Greedy Technique

Activity Selection, Knapsack, Huffman Coding

Introduction

- Applied on optimization problems
- Greedy algorithm always makes a choice that looks best at the moment
- That is, a locally optimal choice is made in the hope of achieving a globally optimal one
- Making local choice does not always yield an optimal solution but as we shall see in this section, for many problems it does
- Whenever applicable it drastically reduces the number of subproblems required to be solved at each level thus, reducing the computation time manifold
- Applied in many graph algorithms as well such as Dijkstra's algorithm for single source shortest path and calculation of minimum spanning tree

Proving that Greedy strategy works

- Greedy problem solving requires two properties to be satisfied by the original problem
  - Optimal substructure
  - Greedy choice property
- For proving that a greedy algorithm works, we need to show that making a locally optimal choice yields a global optimal
- This is usually done by assuming a globally optimal choice and showing that it can be replaced by some greedy choice

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 that can serve only one activity at a time (for example a set of events to be organized in a single lecture hall)
- Each activity has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$
- If an activity a_i is selected for using the resource, it takes place during the half-open interval $[s_i, f_i)$
- Two activities a_i and a_j are said to be ***compatible*** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap that is, if $s_i \geq f_j$ or $s_j \geq f_i$
- In **activity-selection problem**, 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	9	9	10	11	12	14	16

- $\{a_3, a_9, a_{11}\}$ is a mutually compatible set of activities
- $\{a_1, a_4, a_8, a_{11}\}$
- $\{a_2, a_4, a_9, a_{11}\}$
- Above are two largest mutually compatible set of activities

Solution

- We can use a dynamic programming approach for solving this problem
- Suppose, the initial optimal choice is at some point ‘k’ so that the activity a_k belongs to the optimal set of activities
- We will define the optimal solution recursively using this as follows:
- Let us define S_{ij} to be the set of activities that start after a_i finishes and finish before a_j starts
- Let us further define a set A_{ij} that consists of the largest number of mutually compatible activities in S_{ij}

- Let activity a_k belongs to the maximal set A_{ij} of S_{ij}
- We can describe the remaining selection of activities into two subproblems
- One is the finding mutually compatible activities in the set S_{ik} and the other is finding mutually compatible activities in the set S_{kj}
- Now, the two subproblems must be solved optimally in order for the original problem to be solved optimally (optimal substructure holds)
- We can define A_{ik} as - $A_{ik} = A_{ij} \cap S_{ik}$
- Similarly, A_{kj} can be defined as - $A_{kj} = A_{ij} \cap S_{kj}$
- A_{ij} can be defined as the union of three mutually exclusive sets:

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

- The optimal substructure thus yields the following recurrence:

$$c[i, j] = c[i, k] + c[k, j] + 1$$

- The above steps will be repeated for all values of k
- Therefore, the optimal solution can be given as:

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

- Either a top-down approach with memoization can be used for solving this or a bottom up approach can by used by filling the table entries

Greedy Approach

- It turns out that the activity selection problem has the interesting property that a greedy choice can be made at each step thus reducing the number of subproblems to be solved to just 1
- The greedy choice made at each step is dependent on some criteria that gives optimal solution
- For the activity selection problem, greedy choice could be to select the activity that finishes earliest as it would leave maximum space to accommodate more activities
- We assume that the input is sorted in the order of finishing time and therefore the activity with smallest finishing time appears in the front, thus we will choose activity a_1

- If we make the greedy choice at the first step, we don't need to look for all possible values of 'k', thus reducing the number of subproblems that need be solved
- We are left with only one subproblem to be solved S_k – making a greedy choice among activities that start after activity a_1 finishes
- Since the problem has optimal substructure, making a greedy choice among the remaining activities yields an optimal global solution
- We are making a greedy choice at each step and are left with only one subproblem to solve
- Greedy algorithms are therefore, solved in top-down manner
- We can initially write a recursive algorithm to solve the problem and later re-write it as an iterative algorithm

