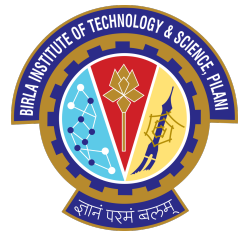
CS F364: Design & Analysis of Algorithms

Lecture-kt13: Dynamic Programming (contd.) + Greedy Algorithm



Dr. Kamlesh Tiwari,
Assistant Professor,
Department of Computer Science and Information Systems,
BITS Pilani, Rajasthan-333031 INDIA

Feb 23, 2017 (Campus @ BITS-Pilani Jan-May 2017)

Recap: Dynamic Programming

- **DP** is applied to optimization problems having 1) Optimal substructure, and 2) Overlapping subproblems

Recap: Dynamic Programming

- **DP** is applied to optimization problems having 1) Optimal substructure, and 2) Overlapping subproblems
- Memorization is the main characteristic

Recap: Dynamic Programming

- **DP** is applied to optimization problems having 1) Optimal substructure, and 2) Overlapping subproblems
- Memorization is the main characteristic
- Let multiplication of **Matrix-chain** $< A_1, A_2, ..., A_n >$ has $P(n)$ number of ways to parenthesize then

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) \times P(n-k) & \text{otherwise} \end{cases}$$

Recap: Dynamic Programming

- **DP** is applied to optimization problems having 1) Optimal substructure, and 2) Overlapping subproblems
- Memorization is the main characteristic
- Let multiplication of **Matrix-chain** $< A_1, A_2, ..., A_n >$ has $P(n)$ number of ways to parenthesize then

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) \times P(n-k) & \text{otherwise} \end{cases}$$

Solution is Catalan number $P(n) = C(n-1)$, where $C(n) = \frac{1}{n+1}{}^{2n}C_n = \Omega(4^n/n^{3/2})$.

Recap: Dynamic Programming

- **DP** is applied to optimization problems having 1) Optimal substructure, and 2) Overlapping subproblems
- Memorization is the main characteristic
- Let multiplication of **Matrix-chain** $< A_1, A_2, ..., A_n >$ has $P(n)$ number of ways to parenthesize then

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) \times P(n-k) & \text{otherwise} \end{cases}$$

  Solution is Catalan number $P(n) = C(n-1)$, where $C(n) = \frac{1}{n+1}{}^{2n}C_n = \Omega(4^n/n^{3/2})$.
- DP maintains two matrices m and s to optimally parenthesize

Recap: Dynamic Programming

- **DP** is applied to optimization problems having 1) Optimal substructure, and 2) Overlapping subproblems
- Memorization is the main characteristic
- Let multiplication of **Matrix-chain** $< A_1, A_2, ..., A_n >$ has $P(n)$ number of ways to parenthesize then

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) \times P(n-k) & \text{otherwise} \end{cases}$$

  Solution is Catalan number $P(n) = C(n-1)$, where $C(n) = \frac{1}{n+1}{}^{2n}C_n = \Omega(4^n/n^{3/2})$.
- DP maintains two matrices m and s to optimally parenthesize
- A **subsequence** of a sequence, is obtained by removing zero or more elements

Longest common subsequence

- In the longest-common-subsequence problem, we are given two sequences $X =< x_1 x_2 x_3 ... x_m >$ and $Y =< y_1 y_2 y_3 ... y_n >$ and wish to find a maximum-length common subsequence of X and Y.

Longest common subsequence

- In the longest-common-subsequence problem, we are given two sequences $X =< x_1 x_2 x_3 ... x_m >$ and $Y =< y_1 y_2 y_3 ... y_n >$ and wish to find a maximum-length common subsequence of X and Y.

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Longest common subsequence

Algorithm 1: LCS-Length(X, Y)

```
1  m = length[X]
2  n = length[Y]
3  for i = 1 to m do
4      c[i,0] = 0
5  for j = 0 to n do
6      c[0,j] = 0
7  for i = 1 to m do
8      for j = 1 to n do
9          if x_i = y_j then
10             c[i,j] = c[i-1,j-1]+1
11             b[i,j] = ↖
12         else
13             if c[i-1,j] ≥ c[i,j-1] then
14                 c[i,j] = c[i-1,j]
15                 b[i,j] = ↑
16             else
17                 c[i,j] = c[i,j-1]
18                 b[i,j] = ←
19 return c and b
```

Longest common subsequence

Algorithm 2: Print-LCS(b, X, i, j)

```
1  if i=0 or j=0 then
2      return
3  if b[i,j] = ↖ then
4      Print-LCS( b, X, i-1, j-1)
5      print x_i
6  else
7      if b[i,j] = ↑ then
8          Print-LCS( b, X, i-1, j)
9      else
10         Print-LCS( b, X, i, j-1)
```

