HANDONG GLOBAL UNIVERSITY

Chapter 16
Greedy Algorithms

Algorithm Analysis

School of CSEE

Dynamic Programming is Blind

- The algorithms we have studied are relatively inefficient:
  - Matrix chain multiplication: $O(n^3)$
  - Longest common subsequence: $O(mn)$
  - Optimal binary search trees (?): $O(n^3)$
- Why?

✓ We have many choices in computing an optimal solution.

✓ We exhaustively (blindly) check all of them.

Example : MCM

$S_{i,j}$

i \ j	1	2	3	4	5	6
1		1		3	3	
2				3	3	3
3				3	3	3
4						
5						5
6						

✓ We would much rather like to have a way to decide which choice is the best or at least restrict the choices we have to try.

✓ Greedy algorithms work for problems where we can decide what's the best choice.

Greedy algorithms

- A design strategy for some optimization problems.
- Make the choice that looks best at the moment.

  --- local optimum
- Not always lead to a globally optimum solutions.
- But come up with the global optimum in many cases.

Coin change

- Want to make change with minimum number of coins.
- Algorithm: use as many coins of the next largest denominations and so forth.
- Example:

coins={half-dollar,quarter,dime,nickel,penny}

74cents = 1(half-dollar)+2(dime)+4(penny)

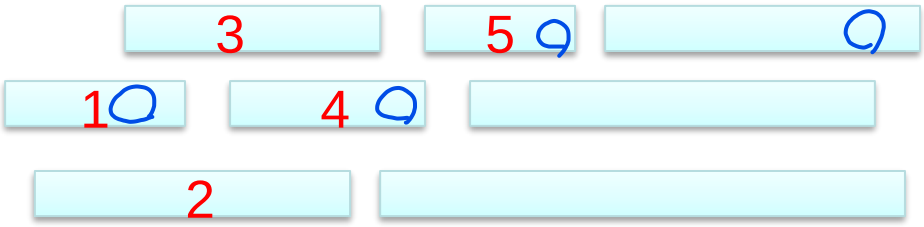
50 10

Activity-Selection Problem

- Problem: get your money's worth at amusement park
  - Lots of rides, each starting and ending at different times
  - Your goal: ride as many rides as possible
    - Another alternative goal that we don't solve here: maximize time spent on rides
- Welcome to the *activity selection problem*.

An activity-selection problem

- We want to schedule several activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities.
- An activity a_i is represented $[s_i, f_i)$, where s_i is a start time and f_i is a finish time.
- a_i and a_j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- Assume that the classes are sorted according to increasing finish times; that is, $f_1 < f_2 < \ldots < f_n$.



An activity-selection problem

- To select a maximum-size subset of mutually compatible activities.
- Example:

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

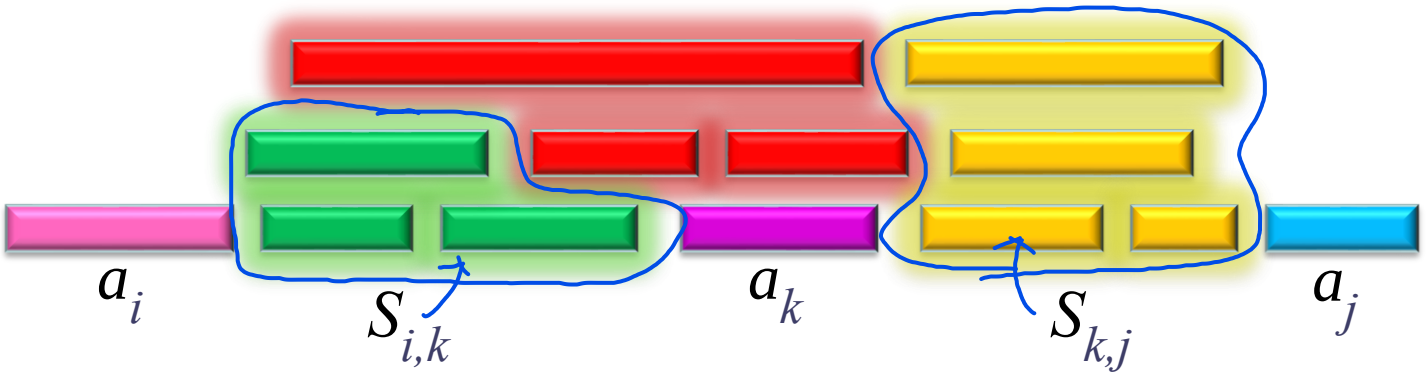
Mutually compatible activities: $\{a_3, a_9, a_{11}\}\{a_1, a_4, a_8, a_{11}\}\{a_2, a_4, a_9, a_{11}\}$

이것 강조해 주세요.

