LeetCode Training Day 2 Sort

Today we will discuss some sort, the most common pattern in sort problem is that you put a bunch of items in a priority queue and then retrieve based on order. The data type for priority queue is in the table below:

|  |  |
| --- | --- |
| Language | Priority Queue |
| C++ | priority\_queue<T> |
| C# | PriorityQueue<TElement, TPriority> (starting .Net 6) |
| Java | PriorityQueue<T> |

Please notice that the implementation of of priority\_queue in the above 3 languages are very different. In C++, it stores the greatest value as the front of the queue, so if you need a lower value comes first, you store 0 – value. C++ also support complex data structure such as pair<T, T>, it sort based on the first value then the second value. This allows us to store a value and the element id together. The priority in C++ also accepts duplicated values. In C#, it allows you store the element with its value sorted, and the lowest value comes first by default. In Java, the priority queue also stores a value (which can be a complex type as well) with lowest first.

In C#, the priority queue allows you to remove by elements or value in the middle of the priority queue, so the internal implementation may be a Sorted Dictionary (such as red black tree). In C++ it does not allow you to peek or remove value in the middle, so the implementation is more like a heap based on array. In Java and Python, you can build the priority queue on any basic collection type.

The common operations are here:

|  |  |
| --- | --- |
| Language | Operations: |
| C++ | Get largest value: value = pq.top();  Remove top value: pq.pop();  Push new value: pq.push(v);  Get size of table: pq.size(); |
| C# | Get minum value: value = pq.Dequeue();  Push new value: pq.Equeue(element, value);  Get size of table: pq.Count; |
| Java | Get minum value: value = pq.peek();  Remove top value: pq.poll();  Push new value: pq.add(element);  Get size of table: pq.Count; |

Some people get confused with priority queue and sorted dictionary. The sorted dictionary for C++, C# and Java is listed below:

|  |  |
| --- | --- |
| Language | Sorted Dictionary |
| C++ | ordered\_map<TValue, TKey>; set<T> |
| C# | SortedDictionary<TValue,TKey>; SortedSet<T> |
| Java | TreeMap<Integer, String>; TreeSet<T> |

In general, they are quite similar, the key difference is that priority queue allow duplicates but sorted dictionary does not, so if you store two integers with same value to a sorted\_set in C++, one will be lost. To avoid it you can consider sorted\_map<T, T> by using value as key and count as value to take duplicate. Another difference is that priority queue normally does not allow you to remove a value in the middle, but sorted dictionary will allow you to do so.

SortedSet is used when you do not care the count or mapping. You can retrieve element by value, or by iterator (in C++). Just like SortedDictionary, it does not hold duplicate values, if you want to store duplicate values from an array, a option is to bundle the index with the value by using pair (in C++) or tuple (in C#), if the value and index are same type you can simply use list. For this case SortedSet can also be used as priority queue with a benefit which you can access the value from the middle.

In C++ ordered\_map<T> and set<T> support binary search, this is another reason I love C++ in LeetCode.

## 414. Third Maximum Number

Easy

Given an integer array nums, return the ***third distinct maximum*** number in this array. If the third maximum does not exist, return the ***maximum*** number.

**Example 1:**

**Input:** nums = [3,2,1]

**Output:** 1

**Explanation:**

The first distinct maximum is 3.

The second distinct maximum is 2.

The third distinct maximum is 1.

**Example 2:**

**Input:** nums = [1,2]

**Output:** 2

**Explanation:**

The first distinct maximum is 2.

The second distinct maximum is 1.

The third distinct maximum does not exist, so the maximum (2) is returned instead.

**Example 3:**

**Input:** nums = [2,2,3,1]

**Output:** 1

**Explanation:**

The first distinct maximum is 3.

The second distinct maximum is 2 (both 2's are counted together since they have the same value).

The third distinct maximum is 1.

**Constraints:**

* 1 <= nums.length <= 104
* -231 <= nums[i] <= 231 - 1

### Analysis:

Since we want distinct value, we use set<T> to dedup. Keep the set

/// <summary>

/// Leet code #414. Third Maximum Number

///

/// Given a non-empty array of integers, return the third maximum number

/// in this array. If it does not exist,

/// return the maximum number. The time complexity must be in O(n).

/// Example 1:

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

/// Output: 1

/// Explanation: The third maximum is 1.

///

/// Example 2:

/// Input: [1, 2]

/// Output: 2

/// Explanation: The third maximum does not exist, so the maximum (2) is

/// returned instead.

