Arrays and Time Complexity

MHS Competitive Programming Leadership

September 24, 2024

1 Arrays

Ideally you should have learnt enough programming by now to understand and use arrays. We will provide an overview here.

Arrays are an ordered collection of elements. They are 0-indexed, which means to refer to the first element of an array you would write `a[0]`. However, in most problem statements, arrays remain 1-indexed, so the first element of an array would be referred to as a_1. Remember this difference to avoid confusion.

1.1 Basic Operations

You should be familiar with the following operations on arrays that are provided by your language:

- Searching for an element in an array
- Sorting an array
- Reversing an array
- Summing all elements in an array
- Finding the maximum or minimum element in an array

Of course, with the exception of sorting, you can code these by yourself with for-loops. But learning the library functions used for these tasks will save a lot of time.

1.2 C++ Array Pitfalls

Arrays in C++ have a few details not present in other langauges.

A static C++ array is not initialized by default. For example, after performing `int a[5];`, the array `a` is not initialized. Its elements contain garbage memory values. If you would like them to be initialized to 0, you can use `int a[5] = {};`.

Going out of bounds of a C++ array does not produce an error message, so you must be vigilant of your array accesses! You can also consider installing an address sanitizer, which will catch these errors for you.

2 Time Complexity

We need a way to quantify how fast a program is. This is done using time complexity analysis.

2.1 Introductory Example

Time complexity is a measure of how long an algorithm takes to run as a function of the size of its input. Consider this example program in Python:

```
a = list(map(int, input().split())) # Python tip: this is how you input a list of space separated integers!
n = len(a)
sum = 0
for i in range(n):
    sum += a[i]
print(sum)
```

So this program will input a list of space-separated integers, and print their sum. Now someone asks, how fast does this program run?

We can answer this question by identifying *how the number of computational operations increases as the input size increases*. Notice that for each element of the array, the computer must perform a summation operation. Given the length of the array is n, there will be n summation operations. So we could say as the input size increases, the number of operations increases at a linear rate.

This is easy to say for such a simple program, but with more complicated programs, whose operations counts can vary polynomially, logarithmically, exponentially, and even as a combination of these, we need a formal notation to describe their running speed.

2.2 Big-O Notation

Big-O notation is a way to describe the upper bound of how the running time of an algorithm changes with respect to input size. It is written as $\mathcal{O}(f(n))$, where $f(n)$ is a function of the input size n. So in this case, the time complexity of the above program is $\mathcal{O}(n)$ since the number of operations increases linearly with the input size.

There are a few rules for finding Big-O notation:

1. Always consider the worst case scenario- the scenario in which the program has to perform the most operations.
2. Find a formula that describes the number of operations as a function of input size
3. Omit constant factors. For example, if your formula is $2n$ it should become n.
4. Omit lower order terms. For example, if your formula is $n^2 + n$, it should become n^2.
5. Put your formula $f(n)$ into the Big-O notation format, $\mathcal{O}(f(n))$.

It may seem strange to remove constant factors and lower order terms, but the idea of Big-O notation is to characterize the scale at which running time increases. When the input size becomes enourmous, these constant coefficients and lower order terms often become insignificant.

2.3 Big-O Notation in Competitive Programming

So now that you can find the Big-O notation of a program's running time, you can use it to determine if a program will run under the time constraints of a problem.

To do this, simply plug in the maximum input size into the expression of the Big-O notation you have (note this isn't a computationally correct usage of Big-O notation, but it suffices). For example if you have $\mathcal{O}(n\sqrt{n})$, and say the maximum bound on n is 10^5, you would plug it into get $10^5\sqrt{10^5} = 10^5 \cdot 10^{2.5} = 10^{7.5}$. Generally, if the time limit for the problem is 2 seconds (which it usually is), then any number less than around $5 \cdot 10^8$ should run in time. So this program is fast enough!