Project 1 report

Partner 1: Yuefan Yu (yy225)
Partner 2: Chentao Wang (cw373)
Date: 10-25-2018

My algorithm behaves as expected for both unsorted and sorted arrays. In each algorithm, I pass all the test cases. Generally, to the sorted arrays, Bubble sort and Insertion sort will be the fastest, since they only have O(n) runtime in this situation, then is the Merge Sort with the runtime O(nlogn), which is very stable, no matter whether the input array is sorted or not. Selection Sort is also always stable with the runtime O(n^2). Quick sort performs worst with a runtime O(n^2), and the coefficient of Quick Sort is larger than Selection Sort, so it is even slower.

To the unsorted arrays, Quick Sort is the fastest one and faster than Merge Sort, which is aligned with our knowledge. The reason for this is the average runtime of Quick Sort is O(nlogn), and the constant inside is much smaller than the one in Merge Sort, which has a stable runtime O(nlogn). Bubble Sort is the worst with runtime O(n^2) since it only swaps number between neighbors.

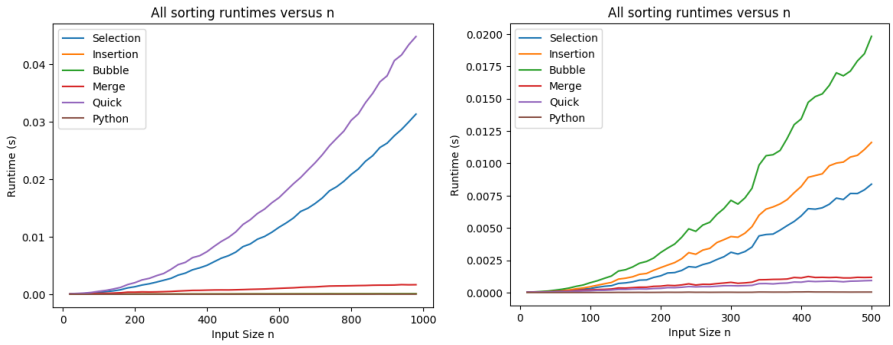

Figure 1. The performance of algorithms in different situation.

Left is inputting sorted array, right is unsorted array

To me, Quick Sort is the best and Bubble Sort is the worst with the consideration of performance and stability. About Quick Sort, although both the runtime of Quick Sort and Merge Sort are the fastest among the algorithms we have tested theoretically and practically (sometimes Quick Sort runs faster and sometimes Merge Sort runs faster in practical, especially merge sort runs faster when inputting sorted array), generally, the Quick Sort has the average runtime O(nlgn), and more likely faster than that, while Merger Sort is very stable, with the input size increasing, Quick Sort seems to have a better performance. Besides, Quick Sort is an in-place algorithm, while Merge Sort is not.

Quick Sort is easier to realize and has a less space complexity. About Bubble Sort, the runtime of it is usually the worst among all algorithms, it is inefficient because it only swaps two neighbors (although it is good when inputting sorted array, it is useless, who want to sort a sorted array?). In the meantime, it is not stable since the computation is too complicated (too many steps), it is easily affected by other factors like running another program in the meantime.

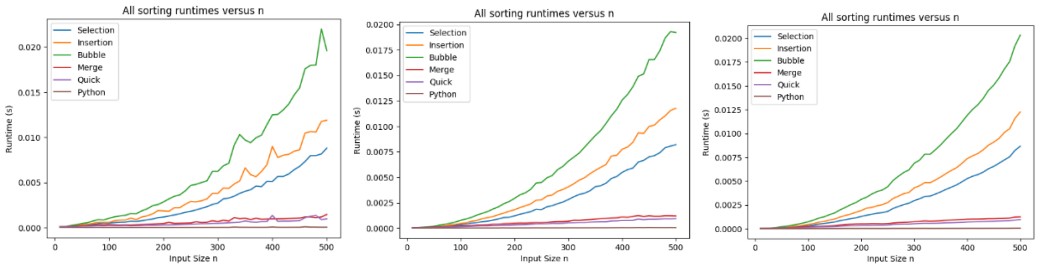

Figure 2 all sorting runtimes versus n with 1 trial (left), 30 trials(middle), 100 trials(right)

