Data Structures and Algorithm

Xiaoqing Zheng
zhengxq@fudan.edu.cn

Activity-selection problem

Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed ***activities*** that wish to use a resource which can be used by only one activity at a time.

Consider the following set S of activities

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

Activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not ***overlap***.

Activity-selection problem

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

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

It is not a ***maximal*** *subset of mutually compatible activities!*

Activity-selection problem

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

Subset $\{a_1, a_4, a_8, a_{11}\}$

*It is a **largest** subset of mutually compatible activities.*

Activity-selection problem

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

Subset $\{a_2, a_4, a_9, a_{11}\}$

*It is a **largest** subset of mutually compatible activities too.*

Brute-force

Activity-selection problem is to select a maximum-size subset of mutually compatible activities.

Analysis

- Checking = $O(n)$ time per subset of S.
- 2^n subset of S.
- Worst-case running time $= O(n2^n)$

  $=$ exponential time.

It is infeasible!

Structure of Activity-selection problem

$S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$ denote the subset of activities in S that can start after activity a_i finishes and finish before activity a_j start.

Suppose now that an optimal solution A_{ij} to S_{ij} includes activity a_k. Then the solutions A_{ik} to S_{ik} and A_{kj} to S_{kj} used within this optimal solution to S_{ij} must be optimal as well.

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

Recursive solution

Let $c[i, j]$ be the number of activities in maximum-size subset of mutually compatible activities in S_{ij}.

Recursive definition of $c[i, j]$ becomes

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

We add fictitious activities a_0 and a_{n+1} and adopt the conventions that $f_0 = 0$ and $s_{n+1} = \infty$, then ***our goal*** is: $c[0, n + 1]$.

Greedy solution

Theorem.

Consider any nonempty subproblem S_{ij}, and let a_m be the activity in S_{ij} with earliest finish time:

$f_m = \min\{f_k : a_k \in S\}$.

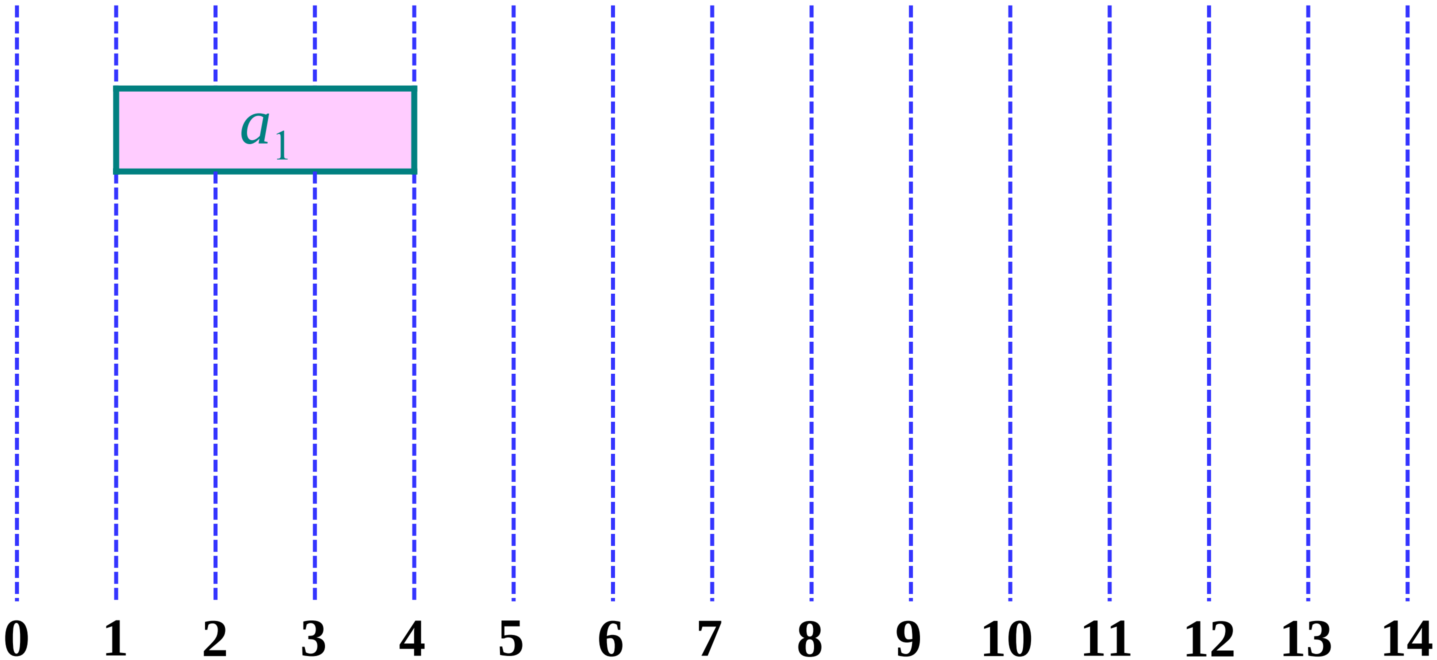
Then

- Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
- The subproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as the only one that may be nonempty.

Computing activity-selection problem

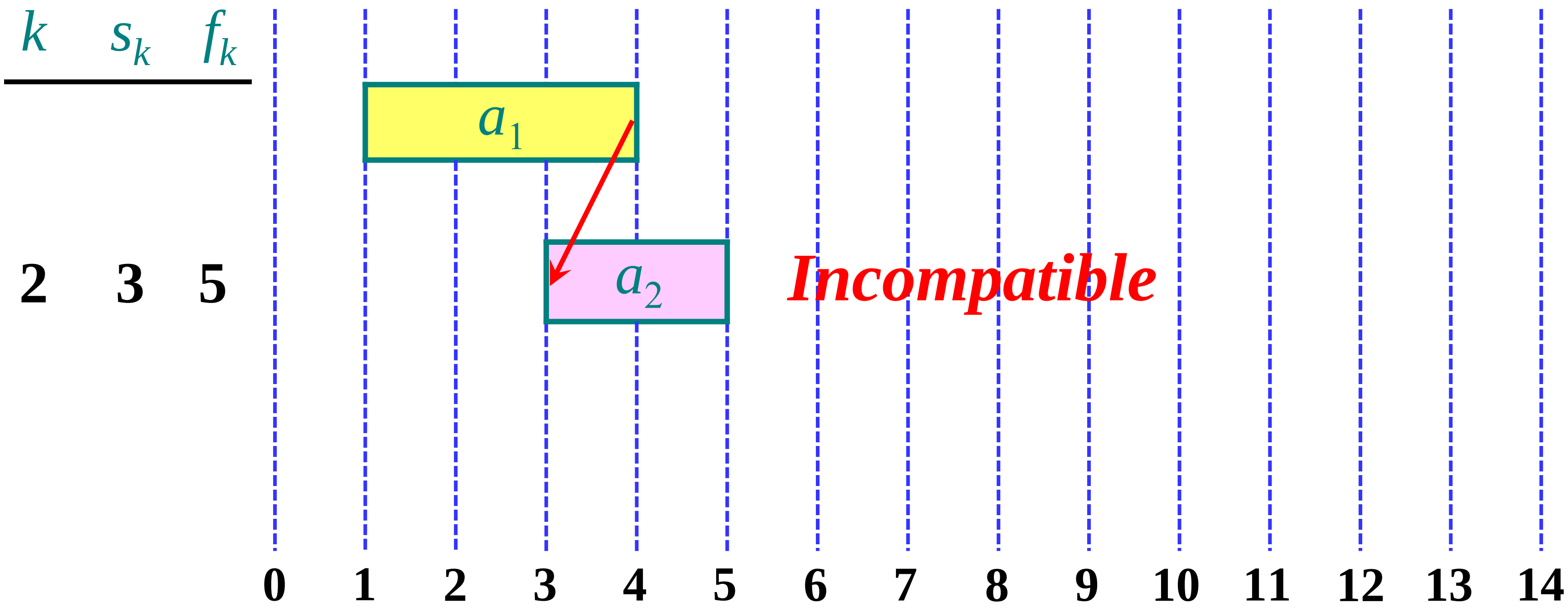
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

