Programming Challenges I

CSCI 485/4930

Time Complexity Notes

Intro

Sometimes, there is more than one way to solve a problem. We need to learn how to compare their performances and choose the best one.

The exact time needed for a program to finish executing depends on lots of things like hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing an algorithm. We will only consider the execution time of the algorithm with respect to the size of input fed to the program. This measurement is called “Time Complexity” or “Order of Growth”.

What is Time Complexity/Order of Growth?

Time Complexity/Order of Growth defines the estimated time taken by a program with respect to the size of the input. Since this function is generally difficult to compute exactly, and the running time is usually not critical for small input, one focuses commonly on the behavior of the complexity when the input size increases; that is, on the asymptotic behavior of the complexity. Therefore, the time complexity is commonly expressed using big O notation, typically O(n), O(n log n), etc.

When solving problems in competitions we always consider the worst-case time complexity.

In this document, we will go through some of the basic and most common time complexities.

Constant Time Complexity O(1)

The code that runs in a fixed amount of time or has fixed number of steps of execution no matter what is the size of input has constant time complexity. For instance, let’s try and derive a Time Complexity for the following code:

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

If we call this function by `my_sum(2, 5)` it will return 7 in 1 step. That single step of computation is summing `a` and `b`. No matter how large is the size of input (i.e. `a` and `b`), it will always return the sum in 1 step.

So, the Time Complexity of the above code is O(1) or Constant Time Complexity.

Let's examine the following code:

```
int multiply_by_1000(int a){
  int res = 0;
  for(int i = 0; i < 1000; i++)
    res += a;
  return res;
}
```

Although the loop will be executed 1000 times, the time complexity of the above code is also O(1), because it runs in a fixed number of steps no matter what the input is (the code will take the same amount of time whether the `a` is 1 or 1000000).

If we compared the above stupid code with the following one:

```
int multiply_by_1000(int a){
  return a*1000;
}
```

We will notice that the second code runs approximately 1000 times faster than the first one, but they both have the time complexity of O(1).

This easy optimization is called "Constant Optimization" (We did not change the complexity of the code but we decreased the constant factor), be careful with the constant factor of your solutions, large constant factor may cause a TLE verdict if the time limit allowed was tight.

Interesting Fact:
An optimizing compiler is a compiler that minimizes the time and/or memory needed to execute the program.
If you run the first code in an optimizing compiler, the optimizer could replace it with the second one to increase the code's efficiency.

Linear Time Complexity O(N)

The code whose Time Complexity or Order of Growth increases linearly as the size of the input is increased has Linear Time Complexity. For instance, let's see this code which returns the sum of an array.

```
int my_array_sum(int arr[], int n){
  int res = 0;
  for(int i=0; i<n; i++)
    res += arr[i];
  return res;
}
```

Here, we are providing an array to the function. If we pass an array of size 10, the number of steps would be:

- 1 step for initializing result.
- The statement inside loop is executed 10 times.
- 1 step for return statement.

In total 1 + 10 + 1 = 12 steps. Now what if the size of the array is 1000000? Then the total steps would be 1 + 1000000 + 1 = 1000002. Do the first and last 1s matter?

For array of size N, Time Complexity would be 1 + N + 1 = (N + 2). We can safely neglect the additional 2 and say that the overall Time Complexity is O(N) (because we saw that as N becomes large the steps with constant time have a negligible effect on running time of the code).

Logarithmic Time Complexity O(log N)

When the size of input is N but the number of steps to execute the code is log(N), such a code is said to be executing in Logarithmic Time. This definition is quite vague but if we take an example, it will be quite clear.

Let's say we have a very large number which is a power of 2 (i.e. we have 2^x). We want to find x. For example: 64 = 2^6. So x is 6.

```
int power_of_2(int n){
  int res = 0;
  while(n){
    res ++;
    n /= 2;
  }
  return res;
}
```

If I call `power_of_2(16)`, the while loop will run 4 times, because we keep dividing a by 2. In first iteration, a will become 8, in second iteration a=4, in third iteration a=2 and in forth iteration a=1. After this iteration the loop will break.

There are 2 instructions inside the loop, so total number of steps would be:
1 instruction for x = 0
2*4 instructions for 2 statements inside the loop
1 instruction for return statement
In total 2*4 + 2.

If we increase the size of a to 1024. It will take 2*10 + 2 steps. Do you notice a pattern here?

log(16) = 4 (Considering log of base 2), log(1024) = 10.

When we keep dividing the size of input N by some value, say b. Then the Time Complexity turns out to be log(N) to the base b.

