→ Complexity

insertion: Log n

Deletion: Log n

SORTING ALGORITHMS

Quicksort Algorithm

→ Invented by C.A.R. Hoare

→ inplace sorting algorithm → It does not take extra place to do sorting

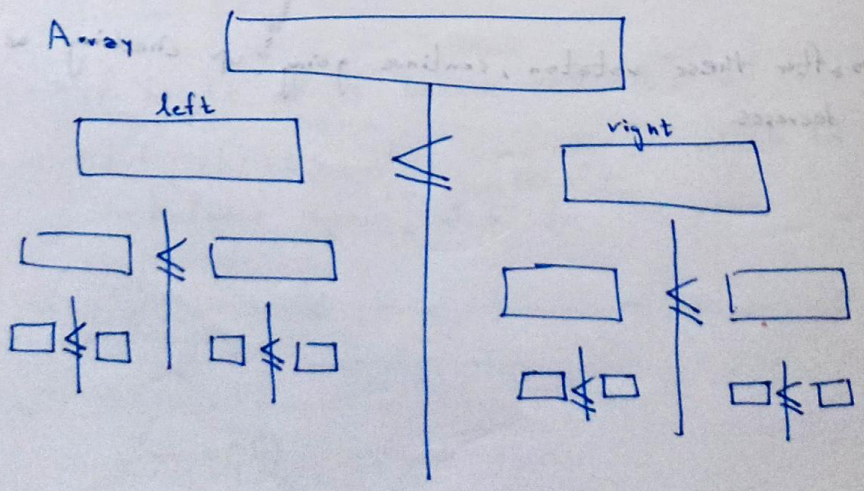
→ Uses comparisions to sort

→ Average Running Time: $O(n \log n)$

→ Worst Case Time: $O(n^2)$

→ Divide and Conquer Algorithm
- → Divide it into two parts
- → solve both part individually
- → Combine them to form a single answer.
- → Done on Recursion.

→ General Outlook



→ then merge

→ elements in right half ≥ elements in left half

→ Not merge sort ~~Case~~ → as in merge sort ~~the~~ this condition is not satisfied, but sorting is done during merge, not while dividing

→ Sorting happens while dividing

Divide Phase

→ Partition the array into two sub array, such that elements in the left half ≤ elements in right half

Conquer Phase

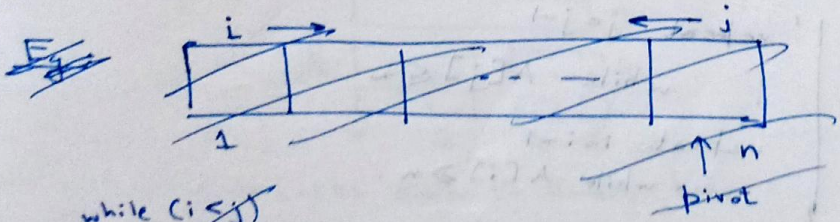
→ Recursively Sort two subarrays

Combine Phase

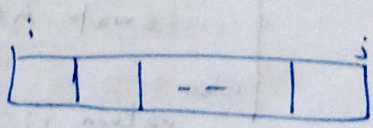
→ Trivial, already sorted by divide

Division

→ Partition done around a pivot element




```
while (i ≤ j)
{ if arr[i] > pivot
      i++
  if arr[j] < pivot
      j++
}
```



→ find i and j such that

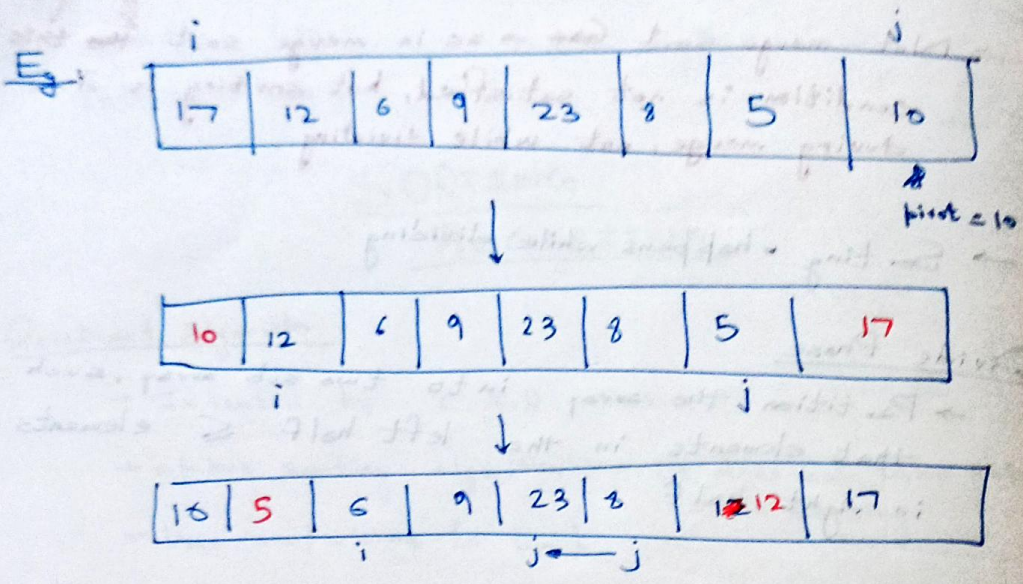
arr[i] > pivot

arr[j] < pivot

then swap.

