LeetCode Training Day 11 DP II

Today we will continue to discuss Dynamic Programming, unlike yesterday we simply **look back** to build the result from previous result. We will do it differently. For example, we can build from beginning, push the intermediate result to future and get aggregated there, then we can process them in the future. We call this methodology as **Carry Forward.**

## 55. Jump Game

Medium

You are given an integer array nums. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return true*if you can reach the last index, or*false*otherwise*.

**Example 1:**

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

**Output:** true

**Explanation:** Jump 1 step from index 0 to 1, then 3 steps to the last index.

**Example 2:**

**Input:** nums = [3,2,1,0,4]

**Output:** false

**Explanation:** You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.

**Constraints:**

* 1 <= nums.length <= 104
* 0 <= nums[i] <= 105

### Analysis:

On every position you can know the right most position you can jump to. We process every position from left to right.

/// <summary>

/// Leet Code 55. Jump Game

///

/// Medium

///

/// You are given an integer array nums. You are initially positioned at

/// the array's first index, and each element in the array represents

/// your maximum jump length at that position.

///

/// Return true if you can reach the last index, or false otherwise.

///

/// Example 1:

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

/// Output: true

/// Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last

/// index.

///

/// Example 2:

/// Input: nums = [3,2,1,0,4]

/// Output: false

/// Explanation: You will always arrive at index 3 no matter what. Its

/// maximum jump length is 0, which makes it impossible to reach the last

/// index.

///

/// Constraints:

/// 1. 1 <= nums.length <= 10^4

/// 2. 0 <= nums[i] <= 10^5

/// </summary>

bool LeetCodeDP::canJump(vector<int>& nums)

{

int right\_most = 0;

int index = 0;

while (index < right\_most + 1 && right\_most < (int)nums.size() - 1)

{

right\_most = max(right\_most, index + nums[index]);

index++;

}

if (right\_most >= (int)nums.size() - 1)

{

return true;

}

else

{

return false;

}

}

## 45. Jump Game II

Medium

Given an array of non-negative integers nums, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

You can assume that you can always reach the last index.

**Example 1:**

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

**Output:** 2

**Explanation:** The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

**Example 2:**

**Input:** nums = [2,3,0,1,4]

**Output:** 2

**Constraints:**

* 1 <= nums.length <= 104
* 0 <= nums[i] <= 1000

### Analysis:

For every step we iterate to right most position from last step.

/// <summary>

/// Leet Code 45. Jump Game II

///

/// Medium

///

/// Given an array of non-negative integers nums, you are initially

/// positioned at the first index of the array.

///

/// Each element in the array represents your maximum jump length at that

/// position.

///

/// Your goal is to reach the last index in the minimum number of jumps.

///

/// You can assume that you can always reach the last index.

///

/// Example 1:

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

/// Output: 2

/// Explanation: The minimum number of jumps to reach the last index is 2.

/// Jump 1 step from index 0 to 1, then 3 steps to the last index.

///

/// Example 2:

/// Input: nums = [2,3,0,1,4]

/// Output: 2

///

/// Constraints:

/// 1. 1 <= nums.length <= 10^4

/// 2. 0 <= nums[i] <= 1000

/// </summary>

int LeetCodeDP::jump(vector<int>& nums)

{

int end = 0;

int index = 0;

int result = 0;

while (end < (int)nums.size() - 1)

{

int right = end;

result++;

while (index <= right)

{

end = max(end, index + nums[index]);

index++;

}

}

return result;

}

## 91. Decode Ways

Medium

A message containing letters from A-Z can be **encoded** into numbers using the following mapping:

'A' -> "1"

'B' -> "2"

...

'Z' -> "26"

To **decode** an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, "11106" can be mapped into:

* "AAJF" with the grouping (1 1 10 6)
* "KJF" with the grouping (11 10 6)

Note that the grouping (1 11 06) is invalid because "06" cannot be mapped into 'F' since "6" is different from "06".

Given a string s containing only digits, return *the****number****of ways to****decode****it*.

The answer is guaranteed to fit in a **32-bit** integer.

