LeetCode 題解

靈魂機器 (soulmachine@gmail.com)

https://github.com/soulmachine/leetcode

最後更新 2020-5-15

版權聲明

內容簡介

本書的目標讀者是準備去北美找工作的碼農，也適用於在國內找工作的碼農，以及剛接觸 ACM 算法競賽的新手。

本書包含了 LeetCode Online Judge(http://leetcode.com/onlinejudge) 所有題目的答案，所有代碼經過精心編寫，編碼規範良好，適合讀者反覆揣摩，模仿，甚至在紙上默寫。

全書的代碼，使用 C++ 11 的編寫，並在 LeetCode Online Judge 上測試通過。本書中的代碼規範，跟在公司中的工程規範略有不同，為了使代碼短（方便迅速實現）：

- 所有代碼都是單一文件。這是因為一般 OJ 網站，提交代碼的時候只有一個文本框，如果還是按照標準做法，比如分為頭文件.h 和源代碼.cpp，無法在網站上提交；
- Shorter is better。能遞歸則一定不用棧；能用 STL 則一定不自己實現。
- 不提倡防禦式編程。不需要檢查 malloc()/new 返回的指針是否為 nullptr；不需要檢查內部函數入口參數的有效性。

本手冊假定讀者已經學過《數據結構》[①]，《算法》[②] 這兩門課，熟練掌握 C++或 Java。

GitHub 地址

本書是開源的，GitHub 地址：https://github.com/soulmachine/leetcode

北美求職微博羣

我和我的小夥伴們在這裏：http://q.weibo.com/1312378

[①]《數據結構》，嚴蔚敏等著，清華大學出版社，http://book.douban.com/subject/2024655/

[②]《Algorithms》，Robert Sedgewick, Addison-Wesley Professional, http://book.douban.com/subject/4854123/

目录

第 1 章
編程技巧

在判斷兩個浮點數 a 和 b 是否相等時，不要用 `a==b`，應該判斷二者之差的絕對值 `fabs(a-b)` 是否小於某個閾值，例如 `1e-9`。

判斷一個整數是否是為奇數，用 `x % 2 != 0`，不要用 `x % 2 == 1`，因為 x 可能是負數。

用 `char` 的值作為數組下標（例如，統計字符串中每個字符出現的次數），要考慮到 `char` 可能是負數。有的人考慮到了，先強制轉型為 `unsigned int` 再用作下標，這仍然是錯的。正確的做法是，先強制轉型為 `unsigned char`，再用作下標。這涉及 C++整型提升的規則，就不詳述了。

以下是關於 STL 使用技巧的，很多條款來自《Effective STL》這本書。

vector 和 string 優先於動態分配的數組

首先，在性能上，由於 `vector` 能夠保證連續內存，因此一旦分配了後，它的性能跟原始數組相當；

其次，如果用 new，意味着你要確保後面進行了 delete，一旦忘記了，就會出現 BUG，且這樣需要都寫一行 delete，代碼不夠短；

再次，聲明多維數組的話，只能一個一個 new，例如：

```
int** ary = new int*[row_num];
for(int i = 0; i < row_num; ++i)
    ary[i] = new int[col_num];
```

用 vector 的話一行代碼搞定，

```
vector<vector<int> > ary(row_num, vector<int>(col_num, 0));
```

使用 reserve 來避免不必要的重新分配

第 2 章
線性表

這類題目考察線性表的操作，例如，數組，單鏈表，雙向鏈表等。

2.1 數組

2.1.1 Remove Duplicates from Sorted Array

描述

Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

For example, Given input array `A = [1,1,2]`,

Your function should return length = 2, and A is now

$$1, 2$$

.

分析

無

代碼 1

```
// LeetCode, Remove Duplicates from Sorted Array
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if (nums.empty()) return 0;

        int index = 0;
        for (int i = 1; i < nums.size(); i++) {
            if (nums[index] != nums[i])
                nums[++index] = nums[i];
```

```
        }
        return index + 1;
    }
};
```

代碼 2

```
// LeetCode, Remove Duplicates from Sorted Array
// 使用 STL, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        return distance(nums.begin(), unique(nums.begin(), nums.end()));
    }
};
```

代碼 3

```
// LeetCode, Remove Duplicates from Sorted Array
// 使用 STL, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        return distance(nums.begin(), removeDuplicates(nums.begin(), nums.end(), nums.begin()))
    }

    template<typename InIt, typename OutIt>
    OutIt removeDuplicates(InIt first, InIt last, OutIt output) {
        while (first != last) {
            *output++ = *first;
            first = upper_bound(first, last, *first);
        }

        return output;
    }
};
```

相關題目

- Remove Duplicates from Sorted Array II，見 §2.1.2

2.1.2 Remove Duplicates from Sorted Array II

描述

Follow up for ”Remove Duplicates”: What if duplicates are allowed at most twice?

For example, Given sorted array `A = [1,1,1,2,2,3]`,

Your function should return length = 5, and A is now

$$1, 1, 2, 2, 3$$

分析

加一個變量記錄一下元素出現的次數即可。這題因為是已經排序的數組，所以一個變量即可解決。如果是沒有排序的數組，則需要引入一個 hashmap 來記錄出現次數。

代碼 1

```
// LeetCode, Remove Duplicates from Sorted Array II
// 時間複雜度 O(n), 空間複雜度 O(1)
// @author hex108 (https://github.com/hex108)
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if (nums.size() <= 2) return nums.size();

        int index = 2;
        for (int i = 2; i < nums.size(); i++){
            if (nums[i] != nums[index - 2])
                nums[index++] = nums[i];
        }

        return index;
    }
};
```

代碼 2

下面是一個更簡潔的版本。上面的代碼略長, 不過擴展性好一些, 例如將 `occur < 2` 改為 `occur < 3`，就變成了允許重複最多 3 次。

```
// LeetCode, Remove Duplicates from Sorted Array II
// @author 虞航仲 (http://weibo.com/u/1666779725)
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        const int n = nums.size();
        int index = 0;
        for (int i = 0; i < n; ++i) {
            if (i > 0 && i < n - 1 && nums[i] == nums[i - 1] && nums[i] == nums[i + 1])
                continue;

            nums[index++] = nums[i];
        }
```

```
        return index;
    }
};
```

相關題目

- Remove Duplicates from Sorted Array，見 §2.1.1

2.1.3 Search in Rotated Sorted Array

描述

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

分析

二分查找，難度主要在於左右邊界的確定。

代碼

```
// LeetCode, Search in Rotated Sorted Array
// 時間複雜度 O(log n)，空間複雜度 O(1)
class Solution {
public:
    int search(const vector<int>& nums, int target) {
        int first = 0, last = nums.size();
        while (first != last) {
            const int mid = first  + (last - first) / 2;
            if (nums[mid] == target)
                return mid;
            if (nums[first] <= nums[mid]) {
                if (nums[first] <= target && target < nums[mid])
                    last = mid;
                else
                    first = mid + 1;
            } else {
                if (nums[mid] < target && target <= nums[last-1])
                    first = mid + 1;
                else
                    last = mid;
            }
        }
        return -1;
    }
};
```

相關題目

- Search in Rotated Sorted Array II，見 §2.1.4

2.1.4 Search in Rotated Sorted Array II

描述

Follow up for "Search in Rotated Sorted Array": What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

分析

允許重複元素，則上一題中如果 `A[m]>=A[l]`，那麼 `[l,m]` 為遞增序列的假設就不能成立了，比如

$$1, 3, 1, 1, 1$$

。

如果 `A[m]>=A[l]` 不能確定遞增，那就把它拆分成兩個條件：

- 若 `A[m]>A[l]`，則區間 `[l,m]` 一定遞增
- 若 `A[m]==A[l]` 確定不了，那就 `l++`，往下看一步即可。

代碼

```
// LeetCode, Search in Rotated Sorted Array II
// 時間複雜度 O(n)，空間複雜度 O(1)
class Solution {
public:
    bool search(const vector<int>& nums, int target) {
        int first = 0, last = nums.size();
        while (first != last) {
            const int mid = first  + (last - first) / 2;
            if (nums[mid] == target)
                return true;
            if (nums[first] < nums[mid]) {
                if (nums[first] <= target && target < nums[mid])
                    last = mid;
                else
                    first = mid + 1;
            } else if (nums[first] > nums[mid]) {
                if (nums[mid] < target && target <= nums[last-1])
                    first = mid + 1;
                else
                    last = mid;
            } else
                //skip duplicate one
```

```
                first++;
            }
            return false;
        }
    };
```

相關題目

- Search in Rotated Sorted Array，見 §2.1.3

2.1.5 Median of Two Sorted Arrays

描述

There are two sorted arrays A and B of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be $O(\log(m+n))$.

分析

這是一道非常經典的題。這題更通用的形式是，給定兩個已經排序好的數組，找到兩者所有元素中第 k 大的元素。

$O(m+n)$ 的解法比較直觀，直接 merge 兩個數組，然後求第 k 大的元素。

不過我們僅僅需要第 k 大的元素，是不需要“排序”這麼昂貴的操作的。可以用一個計數器，記錄當前已經找到第 m 大的元素了。同時我們使用兩個指針 `pA` 和 `pB`，分別指向 A 和 B 數組的第一個元素，使用類似於 merge sort 的原理，如果數組 A 當前元素小，那麼 `pA++`，同時 `m++`；如果數組 B 當前元素小，那麼 `pB++`，同時 `m++`。最終當 m 等於 k 的時候，就得到了我們的答案，$O(k)$ 時間，$O(1)$ 空間。但是，當 k 很接近 $m+n$ 的時候，這個方法還是 $O(m+n)$ 的。

有沒有更好的方案呢？我們可以考慮從 k 入手。如果我們每次都能夠刪除一個一定在第 k 大元素之前的元素，那麼我們需要進行 k 次。但是如果每次我們都刪除一半呢？由於 A 和 B 都是有序的，我們應該充分利用這裏面的信息，類似於二分查找，也是充分利用了“有序”。

假設 A 和 B 的元素個數都大於 $k/2$，我們將 A 的第 $k/2$ 個元素（即 `A[k/2-1]`）和 B 的第 $k/2$ 個元素（即 `B[k/2-1]`）進行比較，有以下三種情況（為了簡化這裏先假設 k 為偶數，所得到的結論對於 k 是奇數也是成立的）：

- `A[k/2-1] == B[k/2-1]`
- `A[k/2-1] > B[k/2-1]`
- `A[k/2-1] < B[k/2-1]`

如果 `A[k/2-1] < B[k/2-1]`，意味着 `A[0]` 到 `A[k/2-1]` 的肯定在 $A \cup B$ 的 top k 元素的範圍內，換句話說，`A[k/2-1]` 不可能大於 $A \cup B$ 的第 k 大元素。留給讀者證明。

因此，我們可以放心的刪除 A 數組的這 $k/2$ 個元素。同理，當 `A[k/2-1] > B[k/2-1]` 時，可以刪除 B 數組的 $k/2$ 個元素。

當 A[k/2-1] == B[k/2-1] 時,說明找到了第 k 大的元素,直接返回 A[k/2-1] 或 B[k/2-1] 即可。因此，我們可以寫一個遞歸函數。那麼函數什麼時候應該終止呢？

- 當 A 或 B 是空時，直接返回 B[k-1] 或 A[k-1]；
- 當 k=1 是，返回 min(A[0], B[0])；
- 當 A[k/2-1] == B[k/2-1] 時，返回 A[k/2-1] 或 B[k/2-1]

代碼

```
// LeetCode, Median of Two Sorted Arrays
// 時間複雜度 O(log(m+n))，空間複雜度 O(log(m+n))
class Solution {
public:
    double findMedianSortedArrays(const vector<int>& A, const vector<int>& B) {
        const int m = A.size();
        const int n = B.size();
        int total = m + n;
        if (total & 0x1)
            return find_kth(A.begin(), m, B.begin(), n, total / 2 + 1);
        else
            return (find_kth(A.begin(), m, B.begin(), n, total / 2)
                    + find_kth(A.begin(), m, B.begin(), n, total / 2 + 1)) / 2.0;
    }
private:
    static int find_kth(std::vector<int>::const_iterator A, int m,
            std::vector<int>::const_iterator B, int n, int k) {
        //always assume that m is equal or smaller than n
        if (m > n) return find_kth(B, n, A, m, k);
        if (m == 0) return *(B + k - 1);
        if (k == 1) return min(*A, *B);

        //divide k into two parts
        int ia = min(k / 2, m), ib = k - ia;
        if (*(A + ia - 1) < *(B + ib - 1))
            return find_kth(A + ia, m - ia, B, n, k - ia);
        else if (*(A + ia - 1) > *(B + ib - 1))
            return find_kth(A, m, B + ib, n - ib, k - ib);
        else
            return A[ia - 1];
    }
};
```

相關題目

- 無

2.1.6 Longest Consecutive Sequence

描述

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example, Given

$$100, 4, 200, 1, 3, 2$$

, The longest consecutive elements sequence is

$$1, 2, 3, 4$$

. Return its length: 4.

Your algorithm should run in $O(n)$ complexity.

分析

如果允許 $O(n \log n)$ 的複雜度，那麼可以先排序，可是本題要求 $O(n)$。

由於序列裏的元素是無序的，又要求 $O(n)$，首先要想到用哈希表。

用一個哈希表 `unordered_map<int, bool> used` 記錄每個元素是否使用，對每個元素，以該元素為中心，往左右擴張，直到不連續為止，記錄下最長的長度。

代碼

```
// Leet Code, Longest Consecutive Sequence
// 時間複雜度 O(n)，空間複雜度 O(n)
class Solution {
public:
    int longestConsecutive(const vector<int> &nums) {
        unordered_map<int, bool> used;

        for (auto i : nums) used[i] = false;

        int longest = 0;

        for (auto i : nums) {
            if (used[i]) continue;

            int length = 1;

            used[i] = true;

            for (int j = i + 1; used.find(j) != used.end(); ++j) {
                used[j] = true;
                ++length;
            }

            for (int j = i - 1; used.find(j) != used.end(); --j) {
```

```
                used[j] = true;
                ++length;
            }

            longest = max(longest, length);
        }

        return longest;
    }
};
```

分析 2

第一直覺是個聚類的操作, 應該有 union,find 的操作. 連續序列可以用兩端和長度來表示. 本來用兩端就可以表示, 但考慮到查詢的需求, 將兩端分別暴露出來. 用 `unordered_map<int, int> map` 來存儲. 原始思路來自於 `http://discuss.leetcode.com/questions/1070/longest-consecutive-sequence`

代碼

```
// Leet Code, Longest Consecutive Sequence
// 時間複雜度 O(n), 空間複雜度 O(n)
// Author: @advancedxy
class Solution {
public:
    int longestConsecutive(vector<int> &nums) {
        unordered_map<int, int> map;
        int size = nums.size();
        int l = 1;
        for (int i = 0; i < size; i++) {
            if (map.find(nums[i]) != map.end()) continue;
            map[nums[i]] = 1;
            if (map.find(nums[i] - 1) != map.end()) {
                l = max(l, mergeCluster(map, nums[i] - 1, nums[i]));
            }
            if (map.find(nums[i] + 1) != map.end()) {
                l = max(l, mergeCluster(map, nums[i], nums[i] + 1));
            }
        }
        return size == 0 ? 0 : l;
    }

private:
    int mergeCluster(unordered_map<int, int> &map, int left, int right) {
        int upper = right + map[right] - 1;
        int lower = left - map[left] + 1;
        int length = upper - lower + 1;
        map[upper] = length;
        map[lower] = length;
        return length;
```

```
    }
};
```

相關題目

- 無

2.1.7 Two Sum

描述

Given an array of integers, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

You may assume that each input would have exactly one solution.

Input: `numbers={2, 7, 11, 15}, target=9`

Output: `index1=1, index2=2`

分析

方法 1： 暴力，複雜度 $O(n^2)$，會超時

方法 2： hash。用一個哈希表，存儲每個數對應的下標，複雜度 $O(n)$.

方法 3：先排序，然後左右夾逼，排序 $O(n \log n)$，左右夾逼 $O(n)$，最終 $O(n \log n)$。但是注意，這題需要返回的是下標，而不是數字本身，因此這個方法行不通。

代碼

```
//LeetCode, Two Sum
// 方法 2 ：hash。用一個哈希表，存儲每個數對應的下標
// 時間複雜度 O(n)，空間複雜度 O(n)
class Solution {
public:
    vector<int> twoSum(vector<int> &nums, int target) {
        unordered_map<int, int> mapping;
        vector<int> result;
        for (int i = 0; i < nums.size(); i++) {
            mapping[nums[i]] = i;
        }
        for (int i = 0; i < nums.size(); i++) {
            const int gap = target - nums[i];
            if (mapping.find(gap) != mapping.end() && mapping[gap] > i) {
                result.push_back(i + 1);
                result.push_back(mapping[gap] + 1);
                break;
```

```
            }
        }
        return result;
    }
};
```

相關題目

- 3Sum, 見 §2.1.8
- 3Sum Closest, 見 §2.1.9
- 4Sum, 見 §2.1.10

2.1.8 3Sum

描述

Given an array S of n integers, are there elements a, b, c in S such that $a + b + c = 0$? Find all unique triplets in the array which gives the sum of zero.

Note:

- Elements in a triplet (a, b, c) must be in non-descending order. (ie, $a \leq b \leq c$)
- The solution set must not contain duplicate triplets.

For example, given array `S = {-1 0 1 2 -1 -4}`.

A solution set is:

```
(-1, 0, 1)
(-1, -1, 2)
```

分析

先排序，然後左右夾逼，複雜度 $O(n^2)$。

這個方法可以推廣到 k-sum，先排序，然後做 $k-2$ 次循環，在最內層循環左右夾逼，時間複雜度是 $O(\max\{n \log n, n^{k-1}\})$。

代碼

```
// LeetCode, 3Sum
// 先排序，然後左右夾逼，注意跳過重複的數，時間複雜度 O(n^2)，空間複雜度 O(1)
class Solution {
    public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> result;
        if (nums.size() < 3) return result;
        sort(nums.begin(), nums.end());
        const int target = 0;
```

```
        auto last = nums.end();
        for (auto i = nums.begin(); i < last-2; ++i) {
            auto j = i+1;
            if (i > nums.begin() && *i == *(i-1)) continue;
            auto k = last-1;
            while (j < k) {
                if (*i + *j + *k < target) {
                    ++j;
                    while(*j == *(j - 1) && j < k) ++j;
                } else if (*i + *j + *k > target) {
                    --k;
                    while(*k == *(k + 1) && j < k) --k;
                } else {
                    result.push_back({ *i, *j, *k });
                    ++j;
                    --k;
                    while(*j == *(j - 1) && *k == *(k + 1) && j < k) ++j;
                }
            }
        }
        return result;
    }
};
```

相關題目

- Two sum, 見 §2.1.7
- 3Sum Closest, 見 §2.1.9
- 4Sum, 見 §2.1.10

2.1.9 3Sum Closest

描述

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

For example, given array `S = {-1 2 1 -4}`, and `target = 1`.

The sum that is closest to the target is 2. (`-1 + 2 + 1 = 2`).

分析

先排序，然後左右夾逼，複雜度 $O(n^2)$。

代碼

```
// LeetCode, 3Sum Closest
// 先排序，然後左右夾逼，時間複雜度 O(n^2)，空間複雜度 O(1)
```

```
class Solution {
public:
    int threeSumClosest(vector<int>& nums, int target) {
        int result = 0;
        int min_gap = INT_MAX;

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

        for (auto a = nums.begin(); a != prev(nums.end(), 2); ++a) {
            auto b = next(a);
            auto c = prev(nums.end());

            while (b < c) {
                const int sum = *a + *b + *c;
                const int gap = abs(sum - target);

                if (gap < min_gap) {
                    result = sum;
                    min_gap = gap;
                }

                if (sum < target) ++b;
                else              --c;
            }
        }

        return result;
    }
};
```

相關題目

- Two sum, 見 §2.1.7
- 3Sum, 見 §2.1.8
- 4Sum, 見 §2.1.10

2.1.10 4Sum

描述

Given an array S of n integers, are there elements a, b, c, and d in S such that $a + b + c + d = target$? Find all unique quadruplets in the array which gives the sum of target.

Note:

- Elements in a quadruplet (a, b, c, d) must be in non-descending order. (ie, $a \leq b \leq c \leq d$)
- The solution set must not contain duplicate quadruplets.

For example, given array `S = {1 0 -1 0 -2 2}`, and `target = 0`.

A solution set is:

```
(-1,  0, 0, 1)
(-2, -1, 1, 2)
(-2,  0, 0, 2)
```

分析

先排序，然後左右夾逼，複雜度 $O(n^3)$，會超時。

可以用一個 hashmap 先緩存兩個數的和，最終複雜度 $O(n^3)$。這個策略也適用於 3Sum 。

左右夾逼

```
// LeetCode, 4Sum
// 先排序，然後左右夾逼，時間複雜度 O(n^3)，空間複雜度 O(1)
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        vector<vector<int>> result;
        if (nums.size() < 4) return result;
        sort(nums.begin(), nums.end());

        auto last = nums.end();
        for (auto a = nums.begin(); a < prev(last, 3); ++a) {
            for (auto b = next(a); b < prev(last, 2); ++b) {
                auto c = next(b);
                auto d = prev(last);
                while (c < d) {
                    if (*a + *b + *c + *d < target) {
                        ++c;
                    } else if (*a + *b + *c + *d > target) {
                        --d;
                    } else {
                        result.push_back({ *a, *b, *c, *d });
                        ++c;
                        --d;
                    }
                }
            }
        }
        sort(result.begin(), result.end());
        result.erase(unique(result.begin(), result.end()), result.end());
        return result;
    }
};
```

map 做緩存

```
// LeetCode, 4Sum
// 用一個 hashmap 先緩存兩個數的和
// 時間複雜度，平均 O(n^2)，最壞 O(n^4)，空間複雜度 O(n^2)
class Solution {
public:
```

```
    vector<vector<int> > fourSum(vector<int> &nums, int target) {
        vector<vector<int>> result;
        if (nums.size() < 4) return result;
        sort(nums.begin(), nums.end());

        unordered_map<int, vector<pair<int, int> > > cache;
        for (size_t a = 0; a < nums.size(); ++a) {
            for (size_t b = a + 1; b < nums.size(); ++b) {
                cache[nums[a] + nums[b]].push_back(pair<int, int>(a, b));
            }
        }

        for (int c = 0; c < nums.size(); ++c) {
            for (size_t d = c + 1; d < nums.size(); ++d) {
                const int key = target - nums[c] - nums[d];
                if (cache.find(key) == cache.end()) continue;

                const auto& vec = cache[key];
                for (size_t k = 0; k < vec.size(); ++k) {
                    if (c <= vec[k].second)
                        continue; // 有重疊

                    result.push_back( { nums[vec[k].first],
                            nums[vec[k].second], nums[c], nums[d] });
                }
            }
        }
        sort(result.begin(), result.end());
        result.erase(unique(result.begin(), result.end()), result.end());
        return result;
    }
};
```

multimap

```
// LeetCode, 4Sum
// 用一個 hashmap 先緩存兩個數的和
// 時間複雜度 O(n^2), 空間複雜度 O(n^2)
// @author 龔陸安 (http://weibo.com/luangong)
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        vector<vector<int>> result;
        if (nums.size() < 4) return result;
        sort(nums.begin(), nums.end());

        unordered_multimap<int, pair<int, int>> cache;
        for (int i = 0; i + 1 < nums.size(); ++i)
            for (int j = i + 1; j < nums.size(); ++j)
                cache.insert(make_pair(nums[i] + nums[j], make_pair(i, j)));

        for (auto i = cache.begin(); i != cache.end(); ++i) {
```

```
            int x = target - i->first;
            auto range = cache.equal_range(x);
            for (auto j = range.first; j != range.second; ++j) {
                auto a = i->second.first;
                auto b = i->second.second;
                auto c = j->second.first;
                auto d = j->second.second;
                if (a != c && a != d && b != c && b != d) {
                    vector<int> vec = { nums[a], nums[b], nums[c], nums[d] };
                    sort(vec.begin(), vec.end());
                    result.push_back(vec);
                }
            }
        }
        sort(result.begin(), result.end());
        result.erase(unique(result.begin(), result.end()), result.end());
        return result;
    }
};
```

方法 4

```
// LeetCode, 4Sum
// 先排序, 然後左右夾逼, 時間複雜度 O(n^3logn), 空間複雜度 O(1), 會超時
// 跟方法 1 相比, 表面上優化了, 實際上更慢了, 切記!
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        vector<vector<int>> result;
        if (nums.size() < 4) return result;
        sort(nums.begin(), nums.end());

        auto last = nums.end();
        for (auto a = nums.begin(); a < prev(last, 3);
                a = upper_bound(a, prev(last, 3), *a)) {
            for (auto b = next(a); b < prev(last, 2);
                    b = upper_bound(b, prev(last, 2), *b)) {
                auto c = next(b);
                auto d = prev(last);
                while (c < d) {
                    if (*a + *b + *c + *d < target) {
                        c = upper_bound(c, d, *c);
                    } else if (*a + *b + *c + *d > target) {
                        d = prev(lower_bound(c, d, *d));
                    } else {
                        result.push_back({ *a, *b, *c, *d });
                        c = upper_bound(c, d, *c);
                        d = prev(lower_bound(c, d, *d));
                    }
                }
            }
        }
```

```
        return result;
    }
};
```

相關題目

- Two sum, 見 §2.1.7
- 3Sum, 見 §2.1.8
- 3Sum Closest, 見 §2.1.9

2.1.11 Remove Element

描述

Given an array and a value, remove all instances of that value in place and return the new length.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

分析

無

代碼 1

```
// LeetCode, Remove Element
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int removeElement(vector<int>& nums, int target) {
        int index = 0;
        for (int i = 0; i < nums.size(); ++i) {
            if (nums[i] != target) {
                nums[index++] = nums[i];
            }
        }
        return index;
    }
};
```

代碼 2

```
// LeetCode, Remove Element
// 使用 remove(), 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int removeElement(vector<int>& nums, int target) {
        return distance(nums.begin(), remove(nums.begin(), nums.end(), target));
    }
};
```

相關題目

- 無

2.1.12 Next Permutation

描述

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

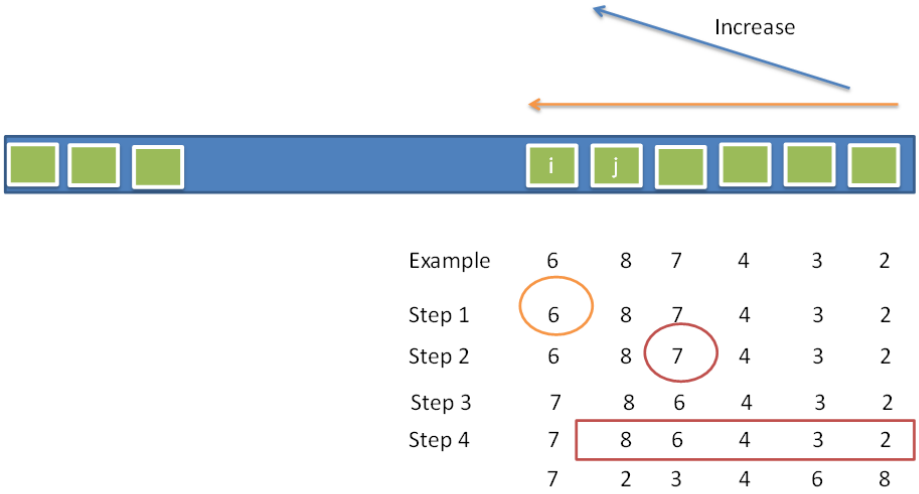
The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

```
1,2,3 → 1,3,2
3,2,1 → 1,2,3
1,1,5 → 1,5,1
```

分析

算法過程如圖 2-1 所示（來自 http://fisherlei.blogspot.com/2012/12/leetcode-next-permutation.html）。



1. From right to left, find the first digit (PartitionNumber)which violate the increase trend, in this example, 6 will be selected since 8,7,4,3,2 already in a increase trend.
2. From right to left, find the first digit which large than PartitionNumber, call it changeNumber. Here the 7 will be selected.
3. Swap the PartitionNumber and ChangeNumber.
4. Reverse all the digit on the right of partition index.

图 2-1 下一個排列算法流程

代碼

```
// LeetCode, Next Permutation
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    void nextPermutation(vector<int> &nums) {
        next_permutation(nums.begin(), nums.end());
    }

    template<typename BidiIt>
    bool next_permutation(BidiIt first, BidiIt last) {
        // Get a reversed range to simplify reversed traversal.
        const auto rfirst = reverse_iterator<BidiIt>(last);
        const auto rlast = reverse_iterator<BidiIt>(first);

        // Begin from the second last element to the first element.
        auto pivot = next(rfirst);

        // Find `pivot`, which is the first element that is no less than its
        // successor.  `Prev` is used since `pivort` is a `reversed_iterator`.
        while (pivot != rlast && *pivot >= *prev(pivot))
```

```
            ++pivot;

        // No such elemenet found, current sequence is already the largest
        // permutation, then rearrange to the first permutation and return false.
        if (pivot == rlast) {
            reverse(rfirst, rlast);
            return false;
        }

        // Scan from right to left, find the first element that is greater than
        // `pivot`.
        auto change = find_if(rfirst, pivot, bind1st(less<int>(), *pivot));

        swap(*change, *pivot);
        reverse(rfirst, pivot);

        return true;
    }
};
```

相關題目

- Permutation Sequence, 見 §2.1.14
- Permutations, 見 §8.3
- Permutations II, 見 §8.4
- Combinations, 見 §8.5

2.1.13 Prev Permutation

描述

Implement previous permutation

```
1,2,3 → 3,2,1
3,2,1 → 3,1,2
1,1,5 → 5,1,1
```

分析

無

代碼

```
// LeetCode, Previous Permutation
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    void prevPermutation(vector<int> &nums) {
```

```
        prev_permutation(nums.begin(), nums.end());
    }

    template<typename BidiIt>
    bool prev_permutation(BidiIt first, BidiIt last) {
        const auto rfirst = reverse_iterator<BidiIt>(last);
        const auto rlast = reverse_iterator<BidiIt>(first);

        auto pivot = next(rfirst);

        while (pivot != rlast && *(pivot) <= *prev(pivot))
            pivot++;

        if (pivot == rlast) {
            reverse(rfirst, rlast);
            return false;
        }

        auto change = find_if(rfirst, pivot, bind1st(greater<int>(), *pivot));

        swap(*change, *pivot);
        reverse(rfirst, pivot);

        return true;
    }
};
```

相關題目

- Permutation Sequence, 見 §2.1.14
- Permutations, 見 §8.3
- Permutations II, 見 §8.4
- Combinations, 見 §8.5

2.1.14 Permutation Sequence

描述

The set `[1,2,3,⋯,n]` contains a total of $n!$ unique permutations.

By listing and labeling all of the permutations in order, We get the following sequence (ie, for $n = 3$):

```
"123"
"132"
"213"
"231"
"312"
"321"
```

Given n and k, return the kth permutation sequence.

Note: Given n will be between 1 and 9 inclusive.

分析

簡單的，可以用暴力枚舉法，調用 $k-1$ 次 `next_permutation()`。

暴力枚舉法把前 k 個排列都求出來了，比較浪費，而我們只需要第 k 個排列。

利用康託編碼的思路，假設有 n 個不重複的元素，第 k 個排列是 $a_1, a_2, a_3, ..., a_n$，那麼 a_1 是哪一個位置呢？

我們把 a_1 去掉，那麼剩下的排列為 $a_2, a_3, ..., a_n$，共計 $n-1$ 個元素，$n-1$ 個元素共有 $(n-1)!$ 個排列，於是就可以知道 $a_1 = k/(n-1)!$。

同理，$a_2, a_3, ..., a_n$ 的值推導如下：

$$\begin{aligned} k_2 &= k\%(n-1)! \\ a_2 &= k_2/(n-2)! \\ &\cdots \\ k_{n-1} &= k_{n-2}\%2! \\ a_{n-1} &= k_{n-1}/1! \\ a_n &= 0 \end{aligned}$$

使用 next_permutation()

```
// LeetCode, Permutation Sequence
// 使用 next_permutation(), TLE
class Solution {
public:
    string getPermutation(int n, int k) {
        string s(n, '0');
        for (int i = 0; i < n; ++i)
            s[i] += i+1;
        for (int i = 0; i < k-1; ++i)
            next_permutation(s.begin(), s.end());
        return s;
    }

    template<typename BidiIt>
    bool next_permutation(BidiIt first, BidiIt last) {
        // 代碼見上一題 Next Permutation
    }
};
```

康託編碼

```
// LeetCode, Permutation Sequence
// 康託編碼, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
```

```
    string getPermutation(int n, int k) {
        string s(n, '0');
        string result;
        for (int i = 0; i < n; ++i)
            s[i] += i + 1;

        return kth_permutation(s, k);
    }
private:
    int factorial(int n) {
        int result = 1;
        for (int i = 1; i <= n; ++i)
            result *= i;
        return result;
    }

    // seq 已排好序, 是第一個排列
    template<typename Sequence>
    Sequence kth_permutation(const Sequence &seq, int k) {
        const int n = seq.size();
        Sequence S(seq);
        Sequence result;

        int base = factorial(n - 1);
        --k;  // 康託編碼從 0 開始

        for (int i = n - 1; i > 0; k %= base, base /= i, --i) {
            auto a = next(S.begin(), k / base);
            result.push_back(*a);
            S.erase(a);
        }

        result.push_back(S[0]); // 最後一個
        return result;
    }
};
```

相關題目

- Next Permutation, 見 §2.1.12
- Permutations, 見 §8.3
- Permutations II, 見 §8.4
- Combinations, 見 §8.5

2.1.15 Valid Sudoku

描述

Determine if a Sudoku is valid, according to: Sudoku Puzzles - The Rules http://sudoku.com.au/TheRules.aspx .

The Sudoku board could be partially filled, where empty cells are filled with the character '.'.

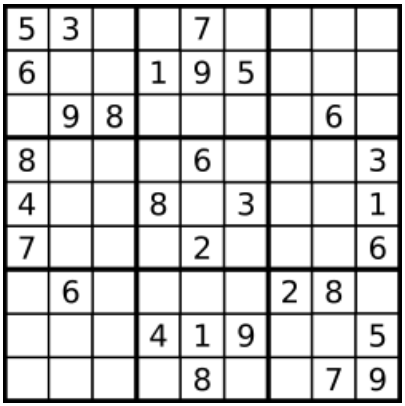


图 2-2 A partially filled sudoku which is valid

分析

細節實現題。

代碼

```
// LeetCode, Valid Sudoku
// 時間複雜度 O(n^2), 空間複雜度 O(1)
class Solution {
public:
    bool isValidSudoku(const vector<vector<char>>& board) {
        bool used[9];

        for (int i = 0; i < 9; ++i) {
            fill(used, used + 9, false);

            for (int j = 0; j < 9; ++j) // 檢查行
                if (!check(board[i][j], used))
                    return false;

            fill(used, used + 9, false);

            for (int j = 0; j < 9; ++j) // 檢查列
                if (!check(board[j][i], used))
                    return false;
        }

        for (int r = 0; r < 3; ++r) // 檢查 9 個子格子
            for (int c = 0; c < 3; ++c) {
                fill(used, used + 9, false);

                for (int i = r * 3; i < r * 3 + 3; ++i)
```

```
                    for (int j = c * 3; j < c * 3 + 3; ++j)
                        if (!check(board[i][j], used))
                            return false;
                }

            return true;
        }

        bool check(char ch, bool used[9]) {
            if (ch == '.') return true;

            if (used[ch - '1']) return false;

            return used[ch - '1'] = true;
        }
    };
```

相關題目

- Sudoku Solver, 見 §10.10

2.1.16 Trapping Rain Water

描述

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example, Given

$$0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1$$

, return 6.

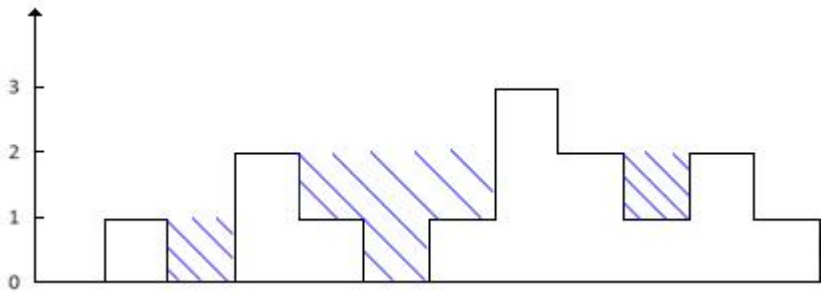


图 2-3 Trapping Rain Water

分析

對於每個柱子，找到其左右兩邊最高的柱子，該柱子能容納的面積就是 `min(max_left, max_right) - height`。所以，

1. 從左往右掃描一遍，對於每個柱子，求取左邊最大值；
2. 從右往左掃描一遍，對於每個柱子，求最大右值；
3. 再掃描一遍，把每個柱子的面積並累加。

也可以，

1. 掃描一遍，找到最高的柱子，這個柱子將數組分為兩半；
2. 處理左邊一半；
3. 處理右邊一半。

代碼 1

```
// LeetCode, Trapping Rain Water
// 思路 1, 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    int trap(const vector<int>& A) {
        const int n = A.size();
        int *max_left = new int[n]();
        int *max_right = new int[n]();

        for (int i = 1; i < n; i++) {
            max_left[i] = max(max_left[i - 1], A[i - 1]);
            max_right[n - 1 - i] = max(max_right[n - i], A[n - i]);

        }

        int sum = 0;
        for (int i = 0; i < n; i++) {
            int height = min(max_left[i], max_right[i]);
            if (height > A[i]) {
                sum += height - A[i];
            }
        }

        delete[] max_left;
        delete[] max_right;
        return sum;
    }
};
```

代碼 2

```
// LeetCode, Trapping Rain Water
// 思路 2, 時間複雜度 O(n), 空間複雜度 O(1)
```

```
class Solution {
public:
    int trap(const vector<int>& A) {
        const int n = A.size();
        int max = 0; // 最高的柱子，將數組分為兩半
        for (int i = 0; i < n; i++)
            if (A[i] > A[max]) max = i;

        int water = 0;
        for (int i = 0, peak = 0; i < max; i++)
            if (A[i] > peak) peak = A[i];
            else water += peak - A[i];
        for (int i = n - 1, top = 0; i > max; i--)
            if (A[i] > top) top = A[i];
            else water += top - A[i];
        return water;
    }
};
```

代碼 3

第三種解法，用一個棧輔助，小於棧頂的元素壓入，大於等於棧頂就把棧裏所有小於或等於當前值的元素全部出棧處理掉。

```
// LeetCode, Trapping Rain Water
// 用一個棧輔助，小於棧頂的元素壓入，大於等於棧頂就把棧裏所有小於或
// 等於當前值的元素全部出棧處理掉，計算面積，最後把當前元素入棧
// 時間複雜度 O(n)，空間複雜度 O(n)
class Solution {
public:
    int trap(const vector<int>& A) {
        const int n = A.size();
        stack<pair<int, int>> s;
        int water = 0;

        for (int i = 0; i < n; ++i) {
            int height = 0;

            while (!s.empty()) { // 將棧裏比當前元素矮或等高的元素全部處理掉
                int bar = s.top().first;
                int pos = s.top().second;
                // bar, height, A[i] 三者夾成的凹陷
                water += (min(bar, A[i]) - height) * (i - pos - 1);
                height = bar;

                if (A[i] < bar) // 碰到了比當前元素高的，跳出循環
                    break;
                else
                    s.pop(); // 彈出棧頂，因為該元素處理完了，不再需要了
            }

            s.push(make_pair(A[i], i));
```

```
        }

        return water;
    }
};
```

相關題目

- Container With Most Water, 見 §12.6
- Largest Rectangle in Histogram, 見 §4.1.3

2.1.17 Rotate Image

描述

You are given an $n \times n$ 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up: Could you do this in-place?

分析

首先想到，純模擬，從外到內一圈一圈的轉，但這個方法太慢。

如下圖，首先沿着副對角線翻轉一次，然後沿着水平中線翻轉一次。

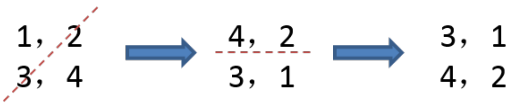


图 2-4 Rotate Image

或者，首先沿着水平中線翻轉一次，然後沿着主對角線翻轉一次。

代碼 1

```
// LeetCode, Rotate Image
// 思路 1, 時間複雜度 O(n^2), 空間複雜度 O(1)
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        const int n = matrix.size();

        for (int i = 0; i < n; ++i)  // 沿着副對角線反轉
            for (int j = 0; j < n - i; ++j)
                swap(matrix[i][j], matrix[n - 1 - j][n - 1 - i]);

        for (int i = 0; i < n / 2; ++i) // 沿着水平中線反轉
```

```
            for (int j = 0; j < n; ++j)
                swap(matrix[i][j], matrix[n - 1 - i][j]);
    }
};
```

代碼 2

```
// LeetCode, Rotate Image
// 思路 2, 時間複雜度 O(n^2), 空間複雜度 O(1)
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        const int n = matrix.size();

        for (int i = 0; i < n / 2; ++i) // 沿着水平中線反轉
            for (int j = 0; j < n; ++j)
                swap(matrix[i][j], matrix[n - 1 - i][j]);

        for (int i = 0; i < n; ++i)  // 沿着主對角線反轉
            for (int j = i + 1; j < n; ++j)
                swap(matrix[i][j], matrix[j][i]);
    }
};
```

相關題目

- 無

2.1.18 Plus One

描述

Given a number represented as an array of digits, plus one to the number.