///

/// Example 3:

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

/// Output: 1

/// Explanation: Note that the third maximum here means the third

/// maximum distinct number.

/// Both numbers with value 2 are both considered as second maximum.

/// </summary>

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

{

set<int> max\_list;

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

{

max\_list.insert(nums[i]);

if (max\_list.size() > 3) max\_list.erase(max\_list.begin());

}

if (max\_list.size() == 3)

{

return \*max\_list.begin();

}

else

{

return \*max\_list.rbegin();

}

}

## 1331. Rank Transform of an Array

Easy

Given an array of integers arr, replace each element with its rank.

The rank represents how large the element is. The rank has the following rules:

* Rank is an integer starting from 1.
* The larger the element, the larger the rank. If two elements are equal, their rank must be the same.
* Rank should be as small as possible.

**Example 1:**

**Input:** arr = [40,10,20,30]

**Output:** [4,1,2,3]

**Explanation**: 40 is the largest element. 10 is the smallest. 20 is the second smallest. 30 is the third smallest.

**Example 2:**

**Input:** arr = [100,100,100]

**Output:** [1,1,1]

**Explanation**: Same elements share the same rank.

**Example 3:**

**Input:** arr = [37,12,28,9,100,56,80,5,12]

**Output:** [5,3,4,2,8,6,7,1,3]

**Constraints:**

* 0 <= arr.length <= 105
* -109 <= arr[i] <= 109

### Analysis:

Put the value in a sorted dictionary and add the index to each distinct value. Then iterate the sorted dictionary

/// <summary>

/// Leet code #1331. Rank Transform of an Array

///

/// Easy

///

/// Given an array of integers arr, replace each element with its rank.

///

/// The rank represents how large the element is. The rank has the

/// following rules:

///

/// Rank is an integer starting from 1.

/// The larger the element, the larger the rank. If two elements are

/// equal, their rank must be the same.

/// Rank should be as small as possible.

///

/// Example 1:

/// Input: arr = [40,10,20,30]

/// Output: [4,1,2,3]

/// Explanation: 40 is the largest element. 10 is the smallest. 20 is

/// the second smallest. 30 is the third smallest.

///

/// Example 2:

/// Input: arr = [100,100,100]

/// Output: [1,1,1]

/// Explanation: Same elements share the same rank.

///

/// Example 3:

/// Input: arr = [37,12,28,9,100,56,80,5,12]

/// Output: [5,3,4,2,8,6,7,1,3]

///

///

/// Constraints:

/// 1. 0 <= arr.length <= 10^5

/// 2. -10^9 <= arr[i] <= 10^9

/// </summary>

vector<int> LeetCodeSort::arrayRankTransform(vector<int>& arr)

{

vector<int> result(arr.size());

map<int, vector<int>> sorted\_arr;

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

{

sorted\_arr[arr[i]].push\_back(i);

}

int order = 1;

for (auto itr : sorted\_arr)

{

for (auto k : itr.second)

{

result[k] = order;

}

order++;

}

return result;

}

## 973. K Closest Points to Origin

Medium

Given an array of points where points[i] = [xi, yi] represents a point on the **X-Y** plane and an integer k, return the k closest points to the origin (0, 0).

The distance between two points on the **X-Y** plane is the Euclidean distance (i.e., √(x1 - x2)2 + (y1 - y2)2).

You may return the answer in **any order**. The answer is **guaranteed** to be **unique** (except for the order that it is in).

**Example 1:**

Chart, line chart

Description automatically generated

**Input:** points = [[1,3],[-2,2]], k = 1

**Output:** [[-2,2]]

**Explanation:**

The distance between (1, 3) and the origin is sqrt(10).

The distance between (-2, 2) and the origin is sqrt(8).

Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.

We only want the closest k = 1 points from the origin, so the answer is just [[-2,2]].

**Example 2:**

**Input:** points = [[3,3],[5,-1],[-2,4]], k = 2

**Output:** [[3,3],[-2,4]]

**Explanation:** The answer [[-2,4],[3,3]] would also be accepted.

**Constraints:**

* 1 <= k <= points.length <= 104
* -104 < xi, yi < 104

### Analysis:

Keep all the distance in a heap and select top K

/// <summary>

/// Leet code #973. K Closest Points to Origin

///

/// We have a list of points on the plane. Find the K closest points to the

/// origin (0, 0).

///

/// (Here, the distance between two points on a plane is the Euclidean

/// distance.)

///

/// You may return the answer in any order. The answer is guaranteed to be

/// unique (except for the order that it is in.)

