Module 4

8. Dynamic Programming

- Dynamic Programming is a technique for solving problems with overlapping subproblems. (optimal solutions)
- Typically, these subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems.
- Rather than solving overlapping subproblems again and again dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained. (Memoization)
- Dynamic Programming is used to solve optimization problems (to find out optimize answer, i.e to find of maximum answer (Eg. max profit) or minimum answer (Eg. min cost)) and overlapping problems.
- Optimal sub structure & Overlapping sub problems.
- Dynamic Programming always gives the optimal answer.

Eg: Fibonacci Numbers : 0, 1, 1, 2, 3, 5, 8, 13, ...

recurrence relation is

$$f(n) = f(n-1) + f(n-2) \quad \text{for } n > 1$$

$$f(n) = 0 \quad \text{if } n = 0$$

$$f(n) = 1 \quad \text{if } n = 1$$

Dynamic Programming techniques

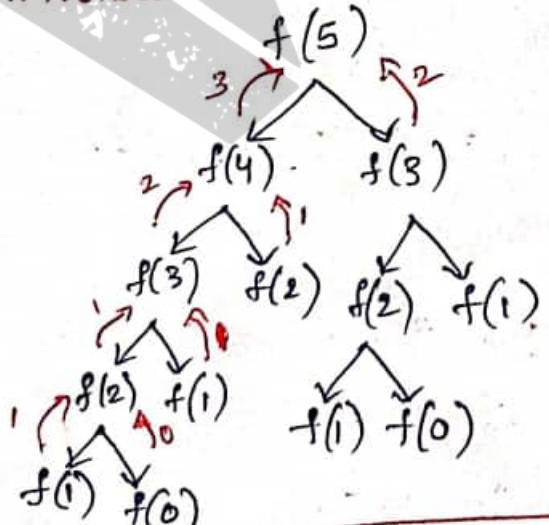
Memoization

- Top down approach
- Recursive implementation
- Caches the results of function calls
- well-suited for small set of i/p
- Used when subproblems have overlapping subproblems

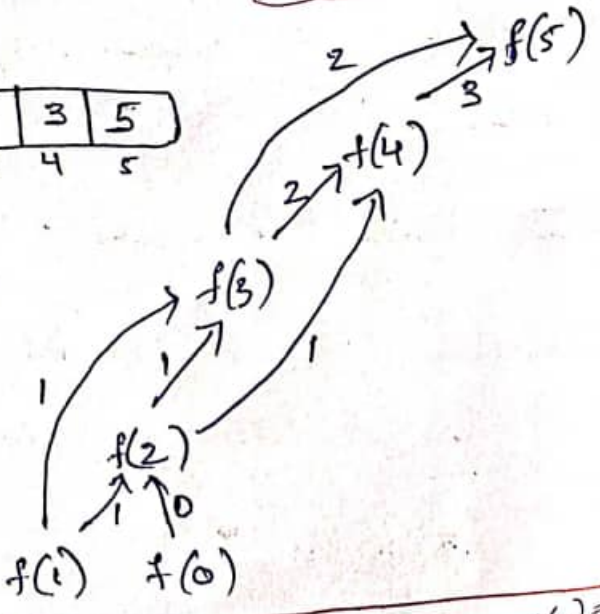
Tabulation

- Bottom up approach
- Iterative implementation (loop)
- Stores the results of subproblems in a table
- Well-suited for large set of inputs
- used when sub problems do not overlap.

Eg: Fibonacci

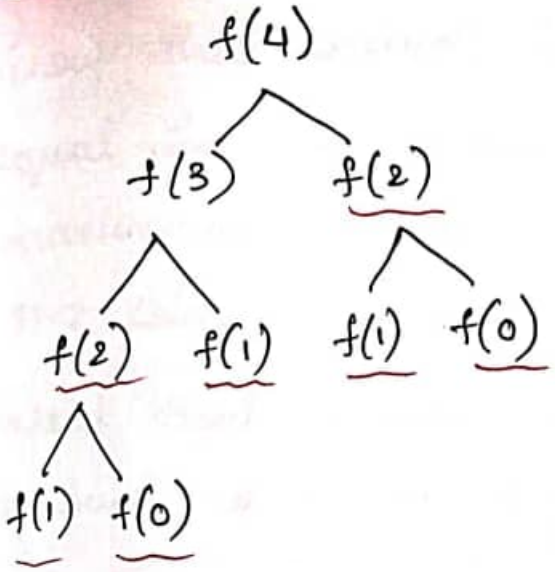


0	1	1	2	3	5
0	1	2	3	4	5



$f(n) = f(n-1) + f(n-2)$ $n>1$, $f(0)=0$, $f(1)=1$

$f(i) = f(i-1) + f(i-2)$, $f(0)=0$, $f(1)=1$



f(2), f(1), f(0) are repeating / overlapping.

- When the sub problems repeat, don't solve it again & again, solve it once & store it in a table (memoization), use the answer from the table when the subproblem repeats.

There are 9 f(n) function calls, if we use dynamic programming approach, we have to solve only 3 f(n) function calls & use them again & again.

8.1 Three Basic Examples (using Dynamic programming)

Example 1 : Coin-row Problem

There is a row of n coins whose values are some positive integers $c_1, c_2, \ldots c_n$, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

- Let F(n) be the maximum amount that can be picked up from the row of n coins.

$c_1, c_2, \ldots, c_{n-2}, c_{n-1}, c_n$

To derive recurrence for F(n), partition all the allowed coin selections into 2 groups:

1) those that include last coin
2) those that doesn't include last coin

Eg: 5 1 9 10 9 2

Greedy strategy: Pick up 10, discard 2 9's
Pick up 5, discard 1
Pick up 2.
10 + 5 + 2 = 17

Dynamic Programming:

$F(n) = \max\{ c[n] + F(n-2), F(n-1) \}$ for $n \geq 1$

$F(0) = 0$

$F(1) = c(1)$

- Group that include last coin, the largest amount is equal to $C_n + F(n-2)$ → the value of the n^{th} coin plus the maximum amount that can be picked up from the first $n-2$ coins.
- Group that doesn't include last coin, the maximum amount is equal to $F(n-1)$.

$$\therefore F(n) = \max\{C_n + F(n-2),\ F(n-1)\} \quad \text{for } n > 1$$
$$F(0) = 0 \qquad \text{// no coins}$$
$$F(1) = C_1 \qquad \text{// only one coin}$$

We can compute $F(n)$ by filling the one-row table left-to right.

Algorithm CoinRow($C[1...n]$)

// Applies formula of $F(n)$ bottom up to find the maximum amount of money that can be picked up from a coin row without picking two adjacent coins

// Input : Array $C[1..n]$ of positive integers indicating the coin values

// Output : The maximum amount of money that can be picked up

```
F[0] ← 0
F[1] ← C[1]
for i ← 2 to n do
    F[i] ← max(C[i] + F[i-2], F[i-1])
return F[n]
```

Eg: Working : Solve the coin row problem by dynamic programming for the coin row 5, 1, 2, 10, 6, 2

$F[0] = 0$

$F[1] = C_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

C_1

$F[2] = \max\{C[2] + F[0], F[1]\}$

$F[2] = \max\{1+0, 5\}$

$= 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

i

$F(1)$

$F[3] = \max\{C[3] + F[1], F[2]\}$

$F[3] = \max\{2+5, 5\}$

$= 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

i

$F[4] = \max\{C[4] + F[2], F[3]\}$

$F[4] = \max\{10+5, 7\}$

$= 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

i

$C_4, F(2)$

$F[5] = \max\{C[5] + F[3], F[4]\}$

$F[5] = \max\{6+7, 15\}$

$= 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

i

$F[6] = \max\{C[6] + F[4], F[5]\}$

$F[6] = \max\{2+15, 15\}$

$= 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	17

i

$C_6, F(4)$

⇑

Backtracking

$F[i] = \max\{C[i] + F[i-2], F[i-1]\}$

Thus, the optimal solution is $\{C_1, C_4, C_6\}$

When we backtrack from bottom up.

To compute $F(6)$, we used c_6 & $F(4)$

To compute $F(4)$, we used c_4 & $F(2)$

To compute $F(2)$, we used $F(1)$

To compute $F(1)$, we used c_1

Thus, optimal solution is $\{c_1, c_4, c_6\}$

Time complexity = $\Theta(n)$ Space complexity = $\Theta(n)$

Fewest coins to make change

Example 2: Change making problem (coin change)

Consider the general instance of the problem, Give change for amount n using the minimum number of coins of denominations $d_1 < d_2 < \ldots < d_m$, assuming availability of unlimited quantities of coins for each of the m denominations $d_1 < d_2 < \ldots < d_m$ where $d_1 = 1$, we consider a dynamic programming algorithm for general case.

- Let $F(n)$ be the minimum number of coins whose values add up to n : $F(0) = 0$
- The amount n can only be obtained by adding one coin of denomination d_j to the amount $n - d_j$ for $j = 1, 2, \ldots m$ such that $n \geq d_j$.
- ∴ We can consider all such denominations and select the one minimizing $F(n - d_j) + 1$.

Since 1 is a constant, find the smallest $F(n - d_j)$ first and then add 1 to it.

1) Amount : 5

coins : 1, 2, 5

5

2 + 2 + 1

2 + 1 + 1 + 1

1 + 1 + 1 + 1 + 1

2) Amount : 11

coins : 1, 5, 6, 9.

