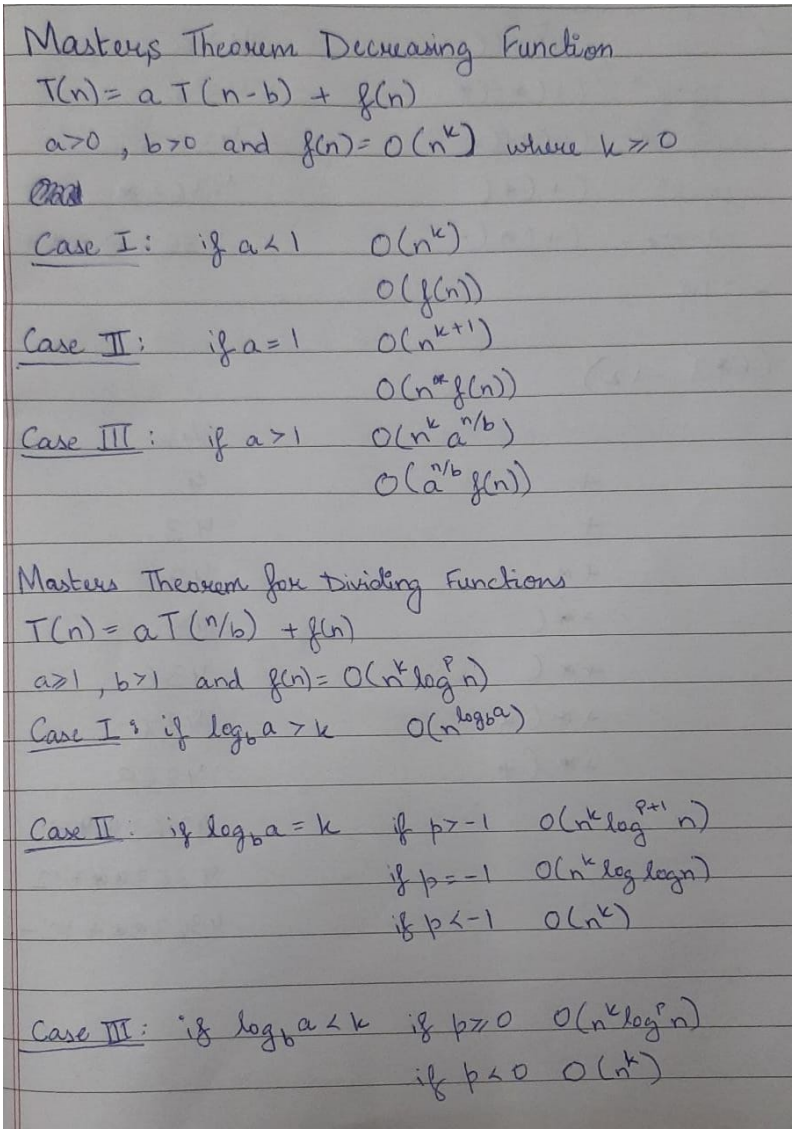
Algorithms

Master Theorem for time complexity



Root Function: O(log(logn))

Binary Search:

1. list should be already sorted
2. find m=(l+h)/2, if number is higher, l=mid+1 and if number is lower, h=mid-1
3. if l>h, then element is not present in the list
4. follows divide and conquer
5. As it is dividing the list everytime, worst case and average case is O(logn)
6. Best case is O(1)

Iterative solution:

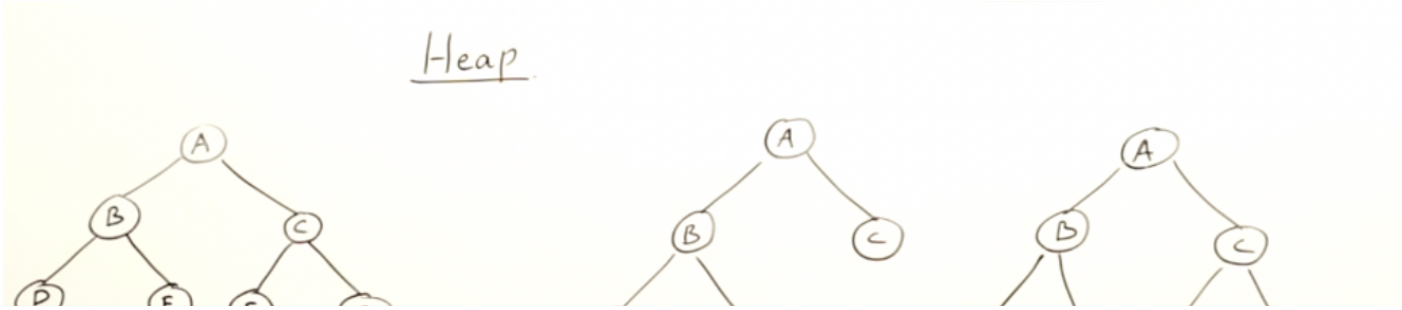
```
int BinSearch(A,n,key){
    int l=1,h=n;
```

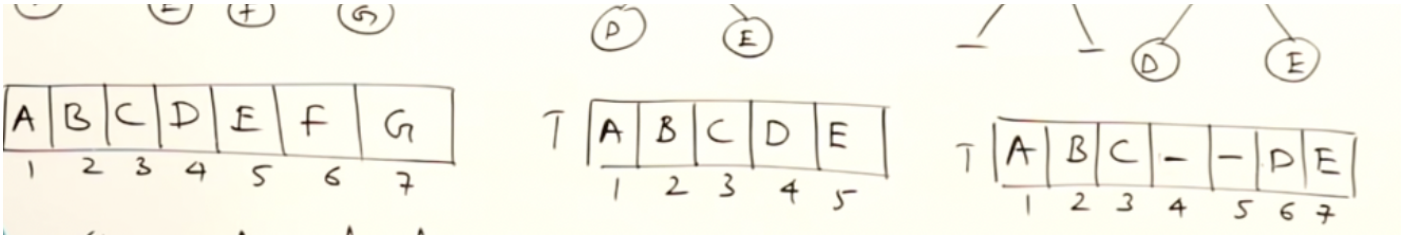
```
    while(l<=h){
        mid=(l+h)/2;
        if(key==A[mid])
            return mid;
        if(key<A[mid])
            h=mid-1;
        else
            l=mid+1;
    }
    return 0;
}
```

Recursive solution:

```
int RBinSearch(l,h,key)
{
    if(l==h){
        if(A[l]==key)
            return l;
        else
            return 0;
    }
    else{
        mid=(l+h)/2;
        if(key==A[mid])
            return mid;
        if(key<A[mid])
            return RBinSearch(l,mid-1,key);
        else
            return RBinSearch(mid+1,h,key);
    }
}
```

Array Representation of Binary true:
if a node is at index i
left child is at 2*i
right child is at 2*i+1
its parent is at i/2



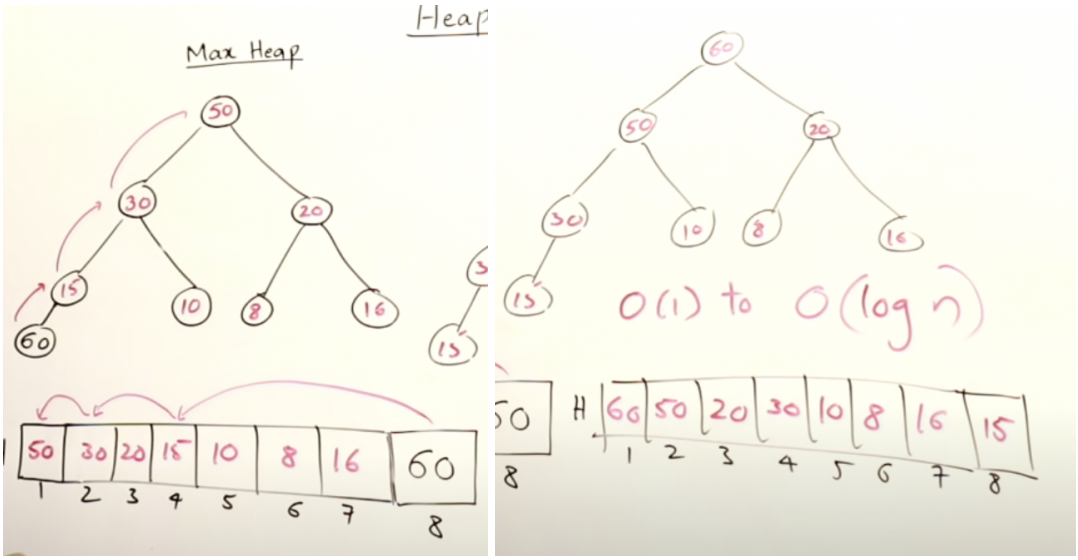


Full Binary Tree: To add a node you need to increase height
No. of nodes: 2^(h+1) - 1

