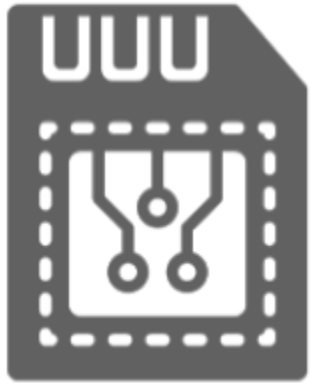
Time Complexity



Space Complexity

CS 2124: Data Structures
Spring 2024

Lecture 2 (Part – II)

Topics: Time and Space Complexity, Runtimes, **Sorting**, Searching

Topics

- Big O (Revision)
- Sorting Algorithms
  - Stable vs Unstable Sorting
  - In-place and Out-of-place Sorting
- Bubble Sort
- Selection Sort
  - Bubble sort vs Selection sort
- Insertion Sort
  - Insertion Sort vs Bubble sort vs Selection sort
- Merge Sort
- Quick Sort
  - Median as pivot
- Summary

Big O notation

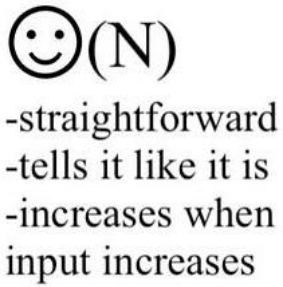
- when analyzing some algorithm, one might find that the time (or the number of steps) it takes to complete a problem of size n is given by $T(n) = 4n^2 - 2n + 2$.
- If we ignore constants (which makes sense because those depend on the particular hardware the program is run on) and slower growing terms, we could say "T(n) grows at the order of n^2 " and write: $T(n) = O(n^2)$.

- **Performance:** how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.
- **Complexity:** how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger?

Complexity affects performance but not the other way around.

Big O notation

- **Sequence of statements:**
  - *Statement 1; statement 2; ... statement k;*
- The total time is found by adding the times for all statements:
- total time = time(statement 1) + time(statement 2) + ... + time(statement k)
- If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1).
- **If-Then-Else**
  - *if (cond)*
  - *then*
    - *block 1 (sequence of statements)*
  - *else*
    - *block 2 (sequence of statements)*
  - *end if;*
- Here, either block 1 will execute, or block 2 will execute.
- Therefore, the worst-case time is the slower of the two possibilities: max(time(block 1), time(block 2)) If block 1 takes O(1) and block 2 takes O(N), the if-then-else statement would be O(N).

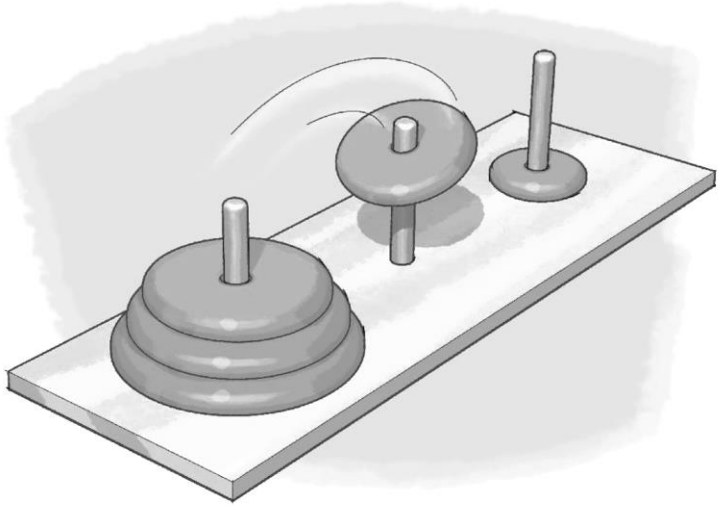


Big O notation

- **Loops:**
    - *for I in 1 .. N loop*
        - *sequence of statements*
    - *end loop;*
- The loop executes N times, so the sequence of statements also executes N times. If we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.
- **Nested loops:**
    - *for I in 1 .. N loop*
        - *for J in 1 .. M loop*
            - *sequence of statements*
            - *end loop;*
    - *end loop;*
- The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times.
- As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M).
- In a common special case where the stopping condition of the inner loop is instead of (i.e., the inner loop also executes N times), the total complexity for the two loops is $O(n^2)$.

Big O notation

- Algorithms with running time *O(2^N) i.e.* $O(2^n)$ are often recursive algorithms that solve a problem of size N by recursively solving two smaller problems of size N-1.
  - For instance to solve the famous "Towers of Hanoi" problem for N disks



Big O notation

- The binary search algorithm accomplishes its task by dividing the search area in half on each iteration.
- So at the start we have N elements to search.
- By the second step we only have N/2 elements to search, and by the third we only have N/4 elements to search.
  - *N = 8, [4, 8, 10, 14, 27, 31, 46, 52] //Compared and divide search area by 2*
  - *N = 4, [27, 31, 46, 52] //Compared and divide search area by 2*
  - *N = 2, [46, 52] //Compared mid to target. They matched, so returned mid.*
- Notice that this took three steps and it's dividing by 2 each time.
- If we multiplied by 2 each time we would have *2 x 2 x 2 = 8, or 23 = 8.*
  - *$23 = 8 \rightarrow \log_2 8 = 3$*
  - *$2^k = N \rightarrow \log_2 N = k$*
- Therefore, the Big O complexity of a binary search is *O(log N).*

Sorting Algorithms

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

- The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner.

- Sorting is also used to represent data in more readable formats.

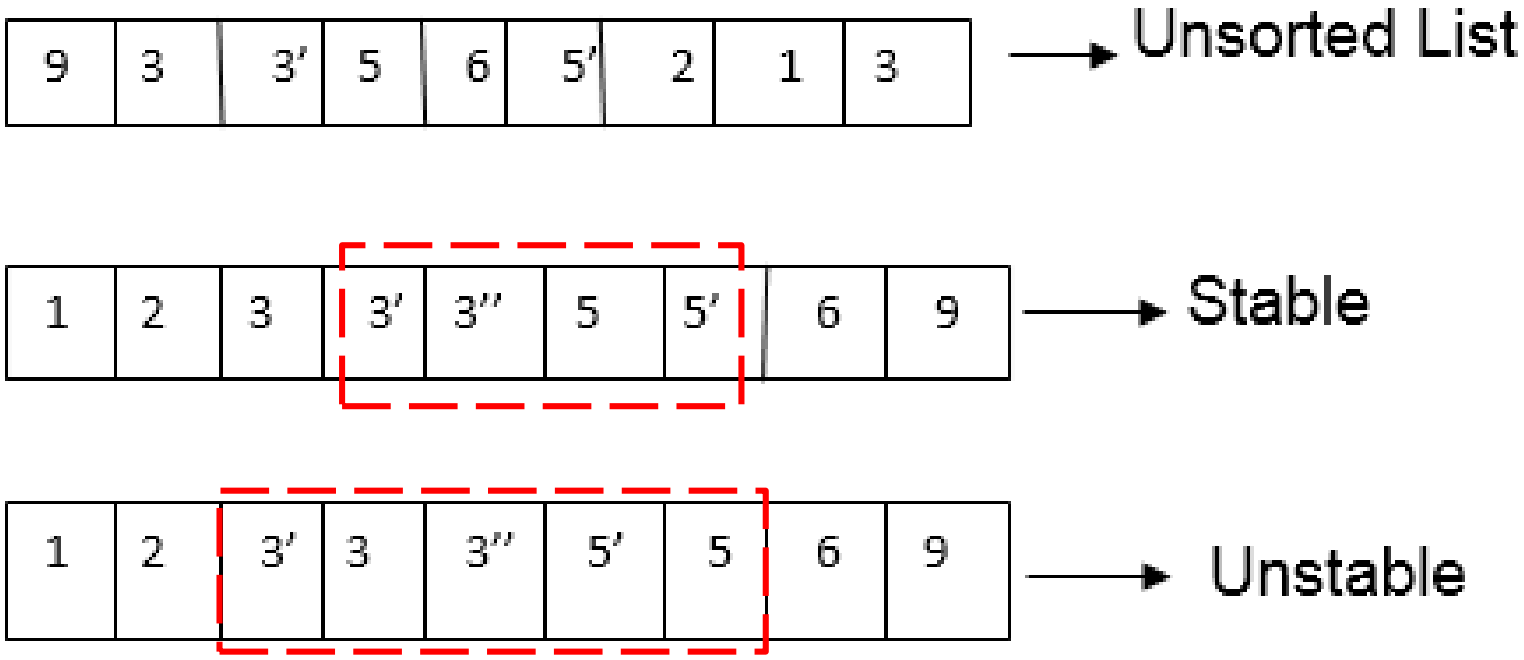
- In general there are **2 approaches** to sort an array of elements:
  1. Some algorithms work by moving elements to their final position, one at a time. You sort an array of size N, put 1 item in place, and continue sorting an array of size N – 1.
  2. Some algorithms put items into a temporary position, close(r) to their final position. You rescan, moving items closer to the final position with each iteration.

