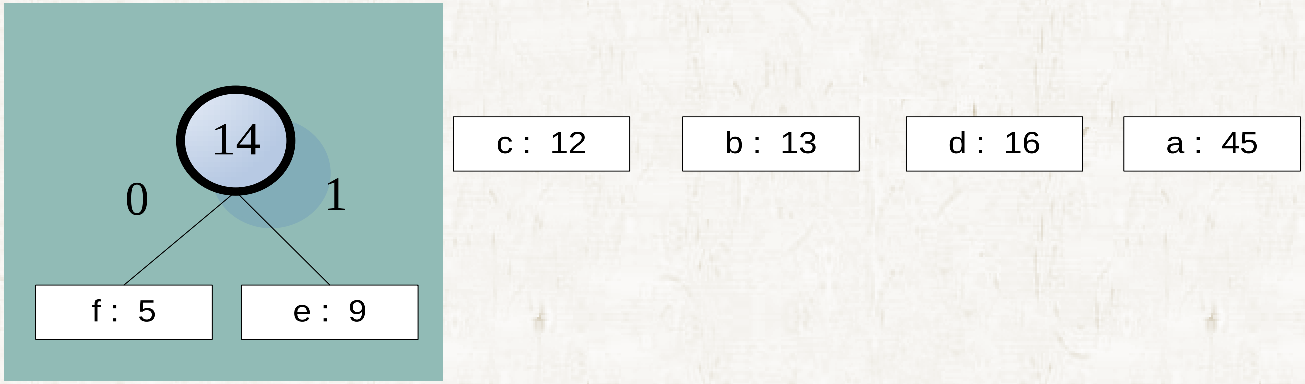
- Huffman invented a greedy algorithm that constructs an optimal prefix code called an ***Huffman code***.

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

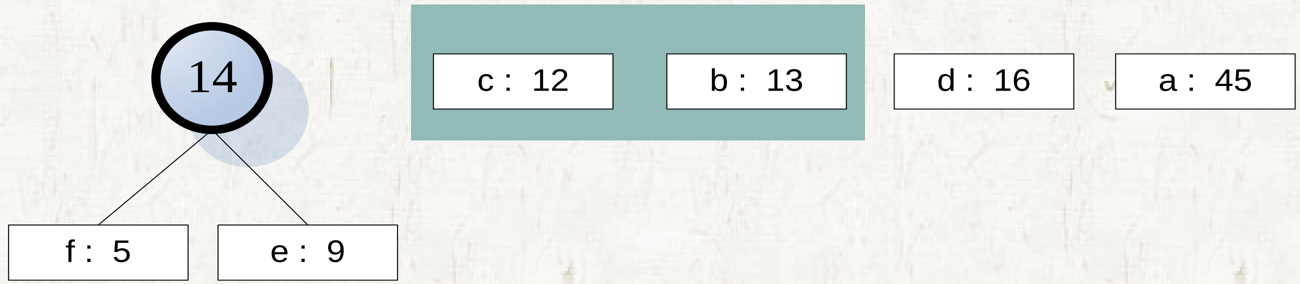
f : 5   e : 9   c : 12   b : 13   d : 16   a : 45