Largest mutually compatible activities: $\{a_1, a_4, a_8, a_{11}\}$ $\{a_2, a_4, a_9, a_{11}\}$

The Structure of an Optimal Schedule

- Let S be set of n proposed activities.
- Let $S_{i,j}$ be the subset of activities in S that begin after activity a_i finishes and end before activity a_j starts.
- We can add two fictitious activities a_0 and a_{n+1} with $f_0 = 0$ and $s_{n+1} = +\infty$. Then $S_{0,n+1}$ is the set of all activities S.



The Structure of an Optimal Schedule

- Assume that activity a_k is part of an optimal schedule of the classes in $S_{i,j}$.
- Then $i < k < j$, and the optimal schedule consists of a maximal subset of $S_{i,k}$, $\{a_k\}$, and a maximal subset of $S_{k,j}$.

it should follow an optimal schedule too. and the optimal schedule should be the same schedule with a schedule which is optimal itself.

⇒ optimal substructure.

The Structure of an Optimal Schedule

- Hence, if c_{ij} is the size of an optimal schedule for set S_{ij}, we have

"Dynamic programming"

$$c_{ij} = \begin{cases} 0 & \text{if } S_{ij} = \phi \\ \max_{i<k<j,\ a_k \in S_{ij}} \{c_{ik} + c_{kj} + 1\} & \text{if } S_{ij} \neq \phi \end{cases}$$

$i<k<j$ 인 k 값들 중에서

$C_{ik} + C_{kj} + 1$ 가 가장 큰 값을 찾는다...

그래서 Greedy Algorithm 으로 a_k를 어떤 경우에 고르겠다. (not-blind)

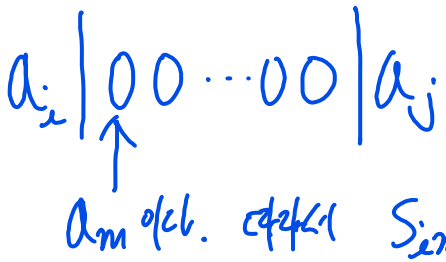
⇒ a_k의 모든 경우를 다 따져 보아야...

또 C_{ik}, C_{kj}를 다시 계산할 필요 없음..

Theorem

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time. Then,

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}
2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.



Theorem

Proof.

2. Suppose there is some $a_k \in S_{im}$. Then $f_i \le s_k < f_k \le s_m$ ⇒ $f_k < f_m$. Then $a_k \in S_{ij}$ and it has an earlier finish time than a_m, which contradicts our choice of a_m.

1. Let A_{ij} be a maximum-size subset of mutually compatible activities in S_{ij}. Order activities in A_{ij} in monotonically increasing order of finish time. Let a_k be the first activity in A_{ij}. If $a_k = a_m$, done. Otherwise, replace a_k by a_m to construct A_{ij}'

