Data Structure

Q-1 Insertion Sort:-

```
void insertionSort (int A[], int n)
{
    for (i=1; i<n; i++)
    {
        j = i-1;
        x = A[i];
        while (j > -1 && A[j] > x)
        {
            A[j+1] = A[j];
            j--;
        }
    }
}
```

Insertion sort performs two operations:-

It scans through the list, comparing each pair of elements, & and it swaps elements if they are out of order. Each operation contributes to the running time of the algorithm. If the order input is already in sorted order, insertion sort compares O(n) elements & performs no swaps (in the above code, the inner loop never triggered). Therefore, in the

Scanned with CamScanner

best case, insertion sort runs in $O(n)$ time.

In an insertion sort algorithm, there are always two constraints in time complexity. One is shifting the element and the other one is comparison of the elements. The time complexity is also dependent on the data structure which is used while sorting. If we use array as data structure then shifting takes $O(n^2)$ in the worst case while using linklist data structure, searching takes $O(n^2)$

Sort 50, 40, 30, 20, 10 using arrays

If we solve above list using array data structure we can use binary search for comparison which will lead to a time complexity of $O(n \log n)$ but shifting takes $O(n^2)$. Therefore, the total time complexity becomes $O(n^2)$

To solve this problem, linklist can be used. In a linklist shifting takes $O(1)$ as new element can be inserted at their right position without shifting. Here as we cannot use binary search

for comparison which will lead to a time complexity of $O(n^2)$ even though shifting takes a constant amount of time. As we have observed in the above example, in both the cases the time complexity is not getting reduced. Hence, we are using an improvised insertion sort taking additional space to sort the elements.

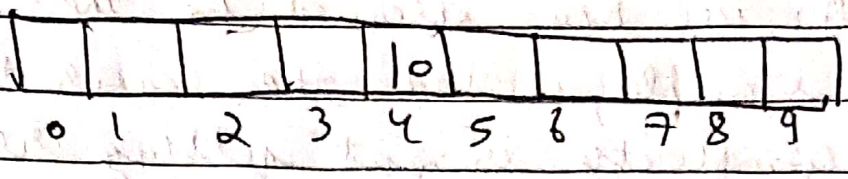
From: UKESSAYS (Provides of free study resources)

In the insertion sort technique proposed here, we will take $2n$ spaces in an array DS, where n is the total no. of elements. The insertion of elements will start from $(n-1)^{th}$ pos. of the array. The same procedure of a standard insertion sort is followed in this technique. Finding the suitable positions of the elements to be inserted will be done using binary search. In the following cases, we will discuss the details of our work:

Case 1: For the best case scenario in a standard insertion sort is the input elements in ascending order using proposed technique:

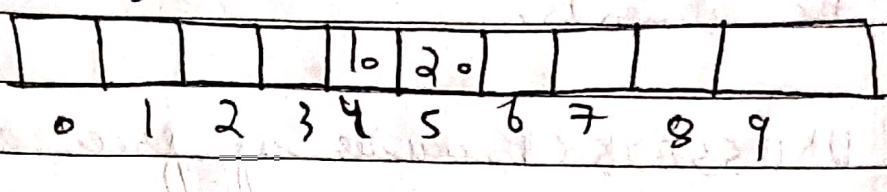
e.g. 10, 20, 30, 40, 50

i)



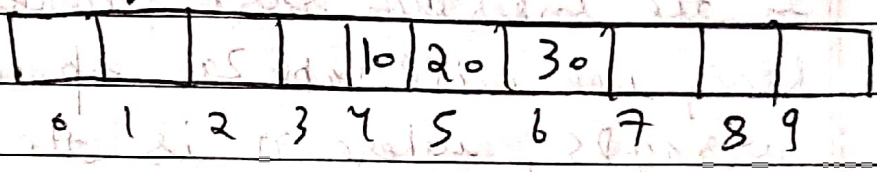
shift = 0 , comparison = 0

ii)



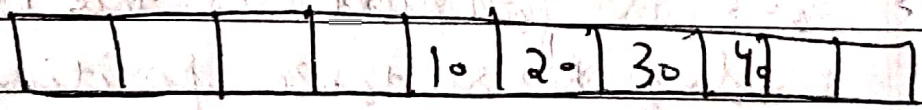
shift = 0 , comparison = 1

iii)



