Date.....
Page.....

DSA LAB-9

Theory

1)A. **Linear search:**

Linear search is a very simple search algorithm. In this type of search, a sequential search is made overall the items one by one. Every item is checked if a match is found then the particular item is returned, otherwise search will continue till the end of data collection.

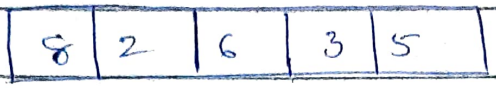
To implement Linear Search:

1. Traverse the array using for loop.
2. In every iteration, compare the target value with the current value of the array.
   - If the values match, return the current index of the array.
   - If the values do not match, move on to the next array element
3. If no match is found, then return -1.

Date.....
Page.....

For example:

Consider an array



If we want to find 5 in the array. Linear search will go step by step in a sequential order starting from first element 8. If the value is not equal, then next element, so when it reaches number 5 the values get equal and it returns 5. else -1.

Date.....
Page.....

Algorithm:

Linear_Search (Array A, N, VAL)

Step 1: Set i = 0
Step 2: Repeat step 3 while i < N
Step 3: If A[i] = VAL then
PRINT A[i]
Go to step 5
[END OF IF]
[END OF LOOP]
Step 4: PRINT " value not present in the array".
Step 5: EXIT

Date.....
Page.....

Binary Search:

Binary search is a fast search algorithm with run-time complexity of o(logn). This search algorithm works on principle of divide and conquer.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is search in sub-array to the left of the middle item. otherwise, the item is searched for is in the sub-array to the right of middle item. This process continues on the sub-array as well until the size of subarray reduces to zero.

Example:

For a binary search to work, it is mandatory for target array to be sorted. (We'll take an example array and we need to search 31 using binary search.

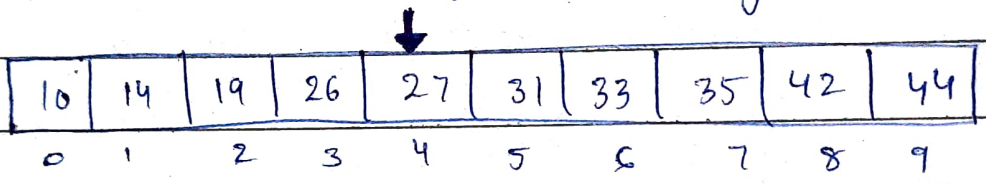
10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

First, we shall determine half of the array by using the formula

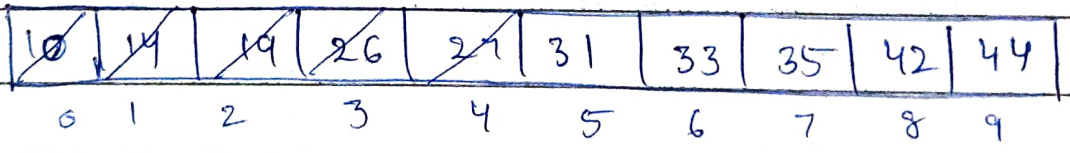
mid = low + (high - low)/2

Here, 0 + (9 - 0)/2 = 4 (integer value of 4.5)

so, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. we find 27 is not a match and less than 31. So, 31 must be in upper portion of array.

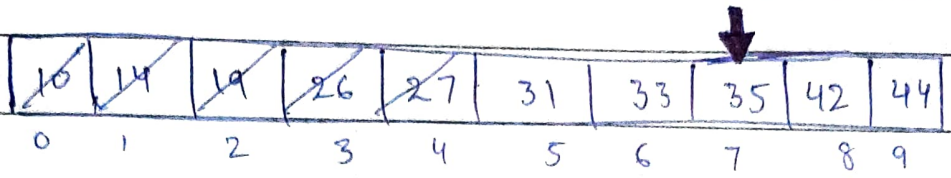


we change our low to mid+1 and find the new mid value again.

low = mid + 1

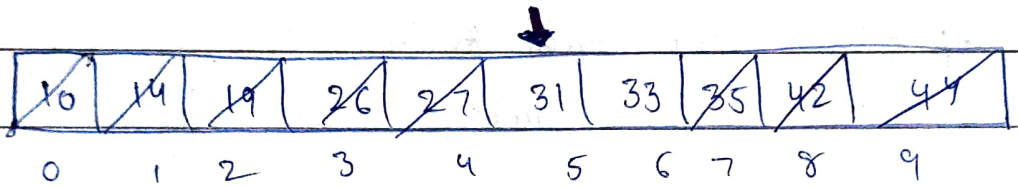
mid = low + (high - low)/2

Now, new mid is 7. we compare value at 7 to 31.



