due Sunday, 27 January 2018, at 11:00 PM

Note: If you choose to work with a partner on this homework, you need to register by 5:00 PM on January 21. Instructions for doing so will be given later, but you will need to follow them exactly. You are allowed to work with a partner in a different section.

When you submit your assignment solutions please take the time to indicate the starting page of each answer. To make life easy for the TA's put the answer to each problem on a separate page (you may put multiple parts of the same problem on one page)

Submissions for the written part of the homework, including your analysis of the programs should be submitted to GradeScope (these will be set up as two distinct assignments – `Homework 1` and `Program 1 Analysis`). The source files for your programs should be submitted to `Assignment 1` in Moodle.

Note. *Keep in mind that the purpose of a proof is to convince the reader. Anything you can do to help the reader understand the ideas behind your argument is worth including. Examples and pictures can be particularly helpful but* ***they are not sufficient.***

[Total points: 53]

1. [8 points] *Purpose: Understanding asymptotic notation*

   Consider the list of nine functions below:

   $$n^{\lg n} \quad (\lg n)^2 \quad n! \quad 2^n \quad n^{0.0001} \quad n^n \quad (n-1)! \quad n^2 \quad 2^{(\lg n)^2}$$

   Sort them into a sequence $g_1(n), \ldots, g_{10}(n)$ so that for $i = 1, \ldots, 8$, you have either $g_i \in \mathrm{o}(g_{i+1})$ or $g_i \in \Theta(g_{i+1})$. Prove all eight relationships using what you know about little-oh and big-theta.

2. [6 points] *Purpose: Understanding asymptotic notation in the context of summations*

   Let

$$f(n) = \sum_{i=1}^{n} i^d (\lg i)^m$$

   where d is any real number > 0 and m is an integer ≥ 1. Give a simple (one term) function $g(n)$ such that $f(n) \in \Theta(g(n))$. Prove your answer using the definitions of big-oh, big-omega, anything you have learned about bounds on summations and, if appropriate, limits.

3. [8 points, not evenly distributed] *Purpose: application of the Master Theorem and ability to recognize and deal with situations where it does not apply*

   Give Θ bounds for $T(n)$ in each of the following recurrences. Assume $T(n)$ is constant for small values of n. In situations where the solution is based on the Master Theorem,[1] please state which case of the Master Theorem applies. If the Master Theorem does not apply, solve the recurrence using the tree/levels method – it is sufficient to describe the intermediate and bottom levels of the tree, set up the summation, and solve. You can choose the value of n at which $T(n)$ is a constant, i.e., you can choose the base case for the recursion to be what ever is most convenient. Also, you can assume that $n = b^k$ for some k to ensure that all smaller n are integers.

   (a) $T(n) = 17 \cdot T(n/4) + n^2 \lg n$

   (b) $T(n) = 2 \cdot T(n/2) + n \lg^2 n$

   (c) $T(n) = 4 \cdot T(n/2) + \dfrac{n^2}{\lg n}$

   (d) $T(n) = 5 \cdot T(n/4) + \dfrac{n^{5/4}}{\lg^2 n}$

[1] There is a more general version of the Master Theorem available on the internet. The term "Master Theorem" here refers to the one in the textbook. If you use the more general version, you have to prove it yourself.

4. [credited to Gary Miller in Jeff Erickson's book] [8 points]

Consider the following sorting algorithm. The parameter $A[1..n]$ in both procedures can refer to any sequence of contiguous elements of A. So, for example, when UNUSUAL($A[1..n]$) calls UNUSUAL($A[n/4+1..3n/4]$), $n/2$ plays the role of n, $A[n/4+1]$ plays the role of A[1], and $A[3n/4]$ plays the role of $A[n]$.

```
CRUEL($A[1..n]$) is          ▷ sorts the array $A$
    if $n \geq 2$ then
        CRUEL($A[1..n/2]$)
        CRUEL($A[n/2+1..n]$)
        UNUSUAL($A[1..n]$)
    endif
end CRUEL
```

and

```
UNUSUAL($A[1..n]$) is       ▷ sorts $A$ assuming both halves have been sorted
    if $n = 2$ then
        if $A[1] > A[2]$ then swap $A[1] \leftrightarrow A[2]$ endif    ▷ the only comparison
    else
        for $i \leftarrow 1$ to $n/4$ do
            swap $A[i+n/4] \leftrightarrow A[i+n/2]$ end do
        UNUSUAL($A[1..n/2]$)                 ▷ recurse on left half
        UNUSUAL($A[n/2+1..n]$)               ▷ recurse on right half
        UNUSUAL($A[n/4+1..3n/4]$)            ▷ recurse on middle half
    endif
end UNUSUAL
```

This sorting algorithm is ***oblivious*** – it's behavior does not depend on the values of the numbers stored in the array. You may assume that the input size n is a power of 2.