Time Complexity

Eg:

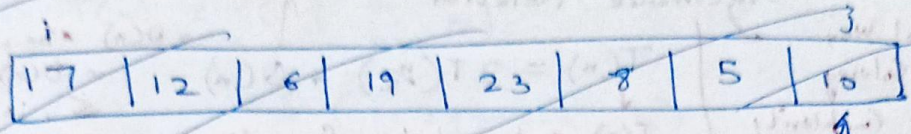


and continue till $i < j$

```
int partition (a A, p, r)        (array, lower bound, upper bound)
{
    x = A[r]  → pivot
    i = p-1 ;
    j = r+1 ;

    while True:
        repeat j = j-1
            while A[j] ≤ x
        repeat i = i+1
            while A[i] ≥ x.
        if i<j:
            swap A[i] & A[j]
        else
            return j;
}
```

Eg:



Quicksort Algo.

```
quick sort (arr, lb, ub)
{
    if (lb < ub):
        then  q = partition (arr, lb, ub)
        quick sort (arr, lb, q)
        quick sort (arr, q+1, ub)
}
```

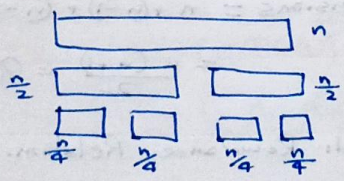
Time Complexity

~~→ Worst Time Complexity:~~

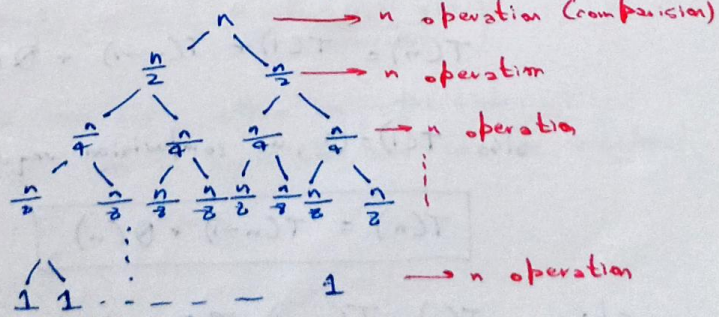
~~→ n time for a partition~~

→ Time Complexity depends on pivot

I) → If pivot median, sub array division exactly half



→ Looks similar to Binary tree



→ In each level, n comparison; no. of level = log n

∴ Time Complexity = n log n (if pivot median)

~~Recurrence Relat~~ in formal way for time complexity

→ Recurrance Relation

Formal Way of Solving Time Complexity

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$ ← Partition for n array

$\Theta(n)$ → because in $O(n)$ it will be $\leq$, in $\Omega(n)$, it will be $\geq$

$T(n)$ → time taken for sorting

Base Case: $T(1) = 0$

$\leq 2T(\frac{n}{2}) + O(n)$

$\geq 2T(\frac{n}{2}) + \Omega(n)$

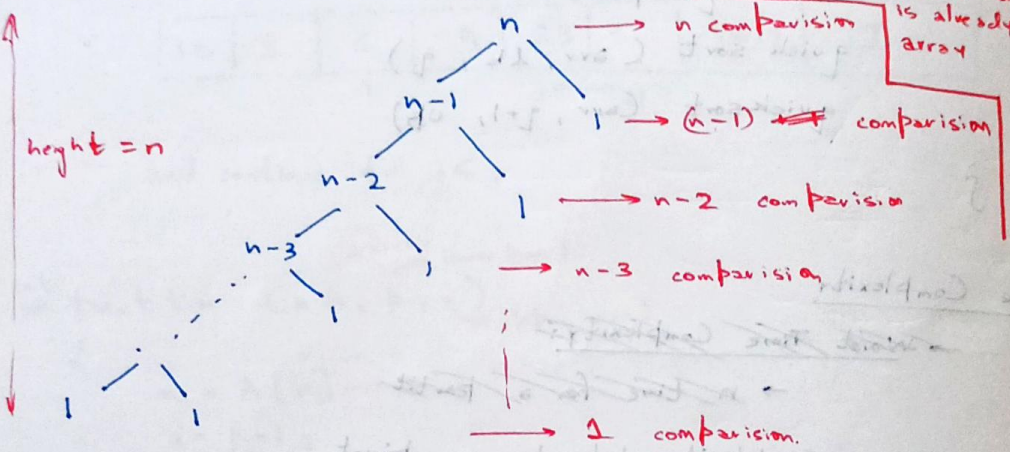
II) Worst Case: Partition of n array = n-1 & 1

→ Happens due to bad pivot : Pivot is max element

↳ combine with fact that pivot element = last element

↳ means worst case is already sorted array

a) Informal method



→ comparisions = $n + (n-1) + (n-2) + (n-3) + \ldots 1$

$= \frac{n(n+1)}{2} = O(n^2)$

b) Formal Method: Recurrance Relation.

$T(n)$: no. of comparision required to sort n value arr

$T(n) = T(1) + T(n-1) + \Theta(n)$

also $T(1) = 0$, no comparision required for single value arr

$$\boxed{T(n) = T(n-1) + \Theta(n)}$$

solving: $T(n) = T(n-1) + \Theta(n-1) + \Theta(n)$

$= T(n-2) + \Theta(n-2) + \Theta(n-\frac{1}{2}) + \Theta(n)$

$\vdots$

$= \Theta(n) + \Theta(n-1) + \Theta(n-2) + \ldots \Theta()$

$= \sum_{k=1}^{n} \Theta(k)$

$$\therefore T(n) = \Theta\left(\sum_{k=1}^{n} k\right)$$

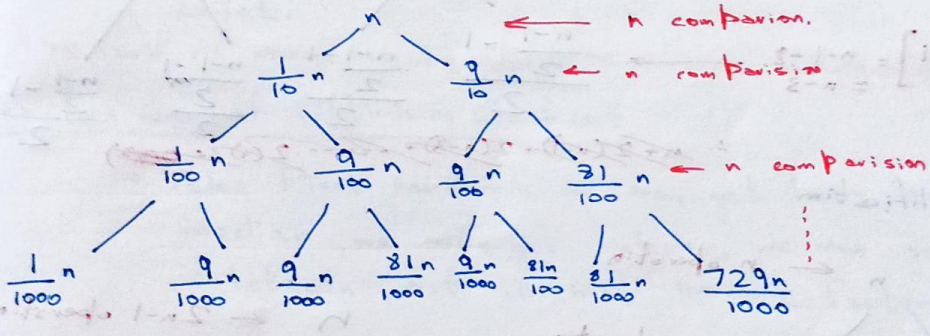
$$= \Theta(1+2+3+\cdots+n)$$

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

→ This worst case occurs when array is already sorted ∧, or reverse sorted. so this worst case comes very rarely.

→ There exist a theorem that states that best Time complexity for sorting Algorithm is $O(n \log n)$

III) 1:9 Partition.



