Hongsup OH
GTID:903232384

Problem1 (Divide and Conquer)

First step (part a), I make optimal algorithm to find optimal sum and corresponding two indices. Second step (part b-(1)) , I design divide and conquer algorithm, in which given array is separated as two parts; left and right, such as left = A[0:m+1], right = A[m+1:len(A)], where m = len(A)//2, and then find maximum sum of central part of array. Third step (part b-(2)), I build an algorithm named max_part to find maximum sum of subarray. In this function, I find optimal sum of half left part of array and half right part of array by optimal algorithm, and find maximum sum of central part's array by divide_conquer algorithm. Finally, I compare three results such as left, right, central parts to get maximum sum.

a)Optimal algorithm

I will show my algorithm

```
def opt_sum(arr):
    b = 0
    opt = float('-inf')
    while b < len(arr) - 1:
        e = 0
        while e < len(arr):
            if sum(arr[b:e+1]) > opt:
                opt = sum(arr[b:e+1])
                s_ind = b
                e_ind = e
            else:
                opt = opt
            e += 1
        b +=1
    return [s_ind+1, e_ind+1], opt
```

If above function operates, then results become:

```
>>> index, opt = opt_sum(A)
```

Index = [4,7] #python index is [3, 6]

opt = 32

where Index means span of subarray which return to maximum sum.

Now I explain opt_sum code. At beginning start index (b) and end index (e) are zero, and assume initial optimal sum (opt) is minus infinity. And then comparing each subarray's sum with optimal sum. If any subarray's sum is larger than optimal sum, then optimal sum is changed as subarray's sum. And update start index and end index. And this procedure is iterated until checking all subarray's sum. Finally, I add one to final value of start index (s_index) and end index (e_index), since the problem's index is start from 1 and python's index start from 0. As I mentioned, Index = [4, 7], If you want to check the result, you need sum from A array's 3 to 6 (A[3] to A[6] or sum(A[3:6+1])).

Hongsup OH
GTID:903232384

b).Divide and Conquer

(1).Find central maximumsum (maxCrossingSum)

In this part, maxCrossingSum function find maximum sum of central part of array.

```
def maxCrossingSum(arr,l,m,h) :
    # Include elements on left of mid.
    s_l = 0; left_sum = float('-inf')
    for i in range(m,l-1,-1) :
        s_l = s_l + arr[i]
        if (s_l > left_sum):
            left_sum = s_l
            start_ind =i
    # Include elements on right of mid
    s_r = 0; right_sum = float('-inf')
    for i in range(m + 1, h + 1) :
        s_r = s_r + arr[i]
        if (s_r > right_sum) :
            right_sum = s_r
            end_ind = i
    # Return sum of elements on left and right of mid
    return [start_ind+1, end_ind+1], left_sum + right_sum;
```

The result of the code is

```
>>> ind, total = maxCrossingSum(A,0,len(A)//2,len(A)-1)
```

```
ind = [4, 7] #python index is [3, 6]
```

```
total = 32
```

where A is array, 0 is lower bound, len(A)//2 is middle point, and len(A)-1 is upper bound.

This function shows same result with simulation of part a). Then I explain this function. First, I find middle point named m (m = len(A)//2). And separate array to left and right by middle point (m) such as left = A[0:m+1] and right = A[m+1:len(A)]. Next, I find maximum sum of left part of array, which start from m to 0, since I need find maximum sum of central part of array. And then I find maximum sum of right part of array, which start from 0 to len(A).

(2). Find Maximum part (max_part)

```
def max_part(arr):
    m = len(arr)//2
    left = arr[0:m+1]
    right = arr[m+1:len(A)]
    l_ind,sum_l = opt_sum(left)
    r_ind,sum_r = opt_sum(right)
    ind,sum_mid = maxCrossingSum(arr, 0, len(arr)//2, len(arr)-1)
    index = [l_ind,r_ind,ind]
    total = [sum_l,sum_r,sum_mid]
```

```
    ans = max(sum_l,sum_r,sum_mid)
    for i in range(len(total)):
        if ans == total[i]:
            L = i
    return index[L], ans
```

The result is

>>> index, total = max_part(A)

Index = [4, 7] #python index is [3, 6]

Total = 32

In this part, I compared sub array's sum and find maximum one. For example, find optimal sum of left half array, find optimal sum of right half array, and then find sum of central part's array, and named sum_l, sum_r, and sum_mid. Finally, find max(sum_l, sum_r, sum_mid), which is the answer.

