Date____/____/_____ Page No.:_____

Data Structure

Q1. 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-- ;
            }
          A[j+1] = x;
        }
    }
```

Insertion sort performs two operations :

it scans through the list, comparing each pair of elements, & it swaps elements if they are out of order. Each operation contributes to the running time of the algorithm. If the input is already in sorted

Chitra

order, insertion sort compares O(n) elements & performs no swaps (in the above code, the inner loop is never triggered). Therefore, in the 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 elements & the other one is comparison of the ". 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 link list 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 $O(n^2)$

Date ___/___/_____ Page No.: _____

To solve this problem, link list can be used. In a link list shifting takes O(1) as new element can be inserted at their right pos. 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 examples above, 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 (Providers of free study resources)

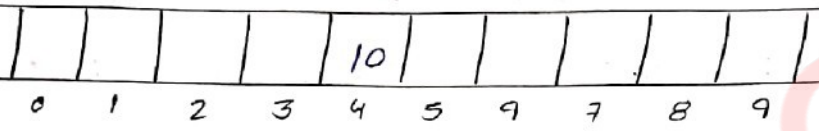
In the insertion sort techniques proposed here, we will take 2n spaces in an array DS, where n is the total no. of elements. The insertion of elements will start from $(n-1)^{th}$ pos. of the array. The same procedure of a standard insertion sort is followed in this technique. Finding the suitable positions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

Date___/___/_____ Page No.:_____

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

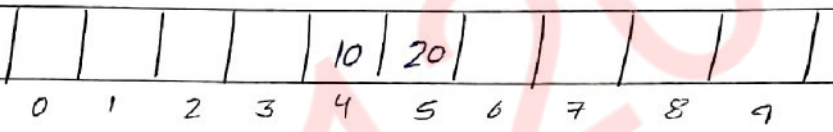
e.g. 10, 20, 30, 40, 50

i)



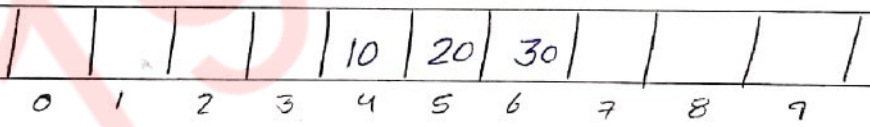
shift = 0 , comparison = 0

ii)



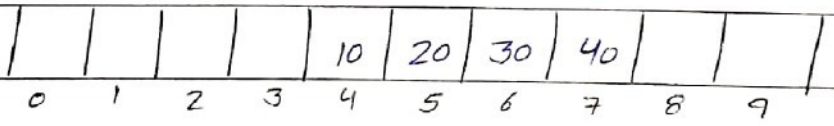
shift = 0 , comparison = 1

iii)



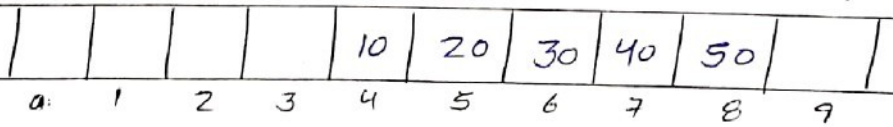
shift = 0 , comparison = 1

iv)



shift = 0 , comparison = 1

v)



shift = 0 , comparison = 1

Chitra

Date __/__/____ Page No.: ____