k	s_k	f_k
1	1	4



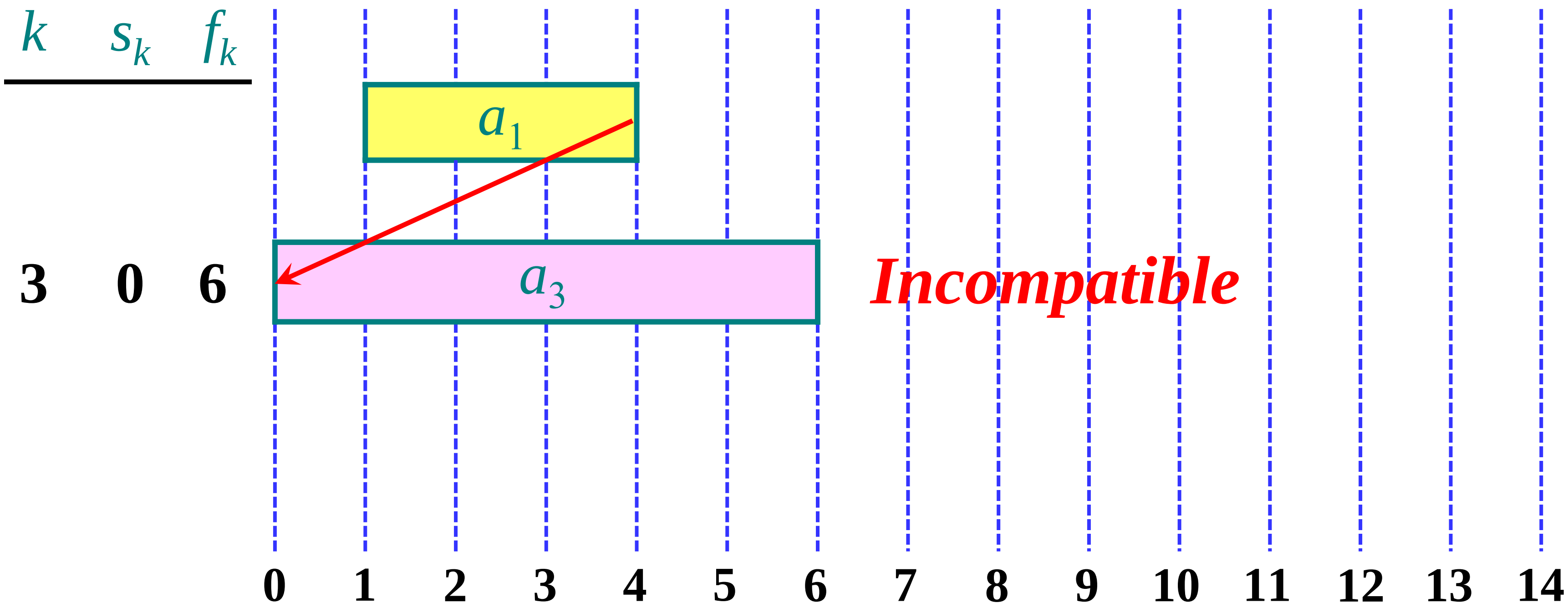
Computing activity-selection problem

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



Computing activity-selection problem

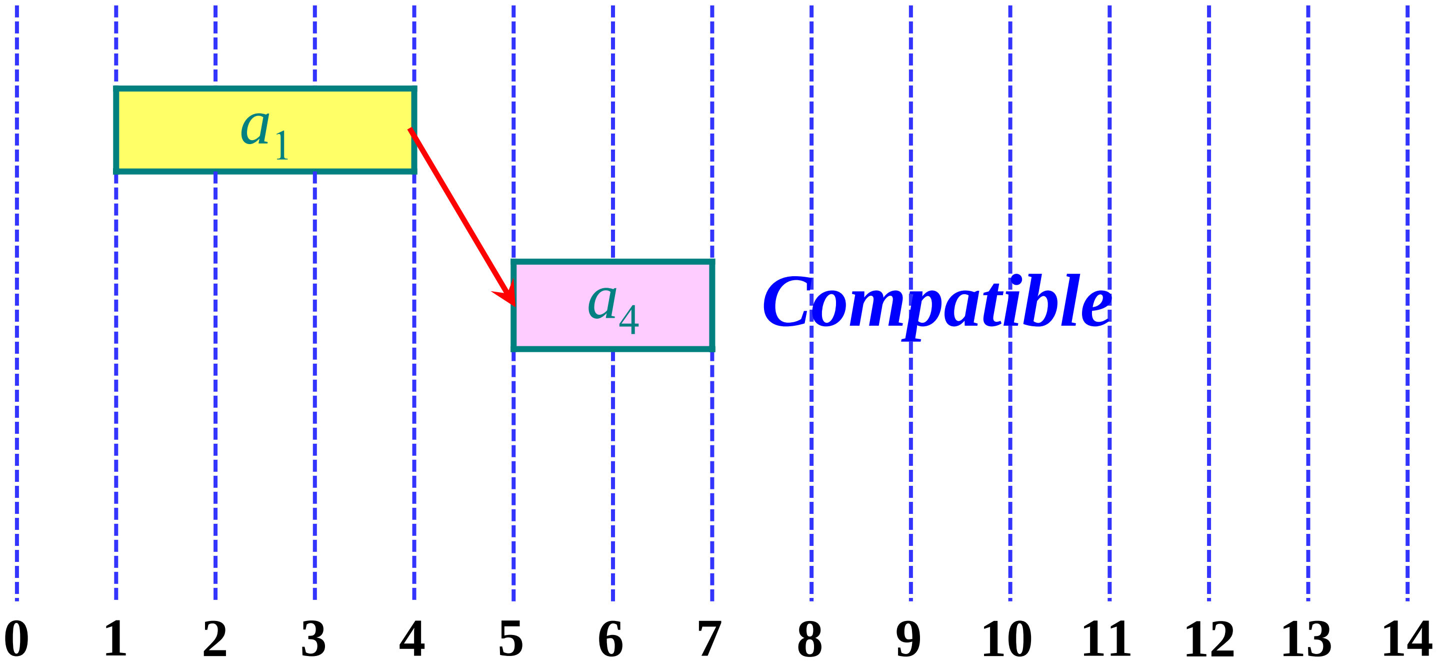
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



