

Searching and Sorting algorithm

Sequential search

The linear search algorithm iteratively searches all elements of the array. It has the best execution time of one and the worst execution time of n , where n is the total number of items in the search array.

It is the simplest search algorithm in data structure and checks each item in the set of elements until it matches the searched element till the end of data collection. When the given data is unsorted, a linear search algorithm is preferred over other search algorithms.

Complexities in linear search are given below:

Space Complexity:

Since linear search uses no extra space, its space complexity is $O(n)$, where n is the number of elements in an array.

Time Complexity:

Best-case complexity = $O(1)$ occurs when the searched item is present at the first element in the search array.

Worst-case complexity = $O(n)$ occurs when the required element is at the tail of the array or not present at all.

Average- case complexity = average case occurs when the item to be searched is in somewhere middle of the Array.

Pseudocode for the linear search algorithm

```
procedure linear_search (list, value)
```

```
    for each item in the list
```

```
        if match item == value
```

```
            return the item's location
```

```
    end if  
end for  
  
end procedure
```

Example,

Let's take the following array of elements:

45, 78, 15, 67, 08, 29, 39, 40, 12, 99

To find '29' in an array of 10 elements given above, as we know linear search algorithm will check each element sequentially till its pointer points to 29 in the memory space. It takes $O(6)$ time to find 29 in an array. To find 15, in the above array, it takes $O(3)$, whereas, for 39, it requires $O(7)$ time.

Binary Search

This algorithm locates specific items by comparing the middlemost items in the data collection. When a match is found, it returns the index of the item. When the middle item is greater than the search item, it looks for a central item of the left sub-array. If, on the other hand, the middle item is smaller than the search item, it explores for the middle item in the right sub-array. It keeps looking for an item until it finds it or the size of the sub-arrays reaches zero.

Binary search needs sorted order of items of the array. It works faster than a linear search algorithm. The binary search uses the divide and conquers principle.

Run-time complexity = $O(\log n)$

Complexities in binary search are given below:

The worst-case complexity in binary search is $O(\log n)$.

The average case complexity in binary search is $O(\log n)$

Best case complexity = $O(1)$

Pseudocode for the Binary search algorithm

```
Procedure binary_search
```

```
    A ← sorted array
```

```
    n ← size of array
```

$x \leftarrow$ value to be searched

Set lowerBound = 1

Set upperBound = n

while x not found

if upperBound < lowerBound

 EXIT: x does not exists.

 set midPoint = lowerBound + (upperBound - lowerBound) / 2

 if A[midPoint] = x

 set upperBound = midPoint - 1

 if A[midPoint] = x

 EXIT: x found at location midPoint

end while

end procedure

Example,

Let's take a sorted array of 08 elements:

09, 12, 26, 39, 45, 61, 67, 78

To find 61 in an array of the above elements,

The algorithm will divide an array into two arrays, 09, 12, 26, 39 and 45, 61, 67, 78

As 61 is greater than 39, it will start searching for elements on the right side of the array.

It will further divide the into two such as 45, 61 and 67, 78

As 61 is smaller than 67, it will start searching on the left of that sub-array.

That subarray is again divided into two as 45 and 61.

As 61 is the number matching to the search element, it will return the index number of that element in the array.

It will conclude that the search element 61 is located at the 6th position in an array.

Binary search reduces the time to half as the comparison count is reduced significantly as compared to the linear search algorithm.

Sorting techniques

Types of Sorting in Data Structures

Comparison-based sorting: In comparison-based sorting techniques, a comparator is defined to compare elements or items of a data sample. This comparator defines the ordering of elements. Examples are: Bubble Sort, Merge Sort.

Counting-based sorting: There's no comparison involved between elements in these types of sorting algorithms but rather work on calculated assumptions during execution. Examples are : Counting Sort, Radix Sort.

In-Place vs Not-in-Place Sorting: In-place sorting techniques in data structures modify the ordering of array elements within the original array. On the other hand, Not-in-Place sorting techniques use an auxiliary data structure to sort the original array. Examples of In place sorting techniques are: Bubble Sort, Selection Sort, Quick Sort.

Some examples of Not in Place sorting algorithms are: Merge Sort

Bubble Sort

The basic idea of bubble sorting is that it repeatedly swaps adjacent elements if they are not in the desired order. YES, it is as simple as that.

If a given array of elements has to be sorted in ascending order, bubble sorting will start by comparing the first element of the array with the second element and immediately swap them if it turns out to be greater than the second element, and then move on to compare the second and third element, and so on.

Time Complexity:

Worst Case: $O(n^2)$

Average Case: $O(n \cdot \log n)$

Best case: $O(n \cdot \log n)$

Auxiliary Space Complexity: $O(1)$

Use Cases

It is used to introduce the concept of a sorting algorithm to Computer Science students.

In computer graphics, bubble sorting is quite popular when it comes to detecting a very small error (like swap of just two elements) in almost-sorted arrays.

