Chapter 2

Searching:

- Searching is the process of finding a given value position in a list of values.
- It decides whether a search key is present in the data or not.
- It is the algorithmic process of finding a particular item in a collection of items.
- It can be done on internal data structure or on external data structure.

There are two popular algorithms available:

1. Linear Search
2. Binary Search

1. Linear Search/ Sequential Search:

- Sequential search is also called as Linear Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

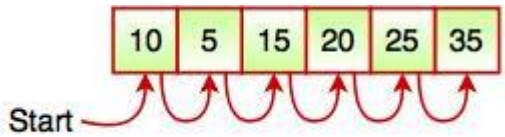


Fig. Sequential Search

The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

Algorithm

```
LinearSearch ( Array A, Value x)
Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Linear Search Example

Let us take an example of an array A[7]={5,2,1,6,3,7,8}. Array A has 7 items. Let us assume we are looking for 7 in the array. Targeted item=7.

Here, we have

A[7]={5,2,1,6,3,7,8}

X=7

At first, When i=0 (A[0]=5; X=7) not matched

i++ now, i=1 (A[1]=2; X=7) not matched

i++ now, i=2(A[2])=1; X=7)not matched

…

….

i++ when, i=5(A[5]=7; X=7) Match Found

Hence, Element X=7 found at index 5.

Linear search is rarely used practically. The time complexity of above algorithm is **O(n)**.

```
int linearSearch(int values[], int target, int n)
{
   for(int i = 0; i < n; i++)
   {
      if (values[i] == target)
      {
         return i;
      }
   }
   return -1;
}
```

Program for Sequential Search

```
#include <stdio.h>
int main()
{
  int arr[50], search, cnt, num;

  printf("Enter the number of elements in array\n");
  scanf("%d",&num);
```

```
  printf("Enter %d integer(s)\n", num);

  for (cnt = 0; cnt < num; cnt++)
  scanf("%d", &arr[cnt]);

  printf("Enter the number to search\n");
  scanf("%d", &search);

  for (cnt = 0; cnt < num; cnt++)
  {
    if (arr[cnt] == search)     /* if required element found */
    {
      printf("%d is present at location %d.\n", search, cnt+1);
      break;
    }
  }
  if (cnt == num)
    printf("%d is not present in array.\n", search);

  return 0;
}
```

2. Binary Search Algorithm

- Binary Search is used for searching an element in a sorted array.
- It is a fast search algorithm with run-time complexity of O(log n).
- Binary search works on the principle of divide and conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.
- It is useful when there are large number of elements in an array.

In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Binary Search is useful when there are large number of elements in an array and they are sorted.

So a necessary condition for Binary search to work is that the list/array should be sorted.

- The above array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.

For example, if searching an element 25 in the 7-element array, following figure shows how binary search works:

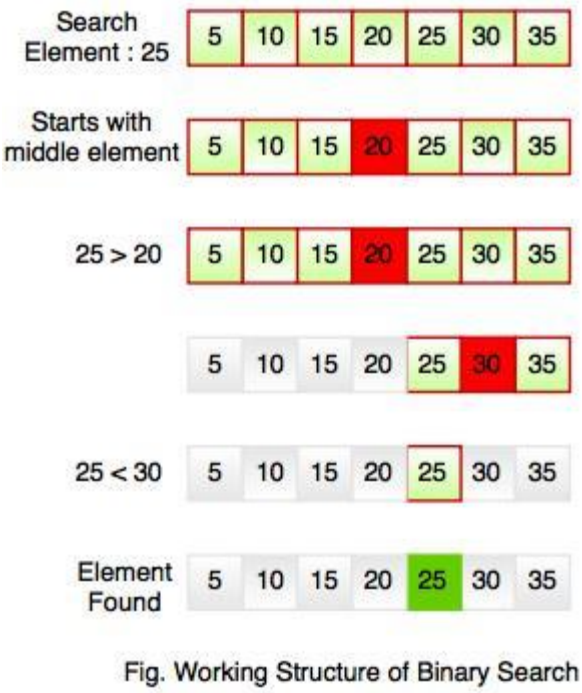


Fig. Working Structure of Binary Search

Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the left of the list. Repeat this, till you find an element.

Features of Binary Search

1. It is great to search through large sorted arrays.
2. It has a time complexity of O(log n) which is a very good time complexity. We will discuss this in details in the Binary Search tutorial.
3. It has a simple implementation.

Algorithm

Step 1: Data list must be ordered list in ascending order.
Step 2: Probe middle of list
Step 3: If target equals list[mid], FOUND.
Step 4: If target < list[mid], discard 1/2 of list between list[mid] and list[last].

Step 5: If target > list[mid], discard 1/2 of list between list[first] and list[mid].
Step 6: Continue searching the shortened list until either the target is found, or there are no elements t o probe.

Binary Search Example

Let us take an example of an array A[16]={1,2,3,4,6,7,8,10,12,13,15,16,18,19,20,22}. The array is sorted and contains 16 items. We are looking for item 19 in this list.

Here, we have

A[16]={1,2,3,4,6,7,8,10,12,13,15,16,18,19,20,22}

X=19

Let us first divide it into two smaller arrays of 8 items each. The first 8 items in a sub array and another 8 in another sub array. A1={1,2,3,4,6,7,8,10} and A2={12,13,15,16,18,19,20,22}

As 10 and 12 are the middle items of this ordered array, we know 19 is greater than the middle numbers that means we don't require to look for the targeted item in first sub array. Let us search in the second subarray A2={12,13,15,16,18,19,20,22}

In the second array we have 8 items. Let's divide them into two equal arrays and check the middle items. Here the middle items are 16 and 18. As 19 is greater than both of them, we don't need to look in the first four items of this subarray. A21={12,13,15,15} and A22={18,19,20,22}

Let us look in the last four items sub array A22={18,19,20,22}. Let us again divide it to subarrays A221={18,19) and A222={20,22}. Here the middle items are 19 and 20. Hence, we found the target item 19 in first subarray.

Program for Binary Search