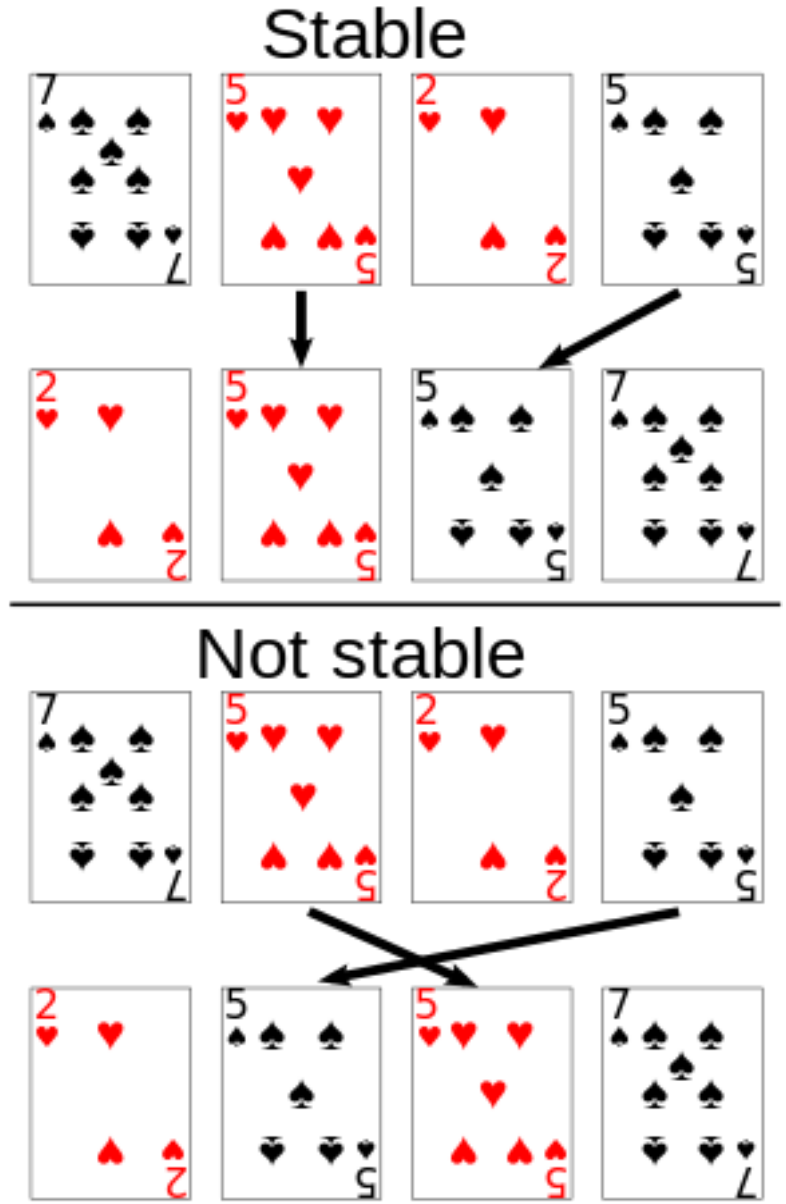
Sorting Algorithms (Complexity And Running Time)

- Factors:
  1. Algorithmic complexity
  2. Additional space requirements
  3. Use of recursion
  4. Worst-case behavior
     - Worst-case behavior is important for real-time systems that need guaranteed performance.
  5. Behavior on already-sorted or nearly-sorted data

Stable vs Unstable Sorting

- A stable sort is one which preserves the original order of the input set, where the [unstable] algorithm does not distinguish between two or more items.





In-place and Out-of-place Sorting

- An **In-place algorithm** modifies the inputs, which can be a list or an array, without using any additional memory. As the algorithm runs, the input is usually overwritten by the output, so no additional space is required.
  - However, the in-place sorting algorithms may take some memory, like using some variables, etc. for its operation.
  - Overall, it takes constant memory for its operation. Since they take constant space, the space complexity of these algorithms is $O(1)$

- An algorithm that is not in place is called a **not-in-place or out-of-place algorithm**. These sorting algorithm uses extra space for sorting, which depends upon the size of the input.
  - The standard merge sort algorithm is an example of the out-of-place algorithm as it requires O(n) extra space for merging.

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

Big O notation

	Cost	Times
i=1;	c1	1
sum = 0;	c2	1
while (i <= n) {	c3	n+1
j=1;	c4	n
while (j <= n) {	c5	n*(n+1)
sum = sum + i;	c6	n*n
j = j + 1;	c7	n*n
}		
i = i +1;	c8	n
}		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

```c
#include <stdio.h>
int main()
{
    int i=1, sum=0,n=2, j=0;
    while (i <= n)
    {
        while (j <= n)
    {
        sum=sum+i;
        j=j+1;
    }
    i=i+1;
    n--;
    }
    printf("%d, %d, %d", i, j, sum);
}
```

Try to find the Big O representation of the algorithm and the out put of the program

