UNIT-III

Change-making problem

Consider the general instance of thefollowing well-known problem. Give change for amount n using the minimum number of coins of denominations $d1 < d2 < \ldots < dm$. For the coin denominations used in the United States, as for those used in most if not all other countries, there is a very simple and efficient algorithm discussed in the next chapter. Here, we consider a dynamic programming algorithm for the general case, assuming availability of unlimited quantities of coins for each of the m denominations
$d1 < d2 < \ldots < dm$ where $d1 = 1$.
Let $F(n)$ be the minimum number of coins whose values add up to n; it is convenient to define $F(0) = 0$. The amount n can only be obtained by adding one coin of denomination dj to the amount $n - dj$ for $j = 1, 2, \ldots, m$ such that $n \geq dj$. Therefore, we can consider all such denominations and select the one minimizing $F(n - dj) + 1$. Since 1 is a constant, we can, of course, find the smallest $F(n - dj)$ first and then add 1 to it. Hence, we have the following recurrence for $F(n)$:

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

We can compute $F(n)$ by filling a one-row table left to right in the manner similar to the way it was done above for the coin-row problem, but computing a table entry here requires finding the minimum of up to m numbers.

ALGORITHM *ChangeMaking(D[1..m], n)*
//Applies dynamic programming to find the minimum number of coins
//of denominations $d1 < d2 < \ldots < dm$ where $d1 = 1$ that add up to a
//given amount n
//Input: Positive integer n and array $D[1..m]$ of increasing positive
// integers indicating the coin denominations where $D[1] = 1$
//Output: The minimum number of coins that add up to n

$F[0] \leftarrow 0$

for $i \leftarrow 1$ to n do

$temp \leftarrow \infty$; $j \leftarrow 1$

while $j \leq m$ and $i \geq D[j]$ do

$temp \leftarrow \min(F[i - D[j]], temp)$

$j \leftarrow j + 1$

$F[i] \leftarrow temp + 1$

return $F[n]$

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3-1], F[3-3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4-1], F[4-3], F[4-4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5-1], F[5-3], F[5-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6-1], F[6-3], F[6-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

FIGURE 8.2 Application of Algorithm *MinCoinChange* to amount $n = 6$ and coin denominations 1, 3, and 4.

Explain Knapsack problem using Dynamic Programming

The problem

- Find the most valuable subset of the given n items that fit into a knapsack of capacity W.

Consider the following sub problem P(i, j)

- Find the most valuable subset of the first i items that fit into a knapsack of capacity j, where 1 □ i □ n, and 1□ j □ W
- Let V[i, j] be the value of an optimal solution to the above subproblem P(i, j). Goal: V[n, W]

Thus, the value of an optimal solution among all feasible subsets of the first I items is the maximum of these two values. Of course, if the ith item does not fit into the knapsack, the value of an optimal subset selected from the first i items is the same as the value of an optimal subset selected from the first $i - 1$ items.
These observations lead to the following recurrence:

$$F(i, j) = \begin{cases} \max\{F(i-1, j),\ v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases} \quad \textbf{(8.6)}$$

It is convenient to define the initial conditions as follows:

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0. \quad \textbf{(8.7)}$$

Figure 8.4 illustrates the values involved in equations (8.6) and (8.7). For $i, j > 0$, to compute the entry in the ith row and the jth column, $F(i, j)$, we compute the maximum of the entry in the previous row and the same column and the sum of v_i and the entry in the previous row and w_i columns to the left. The table can be filled either row by row or column by column.

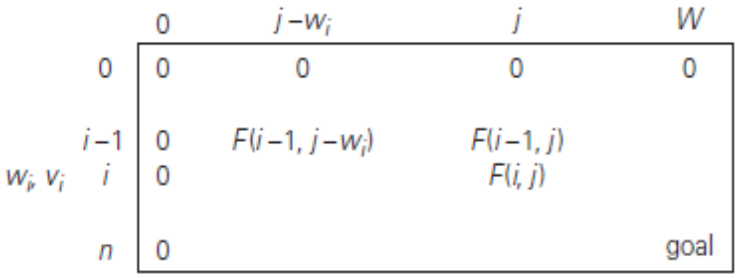


FIGURE 8.4 Table for solving the knapsack problem by dynamic programming.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

FIGURE 8.5 Example of solving an instance of the knapsack problem by the dynamic programming algorithm.

EXAMPLE 1 Let us consider the instance given by the following data:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$.

The dynamic programming table, filled by applying formulas (8.6) and (8.7), is shown in Figure 8.5.

