Sorting algorithms – Bubble selection Insertion Merge Quick Heap (bucket & shell)

Time complexities

Git github

Recursion

Data structures implementation from scratch

- O(1)

C

```c
#include <stdio.h>
int main()
{
    printf("Hello World");
}
```

Output

```
Hello World
```

In above code "Hello World!!!" print only once on a screen. So, time complexity is constant: O(1) i.e. every time constant amount of time require to execute code, no matter which operating system or which machine configurations you are using.

- O(n)

C

```c
#include <stdio.h>
void main()
{
    int i, n = 8;
    for (i = 1; i <= n; i++) {
        printf("Hello Word !!!\n");
    }
}
```

- O(N+M)

CPP

```cpp
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

- O(N * (N-1))

CPP

```cpp
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

- O((n/2)* log(n))

CPP

```cpp
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

In asymptotic analysis, we consider the growth of the algorithm in terms of input size. An algorithm X is said to be asymptotically better than Y if X takes smaller time than y for all input sizes n larger than a value n0 where n0 > 0.

- O(logn)

CPP

```cpp
int a = 0, i = N;
while (i > 0) {
    a += i;
    i /= 2;
}
```

We have to find the smallest x such that N / 2^x N, x = log(N)

- How is time complexity measured?

By counting the number of primitive operations performed by the algorithm on given input size.

Javascript

```javascript
for(var i=0;i<n;i++)
    i*=k
```

1. O(n)
2. O(k)
3. $O(\log_k n)$
4. $O(\log_n k)$

Output:

```
3. O(log_k n)
```

Explanation: because loops for the k^{n-1} times, so after taking log it becomes $\log_k n$.

- O(n* (n-1)/2)

Javascript

```javascript
var value = 0;
for(var i=0;i<n;i++)
    for(var j=0;j<i;j++)
    value += 1;
```

The Big-O notation provides an asymptotic comparison in the running time of algorithms. For $n < n^0$, algorithm A might run faster than algorithm B, for instance.

CPP

```cpp
function(int n)
{
    if (n==1)
       return;
    for (int i=1; i<=n; i++)
    {
        // Inner loop executes only one
        // time due to break statement.
        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}
```

CPP

```cpp
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)

        // Executes O(Log n) times
        for (int j=1; j<=n; j = 2 * j)

            // Executes O(Log n) times
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

CPP

```cpp
void function(int n)
{
    int count = 0;

    // outer loop executes n/2 times
    for (int i=n/2; i<=n; i++)

        // middle loop executes  n/2 times
        for (int j=1; j+n/2<=n; j = j++)

            // inner loop executes logn times
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Time Complexity of the above function $O(n^2 \log n)$.

CPP

```cpp
void function(int n)
{
    int i = 1, s =1;
    while (s <= n)
    {
        i++;
        s += i;
        printf("*");
    }
}
```

Solution: We can define the terms 's' according to relation $s_i = s_{i-1} + i$. The value of 'i' increases by one for each iteration. The value contained in 's' at the i^{th} iteration is the sum of the first 'i' positive integers. If k is total number of iterations taken by the program, then while loop terminates if: $1 + 2 + 3 \ldots. + k = [k(k+1)/2] > n$ So $k = O(\sqrt{n})$. Time Complexity of the above function $O(\sqrt{n})$.

CPP

```cpp
void function(int n)
{
    int count = 0;

    // executes n times
    for (int i=0; i<n; i++)

        // executes O(n*n) times.
        for (int j=i; j< i*i; j++)
            if (j%i == 0)
            {
                // executes j times = O(n*n) times
                for (int k=0; k<j; k++)
                    printf("*");
            }
}
```

- O(N)

What is time complexity of following code :

```
int count = 0;
for (int i = N; i > 0; i /= 2) {
    for (int j = 0; j < i; j++) {
        count += 1;
    }
}
```

- O(n^2)

```
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

- O(2^(R+C))

```
int findMinPath(vector<vector<int> > &V, int r, int c) {
  int R = V.size();
  int C = V[0].size();
  if (r >= R || c >= C) return 100000000; // Infinity
  if (r == R - 1 && c == C - 1) return 0;
  return V[r][c] + min(findMinPath(V, r + 1, c), findMinPath(V, r, c + 1));
}
```

Assume R = V.size() and C = V[0].size().

