# Greedy Algorithms.

A Greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

"The one with maximum benefit from multiple choice is selected" is the basic idea of greedy method. A greedy method arrives at a solution by making a sequence of choices, each of which simply looks the best at the moment.

## Elements of Greedy Algorithm.

### 1. Greedy Choice Property

The first key ingredient is the greedy choice property: we can assemble a globally optimal solution by making locally optimal (greedy) choices.

In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

### 2. Optimal Sub-Structure.

The solution of the given problem ie global choice which is the best of all the local choice is optimal only if the solution at each level or local choice is optimal

# Difference between Dynamic Programming & Greedy Approach

| Dynamic Programming | Greedy Approach |
|---|---|
| 1. It is a powerfull technique and have widely used to solve wider set of problem. | This technique is not as powerful as Dynamic prog. |
| 2. The solution generated is always an optimal sol$^n$. | The solution generated may be an optimal sol$^n$. |
| 3. It follows bottom-up approach to solve a problem. | It uses top-down approach to solve the problem. |
| 4. It ~~follows~~ All the subproblems are dependent on each other. | Generating global solution does not require dependency of subproblems. |
| 5. Cost to solve a problem is high | Cost is less. |

Problems to be covered in Greedy Approach :-

→ Activity Selection problem
→ fractional Knapsack problem
→ Huffman Codes
→ A task scheduling problem.

# Activity Selection Problem.

An activity-selection is the problem of scheduling a resource among several competing activity.
The activity selection problem is defined also as: "Given a set 's' of 'n' activities with start time $s_i$ and $f_i$ as finish time of an $i^{th}$ activity.
The problem is to find the maximum size set of mutually compatible activities. (or non-conflicting activities)

## Steps:

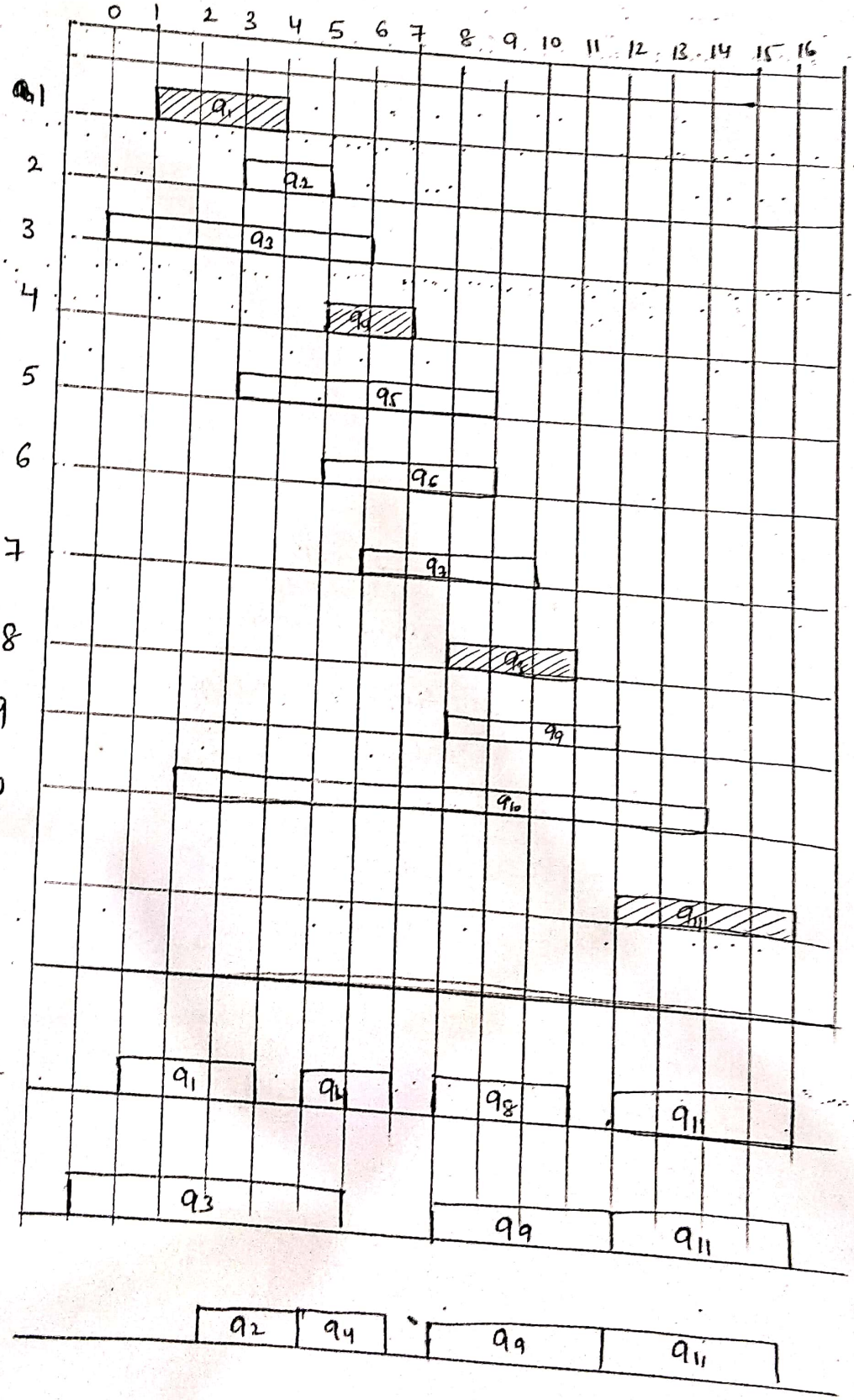
1. Sort the activities as per finishing time in ascending order.
2. Select the 1st activity.
3. Select the new activity if its starting time is greater than or equal to the previously selected activity.
   Repeat step 3 till all activities or checked.

GREEDY-ACTIVITY-SELECTOR (s,f)

1. n = s.length
2. A = {a1}
3. K = 1
4. for m = 2 to n  (m = i, K = j)
5. if s[m] ≥ f[K]
6. A = A ∪ {am}
7. K = m
8. return A.

eg: Given :-

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Si | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| fi | 4 | 5 | 6 | 7 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |




Compatible Activities

1 $a_1$ $a_4$ $a_8$ $a_{11}$

2 $a_3$ $a_9$ $a_{11}$

3 $a_2$ $a_4$ $a_9$ $a_{11}$

Select activity = $a_1$
Starting time = 1
finish time = 4

previously selected activity = $i$
next " " = $j$

Start time of $j$
finish time of $i$

Check ??

is, $s$ of $j >= f$ of $i$

if No, move on to next

if Yes, select $j$.

