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 ← 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. Some examples of Not in Place sorting algorithms are: Merge Sort, Quick 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*logn)

Best case: O(n*logn)

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.

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.

Time Complexity:

Worst Case: O(n*n)

Average Case: O(n*logn)

Best case: O(n*logn)

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*n)
- *Average Case:* O(n*logn)
- *Best case:* O(n*logn)

**Auxiliary Space Complexity:** O(1)

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*logn)
- *Average Case:* O(n*logn)
- *Best case:* O(n*logn)

**Auxiliary 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.

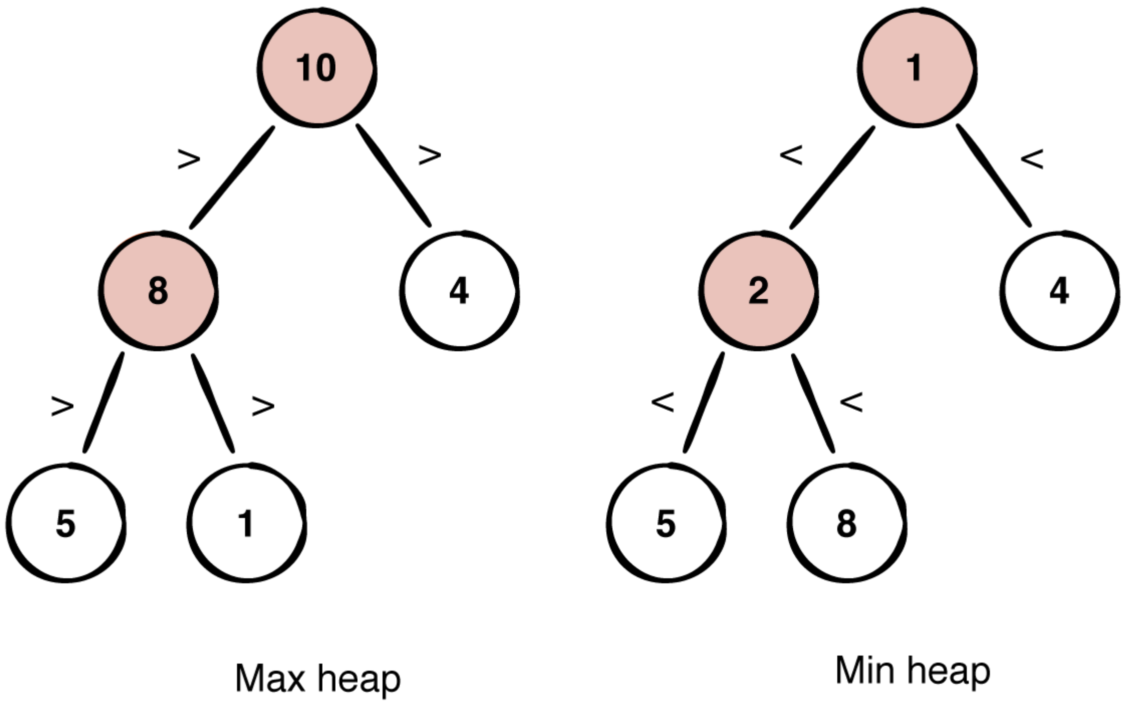
## 8. Heap Sort

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.

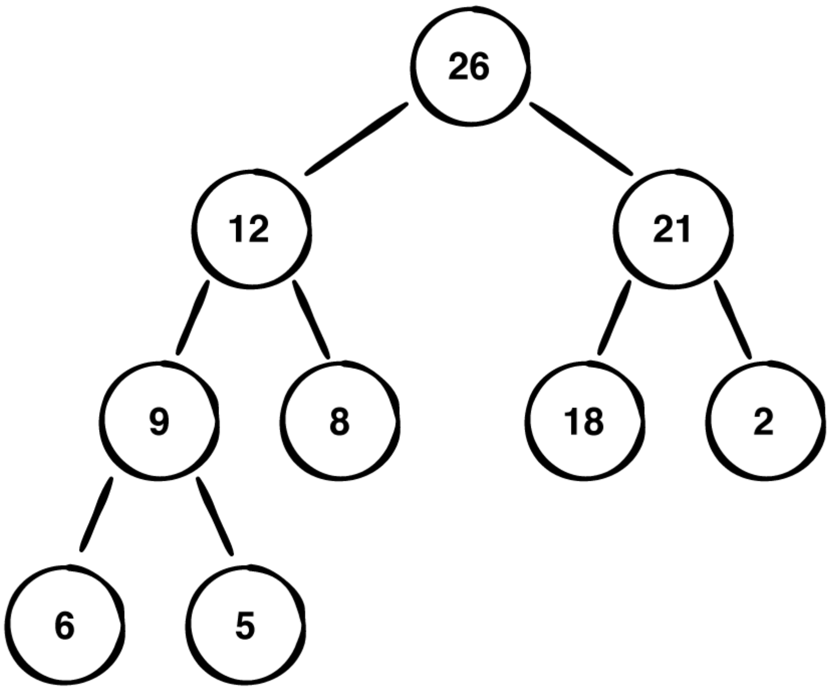
The diagram below shows a heap with parent node values underlined:



