Algorithms – Searching, Greedy Algorithms

SEARCHING ALGORITHMS

In computer science, a search algorithm is an algorithm that retrieves information stored within some data structure. Here we are going to look at linear search and binary search.

Linear Search

The name itself says that we are going to go in a linear fashion. This algorithm is also called **sequential search.**

It sequentially checks each element of the list for a target value until a match is found or until all the elements have been searched.

It works on both sorted and unsorted data.

Problem Statement: Given a list L of n elements with values or records $L_0 \ldots L_{n-1}$ and target value T, the following subroutine uses linear search to find the index of the target T in L.

Basic algorithm:

1. Set i to 0
2. If L_i = T, the search terminates successfully; return i
3. Increase i by 1
4. If i < n, go to step 2. Otherwise, the search terminates unsuccessfully

Time complexity

1. **Approach 1**
    a. **Best case:** number of iterations = 1
    b. **Worst case:** number of iterations = n+1 (we didn't find the element at all)
    c. **Average case :** number of iterations = best + worst / 2 = O(n+2/2) = O(n)
2. **Approach 2: Using recurrence relation**

    T(n) = T(n-1) + 1 if n > 1

    T(1) = 1

    Then by back substitution T(n) = O(n)

Implementation of linear search on a linked list or array is going to give the same worst case time which is O(n)

Let's see the next searching algorithm called Binary Search

Binary Search

Binary search can be only applied when:

1. *The data is stored in an array*
2. *The data is sorted*

Binary search cannot be applied when:

1. *The data is stored in a linked list and sorted*
2. *The data is unsorted*

Binary search, also known as **half-interval search** or **logarithmic search,** is a search algorithm that finds the position of a target value within a sorted array. It does not work on unsorted array.

- Binary search works on the sorted arrays, it begins by comparing the middle element of the array to the target value
- If target value matches the middle element, its position in the array is returned
- If the target value is less than the middle element, then, the search is continued on the left half of the array otherwise the search is continued on the right half of the array

Problem Statement: Given an array A of n elements with values or records $A_0 \dots A_{n-1}$, sorted such that $A_0 \leq \cdots \leq A_{n-1}$, and target value T, the following subroutine uses binary search to find the index of T in A.

Algorithm:

1. Set L to 0 and R to n-1
2. If L > R, the search terminates as unsuccessful
3. Set m (the position of the middle element) to the floor (the largest previous integer) of (L+R)/2
4. If T = A_m, the search is done; return m
5. If T > A_m set L to m+1 and go to step 2
6. If T < A_m set R to m-1 and go to step 2

Time complexity:

Recurrence relation: T(n) = T(n/2) + 1 if n > 1

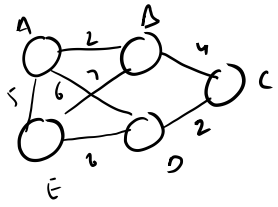
T(n) = 1 if n = 1

⇨ **T(n) = O($\log_2 n$)**

GREEDY ALGORITHMS

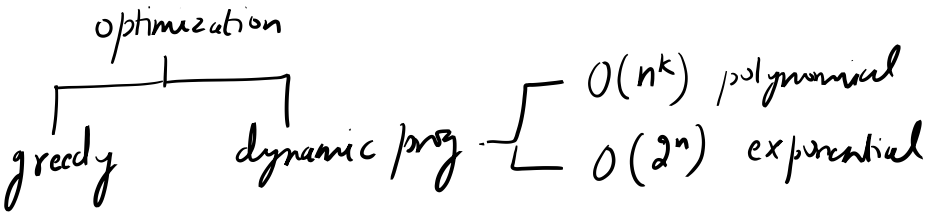
There are problems where we need to optimize some property like; “minimum cost”, “minimum spanning tree”, “maximize profit”, “maximize the reliability”, “minimize risk” etc. Where we want to maximize or minimize, basically we say that these problems are optimization problems.

There are various ways to solve such problems. Let’s understand with an example. Let’s say I have a graph like below. And we want to find out the smallest path from A to C.



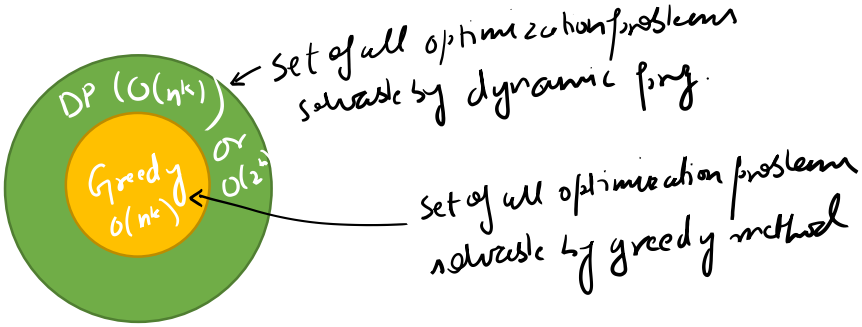
One way to do this is to use **exhaustive search** which means find out all the paths possible and select the minimum from those paths. This is going to be an exponential time or $O(2^n)$ algorithm and we don't want that. We want to see if we can do something better than this.

To solve such optimization problems, we found that there are two programming paradigms.



Using greedy method, we will not be able to solve all the optimization problems. The problem should have some properties then only we can apply greedy method.

We will be able to solve all the optimization problems using dynamic programming, but some programs even after applying DP will give result in exponential time. Therefore, we use dynamic programming where we can get result in polynomial time otherwise it is as good as applying exhaustive search.



Let's see the problems for which greedy algorithms and dynamic programming can be applied to understand them better.

KNAPSACK PROBLEM (Greedy)

Knapsack means a bag. The problem statement is like this.

Problem Statement: We are given a bag that has a holding capacity of 20 units. We are given three objects, ob1, ob2 and ob3. Every object has got some profit provided that you sell respective units of that object. What are the objects you will place in the knapsack to get the maximum profit?

	Ob1	**Ob2**	**Ob3**
Profit	25	24	15
Weight	18	15	10

M (capacity of the bag) = 20 units

1. If we try to put all the objects in the bag, we can't clearly because the maximum capacity is 20.

2. So what I can do next is that I will try to be ***greedy about profits.*** In this way I will try to add those objects first which will give me the maximum profits and move to the next maximum profit making object. Object 1 is giving me the maximum profit after that object 2 is giving me maximum profit, so I am taking all the Object 1 and will fill the remaining space in the knapsack with the object2.

	Weights	Profits
Object 1	18	25
Object 2	2	24*2/15
Total	**20**	**28.2**

But, is this the maximum profit? To answer this, we can go to the next stage.

3. I can now be ***greedy about weights***. Here I will add those objects first which have the least weight and then go to the next object which has the second least weight.

	Weights	Profits
Object 3	10	15
Object 2	10	24 * 10/15
Total	**20**	**31**

I found that if I am being greedy about weights, I am getting more profit. But is this the maximum profit again?

4. Next I can go with ***how much profit I am getting per object/weight.*** This way I can go about the **Profit/Weight ratio**

	Ob1	**Ob2**	**Ob3**
Profit	25	24	15
Weight	18	15	10
Profit/Weight	25/18 = 1.4	24/15 = 1.6	15/10 = 1.5

Intuitively, I will put Object 2 first because I am getting maximum profit per object in this category.

	Weights	Profits
Object 2	15	15*1.6 = 24
Object 3	5	5 * 1.5 = 7.5
Total	**20**	**31.5**

So I found out that neither by being greedy about profit or weights I am getting the maximum profit. I get the maximum profit by using the per weight profit of all the objects and then putting them in the decreasing order of profit.

Let’s see the algorithm and analyze it.

Greedy knapsack algorithm

```
Greedy Knapsack
{
        For i = 1 to n:                                   O(n)
                compute pi/wi
        Sort objects in non-increasing order of p/w       O(nlogn) best complexity I used sorting (merge sort)

        for i=1 to n from sorted list                     O(n)
                if(m > 0 && wi <=m)
                        m = m – wi
                        P = P + pi
                else
                        break;

        if (m>0)                                          O(1)
                P = P + pi(m/wi)
}
```

Time complexity: T(n) = O(n) + O(n) + O(nlogn) + O(1)

T(n) = O(nlogn) (worst case time)

I can also use heap in this problem but **for the worst case** the time complexity will still be O(nlogn).

For using heap sort, we will first make the max heap which takes O(n) and then delete or extract the max from the heap which takes O(logn) time. And this deletion in **worst case** may happen n times.

One can argue that after deletion the depth of the tree may reduce. Yes, that can indeed be the case in which the time complexity may be O(nlog(n-k)) but since we are talking about the **worst case time complexity,** the max heap may be a complete binary tree or almost complete binary tree with (n/2) leaf elements.

Therefore, the complexity = O(n/2 log(n)) which is still **O(nlogn)**

Example for knapsack:

Question: Find the maximum profit.

M = 15, N = 5

Objects	1	2	3	4	5
Profit	2	28	25	18	9
Weight	1	4	5	3	3

The p/w is

Object	1	2	3	4	5
Profit/weight	2	7	5	6	3

Objects sorted by profit/weight = {2, 4, 3, 5, 1}

Now, M = 15

Knapsack = all object2 + all object4 + all object3 + all object5

Profit = 28 + 18 + 25 + 9 = 80

Special case of knapsack: If all of the weights are same, we need not sort them by profit/weight ratio. We can sort them by the profits then and that will be enough

Let's move on to our next greedy method which is called Huffman Coding.

HUFFMAN CODING (Greedy)

Let's say we have a file and we want to store some characters in it. Now these characters are going to take space, obviously. If the characters are "abcd" in the entire file with nothing more than that and I give the two bit representation for each character like:

a = 00 then "abcd" will be 00011011 in the file
b = 01
c = 10
d = 11

Now, if the file has 100 characters, then the number of bits required to represent all of these characters will be: 100 x 2 = 200 bits

Now if the distribution of the characters is not even (number of characters in the file are not same) then we can use Huffman Codes to reduce the size of the file or reduce the number of bits all the characters in the file will take collectively.

To do this:

1. Count the number of characters grouped by character type in the file
   Let's say:
   A = 50
   B = 40
   C = 5
   D = 5
2. The character which has the highest frequency, represent it using the least number of bits.
   A = 0 C = 110
   B = 10 D = 111

We are using codes which are un-equal in the size. This is also called **un-equal sized encoding/different sized encoding/variable sized encoding**.