分析

高精度加法。

代碼 1

```
// LeetCode, Plus One
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    vector<int> plusOne(vector<int> &digits) {
        add(digits, 1);
        return digits;
    }
private:
```

```
    // 0 <= digit <= 9
    void add(vector<int> &digits, int digit) {
        int c = digit;  // carry, 進位

        for (auto it = digits.rbegin(); it != digits.rend(); ++it) {
            *it += c;
            c = *it / 10;
            *it %= 10;
        }

        if (c > 0) digits.insert(digits.begin(), 1);
    }
};
```

代碼 2

```
// LeetCode, Plus One
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    vector<int> plusOne(vector<int> &digits) {
        add(digits, 1);
        return digits;
    }
private:
    // 0 <= digit <= 9
    void add(vector<int> &digits, int digit) {
        int c = digit;  // carry, 進位

        for_each(digits.rbegin(), digits.rend(), [&c](int &d){
            d += c;
            c = d / 10;
            d %= 10;
        });

        if (c > 0) digits.insert(digits.begin(), 1);
    }
};
```

相關題目

- 無

2.1.19 Climbing Stairs

描述

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

分析

設 $f(n)$ 表示爬 n 階樓梯的不同方法數，為了爬到第 n 階樓梯，有兩個選擇：

- 從第 $n-1$ 階前進 1 步；
- 從第 $n-1$ 階前進 2 步；

因此，有 $f(n) = f(n-1) + f(n-2)$。

這是一個斐波那契數列。

方法 1，遞歸，太慢；方法 2，迭代。

方法 3，數學公式。斐波那契數列的通項公式為 $a_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$。

迭代

```
// LeetCode, Climbing Stairs
// 迭代, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int climbStairs(int n) {
        int prev = 0;
        int cur = 1;
        for(int i = 1; i <= n ; ++i){
            int tmp = cur;
            cur += prev;
            prev = tmp;
        }
        return cur;
    }
};
```

數學公式

```
// LeetCode, Climbing Stairs
// 數學公式, 時間複雜度 O(1), 空間複雜度 O(1)
class Solution {
    public:
    int climbStairs(int n) {
        const double s = sqrt(5);
        return floor((pow((1+s)/2, n+1) + pow((1-s)/2, n+1))/s + 0.5);
    }
};
```

相關題目

- Decode Ways, 見 §13.10

2.1.20 Gray Code

描述

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given $n = 2$, return `[0,1,3,2]`. Its gray code sequence is:

```
00 - 0
01 - 1
11 - 3
10 - 2
```

Note:

- For a given n, a gray code sequence is not uniquely defined.
- For example, `[0,2,3,1]` is also a valid gray code sequence according to the above definition.
- For now, the judge is able to judge based on one instance of gray code sequence. Sorry about that.

分析

格雷碼 (Gray Code) 的定義請參考 http://en.wikipedia.org/wiki/Gray_code

自然二進制碼轉換為格雷碼：$g_0 = b_0, g_i = b_i \oplus b_{i-1}$

保留自然二進制碼的最高位作為格雷碼的最高位，格雷碼次高位為二進制碼的高位與次高位異或，其餘各位與次高位的求法類似。例如，將自然二進制碼 1001，轉換為格雷碼的過程是：保留最高位；然後將第 1 位的 1 和第 2 位的 0 異或，得到 1，作為格雷碼的第 2 位；將第 2 位的 0 和第 3 位的 0 異或，得到 0，作為格雷碼的第 3 位；將第 3 位的 0 和第 4 位的 1 異或，得到 1，作為格雷碼的第 4 位，最終，格雷碼為 1101。

格雷碼轉換為自然二進制碼：$b_0 = g_0, b_i = g_i \oplus b_{i-1}$

保留格雷碼的最高位作為自然二進制碼的最高位，次高位為自然二進制高位與格雷碼次高位異或，其餘各位與次高位的求法類似。例如，將格雷碼 1000 轉換為自然二進制碼的過程是：保留最高位 1，作為自然二進制碼的最高位；然後將自然二進制碼的第 1 位 1 和格雷碼的第 2 位 0 異或，得到 1，作為自然二進制碼的第 2 位；將自然二進制碼的第 2 位 1 和格雷碼的第 3 位 0 異或，得到 1，作為自然二進制碼的第 3 位；將自然二進制碼的第 3 位 1 和格雷碼的第 4 位 0 異或，得到 1，作為自然二進制碼的第 4 位，最終，自然二進制碼為 1111。

格雷碼有**數學公式**，整數 n 的格雷碼是 $n \oplus (n/2)$。

這題要求生成 n 比特的所有格雷碼。

方法 1，最簡單的方法，利用數學公式，對從 $0 \sim 2^n - 1$ 的所有整數，轉化為格雷碼。

方法 2，n 比特的格雷碼，可以遞歸地從 $n - 1$ 比特的格雷碼生成。如圖 §2-5 所示。

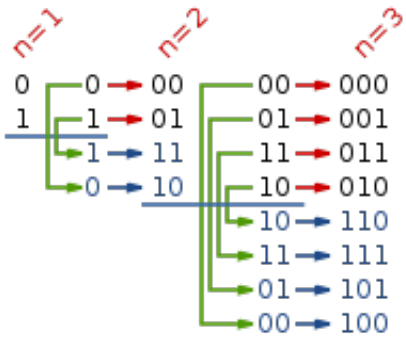


图 2-5 The first few steps of the reflect-and-prefix method.

數學公式

```
// LeetCode, Gray Code
// 數學公式, 時間複雜度 O(2^n), 空間複雜度 O(1)
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int> result;
        const size_t size = 1 << n;  // 2^n
        result.reserve(size);
        for (size_t i = 0; i < size; ++i)
            result.push_back(binary_to_gray(i));
        return result;
    }
private:
    static unsigned int binary_to_gray(unsigned int n) {
        return n ^ (n >> 1);
    }
};
```

Reflect-and-prefix method

```
// LeetCode, Gray Code
// reflect-and-prefix method
// 時間複雜度 O(2^n), 空間複雜度 O(1)
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int> result;
        result.reserve(1<<n);
        result.push_back(0);
        for (int i = 0; i < n; i++) {
            const int highest_bit = 1 << i;
            for (int j = result.size() - 1; j >= 0; j--) // 要反着遍歷, 才能對稱
                result.push_back(highest_bit | result[j]);
        }
```

```
        return result;
    }
};
```

相關題目

- 無

2.1.21 Set Matrix Zeroes

描述

Given a $m \times n$ matrix, if an element is 0, set its entire row and column to 0. Do it in place.

Follow up: Did you use extra space?

A straight forward solution using $O(mn)$ space is probably a bad idea.

A simple improvement uses $O(m + n)$ space, but still not the best solution.

Could you devise a constant space solution?

分析

$O(m + n)$ 空間的方法很簡單，設置兩個 bool 數組，記錄每行和每列是否存在 0。

想要常數空間，可以複用第一行和第一列。

代碼 1

```
// LeetCode, Set Matrix Zeroes
// 時間複雜度 O(m*n), 空間複雜度 O(m+n)
class Solution {
public:
    void setZeroes(vector<vector<int> > &matrix) {
        const size_t m = matrix.size();
        const size_t n = matrix[0].size();
        vector<bool> row(m, false); // 標記該行是否存在 0
        vector<bool> col(n, false); // 標記該列是否存在 0

        for (size_t i = 0; i < m; ++i) {
            for (size_t j = 0; j < n; ++j) {
                if (matrix[i][j] == 0) {
                    row[i] = col[j] = true;
                }
            }
        }

        for (size_t i = 0; i < m; ++i) {
            if (row[i])
                fill(&matrix[i][0], &matrix[i][0] + n, 0);
        }
```

```
        for (size_t j = 0; j < n; ++j) {
            if (col[j]) {
                for (size_t i = 0; i < m; ++i) {
                    matrix[i][j] = 0;
                }
            }
        }
    }
};
```

代碼 2

```
// LeetCode, Set Matrix Zeroes
// 時間複雜度 O(m*n), 空間複雜度 O(1)
class Solution {
public:
    void setZeroes(vector<vector<int> > &matrix) {
        const size_t m = matrix.size();
        const size_t n = matrix[0].size();
        bool row_has_zero = false; // 第一行是否存在 0
        bool col_has_zero = false; // 第一列是否存在 0

        for (size_t i = 0; i < n; i++)
            if (matrix[0][i] == 0) {
                row_has_zero = true;
                break;
            }

        for (size_t i = 0; i < m; i++)
            if (matrix[i][0] == 0) {
                col_has_zero = true;
                break;
            }

        for (size_t i = 1; i < m; i++)
            for (size_t j = 1; j < n; j++)
                if (matrix[i][j] == 0) {
                    matrix[0][j] = 0;
                    matrix[i][0] = 0;
                }
        for (size_t i = 1; i < m; i++)
            for (size_t j = 1; j < n; j++)
                if (matrix[i][0] == 0 || matrix[0][j] == 0)
                    matrix[i][j] = 0;
        if (row_has_zero)
            for (size_t i = 0; i < n; i++)
                matrix[0][i] = 0;
        if (col_has_zero)
            for (size_t i = 0; i < m; i++)
                matrix[i][0] = 0;
    }
};
```

相關題目

- 無

2.1.22 Gas Station

描述

There are N gas stations along a circular route, where the amount of gas at station i is `gas[i]`.

You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

Note: The solution is guaranteed to be unique.

分析

首先想到的是 $O(N^2)$ 的解法，對每個點進行模擬。

$O(N)$ 的解法是，設置兩個變量，`sum` 判斷當前的指針的有效性；`total` 則判斷整個數組是否有解，有就返回通過 `sum` 得到的下標，沒有則返回-1。

代碼

```
// LeetCode, Gas Station
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int canCompleteCircuit(vector<int> &gas, vector<int> &cost) {
        int total = 0;
        int j = -1;
        for (int i = 0, sum = 0; i < gas.size(); ++i) {
            sum += gas[i] - cost[i];
            total += gas[i] - cost[i];
            if (sum < 0) {
                j = i;
                sum = 0;
            }
        }
        return total >= 0 ? j + 1 : -1;
    }
};
```

相關題目

- 無

2.1.23 Candy

描述

There are N children standing in a line. Each child is assigned a rating value.

You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

分析

無

迭代版

```
// LeetCode, Candy
// 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    int candy(vector<int> &ratings) {
        const int n = ratings.size();
        vector<int> increment(n);

        // 左右各掃描一遍
        for (int i = 1, inc = 1; i < n; i++) {
            if (ratings[i] > ratings[i - 1])
                increment[i] = max(inc++, increment[i]);
            else
                inc = 1;
        }

        for (int i = n - 2, inc = 1; i >= 0; i--) {
            if (ratings[i] > ratings[i + 1])
                increment[i] = max(inc++, increment[i]);
            else
                inc = 1;
        }
        // 初始值為 n, 因為每個小朋友至少一顆糖
        return accumulate(&increment[0], &increment[0]+n, n);
    }
};
```

遞歸版

```
// LeetCode, Candy
// 備忘錄法, 時間複雜度 O(n), 空間複雜度 O(n)
// @author fancymouse (http://weibo.com/u/1928162822)
```

```
class Solution {
public:
    int candy(const vector<int>& ratings) {
        vector<int> f(ratings.size());
        int sum = 0;
        for (int i = 0; i < ratings.size(); ++i)
            sum += solve(ratings, f, i);
        return sum;
    }
    int solve(const vector<int>& ratings, vector<int>& f, int i) {
        if (f[i] == 0) {
            f[i] = 1;
            if (i > 0 && ratings[i] > ratings[i - 1])
                f[i] = max(f[i], solve(ratings, f, i - 1) + 1);
            if (i < ratings.size() - 1 && ratings[i] > ratings[i + 1])
                f[i] = max(f[i], solve(ratings, f, i + 1) + 1);
        }
        return f[i];
    }
};
```

相關題目

- 無

2.1.24 Single Number

描述

Given an array of integers, every element appears twice except for one. Find that single one.

Note: Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

分析

異或 (XOR)，不僅能處理兩次的情況，只要出現偶數次，都可以清零。

代碼 1

```
// LeetCode, Single Number
// 時間複雜度 O(n)，空間複雜度 O(1)
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int x = 0;
        for (auto i : nums) {
            x ^= i;
        }
```

```
        return x;
    }
};
```

代碼 2

```
// LeetCode, Single Number
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        return accumulate(nums.begin(), nums.end(), 0, bit_xor<int>());
    }
};
```

相關題目

- Single Number II, 見 §2.1.25

2.1.25 Single Number II

描述

Given an array of integers, every element appears three times except for one. Find that single one.

Note: Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

分析

本題和上一題 Single Number，考察的是位運算。

方法 1：創建一個長度為 `sizeof(int)` 的數組 `count[sizeof(int)]`，`count[i]` 表示在在 i 位出現的 1 的次數。如果 `count[i]` 是 3 的整數倍，則忽略；否則就把該位取出來組成答案。

方法 2：用 `one` 記錄到當前處理的元素為止，二進制 1 出現“1 次”（mod 3 之後的 1）的有哪些二進制位；用 `two` 記錄到當前計算的變量為止，二進制 1 出現“2 次”（mod 3 之後的 2）的有哪些二進制位。當 `one` 和 `two` 中的某一位同時為 1 時表示該二進制位上 1 出現了 3 次，此時需要清零。即**用二進制模擬三進制運算**。最終 `one` 記錄的是最終結果。

代碼 1

```
// LeetCode, Single Number II
// 方法 1, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        const int W = sizeof(int) * 8; // 一個整數的 bit 數, 即整數字長
```

```
        int count[W];  // count[i] 表示在在 i 位出現的 1 的次數
        fill_n(&count[0], W, 0);
        for (int i = 0; i < nums.size(); i++) {
            for (int j = 0; j < W; j++) {
                count[j] += (nums[i] >> j) & 1;
                count[j] %= 3;
            }
        }
        int result = 0;
        for (int i = 0; i < W; i++) {
            result += (count[i] << i);
        }
        return result;
    }
};
```

代碼 2

```
// LeetCode, Single Number II
// 方法 2, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int one = 0, two = 0, three = 0;
        for (auto i : nums) {
            two |= (one & i);
            one ^= i;
            three = ~(one & two);
            one &= three;
            two &= three;
        }

        return one;
    }
};
```

相關題目

- Single Number, 見 §2.1.24

2.2 單鏈表

單鏈表節點的定義如下:

```
// 單鏈表節點
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) { }
};
```

2.2.1 Add Two Numbers

描述

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

Input: `(2 -> 4 -> 3) + (5 -> 6 -> 4)`

Output: `7 -> 0 -> 8`

分析

跟 Add Binary（見 §3.4）很類似

代碼

```
// LeetCode, Add Two Numbers
// 跟 Add Binary 很類似
// 時間複雜度 O(m+n), 空間複雜度 O(1)
class Solution {
public:
    ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
        ListNode dummy(-1); // 頭節點
        int carry = 0;
        ListNode *prev = &dummy;
        for (ListNode *pa = l1, *pb = l2;
             pa != nullptr || pb != nullptr;
             pa = pa == nullptr ? nullptr : pa->next,
             pb = pb == nullptr ? nullptr : pb->next,
             prev = prev->next) {
            const int ai = pa == nullptr ? 0 : pa->val;
            const int bi = pb == nullptr ? 0 : pb->val;
            const int value = (ai + bi + carry) % 10;
            carry = (ai + bi + carry) / 10;
            prev->next = new ListNode(value); // 尾插法
        }
        if (carry > 0)
            prev->next = new ListNode(carry);
        return dummy.next;
    }
};
```

相關題目

- Add Binary, 見 §3.4

2.2.2 Reverse Linked List II

描述

Reverse a linked list from position m to n. Do it in-place and in one-pass.

For example: Given `1->2->3->4->5->nullptr`, $m = 2$ and $n = 4$,

return `1->4->3->2->5->nullptr`.

Note: Given m, n satisfy the following condition: $1 \leq m \leq n \leq$ length of list.

分析

這題非常繁瑣，有很多邊界檢查，15 分鐘內做到 bug free 很有難度!

代碼

```
// LeetCode, Reverse Linked List II
// 迭代版, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode *reverseBetween(ListNode *head, int m, int n) {
        ListNode dummy(-1);
        dummy.next = head;

        ListNode *prev = &dummy;
        for (int i = 0; i < m-1; ++i)
            prev = prev->next;
        ListNode* const head2 = prev;

        prev = head2->next;
        ListNode *cur = prev->next;
        for (int i = m; i < n; ++i) {
            prev->next = cur->next;
            cur->next = head2->next;
            head2->next = cur;  // 頭插法
            cur = prev->next;
        }

        return dummy.next;
    }
};
```

相關題目

- 無

2.2.3 Partition List

描述

Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

For example, Given `1->4->3->2->5->2` and `x = 3`, return `1->2->2->3->4->5`.

分析

無

代碼

```
// LeetCode, Partition List
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        ListNode left_dummy(-1); // 頭結點
        ListNode right_dummy(-1); // 頭結點

        auto left_cur = &left_dummy;
        auto right_cur = &right_dummy;

        for (ListNode *cur = head; cur; cur = cur->next) {
            if (cur->val < x) {
                left_cur->next = cur;
                left_cur = cur;
            } else {
                right_cur->next = cur;
                right_cur = cur;
            }
        }

        left_cur->next = right_dummy.next;
        right_cur->next = nullptr;

        return left_dummy.next;
    }
};
```

相關題目

- 無

2.2.4 Remove Duplicates from Sorted List

描述

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example,

Given `1->1->2`, return `1->2`.

Given `1->1->2->3->3`, return `1->2->3`.

分析

無

遞歸版

```
// LeetCode, Remove Duplicates from Sorted List
// 遞歸版, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
        if (!head) return head;
        ListNode dummy(head->val + 1); // 值只要跟 head 不同即可
        dummy.next = head;

        recur(&dummy, head);
        return dummy.next;
    }
private:
    static void recur(ListNode *prev, ListNode *cur) {
        if (cur == nullptr) return;

        if (prev->val == cur->val) { // 刪除 head
            prev->next = cur->next;
            delete cur;
            recur(prev, prev->next);
        } else {
            recur(prev->next, cur->next);
        }
    }
};
```

迭代版

```
// LeetCode, Remove Duplicates from Sorted List
// 迭代版, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
        if (head == nullptr) return nullptr;
```

```
        for (ListNode *prev = head, *cur = head->next; cur; cur = prev->next) {
            if (prev->val == cur->val) {
                prev->next = cur->next;
                delete cur;
            } else {
                prev = cur;
            }
        }
        return head;
    }
};
```

相關題目

- Remove Duplicates from Sorted List II，見 §2.2.5

2.2.5 Remove Duplicates from Sorted List II

描述

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

For example,

Given `1->2->3->3->4->4->5`, return `1->2->5`.

Given `1->1->1->2->3`, return `2->3`.

分析

無

遞歸版

```
// LeetCode, Remove Duplicates from Sorted List II
// 遞歸版, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
        if (!head || !head->next) return head;

        ListNode *p = head->next;
        if (head->val == p->val) {
            while (p && head->val == p->val) {
                ListNode *tmp = p;
                p = p->next;
                delete tmp;
            }
```

```
            delete head;
            return deleteDuplicates(p);
        } else {
            head->next = deleteDuplicates(head->next);
            return head;
        }
    }
};
```

迭代版

```
// LeetCode, Remove Duplicates from Sorted List II
// 迭代版, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
        if (head == nullptr) return head;

        ListNode dummy(INT_MIN); // 頭結點
        dummy.next = head;
        ListNode *prev = &dummy, *cur = head;
        while (cur != nullptr) {
            bool duplicated = false;
            while (cur->next != nullptr && cur->val == cur->next->val) {
                duplicated = true;
                ListNode *temp = cur;
                cur = cur->next;
                delete temp;
            }
            if (duplicated) { // 刪除重複的最後一個元素
                ListNode *temp = cur;
                cur = cur->next;
                delete temp;
                continue;
            }
            prev->next = cur;
            prev = prev->next;
            cur = cur->next;
        }
        prev->next = cur;
        return dummy.next;
    }
};
```

迭代版

```
// LeetCode, Remove Duplicates from Sorted List II
// 迭代版, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
        if (head == nullptr) return head;
```

```
        ListNode dummy(INT_MIN); dummy.next = head;
        ListNode *prev = &dummy;
        ListNode *cur = head;
        ListNode *next = cur != nullptr ? cur->next : nullptr;

        while (next != nullptr) {
            bool duplicated = false;
            while (next && cur->val == next->val) {
                duplicated = true;
                prev->next = next;
                delete cur;
                cur = next;
                next = cur != nullptr ? cur->next : nullptr;
            }
            if (duplicated) {
                prev->next = next;
                delete cur;
            }
            else
                prev = cur;
            cur = next;
            next = cur != nullptr ? cur->next : nullptr;
        }
        return dummy.next;
    }
};
```

相關題目

- Remove Duplicates from Sorted List，見 §2.2.4

2.2.6 Rotate List

描述

Given a list, rotate the list to the right by k places, where k is non-negative.

For example: Given `1->2->3->4->5->nullptr` and `k = 2`, return `4->5->1->2->3->nullptr`.

分析

先遍歷一遍，得出鏈表長度 len，注意 k 可能大於 len，因此令 $k\% = len$。將尾節點 next 指針指向首節點，形成一個環，接着往後跑 $len - k$ 步，從這裏斷開，就是要求的結果了。

代碼

```
// LeetCode, Remove Rotate List
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode *rotateRight(ListNode *head, int k) {
        if (head == nullptr || k == 0) return head;

        int len = 1;
        ListNode* p = head;
        while (p->next) { // 求長度
            len++;
            p = p->next;
        }
        k = len - k % len;

        p->next = head; // 首尾相連
        for(int step = 0; step < k; step++) {
            p = p->next;  //接着往後跑
        }
        head = p->next; // 新的首節點
        p->next = nullptr; // 斷開環
        return head;
    }
};
```

相關題目

- 無

2.2.7 Rotate List II

描述

Given a list, rotate the list to the left by k places, where k is non-negative.

For example: Given `1->2->3->4->5->nullptr` and `k = 2`, return `4->5->1->2->3->nullptr`.

分析

先遍歷一遍，得出鏈表長度 len，注意 k 可能大於 len，因此令 $k\% = len$。將尾節點 next 指針指向首節點，形成一個環，接着往後跑 $len - k$ 步，從這裏斷開，就是要求的結果了。

代碼

```
// LeetCode, Remove Rotate List
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
```

```
    ListNode *rotateRight(ListNode *head, int k) {
        if (head == nullptr || k == 0) return head;

        int len = 1;
        ListNode* p = head;
        while (p->next) { // 求長度
            len++;
            p = p->next;
        }
        k = k % len; // This is the only difference

        p->next = head; // 首尾相連
        for(int step = 0; step < k; step++) {
            p = p->next;  //接着往後跑
        }
        head = p->next; // 新的首節點
        p->next = nullptr; // 斷開環
        return head;
    }
};
```

相關題目

- 無

2.2.8 Remove Nth Node From End of List

描述

Given a linked list, remove the n^{th} node from the end of list and return its head.

For example, Given linked list: `1->2->3->4->5`, and $n = 2$.

After removing the second node from the end, the linked list becomes `1->2->3->5`.

Note:

- Given n will always be valid.
- Try to do this in one pass.

分析

設兩個指針 p, q，讓 q 先走 n 步，然後 p 和 q 一起走，直到 q 走到尾節點，刪除 `p->next` 即可。

代碼

```
// LeetCode, Remove Nth Node From End of List
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
```

```
    ListNode *removeNthFromEnd(ListNode *head, int n) {
        ListNode dummy{-1, head};
        ListNode *p = &dummy, *q = &dummy;

        for (int i = 0; i < n; i++)  // q 先走 n 步
            q = q->next;

        while(q->next) { // 一起走
            p = p->next;
            q = q->next;
        }
        ListNode *tmp = p->next;
        p->next = p->next->next;
        delete tmp;
        return dummy.next;
    }
};
```

相關題目

- 無

2.2.9 Swap Nodes in Pairs

描述

Given a linked list, swap every two adjacent nodes and return its head.

For example, Given `1->2->3->4`, you should return the list as `2->1->4->3`.

Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

分析

無

代碼

```
// LeetCode, Swap Nodes in Pairs
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode *swapPairs(ListNode *head) {
        if (head == nullptr || head->next == nullptr) return head;
        ListNode dummy(-1);
        dummy.next = head;

        for(ListNode *prev = &dummy, *cur = prev->next, *next = cur->next;
                next;
```

```
                prev = cur, cur = cur->next, next = cur ? cur->next: nullptr) {
            prev->next = next;
            cur->next = next->next;
            next->next = cur;
        }
        return dummy.next;
    }
};
```

下面這種寫法更簡潔，但題目規定了不允許這樣做。

```
// LeetCode, Swap Nodes in Pairs
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        ListNode* p = head;

        while (p && p->next) {
            swap(p->val, p->next->val);
            p = p->next->next;
        }

        return head;
    }
};
```

相關題目

- Reverse Nodes in k-Group, 見 §2.2.10

2.2.10 Reverse Nodes in k-Group

描述

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.

You may not alter the values in the nodes, only nodes itself may be changed.

Only constant memory is allowed.

For example, Given this linked list: `1->2->3->4->5`

For $k = 2$, you should return: `2->1->4->3->5`

For $k = 3$, you should return: `3->2->1->4->5`

分析

無

遞歸版

```
// LeetCode, Reverse Nodes in k-Group
// 遞歸版, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode *reverseKGroup(ListNode *head, int k) {
        if (head == nullptr || head->next == nullptr || k < 2)
            return head;

        ListNode *next_group = head;
        for (int i = 0; i < k; ++i) {
            if (next_group)
                next_group = next_group->next;
            else
                return head;
        }
        // next_group is the head of next group
        // new_next_group is the new head of next group after reversion
        ListNode *new_next_group = reverseKGroup(next_group, k);
        ListNode *prev = new_next_group, *cur = head;
        while (cur != next_group) {
            ListNode *next = cur->next;
            cur->next = prev;
            prev = cur;
            cur = next;
        }
        return prev; // prev will be the new head of this group
    }
};
```

迭代版

```
// LeetCode, Reverse Nodes in k-Group
// 迭代版, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode *reverseKGroup(ListNode *head, int k) {
        if (head == nullptr || head->next == nullptr || k < 2) return head;
        ListNode dummy(-1);
        dummy.next = head;

        for(ListNode *prev = &dummy, *end = head; end; end = prev->next) {
            for (int i = 1; i < k && end; i++)
                end = end->next;
            if (end  == nullptr) break;  // 不足 k 個

            prev = reverse(prev, prev->next, end);
        }

        return dummy.next;
    }
```

```
    // prev 是 first 前一個元素, [begin, end] 閉區間, 保證三者都不為 null
    // 返回反轉後的倒數第 1 個元素
    ListNode* reverse(ListNode *prev, ListNode *begin, ListNode *end) {
        ListNode *end_next = end->next;
        for (ListNode *p = begin, *cur = p->next, *next = cur->next;
                cur != end_next;
                p = cur, cur = next, next = next ? next->next : nullptr) {
            cur->next = p;
        }
        begin->next = end_next;
        prev->next = end;
        return begin;
    }
};
```

相關題目

- Swap Nodes in Pairs, 見 §2.2.9

2.2.11 Copy List with Random Pointer

描述

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

分析

無

代碼

```
// LeetCode, Copy List with Random Pointer
// 兩遍掃描, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    RandomListNode *copyRandomList(RandomListNode *head) {
        for (RandomListNode* cur = head; cur != nullptr; ) {
            RandomListNode* node = new RandomListNode(cur->label);
            node->next = cur->next;
            cur->next = node;
            cur = node->next;
        }

        for (RandomListNode* cur = head; cur != nullptr; ) {
            if (cur->random != NULL)
                cur->next->random = cur->random->next;
```

```
            cur = cur->next->next;
        }

        // 分拆兩個單鏈表
        RandomListNode dummy(-1);
        for (RandomListNode* cur = head, *new_cur = &dummy;
                cur != nullptr; ) {
            new_cur->next = cur->next;
            new_cur = new_cur->next;
            cur->next = cur->next->next;
            cur = cur->next;
        }
        return dummy.next;
    }
};
```

相關題目

- 無

2.2.12 Linked List Cycle

描述

Given a linked list, determine if it has a cycle in it.

Follow up: Can you solve it without using extra space?

分析

最容易想到的方法是，用一個哈希表 `unordered_map<int, bool> visited`，記錄每個元素是否被訪問過，一旦出現某個元素被重複訪問，說明存在環。空間複雜度 $O(n)$，時間複雜度 $O(N)$。

最好的方法是時間複雜度 $O(n)$，空間複雜度 $O(1)$ 的。設置兩個指針，一個快一個慢，快的指針每次走兩步，慢的指針每次走一步，如果快指針和慢指針相遇，則說明有環。參考 http://leetcode.com/2010/09/detecting-loop-in-singly-linked-list.html

代碼

```
//LeetCode, Linked List Cycle
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    bool hasCycle(ListNode *head) {
        // 設置兩個指針, 一個快一個慢
        ListNode *slow = head, *fast = head;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
```

```
            if (slow == fast) return true;
        }
        return false;
    }
};
```

相關題目

- Linked List Cycle II, 見 §2.2.13

2.2.13 Linked List Cycle II

描述

Given a linked list, return the node where the cycle begins. If there is no cycle, return `null`.

Follow up: Can you solve it without using extra space?

分析

當 fast 與 slow 相遇時，slow 肯定沒有遍歷完鏈表，而 fast 已經在環內循環了 n 圈 ($1 \leq n$)。假設 slow 走了 s 步，則 fast 走了 $2s$ 步（fast 步數還等於 s 加上在環上多轉的 n 圈），設環長為 r，則：

$$\begin{aligned} 2s &= s + nr \\ s &= nr \end{aligned}$$

設整個鏈表長 L，環入口點與相遇點距離為 a，起點到環入口點的距離為 x，則

$$\begin{aligned} x + a &= nr = (n-1)r + r = (n-1)r + L - x \\ x &= (n-1)r + (L - x - a) \end{aligned}$$

$L - x - a$ 為相遇點到環入口點的距離，由此可知，從鏈表頭到環入口點等於 $n-1$ 圈內環+相遇點到環入口點，於是我們可以從 `head` 開始另設一個指針 `slow2`，兩個慢指針每次前進一步，它倆一定會在環入口點相遇。

代碼

```
//LeetCode, Linked List Cycle II
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode *slow = head, *fast = head;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
```

```
            if (slow == fast) {
                ListNode *slow2 = head;

                while (slow2 != slow) {
                    slow2 = slow2->next;
                    slow = slow->next;
                }
                return slow2;
            }
        }
        return nullptr;
    }
};
```

相關題目

- Linked List Cycle, 見 §2.2.12

2.2.14 Reorder List

描述

Given a singly linked list $L : L_0 \to L_1 \to \cdots \to L_{n-1} \to L_n$, reorder it to: $L_0 \to L_n \to L_1 \to L_{n-1} \to L_2 \to L_{n-2} \to \cdots$

You must do this in-place without altering the nodes' values.

For example, Given `{1,2,3,4}`, reorder it to `{1,4,2,3}`.

分析

題目規定要 in-place，也就是說只能使用 $O(1)$ 的空間。

可以找到中間節點，斷開，把後半截單鏈表 reverse 一下，再合併兩個單鏈表。

代碼

```
// LeetCode, Reorder List
// 時間複雜度 O(n)，空間複雜度 O(1)
class Solution {
public:
    void reorderList(ListNode *head) {
        if (head == nullptr || head->next == nullptr) return;

        ListNode *slow = head, *fast = head, *prev = nullptr;
        while (fast && fast->next) {
            prev = slow;
            slow = slow->next;
            fast = fast->next->next;
        }
        prev->next = nullptr; // cut at middle
```

```
        slow = reverse(slow);

        // merge two lists
        ListNode *curr = head;
        while (curr->next) {
            ListNode *tmp = curr->next;
            curr->next = slow;
            slow = slow->next;
            curr->next->next = tmp;
            curr = tmp;
        }
        curr->next = slow;
    }

    ListNode* reverse(ListNode *head) {
        if (head == nullptr || head->next == nullptr) return head;

        ListNode *prev = head;
        for (ListNode *curr = head->next, *next = curr->next; curr;
            prev = curr, curr = next, next = next ? next->next : nullptr) {
                curr->next = prev;
        }
        head->next = nullptr;
        return prev;
    }
};
```

相關題目

- 無

2.2.15　LRU Cache

描述

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

`get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

`set(key, value)` - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

分析

為了使查找、插入和刪除都有較高的性能，我們使用一個雙向鏈表 (`std::list`) 和一個哈希表 (`std::unordered_map`)，因為:

- 哈希表保存每個節點的地址，可以基本保證在 $O(1)$ 時間內查找節點
- 雙向鏈表插入和刪除效率高，單向鏈表插入和刪除時，還要查找節點的前驅節點

具體實現細節:

- 越靠近鏈表頭部，表示節點上次訪問距離現在時間最短，尾部的節點表示最近訪問最少
- 訪問節點時，如果節點存在，把該節點交換到鏈表頭部，同時更新 hash 表中該節點的地址
- 插入節點時，如果 cache 的 size 達到了上限 capacity，則刪除尾部節點，同時要在 hash 表中刪除對應的項；新節點插入鏈表頭部

代碼

```
// LeetCode, LRU Cache
// 時間複雜度 O(logn), 空間複雜度 O(n)
class LRUCache{
private:
    struct CacheNode {
        int key;
        int value;
        CacheNode(int k, int v) :key(k), value(v){}
    };
public:
    LRUCache(int capacity) {
        this->capacity = capacity;
    }

    int get(int key) {
        if (cacheMap.find(key) == cacheMap.end()) return -1;

        // 把當前訪問的節點移到鏈表頭部, 並且更新 map 中該節點的地址
        cacheList.splice(cacheList.begin(), cacheList, cacheMap[key]);
        cacheMap[key] = cacheList.begin();
        return cacheMap[key]->value;
    }

    void set(int key, int value) {
        if (cacheMap.find(key) == cacheMap.end()) {
            if (cacheList.size() == capacity) { //刪除鏈表尾部節點（最少訪問的節點）
                cacheMap.erase(cacheList.back().key);
                cacheList.pop_back();
            }
            // 插入新節點到鏈表頭部, 並且在 map 中增加該節點
            cacheList.push_front(CacheNode(key, value));
            cacheMap[key] = cacheList.begin();
        } else {
            //更新節點的值, 把當前訪問的節點移到鏈表頭部, 並且更新 map 中該節點的地址
            cacheMap[key]->value = value;
            cacheList.splice(cacheList.begin(), cacheList, cacheMap[key]);
            cacheMap[key] = cacheList.begin();
        }
```

```
    }
private:
    list<CacheNode> cacheList;
    unordered_map<int, list<CacheNode>::iterator> cacheMap;
    int capacity;
};
```

相關題目

- 無

第 3 章
字符串

3.1 Valid Palindrome

描述

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example,

`"A man, a plan, a canal: Panama"` is a palindrome.

`"race a car"` is not a palindrome.

Note: Have you consider that the string might be empty? This is a good question to ask during an interview.

For the purpose of this problem, we define empty string as valid palindrome.

分析

無

代碼

```
// Leet Code, Valid Palindrome
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    bool isPalindrome(string s) {
        transform(s.begin(), s.end(), s.begin(), ::tolower);
        auto left = s.begin(), right = prev(s.end());
        while (left < right) {
            if (!::isalnum(*left))  ++left;
            else if (!::isalnum(*right)) --right;
            else if (*left != *right) return false;
            else { left++, right--; }
        }
        return true;
    }
};
```

相關題目

- Palindrome Number, 見 §15.2

3.2 Implement strStr()

描述

Implement strStr().

Returns a pointer to the first occurrence of needle in haystack, or null if needle is not part of haystack.

分析

暴力算法的複雜度是 $O(m*n)$，代碼如下。更高效的的算法有 KMP 算法、Boyer-Mooer 算法和 Rabin-Karp 算法。面試中暴力算法足夠了，一定要寫得沒有 BUG。

暴力匹配

```
// LeetCode, Implement strStr()
// 暴力解法, 時間複雜度 O(N*M), 空間複雜度 O(1)
class Solution {
public:
    int strStr(const string& haystack, const string& needle) {
        if (needle.empty()) return 0;

        const int N = haystack.size() - needle.size() + 1;
        for (int i = 0; i < N; i++) {
            int j = i;
            int k = 0;
            while (j < haystack.size() && k < needle.size() && haystack[j] == needle[k]) {
                j++;
                k++;
            }
            if (k == needle.size()) return i;
        }
        return -1;
    }
};
```

KMP

```
// LeetCode, Implement strStr()
// KMP, 時間複雜度 O(N+M), 空間複雜度 O(M)
class Solution {
public:
    int strStr(const string& haystack, const string& needle) {
        return kmp(haystack.c_str(), needle.c_str());
    }
private:
    /*
```

```
     * @brief 計算部分匹配表, 即 next 數組.
     *
     * @param[in] pattern 模式串
     * @param[out] next next 數組
     * @return 無
     */
    static void compute_prefix(const char *pattern, int next[]) {
        int i;
        int j = -1;
        const int m = strlen(pattern);

        next[0] = j;
        for (i = 1; i < m; i++) {
            while (j > -1 && pattern[j + 1] != pattern[i]) j = next[j];

            if (pattern[i] == pattern[j + 1]) j++;
            next[i] = j;
        }
    }

    /*
     * @brief KMP 算法.
     *
     * @param[in] text 文本
     * @param[in] pattern 模式串
     * @return 成功則返回第一次匹配的位置, 失敗則返回-1
     */
    static int kmp(const char *text, const char *pattern) {
        int i;
        int j = -1;
        const int n = strlen(text);
        const int m = strlen(pattern);
        if (n == 0 && m == 0) return 0; /* "","" */
        if (m == 0) return 0;  /* "a","" */
        int *next = (int*)malloc(sizeof(int) * m);

        compute_prefix(pattern, next);

        for (i = 0; i < n; i++) {
            while (j > -1 && pattern[j + 1] != text[i]) j = next[j];

            if (text[i] == pattern[j + 1]) j++;
            if (j == m - 1) {
                free(next);
                return i-j;
            }
        }

        free(next);
        return -1;
    }
};
```

相關題目

- String to Integer (atoi)，見 §3.3

3.3 String to Integer (atoi)

描述

Implement `atoi` to convert a string to an integer.

Hint: Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.

Notes: It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.

Requirements for atoi:

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, `INT_MAX (2147483647)` or `INT_MIN (-2147483648)` is returned.

分析

細節題。注意幾個測試用例:

1. 不規則輸入，但是有效，”-3924x8fc”， ” + 413”,
2. 無效格式，” ++c”, ” ++1”
3. 溢出數據，”2147483648”

代碼

```
// LeetCode, String to Integer (atoi)
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int myAtoi(const string &str) {
        int num = 0;
        int sign = 1;
        const int n = str.length();
```

```
        int i = 0;

        while (str[i] == ' ' && i < n) i++;
        if (i == n) return 0;

        if (str[i] == '+') {
            i++;
        } else if (str[i] == '-') {
            sign = -1;
            i++;
        }

        for (; i < n; i++) {
            if (str[i] < '0' || str[i] > '9')
                break;
            if (num > INT_MAX / 10 ||
                            (num == INT_MAX / 10 &&
                                    (str[i] - '0') > INT_MAX % 10)) {
                return sign == -1 ? INT_MIN : INT_MAX;
            }
            num = num * 10 + str[i] - '0';
        }
        return num * sign;
    }
};
```

相關題目

- Implement strStr()，見 §3.2

3.4 Add Binary

描述

Given two binary strings, return their sum (also a binary string).

For example,

```
a = "11"
b = "1"
```

Return `"100"`.