Thus, the maximal value is $F(4, 5) = \$37$. We can find the composition of an optimal subset by backtracing the computations of this entry in the table. Since $F(4, 5) > F(3, 5)$, item 4 has to be included in an optimal solution along with an optimal subset for filling $5 - 2 = 3$ remaining units of the knapsack capacity. The value of the latter is $F(3, 3)$. Since $F(3, 3) = F(2, 3)$, item 3 need not be in an optimal subset. Since $F(2, 3) > F(1, 3)$, item 2 is a part of an optimal selection, which leaves element $F(1, 3 - 1)$ to specify its remaining composition. Similarly, since $F(1, 2) > F(0, 2)$, item 1 is the final part of the optimal solution {item 1, item 2, item 4}. ■

Explain Memory Function algorithm for the Knapsack problem

- The Knapsack Problem and Memory Functions
  - The Recurrence
    a. Two possibilities for the most valuable subset for the subproblem P(i, j)
       i. It does not include the ith item: $V[i, j] = V[i-1, j]$
       ii. It includes the ith item: $V[i, j] = v_i + V[i-1, j - w_i]$

$$V[i, j] = \begin{cases} \max\{V[i-1, j],\ v_i + V[i-1, j - w_i]\} & \text{if } j - w_i \geq 0 \\ V[i-1, j] & \text{if } j - w_i < 0 \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$

- Memory functions:
  - Memory functions: a combination of the top-down and bottom-up method. The idea is to solve the subproblems that are necessary and do it only once.
  - Top-down: solve common subproblems more than once.
  - Bottom-up: Solve subproblems whose solution are not necessary for the solving the original problem.

ALGORITHM MFKnapsack(i, j)

```
if V[i, j] < 0 //if subproblem P(i, j) hasn't been solved yet.
    if j < Weights[i]
        value ← MFKnapsack(i – 1, j)
    else
        value ← max(MFKnapsack(i – 1, j),
                    values[I] + MFKnapsck( i – 1, j – Weights[i]))
    V[i, j] ← value
return V[i, j]
```

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3, v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2, v_4 = 15$	4	0	—	—	—	—	**37**

FIGURE 8.6 Example of solving an instance of the knapsack problem by the memory function algorithm.

Write short notes on optimal binary search tree. Or Write an algorithm to construct the optimal binary search tree given the roots r(i,j), 0<=i<=j<=n. Also prove that this could be performed in time O(n)

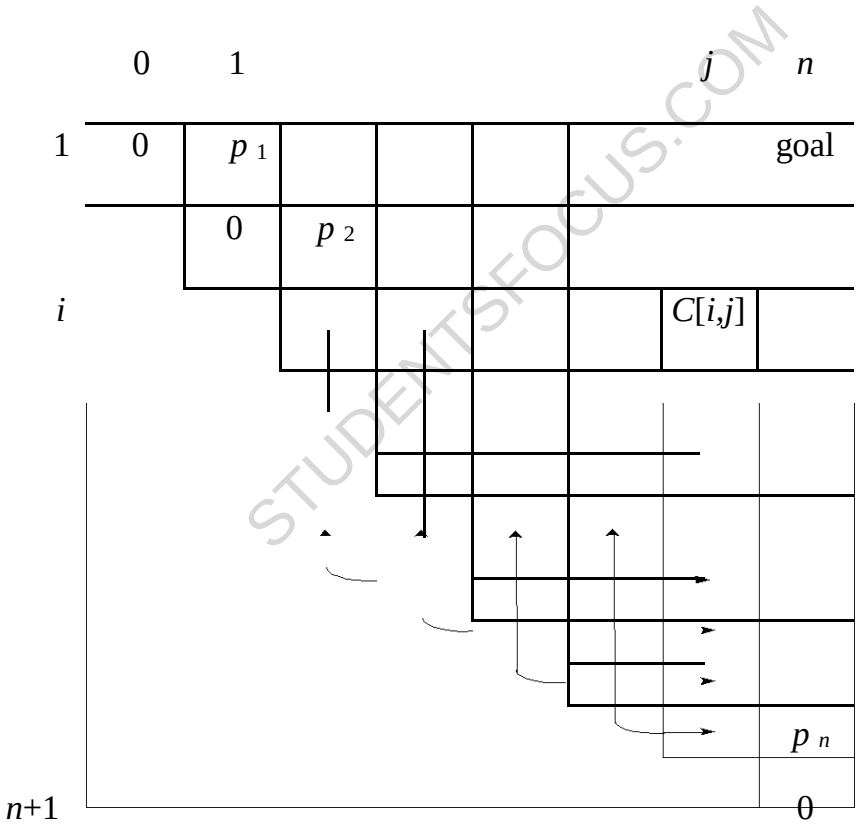
Let C[i,j] be minimum average number of comparisons made in T[i,j], optimal BST for keys ai < …< aj

, where $1 \le i \le j \le n$. Consider optimal BST among all BSTs with some ak ($i \le k \le j$) as their root; T[i,j] is the best among them.