Time complexity of total problem is:

$$part\ a): Left\ and\ right\ half\ of\ array: 2T\left(\frac{n}{2}\right)$$

$$\text{part b)} - (1): Central\ part\ of\ array: O(n)$$

$$part\ b) - (2): Combine: O(1)$$

$$\text{T(n)} = 2T\left(\frac{n}{2}\right) + O(n) + O(1) = O(nlog(n))$$

Above time complexity is my answer, but I just want to mention one more thing based on my code. For part c), my code have for loop to find index, thus its exact time complexity is O(n). However, it does not cause change for my final answer (It is still O(n log(n))).

Problem2 (Master Theorem)

a)

a = 49, b = 7, c = 2

$$\text{a} = 49, b^c = 7^2 = 49, thus\ a = b^c$$

$$\text{T(n)} = O(n^c \log(n)) = O(n^2 \log(n))$$

b)

Master Theorem cannot be used in this case, since a is less than 1, such as 1/4

c)

Mater Theorem cannot be used in this case, since f(n) part is not positive

d)

a = 2, b = 4, c = 0.6

Hongsup OH
GTID:903232384

$$a = 2, b^c = 4^{0.6} = 2.297, thus\ a < b^c$$

$$T(n) = O(n^c) = O(n^{0.6})$$

e)

a = 3, b = 2, c = 0

$$a = 3, b^c = 2^0 = 1, thus\ a > b^c$$

$$T(n) = O(n^{log_b a}) = O(n^{log_2 3})$$

Problme3 (Dynamic Programming)

1.

(1).Subproblem.

Assume G is the sequence of George's grades, for example G[1] is his algorithms grade for the first time, and G[n] is the most recent grade. And $S^*[i_m] = \{G[i_1], \dots, G[i_m]\}$ is the longest subsequence of increasing grades ending with grade $G[i_m]$. That is $S^*[i_m]$ is composed of the grades from the set of years $\{i_1, \dots, i_m\}$.

(2).S^*[j] must contain S^*[i] for i < j to show optimal structure

If $i_m = j$, we have S^*[j] for year j, and $l^*[j] = m$ is the length of S^*[j]. Now we know the last grade in S^*[j] is G[j]. If I consider only the first m-1 grades in S^*[j], I have the subsequence of length m-1 ($\{G[i_1], \dots, G[i_{m-1}]\}$). It must be the subsequence of improving grades, because it is the part of S^*[j]. If $i_{m-1} = i$, then I can know $i = i_{m-1} < i_m = j$.

(3).Prove $\{G[i_1], \dots, G[i_{m-1}]\}$ is the longest subsequence of improving grades that ends with $G[i_{m-1}]$.

If $\{G[i_1], \dots, G[i_{m-1}]\}$ is not the longest ending in grade $i = i_{m-1}$, and there is other subsequence of improving grades named S'[i], which also ends in G[i]. However, its length l'[i] > m – 1. If I just insert S'[i] into original solution S^*[j], then I can get a subsequence length l'[j] = l'[i] + 1 > m-1+1 = m. But S^*[j] is the longest subsequence of improving grades for j, thus I have a contradiction. Therefore, it cannot be that the optimal solution S^*[j] without the optimal solution for the subproblem of finding longest subsequence up to and including grade i_{m-1}.

2.

$$l(i) = \begin{cases} 0 & if\ i = 0 \\ max_{k<i, G[k]<G[i]}\{l(k) + 1\} & otherwise \end{cases}$$

First case is the base case. If i = 0, I need looking for $S^*[0]$, which is the longest subsequence of increasing grades ending with G[0]. This is nothing (null). Thus $S^*[0] = null$ (length is zero). The second case is for any other year i (except i = 0). Based on the observation from the optimal substructure, I define l(i) = 1 + the length of the longest increasing subsequence that ends on an earlier year whose grade is less than the current grade. This definition is iteratively make the longest subsequence starting from year i and working back to

the first year.

