**HOMEWORK 3: SEARCHING – BASIC SORT**

**Name: Trần Gia Bảo**

**ID: 22127034**

**A. Theory part:**

**A1. Linear search must examine, on average, half of the N elements of an array while binary search needs to only check log2N. Thus, binary search is a great deal more efficient than linear search. Why, then, should we still study and use linear search?**

- While it is true that binary search is generally more efficient than linear search, there are still several reasons why we should study and use linear search:

+ Simplicity: Linear search is one of the simplest search algorithms to understand and implement.

+ Unsorted Data: Linear search does not require the data to be sorted beforehand. In contrast, binary search requires the array to be sorted.

+ Small Data Sets: For very small data sets, the difference in efficiency between linear and binary search might not be significant. The overhead of implementing and maintaining a binary search algorithm might not be worth it in such cases, making linear search a practical choice.

+ Sequential Access: Linear search is efficient when the data is stored in a sequential or linked structure, such as a linked list. In such cases, random access is not possible, which makes binary search impractical. Linear search allows us to iterate through the elements one by one until the desired element is found.

+ Debugging and Educational Purposes: Linear search is often used as a teaching tool in computer science and programming courses to introduce basic search algorithms. Understanding linear search can help students grasp fundamental concepts like loops, conditionals, and array traversal. It also serves as a building block for more advanced search algorithms.

+ Auxiliary to Other Algorithms: Linear search is sometimes used as a component of other algorithms or as a fallback when the dataset is small. For example, in certain sorting algorithms like Bubble Sort or Insertion Sort, linear search is used to find the position where an element should be inserted or swapped.

🡪 It's important to note that the choice of search algorithm depends on the specific requirements, characteristics of the data, and the trade-offs between time complexity, space complexity, and code complexity. While binary search is often more efficient, linear search still has its relevance and utility in various scenarios.

**A2. Consider the following array of integers, {1, 4, 6, 7, 9, 10, 14}. For linear search, which elements will be examined to determine that 9 is in the array, and which elements will be examined to determine that 11 is not in the array? Repeat the question for binary search.**

- For linear search:

+ To determine if 9 is in the array, the linear search algorithm will examine each element in the array sequentially until a match is found or the end of the array is reached. In this case, the elements examined will be: 1, 4, 6, 7, 9

+ To determine if 11 is not in the array, the linear search algorithm will examine each element in the array sequentially until the end of the array is reached without finding a match. In this case, the elements examined will be: 1, 4, 6, 7, 9, 10, 14

- For binary search:

+ To determine if 9 is in the array, the binary search algorithm will examine elements in the following order: 7, 10, 9

+ To determine if 11 is not in the array, the binary search algorithm will examine elements in the following order: 7, 10, 14

**A3. Consider an array of 5 integers. What are the maximum number of comparisons and the maximum number of assignments if the array is sorted by bubble sort? Justify your answer. Repeat the question for selection sort, insertion sort, and interchange sort.**

- Bubble Sort:

+ Bubble sort works by repeatedly swapping adjacent elements if they are in the wrong order until the array is sorted. For an array of size 5, bubble sort will require a maximum of 4 iterations to sort the array.

+ Maximum comparisons: In each iteration, bubble sort compares adjacent elements and swaps them if necessary. In the worst case, the first iteration compares all 5 elements, the second iteration compares 4 elements, the third iteration compares 3 elements, and so on. Therefore, the maximum number of comparisons for bubble sort on an array of 5 integers is 5 + 4 + 3 + 2 = 14.

+ Maximum assignments: Bubble sort performs assignments when swapping elements. Similar to comparisons, the first iteration requires 4 assignments, the second iteration requires 3 assignments, the third iteration requires 2 assignments, and the last iteration requires 1 assignment. Thus, the maximum number of assignments for bubble sort on an array of 5 integers is 4 + 3 + 2 + 1 = 10.

