1. Time Complexity of recursive programs

//Lecture 2a

1.1) Introduction to recurrence relation :

```
int fun(n){
    if(n<=1)return 1;
    else return fun(n/2) + n;
}
```

T(n) ← Time taken by program with n input value

} 1 ⟹ $T(n) = T(n/2) + 1$

} $T(n/2)$

Not ~~$T(n) = T(n/2) + n$~~

Similarly,

```
int fun(n){
    ...
    return 2*fun(n/2);
}
```

$T(n) = T(n/2) + 1$

≠

```
int fun(n){
    ...
    return fun(n/2) + fun(n/2);
}
```

$T(n) = 2T(n/2) + 1$

Here you are not calling fun(n/2) twice it's just multiplied it's answer by 2.

There are mainly three types of solving recurrence relations

1.1.1) Iteration method or Repeated substitution method :

Example 1 : T(n) = T(n-1) + 1

$T(n) = T(n-1) + 1$ But $T(n-1) = T(n-2) + 1$

$\therefore T(n) = T(n-2) + 2$ ...

$\therefore T(n) = T(n-k) + k$ if $T(1) = 1$ then $T(n) = T(1) + n - 1 = n$

Example 2 : $T(n) = \begin{cases} T\left(\frac{n}{2}\right) + n, & n > 1 \\ 1, & n \le 1 \end{cases}$

$T(n) = T(n/2) + n = T\left(\frac{n}{4}\right) + \frac{n}{2} + n = T\left(\frac{n}{2^k}\right) + \frac{n}{2^{k-1}} + \ldots + \frac{n}{2} + n$

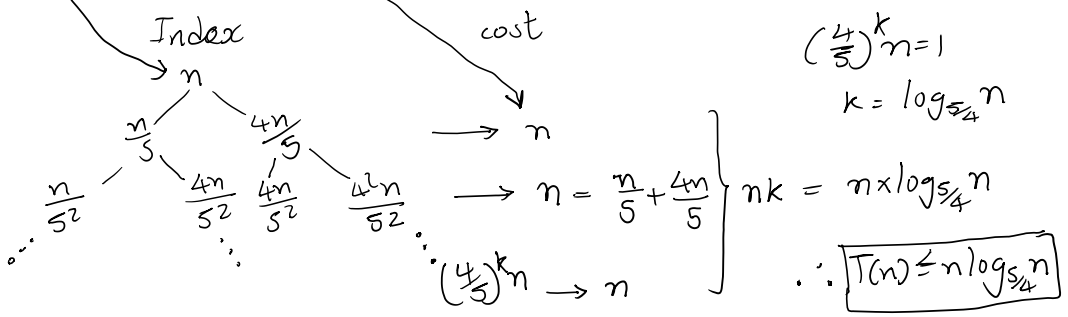
If $2^k = n$ then $T(n) = T(1) + 2 + 4 + \ldots + \frac{n}{2} + n$

$T(n) = T(1) + n\left[1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^{k-1}}\right] = T(1) + n \times \frac{2^k - 1}{2^{k-1}}$

$\therefore T(n) = 2n - 1 = \Theta(n)$

1.1.2) Tree method :

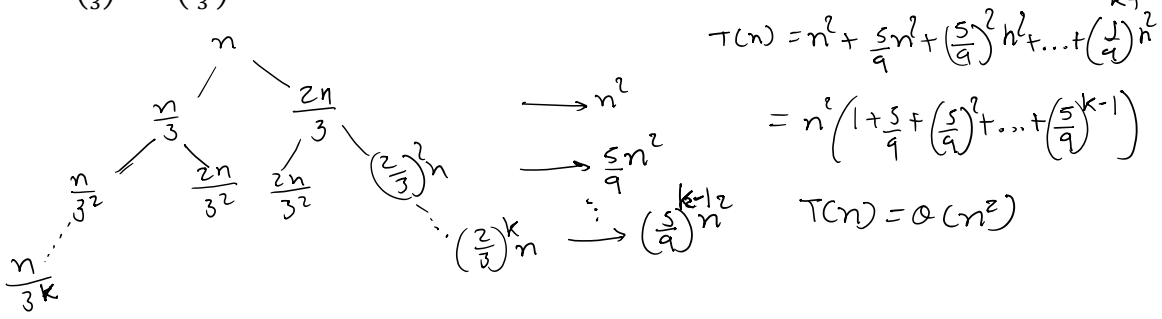
To solve $T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right) + n$ applying iterative method would be lengthy and confusing so here we follow tree method



Now, here where to stop so we stop at in worst case ? longest height leaf. Here sequence $(4/5)^k$ n will produce longest path and at leaves T(1) should appear.

But $n \log_{5/4} n$ is upper bound, lower bound can be found by taking shortest path of the tree, which is formed by $n/5^k$ sequence. So, $n \log_5 n \leq T(n) \leq n \log_{5/4} n$ ➔ $n \log_2 n \leq T(n) \leq n \log_2 n$ as $\log_{5/4} n = \frac{\log_2 n}{\log_2 \frac{5}{4}} = \log_2 n$. Therefore, $T(n) = \theta(nlgn)$

If $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n^2$ is given then



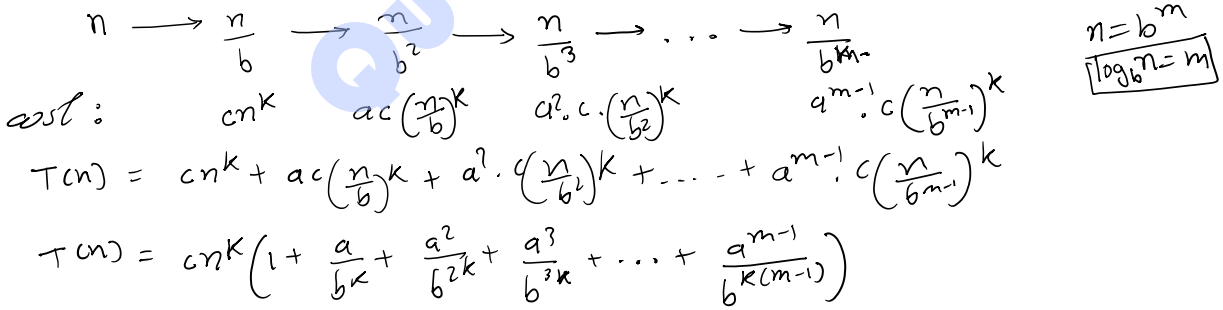
$$1 + c + c^2 + \cdots + c^n = \sum_{i=1}^{n} c^i = \begin{cases} \boldsymbol{\theta(1)}, & \boldsymbol{if\ c < 1} \\ \boldsymbol{\theta(n)}, & \boldsymbol{if\ c = 1} \\ \boldsymbol{\theta(c^n)}, & \boldsymbol{if\ c > 1} \end{cases}$$

NOTE :

1) **If you are using tree method then always find lower and upper bound and then conclude $\theta(f(n))$. $T(n) = T(n-1) + T(n-2) + 1$ here $\Omega\left(2^{\frac{n}{2}}\right) \leq T(n) \leq O(2^n)$ so, $T(n) \neq \theta(2^n)$.**

//Lecture 3d

Q : Solve $T(n) = aT\left(\frac{n}{b}\right) + cn^k$ –



Using GP formula discussed above.

$$\boldsymbol{T(n) = \theta\left(n^k . g(n)\right)}$$

$$\boldsymbol{where\ g(n) = \begin{cases} 1 & if\ a < b^k \\ \log_b n & if\ a = b^k \\ \left(\frac{a}{b^k}\right)^{\log_b n} & if\ a > b^k \end{cases}}$$

//Lecture 4a

1.2) Master theorem :

In above example, we got one general formula where we can say $f(n) = n^k . g(n)$ but we extend this idea to more general function f(n).

We want to solve $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, we will get following results (we will see proof later)

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. → $f(n) \leq n^{\log_b a}$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$. → $f(n) = n^{\log_b a}$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■ → $f(n) \geq n^{\log_b a}$

To apply this method, we first find $n^{\log ba}$ then compare it with f(n).

//Lecture 4b

Here one thing to note that when we compare f(n) with $n^{\log ba}$, f(n) should be polynomial greater than $n^{\log ba}$. For example, $T(n) = 2T\left(\frac{n}{2}\right) + nlogn$

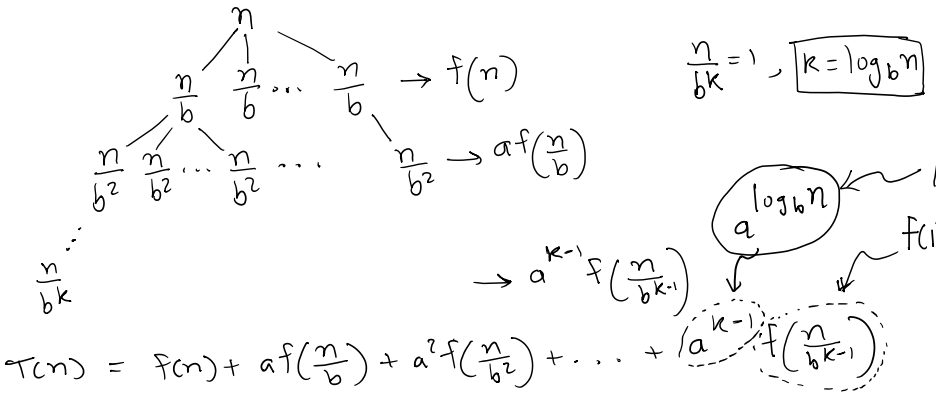
$nlogn \;?\; n^{\log_2 2} \implies \boxed{\log n > 1}$ Not polynomial greater so, can't apply master theo.

If instead of nlogn, $n^2 logn$ were there then $n\log n > 1$, which is polynomially greater so we can apply master theorem here.

If instead of nlogn, 2^n were there then $2^n > 1$ this is also polynomially greater because you can always write $2^n >$ n (polynomial) > 1. Similar case for n!.

//Lecture 4c

Proof of master theorem :



$$\frac{n}{b^k} = 1, \quad \boxed{k = \log_b n}$$

$$T(n) = f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \ldots + a^{k-1} f\left(\frac{n}{b^{k-1}}\right)$$

If $f(n) > a^{\log_b n}$ then series is decreasing means $T(n) = \Theta(f(n))$

If $f(n) < a^{\log_b n}$ then series is increasing means $T(n) = \Theta(a^{\log_b n})$

If $f(n) = a^{\log_b n}$ then $f(n) = f\left(\frac{n}{b}\right) = f\left(\frac{n}{b^2}\right) = \ldots = a^{\log_b n}$

$\therefore T(n) = f(n)\left(1 + a + a^2 + \ldots + a^k\right) = \Theta(f(n) \cdot \underbrace{\log_b n}_{k})$ from GP series

//Lecture 5a

1.2.1) Generalized Master theorem :

We know master theorem cannot solve recurrence relations in which f(n) is not polymonially comparable to n^{logba}. To resolve this problem, we simply change 2nd condition of master theorem as follows :

← after updating 2nd condition it become Generalized MT.

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$. ← $f(n) = \Theta(n^{\log_b a} \log^k n)$ then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Meaning $T(n) = 2T\left(\frac{n}{2}\right) + nlogn$ f(n) ? $n^{\log_2 2} \Rightarrow n$? $n \log n$

$$T(n) = \Theta(n \log^2 n)$$

//Lecture 5b

Now, can we solve $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{logn}, T(2) = 1$? – Yes, using substitution

$T(n/2) = 2T(n/4) + \frac{n/2}{\log n/2}$, $T(n) = 2^2 T\left(\frac{n}{2^2}\right) + \frac{n}{\log n/2} + \frac{n}{\log n}$

$\therefore T(n) = 2^k T\left(\frac{n}{2^k}\right) + \cdots + \frac{n}{\log n/2} + \frac{n}{\log n}$ $\quad \frac{n}{2^k} = 2$ $\quad k = \log n - 1$

$\therefore T(n) = \sum_{i=0}^{\log n - 1} \frac{n}{\log n/2^i} = n \sum_{i=1}^{\log n} \frac{1}{i} = \Theta(n \lg \lg n)$ .... $\sum_{i=1}^{n} \frac{1}{i} = \log n$

We cannot solve it using master theorem because we have solved log n case but it was in numerator, Similarly, can we solve $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{(logn)^2}, T(2) = 1$? – Yes, again we use substitution. This time we will get,

$$T(n) = \sum_{i=1}^{\log n} \frac{n}{i^2} = \Theta(n) \quad \text{... as } \sum_{i=1}^{\log n} \frac{1}{i^2} = \frac{\pi^2}{6} \text{ for } n \to \infty$$

Looks like we have to again extend the master theorem for log n in denominator case. We call this final version of theorem as **Extended master theorem**.

The extended master theorem applies to recurrences of the following form :

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. There are 3 cases :

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \theta(n^{\log_b a})$.
2. If $f(n) = \theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \theta(n^{\log_b a} \log^{k+1} n)$.

- If $f(n) = \theta(\frac{n^{\log_b a}}{\log n})$, then $T(n) = \theta(n^{\log_b a} \log\log n)$.
- If $f(n) = \theta(\frac{n^{\log_b a}}{\log^p n})$ with $p \geq 2$, then $T(n) = \theta(n^{\log_b a})$.

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \theta(f(n))$.

Q : solve $T(n) = 4T\left(\frac{n}{2}\right) + \frac{n}{(logn)^2}$ – here we do not directly say $T(n) = \theta(n)$ by using 2nd part of extended master theorem. We will first check

$\frac{n}{(\log n)^2}$? $n^{\log_2 4} \Rightarrow$ $1 < n\log n$ ← This is polynomially greater

$\therefore f(n) = O(n^{\log_2 4})$ $\therefore T(n) = \theta(n^2)$

//Lecture 5d

1.2.2) Problem that master theorem cannot solve :

Example 1 : $T(n) = \sqrt{n}\, T\left(\frac{n}{2}\right) + n$
Here $a = \sqrt{n}$ is not constant

Example 2 : $T(n) = 2T\left(\frac{n}{\log n}\right) + n^2$
Here b is not constant

Example 3 : $T(n) = \frac{1}{2}T\left(\frac{n}{2}\right) + n^2$
Here $a = 1/2$ is not ≥ 1

Example 4 : $T(n) = 2T\left(\frac{3n}{4}\right) + n$
Here $b = 3/4$ is not > 1.

Example 5 : $T(n) = 3T\left(\frac{n}{2}\right) - n$
Here $f(n) = -n$ is not positive

Example 6 : $T(n) = 2T\left(\frac{n}{2}\right) + n^2 \sin n$
Here Trigonometry not allowed

Example 7 : $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n$
Here : a & b are not fixed

//Lecture 6a

1.3) Change to variable :

As name indicate we replace existing variable by some variable to simplify recurrence relation.

$T(n) = T(\sqrt{n}) + \sqrt{n}$ To solve such type of problem, using substitution it will be difficult. So, we assume n = 2^m.

$T(2^m) = T(2^{m/2}) + 2^{m/2}$ $\xrightarrow{S(m) = T(2^m)}$ $S(m) = S(m/2) + 2^{m/2}$

$\therefore S(m) = \theta(2^{m/2}) \Rightarrow T(2^m) = \theta(2^{m/2}) = T(n) = \theta(\sqrt{n})$

Solve : $T(n) = \left(T\left(\frac{n}{161}\right)\right)^{161} . n$ – When something related to power of T(n) is given we take log on both sides.

$\log(T(n)) = 161 \times \log\left(T\left(\frac{n}{161}\right)\right) + \log n$ $\xrightarrow{S(n) = \log(T(n))}$ $S(n) = 161.S\left(\frac{n}{161}\right) + \log n$

$S(n) = \theta(n) \Rightarrow$ $T(n) = \theta(2^n)$

Solve : $T(n) = \sqrt{n}T(\sqrt{n}) + 100n$ – when coefficient is not constant then try to remove it or try to make it another function. For example,

$\frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + 100$ ... Divide by n

$\Longrightarrow S(n) = S(\sqrt{n}) + 100 \xrightarrow{n=2^m} R(m) = R(m/2) + 100 \Longrightarrow R(m) = \Theta(\log m)$

$S(n) = \Theta(\log\log n) \xrightarrow{T(n) = nS(n)} \boxed{T(n) = \Theta(n \log\log n)}$

2. Divide and Conquer Algorithms

//Lecture 7a

2.1) Introduction to Divide and conquer algorithms :

Break up a problem into smaller subproblems. Solve those subproblems *recursively*. Combine the results of those subproblems to get the overall answer.