Greedy : choose 9, 11 − 9 = 2

choose 1, 2 − 1 = 1

choose 1, 1 − 1 = 0

(9, 1, 1) 3 coins

Dynamic : choose 6 & 5 = 11

3) Amount : 41

coins : 4, 10, 25

Greedy : 25, 41 − 25 = 16

10, 16 − 10 = 6

4, 6 − 4 = 2

− no solution

DP: (25, 4, 4, 4, 4

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 one-row table left to right, but computing a table entry 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 $d_1 < d_2 < \cdots < d_m$ where $d_1 = 1$ that add upto 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 numbe of coins that add up to n

```
F[0] ← 0
for i ← 1 to n do
    temp ← ∞
    j ← 1
    while j ≤ m and i ≥ D[j] do
        temp ← min(F[i - D[j]], temp)
        j ← j + 1
    F[i] ← temp + 1
return F[n]
```

Eg: Working: Minimum Coin Change to amount $n = 6$ and coin denominations 1, 3, 4

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

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

$F[0]=0$

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

$= 1$

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

$F[1]=1$

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

$= 2$

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

$F[2]=2 \quad F[0]=0$

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

$= 1$

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

$F[3], \quad F[1], \quad F[0]$

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

$= 1$

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

$F[4] \quad F[2] \quad F[1]$

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

$= 2$

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

$F[5] \quad F[3] \quad F[2]$

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

$= 2$

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

To find the coins of an optimal solution, we need to back-trace the computations to see which of the denominations produced the minima in formula.

- for $n = 6$, the minimum was produced by $d_2 = 3$. the second minimum (for $n = 6 - 3$) was also produced for a coin of that denomination.
- Thus, the minimum - coin set for $n=6$ is two 3's (3+3=6)

Time complexity = $O(nm)$ Space complexity = $\Theta(n)$

Example 3 : Coin - Collecting Problem

Several coins are placed in cells of an $n \times m$ board, no more than one coin per cell. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current location. When the robot visits a cell with a coin, it always picks up that coin.

Design an algorithm to find the maximum number of coins the robot can collect and a path it needs to follow to do this.

- Let $F(i, j)$ be the largest number of coins the robot can collect and bring to the cell (i, j) in the ith row & jth column of the board. It can reach this cell either from adjacent cell $(i-1, j)$ above it or from the adjacent cell $(i, j-1)$ to the left of it.

The largest numbers of coins that can be brought to these cells are $F(i-1, j)$ and $F(i, j-1)$ respectively. Of course, there are no adjacent cells to the left of first column & above the first row. For such cells, we assume there are 0 neighbors.

Hence, the largest number of coins the robot can bring to cell (i, j) is the maximum of the two numbers $F(i-1, j)$ and $F(i, j-1)$, plus the one possible coin at cell (i, j) itself.

Recurrence :

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \le i \le n, \ 1 \le j \le m$$

$$F(0, j) = 0 \quad \text{for } 1 \le j \le m$$

$$F(i, 0) = 0 \quad \text{for } 1 \le i \le n$$

where $c_{ij} = 1$ if there is a coin in cell (i, j) & $c_{ij} = 0$ otherwise.

Fill in the $n \times m$ table of $F(i, j)$ values either row by row or column by column.

Algorithm RobotCoinCollection ($C[1..n, 1..m]$)

// Applies dynamic programming to compute the largest number of coins a robot can collect on an $n \times m$ board by starting at $(1, 1)$ and moving right & down from upper left to down right corner.

// Input : Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0 for cells with and without a coin, respectively.

// Output : Largest number of coins the robot can bring to cell $[n, m]$

```
F[1,1] ← c[1,1];
for j ← 2 to m do
    F[1,j] ← F[1,j-1] + c[1,j]
for i ← 2 to n do
    F[i,1] ← F[i-1,1] + [i,1]
    for j ← 2 to m do
        F[i,j] ← max(F[i-1,j], F[i,j-1]) + c[i,j]
return F[n,m]
```

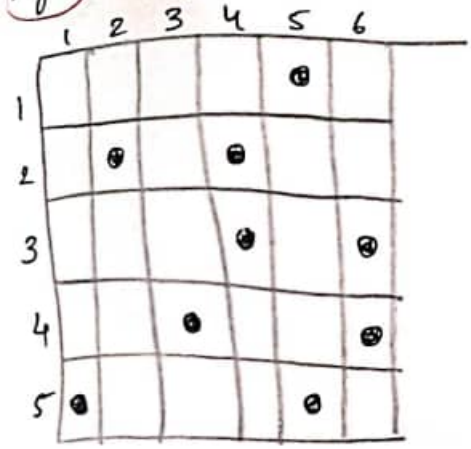
Tracing back the optimal path:

It is possible to trace the computations backwards to get an optimal path.

- If $F(i-1, j) > F(i, j-1)$, an optimal path to cell (i,j) must come down from the adjacent cell above it, (top to down);
- If $F(i-1, j) < F(i, j-1)$, an optimal path to cell (i,j) must come from the adjacent cell on the left;
- If $F(i-1, j) = F(i, j-1)$, it can reach cell (i,j) from either direction. Ties can be ignored by giving preference to coming from the adjacent cell above.
- If there is only one choice, i.e., either $F(i-1, j)$ or $F(i, j-1)$ are not available, use the other available choice.
- The optimal path can be obtained in $\Theta(n+m)$ time

Time complexity: $\Theta(nm)$ Space complexity: $\Theta(nm)$

Eg: 1



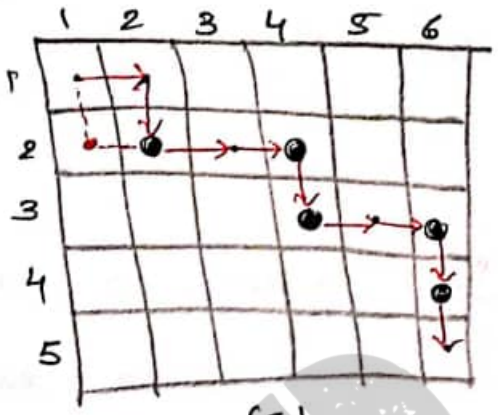
(a)

Coins to collect

	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	5

(b)

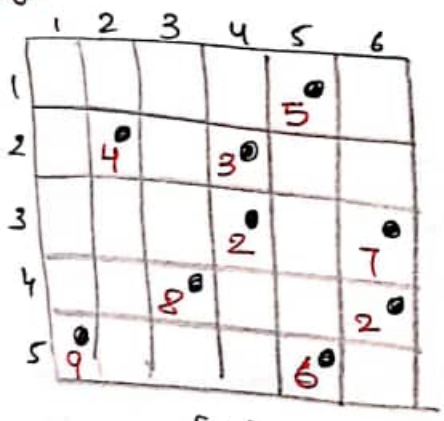
DP. algorithm results



(c)

2 paths to collect 5 coins, the max no of coins possible

Eg: 2



(a)

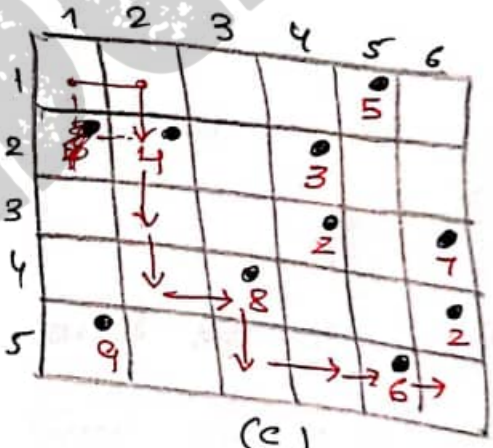
Coins to collect

	1	2	3	4	5	6
1	0	0	0	0	5	5
2	0	4	4	7	7	7
3	0	4	4	9	9	16
4	0	4	12	12	12	18
5	9	9	12	12	18	18

(b)

DP algorithm results

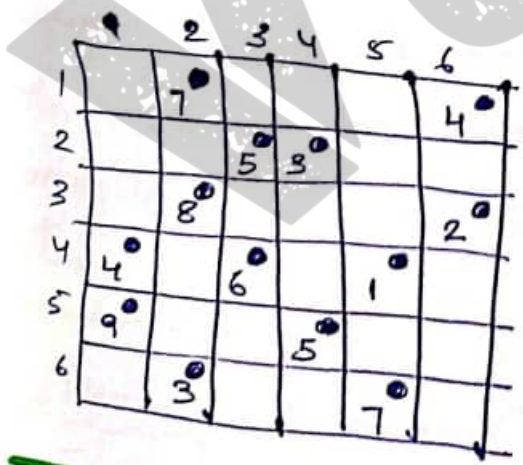
Coins (4, 8, 6)



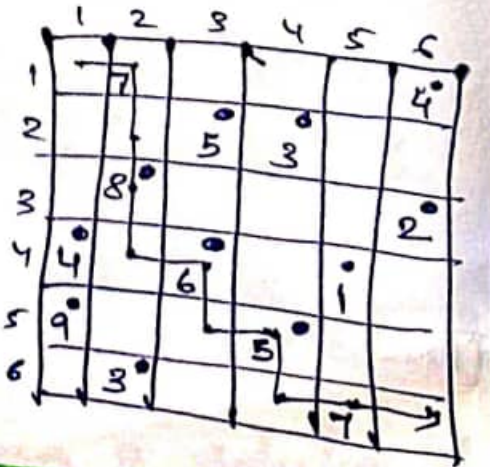
(c)

2 paths to collect 3 coins, max value of the coins possible

Eg: 3



	1	2	3	4	5	6
1	0	7	7	7	7	11
2	0	7	12	15	15	15
3	0	15	15	15	15	17
4	4	15	21	21	22	22
5	13	15	21	26	26	26
6	13	18	21	26	33	33



8.2 The Knapsack Problem and Memory Functions

Knapsack Problem Variants

- 0/1 Knapsack Problem (Items are indivisible)
  - top-down & bottom-up with memory functions
  - bottom-up without memory functions (dynamic Programming approach)
- Fractional Knapsack problem (Items are divisible) (greedy approach)

Given n items of known weights $w_1, \ldots w_n$ and values $v_1, \ldots v_n$ and a knapsack of capacity W (total weight). Find the most valuable subset of the items that fit into the knapsack available. (which items to place in the knapsack such that the weight limit is not exceeded and the total value of the items is as large as possible). We assume all the weights & the knapsack capacity are positive integers, the item values do not have to be integers.

