Maximum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array `[-2,1,-3,4,-1,2,1,-5,4]`,
the contiguous subarray `[4,-1,2,1]` has the largest sum = `6`.

click to show more practice.

More practice:
If you have figured out the O(*n*) solution, try coding another solution using the divide and conquer approach, which is more subtle.

Solution 1

this problem was discussed by Jon Bentley (Sep. 1984 Vol. 27 No. 9 Communications of the ACM P885)

the paragraph below was copied from his paper (with a little modifications)

algorithm that operates on arrays: it starts at the left end (element A[1]) and scans through to the right end (element A[n]), keeping track of the maximum sum subvector seen so far. The maximum is initially A[0]. Suppose we've solved the problem for A[1 .. i - 1]; how can we extend that to A[1 .. i]? The maximum sum in the first I elements is either the maximum sum in the first i - 1 elements (which we'll call MaxSoFar), or it is that of a subvector that ends in position i (which we'll call MaxEndingHere).

MaxEndingHere is either A[i] plus the previous MaxEndingHere, or just A[i], whichever is larger.

```
public static int maxSubArray(int[] A) {
    int maxSoFar=A[0], maxEndingHere=A[0];
    for (int i=1;i<A.length;++i){
        maxEndingHere= Math.max(maxEndingHere+A[i],A[i]);
        maxSoFar=Math.max(maxSoFar, maxEndingHere);
    }
    return maxSoFar;
}
```

written by cbmbbz original link here

Solution 2

Analysis of this problem: Apparently, this is a optimization problem, which can be usually solved by DP. So when it comes to DP, the first thing for us to figure out is the format of the sub problem(or the state of each sub problem). The format of the sub problem can be helpful when we are trying to come up with the recursive relation.

At first, I think the sub problem should look like: `maxSubArray(int A[], int i, int j)`, which means the maxSubArray for A[i: j]. In this way, our goal is to figure out what `maxSubArray(A, 0, A.length - 1)` is. However, if we define the format of the sub problem in this way, it's hard to find the connection from the sub problem to the original problem(at least for me). In other words, I can't find a way to divided the original problem into the sub problems and use the solutions of the sub problems to somehow create the solution of the original one.

So I change the format of the sub problem into something like: `maxSubArray(int A[], int i)`, which means the maxSubArray for A[0:i] which must has A[i] as the end element. Note that now the sub problem's format is less flexible and less powerful than the previous one because there's a limitation that A[i] should be contained in that sequence and we have to keep track of each solution of the sub problem to update the global optimal value. However, now the connect between the sub problem & the original one becomes clearer:

```
maxSubArray(A, i) = maxSubArray(A, i - 1) > 0 ? maxSubArray(A, i - 1) : 0 + A[i];
```

And here's the code

```
public int maxSubArray(int[] A) {
        int n = A.length;
        int[] dp = new int[n];//dp[i] means the maximum subarray ending with A[i]
;
        dp[0] = A[0];
        int max = dp[0];

        for(int i = 1; i < n; i++){
            dp[i] = A[i] + (dp[i - 1] > 0 ? dp[i - 1] : 0);
            max = Math.max(max, dp[i]);
        }

        return max;
}
```

written by FujiwaranoSai original link here

Solution 3

Idea is very simple. Basically, keep adding each integer to the sequence until the sum drops below 0. If sum is negative, then should reset the sequence.

```
class Solution {
public:
    int maxSubArray(int A[], int n) {
        int ans=A[0],i,j,sum=0;
        for(i=0;i<n;i++){
            sum+=A[i];
            ans=max(sum,ans);
            sum=max(sum,0);
        }
        return ans;
    }
};
```

written by lucastan original link here