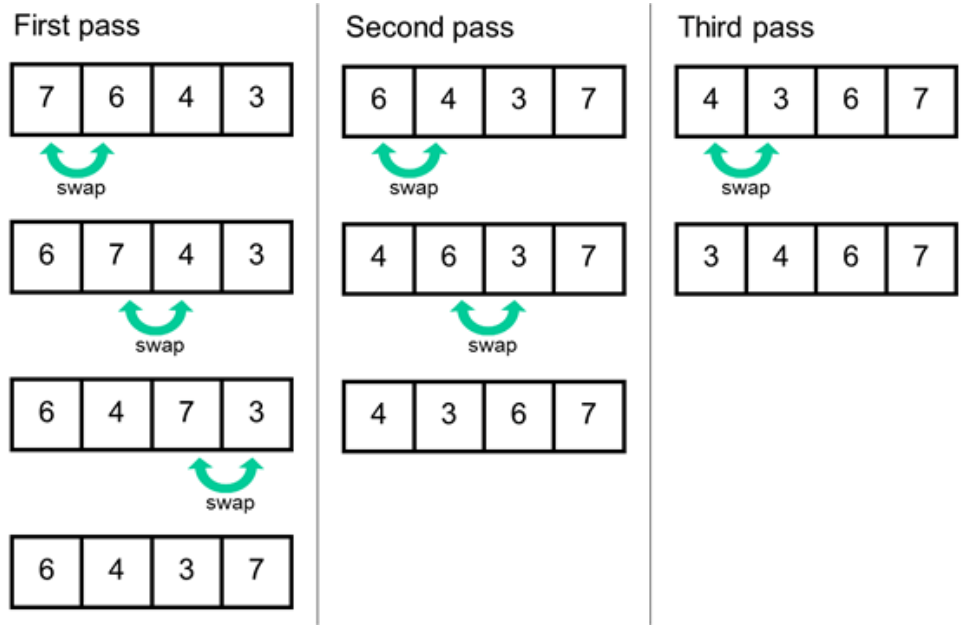
Bubble Sort

- **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.
- This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.
- Steps:
  - Walks through the array n-times.
  - As it walks up the array it checks if the current element and it's neighbor are out of order
  - If they are not, it swaps them so the larger element is right
  - Bubble sort is an In-place and stable sorting algorithm

Bubble Sort

```
1.  begin BubbleSort(list)
2.     for all elements of list
3.        if list[i] > list[i+1]
4.           swap(list[i], list[i+1])
5.        end if
6.     end for
7.        return list
8.     end BubbleSort
```

The worst-case occurs when we want to sort a list in ascending order, but it is arranged in descending order.



Bubble Sort

```
14 int main() {
15   int a[100], n, i, d, swap;
16   printf("Enter number of elements in the array:\n");
17   scanf("%d", &n);
18   printf("Enter %d integers\n", n);
19   for (i = 0; i < n; i++)
20     scanf("%d", &a[i]);
21   bubble_sort(a, n);
22   printf("Printing the sorted array:\n");
23   for (i = 0; i < n; i++)
24     printf("%d\n", a[i]);
25   return 0;
26 }
```

Values are passed to the function
a = Array containing the elements/user input
n= Total number of elements

Bubble Sort

```
21    bubble_sort(a, n);
```

```
1  #include <stdio.h>
2  void bubble_sort(int a[], int n) {
3      int i = 0, j = 0, tmp;
4      for (i = 0; i < n; i++) {   // loop n times - 1 per element
5          for (j = 0; j < n - i - 1; j++) { // last i elements are sorted already
6              if (a[j] > a[j + 1]) {  // swop if order is broken
7                  tmp = a[j];
8                  a[j] = a[j + 1];
9                  a[j + 1] = tmp;
10             }
11         }
12     }
13 }
```

- if list[i] > list[i+1]
- swap(list[i], list[i+1]) //Using a temporary variable ‘tmp’

Bubble Sort

```c
#include <stdio.h>
void bubble_sort(int a[], int n) {
    int i = 0, j = 0, tmp;
    for (i = 0; i < n; i++) {   // loop n times - 1 per element
        for (j = 0; j < n - i - 1; j++) { // last i elements are sorted already
            if (a[j] > a[j + 1]) {  // swop if order is broken
                tmp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = tmp;
            }
        }
    }
}
```

Bubble Sort (Complete Code)

```c
#include <stdio.h>
void bubble_sort(int a[], int n) {
    int i = 0, j = 0, tmp;
    for (i = 0; i < n; i++) {   // loop n times - 1 per element
        for (j = 0; j < n - i - 1; j++) { // last i elements are sorted already
            if (a[j] > a[j + 1]) {  // swop if order is broken
                tmp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = tmp;
            }
        }
    }
}
int main() {
  int a[100], n, i, d, swap;
  printf("Enter number of elements in the array:\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
  bubble_sort(a, n);
  printf("Printing the sorted array:\n");
  for (i = 0; i < n; i++)
     printf("%d\n", a[i]);
  return 0;
}
```

Can the code be optimized ?

Selection Sort

- Selection sort is a simple sorting algorithm. This sorting algorithm is an **in-place** comparison-based algorithm in which the list is divided into two parts:
  1. The Sorted Part At The Left End
  2. The Unsorted Part At The Right End.
- Initially, the sorted part is empty and the unsorted part is the entire list.

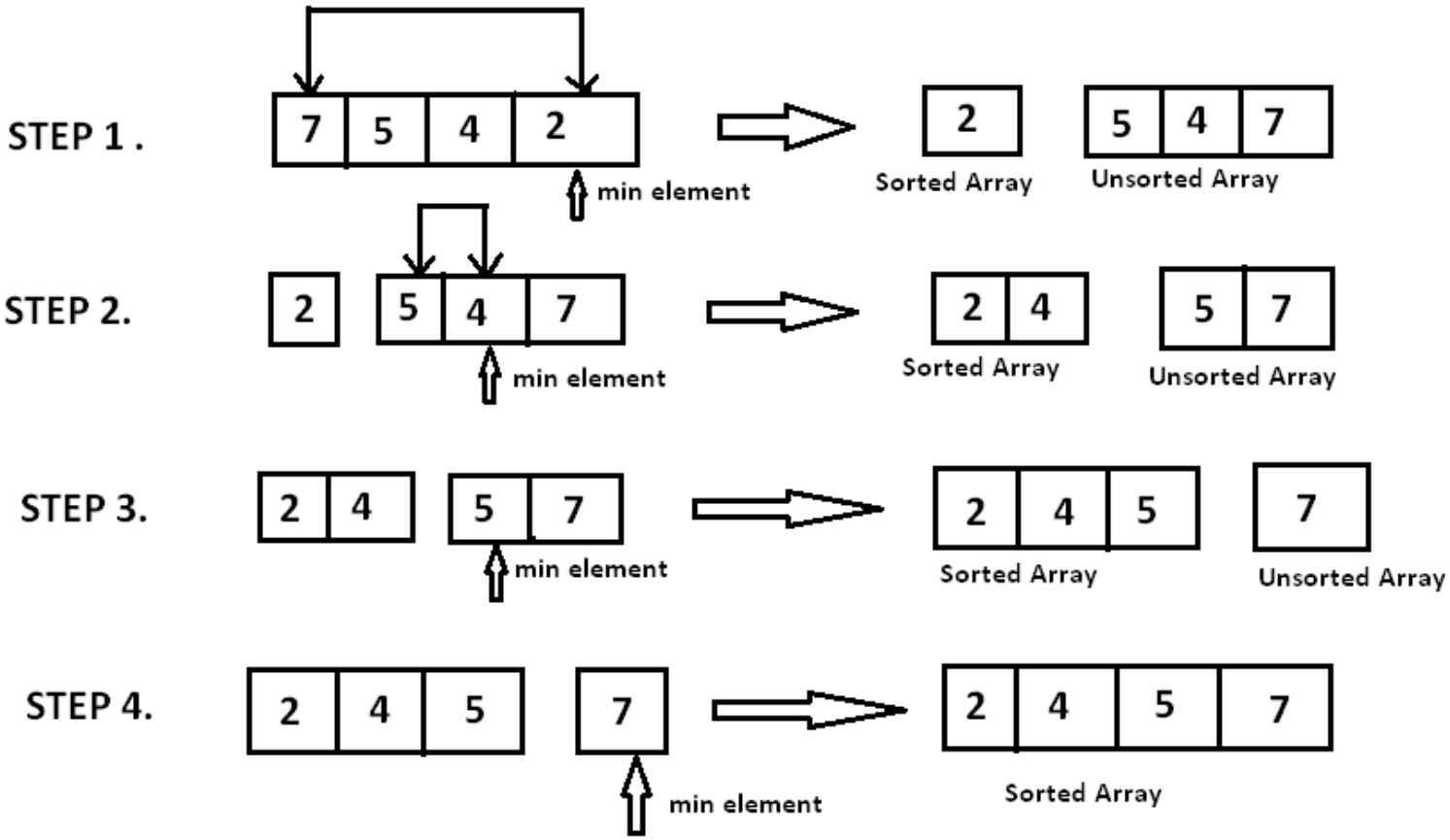
- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.
- This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