Total time to solve whole problem $\boldsymbol{T(n) = divide\ cost + \underbrace{T(n_1) + T(n_2) + \cdots + T(n_k)}_{\text{subproblems}} + Combine\ cost}$.

3 recursion rules :

1) Always have at least one case that can be solved without using recursion. (base case)
2) Any recursive call must progress toward a base case. (finite step loop)
3) Always assume that the recursive call works, and use this assumption to design your algorithms.

//Lecture 7b

2.1.1) Divide and conquer algorithms :

1) Maximum of an array :

```
Maximum(a, l, r){
    if(r == l) return a[r];      //Base case
    m = (l+r)/2;
    max1 = Maximum(a, l, m);     //Divide
    max2 = Maximum(a, m+1, r);
    return max(max1, max2);      //Conquer
}
```

T(n) = Divide + Solving subproblem + combine cost

$T(n) = O(1) + 2T\left(\frac{n}{2}\right) + O(1) = O(n)$

You also modify this problem to find sum of array element.

//Lecture 7c

2) Search in an array :

$T(n) = 2T\left(\frac{n}{2}\right) + 1 \Rightarrow T(n) = O(n)$

```
bool search(int*a, int i, int j, int value){
    if(i==j) return a[i] == value;
    int mid = (i + j)/2;
    return search(a, i, mid, value) || search(a, mid+1, j, value);
}
```

OR because it may not present in subpart 2

//Lecture 7e

3) Dumb sort :

```
void sort(int *a, int i, int j){
    if(i == j) return;
    sort(a, i+1, j);
    for(int k = i + 1; k <= j; k++){
        if(a[k-1] > a[k])swap(a[k-1], a[k]);
    }
}
```

Go to last element & then start comparing & swapping

$T(n) = T(n-1) + O(n)$

$T(n) = T(n-1) + O(n) = O(n^2)$

//Lecture 8a

2.1.2) Introduction to merge sort algorithms :

- Divide array into two halves
- Recursively sort each half
- Merge two halves to make whole array sorted

```
MergeSort(a, i, j){ <-- T(n)
    if(i==j) return;
    mid = (i+j)/2;  <-- divide cost θ(1)
    MergeSort(a, i, mid);   } 2T(n/2)
    MergeSort(a, mid+1, j);
    Merge(a, i, mid, j); <-- combine cost ?
}
```

//Lecture 8b

Merge procedure should combine two array sorted array into one sorted array.

As both arrays are sorted so one of the first index of array contains minimum. We compare both values then put the minimum value in new array of size m+n. Now we increment index of array from which minimum was selected. We do this procedure iteratively till both indexes of array points to last element.

```
Merge(a, i, mid, j){
    l1 = i, l2 = mid+1, k = 0; //k represents index of new array b[]
    while(l1<=mid && l2<=j){
        if(a[l1] <= a[l2]) b[k] = a[l1++];
        else b[k] = a[l2++];
        k++;
    }
    if(l1<=mid) copy remaining elements of 1st half;
    if(l2<=j) copy remaining elements of 2nd half;
    copy b to a;
}
```

O(m+n)

$\therefore T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

$\therefore T(n) = \Theta(n \log n)$

//Lecture 8c

Number of comparisons in merge :

1) At worst we have to compare every element, it is obvious that maximum element would not get compared with anyone at last because it is only remaining element. So, at max m + n – 1 comparison required.
2) At min we can have case where all elements of one array is less than other array. In that case we only do min(n, m) comparisons.

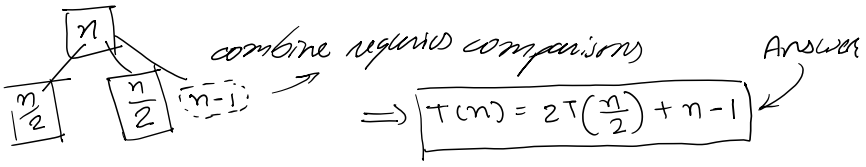
//Lecture 9a

2.1.3) Merge sort questions :

A sorting algorithm falls into the **adaptive sort** family if it takes advantage of existing order in its input. It benefits from the pre-sorted sequence in the input sequence. Example, Merge sort is not adaptive algo but insertion sort is.

//Lecture 9c

Worst case recursive equation for no. of comparisons in merge sort :



T(n) : No. of comparisons in merge sort for n number.

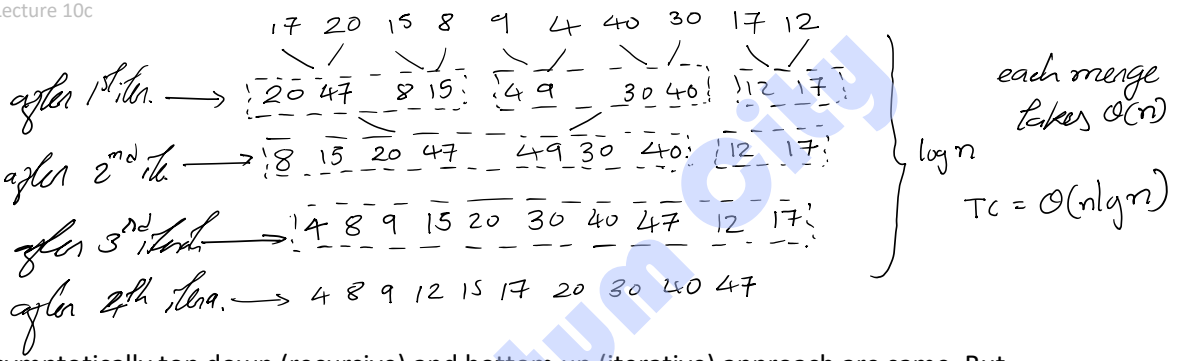
Note that no. of comparisons in merge sort is asked not in merge procedure. If merge is asked then answer will be just n-1.

//Lecture 10a

2.1.4) Iterative merge sort : also known straight two-way merge sort algorithm or bottom up appro.

Idea is to form group of two then four … till array size. And iteratively sorting and merging elements of subarray.

//Lecture 10c



Asymptotically top down (recursive) and bottom up (iterative) approach are same. But

Top down = split + combine (here are splitting it using int mid)

Bottom up = only combines. (here we form group for that also we do groupsize*2 after every iteration)

common in both	Different in both
(i) 'Merge' procedure is same	(i) One is Top down other is bottom up
(ii) Time complexity = O(n log n)	(ii) Top down does split which is extra
(iv) both sorts array	

If we say merge sort by default it is top down (recursive) approach if explicitly mentioned about bottom up by "straight two way" or by using other word.

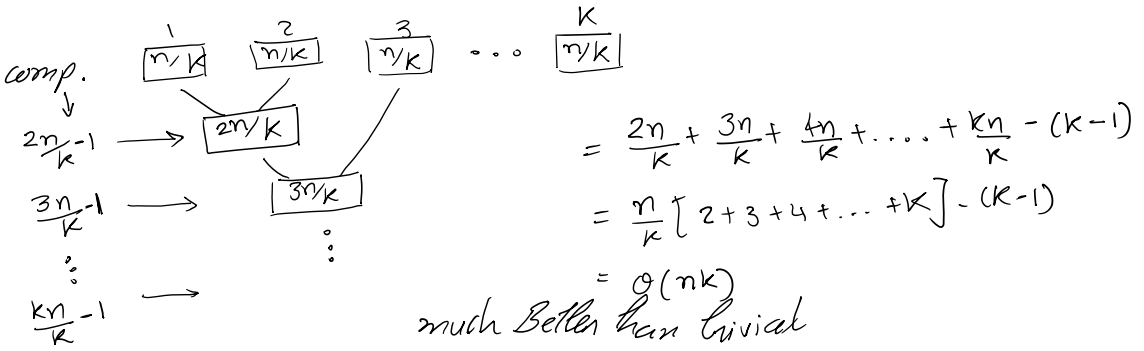
```
for(int grsize = 1; grsize<a.length(); grsize *= 2){
    //Combine pairs of array a of size "grsize"
    for(int i = 0; i<a.length; i += 2*grsize){
        int left = i;
        int mid = i + grsize - 1;          //end of first group
        int right = i + 2*grsize - 1;      //end of second group
        merge(a, left, mid, right);        //combine first and second group
    }
}
```

//Lecture 11a

2.1.5) Merging k sorted arrays :

We will explore 5 methods to merge k sorted array into one sorted array.

1) **Trivial method** : we will combine k sorted array then sort it will take nlogn. But here we are not taking advantage of pre-sorted sequence of element.
2) **Successive Mergeing method** :



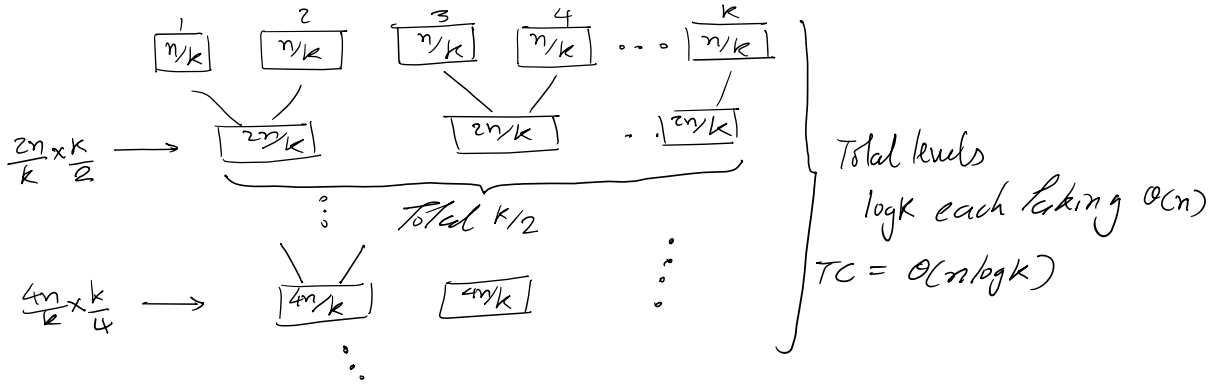
3) **Successive minimum method** : we will take first element of all k sorted array we will find minimum repeatedly. To find first element of final sorted array we need k comparisons. So, for n element we need nk comparison. So, time complexity is $\theta(nk)$. Can we do better ?

we have observed that in successive minimum method if we have selected first minimum then we increment a point to next node in respective sorted array. and we again find minimum and for that we again have to traverse whole k size array. but we have already done comparison between element which was previously used. So is there any method which provides better minimum finding ?

4) **Improvement of 3rd method** : we can improve 3rd method using heap data structure. First, we make min-heap out of all first elements from the k sorted arrays. Which will take O(k) time. Then we extract root, place it into final sorted array and insert new element into min heap which will take O(lg k) time. For each n – k – 1 element we repeat this process which will take total of O(nlog k + k) time = $\theta(n \log k)$.

//Lecture 11b

5) **Bottom up method** : This method is similar to iterative version of mergesort.



Properties of sorting :

- **Stable sorting** : A sorting algorithm is stable if elements with the same key appear in the output array in the same order as they do in the input array.

- **In-place sorting** : A sorting algorithm is said to be an in-place sorting algorithm is the amount of extra space required by the algorithm is $\theta(1)$. That is, the amount of extra space is bounded by a constant, independent of the size of the array.

Note that if sequence of different sorted arrays is to be sorted using optimal merge algorithm then algo chooses smallest sequences for merging.

//Lecture 12a

2.2) Maximum and minimum of numbers :

Finding maximum is like knockout tournament. First, we set first element as max then we compete with second element whoever wins (large value) we update max. we do this till last element.

There can be many methods to find max, one of them is we do pairwise knockout tournament. we will do n/2 + n/4 + … n/n comparisons. TC will be n-1. What if we want 1st runner or second max. We keep tract of second max as when we set some new element to max we give this value to second max and if while traversing we get value greater than second max but less than max then we set that value to the second max.

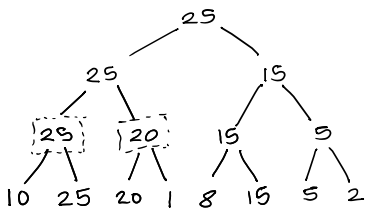
```c
#include<stdio.h>
int main(){
    int arr[] = {4, 2, 5, 6, 18, 1, 7};
    int len = sizeof(arr)/sizeof(arr[0]), max1 = arr[0], max2;
    for(int i = 1;i<len; i++){
        if(max1 < arr[i]){
            max2 = max1;
            max1 = arr[i];
        }
        else if(max2<arr[i]) max2 = arr[i];
    }
    printf("%d %d", max1, max2);
    return 0;
}
```

we will ignore this comparison

Comparison : best when all element are in increasing order = n-1
worst when we need to check 2 if statement = 1 + 2(n-1)
for initialization of i
On average case for one element
No. comp. = $\frac{1}{2}\times1 + \frac{1}{2}\times2 = \frac{3}{2}$
∴ For n-1 element it would be $(3n-1)/2$

//Lecture 12b

Q : Can we have a smatter method ? – yes using, pairwise tournament method. We do competition between two number which one is largest we will move up in tree.



One observation : Second largest element always gets a competition with largest element. In other words second largest can only be defeated by largest

So, first we find maximum using n – 1 comparison and then we will find 2nd largest element from logn element because 25 gets log n competitors in whole tournament. We will keep track of maximum and second maximum for every node using any data structure. So, for second largest element we will have total of n + log n – 2 comparisons.

//Lecture 12d

2.2.1) Maximum and minimum of an array :

Method 1 : We first look at look at trivial approach. We traverse whole array and make two comparison for max and min.

```c
#include<stdio.h>
int main(){
    int arr[] = {1, 2, 3, 6, 4, 8, 0};
    int max, min, len;
    max = min = arr[0];
    len = sizeof(arr)/sizeof(arr[0]);
    for(int i = 1;i<len;i++){
        if(arr[i]>max) max = arr[i];
        else if(arr[i]<min) min = arr[i];
    }
    printf("%d %d", max, min);
    return 0;
}
```

No. of comparison	Best	worst
	$n-1$	$2(n-1)$
Avg.	$\frac{3}{2}(n-1)$	

Method 2 : CLRS

```c
#include<stdio.h>
int main(){
    int arr[] = {1, 0, 3, 2, 9, -1, -2};
    int max, min, len;
    len = sizeof(arr)/sizeof(arr[0]);
    if(len == 1){
        printf("%d %d", arr[0], arr[0]);
        return 0;
    }
    if(arr[0]>arr[1]){
        max = arr[0];
        min = arr[1];}
    else{
        max = arr[1];
        min = arr[0];
    }
    for(int i = 2;i<len-1;i+=2){
        if(arr[i]>arr[i+1]){
            if(arr[i]>max) max = arr[i];
            if(arr[i+1]<min) min = arr[i+1];}
        else{
            if(arr[i+1]>max) max = arr[i+1];
            if(arr[i]<min) min = arr[i];
        }
    }
    printf("%d %d", max, min);
    return 0;
}
```

We make 2 pairs

If length is 1 max = min = arr[0]

Initialization Take 2 pairs whichever is max we set accordingly.

1 comparison

If first is bigger then compare it with max & compare smaller with min

else to reverse of if statement

3 comparison for pair

Total comparison = $3\left(\frac{n-2}{2}\right)+1 = \left\lceil\frac{3n}{2}\right\rceil - 2$

Not included len == 1 because it is not element comp.

//Lecture 13a

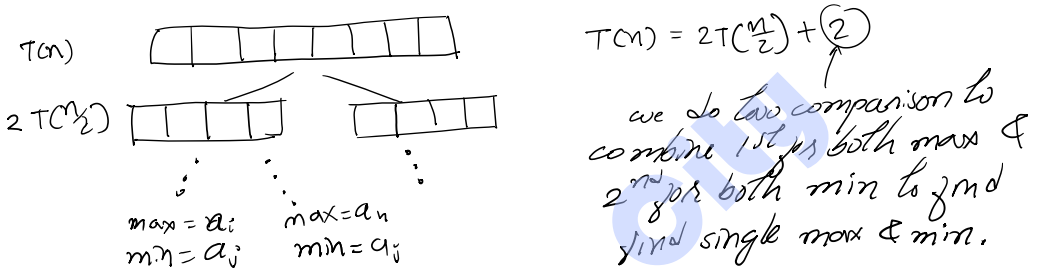
Method 3 : We also have one method which gives same result as CLRS method. But we do not do simultaneous comparison. We use tournament method. We first divide array into two parts then we do match with two number we separate all the winners and losers from first round. Then we find max from winners and min from losers.