Algorithm:

```
Bubble_Sort(int a[], n)
{
    int i, j;
    //number of iteration
    for (i=0; i<n; i++)
    {
        //this loop will compare data for swapping
        for (j=0; j<n-i-1; j++) // n-1-0  n-1-1  n-1-2
        {
            if (a[j] > a[j+1])
            {
                swap (a[j], a[j+1]);
            }
        }
    }
}
```

Modified bubble sort

Algorithm:

```
Bubble_Sort(int a[], n)
{
    int swapped, i, j;
    for (i=0; i<n; i++)
    {

```

```

flag=false;

for (j=0; j<n-i-1; j++)

{
    if (a[j] > a[j+1])

    {
        swap (a[j], a[j+1]);

        flag=true
    }
}

if (!flag)

    break;

}
}

```

Selection Sort

Selection sort is a sorting algorithm in which the given array is divided into two subarrays, the sorted left section, and the unsorted right section.

Initially, the sorted portion is empty and the unsorted part is the entire list. In each iteration, we fetch the minimum element from the unsorted list and push it to the end of the sorted list thus building our sorted array.

Selection Sort Algorithm

Now that you know how selection sort works, following the algorithm steps will be pretty easy:

- **Step 1:** Set minIndex to position 0
(minIndex will hold the index of the smallest number in the unsorted subarray)
- **Step 2:** Search for the smallest element in the unsorted subarray and update minIndex
- **Step 3:** Swap the element at the position minIndex with the first element of the unsorted subarray.
- **Step 4:** Again set minIndex to the first position of the unsorted subarray
- **Step 5:** Repeat steps 2 to 4 until the array gets sorted

Selection Sort Pseudocode

Once you understand the logic, writing the code for selection sort is pretty straightforward. Are you feeling confident yet?

Here's the pseudocode to help you write the code in any language of your choice.

```
procedure selection sort
arr : array of integers
n : size of the array

for i =1 to n-1
    // set current element as minIndex
    minIndex=i
    // check for all other element(right side of current element)
    for j =i+1 to n
        if(arr[j] < arr[minIndex])
            minIndex=j
        end if
    end for

    // swap the current element with the minimum element to the right side
    swap(arr[minIndex], arr[i])
end for

End procedure
```

Time Complexity:

Worst Case: $O(n^2)$

Average Case: $O(n^2)$

Best case: $O(n)$

Auxiliary Space Complexity: $O(1)$

Also read: Time Complexity Simplified with Easy Examples

Use Cases

It is used when the size of a list is small. (Time complexity of selection sort is $O(N^2)$ which makes it inefficient for a large list.)

It is also used when memory space is limited because it makes the minimum possible number of swaps during sorting.

3. Insertion Sort

Insertion sort is a sorting algorithm in which the given array is divided into a sorted and an unsorted section. In each iteration, the element to be inserted has to find its optimal position in the sorted subsequence and is then inserted while shifting the remaining elements to the right.

Time Complexity:

Worst Case: $O(n*n)$

Average Case: $O(n*logn)$

Best case: $O(n*logn)$

Auxiliary Space Complexity: $O(1)$

Use Cases

This algorithm is stable and is quite fast when the list is nearly sorted.

It is very efficient when it comes to sorting very small lists (example say 30 elements.)

4. Quick Sort

Quick Sort is a divide and conquer algorithm. The intuitive idea behind quick sort is it picks an element as the pivot from a given array of elements and then partitions the array around the pivot element. Subsequently, it calls itself recursively and partitions the two subarrays thereafter.

The logical steps involved in Quick Sort algorithm are as follows:

- **Pivot selection:** Picks an element as the pivot (here, we choose the last element as the pivot)
- **Partitioning:** The array is partitioned in a fashion such that all elements less than the pivot are in the left subarray while all elements strictly greater than the pivot element are stored in the right subarray.
- **Recursive call to Quicksort:** Quicksort function is invoked again for the two subarrays created above and steps are repeated.

Time Complexity:

- Worst Case: $O(n^2)$
- Average Case: $O(n \log n)$
- Best case: $O(n \log n)$

Space Complexity

Unlike merge sort quicksort doesn't require any extra space to store arrays but additional memory is required for creating stack frames in recursive calls. So, the space complexity again depends on the pivot.

Worse Case

If we select the largest or smallest element as a pivot, there are in total N recursive calls, so the size of the recursion tree will be N . Thus, the space complexity for such a case is $O(N)$ which is the worst case.

Best Case

If we manage to partition the array in equal halves each time, then the size of the recursion tree is $\log N$. So, in this case, the space complexity is $O(N \log N)$.

Use Cases

- Quicksort is probably more effective for datasets that fit in memory.
- Quick Sort is appropriate for arrays since is an in-place sorting algorithm (i.e. it doesn't require any extra storage)

5. Merge Sort

Merge Sort is a divide-and-conquer algorithm. In each iteration, merge sort divides the input array into two equal subarrays, calls itself recursively for the two subarrays, and finally merges the two sorted halves.

Time Complexity:

- Worst Case: $O(n \log n)$
- Average Case: $O(n \log n)$
- Best case: $O(n \log n)$

Space Complexity: O(n)

Use Cases

- Merge sort is quite fast in the case of linked lists.
- It is widely used for external sorting where random access can be quite expensive compared to sequential access.

6. Counting Sort

Counting Sort is an interesting sorting technique primarily because it focuses on the frequency of unique elements between a specific range (something along the lines of hashing).

