Algorithmic Complexity and Data Structures I

Julian Tu

May 15, 2016

1 Algorithmic Complexity

1.1 What is it?

We use something called Big O notation to define the *time* or *space* it takes for a function or algorithm to run based on the size of its input.

1.1.1 Some Maths

Let f and g be two functions, where the algorithmic complexity of f in terms of Big O notation and g is as follows:

$$\text{If } f(x) = O(g(x))$$
$$\text{Then } |f(x)| \leq M \cdot |g(x)|$$

This is true for all $x \geq x_0$ and $M \geq M_0$ for some x_0 and M_0.

1.1.2 Less Maths

Note that this implies a few things. First, the whole $x \geq x_0$ means that g need not be larger than f for all x. The only requirement is that at some point in the future, g must be greater than f (for *all* input!) and stay greater.

The second implication comes as a result of $M \geq M_0$. This means that *constant factors* do not come into play when using Big O notation. Thus, $O(2N) = O(1000N) = O(N)$.

The third implication is of a result of the first and the general principle that Big O notation strives to be the *worst case* running for any algorithm. Thus an algorithm that runs in $O(N)$ time can be described as $O(2^N)$, though we can see how this is less useful.

However, Big O notation considers summation to be different to multiplication. If an algorithm runs in $O(N^2 + log\, N)$ time, it can be equally correct to write it as $O(N^2)$: taking only the largest term. This is because as N gets large, the greatest, and eventually only distinguishing, impact on the growth of algorithm is its fastest growing term.

1.2 Examples

1.2.1 Linear Search

```
function search(array, searching_for) {
    forall i = [0, n-1] {
        if array[i] = searching_for {
            return i;
        }
    return -1;
}
```

Runs in $O(N)$.

1.2.2 Dumb Linear Search

```
function search(array, searching_for) {
    found = -1
    forall i = [0, 100] {
        forall i = [0, n-1] {
            if array[i] = searching_for {
                found = i;
            }
        }
    }
    return found;
}
```

Still runs in $O(N)$.

1.2.3 Binary Search

```
function search(array, searching_for) {
    low = 0;
    high = n - 1;
    while low <= high {
        mid = (low + high) / 2;
        if array[mid] == searching_for {
            return true;
        } else if array[mid] < searching_for {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1
}
```

Runs in $O(log\, N)$.

1.2.4 Iterative Fibonacci

```
function fib(n) {
    a = 0;
    b = 1;
    forall i = [1, n] {
        tmp = a;
        a = b;
        b = tmp + b;
    }
    return a;
}
```

Runs in $O(N)$.

1.2.5 Recursive Fibonacci

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

Runs in $O(2^N)$.

1.2.6 Finding the highest sum of two pairs of numbers in two arrays

```
function highest_sum(array1, array2) {
    max_sum = 0;
    forall i = [0, size(array1)) {
        forall j = [0, size(array2)) {
            max_sum = max(max_sum, array1[i] + array2[j]);
        }
    }
    return max_sum;
}
```

Runs in $O(NM)$, where N and M are the sizes of the arrays.

1.3 Why you should care

Understanding this is all well and good, but why does it actually matter? Informatics problems will be given with data constraints and time constraints. To solve a problem, one needs to be aware of how these two interact with the processing capacity of a computer. A computer can perform 10^8 calculations in one second. If you're given one second to compute the answer, and your algorithm runs in $O(N^2)$ time with $N \leq 10^5$, your solution will not run in time.

2 Data Structures

Some simple data structures that will come in handy. Be sure to thoroughly understand the principles behind each and potential applications. Concrete implementation is usually left to the standard library of your language of preference.

2.1 Stack

Allows access to elements in order FILO: first in last out. A stack can be imagined as a pile of books, where you only have access to the book on top. Logically, the topmost book will be the last one you put on. A stack provides essentially three methods of interaction: push, pop, peek. The first two insert or remove an item from the stack, and the last looks at the topmost item.

Try implementing a stack with a flat array, given a maximum size.

2.1.1 An application

Consider the following strings:

```
(([])[[]]()()[([()])])
([][()([[()]])])
([]])
```

How can we determine if they are *well-formed*? That is, each opening parenthesis and bracket is matched with its closing pair, and in a way such that they do not intersect.

2.2 Queue

Allows access to elements in order FIFO: first in first out. A queue can be imagined as a fair queue in real life. Consequently, a queue is usually used for applications where an order of operations must be maintained, with some operations spawning more operations - functioning like a buffer. It provides three methods of interaction: enqueue, dequeue, peek. Like the stack, the first two insert or remove an item from the queue, and the last looks at the *front* of the queue.

Try implementing a queue with a flat array, given a maximum size, and without shuffling elements on dequeue.

3 Websites to visit

I recommend exploring the past AIO problems from here above the other links
`http://orac.amt.edu.au/cgi-bin/train/hub.pl`
`https://projecteuler.net/`
`http://codeforces.com/`