分析

無

代碼

```
//LeetCode, Add Binary
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
```

```
public:
    string addBinary(string a, string b) {
        string result;
        const size_t n = a.size() > b.size() ? a.size() : b.size();
        reverse(a.begin(), a.end());
        reverse(b.begin(), b.end());
        int carry = 0;
        for (size_t i = 0; i < n; i++) {
            const int ai = i < a.size() ? a[i] - '0' : 0;
            const int bi = i < b.size() ? b[i] - '0' : 0;
            const int val = (ai + bi + carry) % 2;
            carry = (ai + bi + carry) / 2;
            result.insert(result.begin(), val + '0');
        }
        if (carry == 1) {
            result.insert(result.begin(), '1');
        }
        return result;
    }
};
```

相關題目

- Add Two Numbers, 見 §2.2.1

3.5 Longest Palindromic Substring

描述

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

分析

最長迴文子串，非常經典的題。

思路一：暴力枚舉，以每個元素為中間元素，同時從左右出發，複雜度 $O(n^2)$。

思路二：記憶化搜索，複雜度 $O(n^2)$。設 `f[i][j]` 表示 [i,j] 之間的最長迴文子串，遞推方程如下：

```
f[i][j] = if (i == j) S[i]
          if (S[i] == S[j] && f[i+1][j-1] == S[i+1][j-1]) S[i][j]
          else max(f[i+1][j-1], f[i][j-1], f[i+1][j])
```

思路三：動規，複雜度 $O(n^2)$。設狀態為 `f(i,j)`，表示區間 [i,j] 是否為迴文串，則狀態轉移方程為

$$f(i,j) = \begin{cases} true & ,i = j \\ S[i] = S[j] & ,j = i+1 \\ S[i] = S[j] \text{ and } f(i+1, j-1) & ,j > i+1 \end{cases}$$

思路四：Manacher’s Algorithm，複雜度 $O(n)$。詳細解釋見 http://leetcode.com/2011/11/longest-palindromic-substring-part-ii.html 。

備忘錄法

```
// LeetCode, Longest Palindromic Substring
// 備忘錄法，會超時
// 時間複雜度 O(n^2)，空間複雜度 O(n^2)
typedef string::const_iterator Iterator;

namespace std {
template<>
struct hash<pair<Iterator, Iterator>> {
    size_t operator()(pair<Iterator, Iterator> const& p) const {
        return ((size_t) &(*p.first)) ^ ((size_t) &(*p.second));
    }
};
}

class Solution {
public:
    string longestPalindrome(string const& s) {
        cache.clear();
        return cachedLongestPalindrome(s.begin(), s.end());
    }

private:
    unordered_map<pair<Iterator, Iterator>, string> cache;

    string longestPalindrome(Iterator first, Iterator last) {
        size_t length = distance(first, last);

        if (length < 2) return string(first, last);

        auto s = cachedLongestPalindrome(next(first), prev(last));

        if (s.length() == length - 2 && *first == *prev(last))
            return string(first, last);

        auto s1 = cachedLongestPalindrome(next(first), last);
        auto s2 = cachedLongestPalindrome(first, prev(last));

        // return max(s, s1, s2)
        if (s.size() > s1.size()) return s.size() > s2.size() ? s : s2;
        else return s1.size() > s2.size() ? s1 : s2;
    }

    string cachedLongestPalindrome(Iterator first, Iterator last) {
        auto key = make_pair(first, last);
        auto pos = cache.find(key);

        if (pos != cache.end()) return pos->second;
```

```
        else return cache[key] = longestPalindrome(first, last);
    }
};
```

動規

```
// LeetCode, Longest Palindromic Substring
// 動規, 時間複雜度 O(n^2), 空間複雜度 O(n^2)
class Solution {
public:
    string longestPalindrome(const string& s) {
        const int n = s.size();
        bool f[n][n];
        fill_n(&f[0][0], n * n, false);
        // 用 vector 會超時
        //vector<vector<bool> > f(n, vector<bool>(n, false));
        size_t max_len = 1, start = 0;  // 最長迴文子串的長度, 起點

        for (size_t i = 0; i < s.size(); i++) {
            f[i][i] = true;
            for (size_t j = 0; j < i; j++) {  // [j, i]
                f[j][i] = (s[j] == s[i] && (i - j < 2 || f[j + 1][i - 1]));
                if (f[j][i] && max_len < (i - j + 1)) {
                    max_len = i - j + 1;
                    start = j;
                }
            }
        }
        return s.substr(start, max_len);
    }
};
```

Manacher's Algorithm

```
// LeetCode, Longest Palindromic Substring
// Manacher's Algorithm
// 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    // Transform S into T.
    // For example, S = "abba", T = "^#a#b#b#a#$".
    // ^ and $ signs are sentinels appended to each end to avoid bounds checking
    string preProcess(const string& s) {
        int n = s.length();
        if (n == 0) return "^$";

        string ret = "^";
        for (int i = 0; i < n; i++) ret += "#" + s.substr(i, 1);

        ret += "#$";
        return ret;
    }
```

```
    string longestPalindrome(string s) {
        string T = preProcess(s);
        const int n = T.length();
        // 以 T[i] 為中心, 向左/右擴張的長度, 不包含 T[i] 自己,
        // 因此 P[i] 是源字符串中迴文串的長度
        int P[n];
        int C = 0, R = 0;

        for (int i = 1; i < n - 1; i++) {
            int i_mirror = 2 * C - i; // equals to i' = C - (i-C)

            P[i] = (R > i) ? min(R - i, P[i_mirror]) : 0;

            // Attempt to expand palindrome centered at i
            while (T[i + 1 + P[i]] == T[i - 1 - P[i]])
                P[i]++;

            // If palindrome centered at i expand past R,
            // adjust center based on expanded palindrome.
            if (i + P[i] > R) {
                C = i;
                R = i + P[i];
            }
        }

        // Find the maximum element in P.
        int max_len = 0;
        int center_index = 0;
        for (int i = 1; i < n - 1; i++) {
            if (P[i] > max_len) {
                max_len = P[i];
                center_index = i;
            }
        }

        return s.substr((center_index - 1 - max_len) / 2, max_len);
    }
};
```

相關題目

- 無

3.6 Regular Expression Matching

描述

Implement regular expression matching with support for `'.'` and `'*'`.

`'.'` Matches any single character. `'*'` Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

The function prototype should be:

```
bool isMatch(const char *s, const char *p)
```

Some examples:

```
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "a*") → true
isMatch("aa", ".*") → true
isMatch("ab", ".*") → true
isMatch("aab", "c*a*b") → true
```

分析

這是一道很有挑戰的題。

遞歸版

```
// LeetCode, Regular Expression Matching
// 遞歸版, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    bool isMatch(const string& s, const string& p) {
        return isMatch(s.c_str(), p.c_str());
    }
private:
    bool isMatch(const char *s, const char *p) {
        if (*p == '\0') return *s == '\0';

        // next char is not '*', then must match current character
        if (*(p + 1) != '*') {
            if (*p == *s || (*p == '.' && *s != '\0'))
                return isMatch(s + 1, p + 1);
            else
                return false;
        } else { // next char is '*'
            while (*p == *s || (*p == '.' && *s != '\0')) {
                if (isMatch(s, p + 2))
                    return true;
                s++;
            }
            return isMatch(s, p + 2);
        }
    }
};
```

迭代版

相關題目

- Wildcard Matching, 見 §3.7

3.7 Wildcard Matching

描述

Implement wildcard pattern matching with support for `'?'` and `'*'`.

`'?'` Matches any single character. `'*'` Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

The function prototype should be:

```
bool isMatch(const char *s, const char *p)
```

Some examples:

```
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "*") → true
isMatch("aa", "a*") → true
isMatch("ab", "?*") → true
isMatch("aab", "c*a*b") → false
```

分析

跟上一題很類似。

主要是`'*'` 的匹配問題。`p` 每遇到一個`'*'`，就保留住當前`'*'` 的座標和 `s` 的座標，然後 `s` 從前往後掃描，如果不成功，則 `s++`，重新掃描。

遞歸版

```
// LeetCode, Wildcard Matching
// 遞歸版，會超時，用於幫助理解題意
// 時間複雜度 O(n!*m!)，空間複雜度 O(n)
class Solution {
public:
    bool isMatch(const string& s, const string& p) {
        return isMatch(s.c_str(), p.c_str());
    }
private:
    bool isMatch(const char *s, const char *p) {
        if (*p == '*') {
            while (*p == '*') ++p;  //skip continuous '*'
            if (*p == '\0') return true;
            while (*s != '\0' && !isMatch(s, p)) ++s;

            return *s != '\0';
```

```
        }
        else if (*p == '\0' || *s == '\0') return *p == *s;
        else if (*p == *s || *p == '?') return isMatch(++s, ++p);
        else return false;
    }
};
```

迭代版

```
// LeetCode, Wildcard Matching
// 迭代版, 時間複雜度 O(n*m), 空間複雜度 O(1)
class Solution {
public:
    bool isMatch(const string& s, const string& p) {
        return isMatch(s.c_str(), p.c_str());
    }
private:
    bool isMatch(const char *s, const char *p) {
        bool star = false;
        const char *str, *ptr;
        for (str = s, ptr = p; *str != '\0'; str++, ptr++) {
            switch (*ptr) {
            case '?':
                break;
            case '*':
                star = true;
                s = str, p = ptr;
                while (*p == '*') p++;  //skip continuous '*'
                if (*p == '\0') return true;
                str = s - 1;
                ptr = p - 1;
                break;
            default:
                if (*str != *ptr) {
                    // 如果前面沒有'*', 則匹配不成功
                    if (!star) return false;
                    s++;
                    str = s - 1;
                    ptr = p - 1;
                }
            }
        }
        while (*ptr == '*') ptr++;
        return (*ptr == '\0');
    }
};
```

相關題目

- Regular Expression Matching, 見 §3.6

3.8 Longest Common Prefix

描述

Write a function to find the longest common prefix string amongst an array of strings.

分析

從位置 0 開始，對每一個位置比較所有字符串，直到遇到一個不匹配。

縱向掃描

```
// LeetCode, Longest Common Prefix
// 縱向掃描, 從位置 0 開始, 對每一個位置比較所有字符串, 直到遇到一個不匹配
// 時間複雜度 O(n1+n2+...)
// @author 周倩 (http://weibo.com/zhouditty)
class Solution {
public:
    string longestCommonPrefix(vector<string> &strs) {
        if (strs.empty()) return "";

        for (int idx = 0; idx < strs[0].size(); ++idx) { // 縱向掃描
            for (int i = 1; i < strs.size(); ++i) {
                if (strs[i][idx] != strs[0][idx]) return strs[0].substr(0,idx);
            }
        }
        return strs[0];
    }
};
```

橫向掃描

```
// LeetCode, Longest Common Prefix
// 橫向掃描, 每個字符串與第 0 個字符串, 從左到右比較, 直到遇到一個不匹配,
// 然後繼續下一個字符串
// 時間複雜度 O(n1+n2+...)
class Solution {
public:
    string longestCommonPrefix(vector<string> &strs) {
        if (strs.empty()) return "";

        int right_most = strs[0].size() - 1;
        for (size_t i = 1; i < strs.size(); i++)
            for (int j = 0; j <= right_most; j++)
                if (strs[i][j] != strs[0][j])  // 不會越界, 請參考 string::[] 的文檔
                    right_most = j - 1;

        return strs[0].substr(0, right_most + 1);
    }
};
```

相關題目

- 無

3.9 Valid Number

描述

Validate if a given string is numeric.

Some examples:

```
"0" => true
" 0.1 " => true
"abc" => false
"1 a" => false
"2e10" => true
```

Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

分析

細節實現題。

本題的功能與標準庫中的 `strtod()` 功能類似。

有限自動機

```
// LeetCode, Valid Number
// @author 龔陸安 (http://weibo.com/luangong)
// finite automata, 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    bool isNumber(const string& s) {
        enum InputType {
            INVALID,    // 0
            SPACE,      // 1
            SIGN,       // 2
            DIGIT,      // 3
            DOT,        // 4
            EXPONENT,   // 5
            NUM_INPUTS  // 6
        };
        const int transitionTable[][NUM_INPUTS] = {
                -1, 0, 3, 1, 2, -1, // next states for state 0
                -1, 8, -1, 1, 4, 5,     // next states for state 1
                -1, -1, -1, 4, -1, -1,     // next states for state 2
                -1, -1, -1, 1, 2, -1,     // next states for state 3
                -1, 8, -1, 4, -1, 5,     // next states for state 4
                -1, -1, 6, 7, -1, -1,     // next states for state 5
                -1, -1, -1, 7, -1, -1,     // next states for state 6
                -1, 8, -1, 7, -1, -1,     // next states for state 7
```

```
                -1, 8, -1, -1, -1, -1,     // next states for state 8
                };

        int state = 0;
        for (auto ch : s) {
            InputType inputType = INVALID;
            if (isspace(ch))
                inputType = SPACE;
            else if (ch == '+' || ch == '-')
                inputType = SIGN;
            else if (isdigit(ch))
                inputType = DIGIT;
            else if (ch == '.')
                inputType = DOT;
            else if (ch == 'e' || ch == 'E')
                inputType = EXPONENT;

            // Get next state from current state and input symbol
            state = transitionTable[state][inputType];

            // Invalid input
            if (state == -1) return false;
        }
        // If the current state belongs to one of the accepting (final) states,
        // then the number is valid
        return state == 1 || state == 4 || state == 7 || state == 8;

    }
};
```

使用 strtod()

```
// LeetCode, Valid Number
// @author 連城 (http://weibo.com/lianchengzju)
// 偷懶, 直接用 strtod(), 時間複雜度 O(n)
class Solution {
public:
    bool isNumber (const string& s) {
        return isNumber(s.c_str());
    }
private:
    bool isNumber (char const* s) {
        char* endptr;
        strtod (s, &endptr);

        if (endptr == s) return false;

        for (; *endptr; ++endptr)
            if (!isspace (*endptr)) return false;

        return true;
    }
```

```
};
```

相關題目

- 無

3.10 Integer to Roman

描述

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

分析

無

代碼

```
// LeetCode, Integer to Roman
// 時間複雜度 O(num), 空間複雜度 O(1)
class Solution {
public:
    string intToRoman(int num) {
        const int radix[] = {1000, 900, 500, 400, 100, 90,
                50, 40, 10, 9, 5, 4, 1};
        const string symbol[] = {"M", "CM", "D", "CD", "C", "XC",
                "L", "XL", "X", "IX", "V", "IV", "I"};

        string roman;
        for (size_t i = 0; num > 0; ++i) {
            int count = num / radix[i];
            num %= radix[i];
            for (; count > 0; --count) roman += symbol[i];
        }
        return roman;
    }
};
```

相關題目

- Roman to Integer, 見 §3.11

3.11 Roman to Integer

描述

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

分析

從前往後掃描，用一個臨時變量記錄分段數字。

如果當前比前一個大，說明這一段的值應該是當前這個值減去上一個值。比如 `IV = 5 - 1`；否則，將當前值加入到結果中，然後開始下一段記錄。比如 `VI = 5 + 1, II=1+1`

代碼

```
// LeetCode, Roman to Integer
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    inline int map(const char c) {
        switch (c) {
        case 'I': return 1;
        case 'V': return 5;
        case 'X': return 10;
        case 'L': return 50;
        case 'C': return 100;
        case 'D': return 500;
        case 'M': return 1000;
        default: return 0;
        }
    }

    int romanToInt(const string& s) {
        int result = 0;
        for (size_t i = 0; i < s.size(); i++) {
            if (i > 0 && map(s[i]) > map(s[i - 1])) {
                result += (map(s[i]) - 2 * map(s[i - 1]));
            } else {
                result += map(s[i]);
            }
        }
        return result;
    }
};
```

相關題目

- Integer to Roman, 見 §3.10

3.12 Count and Say

描述

The count-and-say sequence is the sequence of integers beginning as follows:

`1, 11, 21, 1211, 111221, ...`

`1` is read off as `"one 1"` or `11`.

11 is read off as "two 1s" or 21.

21 is read off as "one 2", then "one 1" or 1211.

Given an integer n, generate the nth sequence.

Note: The sequence of integers will be represented as a string.

分析

模擬。

代碼

```
// LeetCode, Count and Say
// @author 連城 (http://weibo.com/lianchengzju)
// 時間複雜度 O(n^2), 空間複雜度 O(n)
class Solution {
public:
    string countAndSay(int n) {
        string s("1");

        while (--n)
            s = getNext(s);

        return s;
    }

    string getNext(const string &s) {
        stringstream ss;

        for (auto i = s.begin(); i != s.end(); ) {
            auto j = find_if(i, s.end(), bind1st(not_equal_to<char>(), *i));
            ss << distance(i, j) << *i;
            i = j;
        }

        return ss.str();
    }
};
```

相關題目

- 無

3.13 Anagrams

描述

Given an array of strings, return all groups of strings that are anagrams.

Note: All inputs will be in lower-case.

分析

Anagram（迴文構詞法）是指打亂字母順序從而得到新的單詞，比如 "dormitory" 打亂字母順序會變成 "dirty room"，"tea" 會變成"eat"。

迴文構詞法有一個特點: 單詞裏的字母的種類和數目沒有改變, 只是改變了字母的排列順序。因此，將幾個單詞按照字母順序排序後，若它們相等，則它們屬於同一組 anagrams 。

代碼

```
// LeetCode, Anagrams
// 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    vector<string> anagrams(vector<string> &strs) {
        unordered_map<string, vector<string> > group;
        for (const auto &s : strs) {
            string key = s;
            sort(key.begin(), key.end());
            group[key].push_back(s);
        }

        vector<string> result;
        for (auto it = group.cbegin(); it != group.cend(); it++) {
            if (it->second.size() > 1)
                result.insert(result.end(), it->second.begin(), it->second.end());
        }
        return result;
    }
};
```

相關題目

- 無

3.14 Simplify Path

描述

Given an absolute path for a file (Unix-style), simplify it.

For example,

path = "/home/", => "/home"

path = "/a/./b/../../c/", => "/c"

Corner Cases:

- Did you consider the case where path = "/../"? In this case, you should return "/".
- Another corner case is the path might contain multiple slashes '/' together, such as "/home//foo/". In this case, you should ignore redundant slashes and return "/home/foo".

分析

很有實際價值的題目。

代碼

```
// LeetCode, Simplify Path
// 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    string simplifyPath(const string& path) {
        vector<string> dirs; // 當做栈

        for (auto i = path.begin(); i != path.end();) {
            ++i;

            auto j = find(i, path.end(), '/');
            auto dir = string(i, j);

            if (!dir.empty() && dir != ".") {// 當有連續 '///' 時, dir 為空
                if (dir == "..") {
                    if (!dirs.empty())
                        dirs.pop_back();
                } else
                    dirs.push_back(dir);
            }

            i = j;
        }

        stringstream out;
        if (dirs.empty()) {
            out << "/";
        } else {
            for (auto dir : dirs)
                out << '/' << dir;
        }

        return out.str();
    }
};
```

相關題目

- 無

3.15 Length of Last Word

描述

Given a string s consists of upper/lower-case alphabets and empty space characters `' '`, return the length of last word in the string.

If the last word does not exist, return 0.

Note: A word is defined as a character sequence consists of non-space characters only.

For example, Given `s = "Hello World"`, return 5.

分析

細節實現題。

用 STL

```
// LeetCode, Length of Last Word
// 偷懶, 用 STL
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int lengthOfLastWord(const string& s) {
        auto first = find_if(s.rbegin(), s.rend(), ::isalpha);
        auto last = find_if_not(first, s.rend(), ::isalpha);
        return distance(first, last);
    }
};
```

順序掃描

```
// LeetCode, Length of Last Word
// 順序掃描, 記錄每個 word 的長度
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int lengthOfLastWord(const string& s) {
        return lengthOfLastWord(s.c_str());
    }
private:
    int lengthOfLastWord(const char *s) {
        int len = 0;
        while (*s) {
            if (*s++ != ' ')
                ++len;
            else if (*s && *s != ' ')
                len = 0;
        }
        return len;
```

```
    }
};
```

相關題目

- 無

第 4 章
棧和隊列

4.1 棧

4.1.1 Valid Parentheses

描述

Given a string containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

The brackets must close in the correct order, `"()"` and `"()[]"` are all valid but `"(]"` and `"([)]"` are not.

分析

無

代碼

```
// LeetCode, Valid Parentheses
// 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    bool isValid (string const& s) {
        string left = "([{";
        string right = ")]}";
        stack<char> stk;

        for (auto c : s) {
            if (left.find(c) != string::npos) {
                stk.push (c);
            } else {
                if (stk.empty () || stk.top () != left[right.find (c)])
                    return false;
                else
                    stk.pop ();
            }
        }
```

```
        return stk.empty();
    }
};
```

相關題目

- Generate Parentheses, 見 §10.9
- Longest Valid Parentheses, 見 §4.1.2

4.1.2 Longest Valid Parentheses

描述

Given a string containing just the characters `'('` and `')'`, find the length of the longest valid (well-formed) parentheses substring.

For `"(()"`, the longest valid parentheses substring is `"()"`, which has length = 2.

Another example is `")()())"`, where the longest valid parentheses substring is `"()()"`, which has length = 4.

分析

無

使用棧

```
// LeetCode, Longest Valid Parenthese
// 使用棧, 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    int longestValidParentheses(const string& s) {
        int max_len = 0, last = -1; // the position of the last ')'
        stack<int> lefts;  // keep track of the positions of non-matching '('s

        for (int i = 0; i < s.size(); ++i) {
            if (s[i] =='(') {
                lefts.push(i);
            } else {
                if (lefts.empty()) {
                    // no matching left
                    last = i;
                } else {
                    // find a matching pair
                    lefts.pop();
                    if (lefts.empty()) {
                        max_len = max(max_len, i-last);
                    } else {
                        max_len = max(max_len, i-lefts.top());
```

```
                    }
                }
            }
        }
        return max_len;
    }
};
```

Dynamic Programming, One Pass

```
// LeetCode, Longest Valid Parenthese
// 時間複雜度 O(n), 空間複雜度 O(n)
// @author 一隻傑森 (http://weibo.com/wjson)
class Solution {
public:
    int longestValidParentheses(const string& s) {
        vector<int> f(s.size(), 0);
        int ret = 0;
        for (int i = s.size() - 2; i >= 0; --i) {
            int match = i + f[i + 1] + 1;
            // case: "((...))"
            if (s[i] == '(' && match < s.size() && s[match] == ')') {
                f[i] = f[i + 1] + 2;
                // if a valid sequence exist afterwards "((...))()"
                if (match + 1 < s.size()) f[i] += f[match + 1];
            }
            ret = max(ret, f[i]);
        }
        return ret;
    }
};
```

兩遍掃描

```
// LeetCode, Longest Valid Parenthese
// 兩遍掃描, 時間複雜度 O(n), 空間複雜度 O(1)
// @author 曹鵬 (http://weibo.com/cpcs)
class Solution {
public:
    int longestValidParentheses(const string& s) {
        int answer = 0, depth = 0, start = -1;
        for (int i = 0; i < s.size(); ++i) {
            if (s[i] == '(') {
                ++depth;
            } else {
                --depth;
                if (depth < 0) {
                    start = i;
                    depth = 0;
                } else if (depth == 0) {
                    answer = max(answer, i - start);
                }
```

```
            }
        }

        depth = 0;
        start = s.size();
        for (int i = s.size() - 1; i >= 0; --i) {
            if (s[i] == ')') {
                ++depth;
            } else {
                --depth;
                if (depth < 0) {
                    start = i;
                    depth = 0;
                } else if (depth == 0) {
                    answer = max(answer, start - i);
                }
            }
        }
        return answer;
    }
};
```

相關題目

- Valid Parentheses, 見 §4.1.1
- Generate Parentheses, 見 §10.9

4.1.3 Largest Rectangle in Histogram

描述

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

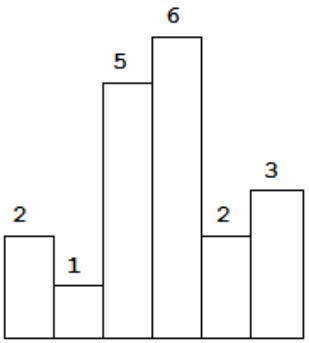


图 4-1 Above is a histogram where width of each bar is 1, given height = `[2,1,5,6,2,3]`.

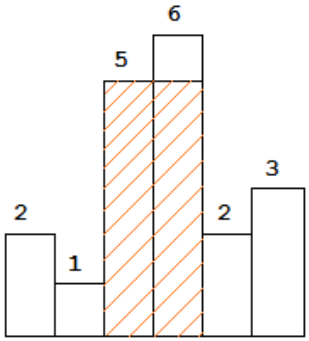


图 4-2 The largest rectangle is shown in the shaded area, which has area = 10 unit.

For example, Given height = `[2,1,5,6,2,3]`, return 10.

分析

簡單的，類似於 Container With Most Water(§12.6)，對每個柱子，左右擴展，直到碰到比自己矮的，計算這個矩形的面積，用一個變量記錄最大的面積，複雜度 $O(n^2)$，會超時。

如圖 §4-2 所示，從左到右處理直方，當 $i = 4$ 時，小於當前棧頂 (即直方 3)，對於直方 3，無論後面還是前面的直方，都不可能得到比目前棧頂元素更高的高度了，處理掉直方 3 (計算從直方 3 到直方 4 之間的矩形的面積，然後從棧裏彈出)；對於直方 2 也是如此；直到碰到比直方 4 更矮的直方 1。

這就意味着，可以維護一個遞增的棧，每次比較棧頂與當前元素。如果當前元素大於棧頂元素，則入棧，否則合併現有棧，直至棧頂元素小於當前元素。結尾時入棧元素 0，重複合併一次。

代碼

```
// LeetCode, Largest Rectangle in Histogram
// 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    int largestRectangleArea(vector<int> &height) {
        stack<int> s;
        height.push_back(0);
        int result = 0;
        for (int i = 0; i < height.size(); ) {
            if (s.empty() || height[i] > height[s.top()])
                s.push(i++);
            else {
                int tmp = s.top();
                s.pop();
                result = max(result,
                        height[tmp] * (s.empty() ? i : i - s.top() - 1));
            }
```

```
        }
        return result;
    }
};
```

相關題目

- Trapping Rain Water, 見 §2.1.16
- Container With Most Water, 見 §12.6

4.1.4 Evaluate Reverse Polish Notation

描述

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are `+, -, *, /`. Each operand may be an integer or another expression.

Some examples:

```
["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6
```

分析

無

遞歸版

```
// LeetCode, Evaluate Reverse Polish Notation
// 遞歸, 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
    int evalRPN(vector<string> &tokens) {
        int x, y;
        auto token = tokens.back();  tokens.pop_back();
        if (is_operator(token))  {
            y = evalRPN(tokens);
            x = evalRPN(tokens);
            if (token[0] == '+')       x += y;
            else if (token[0] == '-')  x -= y;
            else if (token[0] == '*')  x *= y;
            else                       x /= y;
        } else  {
            size_t i;
            x = stoi(token, &i);
        }
        return x;
    }
private:
    bool is_operator(const string &op) {
```

```
        return op.size() == 1 && string("+-*/").find(op) != string::npos;
    }
};
```

迭代版

```
// LeetCode, Max Points on a Line
// 迭代, 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
    int evalRPN(vector<string> &tokens) {
        stack<string> s;
        for (auto token : tokens) {
            if (!is_operator(token)) {
                s.push(token);
            } else {
                int y = stoi(s.top());
                s.pop();
                int x = stoi(s.top());
                s.pop();
                if (token[0] == '+')       x += y;
                else if (token[0] == '-')  x -= y;
                else if (token[0] == '*')  x *= y;
                else                       x /= y;
                s.push(to_string(x));
            }
        }
        return stoi(s.top());
    }
private:
    bool is_operator(const string &op) {
        return op.size() == 1 && string("+-*/").find(op) != string::npos;
    }
};
```

相關題目

- 無

4.2 隊列

第 5 章
樹

LeetCode 上二叉樹的節點定義如下：

```
// 樹的節點
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) { }
};
```

5.1 二叉樹的遍歷

樹的遍歷有兩類：深度優先遍歷和寬度優先遍歷。深度優先遍歷又可分為兩種：先根（次序）遍歷和後根（次序）遍歷。

樹的先根遍歷是：先訪問樹的根結點，然後依次先根遍歷根的各棵子樹。樹的先跟遍歷的結果與對應二叉樹（孩子兄弟表示法）的先序遍歷的結果相同。

樹的後根遍歷是：先依次後根遍歷樹根的各棵子樹，然後訪問根結點。樹的後跟遍歷的結果與對應二叉樹的中序遍歷的結果相同。

二叉樹的先根遍歷有：**先序遍歷** (root->left->right)，root->right->left；後根遍歷有：**後序遍歷** (left->right->root)，right->left->root；二叉樹還有個一般的樹沒有的遍歷次序，**中序遍歷** (left->root->right)。

5.1.1 Binary Tree Preorder Traversal

描述

Given a binary tree, return the preorder traversal of its nodes' values.

For example: Given binary tree `{1,#,2,3}`,

```
  1
   \
    2
   /
  3
```

return

$$1, 2, 3$$

.

Note: Recursive solution is trivial, could you do it iteratively?

分析

用棧或者 Morris 遍歷。

棧

```
// LeetCode, Binary Tree Preorder Traversal
// 使用栈, 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> result;
        stack<const TreeNode *> s;
        if (root != nullptr) s.push(root);

        while (!s.empty()) {
            const TreeNode *p = s.top();
            s.pop();
            result.push_back(p->val);

            if (p->right != nullptr) s.push(p->right);
            if (p->left != nullptr) s.push(p->left);
        }
        return result;
    }
};
```

Morris 先序遍歷

```
// LeetCode, Binary Tree Preorder Traversal
// Morris 先序遍歷, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> result;
        TreeNode *cur = root, *prev = nullptr;

        while (cur != nullptr) {
            if (cur->left == nullptr) {
                result.push_back(cur->val);
                prev = cur; /* cur 剛剛被訪問過 */
                cur = cur->right;
            } else {
                /* 查找前驅 */
                TreeNode *node = cur->left;
                while (node->right != nullptr && node->right != cur)
                    node = node->right;

                if (node->right == nullptr) { /* 還沒線索化, 則建立線索 */
```

```
                    result.push_back(cur->val); /* 僅這一行的位置與中序不同 */
                    node->right = cur;
                    prev = cur; /* cur 剛剛被訪問過 */
                    cur = cur->left;
                } else {    /* 已經線索化，則刪除線索  */
                    node->right = nullptr;
                    /* prev = cur; 不能有這句，cur 已經被訪問 */
                    cur = cur->right;
                }
            }
        }
        return result;
    }
};
```

相關題目

- Binary Tree Inorder Traversal，見 §5.1.2
- Binary Tree Postorder Traversal，見 §5.1.3
- Recover Binary Search Tree，見 §5.1.7

5.1.2 Binary Tree Inorder Traversal

描述

Given a binary tree, return the inorder traversal of its nodes' values.

For example: Given binary tree `{1,#,2,3}`,

```
1
 \
  2
 /
3
```

return

$$1, 3, 2$$

.

Note: Recursive solution is trivial, could you do it iteratively?

分析

用棧或者 Morris 遍歷。

棧

```
// LeetCode, Binary Tree Inorder Traversal
// 使用棧, 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> result;
        stack<const TreeNode *> s;
        const TreeNode *p = root;

        while (!s.empty() || p != nullptr) {
            if (p != nullptr) {
                s.push(p);
                p = p->left;
            } else {
                p = s.top();
                s.pop();
                result.push_back(p->val);
                p = p->right;
            }
        }
        return result;
    }
};
```

Morris 中序遍歷

```
// LeetCode, Binary Tree Inorder Traversal
// Morris 中序遍歷, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> result;
        TreeNode *cur = root, *prev = nullptr;

        while (cur != nullptr) {
            if (cur->left == nullptr) {
                result.push_back(cur->val);
                prev = cur;
                cur = cur->right;
            } else {
                /* 查找前驅 */
                TreeNode *node = cur->left;
                while (node->right != nullptr && node->right != cur)
                    node = node->right;

                if (node->right == nullptr) { /* 還沒線索化, 則建立線索 */
                    node->right = cur;
                    /* prev = cur; 不能有這句, cur 還沒有被訪問 */
                    cur = cur->left;
                } else {    /* 已經線索化, 則訪問節點, 並刪除線索  */
                    result.push_back(cur->val);
```

```
                    node->right = nullptr;
                    prev = cur;
                    cur = cur->right;
                }
            }
        }
        return result;
    }
};
```

相關題目

- Binary Tree Preorder Traversal，見 §5.1.1
- Binary Tree Postorder Traversal，見 §5.1.3
- Recover Binary Search Tree，見 §5.1.7

5.1.3 Binary Tree Postorder Traversal

描述

Given a binary tree, return the postorder traversal of its nodes' values.

For example: Given binary tree `{1,#,2,3}`,

```
1
 \
  2
 /
3
```

return

$$3, 2, 1$$

.

Note: Recursive solution is trivial, could you do it iteratively?

分析

用棧或者 Morris 遍歷。

棧

```
// LeetCode, Binary Tree Postorder Traversal
// 使用栈，時間複雜度 O(n)，空間複雜度 O(n)
class Solution {
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> result;
        stack<const TreeNode *> s;
```

```
        /* p, 正在訪問的結點, q, 剛剛訪問過的結點*/
        const TreeNode *p = root, *q = nullptr;

        do {
            while (p != nullptr) { /* 往左下走*/
                s.push(p);
                p = p->left;
            }
            q = nullptr;
            while (!s.empty()) {
                p = s.top();
                s.pop();
                /* 右孩子不存在或已被訪問, 訪問之*/
                if (p->right == q) {
                    result.push_back(p->val);
                    q = p; /* 保存剛訪問過的結點*/
                } else {
                    /* 當前結點不能訪問, 需第二次進棧*/
                    s.push(p);
                    /* 先處理右子樹*/
                    p = p->right;
                    break;
                }
            }
        } while (!s.empty());

        return result;
    }
};
```

Morris 後序遍歷

```
// LeetCode, Binary Tree Postorder Traversal
// Morris 後序遍歷, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> result;
        TreeNode dummy(-1);
        TreeNode *cur, *prev = nullptr;
        std::function < void(const TreeNode*)> visit =
            [&result](const TreeNode *node){
            result.push_back(node->val);
        };

        dummy.left = root;
        cur = &dummy;
        while (cur != nullptr) {
            if (cur->left == nullptr) {
                prev = cur; /* 必須要有 */
                cur = cur->right;
            } else {
```

```
                TreeNode *node = cur->left;
                while (node->right != nullptr && node->right != cur)
                    node = node->right;

                if (node->right == nullptr) { /* 還沒線索化, 則建立線索 */
                    node->right = cur;
                    prev = cur; /* 必須要有 */
                    cur = cur->left;
                } else { /* 已經線索化, 則訪問節點, 並刪除線索  */
                    visit_reverse(cur->left, prev, visit);
                    prev->right = nullptr;
                    prev = cur; /* 必須要有 */
                    cur = cur->right;
                }
            }
        }
        return result;
    }
private:
    // 逆轉路徑
    static void reverse(TreeNode *from, TreeNode *to) {
        TreeNode *x = from, *y = from->right, *z;
        if (from == to) return;

        while (x != to) {
            z = y->right;
            y->right = x;
            x = y;
            y = z;
        }
    }

    // 訪問逆轉後的路徑上的所有結點
    static void visit_reverse(TreeNode* from, TreeNode *to,
                      std::function< void(const TreeNode*) >& visit) {
        TreeNode *p = to;
        reverse(from, to);

        while (true) {
            visit(p);
            if (p == from)
                break;
            p = p->right;
        }

        reverse(to, from);
    }
};
```

相關題目

- Binary Tree Preorder Traversal，見 §5.1.1

- Binary Tree Inorder Traversal，見 §5.1.2
- Recover Binary Search Tree，見 §5.1.7

5.1.4 Binary Tree Level Order Traversal

描述

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example: Given binary tree `{3,9,20,#,#,15,7}`,

```
  3
 / \
9  20
  /  \
 15   7
```

return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

分析

無

遞歸版

```
// LeetCode, Binary Tree Level Order Traversal
// 遞歸版, 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > levelOrder(TreeNode *root) {
        vector<vector<int>> result;
        traverse(root, 1, result);
        return result;
    }

    void traverse(TreeNode *root, size_t level, vector<vector<int>> &result) {
        if (!root) return;

        if (level > result.size())
            result.push_back(vector<int>());

        result[level-1].push_back(root->val);
        traverse(root->left, level+1, result);
        traverse(root->right, level+1, result);
```

```
    }
};
```

迭代版

```
// LeetCode, Binary Tree Level Order Traversal
// 迭代版, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    vector<vector<int> > levelOrder(TreeNode *root) {
        vector<vector<int> > result;
        queue<TreeNode*> current, next;

        if(root == nullptr) {
            return result;
        } else {
            current.push(root);
        }

        while (!current.empty()) {
            vector<int> level; // elments in one level
            while (!current.empty()) {
                TreeNode* node = current.front();
                current.pop();
                level.push_back(node->val);
                if (node->left != nullptr) next.push(node->left);
                if (node->right != nullptr) next.push(node->right);
            }
            result.push_back(level);
            swap(next, current);
        }
        return result;
    }
};
```

相關題目

- Binary Tree Level Order Traversal II，見 §5.1.5
- Binary Tree Zigzag Level Order Traversal，見 §5.1.6

5.1.5 Binary Tree Level Order Traversal II

描述

Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example: Given binary tree `{3,9,20,#,#,15,7}`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its bottom-up level order traversal as:

```
[
  [15,7]
  [9,20],
  [3],
]
```

分析

在上一題（見 §5.1.4）的基礎上，`reverse()` 一下即可。

遞歸版

```
// LeetCode, Binary Tree Level Order Traversal II
// 遞歸版, 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > levelOrderBottom(TreeNode *root) {
        vector<vector<int>> result;
        traverse(root, 1, result);
        std::reverse(result.begin(), result.end()); // 比上一題多此一行
        return result;
    }

    void traverse(TreeNode *root, size_t level, vector<vector<int>> &result) {
        if (!root) return;

        if (level > result.size())
            result.push_back(vector<int>());

        result[level-1].push_back(root->val);
        traverse(root->left, level+1, result);
        traverse(root->right, level+1, result);
    }
};
```

迭代版

```
// LeetCode, Binary Tree Level Order Traversal II
// 迭代版, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    vector<vector<int> > levelOrderBottom(TreeNode *root) {
        vector<vector<int> > result;
        if(root == nullptr) return result;
```

```
        queue<TreeNode*> current, next;
        vector<int> level; // elments in level level

        current.push(root);
        while (!current.empty()) {
            while (!current.empty()) {
                TreeNode* node = current.front();
                current.pop();
                level.push_back(node->val);
                if (node->left != nullptr) next.push(node->left);
                if (node->right != nullptr) next.push(node->right);
            }
            result.push_back(level);
            level.clear();
            swap(next, current);
        }
        reverse(result.begin(), result.end()); // 比上一題多此一行
        return result;
    }
};
```

相關題目

- Binary Tree Level Order Traversal，見 §5.1.4
- Binary Tree Zigzag Level Order Traversal，見 §5.1.6

5.1.6 Binary Tree Zigzag Level Order Traversal

描述

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example: Given binary tree `3,9,20,#,#,15,7`,

```
  3
 / \
9  20
  /  \
 15   7
```

return its zigzag level order traversal as:

```
[
  [3],
  [20,9],
  [15,7]
]
```

分析

廣度優先遍歷，用一個 bool 記錄是從左到右還是從右到左，每一層結束就翻轉一下。

遞歸版

```
// LeetCode, Binary Tree Zigzag Level Order Traversal
// 遞歸版，時間複雜度 O(n)，空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > zigzagLevelOrder(TreeNode *root) {
        vector<vector<int>> result;
        traverse(root, 1, result, true);
        return result;
    }

    void traverse(TreeNode *root, size_t level, vector<vector<int>> &result,
            bool left_to_right) {
        if (!root) return;

        if (level > result.size())
            result.push_back(vector<int>());

        if (left_to_right)
            result[level-1].push_back(root->val);
        else
            result[level-1].insert(result[level-1].begin(), root->val);

        traverse(root->left, level+1, result, !left_to_right);
        traverse(root->right, level+1, result, !left_to_right);
    }
};
```

迭代版

```
// LeetCode, Binary Tree Zigzag Level Order Traversal
// 廣度優先遍歷，用一個 bool 記錄是從左到右還是從右到左，每一層結束就翻轉一下。
// 迭代版，時間複雜度 O(n)，空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > zigzagLevelOrder(TreeNode *root) {
        vector<vector<int> > result;
        queue<TreeNode*> current, next;
        bool left_to_right = true;

        if(root == nullptr) {
            return result;
        } else {
            current.push(root);
        }

        while (!current.empty()) {
```

```
            vector<int> level; // elments in one level
            while (!current.empty()) {
                TreeNode* node = current.front();
                current.pop();
                level.push_back(node->val);
                if (node->left != nullptr) next.push(node->left);
                if (node->right != nullptr) next.push(node->right);
            }
            if (!left_to_right) reverse(level.begin(), level.end());
            result.push_back(level);
            left_to_right = !left_to_right;
            swap(next, current);
        }
        return result;
    }
};
```

相關題目

- Binary Tree Level Order Traversal，見 §5.1.4
- Binary Tree Level Order Traversal II，見 §5.1.5

5.1.7 Recover Binary Search Tree

描述

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note: A solution using $O(n)$ space is pretty straight forward. Could you devise a constant space solution?