It works by counting the number of elements having distinct key values and then building a sorted array after calculating the position of each unique element in the unsorted sequence.

It stands apart from the algorithms listed above because it literally involves zero comparisons between the input data elements!

Time Complexity:

- *Worst Case: O(n+k)*, where n is the size of input array and k is the count of unique elements in the array

Auxiliary Space Complexity: O(n+k)

Use Cases

- Counting sort is quite efficient when it comes to sorting countable objects (such as bounded integers).
- It is also used when sorting is to be achieved in linear complexity.

8. Heap Sort

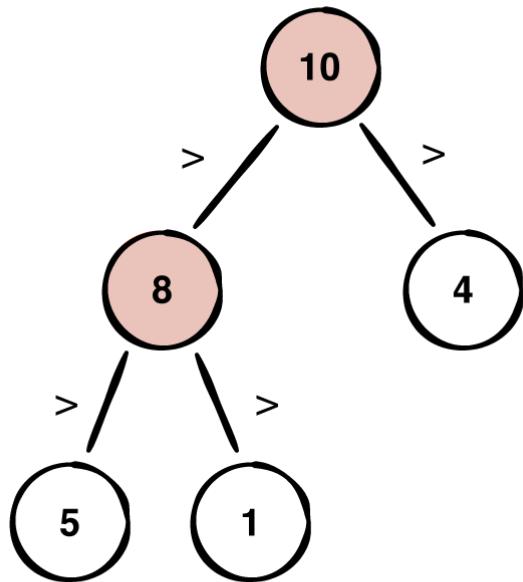
Heapsort is another comparison-based algorithm that sorts an array in ascending order using a heap. This chapter builds on the heap concepts presented in Chapter 12, “Heap Data Structure.”

Heapsort takes advantage of a heap being, by definition, a partially sorted binary tree with the following qualities:

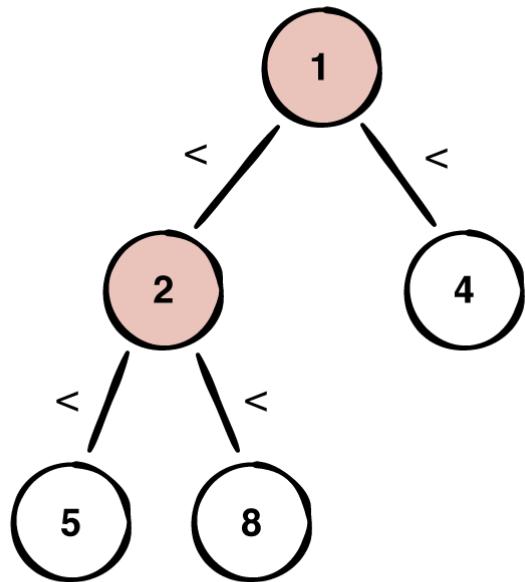
1. In a max heap, all parent nodes are larger than their children.

2. In a min heap, all parent nodes are smaller than their children.

The diagram below shows a heap with parent node values underlined:



Max heap

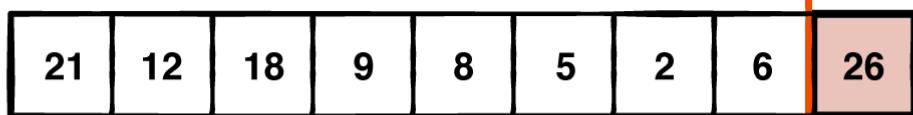
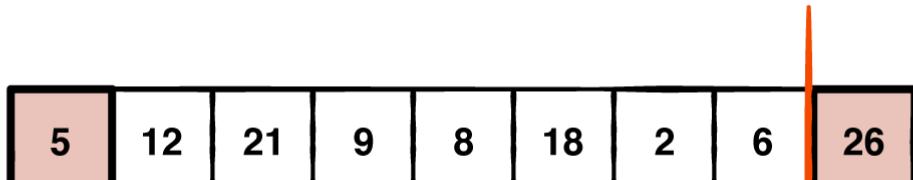
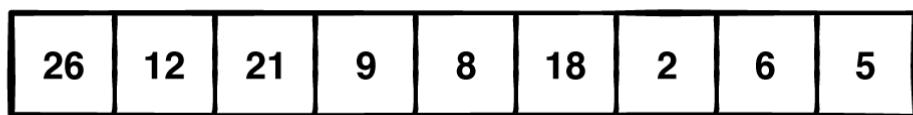
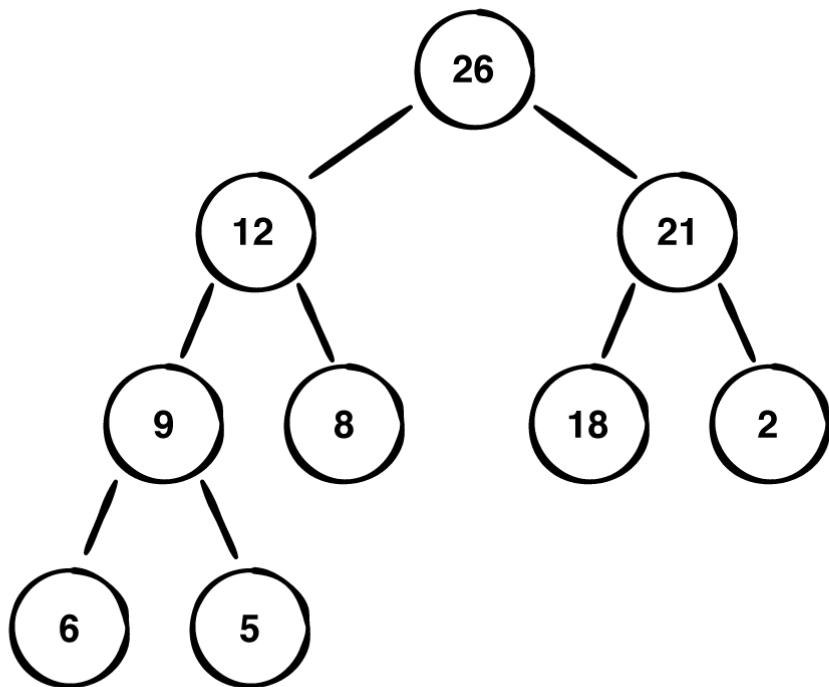