Divide or separating out winners and losers will take n/2 comparison. Finding max from winners will take n/2 – 1 comparison and same for finding min from losers. At the end total comparison = 3n/2 – 2. This is comparison is only if number of elements are even if odd then we do previous procedure with n-1 element as they will form even number group.

$\frac{3(n-1)}{2} - 2 + (2)$ ← For the last element, what if it is more than max or less than min !

For even : $\frac{3n}{2} - 2$ For odd : $\frac{3n}{2} - 1.5$ $\Rightarrow \lceil \frac{3n}{2} \rceil - 2$ comparisons

Method 4 : Using divide and conquer



$T(n) = 2T(n/2) + 2 = 2^k T\left(\frac{n}{2^k}\right) + 2^k + 2^{k-1} + \dots + 2$ ⟶ $\frac{n}{2^k} = 2$

$= 2^k T\left(\frac{n}{2^k}\right) + \frac{2(2^k - 1)}{2-1} = 2^k + 2^{k+1} - 2 = \frac{n}{2} + n - 2$

$T(n) = \frac{3n}{2} - 2$ ← but this is when n is even

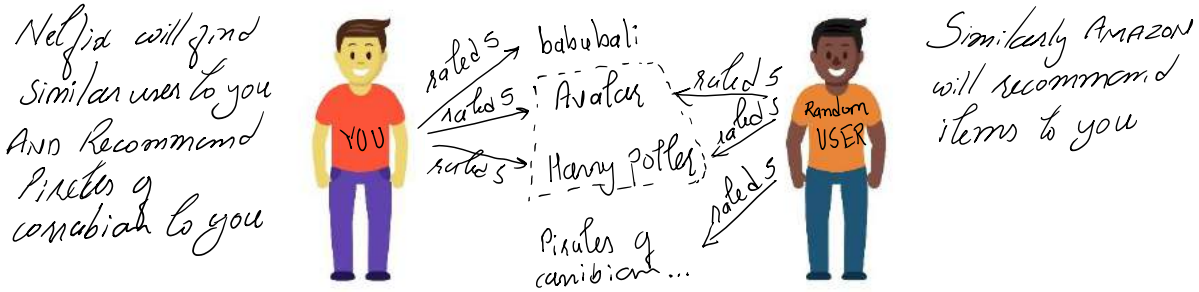
If n is odd $\Rightarrow T(n) = T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + 2 \Rightarrow T(n) = \lceil \frac{3n}{2} \rceil - 2$

Conclusion : *no. of comparison in find min-max when n is even is* $\frac{3n}{2} - 2$ *and n is odd is* $\frac{3n}{2} - 1.5$

//Lecture 15a

2.3) Divide and conquer Algorithm :

Suppose Netflix is recommending some list of movies. How these recommendation works ?



In short, we can say this is how Netflix recommend movies :

Step 1 : Find a similar user to me (this step is crucial)

Step 2 : recommend what I have not watched but user has watched. (this is easy to do)

Q : How to find a similar user to me ? – first we will find user who have watched same movies as you and has given rate according to his choice. You also have rated movies. And suppose we arrange them in sorted order of their rating. We simply count no. of inversion if they are less then we say we have found similar user to you. We call this **counting inversion** problem.

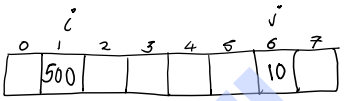
//Lecture 15b

2.3.1) Counting inversions :

We say that two indices i<j forms an inversion if a[i] > a[j].

$$0 \leq Total\ inversions\ with\ n\ elements \leq \frac{n(n-1)}{2}$$

Q : Suppose T is any array of n elements. Show that if there is an inversion (i, j) in T, then T has at least j – i inversions. – It is given that there is inversion (i, j). Suppose we have 500 at i and 10 at j. Two cases are possible,



Case 1 : Number < 10

Suppose at random position between 500 and 10 we have number < 10 then there will definitely be inversion with 500 and not with 10.

Case 2 : Number > 10 and number < 500

If Number is between 500 and 10 then number will have inversion with 500 and 10 as well.

Case 3 : Number > 500 … it will definitely have inversion with 10.

So, in every case we have at least 1 inversion which means for position between j – i we will have at least j – i inversions.

//Lecture 15c

Now, question how computer count inversions ?

Method 1 : Brute force… we will visit each node and for every node will count inversions which will take O(n) time. So, in total time complexity is $O(n^2)$.

Method 2 : Divide and conquer… we will get following recurrence relation at best

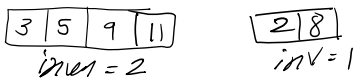
$$T(n) = 2T\left(\frac{n}{2}\right) + O(1) + \frac{n^2}{4}$$

$O(n^2)$ ← $T(n)$; $O(1)$: Divide cost ; $\frac{n^2}{4}$ ← combine cost

//Lecture 16a

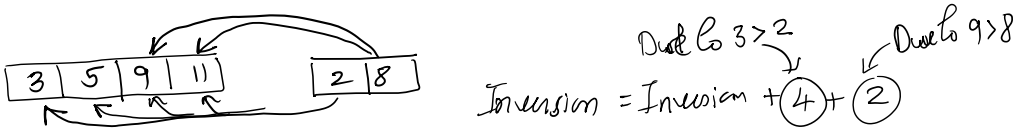
Can we do it better using divide and conquer ? – Yes, using merge sort. In mergesort procedure we will return inversion count.

Suppose at any function call content of array is like this



At this stage return value of first sorted array is 2. And return value of second sorted array is 1. Which means total inversion after this function call must be 2 + 1 + x this is x is number of extra inversion when we merge arrays.

In merge procedure, first we compare 2 with 3 we know 3 and 2 forms inversion but 2 is also forming inversion with rest of the elements. So, we say <u>if some element in second array is less than corr. Element in first array, we will count inversion as inversion + number of elements after 3 (inclusive)</u>.



Similarly, we will do this process until whole array is sorted. Here in merge procedure we just need to add few if else statement to count inversion. So total time complexity of merge procedure will still be o(n) only. Thus, Mergesort will take O(nlogn) and so inversion count.

```
int NewMerge(a, i, mid, j){
    int p = i, q = mid + 1, inversions = 0;
    while(p<=mid && q<=j){
        if(a[p]<=a[q]) b[k] = a[p++];
        else{
            b[k] = a[q++];
            inversions += (mid - p + 1);
        }
        k++;
    }
    if(p>mid) copy remaining elements of 2nd half
    if(q>j) copy remaining elements of 1st half
    copy b to a...
    return inversions;
}
```

Now, merge will add all the right, left tree inversions with NewMerge inversions.

```
int NewMergeSort(a, i, j){
    if(i==j) return 0;
    int mid = (i+j)/2;
    int x = NewMergeSort(a, i, mid);
    int y = NewMergeSort(a, mid+1, j);
    int z = NewMerge(a, i, mid, j);
    return x+y+z;
}
```

//Lecture 17a

2.3.2) Problem which can be solved DAC :

1) Closest Pair in 2D :

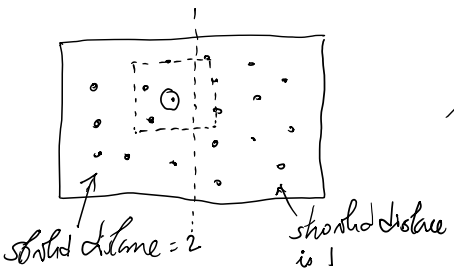
Our target is to find a pair having least distance given n points with (x, y) coordinate.

Brute force approach would be to choose all possible pairs of n points and find their distances and then find min from those distance. So, time taken is $O(n^2)$.

Better approach would be to use divide and conquer. First, we will divide group of points into small subgroup and then we find shortest distance between those pairs and then combine.

Time complexity will be $T(n) = 2T\left(\frac{n}{2}\right) + divide\ cost + Combine\ cost$

//Lecture 17b



Here divides cost is O(1) as we just set limit for dividing purpose. For combine operation, we first form a boundary from the points nearby divide line. And find min distance amongst those points and then combine min. To find min distance first we sort each point according to their x coordinate and y coordinate then find min distance. Assuming we have set our boundary as near as possible to the divide line we can find min distance in o(n) time. Therefore,

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) + O(1) = O(nlogn)$$

//Lecture 17c

2) Exponent of a number :

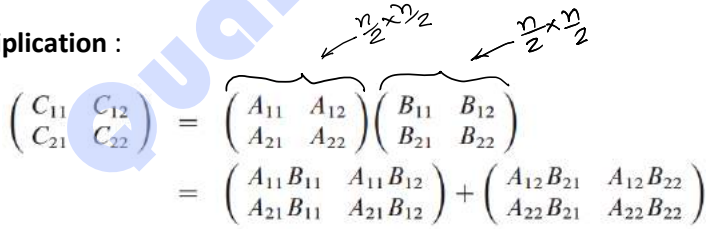
```
//Using brute force :
exponent(a, n){
    if(n==1) return a;
    return a*exponent(a, n-1);
}
T(n) = T(n-1) + 1 = O(n)
```

```
//Using DAC
exponent(a, n){
    if(n==0) return 1;
    if(n==1) return a;
    x = exponent(a, n/2);
    if(n is even) return x*x;
    else return x*x*a;
}
```

T(n) = T(n/2) + 1 = O(log n)

//Lecture 17d

3) Matrix multiplication :



$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$
$$= \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$$

Suppose we use DAC for multiplying matrix so we divide our problem into small sub-matrices. Here matrix C is of order n and then we are dividing matrix into n/2 order. In the recursive case, we partition in O(n) time, perform eight recursive multiplications of n/2 matrices, and finish up with the $O(n^2)$ work from adding two n matrices. Recurrence relation would be

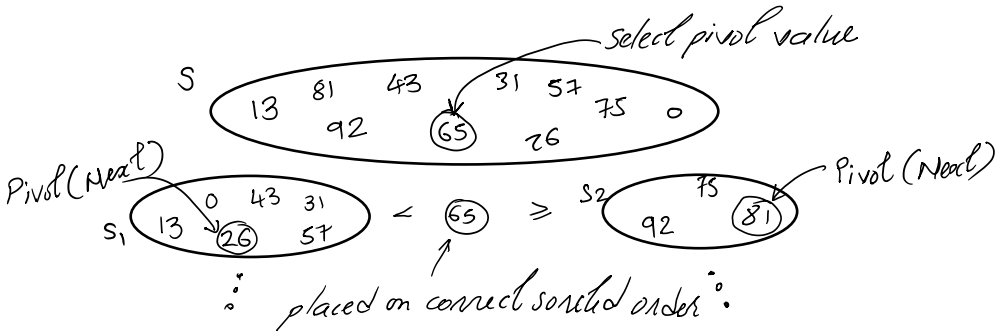
$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2) = \theta(n^3)$$

But Strassen later proved that we need just 7 multiplication so $T(n) = \theta(n^{log7})$

//Lecture 18a

2.3.3) Introduction to quick sort :

Idea is to fix one position of one element. To do that we select one value of element (pivot element) randomly or by some strategy then we make partition of numbers less than and greater than pivot element.



Roughly we are selecting pivot so that we get partition of equal cardinality.

```
quicksort(a, i, j){
    if(i == j) return;
    q = partition(a, i, j);
    quicksort(a, i, q-1);
    quicksort(a, q-1, j);
}
```

function which fixes the pivot and returning index of pivot after fixing and doing partition

Recursively applying quicksort on partitions.

Here partition can be done by any rule. For example, selecting first element of partition as pivot or randomly selecting element as pivot.

```
partition(a, low, high){
    pivot = a[low];
    i = low+1;
    j = high;
    while(i<=j){
        while(i<=j and a[i]<=pivot) i++;          //Finding first element from
left greater than pivot
        while(i<=j and a[j]>pivot) j--;           //Finding first element from
right less than or equal to pivot
        if(i<=j) swap(a[i], a[j]), i++, j--;      //Swap and move ahead
    }
    swap(a[low], a[j]);          //j<i meaning j points to last element less than
pivot which is the position of pivot itself so we swap
    return j;                    //return index of pivot
}
```


Q : Is quicksort stable ? – Consider an array {5, 1, 6, 7, 2(1^{st}), 4, 3, 2(2^{nd}), 7, 8} Let's say pivot is 6.

At some point of time, ↑i ↑j

After swapping, it will become {5, 1, 6, 2(2^{nd}), 2(1^{st}), 4, 3, 7, 7, 8} here you can see that it is not preserving relative position so quicksort is not stable. But it is in-place as can be seen from code.

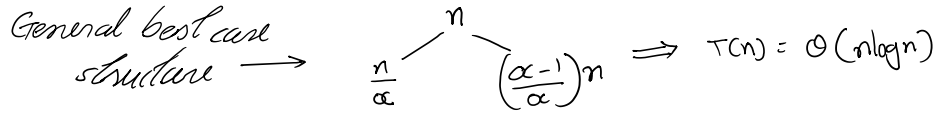
Sorting algo.	Divide cost	Combine cost	Stable	In-place
Mergesort	$\theta(1)$	$\theta(n)$	☑	✗
Quicksort	$\theta(n)$	$\theta(1)$	✗	☑

//Lecture 18c

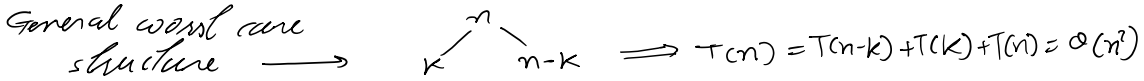
Time complexity :

Best case : if you are very lucky, partition splits the array evenly. $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n) + \theta(1)$

But this is not only the best case when we divide partition into n/1000000 and 999999n/1000000 parts then also time complexity is $\theta(nlogn)$. So, in sort we can say that best case time complexity when partition has following structure,



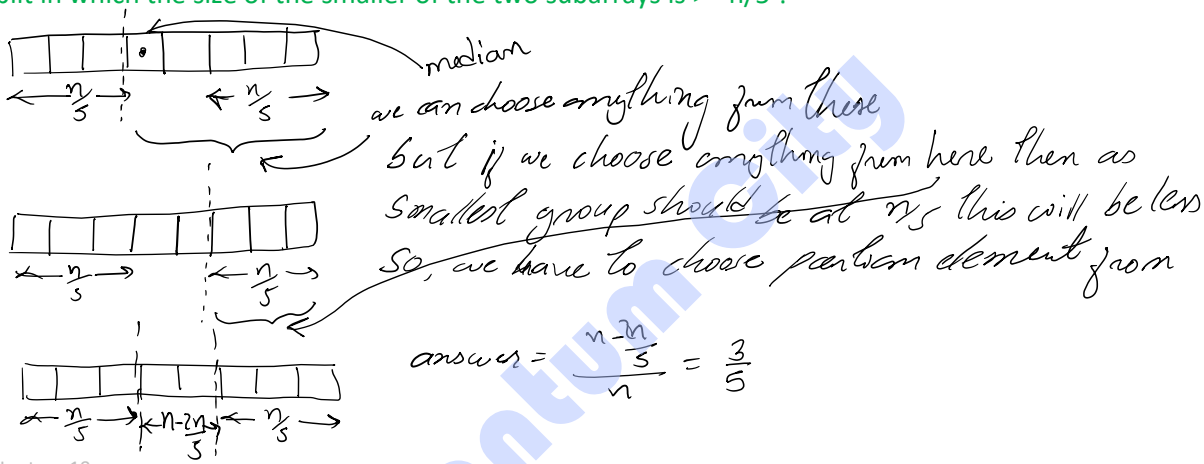
Worst case : On side of the partition has only one element. $T(n) = T(n-1) + T(1) + T(n) = \theta(n^2)$



Expected (average) case : $\theta(nlogn)$

//Lecture 18e

Q : Recall the partition subroutine employed by the quicksort algorithm, as explained in lecture. What is the probability that, with a randomly chosen pivot element, the partition subroutine produces a split in which the size of the smaller of the two subarrays is >= n/5 ? -



//Lecture 19a

2.3.4) The select algorithm :

Input : an unsorted array A of n elements and k

Ex : A : | 7 | 2 | 6 | 9 | 1 | 5 | 4 | 11 |

Output of select algo (select(A, k)) : the kth smallest element of A

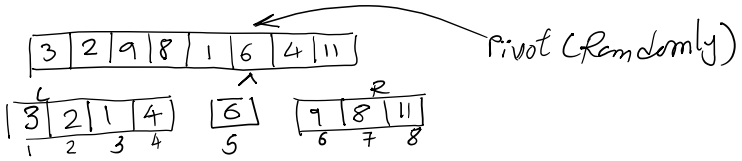
select(A, 1) = 1
select(A, 8) = 11

select(A, 1) = min select(A, n/2) = median & select(A, n) = max

So, select algorithm is the program which can find kth smallest element of A.

