Design and Analysis of Algorithms, Assignment 2

As usual, each part of this assignment is submitted as a separate file. As you finish up the parts, you can turn them in at `https://submit.ncsu.edu/`. Before the deadline, you can submit updated parts of any section whenever you want. They'll automatically replace your previous submission.

This assignment includes a coupole of programming problems and some problems where you get to write up a short answer, an argument or a plot. 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.

1. Substitution Method Proof (15 pts)

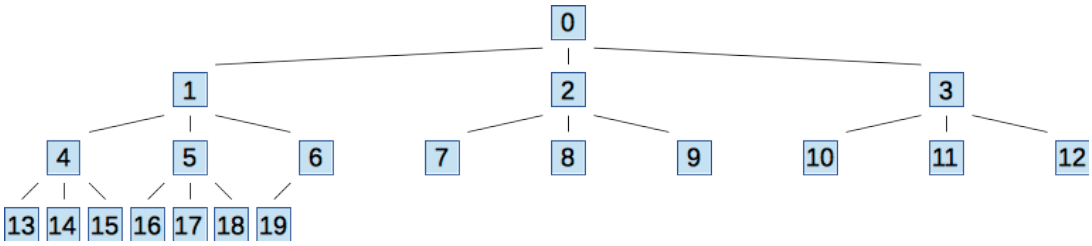
   Use the substitution method to show that the solution to the recurrence $T(n) = 5T(n/3) + \Theta(n^2)$ is $O(n^2)$. Try to give a carefully worded argument of your own. Your argument probably won't need to be more than half a page in length, but be sure to explain the steps of your agrument. I'd like to see a little more explaination than your book normally gives in substitution method arguments (but not too much more).

   Submit your argument as a file called `subst.pdf`.

2. Trinary Heap (25 pts)

   A binary heap is a really efficient data structure, partly because you can embed it in an array without storing or maintaining any of the pointers you'd normally use to build a tree.

   Of course, this doesn't only work for a binary heap. We can build a heap where every node can have three children instead of just two. This is illustrated below, with each node showing its position in the array. We still need to pack elements into the array consecutively, with a complete tree except for a last row that may be partially full from the left.



   You're going to write a program that uses a heap like this. The root node stored in element zero in an array, its three children will be stored in elements $1 \ldots 3$, and so on. You'll need to figure out functions that map from the index of a node to the index of its parent, and other functions that map from the index of a node to the index of each of its children. I'll tell you this is possible and not too difficult.

   Your heap will be a min-heap, stored in a resizable array (e.g., a vector in C++ or an ArrayList in Java). You'll implement two functions (methods) for manipulating the heap. An insert() function will add an arbitrary value to the heap, and a removeMin() function will remove the smallest value and return it.

   Using the heap and these two functions, you'll read a sequence of commands from standard input, one command per line. Each command will look like one of the following:

   - **add** v
     This inserts the given value, v to the heap, where v is a 32-bit, signed value.
   - **remove**
     This removes the minimum value from the heap and prints it on a line to standard output.

Commands will continue until the end-of-file on standard input. You won't have to worry about handling invalid commands.

Call your program heap3.c, heap3.cpp, heap3.java or heap3.py, depending on your implementation language.

Sample Execution

On the course homepage, I've provided three test input files and expected outputs (named things like input_h10.txt and expected_h10.txt). These can help you to make sure your program is behaving correctly. The way to use these files depends a little on your implementation language. In the examples below, I've included shell comments to explain what I'm doing. You don't actually need to enter these when you're trying these examples.

If you're programming in C or C++:

```
# Let the output go to standard output.
eos$ ./heap3 < input_h10.txt
15
35
# capture the output in a file
eos$ ./heap3 < input_h100.txt > output.txt
# see if it exactly matches the expected output.
eos$ diff output.txt expected_h100.txt
```

If you're programming in Java:

```
# Let the output go to standard output.
eos$ java heap3 < input_h10.txt
15
35
# capture the output in a file
eos$ java heap3 < input_h100.txt > output.txt
# see if it exactly matches the expected output.
eos$ diff output.txt expected_h100.txt
```

If you're programming in Python:

```
# Let the output go to standard output.
eos$ python heap3.py < input_h10.txt
15
35
# capture the output in a file
eos$ python heap3.py < input_h100.txt > output.txt
# see if it exactly matches the expected output.
eos$ diff output.txt expected_h100.txt
```

3. Trinary Heap Analysis (10 pts)

   Briefly analyze the running time for your heap's insert() and removeMin() operation. Write up your analysis and submit it in a file named `heap3.pdf`. As part of your analysis, answer the following questions for each operation.