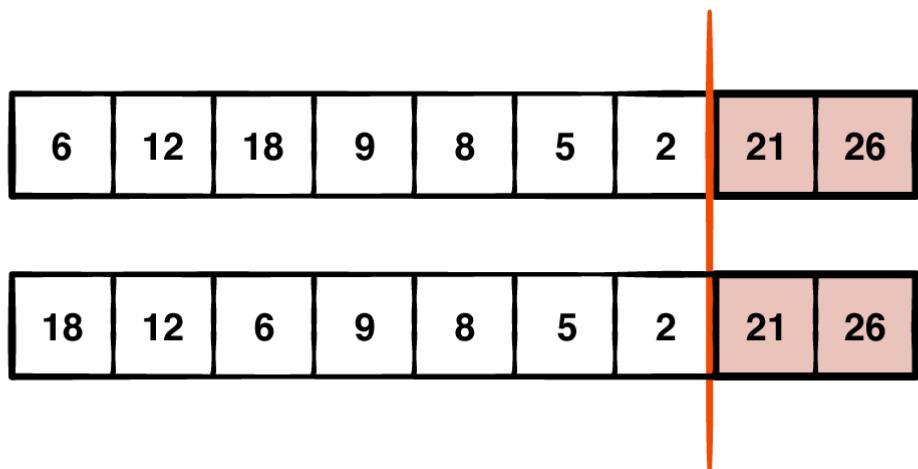
Min heap

Example

For any given unsorted array, to sort from lowest to highest, heap sort must first convert this array into a heap:

6	12	2	26	8	18	21	9	5
---	----	---	----	---	----	----	---	---





Count sort

Algorithm

1. countingSort(array, n) *// 'n' is the size of array*
2. max = find maximum element in the given array
3. create count array with size maximum + 1
4. Initialize count array with all 0's
5. **for** i = 0 to n
 1. find the count of every unique element and
 2. store that count at ith position in the count array
6. **for** j = 1 to max
 1. Now, find the cumulative sum and store it in count array
7. **for** i = n to 1
8. Restore the array elements
9. Decrease the count of every restored element by 1
10. end countingSort

Working of counting sort Algorithm

Now, let's see the working of the counting sort Algorithm.

To understand the working of the counting sort algorithm, let's take an unsorted array. It will be easier to understand the counting sort via an example.

Let the elements of array are -

2	9	7	4	1	8	4
---	---	---	---	---	---	---

1. Find the maximum element from the given array. Let **max** be the maximum element.

max

9	2	7	4	1	8	4
---	---	---	---	---	---	---

2. Now, initialize array of length **max + 1** having all 0 elements. This array will be used to store the count of the elements in the given array.

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Count array

3. Now, we have to store the count of each array element at their corresponding index in the count array.

The count of an element will be stored as - Suppose array element '4' is appeared two times, so the count of element 4 is 2. Hence, 2 is stored at the 4th position of the count array. If any element is not present in the array, place 0, i.e. suppose element '3' is not present in the array, so, 0 will be stored at 3rd position.

Given array	2	9	7	4	1	8	4
Count array	0	1	1	0	2	0	0

Count of each stored element

Now, store the cumulative sum of **count** array elements. It will help to place the elements at the correct index of the sorted array.

