Time and Space Complexity

10 Dec 2021

Time and Space Complexity

- Efficiency
- Asymptotic notation
  - Big O
  - Big Ω
  - Big Θ
- What is time complexity?
- What is space complexity?
- Ascending order of Complexity
- Worst Accepted Algorithm
- Problems
- References

Efficiency

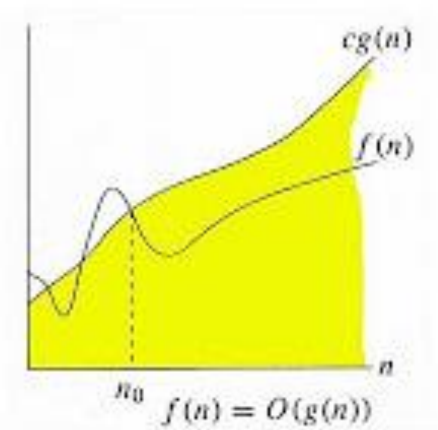
- How efficiency is the program you have written?
  - **Time Complexity** : How much time does it take program to complete?
  - **Space Complexity** : How much memory does this program use?
  - How do these complexities change as the amount of data changes?
    - E.g. From 1 to 10,000,000,000,000
  - What is the difference between the average case and worst case efficiency if any?

Asymptotic notation

Big O , Ω and Θ are formal notational methods for stating the growth of resource needs (efficiency and storage) of an algorithm.

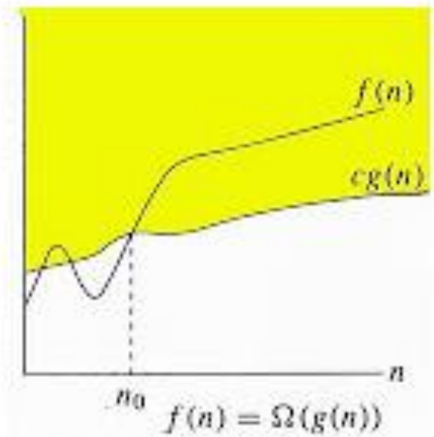
- **Big O - Worst case**
  - Upper bound of an algorithm
  - Rate of growth of an algorithm is less than or equal to a specific value
- **Big Ω Omega – Best case**
  - Lower bound of an algorithm
  - Rate of growth is greater than or equal to a specified value
- **Big Θ Theta – Average case**
  - Tight bound of an algorithm
  - Rate of growth is equal to a specified value

