|  |  |
| --- | --- |
| Name: | Zhuang Jianning |
| Student ID: | A0214561M |

1.

a) N = 500000000 takes about 2 seconds to run

b) When dummy is declared as a double, execution time using same N increased from 2.030s to 3.120s (about 1.5 times)

c) The size of int data type in C is usually 2 or 4 bytes while the double data type is 8 bytes, hence more time might be required per basic operation like addition. Double is also much more precise than int.

---------------------------------------------------------------------------------------------------------------------------------------

workA(N) : O(1)

workB(N) : O(N)

workC(N) : O(N^2)

workD(N) : f(N) = 2f(N/2) + O(N) = O(Nlog2\_N) //mergesort

workE(N) : f(N) = f(N/2) + O(1) = (1 + 1 + … + 1 + 1) log2\_N times = O(log2\_N)

---------------------------------------------------------------------------------------------------------------------------------------

2.

a) When N = 1:

average execution time for workA(N) = 6.000s

average execution time for workB(N) = 0.010s

b) No. The time complexity of workA(N) is O(1) constant as the for loop always runs 567 times only and execution time does not change when N increases. The time complexity of workB(N) is O(N) and execution time grows linearly with respect to N.

Beyond a certain point, in this case N = 567, workB(N) performs worse than workA(N) and continues to get worse

N = 567:

average execution time for workA(N) is 6.000s

average execution time for workB(N) is 6.050s

N = 1000:

average execution time for workA(N) is 6.000s

average execution time for workB(N) is 10.670s

N = 2000:

average execution time for workA(N) is 6.000s

average execution time for workB(N) is 21.340s

Indeed based on the experimental data, execution time of workB(N) grows linearly with respect to N while execution time of workA(N) stays constant

3.

a) Approach A :

Approach B :

b)

|  |  |  |
| --- | --- | --- |
| N | Approach A/s | Approach B/s |
| 1 | 0.260 | 0.020 |
| 2 | 0.530 | 0.060 |
| 3 | 0.800 | 0.120 |
| 4 | 1.060 | 0.210 |
| 5 | 1.330 | 0.320 |
| 6 | 1.590 | 0.440 |
| 7 | 1.860 | 0.590 |
| 8 | 2.130 | 0.760 |
| 9 | 2.400 | 0.960 |
| 10 | 2.660 | 1.170 |

Based on b), for small values of N, Approach B seems to be more efficient as its average execution time is smaller, but that is due to the constant factor in Approach A which is significant when N is small. The time complexity of Approach A is O(N) while the time complexity of Approach B is O(N^2). As N grows, Approach A will be the more efficient approach as its execution time grows much slower with respect to N compared to Approach B. This is actually already evident based on data in b). For Approach A, execution time increases by around 0.260s for each step in N from 1 to 10. For Approach B, when N = 2, execution time only increased by 0.040s but by the time N = 10, execution time increased by 2.010s.

4.

Graphical user interface

Description automatically generated with medium confidence

From the function definition, workC(N) should have a time complexity of O(N^2) due to the nested for loop, each from 1 to N. The actual measurements follow the expected growth rate.

Diagram, line chart

Description automatically generated with medium confidence

From the function definition, workD(N) has the following recursive function:

f(N) = 2f(N/2) + O(N) = N + 2(N/2) + 4(N/4) + …. O(Nlog2\_N)

There are log2\_N levels each doing O(N) work just like divide and conquer algorithms like mergesort

The actual measurements follow the expected growth rate. There are spikes in growth due to powers of two.

Chart, line chart

Description automatically generated with medium confidence

From the function definition, workE(N) has the following recursive function:

f(N) = f(N/2) + O(1) = (1 + 1 + … + 1 + 1) log2\_N times = O(log2\_N)

Function terminates when N/(2^i) = 1 => i = log2\_N. Hence, there are log2\_N levels each doing O(1).

The actual measurements show a slight growth which looks linear, but this is due to small values of N. This follows as logarithm graph seems to grow linearly at the beginning too. Trusting in the time complexity we calculated, we can fit a logarithmic curve with k = 0.1 and n0 = 8.

5.

a) Linear Search is used.

b) O(N). For each element in the dictionary, it compares with the word we wish to find and returns true when strcmp(dictionary[i], word) == 0 (when the 2 strings are lexicographically identical).