## Example

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

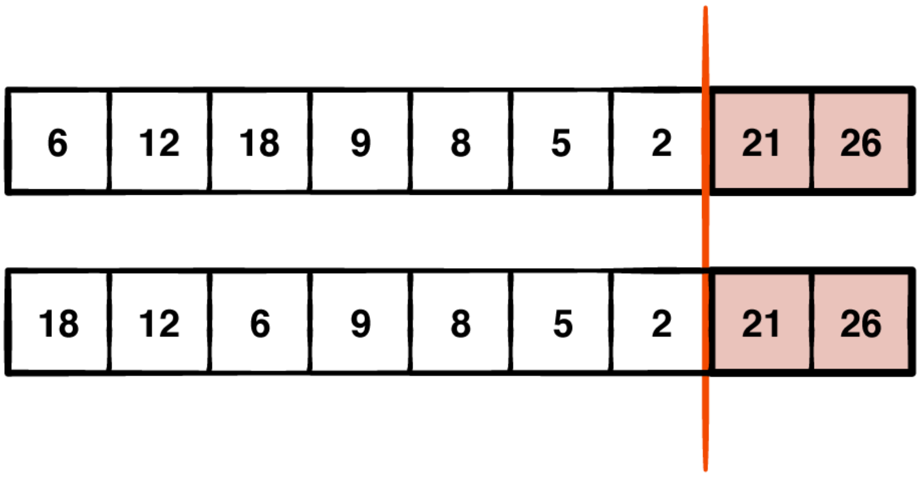
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 6 | 12 | 2 | 26 | 8 | 18 | 21 | 9 | 5 |

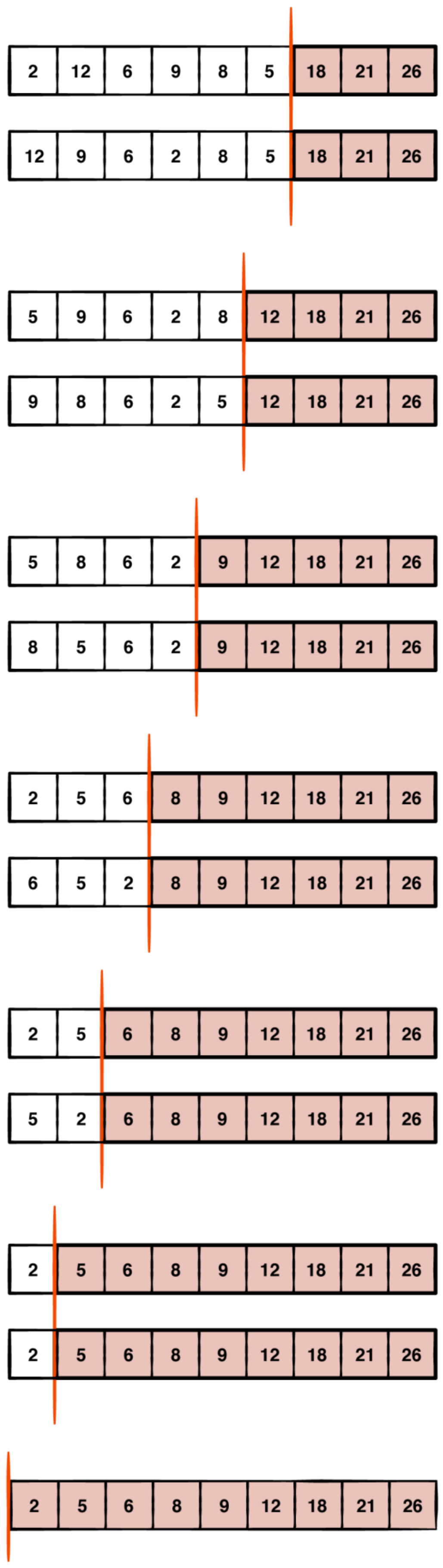


| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 26 | 12 | 21 | 9 | 8 | 18 | 2 | 6 | 5 |