Greedy Algorithm

- DP could overkill

Greedy Algorithm

- DP could overkill
- A greedy algorithm makes a **locally optimal choice** in the hope that the choice will lead to a globally optimal solution

Greedy Algorithm

- DP could overkill
- A greedy algorithm makes a **locally optimal choice** in the hope that the choice will lead to a globally optimal solution
- **Caution:** Greedy algorithms do not always yield optimal solutions

Greedy Algorithm

- DP could overkill
- A greedy algorithm makes a **locally optimal choice** in the hope that the choice will lead to a globally optimal solution
- **Caution:** Greedy algorithms do not always yield optimal solutions

Activity Selection Problem:

- Consider a set $S = \{1, 2, 3, ..., n\}$ of n activities that can happen one activity at a time. Activity i takes place during interval $[s_i, f_i)$.

Greedy Algorithm

- DP could overkill
- A greedy algorithm makes a **locally optimal choice** in the hope that the choice will lead to a globally optimal solution
- **Caution:** Greedy algorithms do not always yield optimal solutions

Activity Selection Problem:

- Consider a set $S = \{1, 2, 3, ..., n\}$ of n activities that can happen one activity at a time. Activity i takes place during interval $[s_i, f_i)$.
- Activity i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap

Greedy Algorithm

- DP could overkill
- A greedy algorithm makes a **locally optimal choice** in the hope that the choice will lead to a globally optimal solution
- **Caution:** Greedy algorithms do not always yield optimal solutions

Activity Selection Problem:

- Consider a set $S = \{1, 2, 3, ..., n\}$ of n activities that can happen one activity at a time. Activity i takes place during interval $[s_i, f_i)$.
- Activity i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
- Select maximum size set of mutually comparable activities.

Activity Selection Problem

Consider following set of activity

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

Activity Selection Problem

Consider following set of activity

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

- $\{3, 9, 11\}$ is a compatible activity

Activity Selection Problem

Consider following set of activity

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

- $\{3, 9, 11\}$ is a compatible activity
- $\{1, 4, 8, 11\}$ is larger compatible activity.

Activity Selection Problem

Consider following set of activity

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

- $\{3, 9, 11\}$ is a compatible activity
- $\{1, 4, 8, 11\}$ is larger compatible activity. In fact it is the largest

Activity Selection Problem

Consider following set of activity

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

- $\{3, 9, 11\}$ is a compatible activity
- $\{1, 4, 8, 11\}$ is larger compatible activity. In fact it is the largest
- Another largest compatible activity is $\{2, 4, 9, 11\}$