**Example 1:**

**Input:** s = "12"

**Output:** 2

**Explanation:** "12" could be decoded as "AB" (1 2) or "L" (12).

**Example 2:**

**Input:** s = "226"

**Output:** 3

**Explanation:** "226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

**Example 3:**

**Input:** s = "0"

**Output:** 0

**Explanation:** There is no character that is mapped to a number starting with 0.

The only valid mappings with 0 are 'J' -> "10" and 'T' -> "20", neither of which start with 0.

Hence, there are no valid ways to decode this since all digits need to be mapped.

**Example 4:**

**Input:** s = "06"

**Output:** 0

**Explanation:** "06" cannot be mapped to "F" because of the leading zero ("6" is different from "06").

**Constraints:**

* 1 <= s.length <= 100
* s contains only digits and may contain leading zero(s).

### Analysis:

For any non-zero digit, we can map to 1 character, thus we carry it to position + 1, for any two digits less than 26, we can also map to 1 character, this we carry it as another result to position +2. At the end point we collect total numbers.

/// <summary>

/// Leet code 91. Decode Ways

///

/// A message containing letters from A-Z is being encoded to numbers using

/// the following mapping:

///

/// 'A' -> 1

/// 'B' -> 2

/// ...

/// 'Z' -> 26

/// Given a non-empty string containing only digits, determine the total

/// number of ways to decode it.

///

/// Example 1:

///

/// Input: "12"

/// Output: 2

/// Explanation: It could be decoded as "AB" (1 2) or "L" (12).

///

/// Example 2:

///

/// Input: "226"

/// Output: 3

/// Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF"

/// (2 2 6).

/// </summary>

int LeetCodeDP::numDecodings(string s)

{

if (s.empty()) return 0;

vector<int> dp(s.size()+1);

dp[0] = 1;

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

{

if (s[i] != '0') dp[i+1] += dp[i];

if (i < s.size() - 1)

{

if ((s[i] == '1') || (s[i] == '2' && s[i + 1] >= '0' && s[i + 1] <= '6'))

{

dp[i + 2] += dp[i];

}

}

}

return dp[s.size()];

}

## 264. Ugly Number II

Medium

An **ugly number** is a positive integer whose prime factors are limited to 2, 3, and 5.

Given an integer n, return *the* nth ***ugly number***.

**Example 1:**

**Input:** n = 10

**Output:** 12

**Explanation:** [1, 2, 3, 4, 5, 6, 8, 9, 10, 12] is the sequence of the first 10 ugly numbers.

**Example 2:**

**Input:** n = 1

**Output:** 1

**Explanation:** 1 has no prime factors, therefore all of its prime factors are limited to 2, 3, and 5.

**Constraints:**

* 1 <= n <= 1690

### Analysis:

We store smallest ugly number for each factor, select the minimum number from all factor candidate and move this index one step forward.

/// <summary>

/// Leet code #264. Ugly Number II

///

/// Write a program to find the n-th ugly number.

///

/// Ugly numbers are positive numbers whose prime factors only

/// include 2, 3, 5.

///

/// Example:

///

/// Input: n = 10

/// Output: 12

/// Explanation: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of

/// the first 10 ugly numbers.

/// Note:

///

/// 1. 1 is typically treated as an ugly number.

/// 2. n does not exceed 1690.

/// </summary>

int LeetCodeDP::nthUglyNumber(int n)

{

vector<int> result;

vector<int> dp(3);

vector<int> factor = { 2, 3, 5 };

result.push\_back(1);

while (result.size() < (size\_t)n)

{

int index = 0;

int min\_num = INT\_MAX;

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

{

if (min\_num > result[dp[i]] \* factor[i])

{

index = i;

min\_num = result[dp[i]] \* factor[i];

}

}

if (min\_num > result.back()) result.push\_back(min\_num);

dp[index]++;

}

return result.back();

}

## 313. Super Ugly Number

Medium

A **super ugly number** is a positive integer whose prime factors are in the array primes.

Given an integer n and an array of integers primes, return *the* nth ***super ugly number***.

