# Heap Sort

When we discuss sort problem, no need to say the heap sort is the top one pattern. We put all the related data in heap and poll them one by one. We can use either TreeMap or priority queue, just watch that the TreeMap (map or set) is single key based, if you have multiple key with same value consider either priority queue or multi\_map (multi\_set). By default the priority queue is from high to low, if you want to do it in reverse, make the negative.

## 373. Find K Pairs with Smallest Sums

Medium

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.

Find the k pairs **(u1,v1),(u2,v2) ...(uk,vk)** with the smallest sums.

**Example 1:**

**Input:** nums1 = [1,7,11], nums2 = [2,4,6], k = 3

**Output:** [[1,2],[1,4],[1,6]]

**Explanation:** The first 3 pairs are returned from the sequence:

  [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]

**Example 2:**

**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]

**Example 3:**

**Input:** nums1 = [1,2], nums2 = [3], k = 3

**Output:** [1,3],[2,3]

**Explanation:** All possible pairs are returned from the sequence: [1,3],[2,3]

### Analysis:

First, we point every item in num1 to the first item in num2, sum them up and put in the heap. When we poll one sum from the heap, we should increase the index in num2 until all the element in num2 are used.

/// <summary>

/// Leet code #373. 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.

/// Find the k pairs (u1,v1),(u2,v2) ...(uk,vk) with the smallest sums.

///

/// Example 1:

/// Given nums1 = [1,7,11], nums2 = [2,4,6], k = 3

/// Return: [1,2],[1,4],[1,6]

/// The first 3 pairs are returned from the sequence:

/// [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]

///

/// Example 2:

/// Given nums1 = [1,1,2], nums2 = [1,2,3], k = 2

/// Return: [1,1],[1,1]

/// 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]

///

/// Example 3:

/// Given nums1 = [1,2], nums2 = [3], k = 3

/// Return: [1,3],[2,3]

/// All possible pairs are returned from the sequence:

/// [1,3],[2,3]

/// </summary>

vector<vector<int>> LeetCodeSort::kSmallestPairs(vector<int>& nums1, vector<int>& nums2, int k)

{

vector<vector<int>> result;

if (nums1.empty() || nums2.empty()) return result;

priority\_queue<vector<int>> priority\_queue;

for (int i = 0; i < (int)nums1.size(); i++)

{

priority\_queue.push({ -(nums1[i] + nums2[0]), i, 0 });

}

for (int i = 0; i < k; i++)

{

if (priority\_queue.empty()) break;

vector<int> sum = priority\_queue.top();

priority\_queue.pop();

result.push\_back({ nums1[sum[1]], nums2[sum[2]] });

sum[2]++;

if (sum[2] < (int)nums2.size())

{

priority\_queue.push({ -(nums1[sum[1]] + nums2[sum[2]]), sum[1], sum[2] });

}

}

return result;

}

## 683. K Empty Slots

Hard

You have N bulbs in a row numbered from 1 to N. Initially, all the bulbs are turned off. We turn on exactly one bulb everyday until all bulbs are on after N days.

You are given an array bulbs of length N where bulbs[i] = x means that on the (i+1)th day, we will turn on the bulb at position x where i is 0-indexed and x is 1-indexed.

Given an integer K, find out the **minimum day number** such that there exists two **turned on** bulbs that have **exactly** K bulbs between them that are **all turned off**.

If there isn't such day, return -1.

**Example 1:**

**Input:**

bulbs: [1,3,2]

K: 1

**Output:** 2

**Explanation:**

On the first day: bulbs[0] = 1, first bulb is turned on: [1,0,0]

On the second day: bulbs[1] = 3, third bulb is turned on: [1,0,1]

On the third day: bulbs[2] = 2, second bulb is turned on: [1,1,1]

We return 2 because on the second day, there were two on bulbs with one off bulb between them.

**Example 2:**

**Input:**

bulbs: [1,2,3]

K: 1

**Output:** -1

**Note:**

1. 1 <= N <= 20000
2. 1 <= bulbs[i] <= N
3. bulbs is a permutation of numbers from 1 to N.
4. 0 <= K <= 20000

### Analysis:

You simulate the bulb which opens every day by putting the open flower index in a heap and look back and forward until you see exactly K empty slot.