分析

$O(n)$ 空間的解法是，開一個指針數組，中序遍歷，將節點指針依次存放到數組裏，然後尋找兩處逆向的位置，先從前往後找第一個逆序的位置，然後從後往前找第二個逆序的位置，交換這兩個指針的值。

中序遍歷一般需要用到棧，空間也是 $O(n)$ 的，如何才能不使用棧？Morris 中序遍歷。

代碼

```
// LeetCode, Recover Binary Search Tree
// Morris 中序遍歷, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    void recoverTree(TreeNode* root) {
        pair<TreeNode*, TreeNode*> broken;
        TreeNode* prev = nullptr;
```

```
        TreeNode* cur = root;

        while (cur != nullptr) {
            if (cur->left == nullptr) {
                detect(broken, prev, cur);
                prev = cur;
                cur = cur->right;
            } else {
                auto node = cur->left;

                while (node->right != nullptr && node->right != cur)
                    node = node->right;

                if (node->right == nullptr) {
                    node->right = cur;
                    //prev = cur; 不能有這句！因為 cur 還沒有被訪問
                    cur = cur->left;
                } else {
                    detect(broken, prev, cur);
                    node->right = nullptr;
                    prev = cur;
                    cur = cur->right;
                }
            }
        }

        swap(broken.first->val, broken.second->val);
    }

    void detect(pair<TreeNode*, TreeNode*>& broken, TreeNode* prev,
            TreeNode* current) {
        if (prev != nullptr && prev->val > current->val) {
            if (broken.first == nullptr) {
                broken.first = prev;
            } //不能用 else，例如 {0,1}，會導致最後 swap 時 second 為 nullptr,
              //會 Runtime Error
            broken.second = current;
        }
    }
};
```

相關題目

- Binary Tree Inorder Traversal，見 §5.1.2

5.1.8 Same Tree

描述

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

分析

無

遞歸版

遞歸版

```
// LeetCode, Same Tree
// 遞歸版, 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
    bool isSameTree(TreeNode *p, TreeNode *q) {
        if (!p && !q) return true;   // 終止條件
        if (!p || !q) return false;  // 剪枝
        return p->val == q->val      // 三方合併
                && isSameTree(p->left, q->left)
                && isSameTree(p->right, q->right);
    }
};
```

迭代版

```
// LeetCode, Same Tree
// 迭代版, 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
    bool isSameTree(TreeNode *p, TreeNode *q) {
        stack<TreeNode*> s;
        s.push(p);
        s.push(q);

        while(!s.empty()) {
            p = s.top(); s.pop();
            q = s.top(); s.pop();

            if (!p && !q) continue;
            if (!p || !q) return false;
            if (p->val != q->val) return false;

            s.push(p->left);
            s.push(q->left);

            s.push(p->right);
            s.push(q->right);
        }
        return true;
    }
};
```

相關題目

- Symmetric Tree，見 §5.1.9

5.1.9 Symmetric Tree

描述

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

分析

無

遞歸版

```
// LeetCode, Symmetric Tree
// 遞歸版, 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
    bool isSymmetric(TreeNode *root) {
        if (root == nullptr) return true;
        return isSymmetric(root->left, root->right);
    }
    bool isSymmetric(TreeNode *p, TreeNode *q) {
        if (p == nullptr && q == nullptr) return true;   // 終止條件
        if (p == nullptr || q == nullptr) return false;  // 終止條件
        return p->val == q->val      // 三方合併
                && isSymmetric(p->left, q->right)
                && isSymmetric(p->right, q->left);
    }
};
```

迭代版

```
// LeetCode, Symmetric Tree
// 迭代版, 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
    bool isSymmetric (TreeNode* root) {
        if (!root) return true;

        stack<TreeNode*> s;
        s.push(root->left);
        s.push(root->right);

        while (!s.empty ()) {
            auto p = s.top (); s.pop();
```

```
            auto q = s.top (); s.pop();

            if (!p && !q) continue;
            if (!p || !q) return false;
            if (p->val != q->val) return false;

            s.push(p->left);
            s.push(q->right);

            s.push(p->right);
            s.push(q->left);
        }

        return true;
    }
};
```

相關題目

- Same Tree，見 §5.1.8

5.1.10 Balanced Binary Tree

描述

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

分析

無

代碼

```
// LeetCode, Balanced Binary Tree
// 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
    bool isBalanced (TreeNode* root) {
        return balancedHeight (root) >= 0;
    }

    /**
     * Returns the height of `root` if `root` is a balanced tree,
     * otherwise, returns `-1`.
     */
    int balancedHeight (TreeNode* root) {
```

```
        if (root == nullptr) return 0;  // 終止條件

        int left = balancedHeight (root->left);
        int right = balancedHeight (root->right);

        if (left < 0 || right < 0 || abs(left - right) > 1) return -1;  // 剪枝

        return max(left, right) + 1; // 三方合併
    }
};
```

相關題目

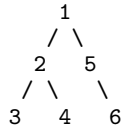
- 無

5.1.11 Flatten Binary Tree to Linked List

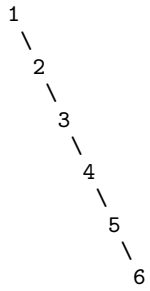
描述

Given a binary tree, flatten it to a linked list in-place.

For example, Given



The flattened tree should look like:



分析

無

遞歸版 1

```
// LeetCode, Flatten Binary Tree to Linked List
// 遞歸版 1, 時間複雜度 O(n), 空間複雜度 O(logn)
```

```
class Solution {
public:
    void flatten(TreeNode *root) {
        if (root == nullptr) return;  // 終止條件

        flatten(root->left);
        flatten(root->right);

        if (nullptr == root->left) return;

        // 三方合併，將左子樹所形成的鏈表插入到 root 和 root->right 之間
        TreeNode *p = root->left;
        while(p->right) p = p->right; //尋找左鏈表最後一個節點
        p->right = root->right;
        root->right = root->left;
        root->left = nullptr;
    }
};
```

遞歸版 2

```
// LeetCode, Flatten Binary Tree to Linked List
// 遞歸版 2
// @author 王順達 (http://weibo.com/u/1234984145)
// 時間複雜度 O(n)，空間複雜度 O(logn)
class Solution {
public:
    void flatten(TreeNode *root) {
        flatten(root, NULL);
    }
private:
    // 把 root 所代表樹變成鏈表後，tail 跟在該鏈表後面
    TreeNode *flatten(TreeNode *root, TreeNode *tail) {
        if (NULL == root) return tail;

        root->right = flatten(root->left, flatten(root->right, tail));
        root->left = NULL;
        return root;
    }
};
```

迭代版

```
// LeetCode, Flatten Binary Tree to Linked List
// 迭代版，時間複雜度 O(n)，空間複雜度 O(logn)
class Solution {
public:
    void flatten(TreeNode* root) {
        if (root == nullptr) return;

        stack<TreeNode*> s;
        s.push(root);
```

```
        while (!s.empty()) {
            auto p = s.top();
            s.pop();

            if (p->right)
                s.push(p->right);
            if (p->left)
                s.push(p->left);

            p->left = nullptr;
            if (!s.empty())
                p->right = s.top();
        }
    }
};
```

相關題目

- 無

5.1.12 Populating Next Right Pointers in Each Node II

描述

Follow up for problem ”Populating Next Right Pointers in Each Node”.

What if the given tree could be any binary tree? Would your previous solution still work?

Note: You may only use constant extra space.

For example, Given the following binary tree,

```
       1
     /  \
    2    3
   / \    \
  4   5    7
```

After calling your function, the tree should look like:

```
       1 -> NULL
     /  \
    2 -> 3 -> NULL
   / \    \
  4-> 5 -> 7 -> NULL
```

分析

要處理一個節點，可能需要最右邊的兄弟節點，首先想到用廣搜。但廣搜不是常數空間的，本題要求常數空間。

注意，這題的代碼原封不動，也可以解決 Populating Next Right Pointers in Each Node I.

遞歸版

```
// LeetCode, Populating Next Right Pointers in Each Node II
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    void connect(TreeLinkNode *root) {
        if (root == nullptr) return;

        TreeLinkNode dummy(-1);
        for (TreeLinkNode *curr = root, *prev = &dummy;
                curr; curr = curr->next) {
            if (curr->left != nullptr){
                prev->next = curr->left;
                prev = prev->next;
            }
            if (curr->right != nullptr){
                prev->next = curr->right;
                prev = prev->next;
            }
        }
        connect(dummy.next);
    }
};
```

迭代版

```
// LeetCode, Populating Next Right Pointers in Each Node II
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    void connect(TreeLinkNode *root) {
        while (root) {
            TreeLinkNode * next = nullptr; // the first node of next level
            TreeLinkNode * prev = nullptr; // previous node on the same level
            for (; root; root = root->next) {
                if (!next) next = root->left ? root->left : root->right;

                if (root->left) {
                    if (prev) prev->next = root->left;
                    prev = root->left;
                }
                if (root->right) {
                    if (prev) prev->next = root->right;
                    prev = root->right;
                }
            }
            root = next; // turn to next level
        }
    }
};
```

相關題目

- Populating Next Right Pointers in Each Node，見 §5.4.6

5.2 二叉樹的構建

5.2.1 Construct Binary Tree from Preorder and Inorder Traversal

描述

Given preorder and inorder traversal of a tree, construct the binary tree.

Note: You may assume that duplicates do not exist in the tree.

分析

無

代碼

```
// LeetCode, Construct Binary Tree from Preorder and Inorder Traversal
// 遞歸，時間複雜度 O(n)，空間複雜度 O(\logn)
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        return buildTree(begin(preorder), end(preorder),
                begin(inorder), end(inorder));
    }

    template<typename InputIterator>
    TreeNode* buildTree(InputIterator pre_first, InputIterator pre_last,
            InputIterator in_first, InputIterator in_last) {
        if (pre_first == pre_last) return nullptr;
        if (in_first == in_last) return nullptr;

        auto root = new TreeNode(*pre_first);

        auto inRootPos = find(in_first, in_last, *pre_first);
        auto leftSize = distance(in_first, inRootPos);
        auto preLeftLast = next(pre_first, leftSize + 1);
        // next(in_first, leftSize) == inRootPos

        root->left = buildTree(next(pre_first), preLeftLast, in_first, inRootPos);
        root->right = buildTree(preLeftLast, pre_last, next(inRootPos), in_last);

        return root;
    }
};
```

相關題目

- Construct Binary Tree from Inorder and Postorder Traversal，見 §5.2.2

5.2.2 Construct Binary Tree from Inorder and Postorder Traversal

描述

Given inorder and postorder traversal of a tree, construct the binary tree.

Note: You may assume that duplicates do not exist in the tree.

分析

無

代碼

```
// LeetCode, Construct Binary Tree from Inorder and Postorder Traversal
// 遞歸，時間複雜度 O(n)，空間複雜度 O(\logn)
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        return buildTree(begin(inorder), end(inorder),
                begin(postorder), end(postorder));
    }

    template<typename BidiIt>
    TreeNode* buildTree(BidiIt in_first, BidiIt in_last,
            BidiIt post_first, BidiIt post_last) {
        if (in_first ==in_last) return nullptr;
        if (post_first == post_last) return nullptr;

        const auto val = *prev(post_last);
        TreeNode* root = new TreeNode(val);

        auto inRootPos = find(in_first, in_last, val);
        auto leftSize = distance(in_first, inRootPos);
        auto postLeftLast = next(post_first, leftSize);

        root->left = buildTree(in_first, inRootPos, post_first, postLeftLast);
        root->right = buildTree(next(inRootPos), in_last, postLeftLast, prev(post_last));

        return root;
    }
};
```

相關題目

- Construct Binary Tree from Preorder and Inorder Traversal，見 §5.2.1

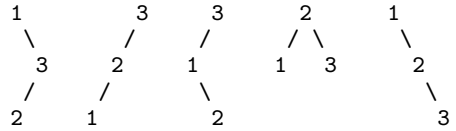
5.3 二叉查找樹

5.3.1 Unique Binary Search Trees

描述

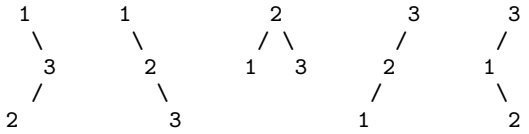
Given n, how many structurally unique BST's (binary search trees) that store values $1...n$?

For example, Given $n = 3$, there are a total of 5 unique BST's.



分析

如果把上例的順序改一下，就可以看出規律了。



比如，以 1 為根的樹的個數，等於左子樹的個數乘以右子樹的個數，左子樹是 0 個元素的樹，右子樹是 2 個元素的樹。以 2 為根的樹的個數，等於左子樹的個數乘以右子樹的個數，左子樹是 1 個元素的樹，右子樹也是 1 個元素的樹。依此類推。

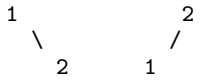
當數組為 $1, 2, 3, ..., n$ 時，基於以下原則的構建的 BST 樹具有唯一性：**以 i 為根節點的樹，其左子樹由 [1, i-1] 構成， 其右子樹由 [i+1, n] 構成。**

定義 $f(i)$ 為以 $[1, i]$ 能產生的 Unique Binary Search Tree 的數目，則

如果數組為空，毫無疑問，只有一種 BST，即空樹，$f(0) = 1$。

如果數組僅有一個元素 1，只有一種 BST，單個節點，$f(1) = 1$。

如果數組有兩個元素 1,2，那麼有如下兩種可能



$$\begin{aligned} f(2) &= f(0) * f(1) \text{ ，1 為根的情況} \\ &+ f(1) * f(0) \text{ ，2 為根的情況} \end{aligned}$$

再看一看 3 個元素的數組，可以發現 BST 的取值方式如下：

$$\begin{aligned} f(3) &= f(0) * f(2) \text{ ，1 為根的情況} \\ &+ f(1) * f(1) \text{ ，2 為根的情況} \\ &+ f(2) * f(0) \text{ ，3 為根的情況} \end{aligned}$$

所以，由此觀察，可以得出 f 的遞推公式為

$$f(i) = \sum_{k=1}^{i} f(k-1) \times f(i-k)$$

至此，問題劃歸為一維動態規劃。

代碼

```
// LeetCode, Unique Binary Search Trees
// 時間複雜度 O(n^2)，空間複雜度 O(n)
class Solution {
public:
    int numTrees(int n) {
        vector<int> f(n + 1, 0);

        f[0] = 1;
        f[1] = 1;
        for (int i = 2; i <= n; ++i) {
            for (int k = 1; k <= i; ++k)
                f[i] += f[k-1] * f[i - k];
        }

        return f[n];
    }
};
```

相關題目

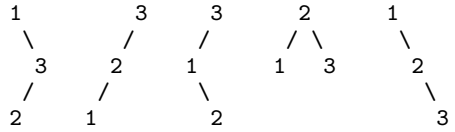
- Unique Binary Search Trees II，見 §5.3.2

5.3.2 Unique Binary Search Trees II

描述

Given n, generate all structurally unique BST's (binary search trees) that store values 1...n.

For example, Given $n = 3$, your program should return all 5 unique BST's shown below.



分析

見前面一題。

代碼

```
// LeetCode, Unique Binary Search Trees II
// 時間複雜度 TODO, 空間複雜度 TODO
class Solution {
public:
    vector<TreeNode *> generateTrees(int n) {
        if (n == 0) return generate(1, 0);
        return generate(1, n);
    }
private:
    vector<TreeNode *> generate(int start, int end) {
        vector<TreeNode*> subTree;
        if (start > end) {
            subTree.push_back(nullptr);
            return subTree;
        }
        for (int k = start; k <= end; k++) {
            vector<TreeNode*> leftSubs = generate(start, k - 1);
            vector<TreeNode*> rightSubs = generate(k + 1, end);
            for (auto i : leftSubs) {
                for (auto j : rightSubs) {
                    TreeNode *node = new TreeNode(k);
                    node->left = i;
                    node->right = j;
                    subTree.push_back(node);
                }
            }
        }
        return subTree;
    }
};
```

相關題目

- Unique Binary Search Trees，見 §5.3.1

5.3.3 Validate Binary Search Tree

描述

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

分析

代碼

```
// Validate Binary Search Tree
// 時間複雜度 O(n)，空間複雜度 O(\logn)
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        return isValidBST(root, LONG_MIN, LONG_MAX);
    }

    bool isValidBST(TreeNode* root, long long lower, long long upper) {
        if (root == nullptr) return true;

        return root->val > lower && root->val < upper
                && isValidBST(root->left, lower, root->val)
                && isValidBST(root->right, root->val, upper);
    }
};
```

相關題目

- Validate Binary Search Tree，見 §5.3.3

5.3.4 Convert Sorted Array to Binary Search Tree

描述

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

分析

二分法。

代碼

```
// LeetCode, Convert Sorted Array to Binary Search Tree
// 分治法，時間複雜度 O(n)，空間複雜度 O(logn)
class Solution {
public:
    TreeNode* sortedArrayToBST (vector<int>& num) {
        return sortedArrayToBST(num.begin(), num.end());
    }

    template<typename RandomAccessIterator>
    TreeNode* sortedArrayToBST (RandomAccessIterator first,
            RandomAccessIterator last) {
        const auto length = distance(first, last);
```

```
        if (length <= 0) return nullptr;  // 終止條件

        // 三方合併
        auto mid = first + length / 2;
        TreeNode* root = new TreeNode (*mid);
        root->left = sortedArrayToBST(first, mid);
        root->right = sortedArrayToBST(mid + 1, last);

        return root;
    }
};
```

相關題目

- Convert Sorted List to Binary Search Tree，見 §5.3.5

5.3.5 Convert Sorted List to Binary Search Tree

描述

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

分析

這題與上一題類似，但是單鏈表不能隨機訪問，而自頂向下的二分法必須需要 RandomAccessIterator，因此前面的方法不適用本題。

存在一種自底向上 (bottom-up) 的方法，見 http://leetcode.com/2010/11/convert-sorted-list-to-balanced-binary.html

分治法，自頂向下

分治法，類似於 Convert Sorted Array to Binary Search Tree，自頂向下，複雜度 $O(n \log n)$。

```
// LeetCode, Convert Sorted List to Binary Search Tree
// 分治法，類似於 Convert Sorted Array to Binary Search Tree,
// 自頂向下，時間複雜度 O(n^2)，空間複雜度 O(logn)
class Solution {
public:
    TreeNode* sortedListToBST (ListNode* head) {
        return sortedListToBST (head, listLength (head));
    }

    TreeNode* sortedListToBST (ListNode* head, int len) {
        if (len == 0) return nullptr;
        if (len == 1) return new TreeNode (head->val);
```

```
        TreeNode* root = new TreeNode (nth_node (head, len / 2 + 1)->val);
        root->left = sortedListToBST (head, len / 2);
        root->right = sortedListToBST (nth_node (head, len / 2 + 2),
                (len - 1) / 2);

        return root;
    }

    int listLength (ListNode* node) {
        int n = 0;

        while(node) {
            ++n;
            node = node->next;
        }

        return n;
    }

    ListNode* nth_node (ListNode* node, int n) {
        while (--n)
            node = node->next;

        return node;
    }
};
```

自底向上

```
// LeetCode, Convert Sorted List to Binary Search Tree
// bottom-up, 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
    TreeNode *sortedListToBST(ListNode *head) {
        int len = 0;
        ListNode *p = head;
        while (p) {
            len++;
            p = p->next;
        }
        return sortedListToBST(head, 0, len - 1);
    }
private:
    TreeNode* sortedListToBST(ListNode*& list, int start, int end) {
        if (start > end) return nullptr;

        int mid = start + (end - start) / 2;
        TreeNode *leftChild = sortedListToBST(list, start, mid - 1);
        TreeNode *parent = new TreeNode(list->val);
        parent->left = leftChild;
        list = list->next;
        parent->right = sortedListToBST(list, mid + 1, end);
```

```
        return parent;
    }
};
```

相關題目

- Convert Sorted Array to Binary Search Tree，見 §5.3.4

5.4 二叉樹的遞歸

二叉樹是一個遞歸的數據結構，因此是一個用來考察遞歸思維能力的絕佳數據結構。

遞歸一定是深搜（見 §10.12.5 節“深搜與遞歸的區別”），由於在二叉樹上，遞歸的味道更濃些，因此本節用“二叉樹的遞歸”作為標題，而不是“二叉樹的深搜”，儘管本節所有的算法都屬於深搜。

二叉樹的先序、中序、後序遍歷都可以看做是 DFS，此外還有其他順序的深度優先遍歷，共有 $3! = 6$ 種。其他 3 種順序是 `root->r->l`，`r->root->l`，`r->l->root`。

5.4.1 Minimum Depth of Binary Tree

描述

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

分析

無

遞歸版

```
// LeetCode, Minimum Depth of Binary Tree
// 遞歸版，時間複雜度 O(n)，空間複雜度 O(logn)
class Solution {
public:
    int minDepth(const TreeNode *root) {
        return minDepth(root, false);
    }
private:
    static int minDepth(const TreeNode *root, bool hasbrother) {
        if (!root) return hasbrother ? INT_MAX : 0;

        return 1 + min(minDepth(root->left, root->right != NULL),
                minDepth(root->right, root->left != NULL));
    }
};
```

迭代版

```
// LeetCode, Minimum Depth of Binary Tree
// 迭代版, 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
    int minDepth(TreeNode* root) {
        if (root == nullptr)
            return 0;

        int result = INT_MAX;

        stack<pair<TreeNode*, int>> s;
        s.push(make_pair(root, 1));

        while (!s.empty()) {
            auto node = s.top().first;
            auto depth = s.top().second;
            s.pop();

            if (node->left == nullptr && node->right == nullptr)
                result = min(result, depth);

            if (node->left && result > depth) // 深度控制, 剪枝
                s.push(make_pair(node->left, depth + 1));

            if (node->right && result > depth) // 深度控制, 剪枝
                s.push(make_pair(node->right, depth + 1));
        }

        return result;
    }
};
```

相關題目

- Maximum Depth of Binary Tree，見 §5.4.2

5.4.2 Maximum Depth of Binary Tree

描述

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

分析

無

代碼

```
// LeetCode, Maximum Depth of Binary Tree
// 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
    int maxDepth(TreeNode *root) {
        if (root == nullptr) return 0;

        return max(maxDepth(root->left), maxDepth(root->right)) + 1;
    }
};
```

相關題目

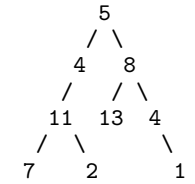
- Minimum Depth of Binary Tree，見 §5.4.1

5.4.3 Path Sum

描述

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example: Given the below binary tree and `sum = 22`,



return true, as there exist a root-to-leaf path `5->4->11->2` which sum is 22.

分析

題目只要求返回 `true` 或者 `false`，因此不需要記錄路徑。

由於只需要求出一個結果，因此，當左、右任意一棵子樹求到了滿意結果，都可以及時 return。

由於題目沒有說節點的數據一定是正整數，必須要走到葉子節點才能判斷，因此中途沒法剪枝，只能進行樸素深搜。

代碼

```
// LeetCode, Path Sum
// 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
```

```
    bool hasPathSum(TreeNode *root, int sum) {
        if (root == nullptr) return false;

        if (root->left == nullptr && root->right == nullptr) // leaf
            return sum == root->val;

        return hasPathSum(root->left, sum - root->val)
                || hasPathSum(root->right, sum - root->val);
    }
};
```

相關題目

- Path Sum II，見 §5.4.4

5.4.4 Path Sum II

描述

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example: Given the below binary tree and `sum = 22`,

```
          5
         / \
        4   8
       /   / \
      11  13  4
     /  \    / \
    7    2  5   1
```

return

```
[
   [5,4,11,2],
   [5,8,4,5]
]
```

分析

跟上一題相比，本題是求路徑本身。且要求出所有結果，左子樹求到了滿意結果，不能 return，要接着求右子樹。

代碼

```
// LeetCode, Path Sum II
// 時間複雜度 O(n)，空間複雜度 O(logn)
class Solution {
public:
    vector<vector<int> > pathSum(TreeNode *root, int sum) {
        vector<vector<int> > result;
```

```
        vector<int> cur; // 中間結果
        pathSum(root, sum, cur, result);
        return result;
    }
private:
    void pathSum(TreeNode *root, int gap, vector<int> &cur,
            vector<vector<int> > &result) {
        if (root == nullptr) return;

        cur.push_back(root->val);

        if (root->left == nullptr && root->right == nullptr) { // leaf
            if (gap == root->val)
                result.push_back(cur);
        }
        pathSum(root->left, gap - root->val, cur, result);
        pathSum(root->right, gap - root->val, cur, result);

        cur.pop_back();
    }
};
```

相關題目

- Path Sum，見 §5.4.3

5.4.5 Binary Tree Maximum Path Sum

描述

Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree. For example: Given the below binary tree,

```
  1
 / \
2   3
```

Return 6.

分析

這題很難，路徑可以從任意節點開始，到任意節點結束。

可以利用“最大連續子序列和”問題的思路，見第 §13.2 節。如果說 Array 只有一個方向的話，那麼 Binary Tree 其實只是左、右兩個方向而已，我們需要比較兩個方向上的值。

不過，Array 可以從頭到尾遍歷，那麼 Binary Tree 怎麼辦呢，我們可以採用 Binary Tree 最常用的 dfs 來進行遍歷。先算出左右子樹的結果 L 和 R，如果 L 大於 0，那麼對後續結果是有利的，我們加上 L，如果 R 大於 0，對後續結果也是有利的，繼續加上 R。

代碼

```
// LeetCode, Binary Tree Maximum Path Sum
// 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
    int maxPathSum(TreeNode *root) {
        max_sum = INT_MIN;
        dfs(root);
        return max_sum;
    }
private:
    int max_sum;
    int dfs(const TreeNode *root) {
        if (root == nullptr) return 0;
        int l = dfs(root->left);
        int r = dfs(root->right);
        int sum = root->val;
        if (l > 0) sum += l;
        if (r > 0) sum += r;
        max_sum = max(max_sum, sum);
        return max(r, l) > 0 ? max(r, l) + root->val : root->val;
    }
};
```

注意，最後 return 的時候，只返回一個方向上的值，為什麼？這是因為在遞歸中，只能向父節點返回，不可能存在 L->root->R 的路徑，只可能是 L->root 或 R->root。

相關題目

- Maximum Subarray，見 §13.2

5.4.6 Populating Next Right Pointers in Each Node

描述

Given a binary tree

```
struct TreeLinkNode {
   int val;
   TreeLinkNode *left, *right, *next;
   TreeLinkNode(int x) : val(x), left(NULL), right(NULL), next(NULL) {}
};
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Note:

- You may only use constant extra space.

- You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example, Given the following perfect binary tree,

```
       1
     /  \
    2    3
   / \  / \
  4  5  6  7
```

After calling your function, the tree should look like:

```
       1 -> NULL
     /  \
    2 -> 3 -> NULL
   / \  / \
  4->5->6->7 -> NULL
```

分析

無

代碼

```
// LeetCode, Populating Next Right Pointers in Each Node
// 時間複雜度 O(n)，空間複雜度 O(logn)
class Solution {
public:
    void connect(TreeLinkNode *root) {
        connect(root, NULL);
    }
private:
    void connect(TreeLinkNode *root, TreeLinkNode *sibling) {
        if (root == nullptr)
            return;
        else
            root->next = sibling;

        connect(root->left, root->right);
        if (sibling)
            connect(root->right, sibling->left);
        else
            connect(root->right, nullptr);
    }
};
```

相關題目

- Populating Next Right Pointers in Each Node II，見 §5.1.12

5.4.7 Sum Root to Leaf Numbers

描述

Given a binary tree containing digits from `0-9` only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path `1->2->3` which represents the number `123`.

Find the total sum of all root-to-leaf numbers.

For example,

```
  1
 / \
2   3
```

The root-to-leaf path `1->2` represents the number `12`. The root-to-leaf path `1->3` represents the number `13`.

Return the sum = `12 + 13 = 25`.

分析

無

代碼

```
// LeetCode, Decode Ways
// 時間複雜度 O(n), 空間複雜度 O(logn)
class Solution {
public:
    int sumNumbers(TreeNode *root) {
        return dfs(root, 0);
    }
private:
    int dfs(TreeNode *root, int sum) {
        if (root == nullptr) return 0;
        if (root->left == nullptr && root->right == nullptr)
            return sum * 10 + root->val;

        return dfs(root->left, sum * 10 + root->val) +
                dfs(root->right, sum * 10 + root->val);
    }
};
```

相關題目

- 無

第 6 章
排序

6.1 Merge Two Sorted Arrays

描述

Given two sorted integer arrays A and B, merge B into A as one sorted array.

Note: You may assume that A has enough space to hold additional elements from B. The number of elements initialized in A and B are m and n respectively.

分析

無

代碼

```
//LeetCode, Merge Sorted Array
// 時間複雜度 O(m+n), 空間複雜度 O(1)
class Solution {
public:
    void merge(vector<int>& A, int m, vector<int>& B, int n) {
        int ia = m - 1, ib = n - 1, icur = m + n - 1;
        while(ia >= 0 && ib >= 0) {
            A[icur--] = A[ia] >= B[ib] ? A[ia--] : B[ib--];
        }
        while(ib >= 0) {
            A[icur--] = B[ib--];
        }
    }
};
```

相關題目

- Merge Two Sorted Lists，見 §6.1
- Merge k Sorted Lists，見 §6.3

6.2 Merge Two Sorted Lists

描述

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

分析

無

代碼

```
//LeetCode, Merge Two Sorted Lists
// 時間複雜度 O(min(m,n)), 空間複雜度 O(1)
class Solution {
public:
    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        if (l1 == nullptr) return l2;
        if (l2 == nullptr) return l1;
        ListNode dummy(-1);
        ListNode *p = &dummy;
        for (; l1 != nullptr && l2 != nullptr; p = p->next) {
            if (l1->val > l2->val) { p->next = l2; l2 = l2->next; }
            else { p->next = l1; l1 = l1->next; }
        }
        p->next = l1 != nullptr ? l1 : l2;
        return dummy.next;
    }
};
```

相關題目

- Merge Sorted Array §6.1
- Merge k Sorted Lists，見 §6.3

6.3 Merge k Sorted Lists

描述

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

分析

可以複用 Merge Two Sorted Lists（見 §6.2）的函數

代碼

```
//LeetCode, Merge k Sorted Lists
// 時間複雜度 O(n1+n2+...), 空間複雜度 O(1)
class Solution {
public:

    ListNode * mergeTwo(ListNode * l1, ListNode * l2){
        if(!l1) return l2;
        if(!l2) return l1;
        ListNode dummy(-1);
        ListNode * p = &dummy;
        for(; l1 && l2; p = p->next){
            if(l1->val > l2->val){
                p->next = l2; l2 = l2->next;
            }
            else{
                p->next = l1; l1 = l1->next;
            }
        }
        p->next = l1 ? l1 : l2;
        return dummy.next;
    }

    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if(lists.size() == 0) return nullptr;

        // multi pass
        deque<ListNode *> dq(lists.begin(), lists.end());
        while(dq.size() > 1){
            ListNode * first = dq.front(); dq.pop_front();
            ListNode * second = dq.front(); dq.pop_front();
            dq.push_back(mergeTwo(first,second));
        }

        return dq.front();
    }
};
```

相關題目

- Merge Sorted Array §6.1
- Merge Two Sorted Lists，見 §6.2

6.4 Insertion Sort List

描述

Sort a linked list using insertion sort.

分析

無

代碼

```
// LeetCode, Insertion Sort List
// 時間複雜度 O(n^2), 空間複雜度 O(1)
class Solution {
public:
    ListNode *insertionSortList(ListNode *head) {
        ListNode dummy(INT_MIN);
        //dummy.next = head;

        for (ListNode *cur = head; cur != nullptr;) {
            auto pos = findInsertPos(&dummy, cur->val);
            ListNode *tmp = cur->next;
            cur->next = pos->next;
            pos->next = cur;
            cur = tmp;
        }
        return dummy.next;
    }

    ListNode* findInsertPos(ListNode *head, int x) {
        ListNode *pre = nullptr;
        for (ListNode *cur = head; cur != nullptr && cur->val <= x;
            pre = cur, cur = cur->next)
            ;
        return pre;
    }
};
```

相關題目

- Sort List, 見 §6.5

6.5 Sort List

描述

Sort a linked list in $O(nlogn)$ time using constant space complexity.

分析

常數空間且 $O(nlogn)$，單鏈表適合用歸併排序，雙向鏈表適合用快速排序。本題可以複用 ”Merge Two Sorted Lists” 的代碼。

代碼

```
// LeetCode, Sort List
// 歸併排序, 時間複雜度 O(nlogn), 空間複雜度 O(1)
class Solution {
public:
    ListNode *sortList(ListNode *head) {
        if (head == NULL || head->next == NULL)return head;

        // 快慢指針找到中間節點
        ListNode *fast = head, *slow = head;
        while (fast->next != NULL && fast->next->next != NULL) {
            fast = fast->next->next;
            slow = slow->next;
        }
        // 斷開
        fast = slow;
        slow = slow->next;
        fast->next = NULL;

        ListNode *l1 = sortList(head);  // 前半段排序
        ListNode *l2 = sortList(slow);  // 後半段排序
        return mergeTwoLists(l1, l2);
    }

    // Merge Two Sorted Lists
    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        ListNode dummy(-1);
        for (ListNode* p = &dummy; l1 != nullptr || l2 != nullptr; p = p->next) {
            int val1 = l1 == nullptr ? INT_MAX : l1->val;
            int val2 = l2 == nullptr ? INT_MAX : l2->val;
            if (val1 <= val2) {
                p->next = l1;
                l1 = l1->next;
            } else {
                p->next = l2;
                l2 = l2->next;
            }
        }
        return dummy.next;
    }
};
```

相關題目

- Insertion Sort List，見 §6.4

6.6 First Missing Positive

描述

Given an unsorted integer array, find the first missing positive integer.

For example, Given `[1,2,0]` return 3, and `[3,4,-1,1]` return 2.

Your algorithm should run in $O(n)$ time and uses constant space.

分析

本質上是桶排序 (bucket sort)，每當 `A[i]!= i+1` 的時候，將 A[i] 與 A[A[i]-1] 交換，直到無法交換為止，終止條件是 `A[i]== A[A[i]-1]`。

代碼

```
// LeetCode, First Missing Positive
// 時間複雜度 O(n)，空間複雜度 O(1)
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        bucket_sort(nums);

        for (int i = 0; i < nums.size(); ++i)
            if (nums[i] != (i + 1))
                return i + 1;
        return nums.size() + 1;
    }
private:
    static void bucket_sort(vector<int>& A) {
        const int n = A.size();
        for (int i = 0; i < n; i++) {
            while (A[i] != i + 1) {
                if (A[i] <= 0 || A[i] > n || A[i] == A[A[i] - 1])
                    break;
                swap(A[i], A[A[i] - 1]);
            }
        }
    }
};
```

相關題目

- Sort Colors, 見 §6.7

6.7 Sort Colors

描述

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?

分析

由於 0, 1, 2 非常緊湊，首先想到計數排序 (counting sort)，但需要掃描兩遍，不符合題目要求。

由於只有三種顏色，可以設置兩個 index，一個是 red 的 index，一個是 blue 的 index，兩邊往中間走。時間複雜度 $O(n)$，空間複雜度 $O(1)$。

第 3 種思路，利用快速排序裏 partition 的思想，第一次將數組按 0 分割，第二次按 1 分割，排序完畢，可以推廣到 n 種顏色，每種顏色有重複元素的情況。

代碼 1

```
// LeetCode, Sort Colors
// Counting Sort
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    void sortColors(vector<int>& A) {
        int counts[3] = { 0 }; // 記錄每個顏色出現的次數

        for (int i = 0; i < A.size(); i++)
            counts[A[i]]++;

        for (int i = 0, index = 0; i < 3; i++)
            for (int j = 0; j < counts[i]; j++)
                A[index++] = i;

    }
};
```

代碼 2

```
// LeetCode, Sort Colors
// 雙指針, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    void sortColors(vector<int>& A) {
        // 一個是 red 的 index, 一個是 blue 的 index, 兩邊往中間走
        int red = 0, blue = A.size() - 1;

        for (int i = 0; i < blue + 1;) {
            if (A[i] == 0)
                swap(A[i++], A[red++]);
            else if (A[i] == 2)
                swap(A[i], A[blue--]);
```

```
            else
                i++;
        }
    }
};
```

代碼 3

```
// LeetCode, Sort Colors
// use partition()
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    void sortColors(vector<int>& nums) {
        partition(partition(nums.begin(), nums.end(), bind1st(equal_to<int>(), 0)),
                nums.end(), bind1st(equal_to<int>(), 1));
    }
};
```

代碼 4

```
// LeetCode, Sort Colors
// 重新實現 partition()
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    void sortColors(vector<int>& nums) {
        partition(partition(nums.begin(), nums.end(), bind1st(equal_to<int>(), 0)),
                 nums.end(), bind1st(equal_to<int>(), 1));
    }
private:
    template<typename ForwardIterator, typename UnaryPredicate>
    ForwardIterator partition(ForwardIterator first, ForwardIterator last,
            UnaryPredicate pred) {
        auto pos = first;

        for (; first != last; ++first)
            if (pred(*first))
                swap(*first, *pos++);

        return pos;
    }
};
```

相關題目

- First Missing Positive, 見 §6.6

第 7 章
查找

7.1 Search for a Range

描述

Given a sorted array of integers, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of $O(\log n)$.

If the target is not found in the array, return

$$-1, -1$$

.

For example, Given

$$5, 7, 7, 8, 8, 10$$

and target value 8, return

$$3, 4$$

.

分析

已經排好了序，用二分查找。

使用 STL

```
// LeetCode, Search for a Range
// 偷懶的做法，使用 STL
// 時間複雜度 O(logn)，空間複雜度 O(1)
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        const int l = distance(nums.begin(), lower_bound(nums.begin(), nums.end(), target));
        const int u =
              distance(nums.begin(), prev(upper_bound(nums.begin(), nums.end(), target)));
        if (nums[l] != target) // not found
            return vector<int> { -1, -1 };
```

```
        else
            return vector<int> { l, u };
    }
};
```

重新實現 lower_bound 和 upper_bound

```
// LeetCode, Search for a Range
// 重新實現 lower_bound 和 upper_bound
// 時間複雜度 O(logn), 空間複雜度 O(1)
class Solution {
public:
    vector<int> searchRange (vector<int>& nums, int target) {
        auto lower = lower_bound(nums.begin(), nums.end(), target);
        auto uppper = upper_bound(lower, nums.end(), target);

        if (lower == nums.end() || *lower != target)
            return vector<int> { -1, -1 };
        else
            return vector<int> {distance(nums.begin(), lower),
                                distance(nums.begin(), prev(uppper))};
    }

    template<typename ForwardIterator, typename T>
    ForwardIterator lower_bound (ForwardIterator first,
            ForwardIterator last, T value) {
        while (first != last) {
            auto mid = next(first, distance(first, last) / 2);

            if (value > *mid)   first = ++mid;
            else                last = mid;
        }

        return first;
    }

    template<typename ForwardIterator, typename T>
    ForwardIterator upper_bound (ForwardIterator first,
            ForwardIterator last, T value) {
        while (first != last) {
            auto mid = next(first, distance (first, last) / 2);

            if (value >= *mid)   first = ++mid;  // 與 lower_bound 僅此不同
            else                 last = mid;
        }

        return first;
    }
};
```

相關題目

- Search Insert Position, 見 §7.2

7.2 Search Insert Position

描述

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

Here are few examples.

```
[1,3,5,6], 5 → 2
[1,3,5,6], 2 → 1
[1,3,5,6], 7 → 4
[1,3,5,6], 0 → 0
```

分析

即 `std::lower_bound()`。

代碼

```
// LeetCode, Search Insert Position
// 重新實現 lower_bound
// 時間複雜度 O(logn), 空間複雜度 O(1)
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        return distance(nums.begin(), lower_bound(nums.begin(), nums.end(), target));
    }

    template<typename ForwardIterator, typename T>
    ForwardIterator lower_bound (ForwardIterator first,
            ForwardIterator last, T value) {
        while (first != last) {
            auto mid = next(first, distance(first, last) / 2);

            if (value > *mid)   first = ++mid;
            else                last = mid;
        }

        return first;
    }
};
```

相關題目

- Search for a Range, 見 §7.1

7.3 Search a 2D Matrix

描述

Write an efficient algorithm that searches for a value in an $m \times n$ matrix. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

For example, Consider the following matrix:

```
[
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
```

Given `target = 3`, return true.

分析

二分查找。

代碼

```
// LeetCode, Search a 2D Matrix
// 時間複雜度 O(logn), 空間複雜度 O(1)
class Solution {
public:
    bool searchMatrix(const vector<vector<int>>& matrix, int target) {
        if (matrix.empty()) return false;
        const size_t  m = matrix.size();
        const size_t n = matrix.front().size();

        int first = 0;
        int last = m * n;

        while (first < last) {
            int mid = first + (last - first) / 2;
            int value = matrix[mid / n][mid % n];

            if (value == target)
                return true;
            else if (value < target)
                first = mid + 1;
            else
                last = mid;
        }

        return false;
    }
};
```

相關題目

- 無

第 8 章
暴力枚舉法

8.1 Subsets

描述

Given a set of distinct integers, S, return all possible subsets.

Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example, If `S = [1,2,3]`, a solution is:

```
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

8.1.1 遞歸

增量構造法 1

每個元素，都有兩種選擇，選或者不選。

```
// LeetCode, Subsets
// 增量構造法，深搜，時間複雜度 O(2^n)，空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        sort(S.begin(), S.end());  // 輸出要求有序
        vector<vector<int> > result;
        vector<int> path;
        subsets(S, path, 0, result);
        return result;
```

```
    }

private:
    static void subsets(const vector<int> &S, vector<int> &path, int step,
            vector<vector<int> > &result) {
        if (step == S.size()) {
            result.push_back(path);
            return;
        }
        // 不選 S[step]
        subsets(S, path, step + 1, result);
        // 選 S[step]
        path.push_back(S[step]);
        subsets(S, path, step + 1, result);
        path.pop_back();
    }
};
```

增量構造法 2

每個元素，都有兩種選擇，選或者不選。

```
// LeetCode, Subsets
// 增量構造法，深搜，時間複雜度 O(2^n)，空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        sort(S.begin(), S.end());  // 輸出要求有序
        vector<vector<int> > result;
        vector<int> path;
        subsets(S, path, 0, result);
        return result;
    }

private:
    static void subsets(const vector<int> &S, vector<int> &path, int step,
                        vector<vector<int> > &result) {
        result.push_back(path);

        for (int i = step; i < S.size(); i++) {
            // 選 S[step]
            path.push_back(S[i]);
            subsets(S, path, i + 1, result);
            // 不選 S[step]
            path.pop_back();
        }
    }
};
```

位向量法

開一個位向量 `bool selected[n]`，每個元素可以選或者不選。

```
// LeetCode, Subsets
// 位向量法, 深搜, 時間複雜度 O(2^n), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        sort(S.begin(), S.end());  // 輸出要求有序

        vector<vector<int> > result;
        vector<bool> selected(S.size(), false);
        subsets(S, selected, 0, result);
        return result;
    }

private:
    static void subsets(const vector<int> &S, vector<bool> &selected, int step,
            vector<vector<int> > &result) {
        if (step == S.size()) {
            vector<int> subset;
            for (int i = 0; i < S.size(); i++) {
                if (selected[i]) subset.push_back(S[i]);
            }
            result.push_back(subset);
            return;
        }
        // 不選 S[step]
        selected[step] = false;
        subsets(S, selected, step + 1, result);
        // 選 S[step]
        selected[step] = true;
        subsets(S, selected, step + 1, result);
    }
};
```

8.1.2 迭代

增量構造法

```
// LeetCode, Subsets
// 迭代版, 時間複雜度 O(2^n), 空間複雜度 O(1)
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        sort(S.begin(), S.end()); // 輸出要求有序
        vector<vector<int> > result(1);
        for (auto elem : S) {
            result.reserve(result.size() * 2);
            auto half = result.begin() + result.size();
            copy(result.begin(), half, back_inserter(result));
            for_each(half, result.end(), [&elem](decltype(result[0]) &e){
                e.push_back(elem);
            });
        }
```

```
        return result;
    }
};
```

二進制法

本方法的前提是：集合的元素不超過 int 位數。用一個 int 整數表示位向量，第 i 位為 1，則表示選擇 $S[i]$，為 0 則不選擇。例如 `S={A,B,C,D}`，則 `0110=6` 表示子集`{B,C}`。

這種方法最巧妙。因為它不僅能生成子集，還能方便的表示集合的並、交、差等集合運算。設兩個集合的位向量分別為 B_1 和 B_2，則 $B_1 \cup B_2, B_1 \cap B_2, B_1 \triangle B_2$ 分別對應集合的並、交、對稱差。

二進制法，也可以看做是位向量法，只不過更加優化。

```
// LeetCode, Subsets
// 二進制法, 時間複雜度 O(2^n), 空間複雜度 O(1)
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        sort(S.begin(), S.end()); // 輸出要求有序
        vector<vector<int> > result;
        const size_t n = S.size();
        vector<int> v;

        for (size_t i = 0; i < 1 << n; i++) {
            for (size_t j = 0; j < n; j++) {
                if (i & 1 << j) v.push_back(S[j]);
            }
            result.push_back(v);
            v.clear();
        }
        return result;
    }
};
```

相關題目

- Subsets II，見 §8.2

8.2 Subsets II

描述

Given a collection of integers that might contain duplicates, S, return all possible subsets.