The nth **super ugly number** is **guaranteed** to fit in a **32-bit** signed integer.

**Example 1:**

**Input:** n = 12, primes = [2,7,13,19]

**Output:** 32

**Explanation:** [1,2,4,7,8,13,14,16,19,26,28,32] is the sequence of the first 12 super ugly numbers given primes = [2,7,13,19].

**Example 2:**

**Input:** n = 1, primes = [2,3,5]

**Output:** 1

**Explanation:** 1 has no prime factors, therefore all of its prime factors are in the array primes = [2,3,5].

**Constraints:**

* 1 <= n <= 106
* 1 <= primes.length <= 100
* 2 <= primes[i] <= 1000
* primes[i] is **guaranteed** to be a prime number.
* All the values of primes are **unique** and sorted in **ascending order**.

### Analysis:

Same as above, We store smallest ugly number for each factor, select the minimum number from all factor candidate and move this index one step forward.

/// <summary>

/// Leet code #313. Super Ugly Number

///

/// Write a program to find the nth super ugly number.

///

/// Super ugly numbers are positive numbers whose all prime factors

/// are in the given prime list primes of size k.

///

/// Example:

///

/// Input: n = 12, primes = [2,7,13,19]

/// Output: 32

/// Explanation: [1,2,4,7,8,13,14,16,19,26,28,32] is the sequence of the

/// first 12 super ugly numbers given primes = [2,7,13,19] of size 4.

/// Note:

///

/// 1. 1 is a super ugly number for any given primes.

/// 2. The given numbers in primes are in ascending order.

/// 3. 0 < k ≤ 100, 0 < n ≤ 106, 0 < primes[i] < 1000.

/// 4. The nth super ugly number is guaranteed to fit in a 32-bit signed

/// integer.

/// </summary>

int LeetCodeDP::nthSuperUglyNumber(int n, vector<int>& primes)

{

vector<int> result;

vector<int> dp(primes.size());

result.push\_back(1);

while (result.size() < (size\_t)n)

{

int index = 0;

int min\_num = INT\_MAX;

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

{

int product = result[dp[i]] \* primes[i];

if (min\_num > product)

{

index = i;

min\_num = product;

}

}

if (min\_num > result.back()) result.push\_back(min\_num);

dp[index]++;

}

return result.back();

}

## 2054. Two Best Non-Overlapping Events

Medium

You are given a **0-indexed** 2D integer array of events where events[i] = [startTimei, endTimei, valuei]. The ith event starts at startTimeiand ends at endTimei, and if you attend this event, you will receive a value of valuei. You can choose **at most** **two** **non-overlapping** events to attend such that the sum of their values is **maximized**.

Return *this****maximum****sum.*

Note that the start time and end time is **inclusive**: that is, you cannot attend two events where one of them starts and the other ends at the same time. More specifically, if you attend an event with end time t, the next event must start at or after t + 1.

**Example 1:**

A picture containing background pattern

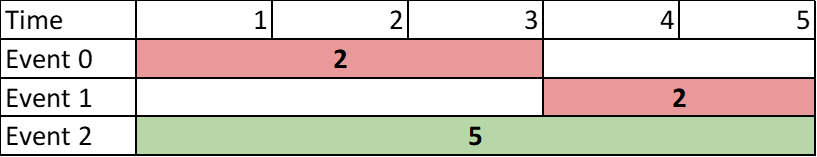
Description automatically generated

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

**Output:** 4

**Explanation:** Choose the green events, 0 and 1 for a sum of 2 + 2 = 4.

**Example 2:**



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

**Output:** 5

**Explanation:** Choose event 2 for a sum of 5.

**Example 3:**

A picture containing diagram

Description automatically generated

**Input:** events = [[1,5,3],[1,5,1],[6,6,5]]

**Output:** 8

**Explanation:** Choose events 0 and 2 for a sum of 3 + 5 = 8.

**Constraints:**

* 2 <= events.length <= 105
* events[i].length == 3
* 1 <= startTimei <= endTimei <= 109
* 1 <= valuei <= 106

### Analysis:

On every event we collect the maximum value before the event and add the current value, we also store the current value in the future to process next.