5 3 4 1 2

Selection Sort

Visual representation on next slide >>

Selection Sort



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

Try code implementation yourself

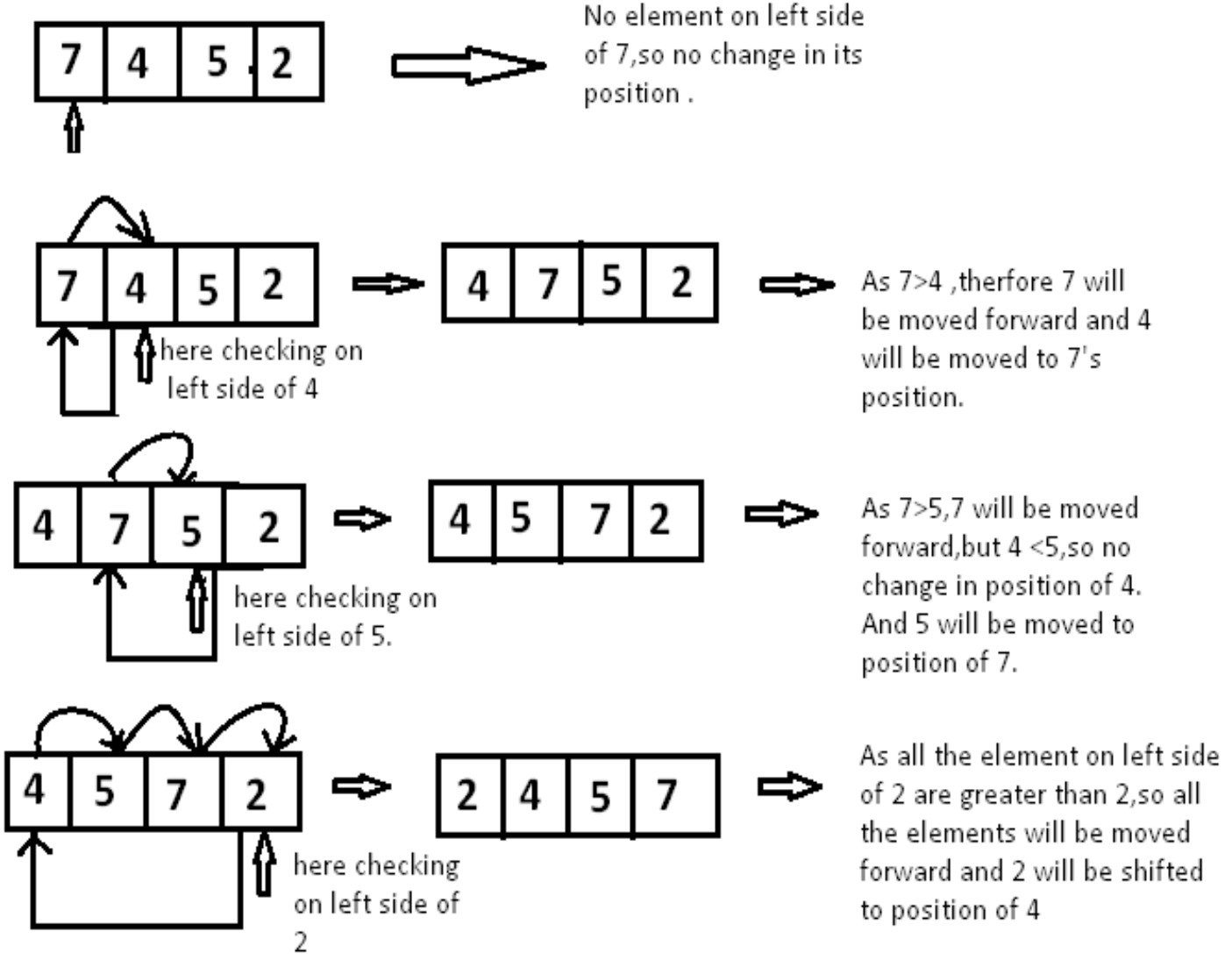
Bubble sort vs Selection sort

Bubble Sort	Selection Sort
Simple Sorting Algorithm	Simple Sorting Algorithm
Compares neighboring elements	Takes the smallest element and moves it into its place
Swap based sorting	In-place sorting

- Which one is fast and efficient ?

Insertion Sort

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



Insertion Sort

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

$O(n)$

The worst case occurs when the array is sorted in reverse order.

