Recursive tree Method

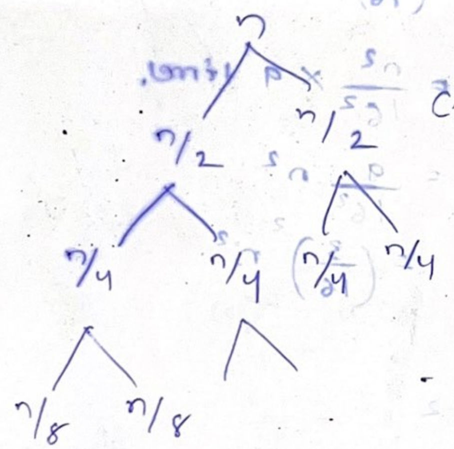
You know about tree and its structure.

$$\boxed{T(n) = 2T(n/2) + cn}$$

Here n is divided into two parts i.e. $n/2$

main problem n to two $n/2$ subproblems.

The cost to perform these divide and conquer technique is cn



cn → To do this step (divide & conquer) cn cost is required

cn — $n/2$ problem is again divided into two. $n/4$

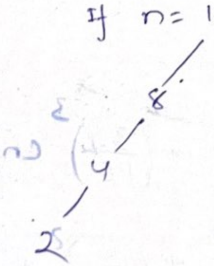
. . cn

To do 3 steps $cn + cn + cn = 3cn$ costs.

3 → depends the height of the tree.

eg.

If $n = 16$ (no. of items)



here height is 4.

If I write $\log_4 16$
$= 4$
$=$ height.

Now

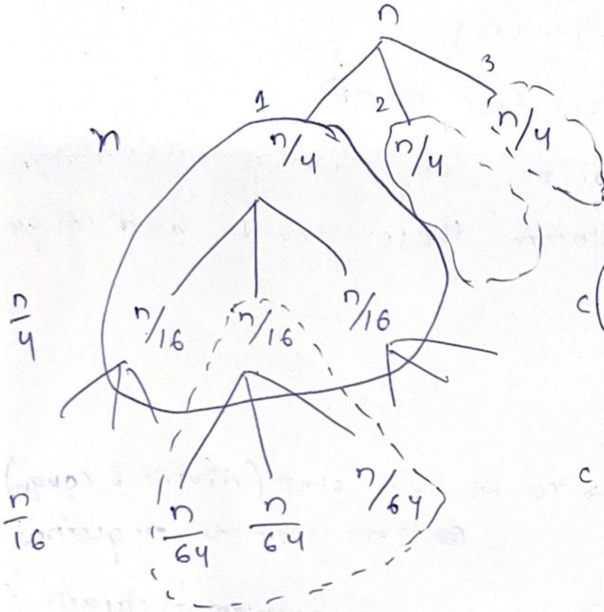
$cn \times \log n$

$= O(n \log n)$

2

$T(n) = 3T(n/4) + cn^2$

$(n/4) \quad n/4 \times 1/4 = n/16$



cn^2

(to divide & conquer & combine cost cn^2

$c\left(\frac{n}{4}\right)^2 = \frac{n^2}{16} \times 3$ times

$= \left(\frac{3}{16}\right) n^2$

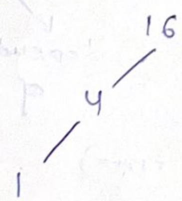
$c\left(\frac{n}{16}\right)^2 = \frac{n^2}{16^2} \times 9$ times.

$= \frac{9}{16^2} n^2$

$= \left(\frac{3}{16}\right)^2 n^2$

$\vdots$

$\left(\frac{3}{16}\right)^3 n^2$



$\log_4 n$

G.P series.

soln

$= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 + \dots$

$= cn^2 \left[1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^2 + \dots\right]$