```
#include<stdio.h>
#include<conio.h>

void main()
{
  int f, l, m, size, i, sElement, list[50]; //int f, l ,m : First, Last, Middle
  clrscr();

  printf("Enter the size of the list: ");
  scanf("%d",&size);

  printf("Enter %d integer values : \n", size);

  for (i = 0; i < size; i++)
    scanf("%d",&list[i]);

  printf("Enter value to be search: ");
  scanf("%d", &sElement);
```

```
  f = 0;
  l = size - 1;
  m = (f+l)/2;

  while (f <= l) {
    if (list[m] < sElement)
      f = m + 1;
    else if (list[m] == sElement) {
      printf("Element found at index %d.\n",m);
      break;
    }
    else
      l = m - 1;
    m = (f + l)/2;
  }
  if (f > l)
    printf("Element Not found in the list.");
  getch();
}
```

Sorting:

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order.

Bubble Sort

Bubble sort is the simplest sorting algorithm. It is based on comparison where each adjacent pair of element is compared and swapped if they are not in order. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. This algorithm is not suitable for huge data sets. Average and worst case time complexity of this algorithm are of **O(n2)** where n is the number of items.

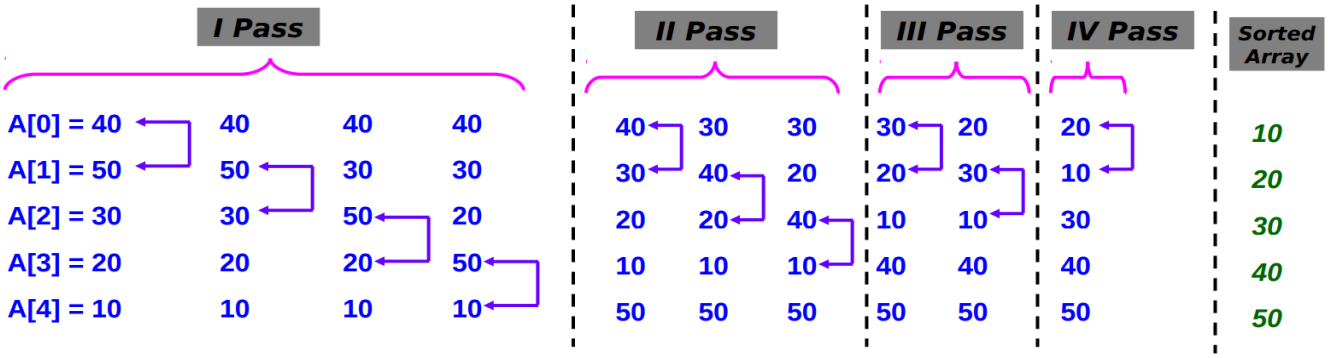
Algorithm

```
for i=N-1 to 2 {
set swap flag to false
for j=1 to i {
if list[j-1] > list[j]
swap list[j-1] and list[j]
set swap flag to true
}
if swap flag is false, break. The list is sorted.
}
```

[NOTE: In each pass, the largest item "bubbles" down the list until it settles in its final position. This is where bubble sort gets its name.]

Example,

Suppose we have a list of array of 5 elements A[5]={40,50,30,20,10}. We have to sort this array using bubble sort algorithm.



We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. After every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements.

```
void bubbleSort(int arr[], int n)
{
   int i, j, temp;
   for(i = 0; i < n; i++)
   {
      for(j = 0; j < n-i-1; j++)
      {
         if( arr[j] > arr[j+1])
         {
// swap the elements
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
}
```

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.

Selection Sort

Selection sort is an in-place comparison sort algorithm. In this algorithm, we repeatedly select the smallest remaining element and move it to the end of a growing sorted list. It is one of the simplest sorting algorithm. Selection sort is known for its simplicity. It has performance advantages over more complicated algorithms in certain situations.

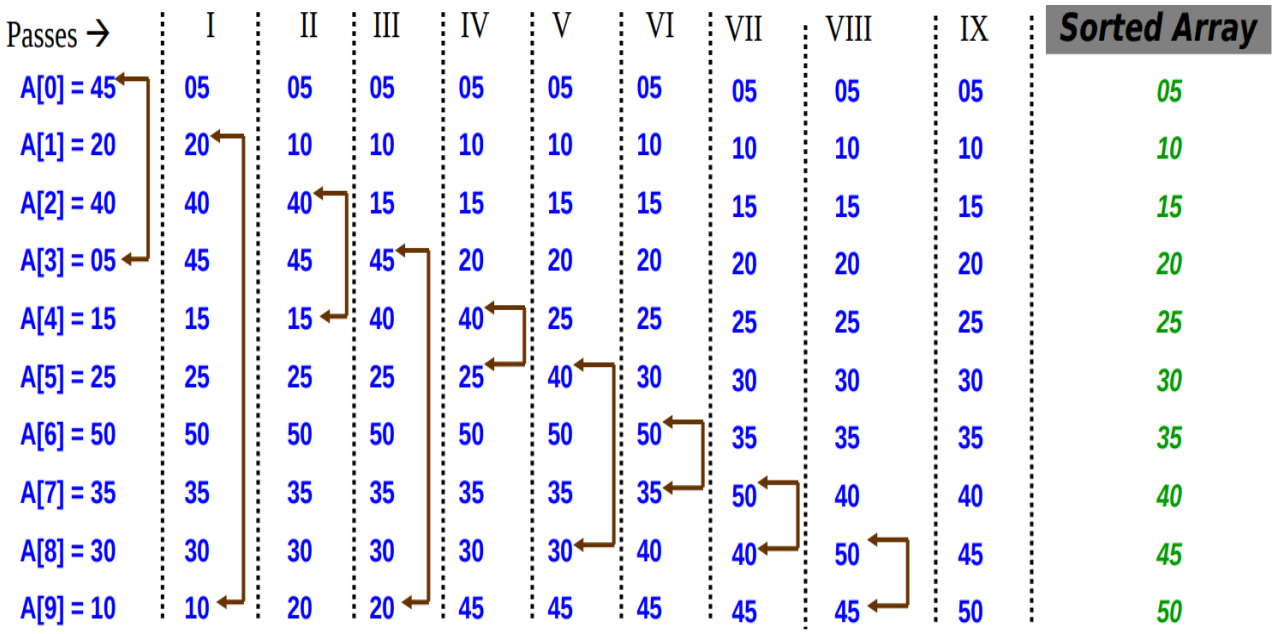
This algorithm is not suitable for large data sets as its average and worst case complexities are of **O(n2)**, where n is the number of items.

Algorithm

Step 1: Set MIN to location 0
Step 2: Search the minimum element in the list
Step 3: Swap with value at location MIN
Step 4: Increment MIN to point to next element
Step 5: Repeat until list is sorted

Example,

Let us assume an array A[10]={45,20,40,05,15,25,50,35,30,10}. We have to sort this array using selection sort.



In this algorithm we have to find the minimum value in the list first. Then, Swap it with the value in the first position. After that, Start from the second position and repeat the steps above for remainder of the list.

```
// function to look for smallest element in the given subarray
int indexOfMinimum(int arr[], int startIndex, int n)
{
   int minValue = arr[startIndex];
   int minIndex = startIndex;
   for(int i = minIndex + 1; i < n; i++)
   {
```

```
      if(arr[i] < minValue)
      {
         minIndex = i;
         minValue = arr[i];
      }
   }
   return minIndex;
}

void selectionSort(int arr[], int n)
{
   for(int i = 0; i < n; i++)
   {
      int index = indexOfMinimum(arr, i, n);
      swap(arr, i, index);
   }
}
```

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.	Quick Sort is much more efficient than selection sort

Insertion Sort

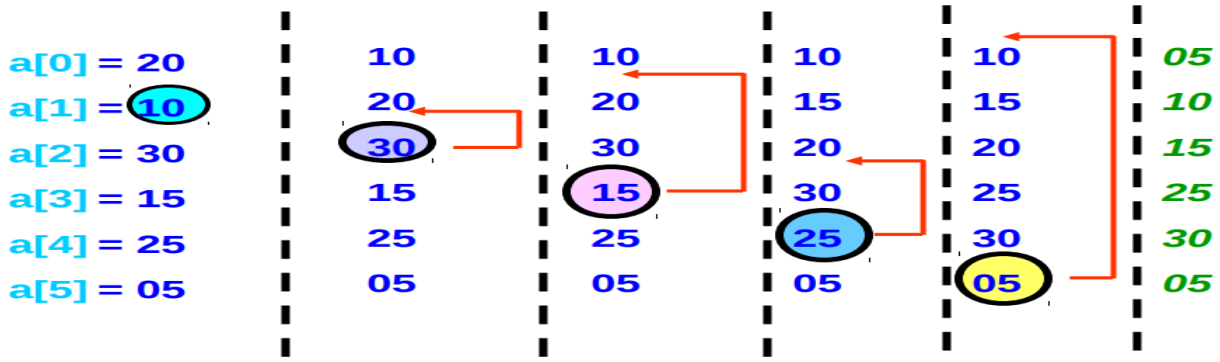
Insertion sort is an in-place sorting algorithm based on comparison. It is a simple algorithm where a sorted sub list is maintained by entering on element at a time. An element which is to be inserted in this sorted sub-list has to find its appropriate location and then it has to be inserted there. That is the reason why it is named so. This algorithm is not suitable for large data set.
Average and worst case time complexity of the algorithm is **O(n2)**, where n is the number of items.

Algorithm

Step 1: If it is the first element, it is already sorted. return 1;
Step 2: Pick next element
Step 3: Compare with all elements in the sorted sub-list
Step 4: Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5: Insert the value
Step 6: Repeat until list is sorted

Example,

Let us take an example of an array A[6]={20,10,30,15,25,05}. We have to sort this array using insertion sort.




```
void insertionSort(int arr[], int length)
{
   int i, j, key;
   for (i = 1; i < length; i++)
   {
      j = i;
      while (j > 0 && arr[j - 1] > arr[j])
      {
         key = arr[j];
         arr[j] = arr[j - 1];
         arr[j - 1] = key;
         j--;
      }
   }
}
```

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.

Quick Sort

Quick sort is a well-known sorting algorithm. It is highly efficient and also known as partition exchange sort. In this sorting algorithm the array is divided into 2 sub array. One contain smaller values than pivot value and other array contain elements having greater values than pivot value. Pivot is an element that is used to compare and divide the elements of the main array into two. Quick sort partitions an array and then calls itself recursively twice to sort the two resulting sub arrays. This algorithm is quite efficient for large data sets.

The Average and worst case complexity are of this algorithm is **O(n2)**, where n is the number of items.

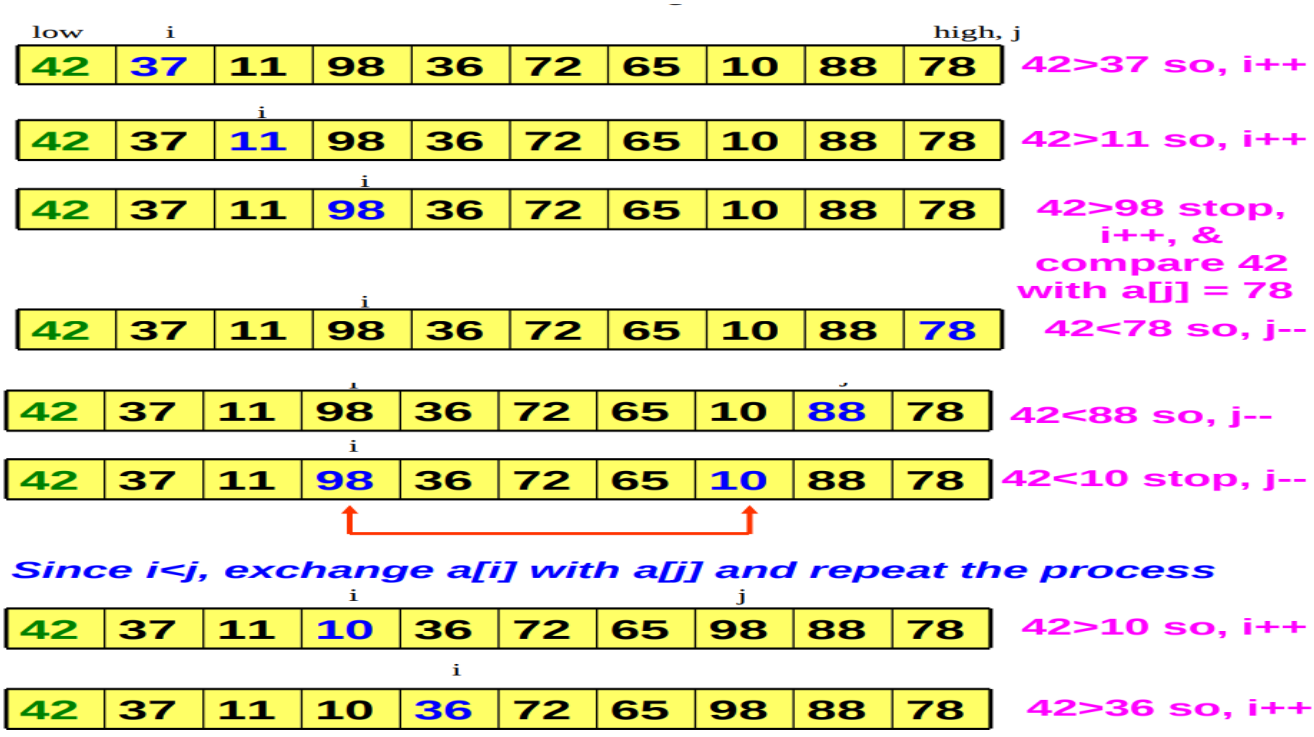
Algorithm

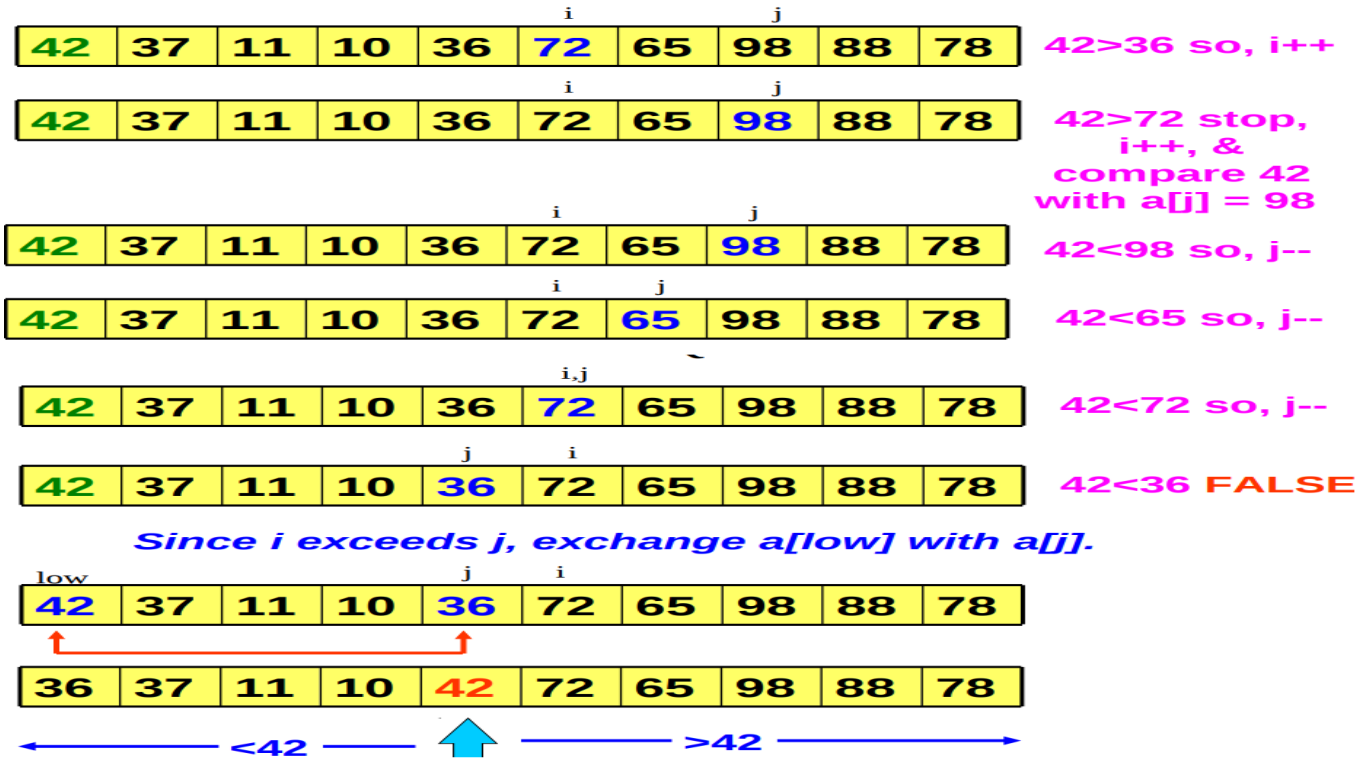
Step 1: Choose the highest index value has pivot
Step 2: Take two variables to point left and right of the list excluding pivot
Step 3: left points to the low index
Step 4: right points to the high
Step 5: while value at left is less than pivot move right
Step 6: while value at right is greater than pivot move left
Step 7: if both step 5 and step 6 does not match swap left and right
Step 8: if left ≥ right, the point where they met is new pivot

In practice, quick sort is faster than other sorting algorithms because its inner loop can be efficiently implemented on most architectures, and in most real-world data it is possible to make design choices which minimize the possibility of require time.

Example,

Let us assume an array A[10]={42,37,11,98,36,72,65,10,88,78}. We have to sort this array using quick sort.