The value stored at location 7 is not a match and greater than 31. So, 31 must be in lower portion.

Now, we calculate mid again. This time it is 5.



Now, it is 31 at position 5. This is the match to what we are searching for. Thus, searching a item is done using Binary search.

Date.....
Page.....

Algorithm:

Binary_search (Array A, lower_bound, upper_bound, Val, pos)

Step 1: set beg = lower_bound, end = upper_bound, pos = -1

Step 2: Repeat step 3 and step 4 while beg <= end

Step 3: set mid = (beg + end) / 2

Step 4: if A[mid] = val then

pos = mid

PRINT POS

Go to step 6.

IF A[mid] > val then

set end = mid - 1

else

set beg = Mid + 1

[END OF IF]

[END OF LOOP]

Step 5: IF POS = -1, then

PRINT " Val is not present in array"

[END OF IF]

Step 6: EXIT.

Date .....
Page .....

2)A. Bubble sort:

Bubble sort is simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

Example: Let us take an array to be sorted.

(5 1 4 2 8)

First pass:

(5 1 4 2 8) → (1 5 4 2 8) Swap Since 5 > 1

(1 5 4 2 8) → (1 4 5 2 8) Swap Since 5 > 4

(1 4 5 2 8) → (1 4 2 5 8) Swap Since 5 > 2

(1 4 2 5 8) → (1 4 2 5 8) Don't swap Since 5 < 8

Second pass:

(1 4 2 5 8) → (1 4 2 5 8) No Swap Since 1 < 4

(1 4 2 5 8) → (1 2 4 5 8) Swap Since 4 > 2

(1 2 4 5 8) → (1 2 4 5 8) No swap Since 4 < 5

(1 2 4 5 8) → (1 2 4 5 8) No Swap Since 5 < 8

The array is sorted but algorithm does not know this, so it continues to proceed.

Third pass:

(1 2 4 5 8) ⟶ (1 2 4 5 8) No swap since 1<2

(1 2 4 5 8) ⟶ (1 2 4 5 8) No swap since 2<4

(1 2 4 5 8) ⟶ (1 2 4 5 8) No swap since 4<5

(1 2 4 5 8) ⟶ (1 2 4 5 8) No swap since 5<8

Algorithm:

Bubble_Sort (Array A, N)

Step 1: Repeat step 2 for I = 0 to N-1

Step 2: Repeat for J = 0 to N-I

Step 3: If A[J] > A[J+1], then
Swap A[J] and A[J+1]
[END OF INNER LOOP]
[END OF OUTER LOOP]

step 4: EXIT

Date .....
Page .....

Insertion Sort:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and unsorted part. Values from the unsorted part are picked and placed at the current position in the sorted part.

Example:

Let us take an array 12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 4 (last element of the array) 'i = 1.

Since 11 is smaller than 12, move 12 and insert 11 before 12.

11, 12, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0, . . . i-1] are smaller than 13.

11, 12, 13, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12, 13 → Array is sorted.

Algorithm:

Insertion_sort (Array A, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N
Step 2: Set temp = A[K]
Step 3: set J = K - 1
step 4: Repeat while temp <= A[J]
set A[J+1] = A[J]
Set J = J - 1
[END OF INNER LOOP]
Step 5: Set A[J+1] = temp
[END OF LOOP]
Step 6: EXIT

Date.....
Page.....

Selection sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Example:

Let us take an array arr[]=64, 25, 12, 22, 11

Now, selection sort finds

minimum element in arr[0,...,4] and place it at beginning

11 25 12 22 64

Find minimum element in arr[1,...,4] and place it at beginning of arr[1,...4]

11 12 25 22 64

Find the minimum element in arr [2,---4] and place it at beginning of arr [2,--4]

11 12 22 25 64

Find the minimum element in arr [3,--4] and place it at begginning of arr [3,--4]

11 12 22 25 64

Algorithm:

Selection_sort to sort an array arr with n elements

Step 1 : Repeat steps 2 and 3 for K=1 to n-1
Step 2 : Call Smallest (arr, K, n, POS)
Step 3 : Swap A[K] with Arr[POS]
[END OF LOOP]
Step 4 : EXIT

Smallest (Arr, K, N, POS)
Step 1 : [Intiliaze] set Small = Arr[K]
Step 2 : Set POS=K
Step 3 : Repeat for J = K+1 to N

Date .....
Page .....

