Workbook - T2A1-B - Nathan Blaga

Question 1:

Sorting algorithms take an array as input and return a sorted array as output. This is achieved through a series of instructions that perform specific operations on the given array (Moore, Pilling & Khim). Bubble sort and Insertion Sort are considered two basic sorting algorithms.

Bubble Sort

Bubble Sort is a simple algorithm used to sort lists and arrays. The below bubble_sort method takes an attribute (an unsorted array) and performs the following functions as discussed below.

```ruby
# Bubble Sort
def bubble_sort(array)
   # Method takes one attribute of a given array
   return array if array.size <= 1
   # If the array contains 0 or 1 elements, return the given array
   swap = true
   # Create a variable swap and set default to true
     while swap
       # While loop occurs while swap is true
       swap = false
       # Set swap to false
       (array.length - 1).times do |x|
           # Iterate through all elements in array
         if array[x] > array[x+1]
           # Starting from the most left element, compare to the closest
element to it's right
           array[x], array[x+1] = array[x+1], array[x]
           # If element x is higher than element x+1, swap the two elements
around, else leave them in the same position
           swap = true
           # If a swap occurs, set variable swap to true.
         end
       end
     end
   array
   # Return array once loop ends and all swaps have been made
 end

array = [34,82,16,46,4,13]
p bubble_sort(array)
# Reference (Michelle 2017)
```

The code will first compare element zero [34] and element one [82]. It then runs a conditional statement that swaps the two elements if the first element is greater than the second. Element zero [34] is less than element one [82], therefore no swap occurs. The code then repeats the loop and compares element one

[82] with element two [16]. Element one is greater than element two and the pairs are swapped within the array. The code returns to the top of the loop and repeats the above steps with the remaining elements. Once the variable swap remains false, the loop is terminated and all elements within the array are ordered from smallest to largest.

In regards to the complexity of the bubble sort, it has the worst-case of O(n²) and best case of O(n). This can be examined using the above example. The buble_sort method consists of two loops, an outer while loop that iterates through the array until there are no elements left to be swapped, and an inner loop that iterates through pairs of elements and sorts them by a conditional statement. The inner loop is defined by length -1 (the entire length of the array) hence it is of linear complexity O(n). If the inner loop is iterating n times to sort out the array, then the while loop is also iterating through the array n times. Hence the while loop has a linear complexity O(n) resulting in the bubble sort having n * n or a worse case of O(n²) quadratic complexity. For the bubble sort to be O(n) (best case) then the array would need to already be sorted, and once executed, the bubble sort would only have one pass of the array with no swaps.

Insertion Sort

Insertion Sort is a sorting algorithm that iterates through an array and swaps each element individually (Moore, Pilling & Khim). The below insertion_sort method takes an attribute (an unsorted array) and demonstrates how the algorithm functions within ruby code.

```ruby
# Insertion Sort
def insertion_sort(array)
    # Method takes one attribute of a given array
    (array.length).times do |j|
    # Iterate through all elements in the array, denoting j as the index
      while j > 0
        # Run a While loop as long as k is greater than index 0 (first
element of array)
        if array[j-1] > array[j]
        # If previous element is greater than j (current element):
          array[j], array[j-1] = array[j-1], array[j]
        # Swap j and j-1  around
        else
          break
        #  Condition breaks if previous element is not larger than current
element
        end
        j-=1
        # Decrease j by 1
      end
    end
    array
  end

array = [34,82,16,46,4,13]
p insertion_sort(array)
# Reference (Michelle 2017)
```

The code first runs a loop that assigns the variable 'j' as an index for the array. It then selects the first element that has an index greater than zero. In the above example, this is [82] not [34] as insertion sort relies on comparing an element to all elements in the array that are left of its position. The code then compares element one [82] denoted as j, with element zero [34] denoted as j-1, and runs the condition statement; if j-1 is greater than j, swap the elements around. Because [34] is less than [82], the code does not swap the two elements around. The if/else statement is terminated and the j counter is decreased by 1. The code then repeats all steps above until it iterates through the entire array and swaps all elements into a sorted array.

Similar to Bubble Sort's Big O complexity, Insertion Sort has the worst-case of $O(n^2)$ and best case of $O(n)$. A simple example to explain this is an array that starts in descending order. The code would need to swap and compare every single element within the array, simplified $O(n^2)$. As such, Insertion Sort has a best case of $O(n)$ when the array is already sorted. The code will run through the array once, make no swaps and terminate. Both Insertion Sort and Bubble Sort share the same Big O runtime of $O(n^2)$ - as elements in an array increase to n, their execution time grows at the same rate of n. Both algorithms achieve their best case $O(n)$ when dealing with a small array that has a limited number of elements requiring sorting, such as array = [1,0,2,3,4]. Even though the two algorithms have the same Big O complexity, this does not indicate identical performances. This can be examined during each pass of the array. Bubble Sort swaps all remaining unsorted elements within the array, while Insertion Sort swaps a value into a group of already-sorted elements, halting once completed (Puryear 2017).

Question 2:

"A search Algorithm is the step-by-step procedure used to locate specific data among a collection of data" (Techopedia 2017).

Search Algorithms are generally classified into two categories; sequential search (Linear Search) and Interval Search (Binary Search) (GeeksforGeeks 2020).