///

/// Example 1:

///

/// Input: points = [[1,3],[-2,2]], K = 1

/// Output: [[-2,2]]

/// Explanation:

/// The distance between (1, 3) and the origin is sqrt(10).

/// The distance between (-2, 2) and the origin is sqrt(8).

/// Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.

/// We only want the closest K = 1 points from the origin, so the answer is

/// just [[-2,2]].

///

/// Example 2:

///

/// Input: points = [[3,3],[5,-1],[-2,4]], K = 2

/// Output: [[3,3],[-2,4]]

/// (The answer [[-2,4],[3,3]] would also be accepted.)

///

/// Note:

///

/// 1. 1 <= K <= points.length <= 10000

/// 2. -10000 < points[i][0] < 10000

/// 3. -10000 < points[i][1] < 10000

/// </summary>

vector<vector<int>> LeetCodeSort::kClosest(vector<vector<int>>& points, int K)

{

map<double, vector<int>> selection;

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

{

double distance = sqrt(pow(points[i][0], 2) + pow(points[i][1], 2));

selection[-distance] = points[i];

if (selection.size() > (size\_t) K) selection.erase(selection.begin());

}

vector<vector<int>> result;

for (auto itr : selection)

{

result.push\_back(itr.second);

}

return result;

}

## 1705. Maximum Number of Eaten Apples

Medium

There is a special kind of apple tree that grows apples every day for n days. On the ith day, the tree grows apples[i] apples that will rot after days[i] days, that is on day i + days[i] the apples will be rotten and cannot be eaten. On some days, the apple tree does not grow any apples, which are denoted by apples[i] == 0 and days[i] == 0.

You decided to eat **at most** one apple a day (to keep the doctors away). Note that you can keep eating after the first n days.

Given two integer arrays days and apples of length n, return the maximum number of apples you can eat.

**Example 1:**

**Input:** apples = [1,2,3,5,2], days = [3,2,1,4,2]

**Output:** 7

**Explanation:** You can eat 7 apples:

- On the first day, you eat an apple that grew on the first day.

- On the second day, you eat an apple that grew on the second day.

- On the third day, you eat an apple that grew on the second day. After this day, the apples that grew on the third day rot.

- On the fourth to the seventh days, you eat apples that grew on the fourth day.

**Example 2:**

**Input:** apples = [3,0,0,0,0,2], days = [3,0,0,0,0,2]

**Output:** 5

**Explanation:** You can eat 5 apples:

- On the first to the third day you eat apples that grew on the first day.

- Do nothing on the fouth and fifth days.

- On the sixth and seventh days you eat apples that grew on the sixth day.

**Constraints:**

* apples.length == n
* days.length == n
* 1 <= n <= 2 \* 104
* 0 <= apples[i], days[i] <= 2 \* 104
* days[i] = 0 if and only if apples[i] = 0.

### Analysis:

On every day you collect the apples and calculated rotten days put them in the priority queue.

/// <summary>

/// Leet code #1705. Maximum Number of Eaten Apples

///

/// Medium

///

/// There is a special kind of apple tree that grows apples every day

/// for n days. On the ith day, the tree grows apples[i] apples that

/// will rot after days[i] days, that is on day i + days[i] the apples

/// will be rotten and cannot be eaten. On some days, the apple tree

/// does not grow any apples, which are denoted by apples[i] == 0

/// and days[i] == 0.

///

/// You decided to eat at most one apple a day (to keep the doctors away).

/// Note that you can keep eating after the first n days.

///

/// Given two integer arrays days and apples of length n, return the

/// maximum number of apples you can eat.

///

/// Example 1:

/// Input: apples = [1,2,3,5,2], days = [3,2,1,4,2]

/// Output: 7

/// Explanation: You can eat 7 apples:

/// - On the first day, you eat an apple that grew on the first day.

/// - On the second day, you eat an apple that grew on the second day.

/// - On the third day, you eat an apple that grew on the second day.

/// After this day, the apples that grew on the third day rot.

/// - On the fourth to the seventh days, you eat apples that grew on the

/// fourth day.

///

/// Example 2:

/// Input: apples = [3,0,0,0,0,2], days = [3,0,0,0,0,2]

/// Output: 5

/// Explanation: You can eat 5 apples:

/// - On the first to the third day you eat apples that grew on the first

/// day.

/// - Do nothing on the fouth and fifth days.

/// - On the sixth and seventh days you eat apples that grew on the sixth

/// day.

