Table of Contents

Leetcode/Lintcode学习笔记

作为一个算法小白，记录整理自己在学习算法过程中的心得，希望对你也有所帮助。

目录

算法学习

如何高效地准备算法面试

- 在刷题时，总结、归类相似题目
- 找出适合同一类题目的模板程序
- 每隔固定时间重做以前的题目

关于代码风格

- 代码块可为三大块：异常处理（空串和边界处理），主体，返回
- 代码风格（可参考 Google 的编程语言规范）
    - 变量名：有意义的变量名
    - 缩进：语句块
    - 空格：二元运算符、括号两侧
    - 可读性：单语句使用花括号等

关于面试

面试官眼中的求职者：你可能是他未来的同事，那么

- 你的代码看起来舒服么
    - TA 需要多少时间 review 你的代码

- 你写代码的习惯好么
  - TA 不会在未来疲于帮你 DEBUG，你不会动不动就搞出 SEV（SEV是什么？）
- 你的沟通能力好么
  - TA 和交流费劲么

面试要考察的编程基本功

- 程序风格（缩进，括号，变量名等）
- Coding 习惯（异常检查，边界处理，数组越界等）
- 沟通（让面试官时刻明白你的意图）
- 测试（主动写出合理的测试例）

在线刷题网站

1. Leetcode，应该是最知名的在线刷题网站了，提供多种编程语言的在线检测，评论区氛围很好，大家会分享思路、讨论问题。
2. Lintcode，和 Leetcode 很像，题目也有一定的重合度，没有评论区。有阶梯训练（ Ladder ），方便专题学习。
3. 牛客网，国内的一个 IT 笔试面试备考平台，有公司真题模考（主要是国内），智能专项练习，在线编程专题多个内容，讨论区有很多校招内容及分享，适合国内找工作的同学。

题目参考解答

1. 九章算法，提供 Lintcode 三种语言（ Java, C++, Python ）的解题答案，我个人在学习过程中大量参考了九章的答案，在此致谢。
2. Leetcode 讨论区，大家会讨论分享交流不同的解法和思路。
3. 数据结构与算法/leetcode/lintcode题解，作者 yuanbin 维护的个人博客，包含算法基础知识、编程题目解析等内容，题目解析部分很详细，本文档的建立也是受其启发，过程中也借鉴学习了很多，在此表示感谢。
4. ProgramCreek，个人博客站点，里面有大量关于算法和 Java 相关的内容。Simple Java 中总结的知识点很适合面试前参考。
5. 水中的鱼，leetcode 题目的答案解析， google 搜索的排名很高。
6. 喜刷刷，跟“水中的鱼”很像。
7. 算法精粹 | soulmachine ，soulmachine 的习题汇编在线阅读地址，有 C++ 和 Java 两个版本。

参考书籍

1. 《算法（第4版）》（豆瓣），Robert Sedgewick 教授的书，讲解了基础的数据结构和算法，配有大量图例，深入浅出，讲解非常细致。结合他老人家在 Coursera 开的两门课程看，效果更佳。课程分别是 Algorithms, Part I (Coursera, Youtube) 和 Algorithms, Part II (Coursera, Youtube) 。顺带提一下 Robert 教授是 Donald Knuth 的学生，功力深厚。

其他资源

1. 刷题版 | 一亩三分地，有很多刷题、准备面试的经验分享。
2. visualgo ，数据结构和算法的可视化演示，对理解算法有帮助。
3. Data Structure Visualizations ，另一个算法可视化演示网站。

strStr

strStr (leetcode lintcode)

```
For a given source string and a target string, you should output
 the first index(from 0) of target string in source string.

If target does not exist in source, just return -1.

Clarification
- Do I need to implement KMP Algorithm in a real interview?
- Not necessary. When you meet this problem in a real interview,
 the interviewer may just want to test your basic implementation
 ability. But make sure your confirm with the interviewer first.

Example
If source = "source" and target = "target", return -1.
If source = "abcdabcdefg" and target = "bcd", return 1.

Challenge
O(n2) is acceptable. Can you implement an O(n) algorithm? (hint:
 KMP)
```

解题思路：

一、穷举法

步骤

- 从源字符串的起始位置开始，逐一与目标字符串进行比较，直至找到，或者搜索结束。
- 具体实现为双重for循环，外循环遍历源字符串，范围为 `0 ~ source.length()-target.length()` ，内循环遍历目标字符串，范围 `0 ~ target.length()-1` 。

算法复杂度：

- 时间复杂度最坏情况为 `O((n - m) \* m)` 次比较，例如 `source = "aaaa…`

`aaaaab", target = "aaab"` 。

```
class Solution {
    /**
     * Returns a index to the first occurrence of target in sour
ce,
     * or -1  if target is not part of source.
     * @param source string to be scanned.
     * @param target string containing the sequence of character
s to match.
     */
    public int strStr(String source, String target) {
        //write your code here
        if (source == null || target == null) {
            return -1;
        }
        int M = target.length();
        int N = source.length();
        //此处应注意i的取值范围
        for (int i = 0; i <= N - M; i++) {
            int j;  //注意此处声明j的位置，要考虑后面j==M判断
            for (j = 0; j < M; j++) {
                //charAt()是一个方法，所以要使用"()”，而不应使用"[]"
                if (source.charAt(i + j) != target.charAt(j)) {
                    break;   //如果遇到不同的元素，从循环中跳出
                }
            }
            if (j == M) {
                return i;   //返回发现字符串的起始位置
            }
        }
        return -1;   //未找到目标字符串
    }
}
```

注：

- num == null 和 num.length == 0的区别是什么？为何在判断边界条件时都需要判断？

- num == null的意思是，你没钱也没有钱包。（nums在内存中没有一个对应的空间，连存储length的地方都没有）

  num.length == 0 的意思是， 你有钱包，你钱包里没钱。（内存中开辟了一片nums的空间，但是一个数都没放进去，但是这篇内存空间中存了length，且length的值是0）

二、**KMP(Knuth-Morris-Pratt)**

类似题目：

延伸阅读：

1. 字符串查找可以用在诸多领域，如：
   - 垃圾邮件检测：识别特定字符串
   - 电子监视：识别网络流量中的特定字符串
   - 屏幕抓取：从网页中提取相关内容
   - Java函数库：indexOf()方法返回指定字符串出现的第一次位置

Reverse Words in a String

Reverse Words in a String (leetcode lintcode)

```
Description
Given an input string, reverse the string word by word.
For example,
Given s = "the sky is blue",
return "blue is sky the".

Clarification
- What constitutes a word?
  A sequence of non-space characters constitutes a word.
- Could the input string contain leading or trailing spaces?
  Yes. However, your reversed string should not contain leading
or trailing spaces.
- How about multiple spaces between two words?
  Reduce them to a single space in the reversed string.
```

解题思路

本题中只有空格字符、非空格字符之分，所以在处理单词时，如何处理空格是比较关键的。

一、使用 `split` 函数

Java 中的 `split` 函数可以根据一定的规则将字符串分割为一个字符串数组，

> 函数声明： `public String[] split(String regex)`
>
> 以字符串 `“boo:and:foo”` 为例，不同的参数可以得到不同的结果。
>
> 参数 结果
>
> `:` `{ "boo", "and", "foo" }`
>
> `o` `{ "b", "", ":and:f" }`

在实现过程还用到了 `StringBuilder` 类型，简要介绍一下它与 `String` ，`StringBuffer` 的异同：

	String	StringBuffer	StringBuilder
存储区域	常量字符串池	堆	堆
可修改性	不可修改	可修改	可修改
线程安全性	线程安全	线程安全	非线程
性能	快	很慢	快
适用情况	字符串无需修改	字符串需修改，且会被多个线程访问	字符串需修改，只会被一个线程访问

Java 实现

```
public class Solution {
    /**
     * @param s : A string
     * @return : A string
     */
    public String reverseWords(String s) {
        if (s == null || s.length() == 0) {
            return "";
        }

        String[] array = s.split(" ");
        StringBuilder sb = new StringBuilder();

        for (int i = array.length - 1; i >= 0; i--) {
            if (!array[i].equals("")) {
                sb.append(array[i]).append(" ");
            }
        }

        // remove the last " "
        return sb.length() == 0 ? "" : sb.substring(0, sb.length
() - 1);
    }
}
```

二、逐个字符处理

题目要求所有单词要逆序排列，从前往后取单词比较麻烦，考虑从后往前处理。在处理空格时需要小心，如何判断处理“字符串最后的空格”和“单词之间的空格”。

- 当遇到“字符串最后的空格”时，此时 `subString` 为空，以此为判断条件跳过空格。
- 当取出单词，遇到第一个“单词之间的空格”时，此时 `subString` 非空，将其添加入 `result` 中。如果此时 `result` 非空要加空格。之后将 `subString` 置为空。

感觉上述实现稍微有点绕 = =!!!

Java 实现

```java
public class Solution {
    /**
     * @param s : A string
     * @return : A string
     */
    public String reverseWords(String s) {
        if (s == null || s.length() == 0) {
            return s;
        }

        StringBuilder subString = new StringBuilder();
        StringBuilder result = new StringBuilder();

        for (int i = s.length() - 1; i >= 0; i--) {
            if (s.charAt(i) != ' ') {
                subString.append(s.charAt(i));
            } else if (subString.length() != 0) {
                if (result.length() != 0) {
                    result.append(" ");
                }
                result.append(subString.reverse());
                subString = new StringBuilder();
            }
        }

        if (subString.length() != 0) {
            if (result.length() != 0) {
                result.append(" ");
            }
            result.append(subString.reverse());
        }
        return result.toString();
    }
}
```

参考

1. Reverse Words in a String | 九章算法

2. Difference Between String , StringBuilder And StringBuffer Classes With Example : Java | JAVA HUNGRY
3. String, StringBuffer, and StringBuilder | stackoverflow
4. [LeetCode] Reverse Words in a String | 努橙刷题编

Rotate String

Rotate String (leetcode lintcode)

```
Description
Given a string and an offset, rotate string by offset. (rotate f
rom left to right)

Example
Given "abcdefg".
offset=0 => "abcdefg"
offset=1 => "gabcdef"
offset=2 => "fgabcde"
offset=3 => "efgabcd"

Challenge
Rotate in-place with O(1) extra memory.
```

解题思路

参考题目 Recover Rotated Sorted Array ，使用三步翻转法。

算法复杂度

- 时间复杂度： `O(n)` 。
- 空间复杂度： `O(1)` 。

易错点

> 1. 在写交换函数时，注意临时变量的类型，这里是 `char` 类型。

Java 实现

```
public class Solution {
    /**
     * @param str: an array of char
     * @param offset: an integer
     * @return: nothing
     */
    public void rotateString(char[] str, int offset) {
        if (str == null || str.length == 0) {
            return;
        }

        offset %= str.length;
        if (offset == 0) {
            return;
        }

        reverse(str, 0, str.length - offset - 1);
        reverse(str, str.length - offset, str.length - 1);
        reverse(str, 0, str.length - 1);
    }

    private void reverse (char[] str, int start, int end) {
        for (int i = start, j = end; i < j; i++, j--) {
            char temp = str[i];
            str[i] = str[j];
            str[j] = temp;
        }
    }
}
```

Binary Search

Binary Search

```
Binary search is a famous question in algorithm.
For a given sorted array (ascending order) and a target number,
find the first index of this number in O(log n) time complexity.

If the target number does not exist in the array, return -1.

Example
If the array is [1, 2, 3, 3, 4, 5, 10], for given target 3, retu
rn 2.
```

解题思路

这里参考了九章算法 `binary search` 的解题模板：

- `start + 1 < end`
- `mid = start + (end - start) / 2`
- `A[mid] ==, <, >`
- `A[start] A[end] ? target`

注：

1. 写为 `start + 1 < end` 主要是为了防止出现死循环。如果写成 `start<end` ，那么当 `start = 1` ， `end = 2` 时， `mid = start + (end - start)/2 = 1` ，如果在判决条件下得到 `start = mid;` ， `start` 不变，会出现死循环。
2. 之所以不使用 `mid = (start + end ) / 2` ，是为了防止 `start` 和 `end` 都很大的时候溢出。

算法复杂度：

- 时间复杂度： `O(log n)` 。
- 空间复杂度： `O(1)` 。

Java 实现

```java
public class Solution {
    /**
     * @param A an integer array sorted in ascending order
     * @param target an integer
     * @return an integer
     */
    public int findPosition(int[] nums, int target) {
        if (nums == null || nums.length == 0) {
            return -1;
        }

        int start = 0, end = nums.length - 1;
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < target) {
                start = mid;
            } else {
                end = mid;
            }
        }

        if (nums[start] == target) {
            return start;
        }
        if (nums[end] == target) {
            return end;
        }
        return -1;
    }
}
```

另一种常规的实现方法，注意比较 `start` 和 `end` 在移动时与第一种方法的区别：

```java
public class Solution {
    /**
     * @param A an integer array sorted in ascending order
     * @param target an integer
     * @return an integer
     */
    public int findPosition(int[] nums, int target) {
        if (nums == null || nums.length == 0) {
            return -1;
        }

        int start = 0, end = nums.length - 1;
        while (start < end) {
            int mid = start + (end - start) / 2;
            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < target) {
                start = mid + 1;
            } else {
                end = mid - 1;
            }
        }

        if (nums[start] == target) {
            return start;
        }
        return -1;
    }
}
```

参考

1. Binary Search | 九章算法

First Position of Target

First Position of Target (leetcode lintcode)

```
For a given sorted array (ascending order) and a target number,
find the first index of this number in O(log n) time complexity.
If the target number does not exist in the array, return -1.

Example
If the array is [1, 2, 3, 3, 4, 5, 10], for given target 3, retu
rn 2.

Challenge
If the count of numbers is bigger than 2^32, can your code work
properly?
```

解题思路

数组中有重复元素，在二分查找确定目标值第一次出现位置时，中间元素 `nums[mid] == target` ，有可能左半部分还有相同元素，所以此时应该把 `end` 位置移动到 `mid` 。

算法复杂度：

- 时间复杂度： `O(log n)` 。
- 空间复杂度： `O(1)` 。

Java 实现

```
class Solution {
    /**
     * @param nums: The integer array.
     * @param target: Target to find.
     * @return: The first position of target. Position starts fr
om 0.
     */
    public int binarySearch(int[] nums, int target) {
        //write your code here
        if (nums == null || nums.length == 0){
            return -1;
        }

        int start = 0, end = nums.length - 1;
        while(start + 1 < end) {
            int mid =  start + (end - start)/2;
            if (nums[mid] == target) {
                end = mid;
            } else if (nums[mid] < target) {
                start = mid;
            } else if (nums[mid] > target) {
                end =  mid;
            }
        }

        if (nums[start] == target) {
            return start;
        }
        if (nums[end] == target) {
            return end;
        }
        return -1;
    }
}
```

Search for a Range

Search for a Range (leetcode lintcode)

```
Given a sorted array of integers, find the starting and ending p
osition of a given target value.
Your algorithm's runtime complexity must be in the order of O(lo
g 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].

Challenge
O(log n) time.
```

解题思路

数组中含重复元素，要寻找某个目标值的范围（起始位置），先找第一次出现的位置，再找最后一次出现的位置。具体实现可参考题目 First Position of Target 。

算法复杂度：

- 时间复杂度： `O(log n)` 。
- 空间复杂度： `O(1)` 。

易错点：

> 1. 在函数返回值时，建立数组并直接进行初始化： `return new int[]{-1, -1};` 。
>
>   直接声明初始化数组： `int[] a = {1, 2, 3, 4};` 。

Java 实现

```
public class Solution {
```

```
    /**
     *@param A : an integer sorted array
     *@param target :  an integer to be inserted
     *return : a list of length 2, [index1, index2]
     */
    public int[] searchRange(int[] A, int target) {

        if (A == null || A.length == 0) {
            return new int[] {-1, -1};     // attention: how Jav
a function return array
        }

        int start = 0, end = A.length - 1;
        int first, last;
        // find first target
        while (start + 1 < end) {
            int mid = start + (end -  start) / 2;
            if (A[mid] >= target) {
                end = mid;
            } else {
                start = mid;
            }
        }
        if (A[start] == target) {
            first = start;
        } else if (A[end] == target){
            first = end;
        } else {
            first = last = -1;
            return new int[] {first, last};     // return two di
mension array
        }

        // find last target
        start = 0;
        end = A.length - 1;
        while (start + 1 < end) {
            int mid = start + (end -  start) / 2;
            if (A[mid] <= target) {
                start = mid;
```

```
            } else {
                end = mid;
            }
        }
        if (A[end] == target) {
            last = end;
        } else if (A[start] == target) {
            last = start;
        } else {
            last = first = -1;
            return new int[] {first, last};
        }
        return new int[] {first, last};    // return the final p
osition
    }
}
```

Search Insert Position

Search Insert Position (leetcode lintcode)

```
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 i
n order.
You may assume NO duplicates in the array.

Example
[1,3,5,6], 5 → 2
[1,3,5,6], 2 → 1
[1,3,5,6], 7 → 4
[1,3,5,6], 0 → 0

Challenge
O(log(n)) time
```

解题思路

该题目变换一下说法，可以是：

- 寻找第一个大于等于 `target` 的位置。
- 寻找最后一个小于等于 `target` 的位置。

此外，插入元素和查找元素的区别在于，可能插入到数组最后一个位置的后面，此时对应 `target > A[n - 1]` （目标值大于数组最大值）。

算法复杂度：

- 时间复杂度： `O(log n)` 。
- 空间复杂度： `O(1)` 。

易错点：

> 1. 在判断边界条件时，如果数组为空，或者不为空但是不含任何元素，那么插入位置就是 0 。
> 2. 记得目标值大于数组中所有元素这种情况！

Java 实现

```
public class Solution {
    public int searchInsert(int[] A, int target) {
        if (A == null || A.length == 0) {
            return 0;    // if there is no elements in array A,
then insert position 0
        }

        int start = 0, end = A.length - 1;
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (A[mid] == target) {
                return mid;
            } else if (A[mid] < target) {
                start = mid;
            } else {
                end = mid;
            }
        }

        if (A[start] >= target) {
            return start;
        } else if (A[end] >= target) {
            return end;
        } else {
            return end + 1;
        }
    }
}
```

另一种实现方法

```
public class Solution {
    /**
     * param A : an integer sorted array
     * param target :  an integer to be inserted
     * return : an integer
     */
    public int searchInsert(int[] A, int target) {

        if (A == null || A.length == 0) {
            return 0;        // if there is no elements in array
 A, then insert position 0
        }
        int start = 0, end = A.length - 1;
        if (A[end] < target) {
            return end + 1;
        }
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (A[mid] == target){
                return mid;
            }else if (A[mid] < target) {
                start = mid;
            }else if (A[mid] > target) {
                end = mid;
            }
        }
        if (A[start] >= target) {
            return start;
        }
        if (A[end] >= target) {
            return end;
        }
        return -1;
    }
}
```

参考

1. Search Insert Position | 九章算法

Search a 2D Matrix

Search a 2D Matrix (leetcode lintcode)

```
Write an efficient algorithm that searches for a value in an m x
 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 integ
er of the previous row.

Example
Consider the following matrix:
[
    [1, 3, 5, 7],
    [10, 11, 16, 20],
    [23, 30, 34, 50]
]
Given target = 3, return true.

Challenge
O(log(n) + log(m)) time
```

解题思路

一、两次二分法

考虑到二维数组的性质，先寻找对应的行，然后在行中寻找对应位置。

其中，寻找行的问题就是寻找最后一个小于 target 的值的位置。

算法复杂度：

- 时间复杂度： `O(log m) + O(log n)` 。
- 空间复杂度： `O(1)` 。

Java 实现：

```java
// Binary Search Twice
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0) {
            return false;
        }
        if (matrix[0] == null || matrix[0].length == 0) {
            return false;
        }

        int row = matrix.length;
        int column = matrix[0].length;

        // find the row index, the last number <= target
        int start = 0, end = row - 1;
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (matrix[mid][0] == target) {
                return true;
            } else if (matrix[mid][0] < target) {
                start = mid;
            } else {
                end = mid;
            }
        }
        if (matrix[end][0] <= target) {
            row = end;
        } else if (matrix[start][0] <= target) {
            row = start;
        } else {
            return false;
        }

        // find the column index, the number equal to target
        start = 0;
        end = column - 1;
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (matrix[row][mid] == target) {
                return true;
```

```
            } else if (matrix[row][mid] < target) {
                start = mid;
            } else {
                end = mid;
            }
        }
        if (matrix[row][start] == target) {
            return true;
        } else if (matrix[row][end] == target) {
            return true;
        }
        return false;
    }
}
```

二、一次二分法

将矩阵的每一行拼接起来，成为一个长为 `m × n` 的排序数组，然后使用二分查找。查找时将位置转换为矩阵坐标即可，此方法更加简洁。

算法复杂度：

- 时间复杂度： `O(log m*n) = O(log m) + O(log n)` 。
- 空间复杂度： `O(1)` 。

Java 实现：

```
public class Solution {
    /**
     * @param matrix, a list of lists of integers
     * @param target, an integer
     * @return a boolean, indicate whether matrix contains targe
t
     */
    public boolean searchMatrix(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0 ||
            matrix[0] == null || matrix[0].length == 0) {
            return false;
        }

        int row = matrix.length;
        int col = matrix[0].length;
        int start = 0;
        int end = row * col - 1;
        int mid;
        // put all rows of the matrix into an array
        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            int temp = matrix[mid / col][mid % col];
            if (temp == target) {
                return true;
            } else if (temp < target) {
                start = mid;
            } else {
                end = mid;
            }
        }
        if (matrix[start / col][start % col] == target) {
            return true;
        }
        if (matrix[end / col][end % col] == target) {
            return true;
        }
        return false;
    }
}
```

参考

1. Search a 2D Matrix | 九章算法

Search a 2D Matrix II

Search a 2D Matrix II (leetcode lintcode)

```
Description
Write an efficient algorithm that searches for a value in an m x
 n matrix, return the occurrence of it.

This matrix has the following properties:
- Integers in each row are sorted from left to right.
- Integers in each column are sorted from up to bottom.
- No duplicate integers in each row or column.

Example
Consider the following matrix:
[
  [1, 3, 5, 7],
  [2, 4, 7, 8],
  [3, 5, 9, 10]
]
Given target = 3, return 2.
```

解题思路

矩阵每一行是递增的，每一列是递增的，且没有重复元素。如果从左上角开始遍历，向右和向下两个方向都是递增，无法有效进行判断。如果从左下角开始遍历，

- 若当前值大于目标值，那么这一行的所有数都大于目标值，该行可以舍弃；
- 若当前值小于目标值，那么这一列的所有数都小于目标值，该列可以舍弃；
- 若当前值等于目标值，计数加一，当前行和当前列全部舍弃。

算法复杂度：

- 时间复杂度：每次向上或向右移动一次，从左下角遍历至右上角，移动次数约为 `m + n` ，复杂度为 `O(m + n)` 。
- 空间复杂度：常数个辅助空间，复杂度为 `O(1)` 。

易错点：

> 1. 边界判断的返回值，要根据求解问题的具体含义来定。比如本题中要求的是矩阵中目标值出现的次数，在矩阵为空时，显然目标值个数为 0 ，所以返回 0 即可。不要总是死脑筋的返回 -1 。

Java 实现

```
public class Solution {
    /**
     * @param matrix: A list of lists of integers
     * @param: A number you want to search in the matrix
     * @return: An integer indicate the occurrence of target in
the given matrix
     */
    public int searchMatrix(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0 ||
            matrix[0] == null || matrix[0].length == 0) {
            return 0;
        }

        int row = matrix.length;
        int col = matrix[0].length;
        int i = row - 1;    // traverse from the left bottom
        int j = 0;
        int count = 0;

        while (i >= 0 && j < col) {
            if (matrix[i][j] > target) {
                i--;
            } else if (matrix[i][j] < target) {
                j++;
            } else {
                count++;
                i--;
                j++;
            }
        }

        return count;
    }
}
```

参考

1. Search a 2D Matrix II | 九章算法

First Bad Version

First Bad Version (leetcode lintcode)

```
Description
The code base version is an integer start from 1 to n.
One day, someone committed a bad version in the code case,
so it caused this version and the following versions are all fai
led in the unit tests.
Find the first bad version.

You can call isBadVersion to help you determine which version is
 the first bad one.
The details interface can be found in the code's annotation part
.

Notice
Please read the annotation in code area to get the correct way t
o call isBadVersion in different language.
For example, Java is SVNRepo.isBadVersion(v).

Example
Given n = 5:
isBadVersion(3) -> false
isBadVersion(5) -> true
isBadVersion(4) -> true
Here we are 100% sure that the 4th version is the first bad vers
ion.

Challenge
You should call isBadVersion as few as possible.
```

解题思路

一般在排序数组中寻找第一个/最后一个目标值都可以使用二分法查找，本题也是如此。

算法复杂度

- 时间复杂度：`O(log n)` 。
- 空间复杂度：`O(1)` 。

Java 实现

```java
/**
 * public class SVNRepo {
 *     public static boolean isBadVersion(int k);
 * }
 * you can use SVNRepo.isBadVersion(k) to judge whether
 * the kth code version is bad or not.
*/
class Solution {
    /**
     * @param n: An integers.
     * @return: An integer which is the first bad version.
     */
    public int findFirstBadVersion(int n) {
        if (n <= 0) {
            return -1;
        }

        int start = 1;
        int end = n;
        int mid;
        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (SVNRepo.isBadVersion(mid)) {
                end = mid;
            } else {
                start = mid;
            }
        }
        if (SVNRepo.isBadVersion(start)) {
            return start;
        }
        if (SVNRepo.isBadVersion(end)) {
            return end;
        }
        return -1;
    }
}
```

Find Peak Element

Find Peak Element (leetcode lintcode)

```
There is an integer array which has the following features:
- The numbers in adjacent positions are different.
- A[0] < A[1] && A[A.length - 2] > A[A.length - 1].

We define a position P is a peek if:
A[P] > A[P-1] && A[P] > A[P+1]
Find a peak element in this array. Return the index of the peak.

Notice
The array may contains multiple peeks, find any of them.

Example
Given [1, 2, 1, 3, 4, 5, 7, 6]
Return index 1 (which is number 2) or 6 (which is number 7)

Challenge
Time complexity O(logN)
```

解题思路

第一个数和最后一个数都小于其相邻数，所以数组一定存在峰值。考虑使用二分法，取中间值后有以下几种情况：

- 中间值比其右边数小，说明其处在上升沿中，峰值在其右侧，`start = mid`。
- 中间值比其左边数小，说明其处在下降沿中，峰值在其左侧，`end = mid`。
- （中间值比两边都小，那么左侧和右侧都有峰值，舍弃哪边都可以。）
- 中间值比两边的数都大，是峰值，直接返回即可。

算法复杂度

- 时间复杂度： `O(log n)` 。
- 空间复杂度： `O(1)` 。

Java 实现：

```
class Solution {
    /**
     * @param A: An integers array.
     * @return: return any of peek positions.
     */
    public int findPeak(int[] A) {
        // 1.答案在之间，2.不会出界
        int start = 1;
        int end = A.length-2;
        while(start + 1 <  end) {
            int mid = (start + end) / 2;
            if(A[mid] < A[mid - 1]) {
                end = mid;
            } else if(A[mid] < A[mid + 1]) {
                start = mid;
            } else {
                end = mid;
            }
        }
        if(A[start] < A[end]) {
            return end;
        } else {
            return start;
        }
    }
}
```

另一种实现，也通过了测试：

```
class Solution {
    /**
     * @param A: An integers array.
     * @return: return any of peek positions.
     */
    public int findPeak(int[] A) {
        if (A.length < 3) {
            return -1;
        }

        int start = 0;
        int end = A.length - 1;
        int mid;
        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (A[mid] < A[mid + 1]) {
                start = mid;
            } else if (A[mid] < A[mid - 1]) {
                end = mid;
            } else {
                return mid;
            }
        }
        return -1;
    }
}
```

参考

1. Find Peak Element | 九章算法

Median of two Sorted Arrays

Median of two Sorted Arrays (leetcode lintcode)

```
Description
There are two sorted arrays A and B of size m and n respectively
.
Find the median of the two sorted arrays.

Example
Given A=[1,2,3,4,5,6] and B=[2,3,4,5], the median is 3.5.
Given A=[1,2,3] and B=[4,5], the median is 3.

Challenge
The overall run time complexity should be O(log (m+n)).
```

解题思路

中位数的概念（维基百科）：

> 对于一组有限个数的数据来说，它们的中位数（ Median ）是这样的一种数：这群数据里的一半的数据比它大，而另外一半数据比它小。
>
> 计算有限个数的数据的中位数的方法是：把所有的同类数据按照大小的顺序排列。如果数据的个数是奇数，则中间那个数据就是这群数据的中位数；如果数据的个数是偶数，则中间那2个数据的算术平均值就是这群数据的中位数。

一、归并排序

比较直观的方法是将两个排序数组合并为一个排序数组，然后再求中位数。

算法复杂度

- 时间复杂度： `O(m + n)` 。
- 空间复杂度： `O(m + n)` 。

Java 实现

```
class Solution {
    /**
     * @param A: An integer array.
     * @param B: An integer array.
     * @return: a double whose format is *.5 or *.0
     */
    public double findMedianSortedArrays(int[] A, int[] B) {
        if ((A == null || A.length == 0) &&
            (B == null || B.length == 0)) {
            return -1.0;
        }

        int m = (A == null) ? 0 : A.length;
        int n = (B == null) ? 0 : B.length;
        int len = m + n;

        // merge sort
        int i = 0, j = 0, k = 0;    // i for A, j for B, k for C
        int[] C = new int[len];
        // case1: both A and B have elements
        while (i < m && j < n) {
            if (A[i] < B[j]) {
                C[k++] = A[i++];
            } else {
                C[k++] = B[j++];
            }
        }
        // case2: only A has elements
        while (i < m) {
            C[k++] = A[i++];
        }
        // case3: only B has elements
        while (j < n) {
            C[k++] = B[j++];
        }

        // return median for even and odd cases
        if (len % 2 == 0) {
            return (C[(len - 1) / 2] + C[len / 2]) / 2.0;
        } else {
```

```
                return C[len / 2];
            }
        }
    }
```

二、二分查找

本题目可以转化为更一般的情况，寻找两个排序数组中第 `k` 大（从 `1` 开始数）的数， `k` 为两个数组长度和的中位数。如果每次比较都能舍弃 `k/2` 个数，那么时间复杂度就是对数线性级别。

我们来比较数组 `A` 和数组 `B` 的第 `k/2` 个数的大小，分为三种情况：

- `A[k/2 - 1] == B[k/2 - 1]` ，不难得到此时两个数组第 `k` 大的数就是 `A[k/2 - 1]` 。
- `A[k/2 - 1] < B[k/2 - 1]` ，那么在合并数组后 `A[0] … A[k/2 - 1]` 一定小于第 k 大的数，舍弃这一部分数，接着从 `A[k/2] … A[A.length - 1]` 中找。由于舍弃了 `k/2` 个数，所以第 `k` 大的数相应变成第 `k - k/2` 大的数。
- `A[k/2 - 1] > B[k/2 - 1]` ，与上述类似。

几个边界条件的判断：

1. 当其中一个数组长度为 `0` 时，直接从另一个数组中取值即可。
2. 当 `k == 1` 时。

算法复杂度

- 时间复杂度： `k = (m + n) / 2` ，所以复杂度为 `O(log(m + n))` 。
- 空间复杂度： `O(1)` 。

Java 实现

```
class Solution {
    /**
     * @param A: An integer array.
     * @param B: An integer array.
     * @return: a double whose format is *.5 or *.0
     */
    public double findMedianSortedArrays(int[] A, int[] B) {
```

```
        int len = A.length + B.length;
        if (len % 2 == 1) {
            return findKth(A, 0, B, 0, len / 2 + 1);
        }
        return (findKth(A, 0, B, 0, len / 2) + findKth(A, 0, B, 0
, len / 2 + 1)) / 2.0;
    }

    // find kth number of two sorted array
    public static int findKth(int[] A, int A_start,
                              int[] B, int B_start,
                              int k) {
        if (A_start >= A.length) {
            return B[B_start + k - 1];
        }
        if (B_start >= B.length) {
            return A[A_start + k - 1];
        }

        if (k == 1) {
            return Math.min(A[A_start], B[B_start]);
        }

        int A_key = A_start + k / 2 - 1 < A.length
                ? A[A_start + k / 2 - 1]
                : Integer.MAX_VALUE;
        int B_key = B_start + k / 2 - 1 < B.length
                ? B[B_start + k / 2 - 1]
                : Integer.MAX_VALUE;

        if (A_key < B_key) {
            return findKth(A, A_start + k / 2, B, B_start, k - k
 / 2);
        } else {
            return findKth(A, A_start, B, B_start + k / 2, k - k
 / 2);
        }
    }
}
```