Computing activity-selection problem

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

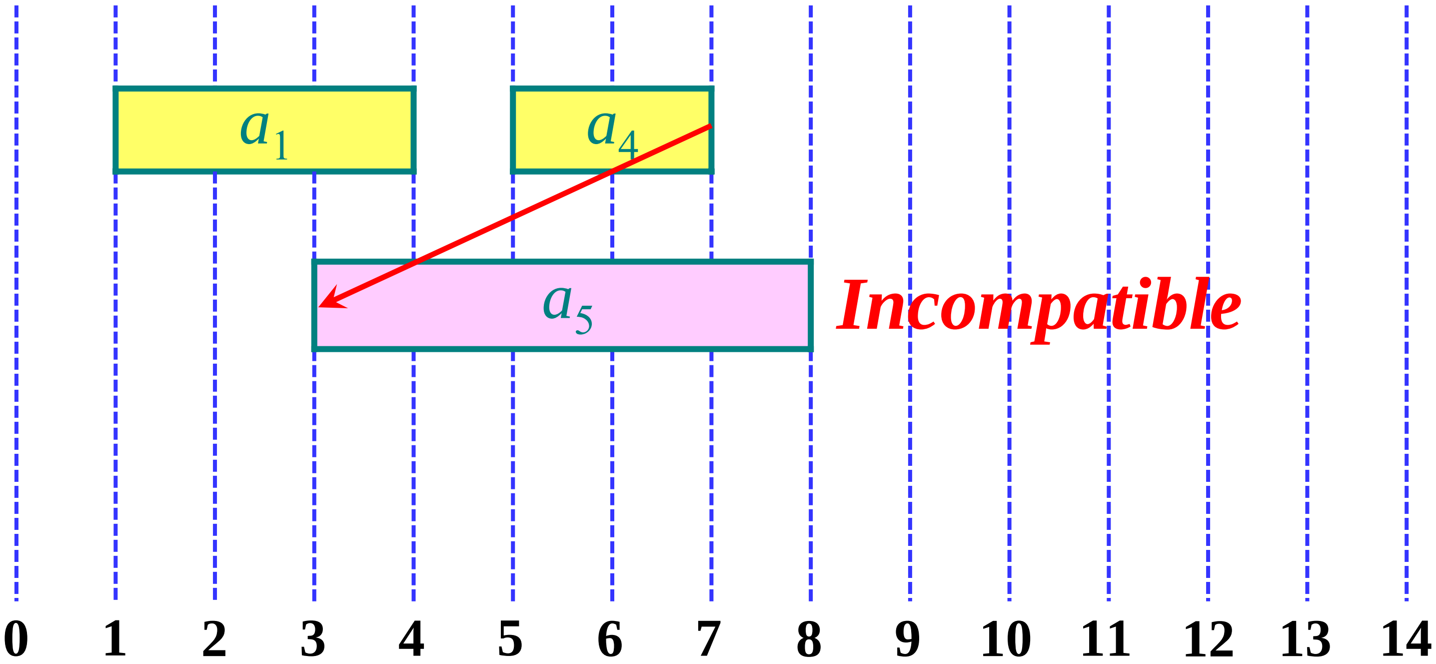
k	s_k	f_k
4	5	7



Computing activity-selection problem

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

k	s_k	f_k
5	**3**	**8**

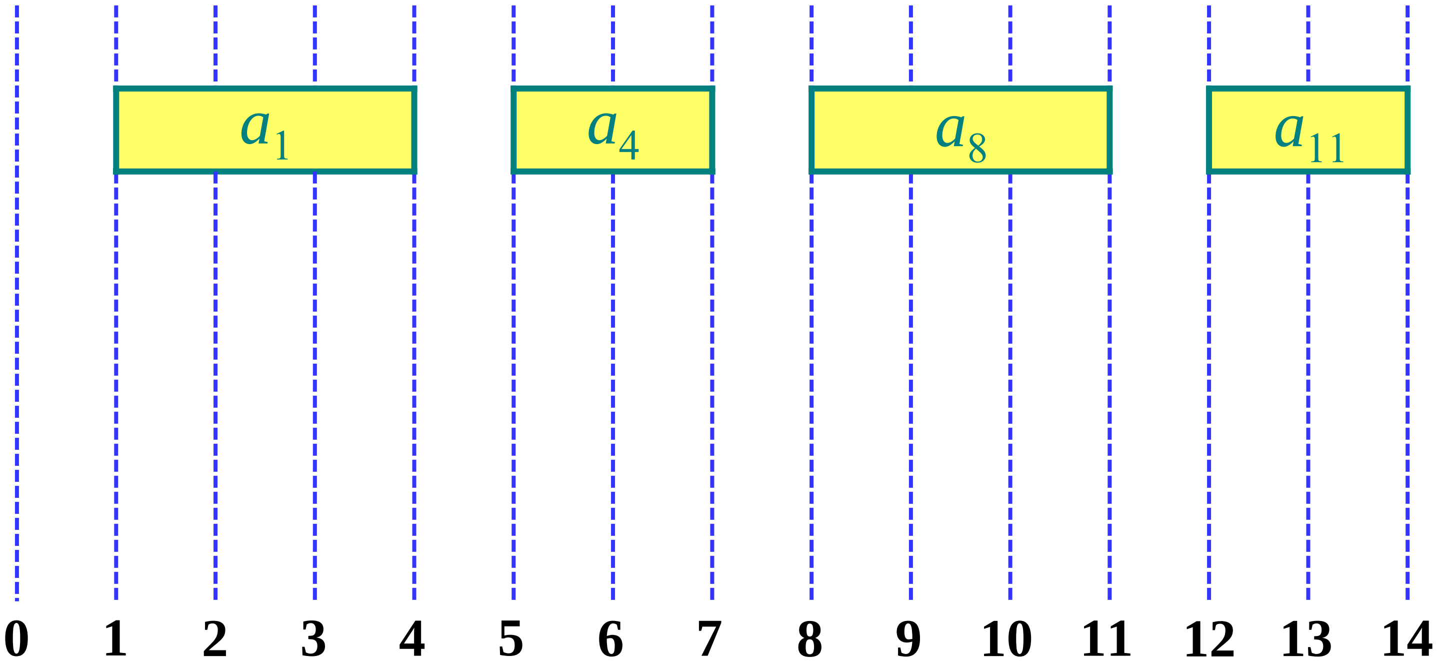


Computing activity-selection problem

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

k s_k f_k

$c[0,12] = \{a_1, a_4, a_8, a_{11}\}$



Matrix-chain multiplication

$m[i,j]$ denote the minimum number of scalar multiplications needed to compute the matrix $A_i \ldots A_j$.

We obtain the ***recursive*** equations

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \le k < j} \{m[i,k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

Our goal is $m[1, n]$.

Activity-selection problem

Let $c[i, j]$ be the number of activities in maximum-size subset of mutually compatible activities in S_{ij}.

Recursive definition of $c[i, j]$ becomes

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

We add fictitious activities a_0 and a_{n+1} and adopt the conventions that $f_0 = 0$ and $s_{n+1} = \infty$, then ***our goal*** is: $c[0, n + 1]$.

Elements of the greedy strategy

Optimal substructure

- An optimal solution to the problem contains within it optimal solutions to subproblems.

Greedy-choice property

- A globally optimal solution can be arrived at by making a locally optimal choice (a greedy choice at each step yields a globally optimal solution).

Steps of the greedy strategy

- Determine the ***optimal substructure*** of the problem.
- Develop a ***recursive*** solution.
- Prove that at any stage of the recursion, one of the optimal choices is the ***greedy choice***. Thus, it is always safe to make the greedy choice.
- Show that all but one of the subproblems induced by having made the greedy choice are ***empty***.
- Develop a ***iterative*** algorithm that implements the greedy strategy.

Knapsack problem

A thief robbing a store finds n items; the ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W.

Which items should he take?

