CSX3001/ITX3001
FUNDAMENTALS OF COMPUTER PROGRAMMING

CLASS 13 SORTING AND SEARCHING ALGORITHM

PYTHON

SORTING ALGORITHM

There are lots of sorting algorithms available for you to use. Each of them has a specific advantage depending on the characteristic of the data collection you are working on.

We also be able to use sort method that is implemented from Python developer. That's the easy way but! do you know what exactly happen behind the scene?

Let's see the straightforward sort algorithm example.

Bubble Sort

Lecturer will explain how this sort works.

```
def bubbleSort(ls):
    n = len(ls)
    for i in range(n):
        for j in range(0, n-i-1):
            if ls[j] > ls[j+1]:
                ls[j], ls[j+1] = ls[j+1], ls[j]
```

SEARCHING ALGORITHM

1. Linear Search Algorithm searches an element in list or other collection structure you are working with, sequentially. For simplicity, we are going to use list for this example.

   Linear search can either start looking from the begin of the list or from the end. When the search key matches an element in the list it stops the code and returns either true, the index of the element or the actual element, depending on the algorithm implementation.

Let's consider the following list of integers for example: [10, 40, 50, 5, 2]
And let's search for the number 40. So the key here is 40.

Step 1, check 40 == 10? If is not true, the code increases an index value and repeat the loop.

Step 2, check 40 == 40 ? If it is true, the code ends and returns True.

Let's now write a code for the linear search algorithm.

```
def linearSearch(key, ls):
    size = len(ls)
    i = 0
    while i < size:
        if ls[i] == key:
            return True
        i += 1
    return False
```

Or even shorter with for-loop.

```
def linearSearch(key, ls):
    for element in ls:
        if element == key:
            return True
    return False
```

Well, the linearSearch method takes the key and the list as arguments. Then it checks if the key matches with one of the elements of the list. Immediately after it finds an element that matches the key, it returns True and ends the code, otherwise, the code returns False. This kind of search will start from the first element in the list.

Imagine we have 100,000,000 elements in the list. How much time it takes for searching a specific element, especially it that element is located towards the end of the list?

If we assume that one loop takes 1 millisecond, and the element we want to look for is the last element. It would take 100,000,000 millisecond!!!! or 1666.67 minutes. That is too long just to find whether a number appears in the list.

The best case would be the element that you looking for is at the first position in the list, the time consumption is O(1). For simplicity, O(1) means it take just 1 round of code execution. But for the worst case would be O(n), where n is the total number of elements in the list. Why?

……

……

……

2. Binary Search Algorithm

Linear search is quite simple, right? Binary search is not that simple, but it is not that complex, although it is not that simple it is more efficient than linear search.

Binary search requires that the list must be sorted, it can either be sorted in ascending or descending order, for this case we are sorting the list in the ascending order.

In a sorted list, it first checks the middle element in the list, if it matches the key the algorithm ends and returns true. If not, it checks if the key is greater than or lesser than the middle element.

If the key is greater than the middle element, then the probability is that the element searched is in the second half of the list, if the key is less than the middle element the probability is that the element searched is in the first half of the list.

Either it is looking in the second half or first half, the algorithm will compare the key against the middle element of that half, and it will repeat

the same process of comparing the key with the middle element until it finds it, or the comparisons are done.

For example, we are searching for number 4 in the list with the following elements: [2,4,1,7,8,15,5]

Step 1, we have to sort the list, and as a result we get

[1,2,4, 5,7,8,15]

Step 2, Find the middle index by adding rightmost index and leftmost index together and calculating a floor division by 2, for example (0+6)//2 = 3

Step 3, Compare the key and the element with a middle index, for example 4 == 5 or 4 > 5 or 4 < 5

Step 4, In this case that 4 is less than 5, so we are going to look for 4 in the first half.

Step 5, Change the rightmost and leftmost indexes to the first and last indexes of the first half: leftmost = 0, rightmost = 2

Step 6, Find the middle position in the first half
(leftmost + rightMost) // 2 → (0 + 2) // 2 = 1

Step 7, Compare the key with the middle position element 4 == 2 or 4 > 2 or 4 < 2

Step 8, In this case, 4 is greater than 2, then let us look for 4 on the right half.

Step 9, Save the rightmost and the leftmost position, in the first half leftMost=2; rightMost=2;

Step 10, Find the middle position (left + right) // 2 → (2 + 2) // 2 = 2

Step 11, Compare the key with the element that index, as a result we get 4 == 4

Step 12, Then the code returns True since we have found the element.

Let's now write a code for the linear search algorithm on the next page.

```
def binarySearch(key, ls):
    ls.sort()
    size = len(ls)
    leftMost = 0
    rightMost = size - 1

    while True:
        middle = round((leftMost + rightMost) / 2)
        if key == ls[middle]:
            return True
        elif key < ls[middle]:
            rightMost = middle - 1
        else:
            leftMost = middle + 1

        if leftMost > rightMost:
            return False
```

First, we sort the list, set leftMost to zero and rightMost to size-1. Then we start a loop that will iterate while the leftMost is less than or equal to rightMost. Within the loop we first get the middle position, then check if the key is equal the element in the middle position, if it is it returns true and ends the code, else it checks if the key is less than the element in the middle position, if it is the rightMost variable is assigned to middle-1 else the leftMost variable is assigned to middle + 1. The loop will run until it matches the key or while the leftMost variable is less than or equal to the rightMost variable.

Observe that we used the round method while calculating the middle position, in the case that leftMost + rightMost is an odd number the middle value will be a real number, so we use round to round it up and make the number an integer.

What is the best case and wort case for binary search time computation? Justify your answer.

……

……

……

EXERCISES

1. Write a recursive function that finds a summation of an element that is divisible by 3 and 7.

2. Write a recursive function that finds the total number of even numbers from the given list.

3. Write a function, namely bidirectionSearch(key, numlist), that prints “It takes {loopcount} loops to find {key}”, otherwise it prints “Sorry a number is not found”.

   Note that this function will look for a key from both directions (leftmost index and rightmost index) of the list.

   With the same numlist, compare bidirectionSearch() with linearSearch()

4. Write a function, namely foursliceSearch(key, numlist), that prints “It takes {loopcount} loops to find {key}”, otherwise it prints “Sorry a number is not found”. Assume that a length of the list must be divisible by 4.

   Note that this function slices a list into four slices and applies a linear search to each slice. Compare bidirectionSearch() and foursliceSearch().

   For example,
   numlist = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
   bidirectionSearch(9, numlist)
   foursliceSearch(9, numlist)

   bidirectionSearch: It takes 8 loops to find 9.
   foursliceSerach: It takes 1 loop to find 9.

ASSIGNMENT:

1. Write a function, namely fourslicebidirectionSearch(), that will slice a list of integers into four slices and applies bidirection search to each slice. Note that a length of list could be any size and must be at least 4.

   Compare linearSearch(), bidirectionSearch(), foursliceSearch() and fourslicebidirectionSearch().