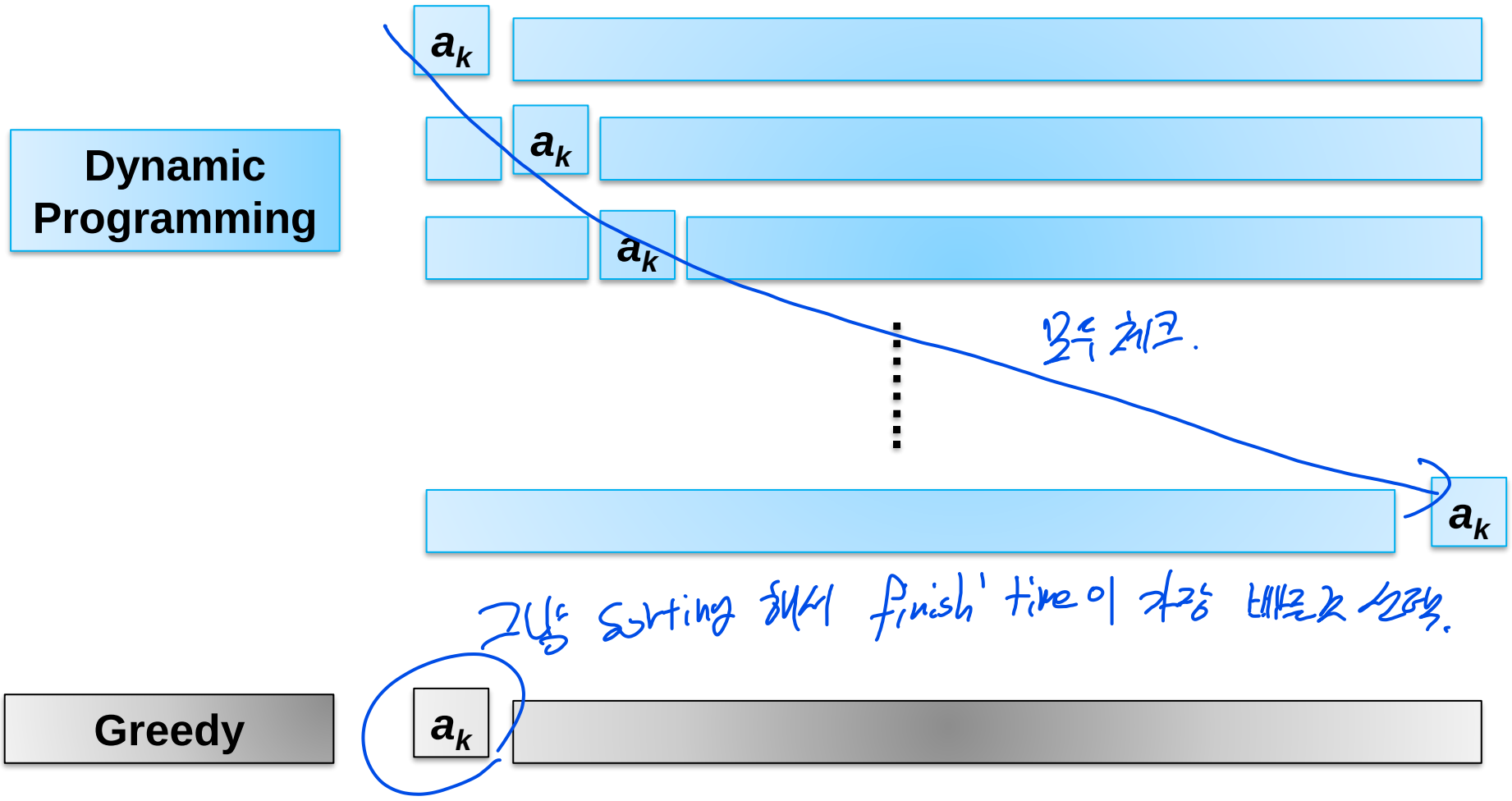
Dynamic Programming vs. Greedy

- Number of subproblems

  - Dynamic programming : Two subproblems are used in an optimal solution and there are $(j\text{-}1) - (i+1) + 1 = j - i - 1$ choices when solving the subproblem $S_{i,j}$.

  - Greedy : Only one subproblem is used in an optimal solution and when solving the subproblems $S_{i,j}$, we need consider only one choice.