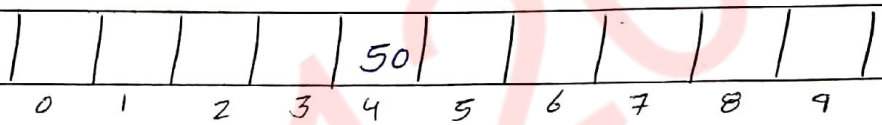
total shift = 0, total comp. = n-1

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

Case 2 for wrost case scenario in a standard insertion sort is the input elements in descending order using proposed technique

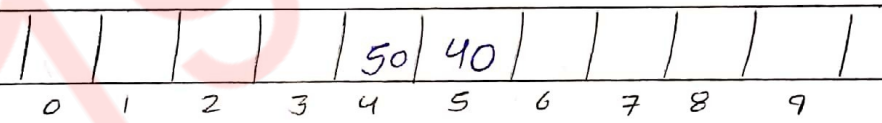
e.g. 50, 40, 30, 20, 10

i)



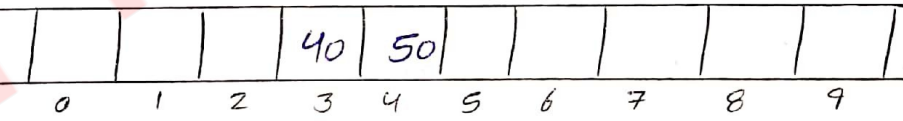
shift = 0, comp. = 0

ii)

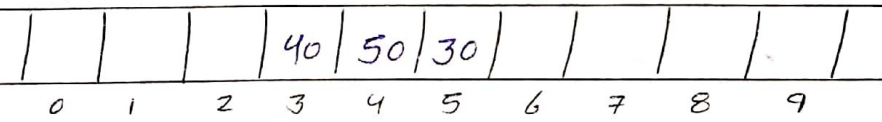


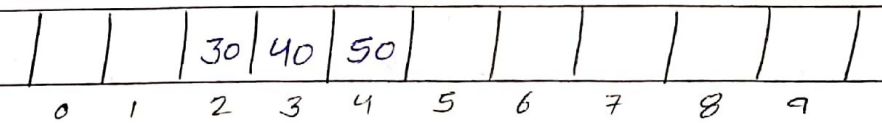
shift = 1

comp = log 1



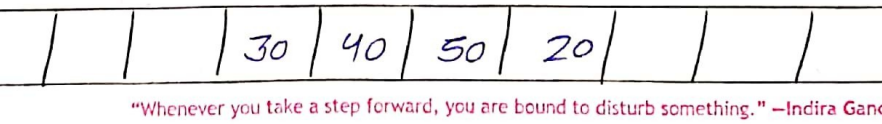
iii)



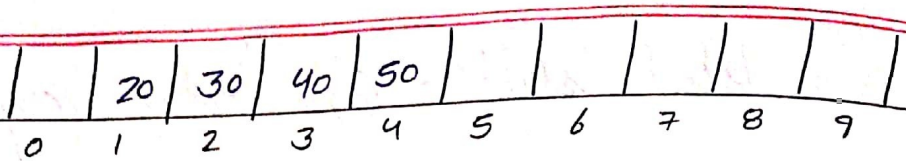


shift = 1, comp = log 2

iv)

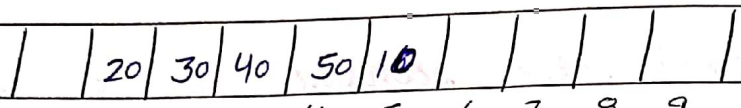


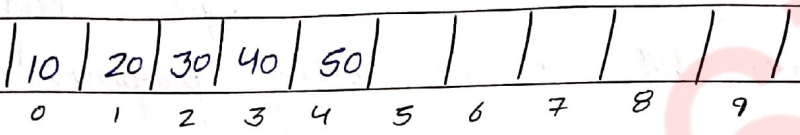
Date __/__/____ Page No.: ____



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)

∴ complexity = O (n log n)

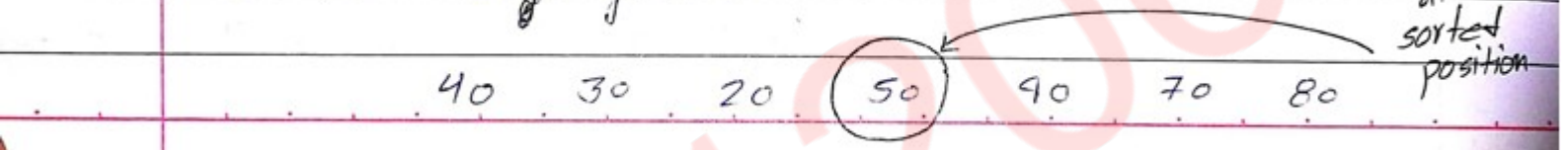
case 3. for the average case scenario in a standard insertion sort, the input elements are in random order. We are following the same procedure but comp. is done via binary search algo., hence it takes O(n log n) for comp., for shifting the elementes the time taken tends to $O(n^2)$ but is not equal to $O(n^2)$. As we have more spaces, there are possibilities that the shifting of some elements

Date ___/___/_____ Page No.: _____

may be reduced because elements may be inserted both at the end as well as in the begining

Q2 Quick sort (in-place sorting technique)

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



pivot (element which have to find its place in list)

50 70 60 90 40 80 10 20 30 ∞

⇒ i will look for element "greater" then pivot element

⇒ j " " " " " "smaller or equal" " "

pivot → 50 70 60 90 40 80 10 20 30 ∞

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

50 30 20 10 ⇒ exchange pivot & 'j' element

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

left side

(40 30 20 10) 50 (80 90 60 70 ∞)

right side

partitioning position

(now sort again it with quicksort)

(sort again it with quicksort.)