Linear Search

Linear Searches traverses an array or list and sequentially checks every element until a match is found (GeeksforGeeks 2020).

```
# Linear Search
def linear_search(list, n)
    # Method takes two attributes; an array of values and target number
denoted by n
        i = 0
        # Start with index 0
    while i < list.size
        # Loop through array while i is less then list size
        return n if list[i] == n
        # Return n(target number) if the element in the array is equal to
n(target number)
        i += 1
        # Whenever an element in the array does not equal to n(target
number) increase index by 1 and restart the loop
        # This prevents an endless loop
```

```
    end
        false
        # After looping through the entire array, if no element equals to
n(target number), return false
end

p linear_search([1,2,3,4,5],4)
# Reference (Castello 2019)
```

The linear_search method takes two attributes; an array or list of values, and the value n (the targeted value required to be found). A variable of 'i' is set to 0 which denotes an index used for the while loop. A while loop is then run and will continue to iterate through any given loop as long as the index of "i" is below the array size. A conditional statement is then set for every iterated element of the array. If that element is equal to n, the code will then return n. For every loop that does not match an element with n, i will increment by 1.

Big O complexity of Linear search has a best case of O(1). This occurs when the target number is found at the first position within the array, resulting in just one comparison. The worst case occurs when the target number is found at the last position or not present within the array at all. If found at the last position, the search is terminated successfully with n comparisons. However, if the target number was not found at all, the search would terminate in failure with n comparisons. Therefore, in a worst case scenario, Linear Search takes O(n) operations.

Binary Search

Binary Search traverses an array or list by repeatedly dividing the search interval in half. It then checks all elements within the divided array until the value is found or the interval is empty (GeeksforGeeks 2020).

```
# Binary Search
def binary_search(list, n)
    low = 0
    # First index (left side of the array) is denoted as low
    high = list.size
    # Second index (right side of the array) is denoted as high

    loop do
        # Conditional loop in which it will only end when it satisfies one
of the below conditions
        mid = (low + high) / 2
        #  To find the middle of the array, a variable mid is assigned to
the whole array

        return n    if list[mid] == n
        # First condition returns the n(target number) when the loop
iterates through the middle of the array and n(target number) is found
        return false if list[mid] == nil
        # Second condition returns nil if code has gone outside the array
due to (n) not existing within the array
        return false if (high - low).abs == 1
        # Third condition returns false when the array only has one element
```

```
  and (n) is not found
          if list[mid] > n
              # If the mid point of the array is higher than n(target number)
              high = mid
              # Loop through elements in low section of the array
          else
              # If the mid point of the array is lower than n(target number)
              low = mid
              # Loop through elements in high section of the array
          end
      end
  end

  p binary_search([5,7,16,23,32,56,73,98],16)
  # Reference (Castello 2019)
```

The binary_search method takes two attributes; an array or list of values, and the value n (the targeted value required to be found). The two variables of low and high are assigned to the left and right of the array. A loop is then run where the midpoint of the array is defined by (low + high)/2. The algorithm divides the array into two, defining the midpoint as; [8/2] = element four [32]. Conditional statements then compare [32] with [16]. Since [32] is larger, the entire high end of the array is removed, leaving a new array of [5,7,16,23]. The code then returns to the top of the loop and runs the same above steps on the new array, and continues to divide the current array in half until the target number (n) is matched within the array.

Big O complexity of Binary Search has a best case running time of O(1)(constant). This occurs when the target number is the first midpoint selected. In the above example, this would be demonstrated if the target number (n) was 32. Binary Search has a worst-case and average-case running of O(log n). When n is the number of elements within the array, it makes O(log N) comparisons. The worst-case occurs when the target element is not present within the array. In other words how many times can n be divided in half to reach the last number (worst-case). The answer to this question is log N. The algorithm can divide n by 2 log N times until the array is down to one element.

Compared to a Linear Search, a Binary Search is a significant improvement as it can cut a list in half on every iteration rather than sequentially searching the entire array one element at a time. Linear Search depends on n size of the array and so for a small dataset, this is acceptable. However Binary Search is more effective and efficient, especially when reaching high volumes of data. One drawback of Binary Search is that it assumes the middle value in the array contains the median of the array and so the array must be sorted. If the array was not sorted, the desired operation could not be performed in O(log N).

Reference List:

Castello, J 9 Novemeber 2019, Binary Search & Linear Search: Algorithms With Ruby, youtube.com, RubyGuides, viewed 5 February 2021, https://www.youtube.com/watch?v=7RPr83FYEkE&ab_channel=JesusCastello

GeeksforGeeks, 17 September 2020, Searching Algorithms, geeksforgeeks.org, viewed 5 February 2021, https://www.geeksforgeeks.org/searching-algorithms/

Michelle, 30 August 2017, Read it, Learn it, Build it:Sorting Algorithms in Ruby,medium.com, viewed 5 February 2021, https://medium.com/@limichelle21/read-it-learn-it-build-it-sorting-algorithms-in-ruby-ead04b04baa6

Moore, K, Pilling, G, Khim, J ND, Sorting Algorithms, brilliant.org, wiki, viewed 5 February 2021, https://brilliant.org/wiki/sorting-algorithms/

Puryear, M, 27 April 2017, Why is insertion sort faster than bubble sort while having the same big 0 notation, quora.com, viewed 5 February 2021, https://www.quora.com/Why-is-insertion-sort-faster-than-bubble-sort-while-having-the-same-big-O-notation

Techopedia, 4 January 2017, Search Algorithm, techopedia.com, definition, viewed 5 February 2021, https://www.techopedia.com/definition/21975/search-algorithm