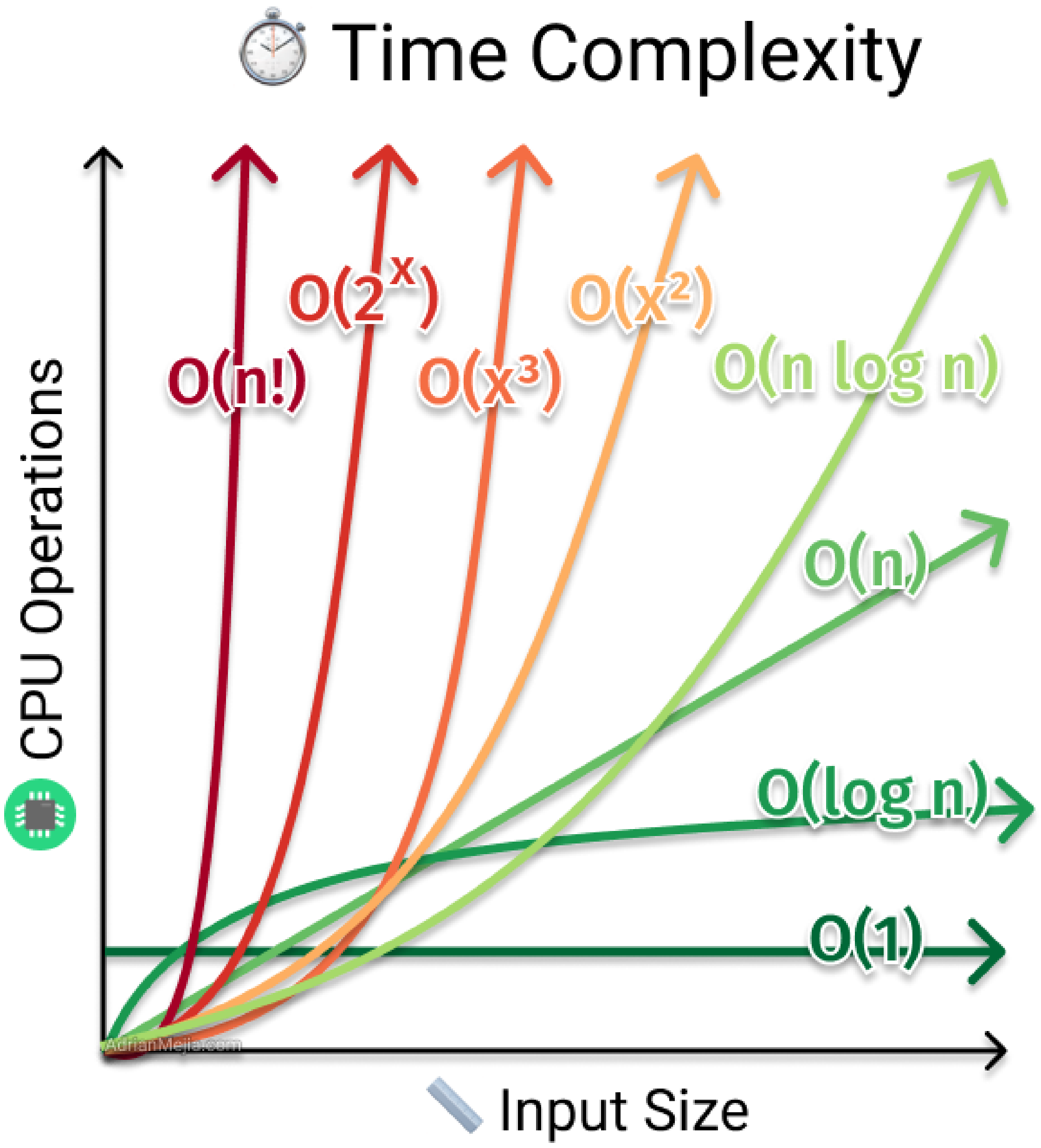
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 12 | 21 | 9 | 8 | 18 | 2 | 6 | 26 |

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


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


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)`.

```
statement1;
statement2;
```

```
...
statementN;
```

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

```
total = 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.