---

1) Dynamic Programming approach (without memory functions)

Derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances.

Let us consider an instance defined by the first i items, $1 \le i \le n$, with weights $w_1, \ldots w_i$, values $v_1, \ldots v_i$, and knapsack capacity j, $1 \le j \le W$.

Let $F(i, j)$ be the value of an optimal solution to this instance i.e. the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.

We can divide all the subsets of the first i items that fit the knapsack capacity j into 2 categories:

1. The subsets that do not include the ith item, the value of an optimal subset is, $F(i-1, j)$.

2. The subsets that do include the ith item (hence, $j - w_i \ge 0$), an optimal subset is made up of this item & an optimal subset of the first $i-1$ items that fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i-1, j - w_i)$.

Thus, the value of an optimal solution among all feasible subsets of the 1st i items is the maximum of these 2 values.

Recurrence: (item not included / item included)

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

← item weighs more than knapsack weight

$F(0, j) = 0$ for $j \ge 0$ ← no items

$F(i, 0) = 0$ for $i \ge 0$ ← no knapsack

goal is to find $F(n, W)$, the maximum value of a subset of the n given items that fit into the knapsack of capacity W, and an optimal subset itself.

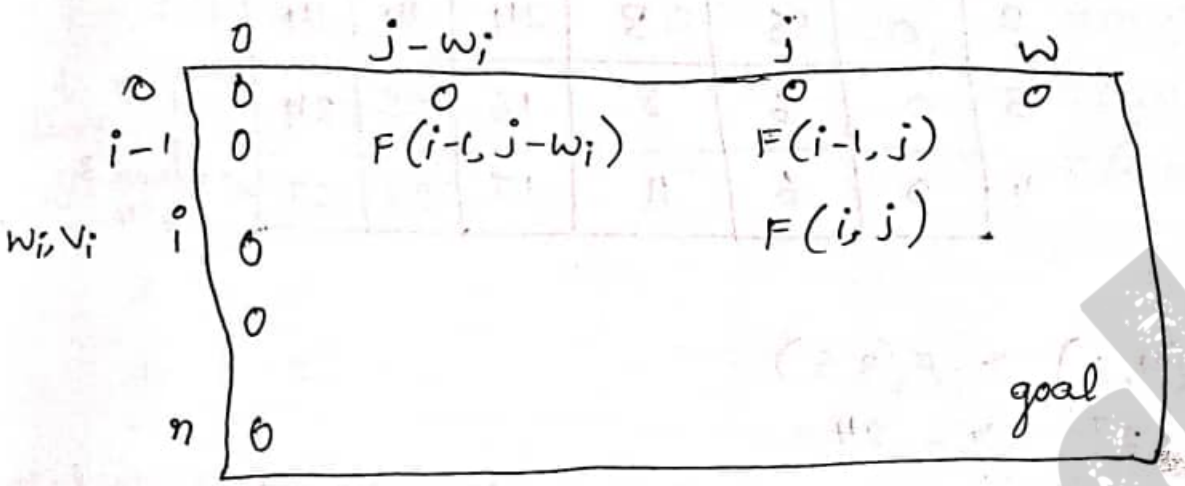


Fig: Table for solving the knapsack problem by dynamic programming

For $i, j > 0$, to compute the entry in the ith row and jth column, $F(i, j)$.

We compute the maximum of the entry in the previous row & the same column & the sum of v_i and the entry in the previous row & w_i columns to the left.

The table can be filled either row by row or column by column.

ex: 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

Knapsack capacity $W = 5$

We start table with (n+1) rows & (W+1) columns.

Fill 0th row & 0th col with zero (∵ F(0,j) = 0 , F(i,0) = 0)

j → weight (Wi)

i ↓ items	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

v(i)

F(0,0), F(0,1), F(0,2)
F(0,3), F(0,4), F(0,5)
F(1,0), F(2,0), F(3,0)
F(4,0) = 0

The start completing the table row wise from left to right by using the formulas:

$F(\overset{i}{1},\overset{j}{1})$:

i = 1
j = 1

$V_i = V_1 = 12$
$W_i = W_1 = 2$

$j - W_i = 1 - 2$
$= -1$

∴ $j - W_i < 0$ ∴(F(i-1, j))

$F(1,1) = F(0,1) = 0$

$F(\overset{i}{1},\overset{j}{2})$:

i = 1 , j = 2

$V_i = V_1 = 12$
$W_i = W_1 = 2$

$j - W_i = 2 - 2$
$= 0$

∴ $j - W_i \geq 0$ F(i-1, j), $V_i + F(i-1, j-w_i)$

$F(1,2) = \max\{F(0,2), V_1 + F(0,0)\}$
$= \max\{0, 12+0\}$

$F(1,2) = \$\ 12$

F(1,3):

i = 1 , j = 3

$V_i = V_1 = 12$
$W_i = W_1 = 2$

$j - W_i = 3 - 2$
$= 1$

∴ $j - W_i \geq 0$

$F(1,3) = \max\{F(0,3), V_1 + F(0,1)\}$
$= \max\{0, 12+0\}$

$F(1,3) = 12$

F(1,4):

i = 1, j = 4

$V_i = V_1 = 12$
$W_i = W_1 = 2$

$j - W_i = 4 - 2$
$= 2$

∴ $j - W_i \geq 0$

$F(1,4) = \max\{F(0,4), V_1 + F(0,2)\}$
$= \max\{0, 12+0\}$

$F(1,4) = 12$

Ex: $W = 5$

item	W	V
1	2	8
2	1	6
3	3	16
4	2	11

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	8	8	8	8
2	0	6	8	14	14	14
3	0	6	8	16	22	24
4	0	6	11	17	22	27

← item 3 & 4

$F(4,5) > F(3,5)$

$27 > 24$

choose item 4, subtract its weight & item

$4-1 = 3, \quad 5-2 = 3$

$F(3,3) > F(2,3)$

$16 \quad 14$

choose item 3, subtract its weight & item

$3-1 = 2, \quad 3-3 = 0$

$F(2,0)$

• F(1,5):

$i=1,\ j=5$ $\quad j-w_i = 5-2 = 3$

$V_i = V_1 = 12$ $\quad \because j-w_i \geq 0$

$w_i = w_1 = 2$

$F(1,5) = \max\{F(0,5),\ \overset{V_1}{12} + F(0,3)\}$

$= \max\{0,\ 12+0\}$

$\boxed{F(1,5) = 12}$

• F(2,4):

$i=2,\ j=4$ $\quad j-w_i = 4-1 = 3$

$V_i = V_2 = 10$ $\quad \because j-w_i \geq 0$

$w_i = w_2 = 1$

$F(2,4) = \max\{F(1,4),\ V_2 + F(1,3)\}$

$= \max\{12,\ 10+12\}$

$\boxed{F(2,4) = 22}$

• F(2,1):

$i=2,\ j=1$ $\quad j-w_i = 1-1 = 0$

$V_i = V_2 = 10$ $\quad \because j-w_i \geq 0$

$w_i = w_2 = 1$

$F(2,1) = \max\{F(1,1),\ \overset{V_2}{10} + F(1,0)\}$

$= \max\{0,\ 10+0\}$

$\boxed{F(2,1) = 10}$

• F(2,5):

$i=2,\ j=5$ $\quad j-w_i = 5-1 = 4$

$V_i = V_2 = 10$ $\quad \because j-w_i \geq 0$

$w_i = w_2 = 1$

$F(2,5) = \max\{F(1,5),\ V_2 + F(1,4)\}$

$= \max\{12,\ 10+12\}$

$\boxed{F(2,5) = 22}$

• F(2,2):

$i=2,\ j=2$ $\quad j-w_i = 2-1 = 1$

$V_i = V_2 = 10$ $\quad \because j-w_i \geq 0$

$w_i = w_2 = 1$

$F(2,2) = \max\{F(1,2),\ \overset{V_2}{10} + F(1,1)\}$

$= \max\{12,\ 10+0\}$

$\boxed{F(2,2) = 12}$

• F(3,1):

$i=3,\ j=1$ $\quad j-w_i = 1-3 = -2$

$V_i = V_3 = 20$ $\quad \because j-w_i < 0$

$w_i = w_3 = 3$

$F(2,5) = F(2,1)$

$\boxed{F(2,5) = 10}$

• F(2,3):