⇒ ∴ quicksort is applied recursively on left & right side

Cpj y jlnl Page
Date / /

⇒ minimum two elements must be there for quick sort

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

∴ quicksort uses partioning method to sort the element with recursive calls

Worst case

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

∴ if list have n elements, j moves n times

for next turn j move (n-1) times

:

2

1

here time is used for comparisons

$1 + 2 + 3 + \dots + n$

time complexity ≈ $\frac{n(n+1)}{2} \approx O(n^2)$

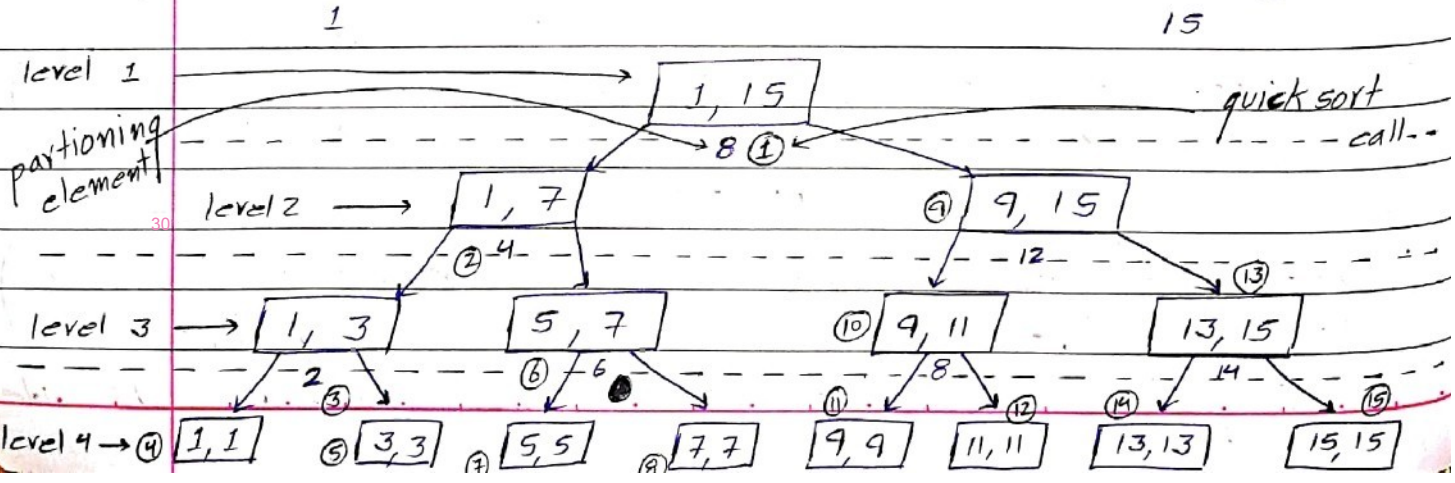
sorted in decending order

comparisons (time complexity) ≈ $O(n^2)$

Best case

⇒ assume element from 1 15 elements

⇒ let partioning is done at middle



⇒ if we slect randomly any element as pivot then this is called randomize quick sort

COmlin Page
Date / /

for level 1 → n comp.
" 2 → "
" 3 → "
" 4 → no comp

let assume there are 16 elements

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

∴ $\log_2(16) = 4$ level (for n=16
$\log_2(n) = n$ " for n=n

∴ total comparisons ≃ $O(n)$
& comparisons are done for ≃ $O(\log n)$ times

∴ time complexity (time for comp.) ≃ $O(n \log n)$

first element → p

Best case → if partitioning is in middle
time → $O(n \log n)$

Wrost case → if partitioning 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 → $O(n \log n)$
wrost " → partioning at any end → $O(n^2)$

⇒ Quick sort is also called as
@ partition exchange sort
@ selection " "

Quick v/s Selection

in quick sort we select a element & then find its position in the list
but for selection we select a position & then find element for it

func^c to find partioning position in the list

```
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]);          // swap pivot & 'j' element
    return j;
}
```

⇒ write quicksort func^c using above partition func^c to sort the elements

30 Bubble sort
(in-place sorting technique)

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

A	8	5	7	3	2	(n = 5)
	0	1	2	3	4	

(swap if second no. is small)

comparison 1 | comparison 2

1st pass

8	5	5	5	5	⇒ 4 comparison
5	8	7	7	7	⇒ 4 swap
7	7	8	3	3	
3	3	3	8	2	← largest element is sorted
2	2	2	2	8	

2nd pass

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

3rd pass ⇒ three element are sorted
⇒ 2 comp.