///

/// Constraints:

/// 1. apples.length == n

/// 2. days.length == n

/// 3. 1 <= n <= 2 \* 10^4

/// 4. 0 <= apples[i], days[i] <= 2 \* 10^4

/// 5. days[i] = 0 if and only if apples[i] = 0.

/// </summary>

int LeetCodeSort::eatenApples(vector<int>& apples, vector<int>& days)

{

priority\_queue<pair<int, int>> pq;

int day = 0;

int result = 0;

int n = apples.size();

while (day < n || !pq.empty())

{

if (day < n && apples[day] != 0)

{

pq.push(make\_pair(0 - (day + days[day]), apples[day]));

}

while (!pq.empty() && 0 - pq.top().first <= day)

{

pq.pop();

}

if (!pq.empty())

{

pair<int, int> pos = pq.top();

pq.pop();

pos.second--;

result++;

if (pos.second > 0) pq.push(pos);

}

day++;

}

return result;

}

## 1921. Eliminate Maximum Number of Monsters

Medium

You are playing a video game where you are defending your city from a group of n monsters. You are given a **0-indexed** integer array dist of size n, where dist[i] is the **initial distance** in kilometers of the ith monster from the city.

The monsters walk toward the city at a **constant** speed. The speed of each monster is given to you in an integer array speed of size n, where speed[i] is the speed of the ith monster in kilometers per minute.

You have a weapon that, once fully charged, can eliminate a **single** monster. However, the weapon takes **one minute** to charge.The weapon is fully charged at the very start.

You lose when any monster reaches your city. If a monster reaches the city at the exact moment the weapon is fully charged, it counts as a **loss**, and the game ends before you can use your weapon.

Return the ***maximum*** number of monsters that you can eliminate before you lose, or n if you can eliminate all the monsters before they reach the city.

**Example 1:**

**Input:** dist = [1,3,4], speed = [1,1,1]

**Output:** 3

**Explanation:**

In the beginning, the distances of the monsters are [1,3,4]. You eliminate the first monster.

After a minute, the distances of the monsters are [X,2,3]. You eliminate the second monster.

After a minute, the distances of the monsters are [X,X,2]. You eliminate the thrid monster.

All 3 monsters can be eliminated.

**Example 2:**

**Input:** dist = [1,1,2,3], speed = [1,1,1,1]

**Output:** 1

**Explanation:**

In the beginning, the distances of the monsters are [1,1,2,3]. You eliminate the first monster.

After a minute, the distances of the monsters are [X,0,1,2], so you lose.

You can only eliminate 1 monster.

**Example 3:**

**Input:** dist = [3,2,4], speed = [5,3,2]

**Output:** 1

**Explanation:**

In the beginning, the distances of the monsters are [3,2,4]. You eliminate the first monster.

After a minute, the distances of the monsters are [X,0,2], so you lose.

You can only eliminate 1 monster.

**Constraints:**

* n == dist.length == speed.length
* 1 <= n <= 105
* 1 <= dist[i], speed[i] <= 105

### Analysis:

Sort the monster by their arriving time

/// <summary>

/// Leet code 1921. Eliminate Maximum Number of Monsters

///

/// Medium

///

/// You are playing a video game where you are defending your city from

/// a group of n monsters. You are given a 0-indexed integer array dist

/// of size n, where dist[i] is the initial distance in kilometers of

/// the ith monster from the city.

///

/// The monsters walk toward the city at a constant speed. The speed of

/// each monster is given to you in an integer array speed of size n,

/// where speed[i] is the speed of the ith monster in kilometers per

/// minute.

///

/// You have a weapon that, once fully charged, can eliminate a single

/// monster. However, the weapon takes one minute to charge.The weapon

/// is fully charged at the very start.

///

/// You lose when any monster reaches your city. If a monster reaches

/// the city at the exact moment the weapon is fully charged, it counts

/// as a loss, and the game ends before you can use your weapon.

///

/// Return the maximum number of monsters that you can eliminate before

/// you lose, or n if you can eliminate all the monsters before they

/// reach the city.

///

/// Example 1:

/// Input: dist = [1,3,4], speed = [1,1,1]

/// Output: 3

/// Explanation:

/// In the beginning, the distances of the monsters are [1,3,4]. You

/// eliminate the first monster.

/// After a minute, the distances of the monsters are [X,2,3]. You

/// eliminate the second monster.

/// After a minute, the distances of the monsters are [X,X,2]. You

/// eliminate the thrid monster.

