Ad-Hoc Problems and Computational Complexity

CS Team Captains

September 24th, 2020

1 Introduction

This week's lecture notes will serve as an introduction to the types of problems you'll see in Competitive Programming competitions starting with **Ad-Hoc Problems**. You'll also learn how to analyze your algorithms' efficiency by looking at their **computational complexity** (sometimes referred to time complexity). After going through this week's notes and putting what you learn into practice, you should be ready to zoom through USACO Bronze contests.

2 Introduction To Computational Complexity

Calculating the computational complexity of your algorithms gives you an idea of how long your solution to a particular problem will take. This is incredibly useful in Competitive Programming because it allows you to ensure that your solution will run in time for all test cases before you write a single line of code. So, you should always attempt to find the complexity of your algorithm before actually implementing it so that your time isn't wasted writing code that isn't efficient. Additionally, knowing the time complexity that's required to solve a problem in a contest can often clue you in on what algorithms might need to be used.

2.1 Big-O Notation

Big-O notation is one of the ways in which you can represent the Computational Complexity of an algorithm. It shows how fast or slow a program will be as a function of the size of its input, which is very useful. You may have seen Big-O notation in the past, it looks something like this: $O(n^2)$, where n is the size of the input.
Here's a more formal way of talking about what is called **Asymptotic Notation**. We will represent the time function $T(n)$ using big-O notation to express an algorithm's run time complexity. For example, the following statement

$$T(n) = O(n^2)$$

says that an algorithm has a quadratic time complexity.

Let's take a quick look at a program and then attempt to figure out its time complexity.

```
#include<iostream>

using namespace std;

int main(){
    int n;
    cin >> n;

    for(int i=0; i<n; i++){
        cout << i << endl;
    }
    return 0
}
```

The code pictured above is very simple. It takes in some input n, then prints every number from 0 to $n-1$. You should notice that the for loop's body will execute n times. The program's running time will increase linearly with n so we can say that its time complexity is $O(n)$. In other words, the program runs in linear time.

2.2 Finding Time Complexity

Here are a few short steps that you can follow to quickly find the worst case Time Complexity of an algorithm:

1. Identify the input (n).
2. Find the maximum number of opeprations the algorithm performs that are depndent on n.
3. Find the highest order term.
4. Remove the constant factors.

2.3 The Definition of Big-O Notation

Here's a formal definition of Big-O notation. For any monotonic functions $f(n)$ and $g(n)$, we say that $f(n) = O(g(n))$ when there exist constants $c > 0$ and $n_0 > 0$ such that

$$f(n) \leq c \cdot g(n) \; \forall \, n \geq n_0$$

Basically, this means that function $f(n)$ does not grow faster than $g(n)$ is an **upper bound** for $f(n)$, for all sufficiently large $n \to \infty$. For those of you who are visual learners, here's a graphic representation of the $f(n) = O(g(n))$ relation, courtesy of Carnegie Mellon University.

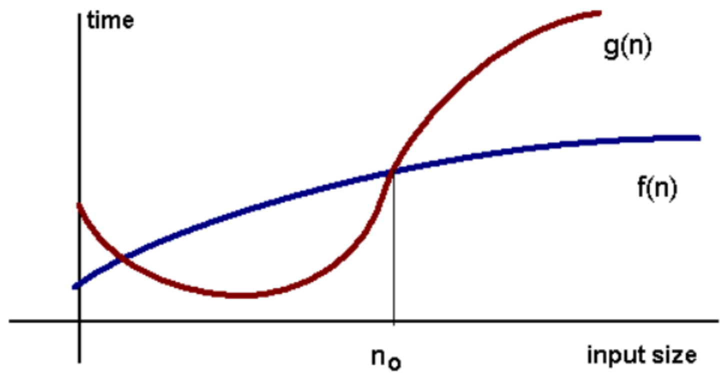

Figure 1: Big O Graphical Representation

Ex. 1 — Prove that $n^2 + 2n + 1 = O(n^2)$.

Answer (Ex. 1) — To approach this, we must first find some c and n_0 such that $n^2 + 2n + 1 \leq c \cdot n^2$. Let $n_0 = 1$, then for $n \geq 1$

$$1 + 2n + n^2 \leq n + 2n + n^2 \leq n^2 + 2n^2 + n^2 = 4n^2$$

$\therefore c = 4$ and $n^2 + 2 \cdot n + 1 = O(n^2)$ $\square$

Ex. 2 — Is $n^2 + n + 1 = O(n^2)$?

