Sorting

- **Sorting is the process of arranging a set of data elements in some logical order. This logical order may be ascending or descending in case of numeric values, or dictionary order in case of alphanumeric values.**
- The biggest advantage of sorting is that searching and retrieval can happen much faster.
- Following are some of the examples of sorting in real-life scenarios:
  - **Telephone Directory** − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
  - **Dictionary** − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.
  - **Ranks based on scores** – such as list of position holders of an examination.

Sorting Approaches:

- **Internal sorting**
- **External sorting**

Internal Sorting:

In internal sorting, all the data to be sorted is held in the main memory of the computer and sorted using an internal sorting method.

External Sorting:

If the data to be sorted is so large that it can't be accommodated completely in the main memory of the computer all at a time then external sorting approach is used. It deals with sorting the data stored in data files.

Internal Sorting Methods: We will learn the following internal sorting methods:

(i) Selection sort
(ii) Bubble sort (or Sorting by Exchange)
(iii) Insertion sort
(iv) Merge sort
(v) Quick sort (or Sorting by Partitioning)
(vi) Heap sort
(vii) Radix sort

First three methods (i.e. Selection sort, Bubble sort and Insertion sort) are simple but inefficient sorting methods, while others are slightly complex but efficient sorting methods.

Note: For all the sorting methods, we will consider a linear array a[n] whose elements are to be sorted in ascending order. You can modify them for the descending order just by reversing the comparison operation. To be consistent with C/C++ language conventions, the elements of the array have index 0 .. n-1.

Selection Sort

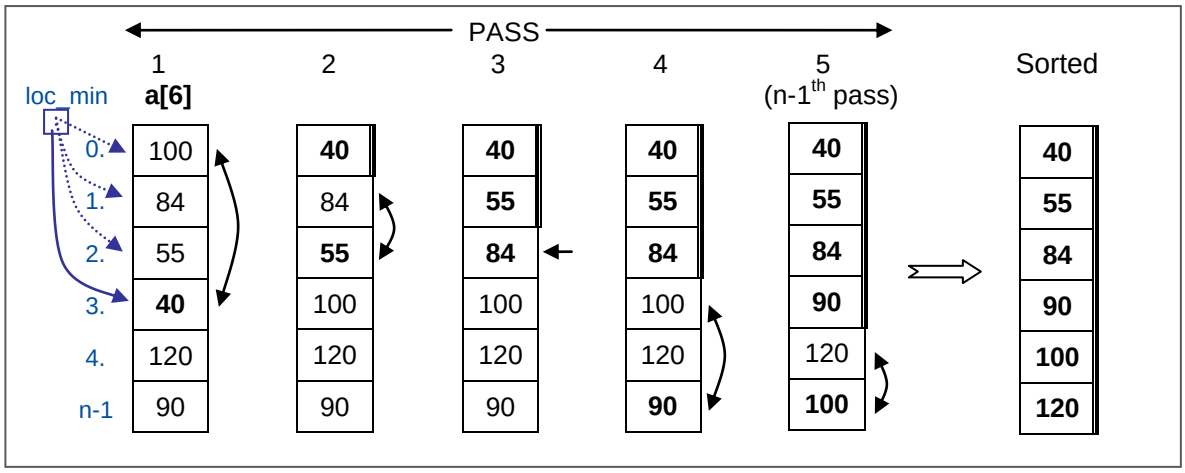
It is one of the easiest methods to sort the data but it is slow and inefficient sorting method. However, it works fine for smaller inputs.

In this method, an n-element linear array is sorted in ascending order in the following manner:

- We begin with the 0^{th} element and a search is performed in the entire array to locate the element having the smallest value.
- When it is found, it is placed in its final position by interchanging it with the 0^{th} element. If 0^{th} element itself contains the smallest value, this means that it is already in its final position and in this case no interchanging of elements is required.
- A search for the element having the smallest value and placing it in its final position is called a pass (or iteration).
- Thus, after 1^{st} pass, the element with the smallest value is placed in its final position (i.e. at location 0 in the array). Now, only n-1 elements are left for sorting.
- Therefore, in the 2^{nd} pass, we begin with the 1^{st} element and a search is again performed to locate the element having the 2^{nd} smallest value. When it is found, it is interchanged with the 1^{st} element. Thus, after 2^{nd} pass, smallest and 2^{nd} smallest elements have been placed in their final positions (at location 0 and 1 respectively in the array) and now only n-2 elements are left for sorting.
- This process of searching the element having the smallest, 2^{nd} smallest and so on value and placing it in its final position in the array continues until all the elements are sorted.
- Total **n-1** such passes are required to sort the array completely.

Example:

- Suppose we have an unsorted linear array a[6] in memory and we want to sort it in ascending order using selection sort method. The selection sort method sorts the array in the following manner:



a) In the 1^{st} pass, we begin with the 0^{th} element and a search is performed in the entire array to locate the element having smallest value. It is found at location 3 whose value is 40. It is placed in its final position by interchanging it with the 0^{th} element. Now only n-1 elements are left for sorting.

b) In the 2nd pass, we begin with the 1st element and a search is performed in the array in the forward direction to locate the element having smallest value. It is found at location 2 whose value is 55. It is placed in its final position by interchanging it with the 1st element. Now only n-2 elements are left for sorting.

c) In the 3rd pass, we begin with the 2nd element and a search is performed in the array in the forward direction to locate the element having smallest value. It is found at location 2 whose value is 84. Since it is already in its final position, hence no interchanging of elements is required.

d) In the 4th pass, we begin with the 3rd element and a search is performed in the array in the forward direction to locate the element having smallest value. It is found at location 5 whose value is 90. It is placed in its final position by interchanging it with the 3rd element.

e) In the 5th (which is the last) pass, we begin with the 4th element and a search is performed in the array in the forward direction to locate the element having smallest value. It is found at location 5 whose value is 100. It is placed in its final position by interchanging it with the 4th element.

f) After 5th pass, the array is sorted. Thus to sort n elements, n-1 passes are required.

Algorithm for Selection Sort

Following Selection sort algorithm sorts the linear array in ascending order.

Algorithm sel_sort(a,n)

```
1.  for pass ← 1 to n-1 do
2.     ▷initialize location of smallest
3.     loc_min ← pass-1
4.     ▷find location of smallest value
5.     for i ← pass to n-1 do
6.        if a[i]<a[loc_min] then
7.           loc_min ← i
8.     ▷interchange elements if required
9.     if loc_min !=(pass-1) then
10.        temp ← a[pass-1]
11.        a[pass-1]←a[loc_min]
12.        a[loc_min]←temp
13. ▷end of for loop of step 1
14. return
```

C++ function for Selection Sort

Assume that we have defined the following *array* class which contains a dynamic linear array a[n] whose elements are to be sorted in ascending order:

```
class array
{ int *a;                          //Dynamic sized array.
  int n;                           //No. of elements in the array.
 public:
  void getdata(int x[],int size);
  void sel_sort();
  void disp();
  ~array()                         //destructor
  { delete [] a; }
 };
```

The C++ function to sort the array a[n] in ascending order using Selection sort is given below:

```
void array::sel_sort()
{ int pass,loc_min,i,temp;
  for(pass=1; pass<=n-1; pass++)
  {  loc_min=pass-1;
     for(i=pass; i<=n-1; i++)
     { if(a[i]<a[loc_min])
           loc_min=i;
     }
     if(loc_min !=(pass-1))
     { temp = a[pass-1];
       a[pass-1]=a[loc_min];
       a[loc_min]=temp;
     }
  }
}
```

Time complexity (running time)

Let f(n) be the number of comparisons needed to sort an *n*-element array using the selection sort algorithm.

Note that the number of comparisons in the selection sort algorithm is independent of the original order of the input values. The algorithm always makes n-1 comparisons in the 1st pass to find the smallest element, n-2 comparisons in the 2nd pass to find the second smallest element and so on.

Accordingly,

No. of comparisons made in the 1st pass = n-1

" " 2nd pass = n-2

: :

: :

" " (n-1)th pass = 1

Thus, number of comparisons f(n) = (n-1)+(n-2)+(n-3)+….+3+2+1

$$f(n) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \mathbf{O(n^2)}$$

C++ program that implements and demonstrates sorting using Selection Sort

```
//selsort1.cpp
#include<iostream>
using namespace std;
class array
{ int *a;                  //Dynamic sized array.
  int n;                   //No. of elements in array a.
public:
  void getdata(int x[],int size);
  void sel_sort();
  void disp();
  ~array()
  { delete [] a; }
};
void array::getdata(int x[],int size)
{ n=size;
  a=new int[n];
  for(int i=0; i<n; i++)
     a[i]=x[i];
}
void array::disp()
{ for(int i=0; i<n; i++)
    cout<<a[i]<<' ';
  cout<<endl;
}
void array::sel_sort()
{ int pass,loc_min,i,temp;
  for(pass=1;pass<=n-1;pass++)
  { loc_min=pass-1;
    for(i=pass; i<=n-1;i++)
    { if( a[i] < a[loc_min] )
         loc_min=i;
    }
    if(loc_min!=(pass-1))
      { temp=a[pass-1];
        a[pass-1]=a[loc_min];
        a[loc_min]=temp;
      }
  }
}
int main()
{ const int n=10;
  int a[n];
  array obj;
  cout<<"Enter "<<n<<" integer values for sorting:\n";
  for(int i=0; i<n; i++)
     cin>>a[i];
  obj.getdata(a,n);
  cout<<"\nArray before sorting:\n";
  obj.disp();
  obj.sel_sort();
  cout<<"\nArray after sorting:\n";
  obj.disp();
  return 0;
}
```

Test run

```
Enter 10 integer values for sorting:
5 42 67 75 42 88 82 33 24 12

Array before sorting:
5 42 67 75 42 88 82 33 24 12

Array after sorting:
5 12 24 33 42 42 67 75 82 88
```

Bubble Sort (or **Sorting by Exchange)**

This is a simple but slow and inefficient sorting method. However, this sorting method works fine for smaller inputs. Since this method relies heavily on exchange mechanism, that's why it is also called sorting by exchange.

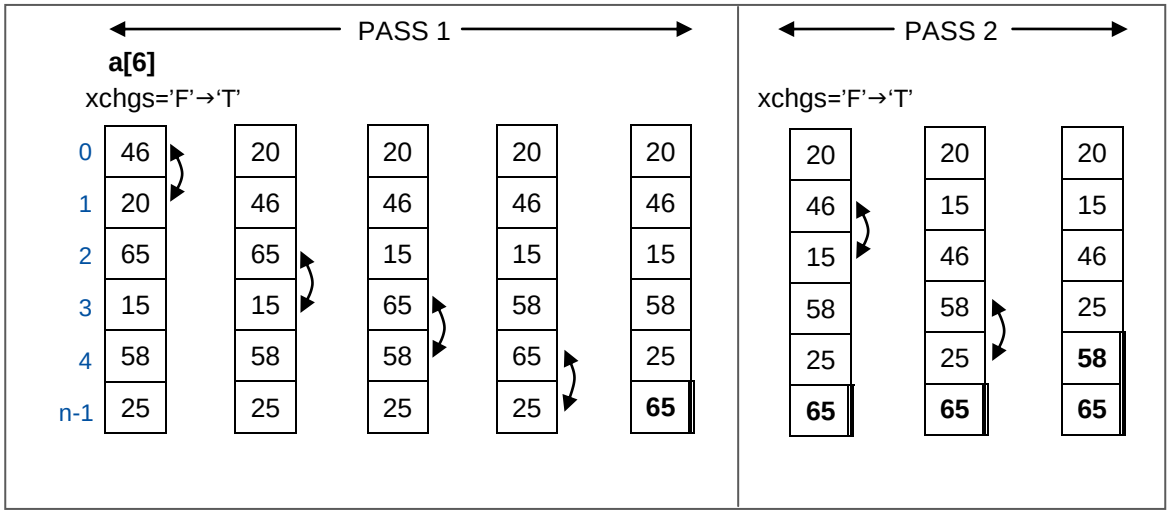
In bubble sort, we compare the two adjacent keys and as soon as it is discovered that the keys are not in the desired order, the two keys are immediately interchanged. This causes the records with smaller keys to bubble up, that's why its name "bubble sort".

