Maximum Sum of 3 Non-Overlapping Subarrays

In a given array `nums` of positive integers, find three non-overlapping subarrays with maximum sum.

Each subarray will be of size `k`, and we want to maximize the sum of all `3*k` entries.

Return the result as a list of indices representing the starting position of each interval (0-indexed). If there are multiple answers, return the lexicographically smallest one.

Example:

```
Input: [1,2,1,2,6,7,5,1], 2
Output: [0, 3, 5]
Explanation: Subarrays [1, 2], [2, 6], [7, 5] correspond to the starting indices [0, 3, 5].
We could have also taken [2, 1], but an answer of [1, 3, 5] would be lexicographically larger.
```

Note:

- `nums.length` will be between 1 and 20000.
- `nums[i]` will be between 1 and 65535.
- `k` will be between 1 and floor(nums.length / 3).

Solution 1

The question asks for three non-overlapping intervals with maximum sum of all 3 intervals. If the middle interval is [i, i+k-1], where k <= i <= n-2k, the left interval has to be in subrange [0, i-1], and the right interval is from subrange [i+k, n-1].

So the following solution is based on DP.

```
posLeft[i] is the starting index for the left interval in range [0, i];
posRight[i] is the starting index for the right interval in range [i, n-1];
```

Then we test every possible starting index of middle interval, i.e. k <= i <= n-2k, and we can get the corresponding left and right max sum intervals easily from DP. And the run time is O(n).

Caution. In order to get lexicographical smallest order, when there are two intervals with equal max sum, always select the left most one. So in the code, the if condition is ">= tot" for right interval due to backward searching, and "> tot" for left interval. Thanks to @lee215 for pointing this out!

```
class Solution {
public:
    vector<int> maxSumOfThreeSubarrays(vector<int>& nums, int k) {
        int n = nums.size(), maxsum = 0;
        vector<int> sum = {0}, posLeft(n, 0), posRight(n, n-k), ans(3, 0);
        for (int i:nums) sum.push_back(sum.back()+i);
       // DP for starting index of the left max sum interval
        for (int i = k, tot = sum[k]-sum[0]; i < n; i++) {
            if (sum[i+1]-sum[i+1-k] > tot) {
                posLeft[i] = i+1-k;
                tot = sum[i+1]-sum[i+1-k];
            }
            else
                posLeft[i] = posLeft[i-1];
        }
        // DP for starting index of the right max sum interval
        // caution: the condition is ">= tot" for right interval, and "> tot" for le
ft interval
        for (int i = n-k-1, tot = sum[n]-sum[n-k]; i >= 0; i--) {
            if (sum[i+k]-sum[i] >= tot) {
                posRight[i] = i;
                tot = sum[i+k]-sum[i];
            }
            else
                posRight[i] = posRight[i+1];
        }
        // test all possible middle interval
        for (int i = k; i <= n-2*k; i++) {
            int l = posLeft[i-1], r = posRight[i+k];
            int tot = (sum[i+k]-sum[i]) + (sum[l+k]-sum[l]) + (sum[r+k]-sum[r]);
            if (tot > maxsum) {
                maxsum = tot;
                ans = {l, i, r};
            }
        }
        return ans;
    }
};
```

Java version

```java
class Solution {
    public int[] maxSumOfThreeSubarrays(int[] nums, int k) {
        int n = nums.length, maxsum = 0;
        int[] sum = new int[n+1], posLeft = new int[n], posRight = new int[n], ans =
new int[3];
        for (int i = 0; i < n; i++) sum[i+1] = sum[i]+nums[i];
        // DP for starting index of the left max sum interval
        for (int i = k, tot = sum[k]-sum[0]; i < n; i++) {
            if (sum[i+1]-sum[i+1-k] > tot) {
                posLeft[i] = i+1-k;
                tot = sum[i+1]-sum[i+1-k];
            }
            else
                posLeft[i] = posLeft[i-1];
        }
        // DP for starting index of the right max sum interval
       // caution: the condition is ">= tot" for right interval, and "> tot" for lef
t interval
        posRight[n-k] = n-k;
        for (int i = n-k-1, tot = sum[n]-sum[n-k]; i >= 0; i--) {
            if (sum[i+k]-sum[i] >= tot) {
                posRight[i] = i;
                tot = sum[i+k]-sum[i];
            }
            else
                posRight[i] = posRight[i+1];
        }
        // test all possible middle interval
        for (int i = k; i <= n-2*k; i++) {
            int l = posLeft[i-1], r = posRight[i+k];
            int tot = (sum[i+k]-sum[i]) + (sum[l+k]-sum[l]) + (sum[r+k]-sum[r]);
            if (tot > maxsum) {
                maxsum = tot;
                ans[0] = l; ans[1] = i; ans[2] = r;
            }
        }
        return ans;
    }
}
```

written by zestypanda original link here

Solution 2

This is a more general DP solution, and it is similar to that buy and sell stock problem.