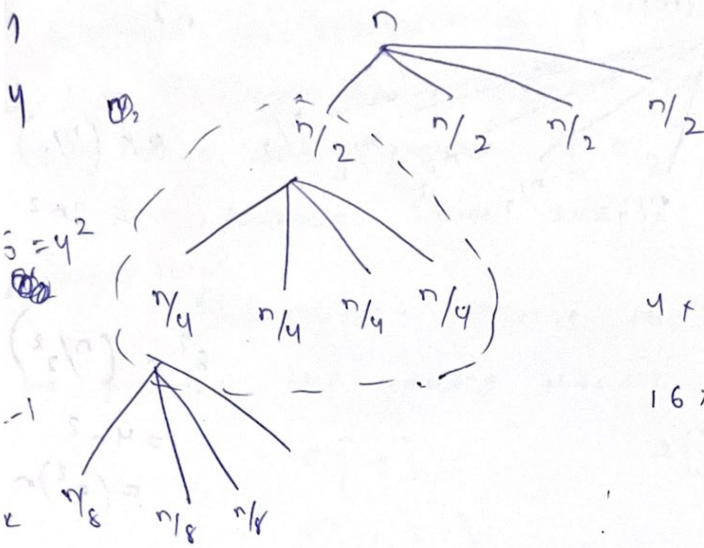
$1 + r + r^2 + r^3 + \dots$

$= \frac{1}{1-r} \quad r < 1$

$= cn^2 \left[\frac{1}{1 - \frac{3}{16}}\right] = \frac{1}{13/16} = \frac{16}{13} cn^2$

$O(n^2)$.

③ $T(n) = 4T(n/2) + n^2$



n^2

$4 \times \left(\frac{n}{2}\right)^2 = 4 \times \frac{n^2}{4} = n^2$

$16 \times \left(n/4\right)^2 = 16 \times \frac{n^2}{16} = n^2$

$4^{k-1} \times \left(\frac{n}{2^{k-1}}\right)^2$

$\frac{n}{2^k}$

$T(n) = 4^k \, T(n/2^k) + kn^2$

$\frac{n}{2^k} = 1 \quad , \quad k = \log n$

$= 4^{\log n} + (1) + n^2 \log n$

$n^{\log_2 4} = n^2 + n^2 \log n = \theta(n^2 \log n)$

---

master method

$T(n) = aT(n/b) + n^{\log_b a} \cdot \log_b^k n$

then $\theta\left(n^{\log_b a} \cdot \log^{k+1} n\right)$

① $T(n) = 4T(n/2) + n^2 (\log n)^2$

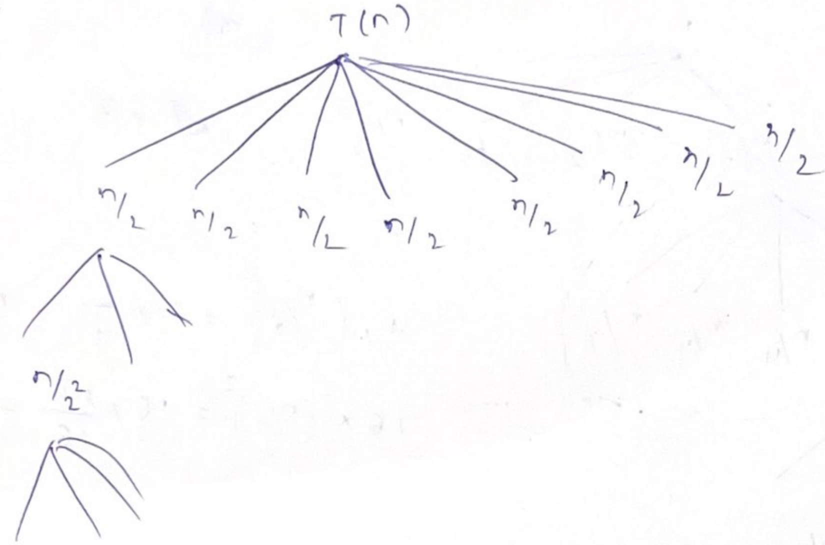
sol^n $\theta(n^2 \log^2 n)$.