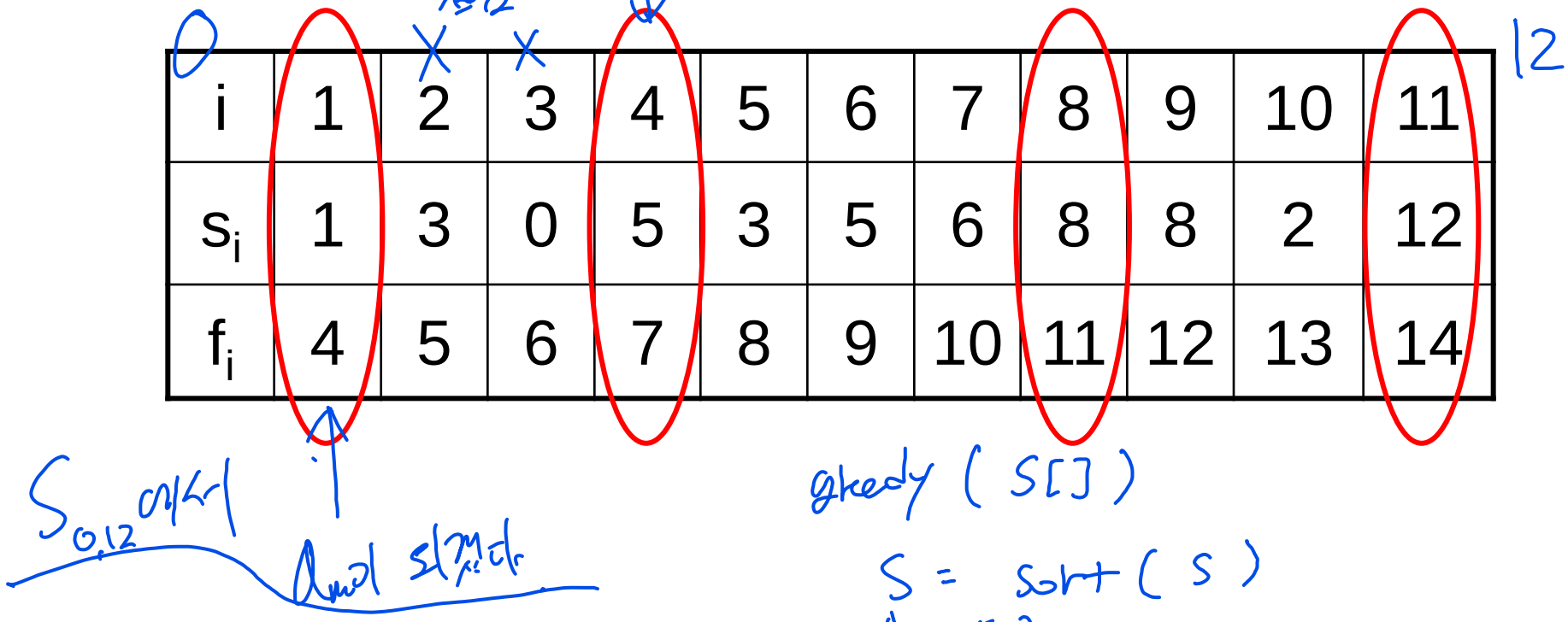
Dynamic Programming vs. Greedy



An activity-selection problem: greedy algorithm

- Example:

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



A Variation of the Problem

- Instead of maximizing the number of classes we want to schedule, we want to maximize the total time the classroom is in use.
- Our dynamic programming approach still remains unchanged.
- But none of the obvious greedy choices would work:
  - Choose the class that starts earliest/latest
  - Choose the class that finishes earliest/latest
  - Choose the longest class

When Do Greedy Algorithms Produce an Optimal Solution?

- Greedy choice property:
  - An optimal solution can be obtained by making choices that seem best at the time, without considering their implications for solutions to subproblems.

Huffman Codes : Text Encoding

- Our next goal is to develop a code that represents a given text as compactly as possible.
- A standard encoding is ASCII, which represents every character using 7 bits:
  - "An English sentence"
  - 1000001 (A) 1101110 (n) 0100000 ( ) 1000101 (E)
    1101110 (n) 1100111 (g) 1101100 (l) 1101001 (i)
    1110011 (s) 1101000 (h) 0100000 ( ) 1110011 (s)
    1100101 (e) 1101110 (n) 1110100 (t) 1100101 (e)
    1101110 (n) 1100011 (c) 1100101 (e)
  - = 133 bits ≈ 17 bytes

Huffman Codes : Text Encoding

- Of course, this is wasteful because we can encode 12 characters in 4 bits:
  - ‹space› = 0000 A = 0001 E = 0010 c = 0011 e = 0100 g = 0101 h = 0110 i = 0111 l = 1000 n = 1001 s = 1010 t = 1011
- Then we encode the phrase as
  - 0001 (A) 1001 (n) 0000 ( ) 0010 (E) 1001 (n) 0101 (g) 1000 (l) 0111 (i) 1010 (s) 0110 (h) 0000 ( ) 1010 (s) 0100 (e) 1001 (n) 1011 (t) 0100 (e) 1001 (n) 0011 (c) 0100 (e)
- This requires 76 bits ≈ 10 bytes

Huffman Codes : Text Encoding

- An even better code is given by the following encoding:

  Variable length.
  non-fixed size.