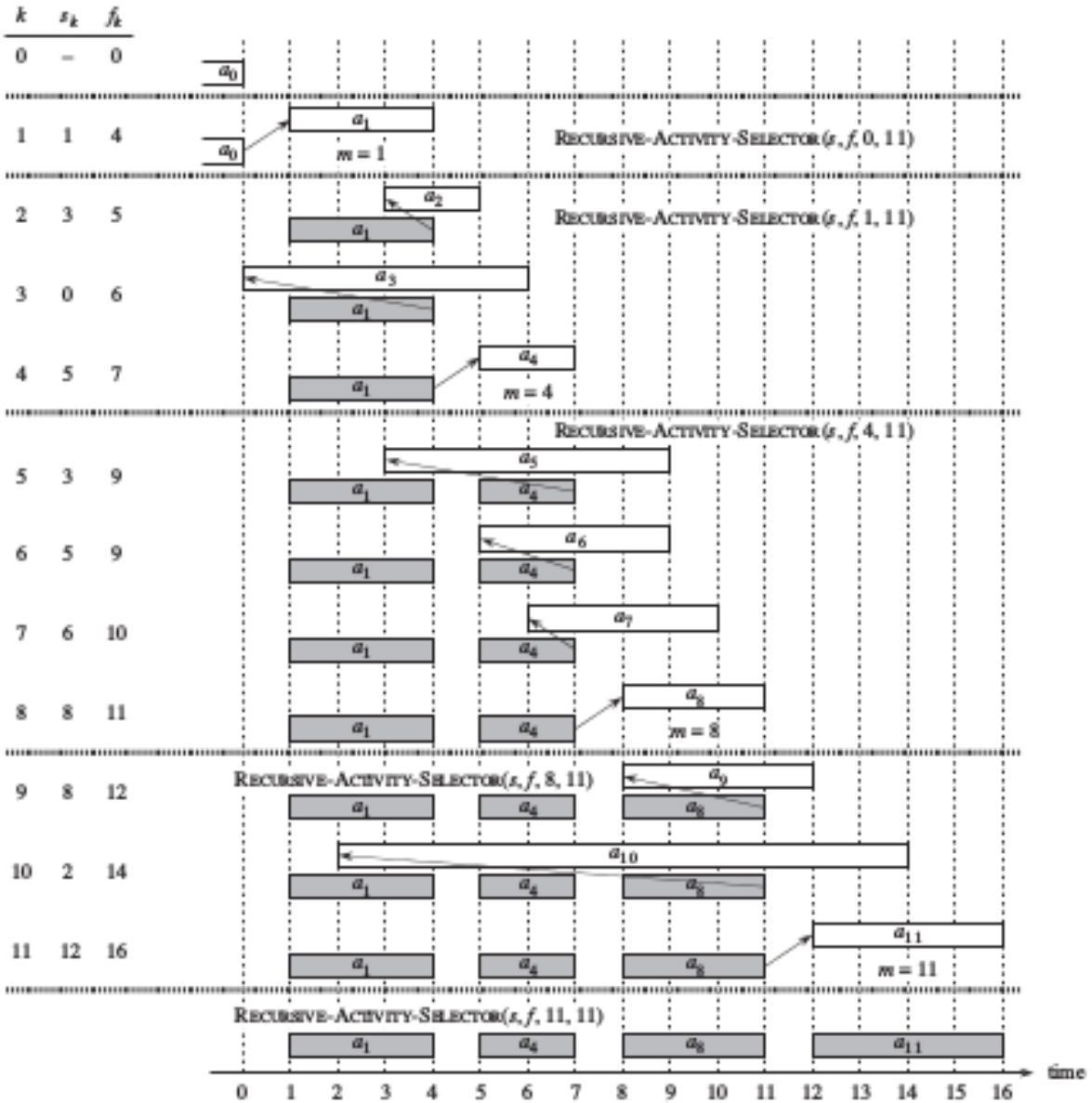
Activity Selection Problem

Assume that activities are in increasing order of their finishing time. If not, then sort it in $O(n \lg n)$ time.

Algorithm 3: Greedy-Activity-Selection(s, f)

```
1 n = length(s)
2 A = { 1 }
3 j = 1
4 for i = 2 to n do
5     if s_i ≥ f_j then
6         A = A ∪ { i }
7         j = i
8 return A
```

Activity Selection Problem



Greedy versus Dynamic programming

- A thief robbing a store finds n items.

Greedy versus Dynamic programming

- A thief robbing a store finds n items.
- The i^{th} item is worth v_i dollars and weighs w_i pounds. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack (consider v_i, w_i, and W as integer).

Greedy versus Dynamic programming

- A thief robbing a store finds n items.
- The i^{th} item is worth v_i dollars and weighs w_i pounds. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack (consider v_i, w_i, and W as integer).
- Which items should he take?

Greedy versus Dynamic programming

- A thief robbing a store finds n items.
- The i^{th} item is worth v_i dollars and weighs w_i pounds. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack (consider v_i, w_i, and W as integer).
- Which items should he take?

Fractional knapsack problem:

He can take fraction of an items as well

Greedy versus Dynamic programming

- A thief robbing a store finds n items.
- The i^{th} item is worth v_i dollars and weighs w_i pounds. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack (consider v_i, w_i, and W as integer).
- Which items should he take?

Fractional knapsack problem:

He can take fraction of an items as well

0-1 knapsack problem:

He can either take complete items or leave

Greedy versus Dynamic programming

- A thief robbing a store finds n items.
- The i^{th} item is worth v_i dollars and weighs w_i pounds. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack (consider v_i, w_i, and W as integer).
- Which items should he take?

Fractional knapsack problem:

He can take fraction of an items as well

0-1 knapsack problem:

He can either take complete items or leave

- Fractional knapsack problem can be solved with greedy but

Greedy versus Dynamic programming

- A thief robbing a store finds n items.
- The i^{th} item is worth v_i dollars and weighs w_i pounds. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack (consider v_i, w_i, and W as integer).
- Which items should he take?