Complete Binary Tree: In array representation, there should be no gap. It is a full binary tree about height h-1 and the last level is filled from left to right

Heap: It is a complete binary tree in a particular order
MaxHeap: Every node is greater than or equal to all its descendants.
MinHeap: Every node is smaller than or equal to all its descendants.

Insertion in Max Heap:
First insert the element in the last position of array representation. Then swap with the parent nodes to reach the correct position.



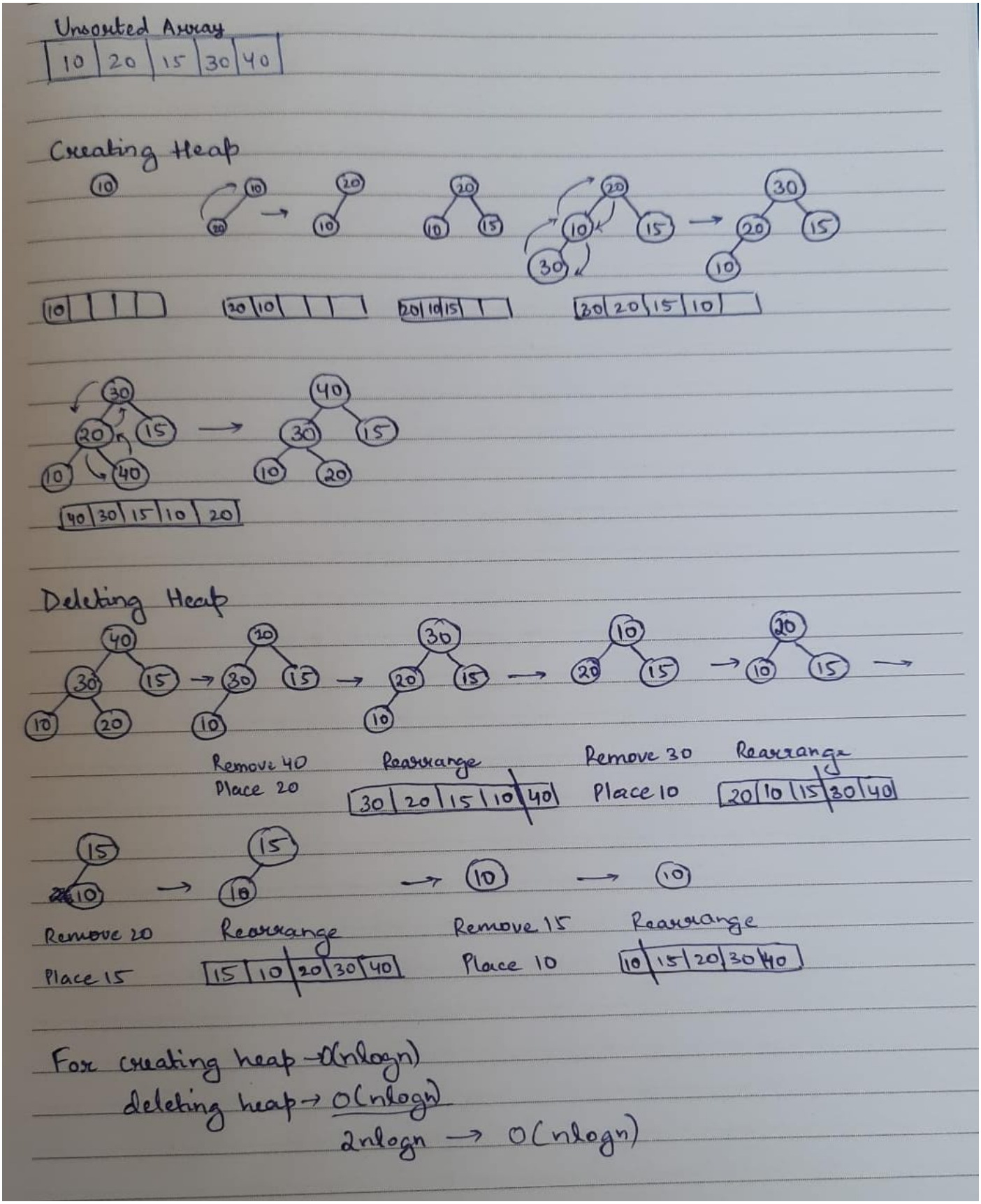
Best case is O(1) where no swap is required and average and worst is O(logn) as for the above example also it took 3 swaps which was logn

Deletion in Max Heap: Only root element is deleted. So the last element in the array representation is taken to root element i.e. first element and the original root is removed. Then we compare descendants and whichever is greater is swapped to maintain max-heap

Sort: When you're deleting instead of removing the root element, store it at the end of the array, this way each time you delete the largest will go to the end of the array left. This way it is sorted

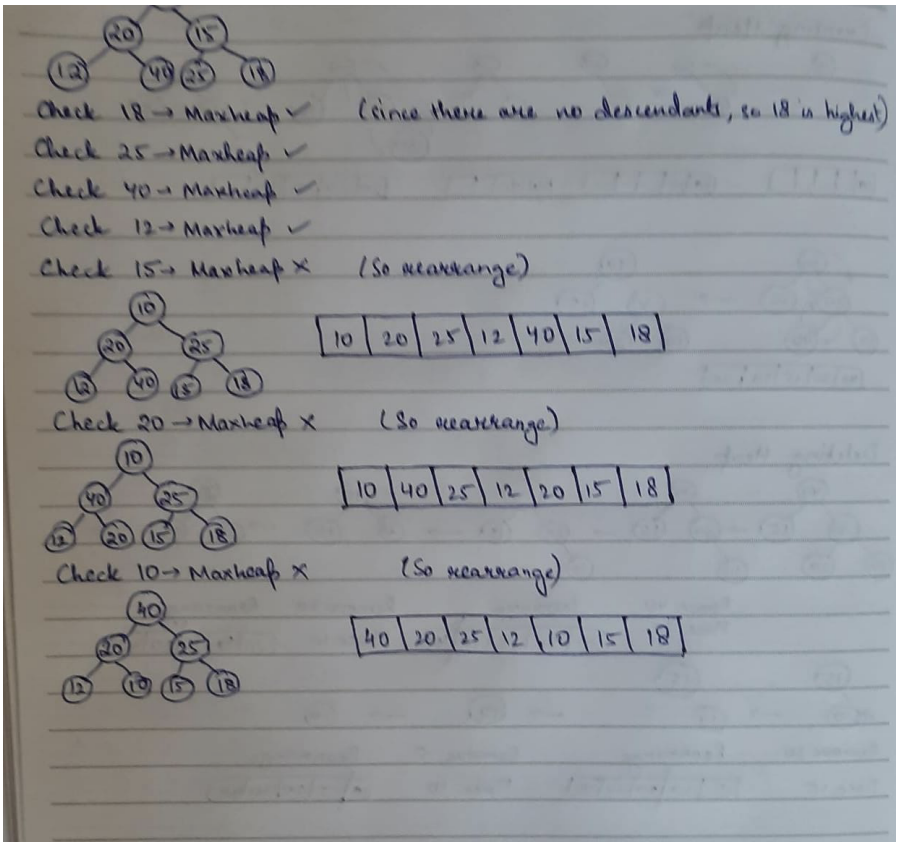
Heap Sort: If you need to sort an array, first create a heap and then delete all elements from the heap and you'll get a sorted array

you'll get a sorted array.