Note:

Elements in a subset must be in non-descending order. The solution set must not contain duplicate subsets. For example, If `S = [1,2,2]`, a solution is:

```
[
  [2],
  [1],
```

```
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

分析

這題有重複元素，但本質上，跟上一題很類似，上一題中元素沒有重複，相當於每個元素只能選 0 或 1 次，這裏擴充到了每個元素可以選 0 到若干次而已。

8.2.1 遞歸

增量構造法

```
// LeetCode, Subsets II
// 增量構造法, 版本 1, 時間複雜度 O(2^n), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        sort(S.begin(), S.end());  // 必須排序

        vector<vector<int> > result;
        vector<int> path;

        dfs(S, S.begin(), path, result);
        return result;
    }

private:
    static void dfs(const vector<int> &S, vector<int>::iterator start,
            vector<int> &path, vector<vector<int> > &result) {
        result.push_back(path);

        for (auto i = start; i < S.end(); i++) {
            if (i != start && *i == *(i-1)) continue;
            path.push_back(*i);
            dfs(S, i + 1, path, result);
            path.pop_back();
        }
    }
};
```

```
// LeetCode, Subsets II
// 增量構造法, 版本 2, 時間複雜度 O(2^n), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        vector<vector<int> > result;
        sort(S.begin(), S.end()); // 必須排序
```

```
        unordered_map<int, int> count_map; // 記錄每個元素的出現次數
        for_each(S.begin(), S.end(), [&count_map](int e) {
            if (count_map.find(e) != count_map.end())
                count_map[e]++;
            else
                count_map[e] = 1;
        });

        // 將 map 裏的 pair 拷貝到一個 vector 裏
        vector<pair<int, int> > elems;
        for_each(count_map.begin(), count_map.end(),
                [&elems](const pair<int, int> &e) {
                    elems.push_back(e);
                });
        sort(elems.begin(), elems.end());
        vector<int> path; // 中間結果

        subsets(elems, 0, path, result);
        return result;
    }

private:
    static void subsets(const vector<pair<int, int> > &elems,
            size_t step, vector<int> &path, vector<vector<int> > &result) {
        if (step == elems.size()) {
            result.push_back(path);
            return;
        }

        for (int i = 0; i <= elems[step].second; i++) {
            for (int j = 0; j < i; ++j) {
                path.push_back(elems[step].first);
            }
            subsets(elems, step + 1, path, result);
            for (int j = 0; j < i; ++j) {
                path.pop_back();
            }
        }
    }
};
```

位向量法

```
// LeetCode, Subsets II
// 位向量法, 時間複雜度 O(2^n), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        vector<vector<int> > result; // 必須排序
        sort(S.begin(), S.end());
        vector<int> count(S.back() - S.front() + 1, 0);
        // 計算所有元素的個數
```

```
        for (auto i : S) {
            count[i - S[0]]++;
        }

        // 每個元素選擇了多少個
        vector<int> selected(S.back() - S.front() + 1, -1);

        subsets(S, count, selected, 0, result);
        return result;
    }

private:
    static void subsets(const vector<int> &S, vector<int> &count,
            vector<int> &selected, size_t step, vector<vector<int> > &result) {
        if (step == count.size()) {
            vector<int> subset;
            for(size_t i = 0; i < selected.size(); i++) {
                for (int j = 0; j < selected[i]; j++) {
                    subset.push_back(i+S[0]);
                }
            }
            result.push_back(subset);
            return;
        }

        for (int i = 0; i <= count[step]; i++) {
            selected[step] = i;
            subsets(S, count, selected, step + 1, result);
        }
    }
};
```

8.2.2 迭代

增量構造法

```
// LeetCode, Subsets II
// 增量構造法
// 時間複雜度 O(2^n), 空間複雜度 O(1)
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        sort(S.begin(), S.end()); // 必須排序
        vector<vector<int> > result(1);

        size_t previous_size = 0;
        for (size_t i = 0; i < S.size(); ++i) {
            const size_t size = result.size();
            for (size_t j = 0; j < size; ++j) {
                if (i == 0 || S[i] != S[i-1] || j >= previous_size) {
                    result.push_back(result[j]);
                    result.back().push_back(S[i]);
```

```
                }
            }
            previous_size = size;
        }
        return result;
    }
};
```

二進制法

```
// LeetCode, Subsets II
// 二進制法, 時間複雜度 O(2^n), 空間複雜度 O(1)
class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        sort(S.begin(), S.end()); // 必須排序
        // 用 set 去重, 不能用 unordered_set, 因為輸出要求有序
        set<vector<int> > result;
        const size_t n = S.size();
        vector<int> v;

        for (size_t i = 0; i < 1U << n; ++i) {
            for (size_t j = 0; j < n; ++j) {
                if (i & 1 << j)
                    v.push_back(S[j]);
            }
            result.insert(v);
            v.clear();
        }
        vector<vector<int> > real_result;
        copy(result.begin(), result.end(), back_inserter(real_result));
        return real_result;
    }
};
```

相關題目

- Subsets，見 §8.1

8.3 Permutations

描述

Given a collection of numbers, return all possible permutations.

For example, `[1,2,3]` have the following permutations: `[1,2,3]`, `[1,3,2]`, `[2,1,3]`, `[2,3,1]`, `[3,1,2]`, and `[3,2,1]`.

8.3.1 next_permutation()

偷懶的做法，可以直接使用 `std::next_permutation()`。如果是在 OJ 網站上，可以用這個 API 偷個懶；如果是在面試中，面試官肯定會讓你重新實現。

代碼

```
// LeetCode, Permutations
// 時間複雜度 O(n!), 空間複雜度 O(1)
class Solution {
public:
    vector<vector<int> > permute(vector<int> &num) {
        vector<vector<int> > result;
        sort(num.begin(), num.end());

        do {
            result.push_back(num);
        } while(next_permutation(num.begin(), num.end()));
        return result;
    }
};
```

8.3.2 重新實現 next_permutation()

見第 §2.1.12 節。

代碼

```
// LeetCode, Permutations
// 重新實現 next_permutation()
// 時間複雜度 O(n!), 空間複雜度 O(1)
class Solution {
public:
    vector<vector<int> > permute(vector<int> &num) {
        vector<vector<int> > result;
        sort(num.begin(), num.end());

        do {
            result.push_back(num);
        // 調用的是 2.1.12 節的 next_permutation()
        // 而不是 std::next_permutation()
        } while(next_permutation(num.begin(), num.end()));
        return result;
    }
};
```

8.3.3 遞歸

本題是求路徑本身，求所有解，函數參數需要標記當前走到了哪步，還需要中間結果的引用，最終結果的引用。

擴展節點，每次從左到右，選一個沒有出現過的元素。

本題不需要判重，因為狀態裝換圖是一顆有層次的樹。收斂條件是當前走到了最後一個元素。

代碼

```
// LeetCode, Permutations
// 深搜, 增量構造法
// 時間複雜度 O(n!), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > permute(vector<int>& num) {
        sort(num.begin(), num.end());

        vector<vector<int>> result;
        vector<int> path;  // 中間結果

        dfs(num, path, result);
        return result;
    }
private:
    void dfs(const vector<int>& num, vector<int> &path,
            vector<vector<int> > &result) {
        if (path.size() == num.size()) {  // 收斂條件
            result.push_back(path);
            return;
        }

        // 擴展狀態
        for (auto i : num) {
            // 查找 i 是否在 path 中出現過
            auto pos = find(path.begin(), path.end(), i);

            if (pos == path.end()) {
                path.push_back(i);
                dfs(num, path, result);
                path.pop_back();
            }
        }
    }
};
```

相關題目

- Next Permutation, 見 §2.1.12
- Permutation Sequence, 見 §2.1.14

- Permutations II, 見 §8.4
- Combinations, 見 §8.5

8.4 Permutations II

描述

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example, `[1,1,2]` have the following unique permutations: `[1,1,2]`, `[1,2,1]`, and `[2,1,1]`.

8.4.1 next_permutation()

直接使用 `std::next_permutation()`，代碼與上一題相同。

8.4.2 重新實現 next_permutation()

重新實現 `std::next_permutation()`，代碼與上一題相同。

8.4.3 遞歸

遞歸函數 `permute()` 的參數 `p`，是中間結果，它的長度又能標記當前走到了哪一步，用於判斷收斂條件。

擴展節點，每次從小到大，選一個沒有被用光的元素，直到所有元素被用光。

本題不需要判重，因為狀態裝換圖是一顆有層次的樹。

代碼

```
// LeetCode, Permutations II
// 深搜, 時間複雜度 O(n!), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > permuteUnique(vector<int>& num) {
        sort(num.begin(), num.end());

        unordered_map<int, int> count_map; // 記錄每個元素的出現次數
        for_each(num.begin(), num.end(), [&count_map](int e) {
            if (count_map.find(e) != count_map.end())
                count_map[e]++;
            else
                count_map[e] = 1;
        });

        // 將 map 裏的 pair 拷貝到一個 vector 裏
        vector<pair<int, int> > elems;
        for_each(count_map.begin(), count_map.end(),
                [&elems](const pair<int, int> &e) {
```

```
                    elems.push_back(e);
                });

        vector<vector<int>> result; // 最終結果
        vector<int> p;  // 中間結果

        n = num.size();
        permute(elems.begin(), elems.end(), p, result);
        return result;
    }

private:
    size_t n;
    typedef vector<pair<int, int> >::const_iterator Iter;

    void permute(Iter first, Iter last, vector<int> &p,
            vector<vector<int> > &result) {
        if (n == p.size()) {  // 收斂條件
            result.push_back(p);
        }

        // 擴展狀態
        for (auto i = first; i != last; i++) {
            int count = 0; // 統計 *i 在 p 中出現過多少次
            for (auto j = p.begin(); j != p.end(); j++) {
                if (i->first == *j) {
                    count ++;
                }
            }
            if (count < i->second) {
                p.push_back(i->first);
                permute(first, last, p, result);
                p.pop_back(); // 撤銷動作，返回上一層
            }
        }
    }
};
```

相關題目

- Next Permutation, 見 §2.1.12
- Permutation Sequence, 見 §2.1.14
- Permutations, 見 §8.3
- Combinations, 見 §8.5

8.5 Combinations

描述

Given two integers n and k, return all possible combinations of k numbers out of $1...n$.

For example, If $n = 4$ and $k = 2$, a solution is:

```
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

8.5.1 遞歸

```
// LeetCode, Combinations
// 深搜, 遞歸
// 時間複雜度 O(n!), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > combine(int n, int k) {
        vector<vector<int> > result;
        vector<int> path;
        dfs(n, k, 1, 0, path, result);
        return result;
    }
private:
    // start, 開始的數, cur, 已經選擇的數目
    static void dfs(int n, int k, int start, int cur,
            vector<int> &path, vector<vector<int> > &result) {
        if (cur == k) {
            result.push_back(path);
        }
        for (int i = start; i <= n; ++i) {
            path.push_back(i);
            dfs(n, k, i + 1, cur + 1, path, result);
            path.pop_back();
        }
    }
};
```

8.5.2 迭代

```
// LeetCode, Combinations
// use prev_permutation()
// 時間複雜度 O((n-k)!), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > combine(int n, int k) {
        vector<int> values(n);
        iota(values.begin(), values.end(), 1);

        vector<bool> select(n, false);
        fill_n(select.begin(), k, true);
```

```
            vector<vector<int> > result;
            do{
                vector<int> one(k);
                for (int i = 0, index = 0; i < n; ++i)
                    if (select[i])
                        one[index++] = values[i];
                result.push_back(one);
            } while(prev_permutation(select.begin(), select.end()));
            return result;
        }
    };
```

相關題目

- Next Permutation, 見 §2.1.12
- Permutation Sequence, 見 §2.1.14
- Permutations, 見 §8.3
- Permutations II, 見 §8.4

8.6 Letter Combinations of a Phone Number

描述

Given a digit string, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below.

图 8-1 Phone Keyboard

Input: Digit string `"23"`

Output:

$$"ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"$$

.

Note: Although the above answer is in lexicographical order, your answer could be in any order you want.

分析

無

8.6.1 遞歸

```
// LeetCode, Letter Combinations of a Phone Number
// 時間複雜度 O(3^n), 空間複雜度 O(n)
class Solution {
public:
    const vector<string> keyboard { " ", "", "abc", "def", // '0','1','2',...
            "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz" };

    vector<string> letterCombinations (const string &digits) {
        vector<string> result;
        if (digits.empty()) return result;
        dfs(digits, 0, "", result);
        return result;
    }

    void dfs(const string &digits, size_t cur, string path,
            vector<string> &result) {
        if (cur == digits.size()) {
            result.push_back(path);
            return;
        }
        for (auto c : keyboard[digits[cur] - '0']) {
            dfs(digits, cur + 1, path + c, result);
        }
    }
};
```

8.6.2 迭代

```
// LeetCode, Letter Combinations of a Phone Number
// 時間複雜度 O(3^n), 空間複雜度 O(1)
class Solution {
public:
    const vector<string> keyboard { " ", "", "abc", "def", // '0','1','2',...
            "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz" };

    vector<string> letterCombinations (const string &digits) {
        if (digits.empty()) return vector<string>();
        vector<string> result(1, "");
        for (auto d : digits) {
            const size_t n = result.size();
            const size_t m = keyboard[d - '0'].size();

            result.resize(n * m);
            for (size_t i = 0; i < m; ++i)
                copy(result.begin(), result.begin() + n, result.begin() + n * i);
```

```
            for (size_t i = 0; i < m; ++i) {
                auto begin = result.begin();
                for_each(begin + n * i, begin + n * (i+1), [&](string &s) {
                    s += keyboard[d - '0'][i];
                });
            }
        }
        return result;
    }
};
```

相關題目

- 無

第 9 章
廣度優先搜索

當題目看不出任何規律，既不能用分治，貪心，也不能用動規時，這時候萬能方法——搜索，就派上用場了。搜索分為廣搜和深搜，廣搜裏面又有普通廣搜，雙向廣搜，A*搜索等。深搜裏面又有普通深搜，回溯法等。

廣搜和深搜非常類似（除了在擴展節點這部分不一樣），二者有相同的框架，如何表示狀態？如何擴展狀態？如何判重？尤其是判重，解決了這個問題，基本上整個問題就解決了。

9.1 Word Ladder

描述

Given two words (start and end), and a dictionary, find the length of shortest transformation sequence from start to end, such that:

- Only one letter can be changed at a time
- Each intermediate word must exist in the dictionary

For example, Given:

```
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
```

As one shortest transformation is `"hit" -> "hot" -> "dot" -> "dog" -> "cog"`, return its length 5.

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.

分析

求最短路徑，用廣搜。

單隊列

```
//LeetCode, Word Ladder
// 時間複雜度 O(n), 空間複雜度 O(n)
struct state_t {
    string word;
    int level;

    state_t() { word = ""; level = 0; }
    state_t(const string& word, int level) {
        this->word = word;
        this->level = level;
    }

    bool operator==(const state_t &other) const {
        return this->word == other.word;
    }
};

namespace std {
    template<> struct hash<state_t> {
    public:
        size_t operator()(const state_t& s) const {
            return str_hash(s.word);
        }
    private:
        std::hash<std::string> str_hash;
    };
}


class Solution {
public:
    int ladderLength(const string& start, const string &end,
            const unordered_set<string> &dict) {
        queue<state_t> q;
        unordered_set<state_t> visited;  // 判重

        auto state_is_valid = [&](const state_t& s) {
            return dict.find(s.word) != dict.end() || s.word == end;
        };
        auto state_is_target = [&](const state_t &s) {return s.word == end; };
        auto state_extend = [&](const state_t &s) {
            unordered_set<state_t> result;

            for (size_t i = 0; i < s.word.size(); ++i) {
                state_t new_state(s.word, s.level + 1);
                for (char c = 'a'; c <= 'z'; c++) {
                    // 防止同字母替換
                    if (c == new_state.word[i]) continue;

                    swap(c, new_state.word[i]);
```

```
                    if (state_is_valid(new_state) &&
                        visited.find(new_state) == visited.end()) {
                        result.insert(new_state);
                    }
                    swap(c, new_state.word[i]); // 恢復該單詞
                }
            }

            return result;
        };

        state_t start_state(start, 0);
        q.push(start_state);
        visited.insert(start_state);
        while (!q.empty()) {
            // 千萬不能用 const auto&, pop() 會刪除元素,
            // 引用就變成了懸空引用
            const auto state = q.front();
            q.pop();

            if (state_is_target(state)) {
                return state.level + 1;
            }

            const auto& new_states = state_extend(state);
            for (const auto& new_state : new_states) {
                q.push(new_state);
                visited.insert(new_state);
            }
        }
        return 0;
    }
};
```

雙隊列

```
//LeetCode, Word Ladder
// 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    int ladderLength(const string& start, const string &end,
            const unordered_set<string> &dict) {
        queue<string> current, next;    // 當前層, 下一層
        unordered_set<string> visited;  // 判重

        int level = -1;  // 層次

        auto state_is_valid = [&](const string& s) {
            return dict.find(s) != dict.end() || s == end;
        };
        auto state_is_target = [&](const string &s) {return s == end;};
        auto state_extend = [&](const string &s) {
```

```
            unordered_set<string> result;

            for (size_t i = 0; i < s.size(); ++i) {
                string new_word(s);
                for (char c = 'a'; c <= 'z'; c++) {
                    // 防止同字母替換
                    if (c == new_word[i]) continue;

                    swap(c, new_word[i]);

                    if (state_is_valid(new_word) &&
                        visited.find(new_word) == visited.end()) {
                        result.insert(new_word);
                    }
                    swap(c, new_word[i]); // 恢復該單詞
                }
            }

            return result;
        };

        current.push(start);
        visited.insert(start);
        while (!current.empty()) {
            ++level;
            while (!current.empty()) {
                // 千萬不能用 const auto&, pop() 會刪除元素,
                // 引用就變成了懸空引用
                const auto state = current.front();
                current.pop();

                if (state_is_target(state)) {
                    return level + 1;
                }

                const auto& new_states = state_extend(state);
                for (const auto& new_state : new_states) {
                    next.push(new_state);
                    visited.insert(new_state);
                }
            }
            swap(next, current);
        }
        return 0;
    }
};
```

相關題目

- Word Ladder II，見 §9.2

9.2 Word Ladder II

描述

Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that:

- Only one letter can be changed at a time
- Each intermediate word must exist in the dictionary

For example, Given:

```
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
```

Return

```
[
    ["hit","hot","dot","dog","cog"],
    ["hit","hot","lot","log","cog"]
]
```

Note:

- All words have the same length.
- All words contain only lowercase alphabetic characters.

分析

跟 Word Ladder 比，這題是求路徑本身，不是路徑長度，也是 BFS，略微麻煩點。

求一條路徑和求所有路徑有很大的不同，求一條路徑，每個狀態節點只需要記錄一個前驅即可；求所有路徑時，有的狀態節點可能有多個父節點，即要記錄多個前驅。

如果當前路徑長度已經超過當前最短路徑長度，可以中止對該路徑的處理，因為我們要找的是最短路徑。

單隊列

```
//LeetCode, Word Ladder II
// 時間複雜度 O(n), 空間複雜度 O(n)
struct state_t {
    string word;
    int level;

    state_t() { word = ""; level = 0; }
    state_t(const string& word, int level) {
        this->word = word;
        this->level = level;
    }
```

```
    bool operator==(const state_t &other) const {
        return this->word == other.word;
    }
};

namespace std {
    template<> struct hash<state_t> {
    public:
        size_t operator()(const state_t& s) const {
            return str_hash(s.word);
        }
    private:
        std::hash<std::string> str_hash;
    };
}


class Solution {
public:
    vector<vector<string> > findLadders(const string& start,
        const string& end, const unordered_set<string> &dict) {
        queue<state_t> q;
        unordered_set<state_t> visited; // 判重
        unordered_map<state_t, vector<state_t> > father; // DAG (树) key: child value: father

        auto state_is_valid = [&](const state_t& s) {
            return dict.find(s.word) != dict.end() || s.word == end;
        };
        auto state_is_target = [&](const state_t &s) {return s.word == end; };
        auto state_extend = [&](const state_t &s) {
            unordered_set<state_t> result;

            for (size_t i = 0; i < s.word.size(); ++i) {
                state_t new_state(s.word, s.level + 1);
                for (char c = 'a'; c <= 'z'; c++) {
                    // 防止同字母替换
                    if (c == new_state.word[i]) continue;

                    swap(c, new_state.word[i]);

                    if (state_is_valid(new_state)) {
                        auto visited_iter = visited.find(new_state);

                        if (visited_iter != visited.end()) {
                            if (visited_iter->level < new_state.level) {
                                // do nothing
                            } else if (visited_iter->level == new_state.level) {
                                result.insert(new_state);
                            } else { // not possible
                                throw std::logic_error("not possible to get here");
                            }
                        } else {
                            result.insert(new_state);
```

```
                    }
                }
                swap(c, new_state.word[i]); // 恢復該單詞
            }
        }

        return result;
    };

    vector<vector<string>> result;
    state_t start_state(start, 0);
    q.push(start_state);
    visited.insert(start_state);
    while (!q.empty()) {
        // 千萬不能用 const auto&, pop() 會刪除元素,
        // 引用就變成了懸空引用
        const auto state = q.front();
        q.pop();

        // 如果當前路徑長度已經超過當前最短路徑長度,
        // 可以中止對該路徑的處理, 因為我們要找的是最短路徑
        if (!result.empty() && state.level + 1 > result[0].size()) break;

        if (state_is_target(state)) {
            vector<string> path;
            gen_path(father, start_state, state, path, result);
            continue;
        }
        // 必須挪到下面, 比如同一層 A 和 B 兩個節點均指向了目標節點,
        // 那麼目標節點就會在 q 中出現兩次, 輸出路徑就會翻倍
        // visited.insert(state);

        // 擴展節點
        const auto& new_states = state_extend(state);
        for (const auto& new_state : new_states) {
            if (visited.find(new_state) == visited.end()) {
                q.push(new_state);
            }
            visited.insert(new_state);
            father[new_state].push_back(state);
        }
    }

    return result;
}
private:
    void gen_path(unordered_map<state_t, vector<state_t> > &father,
        const state_t &start, const state_t &state, vector<string> &path,
        vector<vector<string> > &result) {
        path.push_back(state.word);
        if (state == start) {
            if (!result.empty()) {
                if (path.size() < result[0].size()) {
```

```
                    result.clear();
                    result.push_back(path);
                    reverse(result.back().begin(), result.back().end());
                } else if (path.size() == result[0].size()) {
                    result.push_back(path);
                    reverse(result.back().begin(), result.back().end());
                } else { // not possible
                    throw std::logic_error("not possible to get here ");
                }
            } else {
                result.push_back(path);
                reverse(result.back().begin(), result.back().end());
            }

        } else {
            for (const auto& f : father[state]) {
                gen_path(father, start, f, path, result);
            }
        }
        path.pop_back();
    }
};
```

雙隊列

```
//LeetCode, Word Ladder II
// 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<string> > findLadders(const string& start,
            const string& end, const unordered_set<string> &dict) {
        // 當前層, 下一層, 用 unordered_set 是為了去重, 例如兩個父節點指向
        // 同一個子節點, 如果用 vector, 子節點就會在 next 裏出現兩次, 其實此
        // 時 father 已經記錄了兩個父節點, next 裏重複出現兩次是沒必要的
        unordered_set<string> current, next;
        unordered_set<string> visited; // 判重
        unordered_map<string, vector<string> > father; // DAG

        int level = -1;  // 層次

        auto state_is_valid = [&](const string& s) {
            return dict.find(s) != dict.end() || s == end;
        };
        auto state_is_target = [&](const string &s) {return s == end;};
        auto state_extend = [&](const string &s) {
            unordered_set<string> result;

            for (size_t i = 0; i < s.size(); ++i) {
                string new_word(s);
                for (char c = 'a'; c <= 'z'; c++) {
                    // 防止同字母替換
                    if (c == new_word[i]) continue;
```

```
                    swap(c, new_word[i]);

                    if (state_is_valid(new_word) &&
                            visited.find(new_word) == visited.end()) {
                        result.insert(new_word);
                    }
                    swap(c, new_word[i]); // 恢復該單詞
                }
            }

            return result;
        };

        vector<vector<string> > result;
        current.insert(start);
        while (!current.empty()) {
            ++ level;
            // 如果當前路徑長度已經超過當前最短路徑長度, 可以中止對該路徑的
            // 處理, 因為我們要找的是最短路徑
            if (!result.empty() && level+1 > result[0].size()) break;

            // 1. 延遲加入 visited, 這樣才能允許兩個父節點指向同一個子節點
            // 2. 一股腦 current 全部加入 visited, 是防止本層前一個節點擴展
            // 節點時, 指向了本層後面尚未處理的節點, 這條路徑必然不是最短的
            for (const auto& state : current)
                visited.insert(state);
            for (const auto& state : current) {
                if (state_is_target(state)) {
                    vector<string> path;
                    gen_path(father, path, start, state, result);
                    continue;
                }

                const auto new_states = state_extend(state);
                for (const auto& new_state : new_states) {
                    next.insert(new_state);
                    father[new_state].push_back(state);
                }
            }

            current.clear();
            swap(current, next);
        }

        return result;
    }
private:
    void gen_path(unordered_map<string, vector<string> > &father,
            vector<string> &path, const string &start, const string &word,
            vector<vector<string> > &result) {
        path.push_back(word);
        if (word == start) {
```

```
            if (!result.empty()) {
                if (path.size() < result[0].size()) {
                    result.clear();
                    result.push_back(path);
                } else if(path.size() == result[0].size()) {
                    result.push_back(path);
                } else {
                    // not possible
                    throw std::logic_error("not possible to get here");
                }
            } else {
                result.push_back(path);
            }
            reverse(result.back().begin(), result.back().end());
        } else {
            for (const auto& f : father[word]) {
                gen_path(father, path, start, f, result);
            }
        }
        path.pop_back();
    }
};
```

圖的廣搜

本題還可以看做是圖上的廣搜。給定了字典 dict，可以基於它畫出一個無向圖，表示單詞之間可以互相轉換。本題的本質就是已知起點和終點，在圖上找出所有最短路徑。

```
//LeetCode, Word Ladder II
// 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<string> > findLadders(const string& start,
            const string &end, const unordered_set<string> &dict) {
        const auto& g = build_graph(dict);
        vector<state_t*> pool;
        queue<state_t*> q; // 未處理的節點
        // value 是所在層次
        unordered_map<string, int> visited;

        auto state_is_target = [&](const state_t *s) {return s->word == end; };

        vector<vector<string>> result;
        q.push(create_state(nullptr, start, 0, pool));
        while (!q.empty()) {
            state_t* state = q.front();
            q.pop();

            // 如果當前路徑長度已經超過當前最短路徑長度,
            // 可以中止對該路徑的處理, 因為我們要找的是最短路徑
            if (!result.empty() && state->level+1 > result[0].size()) break;
```

```
            if (state_is_target(state)) {
                const auto& path = gen_path(state);
                if (result.empty()) {
                    result.push_back(path);
                } else {
                    if (path.size() < result[0].size()) {
                        result.clear();
                        result.push_back(path);
                    } else if (path.size() == result[0].size()) {
                        result.push_back(path);
                    } else {
                        // not possible
                        throw std::logic_error("not possible to get here");
                    }
                }
                continue;
            }

            // 擴展節點
            auto iter = g.find(state->word);
            if (iter == g.end()) continue;

            for (const auto& neighbor : iter->second) {
                auto visited_iter = visited.find(neighbor);

                if (visited_iter != visited.end() &&
                    visited_iter->second < state->level + 1) {
                    continue;
                }

                visited[neighbor] = state->level;
                q.push(create_state(state, neighbor, state->level + 1, pool));
            }
        }

        // release all states
        for (auto state : pool) {
            delete state;
        }
        return result;
    }

private:
    struct state_t {
        state_t* father;
        string word;
        int level; // 所在層次, 從 0 開始編號

        state_t(state_t* father_, const string& word_, int level_) :
            father(father_), word(word_), level(level_) {}
    };

    state_t* create_state(state_t* parent, const string& value,
```

```
            int length, vector<state_t*>& pool) {
        state_t* node = new state_t(parent, value, length);
        pool.push_back(node);

        return node;
    }
    vector<string> gen_path(const state_t* node) {
        vector<string> path;

        while(node != nullptr) {
            path.push_back(node->word);
            node = node->father;
        }

        reverse(path.begin(), path.end());
        return path;
    }

    unordered_map<string, unordered_set<string> > build_graph(
            const unordered_set<string>& dict) {
        unordered_map<string, unordered_set<string> > adjacency_list;

        for (const auto& word : dict) {
            for (size_t i = 0; i < word.size(); ++i) {
                string new_word(word);
                for (char c = 'a'; c <= 'z'; c++) {
                    // 防止同字母替換
                    if (c == new_word[i]) continue;

                    swap(c, new_word[i]);

                    if ((dict.find(new_word) != dict.end())) {
                        auto iter = adjacency_list.find(word);
                        if (iter != adjacency_list.end()) {
                            iter->second.insert(new_word);
                        } else {
                            adjacency_list.insert(pair<string,
                                unordered_set<string>>(word, unordered_set<string>()));
                            adjacency_list[word].insert(new_word);
                        }
                    }
                    swap(c, new_word[i]); // 恢復該單詞
                }
            }
        }
        return adjacency_list;
    }
};
```

相關題目

- Word Ladder，見 §9.1

9.3 Surrounded Regions

描述

Given a 2D board containing `'X'` and `'O'`, capture all regions surrounded by `'X'`.

A region is captured by flipping all `'O'`s into `'X'`s in that surrounded region .

For example,

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

分析

廣搜。從上下左右四個邊界往裏走，凡是能碰到的`'O'`，都是跟邊界接壤的，應該保留。

代碼

```
// LeetCode, Surrounded Regions
// BFS, 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    void solve(vector<vector<char>> &board) {
        if (board.empty()) return;

        const int m = board.size();
        const int n = board[0].size();
        for (int i = 0; i < n; i++) {
            bfs(board, 0, i);
            bfs(board, m - 1, i);
        }
        for (int j = 1; j < m - 1; j++) {
            bfs(board, j, 0);
            bfs(board, j, n - 1);
        }
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                if (board[i][j] == 'O')
                    board[i][j] = 'X';
                else if (board[i][j] == '+')
                    board[i][j] = 'O';
    }
private:
    void bfs(vector<vector<char>> &board, int i, int j) {
```

```
        typedef pair<int, int> state_t;
        queue<state_t> q;
        const int m = board.size();
        const int n = board[0].size();

        auto state_is_valid = [&](const state_t &s) {
            const int x = s.first;
            const int y = s.second;
            if (x < 0 || x >= m || y < 0 || y >= n || board[x][y] != 'O')
                return false;
            return true;
        };

        auto state_extend = [&](const state_t &s) {
            vector<state_t> result;
            const int x = s.first;
            const int y = s.second;
            // 上下左右
            const state_t new_states[4] = {{x-1,y}, {x+1,y},
                    {x,y-1}, {x,y+1}};
            for (int k = 0; k < 4;  ++k) {
                if (state_is_valid(new_states[k])) {
                    // 既有標記功能又有去重功能
                    board[new_states[k].first][new_states[k].second] = '+';
                    result.push_back(new_states[k]);
                }
            }

            return result;
        };

        state_t start = { i, j };
        if (state_is_valid(start)) {
            board[i][j] = '+';
            q.push(start);
        }
        while (!q.empty()) {
            auto cur = q.front();
            q.pop();
            auto new_states = state_extend(cur);
            for (auto s : new_states) q.push(s);
        }
    }
};
```

相關題目

- 無

9.4 小結

9.4.1 適用場景

輸入數據：沒什麼特徵，不像深搜，需要有“遞歸”的性質。如果是樹或者圖，概率更大。
狀態轉換圖：樹或者 DAG 圖。
求解目標：求最短。

9.4.2 思考的步驟

1. 是求路徑長度，還是路徑本身（或動作序列）?
   (a) 如果是求路徑長度，則狀態裏面要存路徑長度（或雙隊列+一個全局變量）
   (b) 如果是求路徑本身或動作序列
      i. 要用一棵樹存儲寬搜過程中的路徑
      ii. 是否可以預估狀態個數的上限？能夠預估狀態總數，則開一個大數組，用樹的雙親表示法；如果不能預估狀態總數，則要使用一棵通用的樹。這一步也是第 4 步的必要不充分條件。
2. 如何表示狀態? 即一個狀態需要存儲哪些些必要的數據，才能夠完整提供如何擴展到下一步狀態的所有信息。一般記錄當前位置或整體局面。
3. 如何擴展狀態? 這一步跟第 2 步相關。狀態裏記錄的數據不同，擴展方法就不同。對於固定不變的數據結構（一般題目直接給出，作為輸入數據），如二叉樹，圖等，擴展方法很簡單，直接往下一層走，對於隱式圖，要先在第 1 步裏想清楚狀態所帶的數據，想清楚了這點，那如何擴展就很簡單了。
4. 如何判斷重複? 如果狀態轉換圖是一顆樹，則永遠不會出現迴路，不需要判重；如果狀態轉換圖是一個圖（這時候是一個圖上的 BFS），則需要判重。
   (a) 如果是求最短路徑長度或一條路徑，則只需要讓“點”（即狀態）不重複出現，即可保證不出現迴路
   (b) 如果是求所有路徑，注意此時，狀態轉換圖是 DAG，即允許兩個父節點指向同一個子節點。具體實現時，每個節點要“**延遲**”加入到已訪問集合 `visited`，要等一層全部訪問完後，再加入到 `visited` 集合。
   (c) 具體實現
      i. 狀態是否存在完美哈希方案？即將狀態一一映射到整數，互相之間不會衝突。
      ii. 如果不存在，則需要使用通用的哈希表（自己實現或用標準庫，例如 `unordered_set`）來判重；自己實現哈希表的話，如果能夠預估狀態個數的上限，則可以開兩個數組，head 和 next，表示哈希表，參考第 §**??**節方案 2。

iii. 如果存在，則可以開一個大布爾數組，來判重，且此時可以精確計算出狀態總數，而不僅僅是預估上限。

5. 目標狀態是否已知？如果題目已經給出了目標狀態，可以帶來很大便利，這時候可以從起始狀態出發，正向廣搜；也可以從目標狀態出發，逆向廣搜；也可以同時出發，雙向廣搜。

9.4.3 代碼模板

廣搜需要一個隊列，用於一層一層擴展，一個 hashset，用於判重，一棵樹（只求長度時不需要），用於存儲整棵樹。

對於隊列，可以用 `queue`，也可以把 `vector` 當做隊列使用。當求長度時，有兩種做法：

1. 只用一個隊列，但在狀態結構體 `state_t` 裏增加一個整數字段 `level`，表示當前所在的層次，當碰到目標狀態，直接輸出 `level` 即可。這個方案，可以很容易的變成 A*算法，把 `queue` 替換為 `priority_queue` 即可。
2. 用兩個隊列，`current, next`，分別表示當前層次和下一層，另設一個全局整數 `level`，表示層數（也即路徑長度），當碰到目標狀態，輸出 `level` 即可。這個方案，狀態裏可以不存路徑長度，只需全局設置一個整數 `level`，比較節省內存；

對於 hashset，如果有完美哈希方案，用布爾數組 (`bool visited[STATE_MAX]` 或 `vector<bool> visited(STATE_MAX, false)`) 來表示；如果沒有，可以用 STL 裏的 `set` 或 `unordered_set`。

對於樹，如果用 STL，可以用 `unordered_map<state_t, state_t > father` 表示一顆樹，代碼非常簡潔。如果能夠預估狀態總數的上限（設為 STATE_MAX），可以用數組 (`state_t nodes[STATE_MAX]`)，即樹的雙親表示法來表示樹，效率更高，當然，需要寫更多代碼。