Fractional knapsack problem:

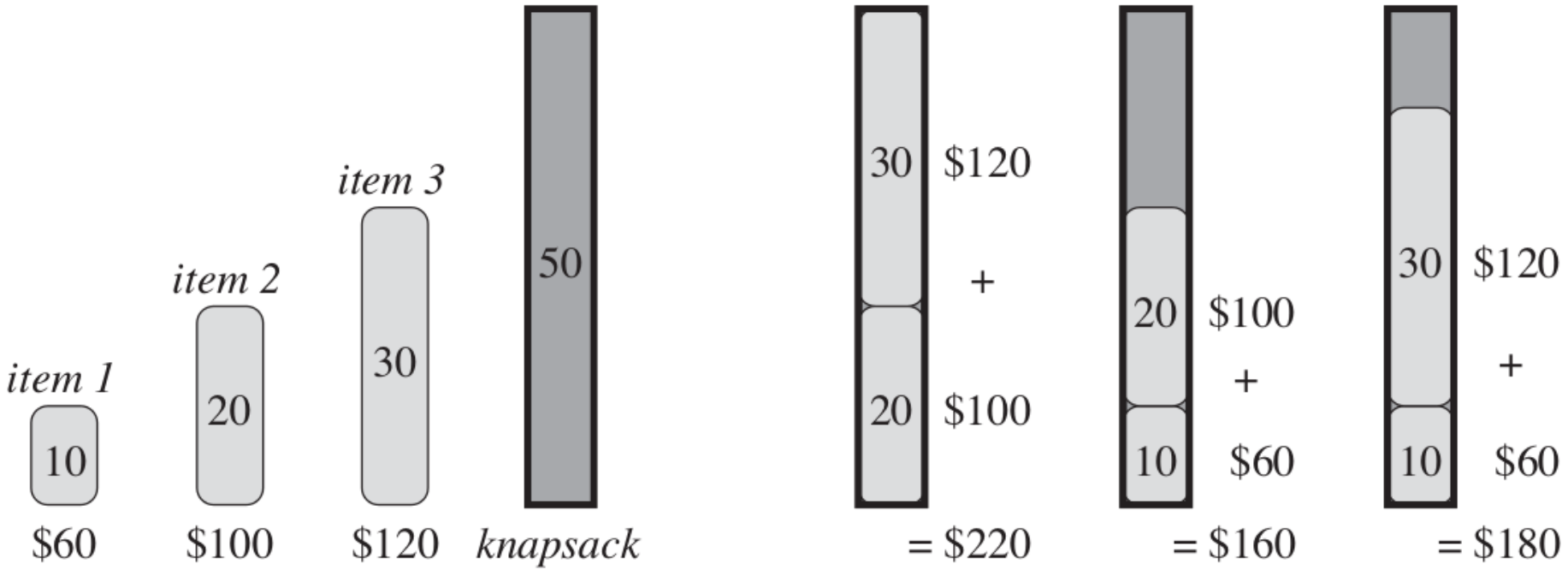
He can take fraction of an items as well

0-1 knapsack problem:

He can either take complete items or leave

- Fractional knapsack problem can be solved with greedy but
- 0-1 knapsack problem needs DP

0-1 knapsack problem needs DP



Huffman codes

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

Huffman codes

- Huffman invented a **greedy algorithm** that constructs an optimal prefix code called a Huffman code.
- It is a variable-length prefix code, useful for lossless data compression

Huffman codes

- Huffman invented a **greedy algorithm** that constructs an optimal prefix code called a Huffman code.
- It is a variable-length prefix code, useful for lossless data compression

Consider for example, a data file of 100,000 characters only containing six characters `a, b, c, d, e, f`

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

Huffman codes

- Huffman invented a **greedy algorithm** that constructs an optimal prefix code called a Huffman code.
- It is a variable-length prefix code, useful for lossless data compression

Consider for example, a data file of 100,000 characters only containing six characters `a, b, c, d, e, f`

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

Huffman codes

- Huffman invented a **greedy algorithm** that constructs an optimal prefix code called a Huffman code.
- It is a variable-length prefix code, useful for lossless data compression

Consider for example, a data file of 100,000 characters only containing six characters `a, b, c, d, e, f`

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

Huffman codes

- Huffman invented a **greedy algorithm** that constructs an optimal prefix code called a Huffman code.
- It is a variable-length prefix code, useful for lossless data compression

Consider for example, a data file of 100,000 characters only containing six characters `a, b, c, d, e, f`

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

- Fixed length code takes 300,000 bits

Huffman codes

- Huffman invented a **greedy algorithm** that constructs an optimal prefix code called a Huffman code.
- It is a variable-length prefix code, useful for lossless data compression

Consider for example, a data file of 100,000 characters only containing six characters `a, b, c, d, e, f`

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

- Fixed length code takes 300,000 bits
- Huffman code needs 224,000 bits ($\sim$25% compression)