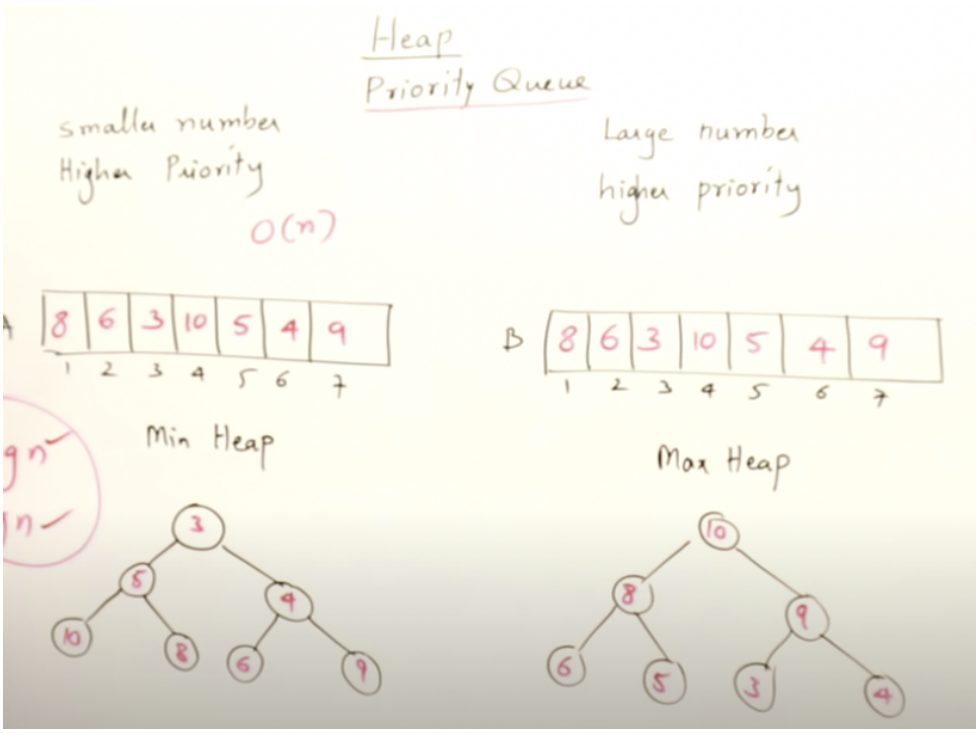


Heapify: Used for creating a heap. Keep checking if the elements are in a heap from the end of the array representation.

Time Complexity: O(n)



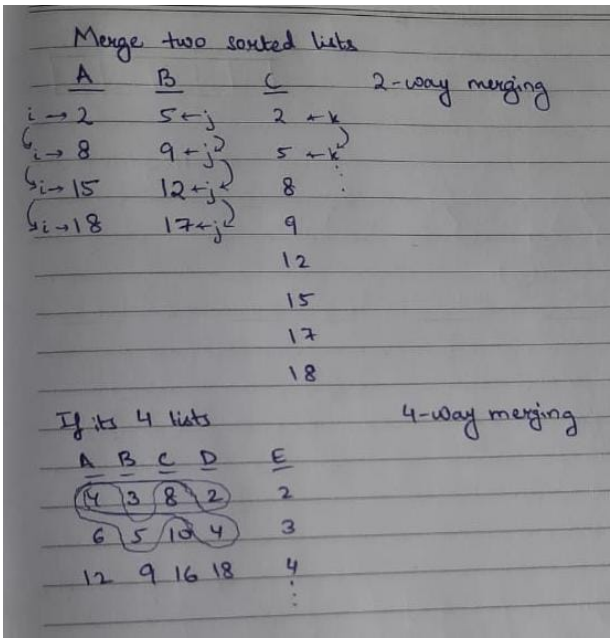
Priority Queue: Best data structure to use for this is heap because faster. Because if you do with array insertion or deletion may take O(n) but in heap it takes O(logn)

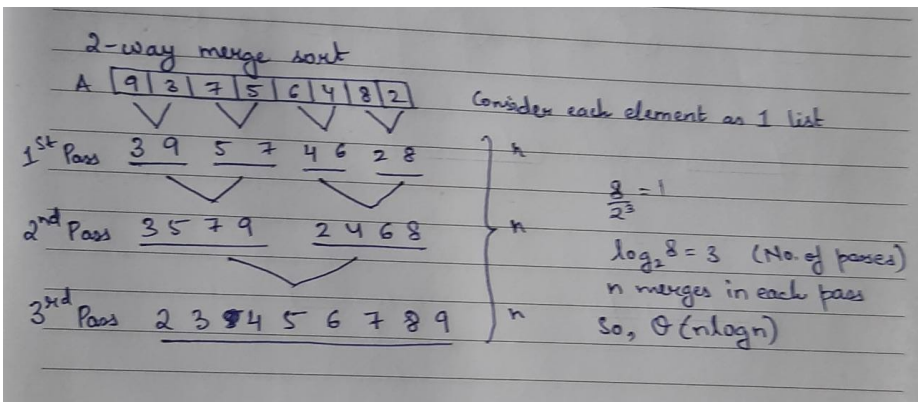


Merge:

```
Algorithm Merge(A,B,m,n){
    i=1,j=1,k=1;
    while(i<=m && j<=n)
    {
```

```
    {
        if(A[i]<B[j])
            C[k++]=A[i++];
        else
            C[k++]=B[j++];
    }
    for(;i<=m;i++)
        C[k++]=A[i++];
    for(;j<=n;j++)
        C[k++]=B[j++];
}
```

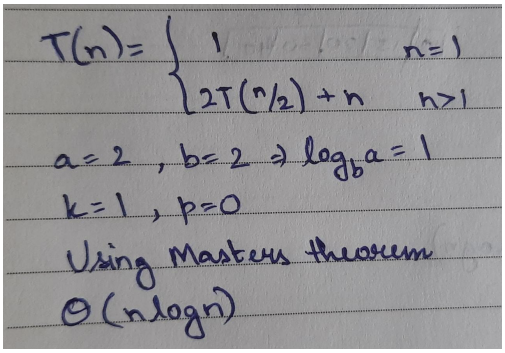




Merge Sort: O(nlogn)

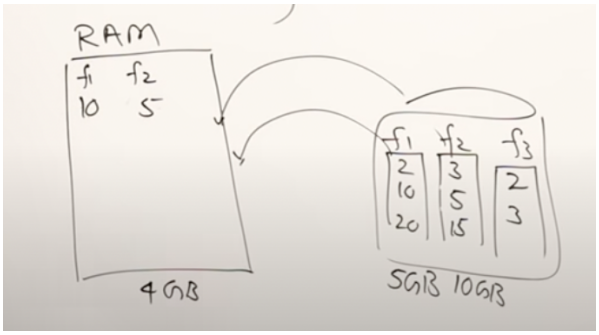
```
Algorithm MergeSort(l,h){        // Let T(n)
    if(l<h){
        mid=(l+h)/2;              //1
        MergeSort(l,mid);         // T(n/2)
        MergeSort(mid+1,h);       // T(n/2)
        Merge(l,mid,h)            // n
    }
}
```