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						

Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

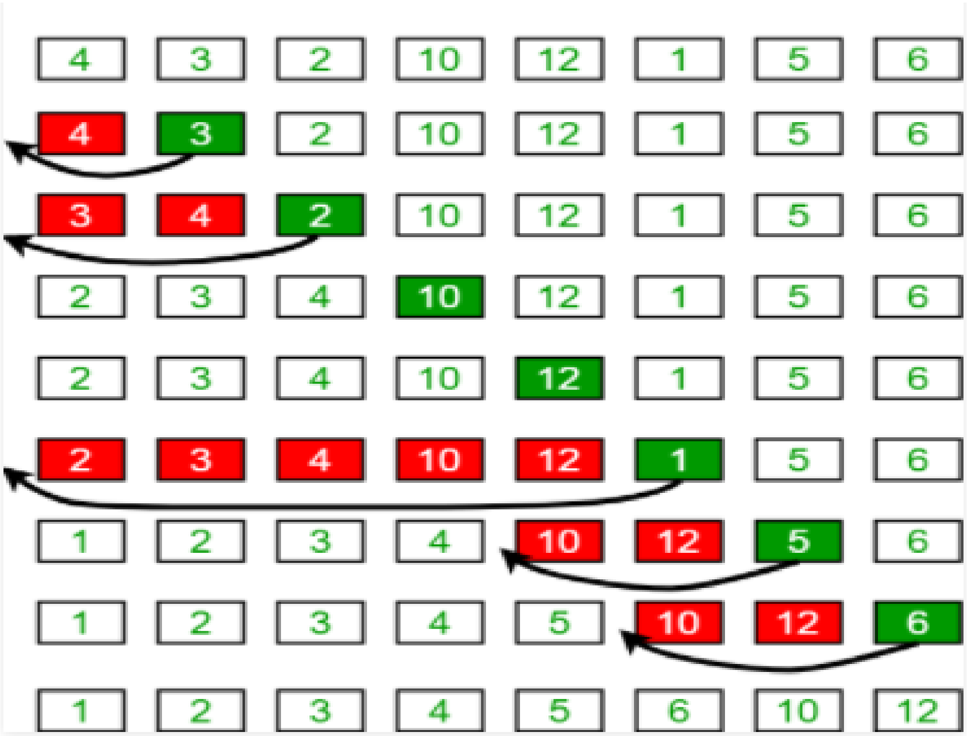
Auxiliary Space: O(1)

Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Sorting In Place: Yes

Stable: Yes

Insertion Sort



Time Complexity: O(n^2)