$s_j = 3$
$f_i = 4$ ⇒ check 3 >= 4 No, next.

$s_j = 0$
$f_i = 4$ ⇒ check 0 >= 4 No, next

$s_j = 5$
$f_i = 4$ ⇒ check 5 >= 4 Yes Select → $a_1$

$s_j = 3$
$f_i = 7$ ⇒ check 3 >= 7 No, next

$s_j = 5$
$f_i = 7$ ⇒ check 5 >= 7 No, next

$s_j = 6$
$f_i = 7$ ⇒ check 6 >= 7 No, next

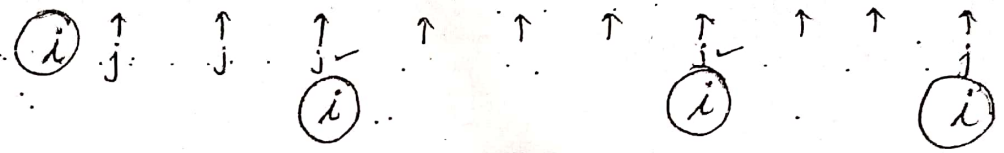
$S_j = 8$ $\quad 8 >= 7$ Yes select $\rightarrow q_4$

$f_i = 7$

accordingly select the activities.

Soln = $\boxed{q_1, q_4, q_8, q_{11}}$

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| s | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| f | 4 | 5 | 6 | 7 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |




$\{q_1, q_4, q_8, q_{11}\}$ → selected activity

# Huffman Coding

Huffman coding is a lossless data compression algo. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

eg:

| Characters | freq. | fixed-length codeword. | Variable length codeword |
|---|---|---|---|
| a | 45 | 000 | |
| b | 13 | 001 | Need to find out ?? |
| c | 12 | 010 | |
| d | 16 | 011 | |
| e | 9 | 100 | |
| f | 5 | 101 | |

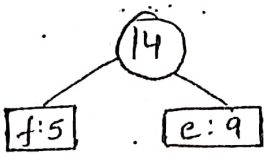
## Steps:

1) Arrange the characters in ascending order of their frequencies
2) Merge the 2 characters with lowest frequencies.
3) Take the next lowest freq. character and add it to the tree:

(a) [f:5] [e:9] [c:12] [b:13] [d:16] [a:45]

(arrange in increasing order)

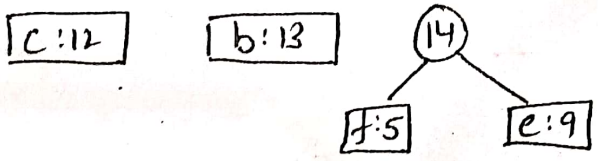
(b)




[c:12] [b:13] [d:16] [a:45]

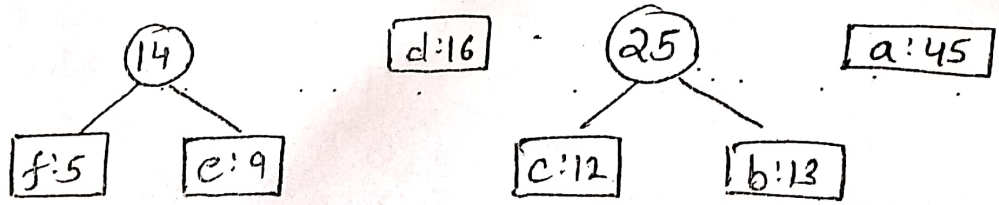
(merge 2 lowest freq. char.)

(c)

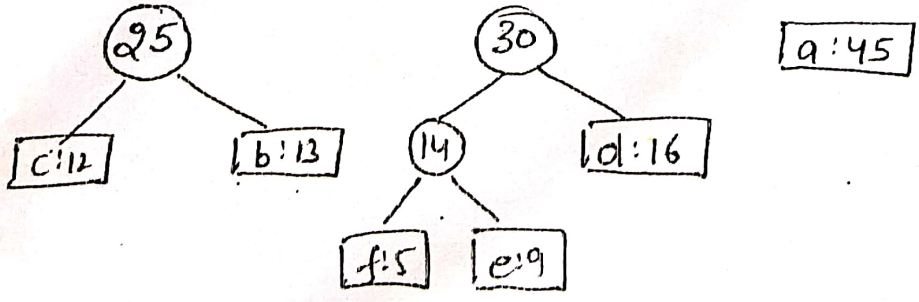



(Rearrange)

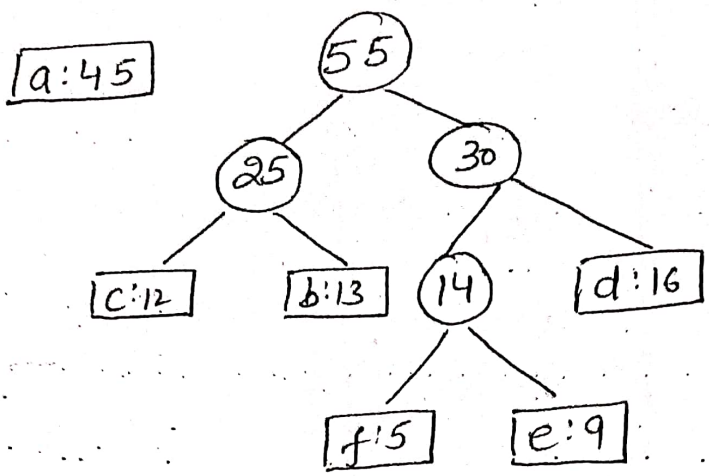
(d)



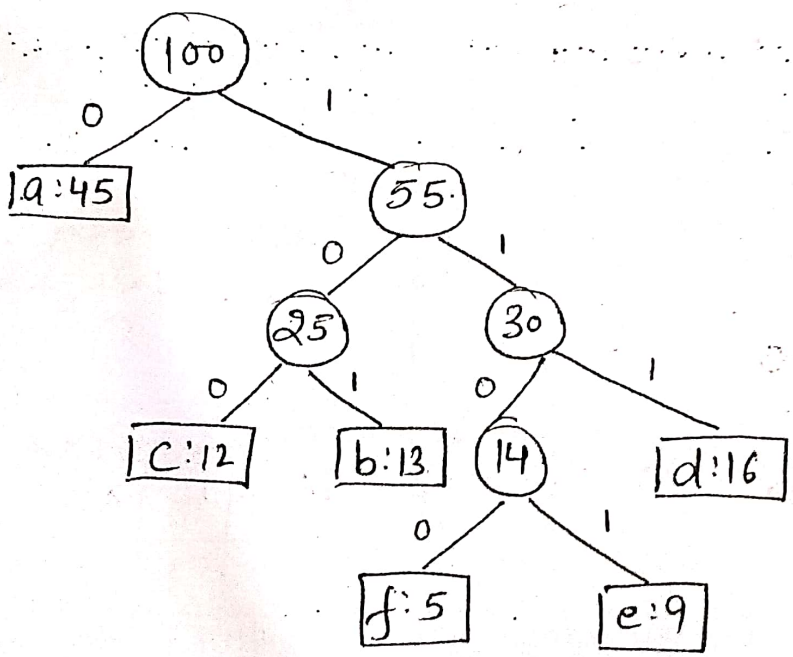

(e)



(f)




(g)




After compliting the tree, assign codes.

