**1.Calculate the complexity of the following code.**

for (int i=n; i>0; i=/2)

{

for(int j=0;j<i;j++)

{

count ++;

}

}

**Solution:**

For the given problem at each updating the value of n is being divided by 2, so

If

i =n loop executes n times

i =n/2 loop executes n/2 times

Now the series will be,

n+n/2+n/4+……….

Sum of given GP series S=a/(1-r) since r<1

a=n, r=1/2

S=n + n/2 + n/4 +……

=n(1 + ½ + ¼ + ……)

=n(1/(1 - ½ ))

=2n

Therefore complexity of the code is O(n).

**2**.**State the algorithm of linear search and analyze its computational complexity in three different cases.**

**Algorithm:**

Linear Search ( Array arr, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if arr[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

**Example:**

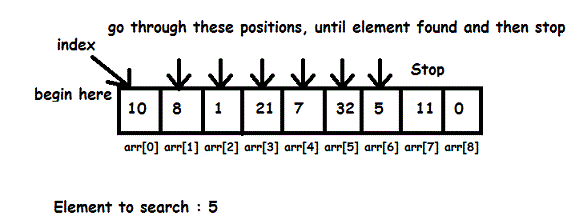
We take the number – 10 8 1 21 7 32 5 11 0

Number to be searched(x)– 5

Start from the leftmost element of arr[] and one by one compare x with each element of arr[]

If x matches with an element, return the index.

If x doesn’t match with any of elements, return -1.



Pass 1: 10🡪 NOT FOUND

Pass 2: 8🡪 NOT FOUND

Pass 3: 1🡪 NOT FOUND

Pass 4: 21🡪 NOT FOUND

Pass 5: 7🡪 NOT FOUND

Pass 6: 32🡪 NOT FOUND

Pass 7: 5🡪 FOUND

**Complexity Analysis :**

In the given problem the condition checks for the search element one by one and if satisfied then returns the location .

Pass1 = 1 + 1 + 1 + 1 + ….. n

= n

i.e. Complexity is O(n)

**Best Case Complexity :**

When the search element is present at the 0th location of the array i.e. a[0] then number of iterations will be O(1).

Example – 1 2 3 4 5

The number to be searched is 1.

Pass 1: 1 🡪 FOUND

**Average Case Complexity :**

When the search element is somewhere in between the 0th and nth element then the complexity will be O(n).

Example- 1 2 3 4 5

The number to be searched is 2.

Pass 1: 1🡪 NOT FOUND

Pass 2: 2🡪 FOUND

**Worst Case Complexity :**

When the search element is in the nthposition then the complexity will be O(n)

Example- 1 2 3 4 5

The number to be searched is 5.

Pass 1: 1🡪 NOT FOUND

Pass 2: 2🡪NOT FOUND

Pass 3: 3🡪 NOT FOUND

Pass 4: 4🡪 NOT FOUND

Pass 5: 5🡪 FOUND

**Remarks**

**3**.**State the algorithm of binary search and analyze its computational complexity in three different cases.**

**Algorithm:**

Step 1: Set BEG := LB,END := UB and MID = BEG+END/2

Step 2: Repeat Step 3 and 4 while BEG<= END and DATA[MID] not equal to ITEM

Step 3: if ITEM<DATA[MID], then

Set END := MID-1

Else

Set BEG := MID+1

[End of if structure]

Step 4: Set MID := INT((BEG+END)/2)

[End of step 2 loop]

Step 5: if DATA[MID]= ITEM then:

Set LOC := MID

Else

Set LOC := NULL

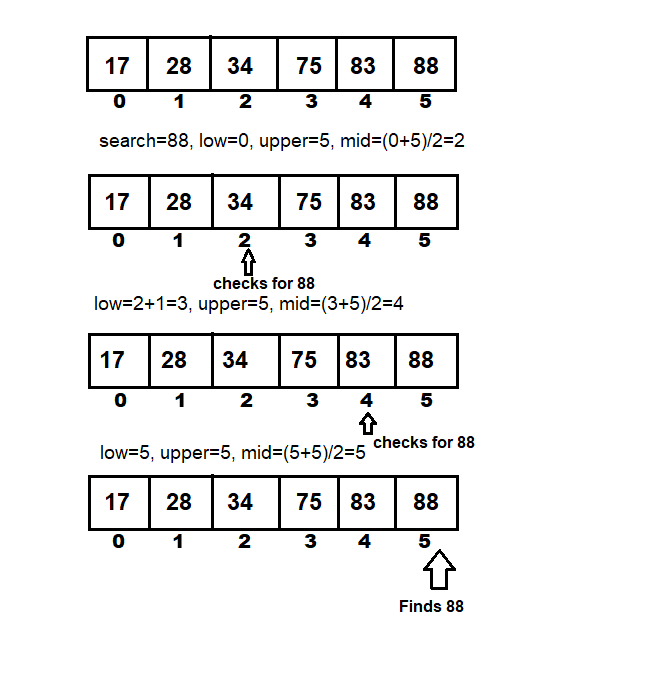
[End of if structure]

Step 6: Exit

**Example:**

The numbers are – 17 28 34 75 83 88

Number to search 88



**Complexity Analysis:**

If an array has n elements, then

n\*(1/2)k=1

or, n/2k=1

or,2k=n

or, k=log2(n)=O(log2n)

**Worst case complexity:**

The key is placed on nth or 0th position, so complexity isO(log2 n)

Example – 1 2 3 4 5

Search 5

Firstly,

Low=0 Up=4

Mid=2 i.e. 3 🡪 NOT FOUND

Secondly,

Low=3 Up=4

Mid=3 i.e. 4🡪 NOT FOUND

Thirdly,

Low=4 Up=4

Mid=4 i.e 5🡪 FOUND

**Best case complexity:**

The key is already placed on the mid position, so complexity is O(1)

Example- 1 2 3 4 5

Search 3

Firstly,

Low =0 Up=4

Mid=2 i.e. 3 🡪 FOUND

**Average case complexity:**

The key is placed near mid, so complexity is O(log2 n)

Example- 1 2 3 4 5

Search 4

Firstly,

Low=0 Up=4

Mid=2 i.e. 3 🡪 NOT FOUND

Secondly,

Low=3 Up=4

Mid=3 i.e. 4🡪 FOUND

**Remarks**

**4.****State the algorithm of bubble sort and analyze its computational complexity in three different cases.**

**Algorithm:**

|  |
| --- |
| Step 1: Repeat Steps 2 and 3 for i=1 to n    Step 2: Set j=1    Step 3: Repeat while j<=n              (A) if  a[i] < a[j]                  Then interchange a[i] and a[j]                  [End of if]              (B) Set j = j+1           [End of Inner Loop]         [End of Step 1 Outer Loop]    Step 4: Exit |
|  |

**Example:**

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required[.](https://adf.ly/KiLQs)

**First Pass:**  
( **5** **1** 4 2 8 ) \to ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) \to ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) \to ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) \to ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**  
( **1** **4** 2 5 8 ) \to ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) \to ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) \to ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) \to ( 1 2 4 **5** **8** )  
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass:**  
( **1** **2** 4 5 8 ) \to ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) \to ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) \to ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) \to ( 1 2 4 **5** **8** )

**Complexity Analysis:**

In the given problem(n-1) comparisons for pass 1,(n-2) in pass 2 upto n

(n-1) + (n-2) +……+ 2 + 1

Sum of series =(n(n-1))/2

= (n2 -n)/2

=O(n2)

**Best Case Complexity :**

Let us consider the elements

[1,2,3,4]

First pass :

1. The algorithm compares 1 and 2, 1<2 is true , so no swapping [1,2,3,4]
2. Now the algorithm compares the next 2 elements 2 and 3 , 2 < 3 is true therefore no swapping [1,2,3,4] .
3. Next it compares 3 and 4 , 3 < 4 is true so no swapping [1,2,3,4]
4. Sorted array obtained [1,2,3,4]

So, the best case complexity is O(n).