⇒ max. 2 swap

4th pass ⇒ four element are sorted, so 5th element will automatically get sorted
⇒ 1 comp
⇒ max. 1 swap

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

⇒ no. of comp = 1 + 2 + 3 + 4 (for 5 elements)
1 + 2 + 3 + ... + n−1 (for n ")
≃ $\frac{n(n-1)}{2}$
≃ $O(n^2)$

⇒ max. no. swap = 1 + 2 + 3 + 4 (for n = 5
1 + 2 + 3 + ... + n−1 (for n = n
≃ $\frac{n(n-1)}{2}$
≃ $O(n^2)$

```
void bubblesort (int A[], int n)
{   int flag;
    for (i = 0; i < n-1; i++)                    // no. of pass
    {   flag = 0;
        for (j = 0; j < n-1-i; j++)              // no. of comp. or swap
        {
            if (A[j] > A[j+1])
            {
                swap (A[j], A[j+1])
                flag = 1;
            }
        }
        if (flag == 0) break;                    // no swaping is done
    }
```

control will come out of outer loop means list is already sorted

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

$\Rightarrow$ Adaptive

already sorted list ($n = 4$)

1st pass comparison

2	2	2	2
3	3	3	3
5	5	5	5
7	7	7	7

no. of comp = $n-1$

no. of swap = 0

time complexity $\simeq O(n)$

$\therefore$ time for bubble sort

minimum $\simeq O(n)$

maximum $\simeq O(n^2)$

$\therefore$ bubble sort is not adaptive by nature, we make it adaptive by using flag

$\Rightarrow$ Stable

8	8	8
8	8	3
3	3	8
5	5	5
4	4	4

($\therefore$ bubble sort is stable)

Merge sort

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

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

Camlin Page
Date / /

(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] & B[j], & store smaller element is C[k]