如何表示狀態

bfs_common.h

```
/** 狀態 */
struct state_t {
    int data1;  /** 狀態的數據，可以有多個字段. */
    int data2;  /** 狀態的數據，可以有多個字段. */
    // dataN;   /** 其他字段 */
    int action; /** 由父狀態移動到本狀態的動作，求動作序列時需要. */
    int level;  /** 所在的層次（從 0 開始），也即路徑長度-1，求路徑長度時需要；
                   不過，採用雙隊列時不需要本字段，只需全局設一個整數 */
    bool operator==(const state_t &other) const {
        return true;  // 根據具體問題實現
    }
};

// 定義 hash 函數

// 方法 1：模板特化，當 hash 函數只需要狀態本身，不需要其他數據時，用這個方法比較簡潔
namespace std {
template<> struct hash<state_t> {
```

```
    size_t operator()(const state_t & x) const {
        return 0; // 根據具體問題實現
    }
};
}

// 方法 2：函數對象，如果 hash 函數需要運行時數據，則用這種方法
class Hasher {
public:
    Hasher(int _m) : m(_m) {};
    size_t operator()(const state_t &s) const {
        return 0; // 根據具體問題實現
    }
private:
    int m; // 存放外面傳入的數據
};

/**
 * @brief 反向生成路徑，求一條路徑.
 * @param[in] father 樹
 * @param[in] target 目標節點
 * @return 從起點到 target 的路徑
 */
vector<state_t> gen_path(const unordered_map<state_t, state_t> &father,
        const state_t &target) {
    vector<state_t> path;
    path.push_back(target);

    for (state_t cur = target; father.find(cur) != father.end();
            cur = father.at(cur))
        path.push_back(cur);

    reverse(path.begin(), path.end());

    return path;
}

/**
 * 反向生成路徑，求所有路徑.
 * @param[in] father 存放了所有路徑的樹
 * @param[in] start 起點
 * @param[in] state 終點
 * @return 從起點到終點的所有路徑
 */
void gen_path(unordered_map<state_t, vector<state_t> > &father,
        const string &start, const state_t& state, vector<state_t> &path,
        vector<vector<state_t> > &result) {
    path.push_back(state);
    if (state == start) {
        if (!result.empty()) {
            if (path.size() < result[0].size()) {
                result.clear();
```

```
                result.push_back(path);
            } else if(path.size() == result[0].size()) {
                result.push_back(path);
            } else {
                // not possible
                throw std::logic_error("not possible to get here");
            }
        } else {
            result.push_back(path);
        }
        reverse(result.back().begin(), result.back().end());
    } else {
        for (const auto& f : father[state]) {
            gen_path(father, start, f, path, result);
        }
    }
    path.pop_back();
}
```

bfs_common.h

求最短路徑長度或一條路徑

單隊列的寫法

bfs_template.cpp

```
#include "bfs_common.h"

/**
 * @brief 廣搜, 只用一個隊列.
 * @param[in] start 起點
 * @param[in] data 輸入數據
 * @return 從起點到目標狀態的一條最短路徑
 */
vector<state_t> bfs(state_t &start, const vector<vector<int>> &grid) {
    queue<state_t> q; // 隊列
    unordered_set<state_t> visited; // 判重
    unordered_map<state_t, state_t> father; // 樹, 求路徑本身時才需要

    // 判斷狀態是否合法
    auto state_is_valid = [&](const state_t &s) { /*...*/ };

    // 判斷當前狀態是否為所求目標
    auto state_is_target = [&](const state_t &s) { /*...*/ };

    // 擴展當前狀態
    auto state_extend = [&](const state_t &s) {
        unordered_set<state_t> result;
        for (/*...*/) {
            const state_t new_state = /*...*/;
            if (state_is_valid(new_state) &&
                    visited.find(new_state) != visited.end()) {
                result.insert(new_state);
```

```
            }
        }
        return result;
    };

    assert (start.level == 0);
    q.push(start);
    while (!q.empty()) {
        // 千萬不能用 const auto&, pop() 會刪除元素,
        // 引用就變成了懸空引用
        const state_t state = q.front();
        q.pop();
        visited.insert(state);

        // 訪問節點
        if (state_is_target(state)) {
            return return gen_path(father, target); // 求一條路徑
            // return state.level + 1; // 求路徑長度
        }

        // 擴展節點
        vector<state_t> new_states = state_extend(state);
        for (const auto& new_state : new_states) {
            q.push(new_state);
            father[new_state] = state;  // 求一條路徑
            // visited.insert(state); // 優化：可以提前加入 visited 集合,
            // 從而縮小狀態擴展。這時 q 的含義略有變化, 裏面存放的是處理了一半
            // 的節點：已經加入了 visited, 但還沒有擴展。別忘記 while 循環開始
            // 前, 要加一行代碼, visited.insert(start)
        }
    }

    return vector<state_t>();
    //return 0;
}
```

bfs_template.cpp

雙隊列的寫法

bfs_template1.cpp

```
#include "bfs_common.h"

/**
 * @brief 廣搜, 使用兩個隊列.
 * @param[in] start 起點
 * @param[in] data 輸入數據
 * @return 從起點到目標狀態的一條最短路徑
 */
vector<state_t> bfs(const state_t &start, const type& data) {
    queue<state_t> next, current; // 當前層, 下一層
    unordered_set<state_t> visited; // 判重
    unordered_map<state_t, state_t> father; // 樹, 求路徑本身時才需要
```

```
    int level = -1;  // 層次

    // 判斷狀態是否合法
    auto state_is_valid = [&](const state_t &s) { /*...*/ };

    // 判斷當前狀態是否為所求目標
    auto state_is_target = [&](const state_t &s) { /*...*/ };

    // 擴展當前狀態
    auto state_extend = [&](const state_t &s) {
        unordered_set<state_t> result;
        for (/*...*/) {
            const state_t new_state = /*...*/;
            if (state_is_valid(new_state) &&
                    visited.find(new_state) != visited.end()) {
                result.insert(new_state);
            }
        }
        return result;
    };

    current.push(start);
    while (!current.empty()) {
        ++level;
        while (!current.empty()) {
            // 千萬不能用 const auto&, pop() 會刪除元素,
            // 引用就變成了懸空引用
            const auto state = current.front();
            current.pop();
            visited.insert(state);

            if (state_is_target(state)) {
                return return gen_path(father, state); // 求一條路徑
                // return state.level + 1; // 求路徑長度
            }

            const auto& new_states = state_extend(state);
            for (const auto& new_state : new_states) {
                next.push(new_state);
                father[new_state] = state;
                // visited.insert(state); // 優化：可以提前加入 visited 集合,
                // 從而縮小狀態擴展。這時 current 的含義略有變化, 裏面存放的是處
                // 理了一半的節點：已經加入了 visited, 但還沒有擴展。別忘記 while
                // 循環開始前, 要加一行代碼, visited.insert(start)
            }
        }
        swap(next, current); //!!! 交換兩個隊列
    }

    return vector<state_t>();
    // return 0;
}
```

bfs_template1.cpp

求所有路徑

單隊列

bfs_template.cpp

```
/**
 * @brief 廣搜, 使用一個隊列.
 * @param[in] start 起點
 * @param[in] data 輸入數據
 * @return 從起點到目標狀態的所有最短路徑
 */
vector<vector<state_t> > bfs(const state_t &start, const type& data) {
    queue<state_t> q;
    unordered_set<state_t> visited; // 判重
    unordered_map<state_t, vector<state_t> > father; // DAG

    auto state_is_valid = [&](const state_t& s) { /*...*/ };
    auto state_is_target = [&](const state_t &s) { /*...*/ };
    auto state_extend = [&](const state_t &s) {
        unordered_set<state_t> result;
        for (/*...*/) {
            const state_t new_state = /*...*/;
            if (state_is_valid(new_state)) {
                auto visited_iter = visited.find(new_state);

                if (visited_iter != visited.end()) {
                    if (visited_iter->level < new_state.level) {
                        // do nothing
                    } else if (visited_iter->level == new_state.level) {
                        result.insert(new_state);
                    } else { // not possible
                        throw std::logic_error("not possible to get here");
                    }
                } else {
                    result.insert(new_state);
                }
            }
        }

        return result;
    };

    vector<vector<string>> result;
    state_t start_state(start, 0);
    q.push(start_state);
    visited.insert(start_state);
    while (!q.empty()) {
        // 千萬不能用 const auto&, pop() 會刪除元素,
        // 引用就變成了懸空引用
        const auto state = q.front();
        q.pop();

        // 如果當前路徑長度已經超過當前最短路徑長度,
```

```
        // 可以中止對該路徑的處理，因為我們要找的是最短路徑
        if (!result.empty() && state.level + 1 > result[0].size()) break;

        if (state_is_target(state)) {
            vector<string> path;
            gen_path(father, start_state, state, path, result);
            continue;
        }
        // 必須挪到下面，比如同一層 A 和 B 兩個節點均指向了目標節點，
        // 那麼目標節點就會在 q 中出現兩次，輸出路徑就會翻倍
        // visited.insert(state);

        // 擴展節點
        const auto& new_states = state_extend(state);
        for (const auto& new_state : new_states) {
            if (visited.find(new_state) == visited.end()) {
                q.push(new_state);
            }
            visited.insert(new_state);
            father[new_state].push_back(state);
        }
    }

    return result;
}
```

bfs_template.cpp

雙隊列的寫法

bfs_template.cpp

```
#include "bfs_common.h"

/**
 * @brief 廣搜，使用兩個隊列.
 * @param[in] start 起點
 * @param[in] data 輸入數據
 * @return 從起點到目標狀態的所有最短路徑
 */
vector<vector<state_t> > bfs(const state_t &start, const type& data) {
    // 當前層，下一層，用 unordered_set 是為了去重，例如兩個父節點指向
    // 同一個子節點，如果用 vector，子節點就會在 next 裏出現兩次，其實此
    // 時 father 已經記錄了兩個父節點，next 裏重複出現兩次是沒必要的
    unordered_set<string> current, next;
    unordered_set<state_t> visited; // 判重
    unordered_map<state_t, vector<state_t> > father; // DAG

    int level = -1;  // 層次

    // 判斷狀態是否合法
    auto state_is_valid = [&](const state_t &s) { /*...*/ };

    // 判斷當前狀態是否為所求目標
    auto state_is_target = [&](const state_t &s) { /*...*/ };
```

```
    // 擴展當前狀態
    auto state_extend = [&](const state_t &s) {
        unordered_set<state_t> result;
        for (/*...*/) {
            const state_t new_state = /*...*/;
            if (state_is_valid(new_state) &&
                    visited.find(new_state) != visited.end()) {
                result.insert(new_state);
            }
        }
        return result;
    };

    vector<vector<state_t> > result;
    current.insert(start);
    while (!current.empty()) {
        ++ level;
        // 如果當前路徑長度已經超過當前最短路徑長度，可以中止對該路徑的
        // 處理，因為我們要找的是最短路徑
        if (!result.empty() && level+1 > result[0].size()) break;

        // 1. 延遲加入 visited，這樣才能允許兩個父節點指向同一個子節點
        // 2. 一股腦 current 全部加入 visited，是防止本層前一個節點擴展
        // 節點時，指向了本層後面尚未處理的節點，這條路徑必然不是最短的
        for (const auto& state : current)
            visited.insert(state);
        for (const auto& state : current) {
            if (state_is_target(state)) {
                vector<string> path;
                gen_path(father, path, start, state, result);
                continue;
            }

            const auto new_states = state_extend(state);
            for (const auto& new_state : new_states) {
                next.insert(new_state);
                father[new_state].push_back(state);
            }
        }

        current.clear();
        swap(current, next);
    }

    return result;
}
```

bfs_template.cpp

第 10 章
深度優先搜索

10.1 Palindrome Partitioning

描述

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

For example, given `s = "aab"`, Return

```
[
  ["aa","b"],
  ["a","a","b"]
]
```

分析

在每一步都可以判斷中間結果是否為合法結果，用回溯法。

一個長度為 n 的字符串，有 $n-1$ 個地方可以砍斷，每個地方可斷可不斷，因此複雜度為 $O(2^{n-1})$

深搜 1

```
//LeetCode, Palindrome Partitioning
// 時間複雜度 O(2^n)，空間複雜度 O(n)
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> path;  // 一個 partition 方案
        dfs(s, path, result, 0, 1);
        return result;
    }

    // prev 表示前一個隔板，start 表示當前隔板
    void dfs(string &s, vector<string>& path,
             vector<vector<string>> &result, size_t prev, size_t start) {
        if (start == s.size()) { // 最後一個隔板
            if (isPalindrome(s, prev, start - 1)) { // 必須使用
                path.push_back(s.substr(prev, start - prev));
```

```
                result.push_back(path);
                path.pop_back();
            }
            return;
        }
        // 不斷開
        dfs(s, path, result, prev, start + 1);
        // 如果 [prev, start-1] 是迴文, 則可以斷開, 也可以不斷開 (上一行已經做了)
        if (isPalindrome(s, prev, start - 1)) {
            // 斷開
            path.push_back(s.substr(prev, start - prev));
            dfs(s, path, result, start, start + 1);
            path.pop_back();
        }
    }

    bool isPalindrome(const string &s, int start, int end) {
        while (start < end && s[start] == s[end]) {
            ++start;
            --end;
        }
        return start >= end;
    }
};
```

深搜 2

另一種寫法，更加簡潔。這種寫法也在 Combination Sum, Combination Sum II 中出現過。

```
//LeetCode, Palindrome Partitioning
// 時間複雜度 O(2^n), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> result;
        vector<string> path;  // 一個 partition 方案
        DFS(s, path, result, 0);
        return result;
    }
    // 搜索必須以 s[start] 開頭的 partition 方案
    void DFS(string &s, vector<string>& path,
            vector<vector<string>> &result, int start) {
        if (start == s.size()) {
            result.push_back(path);
            return;
        }
        for (int i = start; i < s.size(); i++) {
            if (isPalindrome(s, start, i)) { // 從 i 位置砍一刀
                path.push_back(s.substr(start, i - start + 1));
                DFS(s, path, result, i + 1);  // 繼續往下砍
                path.pop_back(); // 撤銷上上行
            }
```

```
        }
    }
    bool isPalindrome(const string &s, int start, int end) {
        while (start < end && s[start] == s[end]) {
            ++start;
            --end;
        }
        return start >= end;
    }
};
```

動規

```
// LeetCode, Palindrome Partitioning
// 動規, 時間複雜度 O(n^2), 空間複雜度 O(1)
class Solution {
public:
    vector<vector<string> > partition(string s) {
        const int n = s.size();
        bool p[n][n]; // whether s[i,j] is palindrome
        fill_n(&p[0][0], n * n, false);
        for (int i = n - 1; i >= 0; --i)
            for (int j = i; j < n; ++j)
                p[i][j] = s[i] == s[j] && ((j - i < 2) || p[i + 1][j - 1]);

        vector<vector<string> > sub_palins[n]; // sub palindromes of s[0,i]
        for (int i = n - 1; i >= 0; --i) {
            for (int j = i; j < n; ++j)
                if (p[i][j]) {
                    const string palindrome = s.substr(i, j - i + 1);
                    if (j + 1 < n) {
                        for (auto v : sub_palins[j + 1]) {
                            v.insert(v.begin(), palindrome);
                            sub_palins[i].push_back(v);
                        }
                    } else {
                        sub_palins[i].push_back(vector<string> { palindrome });
                    }
                }
        }
        return sub_palins[0];
    }
};
```

相關題目

- Palindrome Partitioning II，見 §13.3

10.2 Unique Paths

描述

A robot is located at the top-left corner of a $m \times n$ grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

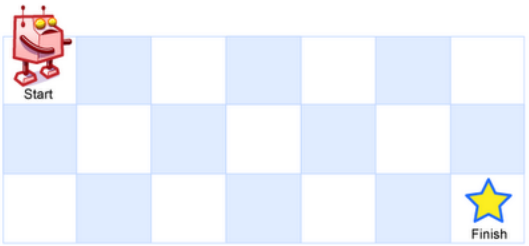


图 10-1 Above is a 3×7 grid. How many possible unique paths are there?

Note: m and n will be at most 100.

10.2.1 深搜

深搜，小集合可以過，大集合會超時

代碼

```
// LeetCode, Unique Paths
// 深搜，小集合可以過，大集合會超時
// 時間複雜度 O(n^4)，空間複雜度 O(n)
class Solution {
public:
    int uniquePaths(int m, int n) {
        if (m < 1 || n < 1) return 0; // 終止條件

        if (m == 1 && n == 1) return 1; // 收斂條件

        return uniquePaths(m - 1, n) + uniquePaths(m, n - 1);
    }
};
```

10.2.2 備忘錄法

給前面的深搜，加個緩存，就可以過大集合了。即備忘錄法。

代碼

```
// LeetCode, Unique Paths
// 深搜 + 緩存, 即備忘錄法
// 時間複雜度 O(n^2), 空間複雜度 O(n^2)
class Solution {
public:
    int uniquePaths(int m, int n) {
        // f[x][y] 表示 從 (0,0) 到 (x,y) 的路徑條數
        f = vector<vector<int> >(m, vector<int>(n, 0));
        f[0][0] = 1;
        return dfs(m - 1, n - 1);
    }
private:
    vector<vector<int> > f;  // 緩存

    int dfs(int x, int y) {
        if (x < 0 || y < 0) return 0; // 數據非法, 終止條件

        if (x == 0 && y == 0) return f[0][0]; // 回到起點, 收斂條件

        if (f[x][y] > 0) {
            return f[x][y];
        } else {
            return f[x][y] = dfs(x - 1, y) +  dfs(x, y - 1);
        }
    }
};
```

10.2.3 動規

既然可以用備忘錄法自頂向下解決, 也一定可以用動規自底向上解決。

設狀態為 `f[i][j]`, 表示從起點 $(1,1)$ 到達 (i,j) 的路線條數, 則狀態轉移方程為:

```
f[i][j]=f[i-1][j]+f[i][j-1]
```

代碼

```
// LeetCode, Unique Paths
// 動規, 滾動數組
// 時間複雜度 O(n^2), 空間複雜度 O(n)
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<int> f(n, 0);
        f[0] = 1;
        for (int i = 0; i < m; i++) {
            for (int j = 1; j < n; j++) {
                // 左邊的 f[j], 表示更新後的 f[j], 與公式中的 f[i][j] 對應
                // 右邊的 f[j], 表示老的 f[j], 與公式中的 f[i-1][j] 對應
                f[j] = f[j] + f[j - 1];
```

```
            }
        }
        return f[n - 1];
    }
};
```

10.2.4 數學公式

一個 m 行，n 列的矩陣，機器人從左上走到右下總共需要的步數是 $m+n-2$，其中向下走的步數是 $m-1$，因此問題變成了在 $m+n-2$ 個操作中，選擇 $m-1$ 個時間點向下走，選擇方式有多少種。即 C_{m+n-2}^{m-1}。

代碼

```
// LeetCode, Unique Paths
// 數學公式
class Solution {
public:
    typedef long long int64_t;
    // 求階乘, n!/(start-1)!, 即 n*(n-1)...start, 要求 n >= 1
    static int64_t factor(int n, int start = 1) {
        int64_t  ret = 1;
        for(int i = start; i <= n; ++i)
            ret *= i;
        return ret;
    }
    // 求組合數 C_n^k
    static int64_t combination(int n, int k) {
        // 常數優化
        if (k == 0) return 1;
        if (k == 1) return n;

        int64_t ret = factor(n, k+1);
        ret /= factor(n - k);
        return ret;
    }

    int uniquePaths(int m, int n) {
        // max 可以防止 n 和 k 差距過大, 從而防止 combination() 溢出
        return combination(m+n-2, max(m-1, n-1));
    }
};
```

相關題目

- Unique Paths II，見 §10.3
- Minimum Path Sum, 見 §13.8

10.3 Unique Paths II

描述

Follow up for ”Unique Paths”:

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

For example,

There is one obstacle in the middle of a 3×3 grid as illustrated below.

```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```

The total number of unique paths is 2.

Note: m and n will be at most 100.

10.3.1 備忘錄法

在上一題的基礎上改一下即可。相比動規，簡單得多。

代碼

```
// LeetCode, Unique Paths II
// 深搜 + 緩存, 即備忘錄法
class Solution {
public:
    int uniquePathsWithObstacles(const vector<vector<int> >& obstacleGrid) {
        const int m = obstacleGrid.size();
        const int n = obstacleGrid[0].size();
        if (obstacleGrid[0][0] || obstacleGrid[m - 1][n - 1]) return 0;

        f = vector<vector<int> >(m, vector<int>(n, 0));
        f[0][0] = obstacleGrid[0][0] ? 0 : 1;
        return dfs(obstacleGrid, m - 1, n - 1);
    }
private:
    vector<vector<int> > f;  // 緩存

    // @return 從 (0, 0) 到 (x, y) 的路徑總數
    int dfs(const vector<vector<int> >& obstacleGrid,
            int x, int y) {
        if (x < 0 || y < 0) return 0; // 數據非法, 終止條件

        // (x,y) 是障礙
        if (obstacleGrid[x][y]) return 0;
```

```
        if (x == 0 and y == 0) return f[0][0]; // 回到起點，收斂條件

        if (f[x][y] > 0) {
            return f[x][y];
        } else {
            return f[x][y] = dfs(obstacleGrid, x - 1, y) +
                dfs(obstacleGrid, x, y - 1);
        }
    }
};
```

10.3.2 動規

與上一題類似，但要特別注意第一列的障礙。在上一題中，第一列全部是 1，但是在這一題中不同，第一列如果某一行有障礙物，那麼後面的行全為 0。

代碼

```
// LeetCode, Unique Paths II
// 動規，滾動數組
// 時間複雜度 O(n^2)，空間複雜度 O(n)
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {
        const int m = obstacleGrid.size();
        const int n = obstacleGrid[0].size();
        if (obstacleGrid[0][0] || obstacleGrid[m-1][n-1]) return 0;

        vector<int> f(n, 0);
        f[0] = obstacleGrid[0][0] ? 0 : 1;

        for (int i = 0; i < m; i++) {
            f[0] = f[0] == 0 ? 0 : (obstacleGrid[i][0] ? 0 : 1);
            for (int j = 1; j < n; j++)
                f[j] = obstacleGrid[i][j] ? 0 : (f[j] + f[j - 1]);
        }

        return f[n - 1];
    }
};
```

相關題目

- Unique Paths，見 §10.2
- Minimum Path Sum, 見 §13.8

10.4 N-Queens

描述

The n-queens puzzle is the problem of placing n queens on an $n \times n$ chessboard such that no two queens attack each other.

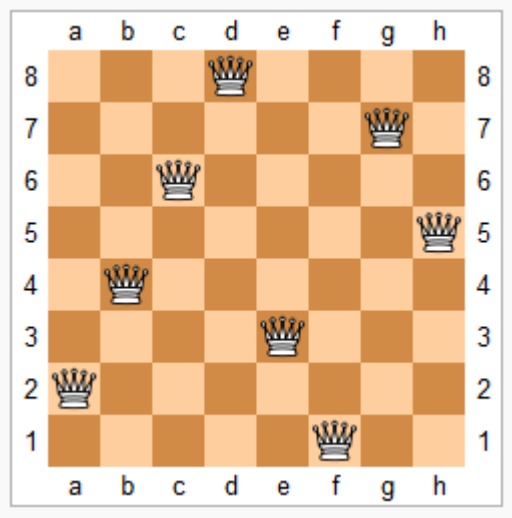


图 10-2 Eight Queens

Given an integer n, return all distinct solutions to the n-queens puzzle.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.

For example, There exist two distinct solutions to the 4-queens puzzle:

```
[
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

分析

經典的深搜題。

設置一個數組 vector<int> C(n, 0), C[i] 表示第 i 行皇后所在的列編號，即在位置 (i, C[i]) 上放了一個皇后，這樣用一個一維數組，就能記錄整個棋盤。

代碼 1

```
// LeetCode, N-Queens
// 深搜+剪枝
// 時間複雜度 O(n!*n), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<string> > solveNQueens(int n) {
        vector<vector<string> > result;
        vector<int> C(n, -1);  // C[i] 表示第 i 行皇后所在的列編號
        dfs(C, result, 0);
        return result;
    }
private:
    void dfs(vector<int> &C, vector<vector<string> > &result, int row) {
        const int N = C.size();
        if (row == N) { // 終止條件, 也是收斂條件, 意味着找到了一個可行解
            vector<string> solution;
            for (int i = 0; i < N; ++i) {
                string s(N, '.');
                for (int j = 0; j < N; ++j) {
                    if (j == C[i]) s[j] = 'Q';
                }
                solution.push_back(s);
            }
            result.push_back(solution);
            return;
        }

        for (int j = 0; j < N; ++j) {  // 擴展狀態, 一列一列的試
            const bool ok = isValid(C, row, j);
            if (!ok) continue;  // 剪枝, 如果非法, 繼續嘗試下一列
            // 執行擴展動作
            C[row] = j;
            dfs(C, result, row + 1);
            // 撤銷動作
            // C[row] = -1;
        }
    }

    /**
     * 能否在 (row, col) 位置放一個皇后.
     *
     * @param C 棋局
     * @param row 當前正在處理的行, 前面的行都已經放了皇后了
     * @param col 當前列
     * @return 能否放一個皇后
     */
    bool isValid(const vector<int> &C, int row, int col) {
```

```
        for (int i = 0; i < row; ++i) {
            // 在同一列
            if (C[i] == col) return false;
            // 在同一對角線上
            if (abs(i - row) == abs(C[i] - col)) return false;
        }
        return true;
    }
};
```

代碼 2

```
// LeetCode, N-Queens
// 深搜+剪枝
// 時間複雜度 O(n!), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<string> > solveNQueens(int n) {
        this->columns = vector<bool>(n, false);
        this->main_diag = vector<bool>(2 * n - 1, false);
        this->anti_diag = vector<bool>(2 * n - 1, false);

        vector<vector<string> > result;
        vector<int> C(n, -1);  // C[i] 表示第 i 行皇后所在的列編號
        dfs(C, result, 0);
        return result;
    }
private:
    // 這三個變量用於剪枝
    vector<bool> columns;  // 表示已經放置的皇后佔據了哪些列
    vector<bool> main_diag;  // 佔據了哪些主對角線
    vector<bool> anti_diag;  // 佔據了哪些副對角線

    void dfs(vector<int> &C, vector<vector<string> > &result, int row) {
        const int N = C.size();
        if (row == N) { // 終止條件, 也是收斂條件, 意味着找到了一個可行解
            vector<string> solution;
            for (int i = 0; i < N; ++i) {
                string s(N, '.');
                for (int j = 0; j < N; ++j) {
                    if (j == C[i]) s[j] = 'Q';
                }
                solution.push_back(s);
            }
            result.push_back(solution);
            return;
        }

        for (int j = 0; j < N; ++j) {  // 擴展狀態, 一列一列的試
            const bool ok = !columns[j] && !main_diag[row - j + N - 1]  &&
                    !anti_diag[row + j];
            if (!ok) continue;  // 剪枝, 如果非法, 繼續嘗試下一列
```

```
            // 執行擴展動作
            C[row] = j;
            columns[j] = main_diag[row - j + N - 1] = anti_diag[row + j] = true;
            dfs(C, result, row + 1);
            // 撤銷動作
            // C[row] = -1;
            columns[j] = main_diag[row - j + N - 1] = anti_diag[row + j] = false;
        }
    }
};
```

相關題目

- N-Queens II，見 §10.5

10.5 N-Queens II

描述

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.

分析

只需要輸出解的個數，不需要輸出所有解，代碼要比上一題簡化很多。設一個全局計數器，每找到一個解就增 1。

代碼 1

```
// LeetCode, N-Queens II
// 深搜+剪枝
// 時間複雜度 O(n!*n), 空間複雜度 O(n)
class Solution {
public:
    int totalNQueens(int n) {
        this->count = 0;

        vector<int> C(n, 0);  // C[i] 表示第 i 行皇后所在的列編號
        dfs(C, 0);
        return this->count;
    }
private:
    int count; // 解的個數

    void dfs(vector<int> &C, int row) {
        const int N = C.size();
        if (row == N) { // 終止條件，也是收斂條件，意味着找到了一個可行解
            ++this->count;
            return;
        }
```

```
        for (int j = 0; j < N; ++j) {  // 擴展狀態, 一列一列的試
            const bool ok = isValid(C, row, j);
            if (!ok) continue;  // 剪枝：如果合法, 繼續遞歸
            // 執行擴展動作
            C[row] = j;
            dfs(C, row + 1);
            // 撤銷動作
            // C[row] = -1;
        }
    }
    /**
     * 能否在 (row, col) 位置放一個皇后.
     *
     * @param C 棋局
     * @param row 當前正在處理的行, 前面的行都已經放了皇后了
     * @param col 當前列
     * @return 能否放一個皇后
     */
    bool isValid(const vector<int> &C, int row, int col) {
        for (int i = 0; i < row; ++i) {
            // 在同一列
            if (C[i] == col) return false;
            // 在同一對角線上
            if (abs(i - row) == abs(C[i] - col)) return false;
        }
        return true;
    }
};
```

代碼 2

```
// LeetCode, N-Queens II
// 深搜+剪枝
// 時間複雜度 O(n!), 空間複雜度 O(n)
class Solution {
public:
    int totalNQueens(int n) {
        this->count = 0;
        this->columns = vector<bool>(n, false);
        this->main_diag = vector<bool>(2 * n - 1, false);
        this->anti_diag = vector<bool>(2 * n - 1, false);

        vector<int> C(n, 0);  // C[i] 表示第 i 行皇后所在的列編號
        dfs(C, 0);
        return this->count;
    }
private:
    int count; // 解的個數
    // 這三個變量用於剪枝
    vector<bool> columns;  // 表示已經放置的皇后佔據了哪些列
    vector<bool> main_diag;  // 佔據了哪些主對角線
```

```
    vector<bool> anti_diag;  // 佔據了哪些副對角線

    void dfs(vector<int> &C, int row) {
        const int N = C.size();
        if (row == N) { // 終止條件，也是收斂條件，意味着找到了一個可行解
            ++this->count;
            return;
        }

        for (int j = 0; j < N; ++j) {  // 擴展狀態，一列一列的試
            const bool ok = !columns[j] &&
                    !main_diag[row - j + N] &&
                    !anti_diag[row + j];
            if (!ok) continue;  // 剪枝：如果合法，繼續遞歸
            // 執行擴展動作
            C[row] = j;
            columns[j] = main_diag[row - j + N] =
                    anti_diag[row + j] = true;
            dfs(C, row + 1);
            // 撤銷動作
            // C[row] = -1;
            columns[j] = main_diag[row - j + N] =
                    anti_diag[row + j] = false;
        }
    }
};
```

相關題目

- N-Queens，見 §10.4

10.6 Restore IP Addresses

描述

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example: Given `"25525511135"`,

return

"255.255.11.135", "255.255.111.35"

. (Order does not matter)

分析

必須要走到底部才能判斷解是否合法，深搜。

代碼

```
// LeetCode, Restore IP Addresses
// 時間複雜度 O(n^4)，空間複雜度 O(n)
```

```
class Solution {
public:
    vector<string> restoreIpAddresses(const string& s) {
        vector<string> result;
        vector<string> ip; // 存放中間結果
        dfs(s, ip, result, 0);
        return result;
    }

    /**
     * @brief 解析字符串
     * @param[in] s 字符串, 輸入數據
     * @param[out] ip 存放中間結果
     * @param[out] result 存放所有可能的 IP 地址
     * @param[in] start 當前正在處理的 index
     * @return 無
     */
    void dfs(string s, vector<string>& ip, vector<string> &result,
             size_t start) {
        if (ip.size() == 4 && start == s.size()) {  // 找到一個合法解
            result.push_back(ip[0] + '.' + ip[1] + '.' + ip[2] + '.' + ip[3]);
            return;
        }

        if (s.size() - start > (4 - ip.size()) * 3)
            return;  // 剪枝
        if (s.size() - start < (4 - ip.size()))
            return;  // 剪枝

        int num = 0;
        for (size_t i = start; i < start + 3; i++) {
            num = num * 10 + (s[i] - '0');

            if (num < 0 || num > 255) continue;  // 剪枝

            ip.push_back(s.substr(start, i - start + 1));
            dfs(s, ip, result, i + 1);
            ip.pop_back();

            if (num == 0) break;  // 不允許前綴 0, 但允許單個 0
        }
    }
};
```

相關題目

- 無

10.7 Combination Sum

描述

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

The same repeated number may be chosen from C unlimited number of times.

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination $(a_1, a_2, ..., a_k)$ must be in non-descending order. (ie, $a_1 \leq a_2 \leq ... \leq a_k$).
- The solution set must not contain duplicate combinations.

For example, given candidate set `2,3,6,7` and target `7`, A solution set is:

```
[7]
[2, 2, 3]
```

分析

無

代碼

```
// LeetCode, Combination Sum
// 時間複雜度 O(n!), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > combinationSum(vector<int> &nums, int target) {
        sort(nums.begin(), nums.end());
        vector<vector<int> > result; // 最終結果
        vector<int> path; // 中間結果
        dfs(nums, path, result, target, 0);
        return result;
    }

private:
    void dfs(vector<int>& nums, vector<int>& path, vector<vector<int> > &result,
            int gap, int start) {
        if (gap == 0) {  // 找到一個合法解
            result.push_back(path);
            return;
        }
        for (size_t i = start; i < nums.size(); i++) { // 擴展狀態
            if (gap < nums[i]) return; // 剪枝

            path.push_back(nums[i]); // 執行擴展動作
            dfs(nums, path, result, gap - nums[i], i);
            path.pop_back();  // 撤銷動作
```

```
        }
    }
};
```

相關題目

- Combination Sum II，見 §10.8

10.8 Combination Sum II

描述

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.

Each number in C may only be used once in the combination.

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination $(a_1, a_2, ..., a_k)$ must be in non-descending order. (ie, $a_1 > a_2 > ... > a_k$).
- The solution set must not contain duplicate combinations.

For example, given candidate set `10,1,2,7,6,1,5` and target `8`, A solution set is:

```
[1, 7]
[1, 2, 5]
[2, 6]
[1, 1, 6]
```

分析

無

代碼

```
// LeetCode, Combination Sum II
// 時間複雜度 O(n!), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > combinationSum2(vector<int> &nums, int target) {
        sort(nums.begin(), nums.end()); // 跟第 50 行配合,
                                        // 確保每個元素最多隻用一次
        vector<vector<int> > result;
        vector<int> path;
        dfs(nums, path, result, target, 0);
        return result;
    }
private:
    // 使用 nums[start, nums.size()) 之間的元素, 能找到的所有可行解
    static void dfs(const vector<int> &nums, vector<int> &path,
```

```
            vector<vector<int> > &result, int gap, int start) {
        if (gap == 0) {  //  找到一個合法解
            result.push_back(path);
            return;
        }

        int previous = -1;
        for (size_t i = start; i < nums.size(); i++) {
            // 如果上一輪循環已經使用了 nums[i]，則本次循環就不能再選 nums[i]，
            // 確保 nums[i] 最多隻用一次
            if (previous == nums[i]) continue;

            if (gap < nums[i]) return;  // 剪枝

            previous = nums[i];

            path.push_back(nums[i]);
            dfs(nums, path, result, gap - nums[i], i + 1);
            path.pop_back();  // 恢復環境
        }
    }
};
```

相關題目

- Combination Sum，見 §10.7

10.9 Generate Parentheses

描述

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given $n = 3$, a solution set is:

```
"((()))", "(()())", "(())()", "()(())", "()()()"
```

分析

小括號串是一個遞歸結構，跟單鏈表、二叉樹等遞歸結構一樣，首先想到用遞歸。

一步步構造字符串。當左括號出現次數 $< n$ 時，就可以放置新的左括號。當右括號出現次數小於左括號出現次數時，就可以放置新的右括號。

代碼 1

```
// LeetCode, Generate Parentheses
// 時間複雜度 O(TODO)，空間複雜度 O(n)
class Solution {
public:
    vector<string> generateParenthesis(int n) {
        vector<string> result;
```

```
        string path;
        if (n > 0) generate(n, path, result, 0, 0);
        return result;
    }
    // l 表示 ( 出現的次數, r 表示 ) 出現的次數
    void generate(int n, string& path, vector<string> &result, int l, int r) {
        if (l == n) {
            string s(path);
            result.push_back(s.append(n - r, ')'));
            return;
        }

        path.push_back('(');
        generate(n, path, result, l + 1, r);
        path.pop_back();

        if (l > r) {
            path.push_back(')');
            generate(n, path, result, l, r + 1);
            path.pop_back();
        }
    }
};
```

代碼 2

另一種遞歸寫法，更加簡潔。

```
// LeetCode, Generate Parentheses
// @author 連城 (http://weibo.com/lianchengzju)
class Solution {
public:
    vector<string> generateParenthesis (int n) {
        if (n == 0) return vector<string>(1, "");
        if (n == 1) return vector<string> (1, "()");
        vector<string> result;

        for (int i = 0; i < n; ++i)
            for (auto inner : generateParenthesis (i))
                for (auto outer : generateParenthesis (n - 1 - i))
                    result.push_back ("(" + inner + ")" + outer);

        return result;
    }
};
```

相關題目

- Valid Parentheses, 見 §4.1.1
- Longest Valid Parentheses, 見 §4.1.2

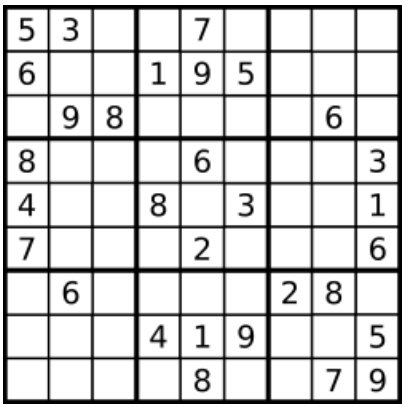
10.10　Sudoku Solver

描述

Write a program to solve a Sudoku puzzle by filling the empty cells.

Empty cells are indicated by the character `'.'`.

You may assume that there will be only one unique solution.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

图 10-3　A sudoku puzzle...

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

图 10-4　...and its solution numbers marked in red

分析

無。

代碼

```
// LeetCode, Sudoku Solver
// 時間複雜度 O(9^4), 空間複雜度 O(1)
class Solution {
public:
    bool solveSudoku(vector<vector<char> > &board) {
        for (int i = 0; i < 9; ++i)
            for (int j = 0; j < 9; ++j) {
                if (board[i][j] == '.') {
                    for (int k = 0; k < 9; ++k) {
                        board[i][j] = '1' + k;
                        if (isValid(board, i, j) && solveSudoku(board))
                            return true;
                        board[i][j] = '.';
                    }
                    return false;
                }
            }
        return true;
    }
private:
    // 檢查 (x, y) 是否合法
    bool isValid(const vector<vector<char> > &board, int x, int y) {
        int i, j;
        for (i = 0; i < 9; i++) // 檢查 y 列
            if (i != x && board[i][y] == board[x][y])
                return false;
        for (j = 0; j < 9; j++) // 檢查 x 行
            if (j != y && board[x][j] == board[x][y])
                return false;
        for (i = 3 * (x / 3); i < 3 * (x / 3 + 1); i++)
            for (j = 3 * (y / 3); j < 3 * (y / 3 + 1); j++)
                if ((i != x || j != y) && board[i][j] == board[x][y])
                    return false;
        return true;
    }
};
```

相關題目

- Valid Sudoku, 見 §2.1.15

10.11 Word Search

描述

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where `"adjacent"` cells are those horizontally or vertically neighbouring. The same letter cell may not be used more than once.

For example, Given board =

```
[
  ["ABCE"],
  ["SFCS"],
  ["ADEE"]
]
```

word = "ABCCED", -> returns true,

word = "SEE", -> returns true,

word = "ABCB", -> returns false.

分析

無。

代碼

```
// LeetCode, Word Search
// 深搜, 遞歸
// 時間複雜度 O(n^2*m^2), 空間複雜度 O(n^2)
class Solution {
public:
    bool exist(const vector<vector<char> > &board, const string& word) {
        const int m = board.size();
        const int n = board[0].size();
        vector<vector<bool> > visited(m, vector<bool>(n, false));
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j)
                if (dfs(board, word, 0, i, j, visited))
                    return true;
        return false;
    }
private:
    static bool dfs(const vector<vector<char> > &board, const string &word,
            int index, int x, int y, vector<vector<bool> > &visited) {
        if (index == word.size())
            return true; // 收斂條件

        if (x < 0 || y < 0 || x >= board.size() || y >= board[0].size())
            return false;  // 越界, 終止條件

        if (visited[x][y]) return false; // 已經訪問過, 剪枝

        if (board[x][y] != word[index]) return false; // 不相等, 剪枝

        visited[x][y] = true;
        bool ret = dfs(board, word, index + 1, x - 1, y, visited) || // 上
                dfs(board, word, index + 1, x + 1, y, visited)    || // 下
                dfs(board, word, index + 1, x, y - 1, visited)    || // 左
                dfs(board, word, index + 1, x, y + 1, visited);      // 右
        visited[x][y] = false;
        return ret;
```

```
    }
};
```

相關題目

- 無

10.12 小結

10.12.1 適用場景

輸入數據：如果是遞歸數據結構，如單鏈表，二叉樹，集合，則百分之百可以用深搜；如果是非遞歸數據結構，如一維數組，二維數組，字符串，圖，則概率小一些。

狀態轉換圖：樹或者圖。

求解目標：必須要走到最深（例如對於樹，必須要走到葉子節點）才能得到一個解，這種情況適合用深搜。

10.12.2 思考的步驟

1. 是求路徑條數，還是路徑本身（或動作序列）？深搜最常見的三個問題，求可行解的總數，求一個可行解，求所有可行解。
   (a) 如果是路徑條數，則不需要存儲路徑。
   (b) 如果是求路徑本身，則要用一個數組 `path[]` 存儲路徑。跟寬搜不同，寬搜雖然最終求的也是一條路徑，但是需要存儲擴展過程中的所有路徑，在沒找到答案之前所有路徑都不能放棄；而深搜，在搜索過程中始終只有一條路徑，因此用一個數組就足夠了。
2. 只要求一個解，還是要求所有解？如果只要求一個解，那找到一個就可以返回；如果要求所有解，找到了一個後，還要繼續擴展，直到遍歷完。廣搜一般只要求一個解，因而不需要考慮這個問題（廣搜當然也可以求所有解，這時需要擴展到所有葉子節點，相當於在內存中存儲整個狀態轉換圖，非常佔內存，因此廣搜不適合解這類問題）。
3. 如何表示狀態？即一個狀態需要存儲哪些必要的數據，才能夠完整提供如何擴展到下一步狀態的所有信息。跟廣搜不同，深搜的慣用寫法，不是把數據記錄在狀態 `struct` 裏，而是添加函數參數（有時為了節省遞歸堆棧，用全局變量），`struct` 裏的字段與函數參數一一對應。
4. 如何擴展狀態？這一步跟上一步相關。狀態裏記錄的數據不同，擴展方法就不同。對於固定不變的數據結構（一般題目直接給出，作為輸入數據），如二叉樹，圖等，擴展方法很簡單，直接往下一層走，對於隱式圖，要先在第 1 步裏想清楚狀態所帶的數據，想清楚了這點，那如何擴展就很簡單了。
5. 終止條件是什麼？終止條件是指到了不能擴展的末端節點。對於樹，是葉子節點，對於圖或隱式圖，是出度為 0 的節點。

6. 收斂條件是什麼？收斂條件是指找到了一個合法解的時刻。如果是正向深搜（父狀態處理完了才進行遞歸，即父狀態不依賴子狀態，遞歸語句一定是在最後，尾遞歸），則是指是否達到目標狀態；如果是逆向深搜（處理父狀態時需要先知道子狀態的結果，此時遞歸語句不在最後），則是指是否到達初始狀態。

   由於很多時候終止條件和收斂條件是是合二為一的，因此很多人不區分這兩種條件。仔細區分這兩種條件，還是很有必要的。

   為了判斷是否到了收斂條件，要在函數接口裏用一個參數記錄當前的位置（或距離目標還有多遠）。如果是求一個解，直接返回這個解；如果是求所有解，要在這裏收集解，即把第一步中表示路徑的數組 `path[]` 複製到解集合裏。

7. 關於判重

   (a) 是否需要判重？如果狀態轉換圖是一棵樹，則不需要判重，因為在遍歷過程中不可能重複；如果狀態轉換圖是一個 DAG，則需要判重。這一點跟 BFS 不一樣，BFS 的狀態轉換圖總是 DAG，必須要判重。

   (b) 怎樣判重？跟廣搜相同，見第 §9.4 節。同時，DAG 說明存在重疊子問題，此時可以用緩存加速，見第 8 步。

8. 如何加速？

   (a) 剪枝。深搜一定要好好考慮怎麼剪枝，成本小收益大，加幾行代碼，就能大大加速。這裏沒有通用方法，只能具體問題具體分析，要充分觀察，充分利用各種信息來剪枝，在中間節點提前返回。

   (b) 緩存。

      i. 前提條件：狀態轉換圖是一個 DAG。DAG=>存在重疊子問題=>子問題的解會被重複利用，用緩存自然會有加速效果。如果依賴關係是樹狀的（例如樹，單鏈表等），沒必要加緩存，因為子問題只會一層層往下，用一次就再也不會用到，加了緩存也沒什麼加速效果。

      ii. 具體實現：可以用數組或 HashMap。維度簡單的，用數組；維度複雜的，用 HashMap，C++有 `map`，C++ 11 以後有 `unordered_map`，比 `map` 快。

拿到一個題目，當感覺它適合用深搜解決時，在心裏面把上面 8 個問題默默回答一遍，代碼基本上就能寫出來了。對於樹，不需要回答第 5 和第 8 個問題。如果讀者對上面的經驗總結看不懂或感覺“不實用”，很正常，因為這些經驗總結是我做了很多題目後總結出來的，從思維的發展過程看，“經驗總結”要晚於感性認識，所以這時候建議讀者先做做前面的題目，積累一定的感性認識後，再回過頭來看這一節的總結，一定會有共鳴。

10.12.3 代碼模板

dfs_template.cpp

```
/**
 * dfs 模板.
 * @param[in] input 輸入數據指針
 * @param[out] path 當前路徑，也是中間結果
 * @param[out] result 存放最終結果
 * @param[inout] cur or gap 標記當前位置或距離目標的距離
 * @return 路徑長度，如果是求路徑本身，則不需要返回長度
 */
void dfs(type &input, type &path, type &result, int cur or gap) {
    if (數據非法) return 0;   // 終止條件
    if (cur == input.size()) { // 收斂條件
    // if (gap == 0) {
        將 path 放入 result
    }

    if (可以剪枝) return;

    for(...) { // 執行所有可能的擴展動作
        執行動作，修改 path
        dfs(input, step + 1 or gap--, result);
        恢復 path
    }
}
```

dfs_template.cpp

10.12.4 深搜與回溯法的區別

深搜 (Depth-first search, DFS) 的定義見 http://en.wikipedia.org/wiki/Depth_first_search，回溯法 (backtracking) 的定義見 http://en.wikipedia.org/wiki/Backtracking

回溯法 = 深搜 + 剪枝。一般大家用深搜時，或多或少會剪枝，因此深搜與回溯法沒有什麼不同，可以在它們之間畫上一個等號。本書同時使用深搜和回溯法兩個術語，但讀者可以認為二者等價。

深搜一般用遞歸 (recursion) 來實現，這樣比較簡潔。

深搜能夠在候選答案生成到一半時，就進行判斷，拋棄不滿足要求的答案，所以深搜比暴力搜索法要快。

10.12.5 深搜與遞歸的區別

深搜經常用遞歸 (recursion) 來實現，二者常常同時出現，導致很多人誤以為他倆是一個東西。

深搜，是邏輯意義上的算法，遞歸，是一種物理意義上的實現，它和迭代 (iteration) 是對應的。深搜，可以用遞歸來實現，也可以用棧來實現；而遞歸，一般總是用來實現深搜。可以說，**遞歸一定是深搜，深搜不一定用遞歸**。

遞歸有兩種加速策略，一種是**剪枝 (prunning)**，對中間結果進行判斷，提前返回；一種是**緩存**，緩存中間結果，防止重複計算，用空間換時間。

其實，遞歸+緩存，就是 memoization。所謂 **memoization**（翻譯為備忘錄法，見第 §**??**節），就

是”top-down with cache”（自頂向下+緩存），它是 Donald Michie 在 1968 年創造的術語，表示一種優化技術，在 top-down 形式的程序中，使用緩存來避免重複計算，從而達到加速的目的。

memoization 不一定用遞歸，就像深搜不一定用遞歸一樣，可以在迭代 (iterative) 中使用 memoization 。**遞歸也不一定用 memoization**，可以用 memoization 來加速，但不是必須的。只有當遞歸使用了緩存，它才是 memoization 。

既然遞歸一定是深搜，為什麼很多書籍都同時使用這兩個術語呢？在遞歸味道更濃的地方，一般用遞歸這個術語，在深搜更濃的場景下，用深搜這個術語，讀者心裏要弄清楚他倆大部分時候是一回事。在單鏈表、二叉樹等遞歸數據結構上，遞歸的味道更濃，這時用遞歸這個術語；在圖、隱式圖等數據結構上，深搜的味道更濃，這時用深搜這個術語。

第 11 章
分治法

11.1 Pow(x,n)

描述

Implement pow(x, n).