0	1	2	3	4	5	6	7	8	9
0	1	2	0	2	0	0	1	1	1
$1+1=2$									
$2+0=2$									
0	1	2	3	4	5	6	7	8	9
0	1	2	2	2	0	0	1	1	1

Similarly, the cumulative count of the count array is -

0	1	2	3	4	5	6	7	8	9
0	1	2	2	4	4	4	5	6	7
Cumulative count									

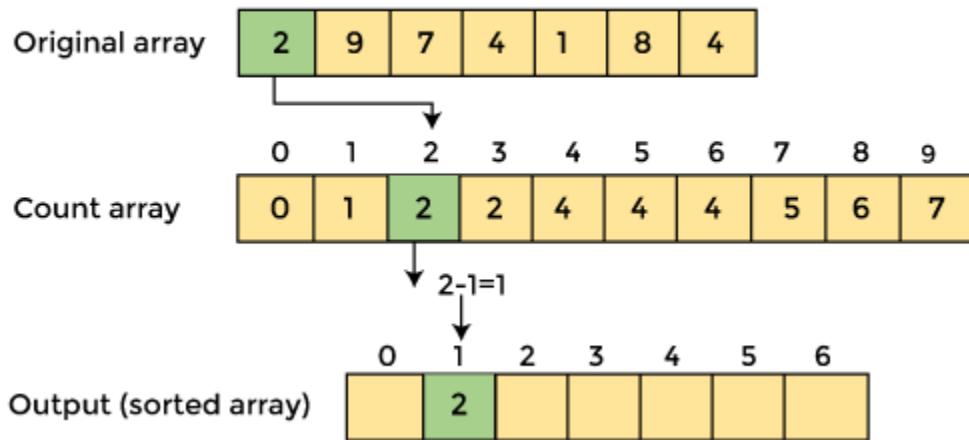
4. Now, find the index of each element of the original array

For element 9

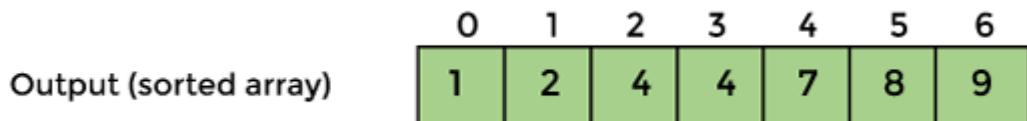
Original array	2	9	7	4	1	8	4			
Count array	0	1	1	2	4	4	4	5	6	7
Output (sorted array)		2						9		

After placing element at its place, decrease its count by one. Before placing element 2, its count was 2, but after placing it at its correct position, the new count for element 2 is 1.

For element 2



Similarly, after sorting, the array elements are -



Now, the array is completely sorted.

```
// Counting sort in Java programming

import java.util.Arrays;

class CountingSort {
    void countSort(int array[], int size) {
        int[] output = new int[size + 1];

        // Find the largest element of the array
        int max = array[0];
        for (int i = 1; i < size; i++) {
            if (array[i] > max)
                max = array[i];
        }
        int[] count = new int[max + 1];

        // Initialize count array with all zeros.
        for (int i = 0; i < max; ++i)
            count[i] = 0;
        }

        // Store the count of each element
        for (int i = 0; i < size; i++) {
```

```

        count[array[i]]++;
    }

    // Store the cumulative count of each array
    for (int i = 1; i <= max; i++) {
        count[i] += count[i - 1];
    }

    // Find the index of each element of the original array in count array, and
    // place the elements in output array
    for (int i = size - 1; i >= 0; i--) {
        output[count[array[i]] - 1] = array[i];
        count[array[i]]--;
    }

    // Copy the sorted elements into original array
    for (int i = 0; i < size; i++) {
        array[i] = output[i];
    }
}

// Driver code
public static void main(String args[]) {
    int[] data = { 4, 2, 2, 8, 3, 3, 1 };
    int size = data.length;
    CountingSort cs = new CountingSort();
    cs.countSort(data, size);
    System.out.println("Sorted Array in Ascending Order: ");
    System.out.println(Arrays.toString(data));
}
}

```

Complexity

Time Complexity

Best

$O(n+k)$

Worst

$O(n+k)$

Average

$O(n+k)$

Space Complexity

$O(\max)$

Counting Sort Algorithm

Counting sort is [a sorting algorithm](#) that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

Working of Counting Sort

1. Find out the maximum element (let it be \max) from the given array.



Given array

2. Initialize an array of length $\max+1$ with all elements 0. This array is used for storing the count of the elements in the array.



Count array

3. Store the count of each element at their respective index in count array

For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of `count` array. If element "5" is not present in the array, then 0 is stored in 5th position.

0	1	2	2	1	0	0	0	1
0	1	2	3	4	5	6	7	8

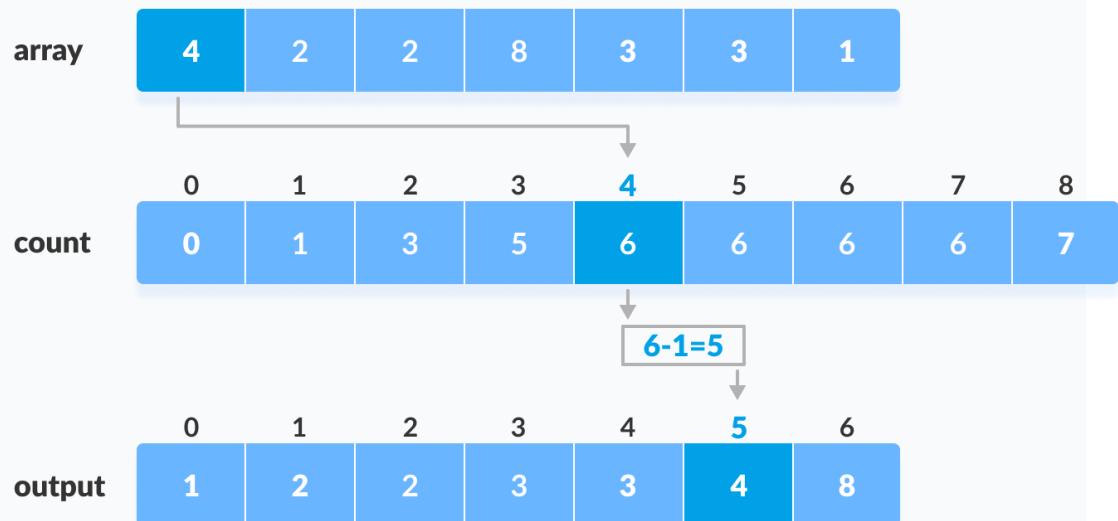
Count of each element stored

4. Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.