参考

1. Median of two Sorted Arrays | 九章算法
2. Median of two Sorted Arrays | 数据结构与算法/leetcode/lintcode题解
3. Median of Two Sorted Array leetcode java | 爱做饭的小莹子

Search in Rotated Sorted Array

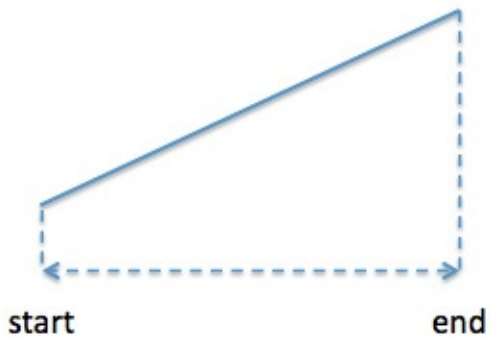
Search in Rotated Sorted Array (leetcode lintcode)

```
Description
Suppose a sorted array is rotated at some pivot unknown to you b
eforehand.
(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.

Example
For [4, 5, 1, 2, 3] and target=1, return 2.
For [4, 5, 1, 2, 3] and target=0, return -1.

Challenge
O(logN) time
```

解题思路

直观的解法是对数组进行遍历，这样需要 `O(n)` 的时间复杂度。如果要在 `O(logN)` 时间内实现需要考虑使用二分查找。

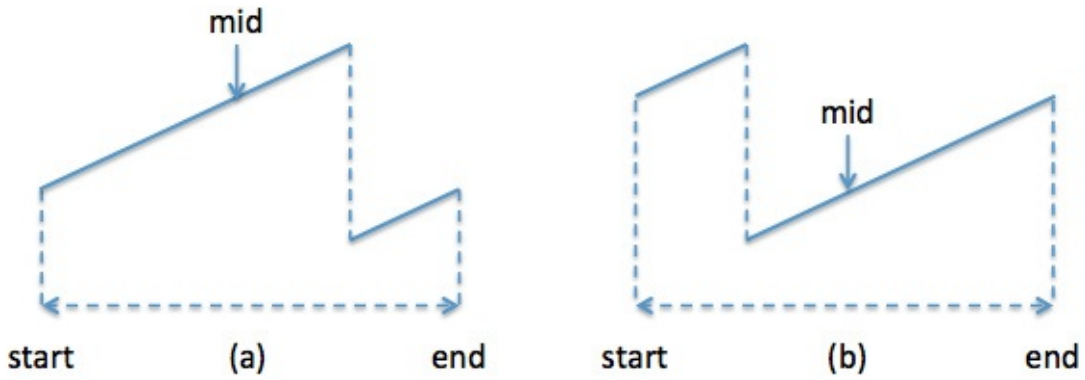
假设原始数组为升序，如图所示。



在旋转之后，可能有以下两种情况：

- 当 `A[mid] > A[start]` 时，对应情况 (a)。

- 当 `A[mid] < A[end]` 时，对应情况 (b) 。



不难看出， `mid` 将旋转序列分为两段，一段是正常的排序数组，另一段是旋转数组，目标值 `target` 可能在任一段。以情况 (a) 为例，当 `A[start] <= target <= A[mid]` 时，当目标值在左侧排序数组上，否则目标值在右侧旋转数组上，以此进行二分查找。

总结一下本题目的思路就是：先判断旋转数组的形状（左边/右边的上升序列更长？），然后判断目标值在哪一侧（正常的排序数组部分上，还是旋转数组部分）。

算法复杂度

- 时间复杂度： `O(log n)` 。
- 空间复杂度： `O(1)` 。

易错点

> 1. 在判断目标值的范围时，考虑到第一个元素 `A[start]` 和最后一个元素 `A[mid]` 可能等于目标值，所以要用大于等于、小于等于。

Java 实现

```
public class Solution {
    /**
     *@param A : an integer rotated sorted array
     *@param target :  an integer to be searched
     *return : an integer
     */
    public int search(int[] A, int target) {
        // write your code here
```

```
        if (A == null || A.length == 0) {
            return -1;
        }

        int start = 0, end = A.length - 1;
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (A[mid] == target) {
                return mid;
            } else if (A[start] < A[mid]) {     // 根据mid所在位置
划分不同情况
                if ((A[start] <= target) && (target <= A[mid]))
{
                    end = mid;
                } else {
                    start = mid;
                }
            } else if (A[mid] < A [end]) {
                if ((A[mid] <= target) && (target <= A[end])) {
                    start = mid;
                } else {
                    end = mid;
                }
            }
        }  // while end
        if (A[start] == target) {
            return start;
        }
        if (A[end] ==  target) {
            return end;
        }
        return -1;
    }
}
```

参考

1. Search in Rotated Sorted Array | 九章算法

Search in Rotated Sorted Array II

Search in Rotated Sorted Array II (leetcode lintcode)

```
Description
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.

Example
Given [1, 1, 0, 1, 1, 1] and target = 0, return true.
Given [1, 1, 1, 1, 1, 1] and target = 0, return false.
```

解题思路

有重复元素，那么考虑一个极端的情况就是数组 `[0, 1, 1, 1, 1, … , 1]` 的旋转数组，该情况无法使用二分法，需要遍历所有元素，时间复杂度为 `O(n)` ，这是最坏情况，可以考虑使用直接遍历。

Java 实现

```
public class Solution {
    /**
     * param A : an integer ratated sorted array and duplicates
are allowed
     * param target :  an integer to be search
     * return : a boolean
     */
    public boolean search(int[] A, int target) {
        // write your code here
        if (A == null || A.length == 0 ) {
            return false;
        }
        for (int i = 0; i < A.length; i++) {
            if (A[i] == target) {
                return true;
            }
        }
        return false;
    }
}
```

当然，其实我们大可不必放弃治疗。考虑到最坏情况不可能一直出现，针对一般情况（只有部分重复元素），我们可以将二分查找和遍历结合起来。

Java 实现

```
public class Solution {
    /**
     * param A : an integer ratated sorted array and duplicates
are allowed
     * param target :  an integer to be search
     * return : a boolean
     */
    public boolean search(int[] A, int target) {
        if (A == null || A.length == 0) {
            return false;
        }

        int start = 0, end = A.length - 1;
```

```
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (A[mid] == target) {
                return true;
            }
            if (A[mid] > A[start]) {
                if (A[start] <= target && target <= A[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else if (A[mid] < A[start]) {
                if (A[mid] <= target && target <= A[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
            } else {
                start++;    // A[start] == A[mid], then skip dup
licate one
            }
        }

        if (A[start] == target) {
            return true;
        } else if (A[end] == target) {
            return true;
        }
        return false;
    }
}
```

参考

1. Search in Rotated Sorted Array II | 九章算法
2. [LeetCode] Search in Rotated Sorted Array II 解题报告 | 水中的鱼

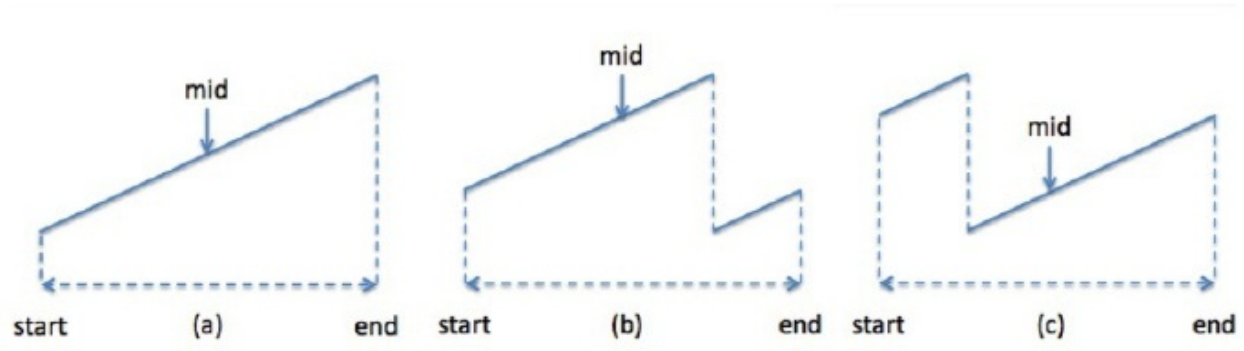
Find Minimum in Rotated Sorted Array

Find Minimum in Rotated Sorted Array (leetcode lintcode)

```
Description
Suppose a sorted array is rotated at some pivot unknown to you b
eforehand.
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
Find the minimum element.

Notice
You may assume no duplicate exists in the array.

Example
Given [4, 5, 6, 7, 0, 1, 2] return 0
```

解题思路

参考题目 Search in Rotated Sorted Array ，在旋转数组中寻找最小值，可能出现以下三种情况：



其中，在情况（a）和情况（c），end 都需要往中间移动，而在情况（b），start 需要往中间移动，由此可以得出我们的解法如下。

算法复杂度

- 时间复杂度： `O(logn)` 。

Java 实现

```
public class Solution {
    /**
     * @param nums: a rotated sorted array
     * @return: the minimum number in the array
     */
    public int findMin(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }

        int start = 0, end = nums.length - 1;
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (nums[mid] < nums[end]) {
                end = mid;
            } else  {
                start = mid;
            }
        }
        if (nums[start] < nums[end]) {
            return nums[start];
        }
        return nums[end];
    }
}
```

Find Minimum in Rotated Sorted Array II

Find Minimum in Rotated Sorted Array II (leetcode lintcode)

```
Description
Suppose a sorted array is rotated at some pivot unknown to you b
eforehand.
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
Find the minimum element.

Notice
The array may contain duplicates.

Example
Given [4,4,5,6,7,0,1,2] return 0.
```

解题思路

本题目要考虑重复元素，那么考虑一个极端的情况原数组 `[0, 1, 1, 1, 1, … , 1]` ，其旋转数组为 `[1, 1, ..., 1, 0, 1, ... , 1, 1]` ，该情况使用二分法无法确保得到正确结果，需要结合遍历来考虑。

Java 实现

```java
public class Solution {
    /**
     * @param num: a rotated sorted array
     * @return: the minimum number in the array
     */
    public int findMin(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }

        int start = 0, end = nums.length - 1;
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (nums[mid] == nums[end]) {
                end--;
            } else if (nums[mid] < nums[end]){
                end = mid;
            } else {
                start = end;
            }
        }
        if (nums[start] <= nums[end]) {
            return nums[start];
        }
        return nums[end];
    }
}
```

参考

1. Find Minimum in Rotated Sorted Array II | 九章算法

Subtree

Subtree (lintcode)

```
Description
You have two every large binary trees: T1, with millions of node
s, and T2, with hundreds of nodes.
Create an algorithm to decide if T2 is a subtree of T1.

Notice
A tree T2 is a subtree of T1 if there exists a node n in T1 such
 that the subtree of n is identical to T2.
That is, if you cut off the tree at node n, the two trees would
be identical.

Example
T2 is a subtree of T1 in the following case:
       1                3
      / \              /
T1 = 2   3      T2 =  4
        /
       4

T2 isn't a subtree of T1 in the following case:
       1                3
      / \                \
T1 = 2   3       T2 =     4
        /
       4
```

解题思路

使用递归的思路，首先判断 `T2` 和 `T1` 是否相等，如果不相等则接着让 `T2` 和 `T1` 左子树、右子树比较。这里面非常重要的一点是判断最基本的情况。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param T1, T2: The roots of binary tree.
     * @return: True if T2 is a subtree of T1, or false.
     */
    public boolean isSubtree(TreeNode T1, TreeNode T2) {
        // write your code here
        if (T2 == null) {
            return true;
        }
        if (T1 == null) {
            return false;
        }

        if (isEqual(T1, T2)) {
            return true;
        }

        // if the tree with root as current node doesn't match t
hen
        // try left and right subtrees one by one
        return isSubtree(T1.left, T2) || isSubtree(T1.right, T2)
;
    }

    private boolean isEqual(TreeNode root1, TreeNode root2) {
        if (root1 == null && root2 == null) {
            return true;
```

```
        }
        if (root1 == null || root2 == null) {
            return false;
        }

        // check if the data of both roots is same and
        // data of left and right subtrees are also same
        return root1.val == root2.val &&
                isEqual(root1.left, root2.left) &&
                isEqual(root1.right, root2.right);
    }

}
```

Binary Tree Preorder Traversal

Binary Tree Preorder Traversal (leetcode lintcode)

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

Example
Given:
    1
   / \
  2   3
 / \
4   5
return [1,2,4,5,3].

Challenge
Can you do it without recursion?
```

解题思路

一、递归法

二叉树本身就是递归定义，所以采用递归的方法实现树的遍历容易理解且代码简介。

实现过程：根据前序遍历访问的顺序，优先访问根结点，然后分别访问左儿子和右儿子。对任一结点，在循环中都可以看作是根结点，可直接访问，访问完之后，若其左儿子非空，按相同规则访问其左儿子，然后访问其右儿子。

```
前序遍历：root -> left -> right
```

根据递归时对返回结果处理方式的不同，可进一步分为遍历和分治两种方法。

面试时不推荐使用递归的方法实现。

算法复杂度

- 时间复杂度：遍历树中的所有结点，时间复杂度 `O(n)` 。
- 空间复杂度：未使用额外空间。

1. 遍历法

实现步骤：

- 访问结点 `node` ，取值，将其视为根结点。
- 若其左儿子非空，按相同规则访问其左儿子。
- 若其右儿子非空，按相同规则访问其右儿子。

其中， `result` 是作为参数传递的。

Java 实现：

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Preorder in ArrayList which contains node values
.
     */
    public ArrayList<Integer> preorderTraversal(TreeNode root) {
        ArrayList<Integer> result = new ArrayList<Integer>();
        // 主循环中无需对边界条件进行检查，递归函数中已完成相关检查
        /*
        if (root == null) {
            return result;
        }
```

```
        */
        traverse(root, result);
        return result;
    }

    private void traverse(TreeNode root, ArrayList<Integer> resu
lt){
        if (root == null) {
            return;
        }
        result.add(root.val);
        traverse(root.left, result);
        traverse(root.right, result);
    }
}
```

2. 分治法

步骤：

- 获得左子树的遍历结果。
- 获得右子树的遍历结果。
- 按照“根结点 + 左子树 + 右子树”的顺序对结果进行合并

其中，`result` 是作为结果返回的。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Preorder in ArrayList which contains node values
.
     */
    public ArrayList<Integer> preorderTraversal(TreeNode root) {
        ArrayList<Integer> result = new ArrayList<Integer>();
        // null or leaf
        if (root == null) {
            return result;
        }

        // divide
        ArrayList<Integer> left = preorderTraversal(root.left);
        ArrayList<Integer> right = preorderTraversal(root.right)
;

        // conquer
        result.add(root.val);
        result.addAll(left);
        result.addAll(right);

        return result;
    }
}
```

二、非递归法

基本原则：用递归可以解决的问题，改用非递归的方法解决，一般都需要使用栈，来模拟递归解法内存中使用的栈操作。

方法一：分层入栈

实现步骤：

- 根结点非空，将根结点 `root` 入栈。
- 栈非空（循环结束的条件）
  - 取栈顶结点 `node` 并进行出栈操作，保存当前结点值。
  - 判断结点 `node` 的右儿子是否为空，若不空，将其入栈。
  - 判断结点 `node` 的左儿子是否为空，若不空，将其入栈。
- 遍历结束，返回结果

算法复杂度

- 时间复杂度：对每个结点遍历一遍，近似为 `O(n)`
- 空间复杂度：使用辅助栈，最坏情况下栈空间与结点数相等，近似为 `O(n)`

易错点

> 1. 注意左子树、右子树的进栈顺序。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Preorder in ArrayList which contains node values
.
```

```
     */
    public ArrayList<Integer> preorderTraversal(TreeNode root) {
        // write your code here
        ArrayList<Integer> result = new ArrayList<Integer>();
        Stack<TreeNode> stack = new Stack<TreeNode>();

        if (root == null) {
            return result;
        }

        stack.push(root);
        while (!stack.empty()) {
            TreeNode node = stack.pop();
            result.add(node.val);

            if (node.right != null) {
                stack.push(node.right);     //注意左子树、右子树的进
栈顺序
            }
            if (node.left != null) {
                stack.push(node.left);
            }
        }
        return result;
    }
}
```

方法二

实现步骤：

- 访问结点 `node` ，并将其入栈。若其左儿子非空，将左儿子置为当前结点，重复上述步骤，直至遇到空结点。
- 取栈顶结点并出栈，将栈顶结点右儿子置为当前结点，重复步骤一。
- 当前结点为空且栈为空，遍历结束。

Java 实现

```java
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Preorder in ArrayList which contains node values
.
     */
    public ArrayList<Integer> preorderTraversal(TreeNode root) {
        ArrayList<Integer> result = new ArrayList<Integer>();
        if (root == null) {
            return result;
        }

        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode node = root;
        while (!stack.empty() || node != null) {
            while (node != null) {
                result.add(node.val);
                stack.push(node);
                node = node.left;
            }
            if (!stack.empty()) {
                node = stack.pop();
                node = node.right;
            }
        }
        return result;
    }
}
```

参考

1. Binary Tree Preorder Traversal | 九章算法
2. Binary Tree Preorder Traversal | 数据结构与算法/leetcode/lintcode题解
3. 二叉树的非递归遍历 | 海子

Binary Tree Postorder Traversal

Binary Tree Postorder Traversal (leetcode lintcode)

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

Example
Given binary tree {1,#,2,3},
   1
    \
     2
    /
   3

return [3,2,1].

Challenge
Can you do it without recursion?
```

解题思路

参考 Binary Tree Preorder Traversal 。

一、递归

1.遍历

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Postorder in ArrayList which contains node value
s.
     */
    public ArrayList<Integer> postorderTraversal(TreeNode root)
{
        // traverse
        ArrayList<Integer> result = new ArrayList<Integer>();
        traverse(root, result);
        return result;
    }

    private void traverse(TreeNode root, ArrayList<Integer> resu
lt) {
        if (root == null) {
            return;
        }

        traverse(root.left, result);
        traverse(root.right, result);

        result.add(root.val);
    }
}
```

2.分治

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Postorder in ArrayList which contains node value
s.
     */
    public ArrayList<Integer> postorderTraversal(TreeNode root)
{
        // divide & conquer
        ArrayList<Integer> result = new ArrayList<Integer>();
        if (root == null) {
            return result;
        }

        ArrayList<Integer> left = postorderTraversal(root.left);
        ArrayList<Integer> right = postorderTraversal(root.right
);

        result.addAll(left);
        result.addAll(right);
        result.add(root.val);

        return result;
    }
}
```

二、迭代

方法一：双栈实现

使用两个辅助栈，一个栈用来添加结点及其左右儿子，另一个栈用来翻转第一个栈的输出。该方法的好处是易于理解，缺点是两个栈使用辅助空间较多。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Postorder in ArrayList which contains node value
s.
     */
    public ArrayList<Integer> postorderTraversal(TreeNode root)
{
        // non-recursion
        ArrayList<Integer> result = new ArrayList<Integer>();
        if (root == null) {
            return result;
        }

        Stack<TreeNode> stack = new Stack<TreeNode>();
        Stack<TreeNode> temp  = new Stack<TreeNode>();

        stack.push(root);
        while (!stack.empty()) {
            TreeNode node = stack.pop();
```

```
            if (node.left != null) {
                stack.push(node.left);
            }
            if (node.right != null) {
                stack.push(node.right);
            }
            temp.push(node);
        }

        while (!temp.empty()) {
            result.add(temp.pop().val);
        }

        return result;
    }

}
```

方法二、单栈实现

后续遍历需要确保根结点出栈是在左儿子和右儿子出栈之后。因此实现步骤如下：

- 对任一非空结点，按照“右儿子-->左儿子”顺序入栈，这样确保左儿子先出栈。
- 对于以下两种情况，出栈并输出至结果。
  - 叶子结点：左儿子、右儿子都为空。
  - 子结点已经出栈的结点，通过 `prev` 指针判断。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
```

```
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Postorder in ArrayList which contains node value
s.
     */
    public ArrayList<Integer> postorderTraversal(TreeNode root)
{
        ArrayList<Integer> result = new ArrayList<Integer>();
        if (root == null) {
            return result;
        }

        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode curr = root;
        TreeNode prev = null;
        stack.push(root);

        while (!stack.empty()) {
            curr = stack.peek();
            // if current node has no child or its two child nod
e have been visited
            if ((curr.left == null && curr.right == null) ||
                (prev != null && (prev == curr.left || prev == c
urr.right))) {
                result.add(curr.val);
                stack.pop();
                prev = curr;
            } else {
                if (curr.right != null) {
                    stack.push(curr.right);
                }
                if (curr.left != null) {
                    stack.push(curr.left);
                }
            }
        }
        return result;
    }
}
```

参考

1. 二叉树的非递归遍历 | 海子
2. Leetcode – Binary Tree Postorder Traversal (Java) | ProgramCreek

Binary Tree Inorder Traversal

Binary Tree Inorder Traversal (leetcode lintcode)

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

Example
Given binary tree {1,#,2,3},
   1
    \
     2
    /
   3
return [1,3,2].

Challenge
Can you do it without recursion?
```

解题思路

一、递归

1.遍历

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Inorder in ArrayList which contains node values.
     */
    public ArrayList<Integer> inorderTraversal(TreeNode root) {
        // traverse
        ArrayList<Integer> result = new ArrayList<Integer>();
        traverse(root, result);
        return result;
    }

    private void traverse (TreeNode root, ArrayList<Integer> res
ult) {
        if (root == null) {
            return;
        }
        traverse(root.left, result);
        result.add(root.val);
        traverse(root.right, result);
    }
}
```

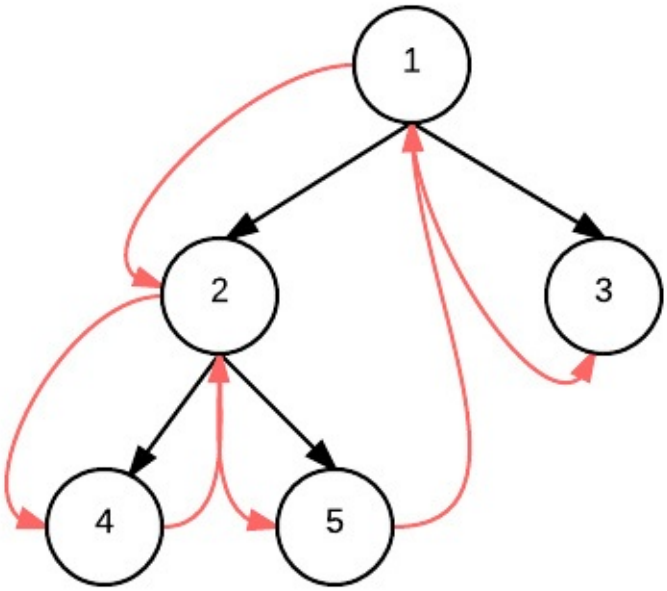
2.分治

```java
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Inorder in ArrayList which contains node values.
     */
    public ArrayList<Integer> inorderTraversal(TreeNode root) {
        // divide & conquer
        ArrayList<Integer> result = new ArrayList<Integer>();
        if (root == null) {
            return result;
        }
        // divide
        ArrayList<Integer> left  = inorderTraversal(root.left);
        ArrayList<Integer> right = inorderTraversal(root.right);

        // conquer
        result.addAll(left);
        result.add(root.val);
        result.addAll(right);

        return result;
    }
}
```

二、迭代

可参考 Binary Tree Preorder Traversal 迭代法方法二的实现，只需要把结点输出至结果的顺序调整一下即可。

使用迭代遍历二叉树一定要使用栈，可以画图理解结点应该进栈顺序、出栈时间，这里借用 ProgramCreek 的图示作为参考。



Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Inorder in ArrayList which contains node values.
     */
    public ArrayList<Integer> inorderTraversal(TreeNode root) {
        // non-recursion
        ArrayList<Integer> result = new ArrayList<Integer>();
        if (root == null) {
            return result;
        }
```

```
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode node = root;

        while (!stack.empty() || node != null) {
            // if it is not null, push to stack and go down the
tree to left
            if (node != null) {
                stack.push(node);
                node = node.left;
            } else {
            // if no left child, pop stack, process the node the
n let node point to the right
                TreeNode tmp = stack.pop();
                result.add(tmp.val);
                node = tmp.right;
            }
        }
        return result;
    }

}
```

参考

1. Leetcode – Binary Tree Inorder Traversal (Java) | ProgramCreek

Maximum Depth of Binary Tree

Maximum Depth of Binary Tree (leetcode lindtcode)

```
Description
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.

Example
Given a binary tree as follow:
  1
 / \
2   3
   / \
  4   5
The maximum depth is 3.
```

解题思路

遍历二叉树最自然的方法是递归，递归调用层数过多可能会导致栈空间溢出，因此，需要适当考虑递归调用的层数。

一、分治法

步骤：

- 求左子树的最大深度。
- 求右子树的最大深度。
- 树的最大深度等于左、右子树的最大深度加 1。

算法复杂度：

- 时间复杂度：遍历树中的所有 `n` 个结点，每个结点所涉及的操作是常数个，故总的时间复杂度为 `O(n)` 。
- 空间复杂度：树的深度最大为 `n` ，最小为 `logn` ，故空间复杂度介于

`O(logn)` 和 `O(n)` 之间。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    public int maxDepth(TreeNode root) {
        // write your code here
        if (root == null) {
            return 0;
        }

        // divide
        int left = maxDepth(root.left);
        int right = maxDepth(root.right);

        // conquer
        int depth = Math.max(left, right) + 1;
        return depth;
    }
}
```

二、遍历法

步骤：

- 访问结点 `node` ，将其视为根结点。
- 结点为空，返回；

  结点非空，比较当前的结点深度 curtdepth 与当前记录的深度 depth，取两者间的最大值。
- 访问左子树，当前结点深度加一。
- 访问右子树，当前结点深度加一。

注： `trick` 在于如何在进入下一层结点时将结点深度加一。

算法复杂度

- 时间复杂度：遍历树中的所有结点，时间复杂度 `O(n)` 。
- 空间复杂度：树的深度最大为 `n` ，最小为 `logn` ，故空间复杂度介于 `O(logn)` 和 `O(n)` 之间。

Java实现：

```java
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    private int depth;

    public int maxDepth(TreeNode root) {
        depth = 0;
        traverse(root, 1);
        return depth;
    }

    private void traverse(TreeNode root, int curtdepth) {
        if (root == null) {
            return;
        }

        if (curtdepth > depth) {
            depth = curtdepth;
        }

        traverse(root.left, curtdepth + 1);
        traverse(root.right, curtdepth + 1);
    }
}
```

参考

1. Maximum Depth of Binary Tree | 九章算法
2. Maximum Depth of Binary Tree | 数据结构与算法/leetcode/lintcode题解

Minimum Depth of Binary Tree

Minimum Depth of Binary Tree (leetcode lintcode)

```
Description
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.

Example
Given a binary tree as follow:
  1
 / \
2   3
   / \
  4   5
The minimum depth is 2.
```

解题思路

一、递归

求根结点到叶子结点的最短路径，当前结点只有一个子结点时，不能返回 0，要接着遍历该子树，直至遇到叶子结点。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    public int minDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        if (root.left == null && root.right == null) {
            return 1;
        }

        if (root.left == null) {
            return minDepth(root.right) + 1;
        }
        if (root.right == null) {
            return minDepth(root.left) + 1;
        }

        return Math.min(minDepth(root.left), minDepth(root.right
)) + 1;
    }
}
```

另一种更简洁的写法

```java
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    public int minDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int left  = minDepth(root.left);
        int right = minDepth(root.right);

        if(root.left == null || root.right == null) {
            return left >= right ? left + 1 : right + 1;
        }

        return Math.min(left, right) + 1;
    }
}
```

参考

1. Find Minimum Depth of a Binary Tree | geeksforgeeks
2. Minimum Depth of Binary Tree leetcode java | 爱做饭的小莹子

Balanced Binary Tree

Balanced Binary Tree (leetcode lintcode)

```
Description
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.

Example
Given binary tree A = {3,9,20,#,#,15,7}, B = {3,#,20,15,7}
A)  3            B)    3
   / \                  \
  9  20                 20
    /  \                / \
   15   7              15  7
The binary tree A is a height-balanced binary tree, but B is not
.
```

解题思路

判断一个二叉树是否平衡，需要判断两个方面：

1. 左子树、右子树是否平衡。
2. 当前结点是否平衡。

所以需要两个信息，一个是树的最大深度，另一个是树是否平衡。

一、ResultType

新建一个类 `ResultType` 保存 `isBalanced` 和 `maxDepth` 信息。

算法复杂度

- 时间复杂度：每个结点遍历一遍，所需的时间是常数，所以总的时间复杂度为

`O(n)` 。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
class ResultType{
    public boolean isBalanced;
    public int maxDepth;
    public ResultType(boolean isBalanced, int maxDepth) {
        this.isBalanced = isBalanced;
        this.maxDepth = maxDepth;
    }
}

public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: True if this Binary tree is Balanced, or false.
     */
    public boolean isBalanced(TreeNode root) {
        return traverse(root).isBalanced;
    }

    private ResultType traverse(TreeNode root) {
        if (root == null) {
            return new ResultType(true, 0);
        }

        ResultType left = traverse(root.left);
        ResultType right = traverse(root.right);
```

```
        // subtree not balance
        if (!left.isBalanced || !right.isBalanced) {
            return new ResultType(false, -1);
        }

        // root not balance
        if (Math.abs(left.maxDepth - right.maxDepth) > 1) {
            return new ResultType(false, -1);
        }

        return new ResultType(true, Math.max(left.maxDepth, righ
t.maxDepth) + 1);
    }

}
```

二、非ResultType

方法一中递归函数需要返回两个信息，考虑把两个信息合并到一个参数中。如果当前子树不平衡，将树的深度 `depth` 置为 `-1` ，每次都对其进行判断即可。该方法很是巧妙。

复杂度分析

- 时间复杂度：遍历树中的所有结点，时间复杂度 `O(n)` 。
- 空间复杂度：使用了部分辅助变量，空间复杂度为 `o(1)` 。

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
```

```
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: True if this Binary tree is Balanced, or false.
     */
    public boolean isBalanced(TreeNode root) {
        int depth;
        depth = maxDepth(root);
        if (depth == -1) {
            return false;
        } else {
            return true;
        }
        // 以上内容可以用一句代替
        // return maxDepth(root) != -1;
    }

    private int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int left = maxDepth(root.left);
        int right = maxDepth(root.right);

        if ((left == -1) || (right == -1) || Math.abs(left - rig
ht) > 1) {
            return -1;
        }

        return Math.max(left, right) + 1;
    }

}
```

参考

1. Balanced Binary Tree | 九章算法
2. Balanced Binary Tree | 数据结构与算法/leetcode/lintcode题解

Binary Tree Maximum Path Sum

Binary Tree Maximum Path Sum (leetcode lintcode)

```
Description
Given a binary tree, find the maximum path sum.
The path may start and end at any node in the tree.