Now when we use this encoding, we need to make sure that proper decoding is possible. Therefore, each encoding will be unique for each character. For example, if A's encoding is 0 then, any other character's encoding can't start from 0.
Technically we are using **prefix codes** here, which means that if I am using a pattern to denote a character, then that pattern should not be a *prefix* of any other pattern.

Are we getting any gain from such an encoding?

Character	Frequency	Bit Representation	Space taken
A	50	0	50x1 bit = 50 bits
B	40	10	40x2 bits = 80 bits
C	5	110	5x3 bits = 15 bits
D	5	111	5x3 bits = 15 bits
Total	**100**	**-**	**160 bits**

Clearly we are getting a gain of 40 bits. So we are being greedy about the allocation of space for characters here.

Therefore, in most of the problems where the frequency of characters in a file is different, Huffman coding is going to give us better results in terms of storage. Therefore, Huffman coding is an algorithm which is used to find the codes which will help us reduce the size of the data.

Let's see the algorithm for Huffman codes.

```
HUFFMAN(C)
{
        N = |c|
        make a min-heap Q with 'c'

        For i = 1 to n-1
                allocate a new node 'z'
                z.left = x = Extract_min(Q)
                z.right = y = Extract_min(Q)
                z.freq = x.freq + y.freq
                INSERT(Q,z)

        Return (Extract_min(Q))
}
```

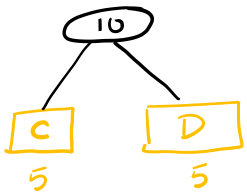
Let us understand this with the help of an example.

Let's say I have a file with 100 characters and their frequency is:

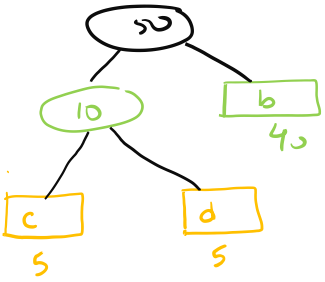
A= 50, B = 40, C = 5 and D = 5, and N(number of distinct characters) = 4

Then, the following are the steps to get the Huffman code for the characters.

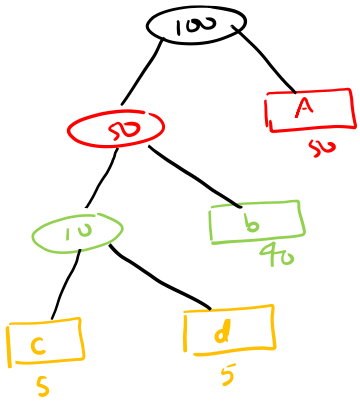
1. Take the first two characters having the least frequency and make them the children of a root in a tree. Here C and D are the characters with least frequencies, the tree will look like.



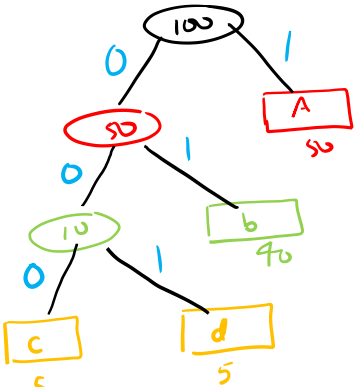
2. Now take this new node and the next character apart from C and D with the least frequency. That will be B with a frequency of 40. Combine these two in a new root node. Least one is the left child.



3. Now take this new node and the next character apart from B, C and D with the least frequency. That will be A with a frequency of 50. Combine these two in a new root node.



4. Now each side of the root node is given either 0 or 1. Let's give the left size 0 and right side 1.



5. Traverse through the tree from the root node to get the Huffman code.
   A = 1
   B = 01
   C = 000
   D = 001

(*steps 2 and 3 should be repeated until we have represented all the characters in the given problem)*

Total bits used without Huffman code = 200

Total bits used with Huffman coding = 160

Bits per character before Huffman coding = 200/100 = 2 bits/character

Bits per character after Huffman coding = 160/100 = 1.6 bits/character

We can also get the number of bits used by a character by calculating **Weighted external path length.** This is calculated as below:

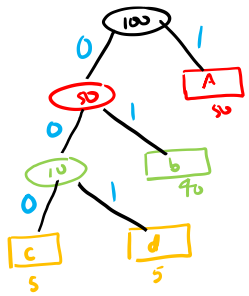
For C: "what is the length of this leaf from the root" = 3

For D: "what is the length of this leaf from the root" = 3

For B: "what is the length of this leaf from the root" = 2

For A: "what is the length of this leaf from the root" = 1

(refer the tree on the right hand side)



Then for each **weighted external path length** you can then multiply with the frequency of the character to get the total bits consumed by this file.

50x1 + 40x2 + 5x3 + 5x3 = 160 bits.

"How can we build a tree so that we can decrease the weight of the external path length? *This is also equivalent to making a tree for the Huffman coding, this way also a question can be asked.*

Let's take another example:

A = 40 D = 5

B = 30 E = 3

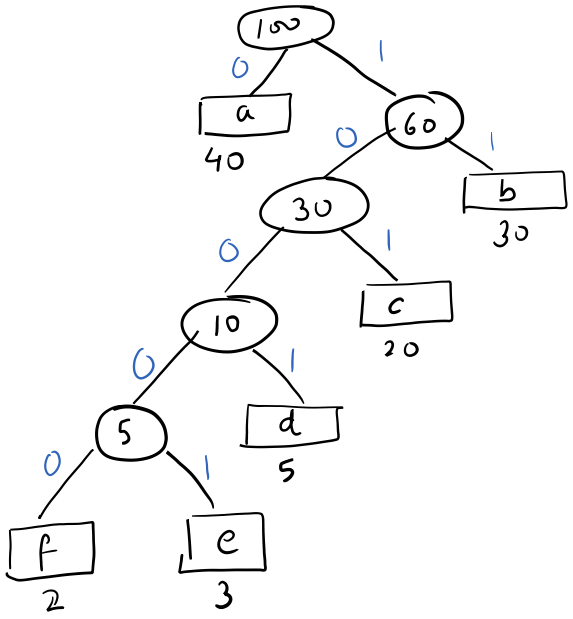
C = 20 F = 2

Let's store them in a data structure:

40	30	20	5	3	2

Take the two most minimum and create the nodes, keeping the minimum one at the left side.

In the next step, these two nodes will be replaced by one node (3, 2) will become (5) and then take the next minimum element from the data structure and repeat this process as we did earlier. We will get the following tree out of it. REMEMBER TO PUT THE SMALLER CHILD ON THE LEFT AND BIGGER CHILD ON THE RIGHT. Then put 0 and 1 on left-right or right-left. I'll put 0 on left and 1 on right.



a = 0 — 1 bit × 40 = 40

b = 11 — 2 bits × 30 = 60

c = 101 — 3 bits × 20 = 60

d = 1001 — 4 bits × 5 = 20

e = 10001 — 5 bits × 3 = 15

f = 10000 — 5 bits × 2 = 10

Total = 100 characters

= 205 bits

We are going to get the prefix code for each character because in the tree each character is a leaf and thus has no children.

Let's see the time complexity of the algorithm. Let's see step by step what has happened.

```
HUFFMAN(C)
{
        N = |c|                                  ← O(1)
        make a min-heap Q with 'c'               ← O(n)
        For i = 1 to n-1                         ← n-1 times
                allocate a new node 'z'
                z.left = x = Extract_min(Q)      ← O(log n)
                z.right = y = Extract_min(Q)     ← O(log n)
                z.freq = x.freq + y.freq
                INSERT(Q,z)                      ← O(log n)
        Return (Extract_min(Q))                  ← O(1)
}
```

$T(n) = 2O(1) + O(n) + 2(n-1)O(\log n) + (n-1)O(\log n)$

$= O(n) + 3(n-1)O(\log n)$

$= O(n) + O(n \log n)$

$= O(n \log n)$

Huffman Coding takes O(nlogn) time.

Space Complexity: We are going to represent the data using the tree. The space required will be ~ O(n)

Therefore, space complexity will be **O(n).**

Question: Instead of using heap why can't I sort the elements to get the minimum?

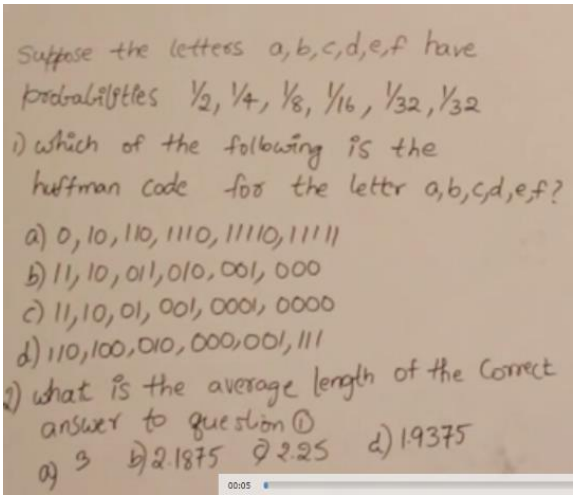
1. Sorting will take O(nlogn) time
2. After deletion, elements have to be inserted in the array, that insertion can take O(n) time.
3. **O(n) times insertion will be done (n-1) time for loop iteration. Therefore if sorting is taken into picture the time complexity will be $O(n^2)$ and we don't want that.**

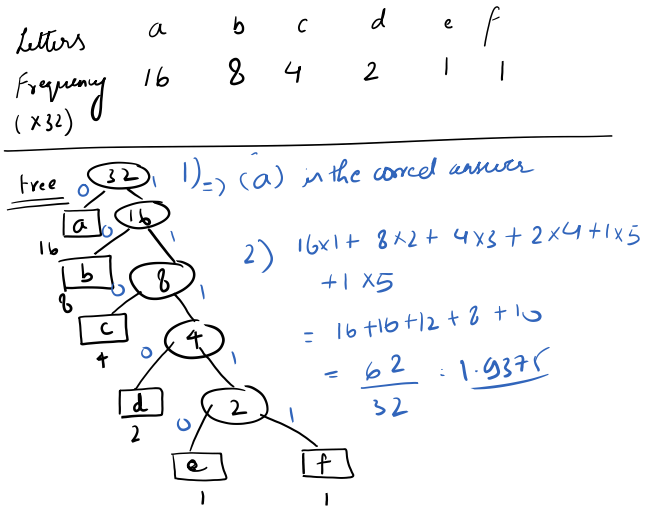
Question: So when should I use sorting and when should I use heaps?