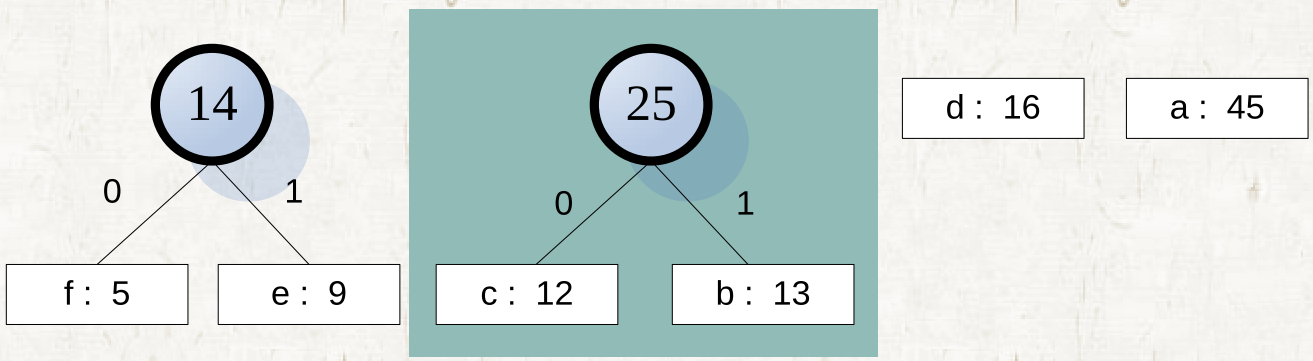
Huffman codes



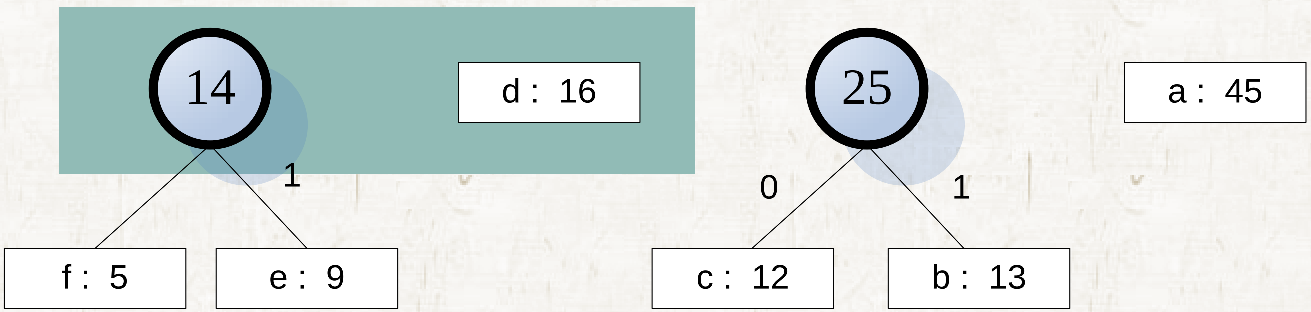


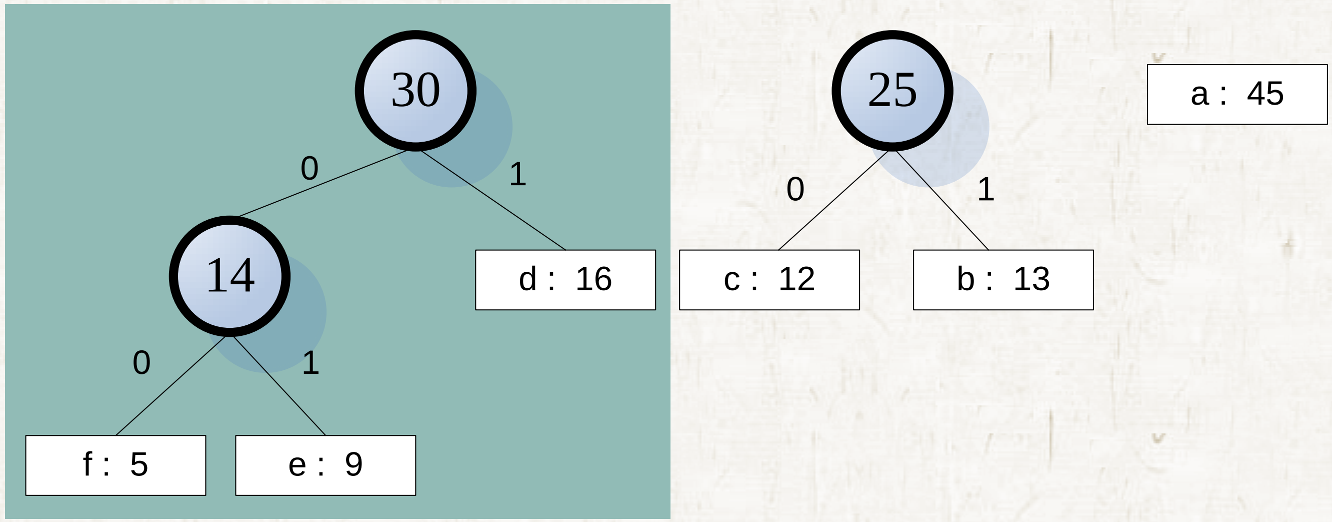
Huffman codes



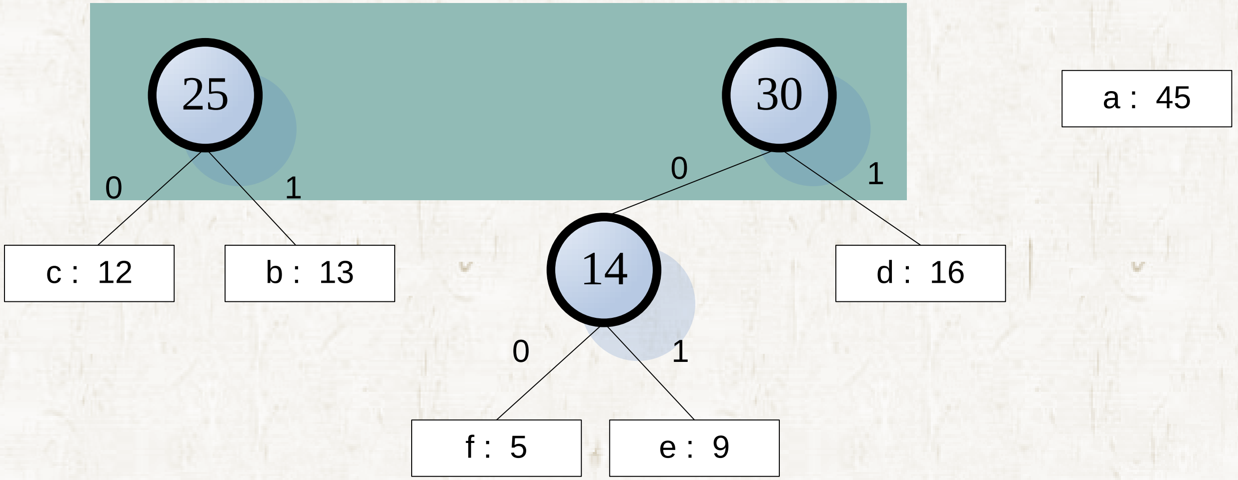


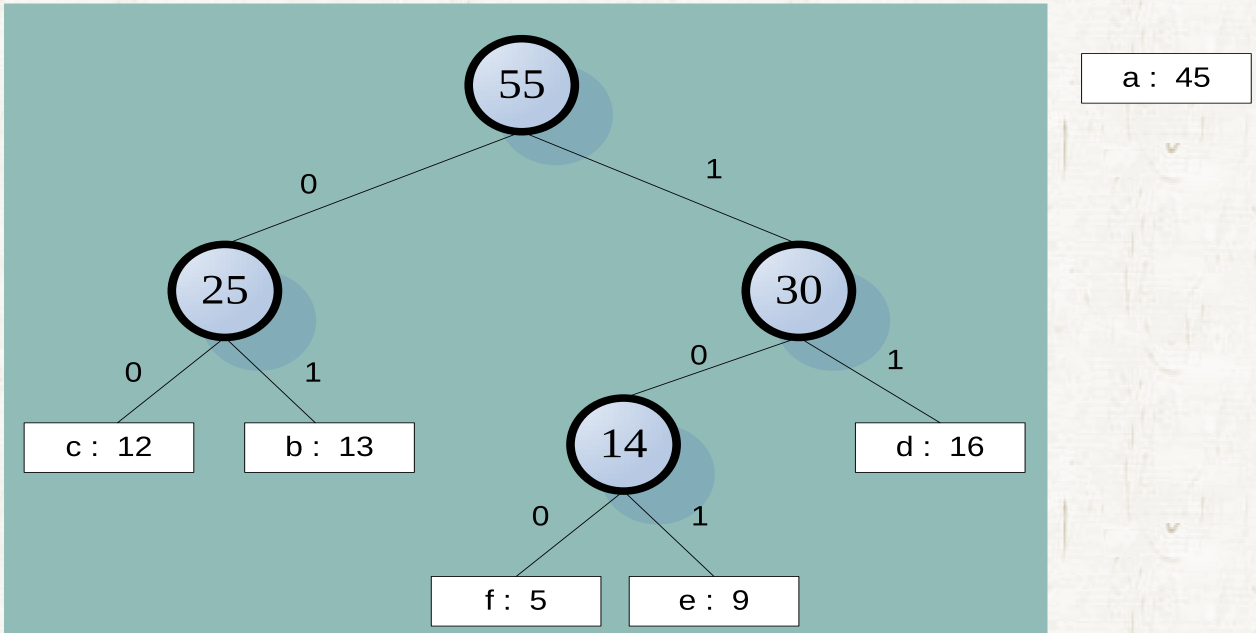
Huffman codes



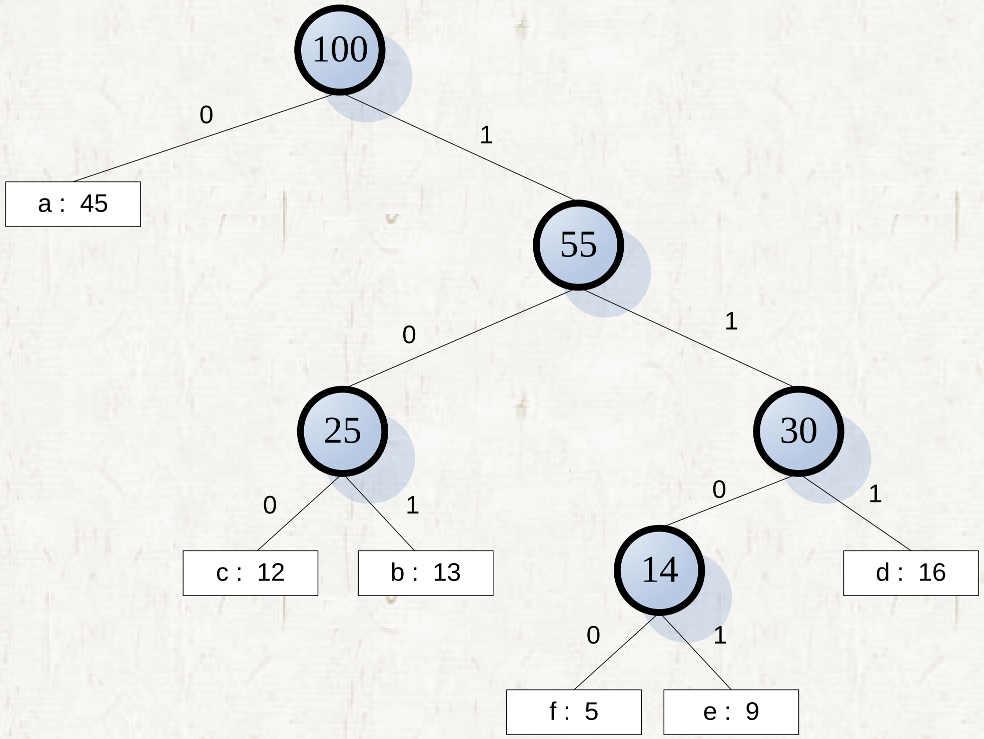