0	1	3	5	6	6	6	6	7
0	1	2	3	4	5	6	7	8

Cumulative count

5. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.



Counting sort

6. After placing each element at its correct position, decrease its count by one.

2	12	6	9	8	5	18	21	26
---	----	---	---	---	---	----	----	----

12	9	6	2	8	5	18	21	26
----	---	---	---	---	---	----	----	----

5	9	6	2	8	12	18	21	26
---	---	---	---	---	----	----	----	----

9	8	6	2	5	12	18	21	26
---	---	---	---	---	----	----	----	----

5	8	6	2	9	12	18	21	26
---	---	---	---	---	----	----	----	----

8	5	6	2	9	12	18	21	26
---	---	---	---	---	----	----	----	----

2	5	6	8	9	12	18	21	26
---	---	---	---	---	----	----	----	----

6	5	2	8	9	12	18	21	26
---	---	---	---	---	----	----	----	----

2	5	6	8	9	12	18	21	26
---	---	---	---	---	----	----	----	----

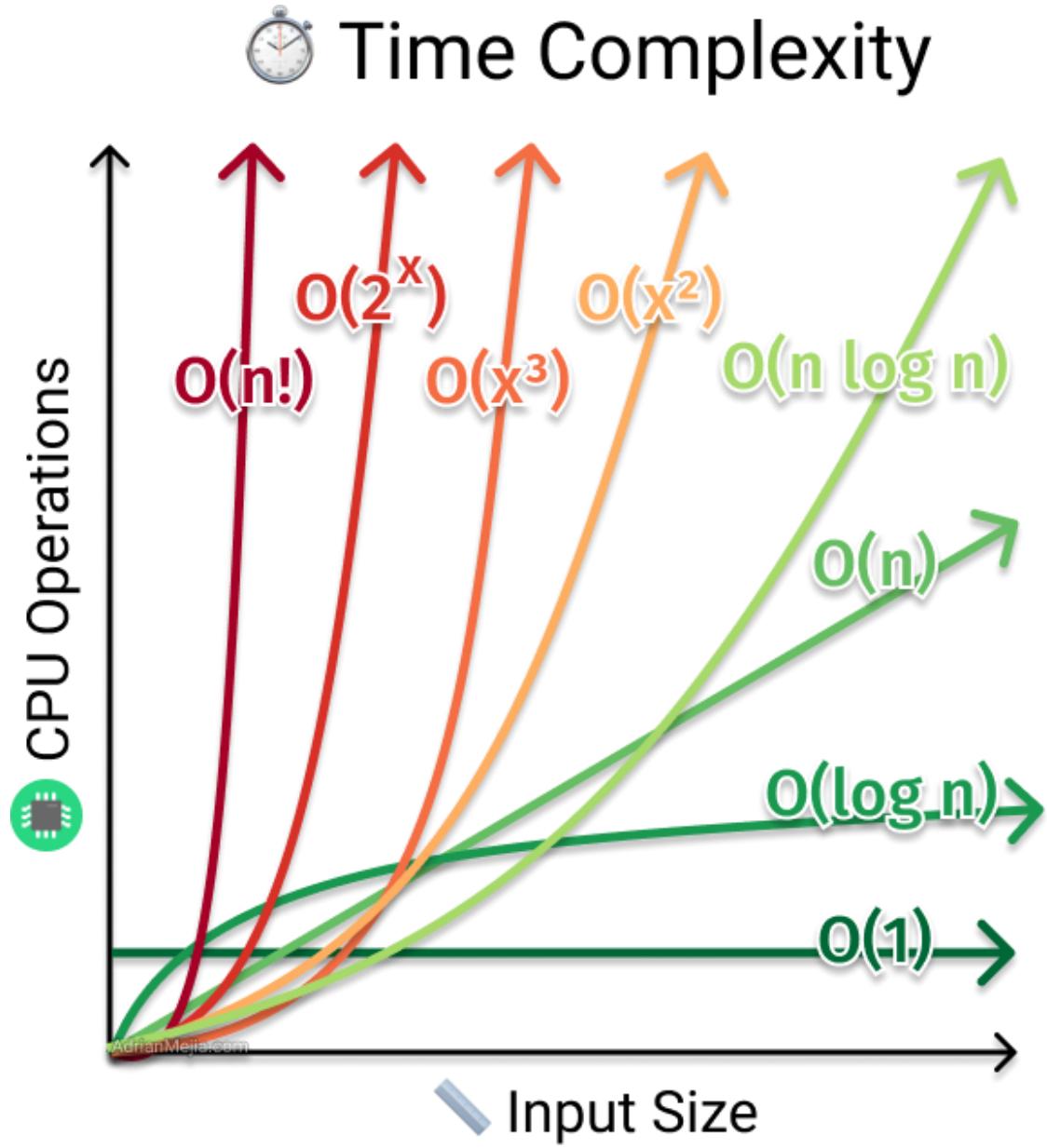
5	2	6	8	9	12	18	21	26
---	---	---	---	---	----	----	----	----