/// <summary>

/// Leet code #683. K Empty Slots

///

/// There is a garden with N slots. In each slot, there is a flower. The

/// N flowers will bloom one by one in N days. In each day, there will be

/// exactly one flower blooming and it will be in the status of blooming

/// since then.

///

/// Given an array flowers consists of number from 1 to N. Each number in

/// the array represents the place where the flower will open in that day.

///

/// For example, flowers[i] = x means that the unique flower that blooms

/// at day i will be at position x, where i and x will be in the range

/// from 1 to N.

///

/// Also given an integer k, you need to output in which day there exists

/// two flowers in the status of blooming, and also the number of flowers

/// between them is k and these flowers are not blooming.

///

/// If there isn't such day, output -1.

///

/// Example 1:

/// Input:

/// flowers: [1,3,2]

/// k: 1

/// Output: 2

/// Explanation: In the second day, the first and the third flower have

/// become blooming.

///

/// Example 2:

/// Input:

/// flowers: [1,2,3]

/// k: 1

/// Output: -1

///

/// Note:

/// The given array will be in the range [1, 20000].

/// </summary>

int LeetCode::kEmptySlots(vector<int>& flowers, int k)

{

int result = -1;

set<int> sorted\_flowers;

for (size\_t i = 0; i < flowers.size(); i++)

{

// insert the flower first then search for position

sorted\_flowers.insert(flowers[i]);

set<int>::iterator itr = sorted\_flowers.find(flowers[i]);

if (itr != sorted\_flowers.begin())

{

set<int>::iterator prev = itr;

prev--;

if ((\*itr - \*prev) == k + 1)

{

result = i + 1;

break;

}

}

set<int>::iterator next = itr;

next++;

if (next != sorted\_flowers.end())

{

if ((\*next - \*itr) == k + 1)

{

result = i + 1;

break;

}

}

}

return result;

}

## 846. Hand of Straights

Medium

Alice has a hand of cards, given as an array of integers.

Now she wants to rearrange the cards into groups so that each group is size W, and consists of W consecutive cards.

Return true if and only if she can.

**Example 1:**

**Input:** hand = [1,2,3,6,2,3,4,7,8], W = 3

**Output:** true

**Explanation:** Alice's hand can be rearranged as [1,2,3],[2,3,4],[6,7,8].

**Example 2:**

**Input:** hand = [1,2,3,4,5], W = 4

**Output:** false

**Explanation:** Alice's hand can't be rearranged into groups of 4.

**Note:**

1. 1 <= hand.length <= 10000
2. 0 <= hand[i] <= 10^9
3. 1 <= W <= hand.length

### Analysis:

Sort the hand in a priority queue or TreeMap, and collect from the lowest number with repeated with window size of W.

/// <summary>

/// Leet code #846. Hand of Straights

///

/// Alice has a hand of cards, given as an array of integers.

///

/// Now she wants to rearrange the cards into groups so that each group

/// is size W, and consists of W consecutive cards.

///

/// Return true if and only if she can.

///

/// Example 1:

/// Input: hand = [1,2,3,6,2,3,4,7,8], W = 3

/// Output: true

/// Explanation: Alice's hand can be rearranged as [1,2,3],[2,3,4],[6,7,8].

///

/// Example 2:

/// Input: hand = [1,2,3,4,5], W = 4

/// Output: false

/// Explanation: Alice's hand can't be rearranged into groups of 4.

///

/// Note:

/// 1. 1 <= hand.length <= 10000

/// 2. 0 <= hand[i] <= 10^9

/// 3. 1 <= W <= hand.length

/// </summary>

bool LeetCodeSort::isNStraightHand(vector<int>& hand, int W)

{

map<int, int> hand\_map;

for (size\_t i = 0; i < hand.size(); i++)

{

hand\_map[hand[i]]++;

}

while (!hand\_map.empty())

{

int start = hand\_map.begin()->first;

int count = hand\_map.begin()->second;

for (int i = 0; i < W; i++)

{

hand\_map[start + i] -= count;

if (hand\_map[start + i] == 0)

{

hand\_map.erase(start + i);

}

else if (hand\_map[start + i] < 0)

{

return false;

}

}

}

return true;

}

## 774. Minimize Max Distance to Gas Station

Hard

On a horizontal number line, we have gas stations at positions stations[0], stations[1], ..., stations[N-1], where N = stations.length.

