Heaps_1

@kbbhatt04

@June 28, 2023

Find K Pairs with Smallest Sums

You are given two integer arrays `nums1` and `nums2` sorted in **ascending order** and an integer `k`.

Define a pair `(u, v)` which consists of one element from the first array and one element from the second array.

Return *the* `k` *pairs* `(u 1 , v 1 ), (u 2 , v 2 ), ..., (u k , v k )` *with the smallest sums*.

Example:
Input: nums1 = [1,1,2], nums2 = [1,2,3], k = 2
Output: [[1,1],[1,1]]
Explanation: The first 2 pairs are returned from the sequence: [1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]

Note: Wrong intuition of using two pointers.

Consider this testcase: `nums1 = [1,1,2], nums2 = [1,2,3], k = 10`

In this when you maintain two pointers (i and j), and say you reach `i = 2` and `j = 0`. So you will add the pair `[nums1[i], nums2[j]]` to the answer vector and then increment j.

Now `i = 2` and `j = 1`. But before `[nums1[i], nums2[j]]` can make pair of `[2, 2]`, you also need pairs of `[nums1[0], nums2[1]]`, `[nums1[1], nums2[1]]` i.e. `[1,2]` and `[1,2]`.

By maintaining two pointer such as i and j, you will miss out on these pairs i.e. we will only get (n+m) pairs instead of (n*m) overall pairs. Thus this approach fails.

- Approach
    - Brute-force
        - Maintain a min-heap that stores two indices and sum of numbers at that indices `[{nums1[i] + nums2[j], {i, j}}]`

        - Push all pairs to the min-heap and pop out k pairs
        - Time Complexity: $O((n*m)log(n*m) + klog(n*m))$ where $klog(n*m)$ is while popping k pairs with smallest sums
        - Space Complexity: $O(n*m)$
    - Better
        - Maintain a max-heap that stores two indices and sum of numbers at that indices `[{nums1[i] + nums2[j], {i, j}}]`
        - For each element in nums1, iterate over all elements of nums2 and compare their sum with the top element of heap
        - If their sum is < sum at top, then pop the heap and insert this new pair
        - Else if their sum is > sum at top, then break inner for loop (i.e. skip remaining elements of nums2)
        - Time Complexity: $O((n*m)logk)$
        - Space Complexity: $O(min(k, n*m))$
    - Optimal

```
                (0, 0)
             /          \
      (0, 1)                (1, 0)
     /      \            /       \
(0, 2)    (1, 1)    (2, 0)     (1, 1)
```

        - These are all the pairs of indices of both arrays possible represented in binary tree form
        - We maintain min-heap and visited set to keep track
        - Initially, we add {nums1[0] + nums2[0], {0, 0}} pair to min-heap and {0, 0} pair to visited as we know it is guaranteed to be the smallest

- Run a loop `while k > 0 && !minHeap.empty()` and pop from minHeap and add it to answer list and also add next two index pair (i.e. {i+1, j} and {i, j+1} where i and j are obtained from the top element we just popped from the minHeap) to minHeap and visited set
- Time Complexity: $O(min(klogk, (n*m)log(n*m)))$
- Space Complexity: $O(min(k, n*m))$

```
# Python3
# Brute-force Solution
class Solution:
    def kSmallestPairs(self, nums1, nums2, k):
        pq = []
        for i in nums1:
            for j in nums2:
                heapq.heappush(pq, (i+j, i, j))
        ans = []
        for i in range(min(k, len(nums1) * len(nums2))):
            top = heapq.heappop(pq)
            ans += [top[1], top[2]],
        return ans
```

```
# Python3
# Better Solution
class Solution:
    def kSmallestPairs(self, nums1, nums2, k):
        pq = [] # max-heap
        for i in nums1:
            for j in nums2:
                if len(pq) < k:
                    # pushing negative sum as python inbuilt hea
                    heapq.heappush(pq, (-i-j, i, j))
                elif -i-j > pq[0][0]:
                    heapq.heappop(pq)
                    heapq.heappush(pq, (-i-j, i, j))
```

```
                else:
                    break
        ans = []
        for summ, i, j in pq:
            ans += [i, j],
        return ans
```

```
// C++
// Better Solution
#include <bits/stdc++.h>
class Solution {
public:
    vector<vector<int>> kSmallestPairs(vector<int>& nums1, vecto
        int n = nums1.size();
        int m = nums2.size();
        priority_queue<pair<int, pair<int, int>>> pq; // max-hea

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (pq.size() < k) {
                    pq.push({nums1[i]+nums2[j], {nums1[i], nums2
                }
                else if (nums1[i] + nums2[j] < pq.top().first) {
                    pq.pop();
                    pq.push({nums1[i]+nums2[j], {nums1[i], nums2
                }
                else {
                    break;
                }
            }
        }

        vector<vector<int>> ans;
        while (!pq.empty()) {
            ans.push_back({pq.top().second.first, pq.top().secon
```

```
            pq.pop();
        }
        return ans;
    }
};
```

```
# Python3
# Optimal Solution
class Solution:
    def kSmallestPairs(self, nums1, nums2, k):
        queue = [] # min-heap
        visited = set()
        def push(i, j):
            if i < len(nums1) and j < len(nums2) and (i, j) not
                heapq.heappush(queue, [nums1[i] + nums2[j], i, j
                visited.add((i, j))
        push(0, 0)
        pairs = []
        while queue and len(pairs) < k:
            _, i, j = heapq.heappop(queue)
            pairs += [nums1[i], nums2[j]],
            push(i, j + 1)
            push(i + 1, j)
        return pairs
```

```
// C++
// Optimal Solution
#include <bits/stdc++.h>
class Solution {
public:
    vector<vector<int>> kSmallestPairs(vector<int>& nums1, vecto
        int n = nums1.size();
        int m = nums2.size();
```

```
        // [[nums1[i], nums2[j]]]
        vector<vector<int>> ans;
        // ({i, j})
        set<pair<int, int>> visited;
        // [{nums1[i] + nums2[j], {i, j}}]
        priority_queue<pair<int, pair<int, int>>, vector<pair<in

        minHeap.push({nums1[0] + nums2[0], {0, 0}});
        visited.insert({0, 0});

        while (k-- && !minHeap.empty()) {
            auto top = minHeap.top();
            minHeap.pop();
            int i = top.second.first, j = top.second.second;
            ans.push_back({nums1[i], nums2[j]});

            if (i + 1 < n && visited.find({i+1, j}) == visited.e
                minHeap.push({nums1[i+1] + nums2[j], {i+1, j}});
                visited.insert({i+1, j});
            }
            if (j + 1 < m && visited.find({i, j+1}) == visited.e
                minHeap.push({nums1[i] + nums2[j+1], {i, j+1}});
                visited.insert({i, j+1});
            }
        }

        return ans;
    }
};
```