- Selection Sort:

+ Selection sort works by repeatedly finding the minimum element and placing it in the correct position. For an array of size 5, selection sort will require a maximum of 4 iterations to sort the array.

+ Maximum comparisons: In each iteration, selection sort compares the current element with the remaining elements to find the minimum. In the worst case, the first iteration compares 5 elements, the second iteration compares 4 elements, the third iteration compares 3 elements, and so on. Hence, the maximum number of comparisons for selection sort on an array of 5 integers is 5 + 4 + 3 + 2 = 14.

+ Maximum assignments: Selection sort performs assignments when swapping the minimum element with the current element. Similar to comparisons, the first iteration requires 1 assignment, the second iteration requires 1 assignment, and so on until the last iteration, which doesn't require any assignment. Therefore, the maximum number of assignments for selection sort on an array of 5 integers is 1 + 1 + 1 + 1 = 4.

- Insertion Sort:

+ Insertion sort works by repeatedly inserting an element into its correct position in the sorted part of the array. For an array of size 5, insertion sort will require a maximum of 4 iterations to sort the array.

+ Maximum comparisons: In each iteration, insertion sort compares the current element with the sorted elements to find its correct position. In the worst case, the first iteration compares 1 element, the second iteration compares 2 elements, the third iteration compares 3 elements, and so on. Hence, the maximum number of comparisons for insertion sort on an array of 5 integers is 1 + 2 + 3 + 4 = 10.

+ Maximum assignments: Insertion sort performs assignments when shifting elements to make space for the current element. In the worst case, the first iteration requires 0 assignments, the second iteration requires 1 assignment, the third iteration requires 2 assignments, and so on until the last iteration, which requires 3 assignments. Therefore, the maximum number of assignments for insertion sort on an array of 5 integers is 0 + 1 + 2 + 3 = 6.

- Interchange Sort (or Bubble Sort variation):

+ Interchange sort is similar to bubble sort but performs exchanges directly, without comparing adjacent elements. For an array of size 5, interchange sort will require a maximum of 4 iterations to sort the array.

+ Maximum comparisons: In each iteration, interchange sort compares the current element with all the remaining elements. In the worst case, the first iteration compares 4 elements, the second iteration compares 3 elements, the third iteration compares 2 elements, and so on. Hence, the maximum number of comparisons for interchange sort on an array of 5 integers is 4 + 3 + 2 + 1 = 10.

+ Maximum assignments:

Interchange sort performs assignments when exchanging elements directly. Similar to comparisons, the first iteration requires 1 assignment, the second iteration requires 1 assignment, and so on until the last iteration, which doesn't require any assignment. Therefore, the maximum number of assignments for interchange sort on an array of 5 integers is 1 + 1 + 1 + 1 = 4.

🡪 To summarize:

- Bubble Sort: Maximum comparisons = 14, Maximum assignments = 10.

- Selection Sort: Maximum comparisons = 14, Maximum assignments = 4.

- Insertion Sort: Maximum comparisons = 10, Maximum assignments = 6.

- Interchange Sort: Maximum comparisons = 10, Maximum assignments = 4.

**A4. Consider the following array of integers, {26, 48, 12, 92, 28, 6, 33}. For bubble sort, show the resulting array for each of the first three iterations of the outer loop. Repeat the question for selection sort, insertion sort, and interchange sort.**

- Bubble Sort:

+ Iteration 1:

Comparisons and Swaps:

26, 48, 12, 92, 28, 6, 33 (No swap)

26, 12, 48, 92, 28, 6, 33 (Swap: 48 and 12)

26, 12, 48, 28, 92, 6, 33 (Swap: 92 and 28)

26, 12, 48, 28, 6, 92, 33 (Swap: 92 and 6)

26, 12, 48, 28, 6, 33, 92 (Swap: 92 and 33)

🡪 Result after the first iteration: {26, 12, 48, 28, 6, 33, 92}

+ Iteration 2:

Comparisons and Swaps:

12, 26, 48, 28, 6, 33, 92 (No swap)