Asymptotic notation



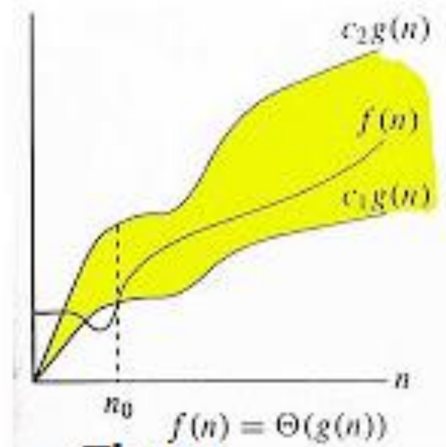
Big Oh

Worst case



Omega

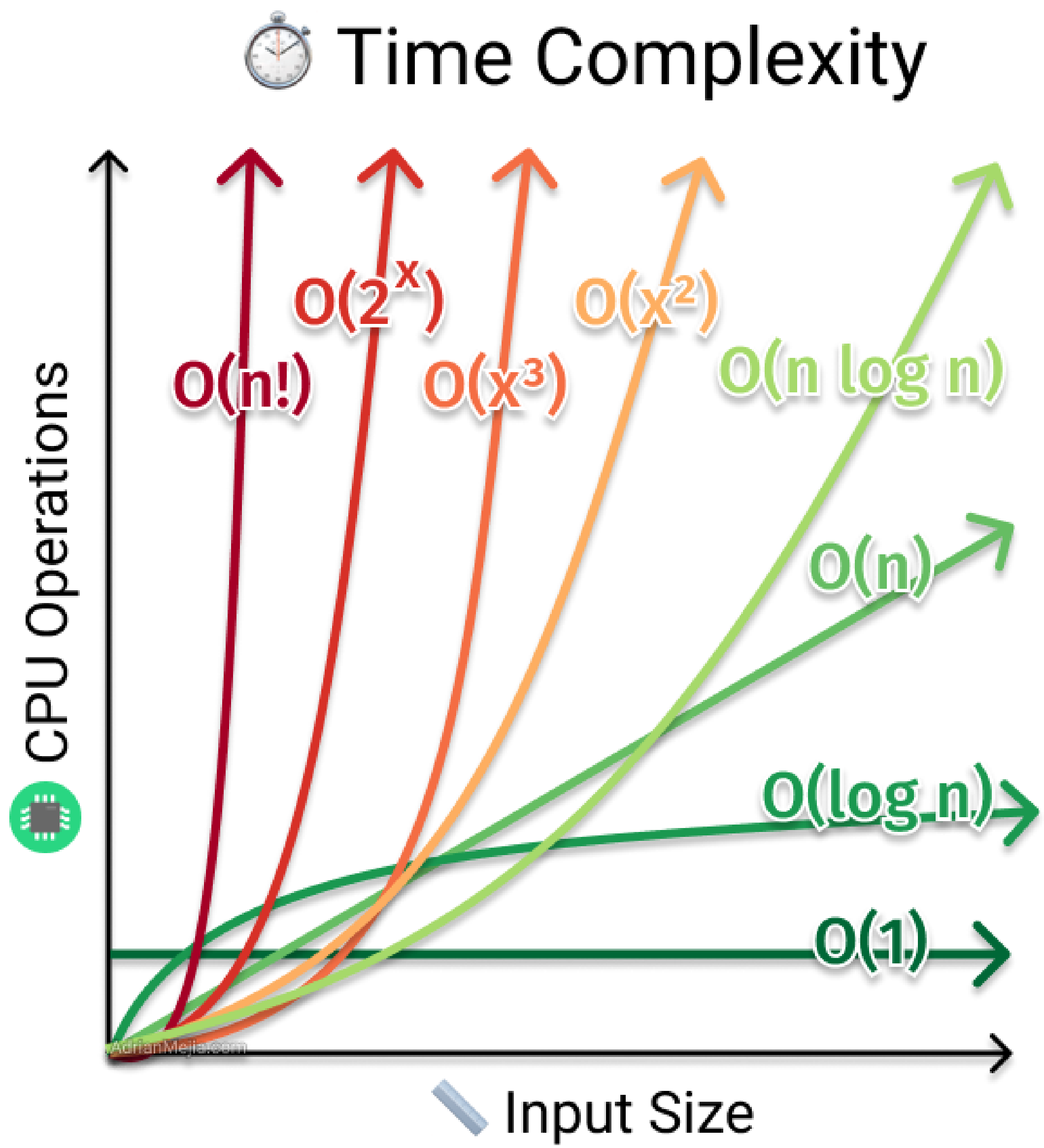
Best case



Theta

Average case

Time Complexity



Horrible

Bad

So so

Good

Excellent

Time Complexity

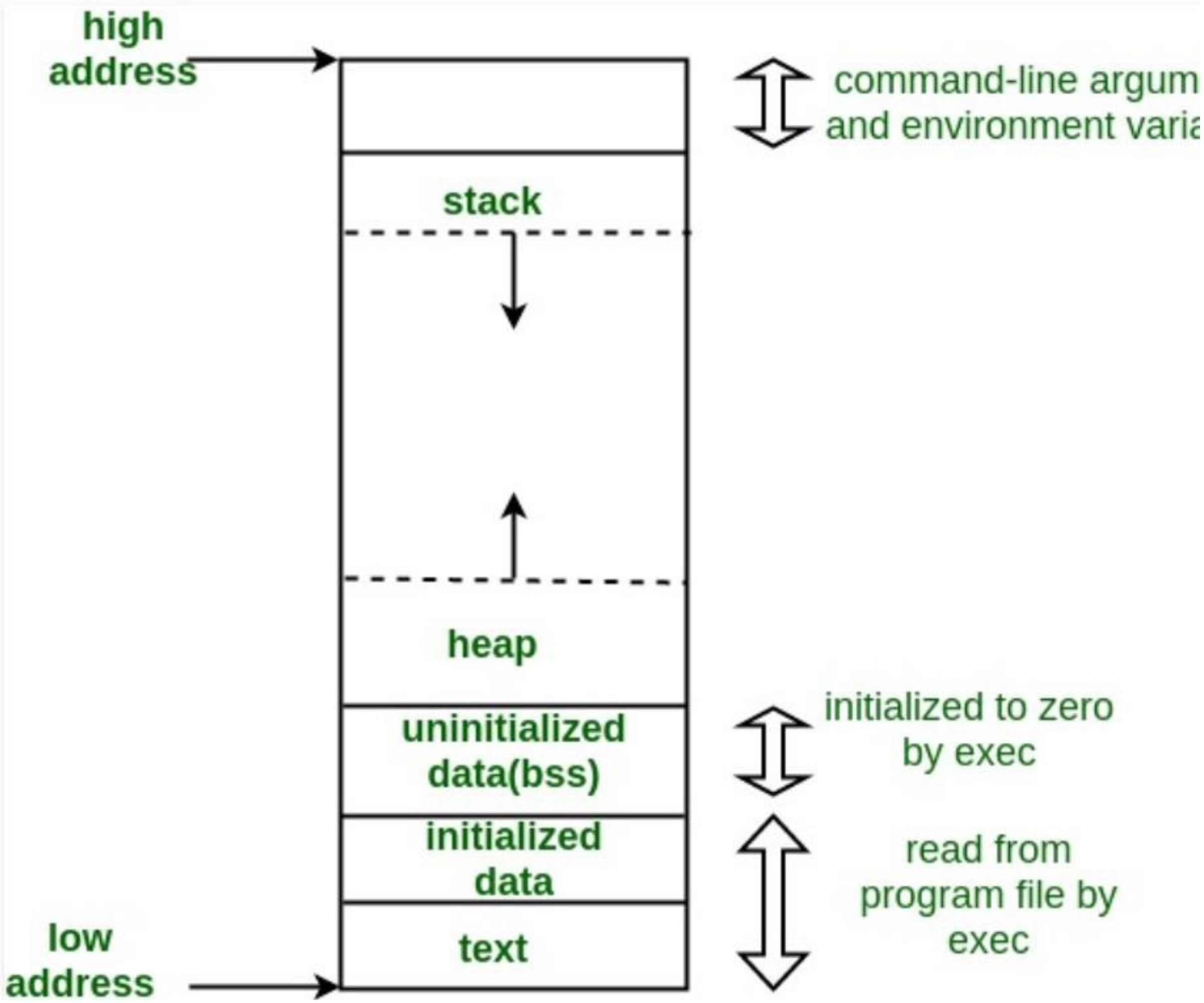
Assume N = 100,000 and processor speed is 1,000,000,000 operations per second

Function	Running Time
2^N	$3.2 \times 10^{30,086}$ years
N^4	3171 years
N^3	11.6 days
N^2	10 seconds
N N	0.032 seconds
N log N	0.0017 seconds
N	0.0001 seconds
N	3.2×10^{-7} seconds
log N	1.2×10^{-8} seconds