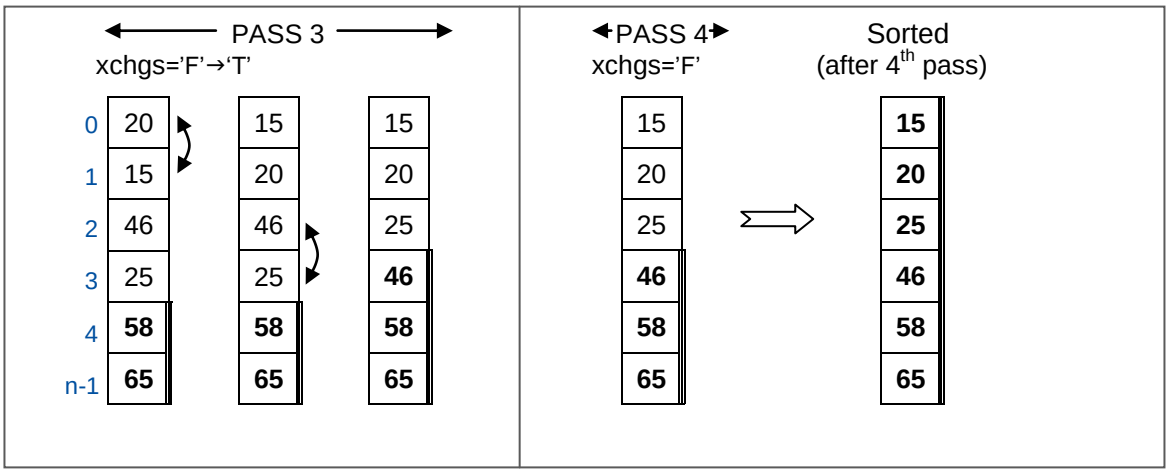
In this method, an n-element array is sorted in ascending order in the following manner:

- To begin with, the 0^{th} element is compared with the 1^{st} element. If it is found that the 0^{th} element is greater than the 1^{st} element (i.e. if they are found to be out of order), they are interchanged. Then 1^{st} element is compared with the 2^{nd} element, if they are found to be out of order, they are interchanged.
- In this way all the elements (excluding the last) are compared with their next element and interchanged if required. This is called a pass (or iteration).
- After 1^{st} pass, the largest value gets placed at the last position (i.e. n-1^{th} position) in the array which is its final position. Now, only first n-1 elements are left for sorting.
- Similarly, in the 2^{nd} pass, the same process is repeated on the first n-1 elements in the array. This places the 2^{nd} largest value in its final position (i.e. in the 2^{nd} last position in the array) and so on.
- After each pass, a check is made to determine whether any interchanges are made during the pass or not. If no, then the array must be sorted and no further passes are required.
- In this way, **at least one** and **at most n-1** such passes are performed by the algorithm to sort the array completely.

Example:

- Suppose we have an unsorted linear array a[6] and we want to sort it in ascending order using bubble sort method. The bubble sort method sorts the array in the following manner:





- In this example, during the 4th pass no elements are interchanged which indicates that the array has been sorted and there is no need to perform further passes (i.e. the last 5th pass in this example).
- Note that if the smallest value is in the last or last but one position in the input array, then all **n-1** passes will be made by the algorithm.

Bubble Sort algorithm

Following Bubble sort algorithm sorts the linear array in ascending order.

Algorithm bubble_sort(a,n)

```
1.  for pass ← 1 to n-1 do
2.    xchgs←'F'
3.    ▷compare successive elements & exchange if required
4.    for i ← 0 to (n-1-pass) do
5.      if a[i]>a[i+1] then
6.         temp←a[i]
7.         a[i]←a[i+1]
8.         a[i+1]←temp
9.         xchgs←'T'
10.   ▷if no exchanges made during the pass, then data is sorted
11.   if xchgs='F' then
12.       exitloop
13.  ▷end of for loop of step 1
14. return
```

Note: *In some books, you may find Bubble sort algorithm not using the xchgs flag. If xchgs flag is not used – then the algorithm will always make n-1 passes like selection sort.*

C++ function for Bubble Sort

Assume that we have defined the following *array* class which contains a dynamic linear array a[n] whose elements are to be sorted in ascending order:

```
class array
{ int *a;
  int n;                          //No. of elements in array a.
 public:
  void getdata(int x[],int size);
  void bubble_sort();
  void disp();
  ~array()                        //Destructor
  { delete [] a; }
 };
```

The C++ function to sort the array a[n] in ascending order using Bubble sort is given below:

```
void array::bubble_sort()
{ int pass,i,xchgs,temp;
  for(pass=1; pass<=n-1; pass++)
  { xchgs=0;
    for(i=0; i<=(n-1-pass); i++)
    { if(a[i]>a[i+1])
       { temp=a[i];
         a[i]=a[i+1];
         a[i+1]=temp;
         xchgs=1;
       }
    }
    if(xchgs==0) break;
  }
}
```

Running time or Computational complexity

Let f(n) be the number of comparisons needed to sort an *n*-element array using the bubble sort algorithm.

Best case: When already sorted data is given as input, the algorithm performs only one pass and therefore in this case the algorithm makes minimum no. of comparisons.

No. of comparisons f(n) = n-1 = **O(n)**

Worst case: When the value of last (or last but one) element of the input array is smallest, then the algorithm performs n-1 passes. Thus in this case the algorithm makes maximum number of comparisons. Accordingly,

No. of comparisons made in the 1^{st} pass = n-1

" " 2^{nd} pass = n-2

: :

: :

" " $(n-1)^{th}$ pass = 1

Thus, number of comparisons $f(n) = (n-1)+(n-2)+(n-3)+\ldots+3+2+1$

$$f(n) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \mathbf{O(n^2)}$$

Average case: $\mathbf{O(n^2)}$

Average case is more difficult to analyse. It is determined by averaging the execution times for all possible instances of input data. That is, take all possible instances of input (all the possible arrangements of all the possible keys), find execution time for each instance and then find their average.

Note: A C++ program *bublsor1.cpp* that implements and demonstrates sorting using Bubble sort is given in *Programs* folder.

Insertion Sort

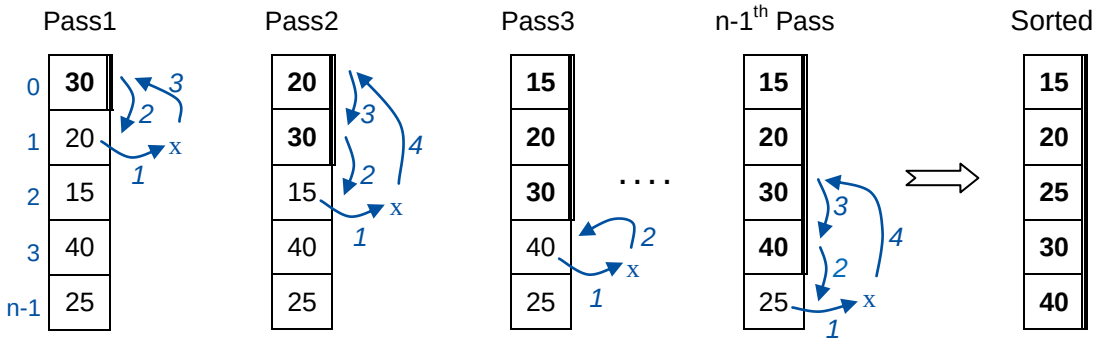
Like selection and bubble sort methods, this is also a simple but slow and inefficient sorting method but works fine for smaller inputs. This sorting technique is often used by card players for arranging the cards into some sequential order while playing cards.

In this method, an n-element array is sorted in ascending order in the following manner:

- This method works by dividing the given array into two parts - sorted part and unsorted part.
- Initially, the sorted part comprises only one element i.e. the 0^{th} element of the array forms the sorted part. All the remaining n-1 elements form the unsorted part.
- The first element from the unsorted part is selected and inserted in the sorted part at its proper position. To accomplish this, first of all the selected element's value is copied in a temporary variable, say **x**, and then x is compared with the elements before it in the sorted part (beginning from last (biggest) element in the sorted part) one by one until we encounter an element whose value is smaller than x or sorted part is exhausted. While performing these comparisons if it is found that x can be inserted at a suitable position in the sorted part then space is created for it by shifting/moving the elements having values greater than x one position forward and inserting x at its proper position in the sorted part.
- Inserting an element (of unsorted part) in its proper position in the sorted part is called a pass (or iteration). Since we need to insert a total of n-1 elements of unsorted part into sorted part, hence n-1 such iterations are required. After **n-1** iterations, the array is sorted.

Example:

- Suppose we want to sort the following integer array a[5] in ascending order using insertion sort method. The insertion sort method sorts the array in the following manner:



a) Initially the sorted part consists of 0^{th} element i.e. 30. In the 1^{st} iteration, 1^{st} element i.e. 20 is selected for insertion in the sorted part. It is copied in a temporary variable x. Now x is compared with the last element in the sorted part i.e. 30. Since 30 is greater than x, it is moved one position forward. After this, no more elements are available in sorted part for comparison (i.e. sorted part is exhausted), therefore x is inserted at 0^{th} location.

b) In the 2^{nd} iteration, 2^{nd} element i.e. 15 (now 1^{st} element in the unsorted part) is selected for insertion in the sorted part. It is copied in **x**. Now x is compared with the last element in the sorted part i.e. 30. Since 30 is greater than x, it is moved one position forward. Then x is compared with 20. 20 is also greater than x, therefore it is also moved one position forward. After this sorted part is exhausted, therefore x is inserted at 0^{th} location.

c) In the 3^{rd} iteration, 3rd element i.e. 40 is selected and copied in x. Now x is compared with 30. Since 30 is smaller than x, hence 30 is not shifted and no further comparisons are required. This means that 40 is already in its proper position.

d) Similarly 4^{th} pass is performed in which last element of the unsorted part is placed in its final position. Thus, after **n-1** passes array is sorted.

Insertion Sort algorithm

Following Insertion sort algorithm sorts the linear array in ascending order.

```
Algorithm ins_sort(a,n)
1.  ▷insert n-1 elements of unsorted part into sorted part
2.  for i ← 1 to n-1 do
3.     x←a[i]
4.     ▷assign location of last element in the sorted part
5.     j←i-1
6.     ▷insert ith element in its proper position in the sorted part
7.     while x<a[j] and j>=0 do
8.        a[j+1]←a[j]
9.        j←j-1
10.    a[j+1]←x
11. ▷end of for loop
12. return
```

C++ function for Insertion Sort

Assume that we have defined the following *array* class which contains a dynamic linear array a[n] whose elements are to be sorted in ascending order:

```
class array
{ int *a;
  int n;                    //No. of elements in array a.
 public:
  void getdata(int x[],int size);
  void ins_sort();
  void disp();
  ~array()                  //destructor
  { delete [] a; }
 };
```

The C++ function to sort the array a[n] in ascending order using Insertion sort is given below:

```
void array::ins_sort()
{ int i,j,x;
  for(i=1; i<=n-1; i++)
  { x=a[i];
    j=i-1;
    while(x<a[j] && j>=0)
    { a[j+1]=a[j];
      j=j-1;
    }
    a[j+1]=x;
  }
}
```

Time complexity (Running time)

Let f(n) be the number of comparisons needed to sort an *n*-element array using insertion sort.

Best case: When input data is already sorted, the algorithm performs only one comparison in each pass for each of the n-1 elements of the unsorted part to determine that they are already in their proper positions. Thus,

number of comparisons f(n) = n−1 = **O(n)**

Worst case: When input data is in reverse order of that in which it is to be sorted, the algorithm makes maximum no. of comparisons in each pass. That is,

number of comparisons f(n) = 1+2+3+….+(n-2)+(n-1)

$$f(n) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \mathbf{O(n^2)}$$

Average case: **$O(n^2)$**

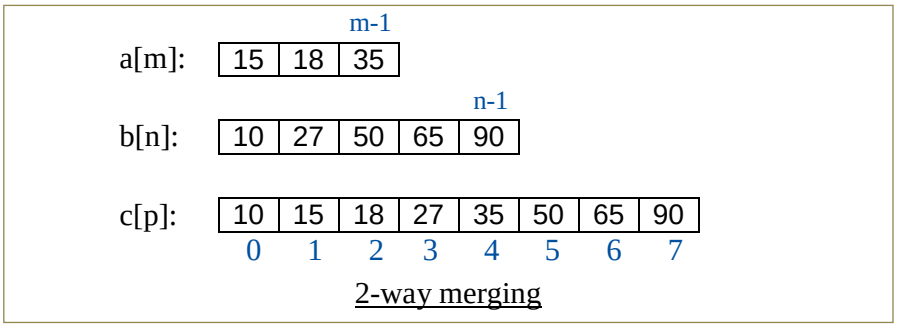
Note: A C++ program *insrsort.cpp* that implements and demonstrates sorting using Insertion sort is given in *Programs* folder.

Merging

- **Merging is the process of combining the sorted elements of two similar structures, such as linear arrays, into a single structure which contains all the elements of both the structures in sorted order.** Merging two sorted collections of data into a single sorted collection is called *2-way merging*.
- In order to merge two sorted linear arrays, smaller values occurring in either of the array are successively selected and placed in the new array.
- This process is repeated until any one of the two arrays is exhausted.
- Then all the remaining unprocessed elements of the other array are simply appended to the end of the new array.
- This new array is the desired merged array which contains all the elements of both the arrays in sorted order.

