Sorting

Sorting is a process that organizes a collection of data into either ascending or descending order. Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.There are many sorting algorithms, such as:

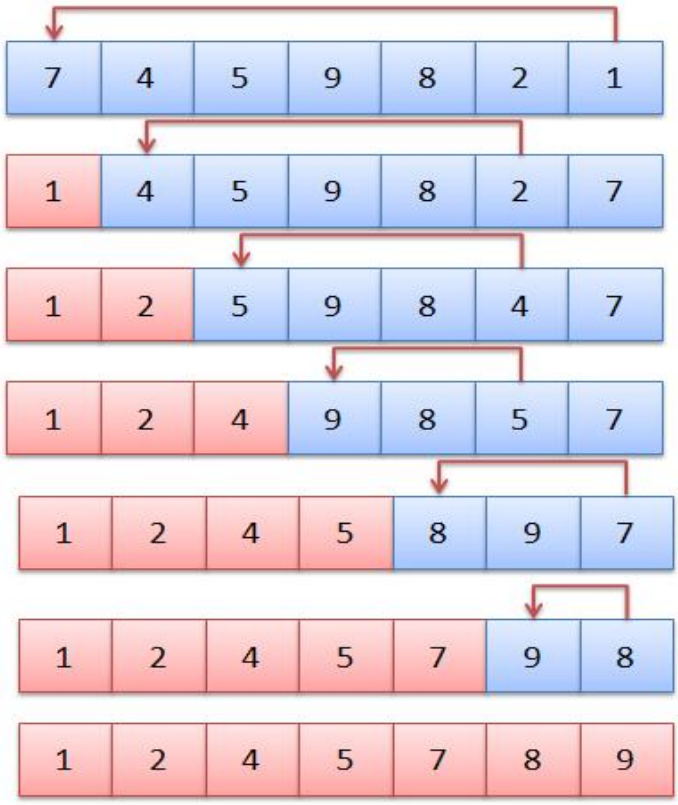
1. Selection Sort
2. Insertion Sort
3. Bubble Sort
4. Merge Sort

Selection Sort:
In selection sort we first check for smallest element in the list and swap it with the first element of the list. Again, we check for the smallest number in a sublist, excluding the first element of the list as it is where it should be (at the first position) and put it in the second position of the list. We continue repeating this process until the list gets sorted.

Selection Sort Algorithm

1. Start from the first element in the list and search for the smallest element in the list.
2. Swap the first element with the smallest element of the list.
3. Take a sublist (excluding the first element of the list as it is at its place) and search for the smallest number in the sublist (second smallest number of the entire list) and swap it with the first element of the list (second element of the entire list).
4. Repeat the steps 2 and 3 with new sublists until the list gets sorted.

Working of selection sort:



Selection Sort Source Code

```
mylist = []
n=input("Number of elements to insert")
for i in range(0,n):
        mylist.append(input("Enter a number"))
for i in range(len(mylist)-1):
    minpos=i
    for j in range(i+1,len(mylist)):
       if mylist[minpos]>mylist[j]:
          minpos = j
    temp = mylist[i]
    mylist[i] = mylist[minpos]
    mylist[minpos] = temp
print(mylist)
```

Time Complexity

- Selection sort best case: $O(n^2)$
- Selection sort worst case: $O(n^2)$
- Selection sort average case: $O(n^2)$

Advantages	**Disadvantages**
The main advantage of the selection sort is that it performs well on a small list.	The primary disadvantage of the selection sort is its poor efficiency when dealing with a huge list of items.
Because it is an in-place sorting algorithm, no additional temporary storage is required beyond what is needed to hold the original list.	The selection sort requires n-squared number of steps for sorting n elements.
Its performance is easily influenced by the initial ordering of the items before the sorting process.	insertion Sort is much more efficient than selection sort

