# Schedule Problem

There objective of this type of problem is to order a series of objects to meet some condition or get maximum return.

## 358. Rearrange String k Distance Apart

Hard

Given a non-empty string **s** and an integer **k**, rearrange the string such that the same characters are at least distance **k** from each other.

All input strings are given in lowercase letters. If it is not possible to rearrange the string, return an empty string "".

**Example 1:**

**Input:** s = "aabbcc", k = 3

**Output:** "abcabc"

**Explanation:** The same letters are at least distance 3 from each other.

**Example 2:**

**Input:** s = "aaabc", k = 3

**Output:** ""

**Explanation:** It is not possible to rearrange the string.

**Example 3:**

**Input:** s = "aaadbbcc", k = 2

**Output:** "abacabcd"

**Explanation:** The same letters are at least distance 2 from each other.

### Solution: On each position we will have a list of qualified characters to place, we choose the one with maximum count, then such character will only be qualified after K positions. We repeat this process until we place all the characters or failed to choose any qualified character.

/// <summary>

/// Leet code #358. Rearrange String k Distance Apart

///

/// Given a non-empty string s and an integer k, rearrange the string such

/// that the same characters are at least distance k from each other.

/// All input strings are given in lowercase letters. If it is not possible

/// to rearrange the string, return an empty string "".

///

/// Example 1:

/// s = "aabbcc", k = 3

/// Result: "abcabc"

/// The same letters are at least distance 3 from each other.

///

/// Example 2:

/// s = "aaabc", k = 3

/// Answer: ""

/// It is not possible to rearrange the string.

///

/// Example 3:

/// s = "aaadbbcc", k = 2

/// Answer: "abacabcd"

/// Another possible answer is: "abcabcda"

/// The same letters are at least distance 2 from each other.

/// </summary>

string LeetCode::rearrangeString(string s, int k)

{

vector<pair<int, int>> char\_map(26, { 0, -1 });

string result(s.size(), 0);

for (char ch : s)

{

char\_map[ch - 'a'].first++;

}

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

{

pair<int, int> max\_pos = make\_pair(0, -1);

for (int j = 0; j < 26; j++)

{

if (char\_map[j].first > max\_pos.first && char\_map[j].second <= (int)i)

{

max\_pos = make\_pair(char\_map[j].first, j);

}

}

if (max\_pos.second == -1) return "";

result[i] = max\_pos.second + 'a';

char\_map[max\_pos.second].first--;

char\_map[max\_pos.second].second = i + k;

}

return result;

}

## 621. Task Scheduler

Medium

Given a char array representing tasks CPU need to do. It contains capital letters A to Z where different letters represent different tasks. Tasks could be done without original order. Each task could be done in one interval. For each interval, CPU could finish one task or just be idle.

However, there is a non-negative cooling interval **n** that means between two **same tasks**, there must be at least n intervals that CPU are doing different tasks or just be idle.

You need to return the **least** number of intervals the CPU will take to finish all the given tasks.

**Example:**

**Input:** tasks = ["A","A","A","B","B","B"], n = 2

**Output:** 8

**Explanation:** A -> B -> idle -> A -> B -> idle -> A -> B.

**Note:**

1. The number of tasks is in the range [1, 10000].
2. The integer n is in the range [0, 100].

/// <summary>

/// Leet code #621. Task Scheduler

///

/// Given a char array representing tasks CPU need to do. It contains

/// capital letters A to Z where different letters represent different

/// tasks. Tasks could be done without original order. Each task could

/// be done in one interval. For each interval, CPU could finish one

/// task or just be idle.

/// However, there is a non-negative cooling interval n that means

/// between two same tasks, there must be at least n intervals that

/// CPU are doing different tasks or just be idle.

/// You need to return the least number of intervals the CPU will take

/// to finish all the given tasks.

///

/// Example 1:

/// Input: tasks = ['A','A','A','B','B','B'], n = 2

/// Output: 8

/// Explanation: A -> B -> idle -> A -> B -> idle -> A -> B.

///

/// Note:

/// 1.The number of tasks is in the range [1, 10000].

/// 2.The integer n is in the range [0, 100].