12, 26, 28, 48, 6, 33, 92 (Swap: 48 and 28)

12, 26, 28, 6, 48, 33, 92 (Swap: 48 and 6)

12, 26, 28, 6, 33, 48, 92 (Swap: 48 and 33)

🡪 Result after the second iteration: {12, 26, 28, 6, 33, 48, 92}

+ Iteration 3:

Comparisons and Swaps:

12, 26, 28, 6, 33, 48, 92 (No swap)

12, 26, 28, 6, 33, 48, 92 (No swap)

12, 26, 6, 28, 33, 48, 92 (Swap: 28 and 6)

🡪 Result after the third iteration: {12, 26, 6, 28, 33, 48, 92}

- Selection Sort:

+ Iteration 1:

Comparisons and Swaps:

12, 48, 26, 92, 28, 6, 33 (Swap: 12 and 6)

🡪 Result after the first iteration: {6, 48, 26, 92, 28, 12, 33}

+ Iteration 2:

Comparisons and Swaps:

6, 12, 26, 92, 28, 48, 33 (No swap)

🡪 Result after the second iteration: {6, 12, 26, 92, 28, 48, 33}

+ Iteration 3:

Comparisons and Swaps:

6, 12, 26, 92, 28, 48, 33 (Swap: 26 and 28)

🡪 Result after the third iteration: {6, 12, 28, 92, 26, 48, 33}

- Insertion Sort:

+ Iteration 1:

Comparisons and Swaps:

26, 48, 12, 92, 28, 6, 33 (No swap)

🡪 Result after the first iteration: {26, 48, 12, 92, 28, 6, 33}

+ Iteration 2:

Comparisons and Swaps:

12, 26, 48, 92, 28, 6, 33 (No swap)

🡪 Result after the second iteration: {12, 26, 48, 92, 28, 6, 33}

+Iteration 3:

Comparisons and Swaps:

12, 26, 48, 92, 28, 6, 33 (Swap: 92 and 28)

12, 26, 48, 28, 92, 6, 33 (Swap: 28 and 6)

🡪 Result after the third iteration: {12, 26, 48, 28, 92, 6, 33}

- Interchange Sort:

+ Iteration 1:

Comparisons and Swaps:

26, 48, 12, 92, 28, 6, 33 (Swap: 26 and 6)

6, 48, 12, 92, 28, 26, 33 (Swap: 26 and 12)

🡪 Result after the first iteration: {6, 48, 12, 92, 28, 26, 33}

+ Iteration 2:

Comparisons and Swaps:

6, 12, 48, 92, 28, 26, 33 (No swap)

🡪 Result after the second iteration: {6, 12, 48, 92, 28, 26, 33}

+ Iteration 3:

Comparisons and Swaps:

6, 12, 48, 92, 28, 26, 33 (Swap: 92 and 28)

6, 12, 48, 28, 92, 26, 33 (Swap: 92 and 26)

🡪 Result after the third iteration: {6, 12, 48, 28, 26, 92, 33}

**A5. Which sorting algorithm(s) is best for each of the following situations? Justify your answer.**

**a. When all elements of the input array are identical, which algorithm(s) will take least time?**

**b. When the swapping is very costly, which algorithm(s) should be preferred so that the number of swaps are minimized in general?**

a. When all elements of the input array are identical, the best sorting algorithm would be Insertion Sort or Selection Sort.

- Insertion Sort: This algorithm has a favorable characteristic when it comes to arrays with identical elements. Since Insertion Sort compares adjacent elements and moves the current element to its correct position, it will require minimal comparisons and swaps in the case of identical elements. The array is already sorted in this scenario, so Insertion Sort will exhibit its best-case time complexity of O(n).