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

- as soon as one of the list is finished & we have some elements remaining in other list, copy all the remaining elements in C[k] array as it is

```
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 $\theta(m+n)$

2. Merging two list in single Array

(i) l			mid	(j)			h
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 remaing element

B	2	3	5	6	7	8	10	12

⇒ & 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++)          // array A = B
        A[i] = B[i];
}
```

3. Merging multiple list

@ M-way merging

i) 4-way merging (no. of sorted list)

A	2	5	15	
B	3	6	18	
C	05	09	12	
D	8	16	20	

sorted list (array)

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

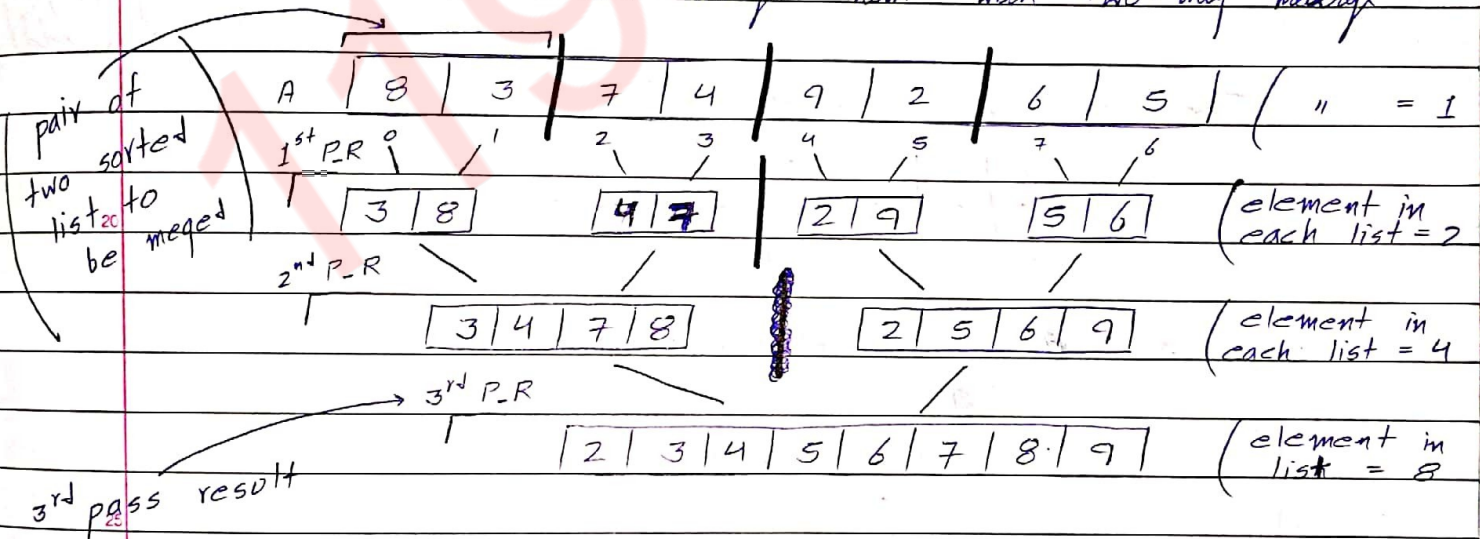
(b) two - way merging

⇒ we will select two sorted array at a time & merge them, this process will continue unless we get a single sorted array

Two - way iterative merge sort

A	8	3	7	4	9	2	6	5	(n = 8)
	0	1	2	3	4	5	6	7	

let A is array containg 8 sorted list (i.e. each element) & each list contain one single element, ∴ each list is sorted & we can merge them with two - way merge



elements merged in 1st pass = O(n)
2nd " = O(n)
3rd pass = O(n)

& merging is done for O(log n) times ($\log_2 8 = 3$)

∴ time complexity O(n log n)

```
void Imerge (int A[], int n)
{ int p, i, l, mid, k;

for (p = 2; p <= n; p = p * 2)          // pass
{
    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);
}
```

// for merging if n is odd

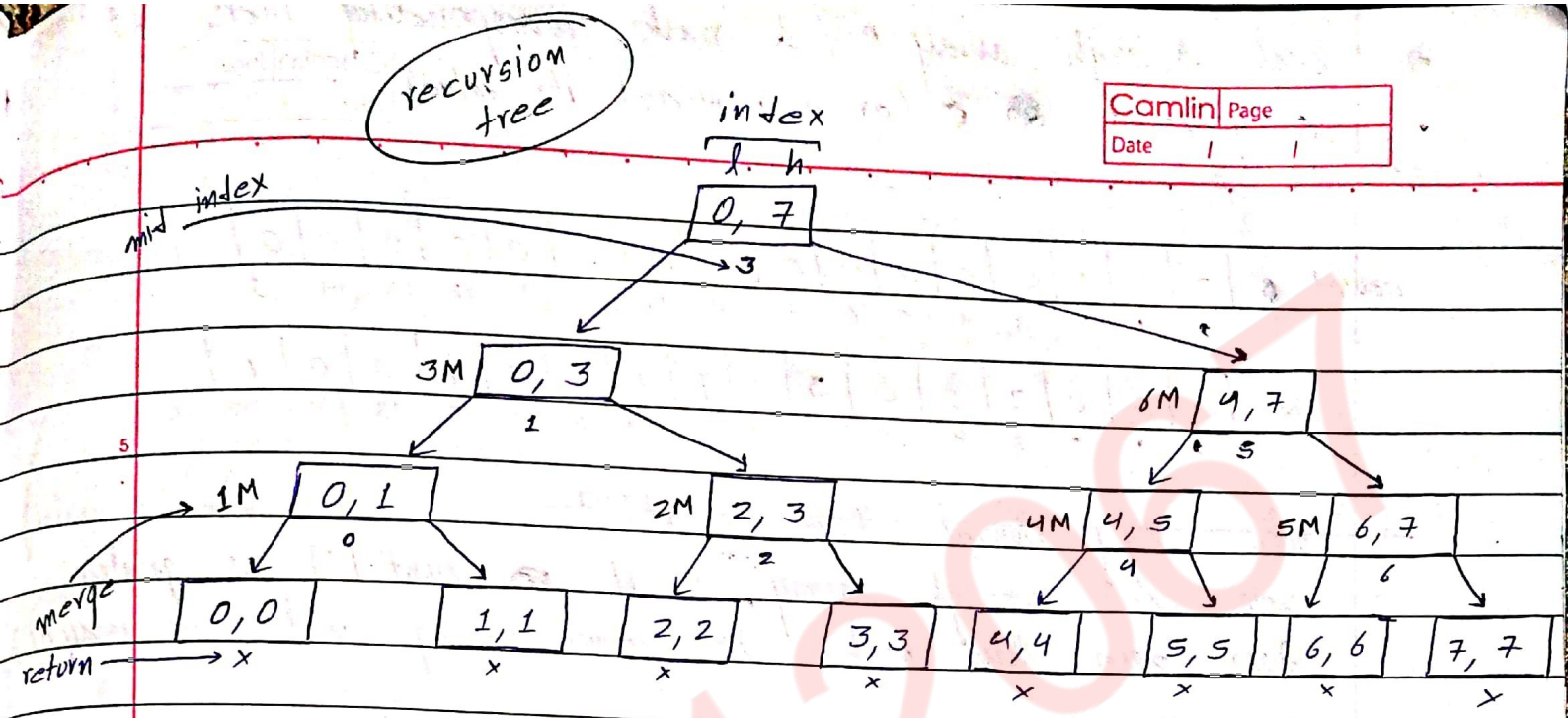