/// </summary>

int LeetCode::leastInterval(vector<char>& tasks, int n)

{

int result = 0;

unordered\_map<char, int> task\_count;

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

for (char task : tasks) task\_count[task]++;

for (auto itr : task\_count)

{

task\_queue.push(make\_pair(0, itr.second));

}

while (!task\_queue.empty())

{

result++;

// get top task;

pair<int, int> task = task\_queue.top();

// if closed task still not ready to schedule, we push a idle;

if (-task.first < result)

{

task\_queue.pop();

task.second--;

if (task.second != 0)

{

task\_queue.push(make\_pair(task.first - n - 1, task.second));

}

}

}

return result;

}

## 1054. Distant Barcodes

Medium

14210FavoriteShare

In a warehouse, there is a row of barcodes, where the i-th barcode is barcodes[i].

Rearrange the barcodes so that no two adjacent barcodes are equal.  You may return any answer, and it is guaranteed an answer exists.

**Example 1:**

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

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

**Example 2:**

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

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

**Note:**

1. 1 <= barcodes.length <= 10000
2. 1 <= barcodes[i] <= 10000

/// <summary>

/// Leet code #1054. Distant Barcodes

///

/// In a warehouse, there is a row of barcodes, where the i-th barcode is

/// barcodes[i].

///

/// Rearrange the barcodes so that no two adjacent barcodes are equal. You

/// may return any answer, and it is guaranteed an answer exists.

///

/// Example 1:

///

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

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

///

/// Example 2:

///

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

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

/// Note:

///

/// 1. 1 <= barcodes.length <= 10000

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

/// </summary>

vector<int> LeetCode::rearrangeBarcodes(vector<int>& barcodes)

{

priority\_queue<pair<int, int>> code\_heap;

unordered\_map<int, int> code\_count;

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

{

code\_count[barcodes[i]]++;

}

for (auto itr : code\_count)

{

code\_heap.push(make\_pair(itr.second, itr.first));

}

vector<int> result;

while (!code\_heap.empty())

{

pair<int, int> code, next;

code = code\_heap.top();

code\_heap.pop();

if (result.empty() || result.back() != code.second)

{

result.push\_back(code.second);

code.first--;

if (code.first > 0) code\_heap.push(code);

}

else

{

next = code\_heap.top();

code\_heap.pop();

result.push\_back(next.second);

next.first--;

if (next.first > 0) code\_heap.push(next);

if (code.first > 0) code\_heap.push(code);

}

}

return result;

}

## 630. Course Schedule III

Hard

There are n different online courses numbered from 1 to n. Each course has some duration(course length) t and closed on dth day. A course should be taken **continuously** for t days and must be finished before or on the dth day. You will start at the 1st day.

Given n online courses represented by pairs (t,d), your task is to find the maximal number of courses that can be taken.

**Example:**

**Input:** [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]

**Output:** 3

**Explanation:**

There're totally 4 courses, but you can take 3 courses at most:

First, take the 1st course, it costs 100 days so you will finish it on the 100th day, and ready to take the next course on the 101st day.

Second, take the 3rd course, it costs 1000 days so you will finish it on the 1100th day, and ready to take the next course on the 1101st day.

Third, take the 2nd course, it costs 200 days so you will finish it on the 1300th day.

The 4th course cannot be taken now, since you will finish it on the 3300th day, which exceeds the closed date.

**Note:**

1. The integer 1 <= d, t, n <= 10,000.
2. You can't take two courses simultaneously.

##### Solution: We sort the deadline first, and check to see if every course can be completed by specific time, if we have any course which can not be completed, and if this course has a smaller duration than any course we have already chosen, we will replace that one. The idea is that if we can not complete all the courses, then we choose the most economic ones, here it means least duration.

/// <summary>

/// Leet code #630. Course Schedule III

///

/// There are n different online courses numbered from 1 to n. Each course

/// has some duration(course length) t and closed on dth day. A course

/// should be taken continuously for t days and must be finished before or

/// on the dth day. You will start at the 1st day.

///

/// Given n online courses represented by pairs (t,d), your task is to

/// find the maximal number of courses that can be taken.

/// Example:

/// Input: [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]

/// Output: 3

/// Explanation:

/// There're totally 4 courses, but you can take 3 courses at most:

/// First, take the 1st course, it costs 100 days so you will finish it on

/// the 100th day, and ready to take the next course on the 101st day.

/// Second, take the 3rd course, it costs 1000 days so you will finish it

/// on the 1100th day, and ready to take the next course on the 1101st day.

/// Third, take the 2nd course, it costs 200 days so you will finish it on

/// the 1300th day.