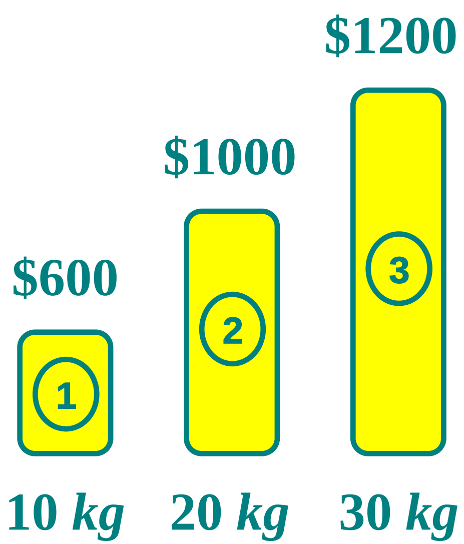
Knapsack problem

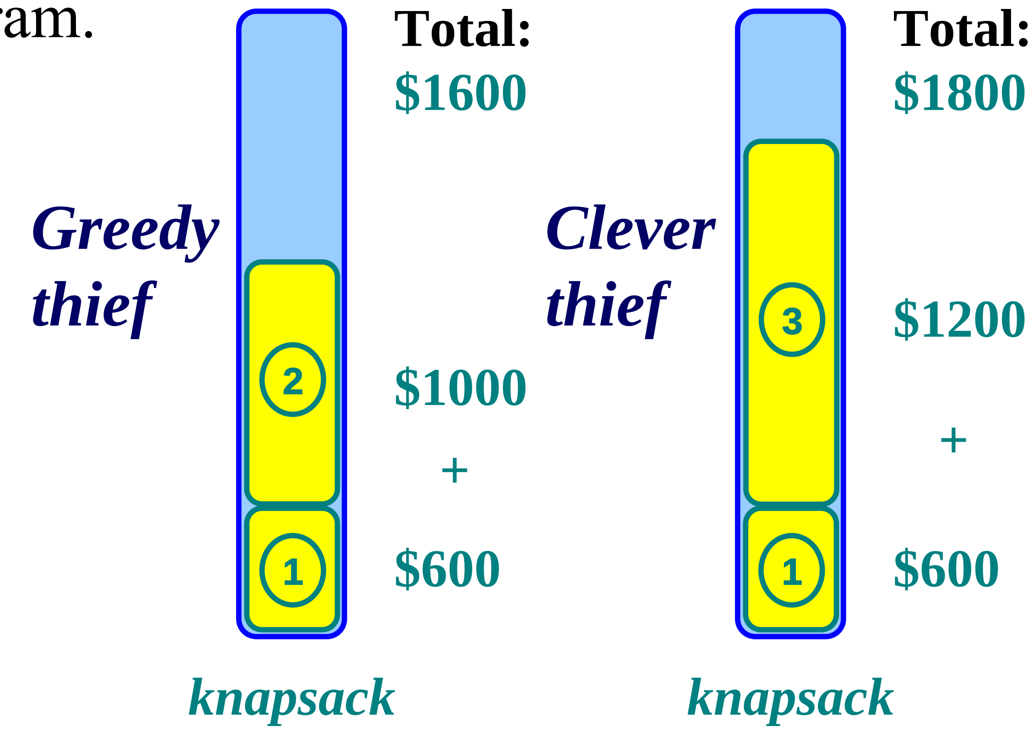
Item 1: **$60** per kilogram.
Item 2: **$50** per kilogram.
Item 3: **$40** per kilogram.

Greedy strategy

- take item 1.
- take item 2.

Thief can hold **50** kilogram.





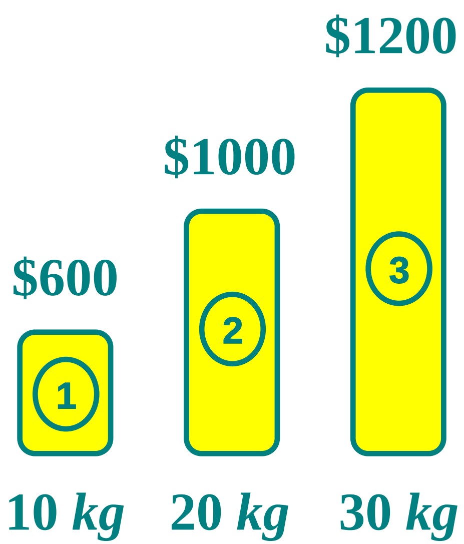
Knapsack problem

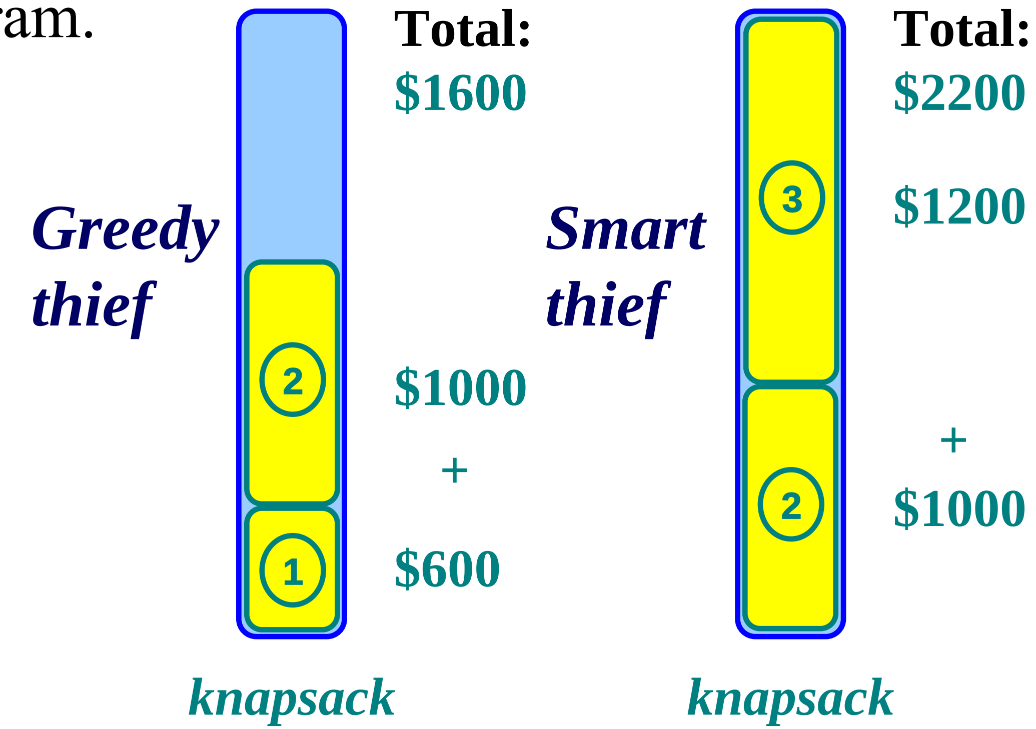
Item 1: **$60** per kilogram.
Item 2: **$50** per kilogram.
Item 3: **$40** per kilogram.

Greedy strategy

- take item 1.
- take item 2.

Thief can hold **50** kilogram.