Example:

Suppose we have two sorted linear arrays a[m] and b[n], where m=3 and n=5. We want to merge them to get the array c containing (m+n) elements in sorted order. The arrays a and b are merged together in the following manner:



2-way merging

(i). Element a[0] i.e. 15 is compared with b[0] i.e. 10. Since 10 is smaller than 15, hence 10 i.e. b[0] is placed (assigned) in the array c at c[0]. Thus, b[0] has been processed.

(ii). Now, element a[0] i.e. 15 is compared with b[1] i.e. 27. Since 15 is smaller than 27, hence 15 i.e. a[0] is placed in the array c at the next position i.e. c[1]. Thus, a[0] has been processed.

(iii). Now, element a[1] i.e. 18 is compared with b[1] i.e. 27. Since 18 is smaller than 27, hence 18 i.e. a[1] is placed in the array c at the next position i.e. c[2]. Thus, a[1] has been processed.

(iv). Now, element a[2] i.e. 35 is compared with b[1] i.e. 27. Since 27 is smaller than 35, hence 27 i.e. b[1] is placed in the array c at the next position i.e. c[3]. Thus, b[1] has been processed.

(v). Now, element a[2] i.e. 35 is compared with b[2] i.e. 50. Since 35 is smaller than 50, hence 35 i.e. a[2] is placed in the array c at the next position i.e. c[4]. Thus, a[2] has been processed.

(vi). Now array a is exhausted, therefore all the remaining unprocessed elements of array b are simply appended to the array c i.e. elements b[2] through b[n-1] whose values are 50, 65, 90 respectively are placed at the next positions i.e. c[5], c[6] and c[7] in the array c.

Array c is the desired merged array.

Note: *2-way merging* is the simplest form of merging. Similarly, *3-way*, *4-way* and *m-way* merging can also be performed.

Algorithm to Merge Two Sorted Linear Arrays

Following algorithm merges two sorted linear arrays a[m] and b[n] into the array c which will contain m+n elements in ascending order.

Algorithm merge(a,b,c,m,n).

```
1.  i←j←k←0
2.  ▷compare corresponding elements and output the smallest into C
3.  while i≤(m-1) and j≤(n-1) do
4.    if a[i]<b[j] then
5.      c[k]←a[i]
6.      i←i+1
7.      k←k+1
8.    else
9.      c[k]←b[j]
10.     j←j+1
11.     k←k+1
12. ▷end of while
13. ▷append all the remaining unprocessed elements into array C
14. if i>(m-1) then
15.   for rem←j to n-1 do
16.     c[k]←b[rem]
17.     k←k+1
18. else
19.   for rem←i to m-1 do
20.     c[k]←a[rem]
21.     k←k+1
22. return
```

C++ function to Merge two Sorted arrays

Assume that we have defined the following *three-arrays* class which contains dynamic linear arrays a[m],b[n] and c[p]. The sorted elements of arrays a and b are to be merged in array c:

```
class three_arrays
{ int *a,m;
  int *b,n;
  int *c,p;
 public:
  void getdata(int x[],int xsize,int y[],int ysize);
  void merge();
  void disp_input_arrays();
  void disp_merged_array();
  ~three_arrays()
  { delete [] a,b,c; }
};
```

C++ function to merge arrays *a[m]* and *b[n]* to get array *c[p]* is given below:

```
void three_arrays::merge()
{ int i,j,k;
  i=j=k=0;
  while (i<=(m-1) && j<=(n-1))
  { if( a[i]<b[j] )
      c[k++]=a[i++];
    else
      c[k++]=b[j++];
  }
  if( i>(m-1) )
    { for(int rem=j; rem<=(n-1); rem++)
        c[k++]=b[rem];
    }
  else
    { for(int rem=i; rem<=(m-1); rem++)
        c[k++]=a[rem];
    }
}
```

Note: *This is known as 2-way merging which is the simplest form of merging. 3-way, 4-way or m-way merging can also be used.*

Time complexity (Running time) of merging operation

Each comparison assigns one element to the array c which eventually has m+n elements. Therefore, the number f(n) of comparisons made by the algorithm can't exceed m+n.

Accordingly, running time of merge algorithm is **O(m+n)**.

Note: A C++ program *merge2.cpp* that implements and demonstrates the merging of two sorted integer arrays is given in *Programs* folder.

Divide-and-Conquer Approach

In divide-and-conquer strategy of problem solving, the problem is divided into a number of subproblems that are similar to the original problem but smaller in size. These subproblems are solved recursively, and then these solutions are combined to get the solution to the original problem.

Divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide: Divide the problem into a number of smaller subproblems.

Conquer: These subproblems are conquered by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine: Solutions to the subproblems are combined to get the solution for the original problem.

Examples of divide and conquer algorithms are: Merge sort algorithm, Quick sort algorithm, Binary search algorithm.

Merge Sort

Merge sort is quite efficient sorting method which performs very well on larger data. It uses the divide-and-conquer approach, which operates as follows:

Divide: Divide the n-element list to be sorted into two sublists of n/2 elements each.

Conquer: Sort the two sublists recursively using merge sort.

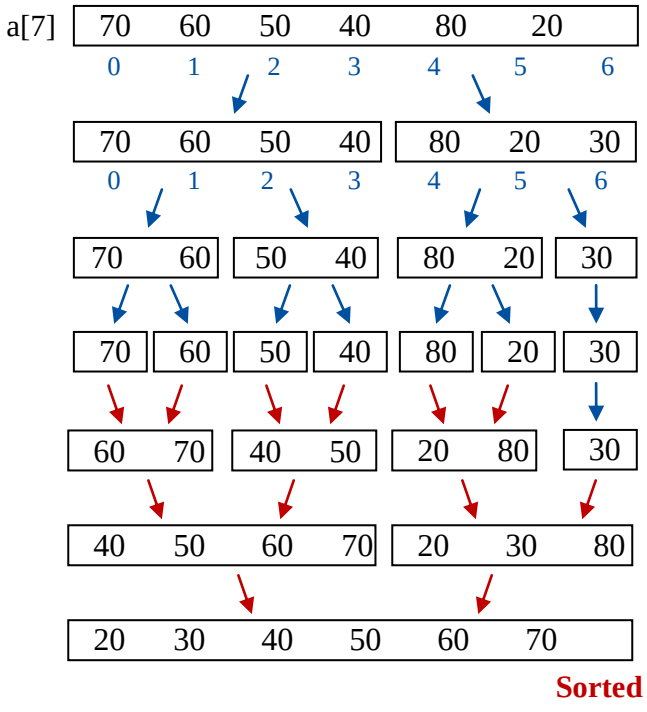
Combine: Merge the two sorted sublists to produce the desired sorted list.

In merge sort, the entire list is divided into two equal sublists recursively until a sublist contains single element.

The key operation of merge sort algorithm is to merge two sorted sublists into a single sorted list. To perform the merging operation the elements of both the sublists are compared and the smaller element is placed in the third list until all the elements from both the sublists are placed in the third list. Third list is required to temporary store the merged data which is copied in the original list after each merging operation.

Example:

Sorting of integer array a[7] using recursive merge sort algorithm is depicted in the following figure:



Recursive Merge Sort algorithm

The recursive algorithm to sort array a[n] with lower bound *lb* and upper bound *ub* in ascending order is as below:

```
Algorithm merge_sort(a,lb,ub)
1. if lb<ub then                    ▷if array contains 2 or more elements
2.   mid←(lb+ub)/2                  ▷then recursively:
3.   merge_sort(a,lb,mid)           ▷(i). split the array into two sub-arrays
4.   merge_sort(a,mid+1,ub)         ▷.     (until they contain only 1 element)
5.   merge(a,lb,mid,mid+1,ub)       ▷(ii). merge the two sub-arrays.
6. return
```

```
Algorithm merge(a,lb1,ub1,lb2,ub2)
1.  ▷merge the two subarrays and store merged data in a temporary array FINAL
2.  i←k←lb1
3.  j←lb2
4.  while i<=ub1 and j<=ub2 do
5.    if a[i]<a[j] then
6.       final[k]←a[i]              ▷size of final[] is same as size of a[]
7.       i←i+1
8.       k←k+1
9.    else
10.      final[k]←a[j]
11.      j←j+1
12.      k←k+1
13. ▷end of while loop
14. if i>ub1 then
15.   for rem←j to ub2 do
16.      final[k]←a[rem]
17.      k←k+1
18. else
19.   for rem←i to ub1 do
20.      final[k]←a[rem]
21.      k←k+1
22. ▷copy merged elements from final[] array  into the original array a[]
23. for i←lb1 to ub2 do
24.   a[i]←final[i]
25. return
```

C++ function for Merge Sort

Assume that we have defined the following *array* class which contains a dynamic linear array a[n] whose elements are to be sorted in ascending order:

```
class array
{ int *a;
  int n;
 public:
  void getdata(int x[],int size);
  void merge_sort(int lb,int ub);
  void merge(int lb1,int ub1,int lb2,int ub2);
  void disp();
  ~array()
  { delete [] a; }
};
```

The C++ function merge_sort() to sort the integer array a[n] in ascending order using merge() function is given below:

```
void array::merge_sort(int lb,int ub)
{ int mid;
  if(lb<ub)                          //if array contains 2 or more elements,
  { mid=(lb+ub)/2;                   //then recursively:
    merge_sort(lb,mid);              //split the array
    merge_sort(mid+1,ub);            //until they contain only 1 element
    merge(lb,mid,mid+1,ub);          //and merge them
  }
}
```

```
void array::merge(int lb1,int ub1,int lb2,int ub2)
{ int i,j,k,rem;
  int *final=new int[n];       //merging requires additional array to hold
  i=k=lb1;                     //merged data temporarily. When merging
  j=lb2;                       //of 2 subarrays is over, it is copied in orig.array a.
  while(i<=ub1 && j<=ub2)
  { if(a[i]<a[j])
      final[k++]=a[i++];
    else
      final[k++]=a[j++];
  }
  if(i>ub1)                                //append remaining elements of the
  {   for(rem=j;rem<=ub2;rem++)            //other subarray
        final[k++]=a[rem];
  }
  else
  {  for(rem=i;rem<=ub1;rem++)
       final[k++]=a[rem];
  }
  for(i=lb1;i<=ub2;i++)            //copy merged elements from temp. array into the
     a[i]=final[i];                //original array a.
  delete [] final;
}
```

Time complexity

To sort just one element, merge sort takes constant time. When we have n>1, we break down the running time as follows:

Divide: The divide step just computes the middle of the sub-array, which takes constant time.

Conquer: We recursively solve two sub-problems, each of size n/2. Time taken to sort left half is T(n/2) and time taken to sort right half is also T(n/2) which contributes 2T(n/2) to the running time.

Combine: Time needed to merge the left half of size n/2 with right half of size n/2 is n.

	Algorithm merge_sort(a,lb,ub)
	1. if lb<ub then
1	2. mid←(lb+ub)/2
T(n/2)	3. **merge_sort**(a,lb,mid)
T(n/2)	4. **merge_sort**(a,mid+1,ub)
n	5. **merge**(a,lb,mid,mid+1,ub)
	6. return

Adding the running times of these three steps, we get the recurrence for the worst-case running time T(n) of merge sort:

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

Now let us solve the recurrence by using substitution method. The recurrence for merge sort is:

T(n) = 2T(n/2)+n -----(1)

Compute T(n/2) by substituting n/2 in place of n in eq.(1):

T(n/2) = 2T(n/4)+n/2 -----(2)

Substitute eq.(2) in eq.(1):

T(n) = 2**{**2T(n/4)+n/2**}**+n

$T(n) = 2^2T(n/2^2)+2n$ -----(3)

Compute T(n/4) by substituting n/4 in place of n in eq.(1):

T(n/4) = 2T(n/8)+n/4 -----(4)

Substitute eq.(4) in eq.(3):

T(n) = 2^2**{**2T(n/8)+n/4**}**+2n

$T(n) = 2^3T(n/2^3)+3n$ -----(5)

Similarly, if we substitute n/8 in place of n in eq.(1) and substitute the equation that we get in eq.(5), the next eq. will be:

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

:

:

$T(n) = 2^kT(n/2^k)+kn$ -----(6)

$n/2^k = 1$ (Since at each level, the sub-array is being divided into two halves, hence ultimately the sub-array will contain only 1 element. So the value of term $n/2^k$ will become 1)

$2^k = n$

$\log_2 2^k = \log_2 n$ (applying $\log_2$ on both hand sides)

$k \log_2 2 = \log_2 n$

$k = \log_2 n$ ($\log_2 2 = 1$)

Now substitute the values of **2^k**, **$n/2^k$** and **k** back in eq.(6):

