Answer Script

Basic Data Structures and Problem Solving Part-II

Week-4, Module 15: Theory Mid Term Exam

(nayeem.cse6.bu@gmail.com)

Question No. 01
Write down the time complexity of Bubble sort, Insertion sort and Merge sort.
Answer No. 01

> Time Complexity of Bubble Sort: O(n^2)

The time complexity of bubble sort is O(n^2) in the worst and average case, and O(n) in the best case.

Bubble sort is an algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

In the worst case, the array is in the reverse order, and the algorithm needs to traverse the entire list for n-1 times to sort it, during each traversal it needs to compare and swap n-1 elements, so the total number of operations is (n-1)*n/2 which is O(n^2). In the average case, the algorithm needs to traverse the list for n-1 times and for each traversal, it needs to compare and swap n/2 elements, which results in a O(n^2) time complexity.

However, the best case is when the array is already sorted, in that case, the algorithm only needs one pass through the list, and no swaps are needed, so the time complexity is O(n).

In conclusion, bubble sort has a time complexity of O(n^2) in the worst and average case, and O(n) in the best case.

› Time Complexity of Insertion Sort: O(n^2)

The time complexity of insertion sort is O(n^2) in the worst and average case, and O(n) in the best case.

Insertion sort is an algorithm that builds a sorted array one element at a time by repeatedly taking the next unsorted element and inserting it into its correct position within the sorted portion of the array.

In the worst case, the array is in reverse order and the algorithm needs to compare and shift n-1 elements for each of the n elements, resulting in (n-1) + (n-2) + ... + 1 = n(n-1)/2 operations, which is O(n^2). In the average case, the algorithm needs to shift n/2 elements for each of the n elements resulting in n(n-1)/4 operations, which is also O(n^2).

However, the best case is when the array is already sorted, in that case, the algorithm does not need to shift any elements, it only needs to compare n-1 elements for each of the n elements, resulting in (n-1) operations, which is O(n)

In conclusion, insertion sort has a time complexity of O(n^2) in the worst and average case, and O(n) in the best case.

› Time Complexity of Merge Sort: O(n log n)

The time complexity of merge sort is O(n*log(n)) in all cases.

Merge sort is a divide-and-conquer algorithm that divides the input array in two, sorts each of the two halves, and then merges the two sorted halves back together.

The time complexity of merge sort is dominated by the merge step, which takes O(n) time to merge two sorted arrays of size n/2 each. Therefore, the time complexity of merge sort is O(n*log(n)), where log(n) is the number of times the array is divided in half, and n is the number of elements in the array.

It's worth noting that the merge step is linear in time, but the recursion process is

logarithmic in time, so the time complexity is O(n*log(n))

In conclusion, merge sort has a time complexity of O(n*log(n)) in all cases, which makes it more efficient than O(n^2) algorithms like bubble sort and insertion sort, especially for large inputs.

Question No. 02

Write two differences between array and linked-list. Why do we need a head/root node in a linked list?

Answer No. 02

> Two differences between array and linked-list:

Array	Linked-List
An array is a data structure that stores a collection of elements, each identified by an index.	A linked-list is a data structure that stores a collection of elements, each with a reference to the next element in the list.
Arrays are stored in contiguous memory locations, this means that elements in an array are physically adjacent to one another in memory	Linked-Lists are stored in non-contiguous memory locations, this means elements in a linked-list are scattered throughout memory and connected by pointers.

> The reason why do we need a head/root node in a linked list:

A head/root node in a linked list is used as a starting point to access the other elements in the list. It contains a reference to the first element in the list, and each subsequent element in the list can be accessed by following the reference in the previous element. In this way, the head node acts as an entry point to the linked list, allowing us to traverse the list and perform operations on its elements. Without

a head node, it would not be possible to access any of the elements in the list, making it impossible to use the linked list for any meaningful purpose.

Question No. 03

What is the basic idea behind the binary search algorithm, and how does it differ from linear search? What is the requirement for an array to be suitable for binary search?

Answer No. 03

› Basic idea behind the binary search algorithm:

The basic idea behind the binary search algorithm is to repeatedly divide the search interval in half. The algorithm begins by comparing the target value to the middle element of the array. If the target value is equal to the middle element, the search is successful and the index of the middle element is returned. If the target value is less than the middle element, the search continues on the lower half of the array; or if the target value is greater than the middle element, the search continues on the upper half of the array. This process continues, narrowing down the search interval until the target value is either found or the search interval is empty.