分析

二分法，$x^n = x^{n/2} \times x^{n/2} \times x^{n\%2}$

代碼

```
//LeetCode, Pow(x, n)
// 二分法, $x^n = x^{n/2} * x^{n/2} * x^{n\%2}$
// 時間複雜度 O(logn), 空間複雜度 O(1)
class Solution {
public:
    double myPow(double x, int n) {
        if (n < 0) return 1.0 / power(x, -n);
        else return power(x, n);
    }
private:
    double power(double x, int n) {
        if (n == 0) return 1;
        double v = power(x, n / 2);
        if (n % 2 == 0) return v * v;
        else return v * v * x;
    }
};
```

相關題目

- Sqrt(x)，見 §11.2

11.2 Sqrt(x)

描述

Implement int `sqrt(int x)`.

Compute and return the square root of `x`.

分析

二分查找

代碼

```
// LeetCode, Sqrt(x)
// 二分查找
// 時間複雜度 O(logn), 空間複雜度 O(1)
class Solution {
public:
    int mySqrt(int x) {
        int left = 1, right = x / 2;
        int last_mid;  // 記錄最近一次 mid

        if (x < 2) return x;

        while(left <= right) {
            const int mid = left + (right - left) / 2;
            if(x / mid > mid) { // 不要用 x > mid * mid, 會溢出
                left = mid + 1;
                last_mid = mid;
            } else if(x / mid < mid) {
                right = mid - 1;
            } else {
                return mid;
            }
        }
        return last_mid;
    }
};
```

相關題目

- Pow(x)，見 §11.1

第 12 章
貪心法

12.1 Jump Game

描述

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.

Determine if you are able to reach the last index.

For example:

`A = [2,3,1,1,4]`, return true.

`A = [3,2,1,0,4]`, return false.

分析

由於每層最多可以跳 `A[i]` 步，也可以跳 0 或 1 步，因此如果能到達最高層，則說明每一層都可以到達。有了這個條件，說明可以用貪心法。

思路一：正向，從 0 出發，一層一層網上跳，看最後能不能超過最高層，能超過，說明能到達，否則不能到達。

思路二：逆向，從最高層下樓梯，一層一層下降，看最後能不能下降到第 0 層。

思路三：如果不敢用貪心，可以用動規，設狀態為 `f[i]`，表示從第 0 層出發，走到 `A[i]` 時剩餘的最大步數，則狀態轉移方程為：

$$f[i] = max(f[i-1], A[i-1]) - 1, i > 0$$

代碼 1

```
// LeetCode, Jump Game
// 思路 1，時間複雜度 O(n)，空間複雜度 O(1)
class Solution {
public:
    bool canJump(const vector<int>& nums) {
        int reach = 1; // 最右能跳到哪裏
        for (int i = 0; i < reach && reach < nums.size(); ++i)
            reach = max(reach,  i + 1 + nums[i]);
```

```
        return reach >= nums.size();
    }
};
```

代碼 2

```
// LeetCode, Jump Game
// 思路 2, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    bool canJump (const vector<int>& nums) {
        if (nums.empty()) return true;
        // 逆向下樓梯, 最左能下降到第幾層
        int left_most = nums.size() - 1;

        for (int i = nums.size() - 2; i >= 0; --i)
            if (i + nums[i] >= left_most)
                left_most = i;

        return left_most == 0;
    }
};
```

代碼 3

```
// LeetCode, Jump Game
// 思路三, 動規, 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    bool canJump(const vector<int>& nums) {
        vector<int> f(nums.size(), 0);
        f[0] = 0;
        for (int i = 1; i < nums.size(); i++) {
            f[i] = max(f[i - 1], nums[i - 1]) - 1;
            if (f[i] < 0) return false;;
        }
        return f[nums.size() - 1] >= 0;
    }
};
```

相關題目

- Jump Game II，見 §12.2

12.2 Jump Game II

描述

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.

For example: Given array `A = [2,3,1,1,4]`

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.)

分析

貪心法。

代碼 1

```
// LeetCode, Jump Game II
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int jump(const vector<int>& nums) {
        int step = 0; // 最小步數
        int left = 0;
        int right = 0;  // [left, right] 是當前能覆蓋的區間
        if (nums.size() == 1) return 0;

        while (left <= right) { // 嘗試從每一層跳最遠
            ++step;
            const int old_right = right;
            for (int i = left; i <= old_right; ++i) {
                int new_right = i + nums[i];
                if (new_right >= nums.size() - 1) return step;

                if (new_right > right) right = new_right;
            }
            left = old_right + 1;
        }
        return 0;
    }
};
```

代碼 2

```
// LeetCode, Jump Game II
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int jump(const vector<int>& nums) {
        int result = 0;
        // the maximum distance that has been reached
        int last = 0;
        // the maximum distance that can be reached by using "ret+1" steps
        int cur = 0;
        for (int i = 0; i < nums.size(); ++i) {
            if (i > last) {
```

```
                last = cur;
                ++result;
            }
            cur = max(cur, i + nums[i]);
        }

        return result;
    }
};
```

相關題目

- Jump Game，見 §12.1

12.3 Best Time to Buy and Sell Stock

描述

Say you have an array for which the i-th element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

分析

貪心法，分別找到價格最低和最高的一天，低進高出，注意最低的一天要在最高的一天之前。

把原始價格序列變成差分序列，本題也可以做是最大 m 子段和，$m = 1$。

代碼

```
// LeetCode, Best Time to Buy and Sell Stock
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        if (prices.size() < 2) return 0;
        int profit = 0;  // 差價, 也就是利潤
        int cur_min = prices[0]; // 當前最小

        for (int i = 1; i < prices.size(); i++) {
            profit = max(profit, prices[i] - cur_min);
            cur_min = min(cur_min, prices[i]);
        }
        return profit;
    }
};
```

相關題目

- Best Time to Buy and Sell Stock II，見 §12.4

- Best Time to Buy and Sell Stock III，見 §13.5

12.4 Best Time to Buy and Sell Stock II

描述

Say you have an array for which the i-th element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

分析

貪心法，低進高出，把所有正的價格差價相加起來。

把原始價格序列變成差分序列，本題也可以做是最大 m 子段和，$m =$ 數組長度。

代碼

```
// LeetCode, Best Time to Buy and Sell Stock II
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        int sum = 0;
        for (int i = 1; i < prices.size(); i++) {
            int diff = prices[i] - prices[i - 1];
            if (diff > 0) sum += diff;
        }
        return sum;
    }
};
```

相關題目

- Best Time to Buy and Sell Stock，見 §12.3
- Best Time to Buy and Sell Stock III，見 §13.5

12.5 Longest Substring Without Repeating Characters

描述

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for `"abcabcbb"` is `"abc"`, which the length is 3. For `"bbbbb"` the longest substring is `"b"`, with the length of 1.

分析

假設子串裏含有重複字符，則父串一定含有重複字符，單個子問題就可以決定父問題，因此可以用貪心法。跟動規不同，動規裏，單個子問題只能影響父問題，不足以決定父問題。

從左往右掃描，當遇到重複字母時，以上一個重複字母的 index+1，作為新的搜索起始位置，直到最後一個字母，複雜度是 $O(n)$。如圖 12-1 所示。

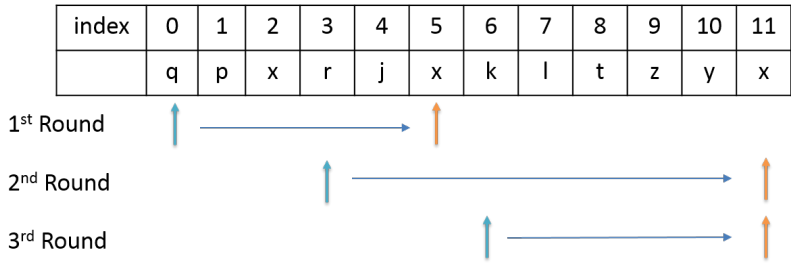


图 12-1 不含重複字符的最長子串

代碼

```
// LeetCode, Longest Substring Without Repeating Characters
// 時間複雜度 O(n), 空間複雜度 O(1)
// 考慮非字母的情況
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        const int ASCII_MAX = 255;
        int last[ASCII_MAX]; // 記錄字符上次出現過的位置
        int start = 0; // 記錄當前子串的起始位置

        fill(last, last + ASCII_MAX, -1); // 0 也是有效位置, 因此初始化為-1
        int max_len = 0;
        for (int i = 0; i < s.size(); i++) {
            if (last[s[i]] >= start) {
                max_len = max(i - start, max_len);
                start = last[s[i]] + 1;
            }
            last[s[i]] = i;
        }
        return max((int)s.size() - start, max_len);  // 別忘了最後一次, 例如"abcd"
    }
};
```

相關題目

- 無

12.6 Container With Most Water

描述

Given n non-negative integers $a_1, a_2, ..., a_n$, where each represents a point at coordinate (i, a_i). n vertical lines are drawn such that the two endpoints of line i is at (i, a_i) and $(i, 0)$. Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container.

分析

每個容器的面積，取決於最短的木板。

代碼

```
// LeetCode, Container With Most Water
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int maxArea(vector<int> &height) {
        int start = 0;
        int end = height.size() - 1;
        int result = INT_MIN;
        while (start < end) {
            int area = min(height[end], height[start]) * (end - start);
            result = max(result, area);
            if (height[start] <= height[end]) {
                start++;
            } else {
                end--;
            }
        }
        return result;
    }
};
```

相關題目

- Trapping Rain Water, 見 §2.1.16
- Largest Rectangle in Histogram, 見 §4.1.3

第 13 章
動態規劃

13.1 Triangle

描述

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

Note: Bonus point if you are able to do this using only $O(n)$ extra space, where n is the total number of rows in the triangle.

分析

設狀態為 $f(i,j)$，表示從從位置 (i,j) 出發，路徑的最小和，則狀態轉移方程為

$$f(i,j) = \min\{f(i+1,j), f(i+1,j+1)\} + (i,j)$$

代碼

```
// LeetCode, Triangle
// 時間複雜度 O(n^2), 空間複雜度 O(1)
class Solution {
public:
    int minimumTotal (vector<vector<int>>& triangle) {
        for (int i = triangle.size() - 2; i >= 0; --i)
            for (int j = 0; j < i + 1; ++j)
                triangle[i][j] += min(triangle[i + 1][j],
                        triangle[i + 1][j + 1]);
```

```
        return triangle [0][0];
    }
};
```

相關題目

- 無

13.2 Maximum Subarray

描述

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array

$$-2, 1, -3, 4, -1, 2, 1, -5, 4$$

, the contiguous subarray

$$4, -1, 2, 1$$

has the largest `sum = 6`.

分析

最大連續子序列和，非常經典的題。

當我們從頭到尾遍歷這個數組的時候，對於數組裏的一個整數，它有幾種選擇呢？它只有兩種選擇：1、加入之前的 SubArray；2. 自己另起一個 SubArray。那什麼時候會出現這兩種情況呢？

如果之前 SubArray 的總體和大於 0 的話，我們認為其對後續結果是有貢獻的。這種情況下我們選擇加入之前的 SubArray

如果之前 SubArray 的總體和為 0 或者小於 0 的話，我們認為其對後續結果是沒有貢獻，甚至是有害的（小於 0 時）。這種情況下我們選擇以這個數字開始，另起一個 SubArray。

設狀態為 `f[j]`，表示以 `S[j]` 結尾的最大連續子序列和，則狀態轉移方程如下：

$$\begin{aligned} f[j] &= \max\{f[j-1] + S[j], S[j]\}, \text{其中 } 1 \le j \le n \\ target &= \max\{f[j]\}, \text{其中 } 1 \le j \le n \end{aligned}$$

解釋如下：

- 情況一，S[j] 不獨立，與前面的某些數組成一個連續子序列，則最大連續子序列和為 $f[j-1]+S[j]$。
- 情況二，S[j] 獨立劃分成為一段，即連續子序列僅包含一個數 S[j]，則最大連續子序列和為 $S[j]$。

其他思路：

- 思路 2： 直接在 i 到 j 之間暴力枚舉，複雜度是 $O(n^3)$
- 思路 3： 處理後枚舉，連續子序列的和等於兩個前綴和之差，複雜度 $O(n^2)$。

- 思路 4： 分治法，把序列分為兩段，分別求最大連續子序列和，然後歸併，複雜度 $O(n \log n)$
- 思路 5： 把思路 2$O(n^2)$ 的代碼稍作處理，得到 $O(n)$ 的算法
- 思路 6： 當成 M=1 的最大 M 子段和

動規

```
// LeetCode, Maximum Subarray
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int result = INT_MIN, f = 0;
        for (int i = 0; i < nums.size(); ++i) {
            f = max(f + nums[i], nums[i]);
            result = max(result, f);
        }
        return result;
    }
};
```

思路 5

```
// LeetCode, Maximum Subarray
// 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    int maxSubArray(vector<int>& A) {
        return mcss(A.begin(), A.end());
    }
private:
    // 思路 5, 求最大連續子序列和
    template <typename Iter>
    static int mcss(Iter begin, Iter end) {
        int result, cur_min;
        const int n = distance(begin, end);
        int *sum = new int[n + 1];  // 前 n 項和

        sum[0] = 0;
        result = INT_MIN;
        cur_min = sum[0];
        for (int i = 1; i <= n; i++) {
            sum[i] = sum[i - 1] + *(begin  + i - 1);
        }
        for (int i = 1; i <= n; i++) {
            result = max(result, sum[i] - cur_min);
            cur_min = min(cur_min, sum[i]);
        }
        delete[] sum;
        return result;
    }
};
```

相關題目

- Binary Tree Maximum Path Sum，見 §5.4.5

13.3 Palindrome Partitioning II

描述

Given a string s, partition s such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of s.

For example, given `s = "aab"`,

Return 1 since the palindrome partitioning

$$"aa","b"$$

could be produced using 1 cut.

分析

定義狀態 `f(i,j)` 表示區間 `[i,j]` 之間最小的 cut 數，則狀態轉移方程為

$$f(i,j) = \min\left\{f(i,k) + f(k+1,j)\right\}, i \le k \le j, 0 \le i \le j < n$$

這是一個二維函數，實際寫代碼比較麻煩。

所以要轉換成一維 DP。如果每次，從 i 往右掃描，每找到一個迴文就算一次 DP 的話，就可以轉換為 `f(i)`=區間 `[i, n-1]` 之間最小的 `cut` 數，n 為字符串長度，則狀態轉移方程為

$$f(i) = \min\left\{f(j+1) + 1\right\}, i \le j < n$$

一個問題出現了，就是如何判斷 `[i,j]` 是否是迴文？每次都從 i 到 j 比較一遍？太浪費了，這裏也是一個 DP 問題。

定義狀態 `P[i][j] = true if [i,j]` 為迴文，那麼

```
P[i][j] = str[i] == str[j] && P[i+1][j-1]
```

代碼

```
// LeetCode, Palindrome Partitioning II
// 時間複雜度 O(n^2), 空間複雜度 O(n^2)
class Solution {
public:
    int minCut(const string& s) {
        const int n = s.size();
        int f[n+1];
        bool p[n][n];
        fill_n(&p[0][0], n * n, false);
        //the worst case is cutting by each char
```

```
        for (int i = 0; i <= n; i++)
            f[i] = n - 1 - i; // 最後一個 f[n]=-1
        for (int i = n - 1; i >= 0; i--) {
            for (int j = i; j < n; j++) {
                if (s[i] == s[j] && (j - i < 2 || p[i + 1][j - 1])) {
                    p[i][j] = true;
                    f[i] = min(f[i], f[j + 1] + 1);
                }
            }
        }
        return f[0];
    }
};
```

相關題目

- Palindrome Partitioning，見 §10.1

13.4 Maximal Rectangle

描述

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

分析

無

代碼

```
// LeetCode, Maximal Rectangle
// 時間複雜度 O(n^2), 空間複雜度 O(n)
class Solution {
public:
    int maximalRectangle(vector<vector<char> > &matrix) {
        if (matrix.empty())  return 0;

        const int m = matrix.size();
        const int n = matrix[0].size();
        vector<int> H(n, 0);
        vector<int> L(n, 0);
        vector<int> R(n, n);

        int ret = 0;
        for (int i = 0; i < m; ++i) {
            int left = 0, right = n;
            // calculate L(i, j) from left to right
            for (int j = 0; j < n; ++j) {
                if (matrix[i][j] == '1') {
```

```
                    ++H[j];
                    L[j] = max(L[j], left);
                } else {
                    left = j+1;
                    H[j] = 0; L[j] = 0; R[j] = n;
                }
            }
            // calculate R(i, j) from right to left
            for (int j = n-1; j >= 0; --j) {
                if (matrix[i][j] == '1') {
                    R[j] = min(R[j], right);
                    ret = max(ret, H[j]*(R[j]-L[j]));
                } else {
                    right = j;
                }
            }
        }
        return ret;
    }
};
```

相關題目

- 無

13.5 Best Time to Buy and Sell Stock III

描述

Say you have an array for which the i-th element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

分析

設狀態 $f(i)$，表示區間 $[0,i](0 \le i \le n-1)$ 的最大利潤，狀態 $g(i)$，表示區間 $[i,n-1](0 \le i \le n-1)$ 的最大利潤，則最終答案為 $\max\{f(i)+g(i)\}, 0 \le i \le n-1$。

允許在一天內買進又賣出，相當於不交易，因為題目的規定是最多兩次，而不是一定要兩次。

將原數組變成差分數組，本題也可以看做是最大 m 子段和，$m = 2$，參考代碼：https://gist.github.com/soulmachine/5906637

代碼

```
// LeetCode, Best Time to Buy and Sell Stock III
// 時間複雜度 O(n)，空間複雜度 O(n)
class Solution {
```

```
public:
    int maxProfit(vector<int>& prices) {
        if (prices.size() < 2) return 0;

        const int n = prices.size();
        vector<int> f(n, 0);
        vector<int> g(n, 0);

        for (int i = 1, valley = prices[0]; i < n; ++i) {
            valley = min(valley, prices[i]);
            f[i] = max(f[i - 1], prices[i] - valley);
        }

        for (int i = n - 2, peak = prices[n - 1]; i >= 0; --i) {
            peak = max(peak, prices[i]);
            g[i] = max(g[i], peak - prices[i]);
        }

        int max_profit = 0;
        for (int i = 0; i < n; ++i)
            max_profit = max(max_profit, f[i] + g[i]);

        return max_profit;
    }
};
```

相關題目

- Best Time to Buy and Sell Stock，見 §12.3
- Best Time to Buy and Sell Stock II，見 §12.4

13.6 Interleaving String

描述

Given $s1$, $s2$, $s3$, find whether $s3$ is formed by the interleaving of $s1$ and $s2$.

For example, Given: `s1 = "aabcc", s2 = "dbbca"`,

When `s3 = "aadbbcbcac"`, return true.

When `s3 = "aadbbbaccc"`, return false.

分析

設狀態 `f[i][j]`，表示 `s1[0,i]` 和 `s2[0,j]`，匹配 `s3[0, i+j]`。如果 s1 的最後一個字符等於 s3 的最後一個字符，則 `f[i][j]=f[i-1][j]`；如果 s2 的最後一個字符等於 s3 的最後一個字符，則 `f[i][j]=f[i][j-1]`。因此狀態轉移方程如下：

```
f[i][j] = (s1[i - 1] == s3 [i + j - 1] && f[i - 1][j])
       || (s2[j - 1] == s3 [i + j - 1] && f[i][j - 1]);
```

Reference https://www.youtube.com/watch?v=ih2OZ9-M3OM

遞歸

```
// LeetCode, Interleaving String
// 遞歸, 會超時, 僅用來幫助理解
class Solution {
public:
    bool isInterleave(const string& s1, const string& s2, const string& s3) {
        if (s3.length() != s1.length() + s2.length())
            return false;

        return isInterleave(begin(s1), end(s1), begin(s2), end(s2),
                begin(s3), end(s3));
    }

    template<typename InIt>
    bool isInterleave(InIt first1, InIt last1, InIt first2, InIt last2,
            InIt first3, InIt last3) {
        if (first3 == last3)
            return first1 == last1 && first2 == last2;

        return (*first1 == *first3
                && isInterleave(next(first1), last1, first2, last2,
                        next(first3), last3))
                || (*first2 == *first3
                        && isInterleave(first1, last1, next(first2), last2,
                                next(first3), last3));
    }
};
```

動規

```
// LeetCode, Interleaving String
// 二維動規, 時間複雜度 O(n^2), 空間複雜度 O(n^2)
class Solution {
public:
    bool isInterleave(const string& s1, const string& s2, const string& s3) {
        if (s3.length() != s1.length() + s2.length())
            return false;

        vector<vector<bool>> f(s1.length() + 1,
                vector<bool>(s2.length() + 1, true));

        for (size_t i = 1; i <= s1.length(); ++i)
            f[i][0] = f[i - 1][0] && s1[i - 1] == s3[i - 1];

        for (size_t i = 1; i <= s2.length(); ++i)
            f[0][i] = f[0][i - 1] && s2[i - 1] == s3[i - 1];

        for (size_t i = 1; i <= s1.length(); ++i)
            for (size_t j = 1; j <= s2.length(); ++j)
                f[i][j] = (s1[i - 1] == s3[i + j - 1] && f[i - 1][j])
                        || (s2[j - 1] == s3[i + j - 1] && f[i][j - 1]);
```

```
        return f[s1.length()][s2.length()];
    }
};
```

動規+滾動數組

```
// LeetCode, Interleaving String
// 二維動規+滾動數組, 時間複雜度 O(n^2), 空間複雜度 O(n)
class Solution {
public:
    bool isInterleave(const string& s1, const string& s2, const string& s3) {
        if (s1.length() + s2.length() != s3.length())
            return false;

        if (s1.length() < s2.length())
            return isInterleave(s2, s1, s3);

        vector<bool> f(s2.length() + 1, true);

        for (size_t i = 1; i <= s2.length(); ++i)
            f[i] = s2[i - 1] == s3[i - 1] && f[i - 1];

        for (size_t i = 1; i <= s1.length(); ++i) {
            f[0] = s1[i - 1] == s3[i - 1] && f[0];

            for (size_t j = 1; j <= s2.length(); ++j)
                f[j] = (s1[i - 1] == s3[i + j - 1] && f[j])
                        || (s2[j - 1] == s3[i + j - 1] && f[j - 1]);
        }

        return f[s2.length()];
    }
};
```

相關題目

- 無

13.7 Scramble String

描述

Given a string $s1$, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of `s1 = "great"`:

```
    great
   /    \
  gr    eat
 / \    /  \
g   r  e   at
```

```
      / \
     a   t
```

To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".

```
    rgeat
   /    \
  rg    eat
 / \    /  \
r   g  e   at
          / \
         a   t
```

We say that "rgeat" is a scrambled string of "great".

Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".

```
    rgtae
   /    \
  rg    tae
 / \    /  \
r   g  ta  e
      / \
     t   a
```

We say that "rgtae" is a scrambled string of "great".

Given two strings $s1$ and $s2$ of the same length, determine if $s2$ is a scrambled string of $s1$.

分析

首先想到的是遞歸（即深搜），對兩個 string 進行分割，然後比較四對字符串。代碼雖然簡單，但是複雜度比較高。有兩種加速策略，一種是剪枝，提前返回；一種是加緩存，緩存中間結果，即 memoization（翻譯為記憶化搜索）。

剪枝可以五花八門，要充分觀察，充分利用信息，找到能讓節點提前返回的條件。例如，判斷兩個字符串是否互為 scamble，至少要求每個字符在兩個字符串中出現的次數要相等，如果不相等則返回 false。

加緩存，可以用數組或 HashMap。本題維數較高，用 HashMap，`map` 和 `unordered_map` 均可。

既然可以用記憶化搜索，這題也一定可以用動規。設狀態為 `f[n][i][j]`，表示長度為 n，起點為 `s1[i]` 和起點為 `s2[j]` 兩個字符串是否互為 scramble，則狀態轉移方程為

```
f[n][i][j]} =  (f[k][i][j] && f[n-k][i+k][j+k])
            || (f[k][i][j+n-k] && f[n-k][i+k][j])
```

遞歸

```
// LeetCode, Scramble String
// 遞歸，會超時，僅用來幫助理解
```

```
// 時間複雜度 O(n^6), 空間複雜度 O(1)
class Solution {
public:
    bool isScramble(const string& s1, const string& s2) {
        return isScramble(s1.begin(), s1.end(), s2.begin());
    }
private:
    typedef string::iterator Iterator;
    bool isScramble(Iterator first1, Iterator last1, Iterator first2) {
        auto length = distance(first1, last1);
        auto last2 = next(first2, length);

        if (length == 1) return *first1 == *first2;

        for (int i = 1; i < length; ++i)
            if ((isScramble(first1, first1 + i, first2)
                 && isScramble(first1 + i, last1, first2 + i))
                    || (isScramble(first1, first1 + i, last2 - i)
                            && isScramble(first1 + i, last1, first2)))
                return true;

        return false;
    }
};
```

動規

```
// LeetCode, Scramble String
// 動規, 時間複雜度 O(n^3), 空間複雜度 O(n^3)
class Solution {
public:
    bool isScramble(const string& s1, const string& s2) {
        const int N = s1.size();
        if (N != s2.size()) return false;

        // f[n][i][j], 表示長度為 n, 起點為 s1[i] 和
        // 起點為 s2[j] 兩個字符串是否互為 scramble
        bool f[N + 1][N][N];
        fill_n(&f[0][0][0], (N + 1) * N * N, false);

        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                f[1][i][j] = s1[i] == s2[j];

        for (int n = 1; n <= N; ++n) {
            for (int i = 0; i + n <= N; ++i) {
                for (int j = 0; j + n <= N; ++j) {
                    for (int k = 1; k < n; ++k) {
                        if ((f[k][i][j] && f[n - k][i + k][j + k]) ||
                                (f[k][i][j + n - k] && f[n - k][i + k][j])) {
                            f[n][i][j] = true;
                            break;
```

```
                    }
                }
            }
        }
        return f[N][0][0];
    }
};
```

遞歸+剪枝

```
// LeetCode, Scramble String
// 遞歸+剪枝
// 時間複雜度 O(n^6), 空間複雜度 O(1)
class Solution {
public:
    bool isScramble(const string& s1, const string& s2) {
        return isScramble(s1.begin(), s1.end(), s2.begin());
    }
private:
    typedef string::const_iterator Iterator;
    bool isScramble(Iterator first1, Iterator last1, Iterator first2) {
        auto length = distance(first1, last1);
        auto last2 = next(first2, length);
        if (length == 1) return *first1 == *first2;

        // 剪枝, 提前返回
        int A[26]; // 每個字符的計數器
        fill(A, A + 26, 0);
        for(int i = 0; i < length; i++) A[*(first1+i)-'a']++;
        for(int i = 0; i < length; i++) A[*(first2+i)-'a']--;
        for(int i = 0; i < 26; i++) if (A[i] != 0) return false;

        for (int i = 1; i < length; ++i)
            if ((isScramble(first1, first1 + i, first2)
                 && isScramble(first1 + i, last1, first2 + i))
                    || (isScramble(first1, first1 + i, last2 - i)
                            && isScramble(first1 + i, last1, first2)))
                return true;

        return false;
    }
};
```

備忘錄法

```
// LeetCode, Scramble String
// 遞歸+map 做 cache
// 時間複雜度 O(n^3), 空間複雜度 O(n^3), TLE
class Solution {
public:
    bool isScramble(const string& s1, const string& s2) {
```

```
        cache.clear();
        return isScramble(s1.begin(), s1.end(), s2.begin());
    }
private:
    typedef string::const_iterator Iterator;
    map<tuple<Iterator, Iterator, Iterator>, bool> cache;

    bool isScramble(Iterator first1, Iterator last1, Iterator first2) {
        auto length = distance(first1, last1);
        auto last2 = next(first2, length);

        if (length == 1) return *first1 == *first2;

        for (int i = 1; i < length; ++i)
            if ((getOrUpdate(first1, first1 + i, first2)
                    && getOrUpdate(first1 + i, last1, first2 + i))
                    || (getOrUpdate(first1, first1 + i, last2 - i)
                            && getOrUpdate(first1 + i, last1, first2)))
                return true;

        return false;
    }

    bool getOrUpdate(Iterator first1, Iterator last1, Iterator first2) {
        auto key = make_tuple(first1, last1, first2);
        auto pos = cache.find(key);

        return (pos != cache.end()) ?
                pos->second : (cache[key] = isScramble(first1, last1, first2));
    }
};
```

備忘錄法

```
typedef string::const_iterator Iterator;
typedef tuple<Iterator, Iterator, Iterator> Key;
// 定製一個哈希函數
namespace std {
template<> struct hash<Key> {
    size_t operator()(const Key & x) const {
        Iterator first1, last1, first2;
        tie(first1, last1, first2) = x;

        int result = *first1;
        result = result * 31 + *last1;
        result = result * 31 + *first2;
        result = result * 31 + *(next(first2, distance(first1, last1)-1));
        return result;
    }
};
}
```

```
// LeetCode, Scramble String
// 遞歸+unordered_map 做 cache, 比 map 快
// 時間複雜度 O(n^3), 空間複雜度 O(n^3)
class Solution {
public:
    unordered_map<Key, bool> cache;

    bool isScramble(const string& s1, const string& s2) {
        cache.clear();
        return isScramble(s1.begin(), s1.end(), s2.begin());
    }

    bool isScramble(Iterator first1, Iterator last1, Iterator first2) {
        auto length = distance(first1, last1);
        auto last2 = next(first2, length);

        if (length == 1)
            return *first1 == *first2;

        for (int i = 1; i < length; ++i)
            if ((getOrUpdate(first1, first1 + i, first2)
                    && getOrUpdate(first1 + i, last1, first2 + i))
                    || (getOrUpdate(first1, first1 + i, last2 - i)
                            && getOrUpdate(first1 + i, last1, first2)))
                return true;

        return false;
    }

    bool getOrUpdate(Iterator first1, Iterator last1, Iterator first2) {
        auto key = make_tuple(first1, last1, first2);
        auto pos = cache.find(key);

        return (pos != cache.end()) ?
                pos->second : (cache[key] = isScramble(first1, last1, first2));
    }
};
```

相關題目

- 無

13.8 Minimum Path Sum

描述

Given a $m \times n$ grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time

分析

跟 Unique Paths (見 §10.2) 很類似。

設狀態為 `f[i][j]`，表示從起點 $(0,0)$ 到達 (i,j) 的最小路徑和，則狀態轉移方程為：

```
f[i][j]=min(f[i-1][j], f[i][j-1])+grid[i][j]
```

Grid	0	1	2
0	1	4	5
1	3	5	7
2	8	2	1
f	0	1	2
0	1	5	9
1	4	6	12
2	12	6	7

图 13-1 Minimum Path Sum

備忘錄法

```
// LeetCode, Minimum Path Sum
// 備忘錄法
class Solution {
public:
    int minPathSum(vector<vector<int> > &grid) {
        const int m = grid.size();
        const int n = grid[0].size();
        this->f = vector<vector<int> >(m, vector<int>(n, -1));
        return dfs(grid, m-1, n-1);
    }
private:
    vector<vector<int> > f;  // 緩存

    int dfs(const vector<vector<int> > &grid, int x, int y) {
        if (x < 0 || y < 0) return INT_MAX; // 越界, 終止條件, 注意, 不是 0

        if (x == 0 && y == 0) return grid[0][0]; // 回到起點, 收斂條件

        return min(getOrUpdate(grid, x - 1, y),
                   getOrUpdate(grid, x, y - 1)) + grid[x][y];
    }

    int getOrUpdate(const vector<vector<int> > &grid, int x, int y) {
        if (x < 0 || y < 0) return INT_MAX; // 越界, 注意, 不是 0
        if (f[x][y] >= 0) return f[x][y];
        else return f[x][y] = dfs(grid, x, y);
```

```
    }
};
```

動規

```
// LeetCode, Minimum Path Sum
// 二維動規
class Solution {
public:
    int minPathSum(vector<vector<int> > &grid) {
        if (grid.size() == 0) return 0;
        const int m = grid.size();
        const int n = grid[0].size();

        int f[m][n];
        f[0][0] = grid[0][0];
        for (int i = 1; i < m; i++) {
            f[i][0] = f[i - 1][0] + grid[i][0];
        }
        for (int i = 1; i < n; i++) {
            f[0][i] = f[0][i - 1] + grid[0][i];
        }

        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                f[i][j] = min(f[i - 1][j], f[i][j - 1]) + grid[i][j];
            }
        }
        return f[m - 1][n - 1];
    }
};
```

動規+滾動數組

```
// LeetCode, Minimum Path Sum
// 二維動規+滾動數組
class Solution {
public:
    int minPathSum(vector<vector<int> > &grid) {
        const int m = grid.size();
        const int n = grid[0].size();

        int f[n];
        fill(f, f+n, INT_MAX); // 初始值是 INT_MAX, 因為後面用了 min 函數。
        f[0] = 0;

        for (int i = 0; i < m; i++) {
            f[0] += grid[i][0];
            for (int j = 1; j < n; j++) {
                // 左邊的 f[j], 表示更新後的 f[j], 與公式中的 f[i[[j] 對應
                // 右邊的 f[j], 表示老的 f[j], 與公式中的 f[i-1][j] 對應
                f[j] = min(f[j - 1], f[j]) + grid[i][j];
```

```
                }
            }
            return f[n - 1];
        }
    };
```

相關題目

- Unique Paths, 見 §10.2
- Unique Paths II, 見 §10.3

13.9 Edit Distance

描述

Given two words `word1` and `word2`, find the minimum number of steps required to convert `word1` to `word2`. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

分析

設狀態為 `f[i][j]`，表示 `A[0,i]` 和 `B[0,j]` 之間的最小編輯距離。設 `A[0,i]` 的形式是 `str1c`，`B[0,j]` 的形式是 `str2d`，

1. 如果 `c==d`，則 `f[i][j]=f[i-1][j-1]`；
2. 如果 `c!=d`，
   (a) 如果將 c 替換成 d，則 `f[i][j]=f[i-1][j-1]+1`；
   (b) 如果在 c 後面添加一個 d，則 `f[i][j]=f[i][j-1]+1`；
   (c) 如果將 c 刪除，則 `f[i][j]=f[i-1][j]+1`；

		a	p	p
	0	1	2	3
a	1	0	1	2
a	2	1	1	2
p	3	2	1	1
p	4	3	2	1

图 13-2 Edit Distance

動規

```
// LeetCode, Edit Distance
// 二維動規, 時間複雜度 O(n*m), 空間複雜度 O(n*m)
class Solution {
public:
    int minDistance(const string &word1, const string &word2) {
        const size_t n = word1.size();
        const size_t m = word2.size();
        // 長度為 n 的字符串, 有 n+1 個隔板
        int f[n + 1][m + 1];
        for (size_t i = 0; i <= n; i++)
            f[i][0] = i;
        for (size_t j = 0; j <= m; j++)
            f[0][j] = j;

        for (size_t i = 1; i <= n; i++) {
            for (size_t j = 1; j <= m; j++) {
                if (word1[i - 1] == word2[j - 1])
                    f[i][j] = f[i - 1][j - 1];
                else {
                    int mn = min(f[i - 1][j], f[i][j - 1]);
                    f[i][j] = 1 + min(f[i - 1][j - 1], mn);
                }
            }
        }
        return f[n][m];
    }
};
```

動規+滾動數組

```
// LeetCode, Edit Distance
// 二維動規+滾動數組
// 時間複雜度 O(n*m), 空間複雜度 O(n)
class Solution {
public:
    int minDistance(const string &word1, const string &word2) {
        if (word1.length() < word2.length())
            return minDistance(word2, word1);

        int f[word2.length() + 1];
        int upper_left = 0; // 額外用一個變量記錄 f[i-1][j-1]

        for (size_t i = 0; i <= word2.size(); ++i)
            f[i] = i;

        for (size_t i = 1; i <= word1.size(); ++i) {
            upper_left = f[0];
            f[0] = i;

            for (size_t j = 1; j <= word2.size(); ++j) {
                int upper = f[j];
```

```
                if (word1[i - 1] == word2[j - 1])
                    f[j] = upper_left;
                else
                    f[j] = 1 + min(upper_left, min(f[j], f[j - 1]));

                upper_left = upper;
            }
        }

        return f[word2.length()];
    }
};
```

相關題目

- 無

13.10 Decode Ways

描述

A message containing letters from `A-Z` is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Given an encoded message containing digits, determine the total number of ways to decode it.

For example, Given encoded message `"12"`, it could be decoded as `"AB"` (1 2) or `"L"` (12).

The number of ways decoding `"12"` is 2.

分析

跟 Climbing Stairs (見 §2.1.19) 很類似，不過多加一些判斷邏輯。

代碼

```
// LeetCode, Decode Ways
// 動規, 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    int numDecodings(const string &s) {
        if (s.empty() || s[0] == '0') return 0;

        int prev = 0;
        int cur = 1;
        // 長度為 n 的字符串, 有 n+1 個階梯
        for (size_t i = 1; i <= s.size(); ++i) {
```

```
            if (s[i-1] == '0') cur = 0;

            if (i < 2 || !(s[i - 2] == '1' ||
                     (s[i - 2] == '2' && s[i - 1] <= '6')))
                prev = 0;

            int tmp = cur;
            cur = prev + cur;
            prev = tmp;
        }
        return cur;
    }
};
```

相關題目

- Climbing Stairs, 見 §2.1.19

13.11 Distinct Subsequences

描述

Given a string S and a string T, count the number of distinct subsequences of T in S.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, `"ACE"` is a subsequence of `"ABCDE"` while `"AEC"` is not).