Example
Given the below binary tree:
  1
 / \
2   3
return 6.
```

解题思路

一、分治法

首先将题目简化，如果是求从根结点出发的最大路径长度，那么直接使用递归法即可：

- 求左子树的最大路径长度。
- 求左子树的最大路径长度。
- 取以上两者的最大值加上当前根结点数值，即为当前根结点的最大路径长度。

现在求解任意结点出发的最大路径，与上述情况相比，多了一种情况：跨越根结点的路径，这里的根结点不一定是整棵树的根结点，也可能只存在于左子树、或右子树。除了比较左、右子树的最大路径长度，还需要比较增加了根结点之后的总路径长度。

所以递归函数需要保存两个信息：子树的最大路径长度、子树中跨越“根结点”路径的最大长度

算法复杂度

- 时间复杂度：每个结点遍历一遍，每个结点的操作是常数个，所以时间复杂度

是 `O(n)` 。

- 空间复杂度：使用常数个辅助变量保存参数，空间复杂度为 `O(1)` 。

注意：

> 1. 题目中未明确说明结点值的正负，所以要考虑结点值为负的情况，一个典型的输入是只含一个结点{-1}。那么 `singlePath` 的值可能为负，这种情况我们可以直接舍弃该子树，认为 `singlePath` 为 `0` 。
> 2. 关于空结点 singlePath 和 maxPath 的初始化取值，可以从实际意义上理解。目前感觉还讲不清楚，可参考方法一和方法二具体实现。

Java 实现：

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */

public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    private class ResultType {
        // singlePath：从root往下走到任意点的最大路径值，这条路径可以不包含任何点
        // maxPath：从树中任意点到任意点的最大路径，这条路径至少包含一个点

        int singlePath, maxPath;
        ResultType (int singlePath, int maxPath) {
            this.singlePath = singlePath;
            this.maxPath = maxPath;
        }
```

```
    }

    public int maxPathSum(TreeNode root) {
        ResultType result = helper(root);
        return result.maxPath;
    }

    private ResultType helper(TreeNode root) {
        if (root == null) {
            // maxPath 取 MIN_VALUE 说明无满足条件路径
            return new ResultType(0, Integer.MIN_VALUE);
        }
        // Divide
        ResultType left = helper(root.left);
        ResultType right = helper(root.right);

        // Conquer
        int singlePath = Math.max(left.singlePath, right.singleP
ath) + root.val;
        singlePath = Math.max(singlePath, 0);    // 舍弃路径值为负
数的部分

        int maxPath = Math.max(left.maxPath, right.maxPath);
        // 至少含有当前根节点的值
        maxPath = Math.max(maxPath, left.singlePath + right.sing
lePath + root.val);

        return new ResultType(singlePath, maxPath);
    }

}
```

二、分治法 II

和解法一类似，具体操作上有些区别。暂时还不是很清楚这么做有什么好处。

Java 实现

```
// Version 2:
```

```
// SinglePath也定义为，至少包含一个点。
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: An integer.
     */
    private class ResultType {
        int singlePath, maxPath;
        ResultType(int singlePath, int maxPath) {
            this.singlePath = singlePath;
            this.maxPath = maxPath;
        }
    }

    public int maxPathSum(TreeNode root) {
        ResultType result = helper(root);
        return result.maxPath;
    }

    private ResultType helper(TreeNode root) {
        if (root == null) {
            return new ResultType(Integer.MIN_VALUE, Integer.MIN
_VALUE);
        }
        // Divide
        ResultType left = helper(root.left);
        ResultType right = helper(root.right);

        // Conquer
        // 与解法一的区别主要在这里
        // singlePath 至少含一个结点就意味着这个结点的值可能为负值
        int singlePath =
            Math.max(0, Math.max(left.singlePath, right.singlePa
th)) + root.val;

        int maxPath = Math.max(left.maxPath, right.maxPath);
        // 由于singlePath可能为负值，所以maxPath需要检查并舍弃这种情况
        maxPath = Math.max(maxPath,
                           Math.max(left.singlePath, 0) +
                           Math.max(right.singlePath, 0) + root.
```

```
val);

        return new ResultType(singlePath, maxPath);
    }
}
```

参考

1. Binary Tree Maximum Path Sum | 九章算法

Lowest Common Ancestor

Lowest Common Ancestor (leetcode lindcode)

```
Description
Given the root and two nodes in a Binary Tree.
Find the lowest common ancestor(LCA) of the two nodes.
The lowest common ancestor is the node with largest depth which
is the ancestor of both nodes.

Notice
Assume two nodes are exist in tree.

Example
For the following binary tree:
  4
 / \
3   7
   / \
  5   6
LCA(3, 5) = 4
LCA(5, 6) = 7
LCA(6, 7) = 7
```

解题思路

一、分治法

思路：使用深度优先搜索，从叶子结点开始，标记子树中出现目标结点的情况。如果子树中有目标结点，那么标记该结点，否则标记为 `null` 。如果左子树、右子树都有标记，说明已经找到最小公共祖先了。如果在根结点为 `A` 的左右子树中寻找 `A` 和 `B` 的公共祖先，就是结点 `A` 本身。

换个角度：如果结点 `C` 的左子树有一个目标结点，右子树没有，则该结点非最小公共祖先。如果结点 `C` 的右子树有一个目标结点，左子树没有，则该节点亦非最小公共祖先。只有当节点 `C` 左子树有一个目标结点，右子树也有一个的时候，才

是最小公共祖先。

在二叉树中寻找结点 `A` 和 `B` 的 `LCA` ：

- 如果左右子树都有返回值，说明当前结点就是 `LCA` ，直接返回。
- 如果只找到 `A` ，返回 `A` 。
- 如果只找到 `B` ，返回 `B` 。
- 如果都没有找到，就返回 `null` 。

算法复杂度

- 时间复杂度：每个结点遍历一遍，每个结点的操作是常数个，所以时间复杂度是 `O(n)` 。
- 空间复杂度：使用了常数个辅助变量保存参数，空间复杂度为 `O(1)` 。

Follow up question

- 如果是二叉搜索树，要怎么做？
- 如果每个结点有父亲指针，要怎么做？

Java 实现：

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of the binary search tree.
     * @param A and B: two nodes in a Binary.
     * @return: Return the least common ancestor(LCA) of the two
 nodes.
     */
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode
```

```
 A, TreeNode B) {
        if (root == null || root == A || root == B) {
            return root;
        }

        // Divide
        TreeNode left = lowestCommonAncestor(root.left, A, B);
        TreeNode right = lowestCommonAncestor(root.right, A, B);

        // Conquer
        // 只有在找到A或B的情况下返回值才不为null，所以以此为条件判断是否
找到LCA
        if (left != null && right != null) {
            return root;
        }
        if (left != null) {
            return left;
        }
        if (right != null) {
            return right;
        }
        return null;
    }
}
```

参考

1. Lowest Common Ancestor | 九章算法
2. [Leetcode] Lowest Common Ancestor of a Binary Tree 最小公共祖先 | segmentfault
3. Lowest Common Ancestor of Two Nodes in a Binary Tree | 爱做饭的小莹子

Binary Tree Level Order Traversal

Binary Tree Level Order Traversal (leetcode lintcode)

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

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]
]

Challenge
Challenge 1: Using only 1 queue to implement it.
Challenge 2: Use DFS algorithm to do it.
```

解题思路

一、单队列实现（**BFS**）

队列中记录每一层的结点，获取当前队列大小，进行分层处理。

- 获取队列中当前层的结点数。
- 对于当前队列中该层的所有结点。
    - 取出当前队列中的头结点，取其值。

    - 如果有左儿子，加入队列。
    - 如果有右儿子，加入队列。

算法复杂度

- 时间复杂度：每个结点遍历一遍，每个结点的操作是常数个，所以时间复杂度是 `O(n)` 。
- 空间复杂度：使用了常数个辅助变量保存参数，空间复杂度为 `O(1)` 。

易错点

> 1. 在初始化 `Queue` 类型时，要用 `LinkedList` ，这是因为在 `Java` 里 `Queue` 是一个接口，不能直接实例化，需要通过已实现 `Queue` 接口的类（如 `Linkedlist` ）来构造。

Java 实现：

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */

public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Level order a list of lists of integer
     */
    public ArrayList<ArrayList<Integer>> levelOrder(TreeNode roo
t) {
        ArrayList result = new ArrayList();
        if (root == null) {
            return result;
        }
```

```
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);

        while (!queue.isEmpty()) {
            ArrayList<Integer> level = new ArrayList<Integer>();
            // 获取当前层的结点个数
            int size = queue.size();
            // 依次取出当前层的结点，并将其左儿子、右儿子放进队列
            // 下一层结点依次加入队列
            for (int i = 0; i < size; i++) {
                TreeNode head = queue.poll();
                level.add(head.val);

                if (head.left != null) {
                    queue.offer(head.left);
                }
                if (head.right != null) {
                    queue.offer(head.right);
                }
            } // end for
            // 将当前层结点对应的一些列值放入result
            result.add(level);
        } // end while

        return result;
    }
}
```

二、DFS

DFS 每次都是在纵向增加层级，需要做的是每次在遍历某一层的时候将该层的结点全部加入到数组中。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
```

```
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */


public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: Level order a list of lists of integer
     */
    public ArrayList<ArrayList<Integer>> levelOrder(TreeNode roo
t) {
        // write your code here
        ArrayList<ArrayList<Integer>> results = new ArrayList<Ar
rayList<Integer>>();

        if (root == null) {
            return results;
        }

        int maxLevel = 0;
        while (true) {
            ArrayList<Integer> level = new ArrayList<Integer>();
            // 将 maxLevel 层的结点放入 level
            dfs(root, level, 0, maxLevel);
            // 在 maxLevel 层找不到结点时结束循环
            if (level.size() == 0) {
                break;
            }
            results.add(level);
            maxLevel++;
        }
        return results;
    }
```

```
    private void dfs (TreeNode root,
                      ArrayList<Integer> level,
                      int curtLevel,
                      int maxLevel) {
        if (root == null || curtLevel > maxLevel) {
            return;
        }
        // 在遍历到 maxLevel 层时，将结点加入队列
        if (curtLevel == maxLevel) {
            level.add(root.val);
            return;
        }

        dfs(root.left, level, curtLevel + 1, maxLevel);
        dfs(root.right, level, curtLevel + 1, maxLevel);
    }
}
```

三、双队列（**BFS**）

四、单队列 + 哑结点（**BFS**）

参考

1. Binary Tree Level Order Traversal | 九章算法
2. How do I instantiate a Queue object in java? | stackoverflow

Binary Tree Level Order Traversal II

Binary Tree Level Order Traversal II (leetcode lintcode)

```
Description
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).

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]
]
```

解题思路

一、BFS

参考 Binary Tree Level Order Traversal ，最后结果反转一下即可。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
```

```
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */


public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: buttom-up level order a list of lists of integer
     */
    public ArrayList<ArrayList<Integer>> levelOrderBottom(TreeNo
de root) {
        // write your code here
        ArrayList<ArrayList<Integer>> result = new ArrayList<Arr
ayList<Integer>>();

        if (root == null) {
            return result;
        }

        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);

        while (!queue.isEmpty()) {
            int size = queue.size();
            ArrayList<Integer> level = new ArrayList<Integer>();

            for (int i = 0; i < size; i++) {
                TreeNode head = queue.poll();
                level.add(head.val);

                if (head.left != null) {
                    queue.offer(head.left);
                }
                if (head.right != null) {
                    queue.offer(head.right);
                }
```

```
                }
                result.add(level);
            }

            Collections.reverse(result);
            return result;
        }

    }
```

参考

1. Binary Tree Level Order Traversal II | 九章算法

Binary Tree Zigzag Level Order Traversal

Binary Tree Zigzag Level Order Traversal (leetcode lintcode)

```
Description
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 a
nd alternate between).

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]
]
```

解题思路

一、BFS（单队列）

和题目 Binary Tree Level Order Traversal 相比，不同在于相邻层结点的排列顺序相反，那么只需要设置一个标志，每到下一层时反转结点顺序即可。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
```

```
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */


public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: A list of lists of integer include
     *          the zigzag level order traversal of its nodes' v
alues
     */
    public ArrayList<ArrayList<Integer>> zigzagLevelOrder(TreeNo
de root) {
        // write your code here
        ArrayList<ArrayList<Integer>> result = new ArrayList<Arr
ayList<Integer>>();
        if (root == null) {
            return result;
        }

        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        // node_level用来记录当前处理的是哪一层根结点
        int node_level = 0;
        while (!queue.isEmpty()) {
            ArrayList<Integer> level = new ArrayList<Integer>();
            int size = queue.size();

            for (int i = 0; i < size; i++) {
                TreeNode head = queue.poll();
                level.add(head.val);

                if (head.left != null) {
                    queue.offer(head.left);
```

```
                }
                if (head.right != null) {
                    queue.offer(head.right);
                }
            }
            // 结点在偶数层，按默认顺序记录；结点在奇数层，顺序反转后记录
            if (node_level % 2 == 0) {
                result.add(level);
            } else {
                Collections.reverse(level);
                result.add(level);
            }
            node_level++;
        }
        return result;
    }
}
```

二、BFS（两个栈）

也可以使用两个栈来实现，这里面的入栈顺序有一些小 trick。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */


public class Solution {
    /**
     * @param root: The root of binary tree.
```

```
     * @return: A list of lists of integer include
     *          the zigzag level order traversal of its nodes' v
alues
     */
    public ArrayList<ArrayList<Integer>> zigzagLevelOrder(TreeNo
de root) {
        // write your code here
        ArrayList<ArrayList<Integer>> result = new ArrayList<Arr
ayList<Integer>>();
        if (root == null) {
            return result;
        }
        // currLevel表示当前在处理的那一层的结点
        // nextLevel表示下一层要处理的结点
        Stack<TreeNode> currLevel = new Stack<TreeNode>();
        Stack<TreeNode> nextLevel = new Stack<TreeNode>();
        Stack<TreeNode> tmp;

        currLevel.push(root);
        boolean normalOrder = true;

        while (!currLevel.isEmpty()) {
            ArrayList<Integer> currLevelResult = new ArrayList<I
nteger>();

            while (!currLevel.isEmpty()) {
                TreeNode node = currLevel.pop();
                currLevelResult.add(node.val);

                if (normalOrder) {
                    if (node.left != null) {
                        nextLevel.push(node.left);
                    }
                    if (node.right != null) {
                        nextLevel.push(node.right);
                    }
                } else {
                    if (node.right != null) {
                        nextLevel.push(node.right);
                    }
```

```
                    if (node.left != null) {
                        nextLevel.push(node.left);
                    }
                }
            }
            result.add(currLevelResult);
            // 交换currLevel和nextLevel，currLevel代表将要处理的那一
层结点
            tmp = currLevel;
            currLevel = nextLevel;
            nextLevel = tmp;

            normalOrder = !normalOrder;
        }

    }
}
```

参考

1. Binary Tree Zigzag Level Order Traversal | 九章算法

Construct Binary Tree from Preorder and Inorder Traversal

Construct Binary Tree from Preorder and Inorder Traversal (lintcode)

```
Description
Given preorder and inorder traversal of a tree, construct the bi
nary tree.

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

Example
Given in-order [1,2,3] and pre-order [2,1,3], return a tree:
  2
 / \
1   3
```

解题思路

根据前序遍历确认当前根结点，然后在中序遍历中分别寻找左子树和右子树，递归求解即可。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
```

```
public class Solution {
    /**
     *@param preorder : A list of integers that preorder traversal of a tree
     *@param inorder : A list of integers that inorder traversal of a tree
     *@return : Root of a tree
     */
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        // write your code here
        if (preorder == null || inorder == null ||
            preorder.length != inorder.length) {
            return null;
        }

        return helper(inorder, 0, inorder.length - 1, preorder, 0, preorder.length - 1);
    }

    private TreeNode helper(int[] inorder, int instart, int inend,
                            int[] prorder, int prstart, int prend) {
        if (instart > inend) {
            return null;
        }

        TreeNode root = new TreeNode(prorder[prstart]);
        int pos = findRootPos(inorder, instart, inend, prorder[prstart]);

        root.left = helper(inorder, instart, pos - 1,
                           prorder, prstart + 1, prstart + pos - instart);
        root.right = helper(inorder, pos + 1, inend,
                           prorder, prend - inend + pos + 1, prend);
        return root;
```

```
    }

    private int findRootPos(int[] A, int start, int end, int tar
get) {
        for (int i = start; i <= end; i++) {
            if (A[i] == target) {
                return i;
            }
        }
        return -1;
    }
}
```

Validate Binary Search Tree

Validate Binary Search Tree (leetcode lintcode)

```
Descriptiion
Given a binary tree, determine if it is a valid binary search tr
ee (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 grea
ter than the node's key.
- Both the left and right subtrees must also be binary search tr
ees.
- A single node tree is a BST

Example
An example:
  2
 / \
1   4
   / \
  3   5
The above binary tree is serialized as {2,1,4,#,#,3,5} (in level
 order).
```

解题思路

一、分治法

在判断时需要三个信息：子树是否平衡，子树的最大值，子树的最小值。需要新建一个类型存储这三种信息。

易错点

> 1. 在判断当前根结点和右子树的最小值时，考虑到空结点的最小值初始化为 Integer.MAX_VALUE，为了防止结点的值等于 Integer.MAX_VALUE，要增加右子树是否为空的判断。做了两遍都犯了这个错误。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of binary tree.
     * @return: True if the binary tree is BST, or false
     */
    private class ResultType {
        boolean isBST;
        int maxValue, minValue;
        ResultType (boolean isBST, int maxValue, int minValue) {
            this.isBST = isBST;
            this.maxValue = maxValue;
            this.minValue = minValue;
        }
    }

    public boolean isValidBST(TreeNode root) {

        ResultType result = helper(root);
        return result.isBST;
    }

    private ResultType helper (TreeNode root) {
        if (root == null) {
```

```
            return new ResultType(true, Integer.MIN_VALUE, Integ
er.MAX_VALUE);
        }

        ResultType left = helper(root.left);
        ResultType right = helper(root.right);

        if (!left.isBST || !right.isBST ) {
            return new ResultType(false, 0, 0);
        }
        if (root.left != null && left.maxValue >= root.val ||
            root.right != null && right.minValue <= root.val) {
            return new ResultType(false, 0, 0);
        }

        return new ResultType(true,
                              Math.max(root.val, right.maxValue)
,
                              Math.min(root.val, left.minValue))
;

    }
}
```

参考

1. Validate Binary Search Tree | 九章算法

Insert Node in a Binary Search Tree

Insert Node in a Binary Search Tree (leetcode lintcode)

```
Description
Given a binary search tree and a new tree node, insert the node
into the tree.
You should keep the tree still be a valid binary search tree.

Notice
You can assume there is no duplicate values in this tree + node.

Example
Given binary search tree as follow, after Insert node 6, the tre
e should be:
  2             2
 / \           / \
1   4   -->   1   4
   /             / \
  3             3   6

Challenge
Can you do it without recursion?
```

解题思路

一、递归法

注意：

> 1. 在遍历二叉树时，有返回值的递归函数的使用方法。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of the binary search tree.
     * @param node: insert this node into the binary search tree
     * @return: The root of the new binary search tree.
     */
    public TreeNode insertNode(TreeNode root, TreeNode node) {
        if (root == null) {
            return node;
        }

        if (node.val < root.val) {
            root.left = insertNode(root.left, node);
        } else if (node.val > root.val) {
            root.right = insertNode(root.right, node);
        }
        return root;
    }
}
```

二、非递归法

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
```

```
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of the binary search tree.
     * @param node: insert this node into the binary search tree
     * @return: The root of the new binary search tree.
     */
    public TreeNode insertNode(TreeNode root, TreeNode node) {
        // write your code here
        if (root == null) {
            root = node;
            return root;
        }

        TreeNode tmp = root;
        TreeNode last = null;
        while (tmp != null) {
            last = tmp;
            if (tmp.val > node.val) {
                tmp = tmp.left;
            } else {
                tmp = tmp.right;
            }
        }
        if (last != null) {
            if (last.val > node.val) {
                last.left = node;
            } else {
                last.right = node;
            }
        }
        return root;
    }
}
```

参考

1. Insert Node in a Binary Search Tree | 九章算法

Search Range in Binary Search Tree

Search Range in Binary Search Tree (leetcode lintcode)

```
Description
Given two values k1 and k2 (where k1 < k2) and a root pointer to
 a Binary Search Tree.
Find all the keys of tree in range k1 to k2.
i.e. print all x such that k1<=x<=k2 and x is a key of given BST
.
Return all the keys in ascending order.

Example
If k1 = 10 and k2 = 22, then your function should return [12, 20
, 22].
    20
   /  \
  8   22
 / \
4   12
```

解题思路

一、DFS

二叉搜索树的中序遍历是升序，根据题目要求，范围内的取值按升序排列，可以对 BST 进行中序 DFS ，并对遍历到的结点值进行判断，满足大小范围加入数组即可。

Java实现 v1 ：将 `ArrayList<Integer> results` 作为一个参数传递

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
```

```
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of the binary search tree.
     * @param k1 and k2: range k1 to k2.
     * @return: Return all keys that k1<=key<=k2 in ascending or
der.
     */
    public ArrayList<Integer> searchRange(TreeNode root, int k1,
int k2) {
        // write your code here
        ArrayList<Integer> results = new ArrayList<Integer>();

        dfs(root, results, k1, k2);
        return results;
    }

    private void dfs(TreeNode root,
                     ArrayList<Integer> results,
                     int k1,
                     int k2) {
        if (root == null) {
            return;
        }

        dfs(root.left, results, k1, k2);
        if (root.val >= k1 && root.val <= k2) {
            results.add(root.val);
        }
        dfs(root.right, results, k1, k2);
    }
}
```

Java实现 v2 ：将 `ArrayList<Integer> results` 定义为一个全局变量

```
public class Solution {

    private ArrayList<Integer> results = new ArrayList<Integer>(
);

    public ArrayList<Integer> searchRange(TreeNode root, int k1,
int k2) {
        dfs(root, k1, k2);
        return results;
    }

    private void dfs(TreeNode root, int k1, int k2) {
        if (root == null) {
            return;
        }

        dfs(root.left, k1, k2);
        if (root.val >= k1 && root.val <= k2) {
            results.add(root.val);
        }
        dfs(root.right, k1, k2);
    }
}
```

Java实现 v3 ：在 v2 版本的实现基础上增加条件判断语句，只有在当前结点取值在要求范围内时，才继续进行遍历，减少不必要的遍历操作

```
public class Solution {

    private ArrayList<Integer> results;

    public ArrayList<Integer> searchRange(TreeNode root, int k1,
int k2) {
        results = new ArrayList<Integer>();
        dfs(root, k1, k2);
        return results;
    }

    private void dfs(TreeNode root, int k1, int k2) {
        if (root == null) {
            return;
        }

        if (root.val > k1) {
            dfs(root.left, k1, k2);
        }

        if (root.val >= k1 && root.val <= k2) {
            results.add(root.val);
        }

        if (root.val < k2) {
            dfs(root.right, k1, k2);
        }
    }
}
```

二、迭代法

参考二叉树中序遍历的迭代实现。注意只能对根结点大于 `k2` 的部分才能剪枝。

Java 实现

```
/**
 * Definition of TreeNode:
```

```
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of the binary search tree.
     * @param k1 and k2: range k1 to k2.
     * @return: Return all keys that k1<=key<=k2 in ascending or
der.
     */
    public ArrayList<Integer> searchRange(TreeNode root, int k1,
int k2) {
        ArrayList<Integer> result = new ArrayList<Integer>();
        if (root == null) {
            return result;
        }
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode node = root;

        while (!stack.isEmpty() || node != null) {
            if (node != null) {
                stack.push(node);
                node = node.left;
            } else {
                node = stack.pop();
                if (node.val >= k1 && node.val <= k2) {
                    result.add(node.val);
                } else if (node.val > k2) {
                    break;
                }
                node = node.right;
            }
        }
        return result;
```

```
        }
    }
```

参考

1. Search Range in Binary Search Tree | 九章算法

Remove Node in Binary Search Tree

Remove Node in Binary Search Tree (leetcode lintcode)

```
Description
Given a root of Binary Search Tree with unique value for each no
de.
Remove the node with given value.
If there is no such a node with given value in the binary search
 tree, do nothing.
You should keep the tree still a binary search tree after remova
l.

Example
Given binary search tree:
    5
   / \
  3   6
 / \
2   4

Remove 3, you can either return:
    5
   / \
  2   6
   \
    4
or
    5
   / \
  4   6
 /
2
```

解题思路

删除一个节点涉及的操作比较复杂，所以需要分析都有哪些情况，进行归类，在不同的情况使用不同的方法。

大的步骤分为2步：

- 找到需要删除的结点。
- 如果结点存在，删除该结点。

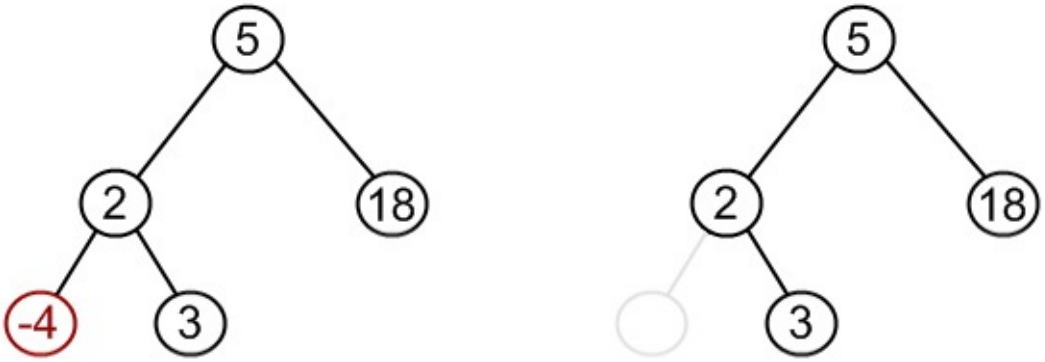
关于查找算法，第一反应就是递归算法，不过涉及删除操作，所以需要记录当前结点的父结点。

关于要删除的结点，有以下几种情况：

1. 叶子结点：

   没有左儿子和右儿子，特殊情况是只有一个根结点。

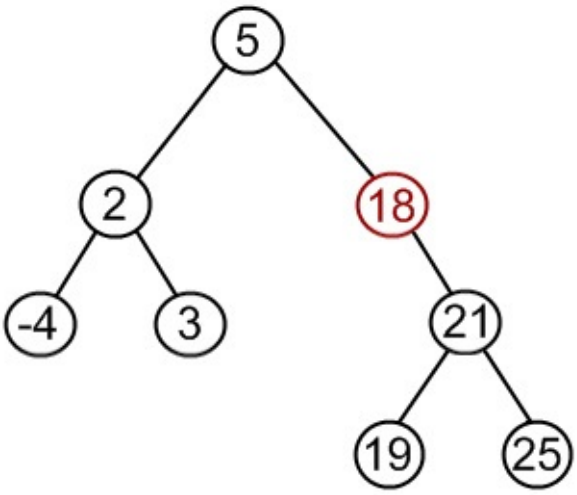
   此时比较简单，直接删除即可。

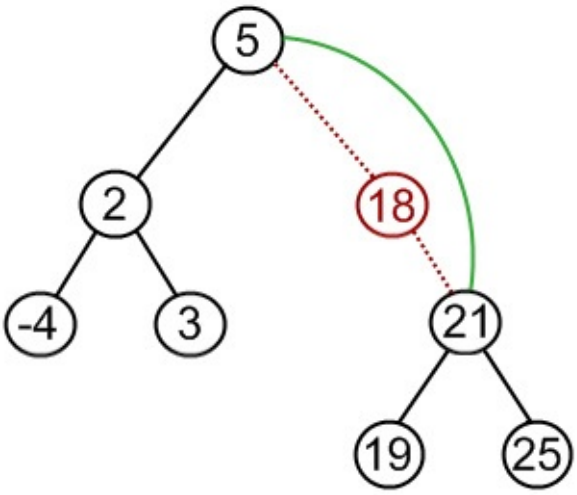


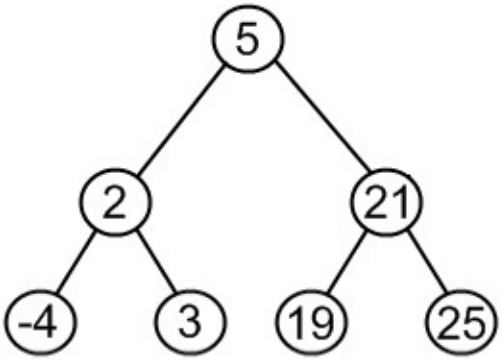
2. 只有一个儿子结点：

   只有左结点或右结点。

   此时也比较简单，将该结点的父结点的儿子指向其儿子结点即可。

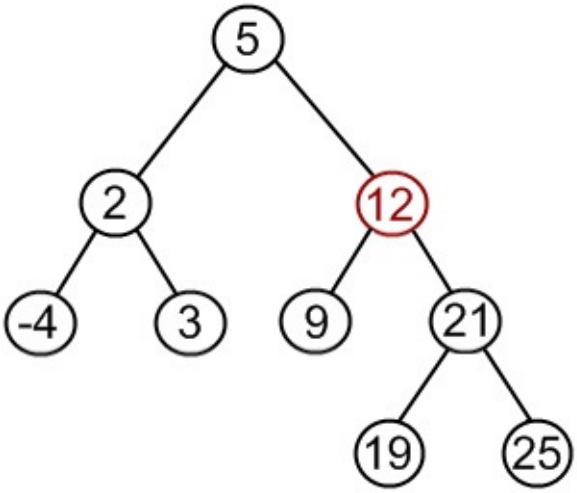


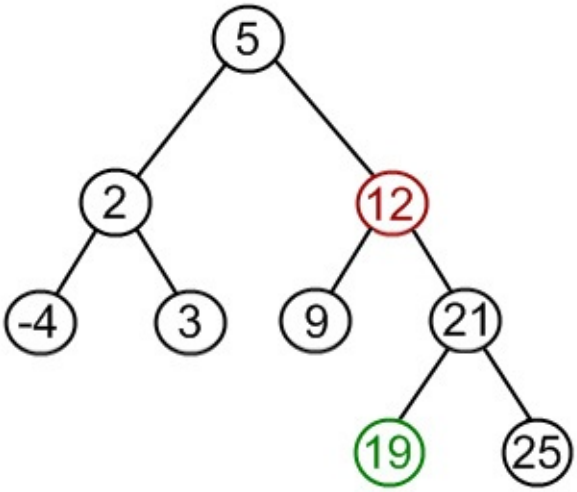


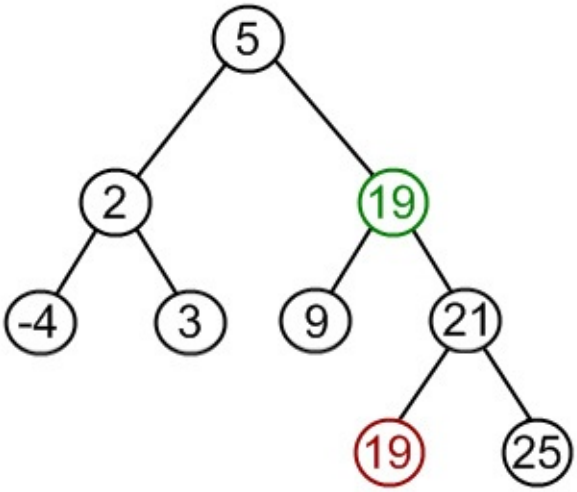


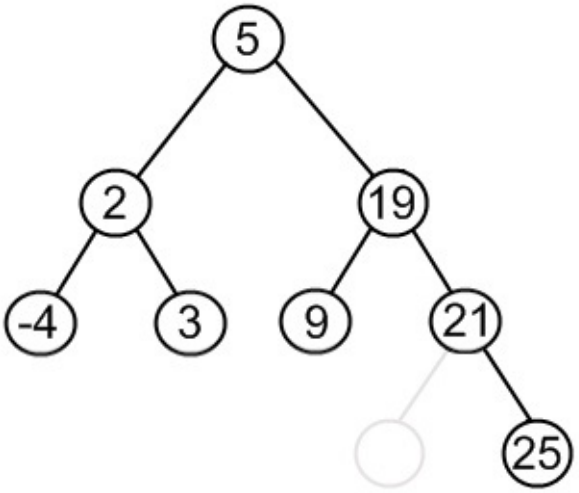
3. 有左儿子和右儿子：

   此时处理稍微复杂一些，要求在删除结点之后仍为二叉查找树，那就意味着树的中序遍历保持升序。为了尽可能减少对树的结点的操作，最简单的方法是找一个结点代替要删除结点，也就是中序遍历中要删除结点的下一个结点——其右子树的最小值结点。根据二叉查找树的性质，不难想到要删除结点右子树的最小值结点一定是叶子结点，这样处理起来就容易了。


5
2
12
-4
3
9
21
19
25


5
2
12
-4
3
9
21
19
25


5
2
19
-4
3
9
21
19
25



Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root: The root of the binary search tree.
     * @param value: Remove the node with given value.
     * @return: The root of the binary search tree after removal
.
     */
    public TreeNode removeNode(TreeNode root, int value) {
        if (root == null) {
            return root;
        } else {
            // 要删除的是根结点，需要使用哑结点辅助
            if (root.val == value) {
                TreeNode dummy = new TreeNode(0);
```

```
                dummy.left = root;
                remove(root, value, dummy);
                root = dummy.left;
                return root;
            } else {
                remove(root, value, null);
                return root;
            }
        }
    }

    public void remove (TreeNode root, int value, TreeNode paren
t) {

        if (value < root.val) {         // value小于当前结点值，向
左子树寻找
            if (root.left != null) {
                remove(root.left, value, root);
            } else {
                return;     // 树中不包含value值，直接返回
            }
        } else if (value > root.val) {  // value大于当前结点值，向
右子树寻找
            if (root.right != null) {
                remove(root.right, value, root);
            } else {
                return;
            }
        } else {
            // 要删除结点左右子树都存在，将结点值置为右子树最小值，并删除
右子树的最小值结点
            if (root.left != null && root.right != null) {
                root.val = minValue(root.right);
                remove(root.right, root.val, root);
            } else if (parent.left == root) {  // 要删除结点只有一
个子树，需要对父结点进行操作
                parent.left = (root.left != null) ? root.left :
root.right;
            } else if (parent.right == root) {
                parent.right = (root.left != null) ? root.left :
```

```
  root.right;
                }
                return;
            }
        }

        public int minValue(TreeNode root){
            if (root.left == null) {
                return root.val;
            } else {
                return minValue(root.left);
            }
        }
    }
```

参考

1. Binary search tree. Removing a node | Algorithms and Data Structures

Binary Search Tree Iterator

Binary Search Tree Iterator (leetcode lintcode)

```
Description
Design an iterator over a binary search tree with the following
rules:
- Elements are visited in ascending order (i.e. an in-order trav
ersal)
- next() and hasNext() queries run in O(1) time in average.

