# Look Forward

The traditional DP will calculate result based on past result, so it is a look back pattern. However there are some situations that looking back is difficult, because you cannot easily determine the exact location what to look, unless you look at every past position, which will lead to at least O(N^2) solution.

In this case we want to push the result on every position to the related future position and use it there. We call this programming pattern as a push forward patten.

## 45. Jump Game II

Hard

Given an array of non-negative integers, 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.

**Example:**

**Input:** [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.

**Note:**

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

### Analysis:

We add all the positions we can visit in one jump into range and in next jump we only visited extended range (skip the positions we already visited in the last jump). This is something like a BFS search, and in some case we can use a queue, but here only two pointers which remember the extended position is good enough.

/// <summary>

/// Leet code #45. Jump Game II

///

/// Given an array of non-negative integers, 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.

///

/// Example:

///

/// Input: [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.

/// Note:

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

/// </summary>

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

{

if (nums.size() <= 1) return 0;

int first = 0;

int last = 0;

int next = last;

int result = 0;

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

{

result++;

for (int i = first; i < last + 1; i++)

{

next = max(next, nums[i] + i);

}

first = min(last + 1, next);

if (next <= last) return -1;

last = next;

}

return result;

}

## 403. Frog Jump

Hard

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:**

* The number of stones is ≥ 2 and is < 1,100.
* Each stone's position will be a non-negative integer < 231.
* 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.

### Analysis:

On every stone you pick up the steps arrive here and push the steps to the next stone.

/// <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;

}

## 313. Super Ugly Number

Medium

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 is a super ugly number for any given primes.
* The given numbers in primes are in ascending order.
* 0 < k ≤ 100, 0 < n ≤ 106, 0 < primes[i] < 1000.
* The nth super ugly number is guaranteed to fit in a 32-bit signed integer.

### Analysis:

We keep track on minimum factor of each prime5, and select the minimum product from them, then increase the corresponding index. If the product is larger than the selected result. We push it as next result.

/// <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();

}

## 514. Freedom Trail

Hard

In the video game Fallout 4, the quest "Road to Freedom" requires players to reach a metal dial called the "Freedom Trail Ring", and use the dial to spell a specific keyword in order to open the door.

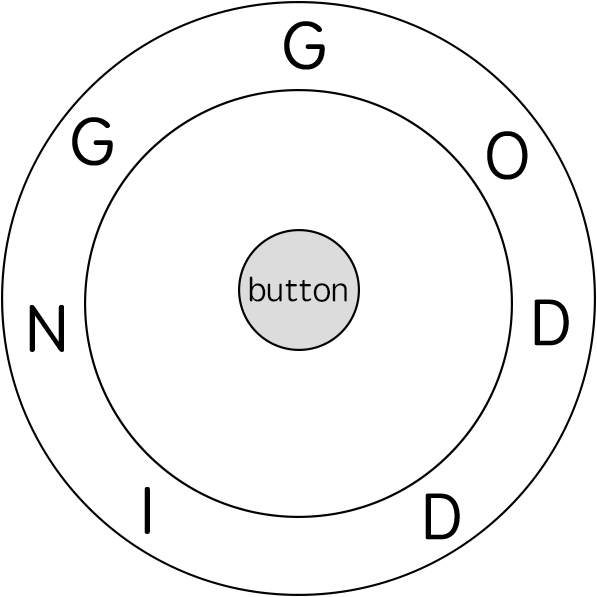
Given a string **ring**, which represents the code engraved on the outer ring and another string **key**, which represents the keyword needs to be spelled. You need to find the **minimum** number of steps in order to spell all the characters in the keyword.

Initially, the first character of the **ring** is aligned at 12:00 direction. You need to spell all the characters in the string **key** one by one by rotating the ring clockwise or anticlockwise to make each character of the string **key** aligned at 12:00 direction and then by pressing the center button.

