Design and Analysis of Algorithms, Assignment 1

Each part of this assignment will be submitted as a separate file. As you finish up the various parts, submit your file using WolfWare Classic, at `https://submit.ncsu.edu/`. If you improve part of your solution, you can re-submit the same file, and we'll automatically grade the latest version of the file you submit.

This assignment includes a programming problem and some problems where you get to write up a short answer, an argument or an explanation. For the non-programming problems, you'll need to turn in a PDF file for your solution. You can prepare the solution in any software package you like (e.g., Microsoft Word, Open Office, LaTeX), but you'll need to use correct mathematical notation in whatever package you choose, and you'll need to create a PDF version of your work when you turn it in.

For the programming problems, be sure your solution compiles and executes successfully on one of the EOS Linux systems. You can access these machines from the public computing labs, and they're available remotely. The university has a page at `http://www.eos.ncsu.edu/remoteaccess/` with brief instructions about accessing and using these systems. I'll post some additional instructions on our course homepage.

Depending on your choice of programming language, here are some things you'll need to consider to make sure your programming solutions work as well for us when we're grading them as they do for you when you're writing them.

- Programming in C

  If you're programming in C, we're going to build your executable using a command like the following. 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 an EOS Linux before you turn it in.

  ```
  gcc -Wall -std=c99 -g -O2 -o program program.c
  ```

- Programming in C++

  If you're programming in C++, we're going to build your executable using a command like the following. 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 an EOS Linux before you turn it in.

  ```
  g++ -Wall -g -O2 -o program program.cpp
  ```

- 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.

  For Java, note that the default compiler on the EOS Linux machines is kind of old, version 1.6. Most of the features we're likely to need will be there in this version of the language, but there are two missing things that sometimes catch people by surprise. In a switch statement, you can't use a string as the switch value (added in 1.7), so you'll have to do this kind of thing with nested if statements. Also, when instantiating a generic class, you won't get the left-to-right type inference to help shorten your code (also added in 1.7). So, for example, you'll have to use:

  ```
  ArrayList< Integer > list = new ArrayList< Integer >();
  ```

  instead of:

  ```
  ArrayList< Integer > list = new ArrayList< >();
  ```

When we compile and run your Java programs, we'll use commands like the following (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 program.java
java program
```

- Python

  Right now, it looks like Python 3 isn't installed on the EOS Linux systems, so we'll have to stick to Python 2 for our programming assignments. When we run your python programs, we'll do it like this:

  ```
  python2 program.py
  ```

- C#

  It looks like we don't have a C# compiler installed on the EOS Linux machines. I'm asking about this, but we'll have to get by without C# on homework assignment 1.

1. For this problem, you'll be comparing functions that describe the running times of two different algorithms (well, at least we'll be pretending they do). For each question, you'll need to do two things. Report which of the two algorithms is asymptotically faster (i.e., which function has a slower rate of growth). Also, report the trade-off point between the two functions, the smallest value n_0 such that, for all $n \geq n_0$, the faster-growing function has a larger value at n than the slower-growing function. Give this answer rounded to one fractional digit of precision (e.g., a number line 25.2 or 1040.0). You can find the (approximate) trade-off point, n_0 however you want. If you can do it with algebra, that's great, or you could sample the two functions at various point or even look closely at a plot of the the functions. Any of these techniques is fine. Turn in your answer as a PDF file named `compare.pdf`. (5 pts each)

   (a) Consider the two functions, $f_1(n)$ and $g_1(n)$.

   $$\begin{aligned} f_1(n) &= 0.5 * n^2 - 10 * n \\ g_1(n) &= 3 * n \end{aligned}$$

   (b) Consider the two functions, $f_2(n)$ and $g_2(n)$.

   $$\begin{aligned} f_2(n) &= 3n \log_2 n \\ g_2(n) &= 2n^{1.5} \end{aligned}$$

2. Asymptotic Bound Proofs

   Write up a short proof of each of the following asymptotic bounds. Submit your argument as a file, `bounds.pdf` (10 pts each)

   (a) $2 * n^2 + 5n \log_2 n \in O(n^2)$

   (b) If $f(n) \in \omega(g(n))$ then $f(n) \in \Omega(g(n))$

3. Recursion Tree Problems

   For each of the following recurrence relations, give a short formula for the cost of the root node, the total cost of the leaves, and the total cost of all the internal nodes (so, you'll be giving three formulas for each). Be sure to label your formulas so we can tell what you're showing us. For the internal nodes, you can leave your answer in summation notation; you don't have to simplify it. Submit your answer as a file named `tree.pdf`. (10 pts each)

   (a) $T(n) = 4T(n/3) + O(n\sqrt{n})$

   (b) $T(n) = 3T(n/3) + O(n)$

4. Master Method

   Use the master method to give a solution to each of the following recurrences. Briefly explain your application of the master method (e.g., which case applies to each problem). Write up your answers in a file called `master.pdf` (5 pts each)

   (a) $T(n) = 2T(n/4) + \Theta(\sqrt{n})$

   (b) $T(n) = 9T(n/3) + \Theta(n)$

   (c) $T(n) = 5T(n/2) + \Theta(n^2)$

   (d) $T(n) = 2T(n/2) + \Theta(n\sqrt{n})$

5. Best-case running time

   We normally don't care too much about the best-case running time for an algorithm; it's just not that useful or informative about the algorithm's performance. Consider the selection sort algorithm we looked at on the first day of class (you'll find it on the slides). Determine what the best-case running time of this algorithm and describe it asymptotically.

   Now, describe a simple modification of this selection sort algorithm so its best-case running time becomes $\Theta(n)$ for an input sequence of length n. This shows why best-case running time isn't that useful. It's usually easy to get an algorithm to handle some special cases quickly, even if this doesn't change its worst-case or average-case behavior.

   Submit your answer as a file, `best.pdf`. (5 points)

6. Common Substring Counting

   Consider the two strings below. The first string has 28 substrings, all of them different. For example "abcd" and "cdef" are both four-character substrings of the first string. The second string has 36 substrings, but some of them are duplicates.

   ```
   abcdefg
   cdefabab
   ```

   Your job is to develop a program (named common.c, common.cpp, common.java or common.py) that counts the number of different strings that are substrings of both of two input strings. For example, "f", "ab" and "cde" are all substrings of the two strings above. The "ab" substring occurs twice in the second string, but you should only count it once.

   Your program will read its input from standard input, one string per line. Input strings will consist of lower-case letters. As output, your program should simply print the number of different strings that are substrings of the two input strings. (15 pts for a working program)

Sample Input

```
abcdefg
cdefabab
```

Sample Output

```
13
```

Test Inputs

On the course homepage, I've provided four test inputs for this program. These can help you to make sure your program is doing the right thing. Here's what I got when I ran my solution on these inputs (implemented in C++):

```
eos$ ./compare < input_1.txt
13
eos$ ./compare < input_2.txt
20
eos$ ./compare < input_3.txt
23
eos$ ./compare < input_4.txt
556
```

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 whithout any change to the program.

If you've implemented your solution in Java or Python, you'll need to run it differently. Try something more like:

```
# for java
eos$ java compare < input_1.txt
13
# for python
eos$ python compare.py < input_1.txt
13
```

Analysis

In addition to your program, you are to write up an asymptotic analysis of its worst case running time. This will probably take you half a page. You should include enough detail to permit someone else to understand your approach even if they couldn't see a copy of your source code. Be sure to explain how you measure the input size. You have two input strings, so you'll probably want to use two different variables to describe the input size, one for each string. Describe how the running time of each stage of your algorithm and your data structures are dependent on input size. Combine all of the contributions to running time into a single formula and simplify it using big-O notation. Your argument can be informal, you're not expected to try to capture the exact run-time of your algorithm (like we did in class) or prove your big-O bound (like you're doing on Problem 2 above). Submit your analysis in a PDF file called `common.pdf`. (10 pts for the analysis)