Example
For the following binary search tree, in-order traversal by usin
g iterator is [1, 6, 10, 11, 12]
   10
 /    \
1      11
 \       \
  6       12

Challenge
Extra memory usage O(h), h is the height of the tree.
Super Star: Extra memory usage O(1)
```

解题思路

其实相当于使用栈来实现树的中序遍历

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
```

```
 *         this.left = this.right = null;
 *     }
 * }
 * Example of iterate a tree:
 * BSTIterator iterator = new BSTIterator(root);
 * while (iterator.hasNext()) {
 *    TreeNode node = iterator.next();
 *    do something for node
 * }
 */

public class BSTIterator {
    private Stack<TreeNode> stack = new Stack<>();
    private TreeNode curt;

    //@param root: The root of binary tree.
    public BSTIterator(TreeNode root) {
        // write your code here
        curt = root;
    }

    //@return: True if there has next node, or false
    public boolean hasNext() {
        // write your code here
        return (curt != null || !stack.isEmpty());
    }

    //@return: return next node
    public TreeNode next() {
        // write your code here
        while (curt != null) {
            stack.push(curt);
            curt = curt.left;
        }

        curt = stack.pop();
        TreeNode node = curt;
        curt = curt.right;

        return node;
```

```
    }
}
```

参考

1. Binary Search Tree Iterator | 九章算法

Remove Linked List Elements

Remove Linked List Elements (leetcode lintcode)

```
Description
Remove all elements from a linked list of integers that have val
ue val.

Example
Given 1->2->3->3->4->5->3, val = 3, you should return the list a
s 1->2->4->5.
```

解题思路

考虑到链表头结点可能被删除，所以需要使用哨兵结点 `dummy node` 。

Java 实现

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    /**
     * @param head a ListNode
     * @param val an integer
     * @return a ListNode
     */
    public ListNode removeElements(ListNode head, int val) {
        if (head == null) {
            return null;
        }

        ListNode dummy = new ListNode(0);
        dummy.next = head;
        head = dummy;

        while (head.next != null) {
            if (head.next.val == val) {
                head.next = head.next.next;
            } else {
                head = head.next;
            }
        }

        return dummy.next;
    }
}
```

Delete Node in the Middle of Singly Linked List

Delete Node in the Middle of Singly Linked List (leetcode lintcode)

```
Description
Implement an algorithm to delete a node in the middle of a singl
y linked list,
given only access to that node.

Example
Given 1->2->3->4, and node 3. return 1->2->4
```

解题思路

本题目的意思是，如果要删除一个链表中的结点，并且你只能从这个结点开始访问链表，无法访问它的前一个结点，应该怎么做。在该结点后续结点存在的情况下，将后续结点值复制过来即可。

Java 实现

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param node: the node in the list should be deleted
     * @return: nothing
     */
    public void deleteNode(ListNode node) {
        if (node == null || node.next == null) {
            return;
        }

        ListNode next = node.next;
        node.val = next.val;
        node.next = node.next.next;
        return;
    }
}
```

Remove Duplicates from Sorted List

Remove Duplicates from Sorted List (leetcode lintcode)

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

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

解题思路

- 比较当前结点值与下一个结点的值，当下一个结点为 `null` 时（尾结点）停止循环
  - 如果相等，将当前结点指向其下下一个结点。
  - 如不等，将当前结点指向下一个结点。

注意：以上两个判别条件构成一个全集，所以要使用 `if … else …` 缺少 `else` 会出现不必要的错误。

Java 实现

```java
/**
 * Definition for ListNode
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param ListNode head is the head of the linked list
     * @return: ListNode head of linked list
     */
    public static ListNode deleteDuplicates(ListNode head) {
        if (head == null) {
            return null;
        }
        ListNode node = head;
        while (node.next != null ) {
            if (node.val == node.next.val) {
                node.next = node.next.next;
            } else {
                node = node.next;
            }
        }
        return head;
    }
}
```

Remove Duplicates from Sorted List II

Remove Duplicates from Sorted List II (leetcode lintcode)

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

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

解题思路

题目要求删除所有重复的结点，那么头结点有可能被删除，所以需要使用哨兵结点 `dummy node` 。

首先判断相邻结点是否相等，如相等，保存结点值，将所有等于该值的结点全部删除。

易错点

> 1. 凡是涉及链表的遍历、删除，都需要确认要访问的结点是否为空，控制流语句（ `while, if` ）的判断条件中尤其要注意。本题两重循环的终止条件都需要注意。

Java 实现

```
/**
 * Definition for ListNode
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
```

```
 *     }
 * }
 */
public class Solution {
    /**
     * @param ListNode head is the head of the linked list
     * @return: ListNode head of the linked list
     */
    public static ListNode deleteDuplicates(ListNode head) {
        // write your code here
        if (head == null || head.next == null) {
            return head;
        }

        ListNode dummy = new ListNode(0);
        dummy.next = head;
        head = dummy;

        while (head.next != null && head.next.next != null) {
            if (head.next.val == head.next.next.val) {
                int val = head.next.val;
                while (head.next != null && head.next.val == val
) {
                    head.next = head.next.next;
                }
            } else {
                head = head.next;
            }
        }

        return dummy.next;
    }
}
```

Merge Two Sorted Lists

Merge Two Sorted Lists (leetcode lintcode)

```
Description
Merge two sorted (ascending) linked lists and return it as a new
 sorted list.
The new sorted list should be made by splicing together the node
s of the two lists and sorted in ascending order.

Example
Given 1->3->8->11->15->null, 2->null , return 1->2->3->8->11->15
->null.
```

解题思路

跟归并排序中的合并类似，要考虑一个链表比另一个链表长的情况。

Java 实现

```java
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param ListNode l1 is the head of the linked list
     * @param ListNode l2 is the head of the linked list
     * @return: ListNode head of linked list
     */
```

```
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null && l2 == null) {
            return null;
        }
        ListNode dummy = new ListNode(0);
        ListNode head = dummy;

        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                head.next = l1;
                l1 = l1.next;
            } else {
                head.next = l2;
                l2 = l2.next;
            }
            head = head.next;
        }

        if (l1 != null) {
            head.next = l1;
        }
        if (l2 != null) {
            head.next = l2;
        }

        return dummy.next;
    }
}
```

Add Two Numbers

Add Two Numbers (leetcode lintcode)

```
Description
You have two numbers represented by a linked list, where each no
de contains a single digit.
The digits are stored in reverse order, such that the 1's digit
is at the head of the list.
Write a function that adds the two numbers and returns the sum a
s a linked list.

Example
Given 7->1->6 + 5->9->2. That is, 617 + 295.
Return 2->1->9. That is 912.
Given 3->1->5 and 5->9->2, return 8->0->8.
```

解题思路

依次将两个链表对应结点值相加，并新建结点即可。有两个需要注意的地方，一是两个链表长度可能不一样，二是加法要考虑进位。

易错点

1. 进位的实现，以及求和、求模的差异。
2. 当一个链表 `l1` 比较长时，多出的部分要把结点复制一遍加到新建链表中，不能直接将结点指向 `l1` 。

Java 实现

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
```

```
 *          next = null;
 *      }
 * }
 */
public class Solution {
    /**
     * @param l1: the first list
     * @param l2: the second list
     * @return: the sum list of l1 and l2
     */
    public ListNode addLists(ListNode l1, ListNode l2) {
        if (l1 == null && l2 == null) {
            return null;
        }

        ListNode dummy = new ListNode(0);
        ListNode head = dummy;

        int sum = 0, carry = 0;
        while (l1 != null && l2 != null) {
            sum = l1.val + l2.val + carry;
            head.next = new ListNode(sum % 10);
            carry = sum / 10;

            head = head.next;
            l1 = l1.next;
            l2 = l2.next;
        }

        while (l1 != null) {
            sum = l1.val + carry;
            head.next = new ListNode(sum % 10);
            carry = sum / 10;
            head = head.next;
            l1 = l1.next;
        }

        while (l2 != null) {
            sum = l2.val + carry;
            head.next = new ListNode(sum % 10);
```

```
            carry = sum / 10;
            head = head.next;
            l2 = l2.next;
        }

        if (carry != 0) {
            head.next = new ListNode(carry);
        }

        return dummy.next;
    }
}
```

Reverse Linked List

Reverse Linked List (leetcode lintcode)

```
Description
Reverse a linked list.

Example
For linked list 1->2->3, the reversed linked list is 3->2->1

Challenge
Reverse it in-place and in one-pass
```

解题思路

这里要注意一点，就是尾结点的 `next` 是 `null`，利用这一点可以简化操作。

Java 实现

```java
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param head: The head of linked list.
     * @return: The new head of reversed linked list.
     */
    public ListNode reverse(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }

        ListNode prev = null;
        while (head != null) {
            ListNode temp = head.next;
            head.next = prev;
            prev = head;
            head = temp;
        }

        return prev;
    }
}
```

Reverse Linked List II

Reverse Linked List II (leetcode lintcode)

```
Description
Reverse a linked list from position m to n.

Notice
Given m, n satisfy the following condition: 1 ≤ m ≤ n ≤ length o
f list.

Example
Given 1->2->3->4->5->NULL, m = 2 and n = 4, return 1->4->3->2->5
->NULL.

Challenge
Reverse it in-place and in one-pass
```

解题思路

对第 `m` 到 `n` 个结点进行反转，会影响到第 `m - 1` 个和第 `n + 1` 个结点。

- 首先找到第 `m - 1` 个结点。
- 反转第 `m` 到 `n` 个结点。
- 处理第 `m - 1` 个和第 `n + 1` 个结点。

建议在做题之前画图表示反转之前和之后的结点关系，减少出错，参考博客喜刷刷的习题解析，举例如下：

```
1.找到第 m-1 个结点。
D-->1-->2-->3-->4-->5-->null
    ^
    |
   head

2.以第 m 个结点为头结点，将长度为 L=n-m 部分反转。
D-->1-->2<--3<--4   5-->null
    ^           ^   ^
    |           |   |
   head       prev curt

3.处理未第 m-1 和第 n+1 个结点。
        |-----------|
        |           v
D-->1   2<--3<--4   5-->null
    |           ^
    |-----------|
```

Java 实现

```java
/**
 * Definition for ListNode
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param ListNode head is the head of the linked list
     * @oaram m and n
     * @return: The head of the reversed ListNode
     */
    public ListNode reverseBetween(ListNode head, int m , int n)
```

```
{
        if (m >= n || head == null) {
            return head;
        }

        ListNode dummy = new ListNode(0);
        dummy.next = head;
        head = dummy;

        // move head to (m-1) node
        for (int i = 1; i < m; i++) {
            head = head.next;
        }

        // reverse list from prev with length n-m
        ListNode prev = head.next;
        ListNode curt = prev.next;
        for (int i = 1; i <= n - m; i++) {
            ListNode tmp = curt.next;
            curt.next = prev;
            prev = curt;
            curt = tmp;
        }

        head.next.next = curt;
        head.next = prev;
        return dummy.next;
    }
}
```

参考

1. [LeetCode] Reverse Linked List II | 喜刷刷

Partition List

Partition List (leetcode lintcode)

```
Description
Given a linked list and a value x,
partition it such that all nodes less than x come before nodes g
reater than or equal to x.
You should preserve the original relative order of the nodes in
each of the two partitions.

Example
Given 1->4->3->2->5->2->null and x = 3,
return 1->2->2->4->3->5->null.
```

解题思路

- 新建两个临时链表，表头使用 `dummy node` （初始化为 `new ListNode(0)` ）。
  - 小于 `x` 的结点，放在 `left` 链表。
  - 大于 `x` 的结点，放在 `right` 链表。
- 合并两个临时链表，将 `right` 链表挂在 `left` 链表尾部，并将 **`right`** 链表尾结点指向 **`null`** 。

易错点：

> 1. `ListNode left = leftDummy = new ListNode(0);` 这种方式是不对的，原因暂时不清楚，应该是和Java语言特性有关。

Java 实现

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
```

```
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param head: The first node of linked list.
     * @param x: an integer
     * @return: a ListNode
     */
    public ListNode partition(ListNode head, int x) {
        if (head == null) {
            return null;
        }
        // 注意dummy node的初始化
        // ListNode left = leftDummy = new ListNode(0); 这种初始化
方式是错误的
        ListNode leftDummy = new ListNode(0);
        ListNode left = leftDummy;
        ListNode rightDummy = new ListNode(0);
        ListNode right = rightDummy;

        while (head != null) {
            if (head.val < x) {
                left.next = head;
                left = left.next;
            } else {
                right.next = head;
                right = right.next;
            }
            head = head.next;
        }

        left.next = rightDummy.next;
        right.next = null;
        return leftDummy.next;
    }
}
```

参考

1. Sort List | 九章算法
2. Partition List -- leetcod | 帮客之家

Sort List

Sort List (leetcode lintcode)

```
Description
Sort a linked list in O(n log n) time using constant space compl
exity.

Example
Given 1->3->2->null, sort it to 1->2->3->null.

Challenge
Solve it by merge sort & quick sort separately.
```

解题思路

一、归并排序

在 `findMiddle` 函数中，让 `fast` 先走一步，是为了取得中间节点的前一个。这样做的目的主要是解决 `1->2->null` 两个结点的情况，如果不这样做，`slow` 就会返回2，就没有办法切割了。

复杂度分析

- 时间复杂度：归并排序的时间复杂度是 `O(nlogn)` 。
- 空间复杂度：由于使用了栈空间，所以空间复杂度是 `O(logn)` 。

Java 实现

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
```

```
 *     }
 * }
 */
public class Solution {
    /**
     * @param head: The head of linked list.
     * @return: You should return the head of the sorted linked
list,
                    using constant space complexity.
     */
    public ListNode sortList(ListNode head) {
        // 至少要有2个结点
        if (head == null || head.next == null) {
            return head;
        }
        // 取得中间结点
        ListNode mid = findMiddle(head);
        // 分割为左右2个链表
        ListNode right = mid.next;
        mid.next = null;
        // 分别对左右2个链表进行排序
        ListNode left = sortList(head);
        right = sortList(right);
        // 合并2个链表
        return merge(left, right);
    }

    private ListNode findMiddle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head.next;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }

    private ListNode merge(ListNode head1, ListNode head2) {
        ListNode dummy = new ListNode(0);
        ListNode tail = dummy;
```

```
        while(head1 != null && head2 != null) {
            if(head1.val < head2.val) {
                tail.next = head1;
                head1 = head1.next;
            } else {
                tail.next = head2;
                head2 = head2.next;
            }
            tail = tail.next;
        }
        if(head1 != null) {
            tail.next = head1;
        } else {
            tail.next = head2;
        }

        return dummy.next;
    }
}
```

二、快速排序

Java 实现

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */

class Pair {
    public ListNode first, second;
    public Pair (ListNode first, ListNode second) {
```

```
        this.first = first;
        this.second = second;
    }
}

public class Solution {
    /**
     * @param head: The head of linked list.
     * @return: You should return the head of the sorted linked
list,
                    using constant space complexity.
     */
    public ListNode sortList(ListNode head) {
        // 至少要有2个结点
        if (head == null || head.next == null) {
            return head;
        }

        ListNode mid = findMedian(head);  // O(n)
        Pair pair = partition(head, mid.val);  // O(n)

        ListNode left = sortList(pair.first);
        ListNode right = sortList(pair.second);

        getTail(left).next = right;  // O(n)

        return left;
    }

    // 1->2->3 return 2
    // 1->2 return 1
    private ListNode findMedian(ListNode head) {
        ListNode slow = head;
        ListNode fast = head.next;

        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
```

```
    }

    private Pair partition(ListNode head, int value) {
        ListNode leftDummy = new ListNode(0), leftTail = leftDum
my;
        ListNode rightDummy = new ListNode(0), rightTail = right
Dummy;
        ListNode equalDummy = new ListNode(0), equalTail = equal
Dummy;

        while(head != null) {
            if(head.val < value) {
                leftTail.next = head;
                leftTail = head;  // 相当于 leftTail = leftTail.n
ext;
            } else if(head.val > value) {
                rightTail.next = head;
                rightTail = head;
            } else {
                equalTail.next = head;
                equalTail = head;
            }
            head = head.next;
        }

        leftTail.next = null;
        rightTail.next = null;
        equalTail.next = null;

        if(leftDummy.next == null && rightDummy.next == null) {
            ListNode mid = findMedian(equalDummy.next);
            leftDummy.next = equalDummy.next;
            rightDummy.next = mid.next;
            mid.next = null;
        } else if(leftDummy.next == null) {
            leftTail.next = equalDummy.next;
        } else {
            rightTail.next = equalDummy.next;  // 包含 left right
 都不为 null 的情况
        }
```

```
        return new Pair(leftDummy.next, rightDummy.next);
    }

    private ListNode getTail(ListNode head) {
        if(head == null) {
            return null;
        }

        while(head.next != null) {
            head = head.next;
        }
        return head;
    }
}
```

参考

1. Sort List | 九章算法
2. LeetCode: Sort List 解题报告 | Yu's garden

Reorder List

Reorder List (leetcode lintcode)

```
Description
Given a singly linked list L: L0 → L1 → … → Ln-1 → Ln
reorder it to: L0 → Ln → L1 → Ln-1 → L2 → Ln-2 → …

Example
Given 1->2->3->4->null, reorder it to 1->4->2->3->null.

Challenge
Can you do this in-place without altering the nodes' values?
```

解题思路

可以分为以下几个步骤实现。

- 将链表分为相等的两部分，如果结点为奇数个，左链表个数比右链表个数多1。
- 反转右半部分链表。
- 合并两个链表。

Java 实现

```java
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */
public class Solution {
```

```
    /**
     * @param head: The head of linked list.
     * @return: void
     */
    public void reorderList(ListNode head) {
        // write your code here
        if (head == null || head.next == null || head.next.next
== null) {
            return;
        }

        ListNode mid = findMedian(head);

        ListNode left = head;
        ListNode right = mid.next;
        mid.next = null;

        right = reverse(right);
        head = merge(left, right);
    }

    private ListNode findMedian(ListNode head) {
        ListNode slow = head;
        ListNode fast = head.next;
        while(fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }

    private ListNode reverse(ListNode head) {
        ListNode prev = null;

        while(head!= null) {
            ListNode temp = head.next;
            head.next = prev;
            prev = head;
            head = temp;
        }
```

```
        return prev;
    }

    private ListNode merge(ListNode left, ListNode right) {
        ListNode dummy = new ListNode(0);
        ListNode head = dummy;
        int i = 0;
        while(left != null && right != null) {
            if(i % 2 == 0) {
                head.next = left;
                left = left.next;
            } else {
                head.next = right;
                right = right.next;
            }
            i++;
            head = head.next;
        }

        if(left != null) {
            head.next = left;
        }
        if(right != null) {
            head.next = right;
        }
        return dummy.next;
    }
}
```

Linked List Cycle

Linked List Cycle (leetcode lintcode)

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

Example
Given -21->10->4->5, tail connects to node index 1, return true

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

解题思路

快慢指针

- 快指针每次走两步，慢指针每次走一步。
- 如果两个指针相遇，说明有循环。
- 如果快指针遇到空结点，说明走到链表末尾，说明无循环。

Java 实现

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param head: The first node of linked list.
     * @return: True if it has a cycle, or false
     */
    public boolean hasCycle(ListNode head) {
        // write your code here
        if(head == null || head.next == null) {
            return false;
        }

        ListNode fast, slow;
        fast = head.next;
        slow = head;
        while(fast != slow) {
            // 之所以需要同时判断fast本身及fast.next，是因为fast每次移动两步
            if(fast == null || fast.next == null) {
                return false;
            }
            fast = fast.next.next;
            slow = slow.next;
        }
        return true;
    }
}
```

参考

1. Linked List Cycle | 九章算法

Linked List Cycle II

Linked List Cycle II (leetcode lintcode)

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

Example
Given -21->10->4->5, tail connects to node index 1，return 10

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

解题思路

快慢指针

- 判断是否链表有循环。
  - 如果快指针遇到空结点，说明到达链表末尾，没有循环。
  - 如果快指针遇到慢指针，说明存在循环。
- 在快慢指针相遇后，同时移动头指针和慢指针，当头指针遇到慢指针的下一个结点时，为循环开始处（此处省去证明）。

Java 实现

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
```

```
 * }
 */
public class Solution {
    /**
     * @param head: The first node of linked list.
     * @return: The node where the cycle begins.
     *           if there is no cycle, return null
     */
    public ListNode detectCycle(ListNode head) {
        // write your code here
        if(head == null || head.next == null) {
            return null;
        }

        ListNode fast = head.next;
        ListNode slow = head;
        while(fast != slow) {
            if(fast == null || fast.next == null) {
                return null;
            }
            fast = fast.next.next;
            slow = slow.next;
        }

        while(head != slow.next) {
            head = head.next;
            slow = slow.next;
        }

        return head;
    }
}
```

参考

1. Linked List Cycle II | 九章算法

Intersection of Two Linked Lists

Intersection of Two Linked Lists (leetcode lintcode)

```
Description
Write a program to find the node at which the intersection of tw
o singly linked lists begins.

Notice
If the two linked lists have no intersection at all, return null
.
The linked lists must retain their original structure after the
function returns.
You may assume there are no cycles anywhere in the entire linked
 structure.

Example
The following two linked lists:
A:          a1 → a2
                   ↘
                     c1 → c2 → c3
                   ↗
B:     b1 → b2 → b3
begin to intersect at node c1.

Challenge
Your code should preferably run in O(n) time and use only O(1) m
emory.
```

解题思路

根据题目中给出的例子，两个链表相交是指从某一结点开始到尾结点，两个链表都指向相同结点。

思路一：比较直观的解法是分别计算两个链表的长度，然后将较长的链表向前移动 L 个结点，L 是两个链表的长度差。然后依次比较两个链表的结点是否相等。

思路二：如果两个链表相交，那么把一个链表接在另一个链表后面，会形成一个环，这样我们就可以参考 Linked List Cycle II 的解法来找相交点。这里我们实现思路二。

Java 实现

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param headA: the first list
     * @param headB: the second list
     * @return: a ListNode
     */
    public ListNode getIntersectionNode(ListNode headA, ListNode
 headB) {
        if (headA == null || headB == null) {
            return null;
        }

        ListNode tail = getTail(headA);
        tail.next = headB;

        ListNode result = listCycle(headA);
        tail.next = null;  // disconnect headA and headB
        return result;

    }

    private ListNode getTail (ListNode head) {
        while (head.next != null) {
```

```
            head = head.next;
        }
        return head;
    }

    private ListNode listCycle(ListNode head) {
        ListNode slow = head, fast = head.next;
        while (fast != slow) {
            if (fast == null || fast.next == null) {
                return null;
            }
            fast = fast.next.next;
            slow = slow.next;
        }

        while (head != slow.next) {
            head = head.next;
            slow = slow.next;
        }
        return head;
    }
}
```

参考

1. Intersection of Two Linked Lists | 九章算法

Merge k Sorted Lists

Merge k Sorted Lists (leetcode lintcode)

```
Description
Merge k sorted linked lists and return it as one sorted list.
Analyze and describe its complexity.

Example
Given lists:
[
  2->4->null,
  null,
  -1->null
],
return -1->2->4->null.
```

解题思路

一、分治法

参考数组归并排序的思路，先分再合，先局部有序，再整体有序。

算法复杂度：

- 时间复杂度：每个链表参与合并的次数为 `logn` ， `n` 个链表合并共需要遍历 `(L1+L2+…+Ln)` 次，所以时间复杂度为 `O(logn*(L1+L2+…+Ln))` 。
- 空间复杂度： `O(1)` 。

Java 实现

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
```

```
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
    public ListNode mergeKLists(List<ListNode> lists) {
        // write your code here
        if(list == null || lists.size() == 0) {
            return null;
        }

        return merge(lists, 0, lists.size() - 1);
    }

    private ListNode merge(List<ListNode> lists, int start, int
end) {
        if(start == end) {
            return lists.get(start);
        }

        int mid = start + (end - start) / 2;
        ListNode left = merge(lists, start, mid);
        ListNode right = merge(lists, mid + 1, end);
        return mergeTwoLists(left, right);
    }

    private ListNode mergeTwoLists(ListNode list1, ListNode list
2) {
        ListNode dummy = new ListNode(0);
        ListNode tail = dummy;
        while(list1 != null && list2 != null) {
            if(list1.val < list2.val) {
                tail.next = list1;
                list1 = list1.next;
            } else {
```

```
                tail.next = list2;
                list2 = list2.next;
            }
            tail = tail.next;
        }

        if (list1 != null) {
            tail.next = list1;
        } else {
            tail.next = list2;
        }

        return dummy.next;
    }
}
```

二、优先队列

使用 Java 优先队列（priority queue）库函数 API 。

- 新建优先队列，将 `n` 个待排序链表的头结点分别装入。
  - 取出队列首结点（最小值），加入链表。
  - 如果取出结点有后续结点，将其装入队列。
- 重复以上两个过程直至队列为空。

Java 实现

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */
public class Solution {
```

```
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
    public ListNode mergeKLists(List<ListNode> lists) {
        // write your code here
        if(lists == null || lists.size() == 0) {
            return null;
        }

        Queue<ListNode> heap = new PriorityQueue<ListNode>(lists
.size(), ListNodeComparator);
        // 共n个链表，heap的大小为n，把n个链表的头结点装入
        for(int i = 0; i < lists.size(); i++) {
            if(lists.get(i) != null) {
                heap.add(lists.get(i));
            }
        }

        ListNode dummy = new ListNode(0);
        ListNode tail = dummy;
        while(!heap.isEmpty()) {
            ListNode head = heap.poll();
            tail.next = head;
            tail = tail.next;
            // 如果从队列取出结点有后续结点，将其加入队列
            if(head.next != null) {
                heap.add(head.next);
            }
        }
        return dummy.next;
    }

    private Comparator<ListNode> ListNodeComparator = new Compar
ator<ListNode>() {
        public int compare(ListNode left, ListNode right) {
            if(left == null) {
                return 1;
            } else if(right == null) {
                return -1;
```

```
                }
                return left.val - right.val;
            }
        };  // 这里为何需要引号？

    }
```

参考

1. Merge k Sorted Lists | 九章算法

Copy List with Random Pointer

Copy List with Random Pointer (leetcode lintcode)

```
Description
A linked list is given such that each node contains an additiona
l random pointer
which could point to any node in the list or null.
Return a deep copy of the list.

Example

Challenge
Could you solve it with O(1) space?
```

解题思路

一、哈希表

哈希表需要使用额外的 `O(n)` 的空间。

将原链表的结点作为键（ `Key` ），将复制链表的结点作为值（ `Value` ），依次向哈希表中存放，在此过程中完成链表的深度复制。

实现逻辑：

- 复制 `next` 指针关系。
    - 原链表的当前结点不在哈希表中。
        - 新建结点，拷贝结点值。
        - 将原始链表结点（ `Key` ）和新建结点（ `Value` ）存放在哈希表中。
    - 原链表当前结点在哈希表中（之前已经作为 `random` 指针指向结点被存储）。
        - 指向该结点（ `Key` ）对应的（ `Value` ）结点。
    - 将新建结点放入新建链表中。

- 复制 `random` 指针关系。
    - 原链表结点的 `random` 指针不为空。
        - `random` 所指结点在哈希表中。
            - 将对应复制结点的 `random` 指针指向哈希表中原始结点 `random` 指向结点的复制结点。
        - `random` 所指结点不在哈希表中。
            - 新建 `random` 指向结点，将原始结点、复制结点 `random` 指向结点对放入哈希表。
- 准备处理下一个结点。

算法复杂度

- 时间复杂度：遍历原链表一次，故为 `O(n)` 。
- 空间复杂度：建立一个哈希表做结点映射，为 `O(n)` 。

Java 实现

```
/**
 * Definition for singly-linked list with a random pointer.
 * class RandomListNode {
 *     int label;
 *     RandomListNode next, random;
 *     RandomListNode(int x) { this.label = x; }
 * };
 */
public class Solution {
    /**
     * @param head: The head of linked list with a random pointe
r.
     * @return: A new head of a deep copy of the list.
     */
    public RandomListNode copyRandomList(RandomListNode head) {
        // write your code here
        if(head == null) {
            return null;
        }
```

```
        HashMap<RandomListNode, RandomListNode> map
          = new HashMap<RandomListNode, RandomListNode>();
        RandomListNode dummy = new RandomListNode(0);
        RandomListNode pre = dummy, newNode;

        while(head != null) {
            // copy next pointer
            if(map.containsKey(head)) {
                newNode = map.get(head);
            } else {
                newNode = new RandomListNode(head.label);
                map.put(head, newNode);
            }
            pre.next = newNode;

            // copy random pointer
            if(head.random != null) {
                if(map.containsKey(head.random)) {
                    newNode.random = map.get(head.random);
                } else {
                    newNode.random = new RandomListNode(head.ran
dom.label);
                    map.put(head.random, newNode.random);
                }
            }
            pre = newNode;
            head = head.next;
        }

        return dummy.next;
    }
}
```

二、复制链表

- 在原链表基础上复制结点。
  - 利用 `next` 指针，在原链表每个结点后面建立复制结点（同时复制 `value` 值， `next` 指针）。
- 调整 `random` 指针关系。

    - 将复制结点的 `random` 指针指向相应的复制结点。
- 拆分链表。

以图示说明（下面的图来自参考链接 2 中的解释）

```
比如链表是
|------------|
|            v
1  --> 2 --> 3 --> 4

第一遍扫描利用 next 指针，扫描过程中先建立 copy 结点(包括 next 指针)，得到
|--------------------------|
|                          v
1  --> 1' --> 2 --> 2' --> 3 --> 3' --> 4 --> 4'

第二遍扫描复制 random 指针的 copy
|--------------------------|
|                          v
1  --> 1' --> 2 --> 2' --> 3 --> 3' --> 4 --> 4'
       |                         ^
       |-------------------------|

最后拆分结点，一边扫描一边拆成两个链表。
拆分后第一个链表为 1-->2-->3-->4，第二链表为 1`-->2`-->3`-->4`。
```

算法复杂度

- 时间复杂度：`O(n)` 。
- 空间复杂度：`O(1)` 。【疑问：复制结点也占用了 O(n) 的空间，为何空间复杂度为 O(1)。因为复制结点占用的空间是一定需要的，而第一种解法中哈希表占用的空间是复制结点以外的额外空间。】

易错点

> 1. 在 `copyNext, copyRandom` 函数中 `head` 是局部变量，不会对 `copyRandomList` 中的 `head` 变量产生影响，所以可以把 `head` 当作局部变量使用；考虑到 `Java` 中函数调用的机制，这两个函数却改变了链表的结点内容，所以无需返回值。【注】受限于对 `Java` 语言的理解，目前还不能解释得很清楚。

Java 实现

```
/**
 * Definition for singly-linked list with a random pointer.
 * class RandomListNode {
 *     int label;
 *     RandomListNode next, random;
 *     RandomListNode(int x) { this.label = x; }
 * };
 */
public class Solution {
    /**
     * @param head: The head of linked list with a random pointe
r.
     * @return: A new head of a deep copy of the list.
     */
    public RandomListNode copyRandomList(RandomListNode head) {
        if (head == null) {
            return null;
        }

        copyNext(head);
        copyRandom(head);
        return split(head);
    }

    private void copyNext (RandomListNode head) {
        while (head != null) {
            RandomListNode newNode = new RandomListNode(head.lab
el);
            newNode.next = head.next;
            head.next = newNode;
            head = head.next.next;
        }
```

```
    }

    private void copyRandom (RandomListNode head) {
        while (head != null) {
            if (head.random != null) {
                head.next.random = head.random.next;
            }
            head = head.next.next;
        }
    }

    private RandomListNode split (RandomListNode head) {
        RandomListNode newHead = head.next;
        while (head != null) {
            RandomListNode tmp = head.next;
            head.next = head.next.next;
            if (tmp.next != null) {
                tmp.next = tmp.next.next;
            }
            head = head.next;
        }
        return newHead;
    }
}
```

参考

1. Copy List with Random Pointer | 九章算法
2. Copy List with Random Pointer | LeetCode题解
3. Copy List with Random Pointer | 数据结构与算法/leetcode/lintcode题解

Convert Sorted List to Balanced BST

Convert Sorted List to Balanced BST (leetcode lintcode)

```
Description
Given a singly linked list where elements are sorted in ascendin
g order,
convert it to a height balanced BST.

