37 和最少为K的最短子数组

更新时间：2019-09-27 09:33:16

> 不安于小成，然后足以成大器；不诱于小利，然后可以立远功。
>
> ——方孝孺

刷题内容

> 难度：**hard**
>
> 题目链接：https://leetcode-cn.com/problems/shortest-subarray-with-sum-at-least-k/

题目描述

返回 A 的最短的非空连续子数组的长度，该子数组的和至少为 K 。
如果没有和至少为 K 的非空子数组，返回 -1 。

示例 1：
输入：A = [1], K = 1
输出：1

示例 2：
输入：A = [1,2], K = 4
输出：-1

示例 3：
输入：A = [2,-1,2], K = 3
输出：3

提示：
1 <= A.length <= 50000
-10 ^5 <= A[i] <= 10 ^5
1 <= K <= 10 ^9

解题方案

思路 1 时间复杂度: O (NlogN) 空间复杂度: O (N)

- 最朴素的想法，可以用左端点和右端点来唯一表示一个连续子数组，只要枚举所有的连续子数组，对于每一个连续子数组，判断改子数组的和是否大于或等于 K，求一个最短满足条件的子数组，这种思路光枚举就要 `o(N^2)` 的复杂度，明显超时
- 只要看到连续子数组，我们自然就会想到前缀和的方法法。令 `sum[i] = A[0]+...+A[i]`，那么每一个连续子数组的和可以表示成 `A[left] + ... + A[right] = sum[right] - sum[left - 1]`，问题就转换成：对于每一个 `sum[i]`，枚举每一个小于 `i` 的下标 `j`，如果 `sum[i] - sum[j] >= K`，那么 `i-j` 就可以拿出来比较
  这么一个思路，是 `o(N^2)` 的时间复杂度，也会超时
- 我们再转换一下关注点，假设固定 `sum[i]`，我们在所有的小于 `i` 的下标 `j` 中，找到那些 `sum[j] <= sum[i] - k`，寻找最大的 `j`，这样 `i-j` 就是最小的。
- 假设两个下标 `m < n` 都小于 `i`，如果 `sum[m] >= sum[n]`，那么显然我们只需要检查 `sum[n]` 就行了，`sum[m]` 就不用检查，因为如果有 `sum[m]<=sum[i]-k`，则 `sum[n]<=sum[m]<=sum[i]-k`，并且 `i-n < i-m`，所以 `m` 在这里没任何意义
- 于是我们可以维护一个递增的栈，把 `sum[m]>=sum[n]` 且 `m<n` 中的 `m` 去掉，每一次遍历 `sum[i]`，就在这个递增的栈中查找最大的 `sum[j] <= sum[i] - k`，这个 j 就是使 `i-j` 最小的值

下面我们看下具体的代码实现：

python python 会超时

```python
class Solution:
    def shortestSubarray(self, A: List[int], K: int) -> int:
        sums = []
        # 由于前缀和需要用到sum[-1]=0，需要特判
        def getSum(idx):
            if idx == -1:
                return 0
            else:
                return sums[idx]

        def bsearch(queue, value):
            l, r = -1, len(queue)
            while r - l > 1:
                mid = (l + r) // 2
                if getSum(queue[mid]) <= value:
                    l = mid
                else:
                    r = mid
            return l

        sums, s = [0] *len(A), 0
        for i in range(len(A)): # 前缀和
            s += A[i]
            sums[i] = s
        res = -1
        queue = [-1] # 单调栈的实现
        for i in range(len(sums)):
            idx = bsearch(queue, sums[i]-K)
            if idx != -1:
                if res == -1 or res > i - queue[idx]:
                    res = i - queue[idx]
            # 维护单调栈的递增性
            while queue and getSum(queue[-1]) > sums[i]:
                queue.pop()
            queue.append(i)
        return res
```