At the stage of rotating the ring to spell the key character **key[i]**:

1. You can rotate the **ring** clockwise or anticlockwise **one place**, which counts as 1 step. The final purpose of the rotation is to align one of the string **ring's** characters at the 12:00 direction, where this character must equal to the character **key[i]**.
2. If the character **key[i]** has been aligned at the 12:00 direction, you need to press the center button to spell, which also counts as 1 step. After the pressing, you could begin to spell the next character in the key (next stage), otherwise, you've finished all the spelling.

**Example:**



**Input:** ring = "godding", key = "gd"

**Output:** 4

**Explanation:**

For the first key character 'g', since it is already in place, we just need 1 step to spell this character.

For the second key character 'd', we need to rotate the ring "godding" anticlockwise by two steps to make it become "ddinggo".

Also, we need 1 more step for spelling.

So the final output is 4.

**Note:**

1. Length of both ring and **key** will be in range 1 to 100.
2. There are only lowercase letters in both strings and might be some duplcate characters in both strings.
3. It's guaranteed that string **key** could always be spelled by rotating the string **ring**.

### Analysis:

We keep track on minimum steps on each iteration with specified character. The result is carried forward. Please watch how to calculate the circular distance of two position (using circle size to round up.)

/// <summary>

/// Leet code #514. Freedom Trail

///

/// In the video game Fallout 4, the quest "Road to Freedom" requires

/// players to reach a metal dial called the "Freedom Trail Ring", and

/// use the dial to spell a specific keyword in order to open the door.

///

/// Given a string ring, which represents the code engraved on the outer

/// ring and another string key, which represents the keyword needs to be

/// spelled. You need to find the minimum number of steps in order to

/// spell all the characters in the keyword.

///

/// Initially, the first character of the ring is aligned at 12:00

/// direction. You need to spell all the characters in the string key one

/// by one by rotating the ring clockwise or anticlockwise to make each

/// character of the string key aligned at 12:00 direction and then by

/// pressing the center button.

///

/// At the stage of rotating the ring to spell the key character key[i]:

///

/// You can rotate the ring clockwise or anticlockwise one place, which

/// counts as 1 step. The final purpose of the rotation is to align one

/// of the string ring's characters at the 12:00 direction, where this

/// character must equal to the character key[i].

/// If the character key[i] has been aligned at the 12:00 direction, you

/// need to press the center button to spell, which also counts as 1 step.

/// After the pressing, you could begin to spell the next character in the

/// key (next stage), otherwise, you've finished all the spelling.

///

/// Example:

/// Input: ring = "godding", key = "gd"

/// Output: 4

/// Explanation:

/// For the first key character 'g', since it is already in place, we just

/// need 1 step to spell this character.

/// For the second key character 'd', we need to rotate the ring "godding"

/// anticlockwise by two steps to make it become "ddinggo".

/// Also, we need 1 more step for spelling.

/// So the final output is 4.

///

/// Note:

/// 1. Length of both ring and key will be in range 1 to 100.

/// 2. There are only lowercase letters in both strings and might be some

/// duplcate characters in both strings.

/// 3. It's guaranteed that string key could always be spelled by rotating

/// the string ring.

/// </summary>

int LeetCodeDP::findRotateSteps(string ring, string key)

{

vector<pair<int, int>> last\_pos\_list = { { 0, 0 } };

unordered\_map<char, vector<pair<int, int>>> char\_map;

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

{

char\_map[ring[i]].push\_back(make\_pair(i, 0));

}

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

{

for (size\_t j = 0; j < char\_map[key[i]].size(); j++)

{

int min\_step = INT\_MAX;

int curr\_pos = char\_map[key[i]][j].first;

for (pair<int, int> last\_pos : last\_pos\_list)

{

min\_step = min(min\_step,

last\_pos.second +

abs(curr\_pos - last\_pos.first));

min\_step = min(min\_step, last\_pos.second + (int)ring.size() -

abs(curr\_pos - last\_pos.first));

}

char\_map[key[i]][j].second = min\_step + 1;

}

last\_pos\_list = char\_map[key[i]];

}

int result = INT\_MAX;

for (size\_t i = 0; i < last\_pos\_list.size(); i++)

{

result = min(result, last\_pos\_list[i].second);

}

return result;

}

## 656. Coin Path

Hard

Given an array A (index starts at 1) consisting of N integers: A1, A2, ..., AN and an integer B. The integer B denotes that from any place (suppose the index is i) in the array A, you can jump to any one of the place in the array A indexed i+1, i+2, …, i+B if this place can be jumped to. Also, if you step on the index i, you have to pay Ai coins. If Ai is -1, it means you can’t jump to the place indexed i in the array.