/// The 4th course cannot be taken now, since you will finish it on the

/// 3300th day, which exceeds the closed date.

/// Note:

/// The integer 1 <= d, t, n <= 10,000.

/// You can't take two courses simultaneously.

/// </summary>

int LeetCode::scheduleCourse(vector<vector<int>>& courses)

{

struct dead\_line\_compare

{

bool operator() (vector<int>& a, vector<int>&b)

{

if (a[1] == b[1]) return (a[0] < b[0]);

else return(a[1] < b[1]);

}

};

sort(courses.begin(), courses.end(), dead\_line\_compare());

int time = 0;

priority\_queue<int> course\_duration;

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

{

pair<int, int> schedule = make\_pair(courses[i][0], courses[i][1]);

// if start time later than now, we are greedy

if (schedule.second >= time + schedule.first)

{

course\_duration.push(schedule.first);

time += schedule.first;

}

// if we can not make it, we replace the longest course

else

{

if ((!course\_duration.empty()) &&

(course\_duration.top() > schedule.first))

{

time -= (course\_duration.top() - schedule.first);

course\_duration.pop();

course\_duration.push(schedule.first);

}

}

}

return course\_duration.size();

}

## 646. Maximum Length of Pair Chain

Medium

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number.

Now, we define a pair (c, d) can follow another pair (a, b) if and only if b < c. Chain of pairs can be formed in this fashion.

Given a set of pairs, find the length longest chain which can be formed. You needn't use up all the given pairs. You can select pairs in any order.

**Example 1:**

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

**Output:** 2

**Explanation:** The longest chain is [1,2] -> [3,4]

**Note:**

1. The number of given pairs will be in the range [1, 1000].

##### Solution: We sort the end positions, for any specific end position, we choose the largest start position. After such list is built, we iterate from beginning to end.

/// <summary>

/// Leet code #646. Maximum Length of Pair Chain

///

/// You are given n pairs of numbers. In every pair, the first number is

/// always smaller than the second number.

/// Now, we define a pair (c, d) can follow another pair (a, b) if and

/// only if b < c. Chain of pairs can be formed in this fashion.

///

/// Given a set of pairs, find the length longest chain which can be

/// formed. You needn't use up all the given pairs. You can select pairs

/// in any order.

///

/// Example 1:

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

/// Output: 2

/// Explanation: The longest chain is [1,2] -> [3,4]

/// Note:

/// The number of given pairs will be in the range [1, 1000].

/// </summary>

int LeetCode::findLongestChain(vector<vector<int>>& pairs)

{

map<int, vector<int>> schedule\_map;

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

{

if ((schedule\_map.count(pairs[i][1]) == 0) ||

(schedule\_map[pairs[i][1]][0] < pairs[i][0]))

{

schedule\_map[pairs[i][1]] = pairs[i];

}

}

int count = 0, last = 0;

for (map<int, vector<int>>::iterator itr = schedule\_map.begin();

itr != schedule\_map.end(); itr++)

{

if (count == 0 || itr->second[0] > last)

{

count++;

last = itr->second[1];

}

}

return count;

}

## 502. IPO

Hard

22021FavoriteShare

Suppose LeetCode will start its IPO soon. In order to sell a good price of its shares to Venture Capital, LeetCode would like to work on some projects to increase its capital before the IPO. Since it has limited resources, it can only finish at most **k** distinct projects before the IPO. Help LeetCode design the best way to maximize its total capital after finishing at most **k** distinct projects.

You are given several projects. For each project **i**, it has a pure profit **Pi** and a minimum capital of **Ci** is needed to start the corresponding project. Initially, you have **W** capital. When you finish a project, you will obtain its pure profit and the profit will be added to your total capital.

To sum up, pick a list of at most **k** distinct projects from given projects to maximize your final capital, and output your final maximized capital.

**Example 1:**

**Input:** k=2, W=0, Profits=[1,2,3], Capital=[0,1,1].

**Output:** 4

**Explanation:** Since your initial capital is 0, you can only start the project indexed 0.

After finishing it you will obtain profit 1 and your capital becomes 1.

With capital 1, you can either start the project indexed 1 or the project indexed 2.

Since you can choose at most 2 projects, you need to finish the project indexed 2 to get the maximum capital.

Therefore, output the final maximized capital, which is 0 + 1 + 3 = 4.