**Average Case Complexity :**

Let us consider the elements

[5,1,4,2]

First pass:

1. The algorithm compares 5 and 1 , 5<1 is not true so swapping occurs [1,5,4,2].
2. Next it compares 5 and 4 , 5 < 4 is false so swapping occur [1,4,5,2].
3. Next it compares 5 and 2 , 5<2 is false so swapping occurs [1,4,2,5]
4. Last element reaches its position and gets fixed .

Second pass:

1. Comparison is done between 1 and 4 , 1< 4 is true so no swapping [1,4,2,5]
2. Next it compares 4 and 2 , 4<2 is false so swaps [1,2,4,5]
3. Next compares 4 and 5 ,4<5 is true so no swapping occurs [1,2,4,5]
4. Sorted array obtained.

So, the average case complexity is O(n2)

**Worst Case Complexity :**

Let us consider the elements

[5,4,2,1]

First Pass

1. The algorithm compares 5 and 4 , 5<4 is false so swapping occurs [4,5,2,1]
2. Next it compares 5 and 2 , 5<2 is false so swaps [4,2,5,1]
3. Next it compares 5 and 1 , 5<1 is false so swaps [4,2,1,5]
4. Last elements get to its fixed position.

Second Pass

1. Comparison is done between 4 and 2 ,4<2 is false so swaps [2,4,1,5]
2. Next it compares 4 and 1 , 4<1 is false so it swaps [2,1,4,5]
3. Next it compares 4 and 5 , 4<5 is true so no swapping [2,1,4,5].
4. Second last element gets fixed [2,1,4,5].

Third Pass

1. Comparison is done between 2 and 1 ,2<1 is false so it swaps [1,2,4,5]
2. Next it compares 2 and 4, 2<4 , true so no swapping [1,2,4,5] .
3. Third Last element gets fixed [1,2,4,5] .
4. The array is sorted [1,2,4,5] .

So, the worst case complexity isO(n2)

**Remarks-**

**5. State the algorithm of selection sort and analyze its computational complexity in three different cases.**

**Algorithm:**

Step 1: Repeat step 2 and 3 for k=1,2,…n-1.

Step 2: Set min 🡨a[k] and loc 🡨k.

Step 3: Repeat for j=k+1,k+2…n

If min> a[j], then:

Set, min 🡨a[j] and loc🡨a[j] and loc🡨 j

[END OF LOOP]

Step 4: Interchange a[k] and a[loc]

Set temp🡨a[k], a[k]🡨a[loc],a[loc]🡨temp

Step 5: Exit

**Example:**

Consider numbers- 7 5 4 2



After sorting,

2 4 5 7

**Complexity analysis:**

For each i from 1 to n - 1, there is one exchange and n - i comparisons, so there is a total of n - 1 exchanges

(n − 1) + (n − 2) + ...+ 2 + 1 = n(n − 1)/2 comparisons. i.e O(n2)

**Best case Complexity:**

If the array is sorted in ascending order and we want to sort them in an ascending order i.e O(n2).

Example- 1 2 3 4

Pass 1: 1 2 3 4 [no change as 1 is smaller than all elements 4 is largest]

Pass 2: 1 2 3 4 [no change as now pointer shifts 2 is smaller than all elements 4 is largest]

Pass 1: 1 2 3 4 [no change as now pointer shifts 3 is smaller then all elements 4 is largest]

Sorted array found.

**Average case complexity:**

It implies that the running time of Selection sort is quite insensitive to the input i.e O(n2).

Example- 1 3 2 4

Pass 1: 1 3 2 4 [no change as 1 is smaller than all elements 4 is largest]

Pass 2: 1 2 3 4 [now pointer shifts and 2 is smaller than all element so it get interchanged]

Pass 1: 1 2 3 4 [no change as now pointer shifts 3 is smaller then all elements 4 is largest]

Sorted array found.

**Worst case complexity:**

If the array is already sorted in a descending order and we want to sort them in an ascending order. This could be quadratic. Therefore Big O: O(n2 ).

Example- 4 3 2 1

Pass 1: 1 3 2 4 [1 is smaller than all elements 4 is largest so it get interchanged]

Pass 2: 1 2 3 4 [now pointer shifts and 2 is smaller than all element so it get interchanged]

Pass 1: 1 2 3 4 [no change as now pointer shifts 3 is smaller then all elements 4 is largest]

Sorted array found.