$i=2,\ j=3$ $\quad j-w_i = 3-1 = 2$

$V_i = 10$ $\quad \because j-w_i \geq 0$

$w_i = 1$

$F(2,3) = \max\{F(1,3),\ V_2 + F(1,2)\}$

$= \max\{12,\ 10+12\}$

$\boxed{F(2,3) = 22}$

• F(3,2):

$i=3,\ j=2$ $\quad j-w_i = 2-3 = -1$

$V_i = V_3 = 20$ $\quad \because j-w_i < 0$

$V_i = W_3 = 3$

$F(3,2) = F(2,2)$

$\boxed{F(3,2) = 12}$

· F(3, 3):

$i = 3,\ j = 3$ $\quad j - w_i = 3 - 3 = 0$

$v_i = v_3 = 20$ $\quad \therefore j - w_i \geq 0$

$w_i = w_3 = 3$

$F(3,3) = \max\{F(2,3),\ v_3 + F(2,0)\}$

$= \max\{22,\ 20 + 0\}$

$F(3,3) = 22$

· F(3, 4):

$i = 3,\ j = 4$ $\quad j - w_i = 4 - 3 = 1$

$v_i = v_3 = 20$ $\quad \therefore j - w_i \geq 0$

$w_i = w_3 = 3$

$F(3,4) = \max\{F(2,4),\ v_3 + F(2,1)\}$

$= \max\{22,\ 20 + 10\}$

$F(3,4) = 30$

· F(3, 5):

$i = 3,\ j = 5$ $\quad j - w_i = 5 - 3 = 2$

$v_i = v_3 = 20$ $\quad \therefore j - w_i \geq 0$

$w_i = w_3 = 3$

$F(3,5) = \max\{F(2,5),\ v_3 + F(2,2)\}$

$= \max\{22,\ 20 + 12\}$

$F(3,5) = 32$

· F(4, 1):

$i = 4,\ j = 1$ $\quad j - w_i = 1 - 2 = -1$

$v_i = v_4 = 15$ $\quad \therefore j - w_i < 0$

$w_i = w_4 = 2$

$F(4,1) = F(3,1)$

$F(4,1) = 10$

· F(4, 2):

$i = 4,\ j = 2$ $\quad j - w_i = 2 - 2 = 0$

$v_i = v_4 = 15$ $\quad \therefore j - w_i \geq 0$

$w_i = w_4 = 2$

$F(4,2) = \max\{F(3,2),\ v_4 + F(3,0)\}$

$= \max\{12,\ 15 + 0\}$

$F(4,2) = 15$

· F(4, 3):

$i = 4,\ j = 3$ $\quad j - w_i = 3 - 2 = 1$

$v_i = v_4 = 15$ $\quad \therefore j - w_i \geq 0$

$w_i = w_4 = 2$

$F(4,3) = \max\{F(3,3),\ v_4 + F(3,1)\}$

$= \max\{22,\ 15 + 10\}$

$F(4,3) = 25$

· F(4, 4):

$i = 4,\ j = 4$ $\quad j - w_i = 4 - 2 = 2$

$v_i = v_4 = 15$ $\quad \therefore j - w_i \geq 0$

$w_i = w_4 = 2$

$F(4,4) = \max\{F(3,4),\ v_4 + F(3,2)\}$

$= \max\{30,\ 15 + 12\}$

$F(4,4) = 30$

· F(4, 5):

$i = 4,\ j = 5$ $\quad j - w_i = 5 - 2 = 3$

$v_i = v_4 = 15$ $\quad \therefore j - w_i \geq 0$

$w_i = w_4 = 2$

$F(4,5) = \max\{F(3,5),\ v_4 + F(3,3)\}$

$= \max\{22,\ 15 + 22\}$

$F(4,5) = 37$

↓ item i	j: weight → 0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

This entry represents the maximum possible value which can be put in a

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 $W - w_i = 5 - w_4 = 5 - 2 = 3$ 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 part of an optimal solution, $W - w_i = 3 - 1 = 2$ remaining units of knapsack capacity

Since $F(1,2) > F(0,2)$, item 1 is part of the optimal solution

optimal solution = {item 1, item 2, item 4}

- Time efficiency = space efficiency = $\Theta(nW)$
- time needed to find composition of optimal solution = $O(n)$

2) 0/1 Knapsack problem with memory functions approach.

- This method solves a given problem in the top-down manner but, in addition, maintains a table of the kind that would have been used by a bottom-up dynamic programming algorithm.
- Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated.
- Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first. if this entry is not "null", it is simply retrieved from the table, otherwise, it is computed by the recursive call whose result is then recorded in the table.
- After initializing the table, the recursive function needs to be called with $i = n$ (the number of items) and $j = W$ (the knapsack capacity).

Algorithm MFKnapsack(i, j)
// Implements the memory function method for the knapsack problem
// Input: A non-negative integer i indicating the number of the first items being considered & a non-negative integer j indicating the knapsack capacity
// Output: The value of an optimal feasible subset of the 1st i items

// Note : Uses as global variables input arrays Weights [1..n], Values [1..n], & table F[0..n, 0..W] whose entries are initialized with -1's except for row 0 & column 0 initialized with 0's

```
if F[i,j] < 0
    if j < weights[i]
        value ← MFKnapsack(i-1, j)
    else
        value ← max(MFKnapsack(i-1, j),
                    values[i] + MFKnapsack(i-1, j - weights[i]))
    F[i,j] ← value
return F[i,j]
```

Ex: 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

Knapsack capacity $W = 5$

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

We should start- compiling

$F(4,5) = \max(F(3,5), 15 + F(3,3))$

$F(3,5) = \max(F(2,5), 20 + F(2,2))$

$F(3,3) = \max(F(2,3), 20 + F(2,0))$

$F(2,5) = \max(F(1,5), 10 + F(1,4))$

$F(2,3) = \max(F(1,3), 10 + F(1,2))$

$F(2,2) = \max(F(1,2), 10 + F(1,1))$

$F(1,5) = \max(F(0,5), 12 + F(0,3))$

$F(1,4) = \max(F(0,4), 12 + F(0,2))$

$F(1,3) = \max(F(0,3), 12 + F(0,1))$

$F(1,2) = \max(F(0,2), 12 + F(0,0))$

$F(1,1) = F(0,1)$

Only 11 out of 20 nontrivial values (i.e, not those in row 0 or in column 0) have been computed.

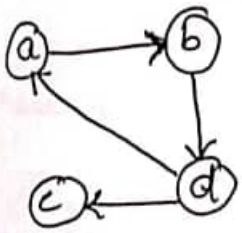
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	-1	12	22	-1	22
3	0	-1	-1	22	-1	32
4	0	-1	-1	-1	-1	37

8.14 Warshall's and Floyd's Algorithms

1.) Warshall's Algorithm

Used for computing the transitive closure of a directed graph. The transitive closure of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the i^{th} row & the j^{th} column is 1 if there exists a nontrivial path (i.e, directed path of a positive length) from the i^{th} vertex to the j^{th} vertex; otherwise, t_{ij} is 0

Ex:



Digraph

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Adjacency matrix

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Transitive closure (vertex is reachable)

Step by Step:

$R_0 =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Ref = 1st row 1st col

$R_1 =$

	0	1	0	0
0	0	1	0	0
0	0	0	0	1
0	0	0	0	0
1	1	1	1	0

Ref = 2nd row 2nd col

$R_2 =$

	0	0	0	1
0	0	1	0	1
0	0	0	0	1
0	0	0	0	0
1	1	1	1	1

Ref = 3rd row 3rd col

$R_3 =$

	0	0	0	0
0	0	1	0	1
0	0	0	0	1
0	0	0	0	0
1	1	1	1	1

Ref = 4th row 4th col

$R_4 =$

	1	1	1	1
1	1	1	1	1
1	1	1	1	1
0	0	0	0	0
1	1	1	1	1

Algorithm Warshall (A[1..n, 1..n])

// Implements warshall's algorithm for computing the transitive closure

// Input : The adjacency matrix A of a digraph with n vertices

// Output : The transitive closure of the digraph

$R^{(0)} \leftarrow A$

for $k \leftarrow 1$ to n do

  for $i \leftarrow 1$ to n do

    for $j \leftarrow 1$ to n do

      $R^{(k)}[i,j] \leftarrow R^{(k-1)}[i,j]$ or $(R^{(k-1)}[i,k]$ and $R^{(k-1)}[k,j])$

return $R^{(n)}$

Formula for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$:

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right)$$

This formula implies the following rules :

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i & column k & the element in its column j & row k are both 1's in $R^{(k-1)}$.

- Warshall's algorithm constructs the transitive closure through a series of $n \times n$ boolean matrices : $R^{(0)}, \ldots R^{(k-1)}, R^{(k)}, \ldots R^{(n)}$

2.) Floyd's Algorithm (All-Pairs Shortest-Paths Problem)

- Given a weighted connected graph (undirected or directed), the Floyd's Algorithm asks to find the distances, i.e the lengths of the shortest paths from each vertex to all other vertices.
- Applications: Communications, transportation networks and operations research.
- Distance matrix: It is a matrix used to record the lengths of shortest paths in an $n \times n$ matrix D. The element d_{ij} in the i^{th} row & j^{th} column of this matrix indicates the length of the shortest path from the i^{th} vertex to j^{th} vertex.
- Floyd's algorithm is appliable to both undirected and directed weighted graphs provided that they do not contain a cycle of a negative length. (The distance between any two vertices in such a cycle can be made arbitrarily small by repeating the cycle enough times.)

- Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices

$$D^{(0)}, \ldots D^{(k-1)}, D^{(k)}, \ldots D^{(n)}$$

- Formula for generating the elements of matrix $D^{(k)}$ from the elements of matrix $D^{(k-1)}$:

$$d_{ij}^{(k)} = \min\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \} \quad \text{for } k \geq 1, \quad d_{ij}^{(0)} = w_{ij}$$

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

$D \leftarrow W$ // is not necessary if W can be overwritten

for $k \leftarrow 1$ to n do

  for $i \leftarrow 1$ to n do

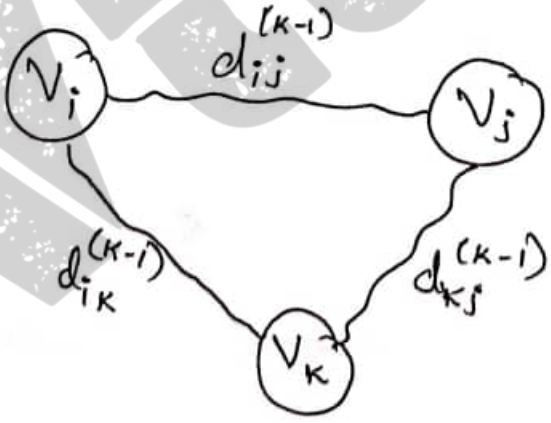
    for $j \leftarrow 1$ to n do

      $D[i,j] \leftarrow \min\{D[i,j], D[i,k] + D[k,j]\}$

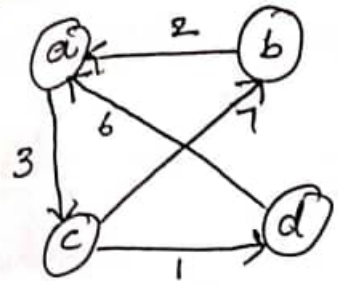
return D

- Underlying idea of Floyd's algorithm, path from V_i to V_j

V_i to V_j, i.e $V_i \rightarrow V_j = d_{ij}^{(k-1)}$ or $\left(d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$



Eg



$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Step by step:

$$D^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

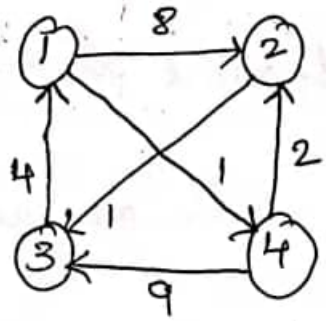
$$D^{(1)} = \begin{array}{c} \\ 0 \\ 2 \\ \infty \\ 6 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & \infty & 3 & \infty \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

$$D^{(2)} = \begin{array}{c} \\ \infty \\ 0 \\ 7 \\ \infty \end{array} \begin{array}{c} \begin{array}{cccc} 2 & 0 & 5 & \infty \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

$$D^{(3)} = \begin{array}{c} \\ 3 \\ 5 \\ 0 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccc} 9 & 7 & 0 & 1 \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 9 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

$$D^{(4)} = \begin{array}{c} \\ 4 \\ 6 \\ 1 \\ 0 \end{array} \begin{array}{c} \begin{array}{cccc} 6 & 16 & 9 & 0 \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Eg :



$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$D^1 = \begin{array}{c} \\ 0 \\ \infty \\ 4 \\ \infty \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 8 & \infty & 1 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$D^2 = \begin{array}{c} \\ 8 \\ 0 \\ 12 \\ 2 \end{array} \begin{array}{c} \begin{array}{cccc} \infty & 0 & 1 & \infty \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$D^3 = \begin{array}{c} \\ 9 \\ 1 \\ 0 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 4 & 12 & 0 & 5 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$D^4 = \begin{array}{c} \\ 1 \\ 6 \\ 5 \\ 0 \end{array} \begin{array}{c} \begin{array}{cccc} 7 & 2 & 3 & 0 \end{array} \\ \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

9. Greedy Technique :

Greedy algorithms are typically used to solve optimization problems only.

- The greedy approach constructs a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.
- On each step, the choice made must be :

1. feasible : it has to satisfy the problem's constraints
2. locally optimal : it has to be the best local choice among all feasible choices available on that step. (greedy choice)
3. irrevocable : once choice is made, it cannot be changed on subsequent steps of the algorithm.

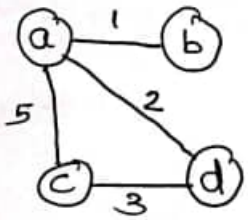
- Prim's algorithm and Kruskal's algorithm : two classic algorithms for the minimum spanning tree problem. These algorithms solve the same problem by applying the greedy approach in two different ways, and both of them always yield an optimal solution.
- There are problems for which a sequence of locally optimal choices (greedy choices) does not yield an optimal solution for every instance of the problem in question, However, a greedy algorithm can still be of value with an approximate solution.

```
Algorithm Greedy(A, n)
// Solves a problem using greedy method
// A is the input, which is a list of n elements
Solution ← ϕ
for i ← 1 to n-1
    x = choose(A)
    if (feasible(x))
        Solution = Union(Solution, x)
return Solution
```

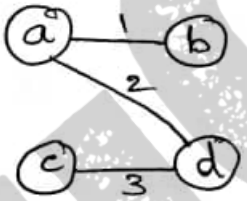
9.1 Prim's Algorithm (Minimum Spanning Tree (MST))

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.

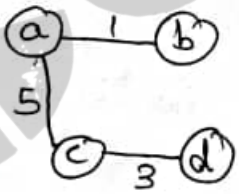
Eg:



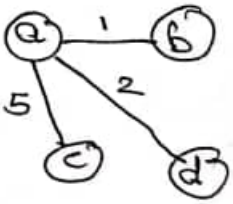
Graph



$W(T_1) = 6$



$W(T_2) = 9$

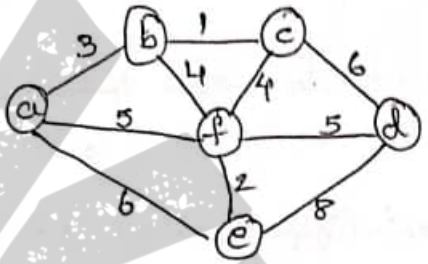


$W(T_3) = 8$

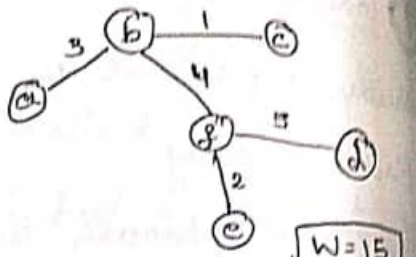
Graph & its spanning trees T_1, T_2, T_3. T_1 is the minimum spanning tree.

- Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees.
- The initial subtree 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 attaching to it the nearest

1.

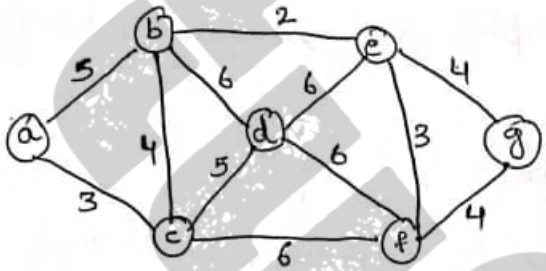


V = 6
E = 10

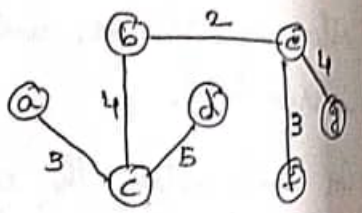


W = 15

2

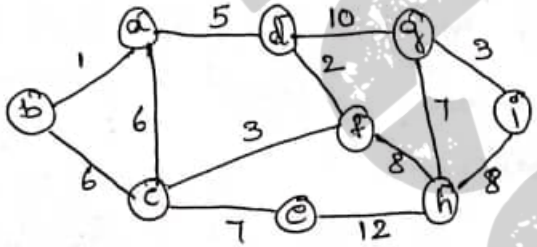


V = 7
E = 12

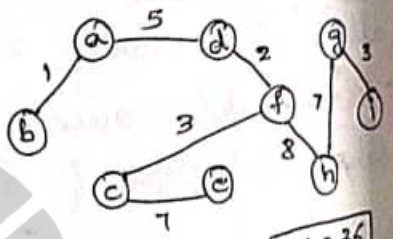


W = 21

3

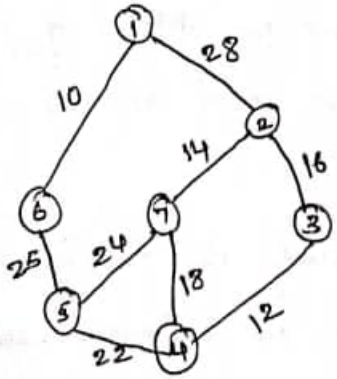


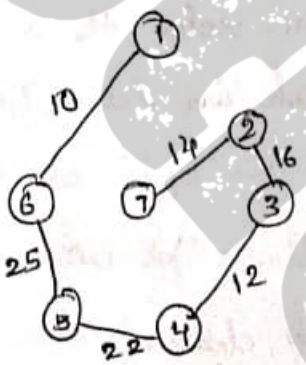
V = 9
E = 13



W = 36

4





W = 99

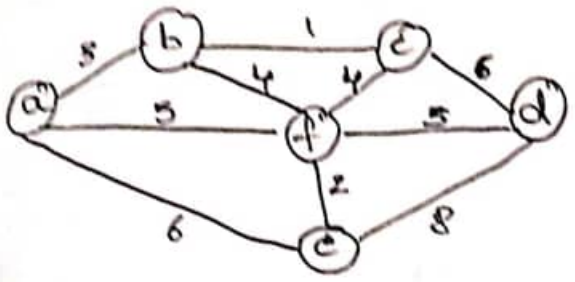
vertex not in that tree. (by an edge of the smallest weight)
- The algorithm stops after all the graph's vertices have been included in the tree being constructed.
- Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is $n-1$, where n is the number of vertices in the graph.

