Workbook Assessment T2A1 - B

Q1 - Identify and explain the workings of TWO sorting algorithms and discuss and compare their performance/efficiency (i.e. Big O)

Sorting Algorithms are used in code to assist with ordering input data. If you have Natural Data proliferate into an array, it may not be in a usable format till it is reordered. A Sorting Alogrithm can do this.

One Sorting Algorithm is called SELECTION SORT. This algorithm works buy by starting with the first item in the array and going through each other item in the array and doing a comparison for the lowest value item. If the 2 items are out of order, they switch places. The Algorithm then moves onto the 2nd item in the array and repeats the comparision process.

For example If you had an array of 4 numbers, out of sequence such as (5 , 8 , 2 , 6) The first step of the algorithm would be to compare the Number 5 with 8, then 2, then 6. Upon seeing that 2 was lower than 5, the numbers would swap places creating an array (2 , 8 , 5 , 6) and the number 6 would not be assessed yet. The next iteration would be to move to the second item in the array and repeat the process. In this case comparing 8 to numbers 5 and 6. When seeing 5 is less than 8, they would swap position and the new array would be (2 , 5 , 8 , 6) A third iteration would have 8 assessed against 6 and swapped resulting in a sorted array of (2 , 5 , 6 , 8).

Below is an example

```
def selection_sort(array)
  for i in 0..array.length-1
    min = i
    for j in i+1..array.length-1
      if array[j] < array[min]
        min = j
      end
    end
    array[i], array[min] = array[min], array[i]
  end
  return array
end

(Altcademy n.d)
```

The industry accepted practice for determining algorithm efficiency and performance is Big O Notation. Things taken into account are 1. the size of the input into the algorithms, 2. Worse case scenario of how long it might take to perform the designed action and 3. The number of operations it will take for the algorithm to run to completion.
This assessment is indepenant of machine performance and there are 7 levels of Big O efficiency you can rate an algorithm. In order from least to most complex they are, O(1) < O(log n) < O(n) < O(n*log n) < O(n^2) < O(2^n) , O(n!)

The sort algorithm SELECTION SORT has big O notation of O(n^2) otherwise called Big O squared or quadratic. Most commonly applied to nested loops. The first command of the algorithm is to go through all data points in the array. This by itself would be O(n). Then inside each iteration of that loop, there is a second loop asking to compare the present data point with every other data point in the array until it does or does not find a data element of lower value. This second command loop by itself would also be considered a O(n), however because it is nested with the original command loop it results in the following O(n)xO(n)=O(n^2)

Another Sorting Algorithm is called MERGE SORT. This algorithm is a two stage system. It works by spliting the array of information in half over and over till it is a collection of single data arrays. Then it merges the individual data points back together and in each action of merging 1+1=2 then 2+2=4 it will also order the data. The end result is the original array but sorted.

For example If you had an array of 4 numbers, out of sequesnce such as (5 . 8 , 2 , 6), first step would be to split them to two arrays of (5 , 8) AND (2 , 6). Then again to (5) (8) (2) (6). Once at this stage the process of mergesort begins and when bringing back 5 and 8, they are compared and ordered. Same with 2 and 6. Lastly mergesort brings back (5,8) and (2,6) ordering it into (2,5,6,8).

Below is an Example.

```
  def merge_sort(array)
   if array.length <= 1
     return array
   end

   array_size = array.length
   middle = (array.length / 2).round

   left_side = array[0...middle]
   right_side = array[middle...array_size]

   sorted_left = merge_sort(left_side)
   sorted_right = merge_sort(right_side)

   merge(array, sorted_left, sorted_right)

   return array
 end

 def merge(array, sorted_left, sorted_right)
   left_size = sorted_left.length
   right_size = sorted_right.length

   array_pointer = 0
   left_pointer = 0
   right_pointer = 0

   while left_pointer < left_size && right_pointer < right_size
     if sorted_left[left_pointer] < sorted_right[right_pointer]
       array[array_pointer] = sorted_left[left_pointer]
       left_pointer+=1
```

```
    else
      array[array_pointer] = sorted_right[right_pointer]
      right_pointer+=1
    end
    array_pointer+=1
  end

  while left_pointer < left_size
      array[array_pointer] = sorted_left[left_pointer]
      left_pointer+=1
      array_pointer+=1
  end

  while right_pointer < right_size
     array[array_pointer] = sorted_right[right_pointer]
     right_pointer+=1
     array_pointer+=1
  end

  return array
end

(dev 2020)
```

The sort algorithm MERGE SORT has big O notation of O(Logn) otherwise called Logarithmic. This is because the very code of the Algorithm halves the Array until it is in individual data points. This makes performing the required task, which in this case is sorting, quicker. Sorting 8 numbers together, for example, would take a lot more computations just like in the above SELECTION Sort method because you would be constantly comparing each one to each other over and over in loop and nested loop. In Merge sort, you just have to sort 2 numbers, then 2 groups of 2 numbers, then 2 groups of 4 numbers. Significantly less computations to achieve the same goal.