Total=2T(n/2)+n

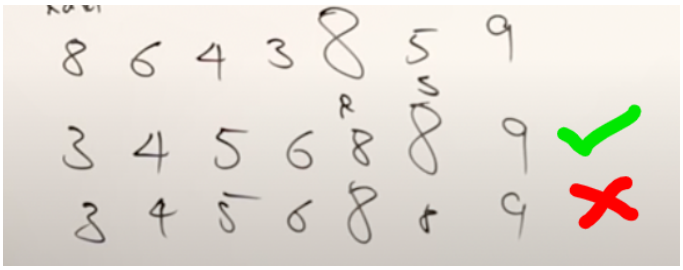


Pros of Merge Sort:

1. Works best in large size list
2. Easy sorting in linked list without need of creating another linked list
3. External Sorting



4. Stable: Duplicates should always preserve their order



Cons of Merge Sort:

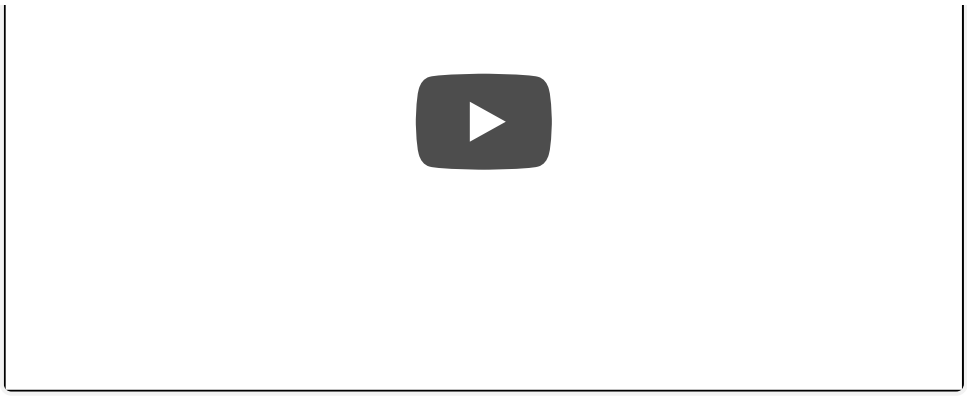
1. Takes extra space for sorting (not in place sorting) (As for merging we put it in a separate array and then we copy it back at the end
2. No small problem (Merge sort is slower in smaller lists like less than 15 elements because it takes up lot of time in recursion. Insertion sort is used at that place)
3. Recursive (Whenever recursive algorithm is used, it creates a stack to keep track of height log(n). So more extra space needed. n for extra array and logn for stack)

So total space used: n+logn = O(n)

Quick Sort:

YouTube Video

2.8.1 QuickSort Algorithm

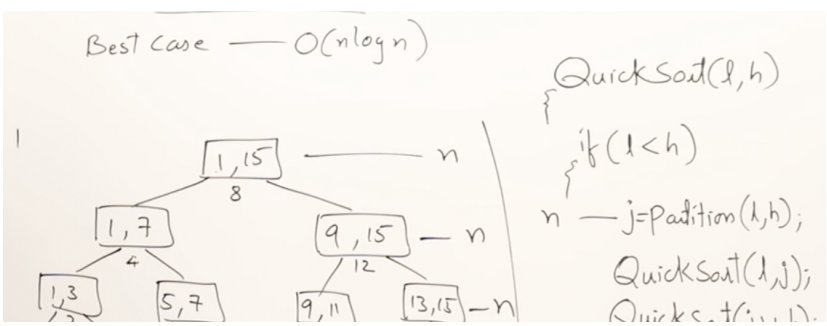


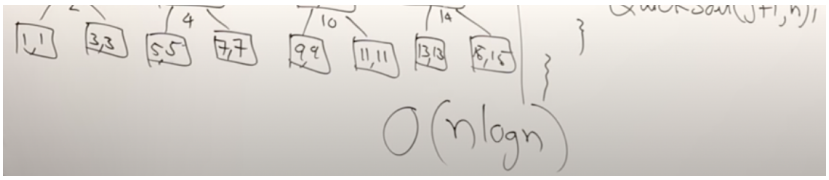
```
Partition(l,h){
	pivot=A[l];
	i=l, j=h;
	while(i<j){
		do{
			i++;
		}while(A[i]<=pivot)
		do{
			j++;
		}while(A[j]>=pivot)
		if(i<j)
			swap(A[i],A[j]);
	}
	swap(A[l],A[j]);
	return j;
}

QuickSort(l,h){
	if(l<h){
		j=Partition(l,h);
		QuickSort(l,j);
		QuickSort(j+1,h);
	}
}
```

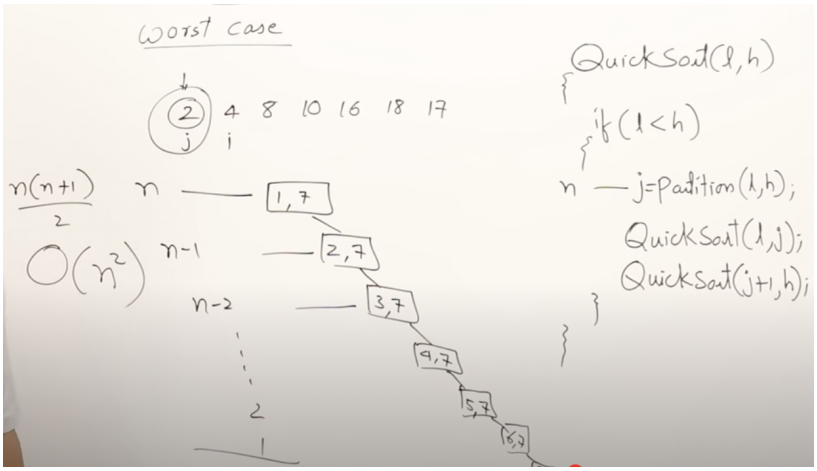
Time Complexity:

When partition is always taken in middle, every time size is reduced to two halves. In that case time complexity is O(nlogn) which is the best case.





If it is an already sorted list, every time only 1 element will be sorted and rest will remain the same. SO time complexity is O(n^2)



So we should always either select middle element as pivot or any random element as pivot

---

For normal matrix multiplication:

```
for(i=0;i<n;i++){
    for(j=0;j<n;j++){
        C[i,j]=0;
        for(k=0;k<n;k++)
            C[i,j]+=A[i,k]*B[k,j];
    }
}
```

Time complexity=O(n^3)

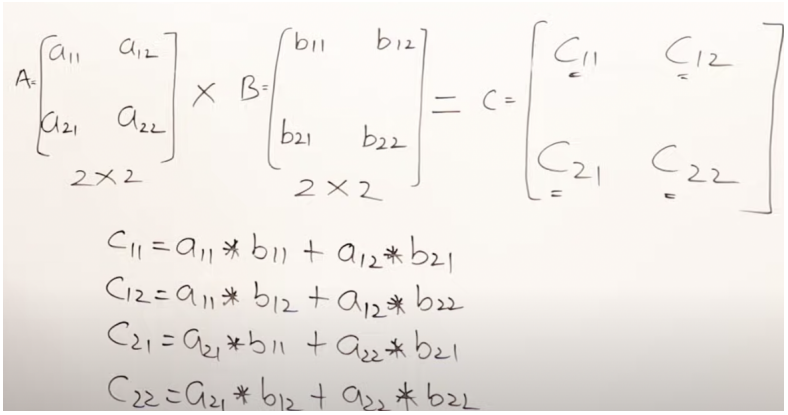
Small problem for recursive algo:

1X1:

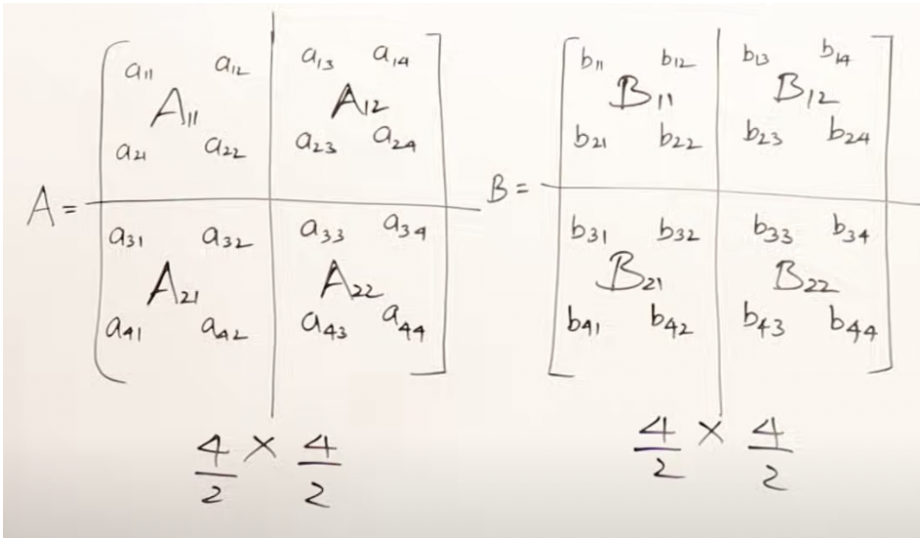
A=[a_{11}] and B=[b_{11}]

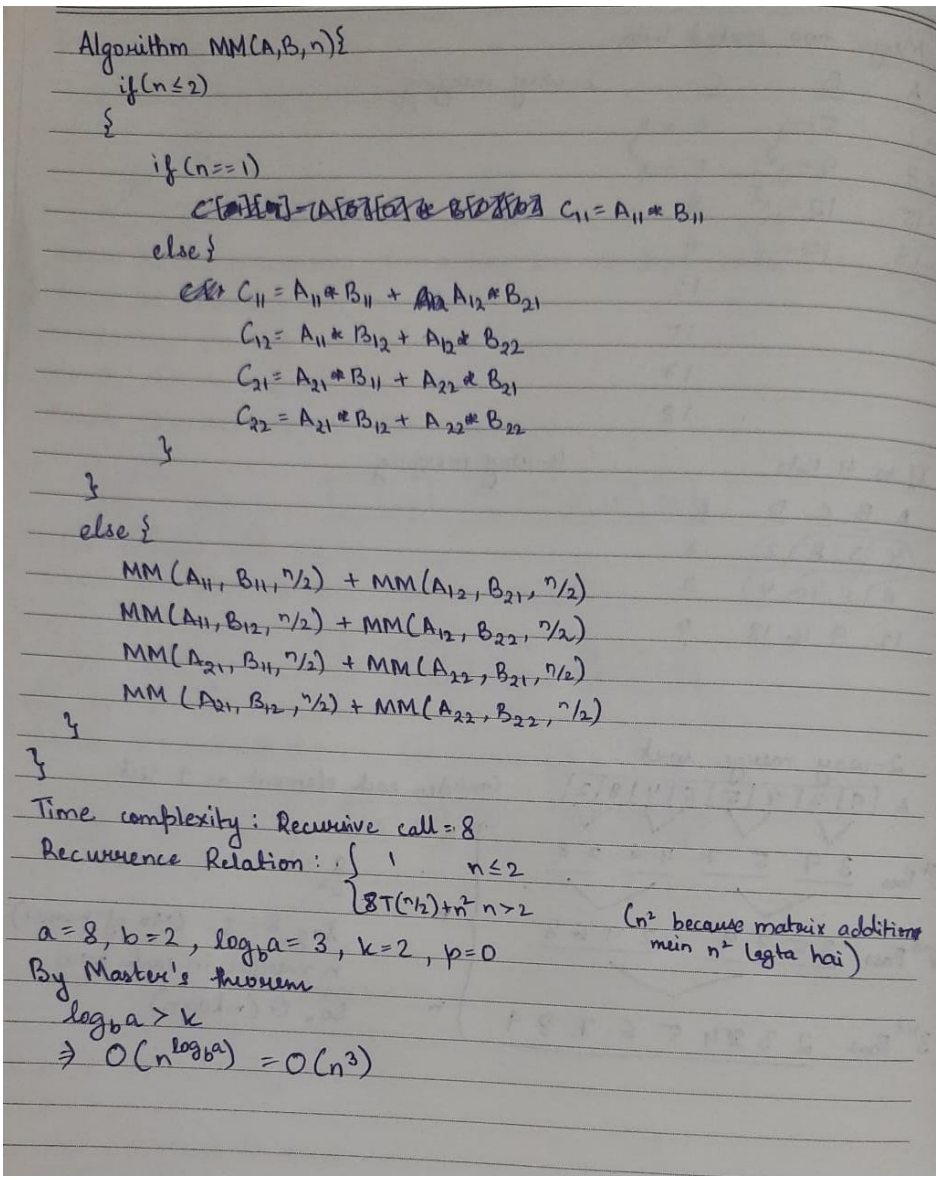
C=[a_{11}*b_{11}]

2X2:

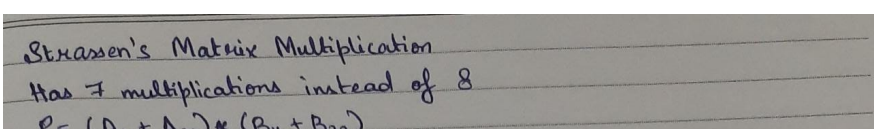


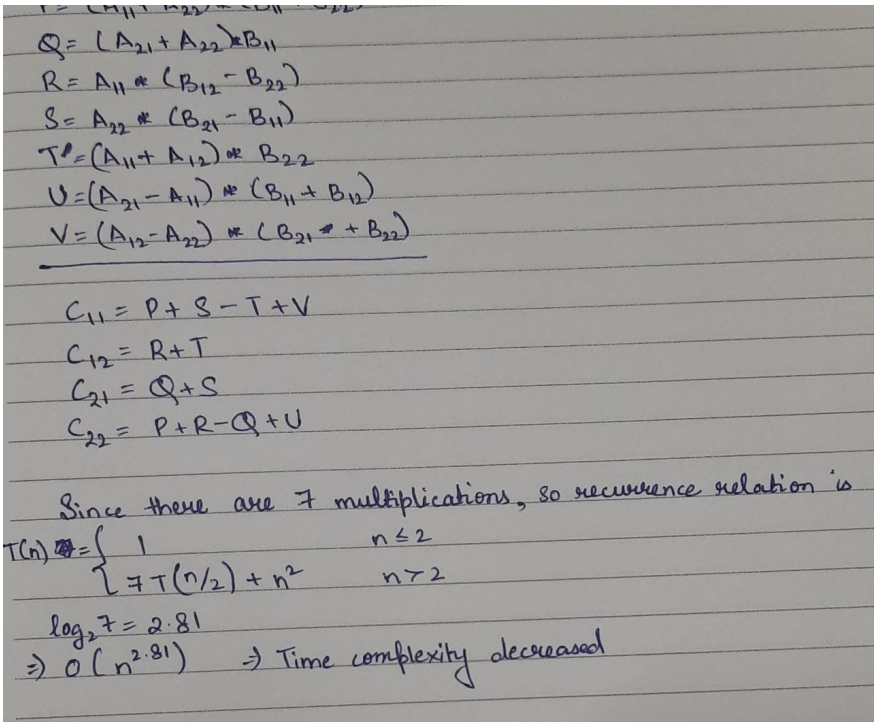
If it is greater than 2X2 then it is a big problem and we need to divide it





Strassens Matrix Multiplication:





To solve optimisation problems you can use 3 methods:

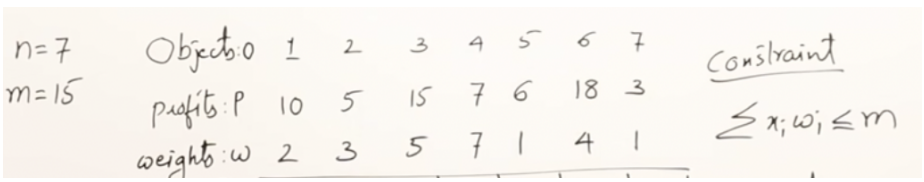
1. Greedy Method
2. Dynamic Programming
3. Branch and Bound