Proof that the Greedy choice works

- Let us consider a subproblem S_k and let a_m be any activity in S_k with the earliest finish time. Then, we will prove that a_m is include in some maximal subset of mutually compatible activities in S_k
- Let A_k be an optimal subset of mutually compatible activities in S_k and let a_j be the first activity to finish in A_k. Then, we have the following:

- Case 1: $a_j = a_m$ then, we have already proved that a_m is included in A_k

- Case 2: $a_j \neq a_m$
- Let us define another set of activities $A'_k = A_k - \{a_j\} \cup \{a_m\}$
- We claim that A'_k defined above is disjoint. This is because A_k is disjoint and a_m is the first activity to finish in S_k. Therefore, $f_m \leq f_j$
- Further, $|A'_k| = |A_k|$ therefore, A'_k is also a maximal subset of mutually compatible activities and hence a solution.
- Thus, we have proved that making a greedy choice at each step always yields an optimal solution for the activity selection problem

Recursive Greedy Algorithm for Activity-Selection Problem

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Iterative Greedy Algorithm

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

- Both the algorithms run in $\theta(n)$ time

Not all greedy choices work

- The greedy choice should be made carefully so as to obtain the global optimum by making greedy choices. For example, the following greedy choices for the activity selection problem do not yield an optimal solution
- Selection of activities with least duration
- Selection of compatible activities with earliest start time
- Selecting activities that overlap with fewest remaining activities

When to apply Greedy Algorithm

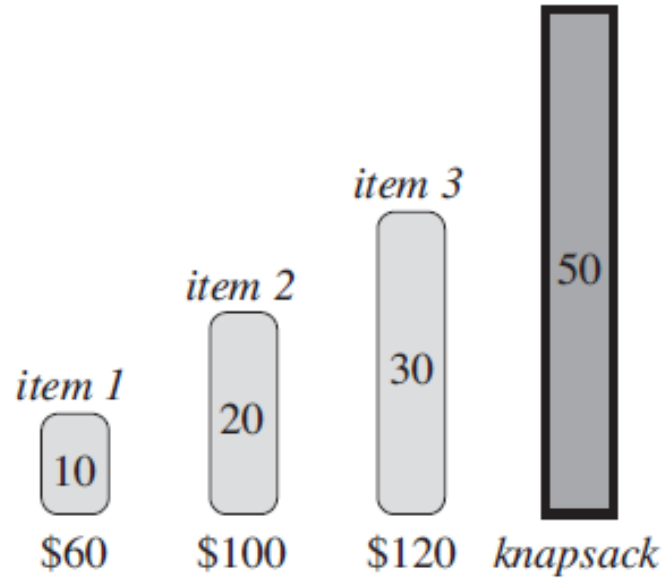
- Optimization Problem
- Greedy choice property – making a greedy choice at each step, yields a global optimum
- Optimal Sub-structure
- We usually prove point no. 2 in the same manner we proved it for activity-selection problem i.e. by assuming a global optimal solution and then proving that a random element of it can be replaced by a greedy choice made at some level and hence, making greedy choices every time will lead to an optimal solution
- Greedy choices can be made quickly if we process the data or use an appropriate data structure. E.g. sorting the activities in decreasing order of finish time causes making of greedy choice much easier