→ to find height:

i) when ratio 1:1, we get height $= \log_2 n$

ii) when ratio 1:9, we get height $= \log_{\frac{10}{9}} n$

Recurrance Relation: $T(n) = T\left(\frac{1}{10}\right) + T\left(\frac{9}{10}\right) + \Theta(n)$

$\therefore$ Time Complexity $= O\left(n \log_{\frac{10}{9}} n\right)$

IV) m : ~~k~~ k Partition

Time Complexity $= O\left(n \log_{\left[\frac{m+m+k}{\max(m,k)}\right]} n\right)$

V) const m element on one side, other on the side.

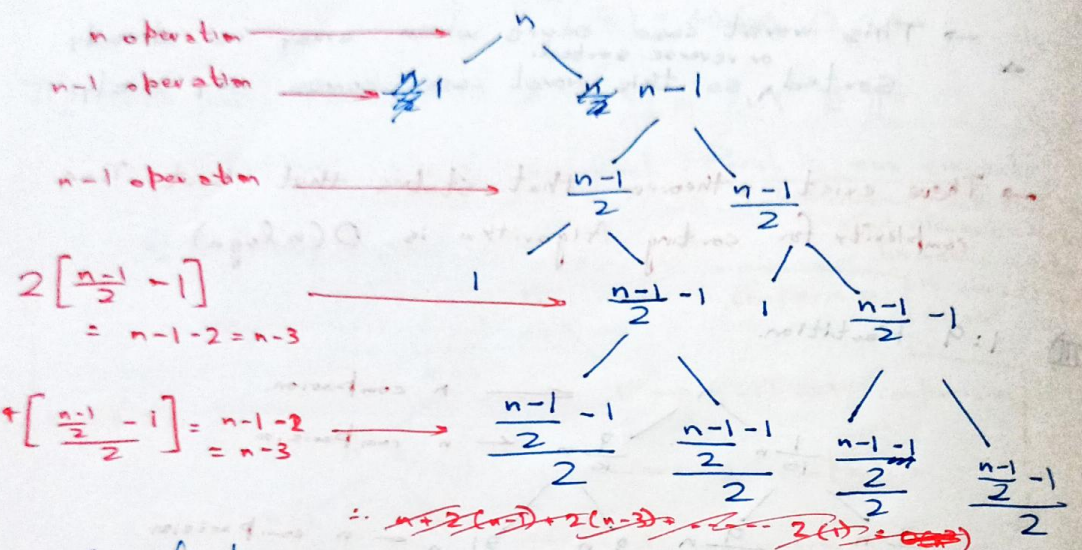
→ as $n \to \infty$, m is negligible and can be considered as worst case scenario

$\therefore O(n^2)$

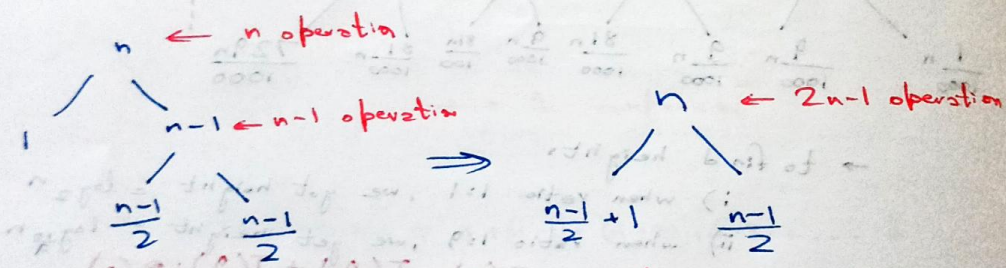
Lucky Case: median pivot : $\frac{n}{2}$: $\frac{n}{2}$

Unlucky Case: Sorted Case : 1 : n-1

VI) alternate Lucky and Unlucky Case



Simplification



→ so we can use recursion.

$$L(n) = 2U\left(\frac{n}{2}\right) + \Theta(n)$$

$$U(n) = L(n-1) + \Theta(n)$$

$$\therefore L(n) = 2\left[L\left[\frac{n}{2} - 1\right] + \Theta\left(\frac{n}{2}\right)\right] + \Theta(n)$$

$$= 2L\left(\frac{n}{2} - 1\right) + 2\Theta\left(\frac{n}{2}\right) + \Theta(n)$$

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

Philosophy

→ We can't always find median

→ But even when we get unlucky and lucky we get $\Theta(n \log n)$

→ So this Algo good.

Way of finding Median

→ We can also find median to make sure we get $O(n \log n)$

→ ~~but~~ we have to make this in $O(n)$, if it is $O(n^2)$, it has no use in this ~~Alg~~ Sorting

→ This is possible, i.e $O(n)$ way of median finding exist.

→ We can't do average, as avg may not be an array, and cause deviation problem

Another way of Optimising Quicksort

→ Take Pivot randomly, we may get lucky and get median, we may get unlucky, we may get pivot in between, but it may become faster.

→ Time Complexity = $\frac{\Sigma \text{ all possible random pivot Algo}}{\Sigma \text{ ~~many way~~ no. of way to selet a pivot}}$

Procedure (Random Pivot Partition)

```
Rvandomized_pivot_partition (arr, p, r)
{
    i= random (p, r)  <- gives random no. between p and r
    swap [A[r], A[i]];
    return partition(arr, p, r);   -> earlier partition fn.
}                                     -> where pivot is taken last element.

randomized_quicksort (arr, p, r)
{
    if (p<r):
        q = randomizer_pivot_partition (arr, p, r);
        randomized_quicksort (arr, p, q);
        randomized_quicksort (arr, q+1, r);
}
```

Randomized Time Complexity

$$T(n) = \sum_{j=1}^{n} \frac{[T(j) + T(n-j)]}{n} + Q(n)$$

→ ratio can be anything : 1 : n-1 , 2 : n-2, ... k : n-k ..... n-1 : 1

(here 1 : n-1 & n-1 : 1 is considered different)

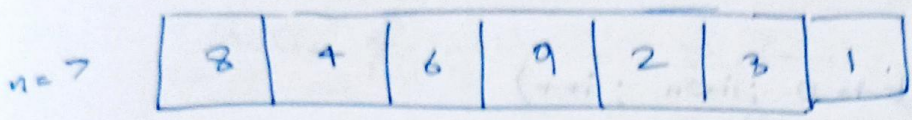
→ solving , we get $T(n) = n \log n$

→ This is why quick sort is prefered, it give $n \log n$ most of time.