2	5	6	8	9	12	18	21	26
---	---	---	---	---	----	----	----	----

2	5	6	8	9	12	18	21	26
---	---	---	---	---	----	----	----	----

2	5	6	8	9	12	18	21	26
---	---	---	---	---	----	----	----	----

Time complexity



Sequential Statements

If we have statements with basic operations like comparisons, assignments, reading a variable. We can assume they take constant time each $O(1)$.

```
1statement1;  
2statement2;
```

```
3...
4statementN;
```

If we calculate the total time complexity, it would be something like this:

```
1total = time(statement1) + time(statement2) + ... time (statementN)
```

Let's use $T(n)$ as the total time in function of the input size n , and t as the time complexity taken by a statement or group of statements.

```
1T(n) = t(statement1) + t(statement2) + ... + t(statementN);
```

If each statement executes a basic operation, we can say it takes constant time $O(1)$. As long as you have a fixed number of operations, it will be constant time, even if we have 1 or 100 of these statements.

Example:

Let's say we can compute the square sum of 3 numbers.

```
1function squareSum(a, b, c) {
2  const sa = a * a;
3  const sb = b * b;
4  const sc = c * c;
5  const sum = sa + sb + sc;
6  return sum;
7}
```

As you can see, each statement is a basic operation (math and assignment). Each line takes constant time $O(1)$. If we add up all statements' time it will still be $O(1)$. It doesn't matter if the numbers are 0 or $9,007,199,254,740,991$, it will perform the same number of operations.

 Be careful with function calls. You will have to go to the implementation and check their run time. More on that later.

Conditional Statements

Very rarely, you have a code without any conditional statement. How do you calculate the time complexity? Remember that we care about the worst-case with Big O so that we will take the maximum possible runtime.

```
1if (isValid) {  
2    statement1;  
3    statement2;  
4} else {  
5    statement3;  
6}
```

Since we are after the worst-case we take whichever is larger:

```
1T(n) = Math.max([t(statement1) + t(statement2)], [time(statement3)])
```

Example:

```
1if (isValid) {  
2    array.sort();  
3    return true;  
4} else {  
5    return false;  
6}
```

What's the runtime? The `if` block has a runtime of $O(n \log n)$ (that's common runtime for [efficient sorting algorithms](#)). The `else` block has a runtime of $O(1)$.

So we have the following:

```
1O([n log n] + [n]) => O(n log n)
```

Since $n \log n$ has a higher order than n , we can express the time complexity as $O(n \log n)$.

Loop Statements

Another prevalent scenario is loops like for-loops or while-loops.

Linear Time Loops

For any loop, we find out the runtime of the block inside them and multiply it by the number of times the program will repeat the loop.

```
1for (let i = 0; i < array.length; i++) {  
2    statement1;
```

```
3 statement2;  
4}
```

For this example, the loop is executed `array.length`, assuming `n` is the length of the array, we get the following:

```
1T(n) = n * [ t(statement1) + t(statement2) ]
```

All loops that grow proportionally to the input size have a linear time complexity `O(n)`. If you loop through only half of the array, that's still `O(n)`. Remember that we drop the constants so $1/2 \cdot n \Rightarrow O(n)$.

Constant-Time Loops

However, if a constant number bounds the loop, let's say 4 (or even 400). Then, the runtime is constant `O(4) → O(1)`. See the following example.

```
1for (let i = 0; i < 4; i++) {  
2    statement1;  
3    statement2;  
4}
```

That code is `O(1)` because it no longer depends on the input size. It will always run statement 1 and 2 four times.

Logarithmic Time Loops

Consider the following code, where we divide an array in half on each iteration (binary search):

```
1function fn1(array, target, low = 0, high = array.length - 1) {  
2    let mid;  
3    while ( low <= high ) {  
4        mid = ( low + high ) / 2;  
5        if ( target < array[mid] )  
6            high = mid - 1;  
7        else if ( target > array[mid] )  
8            low = mid + 1;  
9        else break;  
10    }  
11    return mid;  
12}
```

This function divides the array by its `middle` point on each iteration. The while loop will execute the amount of times that we can divide `array.length` in half. We can calculate this using the `log` function. E.g. If the array's length is 8, then the while loop will execute 3 times because $\log_2(8) = 3$.