So, overall time complexity of the above code is, 2*log(N) + 2. For very large values of N, the multiplication by 2 and the additional 2 can be neglected. Hence, it is log(N) to the base 2.

Log-Linear Time Complexity O(N log N)

Many sorting algorithms complexity is O(N log N) (e.g. Merge sort).

The time complexity of the built-in sort function in C++ is O(N log N)

Polynomial Time Complexity $O(N^X)$

When the computation time increases as a function of N raised to some power, N being the size of the input. Such a code has Polynomial Time Complexity.

For example, let's say we have an array of size N and we have nested loops on that array.

```
for(int i=0; i<n; i++){
  for(int j=0; j<n; j++){
    // some processing
  }
}
```

For each iteration in the outer loop, n iterations are done in the inner loop, so, the processing part is executed N*N times i.e. N^2 times. Such a code has $O(N^2)$ time complexity.

```
for(int i=0; i<n; i++){
  for(int j=0; j<n; j++){
    for(int k=0; k<n; k++){
      // some processing
    }
  }
}
```

The above code has $O(N^3)$ time complexity.

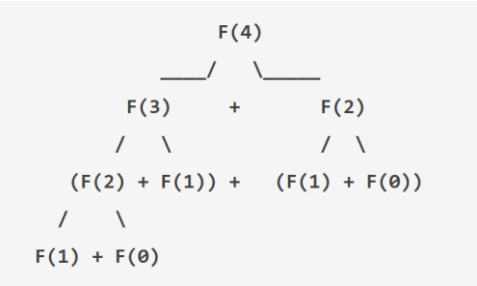
Exponential Time Complexity $O(X^N)$

When the computation time of a code increases as function of X^N, N being the size of the input, such a code has Exponential Time Complexity.

For example, following recursive code to find Nth Fibonacci number has Time Complexity of $O(2^N)$

```
int fib(int n){
  if(n <= 1)
    return 1;
  return fib(n - 1) + fib(n - 2);
}
```

In the above code, for every call to F() we make two more calls to F() unless we reached the base case. So if I call F(4), the tree of calls to F() would look like:



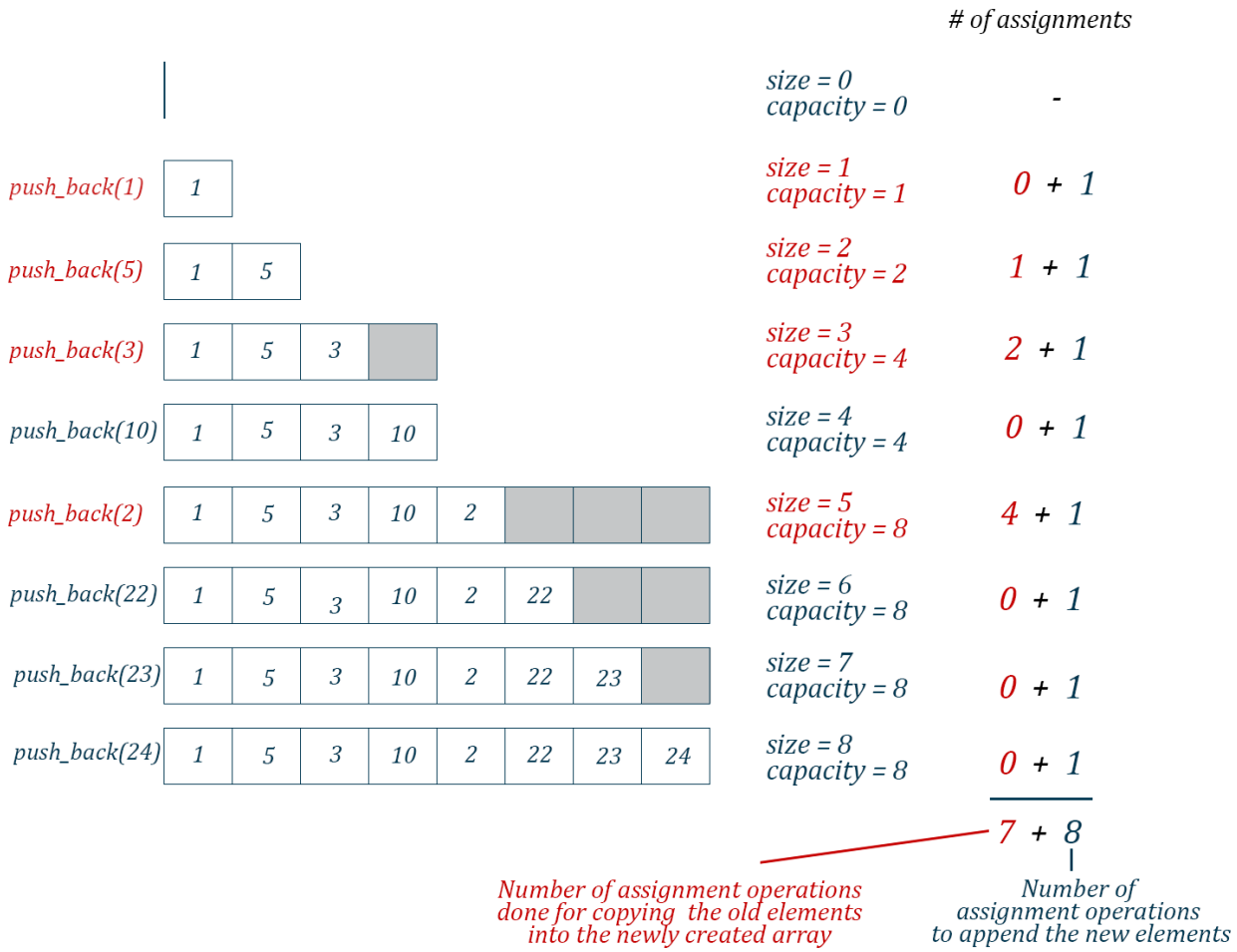
This tree keeps growing exponentially as we increase N. Hence the Time Complexity O(2^N)