Now, we add K more gas stations so that **D**, the maximum distance between adjacent gas stations, is minimized.

Return the smallest possible value of **D**.

**Example:**

**Input:** stations = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], K = 9

**Output:** 0.500000

**Note:**

1. stations.length will be an integer in range [10, 2000].
2. stations[i] will be an integer in range [0, 10^8].
3. K will be an integer in range [1, 10^6].
4. Answers within 10^-6 of the true value will be accepted as correct.

### Analysis:

Of course, you can also do binary search to push down the maximum distance to see if you can accommodate K extra stations. But let’s do it in another way, first you calculate the proportion of the distance between every two stations against the total distance and see how many extra station you will get between these two stations, and get the floor integer. And you may end up with the remaining stations, and you can sort the above new distance in heap and distribute the stations to the longest distance.

/// <summary>

/// Leetcode #774. Minimize Max Distance to Gas Station

///

/// On a horizontal number line, we have gas stations at positions

/// stations[0], stations[1], ..., stations[N-1], where

/// N = stations.length.

///

/// Now, we add K more gas stations so that D, the maximum distance

/// between adjacent gas stations, is minimized.

///

/// Return the smallest possible value of D.

/// Example:

/// Input: stations = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], K = 9

/// Output: 0.500000

/// Note:

///

/// 1.stations.length will be an integer in range [10, 2000].

/// 2. stations[i] will be an integer in range [0, 10^8].

/// 3. K will be an integer in range [1, 10^6].

/// 4. Answers within 10^-6 of the true value will be accepted as correct.

/// </summary>

double LeetCodeSort::minmaxGasDist(vector<int>& stations, int K)

{

if (stations.empty()) return (double)0.0;

double total\_distance = (double)stations.back() - (double)stations[0];

priority\_queue<pair<double, int>> station\_plans;

int remaining\_stations = K;

for (size\_t i = 1; i < stations.size(); i++)

{

int extra\_stations = (int)(((double)stations[i] - (double)stations[i - 1]) \* K / total\_distance);

double planned\_distance = ((double)stations[i] - (double)stations[i - 1]) / ((double)extra\_stations + 1);

station\_plans.push(make\_pair(planned\_distance, (extra\_stations + 1)));

remaining\_stations -= extra\_stations;

}

while (remaining\_stations != 0)

{

pair<double, int> plan = station\_plans.top();

station\_plans.pop();

plan.first = plan.first \* plan.second / ((long long)plan.second + 1);

plan.second++;

station\_plans.push(plan);

remaining\_stations--;

}

double result = station\_plans.top().first;

return result;

}

## 857. Minimum Cost to Hire K Workers

Hard

There are N workers.  The i-th worker has a quality[i] and a minimum wage expectation wage[i].

Now we want to hire exactly K workers to form a *paid group*.  When hiring a group of K workers, we must pay them according to the following rules:

1. Every worker in the paid group should be paid in the ratio of their quality compared to other workers in the paid group.
2. Every worker in the paid group must be paid at least their minimum wage expectation.

Return the least amount of money needed to form a paid group satisfying the above conditions.

**Example 1:**

**Input:** quality = [10,20,5], wage = [70,50,30], K = 2

**Output:** 105.00000

**Explanation**: We pay 70 to 0-th worker and 35 to 2-th worker.

**Example 2:**

**Input:** quality = [3,1,10,10,1], wage = [4,8,2,2,7], K = 3

**Output:** 30.66667

**Explanation**: We pay 4 to 0-th worker, 13.33333 to 2-th and 3-th workers seperately.

**Note:**

1. 1 <= K <= N <= 10000, where N = quality.length = wage.length
2. 1 <= quality[i] <= 10000
3. 1 <= wage[i] <= 10000
4. Answers within 10^-5 of the correct answer will be considered correct.

### Analysis:

We sort by ratio and select the first K workers, and use the highest ratio to adjust the remaining wage, all those wages should be increased because they have lower ratio. But this is not necessarily the best answer. In the next step you pick the workers with higher ratio, who may have lower salary, and replace the selected one who has the highest quality, (i.e. highest wage) and adjust the wage for the selected workers, to do so you do not need to remember wage for everyone, just remember the total quality the selected workers generated. Keep on doing so and record the minimum total wage, you will finally get the answer.