```
T(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.

```
function squareSum(a, b, c) {
  const sa = a * a;
  const sb = b * b;
  const sc = c * c;
  const sum = sa + sb + sc;
  return sum;
}
```

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.

```
if (isValid) {
  statement1;
  statement2;
} else {
  statement3;
}
```

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

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

Example:

```
if (isValid) {
  array.sort();
  return true;
} else {
  return false;
}
```

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:

```
O([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.

```
for (let i = 0; i < array.length; i++) {
  statement1;
  statement2;
}
```

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

```
T(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 n => 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.

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

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):

```
function fn1(array, target, low = 0, high = array.length - 1) {
  let mid;
  while ( low <= high ) {
    mid = ( low + high ) / 2;
    if ( target < array[mid] )
      high = mid - 1;
    else if ( target > array[mid] )
      low = mid + 1;
    else break;
  }
  return mid;
}
```

This function divides the array by its `mid`dle 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 we the while loop will execute 3 times because `log₂(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.

```
for (let i = 0; i < n; i++) {
  statement1;

  for (let j = 0; j < m; j++) {
    statement2;
    statement3;
  }
}
```

For this case, you would have something like this:

```
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:

```
for (let i = 0; i < n; i++) {
  fn1();
  for (let j = 0; j < n; j++) {
    fn2();
    for (let k = 0; k < n; k++) {
      fn3();
    }
  }
}
```

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:

```
function fn(n) {
  if (n < 0) return 0;
  if (n < 2) return n;

  return fn(n - 1) + fn(n - 2);
}
```

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

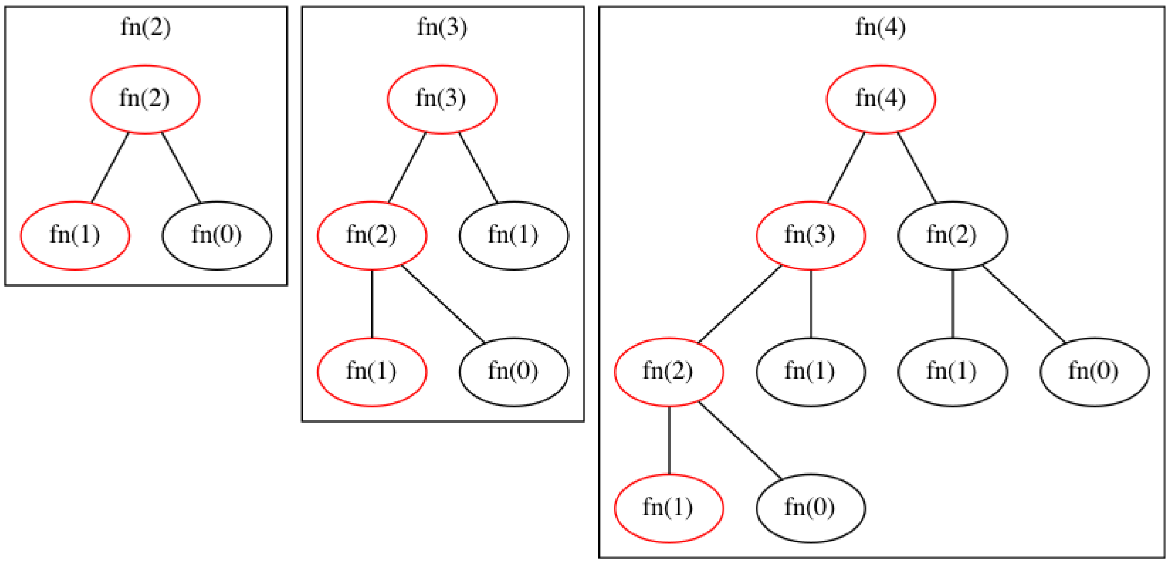
Let's do some examples:

- 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 `fn(4)`, which in turn calls `fn(3)` and `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: `fn(4)`, `fn(3)`, `fn(2)`, `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 `fn(4)`, the tree is not complete. The last level will only have two nodes, `fn(1)` and `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.