CS350 Project

Empirical Analysis of Sorting Algorithms

Hsuan-Chih, Chen

Introduction

In this project, we introduce 7 different algorithms for sorting a list at first. These algorithms are Selection Sort, Insertion Sort, Merge Sort, Radix Sort, Quick Sort, Shell Sort and Tim Sort. We focus on how these algorithms work and their complexity in best, worst, and average case in the description section. After describing each algorithm, we have experimental section to show that how we test these 7 different sorting algorithms on different types of input data. Finally, we discuss experimental performances of these 7 different sorting algorithms whether the performances correspond to what we expect in the description section, and we conclude what we get and learn in this project.

Algorithm Description

Selection Sort

Description

The selection sort is similar to Brute Force algorithm. We traverse all the unsorted data and find the largest one, and we swap the largest item with the last item in the array. After that, we ignore the largest item in the last position. We traverse the rest part of the array again and find the second largest one in the array, and so on. It's a very inefficient sorting algorithm.

Analyzation

This way complexity is absolutely n^2 no matter what kind of input data we use. Even if there is a sorted or nearly sorted array, the comparison time(s) is same as the array with reverse or random data. Therefore, the worst case, the average case, and the best case are the same.

There is still an advantage in the selection sort. In comparison with Insertion Sort, we don't move data so much. We don't have to shift our data many times as the insertion sort while sorting. Instead, we swap the item to the correct place and don't swap that item anymore.

1. Input size: n 2. Basic operation: comparison 3. Complexity:

$C(n) = (n-1) + (n-2) + \ldots + 2 + 1 \rightarrow n \times (n-1)/2 \rightarrow \Theta(n^2)$

Insertion Sort

Description

Insertion Sort has two regions, a sorted region and unsorted region. At each step, we pick one item from the unsorted region, then insert it to the sorted region by comparing the items in the sorted region. We generally use the same way to make cards ordered when we are playing poker.

Analyzation

This algorithm can skip some comparisons when some items are already ordered. Once we find the correct position, we can stop comparing the rest of items in the sorted region. This is the reason that the complexity of the best case is O (n) in the insertion sort. If the list is sorted or nearly sorted, each unsorted item may just need few comparisons for inserting.

In comparison with the selection sort, we move data and swap them frequently. However, currently we just focus on basic operation, comparison, so we don't consider the issue of swapping data.

1. Input size: n. 2. Basic operation: comparison 3. Complexity:

Worst case: reverse list--> C (n) = 1+2+3+…+ (n-2) + (n-1) → n× (n-1)/2→ O (n^2)

Best case: sorted list→ O (n)

Radix Sort

Description

Radix sorting algorithm is forming groups and then combining them for sorting data. It's similar to the bucket sort. However, we just need to create an array which the size is 10 (for integer inputs) for grouping them repeatedly.

Integer example: 195, 203, 8, 11, 55, 6

1<u>1</u>, 20<u>3</u>, 19<u>5</u>, 5<u>5</u>, <u>6</u>, <u>8</u> Compare the units digit of the all numbers

2<u>0</u>3, 6, 8, <u>1</u>1, <u>5</u>5, 1<u>9</u>5 Compare the tens digit of the all numbers

6, 8, 11, 55, <u>1</u>95, <u>2</u>03 Compare the hundreds digit of the all numbers

By using mod 10 and then divide the number by 1, we can make all the units digits ordered. After that, we use mod 100 and then divide the number by 10 to make all the tens digits

ordered, and so on. We finally can get an ordered list by grouping repeatedly. Besides, we also can use the same concept to sort words alphabetically. (ABC, PDX, AAC, SFC…).