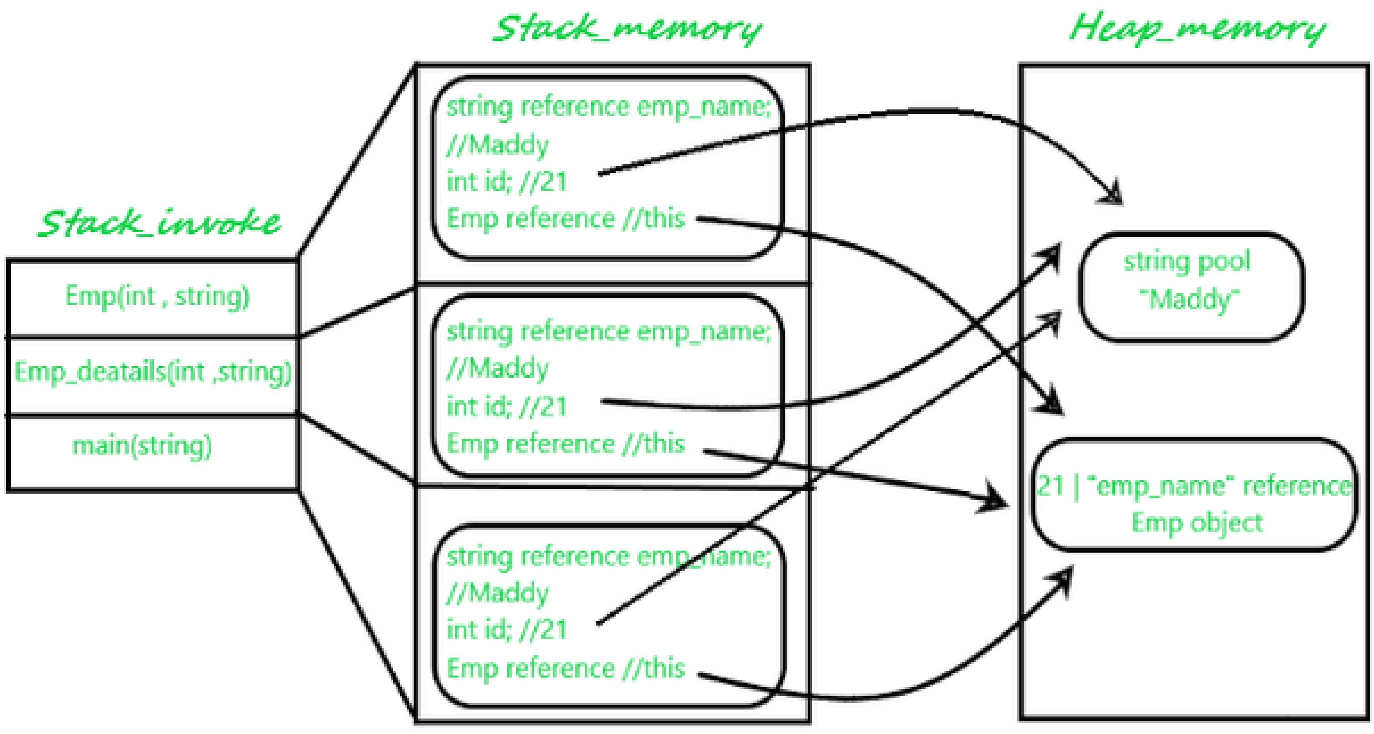
Space Complexity

- The amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.
- **Program instruction** – instruction space
  - amount of memory used to save the compiled version of instructions
- **Environmental stack**
  - One function calls another functions
  - Program stores the current variables to the system stack , while waiting for further execution
- **Data space**
  - Variables – space used by variables and constants constant values , temporary values
  - E.g. int , const, let , etc.