② $T(n) = 2T(n/2) + \underline{n \log n}$

sol^n $= \theta(n \log^2 n)$

$= n(\log n \cdot \log n)$

$$T(n) = 8T(n/2) + n^2$$



1 $\quad T(n) \quad n^2$

8 $\quad 8 \times (n/2)^2 = 2n^2$

$8^2 \quad 8^2 \times (n/2^2)^2 = 4n^2 = (2^2)n^2$

$8^3 \quad 8^3 \times (n/2^3)^2 = 2^3 n^2$

$8^{k-1} \; T(n/2^{k-1}) \quad 8^{k-1}\left(\frac{n}{2^{k-1}}\right)^2 = n^2 \times \frac{1}{2}$

$8^k \; T(n/2^k) \quad 8^k \times \left(\frac{n}{2^k}\right)^2 = 2^k n^2$

$$T(n) = 8^k T(n/2^k) + n^2\left[1 + 2 + 2^2 + 2^3 + \dots + 2^{k-1}\right]$$

$$= 8^{\log_2 n} \cdot T(1) + n^2\left[2^k - 1\right]$$

$\frac{n}{2^k} = 1$

$$= n^3 + n^3 - n^2$$

$$T(n) = \Theta(n^3).$$

Change of variable Method:

$$T(n) = aT(\sqrt{n}) + f(n)$$

By changing variables of above recurrence relation can convert into recurrence relation which can apply master theorem.

$$T(n) = 2T(\sqrt{n}) + 1$$

Assume $n = 2^m \Rightarrow m = \log_2 n$ (convert n value to power of 2)

$$T(2^m) = 2T(2^{m/2}) + 1$$

Assume $T(2^m) = S(m) \Rightarrow S(m) = 2S(m/2) + 1$ Apply master method.

$m^{\log_2 2} = m^1 > m^0$

$5m = \theta(m) \Rightarrow T(n) = \theta(\log_2 n)$

Merge sort

The merge sort algo closely follows the D and C strategy. It operates as follows.

(1) Divide the n element sequence to be sorted into two sub-sequences of $n/2$ elements each.

(2) Conquer - Sort the 2 subsequence using merge sort.

(3) Combine - Merge the 2 sorted sub-sequences to produce the sorted answer.

$a[1] \; ---- \; a[n/2]$

$a[n/2+1] \; ---- \; a[n]$

Algo Merge sort (low, high) ———— $T(n)$

/* a[low, high] is a global array to be sorted. Small (P) is true if there is only one element to be sorted. In this case the list is already sorted

/ {

if (low < high) // if there are more than one element.

// Divide P into two subprob.

// Find where to split the set.

$mid = (low + high)/2$

// Solve the individual subproblems.

Merge sort (low, mid) ———— $T(n/2)$

Merge sort (mid+1, high) ———— $T(n/2)$

// Combine the soln.

Merge (low, mid, high) ———— $\theta(n)$

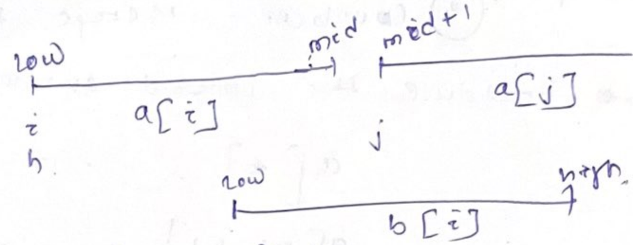
n/2 n/2

Algorithm merge (low, mid, high)

a [low ; high] a global array containing 2 sorted subarray in a [low, mid] and a [mid+1, high]

The goal is to merge these 2 sets into a single set residing in a[low, high]. The array b[ ] is an auxilary global array.

{
h := low
i := low
j = mid+1




```
while ((h ≤ mid) and (j ≤ high) do
{
    if ( a[h] ≤ a[j] ) then
    {
        b[i] = a[h] ; h = h+1 ;
    }
    else
    {
        b[i] = a[j]; j = j+1
    }
    k = k+1
}
if (h > mid) then
    for ( k = j to high ) do
    {
        b[i] = a[k] ; i = i+1
    }
else
    for (k = h to mid) do
    {
        b[i] = a[k]; i = i+1 ;
    }
```