Binary search algorithm is an efficient algorithm for finding an element in a sorted array in O(log n) time. It's one of the most basic and efficient searching algorithms which can be used on a sorted array or list.

› How does binary search differ from linear search:

Linear search and binary search are both algorithms for searching for a specific value in a collection of elements, but they differ in the way they approach the problem.

Linear search is a simple algorithm that starts at the first element of the collection and compares each element in sequence to the target value, until it is either found or the end of the collection is reached. The time complexity of linear search

is O(n), where n is the number of elements in the collection. It's simple to implement but its performance degrades as the size of the collection increases.

Binary search, on the other hand, is a more efficient algorithm that is based on the assumption that the collection is already sorted. It starts by comparing the target value to the middle element of the collection, and then repeatedly dividing the search interval in half, until the target value is either found or the search interval is empty. The time complexity of binary search is O(log n) which is much faster than linear search. However, it requires the collection to be sorted, and it is not well suited for collections with duplicate values.

In summary, linear search is a simple algorithm that works well for small collections or unsorted collections, while binary search is a more efficient algorithm that works well for large, sorted collections.

> An array is suitable for binary search if it satisfies the following requirement:

The array must be sorted: The binary search algorithm relies on the property that the array is sorted. The algorithm repeatedly divides the search interval in half, and it is only able to do this if the elements are in a known order.

The array must have a fixed size: Binary search works on arrays which have a fixed size, if the array is dynamic in size, like a linked list, the algorithm will not work.

The array must be homogeneous: All elements in the array must be of the same data type, otherwise the algorithm will not be able to compare elements.

The array must be accessible by index: The algorithm needs to access the elements in the array by index, so the array must be stored in contiguous memory. If the array is stored in non-contiguous memory, like in a linked list, the algorithm will not work.

In summary, for an array to be suitable for binary search, it must be sorted, have a fixed size, be homogeneous and be accessible by index.

Question No. 04
What is the time complexity of inserting an element at the beginning of a singly linked list? What is the time complexity of inserting an element at any index of a singly linked list? What is the time complexity of deleting an element at the beginning of a singly linked list? What is the time complexity of deleting an element at the beginning of a singly linked list?
Answer No. 04
> Inserting an element at the beginning of a singly linked list: The time complexity of inserting an element at the beginning of a singly linked list is O(1). Inserting an element at the beginning of a singly linked list is a constant-time operation, regardless of the number of elements in the list. This is because the operation only requires a constant amount of work to be done, regardless of the size of the list. **> Inserting an element at any index of a singly linked list:** The time complexity of inserting an element at any index of a singly linked list is O(n). Inserting an element at any index of a singly linked list is an operation that takes linear time, O(n), because it requires traversing the list from the head node to the position where the element is to be inserted. **> Deleting an element at the beginning of a singly linked list:** The time complexity of deleting an element at the beginning of a singly linked list is O(1). Deleting an element at the beginning of a singly linked list is a constant-time operation, regardless of the number of elements in the list. This is because the operation only requires a constant amount of work to be done, regardless of the size of the list. **> Deleting an element at any element of a singly linked list:**

The time complexity of deleting an element at any index of a singly linked list is O(n).

Deleting an element at any index of a singly linked list is an operation that takes linear time, O(n), because it requires traversing the list from the head node to the position where the element is to be deleted.

Question No. 05
Suppose we have an array of 5 integer numbers and a singly linked list of 5 integer numbers. Here the singly linked list of 5 integer numbers takes twice as much memory compared to array. Why is that? Give a proper explanation.
Answer No. 05
A singly linked list takes more memory than an array when storing a collection of elements because each element in a linked list requires an additional reference (or "pointer") to the next element in the list. In the case of an array, each element is stored in a contiguous block of memory, and its position in the array can be determined by its index. This means that the memory required to store an array of n elements is equal to the size of each element multiplied by n. In contrast, each element in a singly linked list is stored in a separate block of memory, and its position in the list is determined by a reference (or "pointer") to the next element in the list. This means that the memory required to store a singly linked list of n elements is equal to the size of each element multiplied by n plus the size of the reference (or "pointer") multiplied by n. So, if we have an array of 5 integer numbers, the memory required would be 5*sizeof(int) And, if we have a singly linked list of 5 integer numbers, the memory required would be 5*sizeof(int) + 5*sizeof(pointer)

Because the size of a pointer is usually larger than the size of an integer, the memory required by a singly linked list is typically larger than the memory required by an array.

