CZ2001 EXAMPLE CLASS 3_GROUP 2

An Zheyuan , Li Zhaochen , Addiennur Hamizah

PROJECT 3A: Empirical Comparison between Insertion Sort and Mergesort

Insertion sort is a simple sorting algorithm that builds a final sorted list from an original unsorted list one item at a time using the incremental approach. Mergesort on the other hand, uses a Divide and Conquer approach. It divides a list into two halves of approximately equal sizes recursively until the sub-list contains single element. Then, it recursively merges two sorted sub-lists into one sorted list. For this project, we aim to perform empirical comparison of time efficiency between Insertion Sort and Mergesort.

1. Algorithm Implementation

We implemented the two sorting algorithms; Mergesort and Insertion Sort in Python.
The function insSort takes an unsorted list 'a' as the only parameter , sorts the list in-place and returns the total number of comparison upon completion of sorting. The code is as shown below:

```
############### INSERTION SORT ################
def insSort(a):
    numComp=0
    for i in range(0, len(a)):
        for j in range(i, 0, -1):
            numComp += 1
            if a[j] < a[j-1]:
                temp = a[j]
                a[j] = a[j-1]
                a[j-1] = temp
            else:
                break
    return numComp
```

The recursive function mergeSort takes three parameters: the unsorted list, first index and last index of the portion to be sorted. It divides the list into two sublists of equal length and recursively calls itself on each sublist. In each recursive call, mergeSort calls the merge function on the unsorted list, and finally returns the total number of comparison to sort the list.

The merge function takes four parameters: an unsorted list 'a', first index, middle index and last index of the portion to be sorted. It assumes that the unsorted list is divided into two sublists, each of which is sorted, separated by the middle index. It merges the two sublists into one sorted list in-place using a two-pointer approach. It returns the total number of comparison between list elements carried out when merging the two sublists. The code is as shown below:

```
################## MERGESORT ##################
def merge(a, lo, mid, hi):
    num_comp = 0
    # a[lo..mid] and a[mid+1..hi] are sorted

    # two pointers that point to the 1st element in each sorted part
    i = lo
    j = mid + 1

    # copy the original array into an auxiliary array
    aux = a.copy()

    # repeat for each slot in a[lo..hi]
    for k in range(lo, hi + 1):
        # all elements in the lower half has been merged
        if i > mid:
            a[k] = aux[j]
            j += 1
        # all elements in the upper half has been merged
        elif j > hi:
            a[k] = aux[i]
            i += 1
        # first element in the lower half is smaller than that in the upper half
        elif aux[i] < aux[j]:
            a[k] = aux[i]
            i += 1
            num_comp += 1
        else:
            a[k] = aux[j]
            j += 1
            num_comp += 1
    return num_comp

def mergeSort(a, lo, hi):
    first_comp = 0
    second_comp = 0
    if hi <= lo:
        return 0
    mid = lo + int((hi-lo) / 2)
    if hi-lo > 1:
        first_comp = mergeSort(a, lo, mid)
        second_comp = mergeSort(a, mid+1, hi)
    total = merge(a, lo, mid, hi) + first_comp + second_comp
    return total
```

2. Generating Input Data based on Random Integers

Before implementing the two sorting algorithms for comparison, we generated lists of sizes 5 000, 10 000, 20 000 and 30 000 respectively with the following types of data for each of the sizes. *(1) Integers 1, 2, …, n sorted in ascending order. (2) Integers n, n−1, …,1 sorted in descending order and (3) Randomly generated datasets of integers in the range [1 … n]*. We then run the Insertion Sort and Mergesort algorithms on the generated datasets and printed the Number of key comparisons and CPU times for each sorting algorithms.

3. Measuring Time Complexity

The statistical results for the above mentioned information are presented in table form below for datasets generated in Ascending, Descending and Random orders.

Ascending Order

Number of Comparisons		Input Size			
		5,000	10,000	20,000	30,000
Sorting algorithm	Insertion Sort	4,999	9,999	19,999	2,9999
	Mergesort	32,004	69,008	148,016	227,728

CPU time(s)		Input Size			
		5,000	10,000	20,000	30,000
Sorting algorithm	Insertion Sort	0.00347	0.00548	0.010934	0.01739
	Mergesort	0.10465	0.40473	1.64772	3.47796

Descending Order

Number of Comparisons		Input Size			
		5,000	10,000	20,000	30,000
Sorting algorithm	Insertion Sort	12,497,500	49,995,000	199,990,000	449,985,000
	Mergesort	29,804	64,608	139,216	219,504

CPU time(s)		Input Size			
		5,000	10,000	20,000	30,000
Sorting algorithm	Insertion Sort	4.17883	17.51924	73.59114	153.94893
	Mergesort	0.14731	0.39481	1.56488	3.58361

