COL100 Assignment 4

2020CS10341

19 January 2021

1 Making a calculator

1.1 Algorithm

The following are the the functions used for the code:

1.1.1 brac

```
3   def brac(s):
4       for i in range(len(s)):
5       #a represents the index value for the rightmost left bracket "(" upto the index 'i' that appears before
        the right bracket ")" whenever a is defined.
6           if s[i] == "(":
7               a = i
8           elif s[i] == ")":
9               return (a,i)
10      #(a,i) represents the indices of the first innermost bracket. Function returns none otherwise (when
        there are no brackets)
```

This function returns the indices of the first innermost bracket that appears in the input string. It stores the index of the the rightmost left bracket `\(" ` and then it stops when it finds a right bracket `\)"` and returns the indices in the form of a tuple `(a,i)` where `a` represents the index of the rightmost left bracket and `i` represents the index of the next right bracket.

1.1.2 oper

This functions returns the index and type of the operator

```
operators = ["+","-","*","/"]
```

present in the string. It does this by checking whether each element of the input string is in the list of operators or not. If it is, then it returns the its index and type in the form of a tuple `(m,i)` where m represents the index of the tuple and `i` represents the index of the tuple in he list `operators` (0 for addition, 1 for subtraction and so on.) The function returns `False` if there is no operator present

1.1.3 eval

This function returns the float point value of an input string containing an operator and two float point numbers. If there is no operator present in the input string (the value of `oper(s)==False`), the function returns the float point value of the input string. Otherwise, it slices the string into two parts- one consisting of the float-point

```
11  def oper(s):
12      #there is no operator symbol for all indices less than m
13      for m in range(len(s)):
14          #checks for the 4 arithmetic operators
15          for i in range(4):
16              if s[m] == operators[i]:
17                  return (m,i)
18      #returns (m,i)-the the index value and the type of the operator
19      return False
20      #returns False if there is no operator
```

```
21  def eval(s):
22      #If there is no operator, return the string as it is after converting it to the appropriate float point
        number
23      if not(oper(s)):
24          return float(s)
25      # y denotes the position (index) of the operator and i denotes the type of the operator
26      (y,i) = oper(s)
27      #a denotes the number present between the operator and the start of the string
28      a = float(s[:y]) if y > 0 else 0
29      #b denotes the number present between the operator and the end of the string
30      b = float(s[y+1:])
31      if i == 0:
32          return a+b
33      elif i == 1:
34          return a-b
35      elif i == 2:
36          return a*b
37      elif i == 3:
38          return a / b
39      # we return the value of the operation represented by the string
```

number before the operator, and the other consisting of the number after it. Then, it performs the appropriate operation on the two numbers and returns the result.

1.1.4 evaluate

This function finds the position of the innermost bracket (if any). Then, it recursively calls itself by substituting the string in the innermost bracket by its calculated value and converting it into a string. This way, it solves the given input by solving one expression in parantheses at a time.

1.2 Proof of Correctness

1.2.1 brac

Pre-Condition: `a represents the index value for the rightmost left bracket '(' upto the index 'i' that appears before the right bracket ')' whenever a is defined.`

```
40 def evaluate(s):
41     if brac(s):
42         (a,b) = brac(s)
43         return evaluate((s[:a])+str(eval(s[(a+1):b])) + s[(b+1):])
44     else:
45         return eval(s)
```

Post-Condition: `(a,i) represents the indices of the first innermost bracket. Function returns none otherwise (when there are no brackets)`

If `a` is defined, it represents the index of the rightmost bracket. If the next letter is neither a `(` or `)` then the value of a remains the same. If the next character is a `(`, then the the value of `a` changes to the `i`, i.e., the current index. So, `a` still represents the index of the rightmost left bracket `(`. If the next character is `)`, the function returns the the tuple `(a,i)` as output.