0 → left child

1 → right child.

a → 0

b → 101

c → 100

d → 111

e → 1101

f → 1100

| characters | freq. | fixed length code | variable length code |
|---|---|---|---|
| a | 45 | 000 | 0 |
| b | 13 | 001 | 101 |
| c | 12 | 010 | 100 |
| d | 16 | 011 | 111 |
| e | 9 | 100 | 1101 |
| f | 5 | 101 | 1100 |

Total Benefit = freq. × Bit length.

$T.B_F = 45\times3 + 13\times3 + 12\times3 + 16\times3 + 9\times3 + 5\times3$

$= 300$

$T.B_V = 45\times1 + 13\times3 + 12\times3 + 16\times3 + 9\times4 + 5\times4$

$= 224$

The total running time of HUFFMAN on a set of n characters is $\boxed{O(n \log n)}$.

# 0/1 Knapsack

Base case : $C[0, J] = 0$ — No object, Bag of size J ∴ Profit = 0

$C[i, 0] = 0$ — i objects, No bag ∴ Profit = 0

$$C[i,j] = \begin{cases} C[i-1, j] & w_i > j \longrightarrow \text{wt of object is more than bag} \\ \max \begin{cases} C[i-1, j] \\ P_i + C[i-1, J - w_i] \end{cases} & w_i \leq j \longrightarrow \text{when object is less than bag} \end{cases}$$

eg

| Objects : | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Weights : | 2 | 3 | 4 | 5 |
| profits : | 3 | 4 | 5 | 6 |

M = 7

What is the max profit = ?

| – | 0 | 1 | 2 | J 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | 3 |
| 2 | 0 | 0 | 3 | 4 | 4 | 7 | 7 | 7 |
| 3 | 0 | 0 | 3 | 4 | 5 | 7 | 8 | 9 |
| 4 | 0 | 0 | 3 | 4 | 5 | 7 | 8 | 9 |

i

max profit = 9.

W P

(2, 3) Object wt = 2, Bag size = 1 ∴ profit = 0

object wt = 2, Bag wt = 2. ∴ (0, 3) max = 3.

object remains same. Chahe bag ka size Kitna hi badha

(3, 4) object wt = 3., bag size = 3.
max [3, 4+0] = 4.
object wt = 3, bag size = 4.
max [3, 4+0] = 4.
object wt = 3, bag size = 5

(4, 5) (3, 4) = wt = 4 J = 4, max [4, 5+0] = 5.
(3, 5) = wt = 4 J = 5, max [7, 5+0] = 7
(3, 6) = wt = 4, J = 6, max (7, 5+3) = 8

# Dynamic Programming

## 0-1 Knapsack problem

Gujar

### Knapsack Problem

Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W.
So we must consider weights of items as well as their value.

### 0-1 Knapsack problem :

→ Items are indivisible ; you either take an item or not.
→ Solved with dynamic programming

Given a knapsack with maximum capacity W, and a set S consisting of n items.
Each item i has some weight wi and benefit value bi. (all wi, bi and W are integer values)

**Problem** : How to pack the knapsack to achieve maximum total value of packed items?

| Weight (wi) | Benefit value (bi) |
|---|---|
| 2 | 3 |
| 3 | 4 |
| 4 | 5 |
| 5 | 8 |
| 9 | 10 |

Max. weight
W = 20.

## Recursive formula for subproblems

$$B[K, w] = \begin{cases} B[K-1, w] & \text{if } w_K > w \\ \max\{B[K-1, w], B[K-1, w-w_K] + b_K\} & \text{else} \end{cases}$$

It means, that the best subset of $S_K$ that has total weight $w$ is:

1) the best subset of $S_{K-1}$ that has total weight $w$, or
2) the best subset of $S_{K-1}$ that has total weight $w-w_K$ plus the item $K$.

Best subset of $S_K$ that has the total weight $w$, either contains item $K$ or not.

First case: $w_K > w$. Item $K$ can't be part of the solution since if it was, the total weight would be $> w$, which is unacceptable.

Second case: $w_K \leq w$. Then the item $K$ can be in the solution and we choose the case with greater value.

# 0-1 Knapsack Algorithm.

```
for w=0 to W
    B[0, w] = 0
for i=1 to n
    B[i, 0] = 0
for i = 1 to n
    for w=0 to W
        if wi <= w // item i can be part of the solution.
        if bi + B[i-1, w-wi] > B[i-1, w]
            B[i, w] = bi + B[i-1, w-wi]
        else
            B[i, w] = B[i-1, w]
        else B[i, w] = B[i-1, w]  // wi > w.
```

eg: $n = 4$ (No. of elements)

$W = 5$ (max weight)

Elements (weight, benefit):

(2, 3), (3, 4), (4, 5), (5, 6)

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | | | | |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |

$i = 1$

$b_i = 3$

$w_i = 2$

$w = 1$

$w - w_i = -1$

items

1: (2, 3)

2: (3, 4)

3: (4, 5)

4: (5, 6)

for $w = 0$ to $W$ : $B[0, w] = 0$

for $i = 1$ to $n$ : $B[i, 0] = 0$

else $B[i, w] = B[i-1, w]$

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

$i = 1, 1, 1, 1$

$b_i = 3, 3, 3, 3$

$w_i = 2, 2, 2, 2$

$w = 2, 3, 4, 5$

$w - w_i = 0, 1, 2, 3$

$B[i, w] = b_i + B[i-1, w-w_i]$

$B[i, w] = b_i + B[i-1, w-w_i]$

$B[i, w] = b_i + B[i-1, w-w_i]$

$B[i, w] = b_i + B[i-1, w-w_i]$

items:

| | |
|---|---|
| 1 | (2, 3) |
| 2 | (3, 4) |
| 3 | (4, 5) |
| 4 | (5, 6) |

| i \ W | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 3 | 3 | 3 | 3 |
| 2 | 0 | 0 | 3 | 4 | 4 | 7 |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |

$i = 2$

$b_i = 4$

$w_i = 3$

$w = 1, 2, 3, 4, 5$

$w - w_i = -2, -1, 0, 1, 2$

else $B[i, w] = B[i-1, w]$ // $w_i > w$

else $B[i, w] = B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

$B[i, w] = b_i + B[i-1, w-w_i]$

$B[i, w] = b_i + B[i-1, w-w_i]$

items:

| | |
|---|---|
| 1 | (2, 3) |
| 2 | (3, 4) |
| 3 | (4, 5) |
| 4 | (5, 6) |

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 3 | 3 | 3 | 3 |
| 2 | 0 | 0 | 3 | 4 | 4 | 7 |
| 3 | 0 | 0 | 3 | 4 | 5 | 7 |
| 4 | 0 | | | | | |

$i = 3$

$b_i = 5$

$w_i = 4$

$w = 1 .. 3, 4, 5$

$w - w_i = 0, 1$

$B[i, w] = B[i-1, w]$ // $w_i > w$

$B[i, w] = b_i + B[i-1, w-w_i]$

$B[i, w] = B[i-1, w]$

| i\W | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 3 | 3 | 3 | 3 |
| 2 | 0 | 0 | 3 | 4 | 4 | 7 |
| 3 | 0 | 0 | 3 | 4 | 5 | 7 |
| 4 | 0 | 0 | 3 | 4 | 5 | (7) |

$i = 4$

$b_i = 6$

$w_i = 5$

$w = 1 \ldots 4, 5$

$w - w_i = 0$

items:

1 : (2, 3)
2 : (3, 4)
3 : (4, 5)
4 : (5, 6)

else $B[i, w] = B[i-1, w]$ // $w_i > w$

$B[i, w] = B[i-1, w]$

Optimal solution.

Max weight = 5

Items = {1, 2}

benefit = 7.

# Fractional Knapsack

Given a knapsack of capasity "W", items 'i' $\{i_1, i_2, i_3 \ldots i_n\}$ of weights $\{w_1, w_2, w_3 \ldots w_n\}$ and benefit $\{v_1, v_2, v_3 \ldots v_n\}$ Our objective is to maximize the benefit such that the total weight inside the knapsack is atmost "W".

Steps:

1.) Calculate value per weight for each item ie density.
2.) Sort the items as per the value density in descending order.
3.) Take as much items as possible, not already taken in the knapsack.

eg:

| Item | Weight | value (Benefit) |
|---|---|---|
| $i_1$ | 6 | 6 |
| $i_2$ | 10 | 2 |
| $i_3$ | 3 | 1 |
| $i_4$ | 5 | 8 |
| $i_5$ | 1 | 3 |
| $i_6$ | 3 | 5 |

Density = $\frac{\text{Value}}{\text{Weight}}$

1.00
0.2
0.3
1.6
3.0
1.667

Sorted List

| I | w | v | Density |
|---|---|---|---|
| $i_5$ | 1 | 3 | 3.00 |
| $i_6$ | 3 | 5 | 1.667 |
| $i_4$ | 5 | 8 | 1.600 |
| $i_1$ | 6 | 6 | 1.000 |
| $i_3$ | 3 | 1 | 0.330 |
| $i_2$ | 10 | 2 | 0.200 |

Our objective is to fill the knapsack with items to get max. benefit without crossing the weight limit W = 16

| item | weight | value | Total weight | Benefit |
|---|---|---|---|---|
| $i_5$ | 1 | 3 | 1.000 | 3.000 |
| $i_6$ | 3 | 5 | 4.000 | 8.000 |
| $i_4$ | 5 | 8 | 9.000 | 16.000 |
| $i_1$ | 6 | 6 | 15.000 | 22.000 |
| $i_3$ | 1 | 0.333 | 16.000 | 22.333 |

Total weight inside Knapsack, initially = 0

check $i_5$

is weight of ($i_5$) + total weight <= 'W'

1 + 0 <= 16

1 <= 16 → Yes (include)

Check $i_6$,

$3+1 <= 16 \rightarrow$ Yes, include

check $i_4$

$5+4 <= 16 \rightarrow$ Yes, include

check $i_1$

$6+9 <= 16 \rightarrow$ Yes, includ

check $i_2$

$3+15 <= 16 \rightarrow$ No.

we will fill the Knapsack with 1 weight of $i_3$ item having value $(1/3) = 0.333$

Total weight inside Knapsack is $= 16$, so we stop here.

**Max Benefit = 22.333**

---

Example 2:

| Items | wt. | value |
|---|---|---|
| $i_1$ | 5 | 6 |
| $i_2$ | 7 | 14 |
| $i_3$ | 2 | 6 |
| $i_4$ | 3 | 9 |
| $i_5$ | 4 | 5 |

Calculate the max. Benefit using fractional Knapsack with max. weight = 15 Kgs.

Sol:

| I | wt. | V | D |
|---|---|---|---|
| 1 | 5 | 6 | 1.200 |
| 2 | 7 | 14 | 2.00 |
| 3 | 2 | 6 | 3.00 |
| 4 | 3 | 9 | 3.00 |
| 5 | 4 | 5 | 1.250 |

// cal. Density = $\frac{\text{Value}}{\text{weight}}$

Now, sort according to density in descending orde.

| I | wt. | V | D |
|---|---|---|---|
| 3 | 2 | 6 | 3 |
| 4 | 3 | 9 | 3 |
| 2 | 7 | 14 | 2 |
| 5 | 4 | 5 | 1.25 |
| 1 | 5 | 6 | 1.20 |

Initiall total weight of Knapsack = 0.

Knapsack

| I | Wt | V | Total weight value | Benefit |
|---|---|---|---|---|
| 3 | 2 | 6 | 2 | 6.00 |
| 4 | 3 | 9 | 5 | 15.00 |
| 2 | 7 | 14 | 12 | 29.00 |
| 5 | 3 | 3.75 | 15 | 32.750 |

$i_3 \Rightarrow$ 2+0 <= 15 ✓
3+2 <= 15 ✓
7+5 <= 15 ✓
4+12 <= 15 ✗ → only 3 units can be used.

4 unit wt. of $i_5$ = 5
1 unit = 5/4 = 1.25 , 3 unit = 1.25 × 3 = 3.75

Benefit = wt taken × $\frac{\text{total value}}{\text{total weight}}$ = 3 × 5/4 = 3.75

# Task - Scheduling Problem

This is the problem of optimally scheduling unit-time tasks on a single processor, where each task has a deadline, along with a penalty / profit.