Answer (Ex. 2) — To prove this, we need to find a constant c such that $cn^2 \geq n^2 + n + 1$. Let $c = 2$. Now we need to find a constant n_0 such that for all $n \geq n_0$, $2n^2 \geq n^2 + n + 1$. Let n_0 be 10. We will proceed with an inductive argument. To make our life simpler, let $f(n) = 2n^2$, and $g(n) = n^2 + n + 1$. When $n = 10$, $f(n) = 200$ and $g(n) = 111$, so $f(n_0) > g(n_0)$. Now let's assume that our statement is true for all values between 10 and n for some n. We already know that this is true for $n = 10$. Let's look at $n + 1$:

$$\begin{aligned} f(n+1) &= 2(n+1)^2 \\ &= 2n^2 + 4n + 2 \\ &= f(n) + 4n + 2 \end{aligned}$$

$$\begin{aligned}
g(n+1) &= (n+1)^2 + (n+1) + 1 \\
&= n^2 + 2n + 1 + n + 1 + 1 \\
&= n^2 + 3n + 3 \\
&= (n^2 + n + 1) + 2n + 2 \\
&= g(n) + 2n + 2
\end{aligned}$$

From our inductive hypothesis, we know that $f(n) \geq g(n)$, thus:

$$f(n) + 4n + 2 \geq g(n) + 4n + 2$$

Since $n \geq 10$, $4n + 2 > 2n + 2$, and therefore:

$$\begin{aligned}
f(n) + 4n + 2 &> g(n) + 2n + 2 \\
f(n+1) &> g(n+1)
\end{aligned}$$

Therefore, for all $n \geq 10$, $2n^2 \geq n^2+n+1$, and therefore $n^2+n+1 = O(n^2)$ $\square$
Try the next one out yourself.

Ex. 3 — Generalizing, is $a \cdot n \cdot n + b + d = O(n \cdot n)$ for $a, b, d > 1$ and $b > d$?

2.4 Common Time Complexities

2.4.1 Constant Time: O(1)

An algorithm runs in constant time if it requires the same amount of time regardless of the input size. Examples:

- array: accessing any element
- fixed-size stack: push and pop methods
- fixed-size queue: enqueue and dequeue methods

2.4.2 Linear Time: O(n)

An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases. Examples:

- array: linear search, traversing, find minimum
- ArrayList: contains method
- queue: contains method

2.4.3 Logarithmic Time: O(log n)

An algorithm is said to run in logarithmic if its time execution is proportional to the logarithm of the input size. Examples:

- Binary Search: The loops in binary search will only loop $logN$ times, so those loops are of size $O(logN)$

2.4.4 Quadratic Time: O(n^2)

An algorithm is said to run in logarithmic time if its time execution is proportional to the square of the input size. Examples:

- bubble sort
- selection sort
- insertion sort

2.5 Big Omega Notation

Big Omega is the notation for the lower bound of an algorithm's time complexity. A capital Ω is used to represent this. We say that $f(n) = \Omega(g(n))$ when there exist constant c that $f(n) \geq c \cdot g(n)$ for all sufficiently large n.

2.6 Big Theta Notation

We say that $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n)$ and $f(n) = \Omega(g(n))$. Here are some examples:

- $2n = \Theta(n)$
- $n^2 + 2n + 1 = \Theta(n^2)$

2.7 Predicting Complexity Required Based on the Problem

You can figure out what time complexity your algorithm will need to be based on the max size of the input, N, as given by the problem.

N	Complexity
10^9	$O(1)$
10^6	$O(N)$
10^5	$O(N)$ or $O(NlogN)$
10^4	$O(N^2)$
10^3	$O(N^2$
10^2	$O(N^2)$

You can also figure out if a solution will be fast enough without looking at this table by substituting the value of N into whatever your Big-O expression should be and then seeing if that result is much less than 10^9.

3 Ad-Hoc Problems

Ad-Hoc problems are the simplest kind of problems and often populate the USACO's Bronze Division. You can also find them at the beginning of Codeforces and At-coder contests. Problem statements for Ad-Hoc problems usually involve a set of instructions that you have to implement in code. As a result they're much more straightforward than other Competitive Programming problems and you most likely won't have to worry too much about the time complexity of your algorithm. Some examples include string manipulation and solving a puzzle.

4 The Problem-Solving Process

The following is by no means a formula for solving contest problems: problem-solving is a personal process, but it's important to take these things into consideration. The process for solving Competitive Programming problems also depends on the situation of the contest. For example, in Codeforces contests you don't have a lot of time so you'll need to solve questions quickly. On the other hand, in the USACO you have a large amount of time which lets you take your time with each problem.

1. **READ THE PROBLEM!**
   - (a) **Read the problem carefully**. USACO problems start with a fun blurb, and then go into a technical description of the problem.
   - (b) **Make sure you understand the problem**. This means playing around with the example cases, understanding the example output.
   - (c) **Evaluate the constraints**. What data types will you need to use (int or long long)? What time and space complexities can you afford? What is the size of your search space?
   - (d) **Generate your own test cases**. Try and break your own solution by coming up with edge cases. Can things be zero? Can the be really large? Does your solution work for every type of input? What might you have missed?
2. **Design an Algorithm**.
   - (a) **Use past experience.** Does this problem look similar to past problems? How does it differ? Classify the problem!
   - (b) Find bijections and use abstractions. Express the problem in pure CS terms, focusing on the general problem rather than the specifics you've been given.
   - (c) **Identify relevant data structures.** Efficient algorithms rely on efficient methods for storing and accessing data.