① min = n/2 runs
② max = n-1

n times

for k := low to high do

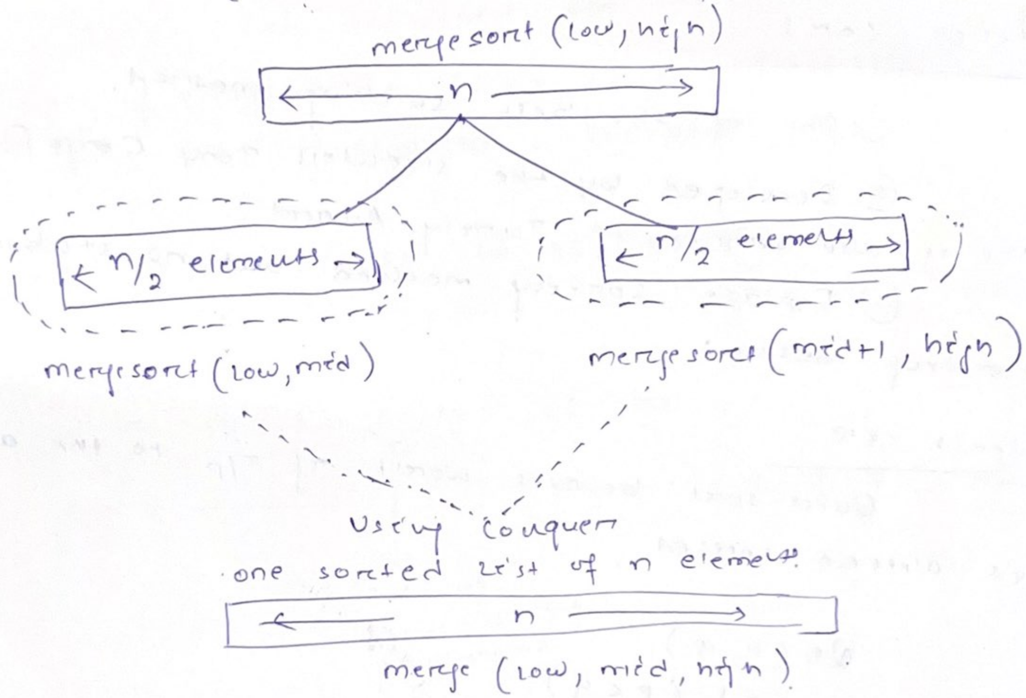
n times

$a[k] := b[k]$

}

Merge Sort

Assume a[low ..... high] array of n elements.



TC

$n + n = 2n$

$\theta(2n) = \theta(n)$

$T(n) \equiv$ TC of mergesort array of n elements

$$T(n) = \begin{cases} 2T(n/2) + n & n > 1 \\ a & n = 1 \end{cases}$$

$f(n) = n^1$ $\quad n^{\log_b a} = n^1$

so case 2

i.e. $\theta(n * \log n)$ in all cases

Depth of recursion of merge sort $= \theta(\log n)$

Space complexity of Merge sort

a[n] and b[n] auxillary array n

$\log_2 n \rightarrow$ stack space

Total $n + \log_2 n$

$= O(n)$

Quick sort

1. One of the best sorting method.
2. Developed by the scientist Tony George [1970] He is also winner of Turing Award.
3. Inplace sorting method but no stable sorting method.

Worst case

Quick sort behaves worst if i/p to the array is already sorted.

```
Qs (p, q)
  if (p<q)
  {
     j = partition (a, p, q)
     Qs (p, j-1) => Qs (1, 0)
     Qs (j+1, q) = Qs (2, n)
```

n-1 element

n

n-1 elements

0   n-2 elements

n-3

0   0

O(n)

T(n) =

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

$$\begin{cases} T(n-1) + n & n > 1 \\ \alpha & n = 3 \end{cases}$$

By solving this we will get $\theta(n^2)$.