If Small > Arr[J], then
Set Small = Arr[J]
Set POS = J
[END OF IF]
[END OF LOOP]
Step 4: EXIT

Merge sort:

Merge sort is a sorting technique based on divide and conquer technique. with worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

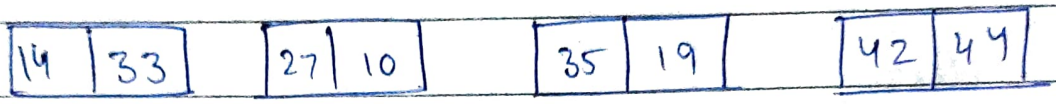
Example:

Let us take an array

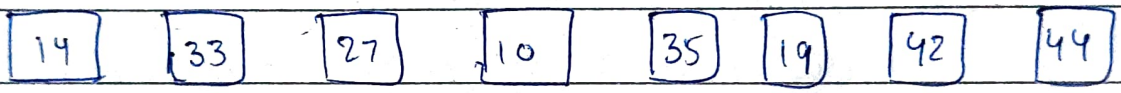
14	33	27	10	35	19	42	44

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. we see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearence of items in the original. Now we divide these two arrays into halves.



Now, we further divide these arrays and we achieve atomic value which can no more be divided.



Now, we combine them in exactly the same manner as they were broken.

we first compare the element for each list and combine them into another list in a sorted manner. we see that 14 and 33 are in sorted positions. we compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. we change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

14	33		10	27		19	35		42	44

In the next iteration of combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

10	14	27	33		19	35	42	44

After final merging, the list is a sorted array

10	14	19	27	33	35	42	44

Algorithm:

Merge_Sort (Arr, Beg, end)

Step 1: If beg < end, then
set mid = (beg + end) / 2
call merge_sort (Arr, beg, mid)
Call merge_sort (Arr, mid+1, end)
MERGE (Arr, beg, ~~end~~ mid, end)
[END OF IF]

Step 2: END

Date.....
Page.....

MERGE (Arr, beg, mid, end)

Step 1: Set i = beg, j = mid + 1, index = 0
Step 2: Repeat while (i <= mid) and (j <= end)
IF Arr[i] < Arr[j] then
Set temp [INDEX] = Arr[i]
Set i = i + 1
ELSE
Set temp [index] = Arr [j]
Set j = j + 1
[END OF IF]
set index = index + 1
[END OF LOOP]

Step 3: [copy the remaining elements of right sub-array, if any]

IF > mid, then
Repeat while j <= end
set temp [index] = Arr [j]
set index = index + 1
set j = j + 1
[END OF LOOP]

[copy remaining elements of sub-array, if any]

ELSE

Repeat while i <= mid

Set temp [index] = Arr [i]

set index = index + 1

Set i = i + 1

[END OF LOOP]

[END OF IF]

Step 4 : [copy the contents of temp back to Arr]

Set K = 0

Step 5 : Repeat while K < index

a. Set Arr [K] = temp [K]

b. Set K = K + 1

[END OF LOOP]

Step 6 : EXIT

Date.....
Page.....

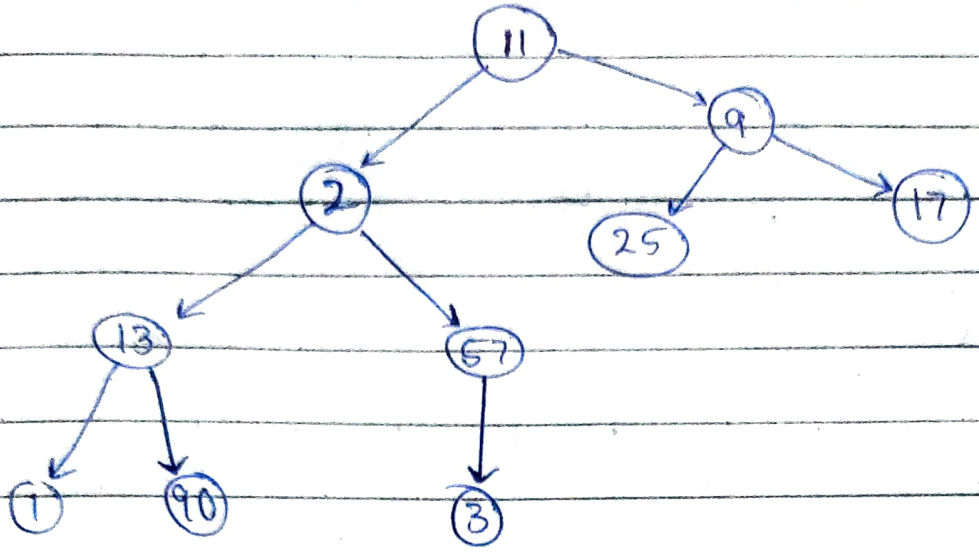
Heap sort:

Heap sort is a comparison-based sorting technique based on Binary heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for remaining elements.

Example:

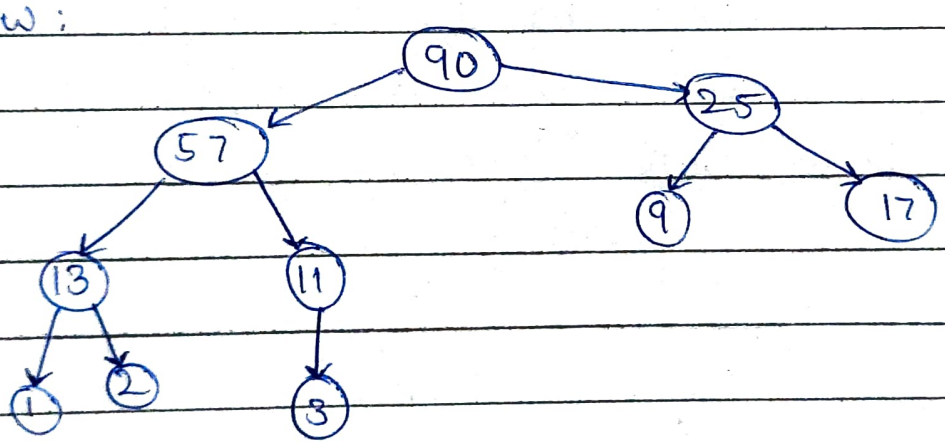
Let us take an array 11, 2, 9, 13, 57, 25, 17, 1, 90, 3.

The first step now is create heap from array ~~from~~ elements. For this consider a binary tree that can built from array elements the zeroth element would be root element and the left and right child of any element would be valued at, $2*i+1$, and $2*i+2^{th}$ index respectively.



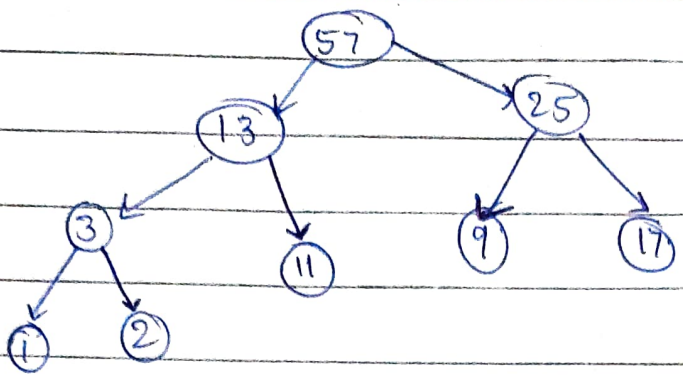
11	2	9	13	57	25	17	1	90	3

The above diagram depicts binary tree of the array we build heap from above binary tree as shown below:



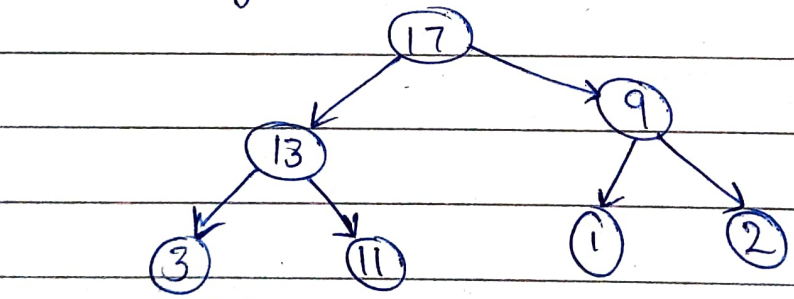
90	57	25	13	11	9	17	1	2	3

Now 90 is moved to last location by exchanging it with 3. Finally, 90 is eliminated from the heap by reducing maximum index value of the array by 1. The balance elements are then rearranged into the heap. The heap and array look like as shown in below:



57	13	25	3	11	9	17	1	2	90

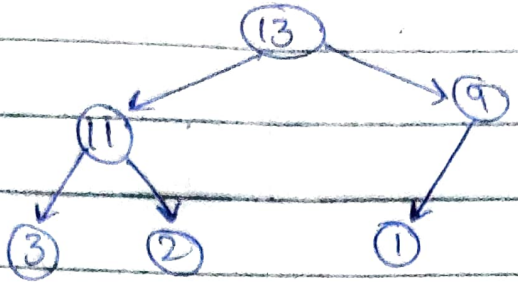
Similarly when the current root is 57 is exchanged with 2, and eliminated from the heap by reducing the maximum index valve of the array by 1. The balance elements are then rearranged into the heap. The heap and array look like as shown in below.



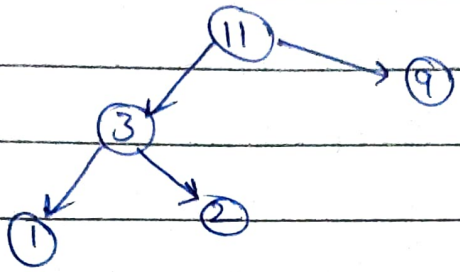
17	13	9	3	11	1	2	25	57	90

Following the same approach, the following phases are followed and untill fully sorted array is achieved.

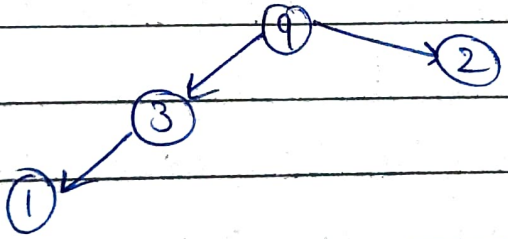
Date.....
Page.....



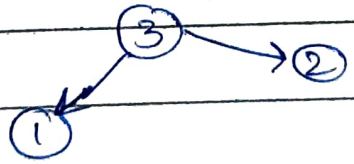
13	11	9	3	2	1	17	25	57	90



11	3	9	1	2	13	17	25	57	90



9	3	2	1	11	13	17	25	57	90

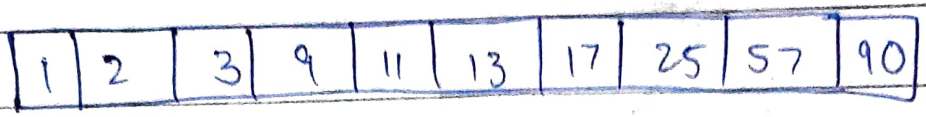


3	1	2	9	11	13	17	25	57	90

2	1	3	9	11	13	17	25	57	90

Date.....
Page.....

(1)



Algorithm:

Heap_Sort (Arr, N)

Step 1: [Build Heap H]
Repeat for i = 0 to N - 1
Call Insert_Heap (Arr, N, Arr[i])
[END OF LOOP]
Step 2: [Repeatedly delete the root element]
Repeat while N > 0
Call Delete_Heap (Arr, N, val)
Set N = N + 1
[END OF LOOP]
Step 3: EXIT

Date.....
Page.....

Quick sort:

Quick sort is a Divide and conquer algorithm It picks an element as pivot and partitions the given array around the picked pivot.

The key process is quicksort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x as its correct position in sorted array and put all smaller elements before x, and put all all greater elements after x, All this should be done in linear time.