We can simply sort array and return index of kth smallest element. But can we do it better ?

We use quicksort partition algo. We know that partition algo returns index of element which has been shorted. So, idea is



We are definitely sure that 5th smallest element is 6 so if we want 3rd smallest it will be left half of partitioned array. Therefore,

- If k = 5, the kth smallest element is the pivot!
- If k < 5, the kth smallest element is in left half

- If k > 5, the kth smallest element is in right half.

```
select(A, p, q, k){
    if(p==q) return A[0];
    m = partition(A, p, q);
    if(m==k) return a[m];
    if(k<m) return select(A, p, m-1, k);
    if(k>m) return select(A, m+1, q, k-m); //k-m because in right array
indexing starts 0 from m+1 index onwards
}
```

Time complexity :

Worst case : $T(n) = T(n-1) + \theta(n) = \theta(n^2)$ not equal partition. So, this is even worst but there is one algo called median of median (which is extremely clever) that gives approximate median. So, you can find kth smallest in $O(n)$.

//Lecture 19b

2.3.5) Binary search :

Given sorted array, search for an element

```
BinarySearch(A, i, j, key){
    if(i==j) if(a[i]==key) return i;
             else return -1;
    mid = (i+j)/2;
    if(a[mid]==key) return mid;
    if(a[mid]>key) return BinarySearch(A, i, mid-1, key);
    else return BinarySearch(A, mid+1, j, key);
}
```

Time complexity
worst case : $O(\log n)$
Best case : $O(1)$
$T(n) = T(n/2) + O(1)$

//lecture 20a

2.4) More sorting algorithm :

2.4.1) Bubble sort :

```
BubbleSort(a){
    for(i = 1 to n-1){
      for(j = 1 to n-1){
        if(a[j]>a[j+1]) swap(a[j], a[j+1]);
      }
    }
}
```

we will not explore last element as it will be sorted

Pass

After every pass maximum elements gets sorted to their place.

But sometimes before ending pass array would be sorted in that case there is no point of traversing array for sorting so we use Boolean variable. At the start of every pass we assign false statement assuming pair is unsorted and then once we swapped, we set value true meaning two pairs are sorted. And after every pass we check if some swapped occurred. If no swapping in inner loop then stop.

```
BubbleSort(a) :
    for(i = 1 to n-1){
        swapped = false;
```

Before this modification
Best case : $O(n^2)$ Worst case : $O(n^2)$ Avg : $O(n^2)$

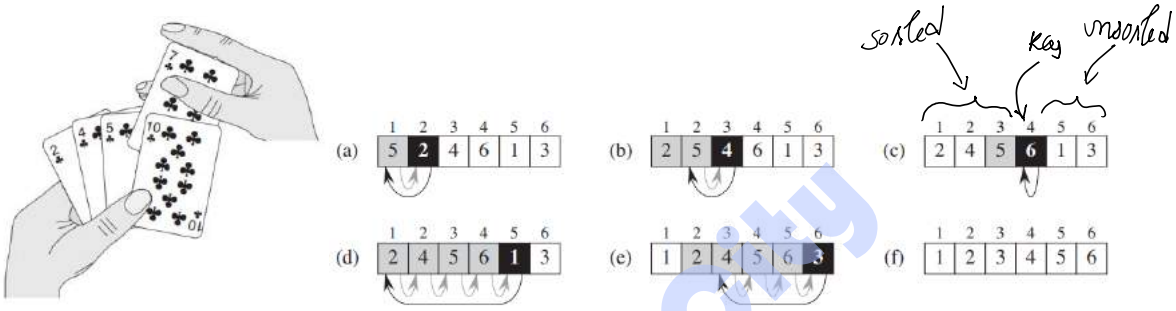
```
        for(j = 1 to n-1){
            if(a[j]>a[j+1]){
                swap(a[j], a[j+1]);
                swapped = true;
            }
        }
        if(swapped == false) break;
    }
```

After modification Time complexity
worst case : array sorted in reverse order $O(n^2)$
Best case : Already sorted $O(n)$

Bubble sort is in-place and stable.

//Lecture 20b

2.4.2) Insertion sort :




```
InsertionSort(A) :
    for(j = 2 to A.length){
        key = A[j];
        //Insert A[j] into the sorted sequence A[1..j-1]
        i = j-1;
        while(i>0 && A[i]>key){
            A[i+1] = A[i];
            i = i-1;
        }
        A[i+1] = key;
    }
```

Time complexity :
Best case : Already sorted $O(n)$
Worst case : Sorted in reverse $O(n^2)$

If array is sorted in reverse order then there will be maximum number of inversions (n-1)+(n-2)+…+1 = n(n-1)/2 and average number of inversions will be $\frac{0+\frac{n(n-1)}{2}}{4} = \frac{n(n-1)}{4}$.

Thus, running time of insertion sort is $\theta(n+d)$ where d is the number of inversions. Merge procedure can be modified to count number of inversions in $\theta(nlogn)$ time.

//Lecture 20d

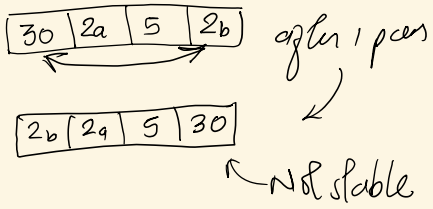
2.4.3) Selection sort :

This is simplest algorithm; on each iteration it finds maximum and put it in last position in array and decreasing size of array after each iteration.

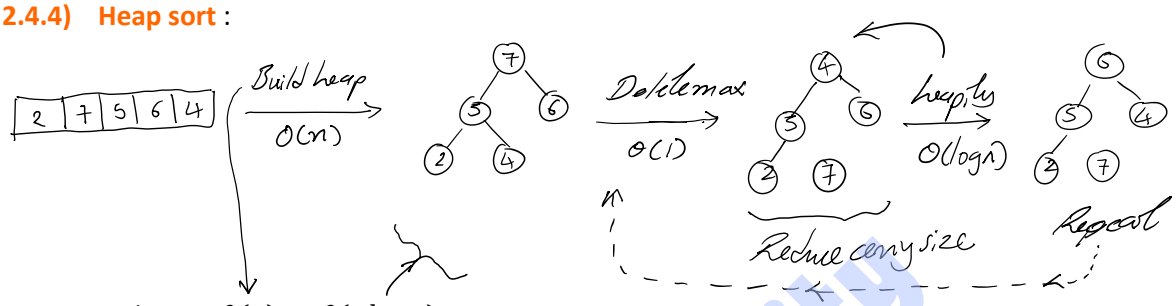
```
SelectionSort(A) :
    length = n;
    for(pass = 1; pass<=n-1; pass++){
        max = A[0];
```

Time complexity :
All case : $O(n^2)$

```
        maxIdx = 0;
        for(j = 1;j<length;j++){
            if(a[j]>max){
                maxIdx = j;
                max = a[j];
            }
        }
        swap(a[maxIdx], a[length-1]);
        length--;
    }
```



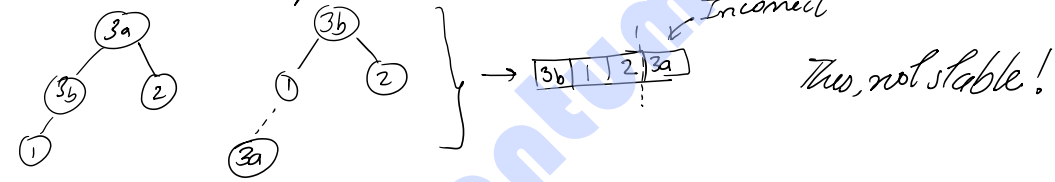
2.4.4) Heap sort :



Time complexity : $\theta(n) + \theta(nlogn)$

Q : Is it stable ? – Consider below case,

But here in heap sort,



Summery,

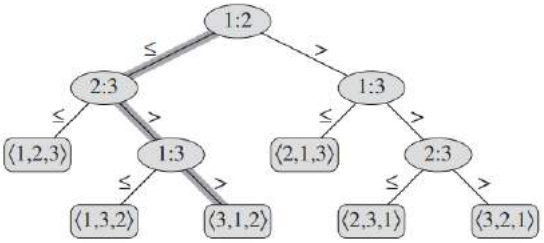
	Best	Average	Worst	Stable?	In place?
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	✗	☑
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	☑	☑
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	☑	☑
Heap sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	✗	☑
Merge sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	☑	✗
Quick sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	✗	☑

Among $O(n^2)$ sorts, insertion sort is usually better than selection sort and both are better than bubble sort.

//Lecture 20e

Theorem : *Any comparison-based sorting algorithm must take atleast n log n time in worst case*.

You have noticed that in every algorithm we are comparing two elements. Any comparison-based algorithm needs to be able to output any one of **n!** possible orderings.



You also know that no. comparisons you are doing to obtain one of the possible ordering is $O(depth)$.

You can observe that above diagram is nearly complete binary tree with L leaves and d depth where L = n! and depth should be $d \geq \log_2 L$ or $2^d \geq L$. d is nothing but comparisons so no. of comparison should $\Omega(nlogn)$. QED.

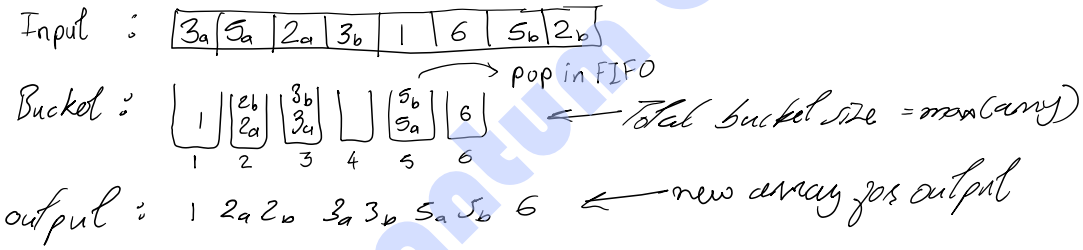
//Lecture 20f

2.4.5) Counting sort and radix Sort :

Assumptions :

- Keys are integers
- Maximum value of any key is <= k

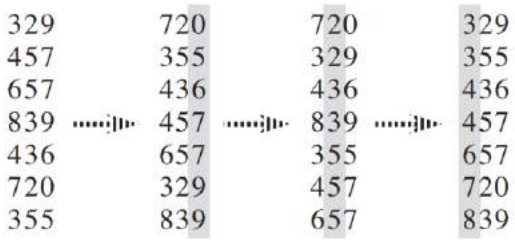
1) **Counting sort** : It takes value and push it into bucket of that index then it pop element from left to right from bucket and insert into new array.



Time complexity : $O(n+k)$

Space complexity : $O(n)$

2) **Radix sort** : We perform counting sort on the least-significant digit first, then perform counting sort on the next least-significant, and so on…



How many iterations are there ? – d

How long does each iteration take ? – O(n+k) but if we are taking only integer then k = 9 so O(n) only.

Total running time : O(nd)

Q : How good is O(nd) ? – O(nd) isn't so great if we are sorting n integers in base 10, each of which is in range {1, 2, 3, 4, …, nt}

Now, how many iterations are needed ? – d = $\log_{10} n^t$

How long does each iteration take ? – O(n) as previous one

What is the total running time ? - $O(nd) = O(nlog_{10}n^t) = O(n * t * log_{10}n)$

Running time of radix sort : $\boldsymbol{O(nt \times \log_k n)}$

3. Graph in algorithm

//Lecture 21a

Notations :

Graph can be represented as G = (V, E) where V = nodes (or vertices) and E = edges.

Undirected-graph connectivity :

An undirected graph is **connected** if for all pairs of vertices u, v there exists a path from u to v.

A undirected graph is complete, aka fully connected if for all pairs of vertices u, v there exists an edge from u to v.

Directed graph connectivity :

A directed graph is **strongly connected** if there is a path from every vertex to every other vertex

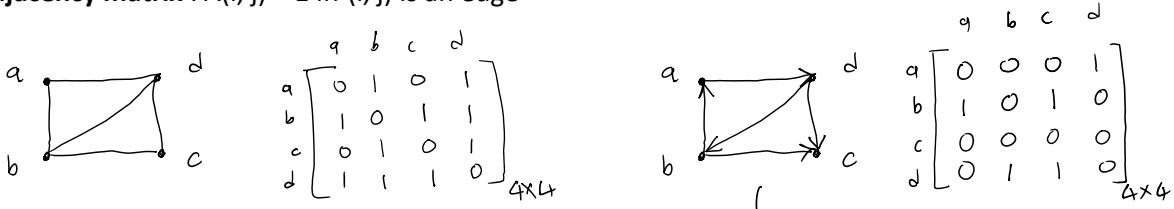
A directed graph is **weakly connected** if there is a path from every vertex to every other vertex ignoring direction of edges.

NOTE : if connected graph is mentioned without mentioning directed or undirected then consider undirected because concept of connected is only exists in undirected for directed we have strongly connected and so on concept.

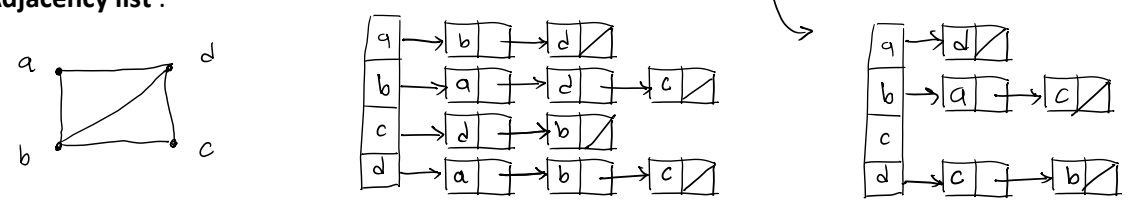
3.1) Introduction to graphs :

Storing graphs : Need to store both the sets of nodes V and set of edges E.

Adjacency matrix : A(i, j) = 1 iff (i, j) is an edge



Adjacency list :



//Lecture 21b

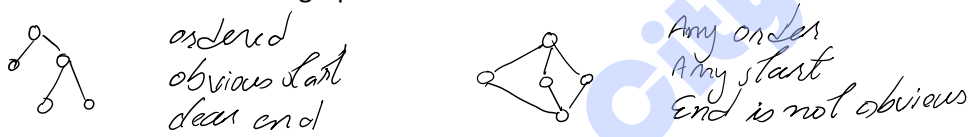
In adjacency list you may have observed that number of external nodes is equal to 2E in case of undirected graph (because each list represents degree of that node). And In case of directed graph, external node = E = sum of out-degree of all vertex.

	Adjacency List	**Adjacency Matrix**
Space	$\theta(V+E)$	$\theta(V^2)$
Test if u → v ∈ E	$\theta(1+\deg(u))$	$\theta(1)$
List v's neighbors	$\theta(1+\deg(v))$	$\theta(V)$
List all edges	$\theta(V+E)$	$\theta(V^2)$
Insert edge uv	$\theta(1)$	$\theta(1)$
Delete edge uv	$\theta(1+\deg(u)+\deg(v))$	$\theta(1)$

Algorithm runtime : Graph algorithm runtimes depend on |V| and |E|. and relation between V and E depend upon density of graph. Sometimes we have complete graph which *dense graph* in that case $|E| \approx |V|^2$. If graph is *spare*, $|E| \approx |V|$.