ALGORITHM Optimal BST(P [1..n])

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//optimal BST and table R of subtrees' roots in the
optimal BST for i ←1 to n do
C[i, i −
1]←0 C[i,
i]←P[i] R[i,
i]←i
C[n + 1, n]←0
```

for d ←1 to n − 1 do //diagonal
count for i ←1 to n − d do
j ←i + d
minval←∞
for k←i to j
do
if C[i, k − 1]+ C[k + 1, j]< minval
minval←C[i, k − 1]+ C[k + 1, j];
kmin←k R[i, j]←kmin
sum←P[i]; for s ←i + 1 to j do
sum←sum + P[s] C[i, j]←minval + sum
return C[1, n], R



Running time of this algorithm: Time Efficiency Θ(n2) and Space Efficiency : Θ(n3) For Example,

EXAMPLE: Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The initial tables are:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Let us compute $C(1, 2)$:

$$C(1, 2) = \min \left\{ \begin{array}{ll} k = 1: & C(1, 0) + C(2, 2) + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k = 2: & C(1, 1) + C(3, 2) + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{array} \right\}$$
$$= 0.4.$$

Thus, out of two possible binary trees containing the first two keys, A and B, the root of the optimal tree has index 2 (i.e., it contains B), and the average number of comparisons in a successful search in this tree is 0.4.

We arrive at the following final tables:

main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

Thus, the average number of key comparisons in the optimal tree is equal to 1.7. Since R(1, 4) = 3, the root of the optimal tree contains the third key, i.e., C. Its left subtree is made up of keys A and B, and its right subtree contains just key D. To find the specific structure of these subtrees, we find first their roots by consulting the root table again as follows. Since R(1, 2) = 2, the root of the optimal tree containing A and B is B, with A being its left child (and the root of the one node tree: R(1, 1) = 1). Since R(4, 4) = 4, the root of this one-node optimal tree is its only key D. Figure 3.10 presents the optimal tree in its entirety.

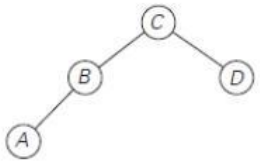


FIGURE 3.10 Optimal binary search tree for the above example.

MINIMUM SPANNING TREE

A spanning tree of an undirected connected graph is its connected acyclic subgraph [i.e., a tree] that contains all the vertices of the graph. If such a graph has

weights assigned to its edges, a minimum spanning tree is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges.

The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

PRIM'S ALGORITHM

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding sub trees. The initial sub tree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree.

ALGORITHM Prim[G]

//Prim's algorithm for constructing a minimum

spanning tree //Input: A weighted connected graph

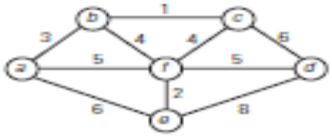
G = V, E

//Output: ET , the set of edges composing a minimum