dp[i][j] stands for in i th sum, the max non-overlap sum we can have from 0 to j
id[i][j] stands for in i th sum, the first starting index for that sum.

```
class Solution {
    public int[] maxSumOfThreeSubarrays(int[] nums, int k) {
        int[][] dp = new int[4][nums.length + 1];
        int sum = 0;
        int[] accu = new int[nums.length + 1];
        for(int i = 0; i < nums.length; i++) {
            sum += nums[i];
            accu[i] = sum;
        }
        int[][] id = new int[4][nums.length + 1];
        int max = 0, inId = 0;
        for(int i = 1; i < 4; i++) {
            for(int j = k-1 ; j < nums.length; j++) {
                int tmpmax = j - k < 0 ? accu[j] : accu[j] - accu[j-k] + dp[i-1][j-
k];
                if(j - k >= 0) {
                    dp[i][j] = dp[i][j-1];
                    id[i][j] = id[i][j-1];
                }
                if(j > 0 && tmpmax > dp[i][j-1]) {
                    dp[i][j] = tmpmax;
                    id[i][j] = j-k+1;
                }
            }
        }
        int[] res = new int[3];
        res[2] = id[3][nums.length-1];
        res[1] = id[2][res[2] - 1];
        res[0] = id[1][res[1] - 1];
        return res;
    }
}
```

written by sharonMoMo original link here

Solution 3

To solve this question, I just think about the start index and the number of Non-Overlapping Subarrays.
Such as the first subarray's index is i, and then later two's start index will be i + k.
And we just think about the Maximum Sum of 2 Non-Overlapping Subarrays start from i + k and so on.
One important trick is we at the first time we think about the Maximum Sum of 2 Non-Overlapping Subarrays and Maximum Sum of 1 Non-Overlapping Subarrays their length should be the maximum one so the memo will be useful. And every time later we can get answer from HashMap. So the time complex is o(n)? right?

```
class Solution {
    HashMap<Integer, Integer> map2 = new HashMap();
    HashMap<Integer, int[]> map2Res = new HashMap();
    HashMap<Integer, Integer> map1 = new HashMap();
    HashMap<Integer, Integer> map1Res = new HashMap();
    public int[] maxSumOfThreeSubarrays(int[] nums, int k) {
        countOne(nums, 2 * k, k);
        int max = 0;
        int [] res = new int[3];
        for (int i = 0; i <= nums.length - 3 * k; i++) {
            int countCur = 0;
            for (int j = 0; j < k; j++) {
                countCur += nums[i + j];
            }
            int help = countTwo(nums, i + k, k);
            if (countCur + help > max) {
                max = countCur + help;
                res[0] = i;
                res[1] = map2Res.get(i + k)[0];
                res[2] = map2Res.get(i + k)[1];
            }
            // System.out.println("index" + i +"max" + max);
        }
        return res;
    }
    public int countTwo(int[] nums, int start, int k) {
        if (map2.get(start) != null) {
            return map2.get(start);
        }
        int max = 0;
        int [] res = new int[2];
        for (int i = nums.length - 2 * k; i >= start; i--) {
            int countCur = 0;
            for (int j = 0; j < k; j++) {
                countCur += nums[i + j];
            }
            int help = countOne(nums, i + k, k);
            if (countCur + help >= max) {
                res = new int[2];
                max = countCur + help;
                res[0] = i;
                res[1] = map1Res.get(i + k);
            }
```

```
            map2.put(i, max);
            map2Res.put(i, res);
        }
        return max;

    }
    public int countOne(int[] nums, int start, int k) {
        if (map1.get(start) != null) {
            return map1.get(start);
        }
        int max = 0;
        int res = 0;
        for (int i = nums.length - k; i >= start; i--) {
            int countCur = 0;
            for (int j = 0; j < k; j++) {
                countCur += nums[i + j];
            }
            if (countCur >= max) {
                max = countCur;
                res = i;
            }
            map1.put(i, max);
            map1Res.put(i, res);
        }
        return max;
    }
}
```

Hope it helpful

written by huaiyan original link here