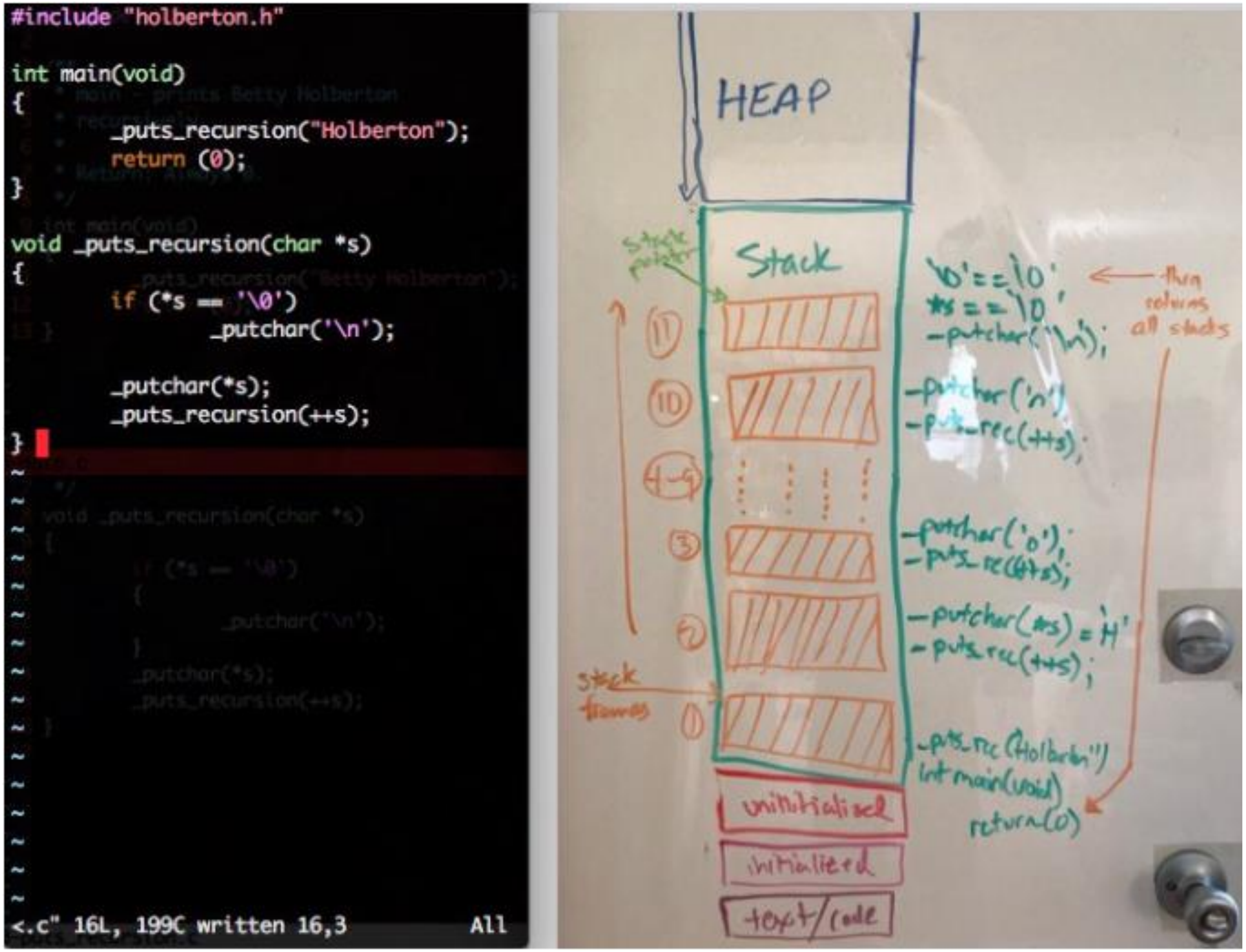
Space Complexity



Space Complexity



Space Complexity



Ascending order of Complexity

Function	Common Name
N!	factorial
2^N	Exponential
N^d, d > 3	Polynomial
N^3	Cubic
N^2	Quadratic
N N	N Square root N
N log N	N log N
N	Linear
N	Root - n
log N	Logarithmic
1	Constant

Running time grows 'quickly' with more input.

Running time grows 'slowly' with more input.