There will be n subproblems. Since the longest increasing subsequence will be ended at any grades, the answer is not necessarily l(n). However, it is the grade i such that the longest increasing subsequence ending in I is the max among all possible last grades. The longest length is $max_{i=1,\dots,n}$l(i)

3.

I will show the algorithm

```
G = [82,77,65,89,83,68,88,71,91,90] #G array
#The longest Improving grade Subsequence
n = len(G) + 1
l = [0]*(n)
for i in range(1,n):
    m = 0
    for j in range(1,i):
        if G[j-1] < G[i-1] and l[j]>l[m]:
            m = j
        l[i] = l[m] + 1
length = max(l)
print(length)

#Reproduce the sequecne
nex = length
seq = [0]*length
for r in range(len(G),0,-1):
    if l[r] == nex:
        seq[nex-1] = G[r-1]
        nex = nex - 1
    if nex < 0 :
        break
print(seq)
```

Then answer is

length = 4

seq = [65, 68, 71, 82]

Space complexity is O(n), time complexity is $O(n^2)$, which is better than $O(2^n)$

Hongsup OH
GTID:903232384

Problem4 (Dynamic Programming)

Given matrices $A_1, A_2, \dots, A_n$

Where A_i is a $d_{i-1} \times d_i$ matrix

Step1. Develop a recursive solution

Define M(i, j) to be the minimum number of multiplications needed to compute

$$A_i \times A_{i+1} \times \dots \times A_j$$

Goal: Find M(1, n)

Basis: M(i, i) = 0

Recursion: Define M(i, j) recursively

Defining M(i, j) recursively

Consider all possible ways to split A_i through A_j into two pieces

Compare the costs of all these splits:

- Best case cost for computing the product of the two pieces
- Plus the cost of multiplying the two products

Take the best one

$$\mathrm{M(i,j)} = min_k(M(i,k) + M(k+1,j) + d_{i-1}d_k d_j)$$

Matrix multiplication becomes:

$$(A_i \times \dots \times A_k) \times (A_{k+1} \times \dots \times A_j)$$

$$\text{where } P_1 = (A_i \times \dots \times A_k) \; and \; P_2 = (A_{k+1} \times \dots \times A_j)$$

$$\text{minimum cost to compute } P_1 \text{is M(i, k)}$$

$$\text{minimum cost to compute } P_2 \text{is M(k + 1, j)}$$

$$\text{cost to compute } P_1 \times P_2 \; is \; d_{i-1}d_k d_j$$

Step2. Defining the Dependencies

Computeing M(i, j) uses

Everything in same row to the left.

M(i, i), M(i, i+1),…, M(i, j-1)

And everything in same column below:

M(i, j), M(i+1, j),…, M(i, j)

Step3. Identify order for solving subproblems

Hongsup OH
GTID:903232384

Recall the dependencies between subproblems just found

Solve the subproblems this way:

- Go along the diagonal
- Start just above the main diagonal
- End in the upper right corner (Goal)

	1	2	3	4	5
1	0				
2	n/a	0			
3	n/a	n/a	0		
4	n/a	n/a	n/a	0	
5	n/a	n/a	n/a	n/a	0

If M is 5by5, then it looks like above chart. And sequence for solving subproblem is from black arrow, orange arrow, blue arrow, and orange arrow (from long to short arrow).

Pseudocode

```
for i = 1 to n:
    M[i, i] = 0

for d = 1 to n-1: #Diagonal

    for i = 1 to n-d #rows with an entry on d-th diagonal

        j = i + d #column correspond to row I on d-th diagonal

        M[i, j] = infinity

        For k = i to j-1:

            M[i, j] = min(M[i, j], M(i, k) + M(k + 1, j) + d_{i-1} d_k d_j)
```

Time complexity is $O(n^3)$

I show how above code works, when M is 4by 4 case

	1	2	3	4
1	0	1200	700	1400
2	n/a	0	400	650
3	n/a	n/a	0	10,000
4	n/a	n/a	n/a	0

A is 30by1, B is 1by 40, C is 40 by 10, and D is 10 by 25. And 400 and 650 is calculated:

400 is B*C: 1*40*10

620 is (B*C)*D: 400 + 1*10*25