Example
                2
1->2->3  =>    / \
              1   3
```

解题思路

一、分治法

自顶向下构建 BST

- 获取当前链表中间结点，作为 BST 的根结点
  - 获取链表左半部分的中间结点，作为左子树的根结点
  - 获取链表右半部分的中间结点，作为右子树的根结点

复杂度分析

- 时间复杂度：由于采用了分治法，所以复杂度分析参考归并排序，约为 `O(nlogn)`

注：

1. 考虑到单向链表的性质，需要获取中间结点的前一个结点。如果有偶数个结点如 6 个，那么找到第 3 个结点，如果是奇数个结点如 7 个，那么找到第 3 个结点。这样将中间结点作为当前子树的根结点，将中间结点的下一个结点作为右半部分的头结点。
2. 自己又把 `findMiddle` 函数中的 `while` 写成了 `if` ，标记一下，得长记性！

java 实现

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param head: The first node of linked list.
     * @return: a tree node
     */
    public TreeNode sortedListToBST(ListNode head) {
        // write your code here
        if(head == null) {
            return null;
        }
        if(head.next == null) {
            return new TreeNode(head.val);
        }

        ListNode mid = findMiddle(head);
        ListNode leftHead = head;
        ListNode rightHead = mid.next.next;
```

```
        // 根结点
        TreeNode root = new TreeNode(mid.next.val);
        mid.next = null;
        // 左子树
        root.left = sortedListToBST(leftHead);
        // 右子树
        root.right = sortedListToBST(rightHead);

        return root;
    }
    // 取中间结点的前一个结点
    private ListNode findMiddle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head.next;
        while(fast != null && fast.next != null && fast.next.nex
t != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }
}
```

以上实现过于繁琐，以下方法实现更加简洁。

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
```

```
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param head: The first node of linked list.
     * @return: a tree node
     */
    public TreeNode sortedListToBST(ListNode head) {
        // write your code here
        return convert(head, null);
    }

    private TreeNode convert(ListNode start, ListNode end) {
        if (start == end) {
            return null;
        }

        ListNode slow = start;
        ListNode fast = start.next;

        while (fast != end && fast.next != end) {
            slow = slow.next;
            fast = fast.next.next;
        }

        TreeNode node = new TreeNode(slow.val);
        node.left = convert(start, slow);
        node.right = convert(slow.next, end);

        return node;
    }
}
```

二、非分治法

考虑到 BST 的中序遍历是一个非递减序列，因此可以使用自底向上的方法构建 BST ，先构建左子树，使用递归的方法从最底层的左子树开始构建，同时不断移动链表的头指针，使得链表的头指针永远是对应当前子树位置的。

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param head: The first node of linked list.
     * @return: a tree node
     */
    private ListNode current;

    public TreeNode sortedListToBST(ListNode head) {
        // write your code here
        int size;

        current = head;
        size = getListLength(head);

        return sortedListToBSTHelper(size);
    }
```

```
    private int getListLength(ListNode head) {
        int size = 0;

        while(head != null) {
            head = head.next;
            size++;
        }
        return size;
    }

    public TreeNode sortedListToBSTHelper(int size) {
        if(size <= 0) {
            return null;
        }

        TreeNode left = sortedListToBSTHelper(size / 2);
        TreeNode root = new TreeNode(current.val);
        current = current.next;
        TreeNode right = sortedListToBSTHelper(size - 1 - size /
2);

        root.left = left;
        root.right = right;

        return root;
    }

}
```

参考

1. Divide&Conquer Java solution Complexity? | leetcode discussion
2. Convert Sorted List to Balanced BST | 九章算法
3. Convert Sorted List to Binary Search Tree | LeetCode题解

Triangle

Triangle (leetcode lintcode)

```
Description
Given a triangle, find the minimum path sum from top to bottom.
Each step you may move to adjacent numbers on the row below.

Notice
Bonus point if you are able to do this using only O(n) extra spa
ce,
where n is the total number of rows in the triangle.

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).
```

解题思路

一、自顶向下

求解矩阵中的最大值/最小值问题，考虑采用动态规划解题。

将三角形矩阵调整一下形状，这样更容易找到两层元素坐标之间的关系。

```
[
[2],
[3,4],
[6,5,7],
[4,1,8,3]
]
```

1. 定义状态：题目求解的是最小值路径，那么可以定义 `f[i][j]` 为从起点 `[0][0]` 走到 `[i][j]` 的最小路径。
2. 定义状态转移函数：画图观察，走到 `[i][j]` 的之前一步可以是 `[i - 1][j]` 或 `[i - 1][j - 1]` ，所以有 `f[i][j] = Math.min(f[i - 1][j], f[i - 1][j - 1]) + triangle[i][j]`
3. 定义起点：从矩阵的左上角开始，所以起点是 `[0][0]` 。同时，边界序列，如最左边的一排，考虑到每个点都只能从上一行的点走到，所以也可以初始化。
4. 定义终点：也就是最终的结果，从最后一行中选取最小值即可。

Java 实现：

```java
public class Solution {
    /**
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    public int minimumTotal(int[][] triangle) {
        // write your code here
        if(triangle == null || triangle.length == 0) {
            return -1;
        }
        if(triangle[0] == null || triangle[0].length == 0) {
            return -1;
        }

        // state: f[x][y] = minimum path value from 0,0 to x,y
        int n = triangle.length;
        int[][] f = new int[n][n];

        // initialize
```

```
        f[0][0] = triangle[0][0];
        for(int i = 1; i < n; i++) {
            f[i][0] = f[i - 1][0] + triangle[i][0];
            f[i][i] = f[i - 1][i - 1] + triangle[i][i];
        }

        // top down
        for(int i = 1; i < n; i++) {
            for(int j = 1; j < i; j++) {
                f[i][j] = Math.min(f[i - 1][j], f[i - 1][j - 1])
 + triangle[i][j];
            }
        }

        // answer
        int best = f[n - 1][0];
        for(int i = 1; i < n; i++) {
            best = Math.min(best, f[n - 1][i]);
        }
        return best;
    }
}
```

另外一种解法，没有初始化最外边两排的元素，而是在求解过程中考虑边界问题。Java 实现如下

```
public class Solution {
    /**
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    public int minimumTotal(int[][] triangle) {
        if (triangle == null || triangle[0] == null ) {
            return -1;
        }

        int n = triangle.length;
        // status
        int[][] f = new int[n + 1][n + 1];
```

```
        // initialize
        f[0][0] = 0;

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= i; j++) {
                f[i][j] = Integer.MAX_VALUE;
                if (j > i - 1) {
                    f[i][j] = f[i - 1][j - 1] + triangle[i - 1][
j - 1];
                } else if (j - 1 <= 0) {
                    f[i][j] = f[i - 1][j] + triangle[i - 1][j - 1
];
                } else {
                    f[i][j] = Math.min(f[i - 1][j - 1], f[i - 1]
[j]) + triangle[i - 1][j - 1];
                }
            }
        }

        int min = Integer.MAX_VALUE;
        for (int i = 1; i <= n; i++) {
            if (f[n][i] < min) {
                min = f[n][i];
            }
        }
        return min;
    }
}
```

二、自底向上

1. 定义状态：题目求解的是最小值路径，那么可以定义 `f[i][j]` 为从起点 `[n - 1][j]` 走到 `[i][j]` 的最小路径。
2. 定义状态转移函数：画图观察， `[i][j]` 出发可到达的点是 `[i + 1][j]` 或 `[i + 1][j + 1]` ，所以有 `f[i][j] = Math.min(f[i + 1][j], f[i + 1][j + 1]) + triangle[i][j]`
3. 定义起点：从矩阵最后一行开始，所以起点是 `[n - 1][j]` 。

4. 定义终点：也就是最终的结果，就是 `[0][0]` 。

java 实现

```
public class Solution {
    /**
     * @param triangle: a list of lists of integers.
     * @return: An integer, minimum path sum.
     */
    public int minimumTotal(int[][] triangle) {
        // write your code here
        if(triangle == null || triangle.length == 0) {
            return -1;
        }
        if(triangle[0] == null || triangle[0].length == 0) {
            return -1;
        }

        // state: f[x][y] = minimum path value from n-1,i to x,y
        int n = triangle.length;
        int[][] f = new int[n][n];

        // initialize
        for(int i = 0; i < n; i++) {
            f[n - 1][i] = triangle[n - 1][i];
        }

        // down top
        for(int i = n - 2; i >= 0; i--) {
            for(int j = i; j >= 0; j--) {
                f[i][j] = Math.min(f[i + 1][j], f[i + 1][j + 1])
 + triangle[i][j];
            }
        }

        // answer
        return f[0][0];
    }
}
```

参考

1. Triangle | 九章算法
2. 什么是动态规划？动态规划的意义是什么？ | 知乎

Minimum Path Sum

Minimum Path Sum (leetcode lintcode)

```
Description
Given a m x n grid filled with non-negative numbers,
find a path from top left to bottom right which minimizes the su
m of all numbers along its path.

Notice
You can only move either down or right at any point in time.
```

解题思路

属于矩阵类型的动态规划。

1. 定义状态：题目求解的是最小和路径，那么可以定义 `f[i][j]` 为从起点 `[0][0]` 走到 `[i][j]` 的最小路径和。
2. 定义状态转移函数：画图观察，走到 `[x][y]` 的前一步可以是 `[i - 1][j]` 或 `[i][j - 1]` ，所以有 `f[i][j] = Math.min(f[i - 1][j], f[i][j - 1]) + triangle[i][j]` 。
3. 定义起点：从矩阵的左上角开始，所以起点是 `[0][0]` 。同时，第一行或第一列，考虑到每个点都只有一种走法，可以根据状态转移特性初始化，本题的初始化需要累加。
4. 定义终点：也就是最终的结果，是右下角位置 `f[i][j]` 。

注：两个方向下标的取值范围务必保证能够遍历到所有位置。

Java 实现

```java
public class Solution {
    /**
     * @param grid: a list of lists of integers.
     * @return: An integer, minimizes the sum of all numbers alo
ng its path
     */
```

```
    public int minPathSum(int[][] grid) {
        // write your code here
        if (grid == null || grid.length == 0) {
            return -1;
        }
        if (grid[0] == null || grid[0].length == 0) {
            return -1;
        }

        // state
        int m = grid.length;
        int n = grid[0].length;
        int[][] f = new int[m][n];

        // start
        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 j = 1; j < n; j++) {
            f[0][j] = f[0][j - 1] + grid[0][j];
        }

        // top down
        // i 和 j 的取值范围务必保证能够遍历所有的位置
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                f[i][j] = Math.min(f[i - 1][j], f[i][j - 1]) + g
rid[i][j];
            }
        }

        // end
        return f[m - 1][n - 1];
    }
}
```

优化空间（滚动数组）：

观察状态转移函数，`f[i][j]` 由 `f[i - 1][j]` 或 `f[i][j - 1]` 决定，与前 `i - 2` 行无关，所以可利用滚动数组进行优化。

```
public class Solution {
    /**
     * @param grid: a list of lists of integers.
     * @return: An integer, minimizes the sum of all numbers alo
ng its path
     */
    public int minPathSum(int[][] grid) {
        if (grid == null || grid.length == 0 ||
            grid[0] == null || grid[0].length == 0) {
            return 0;
        }

        int m = grid.length;
        int n = grid[0].length;
        // status
        int[][] f = new int[2][n];

        // initialize
        f[0][0] = grid[0][0];

        for (int j = 1; j < n; j++) {
            f[0][j] = f[0][j - 1] + grid[0][j];
        }

        for (int i = 1; i < m; i++) {
            f[i % 2][0] = f[(i - 1) % 2][0] + grid[i][0];
            for (int j = 1; j < n; j++) {
                f[i % 2][j] = Math.min(f[(i - 1) % 2][j], f[i % 2
][j - 1]) + grid[i][j];
            }
        }

        return f[(m - 1) % 2][n - 1];
    }
}
```

Unique Paths

Unique Paths (leetcode lintcode)

```
Description
A robot is located at the top-left corner of a m x n grid (marke
d 'Start' in the diagram below).
The robot can only move either down or right at any point in tim
e.
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?

Notice
m and n will be at most 100.

Example
| 1,1 | 1,2 | 1,3 | 1,4 | 1,5 | 1,6 | 1,7 |
|-----------------------------------------
| 2,1 |                                     |
|-----------------------------------------
| 3,1 |                                 | 3,7 |

Above is a 3 x 7 grid. How many possible unique paths are there?
```

解题思路

求方案的总数量，属于矩阵类型的动态规划。

1. 定义状态：题目求解的是到达路径数量，那么可以定义 `f[i][j]` 为从起点 `[0][0]` 走到 `[i][j]` 的路径数量。
2. 定义状态转移函数：画图观察，走到 `[i][j]` 的之前一步可以是 `[i - 1][j]` 或 `[i][j - 1]` ，所以有 `f[i][j] = f[i - 1][j] + f[i][j - 1]` 。
3. 定义起点：从矩阵的左上角开始，所以起点是 `[0][0]` 。同时，在第一行或

第一列，机器人都只有一种走法，所以初始化为 1。

4. 定义终点：也就是最终的结果，右下角位置。

Java 实现

```
public class Solution {
    /**
     * @param n, m: positive integer (1 <= n ,m <= 100)
     * @return an integer
     */
    public int uniquePaths(int m, int n) {
        if (m == 0 || n == 0) {
            return 0;
        }

        // state
        int[][] sum = new int[m][n];

        // initialize
        for (int i = 0; i < m; i++) {
            sum[i][0] = 1;
        }
        for (int j = 1; j < n; j++) {
            sum[0][j] = 1;
        }

        // top down
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                sum[i][j] = sum[i - 1][j] + sum[i][j - 1];
            }
        }

        // end
        return sum[m - 1][n - 1];
    }
}
```

优化空间（滚动数组）：

观察状态转移函数，`sum[i][j]` 由 `sum[i - 1][j]` 或 `sum[i][j - 1]` 决定，与前 `i - 2` 行无关，所以可利用滚动数组进行优化。

```
public class Solution {
    /**
     * @param n, m: positive integer (1 <= n ,m <= 100)
     * @return an integer
     */
    public int uniquePaths(int m, int n) {
        if (m == 0 || n == 0) {
            return 0;
        }

        // status
        int[][] f = new int[2][n];

        // initialize
        for (int i = 0; i < 2; i++) {
            f[i][0] = 1;
        }
        for (int i = 1; i < n; i++) {
            f[0][i] = 1;
        }

        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                f[i % 2][j] = f[(i - 1) % 2][j] + f[i % 2][j - 1
];
            }
        }

        return f[(m - 1) % 2][n - 1];
    }
}
```

Unique Paths II

Unique Paths II (leetcode lintcode)

```
Description
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.

Notice
m and n will be at most 100.

Example
For example,
There is one obstacle in the middle of a 3x3 grid as illustrated
 below.
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
The total number of unique paths is 2.
```

解题思路

解题思路参考 Unique Paths ，不同的地方在于，需要增加对当前位置数值的判断，如果为 1，则路径数量置 0 。

Java 实现：

```
public class Solution {
    /**
     * @param obstacleGrid: A list of lists of integers
     * @return: An integer
     */
```

```
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        // write your code here
        if (obstacleGrid == null || obstacleGrid.length == 0) {
            return -1;
        }
        if (obstacleGrid[0] == null || obstacleGrid[0].length ==
0) {
            return -1;
        }

        // state
        int m = obstacleGrid.length;
        int n = obstacleGrid[0].length;
        int[][] sum = new int[m][n];

        // initialize
        sum[0][0] = (obstacleGrid[0][0] == 1) ? 0 : 1;
        for (int i = 1; i < m; i++) {
            if (obstacleGrid[i][0] != 1) {
                sum[i][0] = sum[i - 1][0];
            } else {
                // sum[i][0] = 0;
                break;
            }
        }
        for (int j = 1; j < n; j++) {
            if (obstacleGrid[0][j] != 1) {
                sum[0][j] = sum[0][j - 1];
            } else {
                // sum[0][j] = 0;
                 break;
            }
        }

        // top down
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (obstacleGrid[i][j] == 1) {
                    sum[i][j] = 0;
                } else {
```

```
                    sum[i][j] = sum[i - 1][j] + sum[i][j - 1];
                }
            }
        }

        // end
        return sum[m - 1][n - 1];
    }
}
```

Climbing Stairs

Climbing Stairs (leetcode lintcode)

```
Description
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 distinc
t ways can you climb to the top?

Example
Given an example n=3 , 1+1+1=2+1=1+2=3
return 3
```

解题思路

一、动态规划

按照正常的动态规划思路实现

Java 实现

```java
public class Solution {
    /**
     * @param n: An integer
     * @return: An integer
     */
    public int climbStairs(int n) {
        // write your code here
        if (n == 0) {
            return 1;
        }

        // state
        int[] ways = new int[n + 1];

        // initialize
        ways[0] = 1;
        ways[1] = 1;

        // top down
        for (int i = 2; i < n + 1; i++) {
            ways[i] = ways[i - 1] + ways[i - 2];
        }

        // end
        return ways[n];
    }
}
```

二、动态规划 II

第一个解法使用了长为 n 的数组，空间复杂度为 `O(n)` ，观察可得，只要保存当前位置的前两步的路径数量即可。此时空间复杂度为常数。

```java
public class Solution {
    /**
     * @param n: An integer
     * @return: An integer
     */
    public int climbStairs(int n) {
        // write your code here
        if (n <= 1) {
            return 1;
        }

        int last = 1, lastlast = 1;
        int now = 0;

        for (int i = 2; i <= n; i++) {
            now = last + lastlast;
            lastlast = last;
            last = now;
        }

        return now;
    }
}
```

参考

1. Climbing Stairs | 九章算法

Jump Game

Jump Game (leetcode lintcode)

```
Description
Given an array of non-negative integers, you are initially posit
ioned 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.

Notice
This problem have two method which is Greedy and Dynamic Program
ming.
The time complexity of Greedy method is O(n).
The time complexity of Dynamic Programming method is O(n^2).

We manually set the small data set to allow you pass the test in
 both ways.
This is just to let you learn how to use this problem in dynamic
 programming ways.
If you finish it in dynamic programming ways, you can try greedy
 method to make it accept again.

Example
A = [2,3,1,1,4], return true.
A = [3,2,1,0,4], return false.
```

解题思路

一、动态规划

1. 定义状态：题目的问题——是否存在一条从起点跳到终点的路径，那么可以定义 `canJump[i]` 为从起点 `[0]` 走到 `[i]` 的路径状态， `true` 为存在， `false` 为不存在。
2. 定义状态转移函数：画图观察，是否能够跳到位置 `[i]` ，依赖于 `i` 之前的

位置 `j` 的状态 `canJump[j]` ，如果存在 `canJump[j]` 为 `true` 且 `A[j] + j >= i` ，那么说明可以跳到位置 `[i]` 。

3. 定义起点：从数组第一个元素开始，所以起点是 `[0]` ，初始化为 `true` 。
4. 定义终点：也就是最终的结果，是数组最后一个元素 `canJump[n - 1]` 。

Java 实现

```
public class Solution {
    /**
     * @param A: A list of integers
     * @return: The boolean answer
     */
    public boolean canJump(int[] A) {
        // wirte your code here
        if (A == null || A.length == 0) {
            return false;
        }

        // state
        // boolean 初始化的默认值是 false
        int n =  A.length;
        boolean[] canJump = new boolean[n];

        // initialize
        canJump[0] = true;

        // top down
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (canJump[j] && (A[j] + j >= i)) {
                    // 此处只需考虑 true 的情况，因为 boolean 默认初始化为 false
                    canJump[i] = true;
                }
            }
        }

        // end
        return canJump[n - 1];
    }

}
```

二、贪心

使用变量 `farthest` 表示能够跳到的最远的点，从左到右扫描，根据当前的位置不断更新 `farthest` ，最后对 `farthest` 与数组长度做比较。

Java 实现

```
public class Solution {
    public boolean canJump(int[] A) {
        // think it as merging n intervals
        if (A == null || A.length == 0) {
            return false;
        }
        int farthest = A[0];
        for (int i = 1; i < A.length; i++) {
            if (i <= farthest && A[i] + i >= farthest) {
                farthest = A[i] + i;
            }
        }
        return farthest >= A.length - 1;
    }
}
```

参考

1. Jump Game | 九章算法
2. LeetCode: Jump Game Total 解题报告 | Yu's garden

Jump Game II

Jump Game II (leetcode lintcode)

```
Description
Given an array of non-negative integers, you are initially posit
ioned 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 ju
mps.

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.)
```

解题思路

一、动态规划

1. 定义状态：题目的问题——从起点跳到终点的最小跳数，定义 `minSteps[i]` 为从起点 `[0]` 走到 `[i]` 的最小跳数。
2. 定义状态转移函数： `i` 之前的位置 `j` ，如果可以跳到 `i` ，那么 `minSteps[i] = minSteps[j] + 1` ，由于可能存在多条路径，所以对 `j` 的所有可能取值，应该找到 `minSteps[j]` 最小的那个位置。
3. 定义起点：从数组第一个元素开始，所以起点是 `[0]` ，初始化为 `0` 。
4. 定义终点：也就是最终的结果，是数组最后一个元素 `minSteps[n - 1]` 。

Java 实现

```
public class Solution {
    /**
     * @param A: A list of lists of integers
     * @return: An integer
     */
    public int jump(int[] A) {
        // write your code here
        if (A == null || A.length == 0) {
            return -1;
        }

        // state
        int n = A.length;
        int[] minSteps = new int[n];

        // initialize
        minSteps[0] = 0;

        // top down
        for (int i = 1; i < n; i++) {
            minSteps[i] = Integer.MAX_VALUE;
            for (int j = 0; j < i; j++) {
                if (minSteps[j] != Integer.MAX_VALUE && A[j] + j
 >= i) {
                    minSteps[i] = minSteps[j] + 1;
                    break;
                }
            }
        }

        // end
        return minSteps[n - 1];
    }
}
```

二、贪心

从左往右扫描，维护一个覆盖区间，每到达一个位置，重新计算覆盖区间的边界。

- 覆盖区间为 `[start, end]` 。
- 计算覆盖区间内每个位置 `A[i] + i` 的值，并将得到的最大值保存为 `farthest` 。
- 将 `farthest` 设为 `end` ，将 `end + 1` 设为 `start` 。

Java 实现

```
public class Solution {
    public int jump(int[] A) {
        if (A == null || A.length == 0) {
            return -1;
        }
        int start = 0, end = 0, jumps = 0;
        while (end < A.length - 1) {
            jumps++;
            int farthest = end;
            for (int i = start; i <= end; i++) {
                if (A[i] + i > farthest) {
                    farthest = A[i] + i;
                }
            }
            start = end + 1;
            end = farthest;
        }
        return jumps;
    }
}
```

参考

1. Jump Game II | 九章算法
2. LeetCode: Jump Game II 解题报告 | Yu's garden

Palindrome Partitioning II

Palindrome Partitioning II (leetcode lintcode)

```
Description
Given a string s, cut s into some substrings such that every sub
string is a palindrome.
Return the minimum cuts needed for a palindrome partitioning of
s.

Example
Given s = "aab",
Return 1 since the palindrome partitioning ["aa", "b"] could be
produced using 1 cut.
```

解题思路

一、动态规划

1. 定义状态：题目求解的是切分字符串的最小数量，那么可以定义 `minCut[i]` 为切分前 `i` 个字符所组成的字符串所需要的最少切分次数。
2. 定义状态转移函数：`minCut[i]` 的上一个状态是 `minCut[j], j < i` ，那么要求 `j + 1 ~ i` 之间的字符也是回文的，我们需要做的是找到满足条件的最小的 `minCut[j]` 。
3. 定义起点：当只有一个字符时，无需切分即满足回文 `minCut[i]` 。当长度为 `n` 的字符串中不存在回文字符串时， `minCut[i] = i - 1` ，作为初始化值。此处 `minCut[0] = -1` 。
4. 定义终点：最终的结果是指前 `n` 个字符切分的最少次数，所以是 `minCut[n]` 。

由于程序实现中已存在双重循环，如果在循环中对字符串是否回文进行判断（时间复杂度 `O(n)` ），会明显提升总体的时间复杂度（ `O(n^3)` ）。因此提前处理字符串，获取子字符串的回文状态，以额外空间换取时间效率。在预处理子字符串回文状态的时候，使用了基于区间的动态规划。

算法复杂度

- 时间复杂度：`O(n^2)`，处理字符串回文状态 `O(n^2)`，动态规划求解最小切分数量 `O(n^2)`。
- 空间复杂度：`O(n^2)`。

Java 实现

```
public class Solution {
    /**
     * @param s a string
     * @return an integer
     */
    public int minCut(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }

        // preparation
        // isPalindrome[start][length]
        // start: start of the substring
        // length: length of the substring
        boolean[][] isPalindrome = getIsPalindrome(s);

        int n = s.length();
        // status
        // minCut[i]: how many cuts do we need to cut the i char
s into palindrome substrings
        int[] minCut = new int [n + 1];

        // initialize : every single char is palindrome, so the
first n chars need n-1 cuts
        for (int i = 0; i <= n; i++) {
            minCut[i] = i - 1;
        }

        // top down
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < i; j++) {
                if ( isPalindrome[j][i - 1] ) {
```

```
                    // find the minimum minCut[j], for j < i
                    minCut[i] = Math.min(minCut[i], minCut[j] + 1
);
                }
            }
        }

        // end
        return minCut[n];
    }

    private boolean[][] getIsPalindrome (String s) {
        int n = s.length();
        boolean[][] isPalindrome = new boolean[n][n];

        // when only one char is processed
        for (int i = 0; i < n; i++) {
            isPalindrome[i][i] = true;
        }

        // when only two chars are processed
        for (int i = 0; i < n - 1; i++) {
            isPalindrome[i][i + 1] = (s.charAt(i) == s.charAt(i
+ 1));
        }
        // dynamic programming based on interval
        // the length of interval >= 3
        for (int length = 2; length < n; length++) {
            for (int start = 0; start + length < n; start++) {
                isPalindrome[start][start + length]
                    = isPalindrome[start + 1][start + length - 1
] &&
                      (s.charAt(start) == s.charAt(start + lengt
h));
            }
        }

        return isPalindrome;
    }
}
```

参考

1. Palindrome Partitioning II | 九章算法

Word Break

Word Break (leetcode lintcode)

```
Description
Given a string s and a dictionary of words dict,
determine if s can be break into a space-separated sequence of o
ne or more dictionary words.

Example
Given s = "lintcode", dict = ["lint", "code"].
Return true because "lintcode" can be break as "lint code".
```

解题思路

动态规划。

1. 定义状态：题目求解的是否能按照字典提供的字符串集合切分字符串，那么可以定义 `canBreak[i]` 为切分前 `i` 个字符的状态，`true` 表示可以切分，`false` 表示无法完成指定切分。
2. 定义状态转移函数：`canBreak[i]` 的上一个状态是 `canBreak[j], j < i`，如果 `j + 1 ~ i` 之间的字符串在字典里，且 `canBreak[j] == true`，那么 `canBreak[i] = true`。由于需要判断 `canBreak[j]`，在初始化时要先置为 `false`。
3. 定义起点：当没有字符时，将状态初始化为 `true`，`canBreak[0] = true`。
4. 定义终点：最终的结果是指前 `n` 个字符是否可以被切分，所以是 `canBreak[n]`。

注意：

1. 程序优化：考虑到一个单词的长度是有限的，所以先获取字典集合中单词的最大长度。然后以切分位置的枚举为外循环，单词长度枚举为内循环。
2. Java 中 `substring(start, end)` 返回的是字符串起始位置是 `start ~ end - 1`。
3. 内层循环中变量是单词长度 `wordLen`，对应的状态函数就是 `canBreak[i -`

`wordLen]` 。

Java 实现

```
public class Solution {
    /**
     * @param s: A string s
     * @param dict: A dictionary of words dict
     */
    public boolean wordBreak(String s, Set<String> dict) {
        if (s.length() == 0 && dict.size() == 0) {
            return true;
        } else if (s == null || s.length() == 0 ||
                   dict == null || dict.size() == 0) {
            return false;
        }

        // state
        int n = s.length();
        boolean[] canBreak = new boolean[n + 1];

        // initialize
        canBreak[0] = true;
        int maxLength = getMaxLength(dict);

        // top down
        for (int i = 1; i <= n; i++) {
            canBreak[i] = false;
            for (int wordLen = 1; wordLen <= maxLength && wordLe
n <= i; wordLen++) {
                // whether (i-wordLen+1, i) is a word?
                // corrisponding to (i-wordLen, i-1) in string s
                String word = s.substring(i - wordLen, i);
                if (canBreak[i - wordLen] && dict.contains(word)
) {
                    canBreak[i] = true;
                    break;
                }
            }
        }
```

```
            // end
            return canBreak[n];
        }

        private int getMaxLength (Set<String> dict) {
            int maxLength = 0;
            for (String word : dict) {
                maxLength = Math.max(maxLength, word.length());
            }
            return maxLength;
        }
    }
```

参考

1. Word Break | 九章算法

Longest Common Subsequence

Longest Common Subsequence (leetcode lintcode)

```
Description
Given two strings, find the longest common subsequence (LCS).
Your code should return the length of LCS.

Clarification
What's the definition of Longest Common Subsequence?
https://en.wikipedia.org/wiki/Longest_common_subsequence_problem
http://baike.baidu.com/view/2020307.htm

Example
For "ABCD" and "EDCA", the LCS is "A" (or "D", "C"), return 1.
For "ABCD" and "EACB", the LCS is "AC", return 2.
```

Longest Common Subsequence (Wekipedia)

> The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences). It differs from problems of finding common substrings: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences.