Fractional knapsack problem

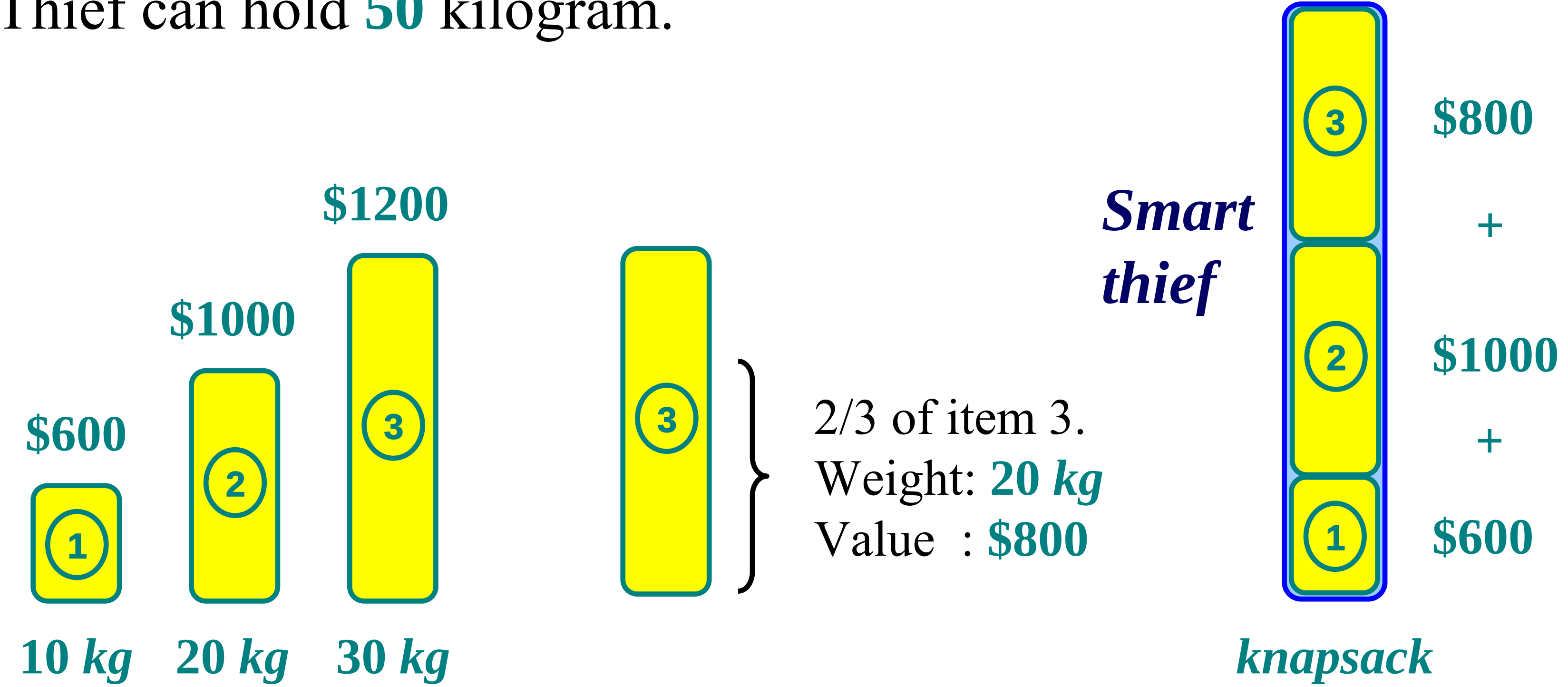
Item 1: **$60** per kilogram.
Item 2: **$50** per kilogram.
Item 3: **$40** per kilogram.

Thief can hold **50** kilogram.

Greedy strategy

- take item 1.
- take item 2.
- take 2/3 of item 3.

Total:
$2400



Character-coding problem

	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

Suppose we have a 100,000-character data file.

- **Fixed-length codeword**

$(45 \cdot 3 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 3 + 5 \cdot 3) \cdot 1{,}000 = 300{,}000$ bits

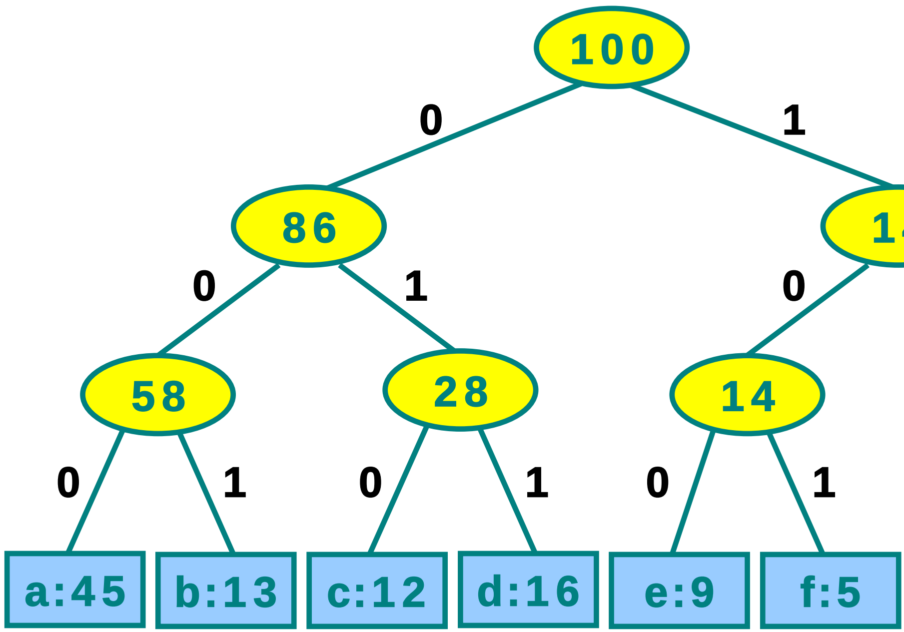
- **Variable-length codeword**

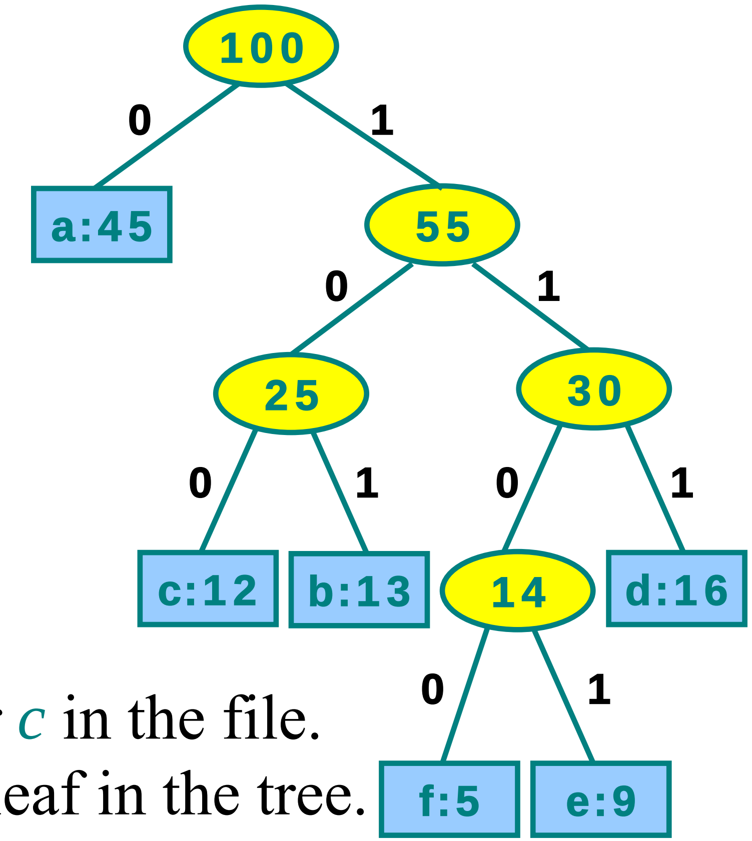
$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000$ bits