(d) **Try finding a pattern for smaller cases.** If applying the four main paradigms (brute force, dynamic programming, divide and conquer and greedy) get you nowhere, keep trying. If you don't know what all of those are yet, don't worry! You'll learn a lot about them in the coming weeks.

(e) **Write a pseudocode solution.** This will keep you aware of edge cases and ensure a bug-free solution. It's also easier to find the time complexity of your solution if you've written it out in pseudocode first.

3. **Implement Your Solution (Code it)** Here are some helpful tips from the USACO training page.

   (a) **Comment, then code.** This might help you organize your solution.

   (b) **Write your solution.** Write out the code!

   (c) **Read over your code!** This will ideally be happening while you're writing. Make sure you're not naming your variables wrong or doing the wrong arithmetic. The best type of debugging is to immediately fix your mistakes.

   (d) **Test your code locally.** Run your code on your computer and then pass it testcases.

   (e) **Submit** Submit your solution and hope for the best.

5 Problem Walkthrough: "Balanced Array"

The following problem is taken from Codeforces and the direct link to the problem is: `https://codeforces.com/problemset/problem/1343/B`.

5.1 Problem Statement

You are given a positive integer n, it is guaranteed that n is even.
You want to construct the array a of length n such that:

- The first $\frac{n}{2}$ elements of a are even;
- the second $\frac{n}{2}$ elements of a are odd;
- **all elements of a are distinct and positive;**
- the sum of the first half equals to the sum of the second half ($\sum_{i=1}^{\frac{n}{2}} a_i = \sum_{i=\frac{n}{2}+1}^{n} a_i$)

If there are multiple answers, you can print any of them. It is **not guaranteed that the answer exists.** You have to answer t independent test cases.

5.2 Input

The first line of the input contains one integer t ($1 \leq t \leq 10^4$) - the number of test cases. Then t test cases follow.
The only line of the test case contains one integer n ($2 \leq n \leq 2 \cdot 10^5$) - the length of the array. It is guaranteed that n is even.
It is guaranteed that sum of n over all test cases does not exceed $2 \cdot 10^5$.

5.3 Output

For each test case, print the answer - “NO” (without quotes), if there is no suitable answer for the given test case or “YES” in the first line and any suitable array $a_1, a_2, ..., a_n$ ($1 \leq a_i \leq 10^9$) satisfying conditions from the problem statement on the second line.

5.4 Example Input/Output

```
//Input
5
2
4
6
8
10
//Output
NO
YES
2 4 1 5
NO
YES
2 4 6 8 1 3 5 11
NO
```

5.5 Solution

After you read over the problem a few times the next thing you should do is look at the max size of the input as pointed out by the problem. We see that the max input size for each test case is $2 * 10^5$. If you look at the table of common time complexities, you'll see that our solution will need to run in O(n) time. Now let's start thinking about the solution to this problem. A good way to start thinking about this is to see in which cases it is impossible to construct an array. You should notice that if n is not divisible by 4 then you cannot construct the array because the parities of the halves will not match. Otherwise, you are always able to construct an array.

But how do you actually construct the array? Let's try the following method: $[2, 4, 6, ..., n, 1, 3, 5, ..., n-1]$. It doesn't quite work, there's a little bit of it that is off. The sums of the two halves are not equal, the sum of the left half is greater than that of the right half by $\frac{n}{2}$. This makes for an easy fix, because all you have to do is add $n/2$ to the last element in the array. The code implementation for this problem is shown below and follows the exact same logic as this solution does. There's a while loop that contains the solution code that will run until there are no more test cases. Within that while loop we check to find the divisibility of n, and from then on we begin to construct our array.

5.6 Code Solution

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    int t;
    cin >> t;
    while (t--) {
        int n;
        cin >> n;
        n /= 2;
        if (n & 1){
            cout << "NO" << endl;
            continue;
        }
        cout << "YES" << endl;
        for(int i=1; i <= n; i++){
            cout << i * 2 << " ";
        }
        for(int i=1; i<n; i++){
            cout << i*2 - 1 << " ";
        }
        cout << 3*n-1 << endl;
    }
    return 0;
}
```

The time complexity for this solution is O(n) and the solution will easily pass all the test cases with time to spare. To make sure this solution is right, submit it to the codeforces online judge.

6 Conclusion

You should now be equipped with the tools you need to solve Ad-Hoc problems. You've also set the base for future explorations in algorithms by learning about time complexity analysis. We suggest that you participate in some weekly Codeforces contests to start getting a feel of what Competitive Programming is. Additionally, it is of utmost importance that you practice regularly in order to ingrain these skills in your mind.