Worst Accepted Algorithm

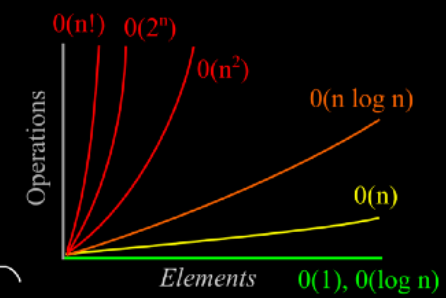
Length of Input (N)	Worst Accepted Algorithm
$\leq [10..11]$	$O(N!), O(N^6)$
$\leq [15..18]$	$O(2^N * N^2)$
$\leq [18..22]$	$O(2^N * N)$
≤ 100	$O(N^4)$
≤ 400	$O(N^3)$
$\leq 2K$	$O(N^2 * logN)$
$\leq 10K$	$O(N^2)$
$\leq 1M$	$O(N * logN)$
$\leq 100M$	$O(N), O(logN), O(1)$

Legend

TIME Complexity VS. SPACE Complexity

Good Fair Bad

Good Fair Bad



<BIG-O-CHEATSHEET>
</>

DATA STRUCTURE Operations

DATA Structure	TIME Complexity: Average – Access	Average – Search	Average – Insertion	Average – Deletion	Worst – Access	Worst – Search	Worst – Insertion	Worst – Deletion	SPACE Complexity: Worst
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

ARRAY SORTING Algorithms

ARRAY Algorithms	TIME Complexity: Best	Average	Worst	SPACE Complexity: Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(1)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Tree Sort	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Shell Sort	O(n log(n))	O(n log(n)^2)	O(n log(n)^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)
Cubesort	O(n)	O(n log(n))	O(n log(n))	O(n)

Data Structures Time and Space Complexity

Data Structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartresian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Array : Sorting -Time and Space Complexity

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Horrible Bad Fair Good Excellent

Problem – 1

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

1. O(N * M) time, O(1) space
2. O(N + M) time, O(N + M) space
3. O(N + M) time, O(1) space
4. O(N * M) time, O(N + M) space

Problem – 1

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

1. O(N * M) time, O(1) space
2. O(N + M) time, O(N + M) space
3. **O(N + M) time, O(1) space**
4. O(N * M) time, O(N + M) space

Problem – 2

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

1. O(N)
2. O(N*log(N))
3. O(N * Sqrt(N))
4. O(N*N)

Problem – 2

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

1. O(N)
2. O(N*log(N))
3. O(N * Sqrt(N))
4. **O(N*N)**

Problem – 3

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

1. O(n)
2. O(nLogn)
3. O(n^2)
4. O(n^2Logn)

Problem – 3

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

1. If n = 32
2. Outer loop
   1. i = 32/2 = 16 , i will be incremented by **1** (**n**)
3. Inner loop
   1. J = 2, j <=n ; j = 2 * 2 = 4 **(log n)**
   2. K = k + n /2

1. O(n)
2. **O(nLogn)**
3. O(n^2)
4. O(n^2Logn)

Problem – 4

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

1. O(N)
2. O(Sqrt(N))
3. O(N / 2)
4. O(log N)

Problem – 4

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

1. O(N)
2. O(Sqrt(N))
3. O(N / 2)
4. **O(log N)**

1. If a = 0 , i = 32
2. i > 0
3. a= 32
4. i = 32/2 = **16**

1. i > 0
2. a= 48
3. i = 16/2 = **8**

1. i > 0
2. a= 56
3. i = 8/2 = **4**

1. i > 0
2. a= 60
3. i = 4/2 = **2**

1. i > 0
2. a= 62
3. i = 2/2 = **1**

1. i > 0
2. a= 63
3. i = 1/2 = **0**

Problem – 4

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

1. O(N)
2. O(Sqrt(N))
3. O(N / 2)
4. O(log N)

Problem – 5 Leetcode

[(3) Sort an Array - LeetCode]

912. Sort an Array

Medium 1349 449 Add to List Share

Given an array of integers `nums`, sort the array in ascending order.