```
// to swap two numbers
void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
}
/* a[] is the array, p is starting index, that is 0, and r is the last index of array. */
void quicksort(int a[], int p, int r)
{
   if(p < r)
   {
      int q;
      q = partition(a, p, r);
      quicksort(a, p, q);
      quicksort(a, q+1, r);
   }
}
int partition (int a[], int low, int high)
{
   int pivot = arr[high]; // selecting highest element as pivot
   int i = (low - 1); // index of smaller element
   for (int j = low; j <= high- 1; j++)
   {
// If current element is smaller than or equal to pivot
      if (arr[j] <= pivot)
```

```
      {
         i++; // increment index of smaller element
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
```

Advantages	Disadvantages
The quick sort is regarded as the best sorting algorithm.	The slight disadvantage of quick sort is that its worst-case performance is similar to average performances of the bubble, insertion or selections sorts.
It is able to deal well with a huge list of items.	If the list is already sorted than bubble sort is much more efficient than quick sort
Because it sorts in place, no additional storage is required as well	If the sorting element is integers than radix sort is more efficient than quick sort.

Radix sort:

Algorithm:

For each digit i where i varies from the least significant digit to the most significant digit of a number
Sort input array using count sort algorithm according to i^{th} digit.

We used count sort because it is a stable sort.

Example: Assume the input array is:
10,21,17,34,44,11,654,123
Based on the algorithm, we will sort the input array according to the **one's digit**
(least significant digit).
0: 10
1: 21 11
2:
3: 123
4: 34 44 654
5:
6:
7: 17
8:
9:

So, the array becomes 10,21,11,123,24,44,654,17
Now, we'll sort according to the **ten's digit**:
0:
1: 10 11 17

2: 21 123
3: 34
4: 44
5: 654
6:
7:
8:
9:

Now, the array becomes : 10,11,17,21,123,34,44,654
Finally , we sort according to the **hundred's digit** (most significant digit):
0: 010 011 017 021 034 044
1: 123
2:
3:
4:
5:
6: 654
7:
8:
9:

The array becomes : 10,11,17,21,34,44,123,654 which is sorted. This is how our algorithm works.

Advantages

1. It is implemented in Java, it would be faster than quicksort or heap.
2. It is stable because it preserves existing order of equals keys.
3. It is good on small keys.

Disadvantages

1. It is not efficient on very long keys because the total sorting time is proportional to key length and to the number of items to sort.
2. We have to write an unconventional compare routine.
3. It requires fixed size keys and some standard way of breaking the keys to pieces.