The Maximum Contiguos Subarray Sum Problem (Kadane's Algorithm)

Definitions

Today we'll be studing a Dynamic Programming problem and applying it to an URI's problem (number 1310 - Profit).

Definition: given an array $A = [a_1, \cdots, a_n]$, a *contiguos subarray of A* is an array $B = [b_1, \cdots, b_m]$ where $b_1 = a_i$ for some i and, for all b_j, if $b_j = a_i$ then $b_{j+1} = a_{i+1}$.

Exemples: let $A = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$.

Exemples of contiguos subarrays of A are:

- $B = [-2, 1, -3]$
- $C = [4, -1, 2, 1, -5]$
- $D = [1]$
- $E = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$

However, these aren't exemples:

- $F = [-2, 1, -3, -5]$
- $G = [14]$
- $H = [-2, 1, -3, 4, -1, 2, 1, -5, 4, 3, 2, 1]$

Can you see why?

Definition: the maximum contiguos subarray sum of an array A is the contiguos subarray of that array which sum is maximum.

Exemple: For the same arrays of last exemple, the sum of each one is:

- sum(B) = -4
- sum(C) = 1
- sum(D) = 1
- sum(E) = 1

But the maximum sum of all possible contiguos subarrays is equal to 6.
That subarray is: $M = [4, -1, 2, 1]$.

Now that we understant the definitions, where's the problem:

The Problem

Problem: given an array A with n integer elements, find the maximum contiguos subarray sum of A.

Naive solution

The solution that comes first to our minds is testing every possibility. By doing that, we'll result in a $O(n^2)$ solution.

The algorithm is:

Keep two pointers in that array: `l` and `r`. Start both at 0 and go through all the array with `r` untin n-1 calculating each sum and keeping track of the maximum one. When `r` achieves n-1, do `l <- l + 1`, `r <- l` and again go through all the array with `r`. It's easy to see that the time complexity is $O(n^2)$. (If you know bubblesort, it's basically the same thing)

```
int const N = 10e5;
int maxCSS(int arr[N]) {
    int l, r;
    int curSum, maxSum;
    maxSum = arr[0];
    for (l = 0; l < n; l++) {
        curSum = 0;
        for (r = l; r < n; r++) {
            curSum += arr[r];
            maxSum = max(curSum, maxSum);
        }
    }
    return maxSum;
}
```

We can, instead of that, use Dynamic Programming to improve the time complexity to $O(n)$ and the time complexity to $O(1)$.

Dynamic Programming solution

The subproblems are: find the max contiguos subarray sum of the array $A_i = [a_1, \cdots, a_i],\ i \leq n$.
The base is when $i = 1$ because $A_1 = [a_1]$ and $maxCSS(A_1) = a_1$.
And we have a recurrence:

$$\forall i > 1\ ,\ maxCSS(A_i) = max\left(maxCSS(A_{i-1}) + a_i,\ a_i\right)$$

That means that in any nonbase case we have two options: continue with the segment we got until the last element or start a wholly new segment starting in a_i.

That leaves us with this implementation:

```
int N = 10e5;

// This is a bottom-up approach
int maxCSS(int arr[N]) {
  // curSeg is value of the maxCSS for the last read value
  // curValue is the value that we're reading in the iteration
  int curSeg, curValue;
  int maxi; //maxi is the maxCSS in every iteration

  // The base case:
  // We just read one value and say that's the maximum
  // That's i = 0 since arrays are 0-indexed
  cin >> curSeg;
  maxi = curSeg;

  // Doing other all iterations
  for (int i = 1; i < n; i++) {
    cin >> curValue; // We read a value
    // Decide if it's better includes the value in the segment
    // or if it's better to start a new segment from that value
    curSeg = max(curSeg + curValue, curValue);

    // And, at every iteration, we keep track of the maximum
    maxi = max(maxi, bestSoFar);
  }

  // In the end of the (n-1)-th iteration, we have the maxCSS
  return maxi;
}
```

Profit Problem in URI

The problem is the following: 1310 - Profit

We only differece for the original problem is the cost for every day. But we can translate the Profit problem into the original one easily by subtracting the cost per day in each element of our original array and creating a new one with the real profit for each day instead of having a revenue array.

Then, if the final maximum profit is negative, it means we should not go at all to the town, printing 0.

The solution is here: Solution

References

I learned about this problem in this video:
Max Contiguous Subarray Sum - Cubic Time To Kadane's Algorithm ("Maximum Subarray" on LeetCode)
made by Back to Back SWE.
All the thanks to the original authors.