Insertion Sort

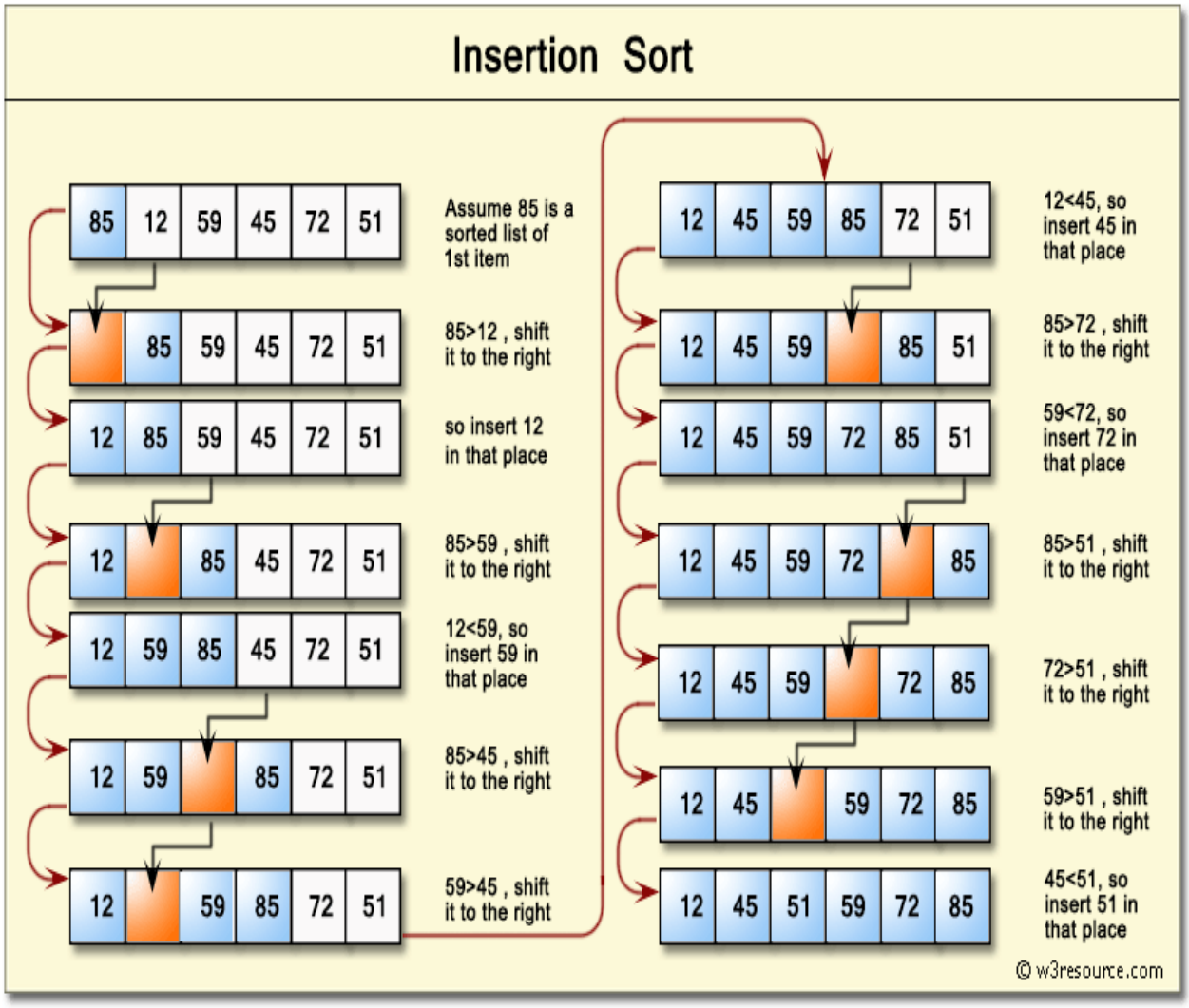
As the name suggests, in Insertion Sort, an element gets compared and inserted into the correct position in the list. To apply this sort, you must consider one part of the list to be sorted and the other to be unsorted. To begin, consider the first element to be the sorted portion and the other

elements of the list to be unsorted. Now compare each element from the unsorted portion with the element/s in the sorted portion. Then insert it in the correct position in the sorted part. Remember, the sorted portion of the list remain sorted at all times.