**Remarks**

**6.State the algorithm of insertion sort and analyze its computational complexity in three different cases.**

**Algorithm:**

Step 1: Set a[0]🡨infinity

Step 2: Repeat step 2 to 5 for k=2,3,….N.

Step 3: Set temp🡨a[k] and ptr🡨k-1

Step 4: Repeat while temp<a[ptr]

1. Set a[ptr + 1]= a[ptr]
2. Set ptr🡨ptr-1

[End of loop]

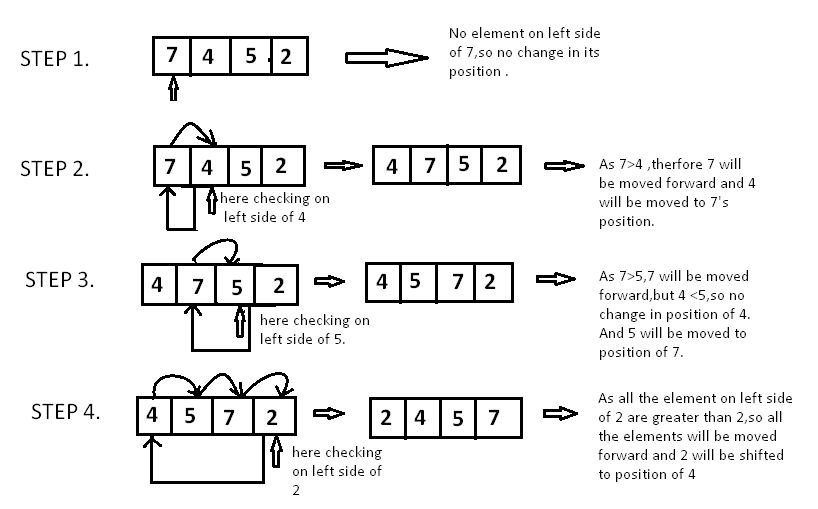
Step 5: Set a[ptr + 1]🡨temp [insert in proper place]

Step 6: Return

**Example:**

Let the numbers be : 7 4 5 2

After sorting: 2 4 5 7



**Complexity analysis:**

The f(n) of comparison of insertion sort algorithm is follow as

f(n)= 1 + 2 +…….+ ( n-1 )

= n(n-1)/2 = O(n2 ).

**Best case Complexity:**

If the array is sorted in ascending order and we want to sort them in an ascending order i.e O(n2).

Example- 1 2 3 4

Pass 1: 1 2 3 4 [no change as 1 is smaller than all elements 4 is largest]

Pass 2: 1 2 3 4 [no change as now pointer shifts 2 is smaller than all elements 4 is largest]

Pass 1: 1 2 3 4 [no change as now pointer shifts 3 is smaller then all elements 4 is largest]

Sorted array found.

**Average case complexity:**

It implies that the running time of Selection sort is quite insensitive to the input i.e O(n2).

Example- 1 3 2 4

Pass 1: 1 3 2 4 [no change as 1 is smaller than all elements 4 is largest]

Pass 2: 1 3 2 4 [as 1 is smaller than 3]

Pass 1: 1 2 3 4 [2 is smaller than 3 so it will interchange with 3 elements 4 is largest]

Sorted array found.

**Worst case complexity:**

If the array is already sorted in a descending order and we want to sort them in an ascending order. This could be quadratic. Therefore Big O: O(n2 ).

Example- 4 3 2 1

Pass 1: 3 4 2 1[3 is smaller than 4 so it get interchanged]

Pass 2: 2 3 4 1 [now pointer shifts and 2 is smaller than all element so it get interchanged]

Pass 1: 1 2 3 4 [1 is smaller than all elements 4 is largest so it get interchanged]

Sorted array found.