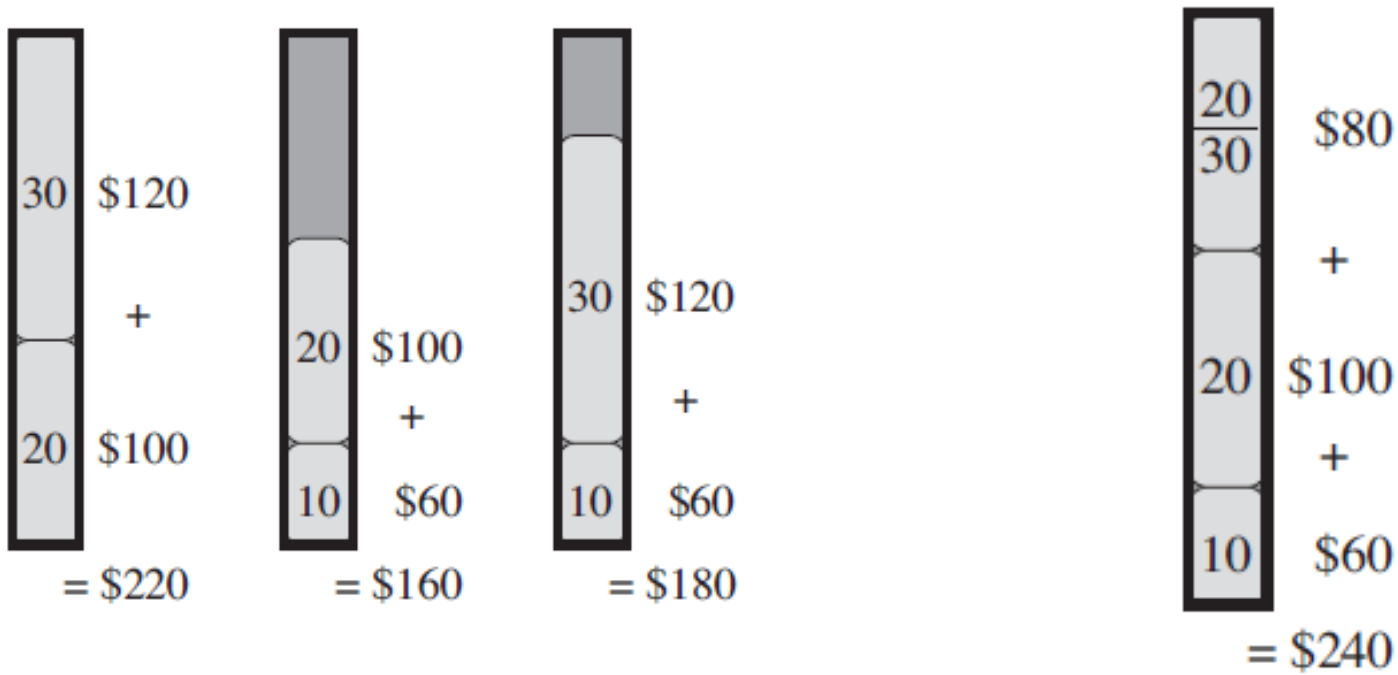
Knapsack Problem

- A thief robbing a store finds *n* items. The ith item costs v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief 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 the thief should take?
- The above problem has two variants:
  - 0/1 Knapsack – each item can either be taken completely or not taken at all and the thief can take at most one item of one type
  - Fractional Knapsack – the thief can take fractions of an item rather than always making a binary (0-1) choice.

Fractional Knapsack

- The fractional knapsack problem satisfies the greedy choice property and can be solved by making a greedy choice at each step.
- The greedy selection is made on the item that gives the largest cost/weight
- After such an item is totally exhausted, the thief can select the next item with largest cost/weight.
- The process is repeated until the total weight of selected items becomes equal to W


item 1
10
$60
item 2
20
$100
item 3
30
$120
50
knapsack


30 $120
+
20 $100
= $220
20 $100
+
10 $60
= $160
30 $120
+
10 $60
= $180
20
30 $80
+
20 $100
+
10 $60
= $240

0-1 Knapsack

- This problem does not obey the greedy choice property
- The 0-1 Knapsack problem is solved using dynamic programming
- The recursive equation is written by assuming that an item ‘k’ is an optimal choice for the knapsack then we are left with one subproblem in which select one item from remaining (n-1) items and add the cost of the item to the total cost of knapsack
- Since we do not know the value of k, this process is repeated for all values of k

$$C_n = \max_{k \in n} \{c_k + C_{n-1}\}$$

Huffman Codes

- Used for data compression
- Causes around 20% to 90% saving
- For encoding, a binary character code is used where each character is encoded with a sequence of unique binary numbers
- Either a fixed length or variable length code can be used
- Variable-length codes are much more efficient than fixed-length codes
- Huffman codes is also a variable length code
- The idea is to use smaller codes for the characters that occur more frequently in data while longer codes are used for less frequent characters

