Analysis of Algorithms

Analysis of Algorithms

- Analysis involves measuring the performance of an algorithm in terms of
  - Time Complexity

    The amount of time needed by an algorithm to execute.
  - Space complexity

    Amount of memory needed by an algorithm to execute.

Time Complexity

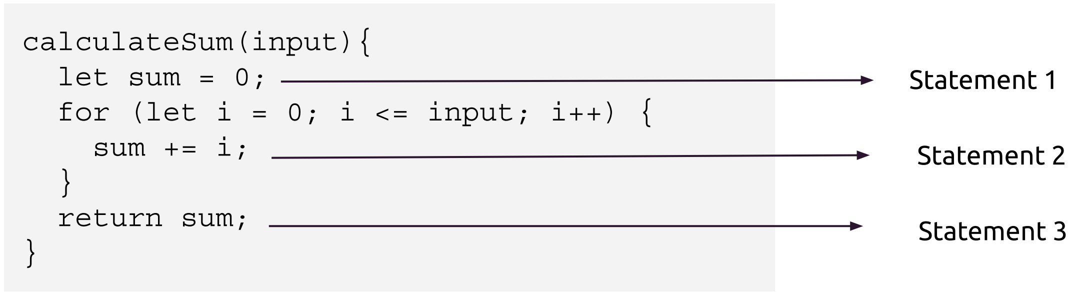
An algorithm's time complexity specifies how long it will take to execute an algorithm as a function of its input size.

Time Complexity

- Since the actual time required to execute an algorithm depends on the details of the program implementing the algorithm and the speed and other characteristics of the machine executing it, it is in general impossible to make an estimation in actual physical time.
- However it is possible to measure the length of the computation in other ways, say by the number of operations performed.

Why is time complexity a function of its input size?

- To perfectly grasp the concept of "as a function of input size," imagine you have an algorithm that computes the sum of numbers based on your input.
  If your input is 4, it will add 1+2+3+4 to output 10; if your input is 5, it will output 15 (meaning 1+2+3+4+5).



```
calculateSum(input){
  let sum = 0;
  for (let i = 0; i <= input; i++) {
    sum += i;
  }
  return sum;
}
```

Looking at the function above, we only have three statements.
Still, because there is a loop, the second statement will be executed based on the input size, so if the input is four, the second statement (statement 2) will be executed four times, meaning the entire algorithm will run six (4 + 2) times.
In plain terms, the algorithm will run **input + 2** times, where input can be any number.
This shows that **it's expressed in terms of the input. In other words, it is a function of the input size**.

Time Complexity Examples

- The following loop performs the statement x := x + 1 exactly n times.

```
for i := 1 to n do
   x := x + 1
```

- The following double loop performs it n2 times:

```
for i := 1 to n do
 for j := 1 to n do
   x := x + 1
```

- The following one performs it $1 + 2 + 3 + \cdots + n = n(n + 1)/2$ times:

```
for i := 1 to n do
  for j := 1 to i do
    x := x + 1
```

Time Complexity

- Since the time that takes to execute an algorithm usually depends on the input, its complexity must be expressed as a function of the input, or more generally as a function of the size of the input.

- Since the execution time may be different for inputs of the same size, we define the following kinds of times:

  - Best Case Time
    Minimum time needed to execute the algorithm among all inputs of a given size n.

  - Worst Case Time
    Maximum time needed to execute the algorithm among all inputs of a given size n.

  - Average Case Time
    Average time needed to execute the algorithm among all inputs of a given size n.

Time Complexity Example

- For instance, assume that we have a list of n objects one of which is colored red and the others are colored blue, and we want to find the one that is colored red by examining the objects one by one.
- We measure time by the number of objects examined.
- In this problem the minimum time needed to find the red object would be 1 (in the lucky event that the first object examined turned out to be the red one).
- The maximum time would be n (if the red object turns out to be the last one).
- The average time is the average of all possible times: 1, 2, 3, . . . , n, which is (1+2+3+· · ·+n)/n = (n+1)/2.

So in this example the best-case time is 1, the worst-case time is n and the average-case time is (n + 1)/2.

Bubble Sort

Since bubble sort is just a double loop its inner loop is executed

$(n - 1) + (n - 2) + \cdot\cdot\cdot + 1 = n(n - 1)/2$ times,

So it requires $n (n - 1)/2$ comparisons and possible swap operations. Hence its execution time is O(n2).

Quick Sort

Let’s look at the space and time complexity of quicksort in the best, average, and worst case scenarios. In general, the time consumed by QuickSort can be written as follows.

T(n) = T(k) + T(n-k-1) + O(n)

Here, T(k) and T(n-k-1)refer to two recursive calls, while the last term O(n) refers to the partitioning process. The number of items less than pivot is denoted by k.

Quicksort’s average case time complexity is **O(n*logn)** and worst case time complexity is **O(n^2).**

Space Complexity

The space complexity of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of characteristics of the input. It is the memory required by an algorithm until it executes completely.

Amount of memory required during program execution as a function of input size.

