CS 3510 – Assignment 2

Due `Friday, June 03, 2022` at `11:59pm` on Canvas

Instructor: Shahrokh Shahi

- Please type your answers (LaTeX is highly recommended) and upload a single PDF file named `<Your-GT-Account>.pdf`, e.g., GBurdell3.pdf, including all your answers. You can submit multiple times. Canvas keeps track of the submissions and append a version number when you re-submit. We always grade your most recent submissions.
- Please read the policies, and do not forget to acknowledge your collaborators and cite your references.
- If you do not understand a question, please ask on Piazza or come to office hours well ahead of the due date.

Problem 1 — Divide and Conquer, Dynamic Programming (50 pts)

Assume there exists a cryptocurrency called *GTCoin*, where its rise and fall rates are known in advance for the next n days into the future. George Burdell wants to use this invaluable information to buy and sell this cryptocurrency and hopefully earn some extra money. However, there is an important rule for using this information: **"One can only buy and sell GTCoin once in the next n days"**! That means, George can buy GTCoin only once and sell it entirely sometime after that.

For instance, let `rates` $= [5.4, 2.3, -1.4, -0.6, 8.1, -4.2, \ldots]$ and George wants to buy \$100 of GTCoin on day 2, where the rate is $+2.3$. Then, if he sells his GTCoins on day 5, he will receive $\$100 \times (2.3 - 1.4 + 0.6 + 8.1)$. In other words, the total earning rate will be the sum of the rates between the buying and selling dates (inclusive). Therefore, he is trying to calculate the maximum total earning that he can get from investing in this cryptocurrency using the information given by the rates array. Let's help George by designing "efficient" algorithms that can solve this problem, which can be defined as following:

Given an array, `rates`, of size n, including positive and negative numbers representing the changes in the GTCoin prices over the next n days, design algorithms to find the sub-array which has the largest sum, then return this largest sum value.

Example 1:
Input: rates = [-1.2, 5.4, 2.5, -1.4, -0.6, 8.1, -4.0, -1.5, 3.9]
Output: largest rate sum = 14.0
Explanation: The maximum rate sum can be obtained by buying on day 2 and selling on day 6: 5.4 + 2.5 -1.6 -0.6 + 8.1 = 14.0

Example 2:
Input: rates = [-1.7, -4.4, -2.5, 3.9, -2.7, 0.0]
Output: largest rate sum = 3.9
Explanation: The maximum rate sum can be obtained by buying on day 4 and selling at the end of the same day.

Example 3:
Input: rates = [1.8, 2.4, 3.5, 0.0, 2.1]
Output: largest rate sum = 9.8
Explanation: The maximum rate sum can be obtained by buying on day 1 and selling on the last day (Because all rate values are non-negative)

Notes:

- A sub-array is defined as a contiguous subset of the given array
- It is not possible to sell before buying anything! But it is possible to buy and sell on the same day (see Example 2).
- You can assume that the problem always has a solution.
- The numbers data type does not matter. In other words, you can assume the rate values are either integer or float (decimal) numbers.

Deliverables:

1. (25 pts) Design an algorithm using the **divide-and-conquer** approach to return the largest sub-array sum. (Hint: You can start by dividing the rates array into two halves. Then, note the sub-array that gives the largest sum can be either in one of these two halves or can start in the first half and end in the second half.)
    - **(a) (Design and Correctness)** Explain your divide-and-conquer algorithm in words and provide the corresponding pseudocode.
    - **(b) (Analysis)** Discuss the time and space complexity of your algorithm.
2. (25 pts) Design another algorithm using the **dynamic programming** approach to return the largest sub-array sum.
    - **(a) (Design and Correctness)** Explain your dynamic programming algorithm in words and provide the corresponding pseudocode.
    - **(b) (Analysis)** Discuss the time and space complexity of your algorithm.

Solutions

This problem is, in fact, a **Maximum Sum Sub-array** problem. In this problem, we have an array of numbers (integers or floating points) and want to find the maximum sum sub-array among all possible (contiguous) sub-arrays.

The naive implementation is using two nested loops to examine all possible cases which will take $O(n^2)$. In the following, **Divide-and-Conquer** and **Dynamic Programming** approaches are employed to solve this problem.

