Imperial College London – Department of Computing

MSc in Computing Science

580: Algorithms
Tutorial: Dynamic Programming

1. The array $A = [A_1, \ldots, A_N]$ contains N integers.

   (a) A *prefix subarray* of A is any continuous subarray that starts with $A[1]$. Write a $\Theta(N)$-time algorithm to find the greatest sum of any prefix subarray of A.

   (b) The greatest sum of *any* continuous subarray of A can be found as follows. For each position i in the array, find s_i, the maximum sum of any continuous subarray starting at i. The solution is then the maximum of those s_i values. There are N such values, so this method will take $\Theta(N^2)$ time using your answer for (1a). This is referred to as a *naive* solution.

   By considering the structure of the naive solution, design a $\Theta(N)$-time solution to the problem.

Answer:

(a) The algorithm needs to loop through the array maintaining two variables: the sum of all the elements seen so far, and the greatest prefix sum found so far. Each time you see a new element, the new total for all elements gives you a possible value for the max prefix sum.

```
procedure MaxPrefix(A = [A_1, ..., A_N])
    Total = A[1]                          ▷ empty prefix not allowed
    Max = A[1]
    for i = 2 to N do
        Total = Total + A[i]
        if Total > Max then
            Max = Total
        end if
    end for
    return Max
end procedure
```

The values of *Total* for each prefix could be stored in another array of size N, and this could be searched for *Max* when it is full. However, it is only ever the value of the previous total that is needed to calculate the next one, so keeping them all is unecessary. In this situation it is simpler and cleaner to just update a single variable, although it will not affect the time complexity if you do use auxiliary arrays.

(b) The key to the $\Theta(N)$ solution is to define s_i in terms of a subproblem. This works as follows. The first subarray starting at i is just $A[i]$. All others are $A[i]$ and some subarray starting at $i+1$. So, value of s_i is either $A[i]$ or $A[i]+s_{i+1}$. So, by working backwards through the array, each s_i can be calculated from the previous one in constant time, and the following algorithm calculates the maximum sum.

```
procedure MaxSum(A = [A_1, ..., A_N])
    Max = A[N]
    S = A[N]                                  ▷ max sum starting at i
    for i = N − 1 to 1 do
        if S > 0 then
            S = S + A[i]
        else
            S = A[i]
        end if
        if S > Max then
            Max = S
        end if
    end for
    return Max
end procedure
```


As before, there is no need to fill another array with the s_i values, although it will not affect the time complexity if you do.