Minimum Operations to Form Subsequence With Target Sum

You are given a **0-indexed** array `nums` consisting of **non-negative** powers of `2`, and an integer `target`.

In one operation, you must apply the following changes to the array:

- Choose any element of the array `nums[i]` such that `nums[i] > 1`.
- Remove `nums[i]` from the array.
- Add **two** occurrences of `nums[i] / 2` to the **end** of `nums`.

Return the ***minimum number of operations*** *you need to perform so that* `nums` *contains a* ***subsequence*** *whose elements sum to* `target`. If it is impossible to obtain such a subsequence, return `-1`.

- Approach
  - Optimal
    - Push elements into the max heap
    - We keep track of the sum to decide whether to break the element in consideration into two halves or not
    - While loop till the target is not achieved
      - Case 1: If top element is less than target, then just subtract the element. No need to divide it at this point
      - Case 2: If top element is greater than target but sum upto top element is less than target. In this case, we should split the top element to two halves
      - Case 3: If our priority queue is empty and we have yet not reached our target, then just return -1
    - Finally we return our ans which we increment at every divide
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$

```
# Python3
# Optimal Solution
import heapq as hp
class Solution:
    def minOperations(self, nums: List[int], target: int) -> int
        tsum = 0
        heap = []
```

```
        for i in nums:
            hp.heappush(heap, -i)
            tsum += i

        ans = 0
        while target:
            front = -heap[0]
            tsum -= front
            hp.heappop(heap)

            if front <= target:
                target -= front
            elif front > target and tsum < target:
                ans += 1
                tsum += front
                hp.heappush(heap, -(front//2))
                hp.heappush(heap, -(front//2))

            if (not heap) and (target != 0):
                return -1
        return ans
```

```
// C++
// Optimal Solution
class Solution {
public:
    int minOperations(vector<int>& nums, int target) {
        priority_queue<int> pq;
        /*We keep a track of sum to decide whether to break the
        element in consideration into two halves or not.*/
        long long sum=0;

        //Push elements into the max heap
        for(int i=nums.size()-1;i>=0;i--){
```

```
            pq.push(nums[i]);
            sum+=nums[i];
        }

        int ans=0;

        //While loop till the target is not achieved
        while(target>0){
            int front = pq.top();
            sum-=front;
            pq.pop();

            /*Case 1: If top element is less than target, then j
            subtract the element. No need to divide it at this
            point.*/

            if(front<=target){
                target-=front;
            }

            /*Case 2: If top element is greater than target but
             upto top element is less than target. In this case,
            should split the top element to two halves.*/

            else if(front>target && sum<target){
                ans++;
                sum+=front;
                pq.push(front/2);
                pq.push(front/2);
            }

            /*Case 3: If our priority queue is empty and we have
            yet not reached our target, then just return -1;*/

            if(pq.empty() && target!=0){
                return -1;
```

```
            }

        }
        //Finally we return our ans which we increment at every
        return ans;
    }
};
```

Template

- Approach
    - Brute-force
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$
    - Better
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$
    - Optimal
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
# Python3
# Better Solution
```

```
# Python3
# Optimal Solution
```

```
// C++
// Optimal Solution
```