Here is an example: S = `"rabbbit"`, T = `"rabbit"`

Return 3.

分析

設狀態為 $f(i, j)$，表示 `T[0,j]` 在 `S[0,i]` 裏出現的次數。首先，無論 `S[i]` 和 `T[j]` 是否相等，若不使用 `S[i]`，則 $f(i, j) = f(i - 1, j)$；若 `S[i]==T[j]`，則可以使用 `S[i]`，此時 $f(i, j) = f(i - 1, j) + f(i - 1, j - 1)$。

		r	a	b	b	i	t	
	1	0	0	0	0	0	0	
r	1	1	0	0	0	0	0	
a	1	1	1	0	0	0	0	
b	1	1	1	1	0	0	0	
b	1	1	1	2	1	0	0	
b	1	1	1	3	3	0	0	
i	1	1	1	3	3	3	0	
t	1	1	1	3	3	3	3	
		r	a	b	b	i	t	
f								
	1	1	0	0	0	0	0	, i = 0
	1	1	1	0	0	0	0	, i = 1
	1	1	1	1	0	0	0	, i = 2

图 13-3 Distinct Subsequences

代碼

```
// LeetCode, Distinct Subsequences
// 二維動規+滾動數組
// 時間複雜度 O(m*n), 空間複雜度 O(n)
class Solution {
public:
    int numDistinct(const string &S, const string &T) {
        vector<int> f(T.size() + 1);
        f[0] = 1;
        for (int i = 0; i < S.size(); ++i) {
            for (int j = T.size() - 1; j >= 0; --j) {
                f[j + 1] += S[i] == T[j] ? f[j] : 0;
            }
        }

        return f[T.size()];
    }
};
```

相關題目

- 無

13.12 Word Break

描述

Given a string s and a dictionary of words dict, determine if s can be segmented into a space-separated sequence of one or more dictionary words.

For example, given

s = "leetcode",

dict = ["leet", "code"].

Return true because "leetcode" can be segmented as "leet code".

分析

設狀態為 $f(i)$，表示 s[0,i] 是否可以分詞，則狀態轉移方程為

$$f(i) = any\_of(f(j)\&\&s[j+1,i] \in dict), 0 \leq j < i$$

深搜

```
// LeetCode, Word Break
// 深搜, 超時
// 時間複雜度 O(2^n), 空間複雜度 O(n)
class Solution {
public:
    bool wordBreak(string s, unordered_set<string> &dict) {
        return dfs(s, dict, 0, 0);
    }
private:
    static bool dfs(const string &s, unordered_set<string> &dict,
            size_t start, size_t cur) {
        if (cur == s.size()) {
            return dict.find(s.substr(start, cur-start+1)) != dict.end();
        }
        if (dfs(s, dict, start, cur+1)) return true;
        if (dict.find(s.substr(start, cur-start+1)) != dict.end())
            if (dfs(s, dict, cur+1, cur+1)) return true;
        return false;
    }
};
```

動規

```
// LeetCode, Word Break
// 動規, 時間複雜度 O(n^2), 空間複雜度 O(n)
class Solution {
public:
    bool wordBreak(string s, unordered_set<string> &dict) {
        // 長度為 n 的字符串有 n+1 個隔板
```

```
        vector<bool> f(s.size() + 1, false);
        f[0] = true; // 空字符串
        for (int i = 1; i <= s.size(); ++i) {
            for (int j = i - 1; j >= 0; --j) {
                if (f[j] && dict.find(s.substr(j, i - j)) != dict.end()) {
                    f[i] = true;
                    break;
                }
            }
        }
        return f[s.size()];
    }
};
```

相關題目

- Word Break II, 見 §13.13

13.13 Word Break II

描述

Given a string s and a dictionary of words dict, add spaces in s to construct a sentence where each word is a valid dictionary word.

Return all such possible sentences.

For example, given

s = `"catsanddog"`,

dict = `["cat", "cats", "and", "sand", "dog"]`.

A solution is `["cats and dog", "cat sand dog"]`.

分析

在上一題的基礎上，要返回解本身。

代碼

```
// LeetCode, Word Break II
// 動規，時間複雜度 O(n^2)，空間複雜度 O(n^2)
class Solution {
public:
    vector<string> wordBreak(string s, unordered_set<string> &dict) {
        // 長度為 n 的字符串有 n+1 個隔板
        vector<bool> f(s.length() + 1, false);
        // prev[i][j] 為 true，表示 s[j, i) 是一個合法單詞，可以從 j 處切開
        // 第一行未用
        vector<vector<bool> > prev(s.length() + 1, vector<bool>(s.length()));
        f[0] = true;
        for (size_t i = 1; i <= s.length(); ++i) {
```

```
            for (int j = i - 1; j >= 0; --j) {
                if (f[j] && dict.find(s.substr(j, i - j)) != dict.end()) {
                    f[i] = true;
                    prev[i][j] = true;
                }
            }
        }
        vector<string> result;
        vector<string> path;
        gen_path(s, prev, s.length(), path, result);
        return result;

    }
private:
    // DFS 遍歷樹, 生成路徑
    void gen_path(const string &s, const vector<vector<bool> > &prev,
            int cur, vector<string> &path, vector<string> &result) {
        if (cur == 0) {
            string tmp;
            for (auto iter = path.crbegin(); iter != path.crend(); ++iter)
                tmp += *iter + " ";
            tmp.erase(tmp.end() - 1);
            result.push_back(tmp);
        }
        for (size_t i = 0; i < s.size(); ++i) {
            if (prev[cur][i]) {
                path.push_back(s.substr(i, cur - i));
                gen_path(s, prev, i, path, result);
                path.pop_back();
            }
        }
    }
};
```

相關題目

- Word Break, 見 §13.12

第 14 章
圖

無向圖的節點定義如下:

```
// 無向圖的節點
struct UndirectedGraphNode {
    int label;
    vector<UndirectedGraphNode *> neighbors;
    UndirectedGraphNode(int x) : label(x) {};
};
```

14.1 Clone Graph

描述

Clone an undirected graph. Each node in the graph contains a `label` and a list of its `neighbours`.

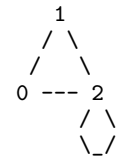
OJ's undirected graph serialization: Nodes are labeled uniquely.

We use `#` as a separator for each node, and `,` as a separator for node label and each neighbour of the node. As an example, consider the serialized graph `{0,1,2#1,2#2,2}`.

The graph has a total of three nodes, and therefore contains three parts as separated by `#`.

1. First node is labeled as 0. Connect node 0 to both nodes 1 and 2.
2. Second node is labeled as 1. Connect node 1 to node 2.
3. Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.

Visually, the graph looks like the following:



分析

廣度優先遍歷或深度優先遍歷都可以。

DFS

```
// LeetCode, Clone Graph
// DFS, 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    UndirectedGraphNode *cloneGraph(const UndirectedGraphNode *node) {
        if(node == nullptr) return nullptr;
        // key is original node, value is copied node
        unordered_map<const UndirectedGraphNode *,
                      UndirectedGraphNode *> copied;
        clone(node, copied);
        return copied[node];
    }
private:
    // DFS
    static UndirectedGraphNode* clone(const UndirectedGraphNode *node,
            unordered_map<const UndirectedGraphNode *,
            UndirectedGraphNode *> &copied) {
        // a copy already exists
        if (copied.find(node) != copied.end()) return copied[node];

        UndirectedGraphNode *new_node = new UndirectedGraphNode(node->label);
        copied[node] = new_node;
        for (auto nbr : node->neighbors)
            new_node->neighbors.push_back(clone(nbr, copied));
        return new_node;
    }
};
```

BFS

```
// LeetCode, Clone Graph
// BFS, 時間複雜度 O(n), 空間複雜度 O(n)
class Solution {
public:
    UndirectedGraphNode *cloneGraph(const UndirectedGraphNode *node) {
        if (node == nullptr) return nullptr;
        // key is original node, value is copied node
        unordered_map<const UndirectedGraphNode *,
                      UndirectedGraphNode *> copied;
        // each node in queue is already copied itself
        // but neighbors are not copied yet
        queue<const UndirectedGraphNode *> q;
        q.push(node);
        copied[node] = new UndirectedGraphNode(node->label);
        while (!q.empty()) {
            const UndirectedGraphNode *cur = q.front();
            q.pop();
            for (auto nbr : cur->neighbors) {
                // a copy already exists
                if (copied.find(nbr) != copied.end()) {
                    copied[cur]->neighbors.push_back(copied[nbr]);
```

```
                } else {
                    UndirectedGraphNode *new_node =
                            new UndirectedGraphNode(nbr->label);
                    copied[nbr] = new_node;
                    copied[cur]->neighbors.push_back(new_node);
                    q.push(nbr);
                }
            }
        }
        return copied[node];
    }
};
```

相關題目

- 無

第 15 章
細節實現題

這類題目不考特定的算法，純粹考察寫代碼的熟練度。

15.1 Reverse Integer

描述

Reverse digits of an integer.

Example1: x = 123, return 321

Example2: x = -123, return -321

Have you thought about this?

Here are some good questions to ask before coding. Bonus points for you if you have already thought through this!

If the integer's last digit is 0, what should the output be? ie, cases such as 10, 100.

Did you notice that the reversed integer might overflow? Assume the input is a 32-bit integer, then the reverse of 1000000003 overflows. How should you handle such cases?

Throw an exception? Good, but what if throwing an exception is not an option? You would then have to re-design the function (ie, add an extra parameter).

分析

短小精悍的題，代碼也可以寫的很短小。

代碼

```
//LeetCode, Reverse Integer
// 時間複雜度 O(logn), 空間複雜度 O(1)
// 考慮 1. 負數的情況 2. 溢出的情況 (正溢出&&負溢出, 比如 x = -2147483648(即-2^31) )
class Solution {
public:
    int reverse (int x) {
        long long r = 0;
        long long t = x;
        t = t > 0 ? t : -t;
        for (; t; t /= 10)
```

```
            r = r * 10 + t % 10;

        bool sign = x > 0 ? false: true;
        if (r > 2147483647 || (sign && r > 2147483648)) {
            return 0;
        } else {
            if (sign) {
                return -r;
            } else {
                return r;
            }
        }
    }
};
```

相關題目

- Palindrome Number, 見 §15.2

15.2 Palindrome Number

描述

Determine whether an integer is a palindrome. Do this without extra space.

Some hints:

Could negative integers be palindromes? (ie, -1)

If you are thinking of converting the integer to string, note the restriction of using extra space.

You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case?

There is a more generic way of solving this problem.

分析

首先想到,可以利用上一題,將整數反轉,然後與原來的整數比較,是否相等,相等則為 Palindrome 的。可是 reverse() 會溢出。

正確的解法是，不斷地取第一位和最後一位（10 進制下）進行比較，相等則取第二位和倒數第二位，直到完成比較或者中途找到了不一致的位。

代碼

```
//LeetCode, Palindrome Number
// 時間複雜度 O(1), 空間複雜度 O(1)
class Solution {
public:
    bool isPalindrome(int x) {
        if (x < 0) return false;
        int d = 1; // divisor
```

```
        while (x / d >= 10) d *= 10;

        while (x > 0) {
            int q = x / d;  // quotient
            int r = x % 10;   // remainder
            if (q != r) return false;
            x = x % d / 10;
            d /= 100;
        }
        return true;
    }
};
```

相關題目

- Reverse Integer, 見 §15.1
- Valid Palindrome, 見 §3.1

15.3 Insert Interval

描述

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

Example 1: Given intervals

$1,3$

,

$6,9$

, insert and merge

$2,5$

in as

$1,5$

,

$6,9$

.

Example 2: Given

$1,2$

,

$3,5$

,

$6,7$

,

$8, 10$

,

$12, 16$

, insert and merge

$4, 9$

in as

$1, 2$

,

$3, 10$

,

$12, 16$

.

This is because the new interval

$4, 9$

overlaps with

$3, 5$

,

$6, 7$

,

$8, 10$

.

分析

無

代碼

```
struct Interval {
    int start;
    int end;
    Interval() : start(0), end(0) { }
    Interval(int s, int e) : start(s), end(e) { }
};

//LeetCode, Insert Interval
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
```

```
public:
    vector<Interval> insert(vector<Interval> &intervals, Interval newInterval) {
        vector<Interval>::iterator it = intervals.begin();
        while (it != intervals.end()) {
            if (newInterval.end < it->start) {
                intervals.insert(it, newInterval);
                return intervals;
            } else if (newInterval.start > it->end) {
                it++;
                continue;
            } else {
                newInterval.start = min(newInterval.start, it->start);
                newInterval.end = max(newInterval.end, it->end);
                it = intervals.erase(it);
            }
        }
        intervals.insert(intervals.end(), newInterval);
        return intervals;
    }
};
```

相關題目

- Merge Intervals，見 §15.4

15.4 Merge Intervals

描述

Given a collection of intervals, merge all overlapping intervals.

For example, Given

$$1, 3$$

,

$$2, 6$$

,

$$8, 10$$

,

$$15, 18$$

, return

$$1, 6$$

,

$$8, 10$$

,

$$15, 18$$

分析

複用一下 Insert Intervals 的解法即可，創建一個新的 interval 集合，然後每次從舊的裏面取一個 interval 出來，然後插入到新的集合中。

代碼

```
struct Interval {
    int start;
    int end;
    Interval() : start(0), end(0) { }
    Interval(int s, int e) : start(s), end(e) { }
};

//LeetCode, Merge Interval
//複用一下 Insert Intervals 的解法即可
// 時間複雜度 O(n1+n2+...), 空間複雜度 O(1)
class Solution {
public:
    vector<Interval> merge(vector<Interval> &intervals) {
        vector<Interval> result;
        for (int i = 0; i < intervals.size(); i++) {
            insert(result, intervals[i]);
        }
        return result;
    }
private:
    vector<Interval> insert(vector<Interval> &intervals, Interval newInterval) {
        vector<Interval>::iterator it = intervals.begin();
        while (it != intervals.end()) {
            if (newInterval.end < it->start) {
                intervals.insert(it, newInterval);
                return intervals;
            } else if (newInterval.start > it->end) {
                it++;
                continue;
            } else {
                newInterval.start = min(newInterval.start, it->start);
                newInterval.end = max(newInterval.end, it->end);
                it = intervals.erase(it);
            }
        }
        intervals.insert(intervals.end(), newInterval);
        return intervals;
    }
};
```

相關題目

- Insert Interval，見 §15.3

15.5 Minimum Window Substring

描述

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity $O(n)$.

For example, `S = "ADOBECODEBANC", T = "ABC"`

Minimum window is `"BANC"`.

Note:

- If there is no such window in S that covers all characters in T, return the emtpy string `""`.
- If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

分析

雙指針，動態維護一個區間。尾指針不斷往後掃，當掃到有一個窗口包含了所有 T 的字符後，然後再收縮頭指針，直到不能再收縮為止。最後記錄所有可能的情況中窗口最小的

代碼

```
// LeetCode, Minimum Window Substring
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    string minWindow(string S, string T) {
        if (S.empty()) return "";
        if (S.size() < T.size()) return "";

        const int ASCII_MAX = 256;
        int appeared_count[ASCII_MAX];
        int expected_count[ASCII_MAX];
        fill(appeared_count, appeared_count + ASCII_MAX, 0);
        fill(expected_count, expected_count + ASCII_MAX, 0);

        for (size_t i = 0; i < T.size(); i++) expected_count[T[i]]++;

        int minWidth = INT_MAX, min_start = 0;  // 窗口大小, 起點
        int wnd_start = 0;
        int appeared = 0;  // 完整包含了一個 T
        //尾指針不斷往後掃
        for (size_t wnd_end = 0; wnd_end < S.size(); wnd_end++) {
            if (expected_count[S[wnd_end]] > 0)  {  // this char is a part of T
                appeared_count[S[wnd_end]]++;
```

```
                if (appeared_count[S[wnd_end]] <= expected_count[S[wnd_end]])
                    appeared++;
            }
            if (appeared == T.size()) {  // 完整包含了一個 T
                // 收縮頭指針
                while (appeared_count[S[wnd_start]] > expected_count[S[wnd_start]]
                        || expected_count[S[wnd_start]] == 0) {
                    appeared_count[S[wnd_start]]--;
                    wnd_start++;
                }
                if (minWidth > (wnd_end - wnd_start + 1)) {
                    minWidth = wnd_end - wnd_start + 1;
                    min_start = wnd_start;
                }
            }
        }

        if (minWidth == INT_MAX) return "";
        else return S.substr(min_start, minWidth);
    }
};
```

相關題目

- 無

15.6 Multiply Strings

描述

Given two numbers represented as strings, return multiplication of the numbers as a string.

Note: The numbers can be arbitrarily large and are non-negative.

分析

高精度乘法。

常見的做法是將字符轉化為一個 int，一一對應，形成一個 int 數組。但是這樣很浪費空間，一個 int32 的最大值是 $2^{31} - 1 = 2147483647$，可以與 9 個字符對應，由於有乘法，減半，則至少可以與 4 個字符一一對應。一個 int64 可以與 9 個字符對應。

代碼 1

```
// LeetCode, Multiply Strings
// @author 連城 (http://weibo.com/lianchengzju)
// 一個字符對應一個 int
// 時間複雜度 O(n*m)，空間複雜度 O(n+m)
typedef vector<int> bigint;

bigint make_bigint(string const& repr) {
```

```
    bigint n;
    transform(repr.rbegin(), repr.rend(), back_inserter(n),
            [](char c) { return c - '0'; });
    return n;
}

string to_string(bigint const& n) {
    string str;
    transform(find_if(n.rbegin(), prev(n.rend()),
            [](char c) { return c > '\0'; }), n.rend(), back_inserter(str),
            [](char c) { return c + '0'; });
    return str;
}

bigint operator*(bigint const& x, bigint const& y) {
    bigint z(x.size() + y.size());

    for (size_t i = 0; i < x.size(); ++i)
        for (size_t j = 0; j < y.size(); ++j) {
            z[i + j] += x[i] * y[j];
            z[i + j + 1] += z[i + j] / 10;
            z[i + j] %= 10;
        }

    return z;
}

class Solution {
public:
    string multiply(string num1, string num2) {
        return to_string(make_bigint(num1) * make_bigint(num2));
    }
};
```

代碼 2

```
// LeetCode, Multiply Strings
// 9 個字符對應一個 int64_t
// 時間複雜度 O(n*m/81), 空間複雜度 O((n+m)/9)
/** 大整數類. */
class BigInt {
public:
    /**
     * @brief 構造函數, 將字符串轉化為大整數.
     * @param[in] s 輸入的字符串
     * @return 無
     */
    BigInt(string s) {
        vector<int64_t> result;
        result.reserve(s.size() / RADIX_LEN + 1);

        for (int i = s.size(); i > 0; i -= RADIX_LEN) {  // [i-RADIX_LEN, i)
```

```
        int temp = 0;
        const int low = max(i - RADIX_LEN, 0);
        for (int j = low; j < i; j++) {
            temp = temp * 10 + s[j] - '0';
        }
        result.push_back(temp);
    }
    elems = result;
}
/**
 * @brief 將整數轉化為字符串.
 * @return 字符串
 */
string toString() {
    stringstream result;
    bool started = false; // 用於跳過前導 0
    for (auto i = elems.rbegin(); i != elems.rend(); i++) {
        if (started) { // 如果多餘的 0 已經都跳過, 則輸出
            result << setw(RADIX_LEN) << setfill('0') << *i;
        } else {
            result << *i;
            started = true; // 碰到第一個非 0 的值, 就說明多餘的 0 已經都跳過
        }
    }

    if (!started) return "0"; // 當 x 全為 0 時
    else return result.str();
}

/**
 * @brief 大整數乘法.
 * @param[in] x x
 * @param[in] y y
 * @return 大整數
 */
static BigInt multiply(const BigInt &x, const BigInt &y) {
    vector<int64_t> z(x.elems.size() + y.elems.size(), 0);

    for (size_t i = 0; i < y.elems.size(); i++) {
        for (size_t j = 0; j < x.elems.size(); j++) { // 用 y[i] 去乘以 x 的各位
            //  兩數第 i, j 位相乘, 累加到結果的第 i+j 位
            z[i + j] += y.elems[i] * x.elems[j];

            if (z[i + j] >= BIGINT_RADIX) { //  看是否要進位
                z[i + j + 1] += z[i + j] / BIGINT_RADIX; //  進位
                z[i + j] %= BIGINT_RADIX;
            }
        }
    }
    while (z.back() == 0) z.pop_back();  // 沒有進位, 去掉最高位的 0
    return BigInt(z);
}
```

```
private:
    typedef long long int64_t;
    /** 一個數組元素對應 9 個十進制位, 即數組是億進制的
     * 因為 1000000000 * 1000000000 沒有超過 2^63-1
     */
    const static int BIGINT_RADIX = 1000000000;
    const static int RADIX_LEN = 9;
    /** 萬進制整數. */
    vector<int64_t> elems;
    BigInt(const vector<int64_t> num) : elems(num) {}
};


class Solution {
public:
    string multiply(string num1, string num2) {
        BigInt x(num1);
        BigInt y(num2);
        return BigInt::multiply(x, y).toString();
    }
};
```

相關題目

- 無

15.7 Substring with Concatenation of All Words

描述

You are given a string, S, and a list of words, L, that are all of the same length. Find all starting indices of substring(s) in S that is a concatenation of each word in L exactly once and without any intervening characters.

For example, given:

```
S: "barfoothefoobarman"
L: ["foo", "bar"]
```

You should return the indices:

$$0, 9$$

.(order does not matter).

分析

無

代碼

```
// LeetCode, Substring with Concatenation of All Words
// 時間複雜度 O(n*m), 空間複雜度 O(m)
```

```
class Solution {
public:
    vector<int> findSubstring(string s, vector<string>& dict) {
        size_t wordLength = dict.front().length();
        size_t catLength = wordLength * dict.size();
        vector<int> result;

        if (s.length() < catLength) return result;

        unordered_map<string, int> wordCount;

        for (auto const& word : dict) ++wordCount[word];

        for (auto i = begin(s); i <= prev(end(s), catLength); ++i) {
            unordered_map<string, int> unused(wordCount);

            for (auto j = i; j != next(i, catLength); j += wordLength) {
                auto pos = unused.find(string(j, next(j, wordLength)));

                if (pos == unused.end() || pos->second == 0) break;

                if (--pos->second == 0) unused.erase(pos);
            }

            if (unused.size() == 0) result.push_back(distance(begin(s), i));
        }

        return result;
    }
};
```

相關題目

- 無

15.8 Pascal's Triangle

描述

Given $numRows$, generate the first $numRows$ of Pascal's triangle.

For example, given $numRows = 5$,

Return

```
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

分析

本題可以用隊列，計算下一行時，給上一行左右各加一個 0，然後下一行的每個元素，就等於左上角和右上角之和。

另一種思路，下一行第一個元素和最後一個元素賦值為 1，中間的每個元素，等於上一行的左上角和右上角元素之和。

從左到右

```
// LeetCode, Pascal's Triangle
// 時間複雜度 O(n^2), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > generate(int numRows) {
        vector<vector<int> > result;
        if(numRows == 0) return result;

        result.push_back(vector<int>(1,1)); //first row

        for(int i = 2; i <= numRows; ++i) {
            vector<int> current(i,1);  // 本行
            const vector<int> &prev = result[i-2];  // 上一行

            for(int j = 1; j < i - 1; ++j) {
                current[j] = prev[j-1] + prev[j]; // 左上角和右上角之和
            }
            result.push_back(current);
        }
        return result;
    }
};
```

從右到左

```
// LeetCode, Pascal's Triangle
// 時間複雜度 O(n^2), 空間複雜度 O(n)
class Solution {
public:
    vector<vector<int> > generate(int numRows) {
        vector<vector<int> > result;
        vector<int> array;
        for (int i = 1; i <= numRows; i++) {
            for (int j = i - 2; j > 0; j--) {
                array[j] = array[j - 1] + array[j];
            }
            array.push_back(1);
            result.push_back(array);
        }
        return result;
```

```
    }
};
```

相關題目

- Pascal's Triangle II，見 §**??**

15.9 Pascal's Triangle II

描述

Given an index k, return the k^{th} row of the Pascal's triangle.

For example, given $k = 3$,

Return

$$1, 3, 3, 1$$

.

Note: Could you optimize your algorithm to use only $O(k)$ extra space?

分析

滾動數組。

代碼

```
// LeetCode, Pascal's Triangle II
// 滾動數組, 時間複雜度 O(n^2), 空間複雜度 O(n)
class Solution {
public:
        vector<int> getRow(int rowIndex) {
                vector<int> array;
                for (int i = 0; i <= rowIndex; i++) {
                        for (int j = i - 1; j > 0; j--){
                                array[j] = array[j - 1] + array[j];
                        }
                        array.push_back(1);
                }
                return array;
        }
};
```

相關題目

- Pascal's Triangle，見 §15.8

15.10 Spiral Matrix

描述

Given a matrix of $m \times n$ elements (m rows, n columns), return all elements of the matrix in spiral order.

For example, Given the following matrix:

```
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
```

You should return `[1,2,3,6,9,8,7,4,5]`.

分析

模擬。

代碼

```
// LeetCode, Spiral Matrix
// @author 龔陸安 (http://weibo.com/luangong)
// 時間複雜度 O(n^2), 空間複雜度 O(1)
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int> >& matrix) {
        vector<int> result;
        if (matrix.empty()) return result;
        int beginX = 0, endX = matrix[0].size() - 1;
        int beginY = 0, endY = matrix.size() - 1;
        while (true) {
            // From left to right
            for (int j = beginX; j <= endX; ++j) result.push_back(matrix[beginY][j]);
            if (++beginY > endY) break;
            // From top to bottom
            for (int i = beginY; i <= endY; ++i) result.push_back(matrix[i][endX]);
            if (beginX > --endX) break;
            // From right to left
            for (int j = endX; j >= beginX; --j) result.push_back(matrix[endY][j]);
            if (beginY > --endY) break;
            // From bottom to top
            for (int i = endY; i >= beginY; --i) result.push_back(matrix[i][beginX]);
            if (++beginX > endX) break;
        }
        return result;
    }
};
```

相關題目

- Spiral Matrix II，見 §15.11

15.11 Spiral Matrix II

描述

Given an integer n, generate a square matrix filled with elements from 1 to n^2 in spiral order.

For example, Given $n = 3$,

You should return the following matrix:

```
[
 [ 1, 2, 3 ],
 [ 8, 9, 4 ],
 [ 7, 6, 5 ]
]
```

分析

這題比上一題要簡單。

代碼 1

```
// LeetCode, Spiral Matrix II
// 時間複雜度 O(n^2), 空間複雜度 O(n^2)
class Solution {
public:
    vector<vector<int> > generateMatrix(int n) {
        vector<vector<int> > matrix(n, vector<int>(n));
        int begin = 0, end = n - 1;
        int num = 1;

        while (begin < end) {
            for (int j = begin; j < end; ++j) matrix[begin][j] = num++;
            for (int i = begin; i < end; ++i) matrix[i][end] = num++;
            for (int j = end; j > begin; --j) matrix[end][j] = num++;
            for (int i = end; i > begin; --i) matrix[i][begin] = num++;
            ++begin;
            --end;
        }

        if (begin == end) matrix[begin][begin] = num;

        return matrix;
    }
};
```

代碼 2

```
// LeetCode, Spiral Matrix II
// @author 龔陸安 (http://weibo.com/luangong)
// 時間複雜度 O(n^2), 空間複雜度 O(n^2)
class Solution {
public:
    vector<vector<int> > generateMatrix(int n) {
        vector< vector<int> > matrix(n, vector<int>(n));
        if (n == 0) return matrix;
        int beginX = 0, endX = n - 1;
        int beginY = 0, endY = n - 1;
        int num = 1;
        while (true) {
            for (int j = beginX; j <= endX; ++j) matrix[beginY][j] = num++;
            if (++beginY > endY) break;

            for (int i = beginY; i <= endY; ++i) matrix[i][endX] = num++;
            if (beginX > --endX) break;

            for (int j = endX; j >= beginX; --j) matrix[endY][j] = num++;
            if (beginY > --endY) break;

            for (int i = endY; i >= beginY; --i) matrix[i][beginX] = num++;
            if (++beginX > endX) break;
        }
        return matrix;
    }
};
```

相關題目

- Spiral Matrix, 見 §15.10

15.12 ZigZag Conversion

描述

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
P   A   H   N
A P L S I I G
Y   I   R
```

And then read line by line: "PAHNAPLSIIGYIR"

Write the code that will take a string and make this conversion given a number of rows:

```
string convert(string text, int nRows);
```

convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".

分析

要找到數學規律。真正面試中，不大可能出這種問題。

n=4:

```
P     I     N
A   L S   I G
Y A   H R
P     I
```

n=5:

```
P       H
A     S I
Y   I   R
P L     I  G
A       N
```

所以，對於每一層垂直元素的座標 $(i,j) = (j+1) * n + i$；對於每兩層垂直元素之間的插入元素（斜對角元素），$(i,j) = (j+1) * n - i$

代碼

```
// LeetCode, ZigZag Conversion
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    string convert(string s, int nRows) {
        if (nRows <= 1 || s.size() <= 1) return s;
        string result;
        for (int i = 0; i < nRows; i++) {
            for (int j = 0, index = i; index < s.size();
                    j++, index = (2 * nRows - 2) * j + i) {
                result.append(1, s[index]);  // 垂直元素
                if (i == 0 || i == nRows - 1) continue;   // 斜對角元素
                if (index + (nRows - i - 1) * 2 < s.size())
                    result.append(1, s[index + (nRows - i - 1) * 2]);
            }
        }
        return result;
    }
};
```

相關題目

- 無

15.13 Divide Two Integers

描述

Divide two integers without using multiplication, division and mod operator.

分析

不能用乘、除和取模，那剩下的，還有加、減和位運算。

最簡單的方法，是不斷減去被除數。在這個基礎上，可以做一點優化，每次把被除數翻倍，從而加速。

代碼 1

```
// LeetCode, Divide Two Integers
// 時間複雜度 O(logn), 空間複雜度 O(1)
class Solution {
public:
    int divide(int dividend, int divisor) {
        // 當 dividend = INT_MIN 時, -dividend 會溢出, 所以用 long long
        long long a = dividend >= 0 ? dividend : -(long long)dividend;
        long long b = divisor >= 0 ? divisor : -(long long)divisor;

        // 當 dividend = INT_MIN 時, divisor = -1 時, 結果會溢出, 所以用 long long
        long long result = 0;
        while (a >= b) {
            long long c = b;
            for (int i = 0; a >= c; ++i, c <<= 1) {
                a -= c;
                result += 1 << i;
            }
        }

        return ((dividend^divisor) >> 31) ? (-result) : (result);
    }
};
```

代碼 2

```
// LeetCode, Divide Two Integers
// 時間複雜度 O(logn), 空間複雜度 O(1)
class Solution {
public:
    int divide(int dividend, int divisor) {
        int result = 0; // 當 dividend = INT_MIN 時, divisor = -1 時, 結果會溢出
        const bool sign = (dividend > 0 && divisor < 0) ||
                (dividend < 0 && divisor > 0); // 異號

        // 當 dividend = INT_MIN 時, -dividend 會溢出, 所以用 unsigned int
        unsigned int a = dividend >= 0 ? dividend : -dividend;
        unsigned int b = divisor >= 0 ? divisor : -divisor;

        while (a >= b) {
            int multi = 1;
            unsigned int bb = b;
            while (a >= bb) {
                a -= bb;
```

```
                result += multi;

                if (bb < INT_MAX >> 1) { // 防止溢出
                    bb += bb;
                    multi += multi;
                }
            }
        }
        if (sign) return -result;
        else return result;
    }
};
```

相關題目

- 無

15.14 Text Justification

描述

Given an array of words and a length L, format the text such that each line has exactly L characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces `' '` when necessary so that each line has exactly L characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

For example,

words:

$$"This", "is", "an", "example", "of", "text", "justification."$$

L: 16.

Return the formatted lines as:

```
[
   "This    is    an",
   "example  of text",
   "justification.  "
]
```

Note: Each word is guaranteed not to exceed L in length.

Corner Cases:

- A line other than the last line might contain only one word. What should you do in this case?
- In this case, that line should be left

分析

無

代碼

```
// LeetCode, Text Justification
// 時間複雜度 O(n), 空間複雜度 O(1)
class Solution {
public:
    vector<string> fullJustify(vector<string> &words, int L) {
        vector<string> result;
        const int n = words.size();
        int begin = 0, len = 0; // 當前行的起點, 當前長度
        for (int i = 0; i < n; ++i) {
            if (len + words[i].size() + (i - begin) > L) {
                result.push_back(connect(words, begin, i - 1, len, L, false));
                begin = i;
                len = 0;
            }
            len += words[i].size();
        }
        // 最後一行不足 L
        result.push_back(connect(words, begin, n - 1, len, L, true));
        return result;
    }
    /**
     * @brief 將 words[begin, end] 連成一行
     * @param[in] words 單詞列表
     * @param[in] begin 開始
     * @param[in] end 結束
     * @param[in] len words[begin, end] 所有單詞加起來的長度
     * @param[in] L 題目規定的一行長度
     * @param[in] is_last 是否是最後一行
     * @return 對齊後的當前行
     */
    string connect(vector<string> &words, int begin, int end,
            int len, int L, bool is_last) {
        string s;
        int n = end - begin + 1;
        for (int i = 0; i < n; ++i) {
            s += words[begin + i];
            addSpaces(s, i, n - 1, L - len, is_last);
        }

        if (s.size() < L) s.append(L - s.size(), ' ');
        return s;
    }

    /**
     * @brief 添加空格.
     * @param[inout]s 一行
```

```
     * @param[in] i 當前空隙的序號
     * @param[in] n 空隙總數
     * @param[in] L 總共需要添加的空額數
     * @param[in] is_last 是否是最後一行
     * @return 無
     */
    void addSpaces(string &s, int i, int n, int L, bool is_last) {
        if (n < 1 || i > n - 1) return;
        int spaces = is_last ? 1 : (L / n + (i < (L % n) ? 1 : 0));
        s.append(spaces, ' ');
    }
};
```

相關題目

- 無

15.15 Max Points on a Line

描述

Given n points on a 2D plane, find the maximum number of points that lie on the same straight line.

分析

暴力枚舉法。兩點決定一條直線，n 個點兩兩組合，可以得到 $\frac{1}{2}n(n+1)$ 條直線，對每一條直線，判斷 n 個點是否在該直線上，從而可以得到這條直線上的點的個數，選擇最大的那條直線返回。複雜度 $O(n^3)$。

上面的暴力枚舉法以“邊”為中心，再看另一種暴力枚舉法，以每個“點”為中心，然後遍歷剩餘點，找到所有的斜率，如果斜率相同，那麼一定共線對每個點，用一個哈希表，key 為斜率，value 為該直線上的點數，計算出哈希表後，取最大值，並更新全局最大值，最後就是結果。時間複雜度 $O(n^2)$，空間複雜度 $O(n)$。

以邊為中心

```
// LeetCode, Max Points on a Line
// 暴力枚舉法，以邊為中心，時間複雜度 O(n^3)，空間複雜度 O(1)
class Solution {
public:
    int maxPoints(vector<Point> &points) {
        if (points.size() < 3) return points.size();
        int result = 0;

        for (int i = 0; i < points.size() - 1; i++) {
            for (int j = i + 1; j < points.size(); j++) {
                int sign = 0;
                int a, b, c;
                if (points[i].x == points[j].x) sign = 1;
```

```
                else {
                    a = points[j].x - points[i].x;
                    b = points[j].y - points[i].y;
                    c = a * points[i].y - b * points[i].x;
                }
                int count = 0;
                for (int k = 0; k < points.size(); k++) {
                    if ((0 == sign && a * points[k].y == c +  b * points[k].x) ||
                        (1 == sign&&points[k].x == points[j].x))
                        count++;
                }
                if (count > result) result = count;
            }
        }
        return result;
    }
};
```

以點為中心

```
// LeetCode, Max Points on a Line
// 暴力枚舉, 以點為中心, 時間複雜度 O(n^2), 空間複雜度 O(n)
class Solution {
public:
    int maxPoints(vector<Point> &points) {
        if (points.size() < 3) return points.size();
        int result = 0;

        unordered_map<double, int> slope_count;
        for (int i = 0; i < points.size()-1; i++) {
            slope_count.clear();
            int samePointNum = 0; // 與 i 重合的點
            int point_max = 1;    // 和 i 共線的最大點數

            for (int j = i + 1; j < points.size(); j++) {
                double slope; // 斜率
                if (points[i].x == points[j].x) {
                    slope = std::numeric_limits<double>::infinity();
                    if (points[i].y == points[j].y) {
                        ++ samePointNum;
                        continue;
                    }
                } else {
                    slope = 1.0 * (points[i].y - points[j].y) /
                        (points[i].x - points[j].x);
                }

                int count = 0;
                if (slope_count.find(slope) != slope_count.end())
                    count = ++slope_count[slope];
                else {
                    count = 2;
```

```
                    slope_count[slope] = 2;
                }

                if (point_max < count) point_max = count;
            }
            result = max(result, point_max + samePointNum);
        }
        return result;
    }
};
```

相關題目

- 無

第 16 章

Remake Data Structure

There are examples to try to remake those data strucutre in standard libraries for a practice purpose.

16.1 Smart Pointer

描述

Know more about smart pointer and reference counting.

分析

Nil

代碼

```
#include <iostream>
#include "DefaultMutex.h"

class ReferenceCount
{
 public:
    ReferenceCount();
    ~ReferenceCount();

    void AddRef();
    int Release();
 private:
    int m_count;
    DefaultMutex m_mutex;
};

ReferenceCount::ReferenceCount()
{
    m_count = 0;
}

ReferenceCount::~ReferenceCount()
{
}
```

```
void ReferenceCount::AddRef()
{
    DefaultLock lock(m_mutex);
    m_count++;
}

int ReferenceCount::Release()
{
    DefaultLock lock(m_mutex);
    if (m_count > 0)
    {
        return --m_count;
    }
    else
    {
        m_count = 0;
        return m_count;
    }
}

template<typename T>
class SmartPointer
{
 public:
    SmartPointer();
    SmartPointer(T* pData);
    SmartPointer(const SmartPointer<T>& smartPointer);
    template<typename U>
        SmartPointer(const SmartPointer<U>& smartPointer, T* pData);
    ~SmartPointer();

    T& operator* ();
    T* operator-> ();
    SmartPointer<T>& operator = (const SmartPointer<T>& smartPointer);

    T* Get();
    T* Get() const;

    template<typename> friend class SmartPointer; // with this friend, we can define the follo
 private:
    T* m_pData;
    ReferenceCount* m_pRC;
};

template<typename T>
SmartPointer<T> MakeSmartPointer();

template<typename T, typename U>
    SmartPointer<T> StaticCast(const SmartPointer<U>& smartPointer);

template<typename T, typename U>
    SmartPointer<T> DynamicCast(const SmartPointer<U>& smartPointer);
```

```
////////////////////////////////////////////////////////////////////////////////
// Start implementation
////////////////////////////////////////////////////////////////////////////////

template<typename T>
SmartPointer<T>::SmartPointer()
    : m_pData(NULL), m_pRC(NULL)
{
    m_pRC = new ReferenceCount();
}

template<typename T>
SmartPointer<T>::SmartPointer(T* pData)
    : m_pData(pData), m_pRC(NULL)
{
    m_pRC = new ReferenceCount();
    m_pRC->AddRef();
}

template<typename T>
SmartPointer<T>::SmartPointer(const SmartPointer<T>& smartPointer)
    : m_pData(smartPointer.m_pData), m_pRC(smartPointer.m_pRC)
{
    m_pRC->AddRef();
}

template<typename T>
template<typename U>
SmartPointer<T>::SmartPointer(const SmartPointer<U>& smartPointer, T* pData)
    : m_pData(pData), m_pRC(smartPointer.m_pRC)
{
    m_pRC->AddRef();
}

template<typename T>
SmartPointer<T>::~SmartPointer()
{
    if (m_pRC->Release() == 0)
    {
        if (m_pData)
            delete m_pData;
        delete m_pRC;
    }
}

template<typename T>
T& SmartPointer<T>::operator* ()
{
    return *m_pData;
}

template<typename T>
```

```
T* SmartPointer<T>::operator-> ()
{
    return m_pData;
}

template<typename T>
SmartPointer<T>& SmartPointer<T>::operator = (const SmartPointer<T>& smartPointer)
{
    if (this != &smartPointer)
    {
        if (m_pRC->Release() == 0)
        {
            if (m_pData)
                delete m_pData;
            delete m_pRC;
        }

        m_pData = smartPointer.m_pData;
        m_pRC = smartPointer.m_pRC;
        m_pRC->AddRef();
    }
    return *this;
}

template<typename T>
T* SmartPointer<T>::Get()
{
    return m_pData;
}

template<typename T>
T* SmartPointer<T>::Get() const
{
 return m_pData;
}

template<typename T>
SmartPointer<T> MakeSmartPointer()
{
    SmartPointer<T> smartPointer(new T());
    return smartPointer;
}

template<typename T, typename U>
SmartPointer<T> StaticCast(const SmartPointer<U>& smartPointer)
{
    T* p = static_cast<T*>(smartPointer.Get());
    return SmartPointer<T>(smartPointer, p);
}

template<typename T, typename U>
SmartPointer<T> DynamicCast(const SmartPointer<U>& smartPointer)
{
```

```
    T* p = dynamic_cast<T*>(smartPointer.Get());
    if (p)
        return SmartPointer<T>(smartPointer, p);
    else
        return SmartPointer<T>();
}
```

相關題目

Nil