COMP 211: Lab 5
Fall, 2014

In this lab, we will introduce *big-O notation*, and use it to summarize the time complexity of functions.

1 Counting Steps

A *step* of computation is any individual instruction; for today, we will count arithmetic (adding or subtracting) and assignments to a variable or array cell.

For example, the following code takes 3 steps:

```
int i = 0; // 1 step for assignment
i = i + 1; // 2 steps, 1 for + and 1 for assignment
```

1.1 Remove Red 1

Consider the following function, which removes the red from an image (modifying it in-place, rather than creating a new array, as we did in the homework):

```
void remove_red(pixel[] A, int width, int height)
{
  int length = width * height;

  for (int i = 0; i < length; i = i + 1)
    {
      A[i] = A[i] & 0xFF00FFFF;
    }
}
```

Task 1.1 Fill in the following table, counting the number of steps of computation when you call `remove_red` on arrays of various sizes:

`width*height`	number of steps of computation
0	
1	
2	
6	

Task 1.2 Give a formula expressing the number of steps of computation in terms of `width*height`.

1.2 Remove Red 2

Now consider the following alternate version of the function, which computes the same thing:

```
void remove_red (pixel[] A, int width, int height)
{
  for (int i = 0; i < height; i = i + 1)
  {
     for (int j = 0; j < width; j = j + 1)
      {
        A[i*width + j] = A[i*width + j] & 0xFF00FFFF;
      }
  }
}
```

Task 1.3 Fill in the table for this version:

`width*height`	number of steps of computation
0	
1	
2	
6	

Task 1.4 Give a formula expressing the number of steps of computation in terms of `width*height`.

2 Big-O Basics

The formulas you arrived at the previous two tasks should be something like

- `5*width*height + 2` for the first version
- `6*width*height + 2*height + 2` for the second version

These are different, but "more or less the same", in the sense that both are proportional to `width*height`.

2.1 Big-O Notation

Big-O notation is a way of categorizing functions based on the idea of *ignoring constant factors*. For example, we say that both of the above running times are $O(\texttt{width} * \texttt{height})$. This gives a summary of the running time of a program, which is inexact, but often useful for choosing between different algorithms that solve the same problem. The reason is that, on very large inputs, the constant factors will be less important that whether the algorithm runs in, say, logarithmic time versus linear time.

Formally, for two functions f, g, we say that

> **Big-O Definition (simplified[1])** f is in $O(g)$ if and only if there exists a constant k such that for all x, $f(x) \leq kg(x)$

That is, f is always less than or equal to some constant multiple of g. Here are some examples of using this definition:

- `5 * width * height` is $O(\texttt{width} * \texttt{height})$, which we can prove by choosing $k = 5$, because

$$5\texttt{width} * \texttt{height} \leq 5\texttt{width} * \texttt{height}$$

- `6 * width * height + 2*height` is $O(\texttt{width} * \texttt{height})$, which we can prove by choosing $k = 8$: $\texttt{height} \leq \texttt{width} * \texttt{height}$ (assuming that $\texttt{width} > 1$), so $2\texttt{height} \leq 2\texttt{width} * \texttt{height}$, so

$$6\texttt{width} * \texttt{height} + 2\texttt{height} \leq 6\texttt{width} * \texttt{height} + 2\texttt{width} * \texttt{height} = 8\texttt{width} * \texttt{height}$$

Task 2.1 Show that the function $3n^2 + 8n$ is $O(n^2)$ by finding a k according to the definition of big-O. You can assume $n \geq 1$.

[1] See the Lecture 10 notes for the full definition; there's one additional twist

2.2 Big-O Facts

For the purposes of this course, we won't really need to use the definition of big-O directly. Mostly, we will use the following facts:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset \ldots$$

That is, the class of *constant functions* ($O(1)$) is contained in the class of *logarithmic functions* ($O(\log n)$), which is contained in the class of *linear functions* ($O(n)$), which is contained in $O(n \log n)$, which is contained in the class of *quadratic* functions ($O(n^2)$), and so on.

To find the big-O, you ignore constants and take the biggest summand of a sum. For example:

- $an^2 + bn + c \log n$ is $O(n^2)$, because the n and $\log n$ terms can be "folded into" the n^2 term by choosing an appropriate constant
- $an + bn \log n$ is $O(n \log n)$

Because big-O is an *upper bound* on the size of a function, the classes are cumulative: if f is $O(n)$, then it is also $O(n^2)$ and $O(n^3)$ etc. If we ask for a "tight big-O bound", that means to give the smallest big-O bound that you can from the above list of options (e.g. don't say $O(n^2)$ if it is also $O(n)$).

3 Algorithm Analysis

Consider the code for linear search:

```
int search(int x, int[] A, int n)
//@ requires \length(A) == n;
//@ requires is_sorted(A,0,\length(A));
/*@ ensures (\result == -1 && ! is_in(x,A,0,\length(A)) ) ||
            (0 <= \result && \result < \length(A) && A[\result] == x); @*/
{
  for (int i = 0; i < n; i = i + 1)
    //@loop_invariant 0 <= i;
    //@loop_invariant i == 0 || A[i-1] < x;
    {
      if (A[i] == x) { return i; }
      if (A[i] > x) {return -1;}
    }

  return -1;
}
```

Task 3.1 Give a tight big-O bound for the worst-case running time of linear search, in terms of n.

Consider the code for binary search:

```
int search(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
        || (0 <= \result && \result < n && A[\result] == x); @*/
{
  int lower = 0;
  int upper = n;

  while (lower < upper)
    //@ loop_invariant 0 <= lower && lower <= upper && upper <= n;
    //@ loop_invariant lower == 0 || (x > A[lower - 1]);
    //@ loop_invariant upper == n || x < A[upper];
    {
      int mid = lower + (upper - lower) / 2;
      //@ assert lower <= mid && mid < upper;
      if (A[mid] == x) {
        return mid;
      }
      else if (A[mid] > x) {
        upper = mid;
      }
      else {
        lower = mid;
      }
    }

  return -1;
}
```

Task 3.2 Give a tight big-O bound for the worst-case running time of binary search, in terms of n.