Running time of Quick sort

$T(n)$ = Time required for recursive calls of i no. of elements + $(n-i-1)$ no. of elements + Partition + combine.

a

65 70 75 80 85 60 55 50 45 ∞ $n=9$

60, 45, 50 55 65 85 80 75 70

$\leftarrow i \rightarrow$

$(n-i-1) = 9-4-1 = 4$

pos of partitioning element.

$$\boxed{T(n) = T(i) + T(n-i-1) + cn} \quad \text{eq } (1)$$

Worst case

Let the partition element be the smallest element or greatest at all the time $\Rightarrow i = 0$

$T(n) = T(0) + T(n-1) + cn$

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

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n-1) + cn & n > 1 \end{cases}$$

Given $T(n) = T(n-1) + cn$

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

$= T(n-3) + c(n-2) + c(n-1) + cn$

$\vdots$

The worst case occurs when the partition process always picks the greatest or smallest element as the pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case is when the array is already sorted in increasing or decreasing order.

$$T(n) = T(n-n) + (1+2+3+\dots+n)c$$
$$= T(0) + \frac{n(n+1)}{2} \cdot c$$
$$= 1 + \frac{n^2+n}{2} \cdot c$$
$$= O(n^2)$$

Best case:

Position of the partitioning element in such a way that it will divide the array into equal part.

$$T(n) = T(i) + T(n-i-1) + c$$
$$= T(n/2) + T(n/2) + cn$$
$$T(n) = 2T(n/2) + cn$$
$$= O(n \log n).$$

ex. →

1	2	3	4	5	6	7	8	9	10	$i \geq j$
65	70	75	80	85	60	55	50	45	∞	2 < 9
65	45	75	80	85	60	55	50	70	∞	3 < 8
65	45	50	80	85	60	55	75	70	∞	4 < 7
65	45	50	55	85	60	80	75	70	∞	5 < 6
60	45	50	55	65	85,	80,	75,	70	∞	6 > 5

X ——— Y

45	50	55	60	65	70	75	80	85

Algorithm Quicksort (p, q)

// sorts the elements a[p] ..... a[q] which reside in the global array a[1:n] into ascending order ; a[n+1] is considered to be defined and must be $\geq$ all the elements in a[1:n]

{

if (p<q) then // If there are more than one elements.

{

// divide P into two subproblems.

j = partition (a, p, q+1)

// q is the position of the partitioning element.

// solve the subproblems.

Quick sort (p, j-1)

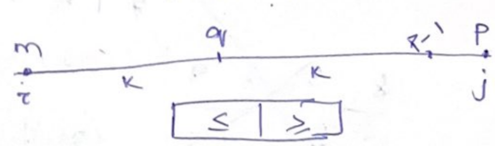
" " (j+1, q)

}

}

Algorithm Partition (a, m, p)

within a[m], a[m+1], ... a[p-1] the elements are rearranged in such a manner that if initially t=a[m] then after complition a[q] = t for some q between m and p-1. Then a[k] $\leq$ t for m $\leq$ k $\leq$ q and a[k] $\geq$ t for q<k<p. q is written as set a[p] = ∞.