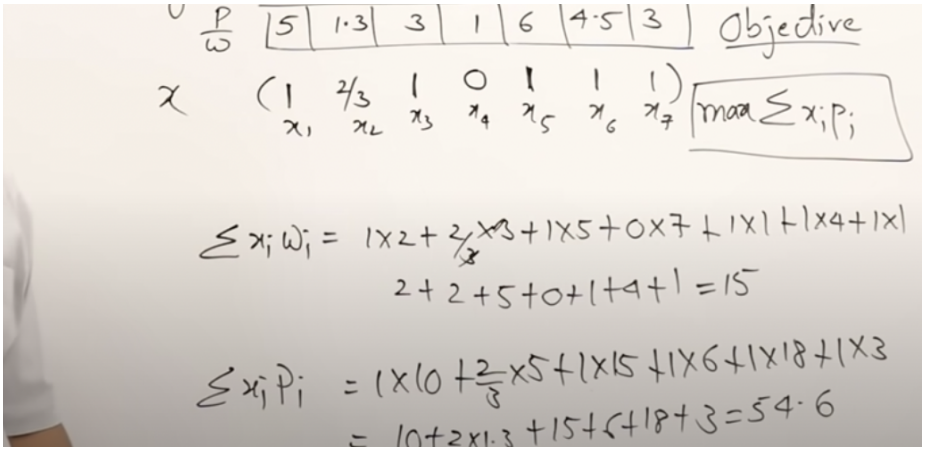
Greedy: The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem.

Basic Algorithm:
Let a be an array of items

```
Algorithm Greedy(a,n){
    for i=1 to n do{
        x=Select(a)l
        if Feasible(x) then
            Solution=Solution+x;
    }
}
```

1. Knapsack problem:
Constraint: weight
Objective: maximize profit
Find p/w and then keep adding from highest to lowest till weight exceeds.





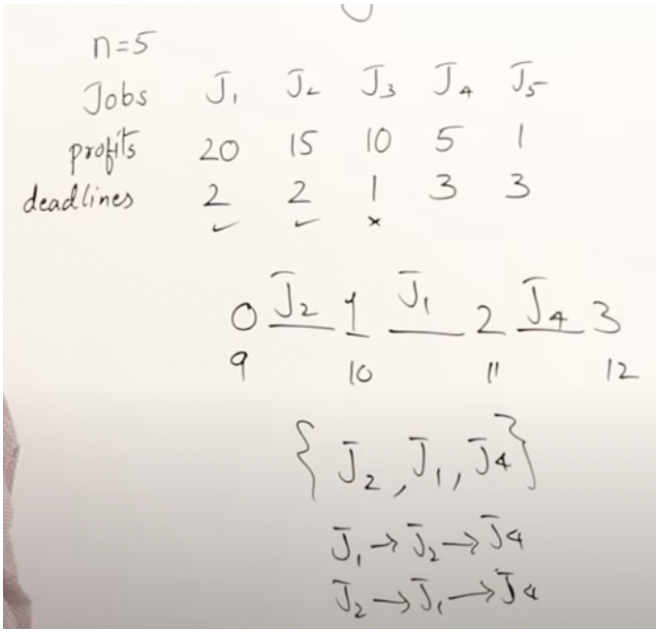
2.Job Scheduling with deadlines:

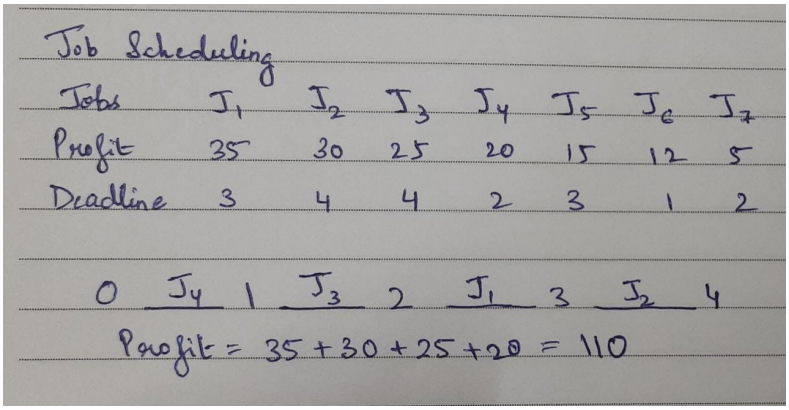
Each job takes 1 unit of time

Complete jobs to get maximum profit following the deadlines

First step arrange jobs in descending order of profit

Second step, pick each job and check if it is feasible





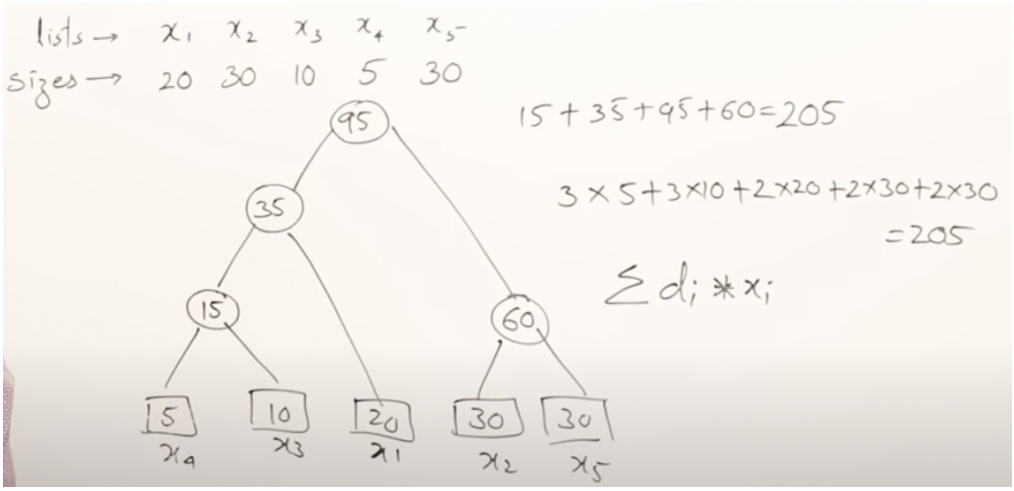
3.Optimal Merge Pattern:

So if you have to do two way merging. The most optimal way is this:

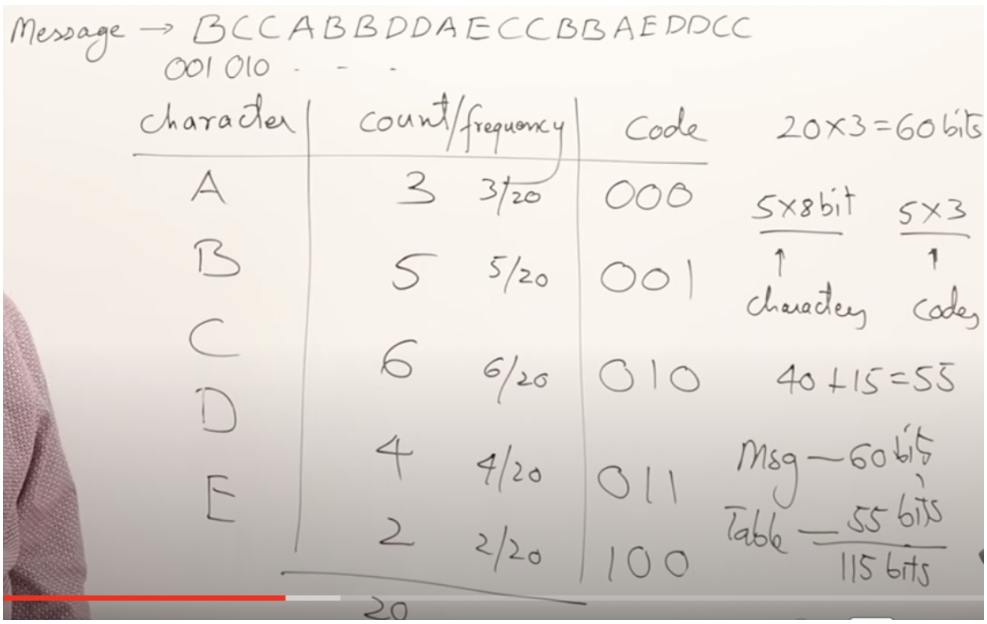
Suppose we have the following lists with their sizes i.e. no of elements in each list.