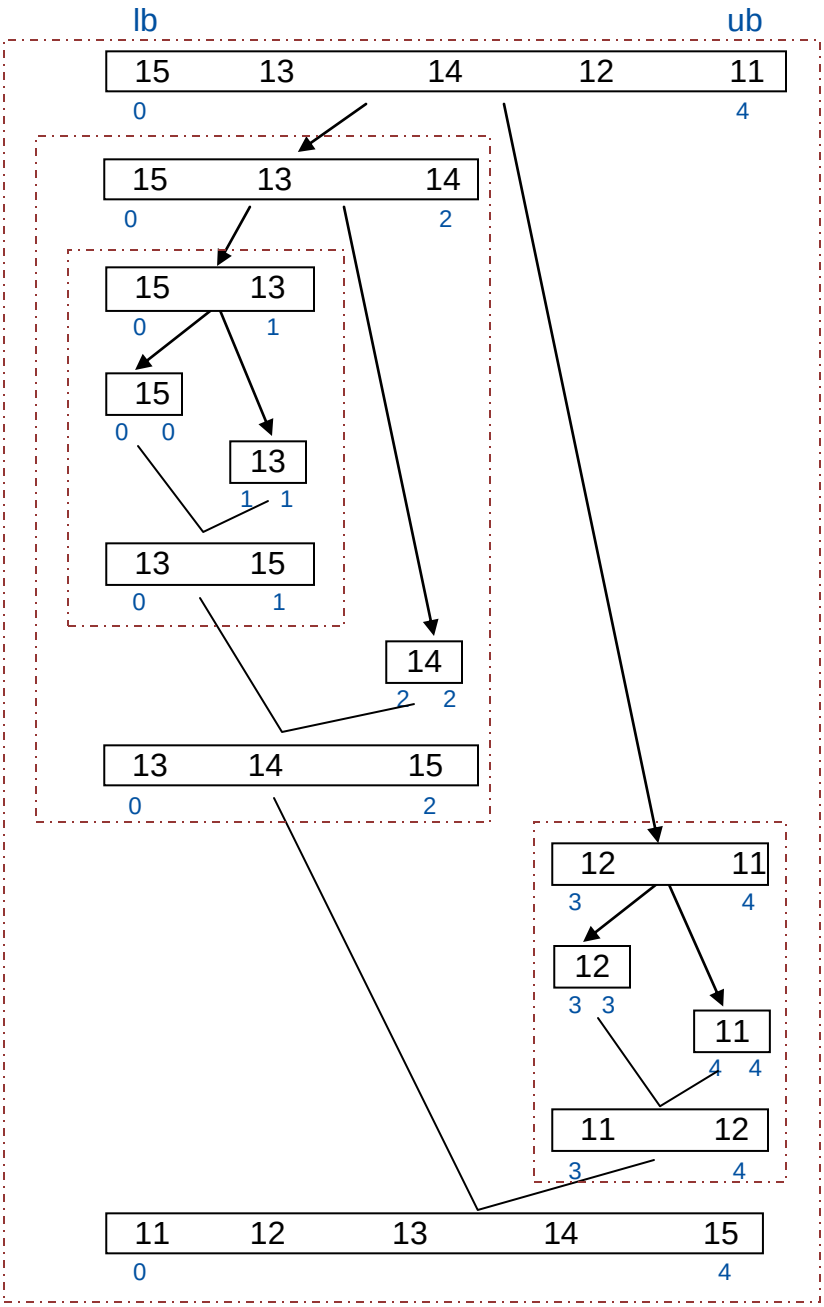
$T(n) = nT(1)+n \log_2 n$

$T(n) = n + n \log_2 n$ This is the solution of the recurrence.

Therefore time complexity of merge sort: $O(n \log_2 n)$.

Time complexity of merge sort is **$O(n \log_2 n)$** as it always divides the array in two halves and takes linear time to merge two halves. Merge sort requires equal amount of extra space because it uses a temporary array of the same size for holding the merged data temporarily, so its space complexity is O(n).

Note: A C++ Program *mergesor.cpp* to demonstrate the sorting of a linear array in ascending order is given in *Programs* folder.



Trace of recursive merge sort

Quick Sort (or Partition-Exchange Sort)

Quick sort is quite efficient sorting method which performs very well on larger data. This sorting method is based on divide-and-conquer problem-solving technique.

To sort a linear array a[n] with lower bound *lb* and upper bound *ub* in ascending order, quick sort uses the following three-step recursive divide-and-conquer process:

- **Divide**: A particular element (usually the 0^{th} element i.e. a[lb]), called pivot (or key), is selected and placed in its final position within the array in such a way that all the elements smaller than the pivot appear in the left part of the array, while all the elements greater than pivot appear in the right part of the array. Thus, this technique partitions the array into two subarrays - first one containing the smaller elements than the pivot, and the second one containing the elements greater than the pivot.
- **Conquer**: Recursively apply quick sort to both left and right subarrays.
- **Combine**: Since the subarrays are sorted in place, no work is needed to combine them, the entire array is now sorted.

Since in this method the array elements are partitioned and exchanged, this sorting technique is also known as partition-exchange sort.

Recursive Quick Sort algorithm to sort array a[n] in ascending order:

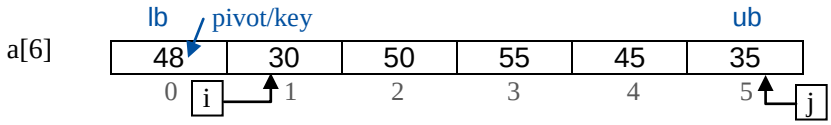
Algorithm quicksort(a,lb,ub)

```
1. if lb<ub then                          ▷if there are 2 or more elements
2.     j←partition(lb,ub)
3.     quicksort(a,lb,j-1)
4.     quicksort(a,j+1,ub)
5. return
```

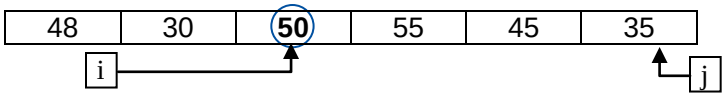
Algorithm partition(a,lb,ub)

```
1.  i←lb+1
2.  j←ub
3.  key←a[lb]
4.  while i<=j do
5.      while a[i]<key and i<=j do    ▷find an element a[i] s.t. a[i]>=key in forward direction
6.          i←i+1
7.      while a[j]>key do             ▷find an element a[j] s.t. a[j]<=key in backward direction
8.          j←j-1
9.      if i<j then
10.         interchange a[i] with a[j]
11.         i←i+1
12.         j←j-1
13.     else
14.         exitloop
15. ▷end of while loop
16. interchange a[lb] with a[j]       ▷place a[lb] (i.e. key)  in its final position
17. return j
```

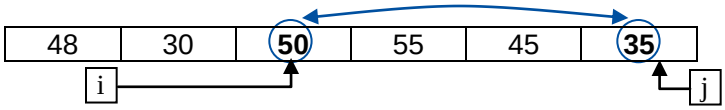
Example: Suppose we want to sort following array a[6] in ascending order using quick sort method:



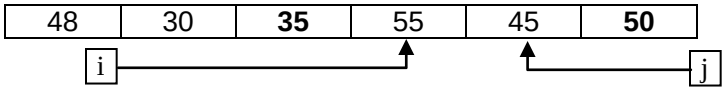
Set i=lb+1, j=ub. Find a[i] s.t. a[i]>=a[lb], in forward direction till i<=j (found at i=2).



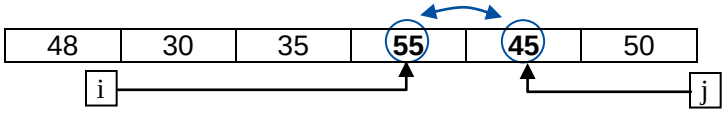
Find a[j] s.t. a[j]<=a[lb], in backward direction. It is found at loc 5.



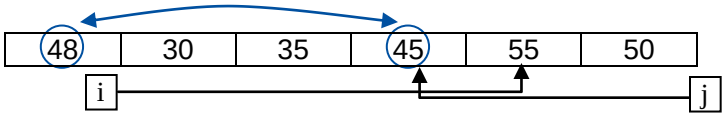
Since i<j, interchange a[i] with a[j], set i=i+1, j=j-1, and repeat above process.



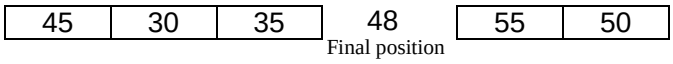
Find a[i] and a[j] again. Now a[i] is found at loc 3 and a[j] at loc 4.



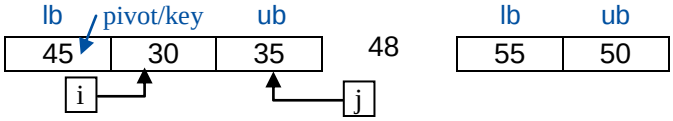
Since i<j, interchange a[i] with a[j], set i=i+1, j=j-1, and repeat above process.



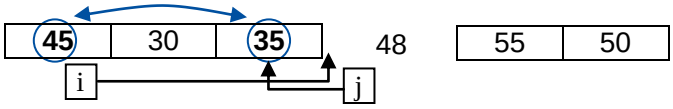
But now i<j is false.
Hence interchange a[lb] with a[j] and divide the array into two subarrays as below.



Repeat the same process on each sub-array one by one until they are sorted.

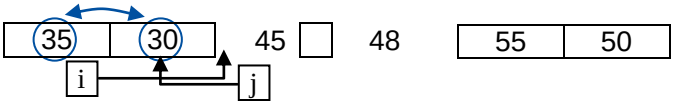


In first subarray, find a[i] and a[j]:



Since i<j is false, hence interchange a[lb] with a[j] and divide the array into two sub-arrays:

Repeat the same process on sub-arrays one by one.



Since i<j is false, hence interchange a[lb] with a[j] and divide the array.

30 35 45 48 55 50

A subarray which is empty or has only one element is a sorted array.

30 35 45 48 55 50

Repeat the same process on the last subarray.

Find location of a[i] and a[j] as below:

Since i<j is false, hence interchange a[lb] with a[j] and divide the array.

30 35 45 48 50 55

30 35 45 48 50 55 Sorted

C++ function for Quick sort

Assume that we have defined the following *array* class which contains a dynamic linear array a[n] whose elements are to be sorted in ascending order:

```
class array
{ int *a;
  int n;
 public:
  void getdata(int x[],int size);
  void quicksort(int lb,int ub);
  int partition(int lb,int ub);
  void disp();
  ~array() { delete [] a; }
};
```

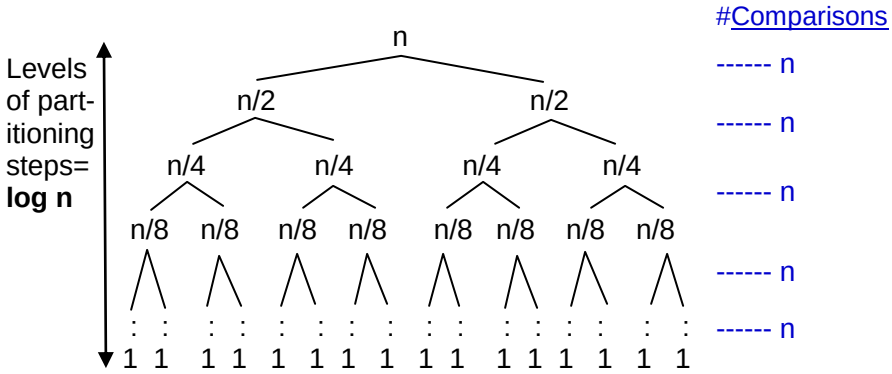
Recursive C++ function to sort the array *a[n]* in ascending order using Quick sort:

```
void array::quicksort(int lb,int ub)
{ int j;
  if(lb<ub)                    //if list contains 2 or more elements then
  { j=partition(lb,ub);        //place pivot in its final position and
    quicksort(lb,j-1);         //divide into 2 subarrays and
    quicksort(j+1,ub);         //sort each one (recursively).
  }
}
int array::partition(int lb,int ub)
{ int i,j,key,temp;
  i=lb+1;
  j=ub;
  key=a[lb];
  while(i<=j)
  { while(a[i]<key&&i<=j) i++;        //find a[i] s.t. a[i]≥a[lb]
    while(a[j]>key) j--;              //find a[j] s.t. a[j]≤a[lb]
    if(i<j)
```

```
        { temp=a[i];                          //interchange a[i] with a[j]
          a[i]=a[j];
          a[j]=temp;
          i++;
          j--;
        }
        else
          break;
    }
    temp=a[lb];                       //interchange a[lb] i.e key with a[j]
    a[lb]=a[j];                       //j is the final position of pivot/key
    a[j]=temp;
    return j;
}
```

Time complexity (Running time)

Best case: When the nature of the input data is such that the **array is always partitioned into two sub-arrays each of size n/2,** we get best case performance and quick sort runs much faster.

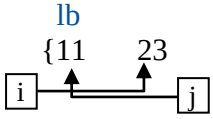


Thus there will be *log n* levels of partitioning steps and *n* comparisons at each partitioning level. Thus, total no. of comparisons f(n) = n log n.

Therefore, best case running time = **O(n log n).**

Worst case: Worst case occurs when the **array is always partitioned into two subarrays such that one subarray is empty and the other contains all the remaining elements** to be sorted. Such a situation, for example, occurs when the input data is already sorted.

For example, sorting of the input data {11, 23, 36, 42} would yield the following sequence of partitions:



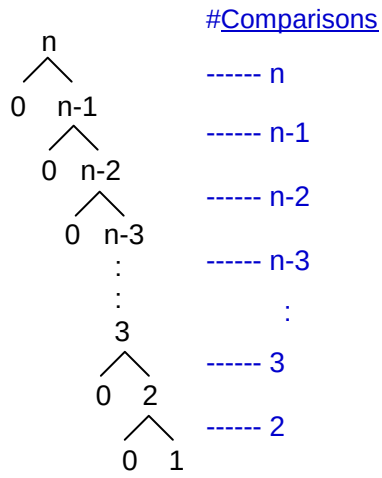
{} **11** {23 36 42}

11 {} **23** {36 42}

11 **23** {} **36** {42}

11 **23** **36** **42**

Due to this unbalanced partitioning the quick sort makes maximum no. of comparisons and its performance is no better than selection sort.



Recursion tree (when one subarray is always empty and the other contains all the remaining elements to be sorted)

Thus, no. of comparisons $f(n) = n+(n-1)+(n-2)+(n-3)+\ldots\ldots+3+2 = \frac{n(n+1)}{2} - 1$

Therefore, worst case running time = **$O(n^2)$**

Average case: **O(n log n)**

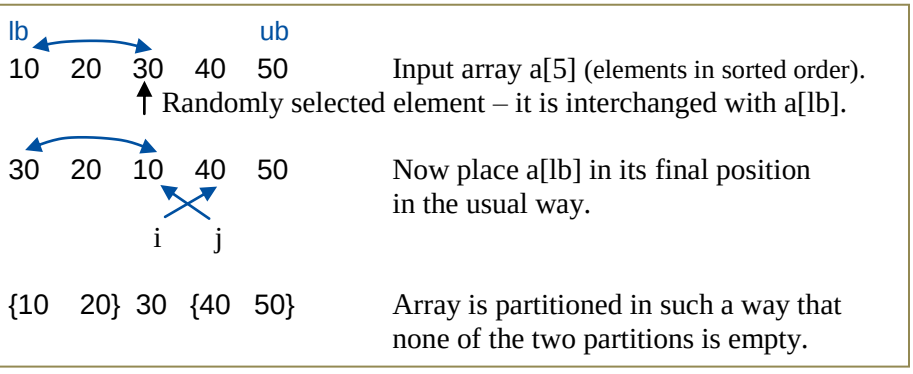
Worst case running time can sometimes be avoided by choosing the pivot/key more carefully for final placement at each step

To achieve this, the key for final placement can be selected based on any one of the following approaches:

1. At the beginning of each step **an element in the array is selected randomly and exchanged with the 0^{th} element i.e. a[lb],** or
2. At the beginning of each step **the approximate middle value is selected and exchanged with the 0^{th} element i.e. a[lb]**.

Then this 0^{th} element i.e. a[lb], is picked and placed in its final position in the usual way. This modification ensures that the quick sort algorithm produces a mixture of good and bad splits which yields a good running time.

Example:



Radix Sort

- Radix sort sorts the data by sorting keys on the corresponding digits, beginning with the least significant digit up to the most significant digit. We think of a key as an *n*-digit number (e.g. 365 is a 3-digit number).
- In the past, Radix sort was used on electro-mechanical equipment, called *Punched-Card Sorter* machine.
- To understand the basic idea of how Radix sort works, let's examine how the sorter machine used to sort the punched cards, each containing a numeric key.

Working of Sorter

Punched-Card Sorter could sort a deck of cards into ascending order on numeric keys. It had 10 receiving pockets (digit bins) corresponding to the 10 decimal digits labeled from 0 to 9. Each pocket acted/worked like a queue.

In order to sort the cards, the card deck was put into the sorter's feed hopper, from where the cards were fed one by one. The sorter used to sort the cards by examining only one digit of the keys at one time in the following manner:

- First of all the sorter was set to select the column of the cards containing unit's digit (least significant digit) of the keys. Similarly in the 2^{nd} pass, it was set to select the column containing ten's digit of the keys and so on.
- Then the *sorter* used to scan each card one by one and based on the digit in the selected column, the cards were distributed in the corresponding pockets. For example, if the digit was 0 then the card was distributed in pocket-0. If it was 1 then the card was distributed in pocket-1 and so on.
- When the entire card deck was processed, the piles of cards were gathered from the pockets and combined into a new deck, with the 0's pile on the top, the 1's pile next to the top, and so on, until the 9's pile was kept on the bottom of the deck. It was essential that order of the cards as they were gathered from the pocket should not be changed. This process was called a pass. Thus in the 1^{st} pass, the sorter used to sort the cards based on unit's digit of the keys.
- If keys on the cards were only 1-digit long then after one pass the cards were in sorted order.
- For sorting cards having 2-digit long keys, a 2^{nd} pass was also required in which ten's digit of the keys were examined and cards were distributed in the corresponding pockets. After 2^{nd} pass, the cards were sorted.
- Similarly to sort 3-digit long keys, three such passes were made. In the 3rd pass, the cards were distributed in corresponding pockets based on their hundred's digit, and so on.
- Thus, in general m passes were required to sort the cards having m-digit long keys.

Example:

- Suppose we have 15 cards, each of which contains a 2-digit long numeric key punched in columns 1-2.
- Suppose the following numeric keys are punched on these cards (one key on each card):

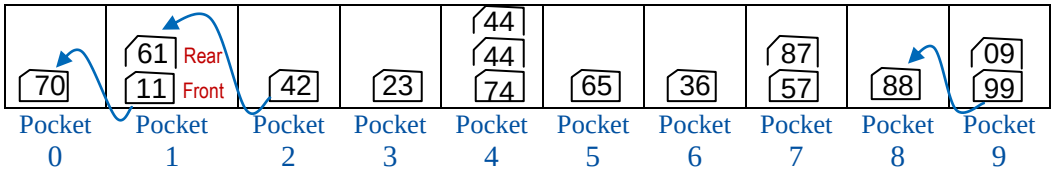
  74, 23, 42, 57, 11, 65, 99, 36, 44, 88, 70, 87, 61, 09, 44

- Since the keys punched on the cards are two-digit long, hence to sort these cards according to the keys in ascending order the sorter makes two passes as described below:

Pass-1:

The card deck is put in the sorter's input hopper and it is set to examine column no. 2 (containing the unit's digit of the keys) of each card. The cards are scanned by the sorter one by one and distributed in the corerresponding pockets based on unit's digit of the keys.

When all the cards get processed, the contents of different pockets will be as below:



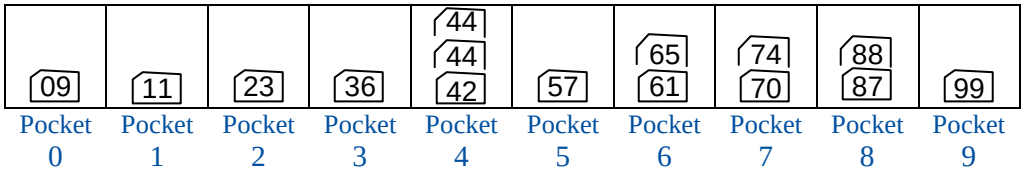
Now the cards are gathered from pocket-0 upto pocket-9 and combined into a new deck. Thus after 1st pass, the order of cards in the new deck will be as below:

70, 11, 61, 42, 23, 74, 44, 44, 65, 36, 57, 87, 88, 99, 09 (sorted on unit's digit)

Pass-2:

This new card deck is put in the sorter's input hopper and it is set to examine column no. 1 (containing the ten's digit of the keys) of each card. In this pass, the cards are distributed into corresponding pockets based on ten's digit of the keys.

When all the cards get processed, the contents of different pockets will be as below:



Again the cards are gathered from pocket-0 upto pocket-9. Now the cards will be in ascending order of their keys as below:

09, 11, 23, 36, 42, 44, 44, 57, 61, 65, 70, 74, 87, 88, 99 (sorted)

Running time (or Computational complexity)

- Radix sort algorithm requires *m* passes, where *m* represents maximum no. of digits in a key.
- In each pass, a particular digit (e.g. in 1st pass unit's digit) of all *n* keys is compared with each of the *r* digits, where *r* represents radix (base) of the number system used.
- Hence, the number of comparisons f(n) can't exceed *m*n*r* i.e.

  f(n) ≤ *m*n*r* = **O(*m*n*),** *r* is ignored because it is a constant and doesn't depend on n, but m does depend on n.

  Worst case: occurs when m=n (i.e. m is large). Therefore f(n) = $O(n^2)$.

  Best case: occurs when m=1 (i.e. keys are short). Therefore f(n) = O(n).

- Thus, radix sort performs well only when *m* is relatively short.

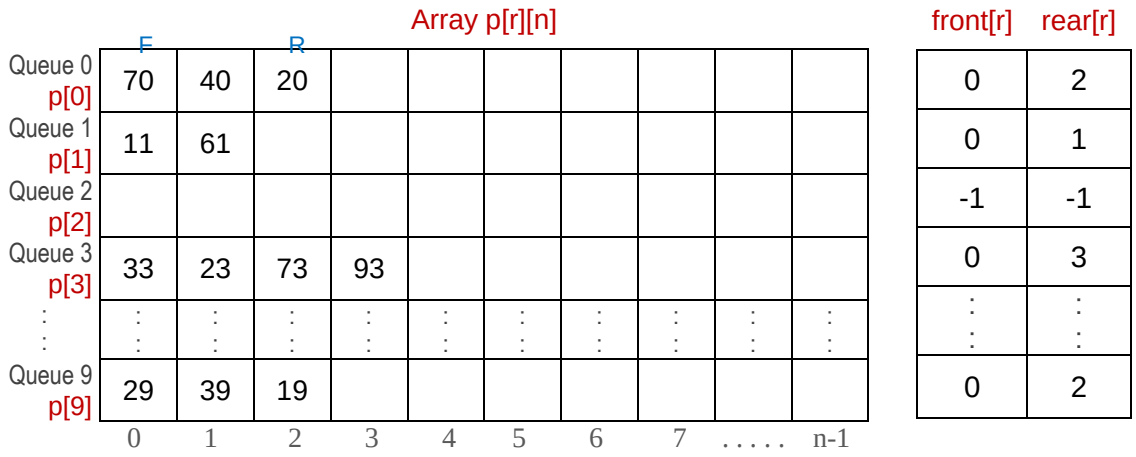
Implementing Radix Sort

Radix sort can be implemented on computer for sorting positive numeric data or alphabetic data, such as names etc. The *r* (equal to radix or base) pockets required may be represented in the following two ways:

1. **Using arrays to represent r queues** (corresponding to r pockets) or
2. **Using linked queues to represent r queues** (corresponding to r pockets).

1. Using Arrays to represent r Queues

- Suppose an integer array a[n] containing decimal numbers is to be sorted (here r=10).
- To represent r queues (pockets), suppose:
  - a 2-D integer array, say p[r][n] is used to represent r queues, and
  - two 1-D integer arrays front[r] and rear[r] are used to represent the location of their front and rear elements.
- Initially all the elements of the arrays front and rear will contain -1 to indicate that the queues (pockets) are presently empty.



- In the 1st pass, the unit's digit of each key in the array a[n] is examined and inserted in the corresponding queue. When all the keys have been inserted in corresponding queues, the keys are deleted from each of the r queues in order and stored back in the array a[n]. Thus, after 1st pass, the keys will get sorted based on their unit's digits, e.g.

  70, 40, 20, 11, 61, 33,23, 73, 93, 29, 39, 19.

- The same process is then applied in the next pass and repeated *m* times (where *m* represents maximum no. of digits in a key) to sort the array a[n].

Radix Sort Algorithm (using arrays to represent queues)

- Let **a[n]** be the array to be sorted and **r** be the radix of the data values.
- Let **`p[r][n]`**, **`front[r]`** and **`rear[r]`** be the three arrays that represent *r* queues.

