T2A1-B - Workbook

Name: Wade Doolan
Student number: 12678

Question 1:

Identify and explain the workings of TWO sorting algorithms and discuss and compare their performance/efficiency (i.e. Big O).

The Bubble Sort algorithm

The following python code is an implementation of the bubble sort algorithm:

```
def bubble_sort(array):
    n = len(array)

    for i in range(n):

        already_sorted = True

        for j in range(n - i - 1):
            if array[j] > array[j + 1]:

                array[j], array[j + 1] = array[j + 1], array[j]

                already_sorted = False

        if already_sorted:
            break

    return array
```

(Real Python, 2020b)

Algorithm description

Essentially, the bubble sort algorithm uses an iterative approach by looping through the elements in a list multiple times and comparing the value of each element with the value of the adjacent element. If the adjacent element's value is less than the current element's value, the elements' positions are swapped. This continues until all the elements have been sorted in ascending order. This process involves using nested loops, with the inner loop working to push the higher valued elements to the right after each iteration. The larger elements are bubbled up with each iteration (Real Python, 2020b). The algorithm is explained in more detail below with an example.

Example: using the bubble sort algorithm to sort elements in the following list `python my_list = [1,7,3,9,2]`

The bubble sort algorithm involves:

1. accepting one argument, a list of values n elements long to be sorted. *In this case my_list.*
2. Once the list is passed into the function, the length of the list is assigned to the variable n. *The length on my_list is 5.*
3. The function then enters the outer loop, which is set to run from 0 to the last element's index value. *The last index value of my_list is 4.*
4. A boolean variable 'already_sorted' is set to True, this is used to make the function more efficient by exiting early if all the elements are sorted.
5. The algorithm now enters the inner loop, which is set to run a reducing number of times with every iteration of the outer loop. *Therefore, the first iteration of the outer loop means the inner loop will run (n-i-1) or (5-0-1 = 4) times. The inner loop will start at index 0 and go up index 3 in my_list.*
6. The purpose of the inner loop is to iterate through the elements and check if the value of an element is greater than the value of the adjacent element. If it is, then the elements are swapped. *The first complete run of the inner loop will result in 3, 7 and 9, 2 being swapped in my_list, with the 9 and 7 being shifted to the right of list*
7. If the elements are swapped, the variable 'already_sorted' is set as False to indicate to the outer loop that elements are still being sorted. *Since elements were swapped, 'already_sorted' is set to False.*
8. Once the inner loop has meet the required condition to exit (n-i-1), the algorithm begins the next iteration of the outer loop and so on until all the elements are sorted. *In the case of my_list, the inner loop has iterated 4 times and outer loop now begins the second iteration, starting the process for the inner loop over again; however, this time the inner loop will only iterate through the first three elements. That is (n-i-1) or (5-1-1 = 3) times. The process is repeated until my_list is sorted [1,2,3,7,9].*

Algorithm performance

The bubble sort algorithm includes two main structures, an inner for loop nested inside an outer for loop. Big(O) notation has been used below to analyse the algorithm complexity (number of steps) for worst case scenario:

Where n is the size of the list to be sorted:

- Before the outer loop begins, the length of the array is assigned to the variable n. This only happens once regardless of the size of the input array. This step has a Big O complexity of n(1) or constant and can be ignored as it has no bearing on the overall performance of the algorithm.
- The outer for loop will iterate n number of time. That is, as n increases, the number of iterations increase linearly. Therefore, the outer loop has a Big(O) notation on O(n).
- The inner loop will iterate by (n - the current iteration number of the outer loop - 1). Even though the inner loop reduces its number of iterations as the outer loop iterates it still increases linearly as n increases. As such, the inner loop also has a Big(O) notation of O(n).
- Since the inner loop is dependent on the outer loop, the algorithm has to make (n - 1) + (n - 2) + (n - 3) + … + 2 + 1 = n(n-1)/2 comparisons before it finishes executing, which can be simplified to \frac{1}{2}21 n^2 - \frac{1}{2}21 n. In relation to Big(O) analysis, constants don't change with the size of the array, as such they can be dropped leaving n^2 - n. Moreover, Big(O) notation is concerned with the term that will grow the fastest, as such the second term can be dropped, leaving n^2.
- Another way to analyse the result is to combine the Big(O) notation from both inner and outer loops: O(n) x O(n), which equals O(n x n) or O(n^2)
- Therefore, the bubble sort algorithm has a Big(O) notation of O(n^2) or quadratic complexity.

(Real Python 2020b; Hijazi 2021)

The Merge Sort algorithm

The following python code is an implementation of the merge sort algorithm:

```
def merge(left, right):

    if len(left) == 0:
        return right

    if len(right) == 0:
        return left

    result = []
    index_left = index_right = 0

    while len(result) < len(left) + len(right):

        if left[index_left] <= right[index_right]:
            result.append(left[index_left])
            index_left += 1
        else:
            result.append(right[index_right])
```

```
            index_right += 1

        if index_right == len(right):
            result += left[index_left:]
            break

        if index_left == len(left):
            result += right[index_right:]
            break

    return result


def merge_sort(array):

    if len(array) < 2:
        return array

    midpoint = len(array) // 2

    return merge(
        left=merge_sort(array[:midpoint]),
        right=merge_sort(array[midpoint:]))
```

(Real Python, 2020b)

Algorithm description

The merge sort algorithm uses a divide-and-conquer approach to sorting arrays by firstly breaking the array down into smaller and smaller parts before sorting and merging the parts back together to produce the sorted array. The merge sort algorithm involves the `merge_sort()` function recursively calling itself, with each iteration halving the array until only two elements remain (a left element and right element). After this, the `merge()` function is constantly called until all elements have been sorted and merged into one array (Real Python, 2020b). The algorithm is explained in more detail below with an example.

Example: using the merge sort algorithm to sort elements in the following list `python my_list = [1,7,3,9,2]`

The merge sort algorithm involves:

1. The `merge_sort()` function is called, passing my_list as the argument. That is, the list to be sorted.
2. The `merge_sort()` function now calculates the midpoint to be used to halve the list. *The midpoint for my_list is 2*.
3. The midpoint represents the index value used to break the list into two halves. And this is used to split the list and then call `merge_sort()` again. *In this case `merge_sort(my_list[:midpoint])` and `merge_sort(my_list[midpoint:])`, with the result from the first iteration being `[1,7]` and `[3,9,2]`*
4. This recursive halving process occurs until my_list has been broken down into individual elements. `[1] [7] [3] [9] [2]`
5. The `merge()` function is now called with the results of left and right halves. That is, the results from the final recursion of `merge_sort()` are passed as the left and right keyword arguments to the `merge()` function. *For example, `merge(left = [1], right =[7])`*
6. This process repeats as the code works it way back up and out of the previous recursion. Therefore, the `merge()` function is next called using the results from the earlier `merge()` function and so on until the list is sorted. *For example, at some point on the way back out of the recursion `merge(left = [1,7], right = [2,3,9])` is called and so on until the final sorted list is returned `[1,2,3,7,9]`.*

Algorithm performance

The merge sort algorithm can be analysed as two separate steps. Using Big(O) notation to analyse the step complexity for worse case scenario:

1. The `merge()` function, which uses a loop to merge two arrays. The length of the two arrays passed into the function can be combined to obtain an overall length of n. As n increases, the number of iterations to be performed by the loop increase in line with the change in n. Therefore, this step has a Big(O) complexity of O(n) or linear complexity.
2. The `merge_sort()` function, that splits the input list in two parts recursively and then calls the `merge()` function. This process involves the number of steps to be performed increasing as n increases in size, however, the rate of increase slows as n gets larger and larger. Notably, The overall number of steps required to split the list in half, down to a single element as n increases can be explained by:
    - Let 'a' represent the number of steps. Therefore, $a = n/2^a$, which equates to a O(log n) complexity.

(Real Python, 2020b)

Since the `merge()` function is combined with the `merge_sort` function, the Big(O) complexity for the merge sort algorithm is O(n) x O(log n) resulting in an overall complexity of O(n log n) (Real Python, 2020b).

Performance comparison between Bubble Sort and Merge Sort

Based on the Big(O) complexity discussed above, The merge sort algorithm has O(n log n) complexity, indicating there are fewer steps required to sort a particular array compared with the bubble sort algorithm $O(n^2)$. For example, to sort an array of 10 elements using the bubble sort algorithm it could take up to 10 X 10 = 100 iterations to sort the array, while the merge sort algorithm could take 10 X $\log_2 10 = 32$ iterations (approx.) to sort the array, making the merge sort algorithm more efficient.

Question 2:

Identify and explain the workings of TWO search algorithms and discuss and compare their performance/efficiency (i.e. Big O).

The Binary Search algorithm

The following python code is an implementation of the binary search algorithm:

```
def find_index(elements, value):
    left, right = 0, len(elements) - 1

    while left <= right:
        middle = (left + right) // 2

        if elements[middle] == value:
            return middle

        if elements[middle] < value:
            left = middle + 1
        elif elements[middle] > value:
            right = middle - 1

    return "No match found"
```

(Real Python, 2020a)

Algorithm description

Similar to the merge sort algorithm above, the binary search algorithm uses an iterative or recursive, divide-and-conquer approach to searching for a particular value in a list of values; however, for the algorithm to work, the list must first be sorted from smallest to largest value. Also, if there are duplicate values in the list, the algorithm will only return the position of the first instance. Once the array is sorted, the algorithm determines the location of the middle element in the array and checks if the middle element's value equals that of the search value. If the middle element is equal to the value, the

algorithm returns the location of the value and stops. If the middle element value is lower than the search value, the algorithm then repeats the search but only on the part of the list above the middle element. And if the middle element value is greater than the search value, the algorithm then repeats the search but only on the part of the list below the middle element. Essentially, the search process involves dividing the list in half after each search attempt until the desired value is found (Bee, 2020).