We keep taking two minimum size lists and merge them

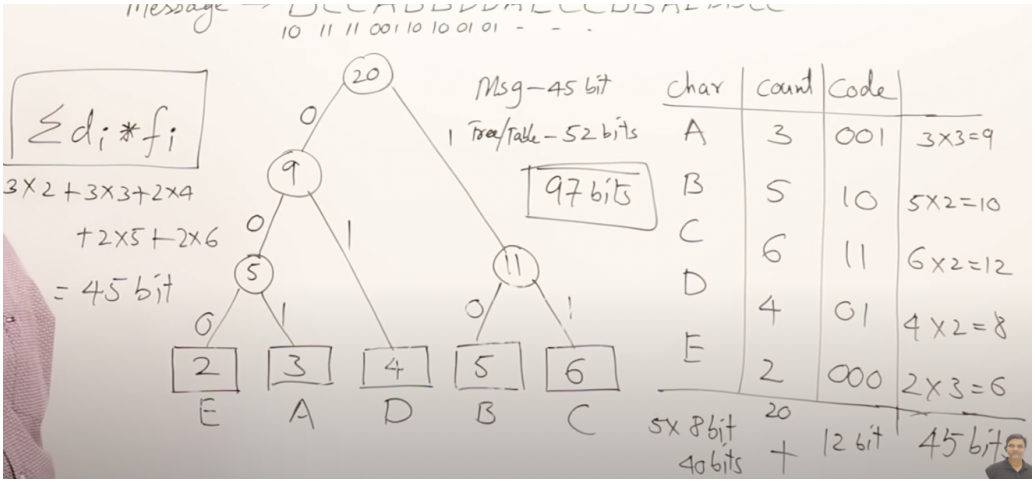
We keep taking two minimum size lists and merge them



4.Huffman Coding: It is compression technique used to reduce the size of message or data.
So if you have a message say AABCDCAABA. Each letter will be converted into its 8bit ASCII code while sending.
Instead you can create your own code like in this case we take 3 bits because we need to fit 5 alphabets.
Now along with the converted message we also need to send the tabla of code so that the destination knows how to decode. So just take 8 bits+3 bits for each character in the case below.



In Huffman, you can take codes of different length depending on how much their frequency is in the message using optimal merge.
So first we do optimal merge and every left edge is labelled 0 and every right edge is labelled 1.
Now from root go down till you reach your character while adding the digit on the edge to your code.
Example: A is 001
Then simply multiply each character's count with the number of digits in its code.
You can also find this by multiplying distance from root to character with frequency.
Then again add 8bits* number of characters and number of bits used in writing your own code that will used in decryption.



To decode the message, simply make the tree from the codes sent and then sequentially go through each digit and whenever you reach a character, you again start from root and add that character to the result

YouTube Video

3.4 Huffman Coding - Greedy Method

5. Minimum Cost Spanning Tree

Let no. of vertices in the graph be V. Then edges in spanning tree should be $|E|=|V|-1$

No. of spanning trees possible= ${}^{|E|}C_{|V|-1}$ - no. of cycles

To find minimum cost spanning tree, you have 2 methods:

1. Prim
2. Kruskal

Prim:

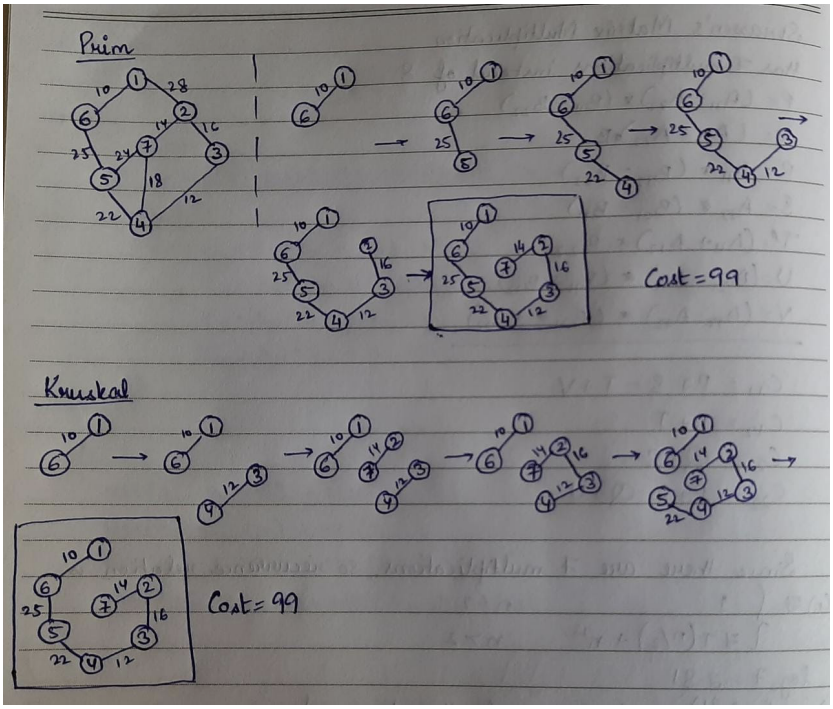
1. Select the minimum cost edge
2. Keep selecting the minimum cost edge making sure the edge is connected to one of the vertices already taken.
3. Prim can't be used for non-connected graph

Kruskal:

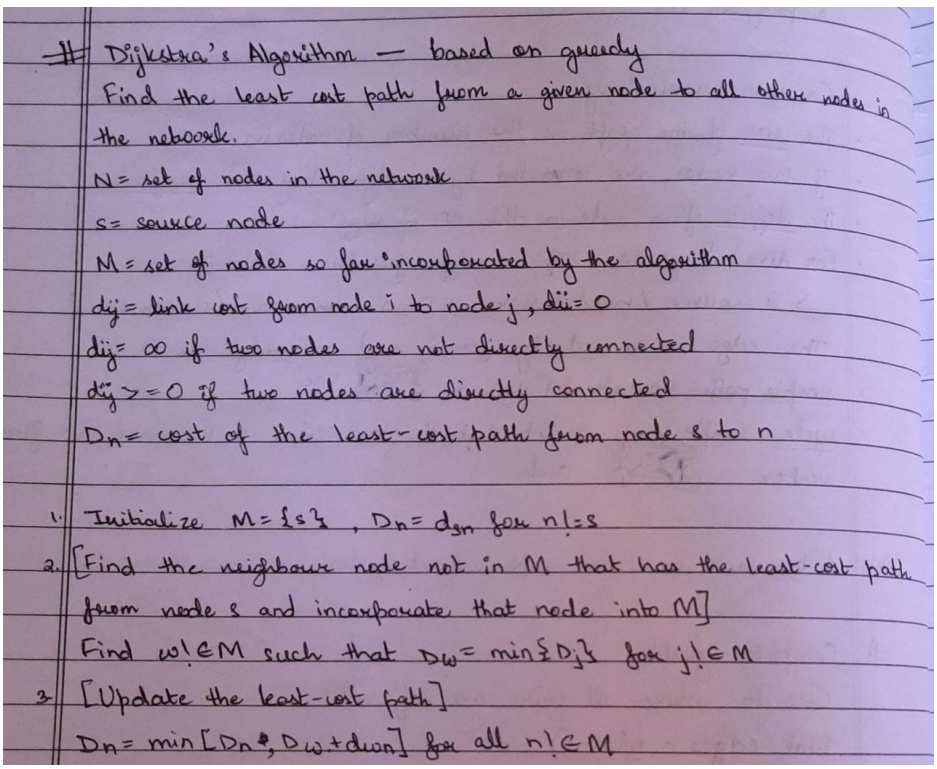
1. Keep selecting the minimum cost edge
2. If selecting an edge creates a cycle, ignore that edge
3. Can be used to solve non-connected graph partially