(a) Is it asymptotically faster or slower than the same operation in a binary heap?

(b) Would you expect it to have a larger or a smaller constant factor than the same operation with a binary heap?

4. Quicksort Base Case Comparison (25 pts)

   In our presentation of quicksort, we assumed the base case was a one-element list. In practice, it may be faster to switch to a different sort when the list size gets small enough, without waiting until it reaches zero. You're going to write a quicksort program, qsort.c, qsort.cpp, qsort.java or qsort.py, to let us examine this.

   Your qsort program will read a sequence of 32-bit numbers form standard input. It will store them in an an arrow, sort them with quicksort, then print the sorted results to standard output, one value per line.

   Your program will take a command-line argument, k. This will tell it when to change sorting strategies. When recursion reaches a list size of k or smaller, your quicksort will just sort the list using insertion sort, bubble sort or selection sort (your choice), rather than continuing to quicksort it recursively. This will let us easily experiment with different trade-off points between quicksort and an asymptotically slower sort that might actually be faster for small enough lists.

Runtime Reporting

To get quicksort to take enough time to measure, we're going to need large input lists of numbers, a million items or more. It takes so long to read and print a list of this size, the total execution time of our programs won't be a good indication of how long they took to perform the sort (especially since all these time measurements will be a little noisy). So, we need to be able to measure the time for just part of our program. We'll do this by measuring the elapsed time in milliseconds and reporting it to standard error. Using stderr here is nice since it will let us send our programs sorted output to a file (or to /dev/null) and still be able to see the execution time on the terminal.

The way to measure execution time in milliseconds depends on your implementation language. Use code like the following to measure the runtime for just your quicksort and report it in milliseconds to standard error.

C or C++

We'll use the following function to capture the current time in milliseconds and return it in a (64-bit) long. You'll need to include the `sys/time.h` header to use this.

```
long getMilliseconds() {
  timeval tv;
  gettimeofday( &tv, NULL );
  long int ms = tv.tv_sec;
  ms = ms * 1000 + tv.tv_usec / 1000;
  return ms;
}
```

Then, using this function we can measure and report the time of any stage of our program using code like the following.

```
long t0 = getMilliseconds();
someOperationYouWantToTime();
long t1 = getMilliseconds();
fprintf( stderr, "%ld\n", t1 - t0 );
```

Java

It's easy to measure time in Java. For consistency with the other languages, we'll define the following simple method to give us the current time in milliseconds:

```
public static long getMilliseconds() {
  return System.currentTimeMillis();
}
```

Then, we can report the runtime for any operation using code like the following:

```
  long t0 = getMilliseconds();
  someOperationYouWantToTime();
  long t1 = getMilliseconds();
  System.err.println( t1 - t0 );
```

Python

In Python, the following function will return the current time in milliseconds. You'll need to import time to get this to work, and sys to get the next block of code to work.

```
def getMilliseconds():
    return int( round( time.time() * 1000 ) )
```

We can use this function in code like the following, to measure the runtime of any part of our program and report it to standard error.

```
t0 = getMilliseconds()
someOperationYouWantToTime()
t1 = getMilliseconds()
sys.stderr.write( "%d\n" % ( t1 - t0 ) )
```

