College of Engineering, Software Department
SE 2221 Algorithms

Laboratory 2
Improving and Comparing Algorithms

Name: Rafael III Prudente Year: 2
Name: Paul Vincent Lerado Year: 2

General Instructions: For every item below, please submit one zip file named "<surname1>_<surname2>_<item number>.zip". For example, if Kaedahara and Sangonomiya were submitting for section number 3, they would name their zip file "Kaedahara_Sangonomiya_3.zip". Please also submit a filled out version of this laboratory named "Kaedahara_Sangonomiya_Lab2.doc" or with any file extension you have .

Objectives:

- To be able to understand pseudocode and implement it in a language of choice.
- To be able to think of ways to improve or modify algorithms.
- To be able to compare some algorithms that essentially do the same task but in different ways.
- To be able to analyze the strengths and weaknesses of algorithms when used in different conditions.

Section 1.
Part 1. To further our understanding of the binary search, you shall be implementing the algorithm yourselves in this section. The following is the pseudocode of the binary search algorithm:

```
function binary_search(A, n, T) is
    L := 0
    R := n − 1
    while L ≤ R do
        m := floor((L + R) / 2)
        if A[m] < T then
            L := m + 1
        else if A[m] > T then
            R := m − 1
        else:
```

```
            return m
    return -1
```

Make sure to add in some test cases that check whether the algorithm actually works. I **highly** recommend you do not check the internet for the code as this will be good practice. You can, however, look at references and docs. You must submit the code for this along with this laboratory doc.

Part 2. After finishing your implementation, test it on an array of 1024 unique numbers. Perform around 100 or more trials on that same array but with changing the target value for every trial. Measure and record the number of splits/halves the algorithm needed to perform on each trial as well as the position of the target value. Compile your results on a visual medium like a chart or a table. (I suggest chartjs or chartjs-cli to help you with this.) Submit this along with the code.

Questions.

1. Were there any patterns or relationships between the position/index of the target value and the number of halves required to find the target value? What kind of values seemed to be easier to find?
    - So while doing the algorithm and based on our discussion, the pattern that we've noticed between the index and number of halves to find the target value is "the closer the value of the target in the middle, the faster it gets to be returned. Vice versa to the values that is the leftmost or the rightmost because it requires several halves to get into the index of it." In conclusion, the number of halves required to find the target value in binary search is proportional to the logarithm of the size of the array. The ease of finding the target value depends on its position in the array and the distribution of values in the array..

Part 3. An algorithm usually will never be the end-all-be-all algorithm to perform a certain task. In fact, research is done continually to find more and more improvements to an existing algorithm. The binary search algorithm is no exception. For this last part, try to add in modifications or improvements to the binary search algorithm that you think might help make it faster. Compare its execution time to the original algorithm on around 50 trials using different kinds of arrays. Please also submit the improved algorithm's code along with the results.

Note: With your modifications, it still has to produce the same **correct** output.

Questions.

1. What was your improvement for the algorithm? Explain why you think this improvement can help.

- Firstly, we optimized the calculation of the mid-point value which is critical for increasing the algorithm's speed. Instead of calculating the midpoint using (low+high)/2, bitwise operators such as right shift should be utilized to optimize the calculation. Because bitwise operations are faster than arithmetic operations, the execution time is reduced. Furthermore, if the target value is equal to the mid-point value, returns the mid-point index immediately. This step can help to improve the speed of the algorithm.

2. Did your modification manage to make the algorithm faster or slower? What did the results tell you?

- For the same test cases that we have implemented for both of them, we can clearly see that the modified binary search algorithm is much faster than the original one.

Section 2.
For this final section, you shall be comparing three sorting algorithms: Bubble Sort, Insertion Sort, and Selection Sort. To save time, you can get their codes at https://github.com/trekhleb/javascript-algorithms. You need to measure their execution times on eight different types of arrays:

A. A 128-length array with randomly filled elements
B. A 256-length array with randomly filled elements
C. A 512-length array with randomly filled elements
D. A 1024-length array with randomly filled elements
E. A 1024-length array which is ALREADY sorted
F. A 1024-length array which is the reverse of an already sorted one
G. A 1024-length array which is ALMOST sorted (around 5~15 elements are out of place)
H. A 2048-length array with randomly filled elements.

Perform 3~5 trials on each array and get the average execution time, then fill in the table below with your results. Please submit the code you used to perform this experiment.