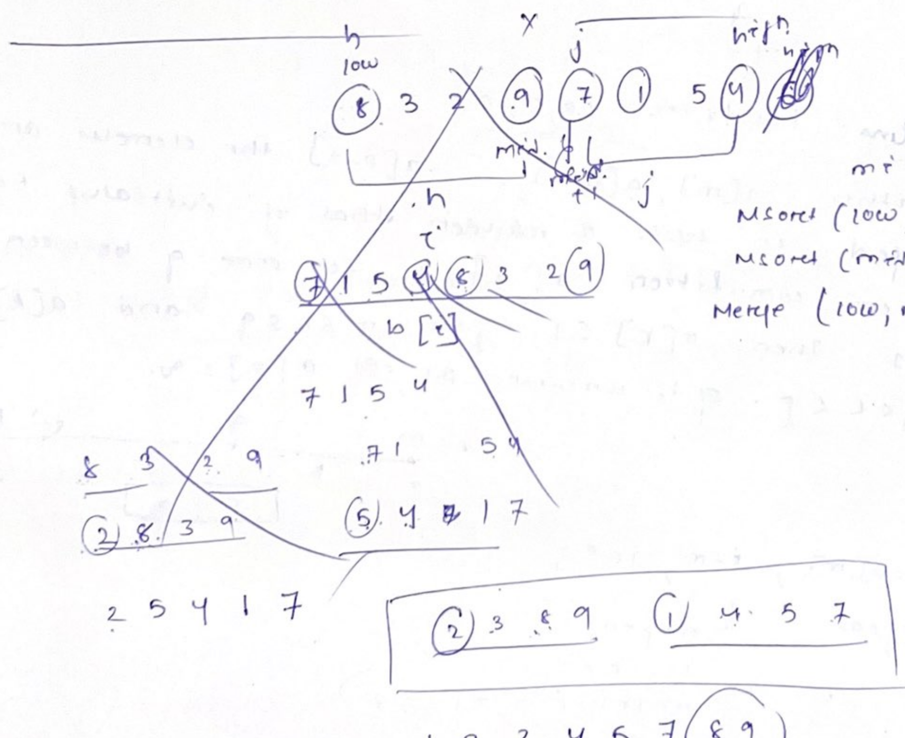
{

v: = a[m] , i=m; j=r ;

repeat {
repeat
i = i + 1
until (a[i] $\geq$ v);
repeat.
j = j - 1 ;
until (a[j] $\leq$ v);

```
    if ( i<j ) then   Interchange (a, i, j);
  }
  until (i ≥ j);
  a[m] := a[j]
  a[j] := v
  return j
}

Algorithm Interchange (a, i, j)
  // Exchange a[i] with a[j]
  {
    p := a[i]
    a[i] := a[j]
    a[j] := p;
  }
```



mid = ⌊low+high

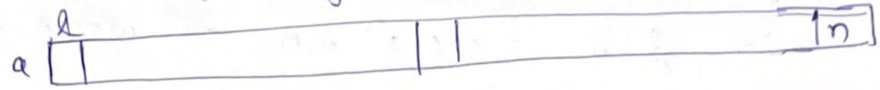
MSort (low, mid)

MSort (mid+1, high)

Merge (low, mid, high)

Binary search

If is applied for sorting elements.



$l = 1, \; h = n \qquad m = \frac{l+h}{2}$

if $(x < a[m])$ then

$h = m - 1$

// B search $a(l, m-1)$

if $(x > a[m])$ then

$l = m + 1$

// B search $a(m+1, n)$

if $(x = a[m])$ then

return (m).

Algo Binsearch (a, n, x)

```
{
   // a[1, ....n] array of n sorted elements
      l = 1 ; h = n ;
   while ( l <= h )
      {
         m = (l+h)/2 ;
         if ( x < a[m] )
              h = m - 1
         else if ( x > a[m] )
              l = m + 1 ;
         else  return (m) ;
      }
   return (-1)   //element not found.
```

Time complexity

Best case $\Rightarrow O(1)$

if x = middle element of array.

worst case

n

$n/2$

$k = \log_2 n \qquad n/2^2$

$\vdots$

$n/2^k$

so $TC = O(\log n)$

for n i/p = log n is worst case.
" 2^n " = $\log 2^n = n$ is worst case.

Average case

$$\frac{1+2+3+\cdots\cdots+\log_2 n}{\log_2 n}$$

$$= \frac{\log n\ (\log n + 1)}{\log n} = \theta(\log n).$$

Space complexity

3 variables are used in the algo.
O(1) = constant space.

Recursive Binary search

```
Algo RBsearch (l, h, x)
{
    // a [l....h] array of n sorted elements.
  if (l == h)        // one element.
  {
    if ( a[ ] == x )
        return (1) ;
    else
        return (-1) ;     // unsuccessfull search.
  }
  else.
  {
      m = (l+h)/n
      if ( x < a [m] )
          RBsearch (l, m-1, x)
      else if (x > a[m])
          RBsearch (m+1, h, x)
      else
          return (m) ;
  }}
```

Best case
The algo. terminates at
time. TC = θ(1)
min. depth of rec
Worst case
T(n) = { a

Best case

The algo. terminates at return(m) in the 1st time.

$$TC = \theta(1)$$

min. depth of recursion = $\theta(1)$

Worst case

$$T(n) = \begin{cases} a & n = 1 \\ T(n/2) + c & n > 1 \end{cases}$$

Because the sol^n is found in one sub half. we are not considering the both side sub half.

$$O(\log_2 n)$$

Space complexity

$$aT(n/2) + f(n)$$

depth of this algo $= \log_b n$

worst case space complexity $= O(\log_2 n)$ # stack space.

