COSC 320 - Advanced Data Structures and Algorithm Analysis
Lab 6

Dr. Joe Anderson

Due: 17 October 2019

1 Objectives

In this lab you will focus on the following objectives:

1. Experiment with randomized versions of standard algorithms
2. Develop experience with statistical data gathering and interpretation

2 Tasks

1. Put your code in a folder to be zipped and turned in at the end.
2. Write a routine called `shuffle` that generates an array of *all* numbers from 1 through n in a random order.
   (a) One way: for each element $i = 1$ to n, swap i with a random element from $\{i, i+1, \ldots, n\}$. This can be proven to yield a uniformly random permutation, assuming the random choice above is uniformly random, of course!
   (b) Another interesting way: use your priority queue from before, and insert each number with a random priority from 1 to n^3. Then simply call Extract-Max (or, equivalently, DeQueue) repeatedly to get the numbers out of the queue in a random order. Asymptotically, what is the difference between this and the above method?
3. **First:** Use your random permutation to simulate the hiring algorithm:
   (a) Shuffle the elements $\{1, 2, \ldots, n\}$ and interpret them as the relative ranks of the candidate assistants.
   (b) Run the Hire-Assistant algorithm and count the number of times that the current assistant is replaced by a better one
   (c) Do the above experiment about 10 times for each different n and report the average.
4. **Second:** Augment your source code from Lab 1 to now implement *randomized Quicksort*. That is, leave your partition function as-is, but instead swap a random element to the partition location prior to calling it from the main `quicksort` routine.
5. Compile together previous labs to study the performance of the following sorting algorithms on large datasets, with the difference of *randomly shuffling* the input prior to running the algorithm. Use your `shuffle` routine.
   (a) Quicksort

(b) Randomized Quicksort

(c) MergeSort

(d) HeapSort

6. Time each of the above on common sets of data, and aggregate your observations in a table. Since the inputs are random, use at least 10 different iterations on each size input array and average the observed timing. Then use this data to answer the questions below.

   You may want to follow the general procedure of:

   (a) For $n = 10,000$ to $1,000,000$, increasing by $50,000$ each time

      i. For $k = 1$ to 20:

         A. Use `shuffle` to generate an array of size n

         B. Time each sorting algorithm on (a copy of) that array

      ii. Average the times for the 20 trials to get an estimate of the expected running time for n elements of each algorithm

7. Include a `Makefile` to build your code.

8. Include a `README` file to document your code, any interesting design choices you made, and **answer the following questions completely and thoroughly**:

   (a) Summarize your approach to the problem, and how your code addresses the abstractions needed.

   (b) What is the theoretical time complexity of your algorithms (best and worst case), in terms of the input size? Be sure to vary the parameters enough to use the observations to answer the next questions!

      i. For example, if your algorithm takes time $T(n) = n^2$, and you double the input (e.g. 1000 to 2000), is the time now $T(2000) = 2000^2 = 2^2 * 1000^2 = 2^2 * T(1000) = T(2 * 1000)$?

      ii. Consider this mode of thought for other complexity classes? How much longer should it take to double the input? Does your experiment reflect this? If not, what could be the reasons that change the performance in practice?

   (c) How does the absolute timing of different algorithms scale with the input? Use the data collected to rectify this with the theoretical time complexity, e.g. what non-asymptotic function of n mostly closely matches the timings that you observe as n grows?

      i. Does the average number of assistants hired look close to the theoretical expectation?

      ii. For quicksort, does the number of swaps look like the expectation proved in class?

   (d) What, roughly, should the worst case running time look like (in seconds)? Estimate this from your observed time (e.g. after counting comparisons/swaps, calculate the time per swap and consider if there were the worst case number of swaps). Do the algorithms ever come close to incurring this worst case cost?

   (e) Use Mathematica or another program to plot your average timing data as a function of array size. Then try to find a function that smoothly approximates your data (e.g. some constant times $n \log n$).

   (f) Can you think of a way to judge the quality of your `shuffle` routine? Do you see ways it could be made to have a more random output?

   (g) How could the code be improved in terms of usability, efficiency, and robustness?

3 Submission

All submitted labs must compile with your provided `Makefile` and run on the COSC Linux environment.

Upload your project files to MyClasses in a single `.zip` file.

Turn in (stapled) printouts of your source code, properly commented and formatted with your name, course number, and complete description of the code.

Also turn in printouts reflecting several different runs of your program (you can copy/past from the terminal output window). Be sure to test different situations, show how the program handles erroneous input and different edge cases.