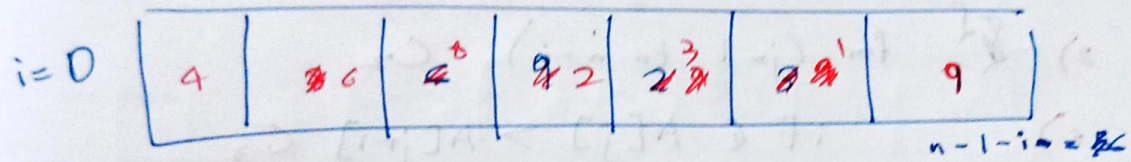
→ Read abot Selection, Insertion and Bubble Sort..

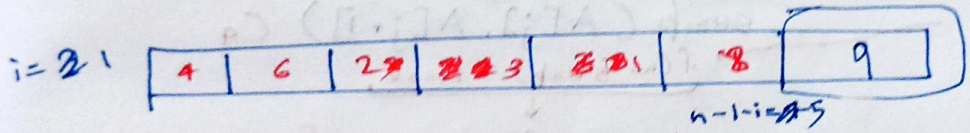
Bubble Sort

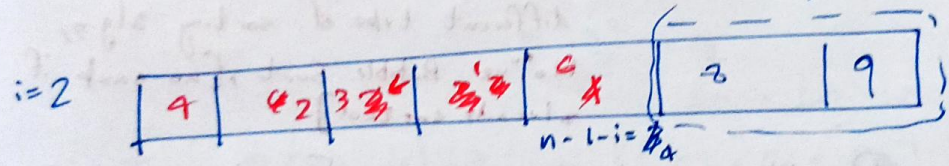
Eg: [Ascending]

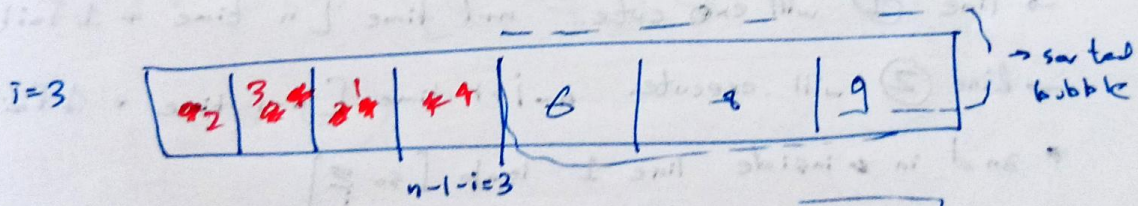


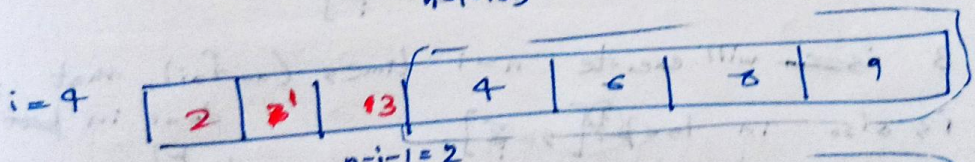
i=0
j=0 compare A[j] & A[j+1], & if A[j] bigger
swap A[j] & A[j+1]

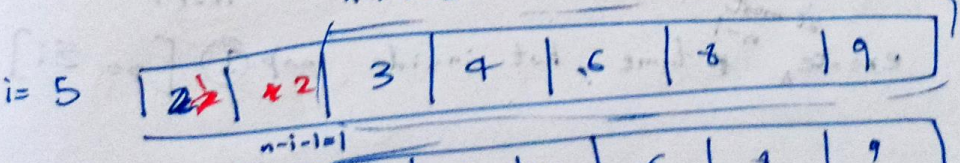


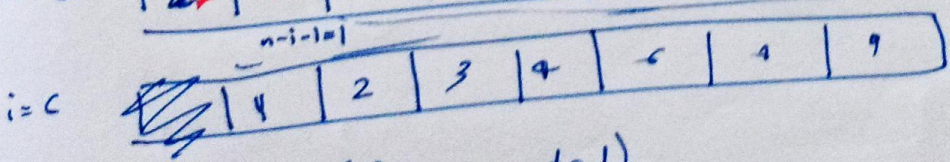












(fully sorted)

Rendadove [1 index array A)

bubblesort (A)

{

~~for (int i = 0 ; i<n ; i++)~~

~~{~~

~~for (int j = 0; j ≤ n-i-1 ; j++)~~

← time to process the line

1) for (i = 1 to n) C_1

2) ~~{~~ for (j = 1 to n - i) C_2

3) if (A[j] > A[j+1] C_3

4) swap (A[j], A[j+1]) C_4

} 3 → if (no swap) → break

different type of sorting algo, Naïve Bubble Sort, [no swap if already sorted]

Complexity

→ line ① will execute n+1 time [n time + 1 fail]

→ line ② will execute n-i+1 time [n-1 time + 1 fail]

* and in & inside line 1 loop [so $\sum i$]

→ line 3 ~~is~~ will execute n-i times (no fail), that done in ~~for~~ loop ①

& is also in loop ① [so $\sum i$]

→ lin 4 execute, at most n-i time, but inside loop ① [so $\sum i$]

∴ Total time taken

$$T = C_1(n+1) + C_2 \sum_{i=1}^{n} (n-i+1) + C_3 \sum_{i=1}^{n} (n-i) + \sum_{i=1}^{n} (n-i)$$

→ now $C_1[n+1]$ is ~~bounded~~, $\Theta(n)$ [avg big-Θ def^n] $\left[\alpha n < C_1[n+1] < \beta n\right]$ [α, β some thing]

$$\therefore T = \Theta(n) + [C_2 + C_3 + C_4] \sum_{i=1}^{n} n - i$$

↓ this also includes $C_2 \sum_{i=1}^{n} 1 = C_2 n$

→ now $\sum_{i=1}^{n} (n-i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i$

$$= n^2 - \frac{n(n+1)}{2}$$

$$= n^2 - \frac{n^2}{2} - \frac{n}{2}$$

$$= \frac{n^2}{2} - \frac{n}{2}$$

$$\therefore T = \Theta(n) + \left[\frac{n^2}{2} - \frac{n}{2}\right][C_2 + C_3 + C_4]$$

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

Rando

Insertion Sort

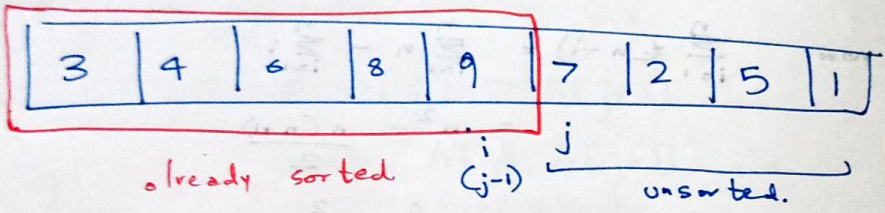
Algo

→ Start Empty Handed

→ Insert a card in the right position of the already sorted cards

→ Continue until all cards are inserted/sorted.