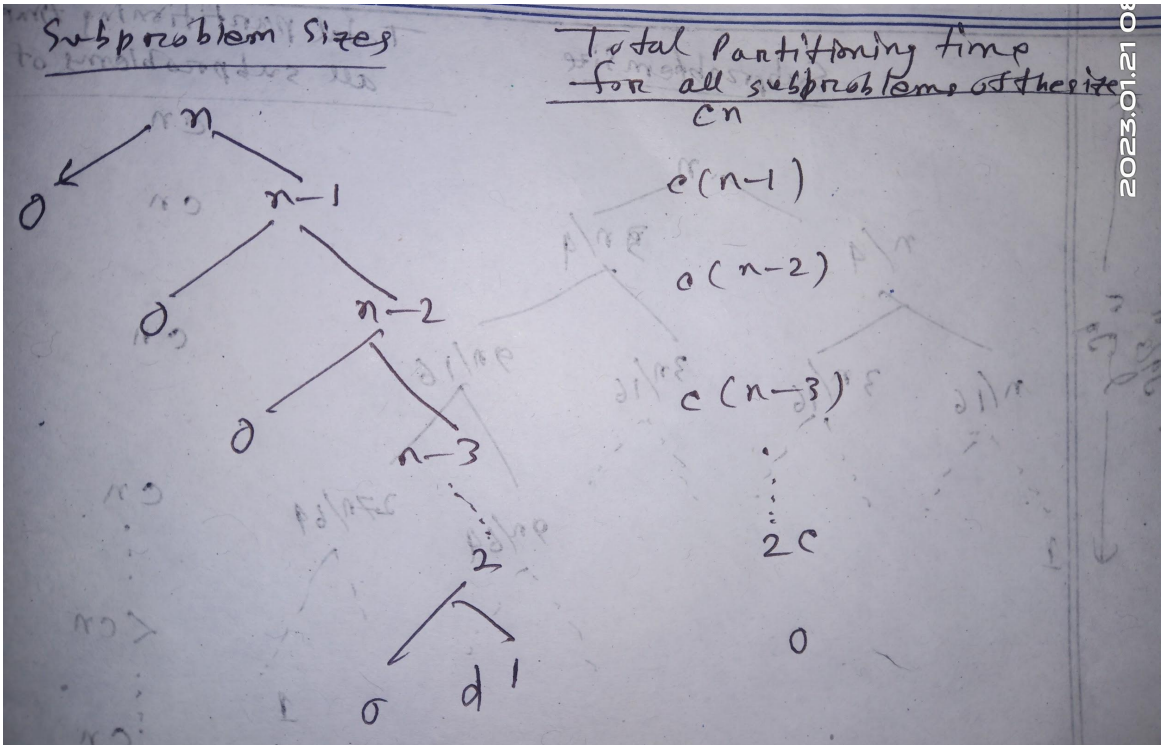
In summary, the singly linked list takes more memory than an array because, in addition to the memory required to store the element, it also requires memory to store the reference (or "pointer") to the next element in the list.

Question No. 06
Derive the worst case and average case time complexity of Quick Sort with proper figures.
Answer No. 06

Quick sort is a divide-and-conquer algorithm that sorts an array by partitioning it into two sub-arrays, one with elements smaller than a pivot element, and one with elements larger than the pivot. The pivot element is chosen in a way such that the partitioning divides the array roughly in half. The algorithm then recursively sorts each partitioned sub-array.