Answer: When there is no requirement of putting an element, heaps can be replaced by sorting algorithms. Otherwise, heaps will be useful.

Let's see some questions on Huffman coding.

Question1:





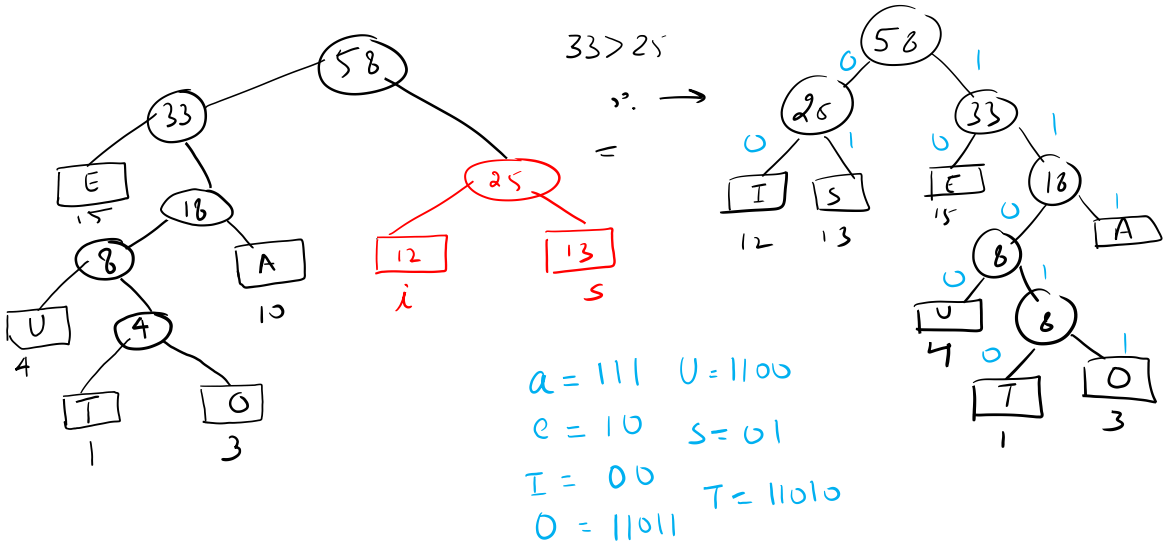
Sometimes it happens that the newly created node need not be chosen immediately after its creation. Let's see and example on that.

Question 2:

A	E	I	O	U	S	T
10	15	12	3	4	13	1

The tree for the given question can be seen below.

A	E	I	O	U	S	T
10	15	12	3	4	13	1



Let's see another greedy algorithm

JOB SEQUENCING WITH DEADLINES(Greedy)

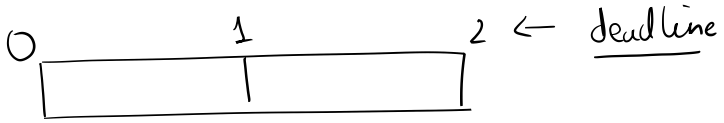
In the job sequencing with deadlines the problem is as follows:

Problem: There are some jobs given and the profit associated with them. The jobs will give those profits if the job is completed within a given deadline. If every job takes one unit of time what is the maximum profit that we can get?

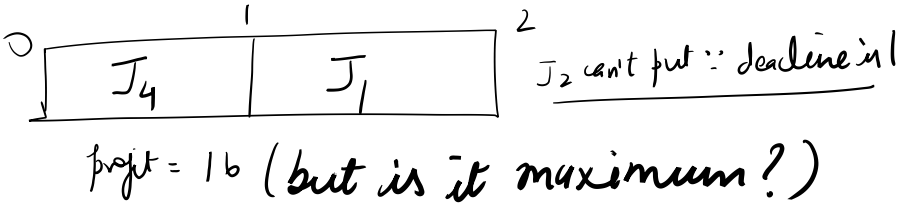
Let's say the jobs, profits and deadlines are given as below.

Jobs	J1	J2	J3	J4
Deadlines	2	1	1	2
Profits	6	8	5	10

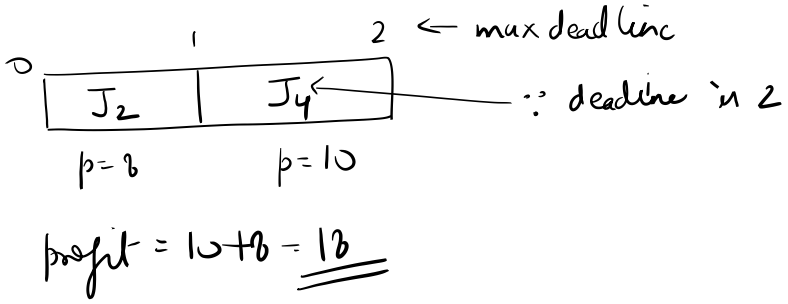
What I want to do is to do jobs in such a way that I get the maximum profits but the jobs should be done within their deadlines. Every job will take 1 unit of time to get completed.



Way 1: To solve this problem I can select the jobs which have their profits arranged in decreasing order.



Way 2: I can select the jobs according to the maximum profits and then place them as far away as possible according to their deadlines.



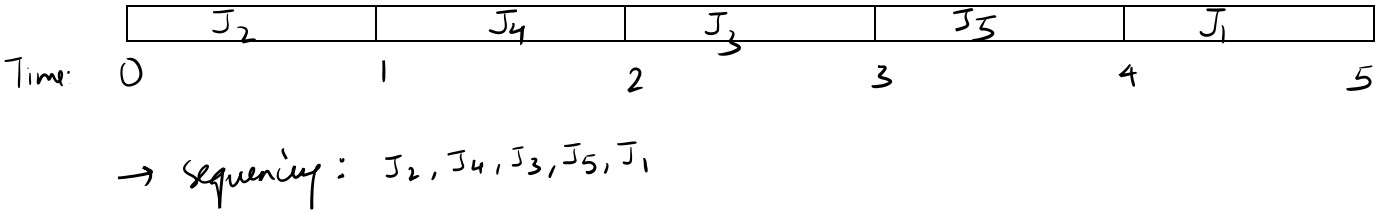
Second way of solving this problem is going to produce the MAXIMUM PROFIT

Let's see another example

Example 2: The jobs, profits and deadlines are given below. Schedule the jobs to get the maximum profit.

Jobs	J1	J2	J3	J4	J5	J6
Deadline	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Max deadline = 5, therefore I can have max of 5 jobs in the sequencing table.



Steps to follow and their complexity analysis:

1. Create the Job Sequencing array with *number of elements = max deadline/max number of jobs.* **If the deadline is given very high and jobs are small, we don't need that big of an array. So be careful when deadlines are very high given <3**
2. Sort out the jobs in the decreasing order of their profit. **[O(nlogn)]**
3. Take each job and then start from the point where its deadline is given and search for an empty location to the left of that deadline. This searching can take O(n) time and for n such jobs, the complexity can be **$O(n^2)$**
4. Place the jobs in the empty position available. If empty position is not available, the job can't be placed.
5. Do this until the job sequencing array is full.

Time complexity = O(nlogn) + $O(n^2)$ = $O(n^2)$

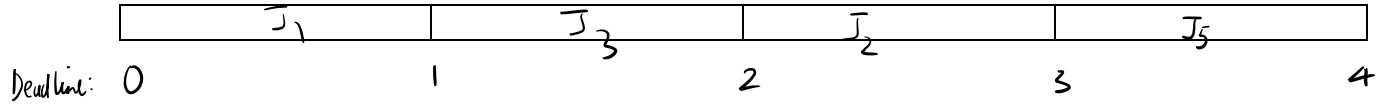
Let's see some more examples

Example 3: The jobs, profits and deadlines are given below. Schedule the jobs to get the maximum profit.

Jobs	J1	J2	J3	J4	J5
Profits	2	4	3	1	10
Deadlines	3	3	3	4	4

Answer: max deadline = 4

Job sequencing array:



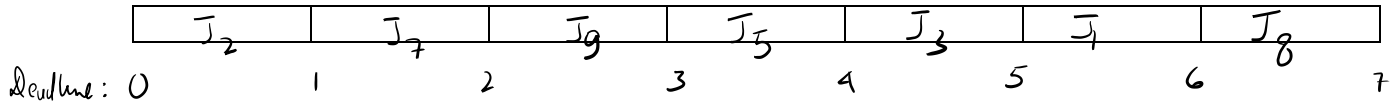
Sequencing = J_1, J_2, J_3, J_5 Profit = 2 + 3 + 4 + 10 = 19

Example 4: The jobs, profits and deadlines are given below. Schedule the jobs to get the maximum profit.

Jobs	J1	J2	J3	J4	J5	J6	J7	J8	J9
Profit	15	20	30	18	18	10	23	16	25
Deadline	7	2	5	3	4	5	2	7	3

Max deadline = 7 Jobs in descending order of profit: J3, J9, J7, J2, J4, J5, J8, J1, J6

Sequencing array



Sequencing = $J_2, J_7, J_9, J_5, J_3, J_1, J_8$ Profit = 20 + 15 + 30 + 18 + 23 + 16 + 25 = 157

Let's see our next greedy optimization problem.

OPTIMAL MERGE PATTERNS (Greedy)

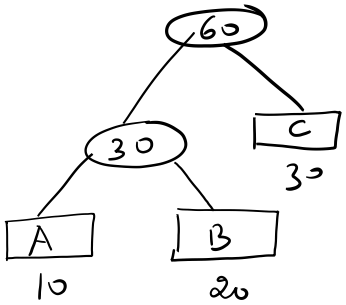
In optimal merge patterns, we have some files with some non-uniform records that we want to merge together. All the files have some data which is in the sorted format, now what we want to do is that we want to merge all of them into a one file. The catch is that this merging should happen with the least number of record movements.

If you recall, merging of records can be done using the MERGE operation of merge sort algorithm. However, we are concerned about the minimum record movement here when the merge operation is done.

Let's understand with an example.

FILES	A	B	C
NO. of RECORDS	10	20	30

Let's do the merging like below:



The movements of the records is like this.

All the 10 records of the file A have been moved once in the parent node of the tree and all the 20 records of the file B have been moved once to the parent node.

This parent node and the records in C have moved their 30 records each once to the final root node of the tree which is the new file size.

Therefore, total movements = 10x1 + 20x1 + 30x1 + 30x1 = 90

BUT IS THIS THE LEAST NUMBER OF MOVEMENTS?