. M.P

So if we compare Bsearch algo with the recursive RBsearch algo, the normal Bsearch is better than the recursive one because space complexity for Bsearch is $O(1)$ where as recursive algo is $O(\log_2 n)$.

Brute force Algo

- Not intelligent programming or logic.
- It computes all feasable solution without avoiding recomputation.
- If exponential number of feasible solution brute force requires expoential time complexity.
- less space complexity.

Dynamic Programming

- Intelligent Programming
- Computes all feasable solution recursively by avoiding recomputation.

(x_1, x_2) = result 1
(x_1, x_2, x_3) = result 2.

- If problem nature can allow for usability even expoential feasible solution problem may runs polynomial time.
- More space complexity because required to store result of every subproblem.

Greedy Method

- Solves problem satically
[Pre-defined Greedy startegy]
- computes only one feasible solution. based on greedy startegy.
- Runs always in polynomial time.
- Every optimized problem solves by greedy method.

e.g. 0/1 knapsack probem fails to solve by greedy method.

Bubble sort

1	2	3	4	5	6	7
5	2	9	2	9	4	3

Pass 1

2 5

2 9

4 9

3 9

2	5	2	9	4	9	3	9

Pass - 2

2 5

4 9

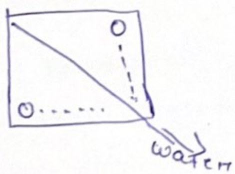
2 9

3 9 9

2	2	5	4	9	3	9	9

Pass - 3

: Apply all passes i.e. (n - 1) passes.



why it is so named bubble?

As passes increases then No. of swaps are decreased. Every pass some element occupies its position.

Algo Bubble (a, n)

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

Time complex

Passes	No. of comparision	No. of swaps
pass - 1	(n - 1)	[0 .... to ..... n - 1]
pass - 2	(n - 2)	[0 ... to n - 2]
pass - 3	(n - 3)	[0 .... to n - 3]
⋮	⋮	⋮
(n - 1) pass	1	[0 to 1]

Time complexity of Bubble sort =

$$= \frac{n(n-1)}{2} + \left[0 \text{ to } \frac{n(n-1)}{2} \right]$$

$$= O(n^2) \quad [\text{all cases}]$$

Selection sort

find position of minimum element from a[to a[n] and swap with a[k] where k = 1 to n.

pass-1

	1	2	3	4	5	6	7
a	9_a	5	6_a	9_b	6_b	1	2

pass-2

$\boxed{1}$ 5 6_a 9_b 6_b 9_a 2

pass-3

$\boxed{1 \mid 2}$ 6_a 9_b 6_b 9_a 5

$\boxed{1 \mid 2 \mid 5}$ 9_a 6_b 9_a $6a$.

⋮

pass n-1

Algo. selection (a, n)

```
{
  for ( i = 1 ;  i ≤ n - 1 ;  i ++ )
  {
     min Pos = i
     for ( j = i + 1 ;  j ≤ n ;  j ++ )
     {
        if ( a[i] < a[min Pos] )
            min Pos = j
     }
     swap ( a[i], a [min Pos] )
  }
}
```

Time complexity

Passes	No. of comparision	No. of swaps
1	n - 1	1
2	n - 2	1
3	n - 3	1
⋮	⋮	⋮

Time complexity of SS

No. of comparision + No. of swap

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

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

I.M.P
if element concatenate big data selection sort best because element movements very less. (less swapping or shift)

Quick sort = $c_1 \cdot n \log n$

selection sort = $c_2 \cdot n^2$

I.M.P
if $(n < 20)$ // small no. of elements then selection sort runs faster than Quick sort.

if $(n > 20)$ // Large data.
Quick sort runs faster then selection sort.

I.M.P
selection sort behaves same as quick sort if partition element is minimum element of the list.