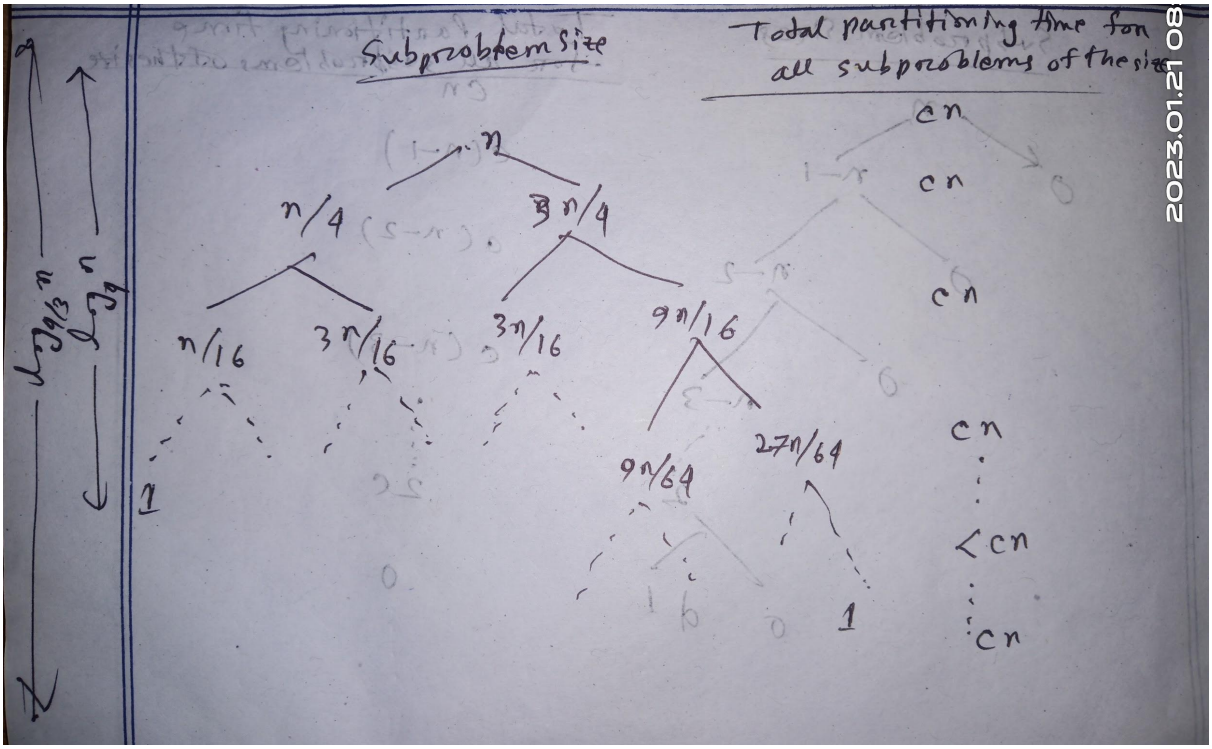
> Worst-case time complexity: O(n^2)

The worst-case time complexity of Quick Sort occurs when the pivot element is chosen as the smallest or largest element in the array, which causes one partition to be empty and the other partition to be the same as the original array. This means that the partitioning does not divide the array in half, but rather causes the size of one partition to be reduced by one element in each recursive call. In this case, the time complexity is O(n^2)

