Lab 9: Algorithmic Complexity

Big O

In computer science there are often many different algorithms that could be used to accomplish any task. For instance to sort a list of numbers a program might use: bubble sort, selection sort, insertion sort, merge sort, heap sort, quicksort, radix sort, bucket sort, bogo sort, or many, many others. They all do the same thing so how can we tell which one is the correct choice.

Efficiency (or complexity) is the most common method of picking the "correct" algorithm to use in a program. This is a measure of how many actions an algorithm performs relative to the number of inputs it is given. Algorithms with higher complexity (lower efficiency) require more and more actions to accommodate a growing number of inputs. This is usually expressed as a **Big O** number.

You'll learn much more about Big O later in your computer science education but let's walk through a simple code example to try to develop some intuition about what it measure.

```
//Code provided on page 464 of Starting out with C++ 9th Edition by Tony Gaddis

int linearSearch (const int arr[], int size, int value){
   int index = 0;
   int position = -1;
   bool found = false;

   while(index < size && !found){
      if(arr[index] == value) {
         found = true;
         position = index;
      }
      else{
         index++;
      }
   }
   return position;
}
```

This is the same code that was provided for Linear Search in the last reading. Let's count how many actions this code takes for an input **arr** that has 3 numbers in it and does not contain the search value:

- variable initialization: 3 actions
- while loop: 3 iterations
  - comparison: 1 action
  - update index: 1 action
- return statement: 1 action

Since the **while** loop runs three times it will perform 3*2 = 6 actions. 6+3+1 gives us 10 actions for a list of 3 numbers. Lets look at how many actions would be performed to search lists of different sizes. Let us assume that no list contains the search value.

Size	Calculation	Actions
3	3 + 3*2 + 1	10
6	3 + 6*2 + 1	16
12	3 + 12*2 + 1	27
24	3 + 24*2 + 1	52

We can see that the calculation is always **(2*n + 4)** where n is the size of the list that is being searched. We can use that calculation to get the number of actions that will be performed to search a list of any size.

We can also see that if we put those points of a graph with Size on the X axis and Actions on the Y axis, there would be a straight line that could be drawn through it. That shows us that the **complexity** of this algorithm is **linear** (or we might say that it runs in "linear time"). To predict how many many actions are needed to search a list with Size=1,000,000 we would just have to plot that line out to the point where Size is 1,000,000

The Big O measurement of Linear Search is the equation from our table **without any constants**. It is written as a function O() with the equation in the parenthesis. So we would say that Linear Search is O(n). Big O isn't meant to tell us exactly how many actions an algorithm performs. Instead it gives us a convenient shorthand to compare the complexity of two algorithms. **An algorithm that is O(n^2) is going to be much slower on long lists of numbers that an algorithm that is O(n)**.

But we cheated a little to get the calculation that we used for Linear Search. Remember that we assumed that the search value is not in the list. But the algorithm stops running when it finds the search value, so it is faster to search a list that contains the search value compared to one that does not. Linear Search does not require that the searched list be sorted so the search value is equally likely to be at the very beginning as it is at the very end. So on average the search value is halfway through the list. We can update our equation to search a list that **does contain** the search value to be (0.5*2*n + 4), since we're only going to search half of the list on average. The Big O for this equation removes all of the constants so it is still O(n).

Remember that Big O is just a rough estimation.

Here are some examples of the Big O complexity of some common classes of algorithms.

Big O	**Algorithms**
O(1)	Simple calculations
O(log n)	Binary Search
O(n)	Linear Search
O(n log n)	"Divide and conquer" searches
O(n^2)	Bubble sort, Insertion sort, Selection sort
O(2^n)	Brute Force attacks

Improving Efficiency

There are many small tricks that a programmer can apply to improve the efficiency of his or her code. You will learn many of these tricks as you continue to study, but here are a few:

- Reduce the layering of loops. Loops in side of loops are a major source of inefficiency.
- Avoid referencing memory as much as possible. Loading and storing values in memory is much slower than calculations that can happen inside of the CPU.
- Avoid repeating complicated calculation. Store and re-use function returns rather than calling the same function several times.
- Quit working as soon as possible. If there are several conditions that could cause your code to halt, check the simplest ones first.
- Use integers whenever you can. Floating point operations require more work that integers.

Notice that many of the tips in that list will not improve the Big O complexity of your code. **A function that performs (1000*n + 1000) actions is the same Big O as a function that performs (10*n + 1)** but will actually take much more time to run, and so it is less efficient.

Often the Big O value of code is determined only by the algorithm that you're implementing but there are still many small improvements that can be made to decrease the execution time of the program. Consider last week's implementation of **Bubble Sort:**

```
//Code provided on page 481 of Starting out with C++ 9th Edition by Tony Gaddis

void bubbleSort(int array[], int size){
   int maxElement;
   int index;

   for(maxElement = size-1; maxElement >0; maxElement--){
      for(index = 0; index < maxElement; index++){
         if(array[index] > array[index+1]) {
            swap(array[index], array[index + 1]);
         }
      }
   }
}

void swap(int &a, int &b) {
   int temp = a;
   a = b;
   b = temp;
}
```

Let’s try sorting a short list using this function: 1, 2, 4, 3

1, 2, 4, 3	index: 0	maxElement: 3	No swap
1, 2, 4, 3	index:1	maxElement: 3	No swap
1, 2, 4, 3	index:2	maxElement: 3	Swap
1, 2, 3, 4	index:0	maxElement:2	No swap
1, 2, 3, 4	index:1	maxElement:2	No swap
1, 2, 3, 4	index:0	maxElement:1	No swap

Notice that the list was correctly sorted after the swap on the third lane. All of the remaining actions by this function were just waited effort. You may recall that the implementation of Bubble Sort that was used in last weeks lab used a **bool** variable to track whether a change had been made in the list. If one "pass" can be made without any swaps taking place, we know the list is already sorted and the sorting function can quit. Let’s alter this Bubble Sort function so that it does the same thing.\

```
//Code adapted from page 481 of Starting out with C++ 9th Edition by Tony Gaddis

void bubbleSort(int array[], int size){
   int maxElement;
   int index;
   bool sorted;

   for(maxElement = size-1; maxElement >0; maxElement--){
      sorted = true;
      for(index = 0; index < maxElement; index++){
         if(array[index] > array[index+1]) {
            swap(array[index], array[index + 1]);
            sorted = false;
         }
      }
      if(sorted) break;
   }
}

void swap(int &a, int &b) {
   int temp = a;
   a = b;
   b = temp;
}
```

Now the function will return as soon as the array is sorted, rather than continuing to process an already sorted list. We do a little more work in every loop (because we have an extra assignment and comparison that happens every iteration) but we can potentially save several iterations of the outer loop which makes this implementation much more efficient even though the algorithm and Big O analyses is unchanged. Always look for these kinds of improvements in your code.