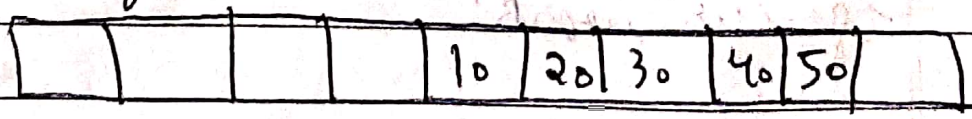
shift = 0 , comparison = 1

iv)



shift = 0 , comparison = 1

v)



shift = 0 , comparison = 1

Scanned with CamScanner

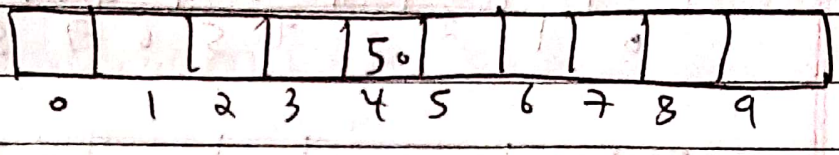
total shift = 0, total comp. = n - 1

∴ time complexity = O(1) + O(n)
= O(n)

Case 2:- for worst case scenario in a standard insertion sort is the input elements in descending order using proposed technique.

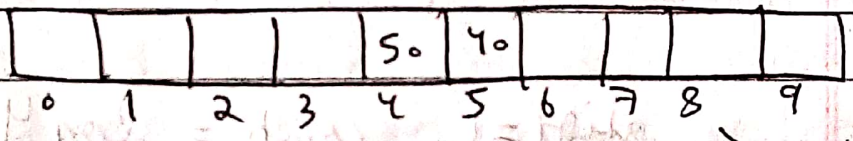
e.g. 50, 40, 30, 20, 10

i)

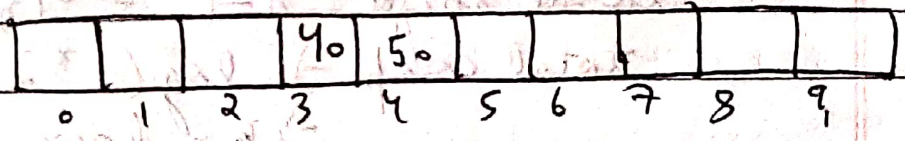


shift = 0 & comp. = 0

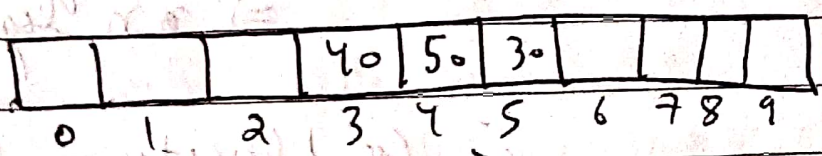
ii)



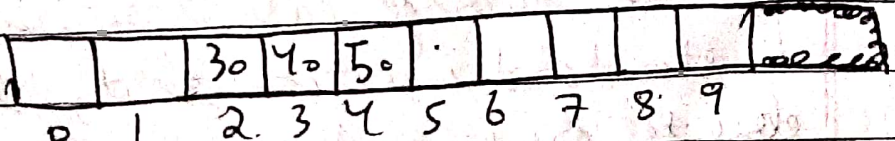
shift = 1
comp. = log 1



iii)