/// <summary>

/// Leet Code 2054. Two Best Non-Overlapping Events

///

/// Medium

///

/// You are given a 0-indexed 2D integer array of events where

/// events[i] = [startTimei, endTimei, valuei]. The ith event

/// starts at startTimei and ends at endTimei, and if you attend this

/// event, you will receive a value of valuei. You can choose at most

/// two non-overlapping events to attend such that the sum of their

/// values is maximized.

///

/// Return this maximum sum.

///

/// Note that the start time and end time is inclusive: that is, you

/// cannot attend two events where one of them starts and the other

/// ends at the same time. More specifically, if you attend an event

/// with end time t, the next event must start at or after t + 1.

///

/// Example 1:

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

/// Output: 4

/// Explanation: Choose the green events, 0 and 1 for a sum of 2 + 2 = 4.

///

/// Example 2:

/// Example 1 Diagram

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

/// Output: 5

/// Explanation: Choose event 2 for a sum of 5.

///

/// Example 3:

/// Input: events = [[1,5,3],[1,5,1],[6,6,5]]

/// Output: 8

/// Explanation: Choose events 0 and 2 for a sum of 3 + 5 = 8.

///

/// Constraints:

/// 1. 2 <= events.length <= 10^5

/// 2. events[i].length == 3

/// 3. 1 <= startTimei <= endTimei <= 10^9

/// 4. 1 <= valuei <= 10^6

/// </summary>

int LeetCodeDP::maxTwoEvents(vector<vector<int>>& events)

{

sort(events.begin(), events.end());

map<int, int> dp;

dp[0] = 0;

int prev = 0;

int result = 0;

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

{

while (!dp.empty() && dp.begin()->first < events[i][0])

{

prev = max(prev, dp.begin()->second);

dp.erase(dp.begin());

}

result = max(result, prev + events[i][2]);

dp[events[i][1]] = max(dp[events[i][1]], events[i][2]);

}

return result;

}

## 2008. Maximum Earnings From Taxi

Medium

There are n points on a road you are driving your taxi on. The n points on the road are labeled from 1 to n in the direction you are going, and you want to drive from point 1 to point n to make money by picking up passengers. You cannot change the direction of the taxi.

The passengers are represented by a **0-indexed** 2D integer array rides, where rides[i] = [starti, endi, tipi] denotes the ith passenger requesting a ride from point starti to point endi who is willing to give a tipi dollar tip.

For**each**passenger i you pick up, you **earn** endi - starti + tipi dollars. You may only drive **at most one**passenger at a time.

Given n and rides, return *the****maximum****number of dollars you can earn by picking up the passengers optimally.*

**Note:** You may drop off a passenger and pick up a different passenger at the same point.

**Example 1:**

**Input:** n = 5, rides = [[2,5,4],[1,5,1]]

**Output:** 7

**Explanation:** We can pick up passenger 0 to earn 5 - 2 + 4 = 7 dollars.

**Example 2:**

**Input:** n = 20, rides = [[1,6,1],[3,10,2],[10,12,3],[11,12,2],[12,15,2],[13,18,1]]

**Output:** 20

**Explanation:** We will pick up the following passengers:

- Drive passenger 1 from point 3 to point 10 for a profit of 10 - 3 + 2 = 9 dollars.

- Drive passenger 2 from point 10 to point 12 for a profit of 12 - 10 + 3 = 5 dollars.

- Drive passenger 5 from point 13 to point 18 for a profit of 18 - 13 + 1 = 6 dollars.

We earn 9 + 5 + 6 = 20 dollars in total.

**Constraints:**

* 1 <= n <= 105
* 1 <= rides.length <= 3 \* 104
* rides[i].length == 3
* 1 <= starti < endi <= n
* 1 <= tipi <= 105

### Analysis:

On every starting point, we collected the maximum earning until current stop, and we add the current ride value and put the maximum earning in the end position for future process.