Huffman codes





Huffman codes



Huffman codes

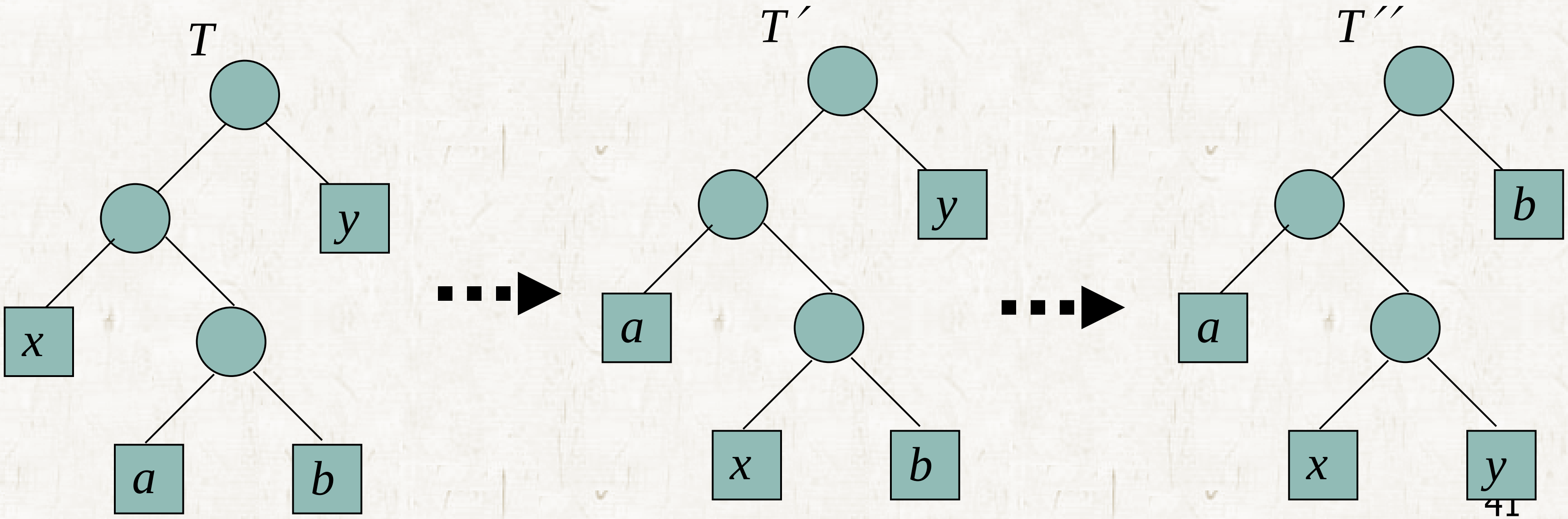
- **Running time**: O($n$lg$n$)
  - Build min heap: $O(n)$
  - Merge: n-1 times
    - Each merge requires two minimum selection: $O(\lg n)$

Huffman codes

- **Correctness**
  - ***Lemma 16.2***
    - Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$.
    - Let x and y be two characters in C having the lowest frequencies.
    - Then there exists an optimal prefix code for C in which the ***codewords for x and y have the same length and differ only in the last bit***.

Huffman codes

- ***Proof***
  - **Idea**: take an arbitrary optimal prefix code tree *T* and modify it and to make a tree representing another optimal prefix code such that the characters *x* and *y* appear as sibling leaves of maximum depth in the new tree.