The way in which this tree has been created to get the least number of movements (yes, 90 is the least number of movements needed for optimal merging in this question) has already been seen by us when we were creating HUFFMAN CODES.

The only difference between optimal merging and Huffman code tree creation is that in optimal merging we don't need to worry about the lesser of the nodes being on the left or right position because we want to just count in this problem.

Since this problem is similar to Huffman code creation, the algorithm is also going to be the same. So I am not elaborating much here. Let's see the time complexity.

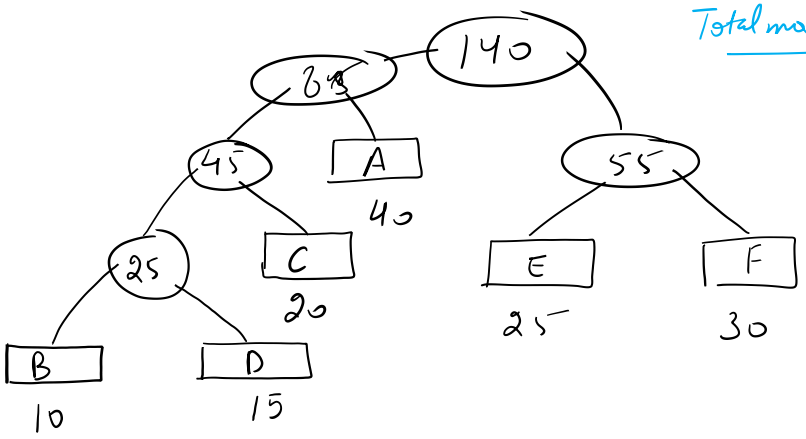
Time complexity

1. Given 'n' files first create a minimum heap (takes O(n) time)
2. Take the two minimums and merge them. To take two minimum, the time taken will be O(logn).
3. We then merge them together and put it back. Time to insert is O(logn)
4. Step 2 and 3 happen (n-1) time. Therefore:

Final time complexity: O(n) + (n-1)O(logn) + (n-1)O(logn) = O(n) + 2(n-1)O(logn) = **O(nlogn)**

Example: What is the optimal merge pattern for the files given below?

File	A	B	C	D	E	F
Records	40	10	20	15	25	30



Total moves: $10 \times 4 + 15 \times 4 + 20 \times 3 + 40 \times 2 + 25 \times 2 + 30 \times 2$

$= 40 + 60 + 60 + 80 + 50 + 60$

$= 340$ moves

Let's move on to the next greedy algorithm which will be on graphs.
Note: the topic of graphs will be discussed in detail in either graph theory or data structures. We will see here some theory of graph required for the respective greedy algorithm on graph.

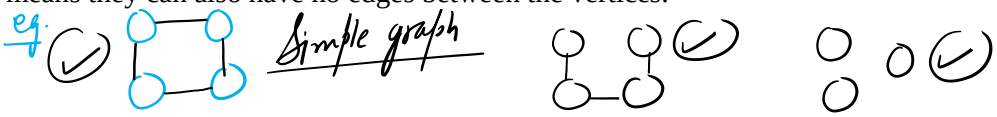
SPANNING TREES AND KIRCHOFF THEOREM

Before moving on to the algorithms, let's clear the theory required to understand the algorithms.

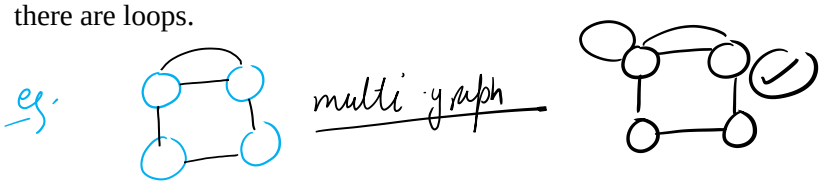
Graph: A graph is a collection of vertices and edges given as an ordered pair (V,E).

Types of graphs can be:

1. **Simple graph:** A graph is a simple graph if between two vertices there is at most one edge. This means they can also have no edges between the vertices.



2. **Multi graph:** A graph is a multi-graph if between any two vertices there is more than one edge or there are loops.

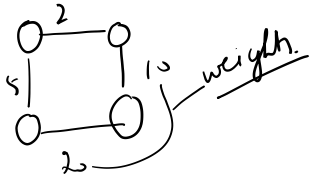


In this topic we care about simple graph.

Properties of simple graphs

1. The minimum number of edges present in a graph = 0
2. The maximum number of edges present in a graph = give all possible edges = $n_{C_2} = \frac{n(n-1)}{2}$ where n is the number of vertices
3. In worst case I can say that
   E = O(n(n-1)/2) = O(n^2) = O(v^2) where v the number of edges
   If I take log on both sides
   v = O(logE) in the worst case.

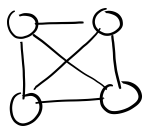
3. **Simple Weighted Graphs**
   In the simple weighted graphs, the edges are given some weights. These meaning of these weights depends on the context.



4. **Simple Unweighted graphs:** The simple graphs which do not have the weights associated with the edges.
5. **Simple Labeled graph:** If all the vertices of a graph are labeled with a name
6. **Simple Unlabeled graph:** If all the vertices of a simple graph are not labeled with anything
7. **Simple Complete graph:** If all the vertices are connected to all the other vertices in a graph with at most one edge, that graph is called a simple complete graph. It is denoted by **K_n** where n is the number of nodes.

SPANNING TREE

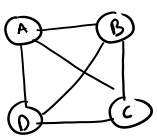
An interesting question that can be asked is *"Given a simple graph what is the minimum number of edges to connect all of the vertices?"*



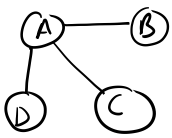
there are 6 edges but do we need all of them to connect all the vertices?

If I answer this question, the result is going to be a **SPANNING TREE.**

A spanning tree is the graph which has minimum number of edges in it such that all the vertices/nodes are connected and any node can be reached from any other node by traveling on a single edge or more than one edge.



spanning tree ⟹



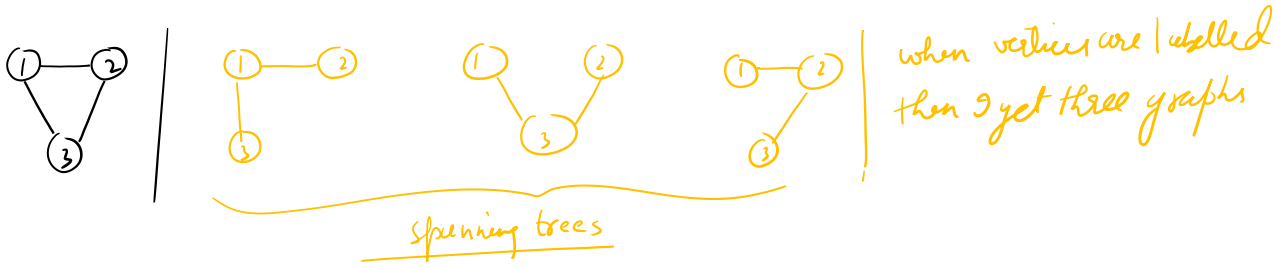
minimum edges = 3

THE MINIMUM NUMBER OF EDGES REQUIED TO CONNECT "n" NODES IN A GRAPH IS "n-1"

Now we know that if a graph contains V vertices then it need a minimum of (V-1) edges to connect all the vertices.

A **spanning tree is also a subgraph of the given graph** from which the spanning tree has been made.

So if I have V vertices in a graph, then how many spanning trees are possible?



In case if the vertices are not labeled, all these spanning trees will be isomorphic.

For K_3 graph number of spanning trees = 3

For K_4 graph number of spanning trees = 4

…

For K_n graph, number of spanning trees = n^{n-2}

What if the graph is not complete?

If the graph is not complete, then the maximum number of spanning trees can't be determined by n^{n-2} because there will be at least one edge missing.

To find out the maximum number of spanning trees for a graph which is not complete we use the **Kirchhoff's Theorem**

KIRCHHOFF'S THEOREM

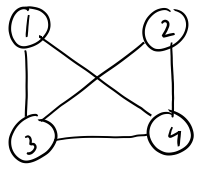
To find out the maximum number of spanning trees we use Kirchhoff's theorem. Here are the steps to do that.

1. Find out the adjacency matrix of the given graph. This graph will be (VxV). Let's call it A.
2. **Theorem**
    a. **Replace all the diagonal 0's elements in A with the degree of the vertex**
    b. **Replace all the non-diagonal 1's with -1**
    c. **Keep all non-diagonal zeros as zeros.**

3. In this modified matrix A, find out the Cofactor of any one of the element.
    a. Cofactor of element A_{11} is calculated as follows
        i. Ignore the row 1 and column 1
        ii. Find the determinant of the remaining matrix
4. The value of the cofactor is the number of spanning trees possible.

Let's take some examples:

Example 1: How many spanning trees are possible for the graph given below



① Adjacency matrix

A	1	2	3	4
1	0	0	1	1
2	0	0	1	1
3	1	1	0	1
4	1	1	1	0

② Theorem application

A	1	2	3	4
1	2	0	-1	-1
2	0	2	-1	-1
3	-1	-1	3	-1
4	-1	-1	-1	3

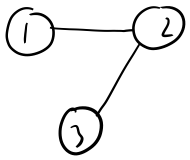
③ Cofactor of A_{11}

$= 2(9-1) - (-1)(-3-1) + (-1)(1+3)$

$= 16 - 2 - 4 = 10$

Therefore, for this graph, 10 spanning trees are possible.

Example 2: How many spanning trees are possible for the graph given below



① Adjacency matrix

A	1	2	3
1	0	1	0
2	1	0	1
3	0	1	0

② Theorem

A	1	2	3
1	1	-1	0
2	-1	2	-1
3	0	-1	1

③ Cofactors

Let us take Cof. of A_{11}

$(2-1) = 1$

Let's understand some more about the spanning trees. Now we know what spanning trees are and now we can see some applications of spanning trees and other algorithms. One of the applications of spanning tree is whenever we want to build something like a network topology we may want to connect all the computers with the least number of wires. But if the problem is like: I want to connect the computers with the minimum total length of the wire (the length of the wire is given for each edge of the tree) then I need to find out **Minimum Spanning Tree.** This means the problem finally translates to finding a minimum spanning tree in a ***weighted graph.*** Let's see how it can be done.