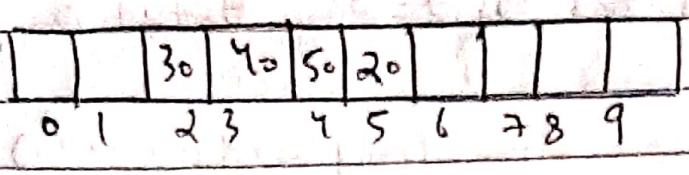


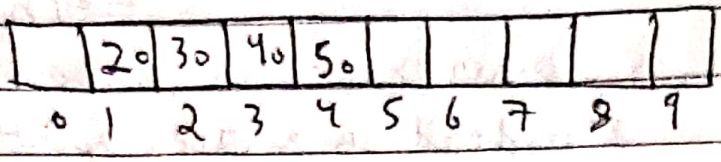
shift = 1, comp. = log 2



Scanned with CamScanner

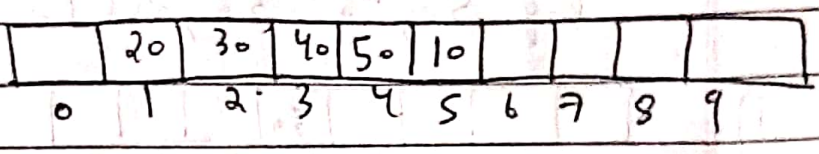
iv)

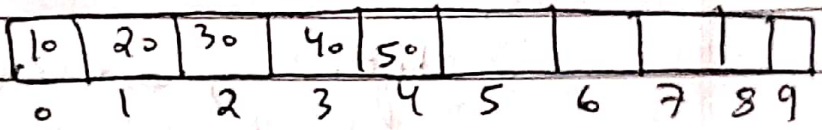




shift = 1 , comp = log 3

v)





shift = 1 , comp = log 4

total shift = n-1 = O(n)

total comp ⇒ log(1 * 2 * 3 * 4)

⇒ n log n

⇒ n log n

∴ Complexity = O(n log n)

Case 3:- for the average case scenario in a standard insertion sort, the input elements are in random order. We are following the same procedure but

Comp. is done via binary search algo., hence it takes $O(n \log n)$ for comp., for shifting the elements the time taken tends to $O(n^2)$ but is not equal to $O(n^2)$. As we have more spaces, there are possibilities that the shifting of some elements may be reduced because elements may be inserted both at the end as well as in the beginning.

Q-2:- Quick sort (in-place sorting technique)

* In quick sort an element is in sorted position if all elements before it are smaller and all the elements after it are greater.

40 30 20 (50) 90 70 80

→ at sorted position

⇒ i will look for element "greater" than pivot element.

⇒ j will look " " "smaller" or equal than pivot element.

(50) 70 60 90 40 80 10 20 30 ∞

pivot i j

1. (50) 70 60 90 40 80 10 20 30 ∞

i → j

(50) 30 60 90 40 80 10 20 70 ∞

i → j

2. (50) 30 60 90 40 80 10 20 70 ∞

i → j

(50) 30 20 90 40 80 10 60 70 ∞

i j

3. (50) 30 20 90 40 80 10 60 70 ∞

i → j

(50) 30 20 10 40 80 90 60 70 ∞

i j

4. (50) 30 20 10 40 80 90 60 70 ∞

j i

⇒ exchange pivot and 'j' elements

if 'i' element is greater than pivot and 'j' element is smaller than pivot, then exchange elements at 'i' and 'j'

left side

right side

(4 3 2 1) 5 (8 9 6 7 ∞)

(now sort again with quicksort) — partitioning position — (sort again with quicksort)

⇒ minimum two elements must be there for quick sort.

⇒ ∞ will act as infinity for right list and partioning element will act as infinity for left side

∴ quicksort uses portioning method to sort the element with recursive calls.

→ Worst case:-

for sorted list (ascending) j move from last to front

∴ if list have n elements, j moves n times for next turn j move (n-1) times

$$1+2+3+\ldots+n$$

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

→ Best case:-

Scanned with CamScanner

⇒ Assume element from 15 elements

⇒ let portioning is done at middle.

1 ——— 15

⇒ if we select randomly any element as pivot then this is called randoming quick sort.

for level 1 → n comp.

2 → 11

3 → 11

4 → 8 no comp.

let assume there are 16 elements

$\frac{16}{2} = \frac{8}{2} = \frac{4}{2} = \frac{2}{2} = 1$

$\therefore \log_2(16) = 4$ level (for n=16)

$\log_2(n) = n$ level (for n=n)

$\therefore$ total comparisons $\simeq o(n)$

& comparisons are done for $\simeq o(\log n)$

$\therefore$ time complexity (time for comp.) $\simeq o(\log n)$

Scanned with CamScanner

Date
Page

First element

Best Case → if pos. is in middle
time → $O(n \log n)$

Worst Case → if pos. is done on either side (i.e., for sorted list)
time → $O(n^2)$

Average Case → $O(n \log n)$

→ if we select middle element as pivot, then

best case → sorted list → $O(n \log n)$
worst " → positioning at any end → $O(n^2)$

→ quick sort is also called as
(a) partition exchange sort
(b) selection " "

→ Quick V/S Selection :-

⇒ in quick sort we select a element and then fixed its position in the list.
⇒ but for selection we select a position and then find element for it.