So the worst case time complexity of insertion sort is $O(n^2)$

Insertion Sort

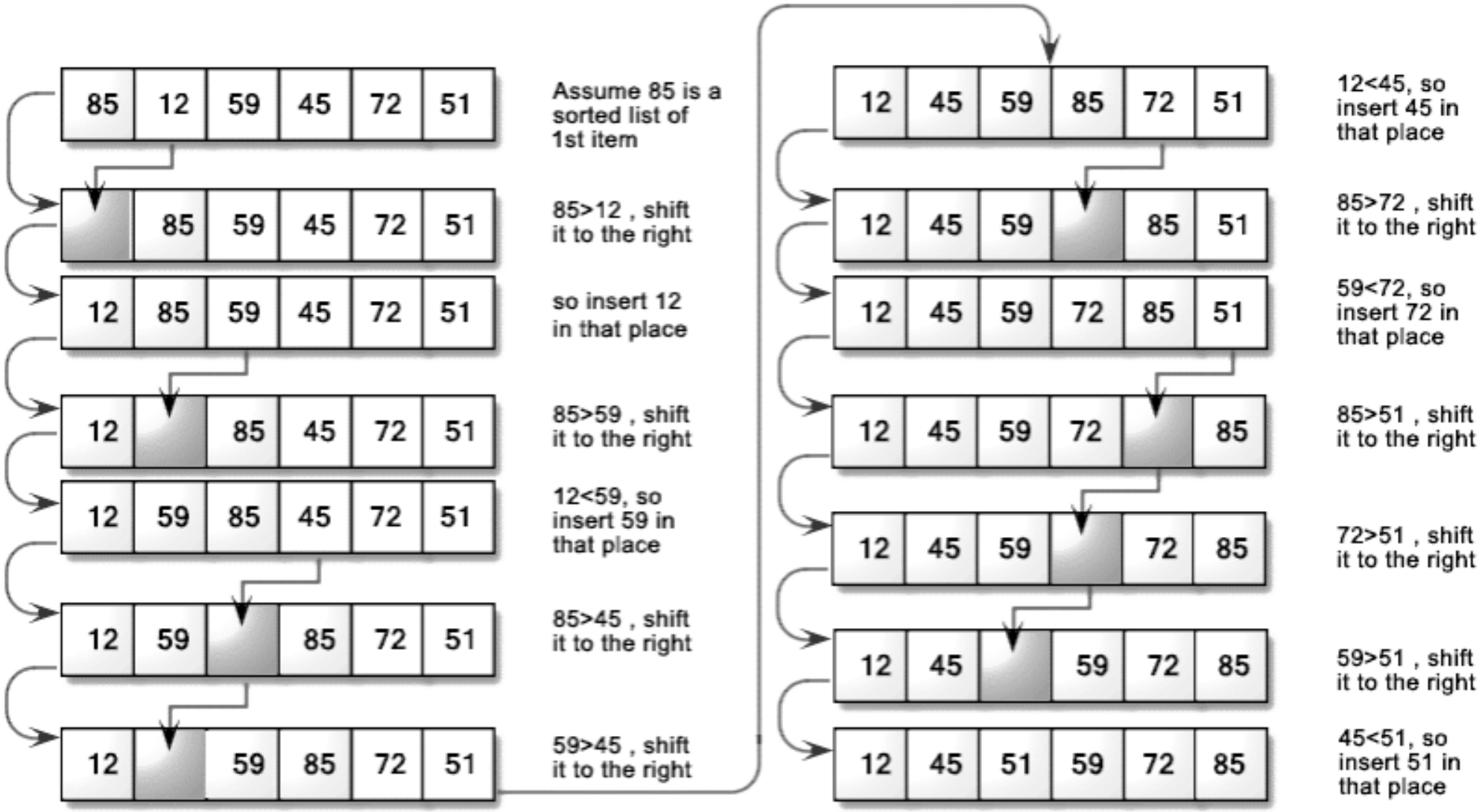
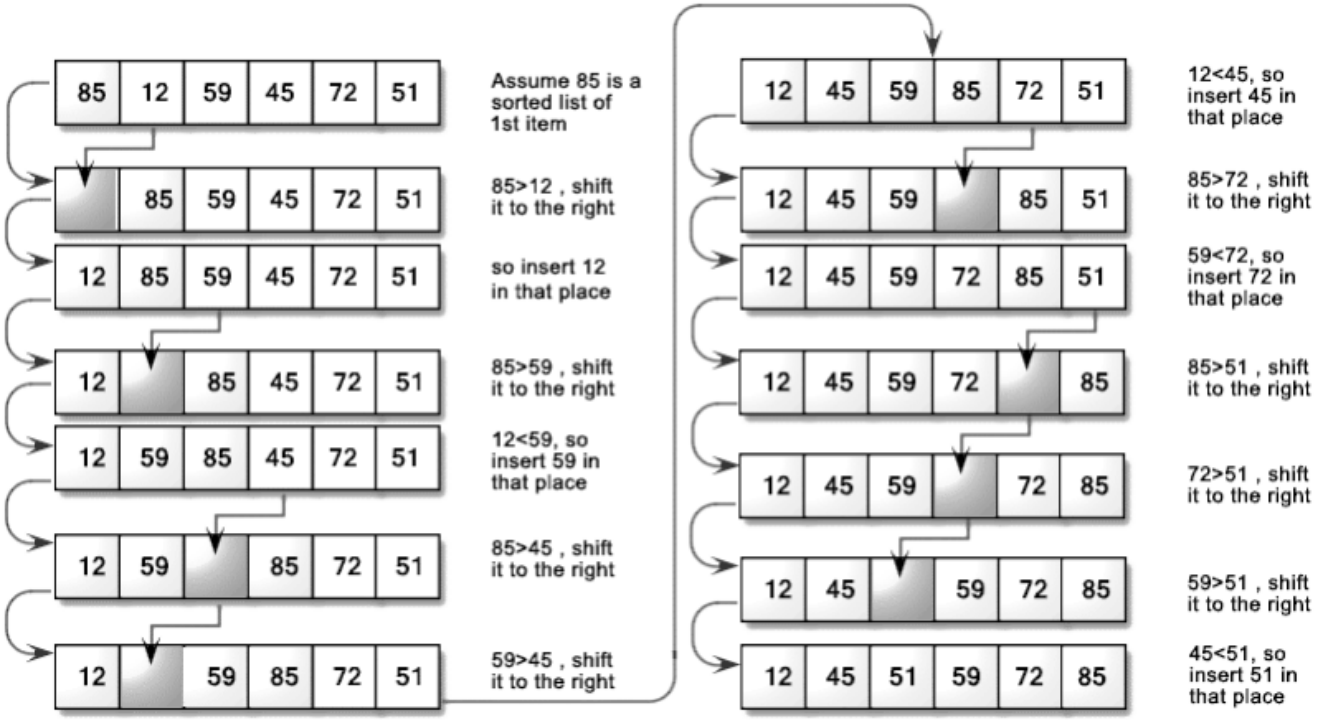


Image Source: [Link]

Insertion Sort

```c
#include<stdio.h>
  int main()
  {
  int  arra[10],i,j,n,array_key;
  printf("Input  no. of values in the array: \n");
  scanf("%d",&n);
  printf("Input  %d array value(s): \n",n);
  for(i=0;i<n;i++)
  scanf("%d",&arra[i]);
    /* Insertion Sort  */
  for (i = 1; i <  n; i++)
  {
  array_key =  arra[i];
  j = i-1;
    while (j >=  0 && arra[j] > array_key)
  {
  arra[j+1] = arra[j];
  j = j-1;
  }
  arra[j+1] =  array_key;
  }
  printf("Sorted  Array: \n");
  for (i=0; i <  n; i++)
  printf("%d  \n", arra[i]);
  }
```



Source: Link

Insertion Sort

```c
#include<stdio.h>
int main() {
    int arra[10], i, j, n, array_key;
    // Input the number of values in the array
    printf("Input no. of values in the array: \n");
    scanf("%d", &n);
    // Input array values
    printf("Input %d array value(s): \n", n);
    for (i = 0; i < n; i++)
        scanf("%d", &arra[i]);
    /* Insertion Sort */
    for (i = 1; i < n; i++) {
        array_key = arra[i];
        j = i - 1;
        // Move elements greater than array_key to one position ahead of their current position
        while (j >= 0 && arra[j] > array_key) {
            arra[j + 1] = arra[j];
            j = j - 1;
        }
        // Insert array_key at its correct position
        arra[j + 1] = array_key;
    }
    // Print the sorted array
    printf("Sorted Array: \n");
    for (i = 0; i < n; i++)
        printf("%d  \n", arra[i]);
}
```

Logical Operators		
Operator	Description	Example
&&	AND	x=6 y=3 x<10 && y>1 Return True
||	OR	x=6 y=3 x==5 || y==5 Return False
!	NOT	x=6 y=3 !(x==y) Return True

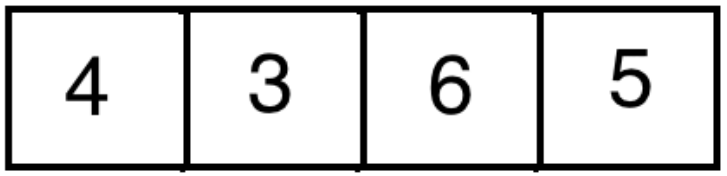
Insertion Sort (Complete Code)

```c
#include<stdio.h>
int main() {
    int arra[10], i, j, n, array_key;
    // Input the number of values in the array
    printf("Input no. of values in the array: \n");
    scanf("%d", &n);
    // Input array values
    printf("Input %d array value(s): \n", n);
    for (i = 0; i < n; i++)
        scanf("%d", &arra[i]);
    /* Insertion Sort */
    for (i = 1; i < n; i++) {
        array_key = arra[i];
        j = i - 1;
        // Move elements greater than array_key to one position ahead of their current position
        while (j >= 0 && arra[j] > array_key) {
            arra[j + 1] = arra[j];
            j = j - 1;
        }
        // Insert array_key at its correct position
        arra[j + 1] = array_key;
    }
    // Print the sorted array
    printf("Sorted Array: \n");
    for (i = 0; i < n; i++)
        printf("%d  \n", arra[i]);
}
```