**Remarks**

**7.State the algorithm of merge sort and analyze its computational complexity in three different cases.**

**Algorithm:**

Step 1: Declare mergesort( a[] )

Check if ( n == 1 ) then return a

Set:= l1[] = a[0] ... a[n/2] and l2[] = a[n/2+1] ... a[n]

Declare l1 = mergesort( l1 ) and l2 = mergesort( l2 )

return merge( l1, l2 )

Step 2: Declare merge( a[], b[] )

Declare c[] and Loop :

Step 3:Loop:

while ( a and b have elements )

check if ( a[0] > b[0] )

add b[0] to the end of c and remove b[0] from b

else

add a[0] to the end of c and remove a[0] from a

[end if]

[end while]

Step 4:Loop: while ( a has elements )

add a[0] to the end of c

remove a[0] from a

[ end while]

Step 5: Loop: while ( b has elements )

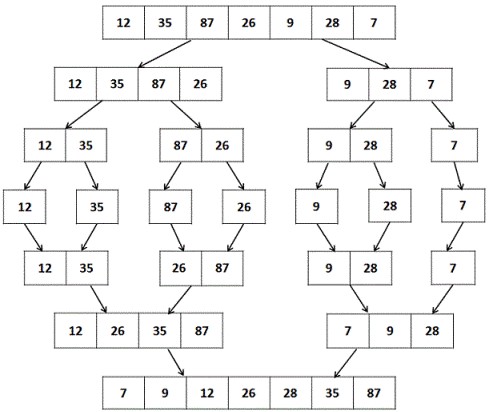
add b[0] to the end of c andremove b[0] from b

[end while]

return c

**Example:**  The numbers: 12 35 87 26 9 28

After sorting : 7 9 12 26 28 35 87



**Complexity Analysis:**

Let us consider, the running time of Merge-Sort as ***T(n)***. Hence,

T(n)={c2xT(n2)+dxnifn⩽1otherwiseT(n)={cifn⩽12xT(n2)+dxnotherwise where *c* and *d* are constants

Therefore, using this recurrence relation,

T(n)=2iT(n2i)+i.d.nT(n)=2iT(n2i)+i.d.n

As, i=logn,T(n)=2lognT(n2logn)+logn.d.ni=logn,T(n)=2lognT(n2logn)+logn.d.n

=c.n+d.n.logn=c.n+d.n.logn

Therefore, T(n)=O(nlogn)

**Best case complexity:**

If the array is sorted in ascending order and we want to sort them in an ascending order i.e O(nlogn).

Example: 1 2 3 4

/ \

1 2 3 4

/ \

1 2 3 4

\ /

1 2 3 4

**Average case complexity:**

If the array is jumbled up with high and low numbers i.e O(nlogn).

Example: 1 3 4 2

/ \

1 3 4 2

/ \

1 3 2 4

\ /

1 2 3 4

**Best case complexity:**

If the array is in descending order and we want to sort them in an ascending order i.e O(nlogn).

Example: 4 3 2 1

/ \

4 3 2 1

/ \

3 4 1 2

\ /

1 2 3 4

**Remarks**

**8.State the algorithm of quick sort and analyze its computational complexity in three different cases.**

**Algorithm:**

**Partition algorithm:**

Step 1: declare partitionFunc(left, right, pivot)

leftPointer = left and rightPointer = right - 1

Step 2: Loop :while True do

while A[++leftPointer] < pivot do

//do-nothing

[end while]

Step 3: Loop: while rightPointer > 0 && A[--rightPointer] > pivot do

//do-nothing

[end while]

Step 4: if leftPointer >= rightPointer

break

else

swap leftPointer,rightPointer

[end if]

[end while]

Step 5: swap leftPointer,right

return leftPointer

end function

**Sorted algorithm:**

Step 1:quickSort(left, right)

if right-left <= 0

return

else

pivot = A[right]

partition = partitionFunc(left, right, pivot)