Amortized Analysis

When we write an algorithm where an occasional operation is very slow, but most of the other operations are faster, it may be helpful to consider amortized analysis.

Amortized analysis is a worst-case analysis of a sequence of operations — to obtain a tighter bound on the overall or average cost per operation in the sequence than that obtained by separately analyzing each operation in the sequence.

Let us consider an example of the dynamically-resizable array (vector) push_backs.
The following figure illustrates the insertions (push_backs) of 8 elements.



The idea is to increase the size of the array whenever it becomes full and we need to append an extra element.
So, each time there is no space for an extra element in the current array the following 3 steps are done:

1- Allocate memory for a larger array of size, typically twice the old array.
2- Copy the contents of the old array to the new one.
3- Free the old array.

If the array has space available, we simply insert the element into the free space.

Let's consider a sequence of n insertions, the worst-case time to execute one insertion is O(n). Therefore, ~~the worst-case time for n insertions is n * O(n) = $O(n^2)$~~

This conclusion is **WRONG,** in fact, the worst-case cost for n insertions is only O(n). Let's see why:

We notice from the above example that the number of times we resized the array is Log(n) = Log(8) = 3 times (in addition to the creation of the first array to hold the first element, this will not require any copying operations, so we will neglect it).

The first resizing needs 2^0 copy operations.
The second resizing needs 2^1 copy operations.
The third resizing needs 2^2 copy operations.

In general, the total number of copying operations for N insertions is $\sum_{i=0}^{\log_2(N)-1} 2^i$ this is equivalent to $2^{\log_2 N} - 1$ = $\boldsymbol{N-1}$

And the total number of the assignment operations to add the new elements is $\boldsymbol{N}$.

So, the total number of operations is $\boldsymbol{2N-1.}$

Thus, the complexity of the N insertions is O(N), and the average cost of each insertion is O(N)/N = O(1).

The single push_back operation out of N push_backs has the amortized complexity of O(1)

To generalize the above rule to hold on any N (not just the power of twos)

The number of times we will resize the array is $\lfloor \log_2(n-1) \rfloor$

$$n + \sum_{i=0}^{\lfloor \log_2(n-1) \rfloor} 2^i$$

Notes

- There are multiple notations to measure the time complexity of your algorithm, in this course we will stick to the tight bound Big O measurement. The rest of measurements are out of scope (If you're curious read this).
- Modern computers are quite fast and can process up to ≈ 100M (or 10^8; 1M = 1,000,000) operations in a few seconds. You can use this information to determine if your algorithm will run in time. For example, if the maximum input size n is 100K (or 10^5; 1K = 1,000), and your current algorithm has a time complexity of O(n2), common sense (or your calculator) will inform you that $(100K)^2$ or 10^{10} is a very large number that indicates that your algorithm will require (on the order of) hundreds of seconds to run.

Resources

1- Rookie's Lab - How to compute Time Complexity or Order of Growth of any program
2- Geeks for Geeks: Amortized Analysis Introduction
3- MIT – OCW 6.046J/18.40J Lecture 13 slides
4- Competitive Programming - 3rd edition

Further Reading

1- Mostafa Saad – Complexity of Algorithms Playlist
2- Competitive Programming - 3rd edition - Section 1.2.3