CSCI B505 SPRING 2020: Programming Assignment 2

Choice of programming language- I chose python for this programming assignment as it offers excellent readability, uncluttered syntax, and has extensive support libraries- I have used one such library called matplotlib to generate the plots in this assignment.

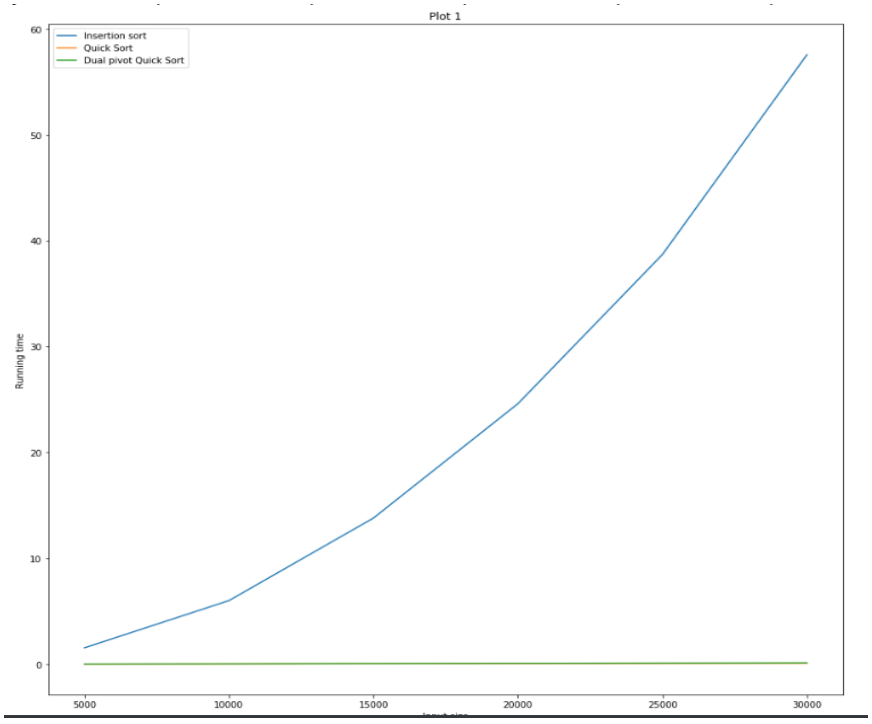
Resources: In this assignment I have implement three sorting algorithms- insertion sort, randomized Quicksort and dual-pivot randomized quick sort. Since, randomized quicksort and dual-pivot randomized quick sort are recursive sorting algorithms, I had to increase the increase the recursion limit and increase the increase the stack size using the following commands in python-

- import sys
  sys.setrecursionlimit (50000).
- import resource
  resource.setrlimit(resource.RLIMIT_STACK , (resource.RLIM_INFINITY, resource.RLIM_INFINITY))

Libraries used: As mentioned above I have used the pyplot module of matplotlib library to generate the plots in parts 1,2,3,4,5 of this assignment. Additionally, I have used the "random" standard library of python to generate inputs, and the "time" standard library to calculate the runtime of sorting algorithms.

Results and Interpretations:

PLOT 1 (LARGE RANDOM INPUTS):