A unit-time task is a job, such a program to be run on a computer, that requires exactly one unit of time to complete.

Given a finite set 'S' of unit-time tasks.
A schedule for 'S' is a permutation of S specifying the order in which these tasks are to be performed.
The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2, and so on.

We are asked to find a schedule for 'S' that minimizes the total penalty incurred for missed deadlines or to maximize the total profit.

## Algorithm

1. Arrange all jobs in decreasing order of penalties / profit in an array J
2. Take an array A of size of max. deadline.
3. Start from right end of array A, search for the job which contains deadlines $\geq$ di in array J for every slot i.

eg: Given activities :

| Activities / tasks | Deadlines | Profit |
|---|---|---|
| 1 | 7 | 15 |
| 2 | 2 | 20 |
| 3 | 5 | 30 |
| 4 | 3 | 18 |
| 5 | 4 | 18 |
| 6 | 5 | 10 |
| 7 | 2 | 23 |
| 8 | 7 | 16 |
| 9 | 3 | 25 |

Max. Deadline = 7.

Sort tasks in Decreasing order of profit

| Tasks | Deadlines | Profit |
|---|---|---|
| 3 | 5 | 30 |
| 9 | 3 | 25 |
| 7 | 2 | 23 |
| 2 | 2 | 20 |
| 4, 5 | 3, 4 | 18, 18 |
| 8 | 7 | 16 |
| 1 | 7 | 15 |
| 6 | 5 | 10 |

Maximum deadline = 7.

| slots | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| | $T_2$ | $T_7$ | $T_9$ | $T_5$ | $T_3$ | $T_1$ | $T_8$ |

Greedily select the task with max. weight/profit, and put it in slots accordingly.

Max profit = 30 of task 3 with deadline 5
∴ put it in 5th slot.

Next profit = 25 of task 9 with deadline 3
∴ put it in 3rd slot.

And so on.

profit = 20 of task 2 with deadline 2.
but slot 2 is already filled. ∴ put it in 1st slot.

profit = 18 of task 4 with deadline 3
but slots are already filled ie. 3rd slot & all slots that are less than 3rd slot. ∴ cancel this task.

profit = 18, of task 5 with deadline 5, put it in 4th slot because slot 5 is already filled.

All slots are filled now.

Sequence : $T_2, T_7, T_9, T_5, T_3, T_1, T_8$

max profit : 20 + 23 + 25 + 18 + 30 + 15 + 16 = 147.

eg. Given, Cal. the max profit.

| Tasks | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Deadline | 3 | 2 | 3 | 1 |
| Profit | 9 | 7 | 7 | 2 |

$T_3 \rightarrow T_2 \rightarrow T_1$

$P = 23$

eg. Given:

| Tasks | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Deadline | 2 | 3 | 1 | 3 |
| Penalty | 10 | 30 | 60 | 40 |

$T_3 \rightarrow T_2 \rightarrow T_4$

Profit 130.

Penalty = 10.

eg *** Given.

| Tasks | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Deadline | 4 | 2 | 4 | 3 | 1 | 4 | 6 |
| Penalty | 70 | 60 | 50 | 40 | 30 | 20 | 10 |

Calculate the penalty & sequence.

Penalty = 50

$q_4 - q_2 - q_3 - q_1 - q_7 - q_6 - q_5$

eg

| Tasks. | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Deadlines | 2 | 2 | 1 | 3 | 3 |
| profit | 20 | 15 | 10 | 5 | 1 |

(1 – 2 – 4 profit = 40)

# Minimum Spanning Trees

A minimum spanning tree (M.ST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graphs that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

## Kruskal's Algorithm (cost or running time = $O(E \log V)$)

It finds a safe edge to add to the growing forest by finding, of all the edges that connects any two trees in the forest, an edge $(u,v)$ of least weight

```
MST-KRUSKAL (G, w)
1. A = ∅
2. for each vertex v ∈ G.V
3.     MAKE-SET (v)
4. sort the edges of G.E into nondecreasing order by weight
5. for each edge (u,v) ∈ G.E, taken in increasing order by w
6.     if FIND-SET (u) ≠ FIND-SET (v)
7.         A = A ∪ {(u,v)}
8         UNION (u,v)
9. return A
```

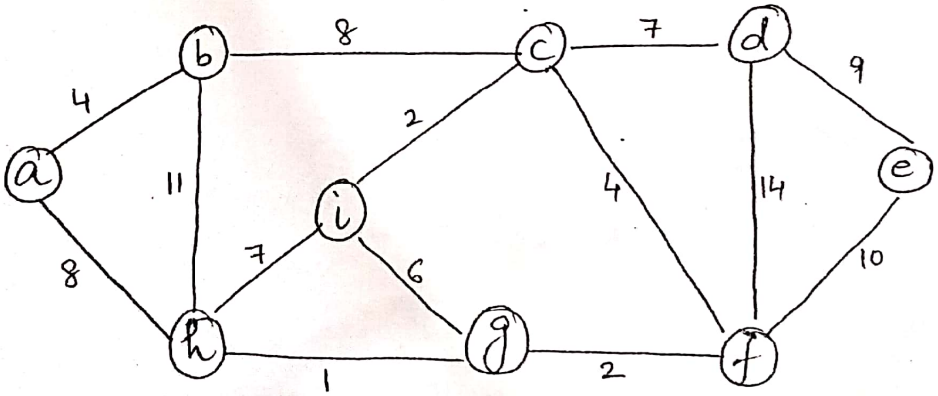
## Explaination

Lines 1-3 initialize the set A to the empty set and create |V| trees, one containing each vertex.

The for loop in lines 5-8 examins edges in order of weight, from lowest to highest. The loops checks for each edge (u,v) whether the endpoints u and v belong to the same tree. If they do, then edge (u,v) cannot be added to the forest without creating a cycle and the edge is discarded. Otherwise, the two vertices belong to different trees.