  - ‹space› = 000 A = 0010 E = 0011 s = 010 c = 0110 g = 0111 h = 1000 i = 1001 l = 1010 t = 1011 e = 110 n = 111

- Then we encode the phrase as

  - 0010 (A) 111 (n) 000 ( ) 0011 (E) 111 (n) 0111 (g) 1010 (l) 1001 (i) 010 (s) 1000 (h) 000 ( ) 010 (s) 110 (e) 111 (n) 1011 (t) 110 (e) 111 (n) 0110 (c) 110 (e)

- This requires 65 bits ≈ 9 bytes

Prefix code

- No codeword is a prefix of some other codeword
- The optimal data compression achievable by a prefix code.
- Easy to decode

	a	b	c	d	e	f
Frequence (in thousands)	45	13	12	16	9	5
Variable-length code	0	101	100	111	1101	1100

001011101→aabe

Why Prefix Codes?

- Consider a code that is not a prefix code:
  - a = 01 m = 10 n = 111 o = 0 r = 11 s = 1 t = 0011
- Now you send a fan-letter to your favourite movie star. One of the sentences is "You are a star." You encode "star" as "1 0011 01 11".
- Your idol receives the letter and decodes the text using your coding table:
- 100110111 = 10 0 11 0 111 = "moron"
- Oops, you have just insulted your idol.
- Non-prefix codes are ambiguous.

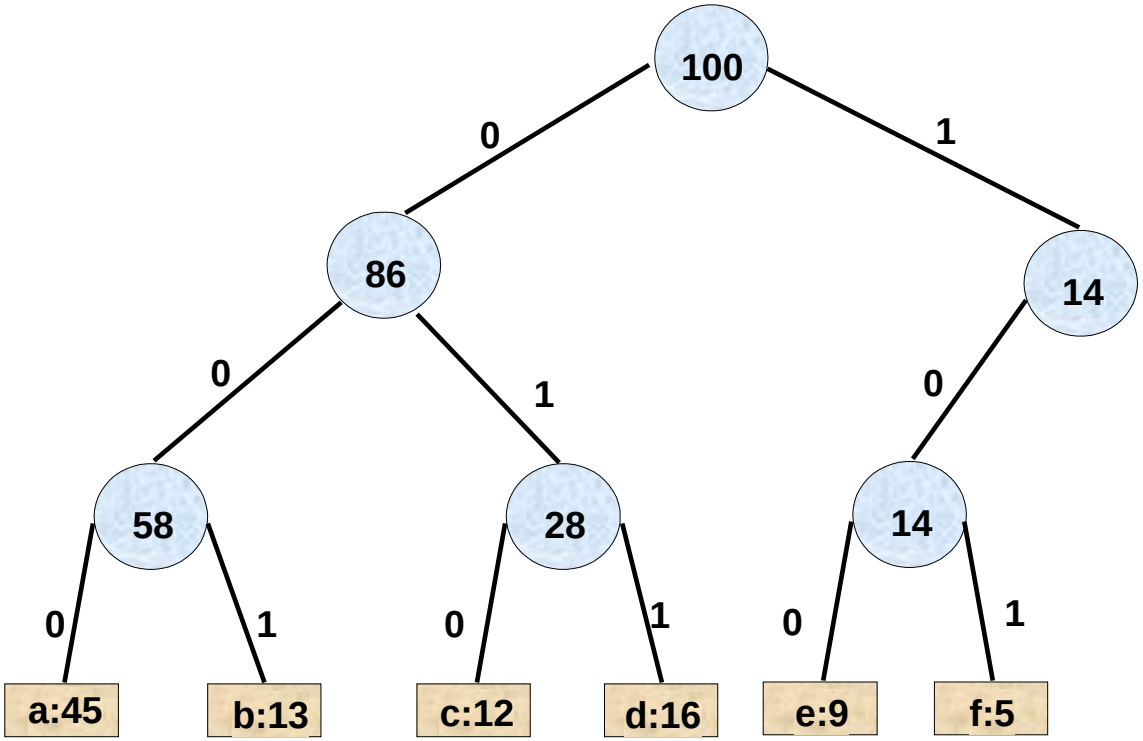
Why Are Prefix Codes Unambiguous?

- It suffices to show that the first character can be decoded unambiguously. We then remove this character and are left with the problem of decoding the first character of the remaining text, and so on until the whole text has been decoded.