Analyzation

The worst case of time complexity: O (w × n)

The worst case of space complexity: O (w + n)

w : the length of the longest integer or word (ex: 203 is 3. PDX is 3)

n: input size(ex: the number of the total integers)

Merge Sort

Description

Merge Sort is a divide and conquer sorting algorithm. We divide unsorted array by 2 and pass this two data into two recursive calls individually. When the recursive calls get the stop condition which only one item left, we start to return and do merging. At each step, we can get two arrays from the two current recursive calls. We compare this two arrays and create a temporary array to store the result of sorting and merging the two arrays, and then we return the temporary and sorted array to the prior call. By this way, we can get the whole sorted array finally. On the other hand, we need to have a temporary array of equal size to the original array. It's probably a problem of the merge sort if the memory is limited.

Analyzation

In the merge sort, we use 2 recursive calls and n-1 (n/2+n/2-1) comparisons at each step. Additionally, no matter what kind of inputs we are supposed to use n-1 comparisons, O (n), to do merging at each step. The number of n is different in different recursive call because we divide n by 2 at each step. Therefore, we can derive the complexity is: T (n) = 2T (n/2) + n. By Master Theorem, we can get the complexity, O (n × log n), in both worst and average cases.

Shell Sort

Description

The shell sort is to compare items that are farther apart and resort a number times. For resorting part, we use "increments" to compares sets of data in the list. Form larger increments to smaller increment, we can choose any increment such as 6->4->2->1 or 7->5->3->1. Finally,

we can get a partially sorted list, and then we use insertion sort to sort this list to get the completely sorted list.

For example, 29 11 13 38 16 5 2 10

1. Sort every 5 numbers: 29 <->5, 11 <->2, 13<->10 (the first increment)

   (a<->b: If a > b, then do swapping)

   ➔ 5, 2, 10, 38, 16, 29, 11, 13
2. Sort every 3 numbers: 5 <-> 38<-> 11, 2<->16 <-> 13, 10<-> 29 (the second increment)
   ➔ 5, 2, 10, 11, 13, 29, 38, 16 (The list is nearly sorted now)
3. Do the insertion sort

Analyzation

In the shell sort, we want to get a nearly sorted list first and then do insertion sort. The reason is that the complexity of the base case is O (n) in the insertion sort. Although we cannot get O (n) in the base case by the shell sort, we can get O ($n \times \log_2 n$) in the best case. The worst case is O (n^2) as the worst case of bubble sort. The worst case of Shell sort depends on gap sequence. Furthermore, in the increment part, a power of 2 is not a good choice for the gaps because we will compare the same items on multiple passes. We should get more different combinations each time.

Quick Sort

Description

Quick Sort is also a divide and conquer sorting algorithm. The idea is that we choose a pivot point (an item), and we partition a list of data by this pivot. At each recursive step, we choose the pivot randomly or strategically. We put the items larger than the pivot on the right side of the pivot, and we put the smaller ones on the left side of the pivot. After that, we use two recursive calls and pass the two sides of the array individually. Once we get the stop condition the range of the array is 1, we also get the array already sorted.

Analyzation

The quick sort and the merge sort both use divide and conquer sorting algorithm. However, the merge sort sorts the list after its recursive calls. The quick sort oppositely sorts the list before its recursive calls. Besides, the quick sort doesn't need a temporary array for sorting. It directly alters the arrangement of the array.

When the pivot's value is the largest or smallest, one of sides (left or right) will become empty. We just can decrease size by only 1 (pivot), and we do size-1 comparisons. Therefore, we will have a lot of useless comparisons. The worst case is as the insertion sort's worst case, O (n^2). The best and average case are O ($n \times \log_2 n$). By using Master theorem, we can derive this result from T (n) = 2T (n/2) + O (n).

Tim Sort

Description

Tim Sort is a standard sorting algorithm in Python. It is a hybrid stable sorting algorithm. It derived from merge sort and insertion sort, and it has good performances on many different types of data in real word.

Complexity→ The best case: O (n) The worse and average case: O ($n \times \log n$)

Experimental Procedure

Programming Language: Python

For making different sorting algorithms run the same input data, I use Python's list( ) function to make a deep copy of my original list. At run time, we always copy the original list first and pass the copy list into each sorting function.