Eg: Consider a situation [particular value of i and j]



procedure

INPUT: A[1:n] – an array of integer

OUTPUT: a permutation of A, which is sorted

```
for (j=2 to n)
{
    key = A[j]
    i = j-1
    while (i>0 & A[i]>key)
    {   A[i+1] = A[i]
        i--
    }
    A[i+1] = key
}
```

C_1 → n times (n-1 go, 1 fail)

C_2 → n-1 time

C_3 → n-1 time

C_4 → inside loop $\sum_{j=2}^{n} (t_j - 1)$

C_5 → $\sum_{j=2}^{n} (t_j - 1)$

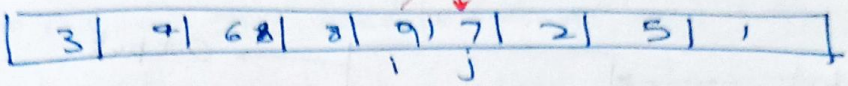
C_6 → $\sum_{j=2}^{n} (t_j - 1)$

C_7 → n-1

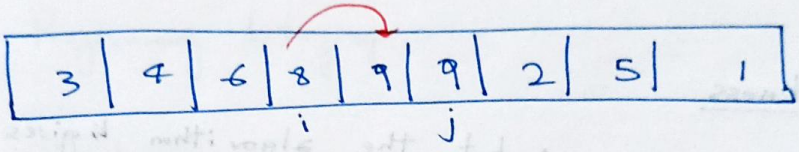
(t_j made as it can take various value due to decision control statement "A[i]>key")

dry run [consider previous ✱ diagram]

I)

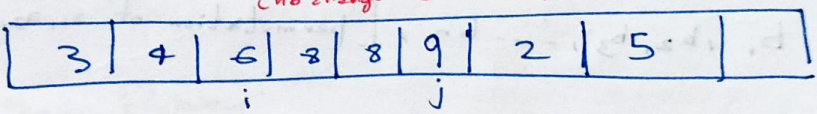


II) key = 7

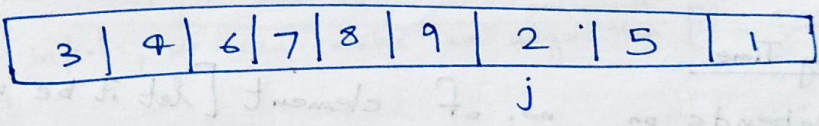


III) key = 7

(no change as 6 ✗ 7)



IV)



⋮

(continue)

Time Complexity

Total time $T = n\left[C_1 + C_2 + C_3 + C_7\right] + \sum_{j=2}^{n} t_j \left[C_4 + C_5 + C_6\right] - \left[C_2 + C_3 + C_5 + C_6 + C_7\right]$

→ Best case: Element already sorted ; $t_j = 1$, $T = f(n)$

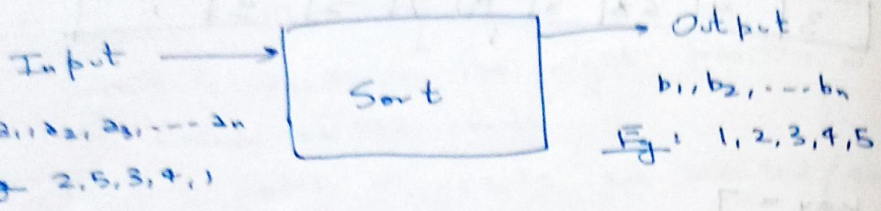
linear time

→ Worst case : Elements in reverse order, $t_j = j$, $T = f(n^2)$

quadratic time.

→ Average case : $t_j = \frac{j}{2}$, $T = f(n^2)$, quadratic

Sorting : Basics



→ **Correctness**

For any given input, the algorithm gives the output

$b_1, b_2, b_3, \ldots b_n$, [permutation of $a_1, a_2, \ldots a_n$]

such that $b_1 < b_2 < \ldots < b_n$

→ **Running Time**

- → Depends on no. of element [let it be 'n']
- → Depends on how partially sorted the input is
- → Also depends on Sorting Algorithm used.

Asymptotic Analysis

Algorithm

→ Outline, the essence of a computati
procedure, step-by-step instructions

Program

→ An implementation of an algorithm in some Programming Language.

Data Structure

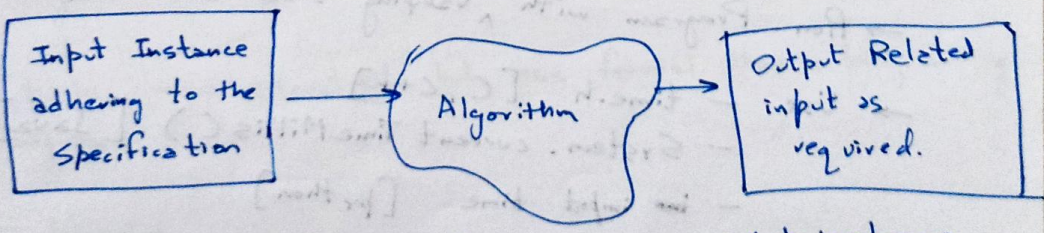
→ Organization of data needed to solve the problem

~~Algori~~ Algorithmic Problem

→ given Input specification and output as function of input, we can make an algorithm [function], such that

$$output = f(input)$$

→ ~~Input can be~~ Infinite no. of input can exist for a set of Input specificat, the function should give correct output, for all possible Inputs.