c++ beats 21.92%

```
class Solution {
public:
    vector<int> sum;
    //由于前缀和需要用到sum[-1]=0，需要特判
    int getSum(int index) {
        if (index == -1) {
            return 0;
        } else {
            return sum[index];
        }
    }
    int bsearch(vector<int>& Q, int value) {
        int left = -1, right = (int)Q.size();
        //二分查找
        while (right - left > 1) {
            int mid = (left + right) / 2;
            if (getSum(Q[mid]) <= value) {
                left = mid;
            } else {
                right = mid;
            }
        }
        return left;
    }
    int shortestSubarray(vector<int>& A, int K) {
        sum.clear();
        int s = 0;
        //前缀和
        for (auto a: A) {
            s += a;
            sum.push_back(s);
        }
        int ret = -1;
        vector<int> Q;//单调栈的实现
        Q.push_back(-1);
        for (int i = 0;i < sum.size();i++) {
            int index = bsearch(Q, sum[i] - K);
            if (index != -1) {
                if (ret == -1 || ret > i - Q[index]) {
                    ret = i - Q[index];
                }
            }
            //维护单调栈的递增性
            while (Q.size() > 0 && getSum(Q[(int)Q.size() - 1]) > sum[i]) {
                Q.pop_back();
            }
            Q.push_back(i);
        }
        return ret;
    }
};
```

java beats 8.43%

```
class Solution {
    private int[] sum;
    //由于前缀和需要用到sum[-1]=0，需要特判
    private int getSum(int index) {
        if (index == -1) {
            return 0;
        } else {
            return sum[index];
        }
    }
    private int bsearch(List<Integer> Q, int value) {
        int left = -1, right = Q.size();
        //二分查找
        while (right - left > 1) {
            int mid = (left + right) / 2;
            if (getSum(Q.get(mid)) <= value) {
                left = mid;
            } else {
                right = mid;
            }
        }
        return left;
    }
    public int shortestSubarray(int[] A, int K) {
        sum = new int[A.length];
        int s = 0;
        //前缀和
        for (int i = 0;i < A.length;i++) {
            s += A[i];
            sum[i] = s;
        }
        int ret = -1;
        List<Integer> Q = new ArrayList<Integer>();//单调栈的实现
        Q.add(-1);
        for (int i = 0;i < sum.length;i++) {
            int index = bsearch(Q, sum[i] - K);
            if (index != -1) {
                if (ret == -1 || ret > i - Q.get(index)) {
                    ret = i - Q.get(index);
                }
            }
            //维护单调栈的递增性
            while (Q.size() > 0 && getSum(Q.get(Q.size() - 1)) > sum[i]) {
                Q.remove(Q.size() - 1);
            }
            Q.add(i);
        }
        return ret;
    }
}
```

go beats 64 %

```
//由于前缀和需要用到sum[-1]=0，需要特判
func getSum(sum []int, index int) int {
 if (index == -1) {
  return 0
 } else {
  return sum[index]
 }
}
func bsearch(Q []int, value int, sum []int) int {
 left := -1
 right := len(Q)
 //二分查找
 for right - left > 1 {
  mid := (left + right) / 2
  if getSum(sum, Q[mid]) <= value {
   left = mid;
  } else {
   right = mid;
  }
 }
 return left;
}
func shortestSubarray(A []int, K int) int {
    sum := make([]int, len(A))
 s := 0
 //前缀和
 for i := 0;i < len(A);i++ {
  s += A[i];
  sum[i] = s
 }
 ret := -1
 Q := make([]int, 0)//单调栈的实现
 Q = append(Q, -1)
 for i := 0;i < len(sum);i++ {
  index := bsearch(Q, sum[i] - K, sum)
  if index != -1 {
   if ret == -1 || ret > i - Q[index] {
    ret = i - Q[index]
   }
  }
  //维护单调栈的递增性
  for len(Q) > 0 && getSum(sum, Q[len(Q) - 1]) > sum[i] {
   Q = Q[0:len(Q) - 1]
  }
  Q = append(Q, i)
 }
 return ret;
}
```

思路 2 时间复杂度: O (N) 空间复杂度: O (N)

- 延续思路 1，递增栈中，假设我们当前遍历到 i，找到了使 `sum [j] <= sum [i] - k`， `i-j` 最大的 `j`，下标小于 `j` 的元素已经可以去掉了，因为后续遍历过程中，不会再用到下标小于 `j` 的元素，原因是，不能提供更小的下标差
- 这样的做法，将递增栈改成递增队列，每个元素至多进队列一次，每个元素至多出队列一次，每一轮 `i`，寻找最大 `j` 的过程，找到后也顺便把小于 `j` 的给删掉。因此时间复杂度是 `o (n)`