1. Pass the same random list which the size is 10000 into the 7 different sorting functions.
2. Use a size 100000 random list and then run the program again.
3. Use a size 1000000 random list and then run the program again.
4. Record all results of the sorting time.
5. Change a random list to be a sorted list and do step 1-4.
6. Change a sorted list and do step 1-4.

Finally, we put the results of the three different types of the input data which their size are all 1000000 together. We should compare and discuss the best and the worst case of the complexity among the three types of the input data.

Experimental Results

Input data : Random List Input size = n Time Unit: seconds							
Size of n	Selection	Insertion	Merge	Radix	Quick	Shell	Tim
10000	5.001	8.704	0.069	0.021	0.037	0.102	0.0039
100000	long time	long time	0.739	0.21	0.5	1.2	0.04
1000000	long time	long time	9.176	3.598	6.678	17.294	0.68

By inputting sorting the random list, the selection sort and the insertion sort are much slower than other sorting algorithms when the input size increase to 100 thousands or 1 million. These performances are already too bad (Over 250 seconds) so I finally put long time on my records. It shows that the O (n^2), worst case of the selection sort and insertion sort, is already too slow for sorting.

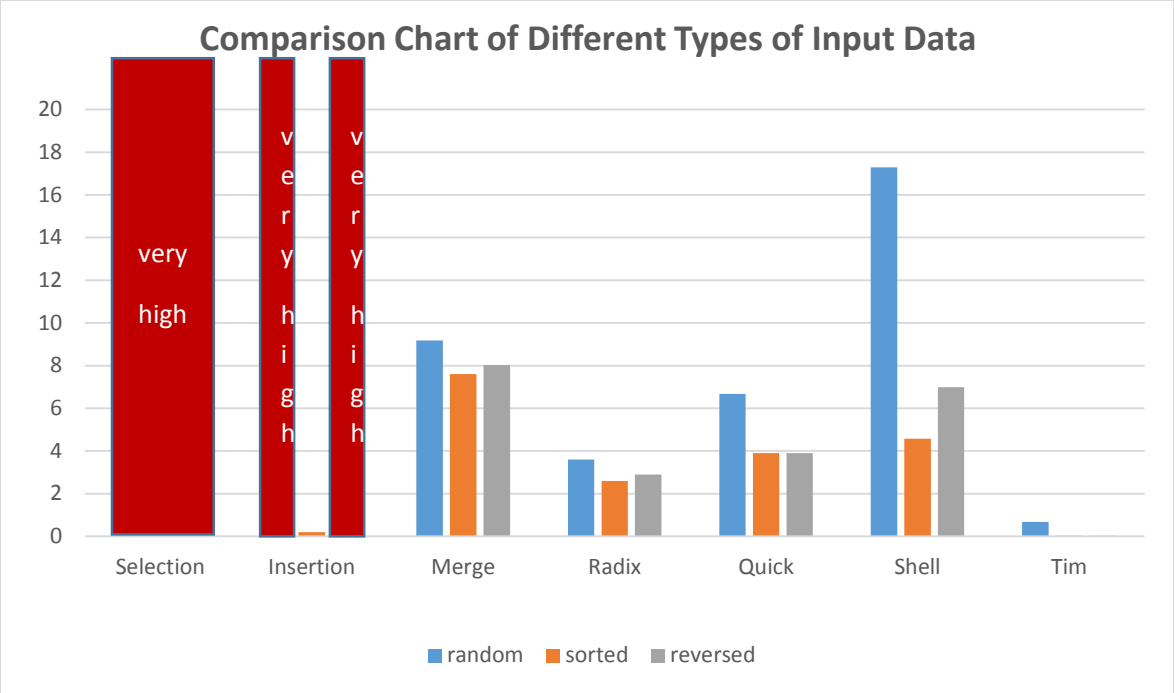
Input data : Sorted List Input size = n Time unit: seconds							
Size of n	Selection	Insertion	Merge	Radix	Quick	Shell	Tim
10000	5.204	0.003	0.1	0.015	0.029	0.03	0.002
100000	long time	0.018	0.678	0.2	0.335	0.404	0.006
1000000	long time	0.195	7.608	2.598	3.892	4.578	0.026

By inputting the sorted list, the Insertion sort obviously has a very good performance. This result prove that the complexity of insertion sort's best case is O (n) and it's really fast. It's much faster than other sorting algorithms besides the python's Tim sort.