→ Algorithm describes actions on the input instance.

→ There can be infinitly many correct algorithms for the same algorithmic Problem

C is a Good Algorithm ?

→ A good Algorithm should be.

i) Efficient in
- Running Time.
- Space Used.

ii) give correct output to given input

→ ~~We can~~ Efficience can be made as a function of input size
- The number of bits in an input number
→ No. of data elements, etc.

Measuring of Running Time of An algorithm

Two Approaches
- Experimental Study
→ Formal / Theoretical Analysis

Experimental Study

→ Write Program implementing the Algorithm
→ Run Program with input with varying size and composition
→ Use
- time.h [C, C++]
- System.currentTimeMillis() [Java]
- import time [python]

to get ~~accurate~~ actual running time

→ Plot Result.

Limitations of Experimental Study

→ It is necessary to implement the algorithm, which may be difficult

→ Results may not be indicative of the running time on other inputs not included in experiment

→ In order to compare two algorithm, same hardware and software environments must be used.

Theoretical Analysis

→ General Methodology to analysing Algorithm

→ Uses high-level description of the algorithm, instead of it's implementation

→ Characterizes Running time as function of input size ('n')

→ Allows us to evaluate speed of Algorithm, irrespecti of Hardware/Software used.

→ Takes all possible inputs

Pseudocode

→ A mixture of natural language and high-level programming ~~language~~ concepts that describes the main idea behind a generic implementation of a data structure or algorithm

→ It is more structured but less formal than a programming language.

→ Expression

→ use standard mathematical symbols to describe numeric and boolean expression.

→ use '←' for assignment ["=" in C]

→ use '=' for equality checking [" == " in C]

→ Method declaration

<Algorithm name> (param 1, param 2)

→ Programming Construct

→ Decision

if ... else ...

→ While loop

while --- do ---

→ Repeat loop

repeat --- until --

→ For loop

for --- do ---

→ Array indexing: A[i], A[i,j]

→ Methods

→ calls: object method (argument)

→ return value.

Analysis of Algorithm

→ Primitive Operation

Low level operation independant of programming language

Eg: → Data Movement [Assignment]

→ Control [branch, subroutine call, return]

→ Arithmetic and logical operations

→ Indexing into array

Primitive operations

→ Basic computations performed by an Algorithm
→ Assumed to take constant amount of time.

Eg: Array Max

Operations

```
Algorithm ArrayMax(A, n)
{
    // Input: Array A with n integer
    // Output: Maximum element in A

    currentMax ← A[0]                          2 [assignment & index]

    for (i ← 1 to n-1)                         2n [n assignment, n loop check]
    {
        if (currentMax < A[i]                  2(n-1) [loop is n-1 time true, indexing, comparition]
            currentMax ← A[i]                  2(n-1)  [n-1 time loop, indexing assignment]
    }                                          ↳ supposing if is true every time
    return currentMax ; 1
}
```

→ Total Primitive operation: $6(n-1)$

→ let
a = Time taken by fastest Primitive operation
b = Time taken by slowest Primitive operation

$T(n)$ = Worst case Time of array Max

$\therefore a(6n-1) \leq T(n) \leq b(6n-1)$

i.e, $T(n)$ is bounded by two linear function.

Best / Worst / Average Case

→ For a specified size of input n, we investigate running time for different input instance.

→ We get a Best Case, a Worst Case, and Average Cases

→ We assume $T(n)$ as worst case, the algo can take less time, but it doesn't matter

Eg: $T(n) = cn^2 + k$, can take less steps, but we don't give a damn.

Why Worst Case?

→ It gives us an upper bound on running time,

→ We don't have to make any guess about running time.

→ For some Algorithm, Worst Case almost always happen

Eg: Seaching in Database.

→ Average case is mostly almost as bad as Worst Case

→ Average Case is very hard to find., since it involves probabilistic arguments and often requires some assumption about distribution of inputs that may be difficult to justify.

Why Average Case?

→ Some time Worst Case is very specific and very bad, so it's better to use Average case, algo. As worst case can be ~~m~~ using worst case will not bring justice to the Algo.

Eg: quick sort → worst case: Already Sorted (n^2) → duh (-__-)

→ Average Case: ~~(n log n)~~ n log n

Asymptotic Analysis

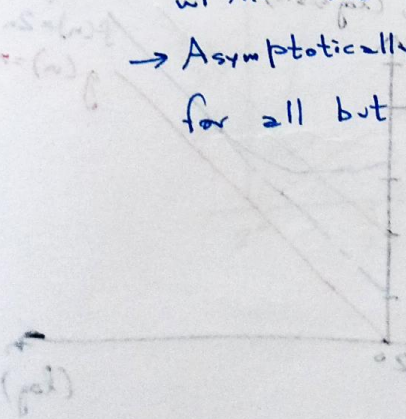
→ Goal

→ Simplify Analysis of Running Time by getting rid of "details" which may be affected by specific implementation and hardware

→ Capture the Essence

→ How the running time of an algorithm increases with the the size of input in the limit

→ Asymptotically more efficient algorithm are best for all but small inputs.



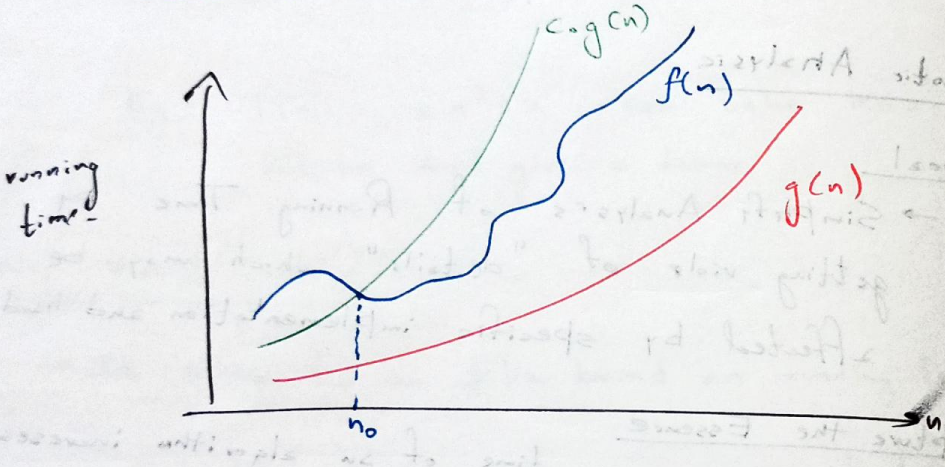
Big - Oh O(n)

→ Asymptotic Upper Bound

→ $f(n)$ is $O[g(n)]$, if there exist c & n_0, such that $f(n) \leq c\,g(n)$, for $n \geq n_0$.

