Date July 18, 2021

WEEK 7

Note : Recursion notes (detailed) in Week 5

Recursive Programs

Program : Find 0 in a list using recursion.

```
def check0(L) :
    if (len(L)==0) :        #if the list is empty return False
        return False
    if (L[0]==0): : (#If first element is zero return True
        return True
    else :
        return check0(L[1:len(L)])
                         # it checks the rest of the list excluding first
                         # element
```

DATASCIENCELASTMINUTEFRIEND

SORTING RECURSIVELY

CODE :

```
def mini(L) :
    #finds the minimum element in the list
    mini = L[0]
    for x in L :
        if (x < mini) :
            mini = x
    return mini

def Sort(L) :
    # recursively sorts the list.
    if (L == []) or (len(L) == 1)
        return L
    #if the list is empty there is nothing to sort

    m = mini(L)
    #m now contains the minimum most element in L
    L.remove(m)
    #we remove that element from L
    return [m] + Sort(L)
    # we recursively sort the smaller list

L = [5, 6, 59, 19, 2, 7]
print (Sort(L))
```

OUTPUT: [2, 5, 6, 7, 19, 59]

Binary Search

What is Binary Search ?

→ Binary search is a searching algorithm for finding an element's position in a sorted array.

→ In this approach, the element is always searched in the middle of a portion of an array.

→ Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

BINARY SEARCH WORKING

Binary search algorithm can be implemented in two ways which are :

(1) Iterative Method

(2) Recursive Method

(The recursive method follows the divide & conquer approach

The general steps for both the methods are :

1. The list in which searching is to be performed is —

[3, 4, 5, 6, 7, 8, 9]

Let x=4 be the element to be searched.

2. Set two pointers low and high at the lowest and highest points respectively

```
[ 3 , 4 , 5 , 6 , 7 , 8 , 9 ]
  ↑                       ↑
 low                     high
```

3. Find the middle element <u>mid</u> of the list, i.e., list[(low + high) //2] = 6

```
[3, 4, 5, 6, 7, 8, 9]
          ↑
         mid
```

4. If x == mid, then return mid. Else, compare the elements to be searched with mid.

5. If $\boxed{x > mid}$, compare x with the middle elements of the elements on the right side of <u>mid</u>. This is done by setting <u>low</u> to <u>low = mid + 1</u>.

6. Else, compare <u>x</u> with the middle element of the elements on the left side of <u>mid</u>. This is done by setting <u>high</u> to <u>high = mid - 1</u>.

```
[ 3 , 4 , 5 , [6], 7 , 8 , 9 ]
  ↑         ↑
 low       high
```

7. Repeat steps 3 to 6 until low meets high.

```
[ 3 , 4 , 5 ]
      ↑
     mid
```

8. x=4 is found. [3 , $\boxed{4}$, 5 , 6 , 7 , 8 , 9]
↑
found

ITERATIVE METHOD

CODE:

```
def binary_search (L,k):

    #we want to shrink our list
    # we will do that by using while loop

    begin = 0          #first element in L , L[0]
    end = len(L)-1     # last element in L , L[len(L)-1]

    # using a while loop to look at the list and keep halving it
    while (end-begin >1)
        #we will handle the case when the no. of elements is
        # less than or equal to 1.
        mid = (begin + end)//2

        if (L[mid]==k):        #if mid is indeed k, then we
            return True        # return True & stop the code.

        if (L[mid] > k):       #if mid element is greater than k,
            end = mid-1        #we will check on the left side

        if (L[mid]<k):         # if mid element is less than k,
            begin = mid+1      # we will check on the right side
```

```
# This is outside while loop. If we are here, it means
# that we haven't found the element. Also, if we are
# here, it means that the while condition is violated.
# Which means end-begin is less than or equal to 1.

# if it is equal to 1, then there are exactly 2 elements

if (L[begin]==k) or (L[end]==k):
    return True
else:
    return False
```

RECURSIVE METHOD

```
def rbinary_search (L, k, begin, end):
    '''This will recursively compute binary search'''
    # if begin and end are same, then we need to
    # just check L[begin]

    if (begin == end):
        if (L[begin] == k):
            return True
        else:
            return False

    # if begin and end are consecutive, then check
    # them individually.
    if (end-begin ==1):
        if (L[begin] == k) or (L[end] == k):
            return True
```

```
else :
        return False

if (end-begin > 1):
    # compute the middle element
    mid = (begin - end) //2
    if (L[mid] > k):
        # discard the right & retain the left
        end = mid-1
    if (L[mid] < k):
        # discard the left & retain the right
        begin = mid+1
    if (L[mid] = = k)
        return True

if (end-begin < 0):
    return False
                                                recursive
                                                    step
return rbinary_search (L, k, begin, end)
```