Algorithm `radix_sort(a,n,r)`

1. find the largest key of the data set
2. find maximum no. of digits m in the largest key
3. for pass←1 to m do
4. for i←0 to r-1 do ▷initialize all r queues
5. front[i]←rear[i]←-1
6. for i←0 to n-1 do ▷insert all n keys into appropriate queues based on passth digit
7. extract and check the passth digit of a[i] & insert it into corresponding queue
8. for i←0 to r-1 do ▷gather all keys from each queue back into the array a
9. delete all keys from ith queue and assign them to array a in order
10. ▷end of for loop (step 3)
11. return

Finding no. of digits m in largest key

```
m←0
repeat
   largest←largest/10
   m←m+1
until largest=0
```

largest=120 , initially m=0

largest= largest/10=12, m=m+1=1
largest= largest/10= 1, m=m+1=2
largest= largest/10= 0, **m**=m+1**=3**

Extracting passth digit from key

Formula: **digit=(key/10^{pass-1})%10**

Ex.1 **Key=1234**

In 1st pass, digit=(key/10^{0})%10 = 4
In 2nd pass, digit=(key/10^{1})%10 = 3
In 3rd pass, digit=(key/10^{2})%10 = 2
In 4th pass, digit=(key/10^{3})%10 = 1

Ex.2 **Key=4**

In 1st pass, digit=(key/10^{0})%10 = 4
In 2nd pass, digit=(key/10^{1})%10 =0%10 =0
In 3rd pass, digit=(key/10^{2})%10 =0%10 =0
….

Note: A C++ Program *radixsor.cpp* to demonstrate the sorting of a positive integer array in ascending order is given in *Programs* folder.

Drawback of Array representation of Queues

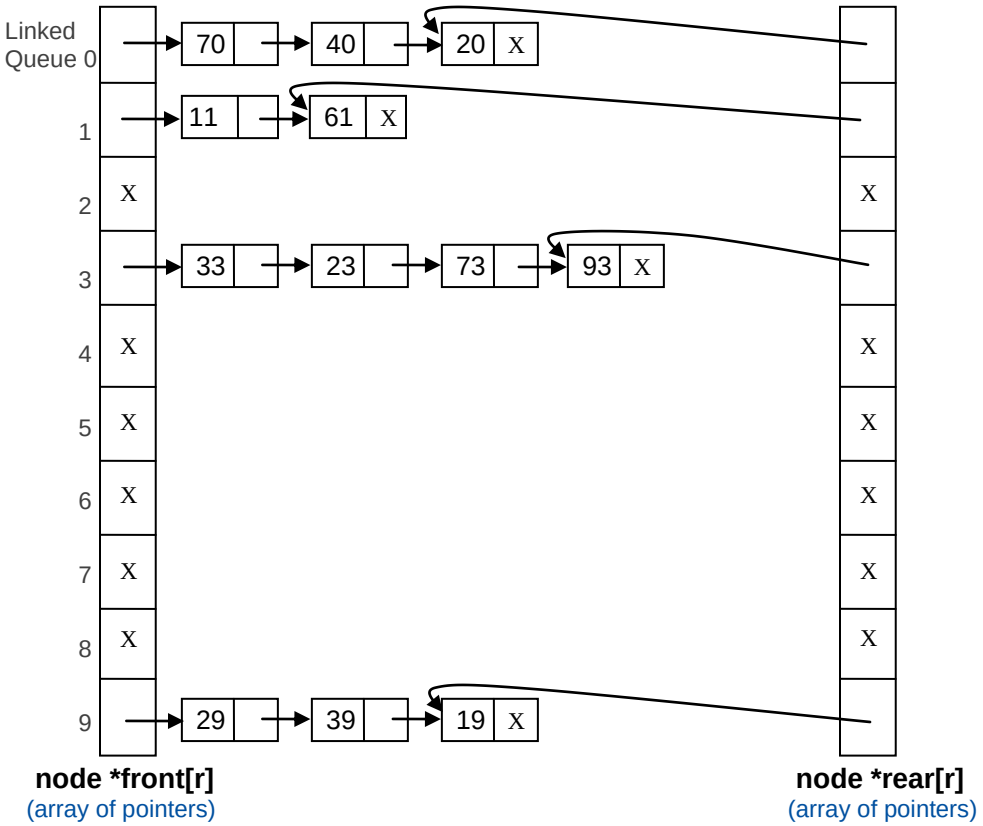
Wastage of storage:

In this representation, we need to use *r* arrays each of size *n* to represent r queues (pockets). This results in a lot of wastage of storage. This is because we can't predict how many elements will occupy a particular pocket during a certain pass.

Thus, this sequential allocation technique is not practical in representing the queues. This wastage of storage can be avoided if the queues are represented as linked queues in the following manner:

2. Using Linked Queues to represent Pockets

Each pocket is represented as a linked FIFO queue as shown below:



Using r=10 linked queues to represent 10 pockets

Comparison of running time of different sorting algorithms for a set of input values

n	**log n**	Selection/Bubble/ Insertion sort **$O(n^2)$**	Merge/Quick/ Heap sort **O(n log n)**	Radix sort **O(m*n)**
10	3.3	100	33	10*m
100	6.6	10,000	660	100*m
1,000	9.9	10^6	9,900	1000*m

Note: ***m*** indicates no. of passes (or maximum no. of digits in the largest key).

Heap Sort

This is an efficient sorting method and is based on a special kind of binary tree structure, called *heap*.

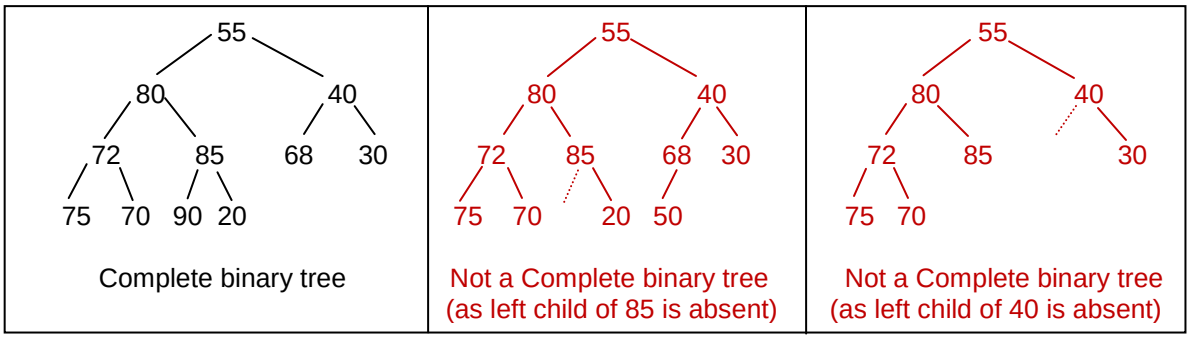
Heap: A heap is a *complete binary tree* in which the value at each node is ≥ the values at its successors (sub-trees).
This implies that the largest value is always in the root node of the heap. That's why a heap is also called a *maxheap*.

Complete binary tree: A binary tree is said to be complete, if

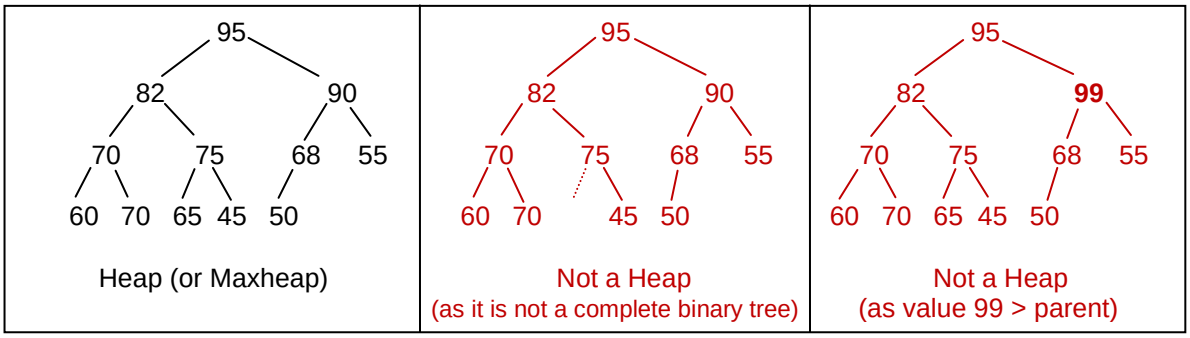
a) All its levels (except possibly the deepest level) are fully filled (i.e. they have maximum no. of possible 2^{level} nodes), and

b) If the deepest level is not fully filled, then the nodes of that level are filled strictly from left to right.

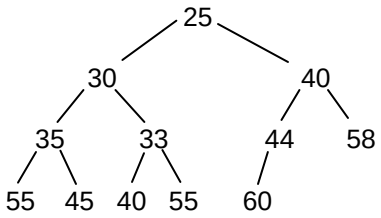
For example:



Following examples illustrate the concept of heap (or maxheap):



Minheap: A ***minheap*** is defined analogously: **t is a complete binary tree in which the value at each node is ≤ the values at its successors**. In a minheap, smallest value is always in the root node.



Minheap

Heap Sort: This sorting method **involves two phases**:

I. **Build a heap** out of the elements to be sorted.

II. **Repeatedly do** the following **until the heap is empty**:

1. **Eliminate the value at the root** from the heap **by interchanging it with the right-most element of the deepest level** to get the largest, 2^{nd} largest and so on element, and
2. **Rebuild the heap**.

Example: Suppose we want to sort the following integer values in ascending order using heap sort:

35, 42, 30, 60, 55, 80, 85, 55

Phase I: Build Heap

To represent the heap in memory, we will use a linear array a[8]. Initially the heap will be empty. Then we will build the heap by inserting the values to be sorted one by one in the heap.

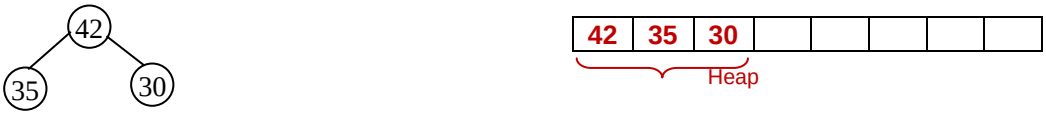
Steps to insert an element in the heap:

1. Insert the new element to the bottom level of the heap.
2. If inserted element ≤ its parent, stop.
3. If not, swap them and repeat step 2.

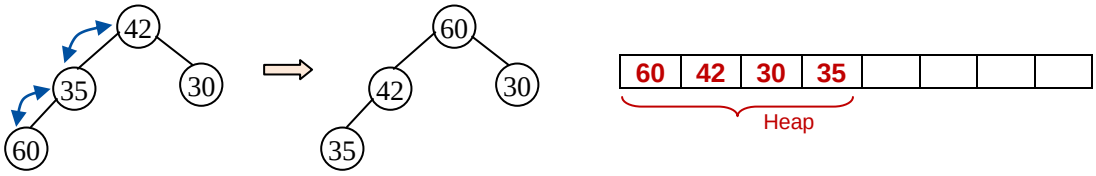
Now let's build the heap step by step. Insert first value i.e. 35 as root of the heap. A binary tree which contains only root node is a heap.

Now insert the second value i.e. 42 as left successor of the root. Since 42 > its parent i.e. 35, hence heap condition is violated which is fixed by interchanging 42 with 35. Now 42 and 35 form the heap.

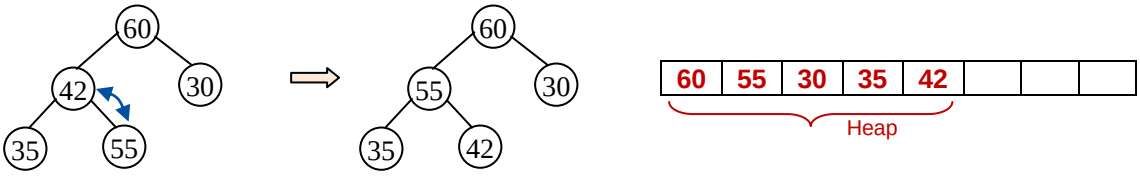
Now insert the third value i.e. 30 as right successor of the root. Since 30 ≤ its parent i.e. 42, hence heap condition is not violated. Now 42, 35 and 30 form the heap.



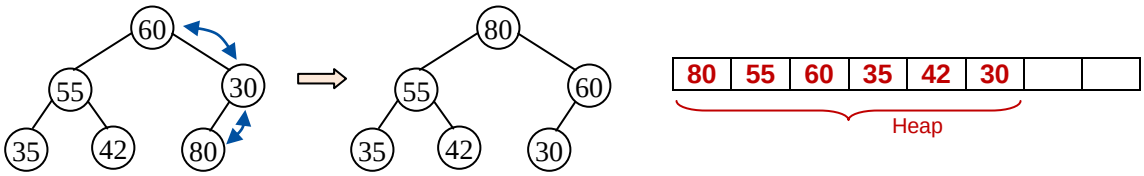
Now insert the next value i.e. 60 as left successor of 35. Since 60 > 35, hence interchange 60 with 35. Then compare 60 with its parent 42. Since 60 is also > 42, hence interchange 60 with 42 also.