→ $f(n)$ & $g(n)$ are function over non-negative numbers.

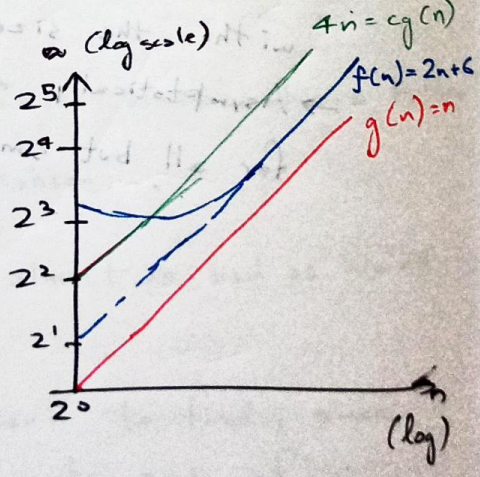
→ Used for worst case analysis



Eg: $f(n) = 2n + 6$

$\therefore 2n+6$ is $O(n)$

with $n_0 = 3$, $c = 4$



Eg: $2n+10$

~~$2n+10 \leq$~~

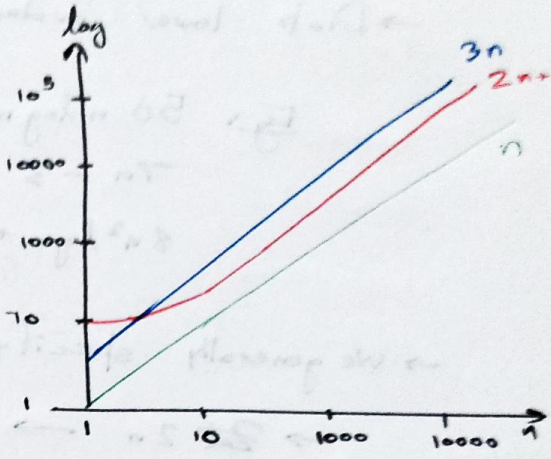
→ Checking $O(n)$

$2n+10 \leq cn$

$(c-2)n \geq 10$

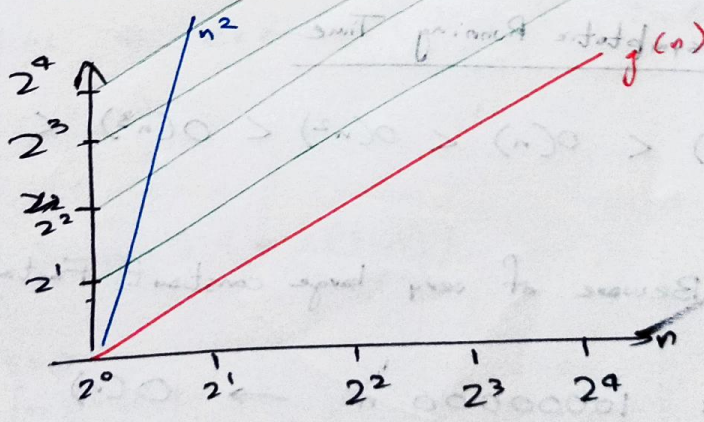
$n \geq \frac{10}{c-2}$

Pick $c=3, n_0=10$



Eg: n^2 is not $O(n)$, as there is no c & n_0, such that $n^2 \leq cn$, $n \geq n_0$

→ ~~There~~ no matter how large c is, there is an n big enough, such that $n^2 > cn$



Simple Rule

→ Drop lower order terms and constants

Eg: $50 n \log n \rightarrow O(n \log n)$

$7n - 3 \rightarrow O(n)$

$8n^2 \log n + 5n^2 + n \rightarrow O(n^2 \log n)$

→ We generally specify tightest bound possible

→ ~~Ze~~ $2n \longrightarrow O(n^2)$ & $O(n)$

but we say $O(n)$

→ Use simplest Expression

Eg: $3n + 5$ → ~~$O(3n)$~~ $O(n)$

Comparing Asymptotic Running Time

$O(\lg n) < O(n) < O(n^2) < O(n^3) < O(2^n)$

Caution! Beware of very large constant Factor.

Eg: $10000000\, n \rightarrow O(n)$

but performs poorly than ~~$O(n^2)$~~

$2n^2 \rightarrow O(n^2)$

Eg 1

Algorithm prefixAverages1 (X):

```
{
1)   for (i=0 to n-1)
     {
2)     a = 0
3)     for (j=0 to n-1)
       {
4)       a = a + X[j]
       }
5)     A[i] = a / (i+1)
     }
6)   return Array A
}
```

Analysis

→ line 2-5 execute n times

→ line 4 execute i time for i = 0, 1, ... n-1

∴ $O(n^2)$

Eg 2

prefixAverages2 (X[1:n])

```
{
1)   s = 0
2)   for (i = 0 to n)
     {
3)     s = s + X[i]
4)     A[i] = s / (i+1)
     }
5)   return Array A
}
```

Analysis

→ line 3-4 execute n+1 times

∴ $O(n)$

Terminology

Logarithmic : $O(\log n)$

Linear : $O(n)$

Quadratic : $O(n^2)$

Polynomial : $O(n^k)$, $k \geq 1$

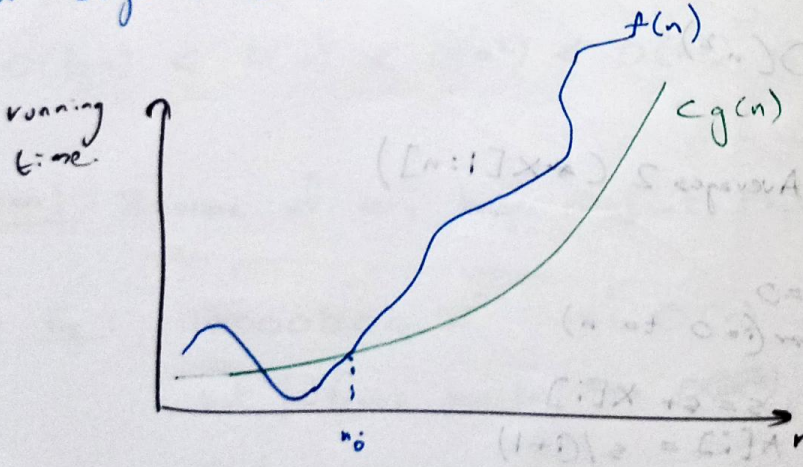
exponential : $O(a^n)$, $a > 1$

Big Omega $\Omega()$

→ Asymptotic lower Bound.

