T2A1-B: Workbook Part B

Q1

Linear Search

Description

Linear Search is a very simple search algorithm that works be checking each element of a list sequentially until the target element is found or the list ends (Ravikiran, 2024).

Example steps:

1. Starting with the first element of the list, compare the current element with the target element.
2. If they match, return the index of the element.
3. If they do not match, move to the next element and repeat the above steps.
4. If the target element is not found return a special value (e.g. -1) indicating that the element is not present in the given list.

Performance

- Best case time complexity: the best case for this algorithm is if the target element is found in the first element of the list. In this case Linear Search completes in constant time and is noted as $O(1)$. This means that the time to complete is always the same no matter the size of the data being searched.
- Worst case time complexity: the worst case for this algorithm is if the target element is not present. In this case every element must be searched and compared before completing with a non-result. Searching every single element in this way is noted as $O(n)$. This is known as linear complexity meaning that the time complexity will always linearly increase or decrease with the size of the data being searched.
- Average case time complexity: on average this algorithm will search and compare half of the elements in the given list. This means that the time complexity of the average case grows linearly as the average will increase as the size of the data being searched increases. This is noted as $O(n)$.
- Space complexity: the space complexity of this algorithm is $O(1)$ because the algorithm is able to function in-place, meaning that it does not require creating or modifying new data structures in order to function. It takes the given list and applies the steps directly to it. (Ravikiran, 2024)

Efficiency

The Linear Search algorithm is simple but very inefficient for large datasets due to its linear time complexity. It can be useful for small unsorted datasets. Edge cases like the target element being the first element (best case) or the target element not being present (worst case) significantly effect the performance (Ravikiran, 2024).

Binary Serach

Description

The Binary Search algorithm works by continually dividing the search interval in a sorted list in half until the target element is found (Yao, 2022).

Example steps:

1. The target element is compared with the middle element of a sorted list. If the target element is less than the middle element the search continues in left (lower) side of the list. If the target element is higher than the target element then the search continues in right (higher) side of the list.
2. The remaining space in the list is what is referred to as the search interval. The middle value of the search interval is then compared in the same way as the previous step.
3. The algorithm continues dividing the list down further and further until it finds the target element or confirms that the target element is not present, in which case it will return a non-result (e.g. -1).

Performance

- Best case time complexity: if the middle element of the sorted list matches the target element in the first comparison then the algorithm completes in constant time, $O(n)$.
- Worst case time complexity: this algorithm cuts the list in half with each iteration which results in a logarithmic time complexity meaning that after log n operations, only one element remains (and is either the target or not, in which case the algorithm returns a non-result). This is noted as $O($log $n)$.
- Average case time complexity: the average case of this algorithm also has logarithmic complexity. Regardless of the structure, as long as the data is sorted, the algorithm will keep halving the search space or search interval. The number of times this halving process needs to happen scales logarithmically, meaning that the dataset has to increase significantly for this number to go up. This is noted as $O($log $n)$.
- Space complexity: this differs depending on whether the version of the algorithm being used. The iterative version has a space complexity of $O(1)$ because no additional memory is needed to compute. The recursive version has a space complexity of $O($log $n)$ because the number of recursions will increase with the number of times the search space is halved, as discussed above. (Yao, 2022)

Efficiency

Binary Search is a very efficient algorithm for large sorted datasets. The main drawback is that it requires a sorted dataset so it is not applicable to dynamic or unsorted data. The edge cases, when the target element is exactly in the middle of the dataset (best case) or when not able to find it at all (worst case), still present a relatively stable performance because of the logarithmic nature of the algorithm. This means that even in the worst case Binary Search is still an efficient algorithm to use in large datasets (Yao, 2022).

Comparison

Linear Search is clearly more suited to small and unsorted datasets. This is because its linear complexity increases too much for larger datasets but stays low for small datasets. The Binary Search is almost the opposite. It must be used on sorted data to work and it is efficient with large datasets because of its logorithmic complexity.

Q2

Bubble sort

Description

Bubble Sort is an algorithm that works by repeatedly comparing and swapping adjacent elements if they are in the wrong order until the list is sorted in ascending order (the lowest value first and the highest value last) (W3Schools, 2023).