Graph traversal and search method :

We've seen traversal before…. But graph traversal is different :



Commonly used search methods : breadth-first search and depth-first search.

3.2) Searching algorithm :

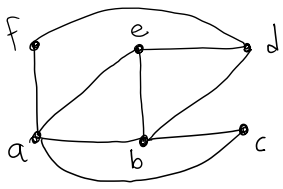
Brute force method is to select one node put it into bag then extract some node from bag if it is unmarked then mark them and put neighbors of that extracted marked node into bag. Repeat this process till all the nodes are marked.

```
Whatever-first-search(G, s) :
    put s in the bag
    while bag is not empty
        Extract v from bag
            if v is unmarked
                mark v
                for each edge (v, w) :
                    put w into the bag
```

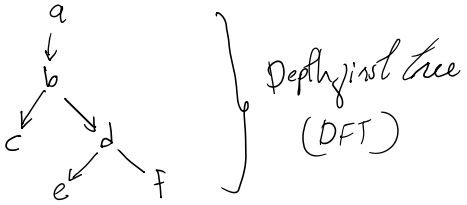
Stack then DFS
queue then BFS

//Lecture 21c

3.2.1) Depth-first search :



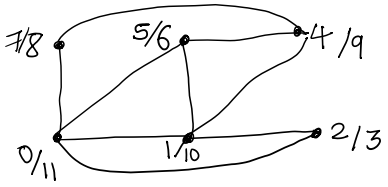
DFS (a) ;



Thumb rule is if there are no new nodes during traversal to visit then backtrack.

Suppose we define start time or discover time (time at which we visit vertex for the first time) and finish time (time at which we visit vertex last time)

Algorithms



Start time / Finish time

//Lecture 21d

1) Implementing DFS :

```
DFS_Visit(u) :
    visited[u] = true
    for each v adjacent to u :
        if not visited[v] : DFS_Visit(v)
```

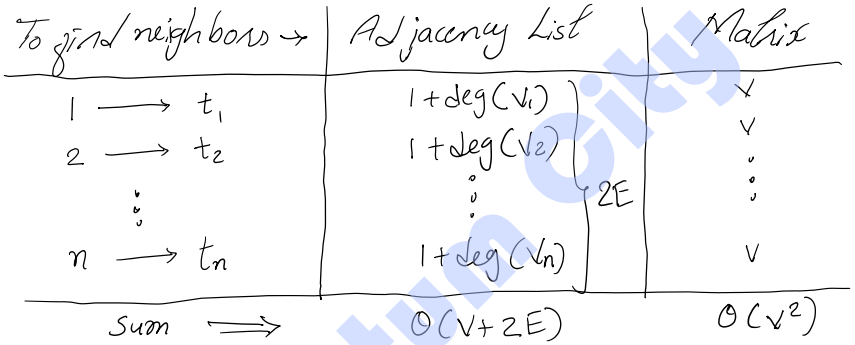
d[u] = time++;

time = time + 1 (& assign time to node)

Above function will mark nodes and visits its neighbors and mark them as well.

```
DFS(G) :
    for each vertex u in vertices
        visited[u] = false
    for each u in vertices
        if(visited[u] == false) DFS_Visit(u)
```

time = 0



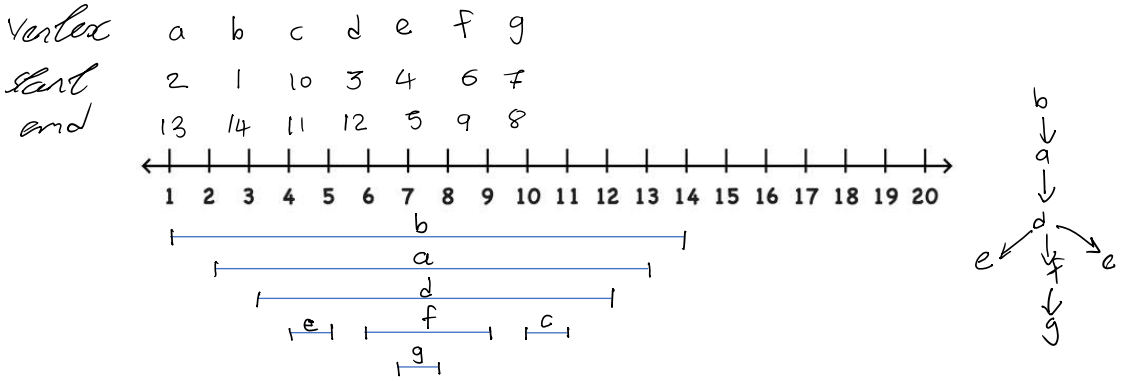
//Lecture 22a

2) DFS parentheses theorem :

In any depth first search of a directed or undirected graph G = (V, E), for any two vertices u and v, exactly one of the following three conditions holds :

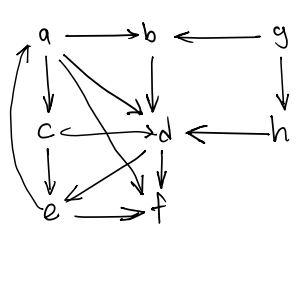
- The intervals [u.d, u.f] and [v.d, v.f] are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest.
- The interval [u.d, u.f] is contained entirely within the interval [v.d, v.f], and u is a descendent of v in a depth-first tree, or
- The interval [v.d, v.f] is contained entirely within the interval [u.d, u.f] and v is a descendent of u in a depth-first tree.

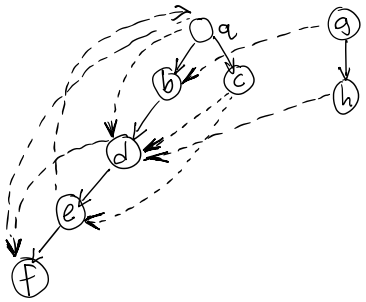
Q : Given the following question on dfs of directed graph. Reconstruct the DFS tree.



//Lecture 22b

3) DFS edge classification :





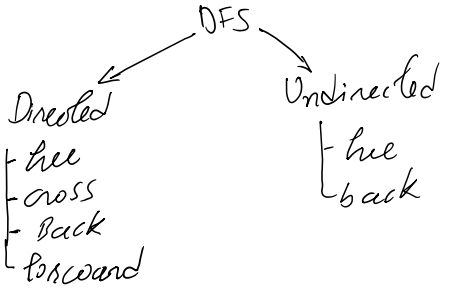
Tree edge : (a,b) (a,c), (b,d) (d,e) (e,f)
Forward edge : (a,d) (a,f) (d,f)
Backward edge : (e,a)
cross edge : (g,h) (c,d) (c,e) (h,d)

Tree edge : edge in DFS forest

Forward edge : an edge which is not a tree edge and goes from ancestor to descendent

Back edge : Not a tree edge and goes from descendent to ancestor.

Cross edge : not a tree edge and neither ancestor nor descendent.



So no cross edge in Undirected graph because it will become subtree as direction of edges is not known. & similarly no forward edge.

//Lecture 23a

A graph has cycle if and only if there is a back edge

//Lecture 23c

4) Applications of depth-first search :

Undirected graphs : Connected components, articulation points, bridges, biconnected components

Directed graphs : Cyclic/acyclic graphs, topological sort, strongly connected components

Topological sort :

- First select node with 0 indegree.
- If not present then topological ordering is not possible.
- If present then, remove it and find next 0 indegree vertex or node. And repeat process.

Finding topological sorting using DFS : We first find vertex with indegree zero then for all neighbor u decrease indegree by 1.

//Lecture 23d

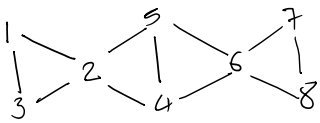
Cut vertex or Articulation point :

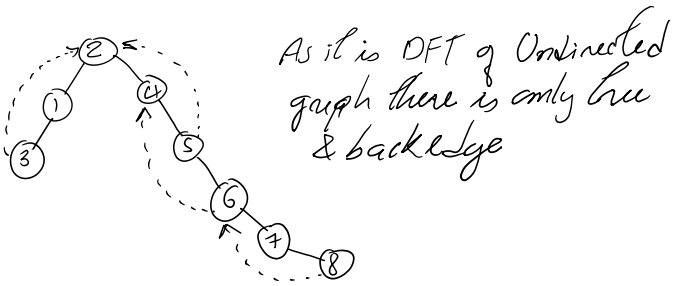
An articulation point of G is a vertex whose removal disconnects G.

Q : How to find articulation points through DFS-based approach ? – we can have several cases based on this …

Case 1 : When can root of DFT be AP ? – only when it has more than one children if it has one child then deletion of that root can't be AP.

Case 2 : when can non-root of DFT be AP ? – if and only if none of descendent are directly connected to ancestor. For example,





But what if 6 has two children and one of its children is connected to 6's ancestor in that case also we have 6 as AP but according to our previous sentence 6 can't be AP so we have to modify above case 2 statement.

A non-root of DFT is AP if and only if there exists one subtree of node such that none of it's descendent are directly connected to ancestor.

Now, talking about implementation so we just note down start time of each node in one array. and another array to maintain lowest start time node connected to a particular node (this will keep tract of back edges).

//Lecture 24a

3.2.2) Breath first search :

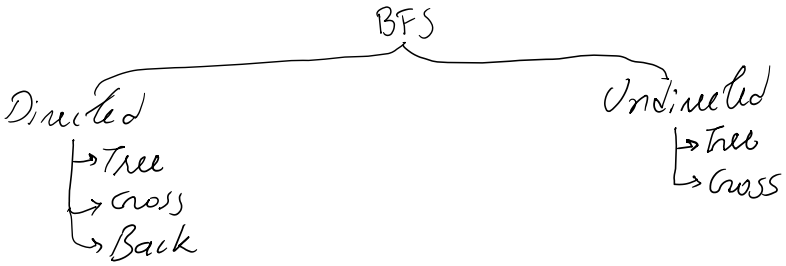
At a time, we will visit neighbor of node separated by one edge.


```
BFS(G, v) :
    //Initialise visited of all nodes to false
    visited[v] = false
    enqueue(Q, v)
    while(Queue is not empty){
        u = dequeue(Q);
        for each adjacent x to u{
            if(visited[x] == false){
                visited[x] = true
                enqueue(Q, x);
            }
        }
    }
```

Time complexity : $O(V+E)$ using adj. list & $O(V^2)$ using Adj. Matrix

//Lecture 24b

1) BFS edge classification :



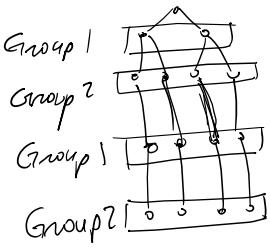
//Lecture 24c

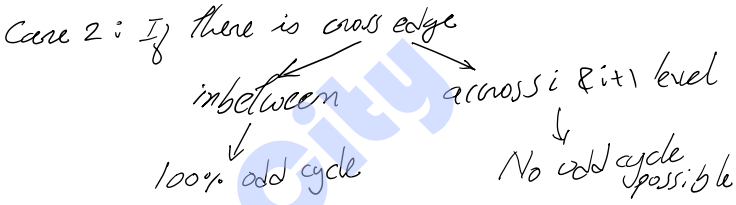
2) Application of BFS :

Bipartite graph : A graph is bipartite if and only if there is not odd length cycle.

Now, we know that in BFS tree only tree and cross edges exists so we have two cases:

Case 1 : no cross edge then definitely graph is bipartite. Because we can always combine alternate level as one group.





Talking about implementation, so we check cross edges in between nodes in same level if yes then graph is not bipartite.

Finding components : Both BFS and DFS can be used to find number of components of graph for DFS we have already seen but for BFS we maintain array in which we assign number which corresponds to a particular component and then we will see if any element is having 0 number which implies separate component.

3) BFS for disconnected graph :

```
vector<bool> Visited(number_of_vertices, false);
    for (int u=0; u<number_of_vertices; u++)
        if (Visited[u] == false)
            BFS(u, adj, Visited);
```

TC = ~~O(V(V+E))~~
Only visiting unvisited node so
TC = O(V+E)

We simply maintain a global array of vertices which keep track of visited vertices.

So, <u>we can use both BFS and DFS for finding path from any two vertex</u>.

4. Greedy algorithms

//Lecture 26a

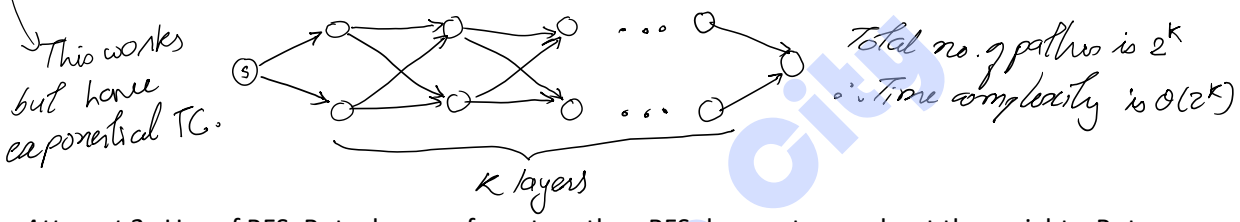
Suppose you went to shop and shopkeeper asks you for change of 30 rupees. Currently you have one 25 Rs., three 10 Rs. So, you follow greedy algorithm which says select best possible ways available at that moment. Your objective is to given change but using as minimal notes as possible. Therefore, you select 25 Rs. Which look best at the moment but now you do not have 5 Rs. Change so this method fails.

Therefore, greedy might not work sometimes.

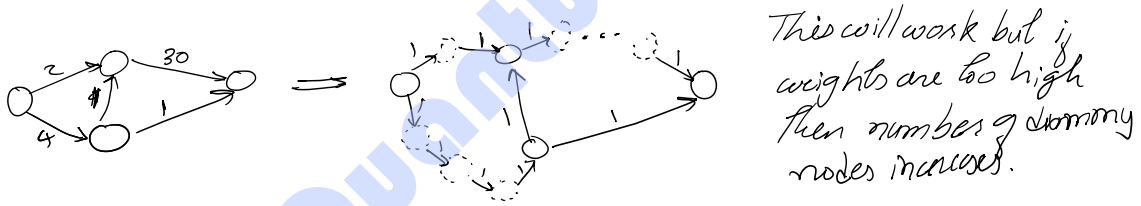
4.1) Shortest path problem :

Here we talk about single source shortest path problem in which we have to find shortest path in terms of weights from single source to final destination.

Attempt 1 : find shortest among all possible paths. This is brute force approach. But consider one scenario



Attempt 2 : Use of BFS. But when we form tree then BFS does not care about the weights. But we can always change the weight of edge into small dummy nodes whose edge weight is 1. Therefore, we can always change the graph into another graph. For example,

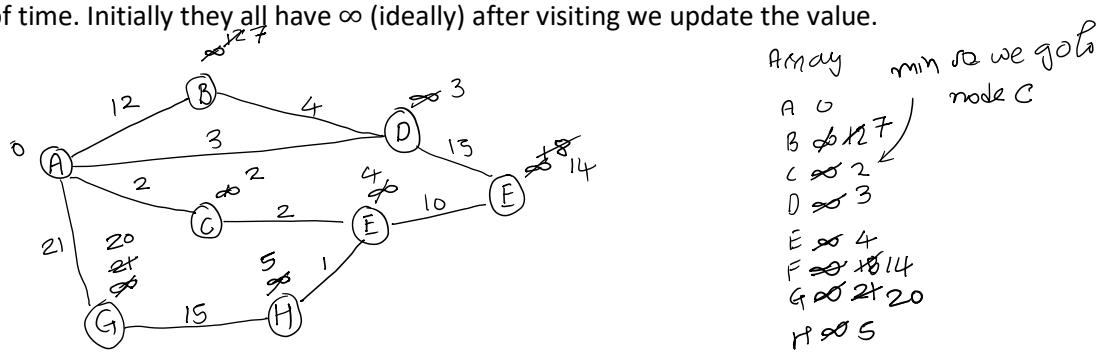


Final attempt : We go to a particular node from source and then try to find min distance from source to that node and we repeat the process till we have traverse every node. This is called Dijkstra algorithm.