1.2.2 oper

Pre-Condition: `there is no operator symbol for all indices less than m`
Post-Condition: `returns (m,i)-the the index value and the type of the operator returns False if there is no operator`
Initially, the value of `m` is `0`, so, there is no operator present before it. If the next character of the string is an operator, it returns the value of the operator and its type. It does this by comparing a character with each of the four elements in the list `operators`.
If none of the character in the string matches with the elements in `operators`, the function returns `False`.

1.2.3 eval

We have defined the rest of functions in a way that the function eval takes a list having atmost one operator as an input. If there is no operator, the function returns the input string as a float point number. Otherwise, it takes the two numbers represented by the strings before and after the operator and returns the number we get after evaluation.

1.2.4 evaluate

If `brac(s)` is not `False`, we separate the input string in separate parts, the part before the brackets, the part within and the part after the brackets.
The function calls itself by solving the part within the brackets using `eval` function and then inputs the resulting string (after the solving the innermost bracket) back into itself.
This way, the function solves one bracket at a time and then when there are no brackets left, it simply returns `eval(s)`, i.e. it returns the value of the number represented by the string that contains at most one operator.

2 A sequence of unique sums

2.1 Algorithm

Let `a` represent the required list. Suppose we have i elements at a step. We find out the (i+1)th element of the list.
We set `c` initially as the number after the last element of the list, i.e., `c = a[i-1] + 1`, `k` as `0`,`l` as `i-1` and `e`

```
47  def sumSequence(n):
48      a = [1,2]
49      for i in range(2,n):
50      #the required list has i items
51          c = a[i-1] + 1
52          #all values less than c can not be the i+1 th element of the required list so c is the minimum bound for the
            required answer. We check whether c can be expressed as a sum of the two elements of the list only once.
53          while c < a[i-1]+a[i-2]:
54              k = 0
55              l = i-1
56              e = 0
57              #values of elements with indices less than k and greater than l cannot be involved in the required sum (c)
58              while k<l:
59                  d = a[k]+a[l]
60                  if d == c:
61                      if e == 0:
62                          e = 1
63                          k += 1
64                          l -= 1
65                      else:
66                          c += 1
67                          break
68                  elif d>c:
69                      l -= 1
70                  elif d<c:
71                      k += 1
72              else:
73                  if e==1:
74                      a.append(c)
75                      break
76                  else:
77                      c += 1
78              #while loop above breaks and we append c if it appears only once else c increases by 1.
79          else:
80              a.append(c)
81          #last possible case and we append it (only after checking the other cases)
82      return a
83      #required list has n items
```

as `0`. `e` represents the current number of ways c has been expressed as the sun of two given elements.
We increment `c` by `1` when we confirm that it is not the next element of the list. We do this by taking the sum of the rightmost and leftmost permissible elements, i.e., `d = a[k] + a[l]`. If `d>c`, we decrease `l` by `1`. On the other hand, if `d<c`, we increase `k` by `1`. If `d==c`, we check whether `c` has already been expressed as the sum of two elements by checking the value of `e`. If `e==1`, that means it has already been expressed as a sum of two elements and thus it is not the required number. So, we stop checking for this `c` and increment it by `1`. On the other hand, if `e==0`, then we set `e` as `1` and continue checking for other values by decreasing `l` by `1`. If we reach a stage where $k \geq l$, we stop checking as this way we would be checking for values that we have already checked for.
We increase `c` till we ensure that it is the required element of the list or till it becomes equal to `a[i-1]+a[i-2]`. This is the largest possible sum unique sum of the elements of the list and must be the required element of the list if lower values have been rejected.

2.2 Proof of Correctness

2.2.1 Outermost For Loop

Pre-Condition: `the required list has i items`
For this loop, we assume that for `i`, we have `i` elements in the list already.
Post-Condition: `the required list has i+1 items`
is that when we reach that when we reach `i = n-1`, we have `n-1` elements and we workout the final element of the list and we append it.
To prove that these invariants hold, we must prove the correctness of the inner loops.