Input data : Reversed list Input size = n Time unit: seconds							
Size of n	Selection	Insertion	Merge	Radix	Quick	Shell	Tim
10000	5.017	16.825	0.097	0.0169	0.029	0.044	0.001
100000	long time	long time	0.704	0.217	0.337	0.6	0.003
1000000	long time	long time	8.017	2.894	3.895	6.992	0.028

By inputting the reversed list, selection sort and insertion sort are still too slow, and Other sorting algorithms has better performance than I expected. I will mention more details about the results of the reverse list in the next section.

Comparison of Different Types of Input Data Input size = 1000000 Time unit: seconds							
Type	Selection	Insertion	Merge	Radix	Quick	Shell	Tim
Random	long time	long time	9.176	3.598	6.678	17.294	0.68
Sorted	long time	0.195	7.608	2.598	3.892	4.578	0.026
Reversed	long time	long time	8.017	2.894	3.895	6.992	0.028



Discussion

For the selection sort, the results correspond what we learn. No matter what kind of the input data is, its complexity is Θ (n^2) which is very slow and not a useful sorting algorithm.
For the insertion sort, the results also apparently shows that the insertion sort are good at sorting sorted or nearly sorted list. Its complexity O(n) is much faster than other algorithms which are O ($n \times \log_2 n$) and O ($w \times n$). Besides, the Tim sort's best case can be O (n). It has even faster performance than the insertion sort. The reason probably is that it's derived from merge sort and insertion sort. It optimizes insertion sort so it can has more powerful and efficient performance.

For the merge sort, the merge sort's best case and worst case are both O ($n \times \log n$). The results also shows that it has stable and not so different performances among the different types of the input data. As I mentioned in the algorithm description section, no matter what kind of data it all use n-1 comparisons while merging. Moreover, although the merge sort and the quick sort are both divide and conquer sorting algorithm, the quick sort has better performances than the merge sort on the different types of the input data.

For the radix sort, the radix sort has good performances at each case. Its worst case and best case are O ($w \times n$). The range of the input value is from 0 to 1000000 so the w is supposed to be 7. I think that's why the radix sort can have better performance than Merge, quick, and shell sort which are O ($n \times \log n$).

For the quick sort, the worst case of quick sort is O (n^2) and the best and average case is O (n × log n). The results shows that the random list spend more time than sorted and reversed list. When it choose many inefficient pivots, it will has worse performance. The choosing pivot function in the quick sort might be the reason. If the choosing pivot function chooses the median of the array each time, we can decrease the size of the sorted and the reversed list quickly, and it won't have a lot of useless comparisons in the two cases. Therefore, I think the sorted and reversed cases are idea inputs, and the random case is a normal and realistic input for this quick sorting algorithm.

For the shell sort, the results shows that the shell sort is a not bad algorithm. In the random case, it spend apparently more time than other algorithms out of the selection and insertion sort. Its worst case is O (n^2), and its best case is O (n × log n). The average case depends on gap sequence we choose. By the results, we can know the gap sequence we chose are not very efficient to this random input. It is acceptable but not ideal.

For the Tim sort, it has the best performance in each cases. It uses effective and complicated ways to combine the insertion sort and the merge sort concepts and then optimizes them strongly for getting power performance.

Conclusions

In this project, we not only realize how these 7 algorithms work but the complexity analyzation is very crucial for each algorithm. After doing the experiments, we find that the experimental results are actually very close to what we learn in the algorithm analyzation. By the effective algorithm analyzations, we can reasonably explain why each algorithm has efficient or inefficient results on different input data. We understand what happen in algorithms such that what make it slower or faster. Furthermore, we can predict different algorithms' performances on different types of data before running them, and choose the best algorithm for different types of input data in the real word.