In my view, we report theoretical runtimes for asymptotically large values of n is because we want to get a general idea about the performance of each algorithm when handling with big data. With a small data, every algorithm is all right, they have the acceptable runtime. However, with a big data input, if n is 1 billion and the computer performs 1 billion operations per second, O(n^2) needs 31 years and O(nlogn) will finish in 30 seconds. Besides, with a larger input size, we are more likely to get a more accurate answer by average. With random input, the larger size will less likely to cause an extreme situation which is specific to each algorithm and cause it to run slower. Therefore, the larger the input size is, the more accurate and stable answer we will obtain. On the opposite, if we use small size n, the performance will vary sharply according to the input array. For instance, some algorithms will run faster with a sorted array than run with a descending array.

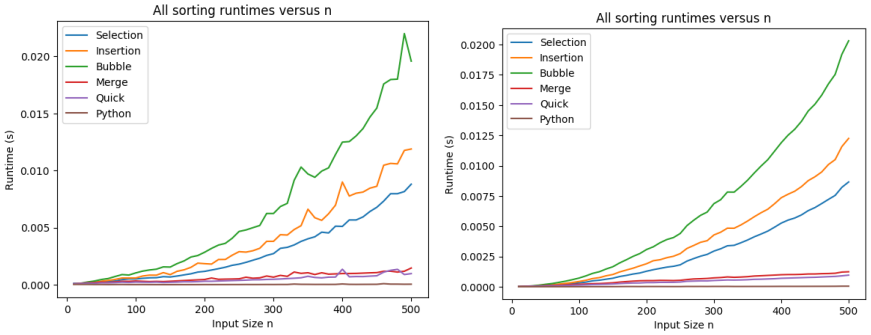

Figure 3. Measure performance with only 1 trial (left), with 100 trials (right)

Using multiple trials is also for accuracy. If there is only one trial like the left diagram, the runtime will differ largely since it is likely that we test the algorithm using only its best case or worst case, which cannot reflect its real performance, therefore, to eliminate such kind of instability, we should use multiple trials like the right diagram, from which we can get an accurate average runtime.

The performance of the computationally expensive task will be affected extremely if doing other things on the computer while operating that task. The more computationally expensive one task is, the more impact it will suffer. In my opinion, it is because our computer has a task schedule, it does context switch to let multi tasks run at the same time, so a computationally expensive task will experience more context switch during its operation, which causes it slower. Or computer will let more emergent task run first and then give a consecutive time to our algorithm to keep the performance, so we can observe that sometimes the algorithm runs much faster than expected.

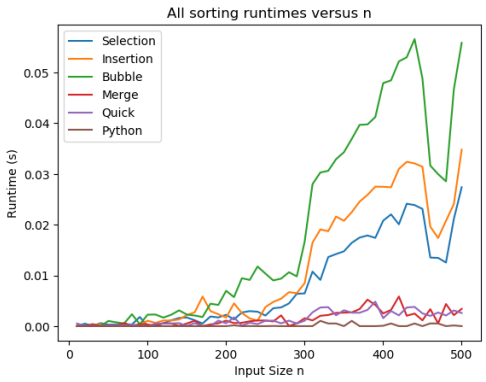

Figure 4. The computationally expensive tasks being affected by other progress

The reason of why do we analyze theoretical runtimes for algorithms is that actual runtimes vary from hardware to hardware or from system to system, like ‘Summit’ will run much faster than an 8-bit Soc when they are running the same algorithm. So, the idea here is to supply general information (standard) of algorithms, the theoretical runtime can let us measure the runtime of different algorithms using the same standard. Thus, we can make sure which one is better.

Therefore, theoretical runtimes provide more useful comparisons with the same standard to measure the performance of algorithms while experimental runtimes not only based on the performance of algorithms but also hardware and system performance.