/// <summary>

/// Leet Code 2008. Maximum Earnings From Taxi

///

/// Medium

///

/// There are n points on a road you are driving your taxi on. The n

/// points on the road are labeled from 1 to n in the direction you

/// are going, and you want to drive from point 1 to point n to make

/// money by picking up passengers. You cannot change the direction

/// of the taxi.

///

/// The passengers are represented by a 0-indexed 2D integer array

/// rides, where rides[i] = [starti, endi, tipi] denotes the ith

/// passenger requesting a ride from point starti to point endi who

/// is willing to give a tipi dollar tip.

///

/// For each passenger i you pick up, you earn endi - starti + tipi

/// dollars. You may only drive at most one passenger at a time.

///

/// Given n and rides, return the maximum number of dollars you can

/// earn by picking up the passengers optimally.

///

/// Note: You may drop off a passenger and pick up a different

/// passenger at the same point.

///

/// Example 1:

///

/// Input: n = 5, rides = [[2,5,4],[1,5,1]]

/// Output: 7

/// Explanation: We can pick up passenger 0 to earn 5 - 2 + 4 = 7

/// dollars.

///

/// Example 2:

///

/// Input: n = 20, rides = [[1,6,1],[3,10,2],[10,12,3],[11,12,2],

/// [12,15,2],[13,18,1]]

/// Output: 20

/// Explanation: We will pick up the following passengers:

/// - Drive passenger 1 from point 3 to point 10 for a profit of

/// 10 - 3 + 2 = 9 dollars.

/// - Drive passenger 2 from point 10 to point 12 for a profit

/// of 12 - 10 + 3 = 5 dollars.

/// - Drive passenger 5 from point 13 to point 18 for a profit

/// of 18 - 13 + 1 = 6 dollars.

/// We earn 9 + 5 + 6 = 20 dollars in total.

///

/// Constraints:

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

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

/// 3. rides[i].length == 3

/// 4. 1 <= starti < endi <= n

/// 5. 1 <= tipi <= 10^5

/// </summary>

long long LeetCodeDP::maxTaxiEarnings(int n, vector<vector<int>>& rides)

{

long long prev = 0;

int index = 0;

vector<long long> dp(n + 1);

long long result = 0;

sort(rides.begin(), rides.end());

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

{

while (index <= rides[i][0])

{

prev = max(prev, dp[index]);

index++;

}

int start = rides[i][0];

int end = rides[i][1];

int tip = rides[i][2];

dp[end] = max(dp[end], prev + (long long)end - (long long)start + (long long)tip);

result = max(result, dp[end]);

}

return result;

}

# Advanced Problems

## 639. Decode Ways II

Hard

A message containing letters from A-Z can be **encoded** into numbers using the following mapping:

'A' -> "1"

'B' -> "2"

...

'Z' -> "26"

To **decode** an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, "11106" can be mapped into:

* "AAJF" with the grouping (1 1 10 6)
* "KJF" with the grouping (11 10 6)

Note that the grouping (1 11 06) is invalid because "06" cannot be mapped into 'F' since "6" is different from "06".

**In addition** to the mapping above, an encoded message may contain the '\*' character, which can represent any digit from '1' to '9' ('0' is excluded). For example, the encoded message "1\*" may represent any of the encoded messages "11", "12", "13", "14", "15", "16", "17", "18", or "19". Decoding "1\*" is equivalent to decoding **any** of the encoded messages it can represent.

Given a string s consisting of digits and '\*' characters, return *the****number****of ways to****decode****it*.

Since the answer may be very large, return it **modulo** 109 + 7.

**Example 1:**

**Input:** s = "\*"

**Output:** 9

**Explanation:** The encoded message can represent any of the encoded messages "1", "2", "3", "4", "5", "6", "7", "8", or "9".

Each of these can be decoded to the strings "A", "B", "C", "D", "E", "F", "G", "H", and "I" respectively.

Hence, there are a total of 9 ways to decode "\*".

**Example 2:**

**Input:** s = "1\*"

**Output:** 18

**Explanation:** The encoded message can represent any of the encoded messages "11", "12", "13", "14", "15", "16", "17", "18", or "19".