Algorithm Prim (G)
// Prim's algorithm for constructing a minimum spanning tree
// Input : A weighted connected graph $G = \langle V, E \rangle$
// Output : E_T, the set of edges composing a minimum spanning tree of G

$V_T \leftarrow \{v_0\}$ // the set of tree vertices can be initialized with any vertex.
$E_T \leftarrow \emptyset$
for $i \leftarrow 1$ to $|V|-1$ do
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

A spanning tree is a ① minimal connected subgraph, removing any edge disconnects it, ② a maximal acyclic subgraph, adding any edge creates a cycle.

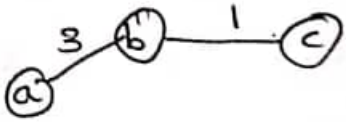
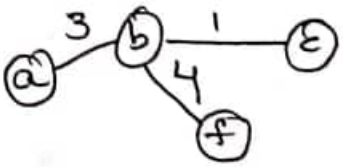
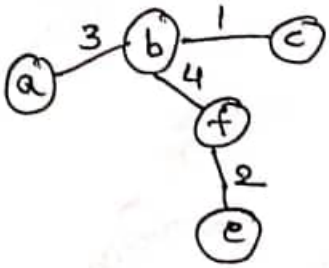
(by) Construct MST (Minimum Spanning Tree) using Prim's Algo



$V = \{a, b, c, d, e, f\}$

$E_T = \phi$ (should contain all edges used in the construction of MST)

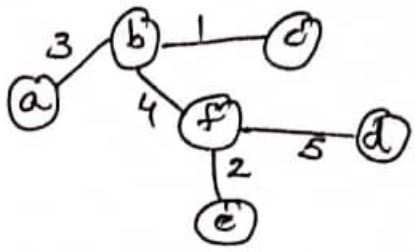
There are 6 vertices, i.e $n = |V| = 6$, there will be $|V| - 1$ (5) iterations.

	Tree vertices	Remaining vertices	MST
(1)	$V_T = \{a\}$	$V - V_T = \{b, c, d, e, f\}$	
	$E_T = \{ab\}$	**b(a, 3)**, c(–, ∞)	
		d(–, ∞), e(a, 6)	
		f(a, 5)	
(2)	$V_T = \{a, b\}$	$V - V_T = \{c, d, e, f\}$	
	$E_T = \{ab, bc\}$	**c(b, 1)**, d(–, ∞),	
		e(a, 6), f(b, 4)	
		f(a, 5)	
(3)	$V_T = \{a, b, c\}$	$V - V_T = \{d, e, f\}$	
	$E_T = \{ab, bc, bf\}$	d(c, 6), e(a, 6)	
		f(b, 4), f(a, 5)	
(4)	$V_T = \{a, b, c, f\}$	$V - V_T = \{d, e\}$	
	$E_T = \{ab, bc, bf, fe\}$	d(c, 6), d(f, 5)	
		e(a, 6), **e(f, 2)**	

⑤ $V_T = \{a, b, c, f, e\}$ $V - V_T = \{d\}$

$E_T = \{ab, bc, bf, fe, fd\}$ $d(c, 6)$, $d(f, 5)$

$d(e, 8)$



$V_T = \{a, b, c, f, e, d\}$, $W(T) = 15$

· Time complexity of Prim's algorithm:

1. If the input graph G is represented as a weighted matrix & the priority queue is implemented as an unordered array, the algorithm running time (efficiency) is $\Theta(|V|^2)$.

2. If the input graph G is represented by its adjacency lists and the priority queue is implemented as a min-heap (on edges), the running time is $O(|E| \log |V|)$, because we need to perform $|V|-1$ deletions of the smallest edge & makes $|E|$ verifications & possibly changes the heap $O(\log |V|)$

$$(|V| - 1 + |E|)(\log |V|) = O = (|E| \log |V|)$$

9.2 Kruskal's Algorithm (MST)

- Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph $G = \langle V, E \rangle$ as an acyclic subgraph with $|V|-1$ edges for which the sum of the edge weights is the smallest.
- The algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.
- The algorithm begins by sorting the graph's edges in nondecreasing order of their weights.
- Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

- **Algorithm** Kruskal(G)

// Kruskal's algorithm for constructing a minimum spanning tree

// Input : A weighted connected graph $G = \langle V, E \rangle$

// Output : E_T, the set of edges composing a MST of G

Sort E in nondecreasing order (increasing order) of the edge weights $w(e_{i_1}) \leq \ldots \leq w(e_{i_{|E|}})$

$E_T \leftarrow \phi$

ecounter $\leftarrow 0$ // initialize the set of tree edges & its size

$K \leftarrow 0$ // initialize the number of processed edges

while ecounter $< |V|-1$ do

$K \leftarrow K+1$

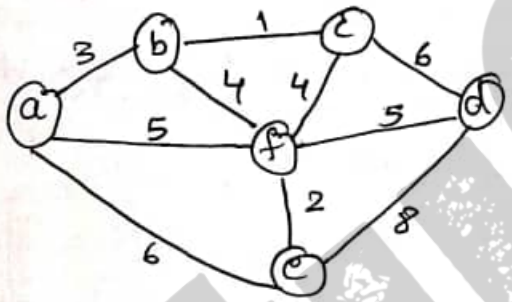
if $E_T \cup \{e_{i_K}\}$ is acyclic

$E_T \leftarrow E_T \cup \{e_{i_K}\}$

ecounter $\leftarrow$ ecounter $+1$

return E_T

Eg: Construct MST using Kruskal's algorithm



$V = \{a, b, c, d, e, f\}$

E

$|V| = 6$, $|V|-1 = 5$ iterations

Sort the edges in increasing order of their weights

Tree edges	Sorted list of edges	Illustration
	bc, ef, ab, bf, cf	
	1 2 3 4 4	
	af, df, ac, cd, ed	
	5 5 6 6 8	

① $E = \{bc\}$

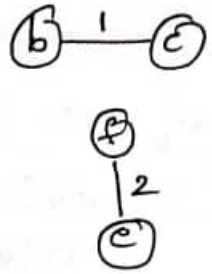
$V = \{b, c\}$

bc, ef, ab, bf, cf, af, df, ac, cd, ed

1 2 3 4 4 5 5 6 6 8

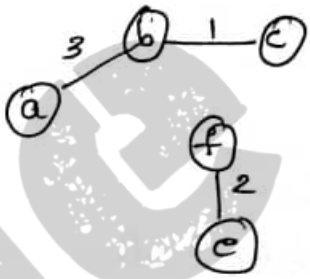
② $E = \{bc, ef\}$ bc, ef, ab, bf, cf, af, df, ae, cd, de
1 2 3 4 4 5 5 6 6 8

$V = \{b, c, e, f\}$



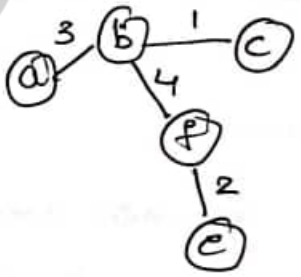
③ $E = \{bc, ef, ab\}$ bc, ef, ab, bf, cf, af, df, ae, cd, de
1 2 3 4 4 5 5 6 6 8

$V = \{b, c, e, f, a\}$



④ $E = \{bc, ef, ab, bf\}$ bc, ef, ab, bf, cf, af, df, ae, cd, de
1 2 3 4 4 5 5 6 6 8

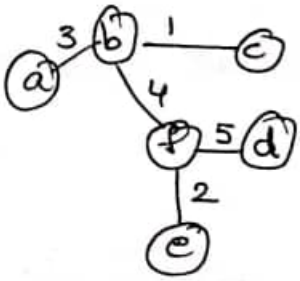
$V = \{b, c, e, f, a\}$



(forms a cycle) Rejected

⑤ $E = \{bc, ef, ab, bf, df\}$ bc, ef, ab, bf, cf, af, df, ae, cd, de
1 2 3 4 4 5 5 6 6 8

$V = \{b, c, e, f, a, d\}$



once all the vertices are constructed, we can stop the algorithm, we have the MST. $W(T) = 15$

- Both Prim's & Kruskal's algorithm MST $W(T) = 15$, with different greedy approches.
- Time complexity = Sort the edges + Extracting the edges $= |E| \log |E| + |E| \approx |E| \log |E|$

Prim's Algorithm	Kruskal's Algorithm
• It starts to build MST from any vertex in the graph.	• It starts to build MST from vertex carrying minimum weight in the graph.
• It traverses one node more than one time to get the minimum distance.	• It traverses one node only once.
• Gives connected component as well as it works only on connected graph.	• Gives forest (disconnected components) at any instant as well as it can work on disconnected components.
• Intermediate steps/answers as well as final/complete answer are always connected.	• Intermediate answer can be disconnected, complete answer will be connected.
• Generates the MST starting from the root vertex	• Generates the MST from the least weighted edge
• Prefers list data structures	• Prefers heap data structures
• Applications : Travelling sales-man problem, Network for roads & rail tracks connecting all the cities, etc.	• Applications : LAN connections, TV network, etc
• Time complexity : $O(V^2)$ or $O(E \log V)$	• Time complexity : $O(\|E\| \log V)$

9.3 Dijkstra's Algorithm (Single-source shortest-path problem)

For a given vertex called the source in a weighted connected graph, find the shortest paths to all its other vertices. Graph $(G(V, E))$, V_0 to all remaining vertices. V_0 is called source and the last vertex is called destination.