- Selection Sort: Selection Sort repeatedly selects the minimum element and places it in the sorted part of the array. In the case of identical elements, the algorithm will consistently find the minimum value in each iteration without the need for swaps. The number of swaps will be minimal, resulting in a more efficient performance compared to algorithms like Bubble Sort or Interchange Sort.

b. When swapping is very costly, the best sorting algorithm to minimize the number of swaps would be Selection Sort or Merge Sort.

- Selection Sort: Selection Sort has a fixed number of swaps for each iteration. In every iteration, it selects the minimum element and places it in its correct position. The number of swaps is directly related to the number of iterations and remains constant regardless of the input array. Therefore, if swapping is costly, Selection Sort can be preferred as it performs fewer swaps compared to other algorithms.

- Merge Sort: Merge Sort is a divide-and-conquer algorithm that divides the input array into smaller subarrays, sorts them, and then merges them to obtain the final sorted array. Merge Sort has a predictable and well-defined number of swaps. In its standard implementation, Merge Sort requires a temporary array to merge the subarrays. The number of swaps is proportional to the number of elements being merged, which is determined by the height of the merge tree. While Merge Sort may require more swaps than Selection Sort, it is still a good choice when swapping is costly due to its efficient overall performance and stable time complexity of O(n log n). Additionally, if an in-place variant of Merge Sort is used, the number of swaps can be minimized.

- Both Selection Sort and Merge Sort offer advantages in minimizing the number of swaps, but the optimal choice between them depends on various factors, such as the specific cost of swapping, the size of the input array, and other performance requirements.

**A6. Choose the best answer for each of the below single-choice questions about selection sort.**

**• Through experiment, you determine that selection sort performs 5000 comparisons when sorting an array of some size k. If you doubled the size of the array to 2k, approximately how many comparisons would you expect it to perform?**

**a) 5000 b) 10000 c) 20000**

**d) 40000 e) The value would depend on the array’s content**

- The number of comparisons in selection sort depends on the size of the array. Since selection sort compares each element with every other element in the unsorted portion of the array, the number of comparisons is directly proportional to the size of the array. Therefore, if the size of the array is doubled from k to 2k, we can expect the number of comparisons to roughly double as well.

- The answer would be: b) 10000

**• Through experiment, you determine that selection sort performs 5000 data moves when sorting an array of some size k. If you doubled the size of the array to 2k, approximately how many moves would you expect it to perform?**

**a) 5000 b) 10000 c) 20000**

**d) 40000 e) The value would depend on the array’s content**

- The number of data moves in selection sort also depends on the size of the array. In each iteration of selection sort, the algorithm finds the minimum (or maximum) element and swaps it with the current position. Each swap involves moving data between array elements. Similar to comparisons, the number of data moves in selection sort is directly proportional to the size of the array. If the size of the array is doubled from k to 2k, we can expect the number of data moves to roughly double as well.

- The answer would be: b) 10000

VD: y = x^2 – z^2 P(4, 7, 3)

F(x, y, z) = x^2 – y – z^2

DelF = <Px, Py, Pz>

= <2x, -1, -2z>

DelF(P) = <8, -1, -6>

Mặt phẳng qua P với ∇F(P) là vectơ pháp tuyến là:

Fx(P)(x-xo) + Fy(y-yo) + Fz(z-zo) = 0

⬄ 8(x-4) - 1(y-7) - 6(z-3) = 0

⬄ 8x - y - 6z -7 = 0

Đường pháp tuyến của mặt (S) tại P(4, 7, 3):

(x-4)/8 = 7-y = (z-3)/-6

VD2: Fx = 3x^2 -3y = 0, Fxy = -3, Fxx = 6x

Fy = 3y^2 -3x = 0, Fyx = -3, Fyy = 6y

Do đó, các điểm dừng của f là (0, 0), (1, 1).

Ta có:

D(x, y) = Fxx.Fyy – Fxy^2 = 36xy - 9

Vì D(0, 0) < 0 và nên (0, 0) là điểm yên ngựa

Vì D(1,1) = 27 và Fxx(1, 1) = 6 > 0 nên (1, 1) là điểm cực tiểu