last element

Two way recursive merge sort

	l			mid				h	
A	8	2	9	6	5	3	7	4	
	0	1	2	3	4	5	6	7	

```
void mergesort (int A[], int l, int h)
{
    if (l < h)
    {
        mid = ⌊(l+h)/2⌋;
        mergesort (A, l, mid);
        mergesort (A, mid+1, h);
        merge (A, l, mid, h);
    }
}
```

merge is done in post order



no. of elements merged in each level = O(n) (expect last level)

merging is done for O(log n)

∴ time complexity = (O n log n)

space complexity

space for array A = n

" " auxilary array B = n

stack size required = log n

∴ total space = O(n + log n)

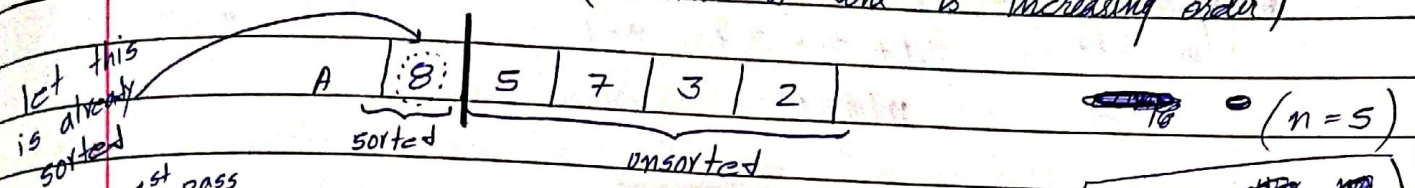
⇒ Out of comparison based sortings only merge sort require extra space

⇒ ~~shifting & comparison is done from back side~~

⇒ shifting & comp. is done from back side

Camlin Page
Date / /

Insertion sort (insertion is done is increasing order)

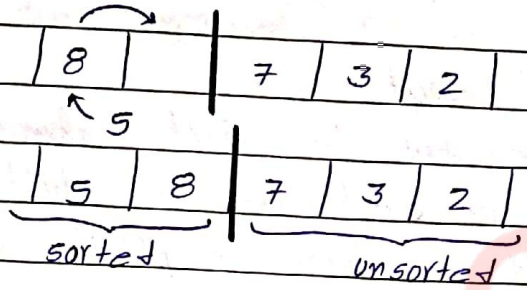


(n = 5)

1st pass

Inset (5)

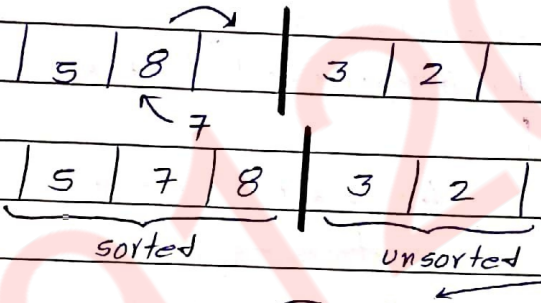
(smaller element in front)



comp. = 1
swap = 1
or shift

2nd pass

insert (7)

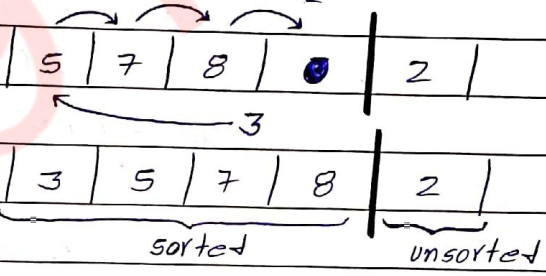


max. comp = 2
max. swap = 2
or shift

3rd pass

insert (3)

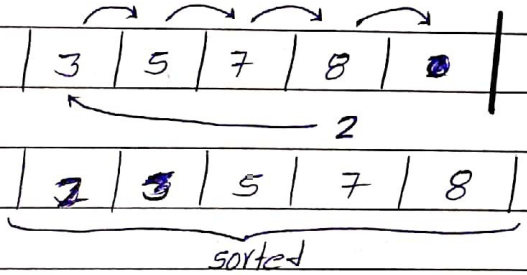
(shifting is done from back side)



3 comp. (max)
3 swap. (max)
or shift

4th pass

insert (2)



comp = 4
swap = 4
or shift

⇒ no. of pass = 4 ≃ n − 1 (for n = 5, for n = n)

⇒ no. of comp. = 1 + 2 + 3 + 4

1 + 2 + 3 + ... + n − 1 (n = 5, n = n)

≃ $\frac{n(n-1)}{2}$ ≃ $O(n^2)$

e.g. | 6 | 8 | 12 | 14 | 18 | 20 | |
j i

Camlin Page
Date / /

no. of swap or shift = $1+2+3+4$ $(n=5)$