**Note:**

1. You may assume all numbers in the input are non-negative integers.
2. The length of Profits array and Capital array will not exceed 50,000.
3. The answer is guaranteed to fit in a 32-bit signed integer.

##### Solution: use the idea of topology sort, and choose the project which is satisfied by the current capital, and choose the one with maximum return (return = profit – capital) and process all the projects.

/// <summary>

/// Leet code # 502. IPO

///

/// Suppose LeetCode will start its IPO soon. In order to sell a good price

/// of its shares to Venture Capital, LeetCode would like to work on some

/// projects to increase its capital before the IPO. Since it has limited

/// resources, it can only finish at most k distinct projects before the IPO.

/// Help LeetCode design the best way to maximize its total capital after

/// finishing at most k distinct projects.

///

/// You are given several projects. For each project i, it has a pure profit

/// Pi and a minimum capital of Ci is needed to start the corresponding project.

/// Initially, you have W capital. When you finish a project, you will obtain

/// its pure profit and the profit will be added to your total capital.

///

/// To sum up, pick a list of at most k distinct projects from given projects

/// to maximize your final capital, and output your final maximized capital.

///

/// Example 1:

///

/// Input: k=2, W=0, Profits=[1,2,3], Capital=[0,1,1].

///

/// Output: 4

/// Explanation: Since your initial capital is 0, you can only start the project

/// indexed 0.

/// After finishing it you will obtain profit 1 and your capital becomes 1.

/// With capital 1, you can either start the project indexed 1 or the project indexed 2.

/// Since you can choose at most 2 projects, you need to finish the project indexed 2

/// to get the maximum capital.

/// Therefore, output the final maximized capital, which is 0 + 1 + 3 = 4.

/// Note:

/// 1.You may assume all numbers in the input are non-negative integers.

/// 2.The length of Profits array and Capital array will not exceed 50,000.

/// 3.The answer is guaranteed to fit in a 32-bit signed integer.

/// </summary>

int LeetCode::findMaximizedCapital(int k, int W, vector<int>& Profits, vector<int>& Capital)

{

priority\_queue<pair<int, int>> capital\_map;

priority\_queue<int> profit\_map;

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

{

capital\_map.push(make\_pair(-Capital[i], Profits[i]));

}

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

{

while (!capital\_map.empty() && -capital\_map.top().first <= W)

{

profit\_map.push(capital\_map.top().second);

capital\_map.pop();

}

if (profit\_map.empty())

{

break;

}

W += profit\_map.top();

profit\_map.pop();

}

return W;

}

## 1642. Furthest Building You Can Reach

Medium

You are given an integer array heights representing the heights of buildings, some bricks, and some ladders.

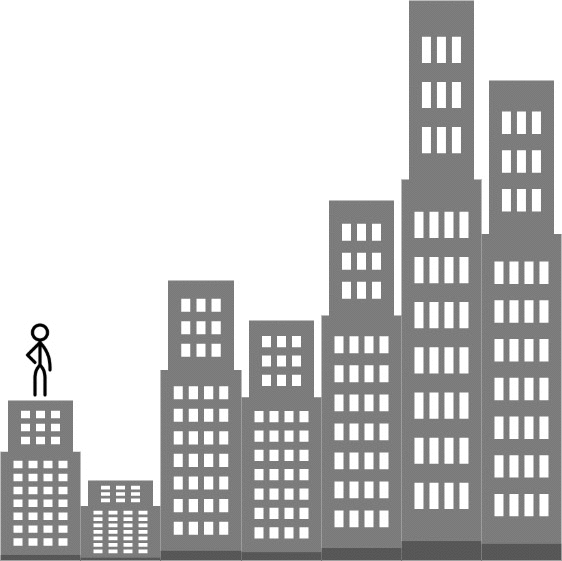
You start your journey from building 0 and move to the next building by possibly using bricks or ladders.

While moving from building i to building i+1 (**0-indexed**),

* If the current building's height is **greater than or equal** to the next building's height, you do **not** need a ladder or bricks.
* If the current building's height is **less than** the next building's height, you can either use **one ladder** or (h[i+1] - h[i]) **bricks**.

*Return the furthest building index (0-indexed) you can reach if you use the given ladders and bricks optimally.*

**Example 1:**



**Input:** heights = [4,2,7,6,9,14,12], bricks = 5, ladders = 1

**Output:** 4

**Explanation:** Starting at building 0, you can follow these steps:

- Go to building 1 without using ladders nor bricks since 4 >= 2.