MINIMUM COST SPANNING TREE

Finding a minimum cost spanning tree is not as straightforward as finding the spanning tree because now we want to find out from all the possible spanning trees of a graph, which one will have the minimum cost.

Revision: For a K_n graph the number of spanning trees = n^{n-2}

Therefore, this problem is of $O(n^{n-2})$ which is exponential. We are lucky that there are two algorithms that tackle the problem statement ***"Given a weighted graph what is the minimum spanning tree".***

1. Prim's algorithm
2. Kruskal's algorithm

Let's see Prim's first.

PRIM'S ALGORITHM FOR FINDING OUT MINIMUM COST SPANNING TREE(Greedy)

The informal steps to apply prim's algorithm are:

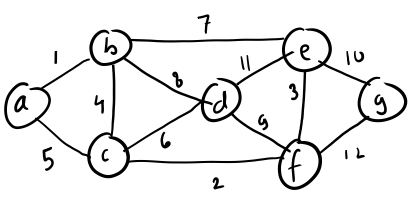
1. Make the skeleton graph meaning draw the graph in the question without its edges.
2. Take the edge which has the minimum weight and join the two nodes containing that edge
3. Look at the connections of these two nodes and add a new node based on the minimum weight
4. The new graph obtained will be a tree with 3 nodes, repeat this process until all the nodes are added in the final graph.

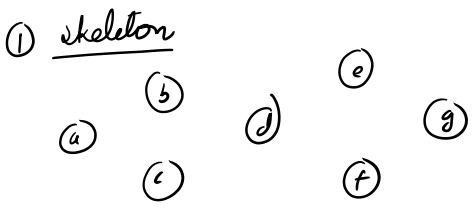
Important things to note:

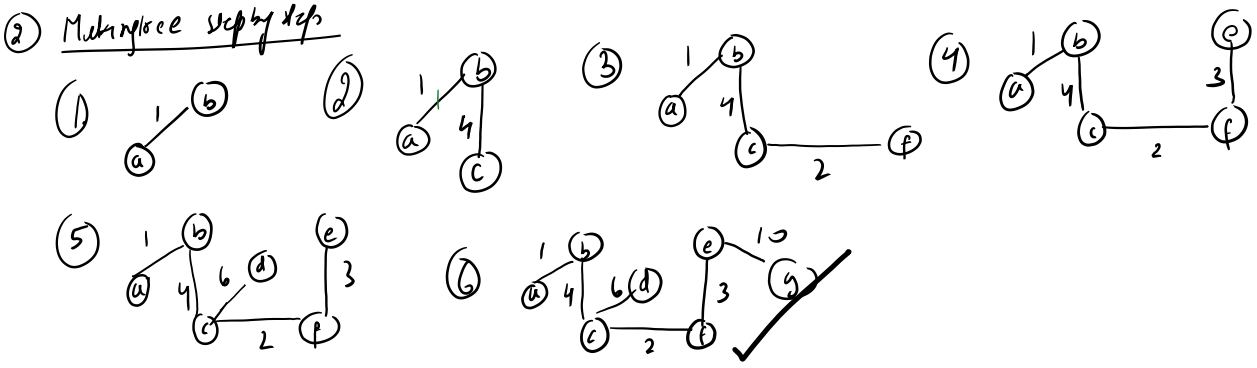
a. At any time of addition of the node a tree is obtained
b. There will be no cycles in this tree because it IS A TREE

Let's take some examples and then we will write the formal algorithm.

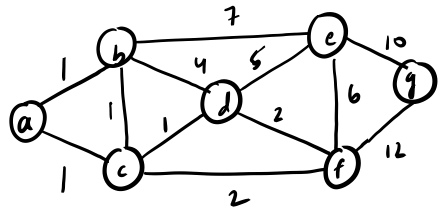
Example 1: Find the minimum cost spanning tree for the graph given below



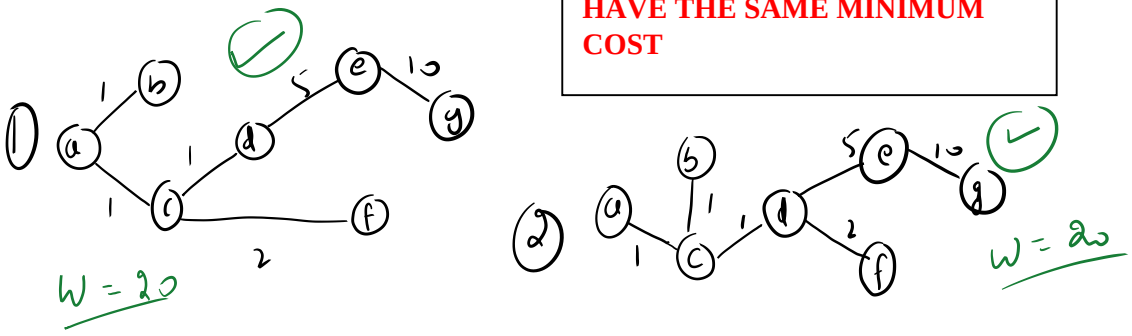




Example 2: An example where all the edge weights are not distinct.

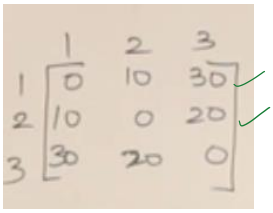


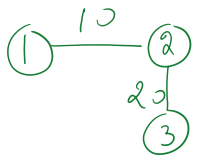
WHENEVER WE HAVE SAME WEIGHTS FOR SOME EDGES THEN THERE IS A CHANCE THAT WE MAY GET MORE THAN ONE MINIMUM SPANNING TREE BUT WILL HAVE THE SAME MINIMUM COST

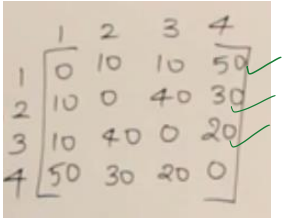


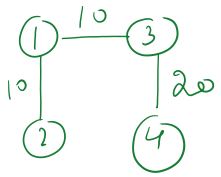
Stop making the graph once the count of the edges is reached (n-1) where n is the number of nodes.

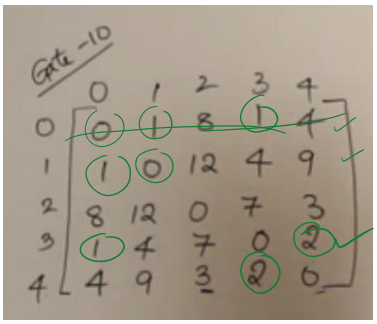
In GATE they are not going to give such simple graphs. They will give the graph in the form of the cost matrix. Let's kick its ass too.





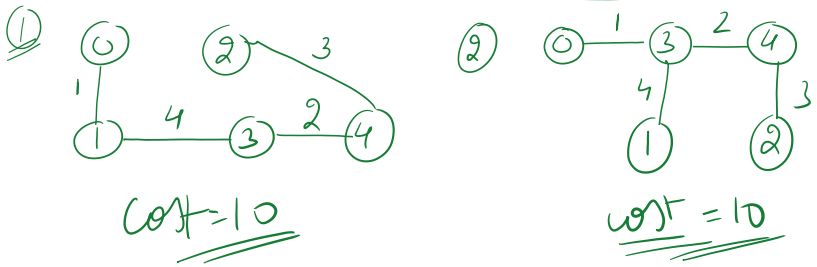






Find the minimum spanning tree such that the node 0 is a leaf

means that 0 can't have more than one connection



Prim's algorithm implementation without minimum heap (Greedy)

This is not necessary for GATE but learning will not hurt. Let's go to the algorithm

```
ALGORITM_PRIMS(E, cost, n, t)

//E is the set of edges. Cost is (nxn) adjacency matrix
//MST is computed and stored in array t[1:n-1, 1:2]

{
    1. Let (k, l) be an edge of min cost in E;
    2. mincost = cost[k, l]
    3. t[1, 1] = k, t[1, 2] = l
    4. for(i = 1to n)
        a. if(cost[i, l] < cost[i, k] then near[i] = l;
        b. else near[i] = k
    5. near[k] = near[l] = 0
    6. for (i=2 to n-1)
        a. let j be an index such that near[j] ≠ 0
           cost[j, near[j]] is minimum;
        b. t[i, 1] = j; t[i,2] = near[j];
        c. mincost = mincost + cost[j, near[j]];
        d. near[j] = 0
        e. for k = 1 to n do
            i. if((near[k] ≠ 0) and (cost [k, near[k]] > cost[k, j]))
                1. then near[k] = j

}
```

The prim's algorithm without min heap is going to take $O(n^2)$ time where n = no. of vertices.

So $O(v^2)$ time.

Let's see a second way to implement prim's algorithm which uses min-heap.

Prim's algorithm implementation using min-heap (Greedy)

```
MST_PRIMS(G, cost, r) //using min heap
{
        For each vertex u ϵ G.vertices
                u.key = ∞
                u.π = NIL

        r.key = 0
        Q = G.vertices
        while(Q ≠ ϕ)
                u = EXTRACT_MIN(Q)
                for each vertec 'v' adjacent to 'u'
                        if vϵQ and cost(u,v} <v.key
                                v.parent = u
                                v.key = w(u,v)
}
```

Total time taken for extract_min = O(vlogv)

Total time taken for build heap = O(v)

Total time taken for Decrease key = O(v.vlogv) = O(v^2logv)

This is actually not correct time taken by decrease key because the method will be called based on the number of edges that a particular node has. And in the end, in the worst case the number of calls to Decrease_Key will be nearly the number of edges.

According to aggregate analysis: Time taken by decrease key = O(ElogV)

Total time = O(VlogV + ElogV + V) = O(ElogV)

Which one is better? O(ElogV) or O(V^2)?