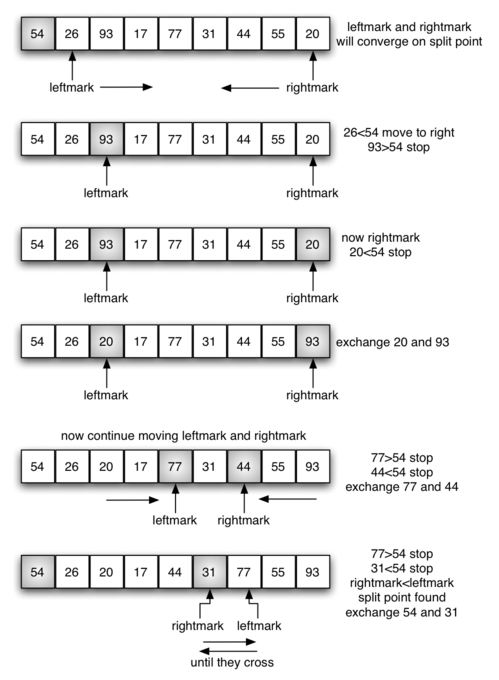
Step 2: Set =: quickSort(left,partition-1)

quickSort(partition+1,right)

[end if]

Step 3: End procedure.

**Example:**



**Complexity analysis:**

Time taken by QuickSort in general can be written as following.

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

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.  
The time taken by QuickSort depends upon the input array and partition strategy.

**Worst case analysis:**

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

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

which is equivalent to

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

The solution of above recurrence is O (n2).

Example: 1 2 3 4

Pivot =4

**Best case complexity:**

The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

T(n) = 2T(n/2) + (n)

The solution of above recurrence is O (nLogn). It can be solved using case 2 of masters theorem.

Example: 1 3 2 4

Pivot= 4

**Average case complexity:**  
To do average case analysis,we can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

T(n) = T(n/9) + T(9n/10) + (n)

Solution of above recurrence is also O(nLogn)

Example: 4 3 2 1

Pivot= 1

**Remarks**

**9.State the algorithm of heap sort and analyze its computational complexity in three different cases.**

**Algorithm:**

**heapify(array, size)**

Step 1: Begin

    for i := 1 to size do

      node := i

      par := floor (node / 2)

Step 2: loop: while par >= 1 do

         if array[par] < array[node] then

            swap array[par] with array[node]

          node := par

          par := floor (node / 2)

      [end of if]

   [end of loop]

Step 3: End

**heapSort(array, size)**

Step 1:Begin

 Step 2:loop:  for i := n to 1 decrease by 1 do

      heapify(array, i)

      swap array[1] with array[i]

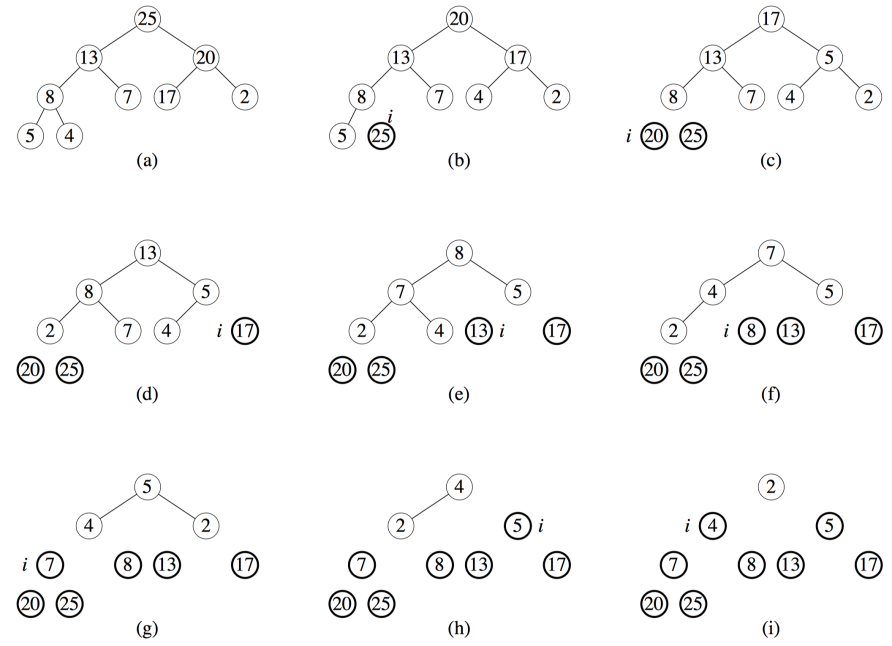
[end of loop]

Step 3: Exit

**Example:** let the numbers be 25 13 20 8 7 17 2 5 4

The MAXHEAP created the number is sorted as follow:

2 4 5 7 8 13 17 20 25



**Complexity analysis:**

Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

T(n) = O(n) + n \* O(log n) = O(n \* log n)

**Best case complexity:**

If the array is sorted in ascending order and we want to sort them in an ascending order i.e O(nlogn).