解题思路

动态规划。

1. 定义状态：定义 `f[i][j]` 为字符串 `A` 的前 `i` 个字符和字符串 `B` 的前 `j` 个字符的最长公共子序列长度。
2. 定义状态转移函数：我们来看 `f[i][j]` 的上一个状态。如果 `A` 的前 `i` 个字符 `A[i - 1]` 和字符串 `B` 的前 `j` 个字符 `B[j - 1]` 相等，那么易得到 `f[i][j] = f[i - 1][j - 1] + 1` 。如果 `A[i - 1]` 和 `B[i - 1]` 不等，那么可以在 `A` 或 `B` 前溯一个位置进行比较，取最大值 `f[i][j] = max(f[i - 1][j], f[i][j - 1])` 。

3. 定义起点：当 `A` 为空或 `B` 为空时，对应的状态函数 `f` 为 `0`，考虑到整数数组默认初始化为 `0`，也可以不显示初始化。
4. 定义终点：最终的结果是 `f[n][m]`。

注：

- 需要注意的地方是，`f[i][j]` 中 `i` 和 `j` 的取值范围分别是 `n + 1` 和 `m + 1`。
- 对字符串的操作，取第 `i` 个值是 *`A.charAt(i)`*。
- 字符串 `A` 的第 `i` 个元素是 `A.charAt(i - 1)`。

Java 实现

```
public class Solution {
    /**
     * @param A, B: Two strings.
     * @return: The length of longest common subsequence of A an
d B.
     */
    public int longestCommonSubsequence(String A, String B) {
        // write your code here
        if (A == null || A.length() == 0) {
            return 0;
        }
        if (B == null || B.length() == 0) {
            return 0;
        }

        // state
        int m = A.length();
        int n = B.length();
        int[][] f = new int[m + 1][n + 1];

        // initialize
        for (int i = 0; i < m; i++) {
            f[i][0] = 0;
        }
        for (int j = 1; j < n; j++) {
            f[0][j] = 0;
        }
```

```
        // top down
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (A.charAt(i - 1) == B.charAt(j - 1)) {
                    f[i][j] = f[i - 1][j - 1] +1;
                } else {
                    f[i][j] = Math.max(f[i - 1][j], f[i][j - 1])
;
                }
            }
        }

        // end
        return f[m][n];
    }
}
```

九章算法的一种实现：

```
public class Solution {
    /**
     * @param A, B: Two strings.
     * @return: The length of longest common subsequence of A an
d B.
     */
    public int longestCommonSubsequence(String A, String B) {
        int n = A.length();
        int m = B.length();
        int f[][] = new int[n + 1][m + 1];
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++){
                f[i][j] = Math.max(f[i - 1][j], f[i][j - 1]);
                if(A.charAt(i - 1) == B.charAt(j - 1))
                    f[i][j] = f[i - 1][j - 1] + 1;
            }
        }
        return f[n][m];
    }
}
```

参考

1. Longest Common Subsequence | 九章算法

Longest Common Substring

Longest Common Substring (leetcode lintcode)

```
Description
Given two strings, find the longest common substring.
Return the length of it.

Notice
The characters in substring should occur continuously in origina
l string. This is different with subsequence.

Example
Given A = "ABCD", B = "CBCE", return 2.

Challenge
O(n x m) time and memory.
```

解题思路

1. 定义状态：如果定义 `f[i][j]` 为字符串 `A` 的前 `i` 个字符和字符串 `B` 的前 `j` 个字符的最长公共子字符串长度，那么当 `A[i - 1]` 和 `B[j - 1]` 相等时，不容易判断 `A[i - 2]` 和 `B[j - 2]` 是否相等，在状态 `f[i][j]` 中不含有相关信息。

   转换一下思路，定义 `f[i][j]` 为以 `A[i - 1]` 和 `B[j - 1]` 为结尾的LCS的长度，这样就可以记忆连续的公共子字符串。

2. 定义状态转移函数：由以上定义，`A[i - 1] == B[j - 1]` 时，则有 `f[i][j] = f[i - 1][j - 1] + 1` ；`A[i - 1] != B[j - 1]` 时，`f[i][j] = 0` 。

3. 定义起点：当 `A` 为空或 `B` 为空时，对应的状态函数 `f` 为 `0`，考虑到整数数组默认初始化为 `0` ，也可以不显式初始化。

4. 定义终点：在 `f[0 ~ n][0 ~ m]` 中取最大值即为最终结果。

Java 实现

```
public class Solution {
    /**
     * @param A, B: Two string.
     * @return: the length of the longest common substring.
     */
    public int longestCommonSubstring(String A, String B) {
        // write your code here
        if (A == null || A.length() == 0) {
            return 0;
        }
        if (B == null || B.length() == 0) {
            return 0;
        }

        // state
        int n = A.length();
        int m = B.length();
        int[][] f = new int[n + 1][m + 1];

        // initialize
        int max = 0;
        // default f[i][j] == 0

        // top down
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (A.charAt(i - 1) != B.charAt(j - 1)) {
                    f[i][j] = 0;
                } else {
                    f[i][j] = f[i - 1][j - 1] + 1;
                }
                if (max < f[i][j]) {
                    max = f[i][j];
                }
            }
        }

        // end
```

```
            return max;
        }
    }
```

参考

Longest Increasing Continuous Subsequence

Longest Increasing Continuous Subsequence (leetcode [lintcode])

```
Description
Give an integer array，find the longest increasing continuous su
bsequence in this array.

An increasing continuous subsequence:
- Can be from right to left or from left to right.
- Indices of the integers in the subsequence should be continuou
s.

Notice
O(n) time and O(1) extra space.
```

解题思路

使用动态规划来考虑本题目，需要从左到右、从右到左遍历两次数组。

易错点：

1. `maxLen` 的初始值是 1，因为最少有一个数字，长度为 1。

Java 实现

```java
public class Solution {
    /**
     * @param A an array of Integer
     * @return  an integer
     */
    public int longestIncreasingContinuousSubsequence(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        // status
        int[] f = new int[A.length + 1];
        f[0] = 0;
        f[1] = 1;

        int maxLen = 1;
        for (int i = 2; i <= A.length; i++) {
            if (A[i - 1] > A[i - 2]) {
                f[i] = f[i - 1] + 1;
            } else {
                f[i] = 1;
            }
            maxLen = Math.max(maxLen, f[i]);
        }

        f[A.length] = 1;
        for (int i = A.length; i >= 2; i--) {
            if (A[i - 2] > A[i - 1]) {
                f[i - 1] = f[i] + 1;
            } else {
                f[i - 1] = 1;
            }
            maxLen = Math.max(maxLen, f[i - 1]);
        }

        return maxLen;
    }
}
```

考虑到状态 `f[i]` 只与 `f[i - 1]` 有关，使用滚动数组优化空间，有如下实现：

```
public class Solution {
    /**
     * @param A an array of Integer
     * @return  an integer
     */
    public int longestIncreasingContinuousSubsequence(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        int n = A.length;
        int length = 1;
        int maxLen = 1;
        for (int i = 1; i < A.length; i++) {
            if (A[i] > A[i - 1]) {
                length++;
            } else {
                length = 1;
            }
            maxLen = Math.max(maxLen, length);
        }

        length = 1;
        for (int i = A.length - 2; i >= 0; i--) {
            if (A[i] > A[i + 1]) {
                length++;
            } else {
                length = 1;
            }
            maxLen = Math.max(maxLen, length);
        }

        return maxLen;
    }
}
```

Longest Increasing Subsequence

Longest Increasing Subsequence (leetcode lintcode)

```
Description
Given a sequence of integers, find the longest increasing subseq
uence (LIS).
You code should return the length of the LIS.

Clarification
What's the definition of longest increasing subsequence?
- The longest increasing subsequence problem is to find a subseq
uence of a given sequence
in which the subsequence's elements are in sorted order, lowest
to highest,
and in which the subsequence is as long as possible.
This subsequence is not necessarily contiguous, or unique.
https://en.wikipedia.org/wiki/Longest_increasing_subsequence

Example
For [5, 4, 1, 2, 3], the LIS is [1, 2, 3], return 3
For [4, 2, 4, 5, 3, 7], the LIS is [2, 4, 5, 7], return 4

Challenge
Time complexity O(n^2) or O(nlogn)
```

解题思路

最长递增子序列的定义（维基百科）：

> 在计算机科学中，最长递增子序列（ longest increasing subsequence, LIS ）问题是指，在一个给定的数值序列中，找到一个子序列，使得这个子序列元素的数值依次递增，并且这个子序列的长度尽可能地大。最长递增子序列中的元素在原序列中不一定是连续的。

一、动态规划

1. 定义状态：定义一维状态变量 `f[i]` ，表示以第 `i` 个数字为结尾的 LIS 的长度。
2. 定义状态转移函数：考虑到 **LIS 是非连续的**，对当前状态 `f[i]` ，上一个状态可能为 `f[j], j < i` ，而且此时需要满足 `nums[i-1] > nums[j-1]` ，在所有满足此条件的状态中取最大值 `f[i] = MAX{f[j] + 1, j < i, nums[j-1] < nums[i-1]}` ；如果所有 `nums[i-1] <= nums[j-1], j = 1...i-1` ，那么 `f[i] = 1` 。
3. 定义起点： `f[0]` 为前 `0` 个数字中 LIS 的长度，为 `0` 。考虑到对于每一个 `i` ， `f[i]` 至少为 `1` ，所以初始化 `f[i] = 1, i = 1,2...,n-1` 。
4. 定义终点：最终结果为 `f[i], i = 1,2...,n-1` 的最大值。

算法复杂度

- 时间复杂度：双重循环，为 `O(n^2)` 。
- 空间复杂度： `O(n)` 。

易错点

> 1. 状态 `f[i]` 的索引和数组的索引 `nums[i - 1]` 有差值，需要注意。
> 2. 需要根据实际意义初始化，可以假设一个极端的情况，一个逆序的排序数组，那么其每个状态都是 `1` 。

Java 实现

```
public class Solution {
    /**
     * @param nums: The integer array
     * @return: The length of LIS (longest increasing subsequenc
e)
     */
    public int longestIncreasingSubsequence(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int len = nums.length;
        // status
        int[] f = new int[len + 1];

        // initialize
        // default : f[1...len] = 1

        for (int i = 1; i <= len; i++) {
            f[i] = 1;
            for (int j = 1; j < i; j++) {
                if (nums[i - 1] > nums[j - 1]) {
                    f[i] = Math.max(f[i], f[j] + 1);
                }
            }
        }

        // result
        int max = 0;
        for (int i = 1; i <= len; i++) {
            if (f[i] > max) {
                max = f[i];
            }
        }
        return max;
    }
}
```

二、二分法

实现思路是，建立一个数组 `minLast` ，依次把数组 `nums` 中的元素 `nums[i]` 放入，放入的规则是：在 `minLast` 中找到第一个比 `nums[i]` 大的元素，然后替换之，这样构造的数组是一个排序数组，所以可以用二分查找。最终 `minLast` 数组长度就是 LIS 长度，需要注意的是 `minLast` 中的数并不是 `nums` 的一个 LIS ，只是长度相等。

具体到实现可能会出现这么两种情况：

- 1） `nums[i]` 比 `minLast` 首元素小，直接替换首元素；
- 2） `nums[i]` 比 `minLast` 所有元素都大，会放在数组尾部，数组的长度增加一；
- 2） `nums[i]` 比 `minLast` 首元素小，比尾元素大，找到第一个比 `nums[i]` 的元素然后替换之。

为什么要这么做？以下是推理过程，主要参考了最长递增子序列(LIS)解法详述，理解起来稍微有点绕。

```
           0 1 2 3 4 5 6 7
nums[i]:   2 5 6 2 3 4 7 4
f[i-1]:  0 1 2 3 1 2 3 4 3
```

我们来分析一下方法一动态规划的计算过程，在求解 `f[i]` 时，将其与每一个 `f[j], j < i && f[j] < f[i]` ，这导致每个 `f[i]` 的求解复杂度是 `O(n)` ，这里是可以改进的。

通过观察可得，在求解 `f[7]` 时，无需和 `nums[3], nums[4]` 比较，只需与 `nums[5]` 比较即可，也就是说我们考察第 `i` 个元素 `nums[i-1]` 时，如果前 `i - 1` 个元素中的任一个递增子序列，其最后一个元素比 `nums[i-1]` 小，那么就可以把 `nums[i-1]` 放在子序列后面，构成一个更长的递增子序列，不需要再和子序列前面的元素比较。

此为启发一，考察第 `i` 个元素 `nums[i-1]` 时，我们只关心前 `i - 1` 个元素中递增子序列的最后一个元素值。

在求解 `f[7]` 时，前 `6` 个元素中有两个长度为 `3` 的递增子序列 `2, 5, 6` 和 `2, 3, 4` ，而 `nums[6] > 6` 且 `nums[6] > 4` ，所以可将 `nums[6]` 放在任一序列后，构成长度为 `4` 的递增子序列。其实，只要 `nums[6] > 4` 即可接在子序列后面，无需和 `6` 比较。

此外启发二，考察第 `i` 个元素 `nums[i-1]` 时，对于同样长度的递增子序列，我们只关心尾元素中的最小值。

算法复杂度

- 时间复杂度： `n` 次二分查找，为 `O(nlogn)` 。
- 空间复杂度： `O(n)` 。

Java 实现

```
public class Solution {
    /**
     * @param nums: The integer array
     * @return: The length of LIS (longest increasing subsequenc
e)
     */
    public int longestIncreasingSubsequence(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int len = nums.length;
        int[] minLast = new int[len + 1];
        minLast[0] = -1;
        for (int i = 1; i <= len; i++) {
            minLast[i] = Integer.MAX_VALUE;
        }

        for (int i = 0; i < len; i++) {
            // find the first number in minLast > nums[i]
            int index = binarySearch(minLast, nums[i]);
            minLast[index] = nums[i];
        }

        for (int i = len; i >= 1; i--) {
            if (minLast[i] != Integer.MAX_VALUE) {
                return i;
            }
        }
        return 0;
```

```
    }

    // find the first number > num
    private int binarySearch (int[] minLast, int num) {
        int start = 0, end = minLast.length - 1;
        while (start + 1 < end) {
            int mid = start + (end - start) / 2;
            if (minLast[mid] < num) {
                start = mid;
            } else {
                end = mid;
            }
        }

        if (minLast[start] > num) {
            return start;
        }
        return end;
    }
}
```

参考

1. Longest Increasing Subsequence | 九章算法
2. [LeetCode] Longest Increasing Subsequence 最长递增子序列 | Grandyang
3. 最长递增子序列(LIS)解法详述 | 杰 & C++ & Python & DM

Edit Distance

Edit Distance (leetcode lintcode)

```
Description
Given two words word1 and word2,
find the minimum number of steps required to convert word1 to wo
rd2.
(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

Example
Given word1 = "mart" and word2 = "karma", return 3.
```

解题思路

动态规划，本题与 LCS 类似。

1. 定义状态：定义 `f[i][j]` 为字符串 `A` 的前 `i` 个字符 匹配 字符串 `B` 的前 `j` 个字符需要执行的最少操作数量 。
2. 定义状态转移函数：对于当前状态 `f[i][j]` ，由于涉及操作较多，所以需要详细讨论：
   - `A[i - 1] == B[j - 1]` 时。
   - 无操作：上一个状态可能是 `f[i - 1][j - 1]` ，此时无需任何操作。
     - 插入操作：如果需要对 `A` 末尾执行一次插入操作，插入之后 `A[i] == B[j - 1]` ，那么执行操作前 `A` 的前 `i` 个字符和 `B` 的前 `j - 1` 个字符相匹配，上一个状态是 `f[i][j - 1]` 。
     - 删除操作：如果需要对 `A` 末尾执行一次删除操作，删除之后 `A[i - 2] == B[j - 1]` ，那么执行操作前 `A` 的前 `i - 1` 个字符和 `B` 的前 `j` 个字符相匹配，上一个状态是 `f[i - 1][j]` 。
     - 综上， 当前状态需要取以上三种情况的最小值。

    - `A[i - 1] != B[j - 1]` 时。
        - 替换操作：将 `A` 的第 `i` 个字符替换为 `B` 的前 `j` 个字符，那么执行操作前 `A` 的前 `i - 1` 个字符和 `B` 的前 `j - 1` 个字符相匹配，上一个状态是 `f[i - 1][j - 1]` 。
        - 插入操作：如果需要对 `A` 末尾执行一次插入操作，插入之后 `A[i] == B[j - 1]` ，那么执行操作前 `A` 的前 `i` 个字符和 `B` 的前 `j - 1` 个字符相匹配，上一个状态是 `f[i][j - 1]` 。
        - 删除操作：如果需要对 `A` 末尾执行一次删除操作，删除之后 `A[i - 2] == B[j - 1]` ，那么执行操作前 `A` 的前 `i - 1` 个字符和 `B` 的前 `j` 个字符相匹配，上一个状态是 `f[i - 1][j]` 。
        - 综上，当前状态需要取以上三种情况的最小值。
3. 定义起点：起点 `f[0][0] == 0` ，当 `A` 为空或 `B` 为空时，对应的状态函数 `f[0][j]` 或 `f[i][0]` ，从起点出发分别需要 `j` 或 `i` 次操作。
4. 定义终点：最终结果即为 `f[n][m]` 。

易错点：

> 1. 在处理边界情况时，如果其中一个字符串长度为 0，程序时可以处理的，所以只要考虑字符串为空的情况即可。

Java 实现

```
public class Solution {
    /**
     * @param word1 & word2: Two string.
     * @return: The minimum number of steps.
     */
    public int minDistance(String word1, String word2) {
        // write your code here
        if (word1 == null || word2 == null) {
            return 0;
        }

        int n = word1.length();
        int m = word2.length();

        // state
        int[][] f = new int[n + 1][m + 1];
```

```
        // initialize
        for (int i = 0; i <= n; i++) {
            f[i][0] = i;
        }
        for (int j = 1; j <= m; j++) {
            f[0][j] = j;
        }

        // top down
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (word1.charAt(i - 1) == word2.charAt(j - 1))
{
                    f[i][j] = Math.min(f[i - 1][j - 1],
                                       Math.min(f[i][j - 1] + 1,
 f[i - 1][j] + 1));
                } else {
                    f[i][j] = Math.min(f[i - 1][j - 1],
                                       Math.min(f[i - 1][j], f[i
][j - 1]))
                                        + 1;
                }
            }
        }

        // end
        return f[n][m];
    }
}
```

优化空间（滚动数组）：

观察状态转移函数，`f[i][j]` 由 `f[i - 1][j]` 或 `f[i][j - 1]` 决定，与前 `i - 2` 行无关，所以可利用滚动数组进行优化。

```
public class Solution {
    /**
     * @param word1 & word2: Two string.
     * @return: The minimum number of steps.
     */
    public int minDistance(String w1, String w2) {
        if (w1 == null || w2 == null ) {
            return 0;
        }

        int m = w1.length();
        int n = w2.length();
        // status
        int[][] f = new int[2][n + 1];
        // initialize
        for (int i = 0; i <= n; i++) {
            f[0][i] = i;
        }

        for (int i = 1; i <= m; i++) {
            f[i % 2][0] = i;
            for (int j = 1; j <= n; j++) {
                if (w1.charAt(i - 1) == w2.charAt(j - 1)) {
                    f[i % 2][j] = Math.min(f[(i - 1) % 2][j - 1]
, Math.min(f[(i - 1) % 2][j], f[i % 2][j - 1]) + 1);
                } else {
                    f[i % 2][j] = Math.min(f[(i - 1) % 2][j - 1]
, Math.min(f[(i - 1) % 2][j], f[i % 2][j - 1])) + 1;
                }
            }
        }

        return f[m % 2][n];
    }
}
```

参考

1. Edit Distance | 九章算法

Distinct Subsequences

Distinct Subsequences (leetcode lintcode)

```
Description
Given a string S and a string T, count the number of distinct su
bsequences of T in S.
A subsequence of a string is a new string which is formed from t
he original string
by deleting some (can be none) of the characters without disturb
ing the relative positions
of the remaining characters. (ie, "ACE" is a subsequence of "ABC
DE" while "AEC" is not).

Example
Given S = "rabbbit", T = "rabbit", return 3.

Challenge
Do it in O(n^2) time and O(n) memory.
O(n^2) memory is also acceptable if you do not know how to optim
ize memory.
```

解题思路

一、动态规划

1. 定义状态：定义 `f[i][j]` 为字符串 `S` 的前 `i` 个字符挑出字符串 `T` 的前 `j` 个字符有多少种方案。
2. 定义状态转移函数：对于当前状态 `f[i][j]` ，详细讨论如下：
    - `S.charAt(i - 1) == T.charAt(j - 1)` 时。
    - 无操作：上一个状态可能是 `f[i - 1][j - 1]` ，此时无需任何操作。
        - 删除操作：如果需要对 `S` 末尾执行一次删除操作，删除之后 `S.charAt[i - 2] == T.charAt[j - 1]` ，那么执行操作前 `S` 的前 `i - 1` 个字符和 `T` 的前 `j` 个字符相匹配，上一个状态是 `f[i - 1][j]` 。不能对 `T` 进行删除操作。

        - 求解的是方案总数，所以 `f[i][j] = f[i - 1][j - 1] + f[i - 1][j]` 。
    - `S.charAt(i - 1) != T.charAt(j - 1)` 时。
        - 删除操作：那么只能对 `S` 末尾执行一次删除操作，删除之后 `S.charAt[i - 2] == T.charAt[j - 1]` ，那么执行操作前 `S` 的前 `i - 1` 个字符和 `T` 的前 `j` 个字符相匹配，上一个状态是 `f[i - 1][j]` 。
        - 所以 `f[i][j] = f[i - 1][j]` 。
3. 定义起点：
    - 当 `S` 为空，`T` 不为空时，无法在 `S` 中找到 `T` ，所以 `f[0][j] = 0, j = 1 ~ m` 。
    - 当 `S` 不空，`T` 为空时，将 `S` 的所有字符删掉可得空字符串，也即空串是任意字符串的子串，所以 `f[i][0] = 1, i = 1 ~ n` 。注：第一次做的时候没疏忽了，没考虑到这种情况。
    - 当 `S` 和 `T` 都为空时，`f[0][0] = 1` 。
4. 定义终点：最终结果即为 `f[n][m]` 。注：初始化时也需要注意取值范围。

Java 实现

```
public class Solution {
    /**
     * @param S, T: Two string.
     * @return: Count the number of distinct subsequences
     */
    public int numDistinct(String S, String T) {
        // write your code here
        if (S == null || T == null) {
            return 0;
        }

        int n = S.length();
        int m = T.length();

        if (m > n) {
            return 0;
        }

        // state
```

```
        int[][] f = new int[n + 1][m + 1];


        // initialize
        for (int i = 0; i <= n; i++) {
            f[i][0] = 1;
        }

        // top down
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (S.charAt(i - 1) == T.charAt(j - 1)) {
                    f[i][j] = f[i - 1][j - 1] + f[i - 1][j];
                } else {
                    f[i][j] = f[i - 1][j];
                }
            }
        }

        // end
        return f[n][m];
    }
}
```

二、动态规划+状态压缩（滚动数组）

Interleaving String

Interleaving String (leetcode lintcode)

```
Description
Given three strings: s1, s2, s3, determine
whether s3 is formed by the interleaving of s1 and s2.

Example
For s1 = "aabcc", s2 = "dbbca"
When s3 = "aadbbcbcac", return true.
When s3 = "aadbbbaccc", return false.

Challenge
O(n2) time or better
```

解题思路

1. 定义状态：可定义 `f[i][j][k]` 为 `s1` 的前 `i` 个字符和 `s2` 的前 `j` 个字符是否能组成 `s3` 的前 `k` 个字符串，考虑到必然有 `i + j = k` ，所以第三个维度 `k` 可略去，直接定义 `f[i][j]` 即可。
2. 定义状态转移函数：对于当前状态 `f[i][j]` ，详细讨论如下。
   - `s1.charAt(i - 1) == s3.charAt(i + j - 1)` 时，那么对应的上一个状态为 `f[i - 1][j]` ，如果上个状态为 `true` ，那么 `f[i][j] = true` 。
   - `s2.charAt(i - 1) == s3.charAt(i + j - 1)` 时，那么对应的上一个状态为 `f[i][j - 1]` ，如果上个状态为 `true` ，那么 `f[i][j] = true` 。
3. 定义起点：
   - 易错点：又一次没考虑清楚初始化情况，这里需要专门说明一下。对于二维的动规问题，**`f[i][0]`** 和 **`f[0][j]`** 都是边界条件，需要进行初始化，初始化时要同时注意边界的状态转移。
   - 当 `s1` 为空，`s2` 不为空时，依次判断 `s2` 与 `s3` 对应字符是否相等，同时还要考虑上一个状态是否为 `true` 完成初始化。
   - 当 `s2` 为空，`s1` 不为空时，依次判断 `s1` 与 `s3` 对应字符是否相

等完成初始化，同时还要考虑上一个状态是否为 `true` 。

4. 定义终点：最终结果即为 `f[n][m]` 。注：初始化时也需要注意取值范围是数组长度加一。

易错点：

> 1. 在初始化时，比较 `s1` 和 `s3` 时，如果要用 substring 函数，需要使用 equals 函数来比较，也即 `s1.substring(0, i).equals(s3.substring(0, i))` 。

Java 实现

```
public class Solution {
    /**
     * Determine whether s3 is formed by interleaving of s1 and
s2.
     * @param s1, s2, s3: As description.
     * @return: true or false.
     */
    public boolean isInterleave(String s1, String s2, String s3)
{
        // write your code here

        int lens1 = s1.length();
        int lens2 = s2.length();
        int lens3 = s3.length();

        if ((s1 == null || lens1 == 0) && s2 == s3) {
            return true;
        }
        if ((s2 == null || lens2 == 0) && s1 == s3) {
            return true;
        }

        if (lens1 + lens2 != lens3) {
            return false;
        }

        // state
        boolean[][] f = new boolean[lens1 + 1][lens2 + 1];
```

```
        // initialize
        f[0][0] = true;
        for (int i = 1; i <= lens1; i++) {
            if (s1.charAt(i - 1) == s3.charAt(i - 1) && f[i - 1][
0]) {
                f[i][0] = true;
            } else {
                f[i][0] = false;
            }
        }
        for (int j = 1; j <= lens2; j++) {
            if (s2.charAt(j - 1) == s3.charAt(j - 1) && f[0][j -
1]) {
                f[0][j] = true;
            } else {
                f[0][j] = false;
            }
        }


        // top down
        for (int i = 1; i <= lens1; i++) {
            for (int j = 1; j <= lens2; j++) {
                if (f[i - 1][j] && s1.charAt(i - 1) == s3.charAt
(i + j - 1) ||
                    f[i][j - 1] && s2.charAt(j - 1) == s3.charAt
(i + j - 1)) {
                    f[i][j] = true;
                }
            }
        }

        // end
        return f[lens1][lens2];
    }
}
```

参考

1. Interleaving String | 九章算法

House Robber

House Robber (leetcode lintcode)

```
Description
You are a professional robber planning to rob houses along a str
eet.
Each house has a certain amount of money stashed,
the only constraint stopping you from robbing each of them is th
at
adjacent houses have security system connected and
it will automatically contact the police
if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of
 money of each house,
determine the maximum amount of money you can rob tonight withou
t alerting the police.
```

解题思路

求解最大值，使用动态规划。

1. 定义状态：可定义 `f[i]` 为抢劫前 `i` 个房子可得到的最大值（最多的钱）。
2. 定义状态转移函数：对于当前状态 `f[i]` ，如果不抢劫第 `i` 个房子，那么上一个状态对应 `f[i - 1]` ；如果抢劫第 `i` 个房子，那么不能抢第 `i - 1` 个房子，上一个状态对应 `f[i - 2]` 。
3. 定义起点：初始化状态转移方程无法计算的情况。
4. 定义终点：最终结果 `f[n]` 。

Java 实现

```
public class Solution {
    /**
     * @param A: An array of non-negative integers.
     * return: The maximum amount of money you can rob tonight
     */
    public long houseRobber(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        int n = A.length;
        // status
        long[] f = new long[n + 1];
        // initialize
        f[0] = 0;
        f[1] = A[0];

        for (int i = 2; i <= n; i++) {
            f[i] = Math.max(f[i - 1], f[i - 2] + A[i - 1]);
        }

        return f[n];
    }
}
```

空间优化（滚动数组）：观察状态转移方程，可以发现 `f[i]` 只依赖于 `f[i - 1]` 和 `f[i - 2]` ，所以大小为 3 的数组就够了。然后通过取模来不断更新数组，求得结果。

Java 实现

```
public class Solution {
    /**
     * @param A: An array of non-negative integers.
     * return: The maximum amount of money you can rob tonight
     */
    public long houseRobber(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        int n = A.length;
        // status
        long[] f = new long[3];
        // initialize
        f[0] = 0;
        f[1] = A[0];

        for (int i = 2; i <= n; i++) {
            f[i % 2] = Math.max(f[(i - 1) % 2], f[(i - 2) % 2] +
 A[i - 1]);
        }

        return f[n % 2];
    }
}
```

House Robber II

```
Description
After robbing those houses on that street,
the thief has found himself a new place for his thievery so that
 he will not get too much attention.
This time, all houses at this place are arranged in a circle.
That means the first house is the neighbor of the last one.
Meanwhile, the security system for these houses remain the same
as for those in the previous street.

Given a list of non-negative integers representing the amount of
 money of each house,
determine the maximum amount of money you can rob tonight withou
t alerting the police.

Notice
This is an extension of House Robber.
```

解题思路

参考 House Robber 的思路，如果房子围城一个圈，那么偷第一个房子，就不能偷最后一个房子，可以进一步分解问题。

```java
public class Solution {
    /**
     * @param nums: An array of non-negative integers.
     * return: The maximum amount of money you can rob tonight
     */
    public int houseRobber2(int[] nums) {
        // write your code here
        if (nums == null || nums.length == 0) {
            return 0;
        }
        if (nums.length == 1) {
            return nums[0];
        }

        int ans1 = houseRobber1(nums, 0, nums.length - 2);
        int ans2 = houseRobber1(nums, 1, nums.length - 1);

        return Math.max(ans1, ans2);
    }

    private int houseRobber1(int[] A, int start, int end) {
        if (start == end) {
            return A[start];
        }
        int[] f = new int[A.length + 1];
        f[start] = A[start];
        f[start + 1] = Math.max(A[start + 1], f[start]);

        for (int i = start + 2; i <= end; i++) {
            f[i] = Math.max(f[i-1], f[i-2] + A[i]);
        }
        return f[end];
    }
}
```

可以使用滚动数组优化空间

```
public class Solution {
    /**
     * @param nums: An array of non-negative integers.
     * return: The maximum amount of money you can rob tonight
     */
    public int houseRobber2(int[] nums) {
        // write your code here
        if (nums == null || nums.length == 0) {
            return 0;
        }
        if (nums.length == 1) {
            return nums[0];
        }

        int ans1 = houseRobber1(nums, 0, nums.length - 2);
        int ans2 = houseRobber1(nums, 1, nums.length - 1);

        return Math.max(ans1, ans2);
    }

    private int houseRobber1(int[] A, int start, int end) {
        if (start == end) {
            return A[start];
        }
        int[] f = new int[2];
        f[start % 2] = A[start];
        f[(start + 1) % 2] = Math.max(A[start + 1], f[start]);

        for (int i = start + 2; i <= end; i++) {
            f[i % 2] = Math.max(f[(i-1) % 2], f[(i-2) % 2] + A[i
]);
        }
        return f[end % 2];
    }
}
```

Maximal Square

Maximal Square (leetcode lintcode)

```
Description
Given a 2D binary matrix filled with 0's and 1's,
find the largest square containing all 1's and return its area.

Example
For example, given the following matrix:
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
Return 4.
```

解题思路

本题目求解最大全"1"正方形边长，考虑使用动态规划，关键点在于如何定义状态。

1. 定义状态：考虑到正方形可以用一个顶点和边长来确定，我们定义 `f[i][j]` 是右下角坐标为 `[i, j]` 的全“1”正方形的最大边长。定义右下角也是为后续的状态转移做准备。
2. 定义状态转移函数：在 `matrix[i][j] == 1` 的情况下，以 `[i, j]` 为右下角的全“1”正方形的最大边长，和以 `[i - 1, j]`, `[i, j - 1]`, `[i - 1, j - 1]` 为右下角的三个全“1”正方形的最大边长有关，而且取决于以上三者的最小值。
3. 定义起点：初始化第一行第一列，需要考虑矩阵只有一行或者一列的情况。
4. 定义终点：`f[i][j]` 的最大值为边长，求平方即可。

Java 实现

```java
public class Solution {
    /**
     * @param matrix: a matrix of 0 and 1
     * @return: an integer
```

```
     */
    public int maxSquare(int[][] matrix) {
        if (matrix == null || matrix.length == 0 ||
            matrix[0] == null || matrix[0].length == 0) {
            return 0;
        }

        int m = matrix.length;
        int n = matrix[0].length;
        // status
        int[][] f = new int[m][n];
        // initialize
        int max = 0;
        for (int i = 0; i < m; i++) {
            f[i][0] = matrix[i][0];
            max = f[i][0] > max ? f[i][0] : max;
        }
        for (int i = 1; i < n; i++) {
            f[0][i] = matrix[0][i];
            max = f[0][i] > max ? f[0][i] : max;
        }

        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                f[i][j] = Integer.MAX_VALUE;
                if (matrix[i][j] == 1) {
                    f[i][j] = Math.min(f[i - 1][j - 1], Math.min
(f[i - 1][j], f[i][j - 1])) + 1;
                } else {
                    f[i][j] = 0;
                }
                if (f[i][j] > max) {
                    max = f[i][j];
                }
            }
        }

        return max * max;
    }
}
```