Now, you start from the place indexed 1 in the array A, and your aim is to reach the place indexed N using the minimum coins. You need to return the path of indexes (starting from 1 to N) in the array you should take to get to the place indexed N using minimum coins.

If there are multiple paths with the same cost, return the lexicographically smallest such path.

If it's not possible to reach the place indexed N then you need to return an empty array.

**Example 1:**

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

**Output:** [1,3,5]

**Example 2:**

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

**Output:** []

**Note:**

1. Path Pa1, Pa2, ..., Pan is lexicographically smaller than Pb1, Pb2, ..., Pbm, if and only if at the first i where Pai and Pbi differ, Pai < Pbi; when no such i exists, then n < m.
2. A1 >= 0. A2, ..., AN (if exist) will in the range of [-1, 100].
3. Length of A is in the range of [1, 1000].
4. B is in the range of [1, 100].

### Analysis:

Carry forward the path and the minimum cost on each position.

/// <summary>

/// Leet code #656. Coin Path

///

/// Given an array A (index starts at 1) consisting of N integers: A1, A2,

/// ..., AN and an integer B. The integer B denotes that from any place

/// (suppose the index is i) in the array A, you can jump to any one of

/// the place in the array A indexed i+1, i+2, …, i+B if this place can

/// be jumped to. Also, if you step on the index i, you have to pay Ai

/// coins. If Ai is -1, it means you can’t jump to the place indexed i in

/// the array.

///

/// Now, you start from the place indexed 1 in the array A, and your aim

/// is to reach the place indexed N using the minimum coins. You need to

/// return the path of indexes (starting from 1 to N) in the array you

/// should take to get to the place indexed N using minimum coins.

///

/// If there are multiple paths with the same cost, return the

/// lexicographically smallest such path.

///

/// If it's not possible to reach the place indexed N then you need to

/// return an empty array.

///

/// Example 1:

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

/// Output: [1,3,5]

///

/// Example 2:

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

/// Output: []

/// Note:

/// Path Pa1, Pa2, ..., Pan is lexicographically smaller than Pb1, Pb2,

/// ..., Pbm, if and only if at the first i where Pai and Pbi differ,

/// Pai < Pbi; when no such i exists, then n < m.

/// A1 >= 0. A2, ..., AN (if exist) will in the range of [-1, 100].

/// Length of A is in the range of [1, 1000].

/// B is in the range of [1, 100].

/// </summary>

vector<int> LeetCodeDP::cheapestJump(vector<int>& A, int B)

{

vector<int> result;

vector<pair<vector<int>, int>> dp(A.size());

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

{

if (A[i] == -1) continue;

if (i == 0)

{

dp[i].first.push\_back(1);

dp[i].second = A[i];

}

else

{

if (dp[i].first.empty()) continue;

}

for (size\_t j = 1; (j <= (size\_t)B) && (i + j < A.size()); j++)

{

if (A[i + j] == -1) continue;

vector<int> new\_path = dp[i].first;

new\_path.push\_back(i + j + 1);

if ((dp[i + j].first.empty()) ||

(dp[i + j].second > dp[i].second + A[i + j]) ||

((dp[i + j].second == dp[i].second + A[i + j]) &&

(dp[i + j].first > new\_path)))

{

dp[i + j] = make\_pair(new\_path, dp[i].second + A[i + j]);

}

}

}

if (A.empty()) return result;

result = dp[A.size() - 1].first;

return result;

}