It turns out that for dense graphs, E = O(V^2) so Elogv = O(V^2logV)

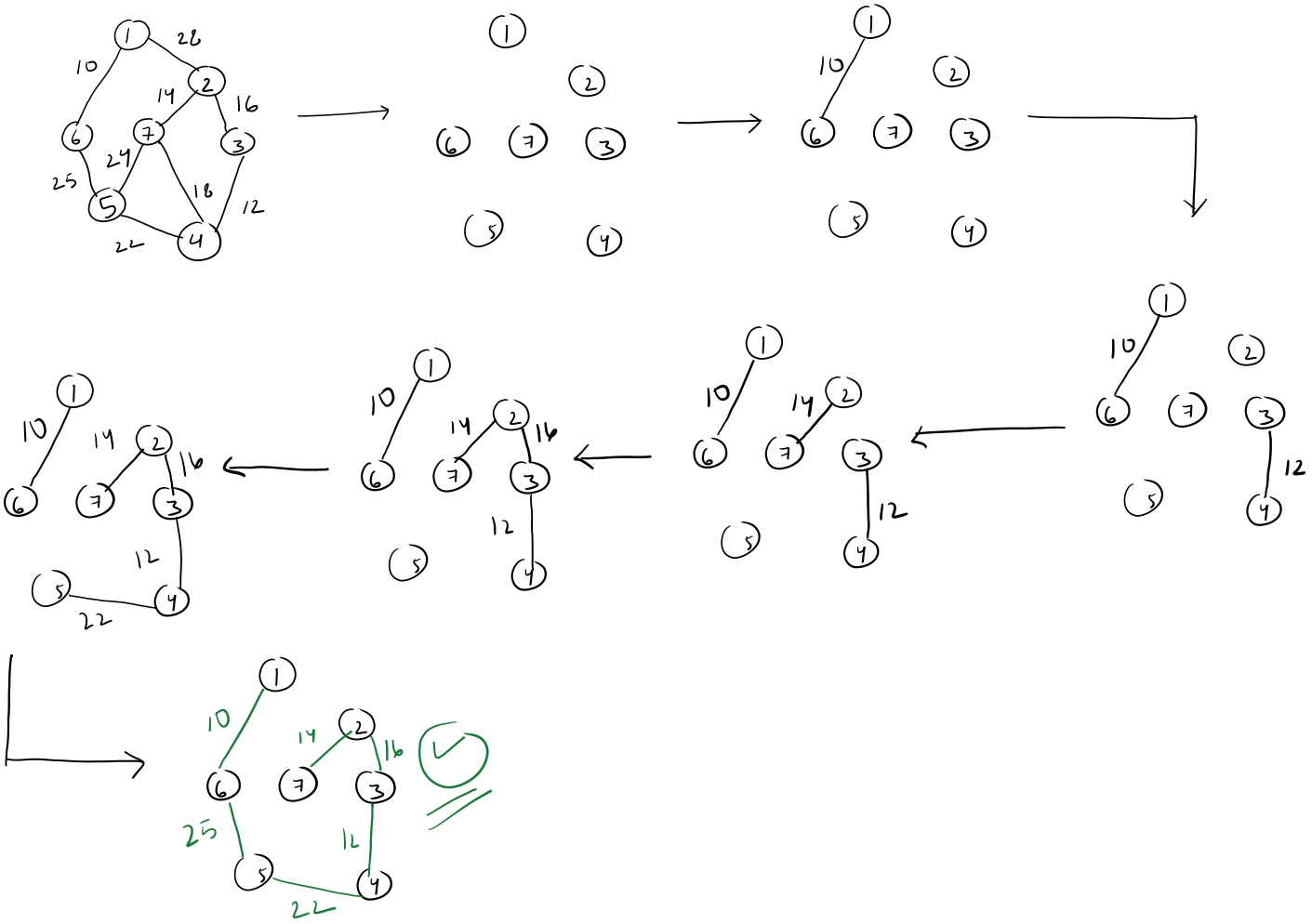
In sparse graph O(ElogV) = O(VlogV).

So for dense graphs, Prim's algorithm without heap is going to be more efficient and in case of sparse graphs Prim's algorithm with heap is going to be efficient.

Let's move on to the next greedy method for solving MST problem: Kruskal's algorithm.

KRUSKAL'S ALGORITHM FOR MINIMUM SPANNING TREE (Greedy)

Let's see an example before we jump to the algorithm.



Steps to make MST using Kruskal's algorithm

1. Get the edge which has minimum weight. Connect the two nodes that have this edge. This step is same as in the Prim's algorithm
2. Now, see which is the next edge which has the minimum weight. Connect the nodes having that edge.
3. Make sure no cycles are formed.
4. Repeat step 2 and 3 until all the vertices have been added.

How Prim's and Kruskal's algorithms are different?

1. In prim's you are building only one tree in each step. In Kruskal's there can be a forest.
2. In prim's you add the vertex/node as the algorithm proceeds. In Kruskal's all the nodes are already present, you are adding the edges in each step.

3. In prim's since it is only one tree and you are adding nodes, a cycle can never be formed. In Kruskal's because you are adding edges, you need to make sure that the cycles are not forming.
4. Applying Prim's and Kruskal's algorithm will give the same minimum spanning tree if all the weights are distinct in the tree. However, if the weights are repeating, you may get different spanning trees but the total weight of all the edges in the respective minimum spanning trees will be **EXACTLY SAME.**
5. **Note: You can find out spanning trees for the graphs which are connected.**

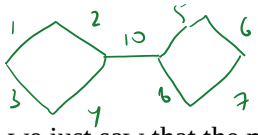
Let's see some more examples.

Gate 2000: Important question since we can explain many concepts in it.

Question: Let 'G' be an undirected connected graph with distinct edge weights. Let max_e be the edge with maximum weight and min_e be the edge with minimum weight. Which of the following is false?

a. Every minimum spanning tree of 'G' must contain min_e
b. If max_e is in a minimum spanning tree, then its removal must disconnect 'G'
c. No minimum spanning tree contains max_e
d. 'G' has a unique minimum spanning tree

Answer: **a. is true** because we always start with the edge with least weight
b. is also true because if we have worst case tree like below, we will have to add the edge with max weight so that the tree remains connected, otherwise it will be disconnected.



c.is false because we just saw that the max_e can be added to MST in worst case.

d.is true because all the weights are distinct

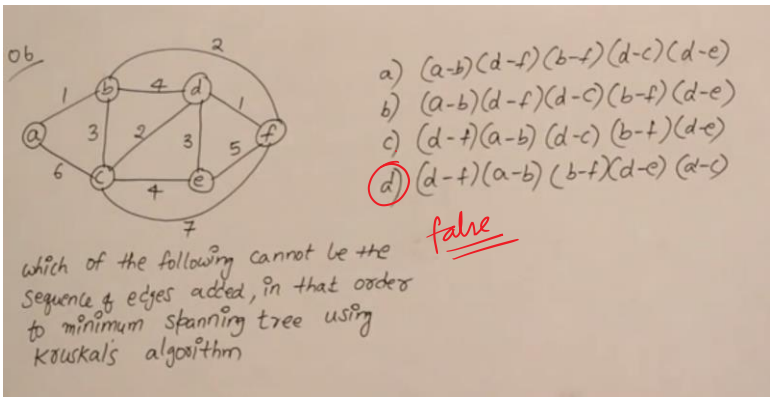
Answer : C.

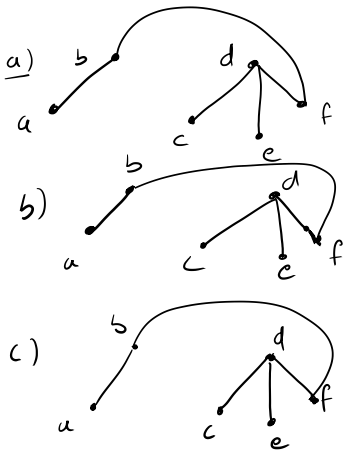
If the question is modified where we are not given if the tree has distinct weight for all edges then (A) and (B) are true and (C) and (D) are false.

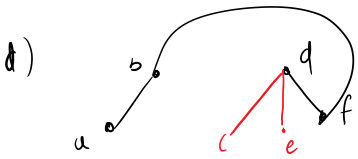
Gate 2007: Let 'w' be the minimum weight among all weights in an undirected connected graph. Let 'e' be a specific edge of weight 'w'. Which of the following is FALSE?

a. There is a minimum spanning tree containing 'e' ← true
b. If 'e' is not in a minimum spanning tree 'T', then in the cycle formed by adding 'e' to 'T', all edges have the same weight ← true
c. Every minimum spanning tree has an edge weight of 'w' ← true
d. 'e' is present in every minimum spanning tree ← false

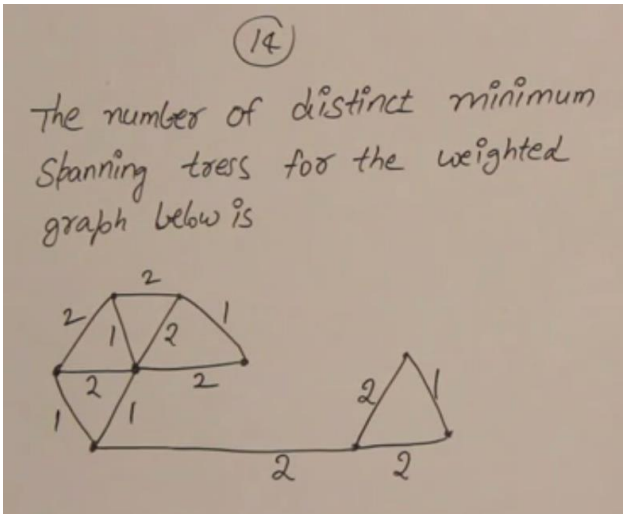
Gate 2006:





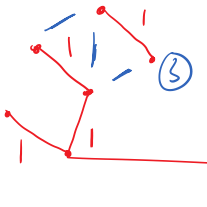


Gate 2014:



① all 1's will be added (there are no cycles)

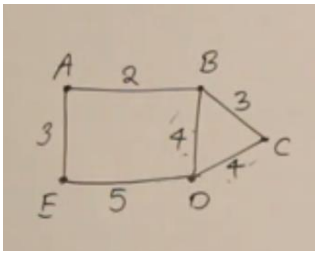
• → choices



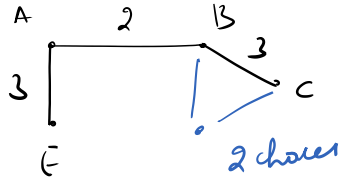
total choices = 3×2 = 6

6 graphs MST

has to be added so the graph is not disconnected



Kruskal's

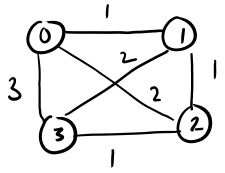


2 choices

∴ total spanning trees possible = 2

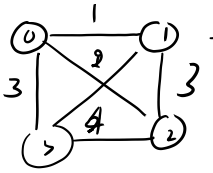
4) A complete, undirected, weighted graph 'G' is given on the vertex {0,1,...n-1} for any fixed 'n'. Draw the minimum spanning tree of G if
a) the weight of the edge (u,v) is |u-v|
b) The weight of the edge (u,v) is (u+v)

a) n=4 → 0 —1— 1 —1— 2 - - - n-1

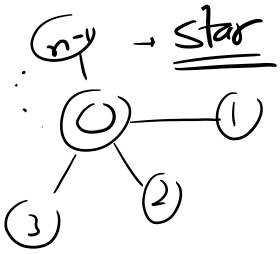


linear

b)



→



star

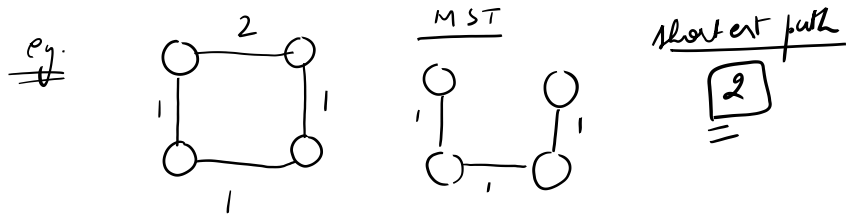
FINDING THE SHORTEST PATH IN A DIRECTED GRAPH (Greedy)

DIJKSTRA'S ALGORITHM or SINGLE SOURCE SHORTEST PATH ALGORITHM

Finding a shortest path is different than finding the minimum spanning tree.