- **Savings of approximately 25%.**

$(300{,}000 - 224{,}000) / 300{,}000 \approx 25\%$

Tree corresponding to the coding



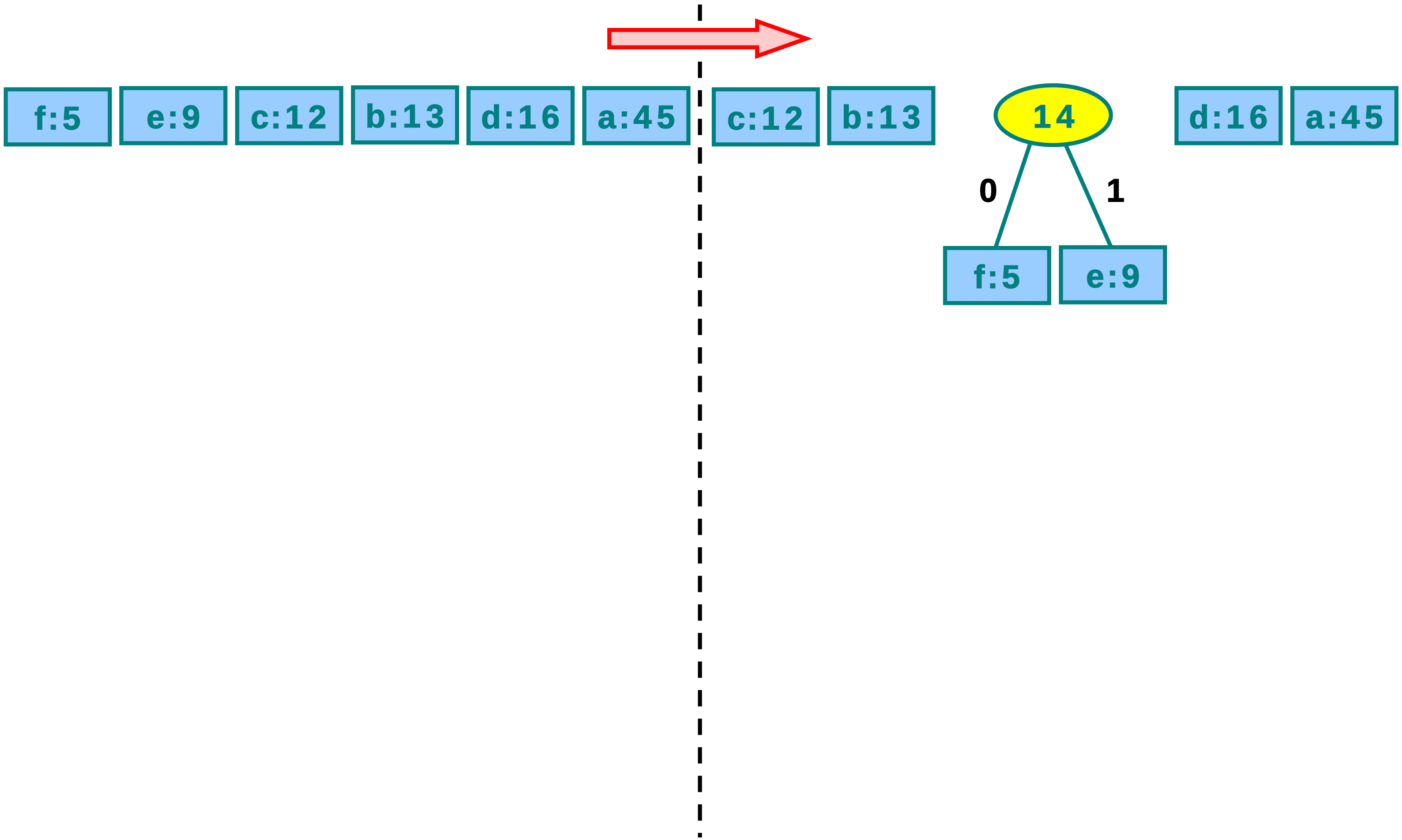


$f(c)$ denote the frequency of character c in the file.

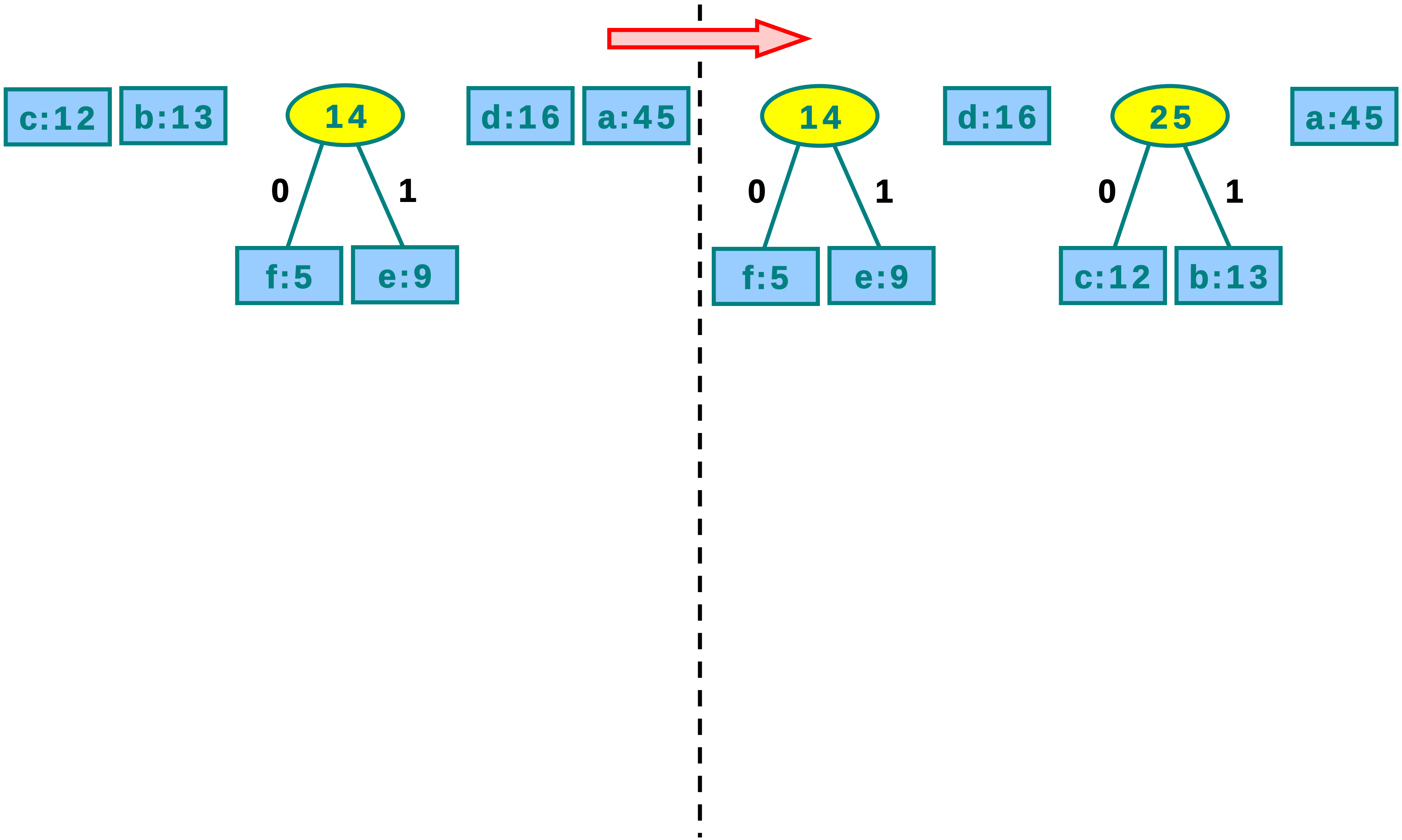
$d_T(c)$ denote the depth of character c's leaf in the tree.

