Search and Sort Algorithms

Linear Search

A linear search is a sequential search - the algorithm begins at one end of a list of data and works its way through, checking whether each item in terms is the required item. If the required item hasn't been found by the end of the search, an error is returned.

A linear search is an example of a 'Brute force' technique - i.e. it relies on raw computing power, rather than using a clever approach to solve the problem quicker. Linear searches are appropriate when searching data which is not ordered in any way.

A linear search is not very efficient. The best case for a linear search is that you find the item you are looking for with the first search, but the worst case is that you have to search every possible location. Therefore, for 100 items, on average a linear search would require: (100 + 1) = 50.5 search steps.

Question 1 - On average, how many search steps would be required if there were 400 search items?

In words, the linear search algorithm is:

1. If the list is empty, stop
2. Start at the beginning of the list
3. Compare the list item with the search item
4. If they are the same, stop, reporting found
5. If they are not the same, move on to the next item
6. Repeat steps 3 to 5 until the end of the list is reached
7. If the end of the list is reached without finding the item, report that the item hasn't been found.

In Python, a linear search looks like this:

```
# Python3 code to linearly search x in arr[].
# If x is present then return its location, otherwise return -1

def search(arr, n, x): #This subroutine performs the search

    for i in range (0, n):
        if (arr[i] == x):
            return i;
    return -1;

# Code to start the algorithm and run the subroutine
arr = [ 2, 3, 4, 10, 40 ]  #The array to search
x = 10
n = len(arr)
result = search(arr, n, x) #This calls the subroutine, result gets
                           #the value that the search subroutine
                           #produces
if(result == -1):
    print("Element is not present in array")
else:
    print("Element is present at index", result)
```

Question 2 - how would you alter the above code so that it asked the user which number they would like to search the array for?

Question 3 - What does the line `n = len(arr)` do, and why is it necessary?

Question 4 - In the line `def search(arr, n, x):` what is the technical name for arr, n and x?

Binary Search

A binary search is much faster than a linear search, however it can only be applied to a list that has been sorted into order. It uses a ‘divide and conquer’ strategy - that is, it keeps dividing the possible locations for the search item in half, rapidly reducing the places left to search.

The search works as follows:

1. Select the median item on the list (that is the ‘middle’ value)
2. Compare this item to the search item
3. If the search item is higher , discard the median and the items lower than it on the list
4. If the search item is lower, discard the median and the items higher on the list.
5. Find the new median
6. Repeat the process until the search item is found, or, if the list becomes empty, stop and report that the search item isn’t present.

The Binary search is quite efficient. For a list of 100 items the fast it could be would be 1 search step, whilst the lowest would be 7. The average steps would be around 6, and usually several steps are used to find the item.

Question 5 - What would be the maximum number of steps required to find an item using a binary search if the initial list contained 400 items?

Question 6 - If you double the number of items in a search list for a linear search, by how much does the maximum search time increase?

Question 7 - If you double the number of items in a search list for a binary search, by how much does the maximum search time increase?

Question 8 - Given the binary search is so much faster than a linear search, give a circumstance under which we might still use a linear search:

Question 9 - In the above case, what could we do so that the binary search could be used? Why might we not use this approach?

Question 10 - Show the steps of a binary search if we were looking for the number 23 in the following list:

2,5,9,15,18,20,23,26,31,43,46,82,85,87,90

A Python implementation for a binary search is shown here:

```
# Returns index of x in arr if present, else -1
def binarySearch (arr, l, r, x):

    # Check base case
    if r >= l:

        mid = l + (r - l) // 2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it can only be
        # present in left subarray
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)

        # Else the element can only be present in right subarray
        else:
            return binarySearch(arr, mid + 1, r, x)

    else:
        # Element is not present in the array
        return -1

# Code that calls the search subroutine
arr = [ 2, 3, 4, 10, 40 ]  # array to search
x = 10

# Function call
result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
    print ("Element is present at index: ", result)
else:
    print ("Element is not present in array")
```

EALING INDEPENDENT COLLEGE

Bubble Sort

Bubble sort is a way of sorting data into order. In a bubble sort, we compare two items which are next to each other, and if they are not in the right order we swap them over. We then compare the next two items, and the next two, until we reach the end of the list, at which point we start again with the first two items. If we complete a full 'pass' without doing any swaps the items have been sorted.

The Bubble sort is a relatively easy algorithm, but it isn't very fast. However, it doesn't take up very much storage space in memory, which can be important.

Question 11 - Bubble sort the following numbers:

8,3,6,1,3,7,9,2

A rough bubble sort algorithm is as follows:

```
SortArray = [5,8,2,9,4,6]
LengthofData = length(SortArray)
HowMany = LengthofData
Passes = 0

NoSwap = False
while NoSwap == False and Passes < LengthofData:
    NoSwap = True
    for i in range(0,HowMany):
            if SortArray[i]>SortArray[i+1]:
                    Temp = SortArray[i]
                    SortArray[i] = SortArray[i+1]
                    SortArray[i+1] = Temp
                    NoSwap = False
    HowMany = HowMany - 1
    Passes = Passes + 1
```

Or another version in Python:

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

Question 12 - This algorithm is not as efficient as the one above. Describe the ways in which it is less efficient:

EALING INDEPENDENT COLLEGE

Merge Sort

The merge sort algorithm breaks a list down into smaller and smaller lists, splitting them in half each time, until every list has only one item in it. Then it merges pairs of lists in turn, creating lists of two, then four, then eight, etc. until there is once again only one list and it is sorted in the correct order.

The merge sort is much faster than a bubble sort, but it does take up a lot more working memory to complete.

Question 13 - Merge sort the following items:

8,3,6,1,3,7,9,2