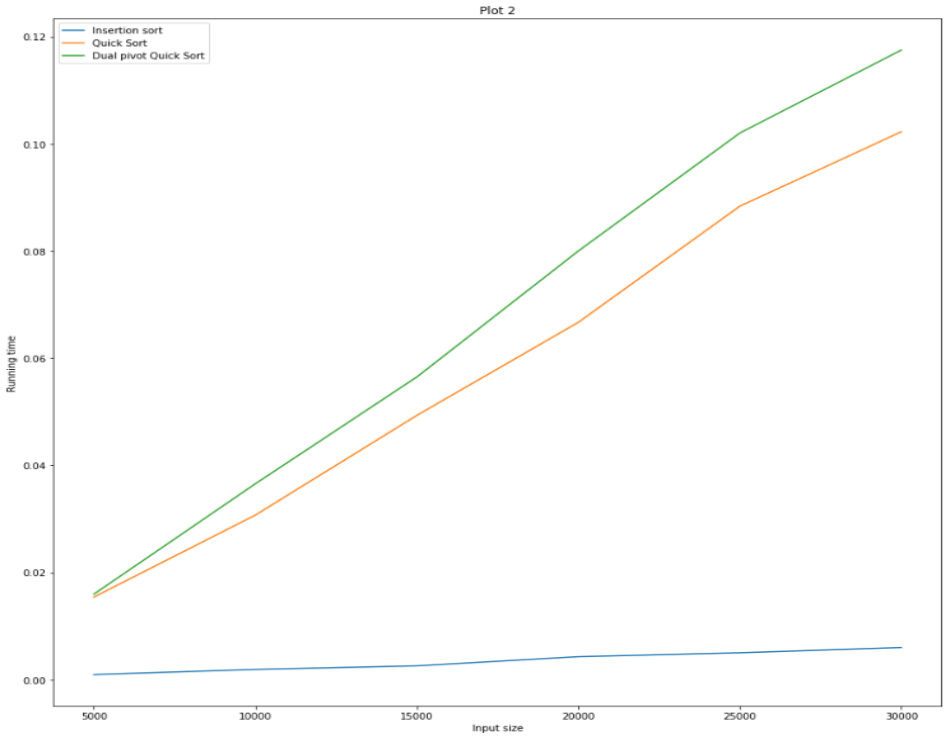
Plot: x-axis: Input Size, **y-axis:** Run-time in seconds.

Average run-times:

1. **Insertion sort:** [1.5470994313557942, 6.008790175120036, 13.80531652768453, 24.605288585027058, 38.6919781366984, 57.55440878868103]
2. **Randomized Quicksort:** [0.014706929524739584, 0.029263734817504883, 0.048584699630737305, 0.06692465146382649, 0.08086339632670085, 0.09826842943827312]
3. **Dual-pivot randomized Quicksort:** [0.01798701286315918, 0.03546015421549479, 0.05938903490702311, 0.08301480611165364, 0.10623757044474284, 0.1246643861134847]

Procedure: The plot above was generated by calculating the average run time of insertion, randomized quicksort, and dual-pivot randomized quicksort algorithms for three input lists of equal size, where the size varied from (5000,10000,15000…30,000), and the elements in each list were random integers between (1, sizeOf(list)).

PLOT 2 (Non-decreasing inputs):



Plot: x-axis: Input Size, **y-axis:** Run-time in seconds.

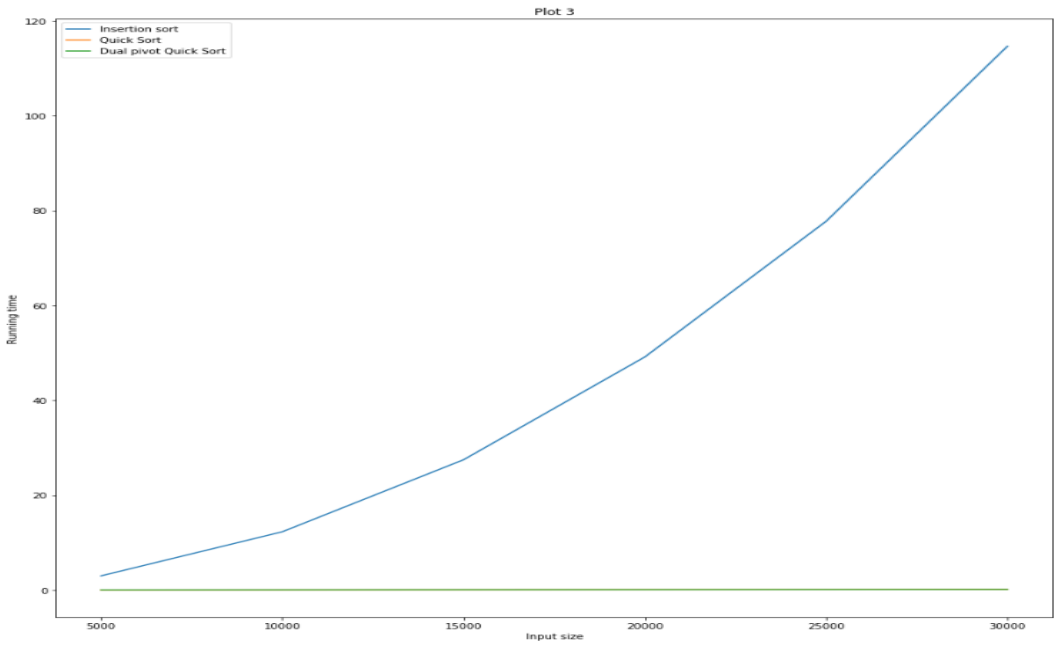
Average run-times:

1. **Insertion sort:** [0.000953515370686849, 0.0019048055013020833, 0.0025779406229654946, 0.004284222920735677, 0.005013227462768555, 0.006002108256022136]
2. **Randomized Quicksort:** [0.015417893727620443, 0.030721346537272137, 0.04932951927185058, 0.0666960875193278, 0.08836809794108073, 0.10225892066955566]

3. **Dual-pivot Randomized Quicksort:** [0.01596999168395996, 0.036568641662597656, 0.056499878565470375, 0.08001573880513509, 0.10202169418334961, 0.11751230557759602]

Procedure: The plot above was generated by calculating the average run time of insertion sort, randomized quick sort , and dual-pivot randomized quicksort algorithms for three input lists of equal sizes, where the size varied from (5000,10000,15000…30,000), and the elements in each list were random integers between (1, sizeOf(list)), and sorted in non-decreasing order.

PLOT 3 (Non-increasing inputs):



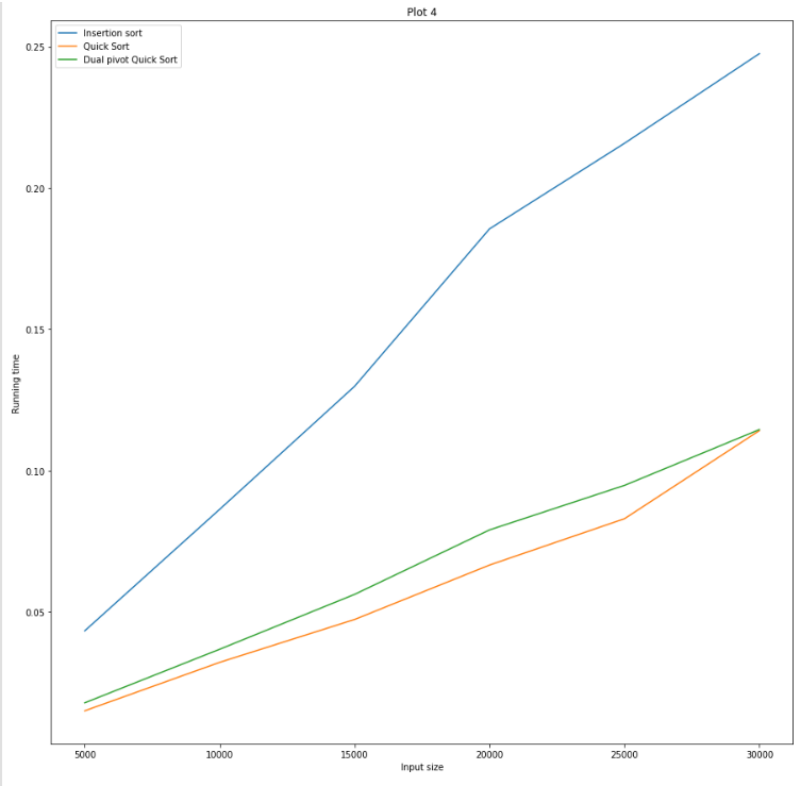
Plot: x-axis: Input Size, **y-axis:** Run-time in seconds.

Average run-times:

1. **Insertion sort:** [2.982149124145508, 12.27924648920695, 27.468767722447712, 49.14463019371033, 77.75112692515056, 114.65872168540955]
2. **Randomized Quicksort:** [0.014469305674235025, 0.031257311503092446, 0.04839650789896647, 0.06709671020507812, 0.0791155497233073, 0.10386983553568523]
3. **Dual-pivot randomized Quicksort:** [0.017745176951090496, 0.03603625297546387, 0.0582726796468099, 0.07118837038675944, 0.09167138735453288, 0.11963367462158203]

Procedure: The plot above was generated by calculating the average run time of insertion, randomized quicksort, and dual-pivot randomized quicksort algorithms for three input lists of equal size, where the-size varied from (5000,10000,15000…30,000) for each data point, and the elements in each list were random integers between (1, sizeOf(list)), and sorted in non-increasing order.