Memory Usage while Execution

An algorithm uses memory space for three reasons

1. Instruction space
   Amount of memory needed to store the instructions of the algorithm.

2. Environmental Stack
   The memory used to store the information/data needed to resume the execution of algorithm after suspension.

3. Data Space
   Amount of space used by the variables and constants.

Instruction Space

- Amount of memory needed to store the executable instructions of the algorithms, generally it is the compiled version of the program.

Environmental Stack

- An algorithm may call another algorithm as part of the execution, in this case when the control is transferred to another algorithm the environmental stack is used to store the information/data needed by current algorithm.
- At later stage when the called algorithm is finished and the calling algorithm has to continue the execution from the point where it left, the information/data from environmental stack would be loaded back.

Example:

If a function A() calls function B() inside it, then all the variables of the function A() will get stored on the system stack temporarily, while the function B() is called and executed inside the function A().

Data Space

- Amount of space used by the variables and constants.
- This includes the space needed to store input and space needed to store the results to be returned.

Calculating Space Complexity

An algorithm's space can be categorized into 2 parts:

1. Fixed Part
   It is independent of the characteristics of input and output.
   It includes instruction(code) space, space for simple variables, fixed-size component variables and constants.

2. Variable Part
   It depends on instance characteristics. It consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables, and the recursion stack space.

Sometimes, *Auxiliary Space* is confused with Space Complexity. The Auxiliary Space is the extra space or the temporary space used by the algorithm during its execution.

Calculating Space Complexity

Space Complexity = Auxiliary Space + Input space

Thus, space requirement S(M) of any algorithm M is: **S(M) = c + Sm** (Instance characteristics), where c is constant.
While analyzing space complexity, we primarily concentrate on estimating Sm.

Calculating Space Complexity

Consider the following algorithm:

```
public int sum(int a, int b) {
    return a + b;
}
```

In this particular method, three variables are used and allocated in memory:

1. The first int argument, a
2. The second int argument, b
3. The returned sum result which is also an int

In Java, a single integer variable occupies 4 bytes of memory. In this example, we have three integer variables. Therefore, this algorithm always takes 12 bytes of memory to complete (3*4 bytes).

We can clearly see that the space complexity is constant, so, it can be expressed in big-O notation as O(1).

Calculating Space Complexity

Consider the following algorithm:

```
public int sumArray(int[] array) {
    int size = array.length;
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += array[i];
    }
    return sum;
  }
```

Let's list all variables present in the above code:

1. Array – the function's only argument – the space taken by the array is equal to 4n bytes where n is the length of the array
2. The int variable, size
3. The int variable, sum
4. The int iterator, i

The total space needed for this algorithm to complete is **4n + 4 + 4 + 4 (bytes)**.

The highest **order** is of **n** in this equation. Thus, the space complexity of that code snippet is **O(n)**.

When the program consists of loops (In case of Iterative algorithms), it will have linear space complexity or O(n).

Calculating Space Complexity

Consider the following algorithm:

```
factorial(n){
    int res = 1;
    for (int i = 1; i <= n; i++) {
      res *= i;
    }
    return res;
}
```

Let’s list all variables present in the above code:

1. res is an integer variable which will take the 4 bytes of space.
2. n is an integer variable which takes 4 bytes
3. i is an iterator variable which will also take 4 bytes of space.
4. Now function call, initialising for loop and return function these all comes under the auxiliary space and lets assume these all will take combinely 4 bytes of space.

Hence, Total Space Complexity = 4*4 = 16 bytes

As there is no variable which just constant value(16) is there so it means that this algorithm will take constant space that is **O(1)**.

Calculating Space Complexity

Algorithm to find the factorial of the number using recursive method.

```
factorial(N){
    if(N<=1) {
      return 1;
    }
    return (N*factorial(N-1));
}
```

Let’s list all the variables used in the algorithm:

1. "N" is an integer variable which stores the value for which we have to find the factorial, so no matter what value will, it will just take "**4 bytes**" of space.
2. Now function call, "if" condition, "else" condition, and return function these all comes under the auxiliary space and lets assume these all will take combinely “4 bytes” of space but the matter of fact here is that here we are calling that function recursively "N" times so here the complexity of auxiliary space will be "**4*N bytes**" where N is the number of which factorial have to be found.

Hence, **Total Space Complexity = (4 + 4*N)** bytes But these 4 bytes are constant so we will not consider it and after removing all the constants(4 from 4*N) we can finally say that this algo have a complexity of **"O(N)".**

Space complexity of few common algorithm

Algorithm	Space Complexity
Bubble Sort	**O(1)** As it is in place sorting algorithm and requires the constant space for variables like flag, temp etc.
Insertion Sort	**O(1)** As it is in place sorting algorithm and requires the constant space for variables like flag, temp etc.
Selection Sort	**O(1)** As it uses constant space for 2 variables to swap elements and 1 for keep pointing on smallest element in unsorted array.
Heap Sort	**O(1)** As in this no extra array is needed because data is rearranged in original array so as to make it sorted.
Quick Sort	**O(n)** As each recursive call will create a stack frame which takes up space, and the number of stack frame is dependent on input size n.
Merge Sort	**O(n)** As in each recursive call 2 arrays are created