Example: using the binary ssearch algorithm to find the index for the value 10 in the following list `python my_list = [1,3,5,7,10,15]`

The binary search algorithm involves:

1. The `find_index()` function is called, passing my_list and the value 10 as arguments.
2. The first step involves determining the upper and lower bounds of my_list and assigning the index values to the variables left and right. *In this case, 0, 5*
3. The function then enters the iterative process of the algorithm, where the middle index value is calculated. *That is, (0 + 5)//2 = 2*
4. The middle index value is used to access the value in my_list at this point and check if this value is equal to the value being searched for. *Since the middle index value = 5, this equates to False and the functions moves to the next block of code.*
5. The middle element value is then checked to see if it is less than or greater than the search value. *Since the middle index value is 5, which is lower than 10, the left variable is recalibrated to 2 + 1 = 3 and effectively ignoring the lower half of my_list*
6. The while loop starts again but this time the middle index is calculated with the new left value. *Therefore, middle = (3 + 5)//2 = 4*
7. The element value at the new middle index position is checked to see if it equals the search value. *The value in my_list at index 4 is 10, which equals the search value, so the function returns the index value 4 and ends.*

Algorithm performance

The binary search algorithm includes a single loop structure. Big(O) notation has been used below to analyse the algorithm complexity (number of steps) for worst case scenario. Note, it is assumed the list has already been sorted in ascending order:

Where n is the size of the list to be searched:

- Since the number of elements being searched is effectively halved after each iteration of the loop structure, the number of steps required to find a given element can be explained as $a = n/2^a$, where 'a' is the number steps and 'n' is the number of elements in the list to be searched. Therefore, on the first iteration, the list is divided in half, on the second iteration it is divided into quarters and so on. The number of steps required can be expressed as $\log_2 n$ and has a Big O value of O(log n). As the value of n increases the number of steps increases, but at a decreasing rate, making the binary search algorithm an efficient search algorithm for large lists.

The Linear Search algorithm

The following python code is an implementation of the linear search algorithm:

```
def find_index(elements, value):
    for index, element in enumerate(elements):
        if element == value:
            return index
```

(Real Python, 2020a)

Algorithm description

The linear search algorithm works by looping over a list of elements, starting with the first element and working through each element until the value is found or the end of the list is reached. Essentially, the function above loops over elements in a list in a consistent order and only checks each element once until the value is found (Real Python, 2020a).

Example: using the linear search algorithm to find the index for the value 10 in the following list `python my_list = [1,3,5,7,10,15]`

1. The `find_index()` function is called, passing my_list and the value 10 as arguments.
2. The first step involves entering the for loop where the enumerate function is used to iterate through the first index and element in the list. *That is index 0 and element 1, with the value 1.*
3. The element value is then checked against the search value. *Since 1 is not equal to 10, the loop begins a second iteration.*
4. The second iteration of the loop performs a check against the second element and the search value. *The value 3 does not equal 10, so the loop begin the third iteration*
5. The third iteration of the loop performs a check against the third element and the search value. *The value 5 does not equal 10, so the loop begin the fourth iteration*
6. The fourth iteration of the loop performs a check against the fourth element and the search value. *The value 7 does not equal 10, so the loop begin the fifth iteration*
7. The fifth iteration of the loop performs a check against the fifth element and the search value. *The value 10 equals 10, so the loop stops, the function returns the index value 4 and ends.*

The linear search algorithm includes a single loop structure. Big(O) notation has been used below to analyse the algorithm complexity (number of steps) for worst case scenario (that is, the search element is at the very end of the list).

Where n is the size of the list to be searched:

- The number of steps required to complete the search using the linear search algorithm is directly proportional to the size of the list or n. As n increases in size, the number of iterations increase by n or linearly. The linear search therefore has a Big(O) notation of O(n).

Performance comparison between Binary Search and Linear Search

Based on the Big(O) complexity discussed above, the binary search algorithm is a more efficient searching algorithm, especially as the size of the list gets larger. The binary search algorithm result, O(log n) indicates there are fewer steps required to search for a value in a list compared with the linear search algorithm O(n). This is particularly true as the size of the list increases. However, it's important to note that the binary search algorithm requires the list to be sorted before the search begins.

References

Bee (2020). All You Need to Know About Big O Notation [Python Examples]. [online] Skerritt.blog. Available at: https://skerritt.blog/big-o/ [Accessed 1 Sep. 2022].

Hijazi, M. (2021). Big O Notation: Bubble Search in Unity - Level Up Coding. [online] Medium. Available at: https://levelup.gitconnected.com/big-o-notation-bubble-search-in-unity-c90ca1f608d0 [Accessed 27 Aug. 2022].

Real Python (2020a). How to Do a Binary Search in Python. [online] Realpython.com. Available at: https://realpython.com/binary-search-python/#iteratively [Accessed 1 Sep. 2022].

Real Python (2020b). Sorting Algorithms in Python. [online] Realpython.com. Available at: https://realpython.com/sorting-algorithms-python/ [Accessed 21 Aug. 2022].