Huffman codes

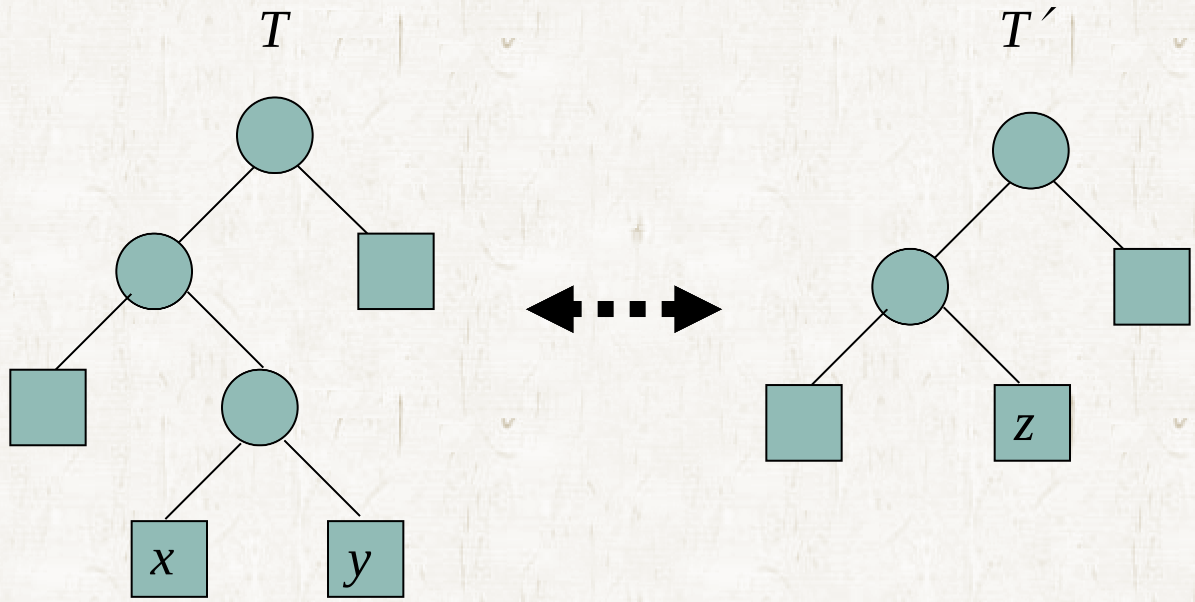
- **The cost of tree *T***
  - $f(c)$: frequency of a character c
  - $d_T(c)$: length of the codeword for c

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

Huffman codes

- ***Lemma 16.3***
  - Let x and y be two characters in a given alphabet C with minimum frequency.
  - Let C' be the alphabet C with characters x, y removed and character z added, so that $C' = C - \{x, y\} \cup \{z\}$; define f for C' as for C, except that $f[z] = f[x] + f[y]$.
  - Let T' be any tree representing an optimal prefix code for the alphabet C'.
  - Then the optimal prefix code tree T for C can be obtained from T' by replacing the leaf node for z with an internal node having x and y as children.

Huffman codes



Huffman codes

- ***Proof***
  - Show $B(T) = B(T') + f[x] + f[y]$
    - For each $c \in C$ - $\{x, y\}$, we have $d_T(c) = d_{T'}(c)$, and hence $f[c]d_T(c) = f[c]d_{T'}(c)$.
    - Since $d_T(x) = d_T(y) = d'(z) + 1$, we have

$$f[x]d_T(x) + f[y]d_T(y) = (f[x] + f[y])(d_{T'}(z) + 1)$$
$$= f[z]d_{T'}(z) + (f[x] + f[y])$$