(a) [4 points] Prove by induction that CRUEL sorts an array correctly.

(b) [1 point] Prove that CRUEL would *not* sort correctly if we removed the **for** loop from UNUSUAL.[2]

(c) [1 point] Prove that CRUEL would *not* sort correctly if we swapped the last two lines of UNUSUAL.

(d) [2 points] Give a Θ bound for the number of comparisons and swaps performed by UNUSUAL when applied to an array of n elements? Prove your answer.

(e) [2 points] Give a Θ bound on the number of comparisons and swaps performed by CRUEL when applied to an array of n elements? Prove your answer.

[2] In order to prove that an algorithm does not produce the correct output, you need to give a counterexample.

Programming Problem: Longest Contiguous Subsequence of Unique Values

[8 points for each of the two programs, 5 points for the analysis.]

For this problem, you get to implement two small programs with different asymptotic performance, and write up a short analysis of each of them. This will give us a chance to make sure everyone is ready to write some programs that we can test, and it will give you a chance to connect the formal study of algorithms to implementation problems you may face as a working software developer.

For this problem, you'll be reading in a sequence of values and looking for the longest contiguous subsequence of values that doesn't contain any duplicates. Consider the following sequence of values. Starting from the beginning, there's a 2-element sequence that doesn't contain any duplicates, 3 and 2. The first three elements are 3, 2 and 3, which contains two copies of the value 3. So, the 2-element subsequence starting at index 0 doesn't contain duplicates, but the 3-element sequence starting at index 0 does contain a duplicate.

```
3 2 3 7 2 5 9 3 7 7
```

If we look for the longest contiguous sequence of values without duplicates, we have a 3-way tie. There's the sequence 3, 7, 2, 5, 9 (starting from index 2), the sequence 7, 2, 5, 9, 3 (starting from index 3) and the sequence 2, 5, 9, 3, 7 (starting from index 4).

Your program will report on the longest contiguous subsequence that's free of duplicates. If there's a tie for length, it will report the one that starts earliest in the sequence. We'll assume values in the sequence are indexed starting from zero. So, for the example above, your solution should report the following, the longest contiguous subsequence that's free of duplicates starts at index 2 and it's 5 elements long:

```
2 5
```

Input Format

Your program will read input from standard input, as if the user is typing in all the input for the program. This should make it easy to test your program, and (as shown below) we can use I/O redirection to get your program to read from in input file instead of reading from the user.

Input will start with a positive integer, n, giving the number of values in the sequence. This will be followed by n non-negative integer values. These values may be split across multiple lines, just to make them easier to look at in a text editor (even for big input files). The example given above would look like the following when given as input (it's actually the sample input, `input-1.txt`):

```
10
3 2 3 7 2 5 9 3 7 7
```

For some tests, the value of n may be large, but you can assume all the input values can be stored in memory at the same time. Values in the sequence will be no larger than 10^9, so each value will fit in an int in any of the implementation languages. You're getting six sample inputs with this assignment. The smallest of these contain sequences of just 10 values, but the largest one has 80,000 values.

Solution Techniques

You're going to solve this problem two ways. The first is an extremely naive and inefficient solution that should be easy to implement. The second is a little harder to implement. It uses more memory than the fist technique, but it achieves a better asymptotic runtime. Both of your solutions are sub-optimal; there are better ways to solve this problem (that you can think about if you want).

Your first solution will be called `unique1.c` (or `unique1.cpp` or `unique1.java`, depending on your implementation language). Your second, more efficient, solution will be called `unique2.c` (or `unique2.cpp` or `unique2.java`). Both solutions will work as follows:

```
for start = 0 to n - 1
     consider longer and longer (contiguous) subsequences starting from
        start until the subsequence contains a duplicate value
     check to see of the subsequence contains a duplicate
     if it doesn't contain duplicate
        see if the subsequence is longer than the longest one previously found
        if it is
            that's a new (better) longest subsequence
 report the longest subsequence found
```

The outer loops of this program look at different portions of the input sequence. In the outermost loop, the starting position is moving through each element of the sequence. In the inner loop, we're looking at longer and longer subsequences starting from that start position of the input.