Scanned with CamScanner

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

⇒ write quicksort func^{n} using above partition func^{n} to sort the elements.

Scanned with CamScanner

→ Bubble Sort :-
(in-place sorting technique)

A	B	5	7	3	2	(n=5)
	0	1	2	3	4	

(1st pass will give max. no
2nd " " " " " " ")

→ 1st pass :-

	comparison1		comparison2		
8	5	5	5	5	(⇒4 comparison
5	8	7	7	7	⇒4 swap
7	7	8	3	3	
3	3	3	8	2	
2	2	2	2	(8)	← largest element is sorted

→ 2nd pass :-

5	5	5	5	(⇒3 comp.
7	7	3	3	⇒3 swap. (max)
3	3	7	2	
2	2	2	(7)	← two element are sorted
8	8	8	(8)	

⇒ 3rd pass :- ⇒ four elemt are sorted, so 5th
⇒ 1 comp
⇒ no swap

Scanned with CamScanner

⇒ no. of pass = 4 ≃ (n-1) passes ≃ O(n)

⇒ no. of comp. = 1+2+3+4
= 1+2+3+...+n-1
= $\frac{n(n-1)}{2}$

⇒ max. no of swap = 1+2+3+4
= 1+2+3+...+n-1
= $\frac{n(n-1)}{2}$
= $O(n^2)$

```
void bubble(int A[], int n)
{ int flag;
  for(i=0; i<n-1; i++)
  { flag=0;
    for(j=0; j<n-1-i; j++)
    {
      if(A[j] > A[j+1])
      {
        swap(A[j], A[j+1])
        flag=1;
      }
    }
    if(flag==0) break;
  }
}
```

control will come out of outer loop

∴ time complexity = $O(n^2)$

→ bubble sort is not adaptive by nature, we make it adaptive by using flag

→ bubble sort is stable

→ Merge Sort :-

1. Merging two list (merging of two sorted list and creating a single sorted list)

Merge:- It is the process of merging multiple sorted list into a new single sorted list.

(i)

	0	1	2	3	4
A	2	10	18	20	23
B	4	9	19	25	
j →	0	1	2	3	

⇒ we will compare A[i] and [j] and store smaller element in C[K]

C	2	4	9	10	18	19	20	23	25
K →	0	1	2	3	4	5	6	7	8

⇒ as soon as one of the list is finished and have we some elements remaining in other list, copy all the remaining elements in C[K] array as it is.

Scanned with CamScanner

```
void merge (int A[], int B[], int m,
            int n)
{ int i, j, k;
  i=j=k=0;
  while (i<m && j<n)
  {
    if (A[i] < B[j])
      C[k++] = A[i++];
    else
      C[k++] = B[j++];
  }
  for ( ; i<m ; i++)
      C[k++] = A[i];
  for ( ; j<n ; j++)
      C[k++] = B[j];
}
```

∴ time complexity $O(m+n)$

2. Merging two list in single array

	(i)				(j)			
A	2	5	8	12	3	6	7	10
	0	1	2	3	4	5	6	7

list 1 list 2

⇒ Here also we compare A[i] & A[j], & will copy smaller element into B[k], & will copy remaining element

B | 2 | 3 | 5 | 6 | 7 | 8 | 10 | 12 |

Scanned with CamScanner

⇒ & copy array B to array A

```
void merge (int A[], int l, int h, int mid)
{ int i, j, k;
  i=l, j=mid+1, k=l;
  int B[h++];
  while (i<=mid && j<=h)
  {
    if (A[i] < A[j])
      B[k++] = A[i++]
    else
      B[k++] = A[j++];
  }
  for (; i<=mid; i++)
    B[k++] = A[i];
  for (; j<=h; j++)
    B[k++] = A[j]
  for (i=l; i<=h; i++)
    A[i] = B[i]
}
```