2.2.2 Outermost while loop

Pre-Condition: `all values less than c can not be the i+1 th element of the required list so c is the minimum bound for the required answer.`
Post-Condition: `last possible case and we append it (only after checking the other cases)`
If we find a `c` that satisfies the required conditions, we append it to `a`. Else, we increment `c` by `1` till it becomes equal to the last possible sum, i.e., `a[i-1] + a[i-2]`.

2.2.3 Innermost While Loop

Pre-Condition: `values of elements with indices less than k and greater than l cannot be involved in the required sum (c)`
Post-Condition: `while loop above breaks and we append c if it appears only once else c increases by 1.`
Initially, `k=0` and `l = i-1`. So, the base case of our pre-condition is valid.
Assume that the precondition holds. We have `d = a[k]+a[l]` for some values of `k,l` such that $k < l$ holds. Now, we compare `d` with `c`. If `d==c`, we check whether this sum `c` has already been found or not by checking what the value of `e` is. If `e==1`, this means `c` is to be rejected and we increase it by `1` . Else, we set `e=1` and increase and decrease k, l by `1` respectively to get a possible new value of `d`. If `d>c`, we decrease `l` by `1` to decrease the value of `d` since, we can't decrease the value of `k` (as those values can not be used for the required sum based on our assumption). Similarly, we increase `k` by `1` for `d<c`. So, by `Principle of Mathematical Induction`, our hypothesis is true. We, continue checking for c this way till $k \geq l$, because that will be a contradiction of our invariant.

2.3 Time Complexity

Claim: The time complexity (TC) of this Algorithm is $O(nm)$, where n is the input number and m is the value of the last element of the list.

Proof

Based on our assumptions, to calculate the i^{th} element, we already have the previous $i-1$ elements.
Let $T(k)$ be the time taken to calculate the k^{th} element of the list, and $f(n)$ denote the n^{th} element of the list. The values of `c` we check for range from $f(i)+1$ to $f(i+1)$ ($f(i) = a[i-1]$). So, we check a total of $f(i+1)-f(i)$ values.
For checking each value, it takes at most $i-1 = O(i)$ steps as we constantly decrease $l-k$ by at least 1 at each step, and when we are finding the value of $f(i+1)$, `l = i-1` and `k = 0` initially. (Note that each step takes $O(1)$ time.)

So, we get the following equation:

$$T(i+1) = T(i) + O(i)(f(i+1) - f(i))$$

$$T(i+1) = T(i) + O(i)f(i+1) - O(i-1)f(i) - f(i)$$

Now, `i` ranges from `2` to `n-1`. So, we get:

$$T(n) = T(n-1) + O(n-1)f(n) - O(n-2)f(n-1) - f(n-1)$$

$$T(n-1) = T(n-2) + O(n-2)f(n-1) - O(n-3)f(n-2) - f(n-2)$$

$$\vdots$$

$$T(3) = T(2) + O(2)f(3) - O(1)f(2) - f(2)$$

Summing both sides, we get

$$T(n) = T(2) + O(n-1)f(n) - O(1)f(2) - \sum_{k=2}^{n-1} f(k)$$

This simplies to

$$T(n) = O(n-1)f(n)$$

Taking $O(n-1) = O(n)$ and $f(n) = m$, we get

$$T(n) = O(n) \times m$$

Or simply,

$$T(n) = O(nm)$$

Hence proved

3 Shortest sublist with sufficient sum

3.1 Algorithm

We consider an element with an index `i` and add the next elements till our sum becomes greater than the required value. Let's call the sum calculated this way as s_i. We do this separately for the first element s_0, store the value of the minimum length required for the sum to exceed the required value and then we proceed.
If we do not get our sum greater than the required value for all but the last element, we store the value of `ans` accordingly, which may be required later in case the value of `j` does not change.

```
if s <= b:
    ans = -1
else:
    ans = n
```

```python
def minLength(a,b):
    n = len(a)
    s = a[0]
    j = 1
    #First While Loop
    while s <= b and j < n:
    #s is less than required sum 'b' and the index of the element we are adding to s 'j' is still in the list.
        s += a[j]
        j += 1
    if s <= b:
        ans = -1
    else:
        ans = n
    i = 1
    #if s <= b then j must have become equal to n for the loop to break. In this case, we have checked all possible sums involving the first element. If s>b, the value of j <= n is stored and we check whether the other sums can exceed b while requiring less than j elements.

    #Second While Loop
    while i < n:
    #j > 1 represents the minimum number of elements required to exceed the required sum. We initialize 's' to the first element.
        s = a[i]
        if s > b:
            return 1
        for k in range(i+1,min(n,i+j)):
        #s<=b and the number of elements we have checked is less than j while also ensuring that these indices are not greater than n-1.
            s += a[k]
            if s > b:
                j = k-i + 1
                break
        #if we get s>b, then the value of j is updated and the for loop is broken. Otherwise, j remains the same as before.
        i += 1
    #the value of j has changed if it was possible to express the sum using less than j elements, otherwise it remains same. Now, we find this sum for the next element.
    if j==n:
        return ans
    else:
        return j
    #we checked the value of the sum s starting from all elements and return the output accordingly.
```

Where b is the given number our sum s needs to exceed.
After checking for the first element separately, we then calculate s_i for $1 \leq i \leq n-1$ by taking at most elements.
If we find a sum $s_i > b$ for a $j' \leq j$, we set this as the new value of `j` and then proceed.
If we find an element which is greater than b, we return `1` as output. Else, we finish by calculating s_i for all elements.
In the end, we return according to the value of `j`,

```
if j==n:
    return ans
else:
    return j
```

If `j<n` then it is the required answer. Else, we are taking all the elements and the answer should be `ans` (as we considered all elements including the first one).

3.2 Proof of Correctness

3.2.1 First While Loop

Pre-Condition: `s is less than required sum 'b' and the index of the element we are adding to s 'j' is still in the list.`
Post-Condition: `if s <= b then j must have become equal to n for the loop to break. In this case, we have checked all possible sums involving the first element.`

If s>b, the value of j <= n is stored and we check whether the other sums can exceed b while requiring less than j elements.
At each step, we increment s by a[j] and then j by 1 till our while loop breaks. Then we set the value of ans assuming that j==n. If at any later step j becomes less than n, ans is not required. Otherwise, ans is our required answer.

3.2.2 Second While Loop

Pre-Condition: j > 1 represents the minimum number of elements required to exceed the required sum. We initialize 's' to the first element.
Post-Condition: the value of j has changed if it was possible to express the sum using less than j elements, otherwise it remains same. Now, we find this sum for the next element.
We calculate the value of $s_i, 1 \leq i < n$ as we did for the first element $i = 0$. Initially, we set s = a[i]. If it exceeds b, we return 1, else we proceed using a *for loop* When this loop breaks, we increment i by 1.

3.2.3 For Loop

Pre-Condition: s<=b and the number of elements we have checked is less than j while also ensuring that these indices are not greater than n-1.
Post-Condition: if we get s>b, then the value of j is updated and the for loop is broken. Otherwise, j remains the same as before.
Note that the invariants hold for our base case. We add the next element and then check whether the sum has exceeded b. If it has, then we update the value of j and break the for loop. Else, we continue till we reach the last element of the list or the number of elements required for the current sum becomes j (as we are trying to exceed b using less than j elements).

Using these invariants (pre-conditions and post-conditions), we can see that we check less than j elements at each step in order to find a j' such that sum of j' continuous elements exceeds the given sum. Then, we return an output accordingly.

3.3 Time Complexity

Claim: The time complexity (TC) of this Algorithm is $O(n^2)$, where n is the length of the input list.

Proof

3.3.1 First While Loop

For the first while loop, we can see that it takes at most n steps for it to break. So, it's time complexity is simply $O(n)$ as each step takes $O(1)$ time.

To calculate the Time Complexity of the next while loop, we first need to calculate the time complexity of the for loop inside it.

3.3.2 For Loop

Each step inside this loop takes constant amount of time, i.e, $O(1)$ time.
Now, this loop runs for k from i+1 to min(n,i+j). Maximum possible value of j is n. So, maximum value of min(n,i+j) is n.
Thus, this loop takes $O(n)$ steps and since each takes $O(1)$ time, Time Complexity of this loop is $O(n)$.