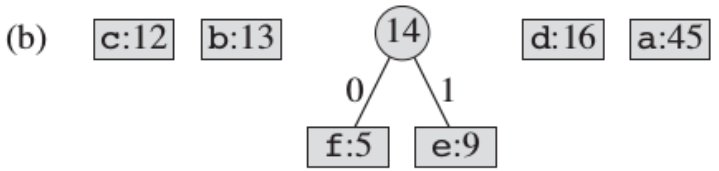
Huffman code

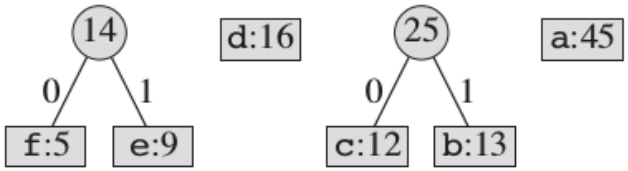
```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)
```

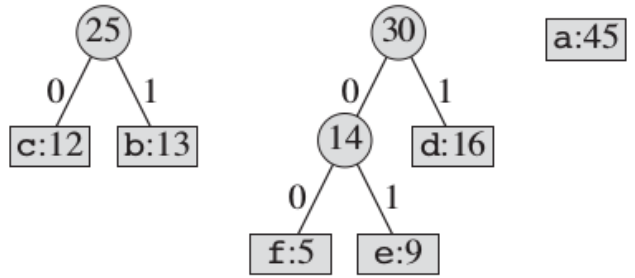
Huffman code

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

(a) 

(b) 

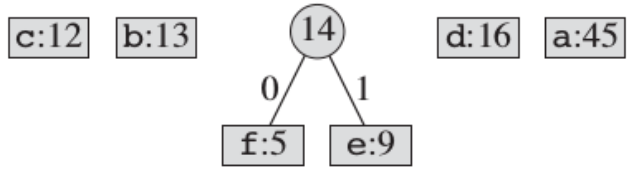
(c) 

(d) 

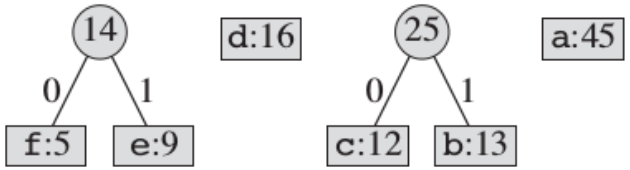
Huffman code

(a)

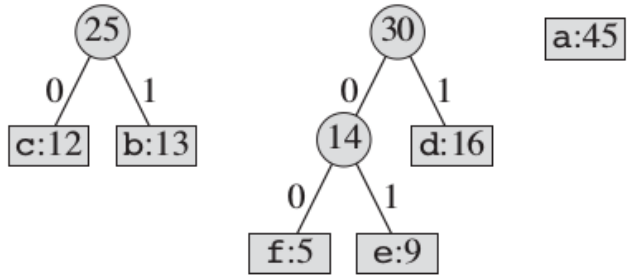
(b)



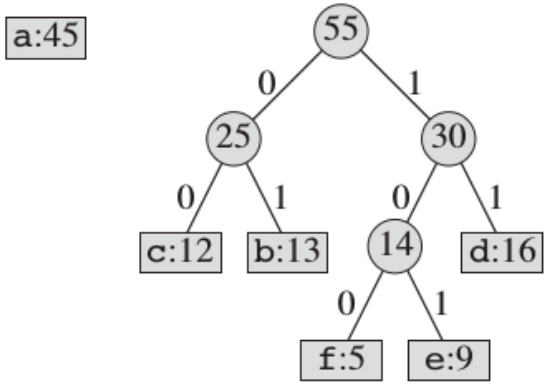
(c)



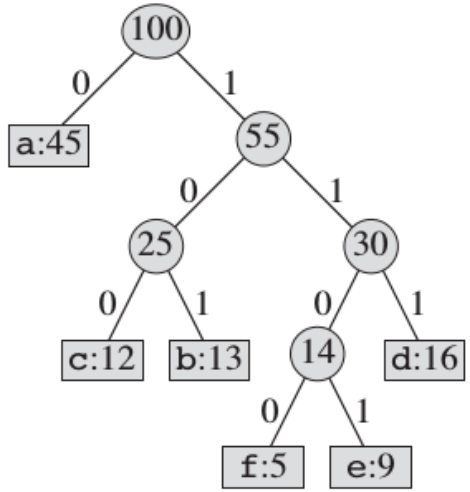
(d)



(e)



(f)



Thank You!

Thank you very much for your attention!

Queries ?