Example:

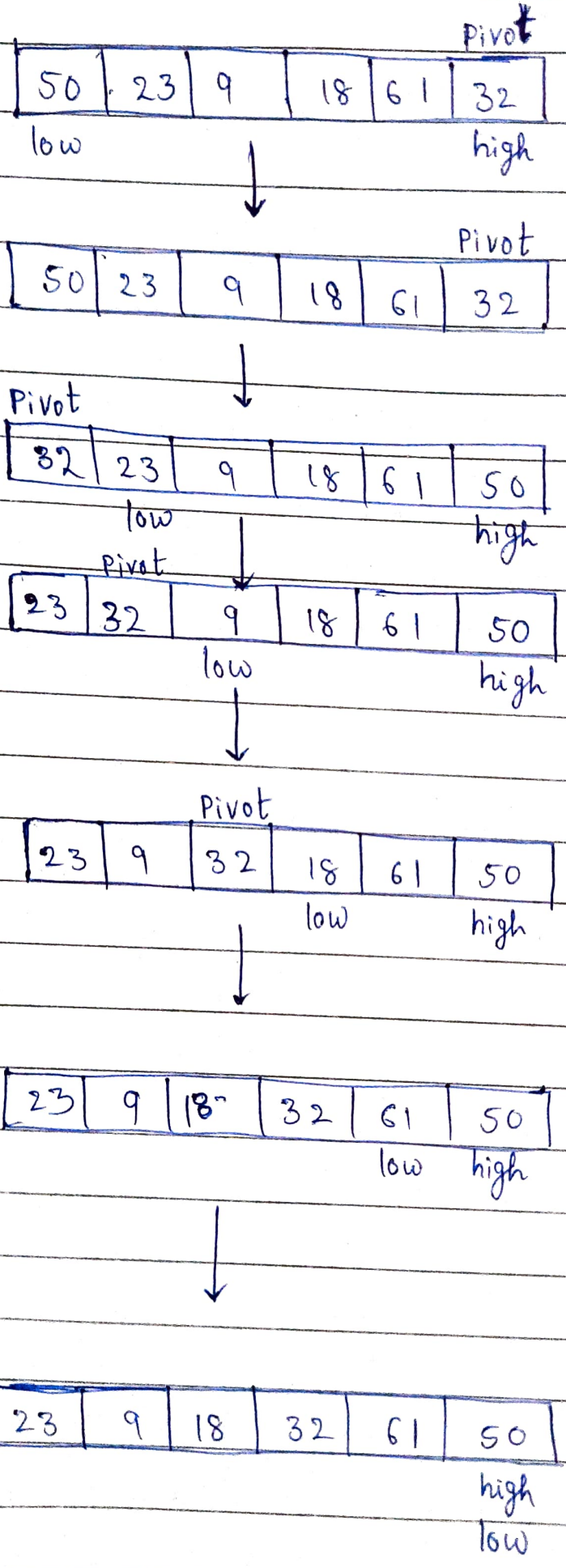
Let us take an array as emaple.