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

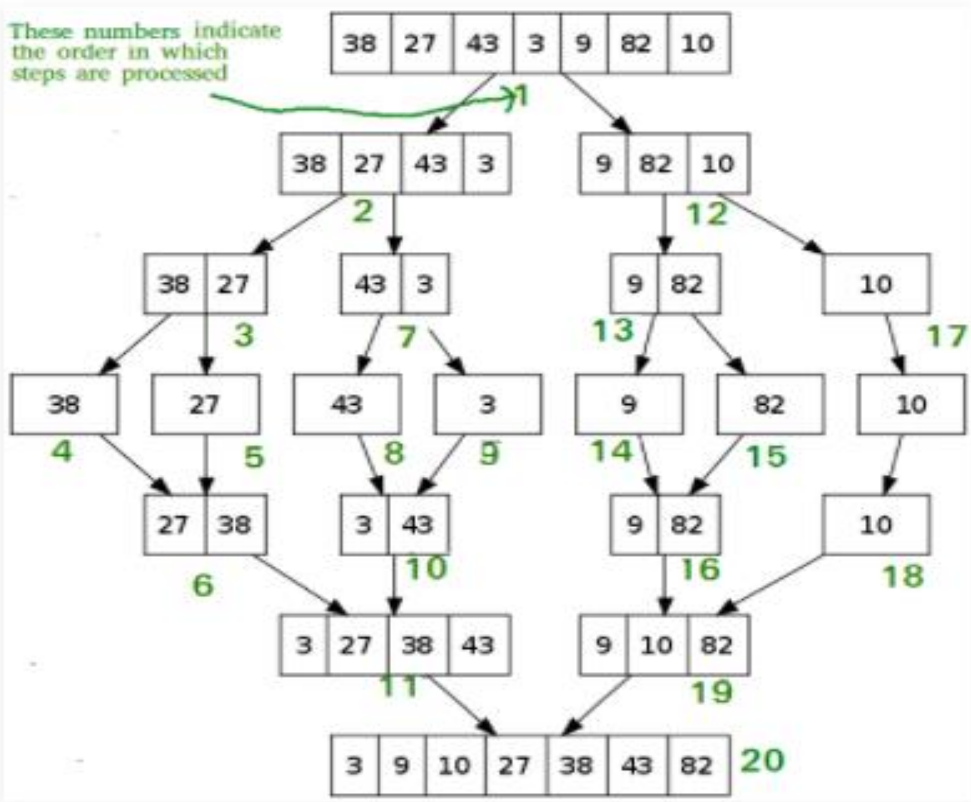
Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

Online: Yes

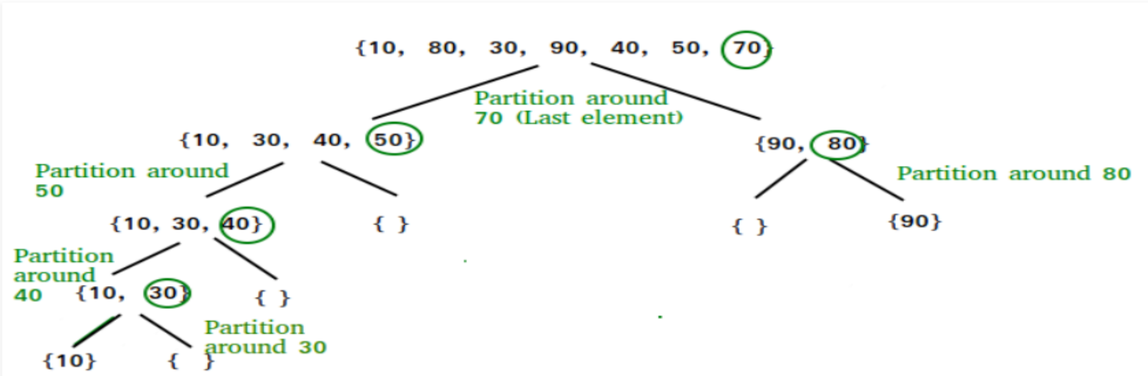
Merge Sort



Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
T(n) = 2T(n/2) + θ(n)

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of Master Method and the solution of the recurrence is θ(nLogn). Time complexity of Merge Sort is θ(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
Auxiliary Space: O(n)
Algorithmic Paradigm: Divide and Conquer
Sorting In Place: No in a typical implementation
Stable: Yes

Quick Sort



https://www.geeksforgeeks.org/quick-sort/

Selection Sort

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Time Complexity: $O(n^2)$ as there are two nested loops.
Auxiliary Space: O(1)
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

Heap Sort

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

Advantages of heapsort –

- **Efficiency –** The time required to perform Heap sort increases logarithmically while other algorithms may grow exponentially slower as the number of items to sort increases. This sorting algorithm is very efficient.
- **Memory Usage –** Memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- **Simplicity –** It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion