dilloncotter_T2A1-B/DillonCotter_T2A1-B.md

Dillon Cotter T2A1-B - Workbook.

Efficiency of an app

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

Bubble sort:
This is possibly the simplest form of a sorting algorithm. It works by swapping elements that are side by side if they are in the incorrect order.
This process iterates over all elements until no more swaps are necessary.
EG. in an array [4, 2, 1, 3] to be placed into ascending order:

- Compares first and second element and swaps positions - [2, 4, 1, 3]
- Next swaps second and third element - [2, 1, 4, 3]
- Swaps third and last element - [2, 1, 3, 4] - largest element is now in correct position.
- Compares and swaps first and second element - [1, 2, 3, 4]
- Compares second and third element and determines swap is not required - [1, 2, 3, 4] - second largest element is now in correct position
- Compares first and second element and determines swap is not required - [1, 2, 3, 4] - 3rd largest and smallest elements are also in position.
  An example of this code in python:

```
def bubble_sort(arr: List[int]):
    # determine the length of array
    n = len(arr)
    # loop through each element in the array
    for i in range(n):
        # swap element positions if the left element is larger than the right element.
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

This sorting algorithm has to loop through each element and contains a nested loop. The best case is an already sorted list, in which case the algorithm only needs to run through the outer list once, giving complexity of O(n), however, given that it contains a nested loop that iterates for each element, the average and worst cases have complexity of O(n^2), as the algorithm has to loop through each element multiple times to ensure proper sorting.

Quick sort:
Quick sort is an algorithm that selects an element then 'pivots' other elements, placing smaller elements to the left and larger elements to the right.
This partition is then recursively applied on each side of the subarray(s).
Python code example:

```
def quick_sort(arr: List[int]):
   # if the list is 0 or 1 element, return the list directly
   if len(arr) <=1:
       return arr

   # get the middle element of the list and store this number as 'pivot'
   pivot = arr[len(arr) // 2]
   #initialise empty lists to append values to depending on their relationship to 'pivot'
   left = []
   middle = []
   right = []

   # loop through each element
   for i in arr:
       # determine which list
       if i < pivot:
           left.append(i)
       elif i == pivot:
           middle.append(i)
       else:
           right.append(i)

   # return the result including recursion of the left and middle
   return quick_sort(left) + middle + quick_sort(right)
```

In this example and using an example [10, 80, 30, 40, 90]:

- Select the pivot = 30 (middle number)
- loop through the array, and place each element to the correct array according to their relationship:
  left = [10]
  middle = [30]
  right = [80, 40, 90]
- recursively call quick_sort:
- for quick_sort[10] the array is length = 1, so returns [10]
- for quick_sort[80, 40, 90] we complete the quick sort again:
  left = []
  middle = [40]
  right = [80, 90]
- recursively call quick_sort:
- quick_sort [] the array is length = 0, returns []
- quick sort [80, 90] we complete the sort again:
  left = [80]
  middle = [90]
  right = []
  with a final recursion each array is now either 0 or 1 element in length, and as a result will output a concatenated list [10, 30, 40, 80, 90]
  The average time complexity for this is O(N log(N)), however as it is possible that when the array splits one part could consist of all except 1 element (middle), and each recursion having the same output (one part has 0 elements, middle element contains the pivot, and right contains all elements except middle).
  In the worst case, this means the time complexity *could* be O(N^2)

Comparison
Bubble sort is generally slower and less efficient than quick sort, however it is possible depending on the array, that both sorting alorithms would have the same time complexity of O(n^2)

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

Linear Search
Linear searching goes through each element in a list one by one until it finds the element you are looking for.
While this could result in O(1) time complexity, when the item is the first element, this is more commonly (and in worst case) O(n) complexity.
EG:

```
def linear_search(arr: List(int), target: int):
    for i in range (len(arr)):
        if arr[i] == target:
            return i
    return 'x'
```

In the above example the code loops through each element in the range of the array, and when the number of the array is equal to the target, return the position of the element in the array

Binary Search
Binary searching works on sorted elements by a divide and conquer algorithm. It begins by checking the middle element works out the relationship between the middle element and target, then searches the half the element will be in, by performing this operation multiple times.
EG.

```
def binary_search(arr: List[int], target: int):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return 'x'
```

In the above code, we first define a low and high value, then loop while the low value is less or equal to high value.

- First define a mid point which will be equal to half of low + high.
- If this number is the target, end and return the position, otherwise
- If the mid is less than the target, set the low value to mid value + 1, or
- If the mid is greater than the target, set the high value to mid value -1.
- Loop this until the mid value = target or low value is higher than the high value (then return anything that the code outside the function can use to compare).

As with Linear searching, it is possible that if the middle element is the target, the complexity is O(n), in the average case (and worst case) the complexity is O(log n).

Comparison
While binary searching on average, and in worst case is faster as it is eliminating half of the array at each step, it requires the elements to first be sorted.