Comparison

It is hard to compare the two algorithms as they seem to be very far away from each other on the scale of time complexity. Looking at the Big O notation of Selection Sort which is O(n^2) and looking at Merge Sort which is O(Logn), both can be applied to non ordered, multi dimensional arrays. But as the numbers go up, so does the time and computations required and because you are comparing logarithmic and quadratic algorithms, the time difference for running is almost exponetial. For Example, The run time difference between Merge and Selection can be 50x to 500x and the skys the limit. For any application, including a marketplace app, the Merge Sorting application, whilst more complex to code than the sorting application, is the proven choice .

Q2 - Identify and explain the workings of TWO search algorithms and discuss and compare their performance/efficiency (i.e. Big O)

Search Algorithms are used in code to assist with searching for specific data needed by the user. There are different algorithms that can do this but the main goal is to perform searches as time efficiently as possible. With large databases or Arrays with data in the Millions, time efficiency becomes very important.

One Search Algorithm is called BINARY SEARCH. Starting with an sorted array, this algorithm halves the size of the array each iteration, removing unrequired data from each search. It sets a midpoint of the array, and compares the midpoint data with the target data. If the target data type is lower than the midpoint data, then the algorithm removes ever piece of data from the right hand side of the midpoint and performs a second iteration and so on and so forth till the target data is found.

For example If you had an ordered array made up of the following (1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11) and a target of 5, first step would be to find the midpoint which is 6. The algorithm would then assess whether 6 is higher or lower than the target of 5. As it is higher this time, then all data to the right of 6 would be removed. Now the second iteration of the algorithm would be looking for target 5 again but the array would look like (1 , 2 , 3 , 4 , 5 , 6). The new midpoint would now be either 3 or 4 as it is not a whole number. If we set it to round up then midpoint would be 4. Comparing target of 5 to midpoint of 4 will lead to the algorithm removing all data to the left of midpoint. The new Iteration would have an Array of (4 , 5 , 6). New Midpoint would be 5 and this woul dmatch target of 5. This would show required data type of 5 is in index position 4 of the original Array. The end result is only 3 iteration to search through 11 data points in an array.

Below is an example

```
  def binary_search_iter(arr, el)
  max = arr.length - 1
  min = 0

  while min <= max
    mid = (min + max) / 2
    if arr[mid] == el
      return mid
    elsif arr[mid] > el
      max = mid - 1
    else
      min = mid + 1
    end
  end

  return nil
end

(dev 2021)
```

The search algorithm BINARY SEARCH has big O notation of O(Logn) otherwise called Logarithmic. This is because for every opertaion the quantity of elements being searched is being halved. All the way down to the result. This means no matter the increase in array size, the amount of calculations relative remains the same.

Another Search Algorithm is called LINEAR SEARCH. This method of searching does NOT require a sorted array, however will continue to repeat until it finds the target data element. For a large database or array in the millions, this would be very time consuming. The process is as follows, the first element is compared to the target element, if it is the same then the search ends. If is is not the same then the algorithm moves to the second element and repeats the comparison so on and so forth untill the target element is found.

For example Lets start with an array of (37 , 45 , 3 , 56 , 5 , 11). If the target data element was 3 then the algorithm would start its first iteration with comparing 37 to 3. As it is not the same, then the second iteration would be comparing 45 to 3. Same negative result so the third Iteration would be comparing 3 to 3. As that is a match, the algorithm comes to an end having found the required data in the array. As mentioned before not very efficient when dealing with large volumes of data.

Below is an example

```
    def linear_search(target, array)
    counter = 0
      while counter < array.length
        if array[counter] == target
          return counter
        else
          counter += 1
        end
      end
    return nil
    end

  (freeCodecamp 2020)
```

The search algorithm LINEAR SEARCH has big O notation of O(n). This is because there is only one command line in loop, which is to compare the element in an array to what you are looking for. O(n) means that the amount of computations required, worse case scenario, is equal to the amount of elements in your data array.

Comparison

Comparing a Linear Search algorithm to Binary search algorithm they have seperate strengths. A Linear search has a higher Big O complexity than a Binary search, more computations are performed to get the same result, however the Binary Search is not as adaptable. A Binary Search algorithm will get to the answer faster, and this becomes very evident as the volume of information needed to go through gets higher and higher. However as this is a RfQ Response for a marketplace app project, other factors have to be taken into consideration. A Binary Search needs the input data to be in Sorted order, whilst a linear search does not. A Binary Search can only work for a single dimensional array, yet a Linear Search can be used with Multidimensional arrays. They Linear search performes equality searches whils tBinary Search performs Ordering Comparisons. So as the user of a marketplace app would be entering Key information to search for in order to create a display of options on the user interface, using Linear Search Algorithm will be the only functional way to go.

Reference

Altcademy n.d, 'Selection Sort Algorithm in Ruby in Ruby', viewed 15 June 2022, https://www.altcademy.com/codedb/examples/selection-sort-algorithm-in-ruby

Megan 2020, 'Merge Sort in Ruby', Dev Blog, Web log post, 25 April, viewed 15 June 2022, https://dev.to/mwong068/merge-sort-in-ruby-28n1

meks 2021, 'Binary Search in Ruby', Dev Blog, Web log post, 9 January, viewed 15 june 2022, https://dev.to/mmcclure11/binary-search-with-ruby-3220

freeCodecamp 2020, 'Linear Search Explained', 27 January, viewed 15 June 2022, https://www.freecodecamp.org/news/linear-search/