Insertion Sort

Bubble Sort	Selection Sort	Insertion Sort
Simple Sorting Algorithm	Simple Sorting Algorithm	Simple Sorting Algorithm
Compares neighboring elements	Takes the smallest element and moves it into its place	Transfer one element at a time to its place
Swap based sorting	In-place sorting	Complex but fast

Insertion Sort



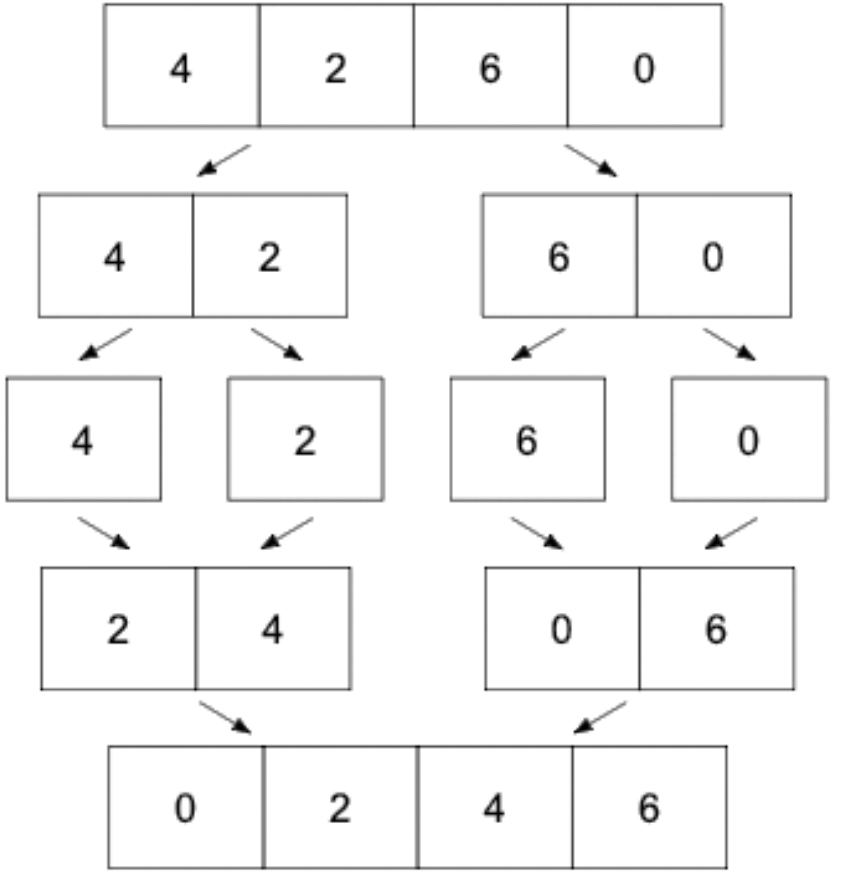
Merge Sort

- Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

- In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

Merge Sort

Step 1: Find the middle index of the array.
Middle = 1 + (last – first)/2
Step 2: Divide the array from the middle.
Step 3: Call merge sort for the first half of the array
MergeSort(array, first, middle)
Step 4: Call merge sort for the second half of the array.
MergeSort(array, middle+1, last)
Step 5: Merge the two sorted halves into a single sorted array.



- Overall time complexity of Merge sort is O(n log n).
- It is more efficient as it is in worst case also the runtime is O(n log n).
- The space complexity of Merge sort is O(n).
- This means that this algorithm takes a lot of space and may slower down operations for the last data sets

Merge Sort

- Which searching algorithm you feel is similar to merge sort ?

- Concerns for Merge Sort?