- This problem is solved by Dijkstra's algorithm.
- Dijkstra's algorithm is applicable to undirected & directed graphs with non negative weights only.
- Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source.
- First, it finds the shored path from the source to a vertex nearest to it, then to a second nearest, and so on.
- In general, before its i^{th} iteration commences, the algorithm has already identified the shortest paths to $i-1$ other vertices neared to the source.
- These vertices, the source, & the edges of the shortest paths leading to them from the source form a subtree T_i of the given graph.
- The next vertex nearest to the source can be found among the vertices adjacent to the vertices of T_i.
- The set of vertices adjacent to the vertices of T_i is called "fringe vertices", from which Dijkstra's algorithm selects the next vertex nearest to the source.

Algorithm Dijkstra(G, s)

// Dijkstra's algorithm for single-source shortest paths

// Input: A weighted connected graph $G = \langle V, E \rangle$ with non-negative weights & its vertex s

// Output: The length d_v of a shortest path from s to v & its penultimate vertex P_v for every vertex v in V

Initialize(Q) // initialize priority queue to empty

for every vertex v in V

$d_v \leftarrow \infty$ // distance of node v is ∞

$P_v \leftarrow$ null // path value of node v is null

Priority Parent

Insert(Q, v, d_v) // initialize vertex priority in the priority queue

$d_s \leftarrow 0$

Decrease(Q, s, d_s) // update priority of s with d_s

$V_T \leftarrow \phi$

for $i \leftarrow 0$ to $|V| - 1$ do

$u^* \leftarrow$ DeleteMin(Q) // delete the minimum priority element

$V_T \leftarrow V_T \cup \{u^*\}$

for every vertex u in $V - V_T$ that is adjacent to u^* do

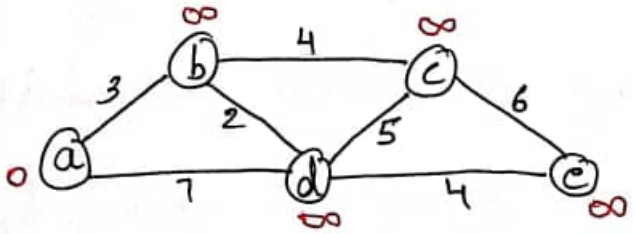
if $d_{u^*} + w(u^*, u) < d_u$ } Relaxation / updation

$d_u \leftarrow d_{u^*} + w(u^*, u)$

$P_u \leftarrow u^*$

Decrease(Q, u, d_u)

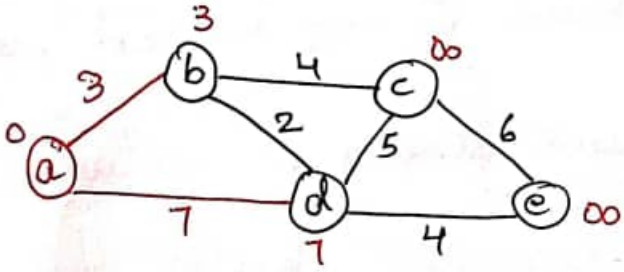
Eg: Consider the following graph & obtain the shortest paths from the vertex "a" (source) to all other vertices using Dijkstra's algorithm.

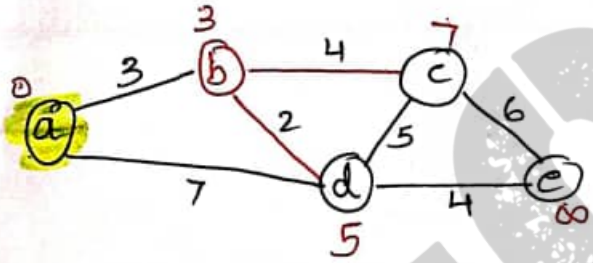


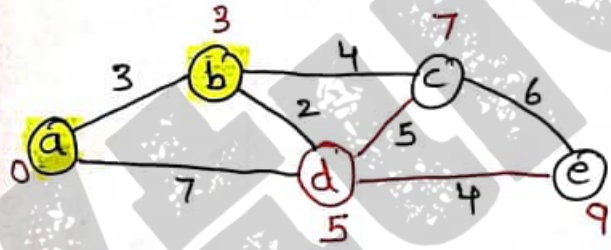
Starting vertex / Source is (a)

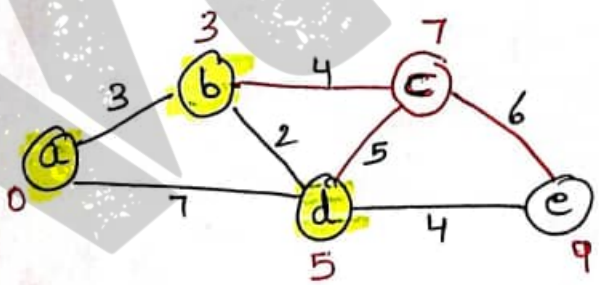
(extract min)

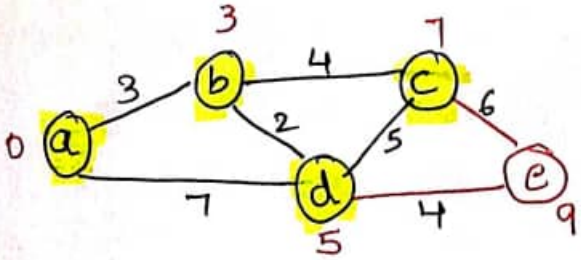
Selected Vertex	b	c	d	e
	∞	∞	∞	∞
(a)	(3)	∞	7	∞
(b)	3	7	(5)	∞
(d)	3	(7)	5	9
(c)	3	7	5	(9)
(e)	3	7	5	9



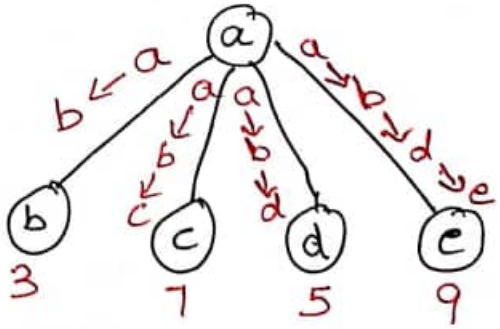


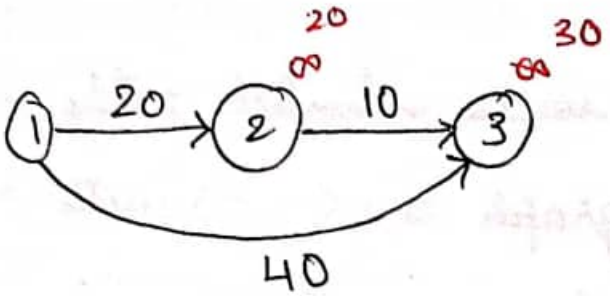


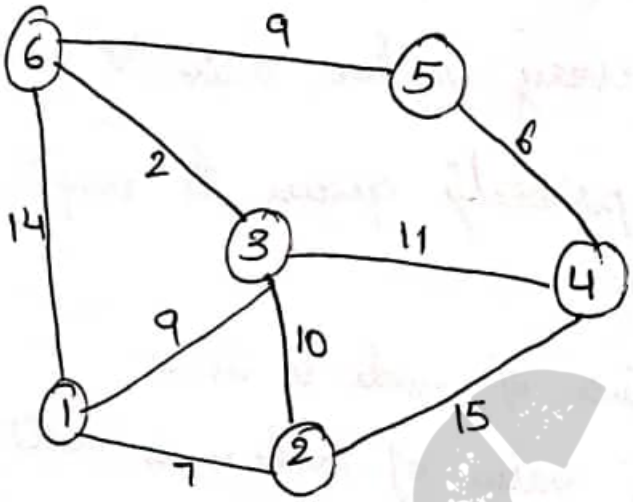




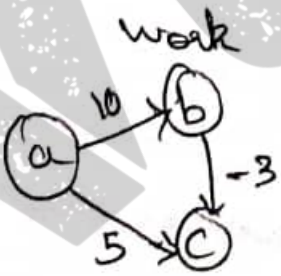
∴

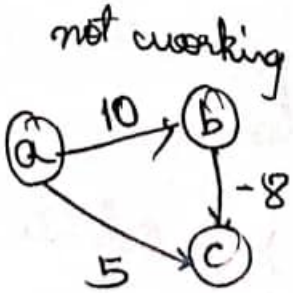






1	2	3	4	5	6
a	∞	∞	∞	∞	∞
(1)	7	9	∞	∞	14
1, (2)	7	(9)	22	∞	14
1,2, (3)	7	9	20	∞	(11)
1,2,3 (6)	7	9	(20)	20	11
1, 2, 3, 6, (4)	7	9	20	(20)	11
1, 2, 3, 6, 4, (5)	7	9	20	20	11





9.4 Huffman Trees and codes

- **Optimal tree problem:** 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 **codeword**.
- There are two ways of encoding: ① fixed-length encoding ② Variable-length encoding.