PLOT 4 (Noisy non-decreasing inputs):



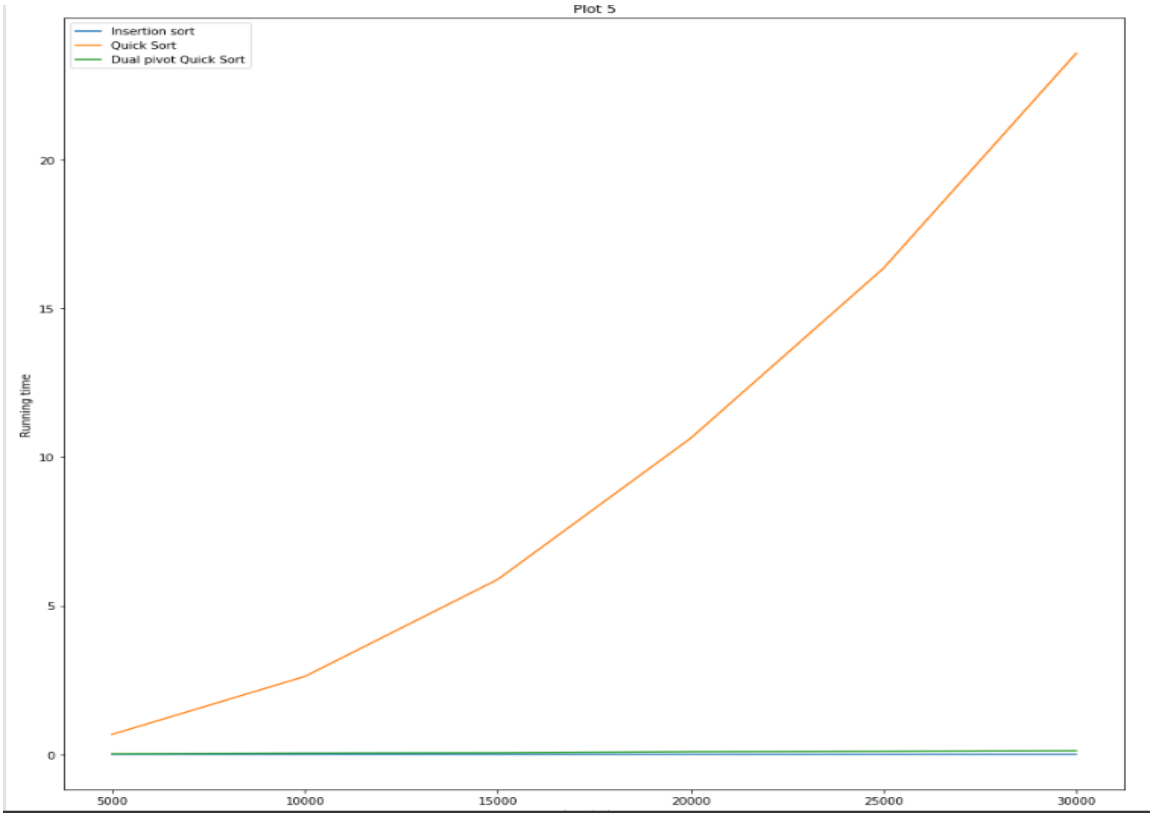
Plot: x-axis: Input Size, **y-axis:** Run-time in seconds.

Average run-times:

1. **Insertion sort:** [0.04340648651123047, 0.08637555440266927, 0.12987653414408365, 0.18554910024007162, 0.21579829851786295, 0.2475114663441976]
2. **Randomized Quicksort:** [0.01511073112487793, 0.03224349021911621, 0.04739960034688314, 0.06664880116780598, 0.08302632967631023, 0.11418747901916504]
3. **Dual-pivot randomized Quicksort:** [0.017932653427124023, 0.036877950032552086, 0.05628506342569987, 0.07907684644063313, 0.09481628735860188, 0.11455480257670085]

Procedure: The plot above was generated by calculating the average run time of insertion, randomized quicksort, and dual-pivot randomized quicksort algorithms for three input lists of equal sizes, where the size varied from (5000,10000,15000…30,000) for each datapoint, and the elements in each list were random integers between (1, sizeOf(list)), and sorted in non-decreasing order with some noise.

INPUT 5 (Constant-value input):



Plot: x-axis: Input Size, **y-axis:** Run-time in seconds.