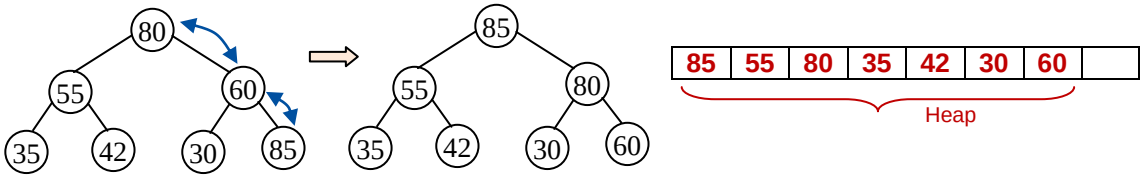
Now insert the next value i.e. 55 as right successor of 42. Since heap condition is violated, hence interchange 55 with its parent to fix the heap condition.



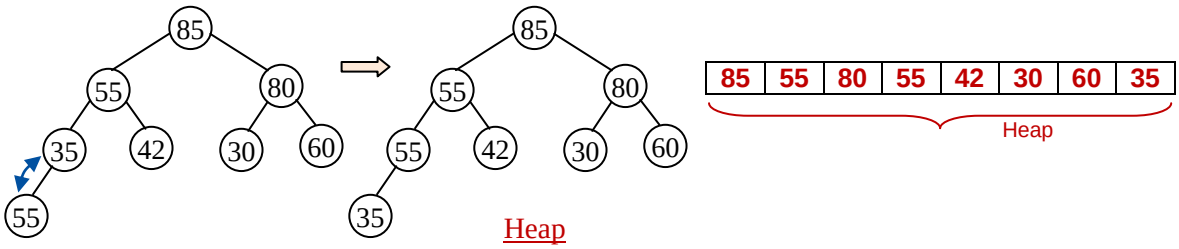
Now insertion of next value i.e. 80 also violates the heap condition which is fixed as below:



Now insertion of next value i.e. 85 also violates the heap condition which is fixed as below:



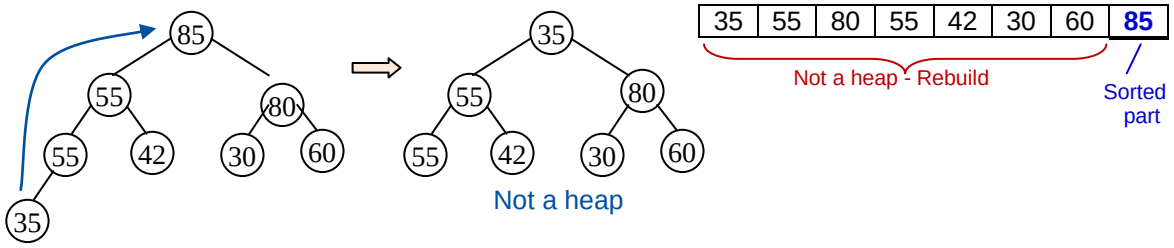
Now insertion of last value i.e. 55 also violates the heap condition which is fixed as below:



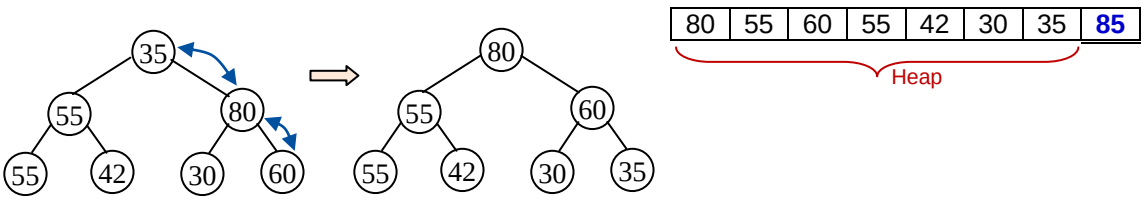
Phase II: **Eliminate the value at the root from the heap by interchanging it with the right-most element of the deepest level to get the largest value and rebuild the heap.**

Steps to rebuild the heap:

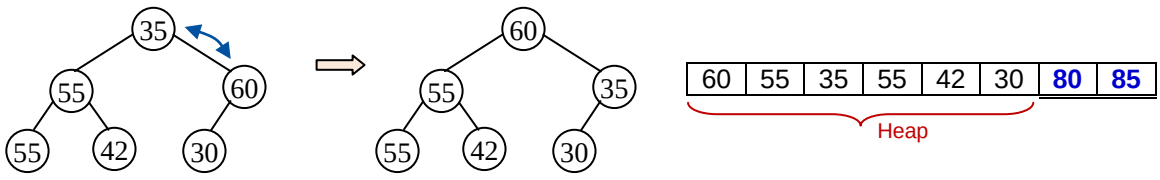
1. Compare the new root with its larger successor ▷smaller successor in case of minheap
2. If new root ≥ larger successor, stop. ▷for sorting in descending order
3. If not, swap them and repeat step 2.



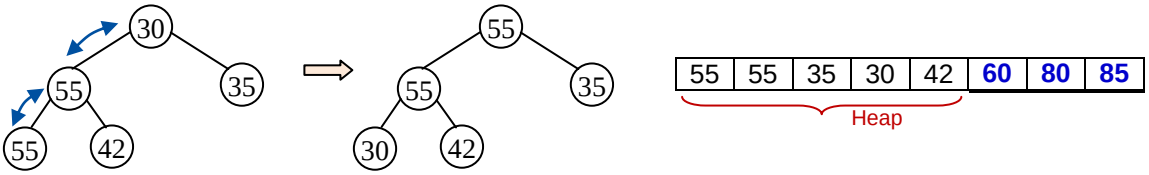
Thus 85 is eliminated from the heap by interchanging it with 35 to get the largest value. Now heap condition is violated which is fixed as below:



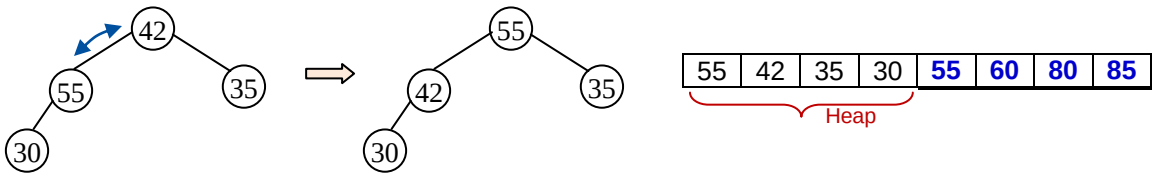
After rebuilding the heap, eliminate 80 from the heap by interchanging it with 35 to get the 2nd largest value. Now heap condition is violated which is fixed as below:



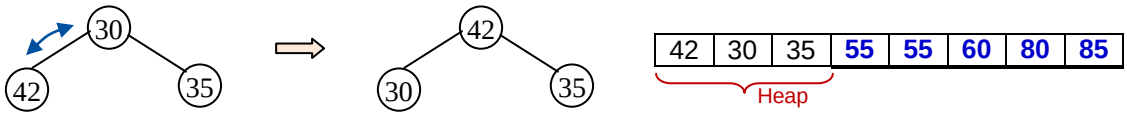
After rebuilding the heap, eliminate 60 from the heap by interchanging it with 30. Now heap condition is violated which is fixed as below:



After rebuilding the heap, eliminate 55 from the heap by interchanging it with 42. Now heap condition is violated which is fixed as below:



Now eliminate 55 by interchanging it with 30. Since heap condition is violated, hence rebuild heap.

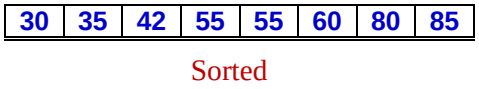


Now eliminate 42 by interchanging it with 35. Heap condition is not violated.

Now eliminate 35 by interchanging it with 30. Thus, heap now contains only the root node.

Eliminate the root to get the smallest value. Now the heap is empty.

Empty.



Sorted

Running time or Computational complexity

Performance of phase-I:

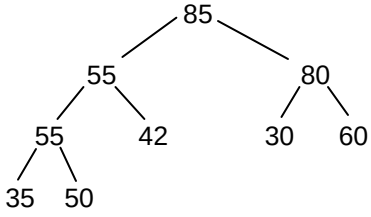
Number of comparisons to find appropriate place of a new element in the heap can't exceed the depth of heap. Since heap is a complete binary tree whose depth is bounded by $\lfloor \log n \rfloor$, hence Number of comparisons to insert n elements into the heap $f(n) \leq n \log n$

Consequently, the running time of phase-I is proportional to **n log n**.

Performance of phase-II:

Running time of phase-II (rebuilding the heap) is also proportional to **n log n**.

Hence, running time of Heap sort is O(n log n).



Heap (depth=3)

Note: A C++ Program *heapsort.cpp* to demonstrate the sorting of an integer array in ascending order is given in *Programs* folder.

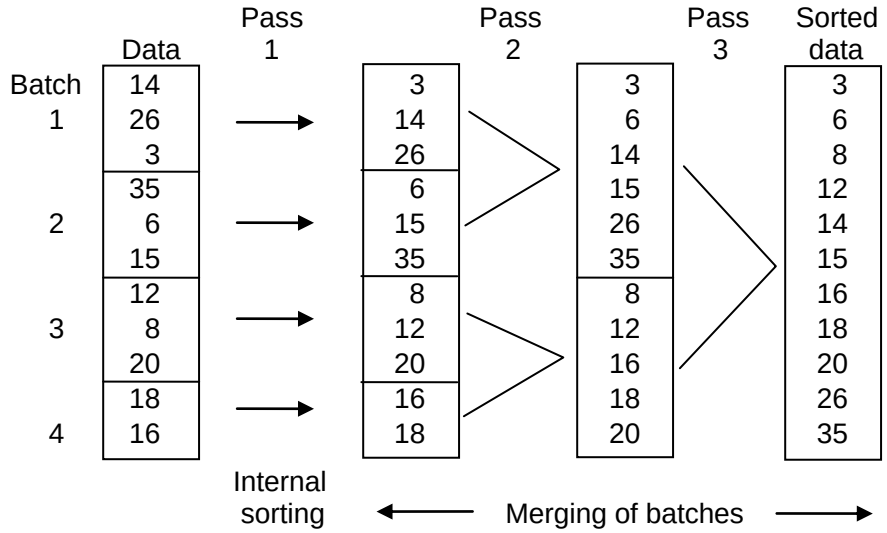
External Sorting

- External sorting is used when size of the data to be sorted is so large that it could not be accommodated in the main memory all at a time. External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, batches of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted subfiles are combined into a single larger file.
- Thus, this sorting approach requires some of the data to be stored in the main memory and some to be kept on a slower secondary storage (usually a disk) while sorting the data.

External sorting on disks

- The records of the table to be sorted are divided into small batches whose size depend upon how much internal memory is set aside to perform the internal sorting.
- These small batches are read into main memory one by one, sorted using an efficient internal sorting method and stored into intermediate file(s) on disk from where they are later retrieved and merged together to form fewer but larger batches.
- This process of merging of batches to form fewer but larger batches continues until we get a single batch which is the desired sorted table.

Example: Suppose we have to sort a table whose size is larger than the internal memory available for sorting. The keys of the records of the table are shown in the following figure. This table is sorted using external sorting method in the following manner:



Trace of external sorting on disk for batch length=3 and 2-way merge.
(If 4-way merge is used, then the table will be sorted after 2nd pass)

a) Suppose depending upon how much internal memory is available for sorting, the records are divided into four batches.

b) In pass1, these small batches are read into main memory one by one, sorted and stored into intermediate file(s) on disk.

c) In pass2, the sorted records of batch1 and batch2 are read in chunks one by one and merged together and stored in an intermediate file on the disk. Similarly, the sorted records of batch3 and batch4 are merged together and stored.

d) This process of merging of sorted batches to form fewer but larger batches is repeated in pass3. After completion of this pass, we get a single batch which is the desired sorted table.

Searching

- **Searching is the process of finding the location of a given element in a set of data.**
- The search is said to be successful if the given element is found (i.e. the element does exists in the array) and unsuccessful otherwise.
- Here, we will discuss two standard searching methods, namely:
    1. **Sequential (or Linear) search**
    2. **Binary search**

1. Sequential (or Linear) Search

- This is the simplest searching technique that is used with unsorted data.
- In sequential search, as its name implies, each array element is checked with the given element (key) to be searched in a sequential manner until the desired key is found or the array is exhausted.