Nested loops statements

Sometimes you might need to visit all the elements on a 2D array (grid/table). For such cases, you might find yourself with two nested loops.

```
1 for (let i = 0; i < n; i++) {  
2     statement1;  
3  
4     for (let j = 0; j < m; j++) {  
5         statement2;  
6         statement3;  
7     }  
8}
```

For this case, you would have something like this:

```
1 T(n) = n * [t(statement1) + m * t(statement2...3)]
```

Assuming the statements from 1 to 3 are $O(1)$, we would have a runtime of $O(n * m)$. If instead of `m`, you had to iterate on `n` again, then it would be $O(n^2)$. Another typical case is having a function inside a loop. Let's see how to deal with that next.

Function call statements

When you calculate your programs' time complexity and invoke a function, you need to be aware of its runtime. If you created the function, that might be a simple inspection of the implementation. If you are using a library function, you might need to check out the language/library documentation or source code.

Let's say you have the following program:

```
1 for (let i = 0; i < n; i++) {  
2     fn1();  
3     for (let j = 0; j < n; j++) {
```

```

4     fn2();
5     for (let k = 0; k < n; k++) {
6         fn3();
7     }
8 }
9

```

Depending on the runtime of fn1, fn2, and fn3, you would have different runtimes.

- If they all are constant $O(1)$, then the final runtime would be $O(n^3)$.
- However, if only `fn1` and `fn2` are constant and `fn3` has a runtime of $O(n^2)$, this program will have a runtime of $O(n^5)$. Another way to look at it is, if `fn3` has two nested and you replace the invocation with the actual implementation, you would have five nested loops.

In general, you will have something like this:

```
t(n) = n * [ t(fn1()) + n * [ t(fn2()) + n * [ t(fn3()) ] ] ]
```

Recursive Functions Statements

Analyzing the runtime of recursive functions might get a little tricky. There are different ways to do it. One intuitive way is to explore the recursion tree.

Let's say that we have the following program:

```

1 function fn(n) {
2     if (n < 0) return 0;
3     if (n < 2) return n;
4
5     return fn(n - 1) + fn(n - 2);
6}

```

You can represent each function invocation as a bubble (or node).

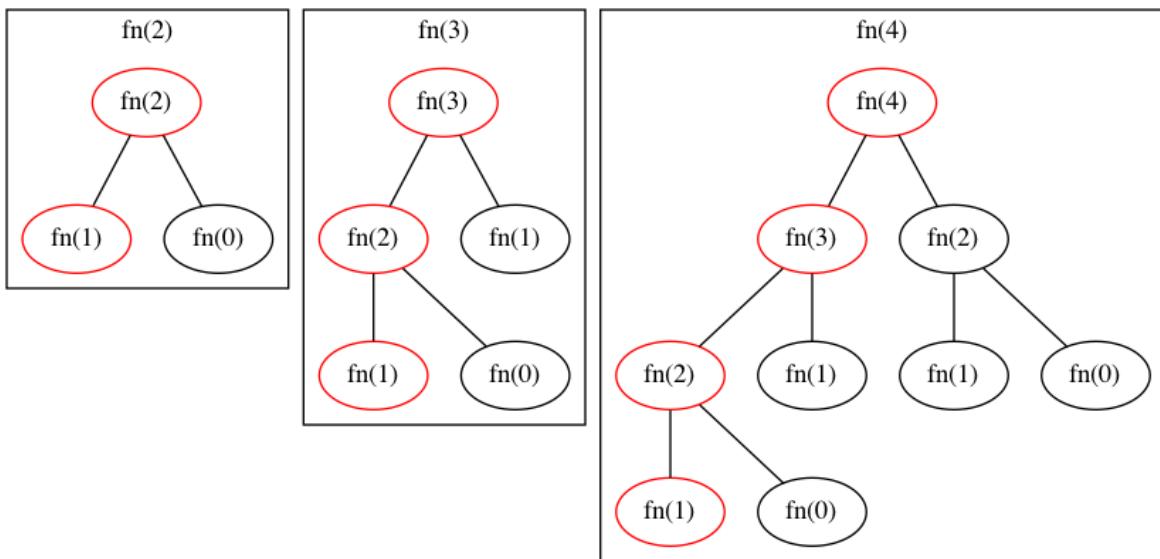
Let's do some examples:

- When you $n = 2$, you have 3 function calls. First `fn(2)` which in turn calls `fn(1)` and `fn(0)`.
- For $n = 3$, you have 5 function calls. First `fn(3)`, which in turn calls `fn(2)` and `fn(1)` and so on.

- For $n = 4$, you have 9 function calls. First $\text{fn}(4)$, which in turn calls $\text{fn}(3)$ and $\text{fn}(2)$ and so on.

Since it's a binary tree, we can sense that every time n increases by one, we would have to perform at most the double of operations.

Here's the graphical representation of the 3 examples:



If you take a look at the generated tree calls, the leftmost nodes go down in descending order: $\text{fn}(4)$, $\text{fn}(3)$, $\text{fn}(2)$, $\text{fn}(1)$, which means that the height of the tree (or the number of levels) on the tree will be n .

The total number of calls, in a complete binary tree, is $2^n - 1$. As you can see in $\text{fn}(4)$, the tree is not complete. The last level will only have two nodes, $\text{fn}(1)$ and $\text{fn}(0)$, while a complete tree would have 8 nodes. But still, we can say the runtime would be exponential $O(2^n)$. It won't get any worst because 2^n is the upper bound.