Cost of the tree T: $$B(T) = \sum_{c \in C} f(c) d_T(c)$$

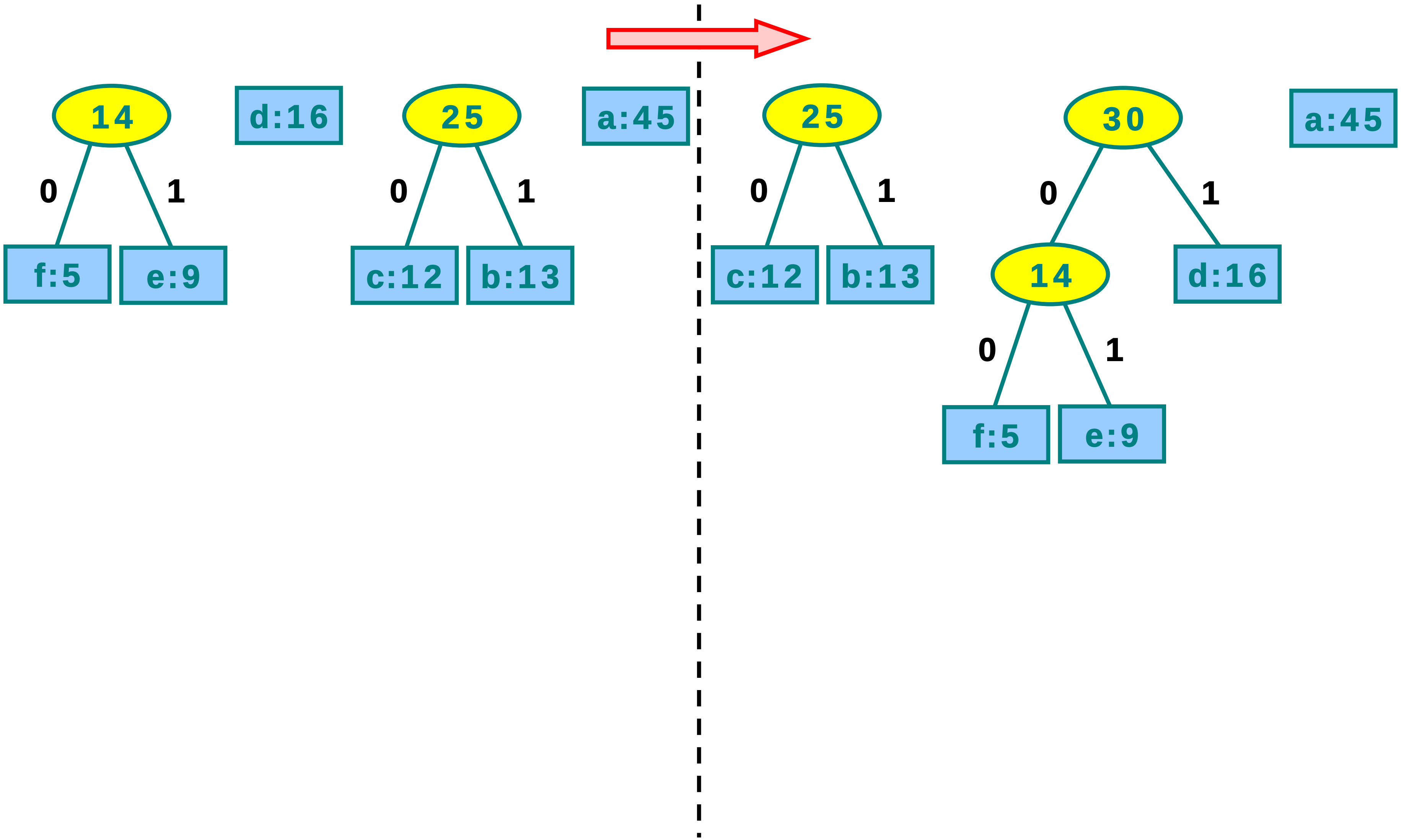
Constructing a Huffman code



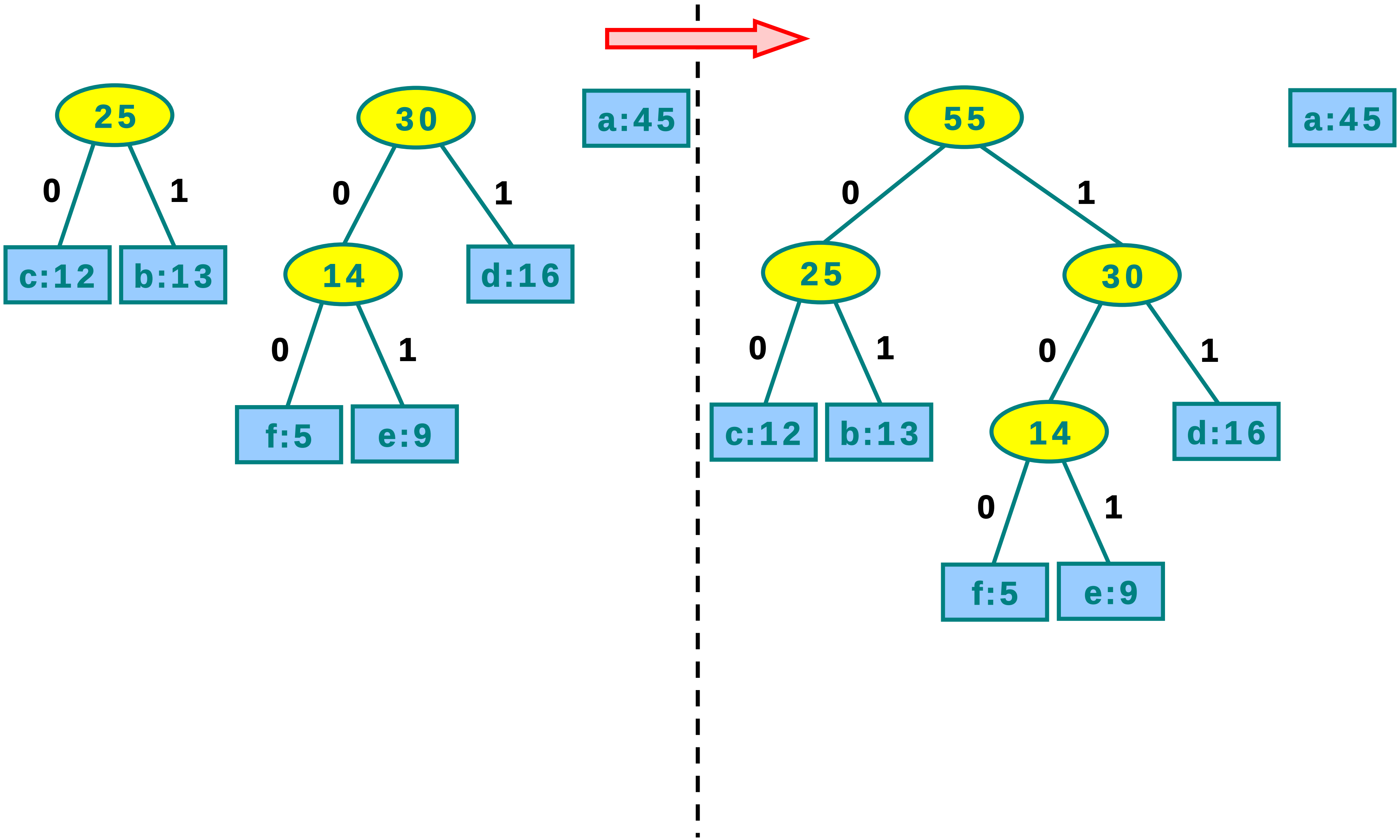
Constructing a Huffman code



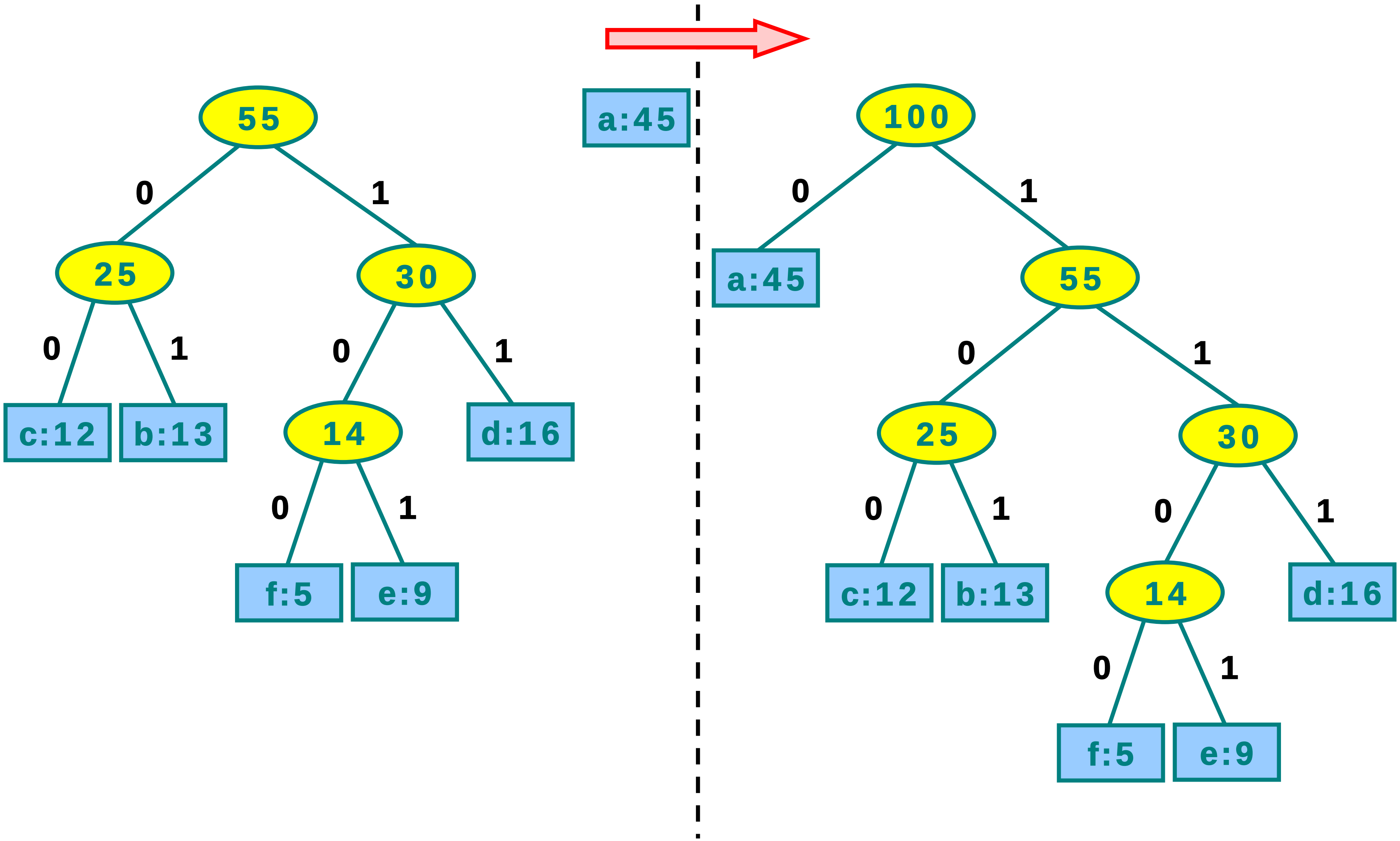
Constructing a Huffman code



Constructing a Huffman code



Constructing a Huffman code



Build Huffman codes

```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n − 1
    do allocate a new node z
        left[z] ← x ← EXTRACT-MIN(Q)
        right[z] ← y ← EXTRACT-MIN(Q)
        f [z] ← f [x] + f [y]
       INSERT(Q, z)