	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

Total number of bits required for encoding:

Case 1: Fixed-length codeword = 1,00,000 x 3 = 3,00,000 bits
Case 2: Variable-length codeword = 45x1 + 13x3 + 12x3 +16x3 +9x4 + 5x4 = 2,24,000 bits

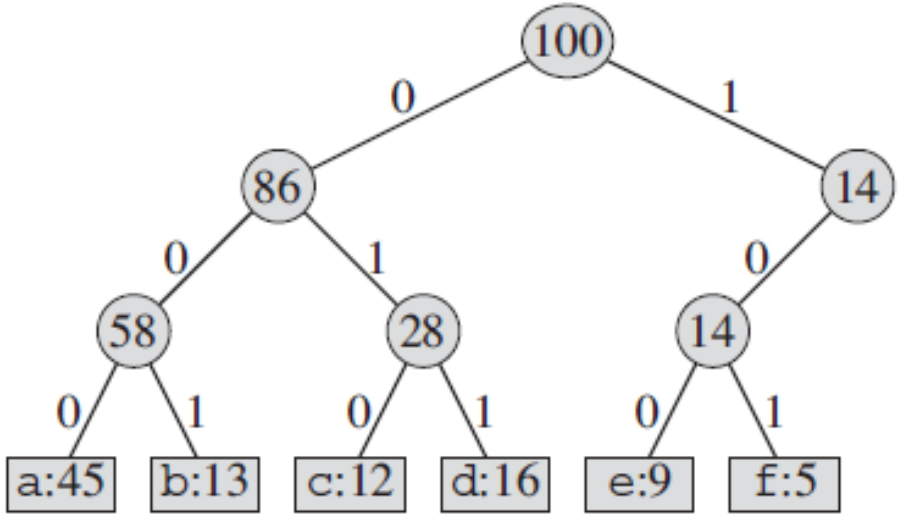
Total savings = (76,000/3,00,000)x100 ≈ 25%

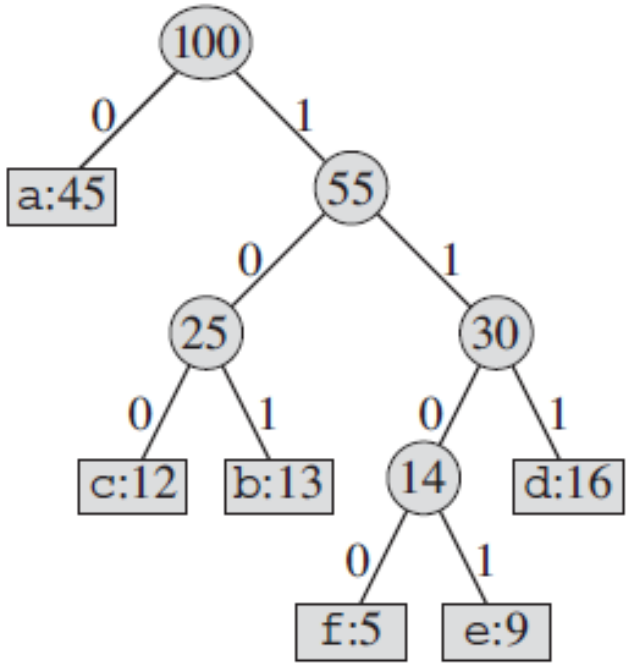
Prefix Codes

- Prefix codes are the ones in which no code is a prefix of other
- Prefix codes always yield optimal data compression
- The document is encoded by concatenating the prefix codes of the characters
- Prefix codes also simplify decoding, since no code is a prefix of other, the decoding is unambiguous
- For decoding, we identify the initial codeword, translate it back to the original character and continue with the next codeword

