Big O Complexity

As we discussed in class, computer scientists use a special shorthand called big-O notation to denote the computational complexity of algorithms. When using big-O notation, the goal is to provide a qualitative insight as to how changes in *N* affect how many units of computation are performed for large amounts of data. Therefore, when computing big-O, we can make the following simplifications:

1. Eliminate any term whose contribution to the total is insignificant as *N* becomes large
   $O(n^2 + n) = O(n^2)$ for large *n*
2. Eliminate any constant factors
   $O(3n) = O(n)$ for large *n*

To compute big-O, it we think about the number of executions that the code will perform in the ***worst*** case scenario. The stragegy for computing Big-O depends on whether or not your program is recursive. For the case of iterative solutions, we try and count the number of executions that are performed. For the case of recursive solutions, we first try and compute the number of recursive calls that are performed.

Basic Examples

Code	**Complexity**

`for (int x = n; x >= 0; x--) {`
`  cout << x << endl;`
`}`
	$O(n)$
	The loop executes *n* times

`for (int i = 0; i < 5; i++) {`
`  for (int j = 0; j < 10; j++) {`
`  cout << j << endl;`
`  }`
`}`
	$O(1)$
	Neither loop depends on the value *n*

`for (int i = 0; i < n; i++) {`
`  for (int j = 0; j < n; j++) {`
`  cout << j << endl;`
`  }`
`}`
	$O(n^2)$
	The outer loop executes *n* times and each iteration, the inner loop executes *n* times, totaling in n^2 loops

`for (int i = 0; i < 5n; i++) {`
`  cout << "hello!" << endl;`
`}`
	$O(n)$
	The loop executes *5n* times and the constant **5** is insignificant as *n* grows

`for (int i = 0; i < n; i++) {`
`  for (int j = 0; j < n*n; j++) {`
`  cout << “tricky!” << endl;`
`  }`
`}`
	$O(n^3)$
	
	The outer loop executes *n* times and each iteration, the inner loop executes n^2 times, totaling in n^3executions

Gauss Summation

In many situations you have a case where you have a code block which executes 1 time, then 2 times, then 3 times until n times. In order to calculate the Big-O for code that follows this format we use the solution for the sum of an arithmetic series.

$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N$$

$$= \frac{N \cdot (N+1)}{2}$$

$$= \frac{1}{2}N^2 + \frac{1}{2}N$$

Which is

$$\mathcal{O}(N^2)$$

In class I incorrectly gave credit for this sum to Carl Gauss. Upon further investigation I found out that was first discovered in 499 AD by Aryabhata, a prominent mathematician-astronomer from the classical age of Indian mathematics and Indian astronomy!

Dividing in half

Binary search worked by dividing a search space of number in half until the algorithm finds the target value. In the worst case scenario it must repeat this process until the search space only has one element. So the question is, how many times can you divide *n* in half until you have only 1 element?

$$\frac{n}{2^x} = 1$$

$$x = \log_2 n$$

And thus the big O of binary search is $O(\log n)$

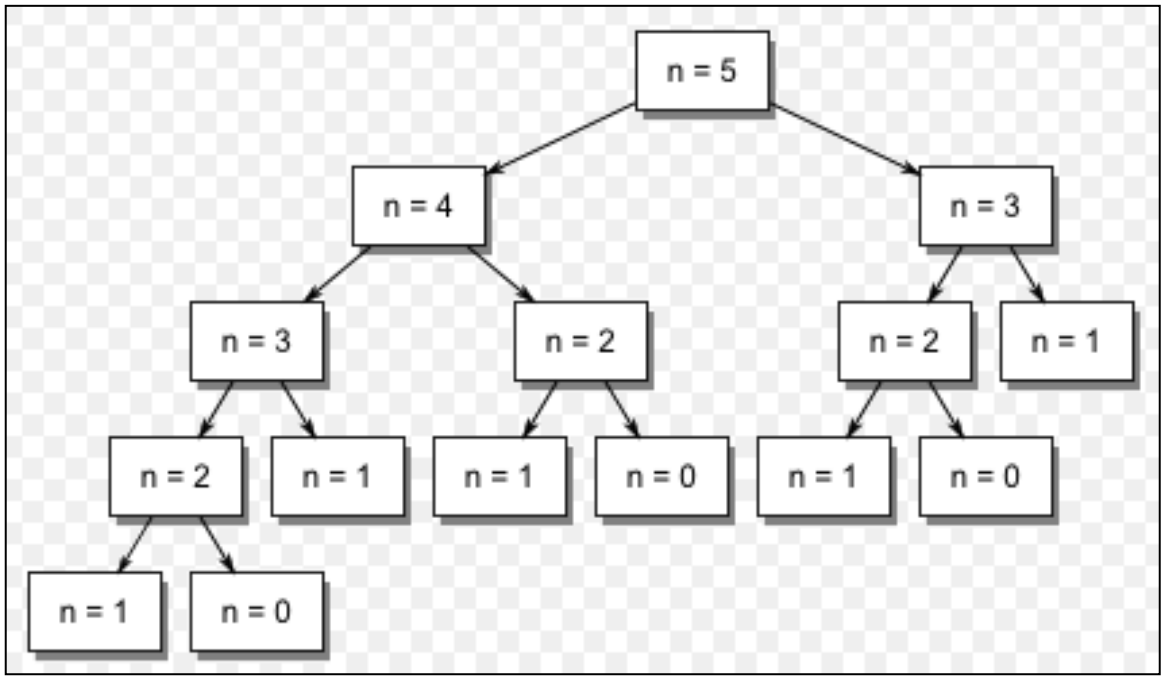
Recursion

If we ask a question on the midterm where you need to compute the Big O of a recursive function it will be of the form where you simply need to calculate the number of calls in the recursion call tree. Often the number of calls is big $O(b^d)$ where b is the branching factor (worst case number of recursive calls for one execution of the function) and d is the depth of the tree (the longest path from the top of the tree to a base case).

For example consider the Fibonacci function:

```
int fib(int n) {
  if(n <= 1) {
    // base case
    return 1;
  } else {
    // recursive case
    return fib(n – 1) + fib(n – 2);
  }
}
```

Here is the call tree of Fib(5);



The branching factor is 2. The depth is *n* where *n* is the number on which the function is called. Thus the big O of this function is:

$$\mathcal{O}(b^d) = \mathcal{O}(2^n)$$