Project 1: Sorting

ECE 590: Fall 2019

Technical Report

Xichen Tan xt24

Jianfei Bi jb577

10/27/2019

1 Implementation

1.1 Overview

Basically, we just follow the instruction mentioned in notes, and translate them into real code to implement these 5 sorting algorithms.

For the Selection Sort, Insertion Sort, and Bubble Sort, we simply follow the steps in notes exactly. For Merge Sort, we recursively split array into 2 halves until there is only one element left in each half. Then we recursively merge each half into one sorted array. For the Quick Sort, we consider that in real world practice, it's almost impossible to encounter some cases that the input is already sorted, and most of them are pretty random. Taking this into account, we simply choose the last element as pivot for each recursive call.

1.2 Performance Estimation

Assuming size of array to be sorted is n. In order to make the results more universal, we only compare the average running time for each sorting algorithm.

1.2.1 Selection Sort

For Selection Sort, we always need to find the smallest of the remaining unsorted elements, thus, the running time is guaranteed to be $O(n^2)$.

1.2.2 Insertion Sort

For each round in Insertion Sort, the thing needs to be done is to insert the current element into right position. Obviously, running time is O(1) if element is already in the right position and $O(k)$ if there are k elements ahead of current element in the worst case.

So, for the k_{th} loop, the average running time is $O(\frac{k}{2})$. Thus, the average asymptotic time Complexity is $O(\frac{n^2}{2})$, which is still $O(n^2)$. However, since each time we do insertion, we need to move every element between current position and inserting position back by one space, the running time could be slightly longer than Selection Sort.

1.2.3 Bubble Sort

The average case is that there are $\frac{n}{2}$ elements out of place, then we will need $\frac{n}{2}$ iterations to do swaps and to swap $n-k$ times at k_{th} iteration. So, the total cost should be $O(n^2)$. But because swaps need more write/read operations, Bubble Sort could be the slowest sorting algorithm.

1.2.4 Merge Sort

To estimate the running time of Merge Sort, we can divide merge sort process into two parts. One is splitting part and the other one is merging part.

For the splitting part, we need $\log n$ levels to split total array into pieces with a single element, and this can take $O(1+2+4+8+n/2) = O(n)$ work.

For the merging part, we need $\log n$ levels to merge pieces into a integrated sorted array and for each level, we need $O(n)$ work to combine to sorted smaller array. So, the total cost should be $O(n \log n)$. Then, we can compute the total cost of Merge Sort which is $O(n \log n + n) = O(n \log n)$.

1.2.5 Quick Sort

For the Quick Sort, the average (also the best) case is that each pivot we choose is in the middle position of the array to be sorted. There are $\log n$ recursive calls and each call takes $O(n)$ running time to scan through the whole array to do partition. Then, the total cost for Quick Sort should be $O(n \log n)$.

1.2.6 Conclusion

According to the estimation above, the best best sorting algorithm is Quick Sort (slightly faster than Merge Sort), and the worst one is Bubble Sort.

2 Results Analysis

2.1 Output Analysis

For unsorted and sorted arrays:

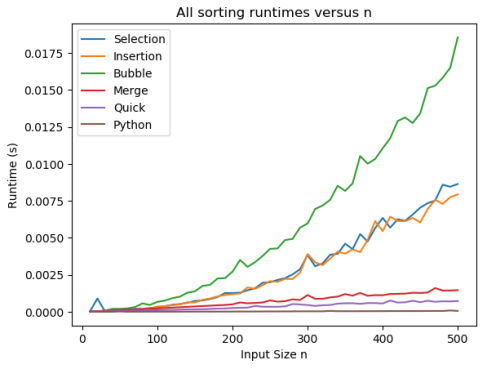

Figure 1: All sorting runtimes(unsorted) vs n

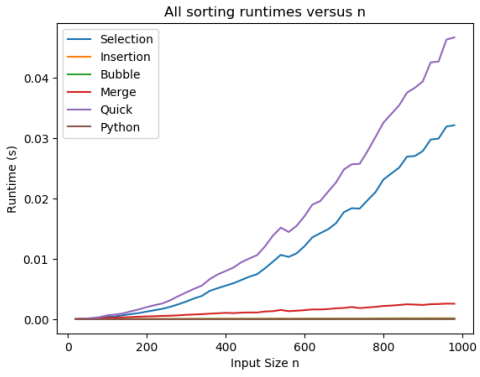

Figure 2: All sorting runtimes(sorted) vs n

We can see from above pictures that when arrays are unsorted, the result of sorting the algorithms from fast to slow is: Python < Quick < Merge < Selection < Insertion < Bubble, which is roughly the same as we estimate in section 1.2.