Example: 1 2 3 4 5. Here we use minheap.

First pass: 1 will be in its position.

Second pass: 1 and 2 will be in its position.

Third pass: 1,2 and 3 will be in its position.

Fourth pass: 1,2, 3 and 4 will be in its position.

Fifth pass: 1,2,3,4,5 will be in its position.

**Average case complexity:**

If the array is jumbled up with high and low numbers i.e O(nlogn).

Its heapify process will be in most steps

Example: 1 3 4 2 5 Here we use minheap.

First pass: 1 will be in its position.

Second pass: 1 and 2 will be in its position.

Third pass: 1,2 and 3 will be in its position.

Fourth pass: 1,2, 3 and 4 will be in its position.

Fifth pass: 1,2,3,4,5 will be in its position.

**Worst case complexity:**

If the array is in descending order and we want to sort them in an ascending order i.e O(nlogn).

Its heapify process will be in each and every steps

Example: 5 4 3 2 1 Here we use minheap.

First pass: 1 will be in its position.

Second pass: 1 and 2 will be in its position.

Third pass: 1,2 and 3 will be in its position.

Fourth pass: 1,2, 3 and 4 will be in its position.

Fifth pass: 1,2,3,4,5 will be in its position.

**Remarks**

**10.State the algorithm of shell sort and analyze its computational complexity in three different cases.**

**Algorithm:**

Step 1: Declare shellSort()

while interval < A.length /3 do:

interval = interval \* 3 + 1

end while

Step 2:loop: while interval > 0 do:

for outer = interval; outer < A.length; outer ++ do:

valueToInsert = A[outer]

inner = outer;

while inner > interval -1 && A[inner - interval] >= valueToInsert do:

A[inner] = A[inner - interval]

inner = inner - interval

[ end while]

A[inner] = valueToInsert

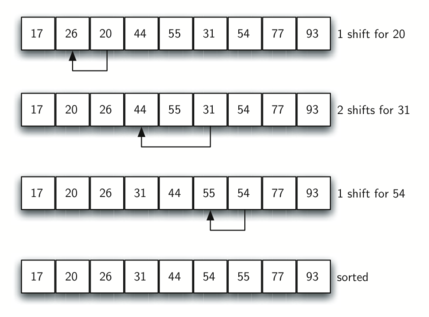
[end for]

interval = (interval -1) /3;

[end while]

Step 3: End

**Example:**



**Complexity analysis:**

Time complexity of Shell Sort depends on gap sequence . Its general time complexity is O(n\* log2n). Time complexity of Shell sort is generally assumed to be near to O(n) and less than O(n2).

Shell’s original sequence: N/2 , N/4 , …, 1 (repeatedly divide by 2);

Hibbard’s increments: 1, 3, 7, …, 2k – 1 ;

Knuth’s increments: 1, 4, 13, …, (3k – 1) / 2 ;

Sedgewick’s increments: 1, 5, 19, 41, 109, ….

**Best case complexity:**

If the array is sorted in ascending order and we want to sort them in an ascending order i.e O(nlogn).

Example: 1 2 3 4 5.

First pass: All the elements are in right position

**Average case complexity:**

If the array is jumbled up with high and low numbers i.e O(n\* log2n)

Example: 1 3 4 2 5

First pass: 1 2 4 3 5

Second pass: 1 2 3 4 5

**Worst case complexity:**

If the array is in descending order and we want to sort them in an ascending order i.e O(n\* log2n)

Example: 5 4 3 2 1

First pass: 1 4 3 2 5

Second pass:1 2 3 4 5

Now list sorted.

**Remarks**