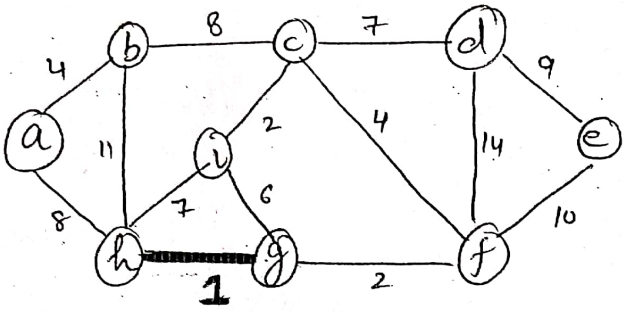
In line 7, it adds the edge (u,v) to A and line 8 merges the vertices in the two trees.

eg:

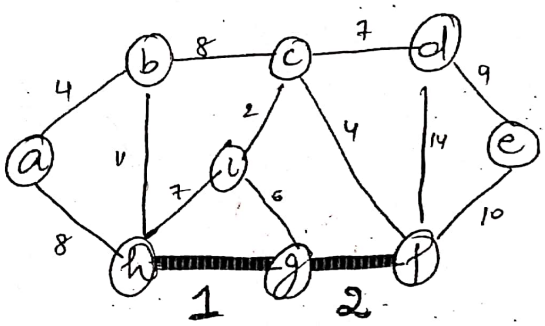



Consider the least weighted edge and add it the MST and Repeat (Don't add edges that make cycles in the graph)

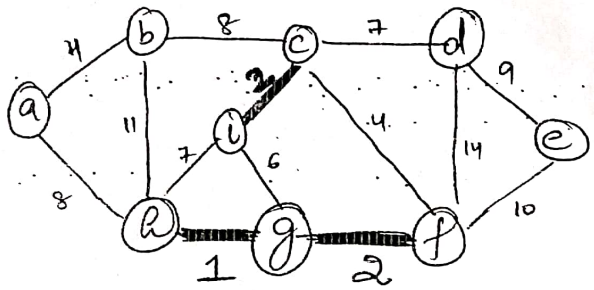
of 2



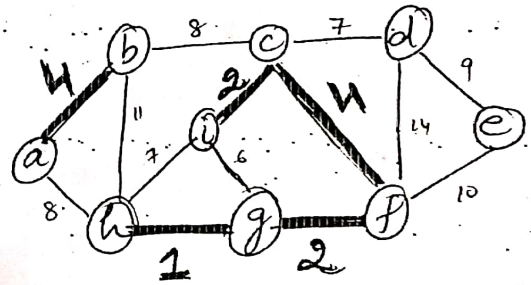

(a)



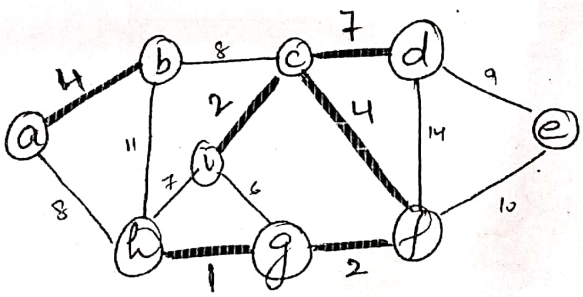

(b)



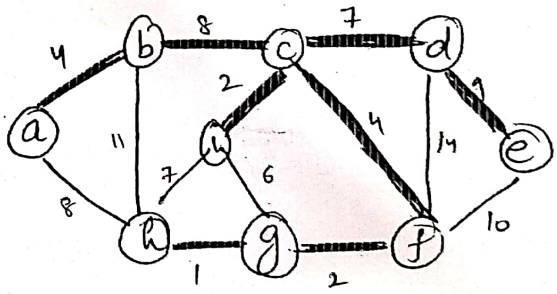

(c)



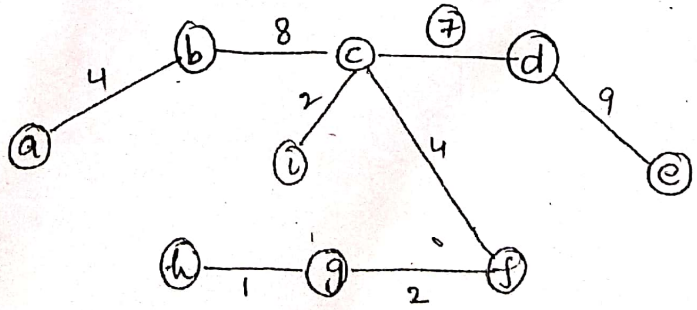

(d)




we will not choose wt. 6 ie-(i,g) bcoz it will form a cycle if added. similarly wt. 7 ie.(h,i). We will add (c,d) of wt. 7.




add '8' of (b,c) edge. ✓
don't add '8' of (g,h) edge ×
add '9' of (d,e) edge

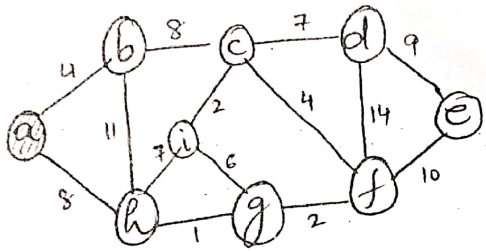



final MST

# PRIM's Algorithm. = $O(E \log V)$

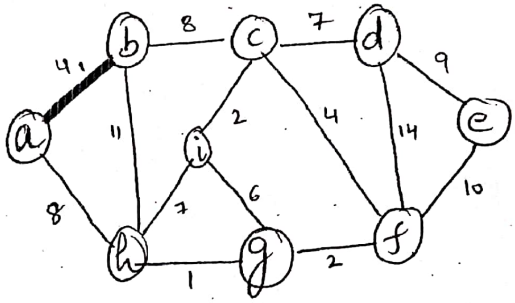
```
MST-PRIM (G, w, r)
1. for each u ∈ G.V
2.     u.key = ∞
3.     u.π = NIL
4.  r.key = 0
5.  Q = G.V
6.  while Q ≠ ∅
7.     u = EXTRACT-MIN (Q)
8.     for each v ∈ G.Adj [u]
9.        if v ∈ Q and w(u,v) < v.key
10.          v.π = u
11.          v.key = w(u, v)
```

Lines 1-5 set the key of each vertex to ∞ (except for the root r, whose key is set to 0 so that it will be the first vertex processed), set the parent of each vertex to NIL and initialize the min-priority queue Q to contain all the vertices.