→ $f(n) \Rightarrow \Omega(g(n))$, if there exist c & n_0, such that $cg(n) \leq f(n)$, $n \geq n_0$

→ Used to describe the best case or lower bounds for algorithmic problems.



Big-Θ Theta $\Theta(n)$

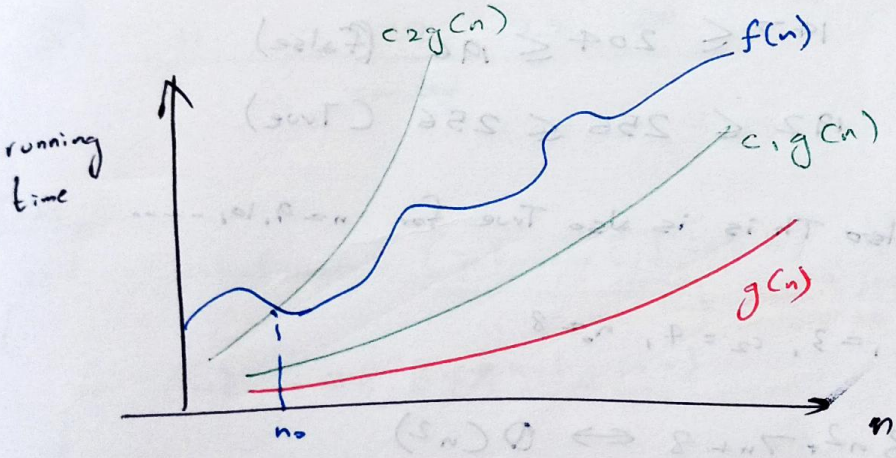
→ Asymptotic Tight Bound.

$f(n) \Rightarrow \Theta(g(n))$, if there exist c_1, c_2, n_0, such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \quad n \geq n_0$$

→ $f(n)$ is $\Theta(g(n))$ iff $f(n)$ is $O(g(n))$ & $f(n)$ is $\Omega(g(n))$

→ $f(n)$ is sandwiched between $c_1 g(n)$ & $c_2 g(n)$



Eg: $3n^2 + 7n + 8 \Leftrightarrow \Theta(n^2)$

Average Case

There should exist c_1, c_2, n_0, such that

$$c_1 n^2 \leq 3n^2 + 7n + 8 \leq c_2 n^2 \quad \forall n \geq n_0$$

$c_1 = 3, \; c_2 = 4$ (let)

$$\therefore 3n^2 \leq 3n^2 + 7n + 8 \leq 4n^2$$

$n = 1 \Rightarrow \quad 3 \leq 18 \leq 4$ (false)

$n = 2 \Rightarrow \quad 147 \leq 204 \leq 196$ (false)

$n = 8 \Rightarrow \quad 192 \leq 256 \leq 256$ (True)

~~also~~ This is also True for $n = 9, 10, \ldots$

$\therefore c_1 = 3, \; c_2 = 4, \; n_0 = 8$

$\therefore 3n^2 + 7n + 8 \Leftrightarrow \Theta(n^2)$

Worst Case

we know that

$$3n^2 + 7n + 8 \leq 3n^2 + 7n^2 + 8n^2$$
$$\leq 18 n^2$$

$\therefore$ we can take $c = 18, \; n_0 = 1$, so that

$3n^2 + 7n + 8 \leq cn^2$ is satisfied

$\therefore 3n^2 + 7n + 8 \Leftrightarrow O(n^2)$

Selection Sort

Eg: I) 64 25 12 22 11
↑ smallest

II) 11 | 25 12 22 64
sorted. unsorted. ← smallest

III) 11 12 | 25 22 64
sorted ← smallest

IV) 11 12 22 | 25 64
sorted. ← smallest

V) 11 12 22 25 | 69
sorted. ← last element no check

Code

```
void selectionsort (int arr[], int n)
{
    for (int i=0; i<n-1; i++)        // see last element, no check
    {
        min_idx = i;
        for (j= i+1; j<n ; j++)
            if (arr[j] < arr[min_idx]
                min_idx = j;
        swap (&arr[min_idx], &arr[i]);
    }
}
```

Time Complexity: Two loop always happens completely

$\therefore O(n^2)$

Eg: Search

→ Divide and Conquer Algo.

Eg: Search 22 14

I)

2	4	5	7	8	9	12	14	17	19	22
↑ low					↑ mid					↑ high

II) 9 < 14, low = mid.

2	4	5	7	8	9	12	14	17	19	22
					↑ low		↑ mid.			↑ high

III) 14 = 14 search done.

Recursive Binary Search Algorithm

```
Algorithm  Binary_Search (A[], k, low, high)
{
    if (low > high)
        return ~~error~~;

    mid ← (low + high)/2

    if (k == A[mid])
        return mid;
    else if (k < A[mid])
        ~~Binary search~~ return Binary_search [A, k, low, mid-1)

    else
        return Binary_Search [~~A, k, mid+1, high~~) (A, k, mid+1, high)

}
```

Prerequisite

→ Needs Sorted Array.

Running Time.

→ The range of candidates to be searchd get halves after each comparision.

→ For array, access takes O(1), thus

Time Complexity = $O(\log n)$ → base 2

Linear Search

Algorithm Linear_Search (A[], q, n)

```
{
    for (int i=0; i< n; i++)
    {
        if (A[i] == q)
            return i;
    }
    return -1;
}
```

Time Complexity

→ Worst : O(n)
→ Average : Θ(n)

→ we can't do better, This is lower Bound.