Python beats 90.53%

```python
class Solution(object):
    def shortestSubarray(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        A = [0] + A
        prefix = [0] * len(A)
        for i in range(1, len(A)):
            prefix[i] = prefix[i-1] + A[i]

        opt = collections.deque() # opt(y) = largest x with prefix[x] <= prefix[y] - K
        res = sys.maxsize

        for x2, px2 in enumerate(prefix):

            #
            while opt and px2 <= prefix[opt[-1]]:
                opt.pop()

            # 新增的规则就在这里
            while opt and px2 - prefix[opt[0]] >= K:
                res = min(res, x2 - opt.popleft())

            opt.append(x2)

        return res if res != sys.maxsize else -1
```

c++ beats 76.06%

```
class Solution {
public:
    vector<int> sum;
    //由于前缀和需要用到sum[-1]=0，需要特判
    int getSum(int index) {
        if (index == -1) {
            return 0;
        } else {
            return sum[index];
        }
    }
    int shortestSubarray(vector<int>& A, int K) {
        sum.clear();
        int s = 0;
        //前缀和
        for (auto a: A) {
            s += a;
            sum.push_back(s);
        }
        int ret = -1;
        deque<int> Q;//改成单调队列，因为需要从两边维护
        Q.push_back(-1);
        for (int i = 0;i < sum.size();i++) {
            int x = -2;
            //改进的部分在这里
            while (Q.size() > 0 && getSum(Q.front()) <= sum[i] - K) {
                x = Q.front();
                Q.pop_front();
            }
            if (x != -2) {
                if (ret == -1 || ret > i - x) {
                    ret = i - x;
                }
            }
            //维护单调队列的递增性
            while (Q.size() > 0 && getSum(Q.back()) > sum[i]) {
                Q.pop_back();
            }
            Q.push_back(i);
        }
        return ret;
    }
};
```

java beats 46.52%

```
class Solution {
    private int[] sum;
    //由于前缀和需要用到sum[-1]=0，需要特判
    private int getSum(int index) {
        if (index == -1) {
            return 0;
        } else {
            return sum[index];
        }
    }
    public int shortestSubarray(int[] A, int K) {
        sum = new int[A.length];
        int s = 0;
        //前缀和
        for (int i = 0;i < A.length;i++) {
            s += A[i];
            sum[i] = s;
        }
        int ret = -1;
        Deque<Integer> Q = new LinkedList<Integer>();//单调队列的实现
        Q.addLast(-1);
        for (int i = 0;i < sum.length;i++) {
            int index = -2;
            //改进的部分在这里
            while (Q.size() > 0 && getSum(Q.getFirst()) <= sum[i] - K) {
                index = Q.pollFirst();
            }
            if (index != -2) {
                if (ret == -1 || ret > i - index) {
                    ret = i - index;
                }
            }
            //维护单调队列的递增性
            while (Q.size() > 0 && getSum(Q.getLast()) > sum[i]) {
                Q.pollLast();
            }
            Q.addLast(i);
        }
        return ret;
    }
}
```

go beats 28 %

```
func getSum(sum []int, index int) int {
 if (index == -1) {
  return 0
 } else {
  return sum[index]
 }
}
func shortestSubarray(A []int, K int) int {
    sum := make([]int, len(A))
 s := 0
 //前缀和
 for i := 0;i < len(A);i++ {
  s += A[i]
  sum[i] = s
 }
 ret := -1
 Q := list.New()//单调队列的实现
 Q.PushBack(-1)
 for i := 0;i < len(sum);i++ {
  index := -2
  //改进的部分在这里
  for Q.Len() > 0 && getSum(sum, Q.Front().Value.(int)) <= sum[i] - K {
   index = Q.Front().Value.(int)
   Q.Remove(Q.Front())
  }
  if index != -2 {
   if ret == -1 || ret > i - index {
    ret = i - index
   }
  }
  //维护单调队列的递增性
  for Q.Len() > 0 && getSum(sum, Q.Back().Value.(int)) > sum[i] {
   Q.Remove(Q.Back())
  }
  Q.PushBack(i)
 }
 return ret;
}
```

小结

关于单调队列和单调栈的题目可以做一做 leetcode 上的 962 题，和这道题的思路很相似，在这里我就不多说了，大家可以自己去尝试下，加油。

}