The two versions of your program will differ in how they detect duplicate values. The first version (`unique1.(c,cpp,java)`) will just compare the last element of the current subsequence against all previous elements in the subsequence to see if its a duplicate. Consider the following sequence of values, for example. Let's say start is index 2 (so, we're pretending we are a few iterations into the execution). When we're (eventually) checking the length-4 subsequence for duplicates, we'll compare the 5 against the previous values in the subsequence (3, 7 and 2). Since it's not a duplicate, we can keep going on, checking longer and longer subsequences that start at index 2.

```
3 2 3 7 2 5 9 3 7 7
```

The second version of our program ((`unique2.(c,cpp,java)`) will improve on this technique a little bit. Instead of doing a linear search through earlier values to look for duplicates, we'll store the values from the current subsequence in a data structure. The data structure you use will depend on your implementation language. If you're using C++ or Java, use a balanced binary tree for storing these values. This data structure is readily available in the form of a TreeMap (in java) or an STL set (for C++). If you're programming in C, you won't have access to these pre-built collections, so you'll need to implement your own data structure. If you're using C, just implement a hash table. That's a little more work for the C programmers, but a hash table is a lot easier to build than a balanced tree.

In any of these languages, these data structures will give you an efficient way to check for duplicates, better than a simple, linear-time search. Remember, each time you move the start ahead, you'll need to clear your data structure.

Implementation Language

For this problem, you get to implement your solution in C, C++ or Java. If you implement your solution in C, call the two versions of your program `unique1.c` and `unique2.c`. If you're programming in C++, name your source files `unique1.cpp` and `unique2.cpp`. If you're using Java, then use `unique1.java` and `unique2.java`.

- Programming in C

  If you're programming in C, we're going to build and test your executable using commands like the following. Your program should be written as if it's going to read input from the user, but we'll use input redirection to get it to read from a file instead.

  The behavior of C and C++ programs can vary from platform to platform, so you'll definitely want to give yourself enough time try out your work on a University EOS Linux before you turn it in.

  ```
  gcc -Wall -std=c99 -g -O2 -o unique1 unique1.c
  ./unique1 < input-1.txt
  ```

- Programming in C++

  If you're programming in C++, we're going to build and test your executable using commands like the following. Your program should be written as if it's going to read input from the user, but we'll use input redirection to get it to read from a file instead.

The behavior of C and C++ programs can vary from platform to platform, so you'll definitely want to give yourself enough time try out your work on a University EOS Linux before you turn it in.

```
g++ -Wall -g -O2 -o unique1 unique1.cpp
./unique1 < input-1.txt
```

- Java

  For a Java program, don't put your classes in a package; just leave them in the default package. This will make it easy for us to compile and test your program without having to match your directory structure. If you use an Integrated Development Environment (IDE, like Eclipse), be sure it doesn't put your code inside a package for you.

  Java is a little more tolerant of differences between platforms (compared to C and C++), but even here, you should try out your solution on a University EOS Linux machine before you turn it in. It's possible that a difference in the compiler version could prevent your program from compiling and running correctly when we try to test. We'll use commands like the following when build and test your solution (this is why you shouldn't put your code inside a package). Here, you can see we expect your main class to be lower-case. This isn't how you normally name classes in Java, but it makes the Java filenames more consistent with the other languages:

  ```
  javac unique1.java
  java unique1 < input-1.txt
  ```

Test Inputs

On the course homepage, We're providing six test input files. These include tests for some special cases and a larger test input to let you see what kind of performance your solution gets. The following shows what you should get for each of these inputs (assuming your solution is in C or C++. You'd run the program differently for java, but you should get the same output). When we test your solutions, we'll use these test inputs, probably along with a few other tests we're not distributing with the assignment.

```
eos$ ./unique1 < input-1.txt
2 5
eos$ ./unique1 < input-2.txt
0 10
eos$ ./unique1 < input-3.txt
1 8
eos$ ./unique1 < input-4.txt
0 1
eos$ ./unique1 < input-5.txt
20 26
eos$ ./unique1 < input-6.txt
69909 1346
```

You can see here, I'm redirecting input from a file, so, even though my program thinks its reading from the user, the shell can trick it into reading from a file without any change to the program.

Program Analysis

In addition to your programs, you will need to write up an asymptotic analysis for the worst-case performance of both versions. The runtime you get will probably depend on the data structure you used in version 2 of the program. Put your analysis in a file called `unique.pdf`.

In your analysis, you should include enough detail to permit someone else to understand your approach even if they're not looking at your source code. Use n to represent the number of values in the input sequence and give your total runtime using big-O notation. Your analysis can be informal; you don't have to prove the asymptotic runtime is correct. I'm expecting your analysis of each version should take about half a page (maybe a little less).

Submitting Your Work

In Moodle, you'll see an assignment named `Assignment 1`. Submit the source code to your two programs using this link. For your analysis, submit that to a GradeScope assignment with the name `Program 1 Analysis`.