Decoding Process

- To simplify the decoding process, it is convenient to represent the codes so that the initial codeword can be picked easily
- The codes are usually represented as the nodes of a binary tree
- The leaves of the binary tree represent the characters
- The code for a character is then defined as a simple path from the root to the leaf
- An optimal code for a file is always represented in the form of a full binary tree (i.e. every non-leaf node has exactly two children)
- While traversing the tree, a ‘0’ means “go to left” and ‘1’ means “go to right”





Fixed-length code => not optimal

Variable-length code => full binary tree => optimal

- For any file having C alphabets, let the frequency of each character c in the file be c.freq and $d_T(c)$ the depth of c's leaf in the tree (thus $d_T(c)$ also denotes the number of bits required to code c)
- Then, the number of bits required to encode the complete file is:

$$B(T) = \sum_{c \in C} c.freq \times d_T(c)$$

Above is defined as the cost of the tree

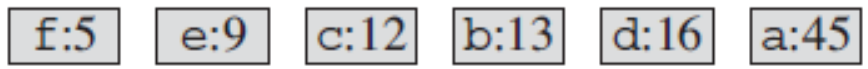
Huffman Code

- Huffman invented a greedy algorithm for generating optimal prefix code also known as Huffman code
- For any optimal prefix code on alphabet C, the full binary tree representation has exactly |C| leaf nodes and exactly |C|-1 internal nodes
- The Huffman coding is based on identifying the characters with smallest frequency and merging them to form an internal node of the tree
- The process is repeated until all the characters are merged into a single tree

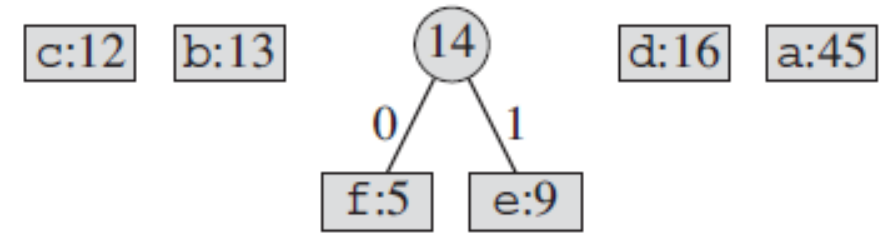
Huffman Code - Recursive Algorithm

```
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)      // return the root of the tree
```

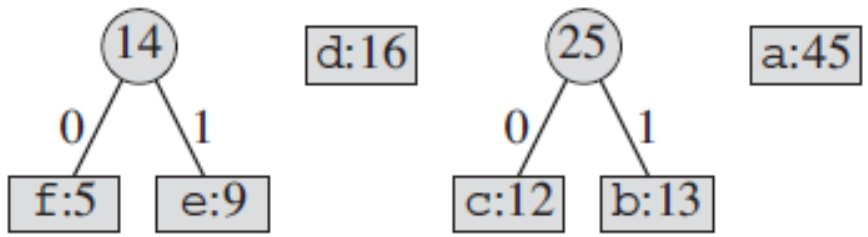
(a)


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

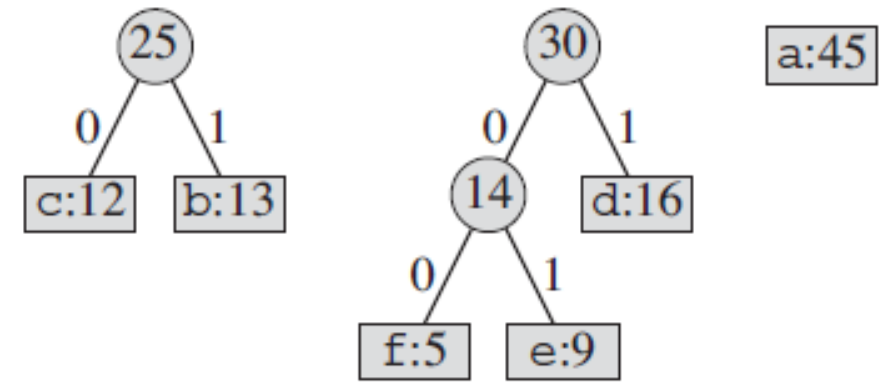
(b)