Random Order

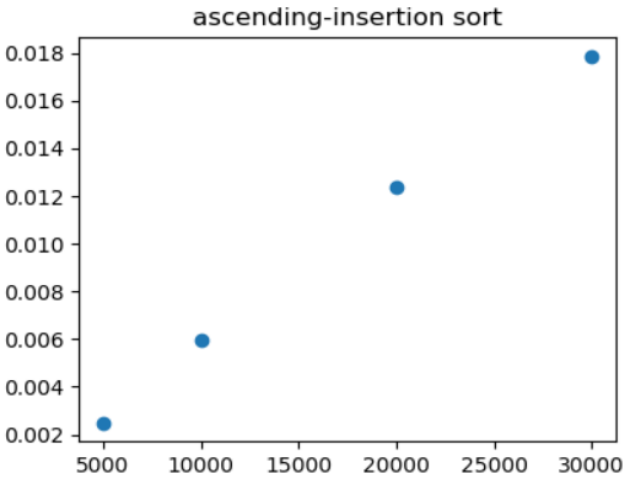
Number of Comparisons		Input Size			
		5,000	10,000	20,000	30,000
Sorting algorithm	Insertion Sort	6,167,477	24,932,923	99,839,695	224,893,606
	Mergesort	55,159	120,467	260,891	408,512

CPU time(s)		Input Size			
		5,000	10,000	20,000	30,000
Sorting algorithm	Insertion Sort	2.12931	8.42556	34.13130	79.66763
	Mergesort	0.12896	0.58825	2.22604	4.91288

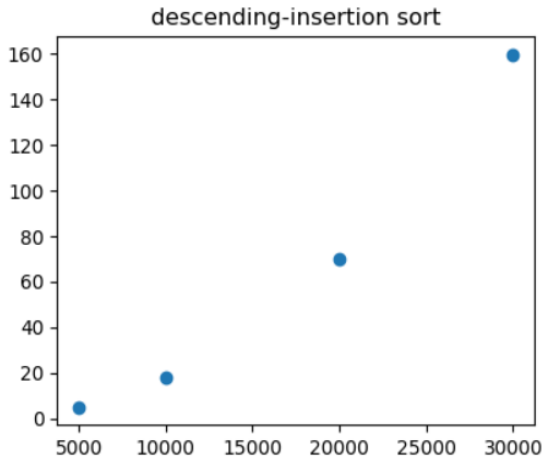
4. Analysis of Results

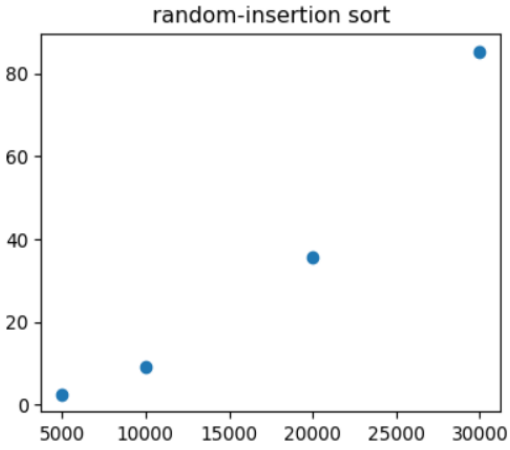
Insertion Sort

For insertion sort, best case is that the list is already sorted in ascending order. The total number of comparison would be n-1, and the time complexity is O(n), where n is the size of the list. From the statistics shown in part 3, when the size increases by 2 times from 5000 to 10000, the CPU time increases by approximately 2 times as well from 0.0035 to 0.0055. The plotted graph is as shown below, it can be observed that the four points more or less lie on a straight line, which proves the time complexity function O(*n*).



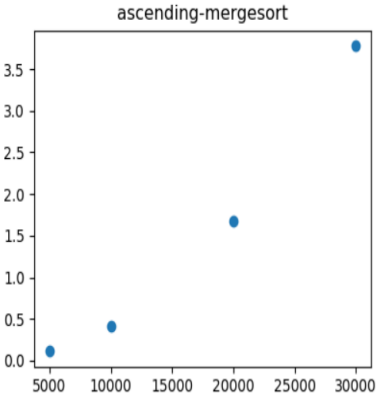
Time complexity function for both worst (list in descending order) and average case (list in random order) are O(n^2). When the size of the list increases by 2 times, CPU time should increase by approximately 4 times.Below are the plotted graph for the two cases.

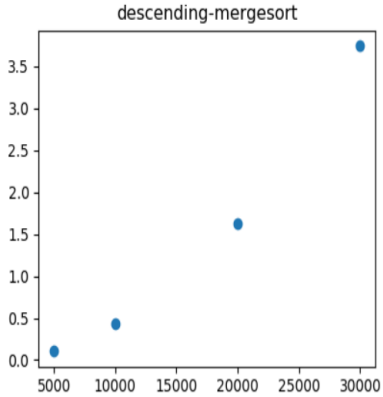


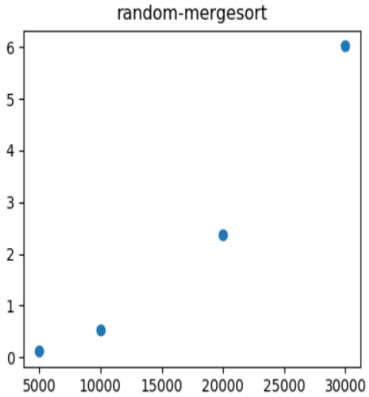


MergeSort

For mergesort, the time complexity functions for all three cases (best, worst, average) are O(nlog(n)) as mergesort always divides the array/list into two halves until there is a single element before merging two sorted sub-lists together and the time taken to merge two halves (or two sorted sub-lists) is linear. The plotted graph is as shown below:







Conclusion

For the list in ascending order, it is the best case for insertion sort. Results show that insertion sort has better performance than mergesort. However, for other cases when the list in random or descending order, mergesort has better performance.