空间优化（滚动数组）：

不难发现，`f[i][j]` 的取值只与 `f[i - 1][j]`, `f[i - 1][j - 1]`, `f[i - 1][j - 1]` 有关，也就是说矩阵 `f` 第 `i` 行的结果只与第 `i - 1` 行有关，所以可以使用滚动数组优化空间。

Java 实现

```
public class Solution {
    /**
     * @param matrix: a matrix of 0 and 1
     * @return: an integer
     */
    public int maxSquare(int[][] matrix) {
        if (matrix == null || matrix.length == 0 ||
            matrix[0] == null || matrix[0].length == 0) {
            return 0;
        }

        int m = matrix.length;
        int n = matrix[0].length;
        // status
        int[][] f = new int[2][n];
        // initialize
        int max = 0;
        for (int i = 0; i < 2; i++) {
            f[i][0] = matrix[i][0];
            max = f[i][0] > max ? f[i][0] : max;
        }
        for (int i = 1; i < n; i++) {
            f[0][i] = matrix[0][i];
            max = f[0][i] > max ? f[0][i] : max;
        }

        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                // f[i][j] = Integer.MAX_VALUE;
                if (matrix[i][j] == 1) {
                    f[i % 2][j] = Math.min(f[(i - 1) % 2][j - 1]
```

```
, Math.min(f[(i - 1) % 2][j], f[i % 2][j - 1])) + 1;
                } else {
                    f[i % 2][j] = 0;
                }
                if (f[i % 2][j] > max) {
                    max = f[i % 2][j];
                }
            }
        }

        return max * max;
    }
}
```

参考

1. Maximal Square | 九章算法

Backpack

Backpack (leetcode lintcode)

```
Description
Given n items with size Ai, an integer m denotes the size of a b
ackpack.
How full you can fill this backpack?

Notice
You can not divide any item into small pieces.

Example
If we have 4 items with size [2, 3, 5, 7], the backpack size is
11, we can select [2, 3, 5],
so that the max size we can fill this backpack is 10.
If the backpack size is 12. we can select [2, 3, 7] so that we c
an fulfill the backpack.
You function should return the max size we can fill in the given
 backpack.

Challenge
O(n x m) time and O(m) memory.
O(n x m) memory is also acceptable if you do not know how to opt
imize memory.
```

解题思路

背包问题：单次选择+最大体积。

一、二维状态

1. 定义状态：定义 `f[i][j]` 为 `A` 的前 `i` 个数是否能填满大小为 `j` 的背包。
2. 定义状态转移函数：对于当前状态 `f[i][j]` ，
    - 如果不计入 `A[i - 1]` ，则 `f[i][j]` 对应的上一个状态是 `f[i - 1]`

`[j]` ；
    - 如果计入 `A[i - 1]` ，满足条件 `j >= A[i - 1]` ，那么 `f[i][j]` 上一个状态是 `f[i - 1][j - A[i - 1]]`
    - 以上两种情况满足其一即可。
3. 定义起点：
    - 对于二维的动规问题，**`f[i][0]`** 和 **`f[0][j]`** 都是边界条件，需要进行初始化，初始化时要同时注意边界的状态转移。
    - 当 `i != 0` ，`j == 0` 时，如果一个数字都不计入，即取前 `i` 个数字中的零个，那么可得结果为零，所以 `f[i][0]` 的状态为 `true` 。
    - 当 `i == 0` ，`j != 0` 时，前零个元素的和显然无法得到值 `j` ，`f[0][j]` 状态为 `false` 。
    - `f[0][0]` 状态为 `true` 。
4. 定义终点：最终结果即为 `f[n][m]` 。注：初始化时也需要注意取值范围是数组长度加一。

易错点：

> 1. 状态函数 `f[][]` 的初始化范围为 `boolean[][] f = new boolean[n + 1][m + 1];` 。
> 2. 双重循环的起始值，结束值。
> 3. `j` 和 `A[i - 1]` 之间的关系，可以等于。

算法复杂度

- 时间复杂度：`O(n*m)` 。
- 空间复杂度：`O(n*m)` 。

Java 实现

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    public int backPack(int m, int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        int n = A.length;
        // state
        boolean[][] f = new boolean[n + 1][m + 1];

        // initialize
        f[0][0] = true;

        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= m; j++) {
                f[i][j] = f[i - 1][j] || (j >= A[i - 1] && f[i -
1][j - A[i - 1]]) ;
            }
        }

        // answer
        for (int j = m; j >= 0; j--) {
            if (f[n][j]) {
                return j;
            }
        }
        return 0;
    }
}
```

使用滚动数组优化空间，可使空间复杂度降至 `O(m)` 。

```
public class Solution {
```

```
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    public int backPack(int m, int[] A) {
        if (m <= 0 || A == null || A.length == 0) {
            return 0;
        }

        int n = A.length;
        // status
        boolean[][] f = new boolean[2][m + 1];
        // initialize
        f[0][0] = true;
        for (int i = 1; i <= m; i++) {
            f[0][i] = false;
        }

        for (int i = 1; i <= n; i++) {
            f[i % 2][0] = true;
            for (int j = 1; j <= m; j++) {
                f[i % 2][j] = f[(i - 1) % 2][j] ||
                                ((A[i-1] <= j) && f[(i - 1) % 2]
[j - A[i - 1]]);
            }
        }

        int res = 0;
        for (int i = m; i >= 0; i--) {
            if(f[n % 2][i]) {
                res = i;
                break;
            }
        }
        return res;
    }
}
```

二、一维状态

1. 定义状态：定义 `f[j]` 为 `A` 的前 `i` 个数是否能填满大小为 `j` 的背包。
2. 定义状态转移函数：对于当前状态 `f[j]` ，
    - 如果不计入 `A[i - 1]` ，则 `f[j]` 保持状态不变；
    - 如果计入 `A[i - 1]` ，满足条件 `j > A[i - 1]` ，那么 `f[j]` 上一个状态是 `f[j - 1]`
    - 注意题目中的隐含条件为每个数只能使用一次。在使用二维数组定义状态是，其中一个维度避免了同一个数字的重复使用。所以在使用一维状态转移函数时，要避免出现这种情况，在内循环的取值需要从大到小。
3. 定义起点：
    - `f[0]` 状态为 `true` 。
4. 定义终点：最终结果即为 `f[m]` 。

算法复杂度

- 时间复杂度： `O(nm)` 。
- 空间复杂度： `O(m)` 。

Java 实现

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    public int backPack(int m, int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        int n = A.length;
        // state
        boolean[] f = new boolean[m + 1];

        // initialize
        f[0] = true;

        for (int i = 1; i <= n; i++) {
            // j starts from m to avoid repeating use of iterm A
[i - 1]
            for (int j = m; j >= 0; j--) {
                if (j >= A[i - 1] && f[j - A[i - 1]]) {
                    f[j] = f[j - A[i - 1]];
                }
            }
        }

        // answer
        for (int j = m; j >= 0; j--) {
            if (f[j]) {
                return j;
            }
        }
        return 0;
    }
}
```

参考

1. Lintcode: Backpack | neverlandly
2. Backpack | lintcode题解

Backpack II

Backpack II (leetcode lintcode)

```
Description
Given n items with size Ai and value Vi, and a backpack with siz
e m.
What's the maximum value can you put into the backpack?

Notice
You cannot divide item into small pieces and
the total size of items you choose should smaller or equal to m.

Example
Given 4 items with size [2, 3, 5, 7] and value [1, 5, 2, 4],
and a backpack with size 10. The maximum value is 9.

Challenge
O(n x m) memory is acceptable, can you do it in O(m) memory?
```

解题思路

背包问题：单次选择+最大价值。

一、动态规划

1. 定义状态：定义 `max[i][j]` 为 `A` 的前 `i` 个数，取出若干个物品后，填满体积为 `j` 的背包，所能获得的最大值。
2. 定义状态转移函数：对于当前状态 `f[i][j]` ，
   - 如果不计入 `A[i - 1]` ，则 `f[i][j]` 对应的上一个状态是 `f[i - 1][j]` ；
   - 如果计入 `A[i - 1]` ，满足条件 `j > A[i - 1]` ，那么 `f[i][j]` 上一个状态是 `f[i - 1][j - A[i - 1]]` ；
   - 以上两种情况取最大值即可。
3. 定义起点：

    - 当 `i != 0` ，`j == 0` 时，如果一个数字都不计入，即取前 `i` 个数字中的零个，那么最大取值为零，所以 `f[i][0] = 0` 。
    - 当 `i == 0` ，`j != 0` 时，前零个元素的和显然无法填满大小为 `j` 的背包， `f[0][j] = 0` 。
    - `f[0][0] = 0` 。
4. 定义终点：最终结果即为 `f[n][m]` 。

算法复杂度

- 时间复杂度： `O(nm)` 。
- 空间复杂度： `O(nm)` 。

Java 实现

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A & V: Given n items with size A[i] and value V[i]
     * @return: The maximum value
     */
    public int backPackII(int m, int[] A, int V[]) {
        // write your code here
        if (A == null || A.length == 0 ||
            V == null || V.length == 0) {
            return 0;
        }

        int n = A.length;

        // state
        int[][] max = new int[n + 1][m + 1];

        // initialize
        // default

        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= m; j++) {
                max[i][j] = max[i - 1][j];
                if (j >= A[i - 1]) {
                    max[i][j] = Math.max(max[i - 1][j], max[i - 1
][j - A[i - 1]] + V[i - 1]);
                }
            }
        }

        // answer
        return max[n][m];
    }
}
```

使用滚动数组优化空间，可使空间复杂度降至 `O(m)` 。

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A & V: Given n items with size A[i] and value V[i]
     * @return: The maximum value
     */
    public int backPackII(int m, int[] A, int V[]) {
        if (m <= 0 || A == null || A.length == 0
            || V == null || V.length == 0) {
            return 0;
        }

        int n = A.length;
        // status
        int[][] f = new int[2][m + 1];
        // initialize
        f[0][0] = 0;
        for (int i = 1; i <= m; i++) {
            f[0][i] = 0;
        }

        for (int i = 1; i <= n; i++) {
            // f[i % 2][0] = 0;
            for (int j = 1; j <= m; j++) {
                f[i % 2][j] = f[(i - 1) % 2][j];
                if (A[i - 1] <= j) {
                    f[i % 2][j] = Math.max(f[i % 2][j],
                                           f[(i - 1) % 2][j - A[i-1
]] + V[i - 1]);
                }
            }
        }
        return f[n % 2][m];
    }
}
```

二、动态规划 II

按照 Backpack 问题中的思路进行优化。

算法复杂度

- 时间复杂度：`O(nm)`。
- 空间复杂度：`O(m)`。

Java 实现

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A & V: Given n items with size A[i] and value V[i]
     * @return: The maximum value
     */
    public int backPackII(int m, int[] A, int V[]) {
        // write your code here
        if (A == null || A.length == 0 ||
            V == null || V.length == 0) {
            return 0;
        }

        int n = A.length;

        // state
        int[] max = new int[m + 1];

        // initialize
        // default

        for (int i = 1; i <= n; i++) {
            for (int j = m; j >= 0; j--) {
                if (j >= A[i - 1]) {
                    max[j] = Math.max(max[j], max[j - A[i - 1]]
+ V[i - 1]);
                }
            }
        }

        // answer
        return max[m];
    }
}
```

参考

Backpack III

Backpack III (lintcode)

```
Description
Given n kind of items with size Ai and value Vi( each item has a
n infinite number available)
and a backpack with size m.
What's the maximum value can you put into the backpack?

Notice
You cannot divide item into small pieces and the total size of i
tems you choose should smaller or equal to m.

Example
Given 4 items with size [2, 3, 5, 7] and value [1, 5, 2, 4], and
 a backpack with size 10.
The maximum value is 15.
```

解题思路

背包问题：重复选择+最大价值。

一、二维状态

1. 定义状态：定义 `f[i][j]` 为 `A` 的前 `i` 个数填满体积为 `j` 的背包可得的最大价值，每个数可重复使用。
2. 定义状态转移函数：对于当前状态 `f[i][j]` ，
   - 如果不计入 `A[i - 1]` ，则 `f[i][j]` 对应的上一个状态是 `f[i - 1][j]` ；
   - 如果计入 `A[i - 1]` ，需满足条件 `j >= A[i - 1]` ，那么 `f[i][j]` 上一个状态是 `f[i][j - A[i - 1]]` ；
   - 以上两种情况取和即可。
3. 定义起点：
   - 当 `i != 0` ，`j == 0` 时，如果一个数字都不计入，即取前 `i` 个数

字中的零个，那么最大取值为零，所以 `f[i][0] = 0` 。
    - 当 `i == 0` ，`j != 0` 时，前零个元素的和显然无法填满大小为 `j` 的背包， `f[0][j] = 0` 。
    - `f[0][0] = 0` 。
4. 定义终点：最终结果即为 `f[n][m]` 。

算法复杂度

- 时间复杂度： `O(nm)` 。
- 空间复杂度： `O(nm)` 。

Java 实现

```
public class Solution {
  public static int backpackIII(int[] A, int[] V, int m) {
      if (A == null || A.length == 0 ||
          V == null || V.length == 0 ||
          m <= 0) {
          return 0;
      }

      int n = A.length;
      // status
      int[][] f = new int[n + 1][m + 1];
      for (int i = 1; i <= n; i++) {
          for (int j = 1; j <= m; j++) {
              f[i][j] = f[i-1][j];
              if (j >= A[i-1]) {
                  f[i][j] = Math.max(f[i][j], f[i][j - A[i-1]] +
 V[i-1]);
              }
          }
      }

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

使用滚动数组优化空间，可使空间复杂度降至 `O(m)` 。

```
public class Solution {
  public static int backpackIII(int[] A, int[] V, int m) {
      if (A == null || A.length == 0 ||
          V == null || V.length == 0 ||
          m <= 0) {
          return 0;
      }

      int n = A.length;
      // status
      int[][] f = new int[2][m + 1];
      for (int i = 1; i <= n; i++) {
          for (int j = 1; j <= m; j++) {
              f[i % 2][j] = f[(i-1) % 2][j];
              if (j >= A[i-1]) {
                  f[i % 2][j] = Math.max(f[i % 2][j], f[i % 2][j
 - A[i-1]] + V[i-1]);
              }
          }
      }

      return f[n % 2][m];
  }
}
```

二、一维状态

1. 定义状态：定义 `f[j]` 为前 `i` 个数填满大小为 `j` 的背包，最大值是多少。
2. 定义状态转移函数：对于当前状态 `f[j]` ，
   - 如果不计入 `A[i]` ，则 `f[j]` 保持状态不变；
   - 如果计入 `A[i]` ，需满足条件 `j > A[i]` ，那么 `f[j]` 上一个状态是 `f[j - A[i]]`
   - 对于当前物品 `i` ，若 `j` 从小到大的话，很可能在 `j` 之前的 `j - A[i]` 时已经放过第 `i` 件物品了，在 `j` 时再放就是重复放入；若 `j` 从大到小，则 `j` 之前的所有情况都没有更新过，不可能放过第 `i` 件物品，所以不会重复放入。【DH：想得不是很明白...】
3. 定义起点：

    - `f[0]` 状态为 `0` 。
4. 定义终点：最终结果即为 `f[m]` 。

算法复杂度

- 时间复杂度： `O(nm)` 。
- 空间复杂度： `O(m)` 。

Java 实现

```
public class Solution {
    /**
     * @param A an integer array
     * @param V an integer array
     * @param m an integer
     * @return an array
     */
    public int backPackIII(int[] A, int[] V, int m) {
        if (A == null || A.length == 0 ||
              V == null || V.length == 0 ||
              m <= 0) {
              return 0;
          }
        int[] f = new int[m + 1];

        for(int i = 0; i < A.length; i++){
            for(int j = 1; j <= m; j++){
                if (j >= A[i]) {
                  f[j] = Math.max(f[j], f[j - A[i]] + V[i]);
                 }
            }
        }
        return f[m];
    }
}
```

参考

1. [LintCode] Backpack I II III IV V VI [背包六问]

2. Backpack III 440 | LintCode题解
3. PPT | 背包问题大汇总 | 九章算法

Backpack IV

Backpack IV ()

```
Description
Given n items with size nums[i] which an integer array and all p
ositive numbers, no duplicates.
An integer target denotes the size of a backpack.
Find the number of possible fill the backpack.
Each item may be chosen unlimited number of times

Example
Given candidate items [2,3,6,7] and target 7,

A solution set is:
[7]
[2, 2, 3]
return 2
```

解题思路

背包计数问题：重复选择+唯一排列+装满可能性总数。

一、二维状态

1. 定义状态：定义 `f[i][j]` 为前 `i` 个数填满大小为 `j` 的背包的方法数，每个数可使用多次。
2. 定义状态转移函数：对于当前状态 `f[i][j]` ，
   - 如果不计入 `A[i - 1]` ，则 `f[i - 1][j]` 保持状态不变；
   - 如果计入 `A[i - 1]` ，需满足条件 `j >= A[i - 1]` ，考虑到每个数可以使用多次，那么 `f[i][j]` 上一个状态是 `f[i][j - A[i-1]]` 。
3. 定义起点：
   - `f[i][0]` 状态为 `1` ，这是根据问题的含义来定义的。
4. 定义终点：最终结果即为 `f[n][m]` 。

算法复杂度

- 时间复杂度：`O(nm)`。
- 空间复杂度：`O(nm)`。

Java 实现

```
public class Solution {
  public static int backPackIV (int[] A, int m) {
      if (A == null || A.length == 0 || m <= 0) {
          return 0;
      }

      int n = A.length;
    // status
    int[][] f = new int[n + 1][m + 1];
    // initialize
    for (int i = 0; i <= n; i++) {
        f[i][0] = 1;
    }
    for (int i = 1; i <= m; i++) {
        f[0][i] = 0;
    }

      for (int i = 1; i <= n; i++) {
          for (int j = 1; j <= m; j++) {
              f[i][j] = f[i-1][j];
              if (j >= A[i-1]) {
                  f[i][j] += f[i][j - A[i-1]];
              }
          }
      }
      return f[n][m];
  }
}
```

使用滚动数组可将空间复杂度降至 `O(m)`。

```
public class Solution {
  public static int backPackIV (int[] A, int m) {
      if (A == null || A.length == 0 || m <= 0) {
          return 0;
      }

      int n = A.length;
    // status
    int[][] f = new int[2][m + 1];
    // initialize
    for (int i = 0; i < 2; i++) {
        f[i][0] = 1;
    }
    for (int i = 1; i <= m; i++) {
        f[0][i] = 0;
    }

      for (int i = 1; i <= n; i++) {
          f[i % 2][0] = 1;
          for (int j = 1; j <= m; j++) {
              f[i % 2][j] = f[(i-1) % 2][j];
              if (j >= A[i-1]) {
                  f[i % 2][j] += f[i % 2][j - A[i-1]];
              }
          }
      }
      return f[n % 2][m];
  }
}
```

二、一维状态

1. 定义状态：定义 `f[j]` 为前 `i` 个数填满大小为 `j` 的背包的方法数。
2. 定义状态转移函数：对于当前状态 `f[j]` ，
   - 如果不计入 `A[i]` ，则 `f[j]` 保持状态不变；
   - 如果计入 `A[i]` ，需满足条件 `j > A[i]` ，那么 `f[j]` 上一个状态是 `f[j - A[i]]`
   - 对于当前物品 `i` ，若 `j` 从小到大的话，很可能在 `j` 之前的 `j -`

`A[i]` 时已经放过第 `i` 件物品了，在 `j` 时再放就是重复放入；若 `j` 从大到小，则 `j` 之前的所有情况都没有更新过，不可能放过第 `i` 件物品，所以不会重复放入。【DH：想得不是很明白...】

3. 定义起点：
    - `f[0]` 状态为 `1` ，这是根据问题的含义来定义的。
4. 定义终点：最终结果即为 `f[m]` 。

易错点

> 1. 初始状态的定义，注意题目的具体含义。

算法复杂度

- 时间复杂度：`O(nm)` 。
- 空间复杂度：`O(m)` 。

Java 实现

```
public class Solution {
    /**
     * @param nums an integer array and all positive numbers, no
 duplicates
     * @param target an integer
     * @return an integer
     */
    public int backPackIV(int[] A, int m) {
        if (A == null || A.length == 0 || m <= 0) {
              return 0;
          }
        int n = A.length;
        // status
        int[] f = new int[m + 1];
        // initialize
        f[0] = 1;

        for(int i = 0; i < n; i++){
            for(int j = A[i]; j <= m; j++){
                f[j] += f[j - A[i]];
            }
        }

        return f[m];
    }
}
```

参考

1. Backpack IV 562 | LintCode题解
2. PPT | 背包问题大汇总 | 九章算法

Backpack V

Backpack V (lintcode)

```
Description
Given n items with size nums[i] which an integer array and all p
ositive numbers.
An integer target denotes the size of a backpack.
Find the number of possible fill the backpack.
Each item may only be used once.

Example
Given candidate items [1,2,3,3,7] and target 7,
A solution set is:
[7]
[1, 3, 3]
return 2
```

解题思路

背包问题：单次选择+装满可能性总数。

一、二维状态

1. 定义状态：定义 `f[i][j]` 为前 `i` 个数填满大小为 `j` 的背包的方法数，每个数只使用一次。
2. 定义状态转移函数：对于当前状态 `f[i][j]` ，
   - 如果不计入 `A[i - 1]` ，则 `f[i - 1][j]` 保持状态不变；
   - 如果计入 `A[i - 1]` ，需满足条件 `j >= A[i - 1]` ，那么 `f[i][j]` 上一个状态是 `f[i - 1][j - A[i-1]]` 。
3. 定义起点：
   - `f[i][0]` 状态为 `1` ，这是根据问题的含义来定义的。
4. 定义终点：最终结果即为 `f[n][m]` 。

算法复杂度

- 时间复杂度：`O(nm)`。
- 空间复杂度：`O(nm)`。

Java 实现

```
public calss Solution {
  /**
     * @param A an integer array and all positive numbers, no du
plicates
     * @param m an integer
     * @return an integer
     */
  public static int backPackV (int[] A, int m) {
      if (A == null || A.length == 0 || m <= 0) {
          return 0;
      }

      int n = A.length;
    // status
    int[][] f = new int[n + 1][m + 1];
    // initialize
    for (int i = 0; i <= n; i++) {
        f[i][0] = 1;
    }
    for (int i = 1; i <= m; i++) {
        f[0][i] = 0;
    }

      for (int i = 1; i <= n; i++) {
          for (int j = 1; j <= m; j++) {
              f[i][j] = f[i-1][j];
              if (j >= A[i-1]) {
                  f[i][j] += f[i-1][j - A[i-1]];
              }
          }
      }
      return f[n][m];
  }
}
```

使用滚动数组进行优化，可将空间复杂度降至 `O(m)` 。

```
public calss Solution {
  /**
     * @param A an integer array and all positive numbers, no du
plicates
     * @param m an integer
     * @return an integer
     */
  public static int backPackV (int[] A, int m) {
      if (A == null || A.length == 0 || m <= 0) {
          return 0;
      }

      int n = A.length;
    // status
    int[][] f = new int[2][m + 1];
    // initialize
    for (int i = 0; i < 2; i++) {
        f[i][0] = 1;
    }
    for (int i = 1; i <= m; i++) {
        f[0][i] = 0;
    }

      for (int i = 1; i <= n; i++) {
          f[i % 2][0] = 1;
          for (int j = 1; j <= m; j++) {
              f[i % 2][j] = f[(i-1) % 2][j];
              if (j >= A[i-1]) {
                  f[i % 2][j] += f[(i-1) % 2][j - A[i-1]];
              }
          }
      }
      return f[n % 2][m];
  }
}
```

二、一维状态

可参考题目 Backpack IV 的解释，由于不同元素只能取一次，所以内层循环需要从大到小遍历。

算法复杂度

- 时间复杂度： `O(nm)` 。
- 空间复杂度： `O(m)` 。

Java 实现

```
public class Solution {
    /**
     * @param A an integer array and all positive numbers, no du
plicates
     * @param m an integer
     * @return an integer
     */
    public int backPackIV(int[] A, int m) {
        if (A == null || A.length == 0 || m <= 0) {
              return 0;
          }
        int n = A.length;
        // status
        int[] f = new int[m + 1];
        // initialize
        f[0] = 1;

        for(int i = 0; i < n; i++){
            for(int j = m; j >= A[i]; j--){
                f[j] += f[j - A[i]];
            }
        }

        return f[m];
    }
}
```

参考

1. PPT | 背包问题大汇总 | 九章算法

Backpack VI

Backpack VI (lintcode)

```
Description
Given an integer array nums with all positive numbers and no dup
licates,
find the number of possible combinations that add up to a positi
ve integer target.

Notice
The different sequences are counted as different combinations.

Example
Given nums = [1, 2, 4], target = 4

The possible combination ways are:
[1, 1, 1, 1]
[1, 1, 2]
[1, 2, 1]
[2, 1, 1]
[2, 2]
[4]
return 6
```

解题思路

背包问题：重复选择+不同排列+装满可能性总数。

本题目也可以视为常规的动规问题，和 leetcode 上的 Combination Sum IV 是同一道题目。

1. 定义状态：定义 `f[i]` 为填满大小为 `i` 的背包，一共有多少种方法。
2. 定义状态转移函数：对于当前状态 `f[i]` ，由于先后选取 `A[j]` 属于不同的方法，只要 `i >= A[j], 0 <= j < n` ，每一个 `A[j]` 都对应一种取法，所以有 `f[i] = sum{f[i - A[j]], i >= A[j], 0<=j<n}` 。
3. 定义起点： `f[0]` 状态为 `1` 。

4. 定义终点：最终结果即为 `f[m]` 。

注：

> 1. 如果数组有负数，就必须要限制每个数使用的次数，否则会得到无穷多种排列方式, 比如 `A = [1, -1], target = 1` 。

算法复杂度

- 时间复杂度： `O(nm)` 。
- 空间复杂度： `O(m)` 。

Java 实现

```
public class Solution {
    /**
     * @param nums an integer array and all positive numbers, no
 duplicates
     * @param target an integer
     * @return an integer
     */
    public int backPackVI(int[] A, int m) {
        if (A == null || A.length == 0 || m <= 0) {
            return 0;
        }

        int n = A.length;
        // status
        int[] f = new int[m + 1];
        // initialize
        f[0] = 1;

        for (int i = 1; i <= m; i++) {
            for (int j = 0; j < n; j++) {
                if (i >= A[j]) {
                    f[i] += f[i - A[j]];
                }
            }
        }
        return f[m];
    }
}
```

参考

1. [leetcode] 377. Combination Sum IV 解题报告 | 小榕流光的专栏

k Sum

k Sum (leetcode lintcode)

```
Description
Given n distinct positive integers, integer k (k <= n) and a num
ber target.
Find k numbers where sum is target. Calculate how many solutions
 there are?

Example
Given [1,2,3,4], k = 2, target = 5.
There are 2 solutions: [1,4] and [2,3].
Return 2.
```

解题思路

一、动态规划

1. 定义状态：由于问题涉及的自变量比较多，前 `i` 个数字，取其中 `j` 个，其和为 `t` 的方案数量，所以定义三维状态变量 `f[i][j][t]` ，完成状态定义是非常关键的一步，如果定义准确，问题基本已经得到解决了。
2. 定义状态转移函数：对于当前状态 `f[i][j][t]` ，详细讨论如下
   - 如果不计入 `A[i - 1]` ，则 `f[i][j][t]` 对应的上一个状态是 `f[i - 1][j][t]` ；
   - 如果计入 `A[i - 1]` ，需满足条件 `j >= A[i - 1]` ，那么 `f[i][j][t]` 上一个状态是 `f[i - 1][j][t]` 或 `f[i - 1][j - 1][t - A[i - 1]]` ；
3. 定义起点：
   - `f[i][0][0]` 从前 `i` 个数字，取其中 `0` 个，其和为 `0` 的方案数量为1。
   - `f[0][j][0], j != 0` 从前 `0` 个数字，取其中 `j` 个，其和为 `0` 的方案数量为0。
   - `f[0][0][t], t != 0` 从前 `0` 个数字，取其中 `0` 个，其和为 `t` 的方案数量为0。

4. 定义终点：最终结果即为 `f[n][k][target]` 。注：初始化时也需要注意取值范围是数组长度加一。

算法复杂度

- 时间复杂度： `O(n*k*target)`
- 空间复杂度： `O(n*k*target)`

易错点：

> 1. 多重循环需确保自变量的取值范围不会造成数组越界，这里的越界包含两部分，一是自变量本身的左右边界，二是自变量之间的边界限定。如本题中变量 `j` 的起始点从 1 开始，因为内部循环出现了 `j - 1` 。

Java 实现

```
public class Solution {
    /**
     * @param A: an integer array.
     * @param k: a positive integer (k <= length(A))
     * @param target: a integer
     * @return an integer
     */
    public int kSum(int A[], int k, int target) {
        // write your code here
        if (A == null || A.length == 0) {
            return 0;
        }

        int n = A.length;
        // state
        // f[i][j][t] take j numbers from the first i numbers, h
ow many combinations' sum is t
        int[][][] f = new int[n + 1][k + 1][target + 1];

        // initialize
        for (int i = 0; i <= n; i++) {
            f[i][0][0] = 1;
        }

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= k && j <= i; j++) {
                for (int t = 1; t <= target; t++) {
                    f[i][j][t] = f[i - 1][j][t];
                    if (t >= A[i - 1]) {
                        f[i][j][t] = f[i - 1][j][t] + f[i - 1][j
 - 1][t - A[i - 1]];
                    }
                }
            }
        }

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

观察不难得到，`f[i]` 仅和 `f[i - 1]` 有关，所以可使用滚动数组进行优化，将空间复杂度降至 `O(k*target)` 。

```
public class Solution {
    /**
     * @param A: an integer array.
     * @param k: a positive integer (k <= length(A))
     * @param target: a integer
     * @return an integer
     */
    public int kSum(int A[], int k, int target) {
        if (A == null || A.length < k || k <= 0) {
            return 0;
        }

        int n = A.length;
        // status
        int[][][] f = new int[2][k + 1][target + 1];
        for (int i = 0; i < 2; i++) {
            f[i][0][0] = 1;
        }

        for (int i = 1; i <= n; i++) {
            f[i % 2][0][0] = 1;
            for (int j = 1; j <= k && j <= i; j++) {
                for (int m = 1; m <= target; m++) {
                    f[i % 2][j][m] = f[(i - 1)%2][j][m];
                    if (m >= A[i - 1]) {
                        f[i % 2][j][m] += f[(i - 1) % 2][j - 1][
m - A[i - 1]];
                    }
                }
            }
        }

        return f[n % 2][k][target];
    }
}
```

二、动态规划 II

考虑到输入是单序列，每次最多增加一个元素，使用二维数组定义状态函数。实现思路参考 Backpack 的解法二。

需要注意的是：循环的嵌套顺序、变量的遍历顺序都需要进行调整，目的是为了避免重复计算。具体细节思考的不是很清楚。

Java 实现

```
public class Solution {
    /**
     * @param A: an integer array.
     * @param k: a positive integer (k <= length(A))
     * @param target: a integer
     * @return an integer
     */
    public int kSum(int A[], int k, int target) {
        // write your code here
        if (A == null || A.length == 0) {
            return 0;
        }

        int n = A.length;
        // state
        // f[i][j][t] indicates the number of solutions for sele
cting j numbers from A[0 … i – 1] with the sum of t.
        int[][] f = new int[k + 1][target + 1];

        // initialize
        f[0][0] = 1;

        for (int i = 1; i <= n; i++) {
            for (int t = target; t >= 0; t--) {
                for (int j = 1; j <= k && j <= i; j++) {
                    if (t >= A[i - 1]) {
                        f[j][t] = f[j][t] + f[j - 1][t - A[i - 1
]];
                    }
                }
            }
        }

        return f[k][target];
    }
}
```

参考

1. lintcode: k Sum 解题报告 | Yu's garden

Minimum Adjustment Cost

Minimum Adjustment Cost (leetcode lintcode)

```
Description
Given an integer array, adjust each integers so that
the difference of every adjacent integers are not greater than a
 given number target.
If the array before adjustment is A, the array after adjustment
is B,
you should minimize the sum of |A[i]-B[i]|

Notice
You can assume each number in the array is a positive integer an
d not greater than 100.