3.3.3 Second While Loop

Now that we know the time complexity of the inner for loop, we can see that Time Complexity of this While loop should be $n \times O(n) = O(n^2)$, as this loop takes $O(n)$ steps and each step takes $O(n)$ time.

Hence proved

4 Merging a contact list (bonus)

4.1 Algorithm

4.1.1 mergeContacts

```
174 def mergeContacts(A):
175     A = sorter(A)
176     i = 0
177     B = []
178     while i < len(A)-1:
179     #all the emails appearing before the index i have been associated to a name
180         c = []
181         if A[i][0] != A[i+1][0]:
182             c.append(A[i][1])
183             B.append((A[i][0],c))
184             i += 1
185         else:
186             temp = A[i][0]
187             c.append(A[i][1])
188             j = i+1
189             while j < len(A):
190             #all the emails corresponding to the first occurence of the current name 'temp' till the index j in the sorted list A have been
                appended to the list c
191                 if A[j][0]==temp:
192                     c.append(A[j][1])
193                     j+=1
194                 else:
195                     i=j
196                     B.append((temp,c))
197                     break
198             #either we run out of list items (j==len(A)) or we find an item with a different name. In this case, we break the while loop and
                append the tuple (name,c) to the output list B.
199             if j==len(A):
200                 B.append((temp,c))
201                 break
202             #we append the tuple (name,c) to the output list B.
203     #each email has been associated with the appropriate name.
204     return B
```

This is our main function. It takes the help of a helper function `sorter` which sorts the input list. Then this function traverses through the sorted list one by one and checks for duplicate names, and if it finds any, appends them to a list `c` and then associates this list to the corresponding name and then appends a tuple `(A[i][0],c)` to the output list `B`. If there are no duplicates, it just adds one element to the list `c`, appends the tuple `(A[i][0],c)` and continues to check for the other elements.

4.1.2 sorter

This function is similar to the function done in our class.
We take two blocks, merge them using `merger`, and the increase the block size by a factor of `2`. Initially, the size of each block is `1`. We take another list of the same size and items initially set to `0`. We then merge these lists and store them in the other list alternatively. *Merging* means merging two sorted blocks to give a bigger sorted block.

```
152 def sorter(L):
153     block=1
154     drn = 0
155     n = len(L)
156     L2 = [0]*n
157     while block<n:
158     #elements in a block of length 'block' are sorted
159         for i in range(0,n,2*block):
160         #all elements with indices less than i are sorted in blocks of length '2*block'
161             if drn%2 == 0:
162                 L2 = merger(L,L2,i,block)
163             else:
164                 L = merger(L2,L,i,block)
165         #all elements in are sorted in blocks of length '2*block'
166         drn += 1
167         block *= 2
168     #the full list is sorted
169     else:
170         if drn %2 == 0:
171             return L
172         else:
173             return L2
```

4.1.3 merger

We take two blocks to merge, compare their first elements, then take the smaller of the elements out and take the next element of the block from which the previous element was taken. Then we take these two elements and proceed in a similar way.

4.2 Proof of Correctness

4.2.1 mergeContacts

Pre-Condition: `all the emails appearing before the index i have been associated to a name`
Post-Condition: `each email has been associated with the appropriate name`
In the sorted list `A`, if the name `A[i][0]` does not match the next name `A[i+1][0]`, this means that this name occurred only once and we add the `A[i][1]` to an empty list `c` and then append the tuple `(A[i][0],c)` to the output list `B`. Else, we add all emails corresponding to the same name to a list using a while loop, and then append the corresponding (name,email list) tuple to the output list `B`.
For this while loop,
Pre-Condition: `all the emails corresponding to the first occurrence of the current name 'temp' till the index j in the sorted list A have been appended to the list c`
Post-Condition: `either we run out of list items (j==len(A)) or we find an item with a different name. In this case, we break the while loop and append the tuple (name,c) to the output list B.`
If `j==len(A)`,
`we append the tuple (name,c) to the output list B.` We do this by considering the next element in the list `A` and comparing `A[j][0]==temp`, where `temp` is the name of the first duplicate entry (name is duplicated for at least two elements).
If this name is different, we set the `i=j` and append the tuple `(A[i][0],c)` to the output list `B`. If this name is the same, we append the email corresponding to it and increment `j` by `1`.