return EXTRACT-MIN(Q)
```

Running time

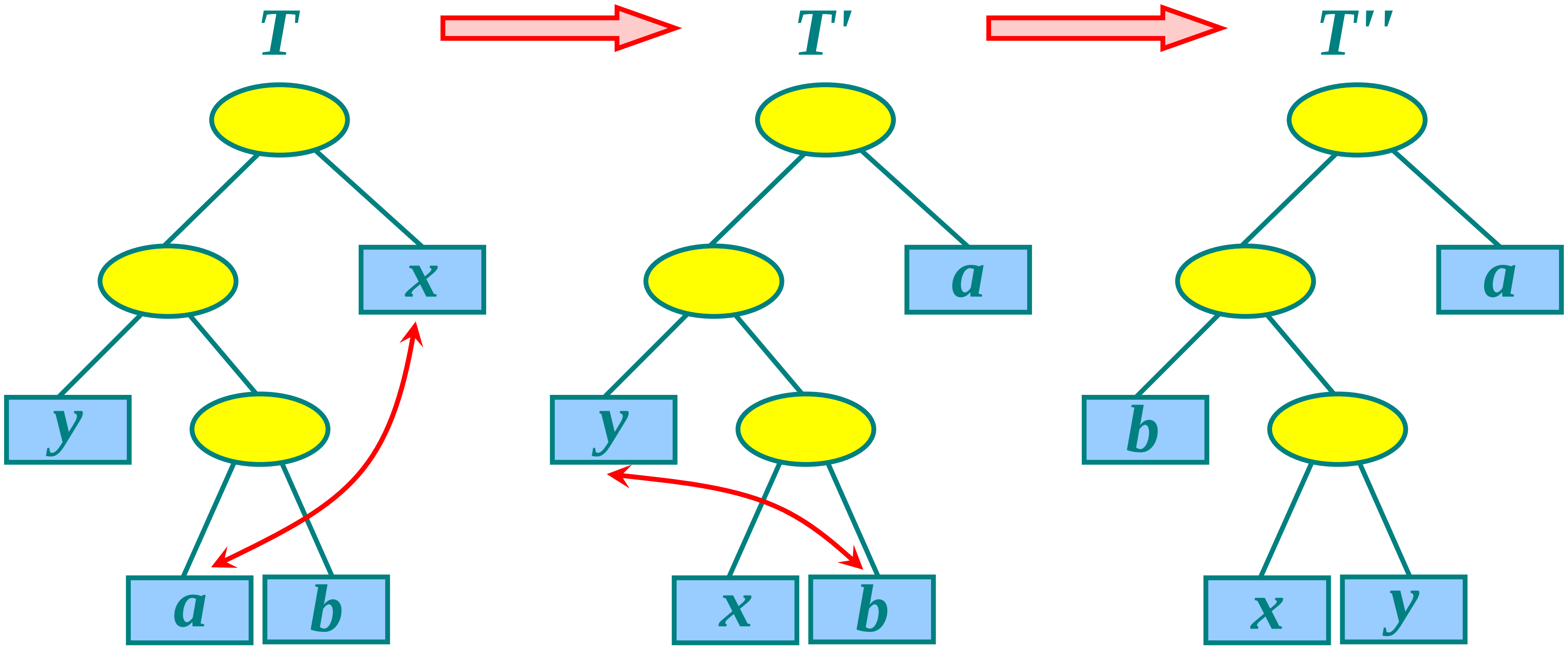
$O(n\lg n)$

Correctness of Huffman's algorithm

Theorem.

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 code words for x and y have the ***same length*** and differ only in the last bit.

Correctness of Huffman's algorithm



Thinking and practice.

- *Why don't greedy algorithms always work?*
- *What are differences between greedy algorithms and dynamic programming.*
- *Try to solve knapsack problem by dynamic programming.*

Any question?

Xiaoqing Zheng
Fundan University