Each of these encoded messages have 2 ways to be decoded (e.g. "11" can be decoded to "AA" or "K").

Hence, there are a total of 9 \* 2 = 18 ways to decode "1\*".

**Example 3:**

**Input:** s = "2\*"

**Output:** 15

**Explanation:** The encoded message can represent any of the encoded messages "21", "22", "23", "24", "25", "26", "27", "28", or "29".

"21", "22", "23", "24", "25", and "26" have 2 ways of being decoded, but "27", "28", and "29" only have 1 way.

Hence, there are a total of (6 \* 2) + (3 \* 1) = 12 + 3 = 15 ways to decode "2\*".

**Constraints:**

* 1 <= s.length <= 105
* s[i] is a digit or '\*'.

### Analysis:

For character ‘\*’, we iterate from 1-9 and carry result forward.

/// <summary>

/// Leet code #639. Decode Ways II

///

/// A message containing letters from A-Z is being encoded to numbers using

/// the following mapping way:

/// 'A' -> 1

/// 'B' -> 2

/// ...

/// 'Z' -> 26

/// Beyond that, now the encoded string can also contain the character '\*',

/// which can be treated as one of the numbers from 1 to 9.

/// Given the encoded message containing digits and the character '\*', return

/// the total number of ways to decode it.

/// Also, since the answer may be very large, you should return the output

/// mod 10^9 + 7.

///

/// Example 1:

/// Input: "\*"

/// Output: 9

/// Explanation: The encoded message can be decoded to the string: "A", "B",

/// "C", "D", "E", "F", "G", "H", "I".

///

/// Example 2:

/// Input: "1\*"

/// Output: 9 + 9 = 18

///

/// Note:

/// The length of the input string will fit in range [1, 10^5].

/// The input string will only contain the character '\*' and digits '0' - '9'.

/// </summary>

int LeetCode::numDecodingsII(string s)

{

if (s.empty()) return 0;

int M = 1000000007;

vector<int> dp(s.size());

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

{

if ((s[i] >= '1') && (s[i] <= '9'))

{

if (i == 0) dp[i] += 1;

else dp[i] = (dp[i] + dp[i - 1]) % M;

}

else if (s[i] == '\*')

{

if (i == 0) dp[i] += 9;

else dp[i] = (int)(((long long)dp[i] + 9 \* (long long)dp[i - 1]) % M);

}

if (i > 0)

{

if ((s[i - 1] == '1' && s[i] >= '0' && s[i] <= '9') ||

(s[i - 1] == '2' && s[i] >= '0' && s[i] <= '6'))

{

if (i == 1) dp[i] += 1;

else dp[i] = (dp[i] + dp[i - 2]) % M;

}

else if (s[i - 1] == '1' && s[i] == '\*')

{

if (i == 1) dp[i] += 9;

else dp[i] = (int)(((long long)dp[i] + 9 \* (long long)dp[i - 2]) % M);

}

else if (s[i - 1] == '2' && s[i] == '\*')

{

if (i == 1) dp[i] += 6;

else dp[i] = (int)(((long long)dp[i] + 6 \* (long long)dp[i - 2]) % M);

}

else if (s[i - 1] == '\*' && s[i] == '\*')

{

if (i == 1) dp[i] += 15;

else dp[i] = (int)(((long long)dp[i] + 15 \* (long long)dp[i - 2]) % M);

}

else if (s[i - 1] == '\*' && s[i] >= '0' && s[i] <= '6')

{

if (i == 1) dp[i] += 2;

else dp[i] = (int)(((long long)dp[i] + 2 \* (long long)dp[i - 2]) % M);

}

else if (s[i - 1] == '\*' && s[i] >= '7' && s[i] <= '9')

{

if (i == 1) dp[i] += 1;

else dp[i] = (int)(((long long)dp[i] + 1 \* (long long)dp[i - 2]) % M);

}

}

}

return dp[s.size() - 1];

}

## 1751. Maximum Number of Events That Can Be Attended II

Hard

You are given an array of events where events[i] = [startDayi, endDayi, valuei]. The ith event starts at startDayiand ends at endDayi, and if you attend this event, you will receive a value of valuei. You are also given an integer k which represents the maximum number of events you can attend.