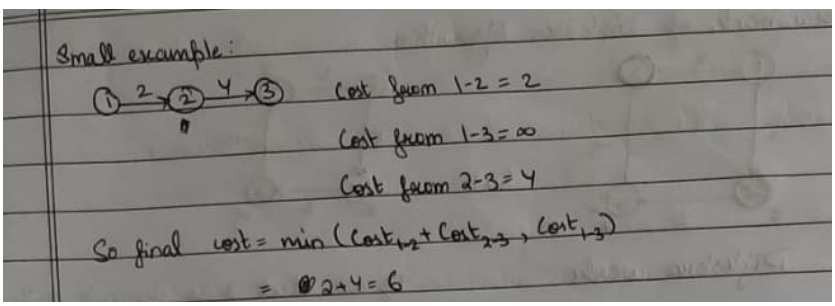
Time Complexity: O(|V|*|E|)=O(n^2)

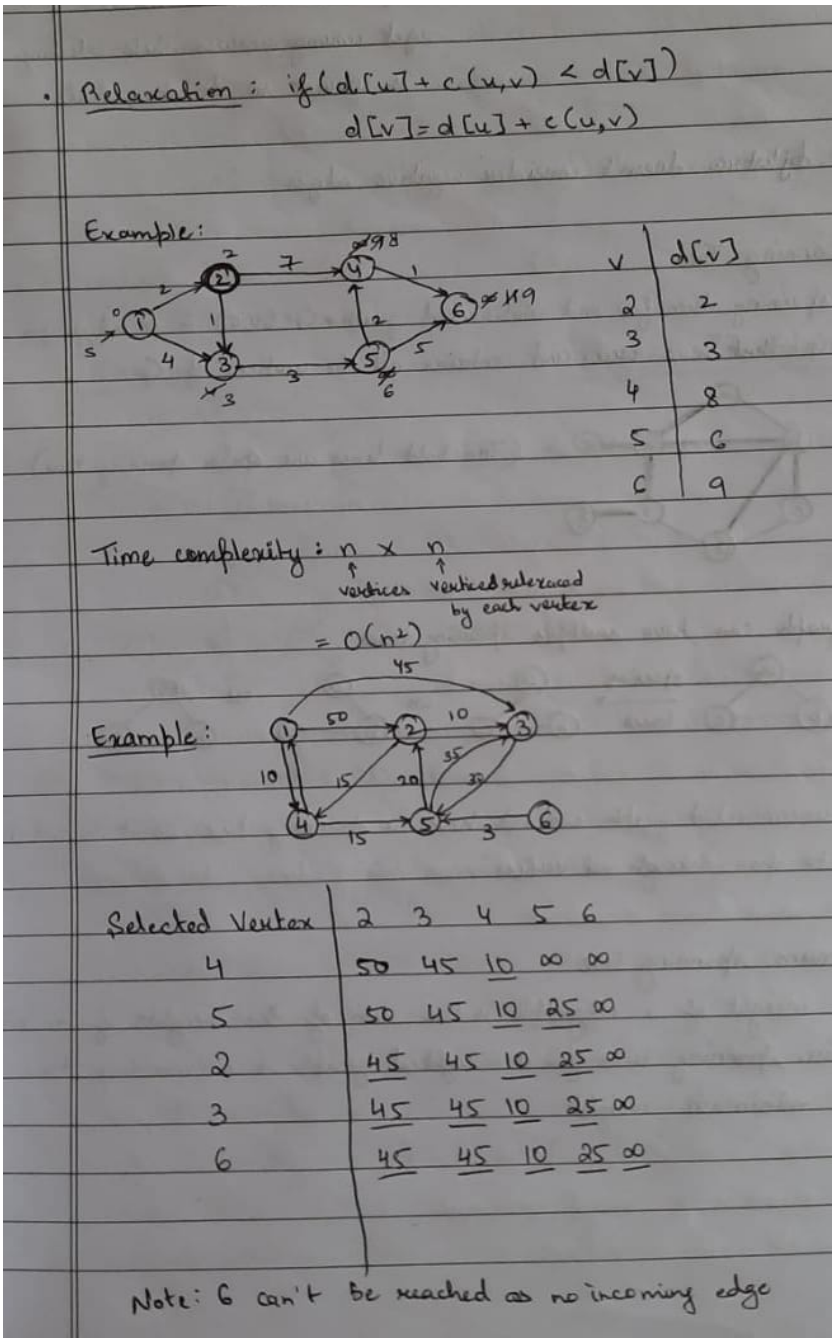
But this can be minimized in Kruskal using min heap. Deleting in min heap takes O(logn) time and it always gives the minimum element. So we store all the edges in the min heap and every step keep deleting one element to get the minimum edge. So time complexity= O(nlogn)

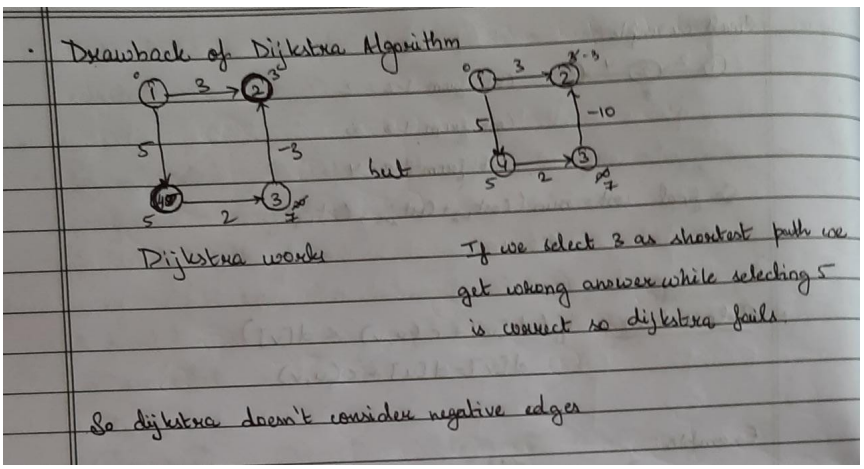


6.Dijkstra Algorithm:









Dynamic Programming: dividing a particular problem into subproblems and then storing the result of these subproblems to calculate the result of the actual problem.

Recursive program for fibonacci:

```
int fib(int n)
{
  if (n <= 1)
    return n;

  return fib(n-1) + fib(n-2);
}
```

```
                      fib(5)
                    /        \
               fib(4)          fib(3)
             /     \           /    \
        fib(3)     fib(2)    fib(2)  fib(1)
       /    \       /   \     /   \
  fib(2)  fib(1) fib(1) fib(0) fib(1) fib(0)
  /   \
fib(1) fib(0)
```

We can see that the function fib(3) is being called 2 times. If we would have stored the value of fib(3), then instead of computing it again, we could have reused the old stored value.
The time complexity is 2^n

Dynamic programming approach for fibonacci

```
int fib(int n)
{
 int f[n+2];
 int i;
 f[0] = 0;
 f[1] = 1;

 for (i = 2; i <= n; i++)
 {
   f[i] = f[i-1] + f[i-2];
 }
 return f[n];
}
```

Time complexity is O(n)

Multistage Graph: weighted directed graph. Find path with minimum cost