Average run-times:

1. **Insertion sort:** [0.0009186267852783203, 0.0015954971313476562, 0.0023442904154459634, 0.0030652681986490884, 0.003945668538411458, 0.004757483800252278]
2. **Randomized Quicksort:** [0.6704974174499512, 2.620270570119222, 5.881783723831177, 10.626001199086508, 16.339920441309612, 23.57404287656148]
3. **Dual-pivot randomized Quicksort:** [0.020084619522094727, 0.04238152503967285, 0.05406483014424642, 0.08779303232828777, 0.10002549489339192, 0.11799764633178711]

Procedure: The plot above was generated by calculating the average run time of insertion, randomized quicksort, and dual-pivot randomized quicksort algorithms for three input lists of equal sizes, where the size varied from (5000,10000,15000…30,000) for each datapoint, and the elements in each list were constant values between (0,sizeOf(list)).

INPUT 6 (Small random inputs):

Results:

- Average time of insertion sort 0.00012380859136581422
- Average time of quick sort 8.845199584960937e-05
- Average time of dual pivot quick sort 0.000122624409198761

Procedure: The results above were generated by calculating the average run-time of insertion sort, quicksort, and dual-pivot randomized quicksort algorithms for a total of 100,000 arrays/lists of size 50, where the elements in each list were random integers between 1 and 50.

Conclusion:

1) **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?**

   Yes, by eyeballing the first plot we can observe the asymptotic running time(O(nlog(n))) of Randomized Quicksort and Dual-pivot Randomized Quicksort. The performance of insertion sort is significantly lower in comparison with both versions of Quicksort. However, between randomized quick-sort and dual-pivot randomized quick sort, there's a marginal difference in performance in terms of run-time.

2) **How about the second plot? Which algorithm do you think is the best one here?**

   The second plot was generated for input sequences in increasing order. This is one of the unfavorable cases for any implementation of Quicksort, as there are O(n^2) comparisons. Insertion sort is the best algorithm for input sequences in increasing order.

3) **How does the third plot compare to the first and second?**

   The third plot was generated for input sequences in decreasing order. This could possibly be the worst-case scenario for all the three algorithms. However, as can be seen in the graph, the performance of insertion sort is significantly lower (higher run-time) than both implementations of Quicksort- I suspect that it's due to the randomness in selection of pivots in both implementations of Quicksort. So, for non-increasing or decreasing inputs the performance of any implementation Quicksort is better than that of insertion sort.

4) **What about the fourth plot? What kind of functions do you think you're observing in the three plots? (linear, logarithmic, quadratic, exponential, etc.)?**

   The fourth plot was generated for input sequences in increasing order with some noise in between elements. So, intuitively, our algorithm should run close to complexities as observed in plot number 2. But, due to noise, Quicksort should perform better than plot 2, while insertion sort should perform worse than plot 2. The curves in this plot reflect that the algorithms have run-time in between linear and quadratic complexities.

5) Can you explain behavior for plot 5? Can you suggest a way to avoid such behavior which is sufficiently general?

The fifth plot was generated for input sequences with constant values, i.e., all the elements in the sequence are equal. By eyeballing the fifth plot we can clearly that quick sort has quadratic time complexity. If all the elements in a sequence are equal, then the sequence is already sorted, and we don't have to feed the sequence to a sorting algorithm. The algorithms implemented in this assignment do not take this property into count and start iterating through the input sequence just as any other sequence. We can check if all the elements in a sequence are equal in O(n) time by iterating through the entire (at-most) sequence once. In general, quicksort is expected to behave similarly when it encounters a group of similar elements in a sequence. In such scenarios we can enhance the performance by choosing the value that's repeating as a pivot and grouping them in the middle. Then we can recursively sort the elements in the left and right of this group- this is exactly what dual-pivot quick sort does, and hence it has a better performance over randomized quick sort.

6) Which algorithm does perform better for input 6? Why?

The quicksort algorithm performs better than insertion sort and dual-pivot insertion sort. I suspect it's because the size of the input sequence is small and range(max-min=50 at-most) of values in the sequence is marginal.

7) Would you use these algorithms for real data and if so why?

In general, quick-sort works better than most sorting algorithms, and if implemented carefully it's stable, has run-time complexity O(nlong(n)), and performs sorting in-place. However, quicksort executes longer for sorting sequences that are already sorted (rare case) or sequences that have a lot of identical elements in them. We can easily avoid such issues by adding conditional checks. So, I would use randomized quicksort sort for real data with a conditional check for input with constant values.