Example 1:

```
Input: nums = [5,2,3,1]
Output: [1,2,3,5]
```

Example 2:

```
Input: nums = [5,1,1,2,0,0]
Output: [0,0,1,1,2,5]
```

Constraints:

- $1 <= nums.length <= 5 * 10^4$
- $-5 * 10^4 <= nums[i] <= 5 * 10^4$

Problem – 5 Leetcode

```java
// Bubble sort
// In i-th pass of Bubble Sort (ascending order),
//last (i-1) elements are already sorted
// i-th largest element is placed at (N-i)-th position
class Solution {
   public int[] sortArray(int[] nums) {
        for(int i = 0 ; i < nums.length; i++){
         for(int j = 0; j < nums.length-1; j++){
            if (nums[j] > nums[j+1]){
                int temp = nums[j];
                nums[j] = nums[j+1];
                nums[j+1] = temp;
            }
         }
      }
      return nums;
   }
}
```

```java
// Selection Sort
// It divides the array into two parts:
//     -- sorted (left) and unsorted (right) subarray.
// It repeatedly selects the next smallest element.
class Solution {
   public int[] sortArray(int[] nums) {
      for(int i = 0 ; i < nums.length; i++){
         int minIndex = i;
         for(int j = i + 1; j < nums.length; j++){
            if (nums[j] < nums[minIndex]){
                int temp = nums[minIndex];
                nums[minIndex] = nums[j];
                nums[j] = temp;
            }
         }
      }
      return nums;
   }
}
```

Problem – 5 Leetcode

```
// Insertion Sort
// Compare current element temp to its predecececessor
// If key < , compare it to the elements before
// Move the greater elements one position up
class Solution {
    public int[] sortArray(int[] nums) {
        for(int i = 1 ; i < nums.length; i++){
            int temp = nums[i];
            int j = i - 1;
            // Move elements of nums, that are greater than temp to
            // one position ahead of their current position
            while(j >= 0 && nums[j] > temp){
                nums[j+1] = nums[j];
                j--;
            }
            nums[j+1] = temp;
        }
        return nums;
    }
}
```

Problem – 5 Leetcode

```
// Insertion Sort
// Compare current element temp to its predecececessor
// If key < , compare it to the elements before
// Move the greater elements one position up
class Solution {
    public int[] sortArray(int[] nums) {
        for(int i = 1 ; i < nums.length; i++){
            int temp = nums[i];
            int j = i - 1;
            // Move elements of nums, that are greater than temp to
            // one position ahead of their current position
            while(j >= 0 && nums[j] > temp){
                nums[j+1] = nums[j];
                j--;
            }
            nums[j+1] = temp;
        }
        return nums;
    }
}
```

Problem – 5 Leetcode

```java
class Solution {
    public int[] sortArray(int[] nums) {
        mergeSortRecursive(nums, 0, nums.length -1);
        return nums;
    }
    private static void mergeSortRecursive(int[] nums, int low, int high){
        if (high - low + 1 <= 1) return;
        // To prevent integer overflow
        if (mid = low + (high - low)/2);
        mergeSortRecursive(nums, low, mid);
        mergeSortRecursive(nums, mid+1, high);
        merge(nums, low, mid, high);
    }
    private static void merge(int[] nums, int low, int mid, int high){
        int[] temp = new int[high - low+1];
        int i = low;
        int j = mid+1;
        int tempIndex = 0;

        while(i <= mid && j <= high){
            if(nums[i] < nums[j]){
                temp[tempIndex++] = nums[i++];
            }else{
                temp[tempIndex++] = nums[j++];
            }
        }
        while(i <= mid){
            temp[tempIndex++] = nums[i++];
        }

        while(j <= high){
            temp[tempIndex++] = nums[j++];
        }
        for(int x = low; x <= high; x++){
            nums[x++] = temp[x-low];
        }
    }
}
```

Array : Sorting -Time and Space Complexity

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Horrible Bad Fair Good Excellent

References

- Know Thy Complexities!, https://www.bigocheatsheet.com/
- What's the Difference Between Big O, Big Omega, and Big Theta?, https://jarednielsen.com/big-o-omega-theta/
- Stack vs Heap Memory Allocation, https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/
- Time And Space Complexity of Data Structure and Sorting Algorithms, https://medium.com/@info.gildacademy/time-and-space-complexity-of-data-structure-and-sorting-algorithms-588a57edf495
- Practice Questions on Time Complexity Analysis, https://www.geeksforgeeks.org/practice-questions-time-complexity-analysis/