Insertion Sort Algorithm

1. Consider the first element to be sorted and the rest to be unsorted
2. Compare with the second element:
    1. If the second element< the first element, insert the element in the correct position of the sorted portion
    2. Else, leave it as it is
3. Repeat 1 and 2 until all elements are sorted

Working of Insertion sort:



Insertion Sort Source code

mylist = []

```
n=input("Number of elements to insert")

for i in range(0,n):

        mylist.append(input("Enter a number"))

for i in range(1,len(mylist)):
    currentvalue = mylist[i]
    position = i
    while position>0 and mylist[position-1]>currentvalue:
      mylist[position]=mylist[position-1]
      position = position-1
    mylist[position]=currentvalue
print(mylist)
```

Time Complexity

- Best-case:O(n)
- Worst-case:O(n 2)
- Average-case:O(n 2)

Advantages	**Disadvantages**
The main advantage of the insertion sort is its simplicity.	The disadvantage of the insertion sort is that it does not perform as well as other, better sorting algorithms
It also exhibits a good performance when dealing with a small list.	With n-squared steps required for every n element to be sorted, the insertion sort does not deal well with a huge list.
The insertion sort is an in-place sorting algorithm so the space requirement is minimal.	The insertion sort is particularly useful only when sorting a list of few items.

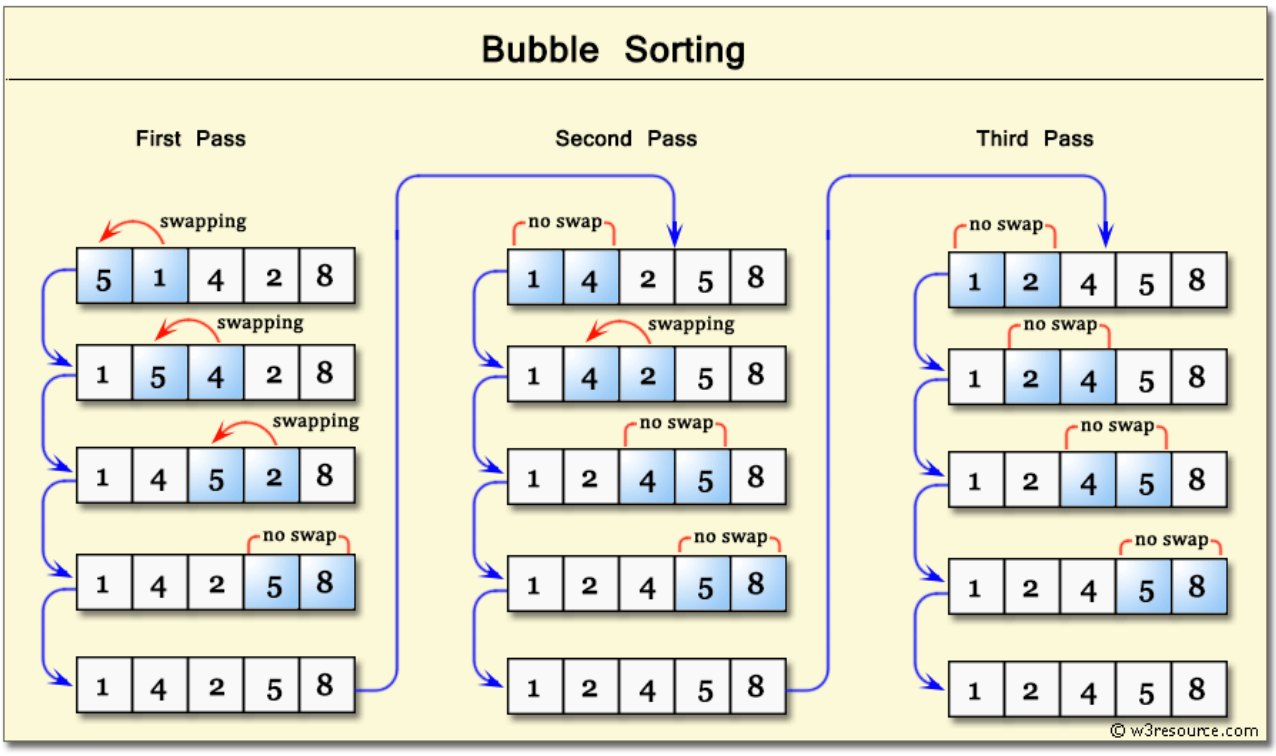
Bubble Sort

Bubble sort is one of the simplest sorting algorithms. The two adjacent elements of a list are checked and swapped if they are in wrong order and this process is repeated until we get a sorted list.