/// <summary>

/// Leet code #857. Minimum Cost to Hire K Workers

///

/// There are N workers. The i-th worker has a quality[i] and a minimum

/// wage expectation wage[i].

///

/// Now we want to hire exactly K workers to form a paid group. When hiring

/// a group of K workers, we must pay them according to the following rules:

///

/// Every worker in the paid group should be paid in the ratio of their

/// quality compared to other workers in the paid group.

/// Every worker in the paid group must be paid at least their minimum wage

/// expectation.

/// Return the least amount of money needed to form a paid group satisfying

/// the above conditions.

///

///

/// Example 1:

/// Input: quality = [10,20,5], wage = [70,50,30], K = 2

/// Output: 105.00000

/// Explanation: We pay 70 to 0-th worker and 35 to 2-th worker.

///

/// Example 2:

/// Input: quality = [3,1,10,10,1], wage = [4,8,2,2,7], K = 3

/// Output: 30.66667

/// Explanation: We pay 4 to 0-th worker, 13.33333 to 2-th and 3-th workers

/// seperately.

///

///

/// Note:

///

/// 1. 1 <= K <= N <= 10000, where N = quality.length = wage.length

/// 2. 1 <= quality[i] <= 10000

/// 3. 1 <= wage[i] <= 10000

/// 4. Answers within 10^-5 of the correct answer will be considered correct.

/// </summary>

double LeetCodeSort::mincostToHireWorkers(vector<int>& quality, vector<int>& wage, int K)

{

vector<pair<double, int>> ratio\_list;

priority\_queue<int> quality\_heap;

for (size\_t i = 0; i < quality.size(); i++)

{

ratio\_list.push\_back(make\_pair((double)wage[i] /

(double)quality[i], quality[i]));

}

sort(ratio\_list.begin(), ratio\_list.end());

unsigned long long sum\_quality = 0;

// if we select K worker, the highest ratio worker is paid as minimum wage,

// for any lower wage, they are upgrade with highest ratio

for (size\_t i = 0; i < (size\_t)K; i++)

{

quality\_heap.push(ratio\_list[i].second);

sum\_quality += ratio\_list[i].second;

}

double ratio = ratio\_list[K - 1].first;

double result = ratio \* sum\_quality;

// we keep on raise the ratio, but because we may have the lower quality

// and wage, we may end up with lower minimum total wage

for (size\_t i = K; i < quality.size(); i++)

{

sum\_quality -= quality\_heap.top();

quality\_heap.pop();

quality\_heap.push(ratio\_list[i].second);

sum\_quality += ratio\_list[i].second;

ratio = ratio\_list[i].first;

result = min(result, ratio \* sum\_quality);

}

return result;

}

## 632. Smallest Range Covering Elements from K Lists

Hard

You have k lists of sorted integers in ascending order. Find the **smallest** range that includes at least one number from each of the k lists.

We define the range [a,b] is smaller than range [c,d] if b-a < d-c or a < c if b-a == d-c.

**Example 1:**

**Input:** [[4,10,15,24,26], [0,9,12,20], [5,18,22,30]]

**Output:** [20,24]

**Explanation:**

List 1: [4, 10, 15, 24,26], 24 is in range [20,24].

List 2: [0, 9, 12, 20], 20 is in range [20,24].

List 3: [5, 18, 22, 30], 22 is in range [20,24].

**Note:**

1. The given list may contain duplicates, so ascending order means >= here.
2. 1 <= k <= 3500
3. -105 <= value of elements <= 105.

### Analysis:

We pick the smallest number in each array, and put in in a priority queue, and keep on pop on the range head, and move to next number until we exhaust the array.

/// <summary>

/// Leet code #632. Smallest Range

///

/// You have k lists of sorted integers in ascending order. Find the

/// smallest range that includes at least one number from each of the

/// k lists.

/// We define the range [a,b] is smaller than range [c,d] if b-a <

/// d-c or a < c if b-a == d-c.

/// Example 1:

/// Input:[[4,10,15,24,26], [0,9,12,20], [5,18,22,30]]

/// Output: [20,24]

/// Explanation:

/// List 1: [4, 10, 15, 24,26], 24 is in range [20,24].