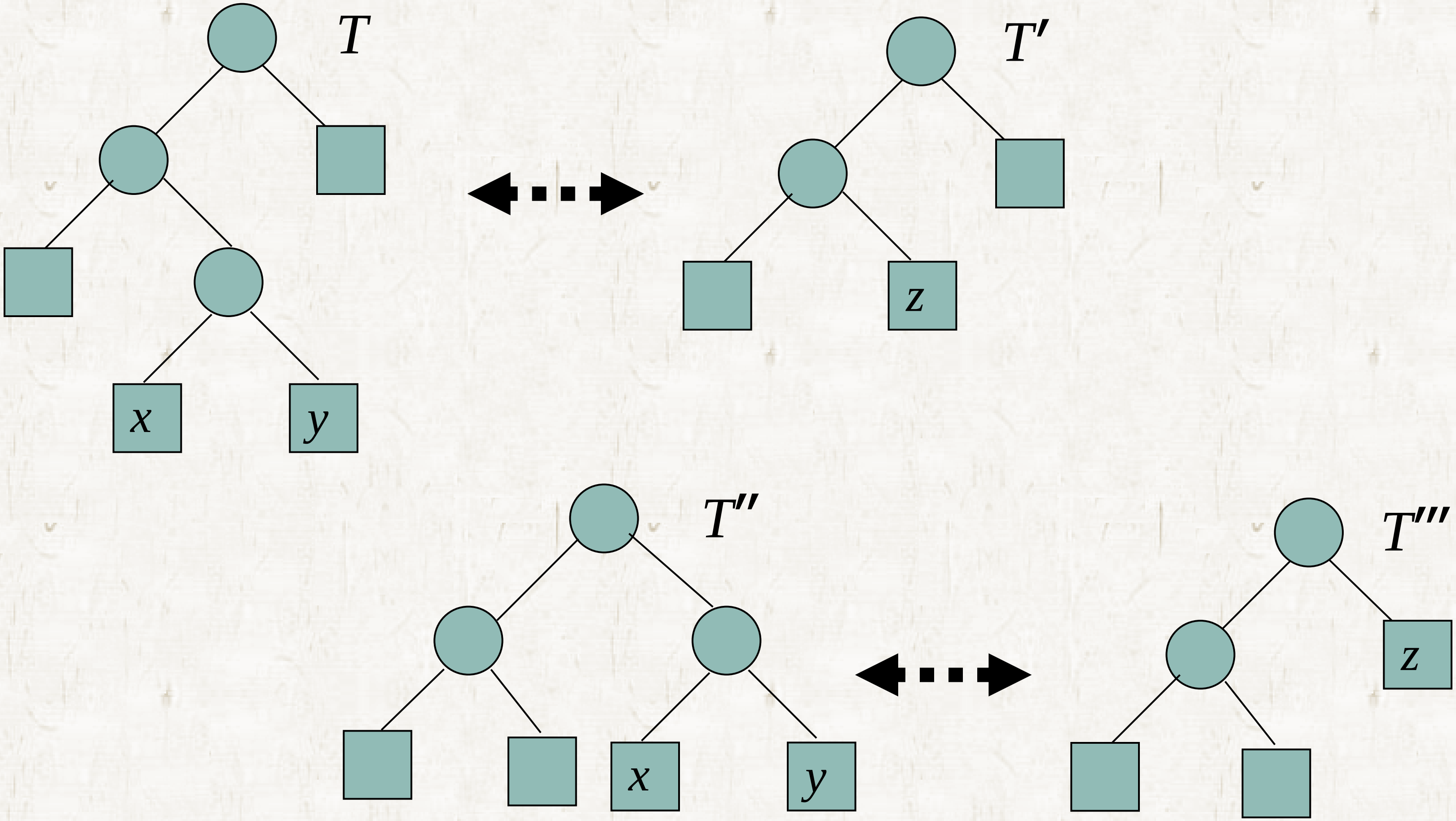
    - From which we conclude that $B(T) = B(T') + f[x] + f[y]$ or, equivalently $B(T') = B(T) - f[x] - f[y]$.

Huffman codes

- ***Proof***
  - Suppose T does not represent an optimal prefix code for C.
  - There exists T'' such that $B(T'') < B(T)$.
  - By Lemma 16.2, there exists T'' having x and y as siblings.
  - Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $f[z] = f[x] + f[y]$.
  - Then, $B(T''') = B(T'') - f[x] - f[y]$
    $< B(T) - f[x] - f[y]$
    $= B(T')$

    ➔Contradiction
  - T must represent an optimal prefix code for the alphabet C.

Huffman codes



Self-study

- **Exercise 16.3-3 (16.3-2 in the 2nd ed.)**
  - Fibonacci number definition is in p. 59 (p. 56 in the 2nd ed.)

- **Exercise 16.3-7 (16.3-6 in the 2nd ed.)**