Bubble sort Algorithm

- Compare the first and the second element of the list and swap them if they are in wrong order.
- Compare the second and the third element of the list and swap them if they are in wrong order.
- Proceed till the last element of the list in a similar fashion.
- Repeat all of the above steps until the list is sorted.

Working of bubble sort:



Bubble Sort Source Code

```
mylist = []
n=input("Number of elements to insert")
for i in range(0,n):
        mylist.append(input("Enter a number"))
for i in range(0,n-1):
    for j in range(0,n-i-1):
        if mylist[j]>mylist[j+1]:
            temp = mylist[j]
            mylist[j] = mylist[j + 1]
            mylist[j + 1] = temp
print(mylist)
```

Time Complexity

- Best-case:O(n)
- Worst-case:O(n 2)
- Average-case:O(n 2)

Advantages	**Disadvantages**
The primary advantage of the bubble sort is that it is popular and easy to implement.	The main disadvantage of the bubble sort is the fact that it does not deal well with a list containing a huge number of items.
In the bubble sort, elements are swapped in place without using additional temporary storage.	The bubble sort requires n-squared processing steps for every n number of elements to be sorted.

The space requirement is at a minimum	The bubble sort is mostly suitable for academic teaching but not for real-life applications.

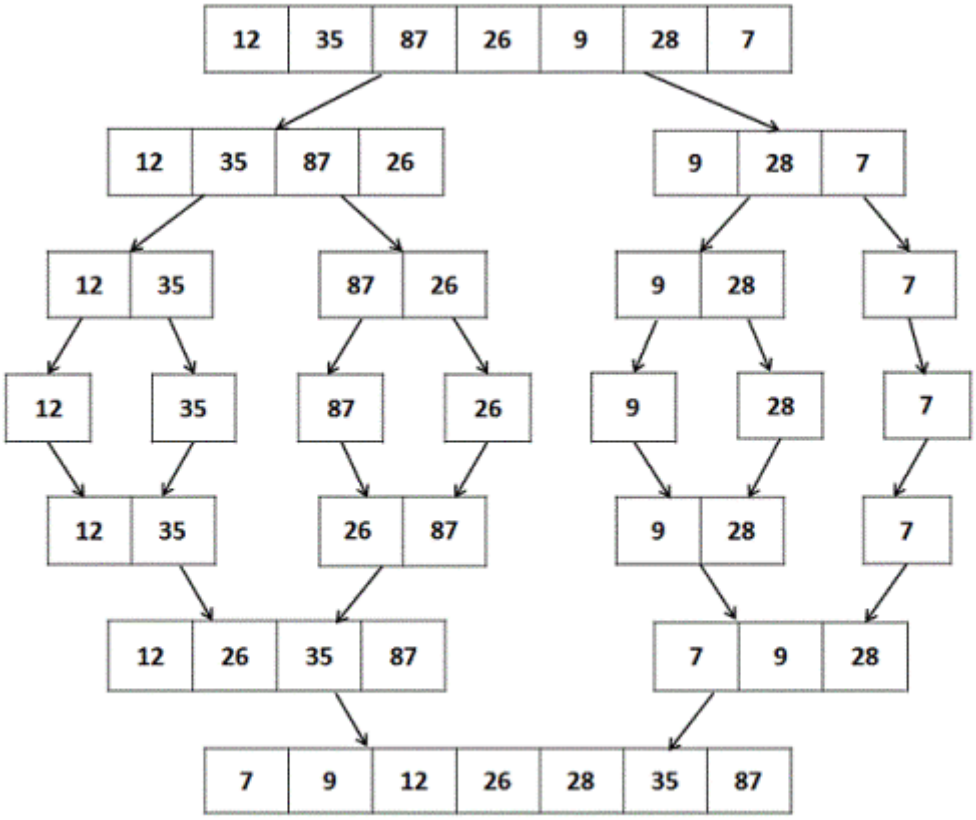
Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves.

Merge sort Algorithm

1. We are given an unsorted list of *n* numbers
2. Divide it into two array of equal size
3. Again Divide those two into another two array of equal size
4. Keep on dividing it until one element is left in each array.
5. Keep Merging the sub arrays to produce sorted arrays until there is only one array remaining.. This will be our sorted array.