---

(1) Divide-and-Conquer

(1.a) We can find the maximum subarray sum of a given array by using the following divide and conquer algorithm:

- Divide the input array into two halves
- Calculate the maximum of the following:
  1. Maximum subarray sum of the left half (recursively)
  2. Maximum subarray sum of the right half (recursively)
  3. Maximum sum of the subarrays start in the left half and end in the right half

Therefore, we just need to define a recursive function (let's call it *maxSub*) for case 1 and 2, and also need another function to run a linear search for case 3 (*maxCross*). The pseudo codes of these two functions are presented in the following:

Algorithm 1: `maxSub`

Input: $A = \{a_1, a_2, ..., a_n\}$, low, $high$
Result: $maxSum$, the maximum sum of sub-array

```
1 if (low == high ) then                                   // base case
2 |  return max{0, A[low]}
3 end
4 mid ← (low+high)/2
5 maxSum ← max{maxSub(A, low, mid), maxSub(A, mid + 1, high), maxCross(A, low, mid, high)}
6 return maxSum
```

Algorithm 2: `maxCross`

Input: $A = \{a_1, a_2, ..., a_n\}$, low, mid, $high$
Result: $maxSum$, the maximum sum of sub-array

```
1  sum_left ← −∞
2  sum_right ← −∞

3  sum ← 0
4  for (i = mid : −1 : low) do
5  |  sum ← sum + A[i]
6  |  if sum > sum_left then
7  |  |  sum_left ← sum
8  |  end
9  end

10 sum ← 0
11 for (i = mid + 1 : 1 : high) do
12 |  sum ← sum + A[i]
13 |  if sum > sum_right then
14 |  |  sum_right ← sum
15 |  end
16 end

17 maxSum ← sum_left + sum_right

18 return maxSum
```

(1.b) The time complexity of this algorithm is dominated by the recursive part (divide and conquer); therefore, we can use master theorem to calculate the complexity:

$$T(n) = a \times T(n/b) + \theta(n^d)$$

In our algorithm, we have two subproblems and in each iteration the subproblem size will be decreased by a factor of two. Therefore, $a = 2$ and $b = 2$. Moreover, it is clear $d = 1$ because we only have a linear search for the third case ($maxCross$), thus

$$T(n) = 2 \times T(n/2) + \theta(n)$$

In this case, $a = b^d$, so

$$T(n) = \theta(n^d \log(n)) = \theta(n \log(n))$$

Space complexity: We did not create any additional storage (array), and we only used some single scalar variables. Therefore, we only require $O(1)$ space for one call of the subroutine. However, if we take all the recursive calls into account, then the overall space complexity is $O(\log n)$.

---

(2) Dynamic Programming

(2.a) Here in dynamic programming, we can use both bottom-up and top-down approaches. But, at first, we need to find the recurrence relation. Let $OPT(i)$ be the maximum subarray sum of the array ending at index i (i.e., $[a_1, a_2, \ldots, a_i]$). In this case, the optimum solution for an element at index i will be the maximum of $OPT(i-1) + A[i]$ and $A[i]$. The former is obtained by adding $A[i]$ to the solution found until

index $i-1$, and the latter is actually starting a new subarray from index i.

$$OPT(i) = \max\{OPT(i-1) + A[i], A[i]\}$$

It is also obvious that $OPT(i=0) = 0$. Therefore, our bottom-up algorithm can be written as follows

Algorithm 3: `DP`

Input: $A = \{a_1, a_2, ..., a_n\}$
Result: $maxSum$, the maximum sum of sub-array

```
1 OPT[0] ← 0

2 for (i = 1 : n) do
3 |   OPT[i] ← max{OPT[i − 1] + A[i], A[i]}
4 end

5 maxSum ← max(OPT)

6 return maxSum
```

(2.b) The time complexity of this algorithm is clearly $O(n)$ since we only have a simple for-loop (It is actually two for-loops; the second one is for finding maximum of OPT). The space complexity of this algorithm is also $O(n)$ because we create an array of size $(n+1)$ for an input of size n to store the maximum sums ending at each index.

Note:
In the DP implementation, at each step of iterations , we only used the results from previous step. Therefore, we can further improve the algorithms to use $O(1)$ space. This implementation is known as Kadane's algorithm:

Algorithm 4: `DP (Kadane)`

Input: $A = \{a_1, a_2, ..., a_n\}$
Result: $maxSum$, the maximum sum of sub-array

```
1 currSum ← −∞
2 maxSum ← −∞

3 for (i = 1 : n) do
4 |   currSum ← max{currSum + A[i], A[i]}
5 |   maxSum ← max(maxSum, currSum)
6 end

7 return maxSum
```