**Report for Project 1 – Sorting**

We have implemented five sorting algorithms including selection sort, insertion sort, bubble sort, merge sort and quick sort in python, got the log-log slope for those algorithms and some plots of runtime versus input size for those five sorting algorithms compared to the python built-in sort function.

Overall, these five sorting algorithms behave as expected for both unsorted and sorted inputs lists. It can be seen from the table below that, for random data, the first three sorting algorithms all have a log-log slope around two, which is consistent with their big-O time complexity of . Since selection sort has the same time complexity for all the cases, it is expectedly that the log-log slope for sorted data is also about two. While the log-log slope of insertion sort and bubble sort is nearly one for sorted data, because if the data are sorted, they will be in their best-case runtime, . Among these five algorithms, it is noticeable that merge sort has a log-log slope a little over one for both random and sorted data. It meets our expectations since merge sort has the same performance for all the cases, whose runtime is . Quick sort behaves similarly to merge sort when dealing with random data, while the log-log slope for sorted data is approximately two, which means it is in the worst case for this algorithm. It is not surprising, because our quick sort chooses the first element of list as the pivot value.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | random data | | sorted data | |
| all n | n>200 | all n | n>400 |
| Selection Sort | 1.757971 | 2.163471 | 1.941543 | 2.076367 |
| Insertion Sort | 1.917691 | 2.044615 | 1.031751 | 1.027307 |
| Bubble Sort | 2.006759 | 2.140757 | 1.030173 | 1.022562 |
| Merge Sort |  | 1.131128 |  | 1.110071 |
| Quick Sort |  | 1.213134 |  | 2.053756 |

Table: log-log Slope for different sorting algorithms (Averaging over 30 Trials)

In our view, quick sort is the best among these five sorting algorithms for three main reasons. The first reason is that it has the time complexity of for both best and average cases, which is better than the first three algorithms whose average runtime is. They will behave much slower than quick sort and merge sort if they are used to sort large lists, which will be common in practice. Another reason is that it is an in-place sorting algorithm. Although it seems that merge sort has the better performance for worst case than quick sort, merge sort is an out-of-place algorithm. It means that it would take much more memory than quick sort and negatively influence the runtime for especially large lists. It can be seen from the figure 1 below that for larger data, quick sort would perform better than merge sort. The last reason is that though it has the worst case of time complexity , it would not be a common case to sort a sorted data in practical use, especially for large lists. Therefore, quick sort would be a better choice when sorting data in practice as long as the data is not sorted, which is a rare situation.

Figure 1. All sorting runtimes VS input size for random data

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generatedChart, line chart

Description automatically generatedChart, line chart

Description automatically generated

* Do your algorithms behave as expected for both **unsorted** and **sorted** input arrays?

yes

* Which sorting algorithm was the best (in your opinion)? Which was the worst? Why do you think that is?

Best: in terms of time complexity, merge sort would be the best, because it’s O(nlogn) for best/average/worst case, while it is an out-of-place algorithm, which means it requires much space. For quick sort, it also takes O(nlogn) for best and average cases, but it is O(nlogn) for worst case if the pivot is not appropriately chosen.

Worst: selection sort, since the runtime will be same for both best and worst case.

* Why do we report **theoretical** runtimes for asymptotically **large values** of n?
* What happens to the runtime for **smaller values** of n? Why do you think this is?

Unstable

Smaller terms except asymptomatically significant part will also have an impact on runtime

* Why do we **average** the runtime across multiple trials? What happens if you use only one trial?

For one specific trial, it may be corner case.

* What happens if you time your code while performing a computationally expensive task in the background (i.e., opening an internet browser during execution)?

?

* Why do we analyze **theoretical runtimes** for algorithms instead of implementing them and reporting actual/experimental runtimes? Are there times when theoretical runtimes provide more useful comparisons? Are their times when **experimental** **runtimes** provide more useful comparisons?

Many other factors will impact the experimental runtimes, such as the computer’s … While theoretical runtimes only ideally consider the runtime for operations. When given the fixed environmental factors, experimental runtimes are more useful