You can only attend one event at a time. If you choose to attend an event, you must attend the **entire** event. Note that the end day is **inclusive**: that is, you cannot attend two events where one of them starts and the other ends on the same day.

Return *the****maximum sum****of values that you can receive by attending events.*

**Example 1:**

Chart, bar chart

Description automatically generated

**Input:** events = [[1,2,4],[3,4,3],[2,3,1]], k = 2

**Output:** 7

**Explanation:** Choose the green events, 0 and 1 (0-indexed) for a total value of 4 + 3 = 7.

**Example 2:**

Chart, bar chart

Description automatically generated

**Input:** events = [[1,2,4],[3,4,3],[2,3,10]], k = 2

**Output:** 10

**Explanation:** Choose event 2 for a total value of 10.

Notice that you cannot attend any other event as they overlap, and that you do **not** have to attend k events.

**Example 3:**

**Chart, waterfall chart

Description automatically generated**

**Input:** events = [[1,1,1],[2,2,2],[3,3,3],[4,4,4]], k = 3

**Output:** 9

**Explanation:** Although the events do not overlap, you can only attend 3 events. Pick the highest valued three.

**Constraints:**

* 1 <= k <= events.length
* 1 <= k \* events.length <= 106
* 1 <= startDayi <= endDayi <= 109
* 1 <= valuei <= 106

### Analysis:

We sort the events based on starting time, before processing every event, we harvest the accumulated event values happened before the event and when we process the event, we set event value based on the previous harvest value and current event at end time (if end time has conflict, we need to choose maximum), such end time event value can be harvested later again.

/// <summary>

/// Leet code 1751. Maximum Number of Events That Can Be Attended II

///

/// Hard

///

/// You are given an array of events where events[i] = [startDayi,

/// endDayi, valuei]. The ith event starts at startDayi and ends at

/// endDayi, and if you attend this event, you will receive a value of

/// valuei. You are also given an integer k which represents the

/// maximum number of events you can attend.

///

/// You can only attend one event at a time. If you choose to attend

/// an event, you must attend the entire event. Note that the end day

/// is inclusive: that is, you cannot attend two events where one of

/// them starts and the other ends on the same day.

///

/// Return the maximum sum of values that you can receive by attending

/// events.

///

/// Example 1:

/// Input: events = [[1,2,4],[3,4,3],[2,3,1]], k = 2

/// Output: 7

/// Explanation: Choose the green events, 0 and 1 (0-indexed) for a total

/// value of 4 + 3 = 7.

///

/// Example 2:

///

/// Input: events = [[1,2,4],[3,4,3],[2,3,10]], k = 2

/// Output: 10

/// Explanation: Choose event 2 for a total value of 10.

/// Notice that you cannot attend any other event as they overlap, and

/// that you do not have to attend k events.

///

/// Example 3:

/// Input: events = [[1,1,1],[2,2,2],[3,3,3],[4,4,4]], k = 3

/// Output: 9

/// Explanation: Although the events do not overlap, you can only attend

/// 3 events. Pick the highest valued three.

///

/// Constraints:

/// 1. 1 <= k <= events.length

/// 2. 1 <= k \* events.length <= 10^6

/// 3. 1 <= startDayi <= endDayi <= 10^9

/// 4. 1 <= valuei <= 10^6

/// </summary>

int LeetCodeDP::maxValue(vector<vector<int>>& events, int k)

{

map<int, vector<int>> event\_values;

sort(events.begin(), events.end());

vector<int> prev\_event = { 0 };

int result = 0;

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

{

int start = events[i][0];

int end = events[i][1];

int value = events[i][2];

// collect all previous event maximum values

while (!event\_values.empty() && event\_values.begin()->first < start)

{

auto itr = event\_values.begin();

for (size\_t j = 0; j < itr->second.size(); j++)

{

if (j >= prev\_event.size())

{

prev\_event.push\_back(itr->second[j]);