/// List 2: [0, 9, 12, 20], 20 is in range [20,24].

/// List 3: [5, 18, 22, 30], 22 is in range [20,24].

///

/// Note:

/// The given list may contain duplicates, so ascending order means

/// >= here.

/// 1 <= k <= 3500

/// -10^5 <= value of elements <= 10^5.

/// For Java users, please note that the input type has been changed to

/// List<List<Integer>>. And after you reset the code template, you'll see

/// this point.

/// </summary>

vector<int> LeetCodeSort::smallestRange(vector<vector<int>>& nums)

{

vector<int> range = { 0, INT\_MAX };

map<pair<int, int>, int> sort\_map;

for (size\_t i = 0; i < nums.size(); i++)

{

sort\_map[make\_pair(nums[i][0], i)] = 0;

}

while (true)

{

pair<int, int> start = sort\_map.begin()->first;

pair<int, int> end = sort\_map.rbegin()->first;

if (end.first - start.first < range[1] - range[0])

{

range[0] = start.first;

range[1] = end.first;

}

int index = sort\_map.begin()->second + 1;

// exhaust one list, game over

if (index == nums[start.second].size()) break;

// delete the head and move the position in that list to next

sort\_map.erase(start);

sort\_map[make\_pair(nums[start.second][index], start.second)] = index;

}

return range;

}

## 295. Find Median from Data Stream

Hard

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

For example,

[2,3,4], the median is 3

[2,3], the median is (2 + 3) / 2 = 2.5

Design a data structure that supports the following two operations:

* void addNum(int num) - Add a integer number from the data stream to the data structure.
* double findMedian() - Return the median of all elements so far.

**Example:**

addNum(1)

addNum(2)

findMedian() -> 1.5

addNum(3)

findMedian() -> 2

**Follow up:**

1. If all integer numbers from the stream are between 0 and 100, how would you optimize it?
2. If 99% of all integer numbers from the stream are between 0 and 100, how would you optimize it?

### Analysis:

This is a famous problem, to find the median you need to keep two heap, one for the smaller half, another is for the larger half, normally you keep these two heap size balanced, and when you need a median, you can take the largest from small heap and the smallest from the large heap and calculate the average.

Follow up: if the value is known in certain range you can do bucket count.

/// <summary>

/// Leet code #295. Find Median from Data Stream

///

/// Median is the middle value in an ordered integer list. If the size of

/// the list is even, there is no middle value. So the median is the mean

/// of the two middle value.

///

/// For example,

/// [2,3,4], the median is 3

///

/// [2,3], the median is (2 + 3) / 2 = 2.5

///

/// Design a data structure that supports the following two operations:

///

/// void addNum(int num) - Add a integer number from the data stream to

/// the data structure.

/// double findMedian() - Return the median of all elements so far.

///

/// Example:

///

/// addNum(1)

/// addNum(2)

/// findMedian() -> 1.5

/// addNum(3)

/// findMedian() -> 2

///

/// Follow up:

///

/// If all integer numbers from the stream are between 0 and 100, how

/// would you optimize it?

/// If 99% of all integer numbers from the stream are between 0 and

/// 100, how would you optimize it?

/// </summary>

class MedianFinder

{

private:

priority\_queue<int, vector<int>, greater<int>> m\_Large;

priority\_queue<int> m\_Small;

public:

// Default constructor.

MedianFinder()

{

}

// Adds a number into the data structure.

void addNum(int num)

{

if ((m\_Small.size() == 0) || (m\_Small.top() > num))

{

m\_Small.push(num);

}

else

{

m\_Large.push(num);

}

if (m\_Small.size() > m\_Large.size() + 1)

{

m\_Large.push(m\_Small.top());

m\_Small.pop();

}

if (m\_Large.size() > m\_Small.size())

{

m\_Small.push(m\_Large.top());

m\_Large.pop();

}

}

// Returns the median of current data stream

double findMedian()

{

double value;

if (m\_Small.size() == m\_Large.size() + 1)

{

value = (double)m\_Small.top();

}

else

{

value = ((double)m\_Small.top() + (double)m\_Large.top()) / 2;

}

return value;

}

};