- If code is ambiguous, then it is not prefix code. Thus, if it is prefix code, it is unambiguous

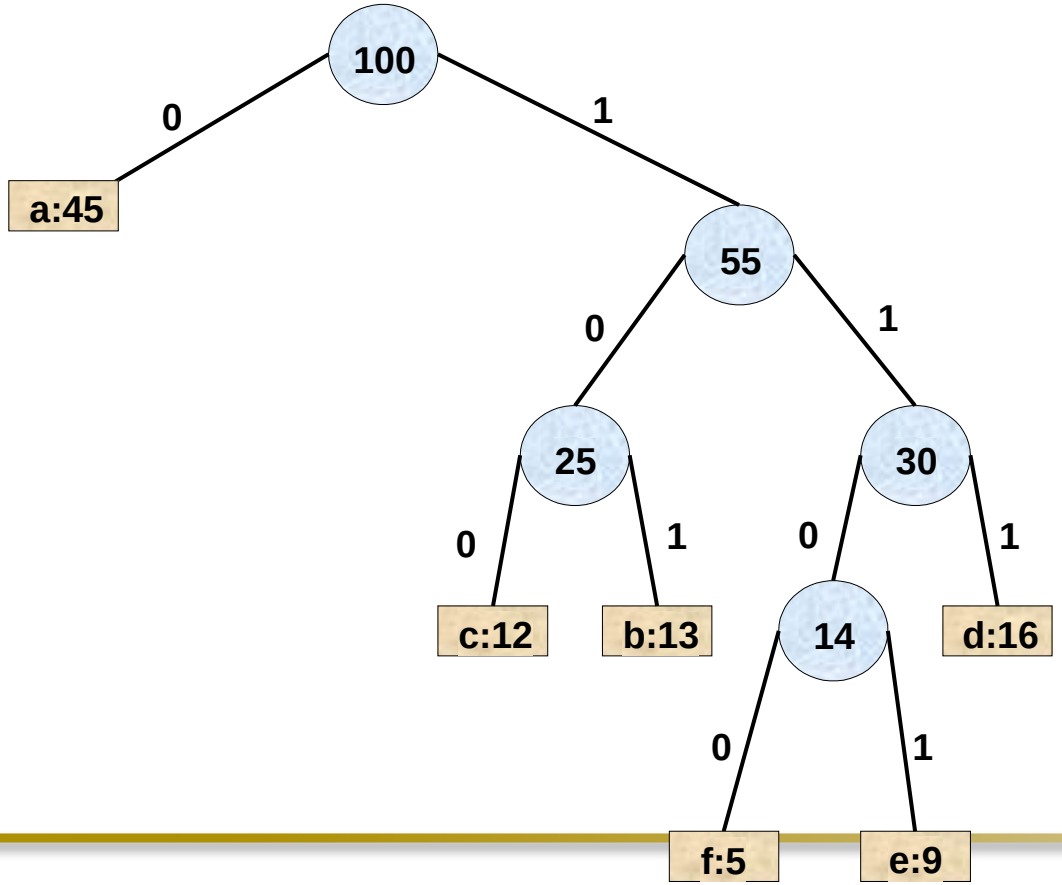
Decoding/encoding tree : Fixed-length code

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



Decoding/encoding tree : Variable-length code

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Variable-length code	0	101	100	111	1101	1100



Cost of encoding a file

- T: tree corresponding to the coding scheme c in the alphabet C.
- $f(c)$: the frequency of c in the file.
- $d_T(c)$: the depth of c's leaf in the tree.
- $B(T)$: the number of bits required to encode a file.

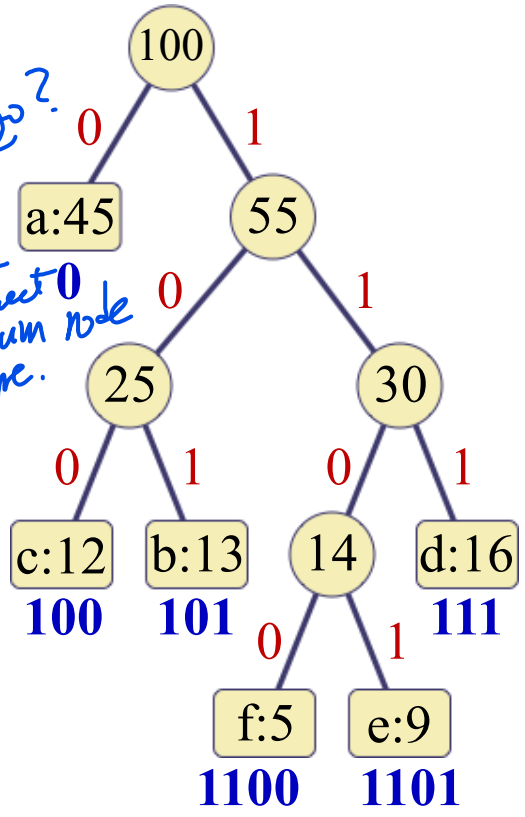
$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