Subpaths of shortest path is also shortest path

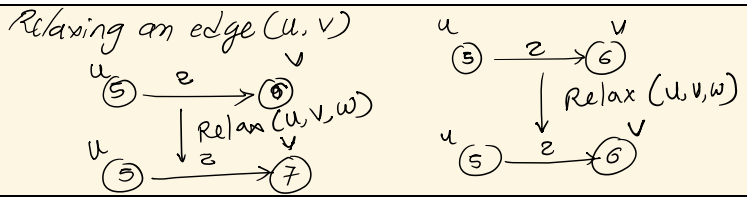
4.1.1) Dijkstra algorithm :

In Dijkstra algorithm each node have one value which denotes that shortest possible distance at that moment of time. Initially they all have ∞ (ideally) after visiting we update the value.



So, we pick min then go to that node then relax all outgoing edges and we repeat this process until we have visited all nodes. Relax means assign min distance to neighbor of selected node.

```
RELAX(u, v, w) :
    if (d[v]>d[u]+w){
        d[v] = d[v] + w
        parent[v] = u
    }
```

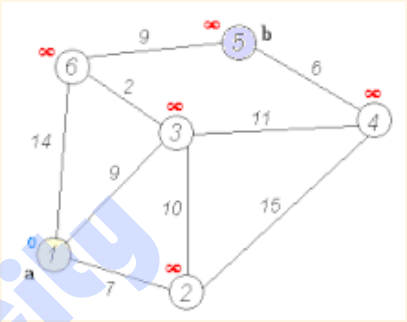


Dijkstra code :

```
Dijkstra(G, s):
1    key[v] = ∞ for all v in V
2    key[s] = 0
3    S = ∅
4    initialize priority queue Q to all vertices

5    while Q is not empty :
6        u = EXTRACT-MIN(Q)
7        S = S U {u}

8        for each adjacent v of u
9            RELAX(u, v, w)
```



//Lecture 27c

Time complexity :

One thing you may have noticed that we are relaxing any edge exactly once.

Algorithm maintains the min-priority queue Q by calling three priority-queue operations: INSERT (implicit in line 4), EXTRACT-MIN (line 6), and DECREASE-KEY (implicit in RELAX, which is called in line 9).

Total time complexity = $\sum_{u \in V} T(\boldsymbol{ExtractMin}) + \sum_{u \in V}(\sum_{v \in u.neighbors} T(\boldsymbol{Decrease\ key}))$

If Adjacency list + Heap : $V \log V + E \log V$

If Adjacency matrix + heap : $V \log V + ?$

In adj. matrix for neighbours you are traversing whole row ∴ 1's & 0's

$[\ldots 1\,0\,0\,1\,0\,0\,1\,0\,0 \ldots] \rightarrow deg^-(v) \times T(Decrease\ key)$ for 1's $+ O(1) \times (V - deg^-(v)) \rightarrow EXT(Dec.key) + V^2 - E$ for 0's

$\therefore TC = V \log V + E \log V + V^2$

In general, we can say that, Dijkstra time complexity :

Adjacency list : $V \times T(ExtractMin) + E \times T(DecreaseKey)$

Adjacency matrix : $V \times T(ExtractMin) + E \times T(DecreaseKey) + V^2$

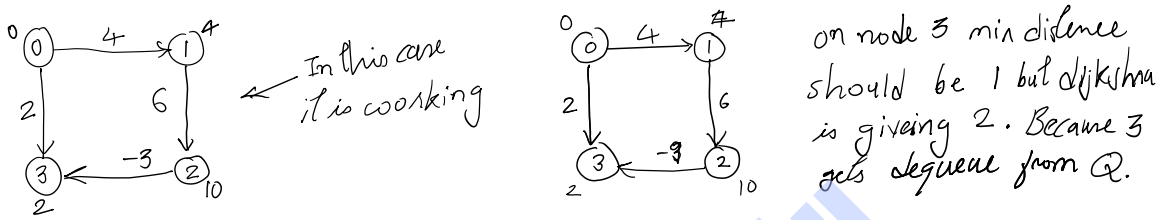
Graph representation	Priority Queue data structure	Time Complexity Dijkstra
Adjacency List	Heap	$(E + V) \log V$

Adjacency Matrix	Heap	$(E+V)\log V + V^2$
Adjacency List	Unsorted Array	$V^2 + E$
Adjacency Matrix	Unsorted Array	$V^2 + E$
Adjacency List	Sorted Array	$V + EV$
Adjacency Matrix	Sorted Array	$V + EV + V^2$
Adjacency List	Fibonacci Heap	$V \log V + E$
Adjacency Matrix	Fibonacci heap	$V \log V + E + V^2$

Note that for RELAX procedure if we take heap then it will take log n time for decrease-min operation and if it is implemented by array then O(1).

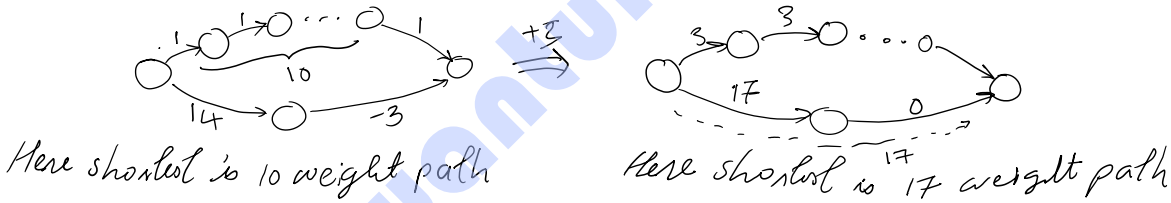
//Lecture 27a

Q : Does Dijkstra work on negative edges ? –

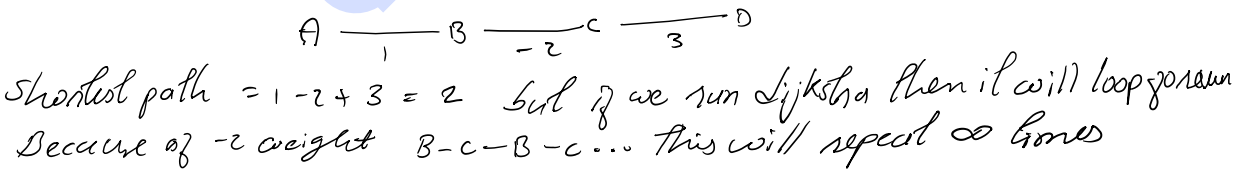


In first iteration Dijkstra will assume that 2 is min and there is no way that there will be a path from node 4 also because subpath of shortest path should be shortest path. So, it will ignore node 3 by removing it from queue. So how to overcome this issue.

Idea : *add some weight to make every weight as non-negative and apply Dijkstra ?* – this looks correct and do work for some graph but consider below counter,



You may encounter cycle in graph, but one thing to note that here from beginning we have assumed that graph is directed if graph is undirected then one negative edge can make cycle. For example,

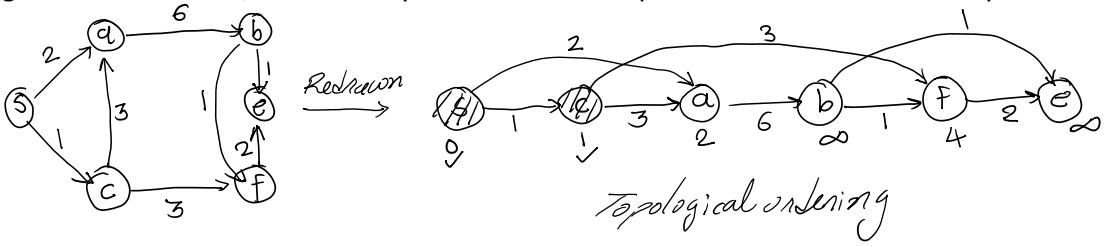


We want to build shortest path algorithm so that it can at least detect negative weight because if there is negative cycle then it may go to infinite loop.

//Lecture 28a

4.1.2) Shortest path in directed Acyclic graph :

You can apply Dijkstra algorithm but there is special thing about this graph that if negative weight edge is there than also, we can every time find shortest path because of absence of cycle.



One thing to note that when we come at node "a" then we are sure that it is shortest possible path result so there is no need to find traverse to a from any other vertex because there is no cycle.

```
Shortest path of DAG :
1: Set d[s] = 0 and d[v] = ∞ for al v ≠ s.
2: topologically sort the vertices of G
3: for each vertex u, taken in topologically sorted order do
4:      for every edge (u, v) do
5:          RELAX(u, v)
6:      end for
7: end for
```

$\theta(V+E)$

For every vertex we are doing $1+deg^{+}(V)$

$\therefore$ TC $= \theta(V+E)$

Therefore, we can say that Dijkstra : Pick min and relax outgoing edges. And DAG shortest path : pic in topological order and relax outgoing edges.

There is one algo Bellman ford : relax edges in any order (V-1) times.

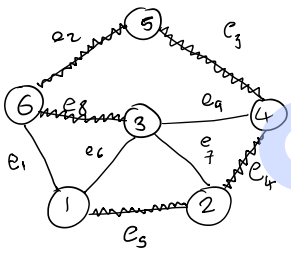
//Lecture 28b

4.1.3) Bellman ford algorithm :

If we relax in the order of shortest path (along with intermixed other relaxation) then we will get shortest path cost

This is true because there is always one shortest relaxation sequence exists. If we insert more relaxation in between this sequence then also final shortest path cost will be same. To find shortest path shortest relaxation sequence at least must be covered. This is called **path-relaxation property**.

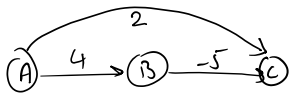
For example,



Here relaxation path is $e_5\ e_4\ e_3\ e_2\ e_8$ (in any order)
Now you can add any relaxation (edges) between this sequence it will still give shortest path cost.

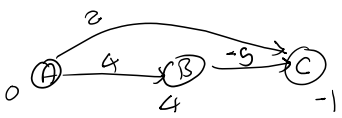
So, if you relax all element of relaxation path sequence then we are sure that all edges are relaxed and we have shortest path cost. Therefore, we first relax all the edges we know that 5 edges must be relaxed so we run relaxation for each edge V – 1 (here 4 times).

Example,



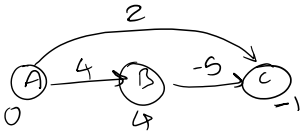
← In this Dijkstra fails

By Bellman ford, AB BC AC



Relaxation order

BC AC AB



Bellman ford works on the negative edge.

We can extend the relaxation idea to find negative cycle in graph. To do that we first relax all edge V -1 times then we know that we definitely have shortest path. And if do one more time then also it should not matter as all edges are already relaxed. But in case of negative cycle if we do one more time then it will be given different cost at some nodes so after V -1 relaxation, we do it one more time to just check negative cycle. If cost is same then no negative cycle present if different then negative cycle definitely exists.

```
Bellman_ford(G, s):
    d[v] = ∞ for all v in V
    d[s] = 0
    for i = 1 to V - 1
        for each edge (u, v) in E
            RELAX(u, v, w)
    for each edge (u, v) in E
        if(d[v]>d[u]+w)
            return false
    return true
```

O(VE)

worst case : when $E = O(V^2)$

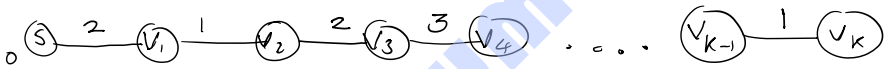
$TC : O(V^3)$

Dijkstra is fast but it does not work for negative weight edge.

If we run k times then we have shortest path to all vertices which are having at most k edges

//Lecture 28d

Q : Why bellman ford works ? – Suppose below is shortest path



Claim : our algorithm will discover shortest path not more than k steps.

In 1st iteration : we are sure that somewhere S-V1 this edge will relax and never gets changed because it is shortest.

In 2nd iteration : we are sure that somewhere in relaxation sequence v1-v2 edge will relax and its min cost will never get changed because it made up of two shortest paths.

Same up to k th iteration : we are sure that V_{k-1}-V_k edge will definitely relax. Thus, we have relaxed all the shortest path till kth iteration.

Bellman-ford : Early termination

```
Bellman_ford(G, s):
    d[v] = ∞ for all v in V
    d[s] = 0
    for i = 1 to V - 1
        relaxed = false
        for each edge (u, v) in E
            RELAX(u, v, w)
        if(relaxed == false) break

    for each edge (u, v) in E
        if(d[v]>d[u]+w)
            return false
```

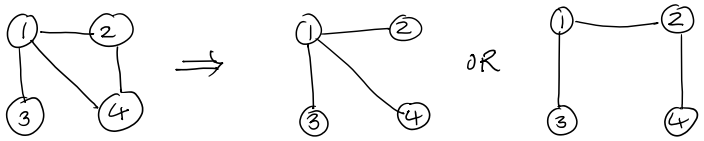
```
RELAX(u, v, w):
    if(d[v]>d[u]+w)
        d[v] = d[u] + w
        relaxed = true
```

```
    return true
```

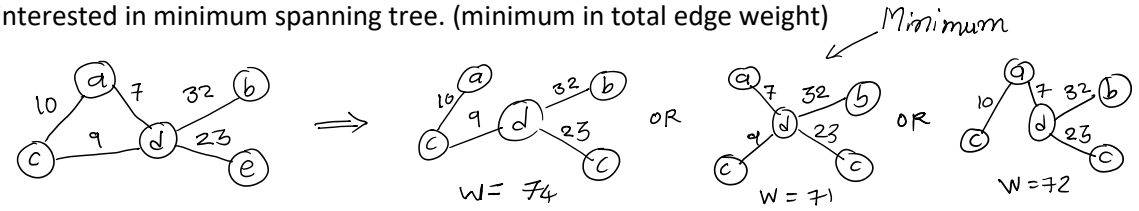
//Lecture 29a

4.2) Minimum Spanning trees :

A spanning tree of an undirected graph is a connected subgraph which contains all the vertices and has no cycles.



But we are not interested in spanning tree because there can be many spanning trees we are interested in minimum spanning tree. (minimum in total edge weight)



A MST of a graph connects all the vertices together while minimizing the number of edges used (and their weights).

Brute force way to find MSTs :

If there are n vertices in any graph then we have 2^n possible combination of building tree because we can either ignore edge from n vertices.

Spanning tree in complete graph (K_n) = n^{n-2}

Listing down all spanning trees and taking min is exponential in time.

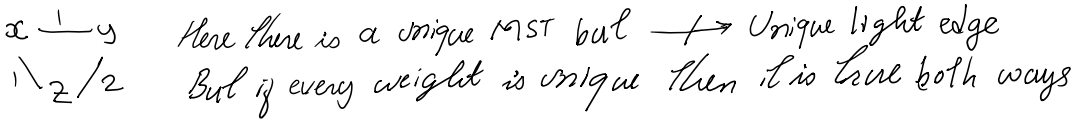
//Lecture 29b

Two properties of MSTs : If all edge weights are distinct

- ***Cut property*** : The smallest edge crossing any cut must be part of all MSTs
- ***Cycle property*** : The largest edge on any cycle is never in any MST

If we have graph with n vertices then total number of cuts will be 2^n but here we are counting {a} {b, c,…} and {b, c,…} {a} twice therefore, we divide it by 2 and also we are counting { } {a, b, c,…} this is not valid partition of due to cut. Therefore, finally we say **total cuts = $2^{n-1} - 1$**.

Proposition from cut property : A graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. But converse is not true as



Proposition from cycle property : An edge is not in any MST if and only if it is the unique heaviest edge in some cycle.

//Lecture 29c

4.2.1) Kruskal's Algorithm :

In these following sections we will answer how to find MST.

```
MST(G, w) :
    A = Ø
    while(A does not form a spanning tree)
        find an edge (u, v) that is safe //safe = it must be included in MST
(by some property or criteria)
        A = A U {(u, v)}
    return A
```

There are mainly two algorithm which guarantees safe edge after every iteration. Kruskal's algorithm, Prim's algorithm.