Example steps:

1. Starting with the first element, this element is compared to the second element. If the first element is larger it swaps places with the second element. If the second element is larger no changes are made.
2. This process repeats until an element is left in its place, meaning that element is sorted. Then, the algorithm moves onto the next element and begins comparing that one until it is sorted into its correct place.
3. Once each element has been sorted the algorithm is complete and the list is now sorted into ascending order.

Performance

- Best case time complexity: if the list is already sorted, Bubble Sort only makes one pass through the list, checking but not actually swapping any of the elements. This case has linear complexity and is noted as $O(n)$.
- Worst case time complexity: this occurs when the list is in reverse (descending order), meaning every adjacent pair must be compared and swapped. Each pass through performs $n - 1$ comparisons, reducing by one in each subsequent pass. The time complexity in this case is known as quadratic complexity and is noted as $O(n^2)$.

- Average case time complexity: in the average case, about half of the elements will need to compared and swapped. This gives the same complexity as the worst case: $O(n^2)$.
- Space compelxity: Bubble Sort operates in-place, it does not require any creating or modifying of new data structures except a constant use of memory to swap elements in the given list. This is noted as $O(1)$. (W3Schools, 2023)

Efficiency

Bubble Sort is an inefficient algorithm for large datasets becasue of its quadratic complexity. It becomes too impractical as the dataset grows. For smaller datasets, or mostly sorted lists which are closer to the best case, it can be quite efficient. This discrepancy in efficiency significantly reduces the number of plausible and efficient applications for Bubble Sort (W3Schools, 2023).

Merge sort

Description

Merge Sort works by dividing a list into smaller sublists until each list contains only one element. Then the lists are merged back together is a sorted order. This algorithm follows the 'divide and conquer' approach (Geeks For Geeks, 2024).

Example steps:

1. Divide: the given list is recursively split in half until each element is in its own sublist.
2. Conquer: the sublists are then sorted.
3. Merge: the sorted sublists are then merged together to recreate the original list in a sorted order.

Performance

- Worst case time complexity: the original list is split log n times (since each division reduces the problem size by half), and during each merge, n elements are processed. This is noted as $O(n$ log $n)$.
- Best case time complexity: Even if the list is already sorted, the algorithm will still divide and then merge the elements in the same way it would in the worst case. The time complexity is the same: $O(n$ log $n)$. This is also true of the average case.
- Space complexity: Merge Sort requires additional memory for the temoprary sublists that it will create during the merging process. This leads to a linear space complexity noted as $O(n)$. (Geeks For Geeks, 2024)

Efficiency

Merge Sort's efficiency remains the same across best, worst, and average use cases. This is a great advantage when working with large datasets. It works fine for smaller datasets but is not the best option. The main drawback of this algorithm is its space complexity that increases linearly with the size of the dataset. In memory constrained environments other algorithms could be better suited (Geeks For Geeks, 2024).

Comparison

Bubble Sort is best used in small datasets. It can quickly sort through a small list without using much memory. It is not well suited for large datasets at all. Merge Sort on the other hand is excellent for large datasets because of its logarithmic time complexity. This algorithm should always be used on large lists over an algorithm like Bubble Sort that is very inefficient with large lists. The only advantage Bubble Sort might have over Merge Sort is that it uses a fixed amount of memory which could be preferntial if memory is an issue over speed. Otherwise, if speed is the main issue and not memory, Merge Sort is much better suited.

References:

DSA - Bubble Sort (2023) W3Schools Online Web Tutorials. Available at: https://www.w3schools.com/dsa/dsa_algo_bubblesort.php (Accessed: 03 September 2024).

Merge sort - data structure and algorithms tutorials (2024) GeeksforGeeks. Available at: https://www.geeksforgeeks.org/merge-sort/ (Accessed: 02 September 2024).

Ravikiran, A.S. (2024) What is Linear Search Algorithm: Time complexity, Simplilearn. Available at: https://www.simplilearn.com/tutorials/data-structure-tutorial/linear-search-algorithm (Accessed: 04 September 2024).

Yao, D. (2022) Binary search, USACO Guide. Available at: https://usaco.guide/silver/binary-search?lang=cpp (Accessed: 05 September 2024).