Working of Merge sort:



Merge Sort

Merge sort Source code

```
def mergeSort(alist):
  if len(alist)>1:
     mid = len(alist)/2
     lefthalf = alist[:mid]
```

```
    righthalf = alist[mid:]
    mergeSort(lefthalf)
    mergeSort(righthalf)
    i,j,k=0,0,0
    while i < len(lefthalf) and j < len(righthalf):
       if lefthalf[i] < righthalf[j]:
          alist[k]=lefthalf[i]
          i=i+1
       else:
          alist[k]=righthalf[j]
          j=j+1
       k=k+1
    while i < len(lefthalf):
       alist[k]=lefthalf[i]
       i=i+1
       k=k+1
    while j < len(righthalf):
       alist[k]=righthalf[j]
       j=j+1
       k=k+1
alist = [1,5,2,0,7,9]
mergeSort(alist)
print(alist)
```

Time Complexity

- Best case : O (n log n)
- Average case : O(n log n)
- Worst case: O(n log n)

Advantages	**Disadvantages**
It can be applied to files of any size.	Requires extra space»N
Reading of the input during the run-creation step is sequential ==> Not much seeking.	Merge Sort requires more space than other sort.
If heap sort is used for the in-memory part of the merge, its operation can be overlapped with I/O	Merge sort is less efficient than other sort

Searching

Searching is a process of finding an object among a group of objects Here the element is also referred to as ‘Key’ Searching is one of the important application of Arrays i.e., we can search a particular element present in a list or in a record**.**
There are 2 important Searching Techniques
1. Linear Search [Sequential Search]
2. Binary Search [Logarithmic Search]

Necessary components to search a list of data

- Array containing the list
- Length of the list

- Item for which you are searching

After search completed

- If item found, report “success,” return location in array
- If item not found, report “not found” or “failure”

Linear/Sequential searching

The linear search is used to find an item in a list. The items do not have to be in order. To search for an item, start at the beginning of the list and continue searching until either the end of the list is reached or the item is found.

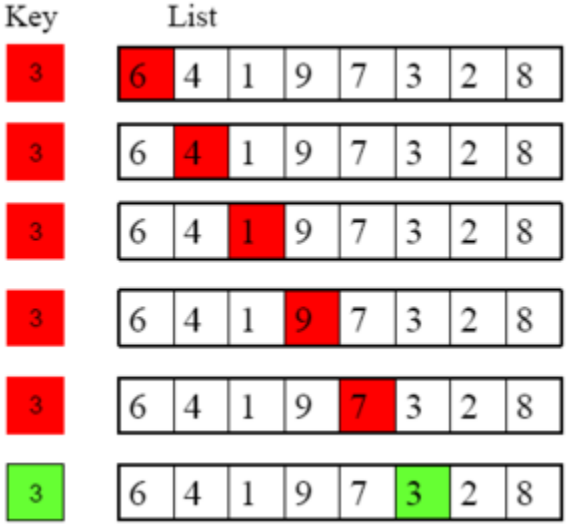
Linear Search Algorithm

Step 1: Start at first element of array – Search Element.
Step 2: Compare value to value (key) for which you are searching
Step 3: Continue with next element of the array until you find a match or reach the last element in the array. (iteration)
Step 4: Return Failure , if element not found in the list

Performance of the Linear Search

- Suppose that the first element in the array list contains the variable key, then we have performed one comparison to find the key.
- Suppose that the second element in the array list contains the variable key, then we have performed two comparisons to find the key.
- Carry on the same analysis till the key is contained in the last element of the array list. In this case, we have performed N comparisons (where N is the size of the array list) to find the key.
- Finally if the key is NOT in the array list, then we would have performed N comparisons and the key is NOT found and we would return -1.

Working of Linear Search



Linear Search Source Code

```
mylist=[]
c=0
```

```
n=input("Enter number of elements")
for i in range(0,n):
        mylist.append(input("Enter a number"))
search=input("Enter a number to search")
for i in range(0,n):
        if(mylist[i]==search):
                print"Element {} found at position {}".format(search,i)
                c=1
                break
if(c==0):
        print "not found"
```

Linear search Time Complexity

- Best-case:O(1)
- Worst-case:O(n)
- Average-case:O(n)

Advantages

- It is easy to understand
- Easy to implement
- Does not require the array to be in sorted order

Disadvantages

- If there are 20,000 items in the array and what you are looking for is in the 19,999 th element, you need to search through the entire list.

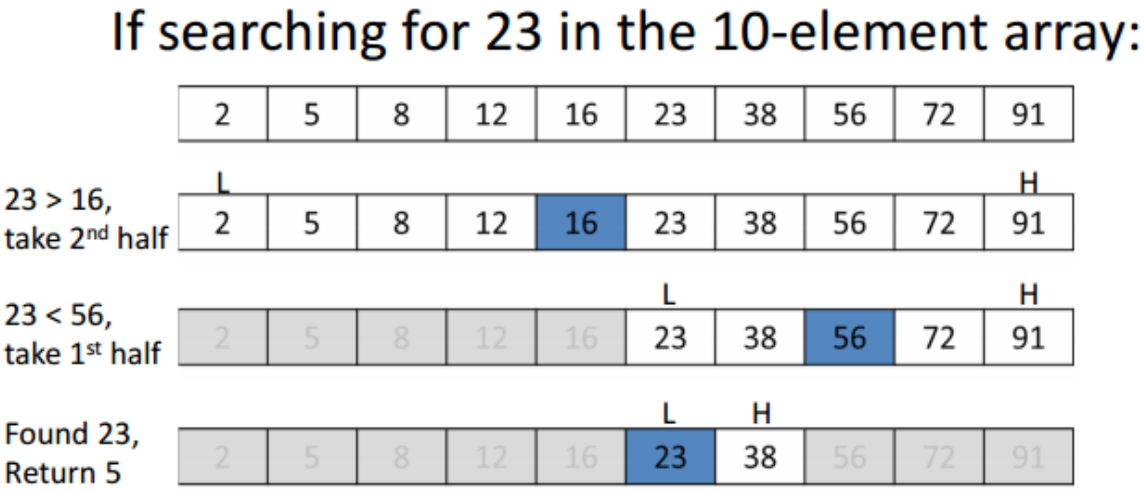
Binary search

- Search as the name suggests, is an operation of finding an item from the given collection of items. Binary Search algorithm is used to find the position of a specified value (an „Input Key) given by the user in a sorted list.
- Binary Search can only be performed if the data is sorted, whether it is in form of a Array, list or any other structure
- This concept is generally used by Electricians to locate a fused bulb in a serial set & in searching a dictionary, phonebook
- The concept is splitting the list into two halves & then checking the middle item
- Binary Search is also referred to as a Logarithmic Search

Binary search algorithm

Step 1: It starts with the middle element of the list.
Step 2: If the middle element of the list is equal to the „input key then we have found the position the specified value.
Step 3: Else if the „input key is greater than the middle element then the „input key‟ has to be present in the last half of the list.
Step 4: Or if the „input key is lesser than the middle element then the input key has to be present in the first half of the list.
Step 5: Failure if element not found or empty

Working of Binary Search



Binary Search Source Code

```
mylist=[]
c=0
n=input("Enter number of elements")
for i in range(0,n):
        mylist.append(input("Enter a number"))
search=input("Enter a number to search")
mylist.sort()
first=0
last=n-1
mid=(first+last)/2
while(first <= last):
        if(search>mylist[mid]):
                first=mid+1
        elif(search==mylist[mid]):
                print "Element {} found at poition {}".format(search,mid)
                break
        else:
                last=mid-1
        mid=(first+last)/2
if(first>last):
        print "Search elements {} not found".format(search)
```

Binary Search Time Complexity

- Best-case:O(1)
- Worst-case:O(log n)
- Average-case:O(log n)

Advantages

- Compared to linear search (checking each element in the array starting from the first), binary search is *much* faster.
- It's a fairly simple algorithm, though people get it wrong all the time.It's well known and often implemented for you as a library routine.

Disadvantages

- It’s more complicated than linear search, and is overkill for very small numbers of elements.
- It works only on lists that are sorted and kept sorted. That is not always feasable, especially if elements are constantly being added to the list.