Kruskal's algorithm :

```
Kruskal(G) :
    sort edges in increasing order
    T = Ø
    for each edge e1 in the sorted order
        if e1 U T is not making a cycle :
            T = T U {e1}
    return T
```

using DFS

ElogE

TC = ElogE + E∝

suppose ∝

Time to check cycle

TC = ElogE + Ex O(V+E)

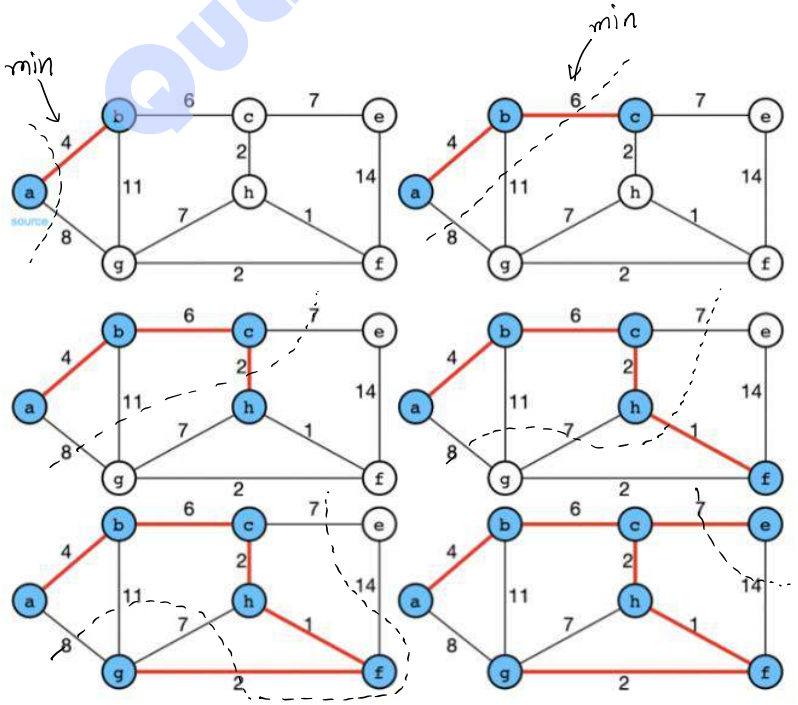
but there is a method called union-find data structure which calculate cycle in O(1) so Kruskal's algo's TC is O(E lg E).

Q : Can Kruskal's handle negative weights ? – will not impact anything because here we are not finding path or something, we are simply sorting edges.

//Lecture 30b

4.2.2) Prims algorithm :

Take any set of vertices, least weight edge in cut is always part of MST. For example,



Q : Do we need to detect cycles in prime's ? – No, as you can see, we are just forming cut set and detecting min along them without carrying about cycle.

Prim's algorithm is exactly same as Dijkstra algorithm but small difference is that in relax we were carrying weight from the source in Dijkstra and in prim's we will only carry weight of node to adjacent node.

```
Dijkstra(G, s):
1    key[v] = ∞ for all v in V
2    key[s] = 0
3    S = ∅
4    initialize priority queue Q to all
vertices

5    while Q is not empty :
6        u = EXTRACT-MIN(Q)
7        S = S U {u}

8        for each adjacent v of u
9            if v ∈ Q and d[v] > d[u] + w
10               d[v] = d[u] + w
11               //decrease Key
12               parent[v] = u
```

Relax (u, v, w):

```
Prims(G, s):
1    key[v] = ∞ for all v in V
2    key[s] = 0
3    S = ∅
4    initialize priority queue Q to all
vertices

5    while Q is not empty :
6        u = EXTRACT-MIN(Q)
7        S = S U {u}

8        for each adjacent v of u
9            if v ∈ Q and d[v] > w
10               d[v] = w
11               //decrease Key
12               parent[v] = u
```

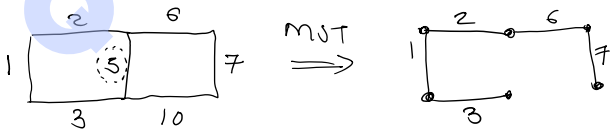
Main distinction :

Prim's : closest vertex to the tree

Dijkstra's : Closest vertex to the source … therefore prim's have same time complexity as Dijkstra's.

//Lecture 31a

Q : If G has a cycle and there is unique edge e which has the minimum weight on this cycle, then e must be part of every MST. – This looks true but it is false.



//Lecture 31c

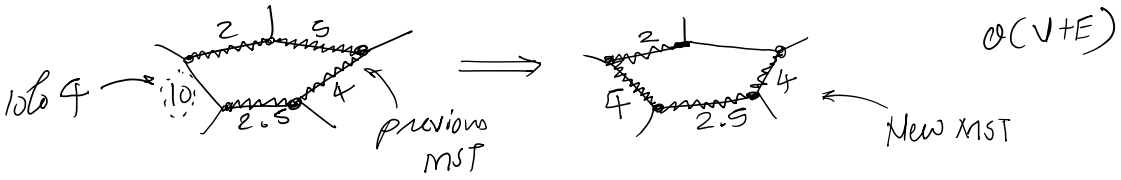
4.2.3) MST with modification of edge :

There are 4 cases :

1) **Edge is in MST and you decreasing value of edge** : This is trivial case where decreasing value of edge (we are talking about specific edge) will create same MST as previous.
2) **Edge is not in MST and you decreasing value of edge** :

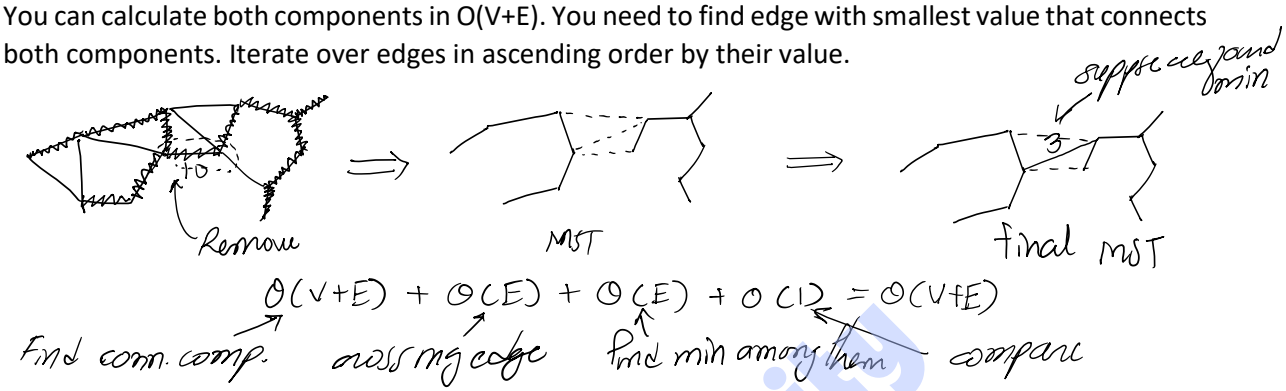
Add this edge to the MST. Now you've got exactly 1 cycle.

Based on cycle property in MST you need to find and remove edge with highest value that is on that cycle. You can do it using DFS or BFS. Complexity O(V+E).



3) Edge is in MST and you increasing its value of edge :

Remove this edge from MST. Now you have 2 disconnected component that should be connected. You can calculate both components in O(V+E). You need to find edge with smallest value that connects both components. Iterate over edges in ascending order by their value.



4) Edge is not in MST and you increasing its value of edge : This is also trivial as edge is not in MST and when you increase edge value then we will not include that edge in MST so MST will remain same as previous.

//Lecture 32a

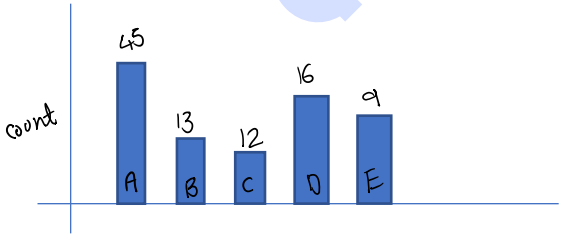
4.3) Some Greedy algorithm :

4.3.1) Huffman encoding :

Optimal codes problem :

Input : some distribution on characters (frequencies of characters)

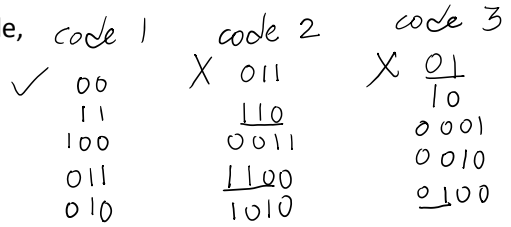
Output : A way to encode the characters as efficiently as possible.



We want to store this info in form of bits so if we use 3 bits to represent A then we need total of 45x3 = 135 bits. Similarly, for other but we want to minimize it AEAP.

Prefix free code : A code is a prefix-free code if any code is not a prefix of another code.

Example,

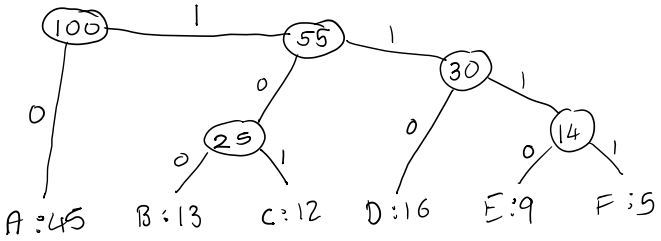


Solution : **Huffman coding !**

Greedily build subtrees by merging, starting with the 2 most infrequent letters. This will result in **Prefix free code and it is optimal code.**

Example, count of A in some file

A : 45 | B : 13 | C : 12 | D : 16 | E : 9 | F : 5



There are many variations of Huffman code. Instead of count of word we are may be given probability of occurrence of symbol.

```
HUFFMAN(C) :
    n = |C|
    Q = C
    for i = 1 to n - 1
        x = Extract-Min(Q)
        y = Extract-Min(Q)
        z.freq = x.freq + y.freq
        Insert(Q, z)
    return Extract-Min(Q)
```

//Lecture 33c

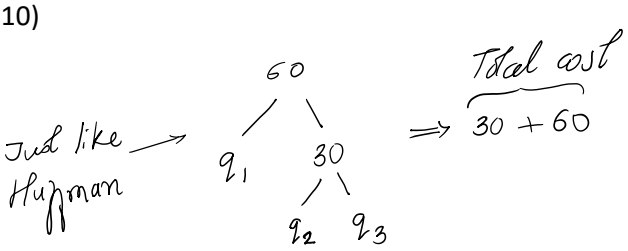
Optimal merge pattern :

Input : A set of files of different lengths.

Output : an optimal sequence of two-way merges to obtain a single file

Example, N = 3 and (q1, q2, q3) = (30, 20, 10)

1, 2, 3	50+60 = 110
1, 3, 2	40+60 = 100
2, 1, 3	50+60 = 110
2, 3, 1	30+60 = 90
3, 1, 2	40+60 = 100
3, 2, 1	30+60 = 90



//Lecture 34a

4.3.2) Interval scheduling problem :

Input : set of intervals

Output : Subset of nonoverlapping intervals of maximum size

Suppose we have a set S = {a1, a2, …, an} of n proposed activities that wish to use a resource, such as a lecture hall, which can serve only on activity at a time. Each activity ai has a start time si and a finish time fi, where $0 \leq s_i < f_i < \infty$. If selected, activity ai takes place during the half-open time interval $[s_i, f_i)$. Activities ai and aj are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. This is, ai

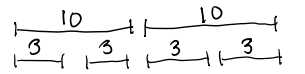
and aj are compatible if $s_i \geq f_j$ or $s_j \geq f_i$. In the activity selection problem, we wish to select a maximum-size subset of mutually compatible activities.

For example,

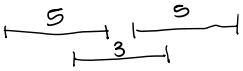
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities; another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

Idea 1 : let's pick intervals of min size.



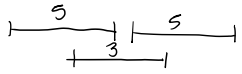
works for this case



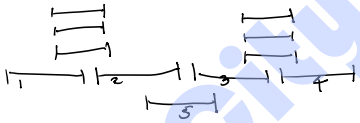
{5,5} is more

Not working as algo will take only 3

Idea 2 : Pick an interval that overlaps with minimum number of intervals.



working for this but

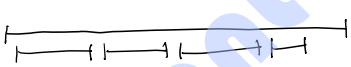


correct answer {1,2,3,4} but algo is giving {1,5,4}

Idea 3 : *Pick the interval with earliest finish time*.

Yes, it is working for last all counters. Surprisingly, this works for every case and this is optimal.

Q : Does following greedy algorithm works ? Choose the interval x that ends last, discard classes that conflict with x, and recurse – No.



← Not working

There is another optimal choice which says "choose the interval x that **stats** last, discard al classes that conflict with x, and recuse."

//Lecture 34e

Job scheduling with deadlines :

Job	**J1**	**J2**	**J3**	**J4**	**J5**	**J6**	**J7**
Deadline	9	1	5	7	4	10	5
Profit	15	2	18	1	25	20	8

Step 1 : sort all the jobs in decreasing order of profit. ← $n \log n$

Step 2 : Find maximum deadline and initialize array ← n

Step 3 : start from right side of the array and search for the slot i to find job which contain deadline <= i (linear search)

Step 4 : Repeat step 3 for all jobs

← n^2

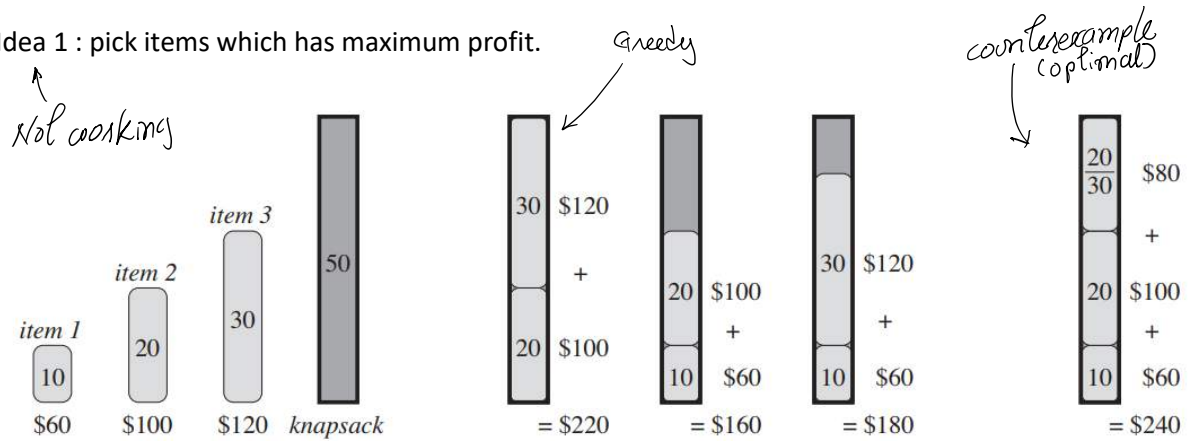
J2		J7	J5	J3		J4		J1	J6
0	1	2	3	4	5	6	7	8	9	10

//Lecture 35a

4.3.3) Fractional knapsack :

Goal : choose items such that total profit is maximized

Idea 1 : pick items which has maximum profit.



Idea 2 : balancing profit and weight i.e. profits/weight meaning per 1 unit weight how much profit you are getting.

So, first we will find profit/weight for all items and then we will choose item in decreasing order of p/w i.e. largest ratio first.

Surprisingly **time complexity is O(n).**

5. Dynamic Programming

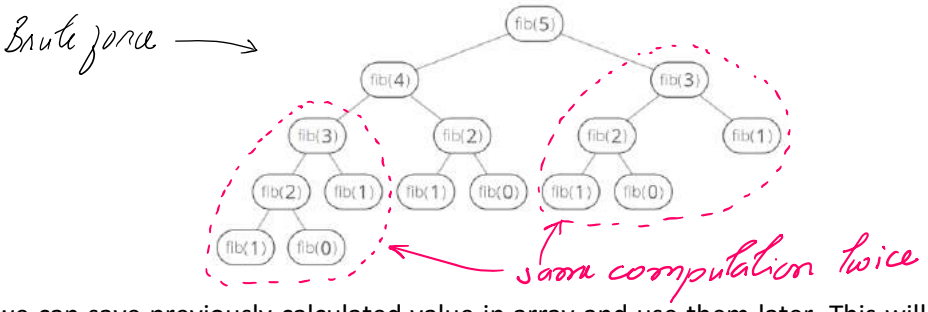
//Lecture 36a

Consider a program generating Fibonacci sequence,

```
int fibo(n) :
    if(n == 0 or n == 1) return 1;
    return fibo(n-1) + fibo(n-2);
```

$T(n) = T(n-1) + T(n-2) + 1$

$T(n) = O(2^n)$



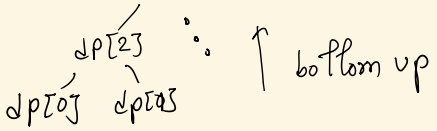
Instead we can save previously calculated value in array and use them later. This will save our time but at some array cost (space).

```
//dp is initialized all -1
int fibo(n) :
    if(d[n] != -1) return dp[n];
    if(n == 0 or n == 1) dp[n] = 1;
    else dp[n] = fibo(n-1) + fibo(n-2);
    return dp[n];
```

Top down method because we are going n to 1

bottom-up fashion :

```
int fibo(n) :
    dp[0] = dp[1] = 1;
    for(i = 2 to n)
        dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
```

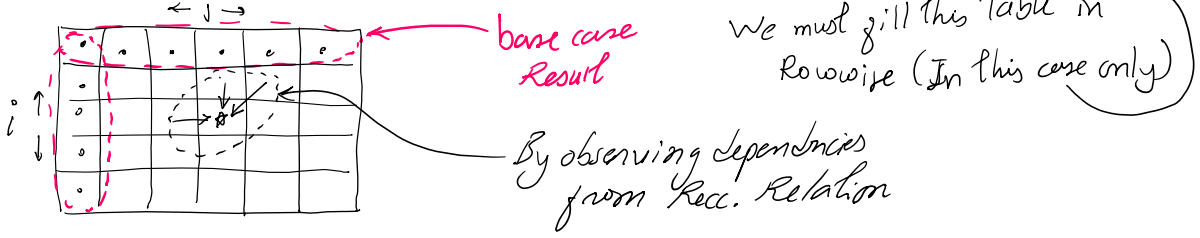


similarly, to implement dynamic programming problems we use top down and bottom up approach.

//Lecture 37b

After getting recurrence relation from problem statement we use tabular approach to find value of specifics asked in question.

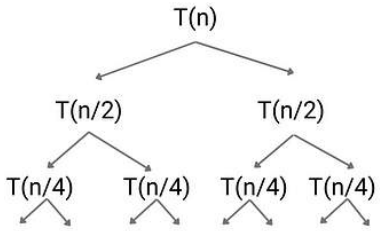
For example, $A(i,j) = \min(A(i,j-1), A(i-1,j-1), A(i-1,J+1))$



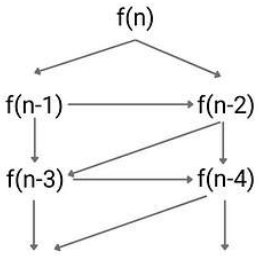
//Lecture 37c

Elements of dynamic programming :

- Solving smaller subproblems of same kind. – **Optimal substructure**
- Check if problem is being solved again. If so, keep a table and save time. – **Overlapping subproblems**



Independent sub-problems in merge sort recursion tree



Dependent sub-problems in nth Fibonacci recursion tree

//Lecture 38a

5.1) Longest common subsequence :

A sequence Z is a subsequence of X if Z can be obtained from X by dropping symbols.

BDFH is a subsequence of ABCDEFGH. We can say that every non-empty substring is subsequence.

Task : Given sequences X and Y, find the length of their Longest common subsequence Z.

Naïve algorithm :

- Enumerate all possible subsequences of X ← 2^n
- Find each subsequence of X check if is also a subsequence of Y – to do this we take two pointer i and j where

i → m = ABC

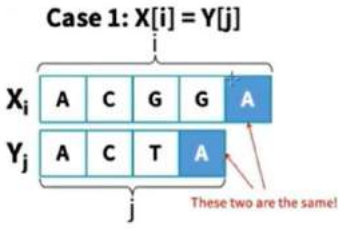
X = PQADBXC, j →

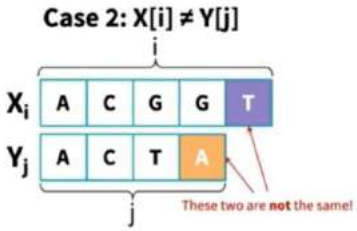
if symbols ≠ then increment j
if matching then increment both

- Keep track of the longest common subsequence found and its length.
- Running time : $O(n2^n)$

//Lecture 38b

5.1.1) Using dynamic programming :





Suppose, $LCS[i,j]$ = length of longest common subsequence of $X[1,\dots,i]$ and $Y[1,\dots,j]$.

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1,j-1], & if\ x_i = y_j \\ \max\begin{cases} LCS[i-1,j] \\ LCS[i,j-1] \end{cases}, & if\ x_i \neq y_j \\ 0, & if\ i = 0\ or\ j = 0 \end{cases}$$ ← base case

Using dynamic programming we can pre-calculate subproblems and store it in array (say dp[i][j]).

```
LCS[i,j] :
    if(dp[i][j] != NULL) return dp[i][j];
    if(i==0 or j==0)
        dp[i][j] = 0;
    else if(x[i] == y[j])
        dp[i-1][j-1] = LCS[i-1, j-1];
        dp[i][j] = 1 + dp[i-1][j-1]
    else  dp[i-1][j] = LCS(i-1,j);
        dp[i][j-1] = LCS(i,j-1);
        dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    return dp[i][j];
```

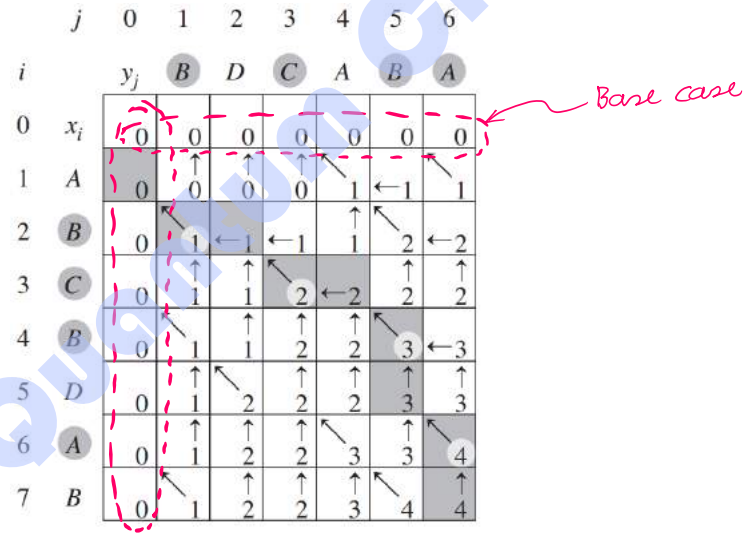
//Lecture 38c

Q : How many recursive calls are there ? – Since it forms binary tree and we are decreasing index by 1 so final time complexity would be $O(\max(2^m, 2^n))$.

Q : How many unique calls are there ? – m x n unique calls.

//Lecture 39a

Example,



//Lecture 40a

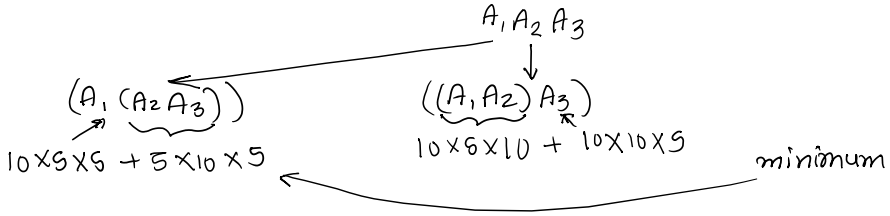
5.2) Matrix chain n matrices and Knapsack :

We know that when we multiply two matrix A_{pxq} and B_{qxr} then total multiplication required is p x q x r.

Now, consider 3 matrices A1, A2, A3 such that

A1 : 10 x 5, A2 : 5 x 10, A3 : 10 x 5.

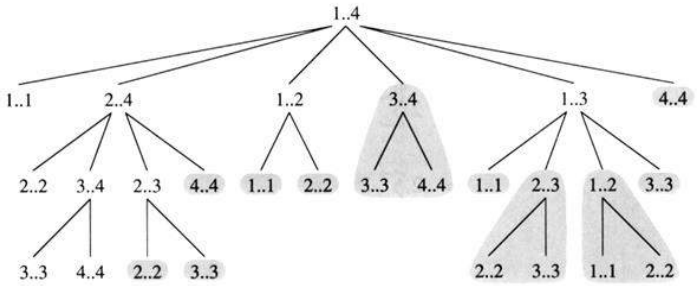
Q : What is the cost of different ordering of A1, A2 and A3 ? and what is least cost among them ? –



Problem statement :

Input : A sequence (chain) of (A1, A2, …, An) of matrices.

Output : Full parameterization (ordering) for the product A1 . A2 … An that minimizes the number of (scaler) multiplications.



Number of children are $\Omega(2^n)$. And thus, find min we take $\Omega(2^n)$ comparisons and that is why our worst-case time complexity is worse than any other algo.

Before solving the matrix-chain multiplication problem by dynamic programming, let us convince ourselves that exhaustively checking all possible parenthesizations does not yield an efficient algorithm.

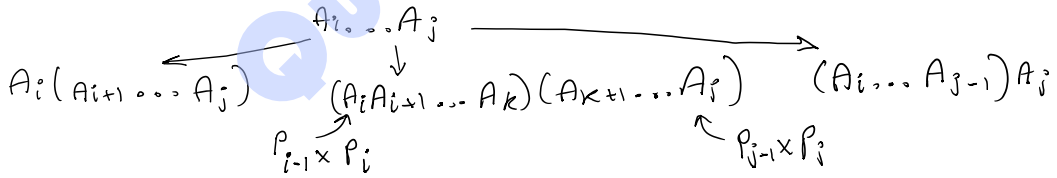
We see that no. of ways to multiply n matrices ≡ no. of ways to put n – 1 balanced parenthesis

Which in turn equal to Catalan number = $\frac{(2n)!}{(n+1)!n!}$.

//Lecture 40c

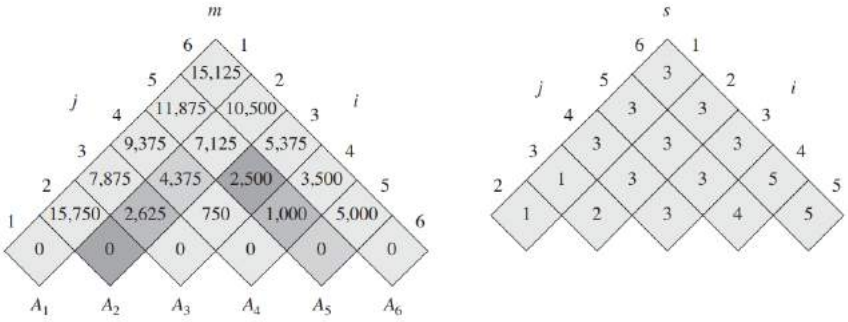
5.2.1) Using dynamic programming :

Recursive formulation : let m[i,j] represent minimum cost of multiplying matrices Ai … Aj. Where matrix A_k has dimension of P_{k-1} x P_k.



$$m[i,j] = \underbrace{min}_{k=i\ to\ j-1} \left(m[i,k] + m[k+1,j] + P_{i-1}.P_k.P_j\right) \quad \dots if\ i < j$$

$$m[i,j] = 0 \quad \dots if\ i = j$$



matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

Total time complexity of matrix chain multiplication : table size × time for one entry

$O(n^2)$ $\quad$ $O(n)$ $\Longrightarrow O(n^3)$

5.2.2) 0/1 knapsack problem :

NOTE : greedy only gives optimal solution in fractional knapsack as we have seen previously but 0/1 knapsack may or may not give optimal solution in 0/1 knapsack so we use dp.

Let $ks[j, x]$ = optimal value you can fit in a knapsack of capacity x with items 1 through j. Our aim is to find $ks[n, C]$.

$ks[<x_1, x_2, x_3>, 5]$ — Takes x_3 → $p_3 + ks[<x_1, x_2>, 5 - |x_3|]$; Don't take x_3 → $ks[<x_1, x_2>, 5]$

$$ks[i, C] = \begin{cases} 0, & \text{if } i = 0 \text{ or } C = 0 \\ ks[i-1, C], & \text{if } w_i > C \\ max\begin{cases} ks[i-1, C-w_i] + p_i & \leftarrow \text{Take} \\ ks[i-1, C] & \leftarrow \text{Don't Take} \end{cases} \end{cases}$$

//Lecture 41d

Q : How many total calls are possible ? – $O(2^n)$ because at worst all the weights are 1 unit and, in any situation, we can either take or ignore weights which gives 2 choice for all n weights.

Q : How many unique calls are possible ? – nC.

Not polynomial (it's pseudo-polyno...)

Time complexity : size of table × time to fill one entry = $nC \times O(1) = O(nC)$.

//Lecture 41e

Q : What will be the maximum profit obtained using 0-1 knapsack with capacity of 5 ? –

C

Item number	profits	Weights
X1	2	3
X2	3	4
X3	4	5
X4	5	2

i \ C	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	2	2	2
2	0	0	0	2	3	3
3	0	0	0	2	3	4
4	0	0	5	5	5	7

Here, cell[3, 2] represents max profit using first 3 items with capacity 2. For example, cell[1,1] is 0 because with capacity 1 we cannot fill any item weights.

//Lecture 42a

5.3) More dynamic programming algorithm :

In this section we will study subset sum problem, coin change problem, Floyd Warshall algorithm, travelling salesman problem.

5.3.1) Subset sum problem :

Input : given a set of integers {6, 8, 9, 11, 13, 16, 18, 24}, and some integer K

Output : find a subset that has as large sum as possible, without exceeding K.

Let mss[i, W] defines maximum subset sum (not exceeding W) given 1, 2, …, i elements.

$$mss[i,W] = \begin{cases} 0, & if\ i = 0\ or\ W = 0 \\ mss[i-1,W], & if\ w_i > W \\ max\begin{cases} mss[i-1,W-w_i] + w_i \\ mss[i-1,W] \end{cases}, & if\ W > w_i \end{cases}$$

If our output were

Check whether a subset that has as sum equal to k. Then,

Let ss(i, S) denotes if there is a subset with i element that has sum equal to k. this will return true if there exists one and false otherwise.

$$ss[i,S] = \begin{cases} False, & if\ i = 0, S \neq 0 \\ True, & if\ S = 0 \\ ss[i-1,S], & if\ a_i > S \\ ss[i-1,S-a_i] \vee ss[i-1,S], & if\ S > a_i \end{cases}$$

//Lecture 42b

5.3.2) Coin change problem :

You are a shopkeeper and you need to return 31 rupees to customer.

You have notes of [25, 10, 5, 1] rupees. (here there can be multiple notes of 25, 10 ,5, 1)

What is minimum number of notes you give to customer ?

$$CC[i,S] = \begin{cases} 0, & if\ i = 0\ or\ S = 0 \\ CC[i-1,S], & if\ v_i > S \\ min\begin{cases} 1 + CC[i,S-v_i] \\ CC[i-1,S] \end{cases} & Otherwise \end{cases}$$

//Lecture 42c

Q : Do we have optimal substructure in divide and conquer ? – We know we have subproblems of same kinds or same structure so yes optimal substructure is there in DAC. But we do not have overlapping subproblems in DAC because two subproblems are independent.

Q : Do we have optimal substructure in greedy ? – Yes, because after selecting one element we decrease out search key area to n-1 but we do not have overlapping subproblems because we are doing one move at a time not multiple moves at a time unlike DP.

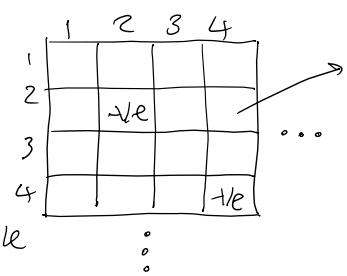
//Lecture 42d

5.3.3) Floyd Warshall Algorithm : All pair shortest path with TC $O(V^3)$

Input : G(V, E)

Output : shortest path between all pair of vertices.

Output of FWA (Floyd…) :



If negative cycle exist then one of the Diogonal is -ve

It works on negative weights also

//Lecture 42e

5.3.4) Travelling salesman :

We have to find min time required to reach from one place to another but for this there is exponential time algorithm present but no polynomial time algo.