① Fixed-length encoding:

- This method assigns to each symbol a bit string of the same length m ($m \geq \log_2 n$). This is exactly what the standard ASCII code does.
- One way of getting a coding scheme that yields a shorter bit string on the average is based on the old idea of assigning shorter codewords to more frequent symbols & longer codewords to less frequent symbols.

② Variable-length encoding:

- This method assigns codewords of different lengths to different symbols, introduces a problem that fixed-length encoding doesn't have. How can we tell how many bits of an encoded text represent the 1st (or, more generally, the ith) symbol?
- To avoid this complication, can limit ourselves to the so called prefix-free (or simply prefix) codes.

Fixed length:

-B C C A B B D D A E C C B B A E D D C C — 20

character	count/frequency		code
A	3	3/20	000
B	5	5/20	001
C	6	6/20	010
D	4	4/20	011
E	2	2/20	100

using ASCII - 20 × 8 = 160 bits

using owncode - 20 × 3 = 60 bits

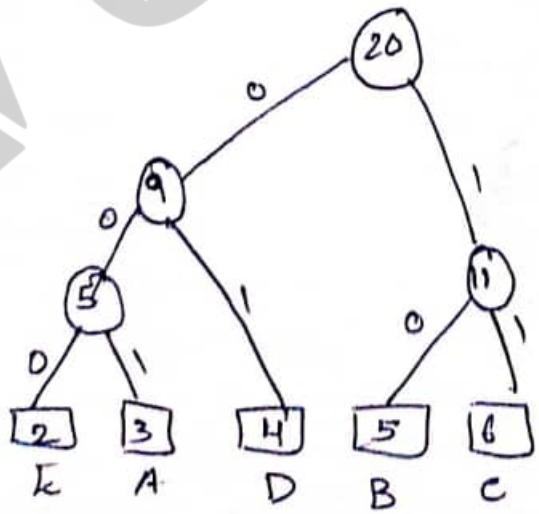
5 × 8 + 5 × 3 — to store the table

40 + 15 = 55 bits

60 + 55 = 115 bits Total

Variable length: optimal merge pattern (Huffman)

$\Sigma d_i \times f_i$



char	count	code
A	3	001
B	5	10
C	6	11
D	4	01
E	2	000
	5×8=40	12 bits = 52

3 × 3 = 9
5 × 2 = 10
6 × 2 = 12
4 × 2 = 8
2 × 3 = 6
} merg 45 bits

table 45+52 = 97 bits

- In a prefix code, no codeword is a prefix of codeword ~~some~~ symbol of another symbol. Hence, with such an encoding, we can simply scan a bit string until we get the first group of bits that is a codeword for some symbol, replace these bits by this symbol & repeat this operation until the bit string's end is reached.
- If we want to create a binary prefix code for some alphabet, associate the alphabet's symbols with leaves of a binary tree in which all the left edges are labeled 0 & all the right edges are labeled 1. The codeword of a symbol can then be obtained by recording the labels on the simple path from the root to the symbol's leaf. Since there is no simple path to a leaf that continues to another leaf, no codeword can be a prefix of another codeword, hence, any such tree yields a prefix code.
- How can we construct a tree that would assign shorter bit strings to high-frequency symbols & longer ones to low frequency symbols?
- It can be done by the following greedy algorithm, Huffman's algorithm.

Huffman's coding algorithm is use to

1. Encoding data
2. Data compression.

Huffman's algorithm (Compression technique to reduce data size)

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. (the weight of a tree will be equal to sum of the frequencies in the tree's leaves)

Step 2: Repeat the following operation until a single tree is obtained.

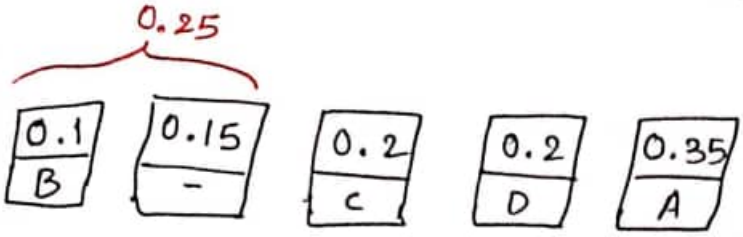
- Find two trees with the smallest weight (ties can be broken arbitrarily). Make them the left & right subtree of a new tree & record the sum of their weights in the root of the new tree as its weight.

A tree constructed by Huffman's algorithm is called a **Huffman tree**. It defines a Huffman code.

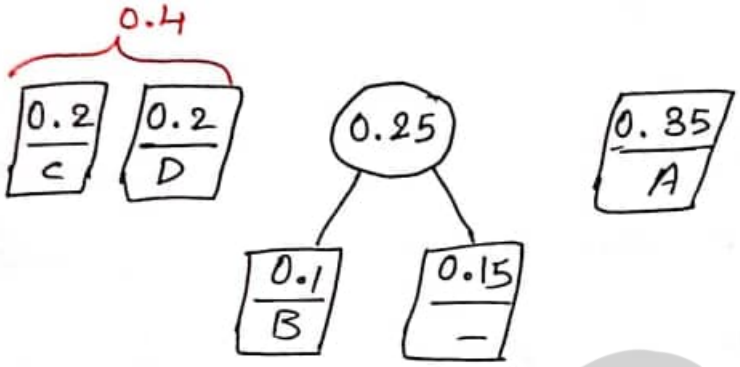
Ex: Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

Symbol	A	B	C	D	_
Frequency	0.35	0.1	0.2	0.2	0.15

Arrange the data in ascending order of weight

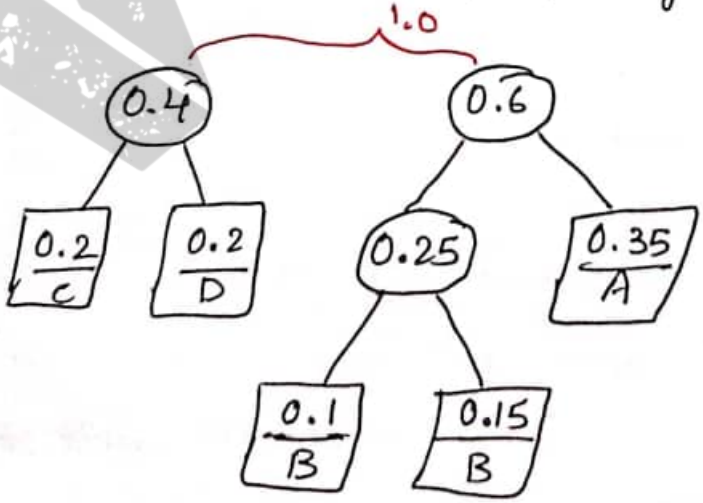


Extract two min frequency nodes & add a new internal node with the sum of their frequencies.

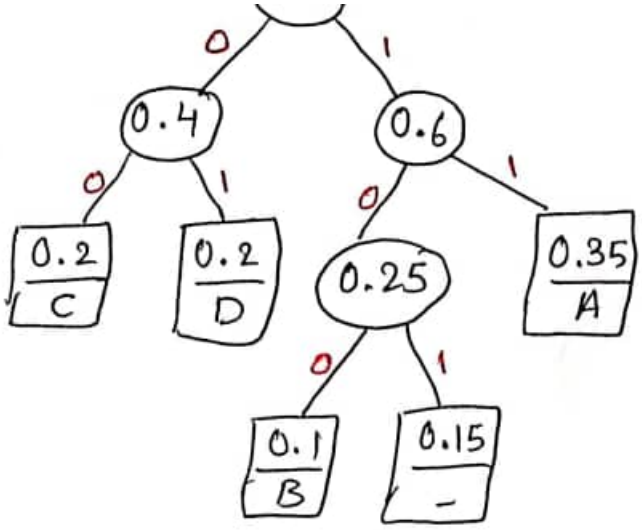


Extract next 2 min frequency nodes & add them

Extract next 2 min frequency nodes & add them



Extract next 2 min frequency nodes & add them



Now encode the above created Huffman tree. The left branch is labeled 0 & right branch is labeled 1.

The resulting codewords are as follows:

Symbol	A	B	C	D	-
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Ex: Encode : DAD - 01 11 01 - Avg no of bits per symbol : $(2 \times 0.2)_{D} + (2 \times .35)_{A} + (2 \times 0.2)_{D} = \boxed{1.5}$

Decode : 100 11 01 - BAD

$$\left(\begin{matrix}\text{no of bits} \\ \text{in codeword}\end{matrix} \times \text{frequency}\right)$$

Compression ratio

- Dynamic Huffman encoding :

coding tree is updated each time a new symbol is read from the source text.

- Lempel - Ziv algorithm (modern alternatives)