c:12
b:13
14
0
1
f:5
e:9
d:16
a:45

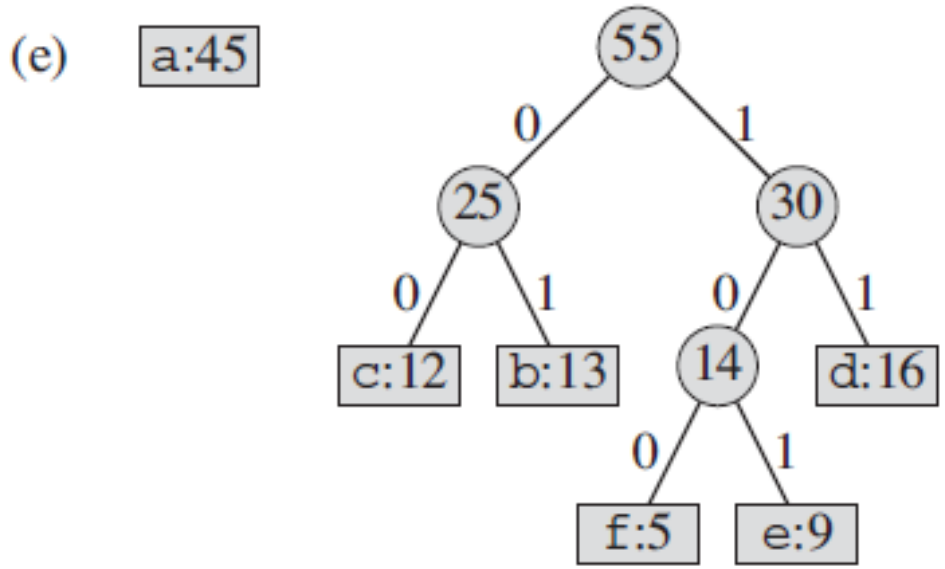
(c)


14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45

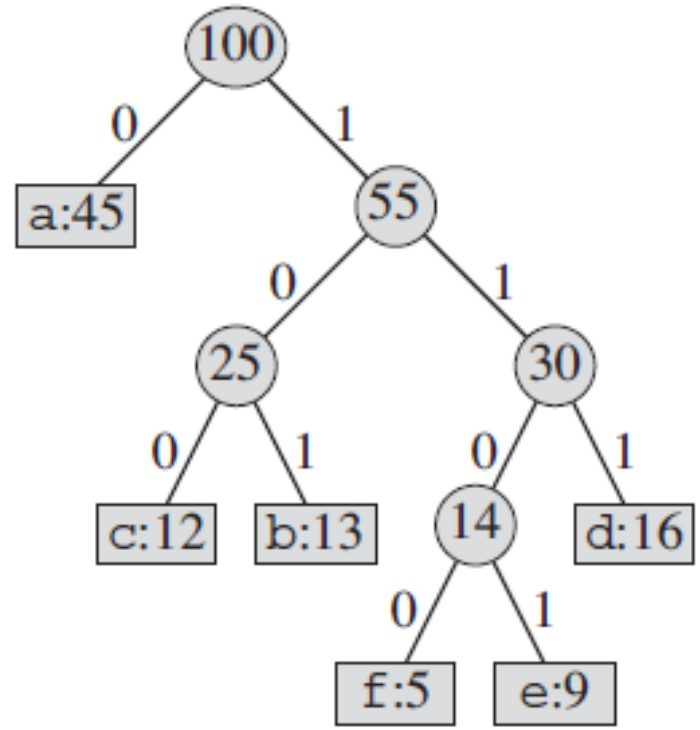
(d)


25
0
1
c:12
b:13
30
0
1
14
d:16
0
1
f:5
e:9
a:45

(e)


a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

(f)


100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9