If arrays are not presorted, the result of sorting the algorithms from fast to slow is: Python < Insertion ≈ Bubble < Merge < Selection < Quick. For Quick Sort, we just simply scan through whole array for one time, the cost is $O(n)$. For Bubble Sort, we do a optimization for presorted arrays. We set a flag and set it to true first, and only when swap happens, we set it to true again, otherwise we set it to false and we stop looping. So, when the array is presorted, the running time is $O(n)$, which is nearly same as Insertion Sort. For Merge Sort, whether or not the array is sorted doesn't matter, the total cost is still $O(n \log n)$. For Selection Sort, the running of them are still $O(n^2)$. The slowest algorithm is Quick Sort, since in worst case, each pivot we choose is always the biggest one. Thus, we need to do n recursions and the total cost is $O(n^2)$. Meanwhile, Quick Sort needs some time to make recursive calls, which makes Quick Sort the slowest one(personal perspective).

To verify our estimation of time complexity, we can check the slope of running time versus array size on a log-log scale. Which as follows:

For sorted arrays:

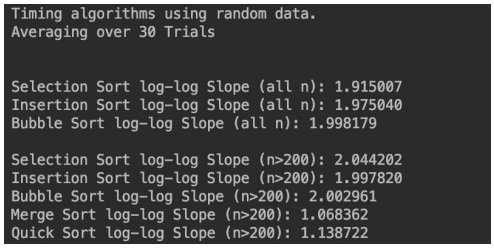

Figure 3: Slope for runtimes(unsorted) vs n

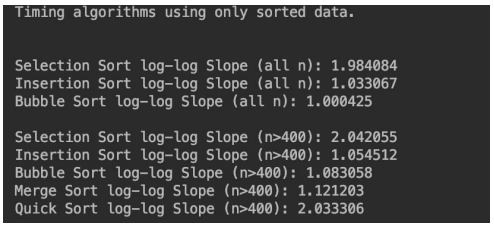

Figure 4: Slope for runtimes(sorted) vs n

We can see that for unsorted arrays, slope of Selection, Insertion, and Bubble Sort is around 2, and slop of Merge and Quick Sort is around 1, which is consistent with our expectation.

For presorted arrays, slope of Selection and Quick Sort is around 2, and slope of other Sorts is around 1, which is still consistent with our expectation.