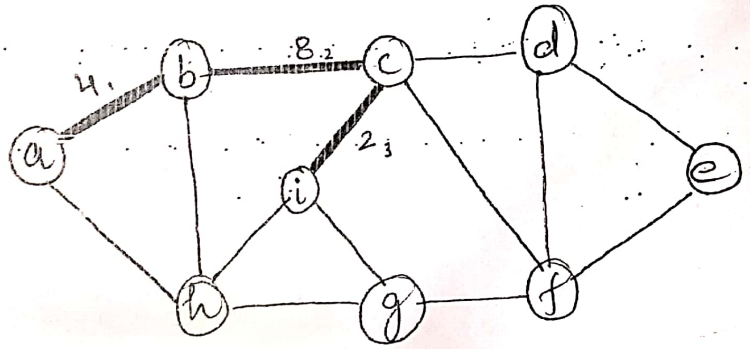

(1)

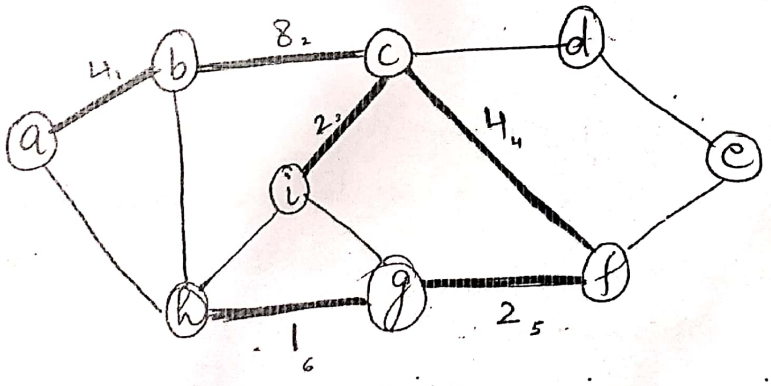
(a) is the root vertex.

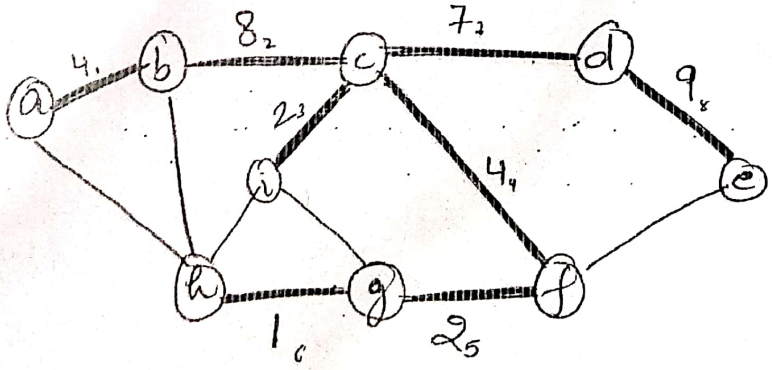



select the min. wt edge from root vertex.

Now select min wt. edge from that vertex ie. (b).



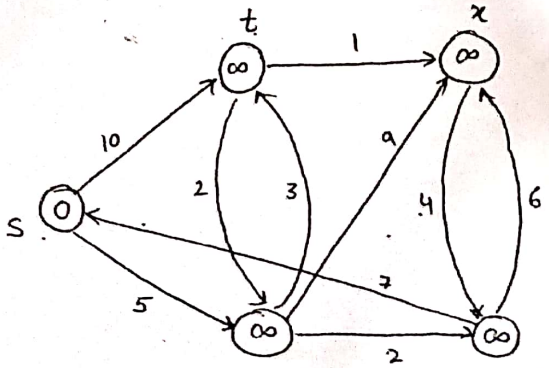



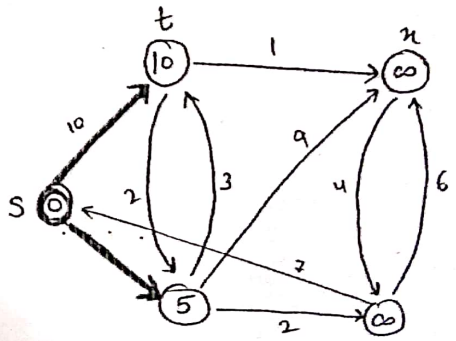
# Dijkastra's Algorithm

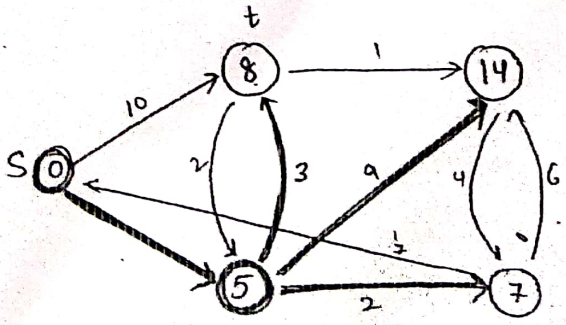
Dijkastra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are non-negative.

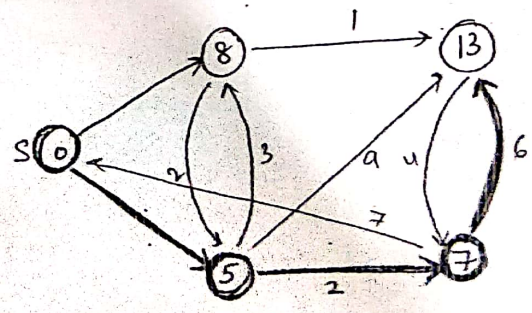
DIJKSTRA $(G, w, s)$ $= O(E \log V)$

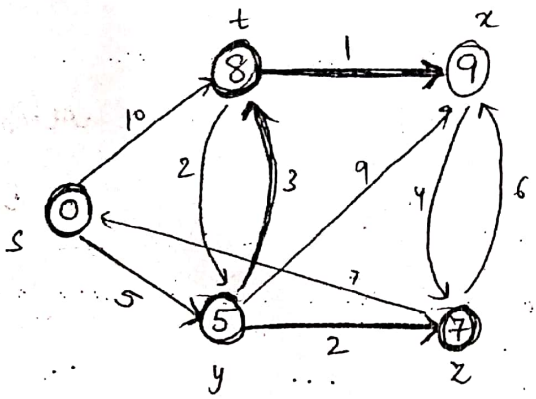
1. INITIALIZE-SINGLE-SOURCE $(G, s)$
2. $S = \emptyset$
3. $Q = G.V$
4. while $Q \neq \emptyset$
5. $\quad u =$ EXTRACT-MIN $(Q)$
6. $\quad S = S \cup \{u\}$
7. $\quad$ for each vertex $v \in G.Adj[u]$
8. $\quad$ RELAX $(u, v, w)$

