Project 1 Sorting

Name: Qingyang Xu, Yue Yang   netID: qx37, yy258

1. Abstract

In this project, we design and implement five sorting algorithms to obtain an array of ascending order. According to the plot of runtime versus input size, the performances of different algorithms on both sorted and unsorted test cases are clearly displayed. Then we have a through discussion.

2. Experiment

Selection Sort algorithm repeatedly looks for the minimum element in the unsorted part and then move it to the end of the sorted portion in each iteration. The number of iterations equals the length of the array. For best case and worst case, the time complexity is: $n + (n-1) + \cdots + 2 + 1 \in O(n^2)$.

Insertion Sort algorithm inserts the first element of the unsorted component into sorted part, and keeps swapping it to the correct position in each iteration. For the best case, if the array is sorted, the step to move is 1 in each iteration, the time complexity is O(n). For the worst case and average case, the time complexity is: $1 + 2 + \cdots + (n-1) + (n-2) \in O(n^2)$.

Bubble Sort algorithm iterates through the array, compares every two adjacent elements, and swaps them if they are in wrong order. The kth largest element will be placed in the correct position at the kth iteration. For best case, if the array is sorted, the time complexity is O(n). For the worst and average case, the time complexity is $n + (n-1) + (n-2) + \cdots + 2 + 1 \in O(n^2)$.

Merge Sort algorithm recursively splits the array into two halves until the base case (one or two elements are left), then the sorting operations begin to merge the already sorted halves back until the complete array is merged. According to T(n)=2T(n/2)+O(n), the time complexity is $O(n \log n)$.

Quick Sort algorithm partitions the array based on a pivot until only one element left. It puts elements smaller than the pivot in front of the pivot and element larger than the pivot behind the pivot. For average and best case, according to T(n)=2T(n/2)+O(n), $T(n) \in O(n \log n)$. For worst case, pivot is the smallest or the largest one of all the elements in every partition, the time complexity is $O(n^2)$.

3. Discussion

3.1 Log-log Plots and Fitted Slope

Figure 1 and 2 show the log-log plots of runtime versus n. Figure 3 and 4 show the plots of runtime versus n on both unsorted and sorted data. Table 1 shows the fitted slopes.

Using the figures, we can first conclude the algorithms behave as expected. We can derive this conclusion by looking at the slopes of the fitted lines. When the data is unsorted, the theoretical runtime of Selection Sort, Insertion Sort and Bubble Sort is $O(n^2)$. By plotting log-log plots, we are supposed to get a straight line whose slope is 2. For Merge Sort and Quick Sort, the theoretical runtime is $O(nlogn)$, the slope of the fitted line is supposed to be 1.

When the data is sorted, the theoretical runtime of Selection Sort, Insertion Sort, Bubble Sort, Merge Sort and Quick Sort are $O(n^2)$, $O(n)$, $O(n)$, $O(nlogn)$ and $O(nlogn)$ respectively. The slopes of the fitted lines are supposed to be 2, 1, 1, 1 and 1. (For Quick Sort, we choose the element in the middle of the array as the pivot, so the runtime is still $O(nlogn)$ when given sorted data).

From Figure 1, Figure 2 and Table 1, we can know that the slopes of the fitted lines we get are the same as what they are supposed to be. The algorithms behave as expected.