2.2 Discussion on values of n

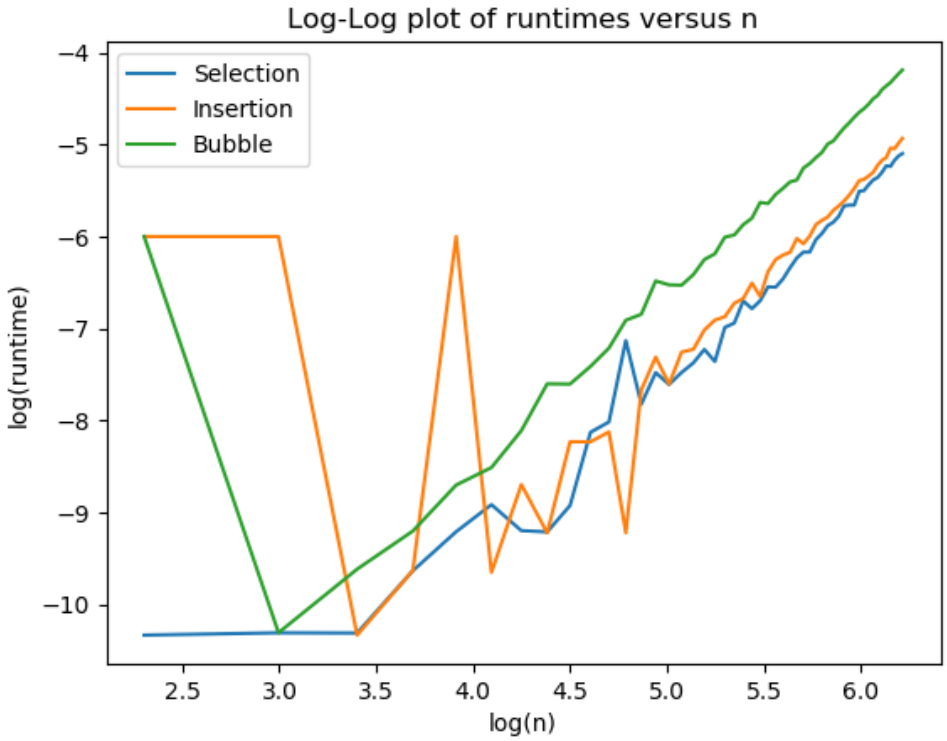

Figure 5: Log-log plot for runtimes(unsorted) vs n

We can see that when the n is relatively small, log-log plot for sorting algorithms has a very large vibration, which cannot reflect the real time complexity. However, when n gets larger, the lines tend to be smooth and the slope tends to a constant which is 2 here.

The reason for the vibration when n is small could be the random input. To tell in more details, when the n is small, time complexity can vary greatly due to the random generated arrays. Thus, the consequence is that log-log plot will not converge until value of n is large enough.

3 Number of Trials

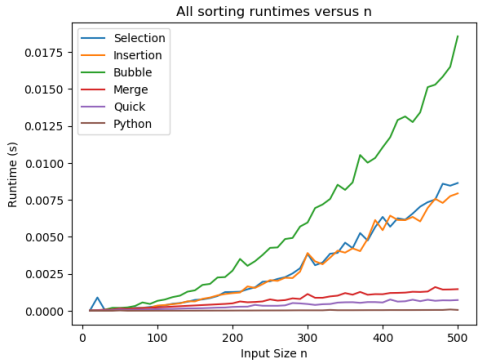

Figure 6: All sorting runtimes(unsorted) vs n

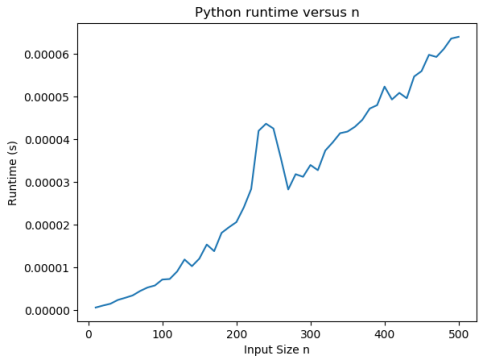

Figure 7: Python sorting runtimes(unsorted) vs n

While testing the performance of these sorting algorithms, we must average the runtime across multiple trials. This act serves to keep the occasionality at the lowest level. Inputs with different features may cause algorithms to yield dramatically different performances. Hence, the performance shown by one trial may be highly unreliable, as shown in the above graph. While over a decent number of trials, such as 30, the impact of outliers will be minimized, so that the results we get will be more precise.

4 Performing expensive tasks meanwhile

If I time the code while performing a computationally expensive task, in my case open several software at the same time in the background, there exists a rapid increase in the runtime. This is expected because as the computer allocates more computational resources for other tasks, python will get fewer resources and thus need more runtime to finish the task.

5 Conclusion

Analyzing theoretical runtimes for algorithms is essential. The computational power of computers can vary greatly, causing the actual runtime of one same algorithm on different machines to vary greatly. Analyzing theoretical runtime can set up a generalized uniform standard for us to evaluate the efficiency of an algorithm. Theoretical runtimes, namely big O notation, provides us a way to describe how fast the runtime will grow with the input size n increasing. In practice though, the actual runtimes of an algorithm depends much on the input size, so it is possible that an algorithm with a worse complexity has better performance on small inputs than an algorithm with better complexity.