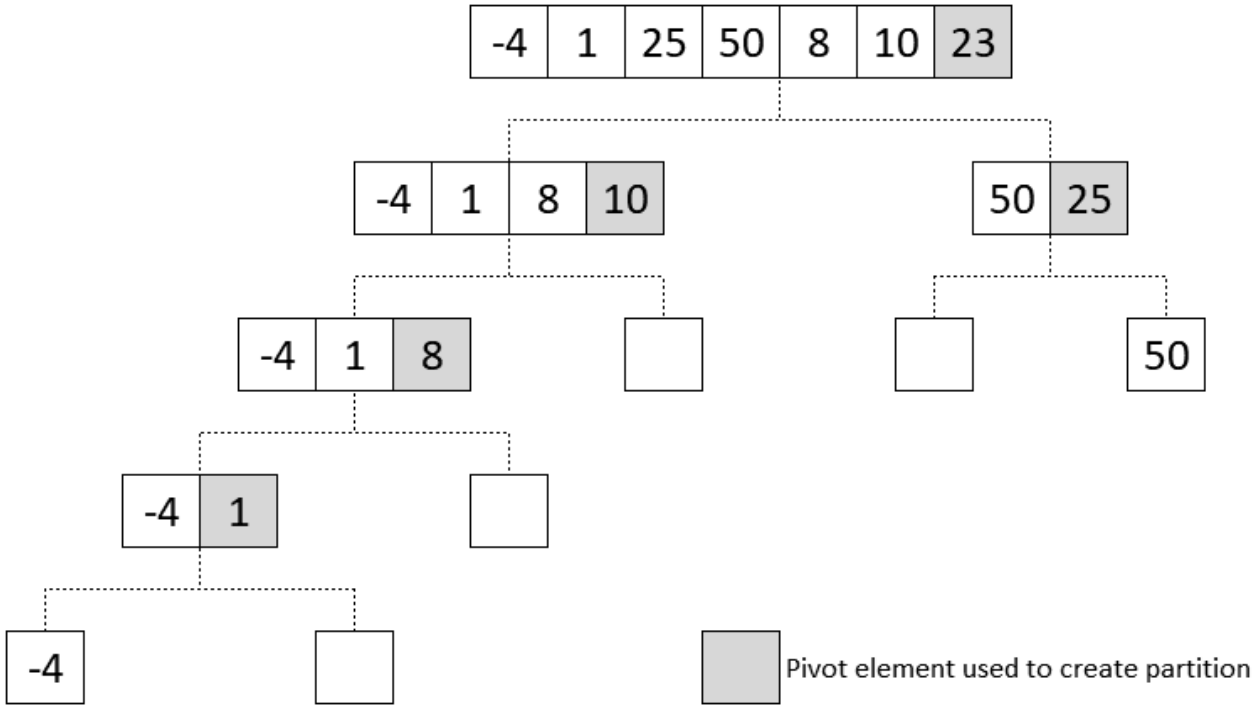
Try to Implement it your self

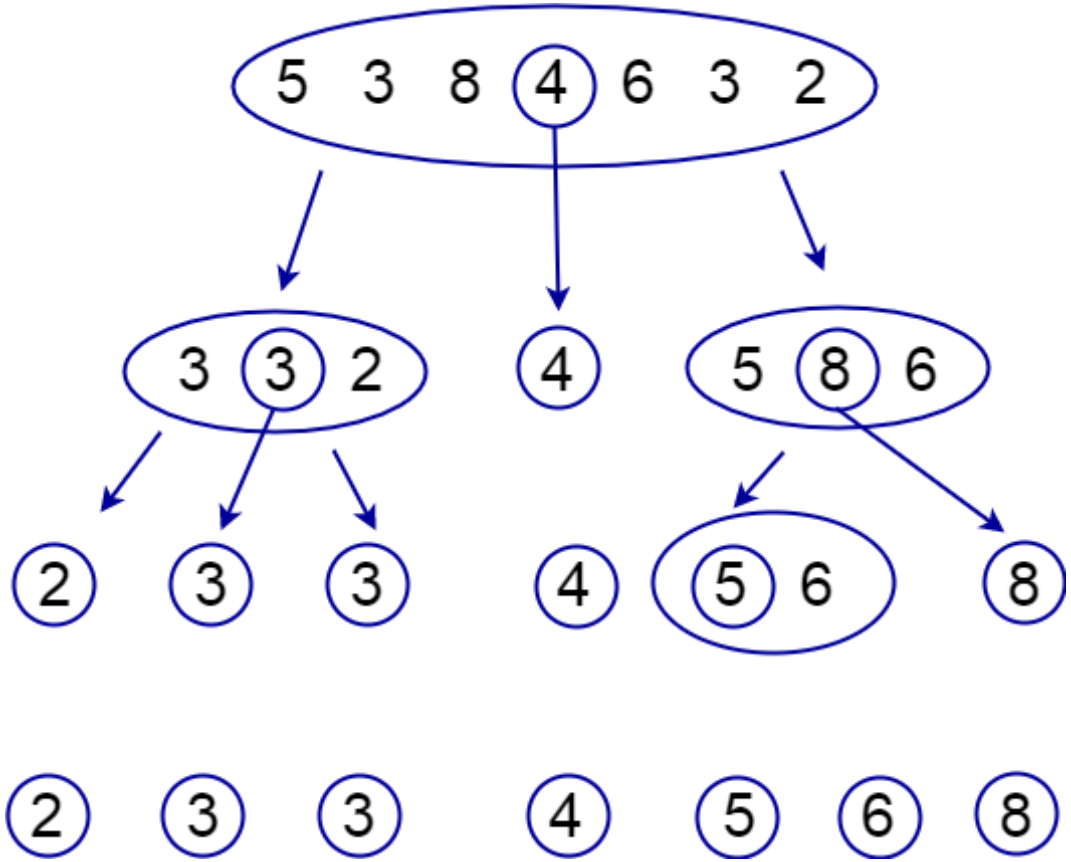
Quick Sort

- Like Merge Sort, QuickSort is a Divide and Conquer algorithm.
- It picks an element as a pivot and partitions the given array around the picked pivot.
- There are many different versions of quickSort that pick pivot in different ways.

1. Always pick the first element as a pivot.
2. Always pick the last element as a pivot (implemented below)
3. Pick a random element as a pivot.
4. Pick median as the pivot.

Quick Sort
(Visual representation)





Quick Sort

- Complexity: O(n log n).
- Highly dependent on the selection of pivot
- Worst Case $O(n^2)$: when the picked pivot is always an extreme (smallest or largest) element. This happens when input array is sorted or reverse sorted and either first or last element is picked as pivot.

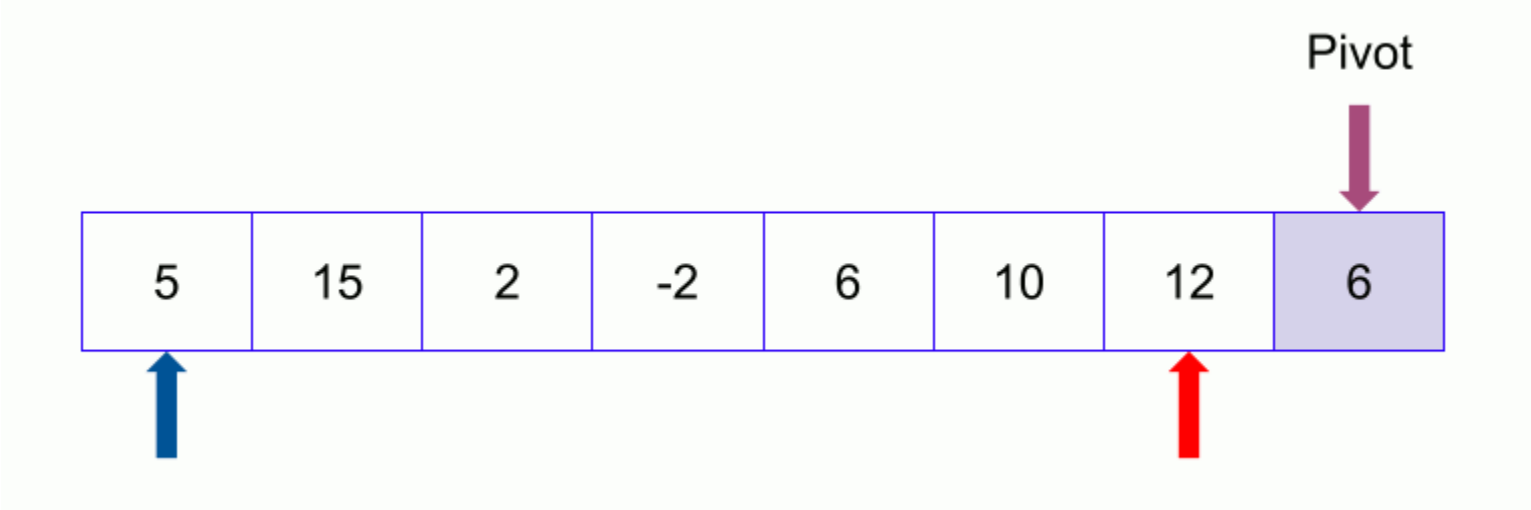


Image Source: Link

Quick Sort

```
24  int main(){
25      int i, count, number[25];
26      printf("How many elements are u going to enter?: ");
27      scanf("%d",&count);
28      printf("Enter %d elements: ", count);
29      for(i=0;i<count;i++)
30          scanf("%d",&number[i]);
31      quicksort(number,0,count-1);
32      printf("Order of Sorted elements: ");
33      for(i=0;i<count;i++)
34          printf(" %d",number[i]);
35  }
```

Quick Sort

```c
#include<stdio.h>
void quicksort(int number[25],int first,int last){
   int i, j, pivot, temp;
   if(first<last){
      pivot=first;
      i=first;
      j=last;
      while(i<j){
         while(number[i]<=number[pivot]&&i<last)
            i++;
         while(number[j]>number[pivot])
            j--;
         if(i<j){
            temp=number[i];
            number[i]=number[j];
            number[j]=temp;
         }      }
      temp=number[pivot];
      number[pivot]=number[j];
      number[j]=temp;
      quicksort(number,first,j-1);
      quicksort(number,j+1,last);
   }   }
```

```c
31      quicksort(number,0,count-1);
```

1) Pick an element from the array, this element is called as pivot element.
2) Divide the unsorted array of elements in two arrays
   a) Values less than the pivot come in the first sub array
   b) Values greater than the pivot come in the second sub-array (equal values can go either way).
3) Recursively repeat the step 2(until the sub-arrays are sorted) to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

- The same logic we have implemented in the following C program.