Sample Execution

I'm providing a few sample inputs and expected outputs for your sorting program. For C and C++, you should be able to compile your program and run it like the following. As with the earlier examples, I'm using shell comments to explain what I'm doing. If you're programming in Java or Python, you can modify these examples (like the ones above), to try out your solution.

```
# Run the qsort on a the smallest input, switching to
# a naive sort when the list size is 2 or less.  The zero
# printed out at the start is the execution time in milliseconds
# (too small to register)
$ ./qsort 2 < input_s10.txt
```

```
0
3
4
12
27
52
62
65
68
80
95
# Here, I'm running the program on the largest sample input, with
# a transition to naive sort at 10 elements, sending the sorted
# output to a file, but letting the runtime go to the terminal.
$ ./qsort 10 < input_s1000000.txt > output.txt
88
# Capturing output lets us see if we got exactly the right output.
$ diff output.txt expected_s1000000.txt
```

5. Quicksort Performance Evaluation (10 pts)

   Let's see how changing the trade-off point affects quicksort. Use your quicksort program to sort the 1,000,000-element test case, with different base case sizes for switching to an $O(n^2)$ sort. Try with the each of the values 1, 3, 10, 30 and 100 for k (you can sample additional points if you'd like).

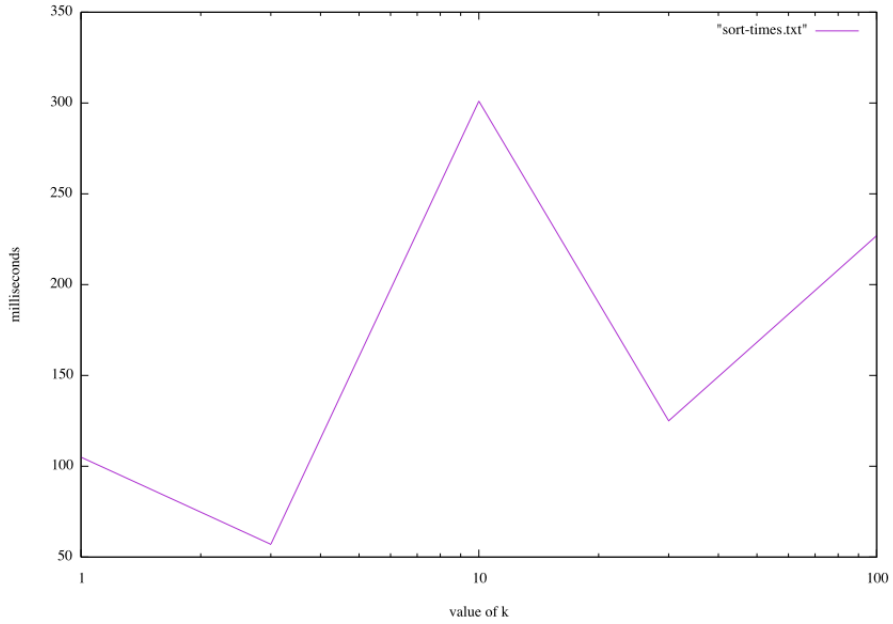
   For each k value, run the test five times and use the median runtime for your quicksort to generate your plot (see below). This will help to reduce the noise in your measurement. Be sure to run your tests on a non-shared machine (e.g., an EOS linux desktop system or a VCL image). That way, other load on the system won't affect your measurements as much.

   You'll have to sort the million-element list lots of times, but you can automate the execution of your program to make this easier. A loop like the following, written in shell, will run your program 5 times with a base case of 10, letting the sorted output go to `/dev/null` (just discarded) and letting the runtime go to the terminal. A little shell programming like this will let you run a bunch of tests in a row.

   ```
   for (( i = 0; i < 5; i++ ))
   do
     ./qsort 10 input_s1000000.txt >/dev/null
   done
   ```

   Generate a plot of our sort's runtime across this range k values (always using the 1,000,000-element input). Plot the sorting time in milliseconds on the Y axis and the value of k on the X axis. You can use a log scale on the X axis if you want (but it's not required). Call your plot `plot.pdf`.

   Your plot should look something like the following (but with different data; I just made up random values for this plot):



You can use any tool you want to create your plot. I used gnuplot, which should be available on the EOS Linux desktop machines. You could also use Microsoft Excel or LibreOffice Chart if you want. In your plot, you may not see too much variation in the runtime (especially for smaller k values) but you should see some.