/// All 3 monsters can be eliminated.

///

/// Example 2:

/// Input: dist = [1,1,2,3], speed = [1,1,1,1]

/// Output: 1

/// Explanation:

/// In the beginning, the distances of the monsters are [1,1,2,3].

/// You eliminate the first monster.

/// After a minute, the distances of the monsters are [X,0,1,2], so you

/// lose.

/// You can only eliminate 1 monster.

///

/// Example 3:

/// Input: dist = [3,2,4], speed = [5,3,2]

/// Output: 1

/// Explanation:

/// In the beginning, the distances of the monsters are [3,2,4].

/// You eliminate the first monster.

/// After a minute, the distances of the monsters are [X,0,2], so you lose.

/// You can only eliminate 1 monster.

///

/// Constraints:

/// 1. n == dist.length == speed.length

/// 2. 1 <= n <= 10^5

/// 3. 1 <= dist[i], speed[i] <= 10^5

/// </summary>

int LeetCodeSort::eliminateMaximum(vector<int>& dist, vector<int>& speed)

{

priority\_queue<pair<double, int>> pq;

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

{

pq.push(make\_pair(-((double)dist[i]) / speed[i], i));

}

int time = 0;

while (!pq.empty())

{

if (speed[pq.top().second] \* time >= dist[pq.top().second])

{

break;

}

pq.pop();

time++;

}

return time;

}

## 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.

Return 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]

**Constraints:**

* 1 <= nums1.length, nums2.length <= 105
* -109 <= nums1[i], nums2[i] <= 109
* nums1 and nums2 both are sorted in **ascending order**.
* 1 <= k <= 1000

### Analysis:

We select every element in nums1 and add with nums2[0] and push them to the priority queue and keep track the index of both arrays. Then we retrieved a smallest value from the heap, count the order and add the index in num2 if not reach end and calculate a new sum and throw it in the pq again. In this case the priority queue will never exceed the size of nums1.

Please notice we can not pick a pair of index and store back the next element in each array, because in this case one pair index pick will cause 2 new pairs inserted and your priority queue size will get bursted.

/// <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 < k); 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;

## 1882. Process Tasks Using Servers

Medium

You are given two **0-indexed** integer arrays servers and tasks of lengths n​​​​​​ and m​​​​​​ respectively. servers[i] is the **weight** of the i​​​​​​th​​​​ server, and tasks[j] is the **time needed** to process the j​​​​​​th​​​​ task **in seconds**.

Tasks are assigned to the servers using a **task queue**. Initially, all servers are free, and the queue is **empty**.

At second j, the jth task is **inserted** into the queue (starting with the 0th task being inserted at second 0). As long as there are free servers and the queue is not empty, the task in the front of the queue will be assigned to a free server with the **smallest weight**, and in case of a tie, it is assigned to a free server with the **smallest index**.

If there are no free servers and the queue is not empty, we wait until a server becomes free and immediately assign the next task. If multiple servers become free at the same time, then multiple tasks from the queue will be assigned **in order of insertion** following the weight and index priorities above.

A server that is assigned task j at second t will be free again at second t + tasks[j].

Build an array ans​​​​ of length m, where ans[j] is the **index** of the server the j​​​​​​th task will be assigned to.

Return the array ans​​​​.

**Example 1:**

**Input:** servers = [3,3,2], tasks = [1,2,3,2,1,2]

**Output:** [2,2,0,2,1,2]

**Explanation:** Events in chronological order go as follows:

- At second 0, task 0 is added and processed using server 2 until second 1.

- At second 1, server 2 becomes free. Task 1 is added and processed using server 2 until second 3.

- At second 2, task 2 is added and processed using server 0 until second 5.

- At second 3, server 2 becomes free. Task 3 is added and processed using server 2 until second 5.

- At second 4, task 4 is added and processed using server 1 until second 5.

- At second 5, all servers become free. Task 5 is added and processed using server 2 until second 7.

**Example 2:**

**Input:** servers = [5,1,4,3,2], tasks = [2,1,2,4,5,2,1]

**Output:** [1,4,1,4,1,3,2]

**Explanation:** Events in chronological order go as follows:

- At second 0, task 0 is added and processed using server 1 until second 2.

- At second 1, task 1 is added and processed using server 4 until second 2.

- At second 2, servers 1 and 4 become free. Task 2 is added and processed using server 1 until second 4.

- At second 3, task 3 is added and processed using server 4 until second 7.

- At second 4, server 1 becomes free. Task 4 is added and processed using server 1 until second 9.