- Go to building 2 using 5 bricks. You must use either bricks or ladders because 2 < 7.

- Go to building 3 without using ladders nor bricks since 7 >= 6.

- Go to building 4 using your only ladder. You must use either bricks or ladders because 6 < 9.

It is impossible to go beyond building 4 because you do not have any more bricks or ladders.

**Example 2:**

**Input:** heights = [4,12,2,7,3,18,20,3,19], bricks = 10, ladders = 2

**Output:** 7

**Example 3:**

**Input:** heights = [14,3,19,3], bricks = 17, ladders = 0

**Output:** 3

**Constraints:**

* 1 <= heights.length <= 105
* 1 <= heights[i] <= 106
* 0 <= bricks <= 109
* 0 <= ladders <= heights.length

##### Solution: We use all ladders first and then pop up the minimum gap to replace by bricks, keep on doing so until all ladders and bricks are used.

/// <summary>

/// Leet code #1642. Furthest Building You Can Reach

///

/// Medium

///

/// You are given an integer array heights representing the heights of

/// buildings, some bricks, and some ladders.

///

/// You start your journey from building 0 and move to the next building

/// by possibly using bricks or ladders.

///

/// While moving from building i to building i+1 (0-indexed),

///

/// If the current building's height is greater than or equal to the next

/// building's height, you do not need a ladder or bricks.

/// If the current building's height is less than the next building's height,

/// you can either use one ladder or (h[i+1] - h[i]) bricks.

/// Return the furthest building index (0-indexed) you can reach if you use

/// the given ladders and bricks optimally.

///

/// Example 1:

/// Input: heights = [4,2,7,6,9,14,12], bricks = 5, ladders = 1

/// Output: 4

/// Explanation: Starting at building 0, you can follow these steps:

/// - Go to building 1 without using ladders nor bricks since 4 >= 2.

/// - Go to building 2 using 5 bricks. You must use either bricks or

/// ladders because 2 < 7.

/// - Go to building 3 without using ladders nor bricks since 7 >= 6.

/// - Go to building 4 using your only ladder. You must use either bricks or

/// ladders because 6 < 9.

/// It is impossible to go beyond building 4 because you do not have any

/// more bricks or ladders.

///

/// Example 2:

/// Input: heights = [4,12,2,7,3,18,20,3,19], bricks = 10, ladders = 2

/// Output: 7

///

/// Example 3:

/// Input: heights = [14,3,19,3], bricks = 17, ladders = 0

/// Output: 3

///

/// Constraints:

/// 1. 1 <= heights.length <= 10^5

/// 2. 1 <= heights[i] <= 10^6

/// 3. 0 <= bricks <= 10^9

/// 4. 0 <= ladders <= heights.length

/// </summary>

int LeetCodeGreedy::furthestBuilding(vector<int>& heights, int bricks, int ladders)

{

priority\_queue<int> pq;

int result = 0;

while (result < (int)heights.size() -1)

{

int next = result + 1;

if (heights[next] <= heights[result])

{

result++;

}

else

{

int gap = heights[result] - heights[next];

pq.push(gap);

if ((int)pq.size() > ladders)

{

bricks += pq.top();

pq.pop();

if (bricks < 0) break;

}

result++;

}

}

return result;

}

## 1665. Minimum Initial Energy to Finish Tasks

Hard

You are given an array tasks where tasks[i] = [actuali, minimumi]:

* actuali is the actual amount of energy you **spend to finish** the ith task.
* minimumi is the minimum amount of energy you **require to begin** the ith task.

For example, if the task is [10, 12] and your current energy is 11, you cannot start this task. However, if your current energy is 13, you can complete this task, and your energy will be 3 after finishing it.

You can finish the tasks in **any order** you like.

Return *the****minimum****initial amount of energy you will need* *to finish all the tasks*.

**Example 1:**

**Input:** tasks = [[1,2],[2,4],[4,8]]

**Output:** 8

**Explanation:**

Starting with 8 energy, we finish the tasks in the following order:

- 3rd task. Now energy = 8 - 4 = 4.

- 2nd task. Now energy = 4 - 2 = 2.

- 1st task. Now energy = 2 - 1 = 1.

Notice that even though we have leftover energy, starting with 7 energy does not work because we cannot do the 3rd task.

