CSCI B505 Spring 20: Programming assignment 2

Due online: February 21, 11:59pm EST

Submit your work via Canvas, using "File upload". All work is strictly individual. If you have difficulties, please ask AI's for help during their office hours.

What to submit

The setup of this assignment is similar to Programming Assignment 1, so you will be able to reuse your code a lot.

For this assignment you will be submitting the following:

1. Source code. Please, follow good coding practices: use indentation, write comments, etc.
2. Write-up in PDF/Word format. **Handwriting is not allowed.** The file should contain:
   - Plots demonstrating performance of your code.
   - Justifications for any choices you've made.
   - Conclusions and analysis of the results.
3. Please submit tests which show correctness of your algorithm.
4. If you write a code for drawing the plots in a separate file, please submit this file too.

Insertion Sort vs. Randomized Quicksort vs. Dual-Pivot Randomized Quicksort

Implement Insertion Sort, Randomized QuickSort (as described in lectures and in CLRS, Chapter 7.3) and Dual-Pivot Randomized Quicksort (described below). You can use one of the following programming languages: C/C++, Java or Python. Please test your algorithms thoroughly.

Dual-Pivot Randomized Quicksort

Dual-Pivot Randomized Quicksort is similar to Randomized Quicksort. The main difference is that instead of selecting 1 random pivot which divides the array into 2 parts, it selects 2 random pivots which divide the array into 3 parts. You should divide the array into 3 parts using only 1 pass over the array in the following way.

Assume that we partition an array a from position ℓ (included) up to position r (excluded). Let our pivots be $k_1 = a[\ell]$ and $k_2 = a[\ell + 1]$ such that $k_1 < k_2$. Assume that we have processed all elements up to i excluded. During the algorithm execution, we maintain two indices s_1 and s_2 which split the processed elements by k_1 and k_2:

- For each $j : \ell + 2 \leq j < s_1$ we have $a[j] < k_1$.

- For each $j : s_1 \le j < s_2$ we have $k_1 \le a[j] < k_2$.
- For each $j : s_2 \le j < i$ we have $k_1 \le k_2 \le a[j]$.

We currently process an index i. Consider the following cases:

- $k_2 \le a[i]$ – $a[i]$ is in correct group ($\ge k_2$), so do nothing.
- $k_1 \le a[i] < k_2$ – swap $a[i]$ and $a[s_2]$, increase s_2 (similarly to QuickSort).
- $a[i] < k_1$ – the complicated case. We can't just swap $a[i]$ and $a[s_1]$: if $s_1 < s_2$, then $k_1 \le a[s_1] < k_2$, and after swapping $a[s_1]$ with $a[i]$ it would belong to group $\ge k_2$, destroying the invariant. Therefore, we first swap $a[i]$ and $a[s_2]$ and then swap $a[s_2]$ and $a[s_1]$. After that, we increase s_1 and s_2.

Input/Output:

Your sorting functions should accept an array of n numbers $x_1, x_2, \ldots, x_n$ (in this order). Each number will be an integer between 1 and n. Your function should sort the array in non-decreasing order. Note that you can sort the array in place, without creating a new array.

Plots

Your plots should show **average running times** of the sorting algorithms. For each of the first 5 input types below you should plot the average running time of each algorithm for inputs of size $n = 5000, 10000, 15000, \ldots$ up to 30000. For each data point on the plots, take the average running time of 3 runs (each time generating a new random input). Plot all three algorithms on the same chart so that it is easy to compare. For the last input type (small random inputs) see instructions below. **When measuring the running time you should only measure the time of sorting, not the time spent reading, generating or writing the data.**

Input/Plot 1: Large random inputs. Generate each x_i to be a uniformly random integer between 1 and n.

Input/Plot 2: Non-decreasing inputs. Generate each x_i to be a uniformly random integer between 1 and n and sort the resulting sequence in non-decreasing order ($x_1 \le x_2 \le \ldots \le x_n$).

Input/Plot 3: Non-increasing inputs. Generate each x_i to be a uniformly random integer between 1 and n and sort the resulting sequence in non-increasing order ($x_1 \ge x_2 \ge \ldots \ge x_n$).

Input/Plot 4: Noisy non-decreasing inputs. Generate input in two steps:

1. Generate input as in Plot 2.
2. Repeat the following 50 times: Pick two random integers i and j and exchange x_i and x_j.

Input/Plot 5: Constant-value input. Generate all x_i to be the same number (e.g. 1).

Input 6: Small random inputs. Generate 100,000 inputs as in Input/Plot 1 for $n = 50$ (i.e. you need to generate 100K sequences of random numbers between 1 and 50, of length 50 each). Measure the overall runtime of sorting these sequences. There is no plot for this type of input, just compare the two resulting runtimes.

Python: stack size manipulations

When writing recursive programs in Python, you can encounter the following limitations:

- Recursion limit – the largest possible number of nested recursion calls, which equals 1000 by default. Therefore, if you have more than 1000 nested calls, your program will fail with RecursionError: maximum recursion depth exceeded in comparison.

- Limited stack size – your program has limited stack size: if you try to create too many variables on a stack, the program will crash (without any chance of recovery). In my case, I get an error Process finished with exit code 139, yours may be different. In a sense, recursion limit is set to avoid this problem, to make the error recoverable.

In general, in Python, you should just avoid programs which use too much recursive calls, but it doesn't suit our goals. If you have a large recursion depth, you should increase both limits. Changing the recursion limit is simple: just add the following in the beginning of the file:

```
import sys

sys.setrecursionlimit(50000)
```

Now you can have 50000 recursive call, which is enough for our purposes.

Changing stack size is more challenging an system-dependent. If you are lucky, the following (in the beginning of the file) would work:

```
import resource

resource.setrlimit(resource.RLIMIT_STACK, (resource.RLIM_INFINITY, resource.RLIM_INFINITY))
```

If you get ValueError: current limit exceeds maximum limit, please use the following code instead. The code sets stack size for all new threads to be about 100MB and creates a new thread with this stack size.

```
import threading

def run():
    # your code here

threading.stack_size(int(1e8))
t = threading.Thread(target=run)
t.start()
t.join()
```

If still nothing works, just report that: don't show the corresponding points on the plots, and specify that for such points the recursion limit is exceeded.

Write-up

Explain your choices: Explain any platform/language choices that you made for your code and plots. How did you create and store the data you used to make the plots. Did you run into any difficulties or made any interesting observations?

Conclusions:

- The last two algorithms have asymptotic running time of $O(n \log n)$. Does the first plot reflect this? How do the three algorithms compare in terms of the running time?
- How about the second plot? Which algorithm do you think is the best one here?
- How does the third plot compare to the first and second?
- What about the fourth plot? What kind of functions do you think you're observing in the three plots (linear, logarithmic, quadratic, exponential, etc.)?
- Can you explain behavior for plot 5? Can you suggest a way to avoid such behavior which is sufficiently general?
- Which algorithm does perform better for input 6? Why?
- Would you use these algorithms for real data and if so why?