- At second 5, task 5 is added and processed using server 3 until second 7.

- At second 6, task 6 is added and processed using server 2 until second 7.

**Constraints:**

* servers.length == n
* tasks.length == m
* 1 <= n, m <= 2 \* 105
* 1 <= servers[i], tasks[j] <= 2 \* 105

### Analysis:

We put all the free servers based on weight and busy servers based on their next available time (all of them starting with zero) in a priority queue and select the next available server from priority queue.

/// <summary>

/// Leet code 1882. Process Tasks Using Servers

///

/// Medium

///

/// You are given two 0-indexed integer arrays servers and tasks of lengths

/// n and m respectively. servers[i] is the weight of the ith server, and

/// tasks[j] is the time needed to process the jth task in seconds.

///

/// Tasks are assigned to the servers using a task queue. Initially, all

/// servers are free, and the queue is empty.

///

/// At second j, the jth task is inserted into the queue (starting with

/// the 0th task being inserted at second 0). As long as there are free

/// servers and the queue is not empty, the task in the front of the queue

/// will be assigned to a free server with the smallest weight, and in

/// case of a tie, it is assigned to a free server with the smallest index.

///

/// If there are no free servers and the queue is not empty, we wait until

/// a server becomes free and immediately assign the next task. If multiple

/// servers become free at the same time, then multiple tasks from the

/// queue will be assigned in order of insertion following the weight and

/// index priorities above.

///

/// A server that is assigned task j at second t will be free again at

/// second t + tasks[j].

///

/// Build an array ans of length m, where ans[j] is the index of the

/// server the jth task will be assigned to.

///

/// Return the array ans.

///

/// Example 1:

/// Input: servers = [3,3,2], tasks = [1,2,3,2,1,2]

/// Output: [2,2,0,2,1,2]

/// Explanation: Events in chronological order go as follows:

/// - At second 0, task 0 is added and processed using server 2 until

/// second 1.

/// - At second 1, server 2 becomes free. Task 1 is added and

/// processed using server 2 until second 3.

/// - At second 2, task 2 is added and processed using server 0 until

/// second 5.

/// - At second 3, server 2 becomes free. Task 3 is added and processed

/// using server 2 until second 5.

/// - At second 4, task 4 is added and processed using server 1 until

/// second 5.

/// - At second 5, all servers become free. Task 5 is added and processed

/// using server 2 until second 7.

///

/// Example 2:

/// Input: servers = [5,1,4,3,2], tasks = [2,1,2,4,5,2,1]

/// Output: [1,4,1,4,1,3,2]

/// Explanation: Events in chronological order go as follows:

/// - At second 0, task 0 is added and processed using server 1 until

/// second 2.

/// - At second 1, task 1 is added and processed using server 4 until

/// second 2.

/// - At second 2, servers 1 and 4 become free. Task 2 is added and

/// processed using server 1 until second 4.

/// - At second 3, task 3 is added and processed using server 4 until

/// second 7.

/// - At second 4, server 1 becomes free. Task 4 is added and processed

/// using server 1 until second 9.

/// - At second 5, task 5 is added and processed using server 3 until

/// second 7.

/// - At second 6, task 6 is added and processed using server 2 until

/// second 7.

///

/// Constraints:

/// 1. servers.length == n

/// 2. tasks.length == m

/// 3. 1 <= n, m <= 2 \* 10^5

/// 4. 1 <= servers[i], tasks[j] <= 2 \* 10^5

/// </summary>

vector<int> LeetCodeSort::assignTasks(vector<int>& servers, vector<int>& tasks)

{

set<pair<int, int>> free\_servers;

set<pair<int, int>> busy\_servers;

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

{

free\_servers.insert(make\_pair(servers[i], i));

}

vector<int> result;

int time = 0;

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

{

if (time <= (int)i) time = i;

int server\_id = -1;

if (free\_servers.empty())

{

time = busy\_servers.begin()->first;

}

while (!busy\_servers.empty() && busy\_servers.begin()->first <= time)

{

server\_id = busy\_servers.begin()->second;

free\_servers.insert(make\_pair(servers[server\_id], server\_id));

busy\_servers.erase(busy\_servers.begin());

}

server\_id = free\_servers.begin()->second;

result.push\_back(server\_id);

free\_servers.erase(free\_servers.begin());

busy\_servers.insert(make\_pair(time + tasks[i], server\_id));

}

return result;