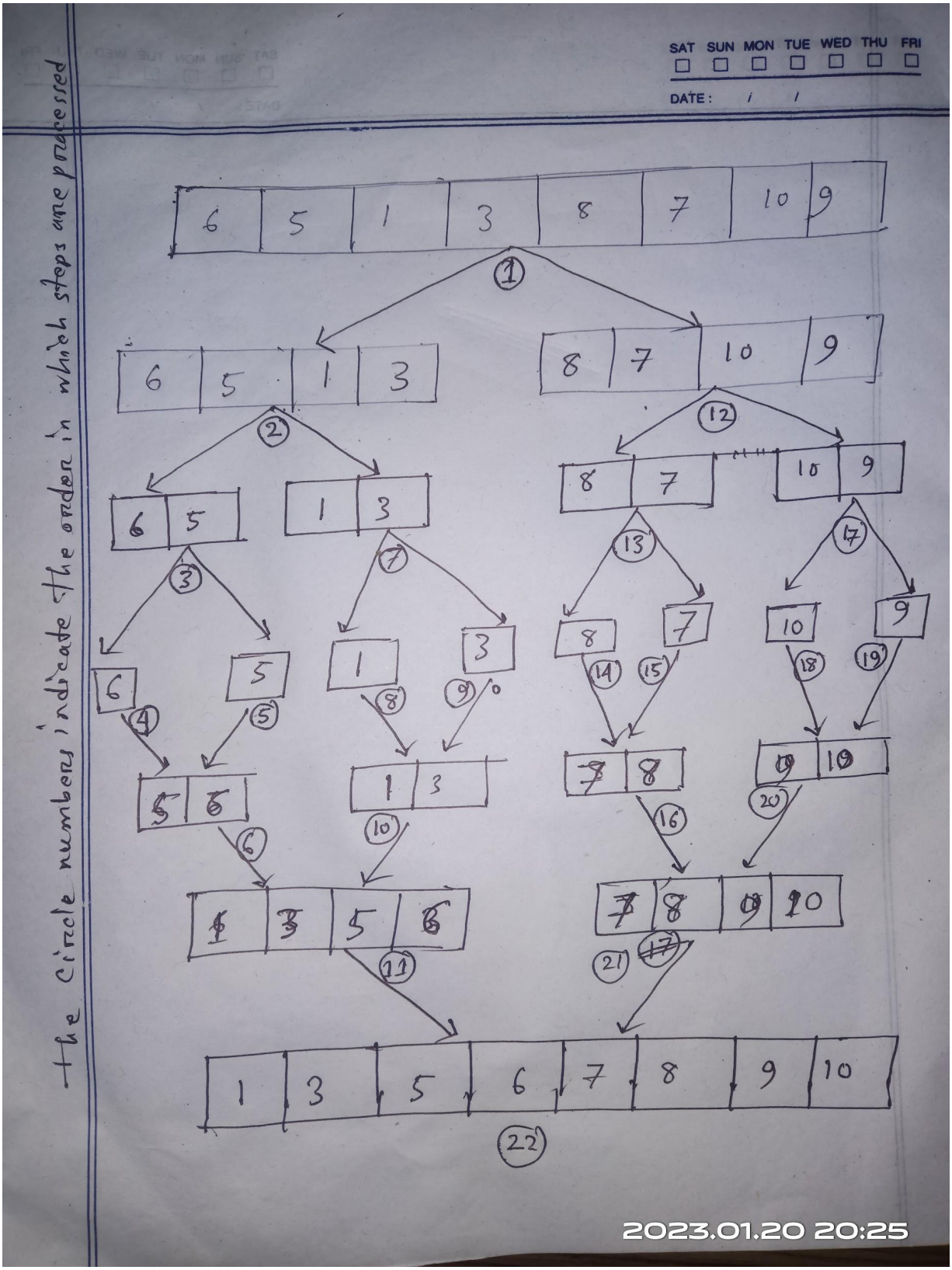


>Average-case time complexity: O(n log n)

In the average-case, the pivot element is chosen such that it divides the array roughly in half. The time complexity of the partitioning step is O(n), and the number of recursive calls is log(n) because the size of the array is halved in each recursive call. So, the total time complexity of Quick Sort in the average case is O(n log n)



Question No. 07
Draw the recursion call tree with return values for Merge Sort in the array [6, 5, 1, 3, 8, 7, 10, 9]
Answer No. 07
Drawing below the recursion call tree with return values for Merge Sort in the array [6, 5, 1, 3, 8, 7, 10, 9]: The merge sort algorithm works by recursively dividing the array into two halves, sorting each half, and then merging the sorted halves back together.


SAT SUN MON TUE WED THU FRI
DATE : / /
the circle numbers indicate the order in which steps are processed
6 5 1 3 8 7 10 9
1
6 5 1 3
8 7 10 9
2
12
6 5
1 3
8 7
10 9
3
7
13
17
6
5
1
3
8
7
10
9
4
5
8
14
15
18
19
5 6
1 3
7 8
9 10
6
10
16
20
1 3 5 6
7 8 9 10
11
21
1 3 5 6 7 8 9 10
22
2023.01.20 20:25

The recursion call tree for merge sort on the array [6, 5, 1, 3, 8, 7, 10, 9] would look like this:

The initial call to merge sort is made on the entire array [6, 5, 1, 3, 8, 7, 10, 9].

The array is then divided into two halves: [6, 5, 1, 3] and [8, 7, 10, 9].