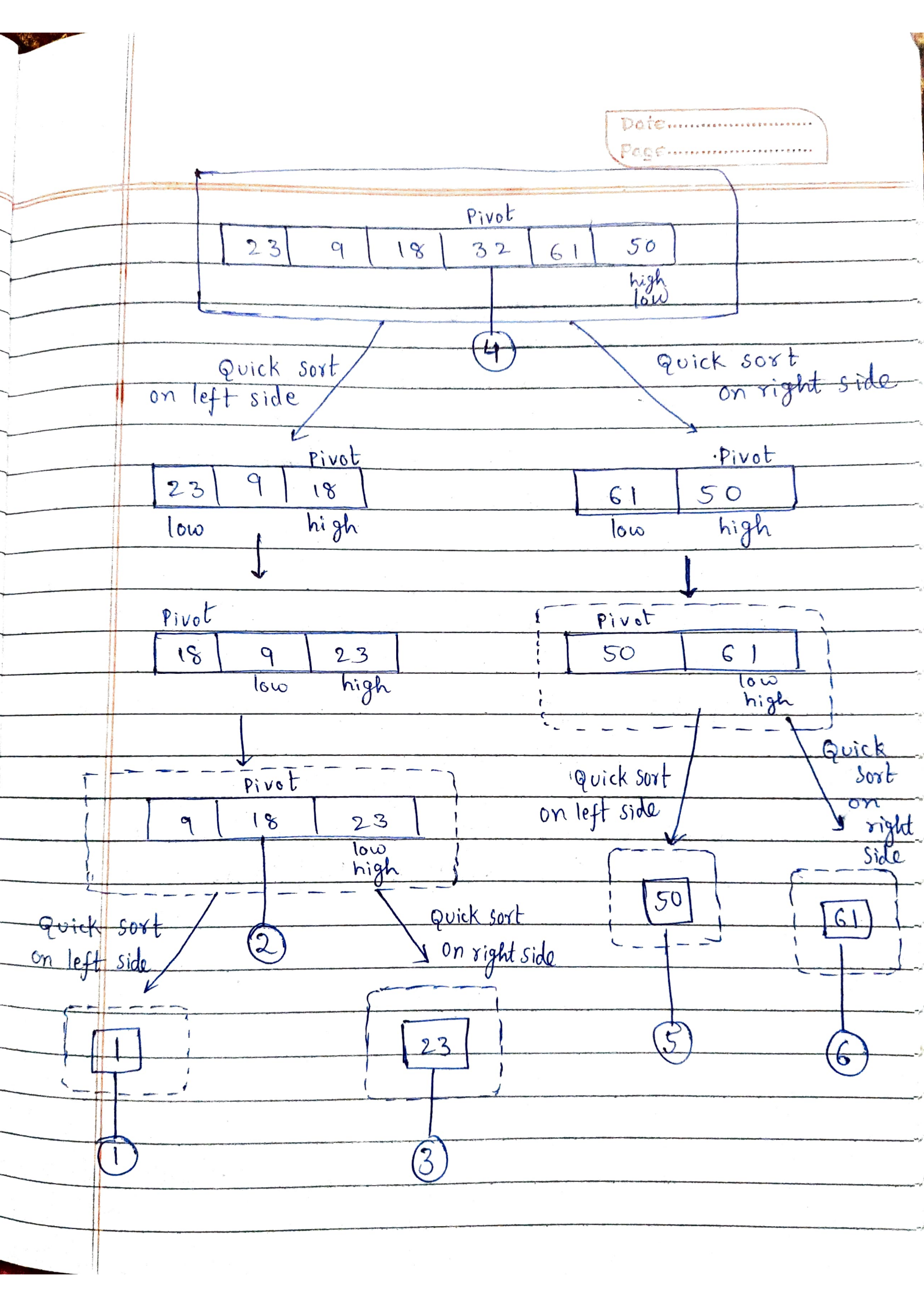
50, 23, 9, 18, 61, 32.

Select a pivot, for ease of code we ofttten select rightmost element as pivot.

set two pointers low and high to beginning and end of array respectively.

Move those depending upon less than or greater than pivot value until sorted ~~element~~ array is obtained as shown below:


Pivot
50 23 9 18 61 32
low
high
Pivot
50 23 9 18 61 32
Pivot
32 23 9 18 61 50
low
high
Pivot
23 32 9 18 61 50
low
high
Pivot
23 9 32 18 61 50
low
high
23 9 18 32 61 50
low
high
23 9 18 32 61 50
high
low


Pivot
23 9 18 32 61 50
high
low
4
Quick sort on left side
Quick sort on right side
Pivot
23 9 18
low
high
Pivot
61 50
low
high
Pivot
18 9 23
low
high
Pivot
50 61
low
high
Pivot
9 18 23
low
high
Quick sort on left side
Quick sort on right side
Quick sort on left side
Quick Sort on right Side
2
50
61
Quick sort on left side
Quick sort on right side
1
23
5
6
1
3

Date.....
Page.....

Algorithm:

Quick_Sort (Arr, beg, end)

Step 1: If (beg < end), then
CALL Partition (Arr, beg, end, loc)
Call quick_sort (Arr, beg, loc - 1)
Call quick_sort (Arr, loc+1, end)
[END OF IF]

Step 2: END

PARTITION (Arr, beg, end, loc)

Step 1: Set left = beg, right = end, loc = beg,
flag = 0

Step 2: Repeat steps 3 to while flag = 0

Step 3: Repeat while Arr[loc] <= Arr[right] and
loc != right
Set right = right - 1
[END OF LOOP]

Step 4: If loc == right then
set flag = 1
ELSE IF Arr[loc] > Arr[RIGHT] then

Swap Arr[loc] with Arr[right]
Set loc = Right
[END OF IF]
Step 5: IF flag = 0, then
Repeat while Arr[Loc] >= Arr[left]
and loc != left
Set left = left + 1
[END OF LOOP]
Step 6: IF Loc == left, then
set flag = 1
ELSE IF Arr[Loc] < Arr[Left], then
Swap Arr[loc] with Arr[left]
Set loc = Left
[END OF IF]
Step 7: [END OF LOOP]
Step 8: EXIT