In minimum cost spanning tree:

1. There is no concept of source and destination but in shortest path problem we have a source from where we will start and reach the destination using the shortest path
2. Minimum cost spanning tree is not going to solve the shortest path problem. They may give you shortest path but that will be purely coincidental.



Problem: There will be a graph which will be weighted and directed and a source and a destination are given. We need to reach the destination from the source using the shortest path.

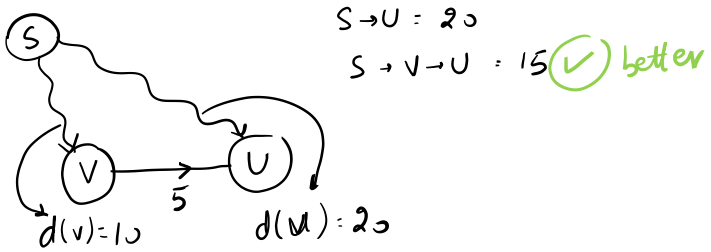
This is also called *Single source shortest path* problem. Here we find out the shortest path from one source to all the other nodes. ☺

Let's build the algorithm in parts.

One of the operations that we use is called **RELAXING AN EDGE.**

Relaxing an edge

We are in the algorithm until some phase and during this we found out that from S to reach V the total distance taken is d(V) = 10 and from S to reach U the total distance taken is d(U) = 20. Let's say there is an edge between V and U with a weight of 5. Then relaxing this edge will mean: in case if I use this V -> U edge in the calculation of minimum path from S to U or V to U then is there going to be any better result?



We will use the relax operation:

$$if\ d(u) > d(v) + c(u,v)\ then,$$

$$d(a) = d(v) + c(v,u)$$

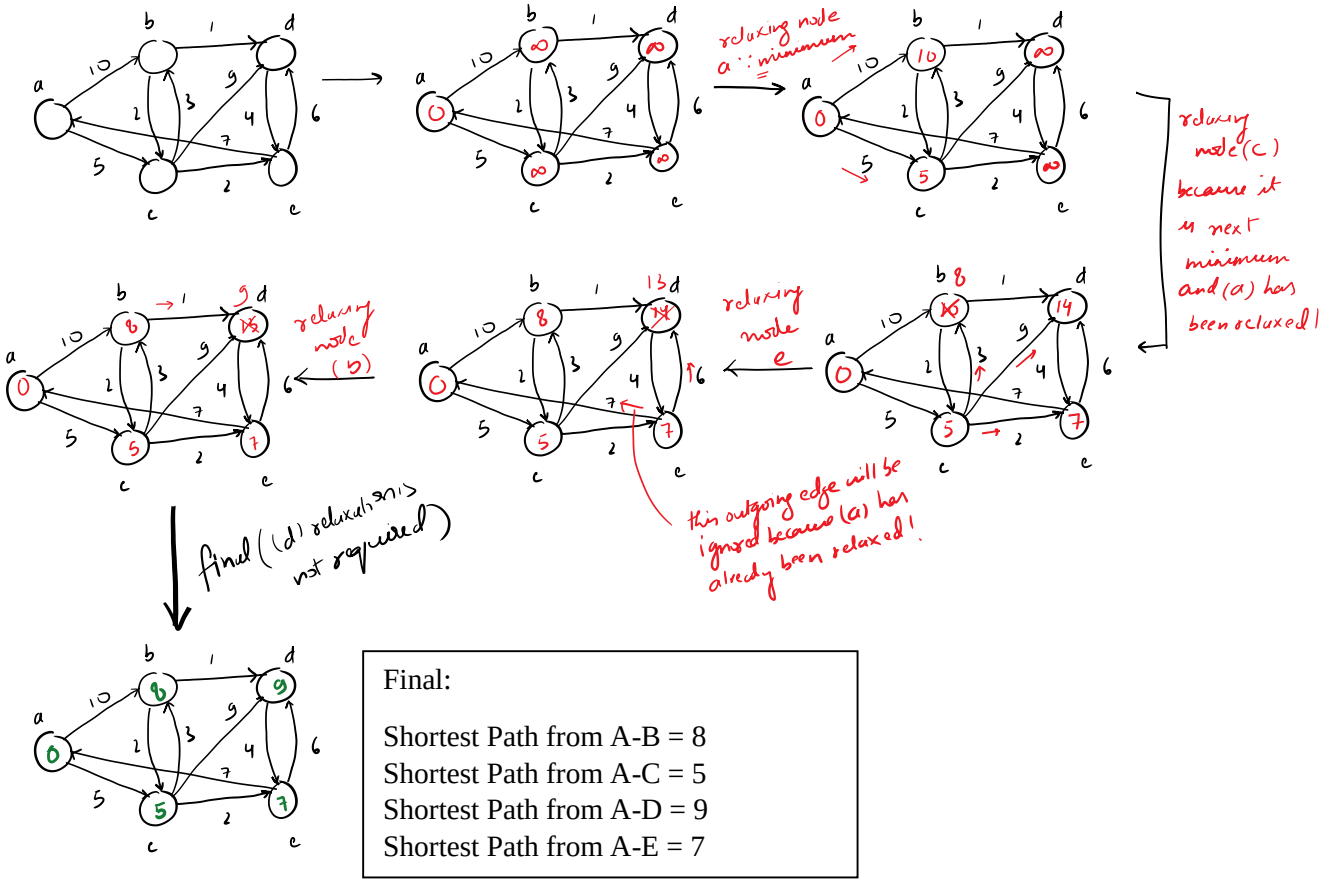
Dijkstra's algorithm should not be used when the edge weights are negative

Steps for Dijkstra's algorithm

1. Make sure there are no edges with negative weights
2. Take source node, the distance of the source node to itself is going to be 0
3. Take all the nodes and make the distance from source to all of them as ∞
4. From the source node, see what outgoing edges are there and relax them.
5. Take the next node which has the minimum of all the current path and has not been calculate the relaxation before.
6. Repeat step 4 until all the outgoing edges of all the nodes have been relaxed.
7. Constantly ask question: "If I relax this node will the distance get better (better means lesser)."

Let's understand with and example.

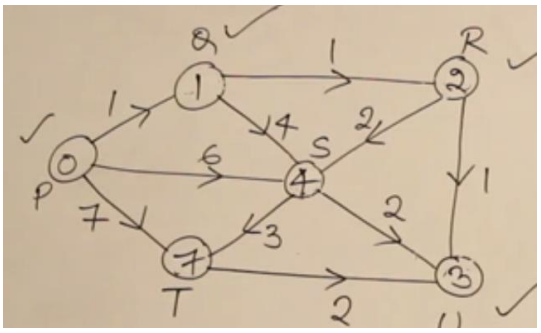
Example: Calculate single source shortest path in the following graph



Final:

Shortest Path from A-B = 8
Shortest Path from A-C = 5
Shortest Path from A-D = 9
Shortest Path from A-E = 7

Example 2: What is the order of relaxation?



Order: P,Q,R,U,S,T

DIJKSTRA'S ALGORITHM

Let's write the Dijkstra's Algorithm, but before let's see some points

1. Since we are extracting the minimum for relaxation, heap sounds like a good data structure to implement Dijkstra's algorithm.
2. Relaxing operation is nothing but the *decrease_key* operation in the heap.

DIJKSTRA(G, W, S)

1. Initialize_single_source(G,S) ← O(V)
2. S = ϕ
3. Q = G.V ← Build heap = O(V)
4. While Q ≠ ϕ runs V times
   a. U = Extract_Min(Q) ← O(logV)
   b. S = $S \cup \{Q\}$
   c. For each vertex $v \epsilon G \cdot Adj(U)$
      i. Relax(u, r,w) ← O(logV)

S = set of all vertices whose shortest path has been decided

Relax() is run on each edge at most one and every call to Relax() runs Decrease_Key() in heap.

→ Extract_min = O(VlogV) will run

→ Relax( ) will run = O(ElogV)

why?