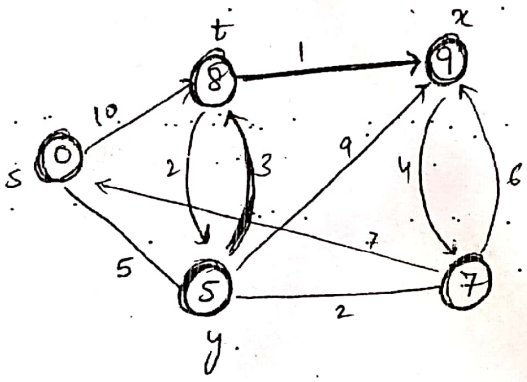



t
x
8
1
9
10
2
3
9
4
6
0
s
7
5
5
2
7
y
z


t
x
8
1
9
10
0
s
2
3
9
4
6
7
5
5
2
7
y

Guruja

# Bellman - Ford Algorithm

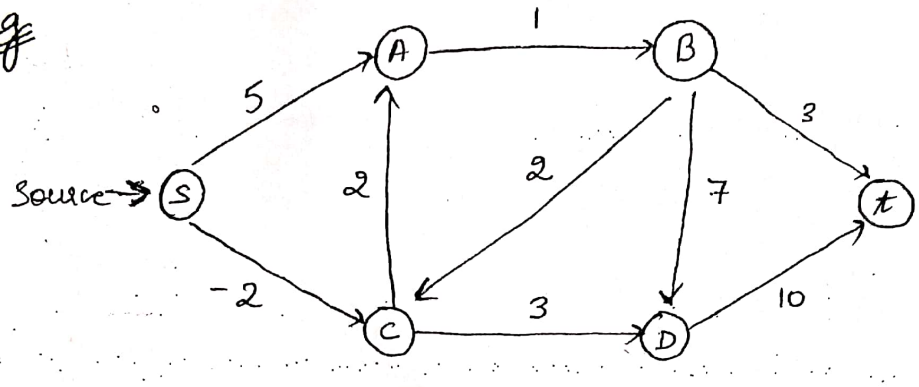
It solves the single - source shortest path problem in general case in which edge weights may be negative. Given a weighted, directed graph $G = (V, E)$ with source $s$ and weight function $w : E \rightarrow \mathbb{R}$, the Bellman ford Algorithm returns a boolean value indicating whether or not there is a negative weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicate that no solution exists. If there is no such cycle, the algo produces the shortest path and their weights.

```
BELLMAN - FORD (G, w, s)
1. INITIALIZE - SINGLE - SOURCE (G, s)
2. for i = 1 to |G.V| - 1
3. for each edge (u, v) ∈ G.E
4. RELAX (u, v, w)
5  for each edge (u, v) ∈ G.E
6. if v.d > u.d + w(u.v)
7. return false
8. return true.
```

u → start vertix
v → end vertix
u, v → edge
w → weights.

Running Time = $O(VE)$

eg




We will maintain a table which includes $d[v]$ and $\pi[v]$ where $d[v]$ is weight of vertices and $\pi[v]$ is the predecessor or parent of that vertex.

| | S | A | B | C | D | t |
|---|---|---|---|---|---|---|
| $d[v]$ | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| $\pi[v]$ | | | | | | |

If there are N vertices, then we will iterate N-1 times to get the shortest distance.
and
we do the $N^{th}$ iteration to check if there is any negative cycle.

$I^{st}$ iteration

| | S | A | B | C | D | t |
|---|---|---|---|---|---|---|
| $d[v]$ | 0 | ∞ 5 | ∞ 6 | ∞ -2 | ∞ 1 | ∞ 9 |
| $\pi[v]$ | 0 | S | A | S | C | B |

## 2nd Iteration

| | S | A | B | C | D | t |
|---|---|---|---|---|---|---|
| d[v] | 0 | ~~∞~~ 0 | ~~∞~~ 1 | -2 | 1 | ~~∞~~ 4 |
| π[v] | 0 | ~~S~~ C | ~~A~~ A | S | C | ~~B~~ B |

and so on till 5th iteration.

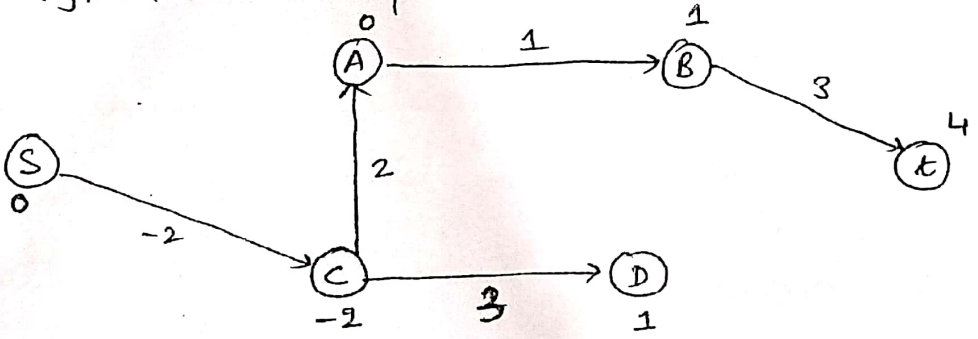
There is No change

## 6th iteration

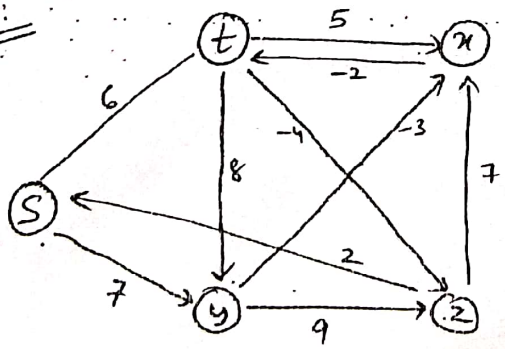
| | S | A | B | C | D | t |
|---|---|---|---|---|---|---|
| d[v] | 0 | 0 | 1 | -2 | 1 | 4 |
| π[v] | 0 | C | A | S | C | B |

There is no -ve cycle.

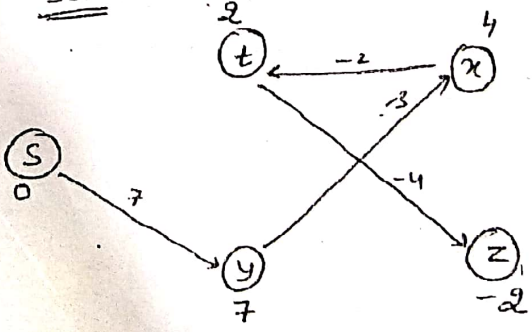
final shortest path is :



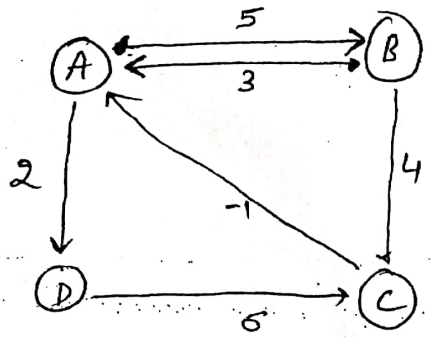

eg:




Sol^n



eg




Soln