spanning tree of G VT←{v0} //the set of tree vertices can be
initialized with any vertex ET←∅

for i ←1 to |V| − 1 do

find a minimum-weight edge e∗ = [v∗ , u∗] among all
the edges [v, u] such that v is in VT and u is in V − VT
VT←VT∪ {u∗ }
ET←ET∪ {e∗ }

return ET



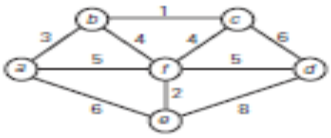
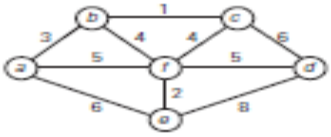
Tree vertices	Remaining vertices	Illustration
a(–, –)	**b(a, 3)** c(–, ∞) d(–, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(–, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

FIGURE 9.3 Application of Prim's algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.

FLOYD'S ALGORITHM

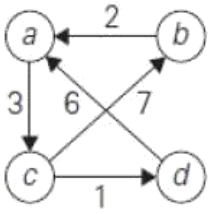
Given a weighted connected graph (undirected or directed), the all-pairs shortest paths problem asks to find the distances—i.e., the lengths of the shortest paths— from each vertex to all other vertices.

ALGORITHM Floyd(W[1..n, 1..n])

//Implements Floyd's algorithm for the all-pairs shortest-paths problem //Input: The weight matrix W of a graph with no negative-length cycle //Output: The distance matrix of the shortest paths' lengths

D ←W //is not necessary if W can be overwritten
for k←1 to n do
for i ←1 to n do
for j ←1 to n do
D[i, j]←min{D[i, j], D[i, k]+ D[k, j]}
return D

Efficiency of Floyd's Algorithm: Time efficiency Θ(n3) and Space Efficiency is Θ(n2)
For Example,



$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., **a** and *b* (note a new shortest path from *c* to *a*).

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., **a**, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., **a**, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

FIGURE 3.5 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

Write the procedure to compute Huffman code.

Suppose we have to encode a text that comprises symbols from some n-symbol alphabet by assigning to each of the text‘s symbols some sequence of bits called the code word. For example, we can use a fixed-length encoding that assigns to each symbol a bit string of the same length m
($m \geq \log_2 n$).

Step 1 Initialize n one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree‘s root to indicate the tree‘s **weight**. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree‘s leaves.)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight. Make them the left and right sub tree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called a **Huffman tree**. It defines—in the mannerdescribed above—a **Huffman code**.

EXAMPLE Consider the five-symbol alphabet {A, B, C, D, _} with the following occurre frequencies in a text made up of these symbols:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for this input is shown in Figure 3.18

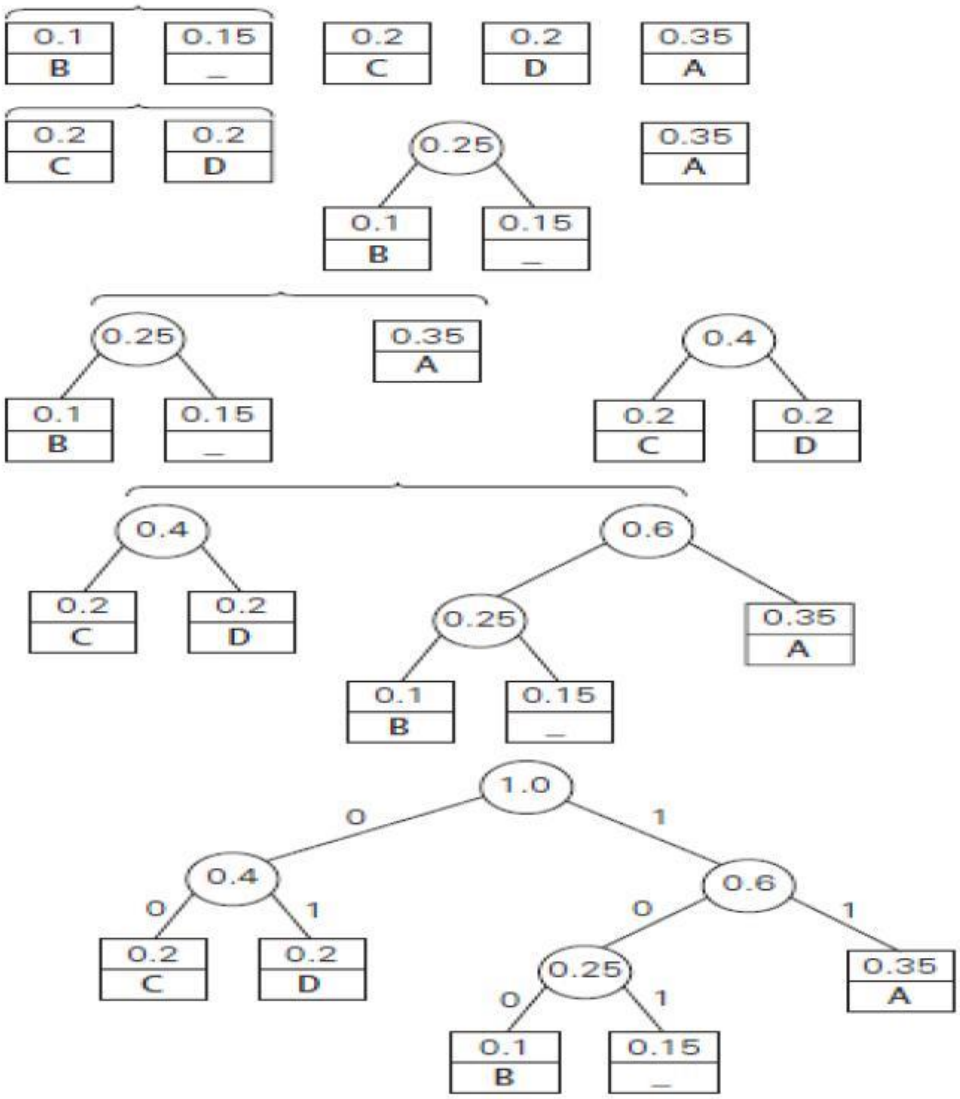


FIGURE 3.18 Example of constructing a Huffman coding tree.