∴ Time complexity is defined below

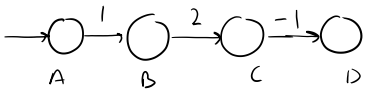
Time complexity for Dijkstra's algorithm = O(V + V + VlogV + ElogV) (using aggregate analysis)

⇨ **O(ElogV) (WORST CASE SCENARIO)**

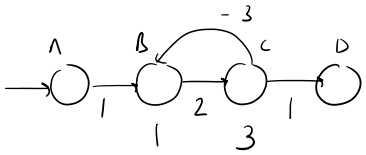
NEGATIVE WEIGHT EDGES AND DIJKSTRA'S ALGORITHM

Shortest paths will not be found when:

1. The graph is disconnected
2. If the graph has negative weight cycles



A-D : 2 ✓ Dijkstra's applicable



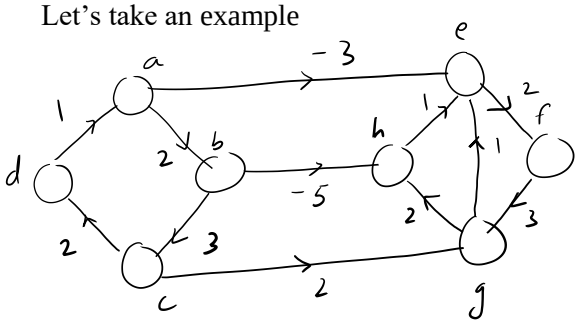
Dijkstra's not applicable

1 3
0 2
-1 1
-2 0
-3 -1

Important note: *If we give Dijkstra's algorithm* ***graphs with negative weight cycles*** *then* ***it is will only work in some cases and find the shortest path.*** *However, if there are graphs with only* ***negative edges and no negative weight cycles****, the algorithm* ***will be able to find shortest path****.*

This is because Dijkstra's algorithm doesn't have the power to differentiate if the graph has negative cycles or not. If an algorithm has this power, then we can find the shortest paths in graphs with negative cycles too. ***Therefore, if there is a cycle with negative weights, then Dijkstra's will not be able to decrease the value of the key as the key has already been deleted from the heap. So Dijkstra's algorithm will fail whenever we delete a note and then for the same node later on we have to decrease the value.***

Let's take an example



Gate 2008

Let's understand with the table

Nodes	A	B	C	D	E	F	G	H
Initialization	**0**	∞	∞	∞	∞	∞	∞	∞
Delete A		2	∞	∞	**-3**	∞	∞	∞
Delete E		2	∞	∞		**-1**	∞	∞
Delete F		**2**	∞	∞			2	∞
Delete B			5	∞			2	**-3**
Delete H			5	∞			**2**	
Delete G			**5**	∞				
Delete C				**7**				
Delete D								

Order: AEFBHGCD
Shortest distances:

A-A = 0	A-E = -3
A-B = 2	A-F = -1
A-C = 5	A-G = 2
A-D = 7	A-H = -3

Now let’s talk about the algorithm that has the capability to find out whether a given graph has any negative weight cycles in it or not.

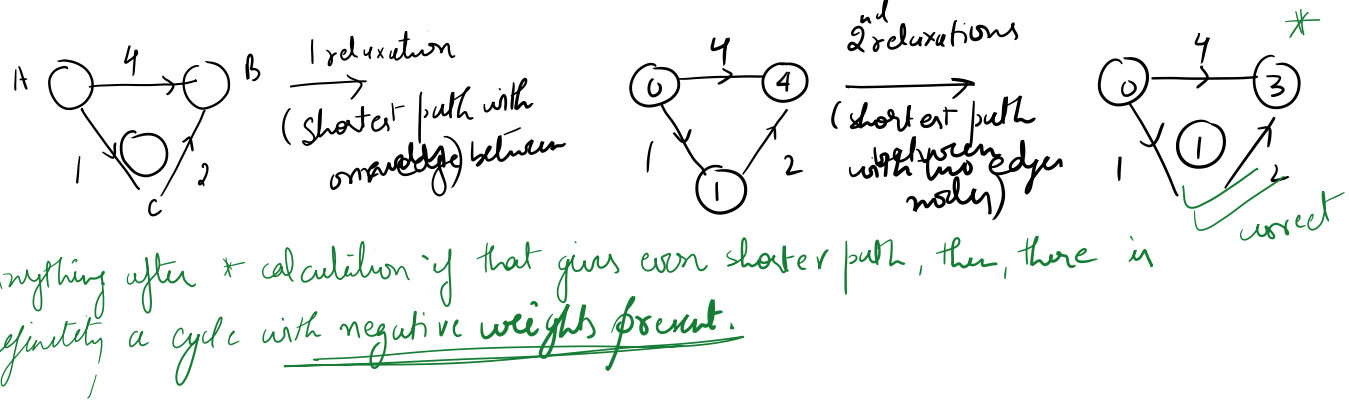
BELLMAN FORD ALGORITHM

This algorithm can say that "No I can't find the shortest path because the negative weight cycle is present" and also it can find out the shortest path when the negative weight cycle is not present. Such a power doesn't come cheap.

Compared to Dijkstra's algorithm, Bellman Ford algorithm is a bit slower so its complexity is more compared to Dijkstra's.

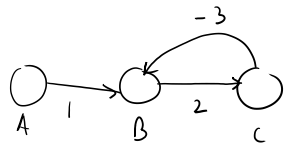
***Bellman Ford works on this**: If you have a graph with 'n' nodes then the shortest path in the graph will never contain more than "n-1" edges. If it is more than n-1 and the shortest path already calculated for the node is decreasing, then there is definitely a cycle with negative weights present.*

This is the basic algorithm which tells about the negative cycles in the graph.

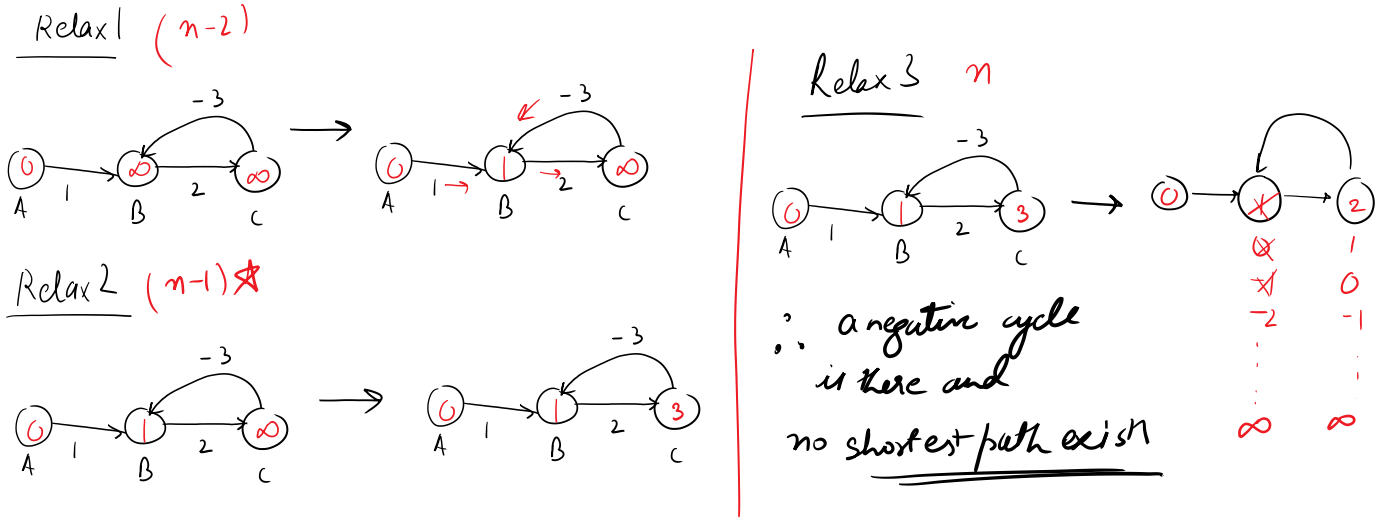


Let's see with an example how Bellman Ford algorithm is going to tell if the negative weight cycle is present or not.

Example:



Number of nodes = 3, so Number of times each edge should be relaxed = 3 – 1 = 2 times.
Bellman Ford algorithm will run : (V-1) + 1 (check the negative cycle) = V times.



Total number of relax operations by Bellman Ford is:

((V+1) -1) x E = VE x (Relax()) times. **Therefore, the time complexity = O(VE)**

Here Relax() will be in an array instead of a heap because for Bellman Ford we don't need a heap as there is no **decrease_key** or **Extract_min** operation. This relax can be done is O(1) or constant time. Therefore, total complexity is O(VE) which is higher than the Dijkstra's algorithm.

Let's take a look at the formal algorithm.

```
BELLMAN_FORD(G, w, S)
{
    1. Initialize_single_source(G, S)
    2. For i=1 to |G.V| -1
        a. For each edge (u, v) ϵ G.E
            i. RELAX(u, v, w)
    3. For each edge (u, v) ϵ G.E
        a. If(v.d > u.d + w(u, v))
            i. Return FALSE
    4. Return TRUE
}
```

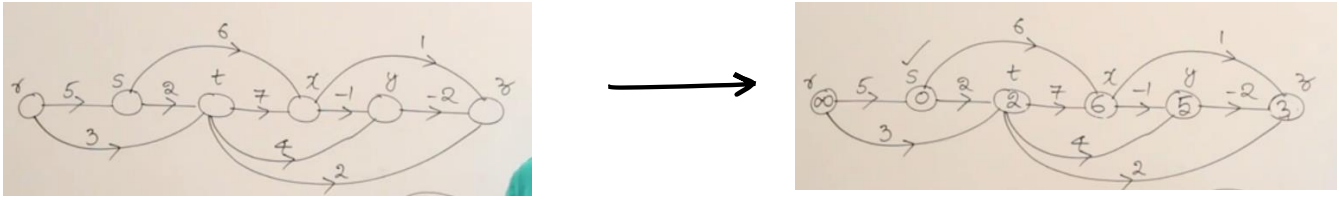
Complexity:
$O(V + VE + E) = O(VE)$

DIRECTED ACYCLIC GRAPH

As the name suggests, it is a directed graph where there are no cycles.

To find the shortest path in Directed acyclic graphs or DAGs

1. Make sure that the graph is topologically sorted (covered in data structure) which take O(V+E) time. After topologically sorting the graph, all the edges will move from left to right.
2. Take each vertex one by one and relax all the edges outgoing from that edge



Analysis:

1. For topological sorting time: O(V+E)
2. For relaxing all the edges outgoing from all the vertices (basically means relaxing all the edges once) = O(E)
3. **Time complexity = O(V+E)**

SHORTEST PATH IN DIRECTED ACYCLIC GRAPHS

DAG_SHORTEST_PATHS(G, W, S)

1. Topologically sort the vertices of 'G' ← O(V+E)
2. Initialize_Single_Source(G, S) ← O(V)
3. For each vertex u, taken in topologically sorted order
    a. For each vertex v ϵ G.adj[u]
        i. Relax(u, v, W)

} O(E)

because array implementation of data structure

Note: ***Don't assume that the fastest shortest path algorithm is DAG one because that will work only in specific cases. So we have seen:***

Dijkstra's Algorithm: O(ElogV)

Bellman Ford Algorithm: O(VE)

DAG Shortest Path: O(V+E)

Use DAG Shortest path algorithm only when you are sure that you don't have any cycles in your graph. Therefore, this will work also when we have negative weights.

This concludes our discussion on optimization with GEEDY ALGORITHMS. Next up is Dynamic Pogramming.