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

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

$\approx O(n^2)$

⇒ 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 comptiable with LL than array

```
void insertionsort (int A[], int n)
{
    for (i = 1 ; i < n ; i++)          // pass
    {
        j = i - 1 ;
        x = A[i] ;
        while (j > -1 && A[j] > x)     // shift or comp.
        {
            A[j+1] = A[j] ;
            j-- ;
        }
        A[j+1] = x
    }
}
```

space complexity ≈ $O(1)$

∴ time complexity ≈ $O(n^2)$

⇒ Adaptive

| A | 2 | 5 | 8 | 10 | 12 |

already sorted

insert (5)

| | 2 | | 8 | 10 | 12 |

5

| 2 | 5 | 8 | 10 | 12 |

⋮

Camlin Page
Date / /

no. of comp ≈ n-1 ≈ O(n)
no. of shift ≈ 0
∴ time complexity ≈ O(n)

⇒ ∴ insertion sort is adaptive by nature, we don't have used flag here to make it adaptive

time required for insertion sort
max ≈ $O(n^2)$
min ≈ O(n)

⇒ if list is sorted in asecending order (best case)
(time) comp. ≈ O(n)
shift ≈ O(1)

⇒ if list is sorted is decending order (wrost case)
shift ≈ $O(n^2)$
(time complexity) comp. ≈ $O(n^2)$

⇒ stable

A | 3 | 5 | 8 | 10 | 12 | 5 |

insert (5) | 3 | 5 | 8 | 10 | 12 | | 5

| 3 | 5 | 5 | 8 | 10 | 12 |

∴ insertion sort is stable

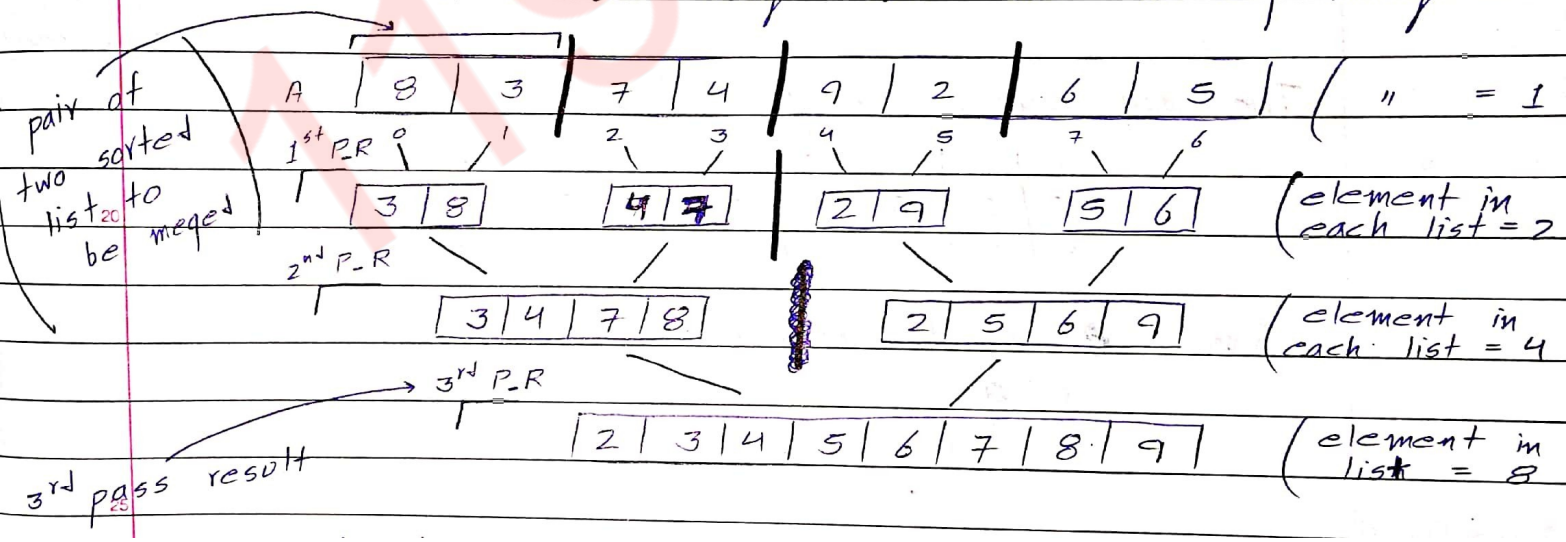
Q3 two - way merging

⇒ we will select two sorted array at a time & merge them, this process will continue unless we get a single sorted array

Two-way iterative merge sort

A	8	3	7	4	9	2	6	5	(n = 8)
	0	1	2	3	4	5	6	7	

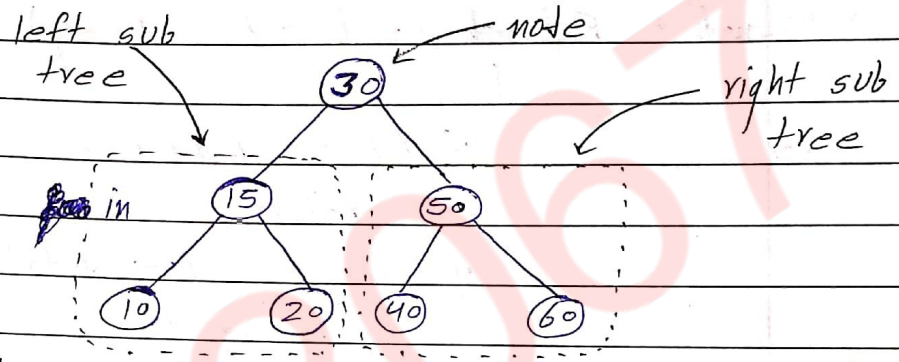
let A is array containg 8 sorted list (i.e. each element) & each list contain one single element, ∴ each list is sorted & we can merge them with two-way merge



So, we have used the above two-way merge sort, the only difference is, here we are handling the strings, in place of numbers.

To compare capital letters with small letters, we have compared capital letters as small letters by adding 32 to each capital letter when encountered in the string.

Q4 BST intro



it is a binary tree in which for every node all the elements on its left sub tree are smaller than element on that node & all the elements on its right sub tree are greater than element on that node.

⇒ it will not have duplicate elements

⇒ inorder traversal give elements in sorted order

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

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

```
node * Rinsert (node *p, int key)
    {
      node * t;
      if (p == NULL)
        {
      t. = malloc ( );
      t → data = key;
      t → lchild = t → rchild = NULL;
        return t;
        }
      if (key < p → data)
            p → lchild = insert (p → lchild, key);
      else if (key > p → data)
            p → rchild = insert(p → rchild, key);
      return p;
    }
int  main ( )
    {
      node * root = NULL;
      root = insert (root, 30);
             insert (root, 20);
               :
    }
```

For this specific func we have created dynamic array by using malloc to store name during runtime.
And then we compare each letter of name to decide weather it will be left or right child of the root name.