In the worst case, the function has to search through every element in the dictionary if the word we are looking for is at the end. Hence, the time complexity of searchDictionary() is linear in the size of the dictionary, in this case N. O(N)

Length of each word is constant/max of 32 chars, so can be ignored in complexity analysis.

c)

|  |  |  |  |
| --- | --- | --- | --- |
| N (Dictionary Size) | Time for 10000 query/s | Average time per query/s | # of found words |
| 25000 | 2.350 | 0.000235 | 6 |
| 50000 | 4.340 | 0.000434 | 6 |
| 100000 | 7.950 | 0.000795 | 8 |
| 200000 | 15.500 | 0.00155 | 66 |

Indeed, the growth of time taken follows a linear trend with respect to N.

d)

Each word used to search in the dictionary is read in from words\_query.txt. If we did just 1 query, it could be the case where the first word is found near the start of the dictionary when linear search is performed, hence the timings would be artificially fast. It is better to run 10,000 queries and take the average to get a better picture of the time complexity of the linear search function.

6.

a) Insertion Sort is used. We maintain a sorted subarray from a[0:i-1]. For each element from index 1 to N-1, we find the insertion point into the sorted array on the left by comparison with the previous elements and swapping if it is smaller. This grows the sorted subarray until it becomes the whole array.

b)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N (Dictionary Size) | Sorting/s | Searching/s | Overall/s | # words found |
| 25000 | 4.670 | 0.010 | 4.680 | 6 |
| 50000 | 26.870 | 0.010 | 26.880 | 6 |
| 100000 | 93.310 | 0.010 | 93.320 | 8 |
| 200000 | 238.770 | 0.010 | 238.780 | 66 |

Overall time follows closely with Sort time hence is omitted from plot as it is almost a 100% overlap due to the axes division

Insertion sort has a worse case time complexity of O(N^2). The Sort trend line follows a quadratic growth trend which agrees with our time complexity

Binary Search on a sorted array takes logarithmic time O(log2\_N). The search trend line should follow the logarithmic growth trend but it is hard to see due to the scale and the small values.

Overall, the quadratic time complexity is significant hence overall time complexity is O(N^2) which follows from the trendline of sort.

c)

It depends on the number of queries we are making. Let q be the number of queries we make.

The actual time complexity of Approach 1 becomes O(q\*N) as we search through the dictionary in O(N) for each of the q queries.

The actual time complexity of Approach 2 becomes O(N^2 + q\*log2\_N). Sorting the dictionary is independent of number of queries and insertion sort takes O(N^2) time. Binary search takes O(log2\_N) time for each of the q queries. Hence overall complexity is O(N^2 + q\*log2\_N).

In this lab, q was harcoded to a constant value of 10000 queries, hence we seem to be comparing linear time complexity O(N) for Approach 1 with quadratic time complexity O(N^2) for Approach 2. However, when the value of q is not kept constant or significant enough to cover even the N^2 term, Approach 2 might become a more efficient approach than Approach 1 since we just sort once and enjoy faster search compared to performing linear search for each query. The cost of sorting can be amortized by the number of queries especially in this case where N is bounded by 220000 due to memory limitation.

Some experimental data when increasing q from 10,000 to 100,000:

N = 25000

Text

Description automatically generated

Time taken for Approach 1 increased from 2.350s to 19.100s

Time taken for Approach 2 increased from 4.710s to 4.720s

N = 50000

Text

Description automatically generated

Time taken for Approach 1 increased from 4.320s to 33.940s

Time taken for Approach 2 increased from 26.880 to 27.030s

Challenge Section

7. Merge Sort O(Nlog2\_N)

All the types of sorts so far use the comparison model, while sorts like selection, bubble or insertion have a upper bound complexity of O(N^2), Merge Sort is a divide and conquer algorithm and has upper bound of O(Nlog2\_N) regardless of the type of input.

But there is a lower bound to comparison based sorts which can be proven with a binary decision tree with N! leaves due to permutation. The height of such a tree is then log(N!) which can be simplified with stirlings approximation of N! -> (N/e)^N. Height of tree becomes Nlog2\_N which corresponds to time complexity of getting from root to leaf. It is a lower bound considering the binary tree is complete

8.

a) It is more efficient