}

|  |  |  |  |
| --- | --- | --- | --- |
| Sorting Algorithm | Complexity | Worst Case | How the partial result looks like |
| Selection | O(n^2) | O(n^2) | Always the minimum is in partial result |
| Insertion | O(n^2) | O(n^2) | Partial result is always sorted, but minimum is not necessary in partial result. |
| Bubble | O(n^2) | O(n^2) | In every iteration, the larger number is exchange one level up. |
| Merge sort | O(nlog(n)) | O(nlog(n)) | You first get a sorted pair of two, then group of 4, then group of 8… |
| Quick Sort | O(nlog(n)) | O(n^2) | The first number in the list will be put in middle, with left part less than it, right part greater than it, slowly you are building a BST. |
| Heap sort | O(nlog(n)) | O(nlog(n)) | close to quick sort, but only the left part build up the BST.  Please remember if you have a small M groups of sorted group, heap sort can give you a O(n) solution. |
| Binary Tree sort | O(nlog(n)) | O(n^2) | Almost same as quick sort |
| Bucket sort | O(n)\*O(mlog(m)) | O(n)\*O(mlog(m)) | bucket sort is normally used for a large data which should be sorted externally. |

In the above list, you need to pay attention to **merge sort, heap sort, tree sort** and **bucket sort**

## Bucket Sort

Bucket Sort is to divide the original data into multiple groups, the groups themselves are in order. After the data are in groups you can further do the next level bucket sort or simply select the minimum or maximum data in each group. Please look at the following example:

## 164. Maximum Gap

[Maximum Gap - LeetCode](https://leetcode.com/problems/maximum-gap/description/)

/// <summary>

/// Leet code #164. Maximum Gap

/// Given an unsorted array, find the maximum difference between the successive

/// elements in its sorted form.

/// Try to solve it in linear time/space.

/// Return 0 if the array contains less than 2 elements.

/// You may assume all elements in the array are non-negative integers and fit

/// in the 32-bit signed integer range.

/// </summary>

int LeetCode::maximumGap(vector<int>& nums)

{

if (nums.size() < 2) return 0;

int minimum = INT\_MAX;

int maximum = INT\_MIN;

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

{

if (nums[i] < minimum) minimum = nums[i];

if (nums[i] > maximum) maximum = nums[i];

}

// The plus one is to make sure we do not have a slot\_gap as 0.

int slot\_gap = (maximum - minimum) / nums.size() + 1;

// In case the slot gap is not divisible by the range.

vector<pair<int, int>> slot((maximum - minimum) / slot\_gap + 1, make\_pair(-1, -1));

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

{

int index = (nums[i] - minimum) / slot\_gap;

pair<int, int> pair = slot[index];

if (pair.first == -1)

{

pair.first = nums[i];

pair.second = nums[i];

}

else

{

if (nums[i] < pair.first)

{

pair.first = nums[i];

}

else if (nums[i] > pair.second)

{

pair.second = nums[i];

}

}

slot[index] = pair;

}

int max\_gap = INT\_MIN;

int previous = -1;

pair<int, int> pair;

if (slot.size() == 1)

{

pair = slot[0];

max\_gap = pair.second - pair.first;

}

else

{

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

{

pair = slot[i];

if (pair.first == -1) continue;

if ((previous != -1) && (pair.first - previous) > max\_gap)

{

max\_gap = pair.first - previous;

}

previous = pair.second;

}

}

return max\_gap;

}

## 274. H-Index

[H-Index - LeetCode](https://leetcode.com/problems/h-index/description/)

/// <summary>

/// Leet code #274. H-Index

/// Given an array of citations (each citation is a non-negative integer) of

/// a researcher, write a function to compute the researcher's h-index.

/// According to the definition of h-index on Wikipedia: "A scientist has index h

/// if h of his/her N papers have at

/// least h citations each, and the other N − h papers have no more than h citations each."

/// For example, given citations = [3, 0, 6, 1, 5], which means the researcher has 5 papers

/// in total and each of

/// them had received 3, 0, 6, 1, 5 citations respectively. Since the researcher has 3 papers

/// with at least 3

/// citations each and the remaining two with no more than 3 citations each, his h-index is 3.

/// Note: If there are several possible values for h, the maximum one is taken as the h-index.

/// Hint:

/// 1.An easy approach is to sort the array first.

/// 2.What are the possible values of h-index?

/// 3.A faster approach is to use extra space.