}

else

{

prev\_event[j] = max(prev\_event[j], itr->second[j]);

}

}

event\_values.erase(event\_values.begin());

}

if (event\_values.count(end) == 0)

{

event\_values[end] = { 0 };

}

for (int j = 0; (j < prev\_event.size() && j < k); j++)

{

if (j + 1 >= event\_values[end].size())

{

event\_values[end].push\_back(prev\_event[j] + value);

}

else

{

event\_values[end][j+1] = max(event\_values[end][j + 1], prev\_event[j] + value);

}

result = max(result, event\_values[end][j + 1]);

}

}

return result;

}

## 403. Frog Jump

Hard

A frog is crossing a river. The river is divided into some number of units, and at each unit, there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.

Given a list of stones' positions (in units) in sorted **ascending order**, determine if the frog can cross the river by landing on the last stone. Initially, the frog is on the first stone and assumes the first jump must be 1 unit.

If the frog's last jump was k units, its next jump must be either k - 1, k, or k + 1 units. The frog can only jump in the forward direction.

**Example 1:**

**Input:** stones = [0,1,3,5,6,8,12,17]

**Output:** true

**Explanation:** The frog can jump to the last stone by jumping 1 unit to the 2nd stone, then 2 units to the 3rd stone, then 2 units to the 4th stone, then 3 units to the 6th stone, 4 units to the 7th stone, and 5 units to the 8th stone.

**Example 2:**

**Input:** stones = [0,1,2,3,4,8,9,11]

**Output:** false

**Explanation:** There is no way to jump to the last stone as the gap between the 5th and 6th stone is too large.

**Constraints:**

* 2 <= stones.length <= 2000
* 0 <= stones[i] <= 231 - 1
* stones[0] == 0
* stones is sorted in a strictly increasing order.

### Analysis:

On every stone, we collect all possible steps jump to this stone, then, we store all future steps in target stone for future iterations.

/// <summary>

/// Leet code #403. Frog Jump

///

/// A frog is crossing a river. The river is divided into x units and at

/// each unit there may or may not exist a stone. The frog can jump on a

/// stone, but it must not jump into the water.

///

/// Given a list of stones' positions (in units) in sorted ascending order,

/// determine if the frog is able to cross the river by landing on the

/// last stone. Initially, the frog is on the first stone and assume the

/// first jump must be 1 unit.

///

/// If the frog's last jump was k units, then its next jump must be either

/// k - 1, k, or k + 1 units. Note that the frog can only jump in the

/// forward direction.

///

/// Note:

///

/// 1. The number of stones is ≥ 2 and is < 1,100.

/// 2. Each stone's position will be a non-negative integer < 231.

/// 3. The first stone's position is always 0.

///

/// Example 1:

///

/// [0,1,3,5,6,8,12,17]

///

/// There are a total of 8 stones.

/// The first stone at the 0th unit, second stone at the 1st unit,

/// third stone at the 3rd unit, and so on...

/// The last stone at the 17th unit.

///

/// Return true. The frog can jump to the last stone by jumping

/// 1 unit to the 2nd stone, then 2 units to the 3rd stone, then

/// 2 units to the 4th stone, then 3 units to the 6th stone,

/// 4 units to the 7th stone, and 5 units to the 8th stone.

///

/// Example 2:

///

/// [0,1,2,3,4,8,9,11]

///

/// Return false. There is no way to jump to the last stone as

/// the gap between the 5th and 6th stone is too large.

/// </summary>

bool LeetCodeDP::canCross(vector<int>& stones)

{

size\_t size = stones.size();

unordered\_map<int, unordered\_set<int>> dp;

// initalize

for (size\_t i = 0; i < stones.size(); i++) dp[stones[i]];

dp[0].insert(0);

// iterate every stone

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

{

for (auto s : dp[stones[i]])

{

for (int j = max(1, s - 1); j <= s + 1; j++)

{

int next = stones[i] + j;

if (next == stones.back()) return true;

if (dp.count(next) > 0)

{

dp[next].insert(j);

}

}

}

}

return false;

}