3. Merging multiple list:-

We will compare A[i], B[j], C[k], D[l] elements and copy smallest element in E[m]; & copy remaining elements

Scanned with CamScanner

→ Two way iterative merge sort :-

```
void Imerge (int A[], int n)
{ int p, i, l, mid, k;
  for (p=2, p<=n; p=p*2)
  {
    for (i=0; i+p-1<n; i=i+p)
    {
      l=i;
      h=i+p-1;
      mid=(l+h)/2;
      merge(A, l, mid, h);
    }
  }
  if (p/2<n)
    merge (A, 0, p/2-1, n-1);
}
```

→ Time complexity = $O(n \log n)$

→ Two way recursive merge sort :-

```
void mergesort(int A[], int l, int h)
{
  if(l<h)
  {
    mid=(l+h)/2;
    mergesort(A, l, mid);
```

Scanned with CamScanner

```
        mergesort (A, mid+1, h);
        merge (A, l, mid, h);
    }
}
```

→ no. of elements merged in each level = $o(n)$

→ Merging is done for $o(\log n)$

∴ time complexity = $o(n \log n)$

Space Complexity:-

space for array A = n
" " auxiliary array B = n
stack size required = $\log n$
∴ total space = $o(n + \log n)$

⇒ Out of comparison based sortings only merge sort require extra space.

Scanned with CamScanner

shifting and comparison done from back side

→ Insertion sort:- (insertion is done in increasing order)

no. of pass = n-1

no. of comparison = $O(n^2)$

no. of swap = $O(n^2)$

⇒ no intermediate result

→ with array we have to shift elements for insertion sort.

→ But with LL we don't have to shift elements.

→ Insertion sort is more compatible with LL then array.

```
void insertsort (int A[], int n)
{
  for(i=1; i<n; i++)
  {
    j = i-1;
    x = A[i];
    while (j>-1 && A[j]>x)
    {
      A[j+1] = A[j];
      j--;
    }
    A[j+1] = x
  }
}
```

Scanned with CamScanner

∴ time complexity $\simeq O(n^2)$

→ Insertion sort is adaptive by nature

time required for insertion sort

max $\simeq O(n^2)$

min $\simeq O(n)$

→ Insertion-sort is stable

Best Case

→ if list is sorted in ascending order (i.e.

time (comp) $\simeq O(n)$

shift $\simeq O(1)$

Worst Case

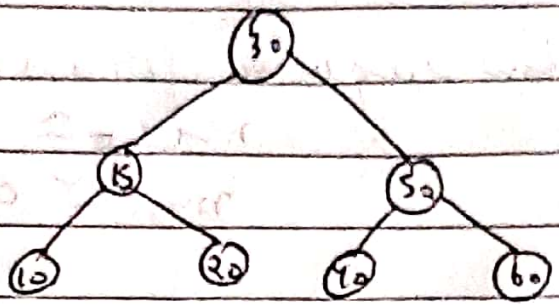
→ if list is sorted in descending order

shift $\simeq O(n^2)$

time complexity $\simeq O(n^2)$

Scanned with CamScanner

BST



It is binary tree in which for every node all the elements on its left sub tree are smaller than ~~or equal to~~ element on that node and all the elements on its right sub tree are greater than element on that node.

→ no. of BST for 'n' nodes (given in nodes)

$$= \left(\frac{^{2n}C_n}{n+1} \right) \text{trees}$$

→
```
node * Rinsert (node * p, int key)
{
    node *t;
    if (p == NULL)
    {
        t = malloc ();
        t → data = key;
        t → l.child = t → r.child = NULL;
```

Scanned with CamScanner

```
return t;
}
if (key < p -> data)
        p -> l.child = insert (p -> l.child,
                                key);
else if (key > p -> data)
        p -> r.child = insert (p -> r.child,
                                key);
return p;
}
int main()
{
node * root = NULL;
root = insert (root, 30);
        insert (root, 20);
}
```

→ For this specific func^n, we have created dynamic array by using malloc to store name during runtime.

And then we compare each letter of name to decide ~~whether~~ whether it will be left or right child of the root name.

Scanned with CamScanner