4.2.2 Sorter

Initially we set the value of `block` to be `1`, as a such a block is sorted by definition. `drn` represents whether the elements after being sorted by a step further are in the first list or the second (`drn` is even for the first case and

```
85  > def minLength(a,b):…
    Run Cell | Run Above | Debug Cell
121 # %% Question 4
122 def merger(L,L2,i,block):
123     n = len(L)
124     j = i
125     k = i + block
126     l = i
127     while k < min(n,i+(2*block)) and j<i+block:
128     #all the elements occuring before k and j in their respective blocks have been added to the second list in a sorted manner.
129         if L[j][0]<L[k][0]:
130             L2[l]=L[j]
131             l += 1
132             j += 1
133         else:
134             L2[l]=L[k]
135             l += 1
136             k += 1
137     #all the elements that have been added to the second list till we run out of elements in one of the blocks have been added in a sorted
        manner and the remaining elements are larger than these elements.
138     else:
139         while j < min(n,i+block):
140         #elements remaining in the first block before j have been added in a sorted manner
141             L2[l]=L[j]
142             l += 1
143             j += 1
144         #elements of the first block have been placed in a sorted manner
145         while k < min(n,i+2*block):
146         #elements remaining in the second block before k have been added in a sorted manner
147             L2[l]=L[k]
148             l += 1
149             k += 1
150         #elements of the second block have been placed in a sorted manner
151     return L2
```

odd for the second). For the outer while loop, we have:
Pre-Condition: `elements in a block of length 'block' are sorted`
Post-Condition: `the full list is sorted`
Assuming that the inner loop functions as desired, the value of `block` is increasing by a factor of `2`, and when it exceeds `n`, our list will be sorted based on our pre-condition.
For, the inner while loop:
Pre-Condition: `all elements with indices less than i are sorted in blocks of length '2*block'`
Post-Condition: `all elements in are sorted in blocks of length '2*block'`
Assuming that the `merger` function merges correctly, we get blocks of twice the length that are merged. Since this for loop runs in steps of `2*block`, we get many blocks of these lengths that are sorted.

4.2.3 Merger

It suffices to prove that this function merges correctly to complete our proof of correctness. `j` denotes the indices of the elements of the first block and `k` denotes the indices of the elements of the second block. For the first while loop,
Pre-Condition: `all the elements occurring before k and j in their respective blocks have been added to the second list in a sorted manner.`
Post-Condition: `all the elements that have been added to the second list till we run out of elements in one of the blocks have been added in a sorted manner and the remaining elements are larger than these elements.`
Since no elements have been added to the second list initially, our base condition holds for the initial values of `j,k`. Then, we compare these elements. The smaller of these elements gets appended to the list in the position

`l`. We increase the index of the smaller element by 1 and l by 1 as well. Since both the blocks are sorted, the next element that appears must be greater than the previous element. Thus, we continue sorting till one of the blocks are exhausted and all their elements have been added to the second list.
When one of the blocks gets exhausted, we add the remaining items to the list using a while loop.
Pre-Condition: `elements remaining in the first block before j have been added in a sorted manner`
Post-Condition: `elements of the first block have been placed in a sorted manner`
So, we check whether any elements remain in any block and add them in a sequential order to the second list.

4.3 Time Complexity

We already know that *Merging and Sorting* in this manner takes $O(nlog(n))$ time. The main function first sorts the input list and then traverses the list items one by one. Traversing and appending in this manner takes constant time for each step, so this whole process takes $O(n)$ time. Hence, our net time complexity is $O(nlog(n))$ time.