Huffman codes

- Invented by Huffman
- Widely used
- Very effective technique for compressing data
- An optimal prefix code: proof in textbook

Huffman's Algorithm

```
Huffman-Code(C)
1   n ← |C|
2   Q ← C
3   for i = 1..n – 1
4       do allocate a new node z
5          left[z] ← Extract-Min(Q)
6          right[z] ← Extract-Min(Q)
7          f[z] ← f[left[z]] + f[right[z]]
8          Insert(Q, z)
9   return Extract-Min(Q)
```



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

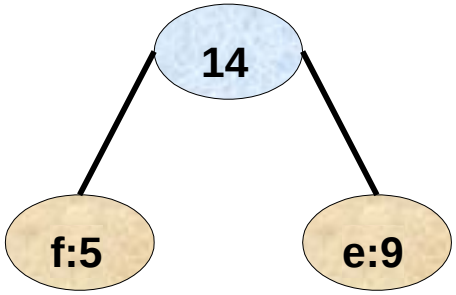
Huffman tree: example

Step 1:

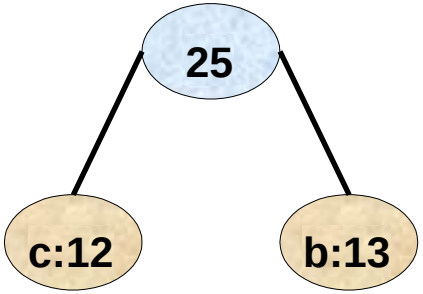
	a	b	c	d	e	f
Frequence	45	13	12	16	9	5

Step 2:

	a	b	c	d	
Frequence	45	13	12	16	14



	a	d		
Frequence	45	16	14	25

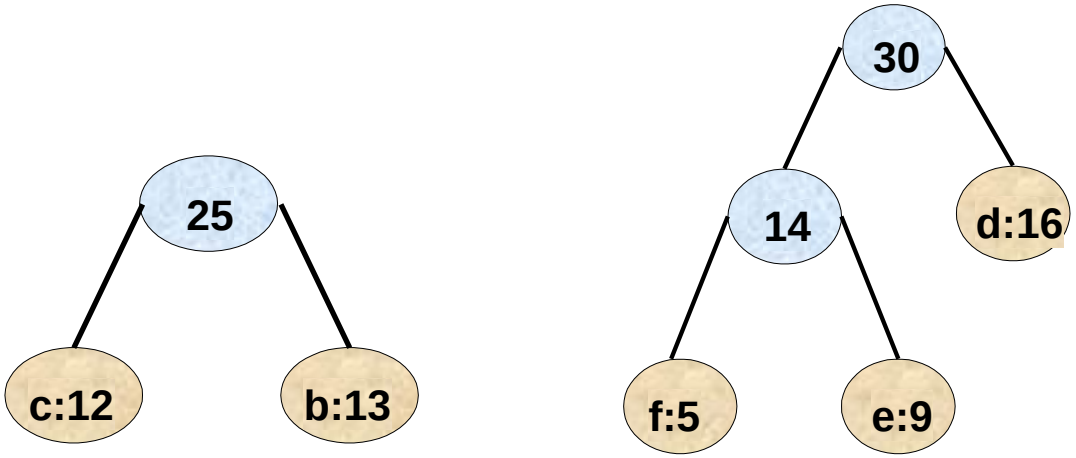


Huffman tree: example

Step 3:

	a	d		
Frequence	45	16	14	25

	a		
Frequence	45	30	25

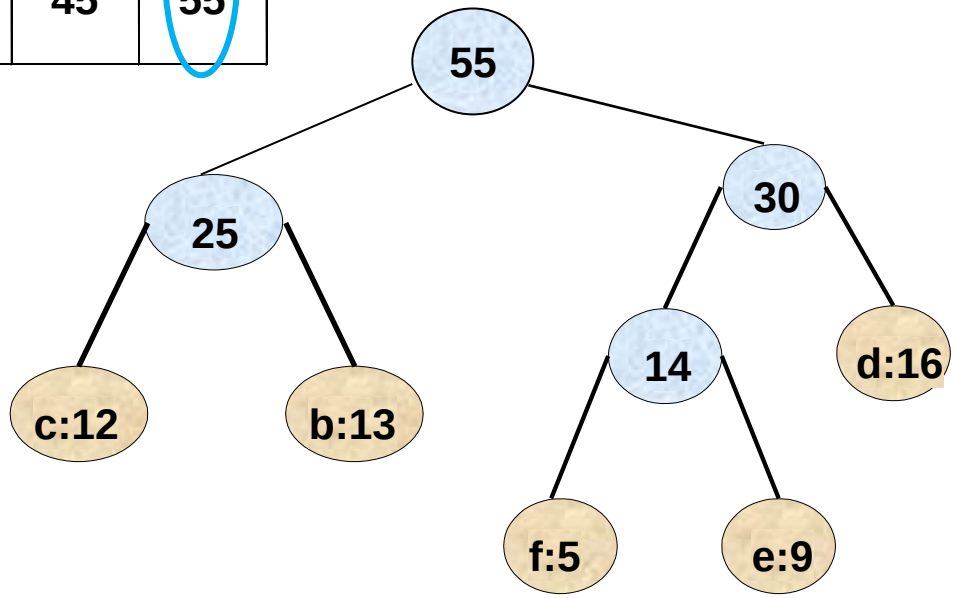


Huffman tree: example

Step 4:

	a		
Frequence	45	30	25

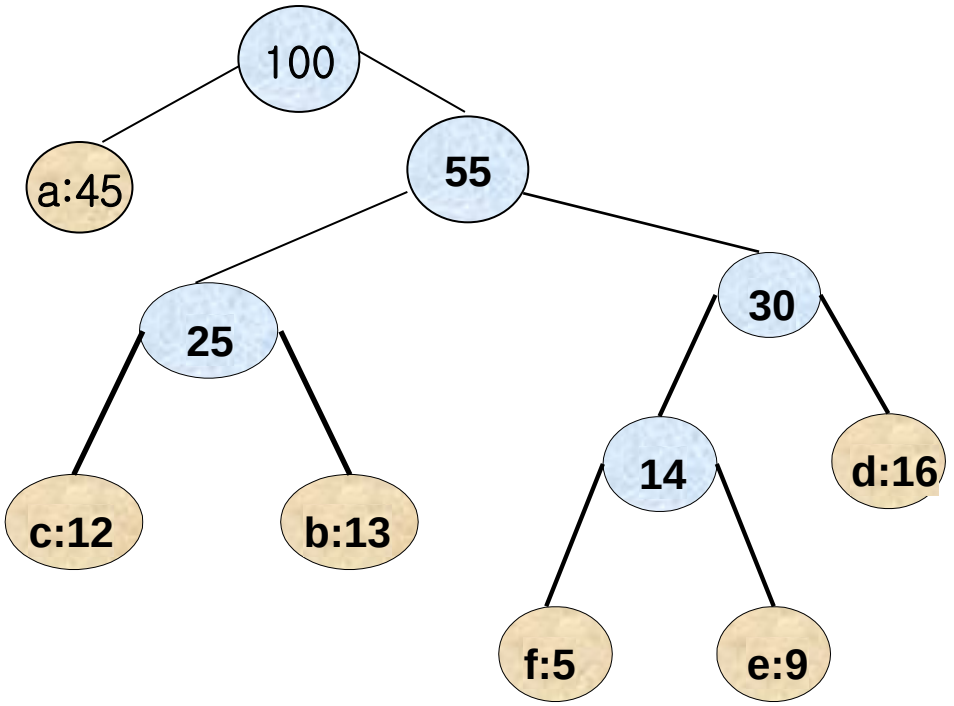
	a	
Frequence	45	55



Huffman tree: example

Step 5:

	a	
Frequence	45	55



Huffman tree: example

Coding scheme

0 for the left branch

1 for the right branch