The first recursive call is made on the left half of the array: [6, 5, 1, 3]

The left half of the array is then divided into two halves: [6, 5] and [1, 3]

The first recursive call on the left half is then made on the left half of the left half of the array: [6, 5]

The array [6, 5] is already sorted, so the left and right halves are merged back together: [5, 6]

The second recursive call on the left half is then made on the right half of the left half of the array: [1, 3]

The array [1, 3] is already sorted, so the left and right halves are merged back together: [1, 3]

The left and right halves of the left half of the array [5, 6] and [1, 3] are then merged back together: [1, 3, 5, 6]

The second recursive call on the original array is then made on the right half of the array: [8, 7, 10, 9]

The right half of the array is then divided into two halves: [8, 7] and [10, 9]

The first recursive call on the right half is then made on the left half of the right half of the array: [8, 7]

The array [8, 7] is already sorted, so the left and right halves are merged back together: [7, 8]

The second recursive call on the right half is then made on the right half of the right half of the array: [10, 9]

The array [10, 9] is already sorted, so the left and right halves are merged back together: [9, 10]

The left and right halves of the right half of the array [7, 8] and [9, 10] are then merged back together: [7, 8, 9, 10]

The left and right halves of the original array [1, 3, 5, 6] and [7, 8, 9, 10] are then merged back together: [1, 3, 5, 6, 7, 8, 9, 10]

So, the final sorted array is [1, 3, 5, 6, 7, 8, 9, 10]

Question No. 08

Write down the time complexity with proper explanation of the following code segment.

```
for (int i = 1; i * i <= n; i++) {
    if (n % i == 0) {
        cout<< i <<"\n";
        cout<< (n/i) <<"\n";
    }
}
```

Answer No. 08

The time complexity of this code segment is O(sqrt(n)).

The outer loop runs from 1 to the square root of "n" (i.e. i*i < = n), so it runs for sqrt(n) iterations. Within the loop, there is a single if statement that takes constant time to execute, regardless of the input size. Therefore, the total time taken by the loop is directly proportional to the number of iterations, which is sqrt(n).

Since the time complexity of an algorithm is represented by the function that describes the relationship between the input size and the running time, we can say that the time complexity of this code segment is O(sqrt(n)), where n is the input size.

Question No. 09

Suppose you are working in a project where you need to do many random memory accesses and binary search. Array vs Linked list which is more suitable in this case? Why?

Answer No. 09

In a project where I need to do many random memory accesses and binary search, an array would be more suitable than a linked list.

Arrays have a constant time complexity for random access, meaning that accessing any element in the array takes the same amount of time, regardless of the location of the element in the array. This is because all elements in an array are stored in a contiguous block of memory, and the memory address of each element can be calculated using a simple mathematical formula (i.e. memory address of element i = base address + i * size of element).

In contrast, linked lists have a linear time complexity for random access because in order to access a specific element, we need to traverse the list from the head, element by element, until we reach the desired element.

Regarding binary search, as long as the array is sorted, it can be done in O(log(n)) time, where n is the number of elements in the array. On the other hand, binary search is not possible on a linked list as its elements are not guaranteed to be contiguous in memory, and does not have an indexing system like an array.

In summary, an array is more suitable for the case where we need to do many random memory accesses and binary search because of its constant time complexity for random access and its ability to perform binary search in logarithmic time.

Question No. 10

Alice is using a singly linked list to implement undo-redo functionality in a text editor. Bob advised Alice to use a doubly linked list in this scenario. Which approach seems more suitable to you? Give a proper explanation.

Answer No. 10

I agree with Bob's advice. In the scenario where undo-redo functionality is being implemented in a text editor using a singly linked list, using a doubly linked list would be more suitable.

A singly linked list only has a single pointer to the next element in the list, so when we are traversing the list, we can only go in one direction (forward). This means that in order to implement undo functionality, we would need to keep track of the previous elements in the list in a separate data structure, such as a stack. This would add additional complexity to the implementation and increase the amount of memory used.

On the other hand, a doubly linked list has a pointer to both the next and previous elements in the list. This means that we can traverse the list in both directions, and undo functionality can be implemented by simply moving the current element pointer to the previous element. This approach is much simpler and more memory efficient.

Additionally, doubly linked list allows us to implement redo functionality too, as we can traverse the list in both directions, so when undo is performed, the current pointer points to the previous element and the next pointer points to the current element, so with the next pointer we can traverse back to the elements we undone.

In summary, using a doubly linked list would be more suitable for the scenario of implementing undo-redo functionality in a text editor because it allows for simple and memory efficient implementation of both undo and redo functionality.