Quick Sort

```c
#include<stdio.h>
void quicksort(int number[25],int first,int last)
{
   int i, j, pivot, temp;
   if(first<last)
   {
      pivot=first; //First element as pivot
      i=first;
      j=last;

      while(i<j)
      {
         while(number[i]<=number[pivot]&&i<last)
            i++;
         while(number[j]>number[pivot])
            j--;
         if(i<j)
         {
            temp=number[i];
            number[i]=number[j];
            number[j]=temp;
         }
      }
      temp=number[pivot];
      number[pivot]=number[j];
      number[j]=temp;
      quicksort(number,first,j-1);
      quicksort(number,j+1,last);

   }
}
int main(){
   int i, count, number[25];
   printf("How many elements are u going to enter?: ");
   scanf("%d",&count);
   printf("Enter %d elements: ", count);
   for(i=0;i<count;i++)
      scanf("%d",&number[i]);
   quicksort(number,0,count-1);
   printf("Order of Sorted elements: ");
   for(i=0;i<count;i++)
      printf(" %d",number[i]);
   return 0;
}
```

Line 7:

- pivot=last;
- mid = (first + last) / 2;
- Pivot = rand()

Median

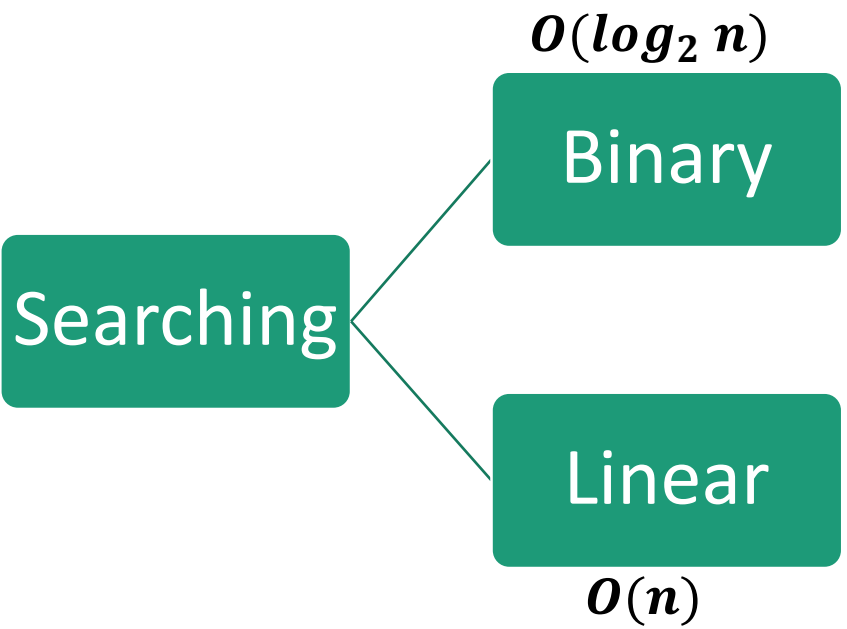
```c
#include <stdio.h>
void swap(int *p,int *q)
{
   int t;
   t=*p;
   *p=*q;
   *q=t;
   printf("T = %d, p = %d , q = %d \n", t, *p, *q);

}
void sort(int a[],int n)
{
   int i,j,temp;
   for(i = 0;i < n-1;i++)
   {
      for(j = 0;j < n-i-1;j++)
      {
         if(a[j] > a[j+1])
            swap(&a[j],&a[j+1]);
      }
   }
}
```

```c
int main() {
   int a[] = {2,3,8,5,1};
   int n = 5;
   int sum,i;
   sort(a,n);
   for(int i = 0;i < 5;i++)
   printf("Array[%d] = %d ", i, a[i]);
   n = (n+1) / 2 - 1;       // array indexing
   printf("\n Median = %d ", a[n]);
   return 0;
}
```

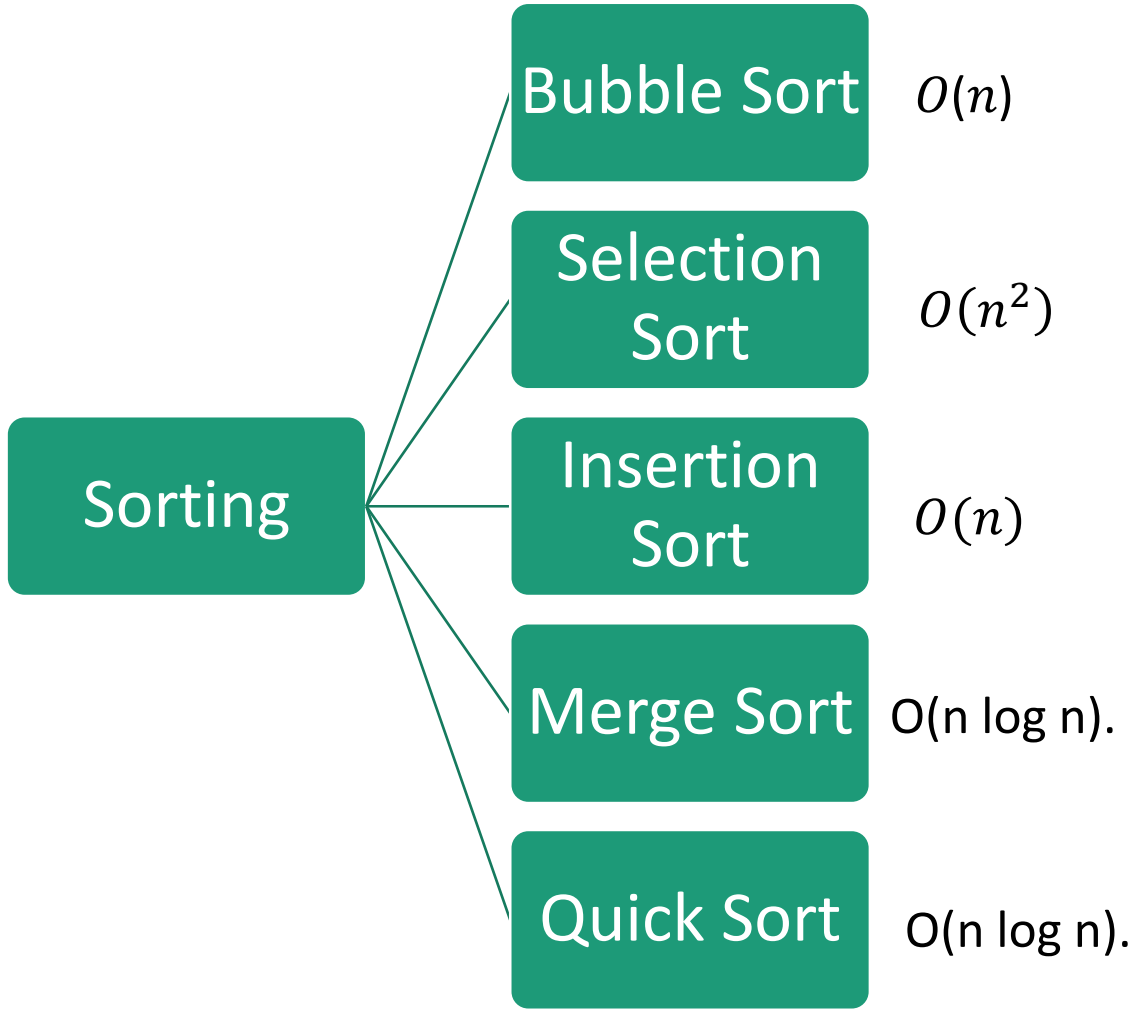
- What will be the output?

Summary

- The list/array must be sorted
- Divide and conquer method



- Large Data an issue



End of Second week

Do try to implement the codes by your self to better understand the working

- Assignment – 1 (Plagiarism check is enable on the canvas)
  - Assigned: 22nd Jan
  - Due: 29th Jan (End of day as per Canvas)