**Example 2:**

**Input:** tasks = [[1,3],[2,4],[10,11],[10,12],[8,9]]

**Output:** 32

**Explanation:**

Starting with 32 energy, we finish the tasks in the following order:

- 1st task. Now energy = 32 - 1 = 31.

- 2nd task. Now energy = 31 - 2 = 29.

- 3rd task. Now energy = 29 - 10 = 19.

- 4th task. Now energy = 19 - 10 = 9.

- 5th task. Now energy = 9 - 8 = 1.

**Example 3:**

**Input:** tasks = [[1,7],[2,8],[3,9],[4,10],[5,11],[6,12]]

**Output:** 27

**Explanation:**

Starting with 27 energy, we finish the tasks in the following order:

- 5th task. Now energy = 27 - 5 = 22.

- 2nd task. Now energy = 22 - 2 = 20.

- 3rd task. Now energy = 20 - 3 = 17.

- 1st task. Now energy = 17 - 1 = 16.

- 4th task. Now energy = 16 - 4 = 12.

- 6th task. Now energy = 12 - 6 = 6.

**Constraints:**

* 1 <= tasks.length <= 105
* 1 <= actual​i <= minimumi <= 104

##### Solution: Ideally, we add up all used energy will become the answer, but we may have some gaps. We deduct used energy from minimum energy required, to get the final energy, and we then count from zero to reach all these final energies to get the gaps if any.

/// <summary>

/// Leet code #1665. Minimum Initial Energy to Finish Tasks

///

/// Hard

///

/// You are given an array tasks where tasks[i] = [actual[i], minimum[i]]:

///

/// actual[i] is the actual amount of energy you spend to finish the ith

/// task.

/// minimum[i] is the minimum amount of energy you require to begin the

/// ith task.

/// For example, if the task is [10, 12] and your current energy is 11,

/// you cannot start this task. However, if your current energy is 13,

/// you can complete this task, and your energy will be 3 after finishing

/// it.

///

/// You can finish the tasks in any order you like.

///

/// Return the minimum initial amount of energy you will need to finish

/// all the tasks.

///

/// Example 1:

/// Input: tasks = [[1,2],[2,4],[4,8]]

/// Output: 8

/// Explanation:

/// Starting with 8 energy, we finish the tasks in the following order:

/// - 3rd task. Now energy = 8 - 4 = 4.

/// - 2nd task. Now energy = 4 - 2 = 2.

/// - 1st task. Now energy = 2 - 1 = 1.

/// Notice that even though we have leftover energy, starting with 7

/// energy does not work because we cannot do the 3rd task.

///

/// Example 2:

/// Input: tasks = [[1,3],[2,4],[10,11],[10,12],[8,9]]

/// Output: 32

/// Explanation:

/// Starting with 32 energy, we finish the tasks in the following order:

/// - 1st task. Now energy = 32 - 1 = 31.

/// - 2nd task. Now energy = 31 - 2 = 29.

/// - 3rd task. Now energy = 29 - 10 = 19.

/// - 4th task. Now energy = 19 - 10 = 9.

/// - 5th task. Now energy = 9 - 8 = 1.

///

/// Example 3:

/// Input: tasks = [[1,7],[2,8],[3,9],[4,10],[5,11],[6,12]]

/// Output: 27

/// Explanation:

/// Starting with 27 energy, we finish the tasks in the following order:

/// - 5th task. Now energy = 27 - 5 = 22.

/// - 2nd task. Now energy = 22 - 2 = 20.

/// - 3rd task. Now energy = 20 - 3 = 17.

/// - 1st task. Now energy = 17 - 1 = 16.

/// - 4th task. Now energy = 16 - 4 = 12.

/// - 6th task. Now energy = 12 - 6 = 6.

///

/// Constraints:

/// 1. 1 <= tasks.length <= 10^5

/// 2. 1 <= actual​[i] <= minimum[i] <= 10^4

/// </summary>

int LeetCodeGreedy::minimumEffort(vector<vector<int>>& tasks)

{

priority\_queue<pair<int, int>> task\_pq;

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

{

task\_pq.push(make\_pair(tasks[i][0] - tasks[i][1], i));

}

int result = 0;

while (!task\_pq.empty())

{

pair<int, int> task = task\_pq.top();

task\_pq.pop();

int point = 0 - task.first;

if (point > result) result = tasks[task.second][1];

else result = result + tasks[task.second][0];

}

return result;

}