Example: Suppose we have an unsorted linear array a[10] and want to find the location of the given element, say ***item***=55. In linear search, *item* is compared with the elements of the array sequentially starting with 0^{th} element and the searching process ends either when *item* is found or the array is exhausted. In this example, the *item* is found at location 2. The following figure illustrates this:

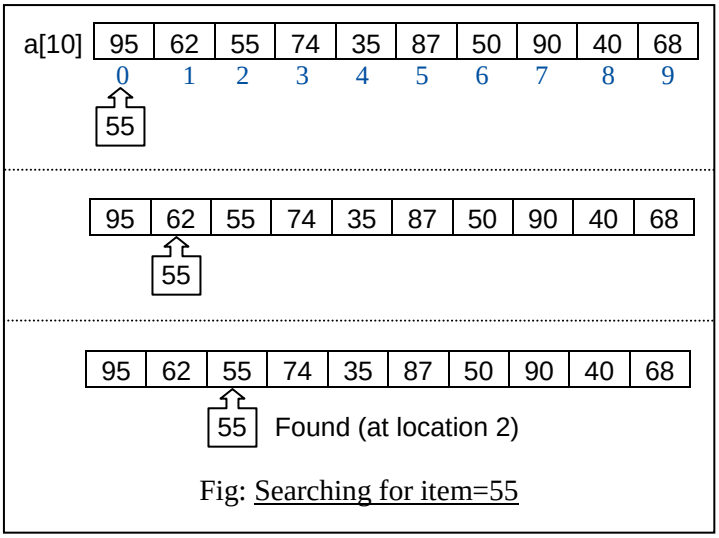


Fig: Searching for item=55

Below given Sequential Search algorithm searches the desired element *item* in unsorted linear array a[n].

Algorithm seq_search(a,n,item)

```
1. for i←0 to n-1 do
2.    if item=a[i] then
3.       return i          ▷search successful, return location
4. return -1               ▷search unsuccessful
```

C++ function for Sequential Search

Assume that we have defined the following *array* class which contains a dynamic linear array a[n] as its member:

```
class array
{ int *a;
  int n;
 public:
  void getdata(int b[],int size);
  int seq_search(int item);
  void disp();
  ~array() { delete [] a; }
};
```

The C++ function for Sequential Search is given below:

```
int array::seq_search(int item)
{ for(int i=0;i<=n-1;i++)
  { if(item==a[i])
      return i;
  }
  return -1;
}
```

Time complexity (Running time)

The running time of sequential (linear) search can be measured by counting the key comparisons performed to find the desired element in the array.

Best case: It occurs when the element being searched occurs at location 0 in the array.

No. of comparisons f(n) = 1 = **O(1).**

Worst case: When the element either occurs at the last (i.e. n-1th) location or is not present in the array at all. In either situation:

No. of comparisons f(n) = n = **O(n).**

Average case: **O(n).**

C++ program that implements sequential (linear) search and demonstrates its working

```
//seqsrch.cpp
#include <iostream>
using namespace std;
class array
{ int *a;
  int n;
 public:
  void getdata(int b[],int size);
  int seq_search(int item);
  void disp();
  ~array() { delete [] a; }
};
void array::getdata(int b[],int size)
{ n=size;
  a=new int[n];
  for(int i=0; i<n; i++)
     a[i]=b[i];
}
void array::disp()
{ for(int i=0; i<n; i++)
    cout<<a[i]<<' ';
  cout<<endl;
}
int array::seq_search(int item)
{ for(int i=0;i<=n-1;i++)
  { if(item==a[i])
       return i;
  }
  return -1;                          //search unsuccessful
}
int main()
{ const int n=10;
  int a[n],item,loc;
  array obj;
  cout<<"Enter "<<n<<" integer values to be stored in the array:\n";
  for(int i=0; i<n; i++)
     cin>>a[i];
  obj.getdata(a,n);
  cout<<"\nEnter the integer value to be searched: ";
  cin>>item;
  cout<<"\nArray contains following data elements:\n";
  obj.disp();
  loc=obj.seq_search(item);
  if(loc== -1)
    cout<<"\nGiven item "<<item<<" -does not occur in the array\n";
  else
    cout<<"\nGiven item "<<item<<" -occurs at loc: "<<loc<<" in the array"<<endl;
  return 0;
}
```

Test run

Enter 10 integer values to be stored in the array:
15 25 98 67 43 89 75 10 36 62

Enter the integer value to be searched: **36**

Array contains following data items:
15 25 98 67 43 89 75 10 36 62

Given value **36 -occurs at loc: 8** in the array

Binary Search

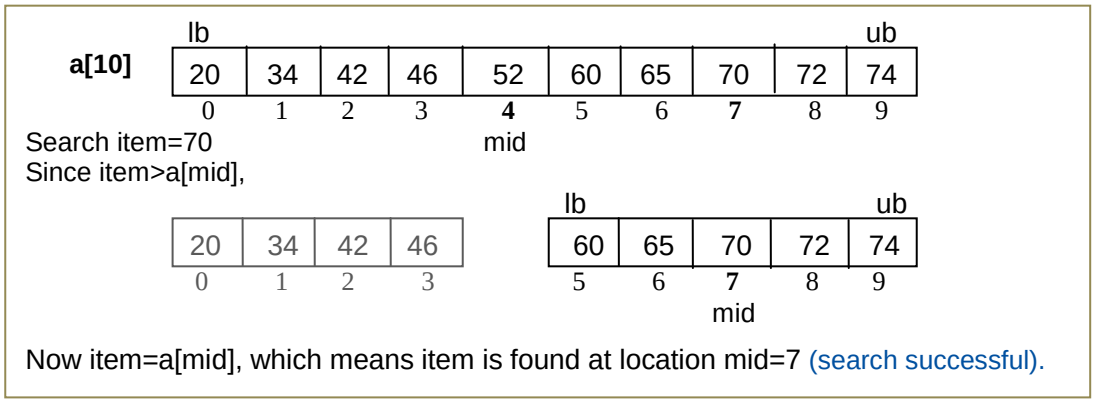
This is a very fast and efficient searching technique but it requires that the array (or table) must be in sorted order. That is, it can be used for finding the location of a particular element *item* in a sorted array or sorted table only.

Binary search finds the location of a given element *item* in a sorted array a[n] with lowerbound *lb* and upperbound *ub* in the following manner:

- The location of approximate middle element of the array is determined using the formula: mid=(lb+ub)/2.
- This middle element a[mid] is then compared with the element *item* to be searched. If it matches then the search is successful. Otherwise, the array is divided into two halves: one from lb to the mid-1th element (first half), and another from mid+1th element to the ub element (second half). Obviously, all the elements of 1st half are smaller than the middle element and all the elements of 2nd half are greater than the middle element. The searching then proceeds in either of the two halves depending upon whether item<a[mid] or item>a[mid] as below:
  - If item<a[mid], then item may exist only in the first half of the array. Therefore, 2nd half of the array is discarded and only 1st half is considered for continuing the searching operation. This is done by changing the value of ub=mid-1 and finding the new value of mid.
  - If item>a[mid], then item may exist only in the second half of the array. Therefore, further searching is restricted only to the second half of the array. This is done by changing the value of lb=mid+1 and finding the new value of mid.
  - This process of finding the location of approximate middle element and comparing it with item continues until the desired element is found (i.e. item=a[mid]) or division of the sub-array is not possible (i.e. lb>ub) which indicates that the element being searched does not occur in the array.

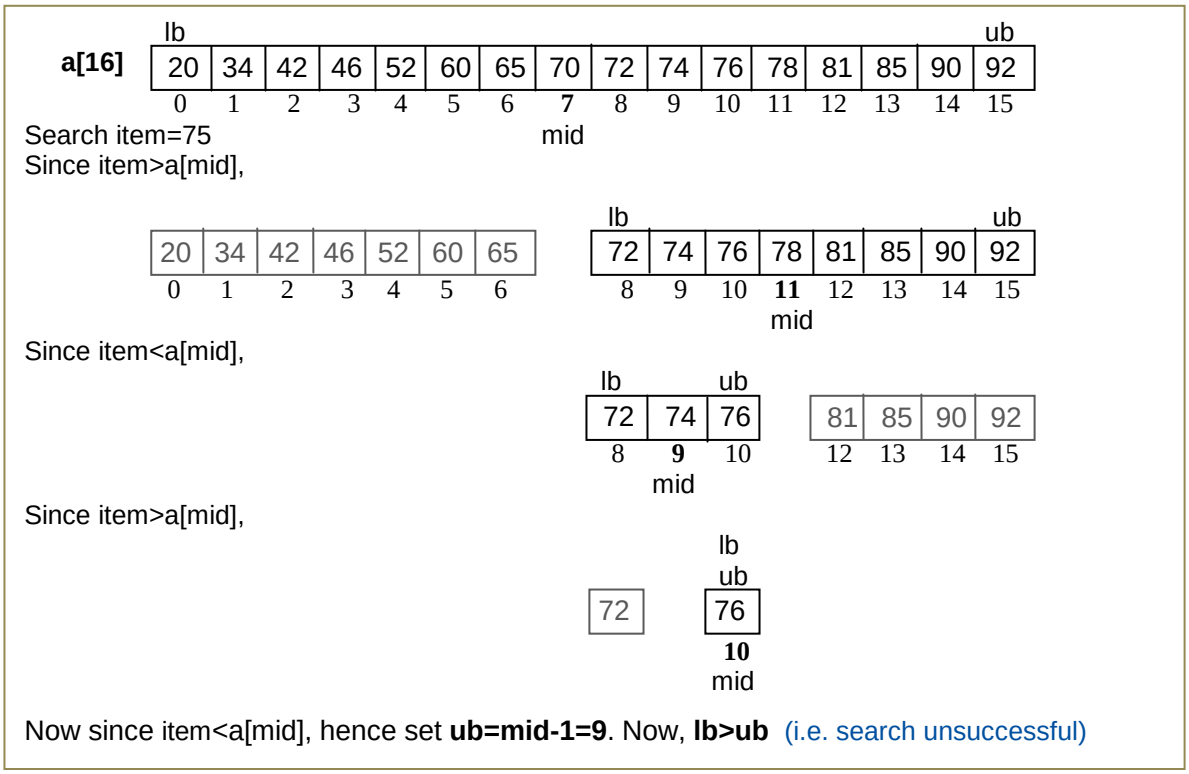
Example-1:

- Suppose we have a sorted array a[10] and we want to find the location of a given element *item*=70. The following figure illustrates the binary search operation:



Example-2:

- Similarly if we want to search for a given element *item*=75 in the following sorted array a[16]. The following figure illustrates the binary search operation:



Now since item<a[mid], hence set **ub=mid-1=9**. Now, **lb>ub** (i.e. search unsuccessful)

Below given Binary Search algorithm searches the desired element *item* in sorted linear array a[n].

Algorithm binary_search(a,n,item)

```
1.  lb←0
2.  ub←n-1
3.  while lb<=ub do
4.    mid←(lb+ub)/2
5.    if item=a[mid] then
6.       return mid                  ▷search successful
7.    else if item<a[mid] then
8.       ub←mid-1
9.    else
10.      lb←mid+1
11. ▷end of while loop
12. return -1                        ▷search unsuccessful
```

C++ Function for Binary Search

Assume that we have defined the following *array* class which contains a dynamic linear array a[n] as its member:

```
class array
{ int *a;
  int n;
 public:
  void getdata(int b[],int size);
  int binary_search(int item);
  void disp();
  ~array() { delete [] a; }
};
```

The C++ function for Binary search is given below:

```
int array::binary_search(int item)
{ int lb=0,ub=n-1,mid;
  while(lb<=ub)
  { mid=(lb+ub)/2;
    if(item==a[mid])
       return mid;
    else if(item<a[mid])
       ub=mid-1;
    else lb=mid+1;
  }
  return -1;
}
```

Limitations of Binary search

1. List must be sorted.
2. One must have direct access to the middle element in the list/sublist.

Time complexity (Running time)

Best case: It occurs when the item being searched is the approximate middle element of the array.

Number of comparisons f(n) =1 = **O(1).**

Worst case: It occurs when the item being searched is not present in the array, then the process of dividing the list in two halves continues until lb>ub. Hence the algorithm makes maximum number of comparisons.

Number of comparisons f(n) = $\lfloor \log n \rfloor + 1$ = **O(log n).**

Average case: **O(log n).**

Note: floor(x) = $\lfloor x \rfloor$ is the largest integer not greater than x (or largest-previous int).

Note: A C++ Program *binsrch.cpp* to demonstrate the searching of an item in an array using binary search algorithm is given in *Programs* folder.

C++ recursive function for Binary search

```
int array::binary_search(int lb,int ub,int item)
{ int mid;
  if (lb>ub)
     return -1;                    //search unsuccessful
  mid=(lb+ub)/2;
  if(item==v[mid])
     return mid;                   //search successful
  else if(item<v[mid])
    { ub=mid-1;
      return(binary_search(lb,ub,item));
    }
  else
    { lb=mid+1;
      return(binary_search(lb,ub,item));
    }
}
```

Table showing running times of sorting and searching algorithms

Algorithm	Best case	Average case	Worst case
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Merge sort	O(n log n)	O(n log n)	O(n log n)
Quick sort	O(n log n)	O(n log n)	$O(n^2)$
Heap sort	O(n log n)	O(n log n)	O(n log n)
Radix sort	O(m*n)	O(m*n)	O(m*n)
Linear search	O(1)	O(n)	O(n)
Binary search	O(1)	O(log n)	O(log n)

Note: ***m*** indicates no. of passes (or no. of digits in the largest key).

****