Array type	Bubble Sort	Insertion Sort	Selection Sort
A128	1.069ms	0.121ms	0.579 ms
B256	3.279ms	0.062ms	3.708ms
C512	2.832ms	0.131ms	3.297ms
D1024	2.84 ms	0.17ms	0.667ms
E1024Sorted	1.175ms	0.192ms	0.803ms
F1024Reversed	1.472ms	0.106ms	1.09ms
G1024Almost	1.123ms	0.115ms	0.927ms
H2048	6.782ms	0.341ms	3.7ms

Questions:

1. As the length of the array increases, which sorting algorithm tends to perform poorly? Why does that algorithm perform poorly?

Bubble sort performs poorly as array length increases because it has a time complexity of O(n2), which means it must execute n2 comparisons and swaps to sort an array of length n. Insertion sort and selection sort have the same time complexity as bubble sort, but they perform better in practice since they make fewer comparisons and swaps. For really large arrays, however, all three sorting algorithms will perform poorly, and it is preferable to use a more efficient sorting algorithm, such as merge sort or quicksort, which have a time complexity of O. (n log n).

2. For an already sorted array, which algorithm performed the best? Why did that algorithm perform well?
   Insertion sort is the best choice for an already sorted array. This is due to the fact that insertion sort has a best-case time complexity of O(n), which means that it only requires a single run around the array to determine that it is already sorted. On the other hand, bubble and selection sort have a time complexity of O(n2) in the best situation, which means that even if the array is already sorted, they must do numerous unnecessary comparisons and swaps. As a result, for an already sorted array, insertion sort is the most efficient sorting method.

3. How about the array that was almost sorted? Did the same algorithm in #2 perform the best for the almost sorted array as well? Why is that so (or not so)?

For an almost sorted array, insertion sort performs bubble sort, insertion sort, and selection sort. This is due to the average-case time complexity of insertion sort being O(n+k), where k is the number of inversions or out-of-place entries in the almost sorted array. There are few inversions or out-of-place elements in the array because it is almost sorted, and k is small. As a result, insertion sort only requires a few comparisons and swaps to sort the array, making it faster than bubble sort and selection sort, which have an average time complexity of O(n2). As a result, even for almost sorted arrays, insertion sort is the most efficient algorithm of the three.

4. What are your observations for the reversed sorted array? What are your thoughts on the general execution time when compared to a randomly filled array?

Bubble sort, insertion sort, and selection sort all have a worst-case time complexity of O(n2) for a reversed sorted array, making them relatively slow for large arrays. This is because these algorithms must perform numerous comparisons and swaps to relocate each member to its correct location, resulting in a significant amount of work for an already reversed array. Because of the number of comparisons and swaps they do, bubble sort and selection sort are slightly slower than insertion sort in practice.
A reversed sorted array requires more comparisons and swaps to sort than a randomly filled array, increasing the execution time of bubble sort, insertion sort, and selection sort. Because a randomly filled array is more likely to be partially sorted and has fewer inversions or out-of-place elements than a reversed sorted array, the number of comparisons and swaps required to sort the array is reduced. For best performance, utilize more efficient sorting algorithms such as merge sort or quicksort, which have a worst-case time complexity of O(n log n) and can handle reversed sorted arrays and randomly populated arrays more efficiently.

5. In a general project, which sorting algorithm would you use among the three? Why?

In general, I would choose insertion sort over bubble sort, insertion sort, and selection sort if the data set is not too large and has not already been sorted. This is due to the fact that insertion sort is simple, easy to implement, and has good performance characteristics for small and almost sorted arrays. Its O(n+k) average-case time complexity makes it ideal for datasets with few inversions or out-of-place elements. However, if the dataset is very large or has already been sorted, or if efficiency is essential, I would explore more efficient sorting algorithms such as merge sort or quicksort, which have lesser time complexity in the worst scenario and are better suited for large datasets.

Conclusions.

In conclusion, through this laboratory, it was found that the ability to understand pseudocode and implement it in a language of choice is a critical skill for success in these understanding algorithms just like what we did in the binary search algorithm. Additionally, the ability to think of ways to improve or modify algorithms is essential for optimizing performance and efficiency. We did conclude here that our modified binary search is much faster than the original one for the reasons we've discussed above.

For the second part, comparing different algorithms that perform similar tasks and analyzing their strengths and weaknesses under other conditions is also critical for selecting the most appropriate algorithm for a particular task. Overall, the results of this laboratory highlight the importance of these skills for anyone seeking to be proficient in algorithms. By developing these skills, individuals can successfully understand, develop, and optimize algorithms and software systems to perform tasks more efficiently and effectively, resulting in more successful and impactful projects.