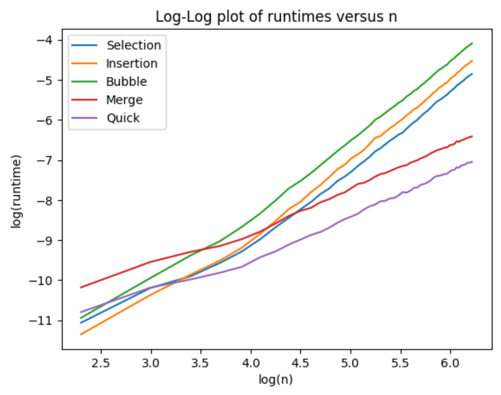

Figure 1. Log-log plot of runtime on the unsorted test cases

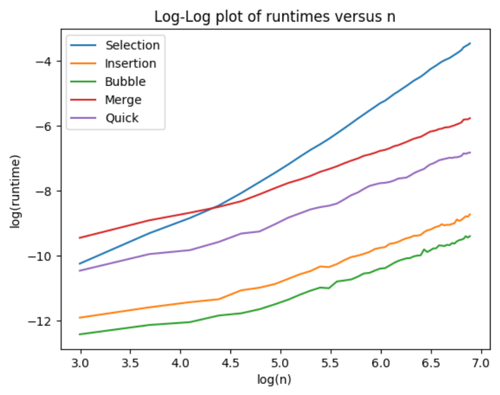

Figure 2. Log-log plot of runtime on the sorted test cases

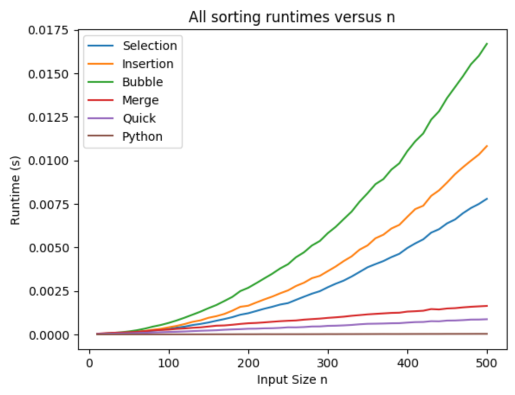

Figure 3. Runtime versus n on unsorted test cases

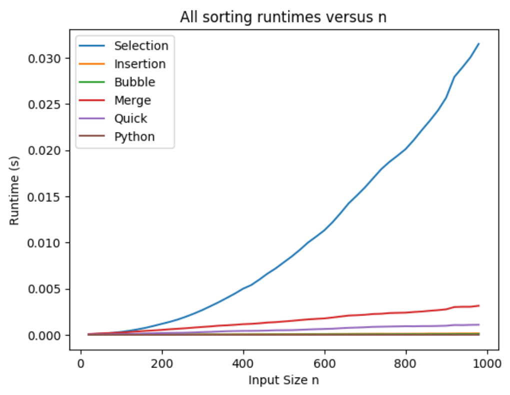

Figure 4. Runtime versus n on sorted test cases

Table 1. Slope of the fitted line

algorithm	unsorted data		sorted data	
	all n	n>200	all n	n>400
Selection Sort	1.749337	2.066442	1.886689	2.048392
Insertion Sort	1.872997	2.070640	0.923102	1.103314
Bubble Sort	1.862323	2.019204	0.907443	1.097577
Merge Sort	1.029884	1.067392	1.018970	1.121385
Quick Sort	1.038378	1.141602	1.042807	1.162206

3.2 Large N and Small N

From Table 1, we can conclude that the fitted line is more accurate when using only large n instead of using all n. Also, in Figure 1 and 2, when n is small, the slope of the fitted line is not as accurate as it is when n is large. To explain this phenomenon, we need to think about Big-O notation.

Big-O only considers what happens for large input sizes and ignores constant factors. This can be explained for two reasons. First, when n is small, the computer finishes the task in a really short time.

If the computer finishes a task in 100ns versus 50ns, then the difference doesn't matter. Second, when n is large, we can ignore the constant. If one function grows faster than another by more than a constant factor, then for an enough large n, the faster growing function will exceed the slower one—the gap increase as n grows. This asymptotic dominance occurs no matter what the constant factors are—the only difference they will make is where the faster growing one takes the slower one.

3.3 Actual Runtime and Theoretical Runtime

In this project, we get the actual runtime. The actual runtime is sensible to different processors, other computer programs that runs on the computer and so on. So, we need to average the runtime across multiple trials. If we only use one trial, the result we get will be very inaccurate.

In most cases, we analyze the theoretical runtime instead of the actual runtime. First, the actual runtime is sensible and not reliable. Second, when n is large or when the runtime increases very fast, it's impossible for us to get the actual time. Third, Big-O notation abstracts away unimportant distinctions caused by factors such as processors and goes to the key differences between algorithms.

However, there still are times when experimental runtimes are more useful. The first situation is when two algorithms have same theoretical runtime, we may want to compare their actual runtimes. The second situation is when n is small. At this time, constant matters. If algorithm A is $10000000n$ and algorithm B is $2n^2$. When n>5000000, we prefer A than B. However, when n=100, A requires less steps than B. In this situation, we need actual runtimes which tells us that B is better.

3.4 Runtime on sorted and unsorted data

For unsorted input arrays, as shown in both Figure 1 and Figure 3, Quick Sort can realize sorting with the shortest runtime among the five algorithms and Selection Sort is the most time-consuming one. The difference will be more obvious as the size of the input array becomes larger.

For sorted input arrays, as shown in both Figure 2 and Figure 4, Bubble Sort has the best performance while Selection Sort is still the worst one as the size becomes large.

For the overall performance, our Quick Sort algorithm displays great efficiency and works well in both unsorted and sorted cases. But it depends on the selection of pivot: here we choose the middle point of each partition as a pivot, so the time complexity of Quick Sort is still $O(n \log n)$. If the first or the last one of each partition is chosen as pivot, then the sorted input will become the worst cases so that the time complexity will become $O(n^2)$.

4. Conclusion

In this project, five sorting algorithms are implemented and the plots of runtime versus input size demonstrate their performances for different test cases. We deduce the time complexity before conducting experiments and then verify our theory analysis by calculating the slope of log-log plot for asymptotically large values of input size. On the basis of the experiment results, the algorithms do behave as expected. Besides, we discussed why we consider large n during analysis. What' more, we discussed when we need actual runtime and when we need theoretical runtime. In addition, we discuss about the best and worst algorithms separately. Our Quick Sort algorithm has the best and stable performance in both sorted and unsorted cases by choosing the middle point of each partition as pivot.