/// </summary>

int LeetCode::hIndex(vector<int>& citations)

{

vector<int> count\_map(citations.size() + 1);

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

{

if (citations[i] >= (int)citations.size())

{

count\_map[citations.size()]++;

}

else

{

count\_map[citations[i]]++;

}

}

int index;

for (index = citations.size(); index >= 0; index--)

{

if (index < (int)citations.size()) count\_map[index] += count\_map[index + 1];

if (count\_map[index] >= index) break;

}

return index;

}

## Dutch Flag Sort

Assume you have only 3 unique number with duplication in a long list, and you want to sort it. The solution is using 3 pointers, low, from left to right, keep track on the smallest number, high, from right to left, keep track on the largest number, mid is used to scan from left to right in case both end stuck on the middle number.

## 75. Sort Colors

[Sort Colors - LeetCode](https://leetcode.com/problems/sort-colors/description/)

/// <summary>

/// Leet code #75. Sort Colors

/// Given an array with n objects colored red, white or blue, sort them so

/// that objects of the same color are adjacent, with the colors in the order

/// red, white and blue.

/// Here, we will use the integers 0, 1, and 2 to represent the color red,

/// white, and blue respectively.

/// Note:

/// You are not suppose to use the library's sort function for this problem.

/// Follow up:

/// A rather straight forward solution is a two-pass algorithm using counting

/// sort.

/// First, iterate the array counting number of 0's, 1's, and 2's, then overwrite

/// array with total number of 0's, then 1's and followed by 2's.

/// Could you come up with an one-pass algorithm using only constant space?

/// </summary>

void LeetCode::sortColors(vector<int>& nums)

{

int low, high, mid;

low = 0;

high = nums.size() - 1;

mid = low;

while ((low <= high) && (mid <= high))

{

if (nums[low] == 0)

{

low++;

mid = low;

}

else if (nums[high] == 2)

{

high--;

}

else if (nums[mid] == 0)

{

swap(nums[low], nums[mid]);

low++;

}

else if (nums[mid] == 2)

{

swap(nums[mid], nums[high]);

high--;

}

else

{

mid++;

}

}

}

##### **A more shorter solution**

void LeetCode::sortColors(vector<int>& nums)

{

int low, high, mid;

low = 0;

high = nums.size() - 1;

mid = low;

while (mid <= high)

{

if (nums[mid] == 0)

{

swap(nums[low], nums[mid]);

low++;

mid++;

}

else if (nums[mid] == 2)

{

swap(nums[mid], nums[high]);

high--;

}

else

{

mid++;

}

}

}

## Wiggle Sort

/// <summary>

/// Leet code #280. Wiggle Sort

///

/// Given an unsorted array nums, reorder it in-place such that

/// nums[0] <= nums[1] >= nums[2] <= nums[3]....

/// For example, given nums = [3, 5, 2, 1, 6, 4],

/// one possible answer is [1, 6, 2, 5, 3, 4].

/// </summary>

void LeetCode::wiggleSort(vector<int>& nums)

{

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

{

if (i % 2 == 1)

{

if (nums[i] < nums[i - 1]) swap(nums[i], nums[i - 1]);

}

else

{

if (nums[i] > nums[i - 1]) swap(nums[i], nums[i - 1]);

}

}

}

/// <summary>

/// Leet code #324. Wiggle Sort II Add to List

/// Given an unsorted array nums, reorder it such that nums[0]

/// < nums[1] > nums[2] < nums[3]....

/// Example:

/// (1) Given nums = [1, 5, 1, 1, 6, 4], one possible answer is

/// [1, 4, 1, 5, 1, 6].

/// (2) Given nums = [1, 3, 2, 2, 3, 1], one possible answer is

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

/// Note:

/// You may assume all input has valid answer.

/// Follow Up:

/// Can you do it in O(n) time and/or in-place with O(1) extra space?

/// </summary>

void LeetCode::wiggleSortII(vector<int>& nums)

{

int n = nums.size();

// Find a median.

vector<int>::iterator middle\_ptr = nums.begin() + n / 2;

nth\_element(nums.begin(), middle\_ptr, nums.end());

int value = \*middle\_ptr;

// Index-rewiring.

#define A(i) nums[(1+2\*(i)) % (n|1)]

// 3-way-partition-to-wiggly in O(n) time with O(1) space.

int low = 0, mid = 0, high = n - 1;

while (mid <= high)

{

if (A(mid) > value)

{

swap(A(low), A(mid));

low++; mid++;

}

else if (A(mid) < value)

{

swap(A(mid), A(high));

high--;

}

else

{

mid++;

}

}

}