Example
Given [1,4,2,3] and target = 1, one of the solutions is [2,3,2,3
],
the adjustment cost is 2 and it's minimal.
Return 2.
```

解题思路

1. 定义状态：定义二维状态变量 `f[i][k]` ，前 `i` 个数字，第 `i` 个数调整为 `k` 时，满足相邻两数之差 `<= target` ，所需要的最小代价。
2. 定义状态转移函数：对于当前状态 `f[i][k]` ，其上一个状态为 `f[i - 1][j]` ，从上一个状态到下一个状态需要做的调整为 `f[i - 1][j] + abs(A.get(i - 1) - k)` ，根据题意 `k` 的范围是 `0 <= k <= 100` ，所以在满足 `abs(j - k)` 的条件下，找出做出最小调整的 `k` 值。
3. 定义起点：
   - `f[0][i]` 从前 `0` 个数字，取第 `0` 个，调整为 `i` 的方案数量为0。
4. 定义终点：最终结果即为 `f[n][i], i = 0,1,...100` 中的最小值。

Java 实现

```
public class Solution {
    /**
     * @param A: An integer array.
     * @param target: An integer.
     */
    public int MinAdjustmentCost(ArrayList<Integer> A, int targe
t) {
        int n = A.size();

        // state
        int[][] f = new int[n + 1][101];

        // initialize
        for (int i = 0; i <= n; i++) {
            for (int j = 1; j <= 100; j++) {
                f[i][j] = Integer.MAX_VALUE;
            }
        }
        for (int i = 1; i <= 100; i++) {
            f[0][i] = 0;
        }

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= 100; j++) {
                if (f[i - 1][j] != Integer.MAX_VALUE) {
                    for (int k = 1; k <= 100; k++) {
                        if (Math.abs(j - k) <= target) {
                            /*
                            if (f[i][k] > f[i - 1][j] + Math.abs
(A.get(i - 1) - k)) {
                                f[i][k] = f[i - 1][j] + Math.abs
(A.get(i - 1) - k);
                            }*/
                            int tmp = f[i - 1][j] + Math.abs(A.g
et(i - 1) - k);
                            f[i][k] = Math.min(f[i][k], tmp);
                        }
                    }
                }
```

```
            }
        }

        int ans = Integer.MAX_VALUE;
        for (int i = 1; i <= 100; i++) {
            if (f[n][i] < ans) {
                ans = f[n][i];
            }
        }
        return ans;
    }
}
```

参考

1. Minimum Adjustment Cost | Yu's garden

Maximum Subarray

Maximum Subarray (leetcode lintcode)

```
Description
Given an array of integers, find a contiguous subarray which has
 the largest sum.

Notice
The subarray should contain at least one number.

Example
Given the array [−2,2,−3,4,−1,2,1,−5,3],
the contiguous subarray [4,−1,2,1] has the largest sum = 6.

Challenge
Can you do it in time complexity O(n)?
```

解题思路

一、动态规划

1. 定义状态：定义一维状态变量 `f[i]` ，表示以第 `i` 个数字 `nums[i - 1]` 为结尾的子数组的最大和。
2. 定义状态转移函数：对于当前状态 `f[i]` ，其上一个状态为 `f[i - 1]` ，
   - 当 `f[i - 1] <= 0` 时，舍弃之前的部分，`f[i]` 取第 `i` 个数字 `nums[i - 1]` 即可。
   - 当 `f[i - 1] > 0` 时，连续求和即可。
   - 根据上述讨论可得 `f[i] = f[i - 1] <= 0 ? nums[i - 1] : f[i - 1] + nums[i - 1]` 。考虑到当前状态只取决于上一个状态，所以使用一个变量保存状态可以节省内存开销。
3. 定义起点：
   - `f[0] = 0` 。
4. 定义终点：最终结果即为 `f[i], i = 0,1,...n` 的最大值。

Java 实现

```java
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A integer indicate the sum of max subarray
     */
    public int maxSubArray(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        // state
        int curSum = nums[0];
        int maxSum = nums[0];

        for (int i = 1; i < nums.length; i++) {
            curSum = curSum <= 0 ? nums[i] : curSum + nums[i];
            // curSum = Math.max(nums[i], curSum + nums[i]);
            maxSum = Math.max(curSum, maxSum);
        }

        return maxSum;
    }
}
```

二、前缀和

对前缀和 `sum[i] = nums[0] + nums[1] + … + nums[i]` ，

那么第 `i + 1` 到 `j + 1` 个连续数字的和为 `sum[i … j] = sum[j] - sum[i - 1]` 。

以上是使用前缀和进行解题的基础。

具体到本题，以第 `i` 个数字结尾的子数组的最大值，等于当前的前缀和 `curSum` 减其子数组的最小前缀和 `minSum` 。

Java 实现

```java
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A integer indicate the sum of max subarray
     */
    public int maxSubArray(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int maxSum = Integer.MIN_VALUE;
        int curSum = 0, minSum = 0;
        for (int i = 0; i < nums.length; i++) {
            curSum += nums[i];
            maxSum = Math.max(maxSum, curSum - minSum);
            minSum = Math.min(minSum, curSum);
        }

        return maxSum;
    }
}
```

参考

1. [LeetCode] Maximum Subarray | 喜刷刷
2. Maximum Subarray | 九章算法
3. Maximum Subarray — LeetCode | Code Ganker

Maximum Subarray II

Maximum Subarray II (leetcode lintcode)

```
Description
Given an array of integers, find two non-overlapping subarrays w
hich have the largest sum.
The number in each subarray should be contiguous.
Return the largest sum.

Notice
The subarray should contain at least one number

Example
For given [1, 3, -1, 2, -1, 2],
the two subarrays are [1, 3] and [2, -1, 2] or [1, 3, -1, 2] and
 [2],
they both have the largest sum 7.

Challenge
Can you do it in time complexity O(n) ?
```

解题思路

题目要求找到两个不重叠的子数组和的最大值，关键在于不重叠。考虑如何利用这个条件将题目转化为 Maximum Subarray ，也即在数组中寻找最大子数组。可以将一个数组用隔板划分为两个，然后分别在左、右两个子数组中寻找最大子数组。

如果在遍历划分点的时候，同时寻找两个最大子数组，双重循环的时间复杂度为 `O(n^2)` 。我们可以先寻找不同划分左数组的最大子数组，将和存储在数组中；然后寻找不同划分右数组的最大子数组，将和存储在数组中；最后两个数组分别求和取最大值即可。

易错点：

1. 最后求和的时候需注意，由于有划分的存在，所以两个数组的索引差 `1` 。
2. 在求最大值、最小值时，变量的初始化需要使用 `Integer.MIN_VALUE` 或 `Integer.MAX_VLUAE` 。

Java 实现

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: An integer denotes the sum of max two non-overla
pping subarrays
     */
    public int maxTwoSubArrays(ArrayList<Integer> nums) {
        if (nums == null || nums.size() == 0) {
            return 0;
        }

        int curSum = 0;
        int minSum = 0;
        int max = Integer.MIN_VALUE;
        int[] maxSumLeft = new int[nums.size()];
        for (int i = 0; i < nums.size(); i++) {
            curSum += nums.get(i);
            max = Math.max(max, curSum - minSum);
            minSum = Math.min(minSum, curSum);
            maxSumLeft[i] = max;
        }

        curSum = 0;
        minSum = 0;
        max = Integer.MIN_VALUE;
        int[] maxSumRight = new int[nums.size()];
        for (int i = nums.size() - 1; i >= 0; i--) {
            curSum += nums.get(i);
            max = Math.max(max, curSum - minSum);
            minSum = Math.min(minSum, curSum);
            maxSumRight[i] = max;
        }
```

```
        max = Integer.MIN_VALUE;
        for (int i = 0; i < nums.size() - 1; i++) {
            max = Math.max(max, maxSumLeft[i] + maxSumRight[i + 1
]);
        }

        return max;
    }
}
```

以下实现似乎更简洁

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: An integer denotes the sum of max two non-overla
pping subarrays
     */
    public int maxTwoSubArrays(ArrayList<Integer> nums) {
        // write your code
        if (nums == null || nums.size() == 0) {
            return Integer.MIN_VALUE;
        }
        int n = nums.size();

        int curtSum = nums.get(0);
        int[] left = new int[n];
        left[0] = nums.get(0);
        for (int i = 1; i < n; i++) {
            curtSum = Math.max(curtSum + nums.get(i), nums.get(i
));
            left[i] = Math.max(left[i-1], curtSum);
        }

        curtSum = nums.get(n - 1);
        int[] right = new int[n];
        right[n - 1] = nums.get(n - 1);
        for (int i = n - 2; i >= 0; i--) {
            curtSum = Math.max(curtSum + nums.get(i), nums.get(i
```

```
));
            right[i] = Math.max(right[i+1], curtSum);
        }

        int max = Integer.MIN_VALUE;
        for (int i = 0; i < n - 1; i++) {
            int temp = left[i] + right[i + 1];
            max = Math.max(temp, max);
        }

        return max;
    }
}
```

参考

1. Maximum Subarray II | 九章算法

Maximum Subarray III

Maximum Subarray III (leetcode lintcode)

```
Description
Given an array of integers and a number k,
find k non-overlapping subarrays which have the largest sum.
The number in each subarray should be contiguous.
Return the largest sum.

Notice
The subarray should contain at least one number

Example
Given [-1,4,-2,3,-2,3], k=2, return 8
```

解题思路

本题目是典型的划分型动态规划。

一、常规解法

1. 定义状态：定义二维状态变量 `f[i][j]` ，表示前 `i` 个数字，取 `j` 个子数组时，得到的最大和。
2. 定义状态转移函数：对于当前状态 `f[i][j]` ，由于划分是任意的，所以不止和当前的元素相关。其上一个状态可能为 `f[m][j - 1]` ，考虑到 `i >= j` ， `m` 的取值范围是 `j - 1 <= m < i` ，则有 `f[i][j] = max{f[m][j - 1] + maxSubArray(m + 1, i)}` 。
3. 定义起点：
   - `f[i][0]` 从前 `i` 个数字， `0` 个子数组，得到的最大和为 `0` 。
   - `f[0][i]` 从前 `0` 个数字， `i` 个子数组，得到的最大和为 `0` 。
4. 定义终点：最终结果即为 `f[n][k]` 。

算法复杂度

- 时间复杂度： `O(k*n^3)` 。

- 空间复杂度：`O(kn)`。

易错点

> 1. 涉及划分时，需要注意几个自变量 `i, j, m` 之间的大小关系。

Java 实现

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @param k: An integer denote to find k non-overlapping sub
arrays
     * @return: An integer denote the sum of max k non-overlappi
ng subarrays
     */
    public int maxSubArray(int[] nums, int k) {
        if (nums == null || nums.length == 0
            || k <= 0 || k > nums.length) {
            return -1;
        }

        int n = nums.length;
        // status
        int[][] f = new int[n + 1][k + 1];
        // initialize

        for (int j = 1; j <= k; j++) {
            for (int i = j; i <= n; i++) {
                f[i][j] = Integer.MIN_VALUE;
                for (int m = j-1; m < i; m++) {
                    f[i][j] = Math.max(f[i][j], f[m][j-1] + maxS
ubArray(nums, m, i-1));
                }
            }
        }

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

```
    private int maxSubArray(int[] nums, int start, int end) {
        int curSum = 0;
        int maxSum = Integer.MIN_VALUE;
        for (int i = start; i <= end; i++) {
            curSum = Math.max(curSum + nums[i], nums[i]);
            maxSum = Math.max(maxSum, curSum);
        }

        return maxSum;
    }
}
```

时间复杂度太高，进一步思考，是否可以简化求数组最大和的操作，考虑到一个数组从两个方向求最大和子数组的结果是一样的，所以把 `maxSubArray` 函数和自变量 `m` 循环结合在一起，将时间复杂度将至 `O(k*n^2)` 。

Java 实现

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @param k: An integer denote to find k non-overlapping sub
arrays
     * @return: An integer denote the sum of max k non-overlappi
ng subarrays
     */
    public int maxSubArray(int[] nums, int k) {
        if (nums == null || nums.length == 0
            || k <= 0 || k > nums.length) {
            return -1;
        }

        int n = nums.length;
        // status
        int[][] f = new int[n + 1][k + 1];
        // initialize

        for (int j = 1; j <= k; j++) {
            for (int i = j; i <= n; i++) {
```

```
                f[i][j] = Integer.MIN_VALUE;

                int curSum = 0;
                int maxSum = Integer.MIN_VALUE;
                for (int m = i-1; m >= j-1; m--) {
                    curSum = Math.max(curSum + nums[m], nums[m])
;
                    maxSum = Math.max(maxSum, curSum);

                    f[i][j] = Math.max(f[i][j], f[m][j-1] + maxS
um);
                }
            }
        }

        return f[n][k];
    }

}
```

二、划分型动规解法

对方法一进行思考，其实仍然可以进行优化。对于 `f[i][j] = max{f[m][j - 1] + maxSubArray(m + 1, i)}, j - 1 <= m < i` ，每当新增一个元素的时候，之前的最大值仍是最大值，但是需要和新加进来的元素进行对比，所以只需要把之前遍历得到的结果用一个变量存储起来，就无需反复遍历了。

1. 定义状态：
    - 定义 `localMax[i][j]` ，表示前 `i` 个数字，取 `j` 个划分时，包含第 `i` 个数字的最大和。
    - 定义 `globalMax[i][j]` ，表示前 `i` 个数字，取 `j` 个划分时，可以不包含第 `i` 个数字的最大和。
2. 定义状态转移函数：
    - 对 `localMax[i][j]` ，由于一定包含第 `i` 个数字，那么有两种情况，
        - 第 `i` 个数字属于第 `j` 个子数组，对应 `localMax[i - 1][j] + nums[i - 1]` ；
        - 第 `i` 个数字不属于第 `j` 个子数组，对应 `localMax[i - 1][j - 1] + nums[i - 1]` ；

        - 比较两者取最大值皆可。
    - 对 `globalMax[i][j]`，比较 `localMax[i][j]` 和 `globalMax[i - 1][j]` 中的最大值即可。
3. 定义起点：
    - 对 `localMax[i][j]`，在处理时会出现 `localMax[i - 1][i]`，根据逻辑此种情况不存在，由于涉及取最大值操作，所以初始化为 `Integer.MIN_VALUE`。
4. 定义终点：最终结果即为 `globalMax[n][k]`。

算法复杂度

- 时间复杂度：`O(nk)`。
- 空间复杂度：`O(nk)`。

注：本题目可以对 `localMax[i][j]` 进行滚动数组优化。

Java 实现

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @param k: An integer denote to find k non-overlapping sub
arrays
     * @return: An integer denote the sum of max k non-overlappi
ng subarrays
     */
    public int maxSubArray(int[] nums, int k) {
        if (nums == null || nums.length == 0
            || k <= 0 || k > nums.length) {
            return -1;
        }

        int n = nums.length;
        // status
        int[][] globalMax = new int[n + 1][k + 1];
        int[][] localMax = new int[n + 1][k + 1];
        // initialize

        for (int j = 1; j <= k; j++) {
            localMax[j - 1][j] = Integer.MIN_VALUE;
            for (int i = j; i <= n; i++) {
                localMax[i][j] = Math.max(localMax[i - 1][j], gl
obalMax[i - 1][j - 1]) + nums[i - 1];
                if (i == j) {
                    globalMax[i][j] = localMax[i][j];
                } else {
                    globalMax[i][j] = Math.max(globalMax[i - 1][
j], localMax[i][j]);
                }
            }
        }

        return globalMax[n][k];
    }

}
```

参考

1. LintCode-Maximum Subarray III | LiBlog
2. Lintcode: Maximum Subarray III | 姜糖水·码农
3. Lintcode - Maximum Subarray III | 雯雯熊孩子
4. LintCode 43 [Maximum Subarray III] | Jason_Yuan
5. [LintCode]Maximum Subarray III | 今際の国の呵呵君
6. Maximum Subarray III | 九章算法

Maximum Product Subarray

Maximum Product Subarray (leetcode lintcode)

```
Description
Find the contiguous subarray within an array (containing at leas
t one number)
which has the largest product.

Example
For example, given the array [2,3,-2,4],
the contiguous subarray [2,3] has the largest product = 6.
```

解题思路

可参考题目 Maximum Subarray，不过有一点需要注意的是，乘积有正有负，负负得正，所以在本题中除了记录当前的乘积最大值外，还需要记录当前的乘积最小值。

1. 定义状态：定义一维状态变量 `curtMax[i]` 和 `curtMin[i]` ，表示以第 `i` 个数字 `nums[i - 1]` 为结尾的子数组的最大乘积和最小乘积，定义变量 `max` 存储每次循环计算后的最大值。
2. 定义状态转移函数：
   - 对于 `curtMax[i]` ，它应该是 `curtMax[i - 1] * nums[i - 1], curtMin[i - 1] * nums[i - 1], nums[i - 1]` 三者之中的最大值。
   - 对于 `curtMin[i]` ，它应该是 `curtMax[i - 1] * nums[i - 1], curtMin[i - 1] * nums[i - 1], nums[i - 1]` 三者之中的最小值。
   - 每次循环记录当前得到的最大乘积值为 `max` 。
   - 根据状态转移函数，当前状态只取决于上一个状态，所以使用滚动数组进行内存优化。
3. 定义起点：三者的起点都为 `nums[0]` 。
4. 定义终点：最终结果即为 `max` 。

```
public class Solution {
    /**
     * @param nums: an array of integers
     * @return: an integer
     */
    public int maxProduct(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int[] curtMinProduct = new int[2];
        int[] curtMaxProduct = new int[2];
        curtMinProduct[0] = nums[0];
        curtMaxProduct[0] = nums[0];
        int maxProduct = nums[0];

        for (int i = 1; i < nums.length; i++) {
            curtMinProduct[i % 2] = Math.min(nums[i],
                                Math.min(curtMinProduct[(i - 1)
% 2] * nums[i], curtMaxProduct[(i - 1) % 2] * nums[i]));
            curtMaxProduct[i % 2] = Math.max(nums[i],
                                Math.max(curtMinProduct[(i - 1)
% 2] * nums[i], curtMaxProduct[(i - 1) % 2] * nums[i]));
            maxProduct = Math.max(maxProduct, curtMaxProduct[i %
2]);
        }

        return maxProduct;
    }
}
```

Best Time to Buy and Sell Stock

Best Time to Buy and Sell Stock (leetcode lintcode)

```
Description
Say you have an array for which the ith 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 algorit
hm to find the maximum profit.

Example
Given array [3,2,3,1,2], return 1.
```

解题思路

记录当前的最大利润，以及最小买入值。

```java
public class Solution {
    /**
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length < 2) {
            return 0;
        }

        int profit = 0;
        int min = prices[0];
        for (int i = 1; i < prices.length; i++) {
            profit = Math.max(profit, prices[i] - min);
            min = Math.min(min, prices[i]);
        }
        return profit;
    }
}
```

Best Time to Buy and Sell Stock II

Best Time to Buy and Sell Stock II (leetcode lintcode)

```
Description
Say you have an array for which the ith 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).

Example
Given an example [2,1,2,0,1], return 2
```

解题思路

一、动态规划

1. 定义状态：`f[i]` 第 `i` 天卖出股票能获得的最大收益。
2. 定义状态转移函数：如果第 `i` 天卖出，那么可以是第 `j + 1, j < i` 天买入，这样 `f[i]` 对应的上一个状态是 `f[j]` ，`f[i] = MAX{f[j] + prices[i-1] - prices[j]}` 。
3. 定义起点：`f[0] = 0` 。
4. 定义终点：为 `f[n]` 。

Java 实现

```
class Solution {
    /**
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length < 2) {
            return 0;
        }

        int n = prices.length;
        // status
        int[] f = new int[n + 1];
        // initialize
        f[0] = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < i; j++) {
                f[i] = Math.max(f[i], f[j] + prices[i-1] - price
s[j]);
            }
        }

        return f[n];
    }
}
```

二、贪心

在方法一中，会出现较多的重复计算。从另外一个角度考虑，如果可以无限多次的交易，那么只要第二天的价格比第一天的价格高，就可以进行交易。

Java 实现

```java
class Solution {
    /**
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length < 2) {
            return 0;
        }

        int n = prices.length;
        int profit = 0;
        for (int i = 1; i < n; i++) {
            int diff = prices[i] - prices[i - 1];
            if (diff > 0) {
                profit += diff;
            }
        }

        return profit;
    }
}
```

参考

1. Best Time to Buy and Sell Stock II leetcode java | 爱做饭的小莹子

Best Time to Buy and Sell Stock III

Best Time to Buy and Sell Stock III (lintcode)

```
Description
Say you have an array for which the ith 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.

Notice
You may not engage in multiple transactions at the same time
(ie, you must sell the stock before you buy again).

Example
Given an example [4,4,6,1,1,4,2,5], return 6.
```

解题思路

使用隔板法，将数组分为两部分，然后分别求一次交易的最大利润。考虑到题目要求是最多可以进行两次交易，也即可以进行一次交易，所以在求最终结果时，隔板两侧的数组可以有重合。

```java
class Solution {
    /**
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int[] prices) {
        // write your code here
        if (prices == null || prices.length < 2) {
            return 0;
        }

        int n = prices.length;
        int[] left = new int[n];
        int min = prices[0];
        left[0] = 0;
        for (int i = 1; i < n; i++) {
            left[i] = Math.max(left[i-1], prices[i] - min);
            min = Math.min(min, prices[i]);
        }

        int[] right = new int[n];
        int max = prices[n-1];
        right[n-1] = 0;
        for (int i = n-2; i >= 0; i--) {
            right[i] = Math.max(right[i+1], max - prices[i]);
            max = Math.max(max, prices[i]);
        }

        int ans = Integer.MIN_VALUE;
        for (int i = 0; i < n - 1; i++) {
            ans = Math.max(left[i] + right[i], ans);
        }

        return ans;
    }
};
```

Best Time to Buy and Sell Stock IV

Best Time to Buy and Sell Stock IV (leetcode lintcode)

```
Description
Say you have an array for which the ith element is the price of
a given stock on day i.
Design an algorithm to find the maximum profit. You may complete
 at most k transactions.

Notice
You may not engage in multiple transactions at the same time
(i.e., you must sell the stock before you buy again).

Example
Given prices = [4,4,6,1,1,4,2,5], and k = 2, return 6.
```

解题思路

本题使用动态规划求解，关键在于如何定义状态。

一、常规解法

1. 定义状态：定义二维状态变量 `f[i][j]` 为前 `i` 天至多进行 `j` 次交易，能够获得的最大收益。
2. 定义状态转移函数：对于 `f[i][j]` 上一个状态可能是 `f[x][j - 1], 0 < x < i`，第 `j` 次交易的利润为 `profit(x + 1, i)`，那么状态转移函数为 `f[i][j] = max{f[x][j - 1] + profit(x + 1, i)}, 0 < x < i` 。
3. 定义起点：`f[i][0] = 0, f[0][j] = Integer.MIN_VALUE` 。
4. 定义终点：最终结果为 `f[n][k]` 。

算法复杂度

- 时间复杂度：上述状态定义下，需要三重循环 `i:0~n, j:0~k, x:0~(i-1)`，所以是 `O((n^2)*k)` 。
- 空间复杂度：`O(nk)` 。

二、优化时间解法

常规解法中定义的状态可以视为全局变量，可以增加一个变量保存局部变量，这样不必每次都对之前的状态进行 遍历。

1. 定义状态：
   - `mustsell[i][j]` 表示前 `i` 天，至多进行 `j` 次交易，第 `i` 天必须卖出，获得的最大收益。
   - `globalbest[i][j]` 表示前 `i` 天，至多进行 `j` 次交易，第 `i` 天可以不卖出，获得的最大收益。
2. 定义状态转移函数：
   - 定义 `gain` 为第 `i - 1` 天买入，第 `i` 天卖出的收益，`gain = prices[i] - prices[i - 1]` 。
   - `mustsell[i][j] = Max{globalbest[i - 1][j - 1] + gain, mustsell[i - 1][j] + gain}` 。
     - 其中 `globalbest[i - 1][j - 1] + gain = a. mustsell[i - 1][j - 1] + gain; b. mustsell[x][j - 1] + gain, x < i - 1` 。
     - 对于情况 `a` ，相当于 `i - 1` 天卖出，又买入，也就是把卖出拖延到第 `i` 天。所以可以是 `mustsell[i][j]` 的最优解，题目要求是至多交易 `k` 次，符合题意。
     - 对于情况 `b` ，本来其实要 `x` 遍历一遍 `0 ~ i-1` 找到 `mustsell[x][j - 1]` 的最优解。考虑到 `globalbest[i][j]` 已保存之前所有的最优解，所以时间得到优化。
   - `globalbest[i][j] = Max{globalbest[i - 1][j], mustsell[i][j]}` 。
3. 定义起点：`mustsell[0][i] = globalbest[0][i] = 0` 。
4. 定义终点：`globalbest[n - 1][k]` 。

Java 实现

```
class Solution {
    /**
     * @param k: An integer
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int k, int[] prices) {
```

```
        if (prices == null || prices.length == 0 || k == 0) {
            return 0;
        }

        if (k >= prices.length / 2) {
            int profit = 0;
            for (int i = 1; i < prices.length; i++) {
                if (prices[i] > prices[i - 1]) {
                    profit += prices[i] - prices[i - 1];
                }
            }
            return profit;
        }

        int n = prices.length;
        // mustsell[i][j] 表示前i天，至多进行j次交易，第i天必须sell的
最大获益
        int[][] mustsell = new int[n + 1][k + 1];
        // globalbest[i][j] 表示前i天，至多进行j次交易，第i天可以不sel
l的最大获益
        int[][] globalbest = new int[n + 1][k + 1];

        mustsell[0][0] = globalbest[0][0] = 0;
        for (int i = 1; i <= k; i++) {
            mustsell[0][i] = globalbest[0][i] = 0;
        }

        for (int i = 1; i < n; i++) {
            int gain = prices[i] - prices[i - 1];
            mustsell[i][0] = 0;
            for (int j = 1; j <= k; j++) {
                mustsell[i][j] = Math.max(globalbest[i - 1][j - 1
] + gain,
                                          mustsell[i - 1][j] +
 gain);
                globalbest[i][j] = Math.max(globalbest[i - 1][j]
, mustsell[i][j]);
            }
        }
        return globalbest[n - 1][k];
```

```
    }
};
```

参考

1. Best Time to Buy and Sell Stock IV | 九章算法

Longest Increasing Continuous subsequence II

Longest Increasing Continuous subsequence II (lintcode)

```
Description
Give you an integer matrix (with row size n, column size m)，
find the longest increasing continuous subsequence in this matri
x.
(The definition of the longest increasing continuous subsequence
 here
can start at any row or column and go up/down/right/left any dir
ection).

Example
Given a matrix:
[
  [1 ,2 ,3 ,4 ,5],
  [16,17,24,23,6],
  [15,18,25,22,7],
  [14,19,20,21,8],
  [13,12,11,10,9]
]
return 25

Challenge
O(nm) time and memory.
```

解题思路

本题与滑雪问题是同一类问题，考虑到以下几个因素，需要使用记忆化搜索解题。

1. 初始化不容易找到。
2. 状态转移比较麻烦，没有清晰的顺序性。

`dp[i][j]` 表示从 `A[i][j]` 出发向上下左右四个方向寻找，得到的最长连续子序列的长度。

`flag[i][j]` 来标记 `dp[i][j]` 求解状态，为 `1` 时表示已完成求解，直接返回 `dp[i][j]`。为 `-1` 时则为了防止重复寻找。

Java 实现

```
// write your code here
public class Solution {
    /**
     * @param A an integer matrix
     * @return an integer
     */

    public int longestIncreasingContinuousSubsequenceII(int[][]
A) {
        if (A == null || A.length == 0) {
            return 0;
        }
        int n = A.length;
        int m = A[0].length;
        int ans = 0;
        int[][] dp = new int[n][m];
        int[][] flag = new int[n][m];

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                dp[i][j] = dfs(A, dp, flag, i, j, n, m);
                ans = Math.max(ans, dp[i][j]);
            }
        }
        return ans;
    }

    int[] dx = {1, -1, 0, 0};
    int[] dy = {0, 0, 1, -1};

    private int dfs(int[][] A, int[][] dp, int[][] flag, int i,
int j, int n, int m) {
```

```
        if (flag[i][j] == 1) {
            return dp[i][1];
        }

        int ans = 1;
        flag[i][j] = -1;
        for (int k = 0; k < 4; k++) {
            int id = i + dx[k];
            int jd = j + dy[k];
            if (0 <= id && id < n && 0 <= jd && jd < n) {
                if (flag[id][jd] != -1 && A[i][j] < A[id][jd]) {
                    ans = Math.max(ans, dfs(A, dp, flag, id, jd,
 n, m) + 1);
                }
            }
        }
        flag[i][j] = 1;
        dp[j][j] = ans;
        return dp[i][j];
    }
}
```

参考

1. Longest Increasing Continuous subsequence II.java

Coins in a Line

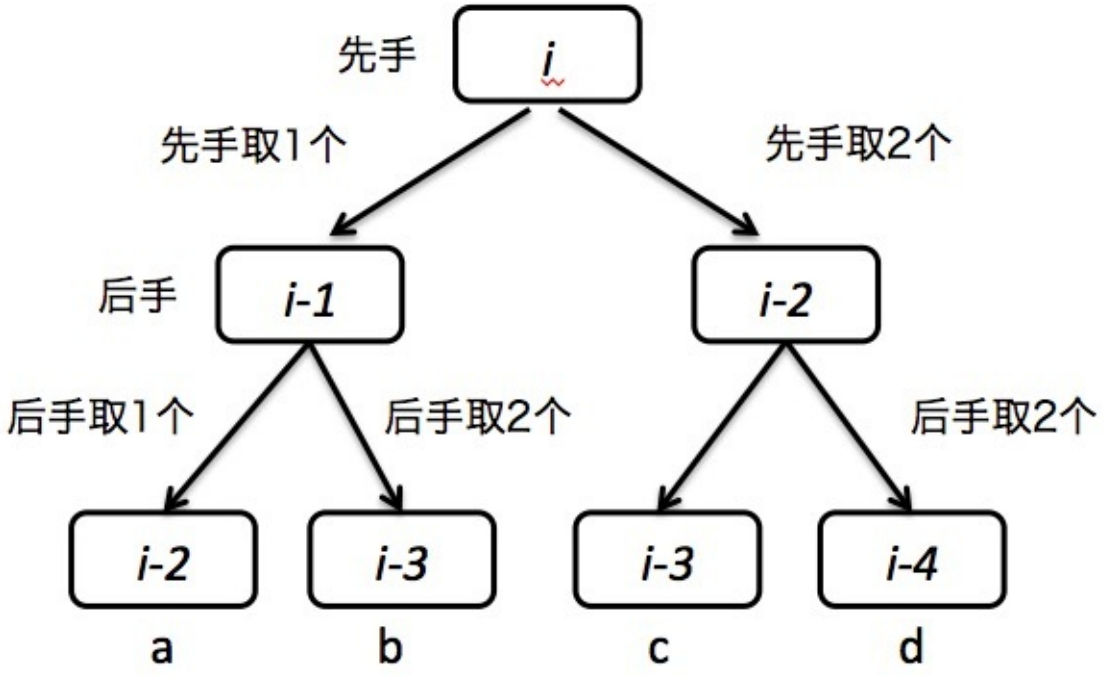
Coins in a Line (leetcode lintcode)

```
Description
There are n coins in a line.
Two players take turns to take one or two coins from right side
until there are no more coins left.
The player who take the last coin wins.

Could you please decide the first play will win or lose?

Example
n = 1, return true.
n = 2, return true.
n = 3, return false.
n = 4, return true.
n = 5, return true.

Challenge
O(n) time and O(1) memory
```

解题思路

一、动态规划（记忆化搜索）

1. 定义状态： `f[i]` 为现在还剩 `i` 个硬币，先手取硬币最终的输赢状况。
2. 定义状态转移函数： 可参考以下图示，在还剩 `i` 个硬币的时候，先手有两种取法，取 1 个或者取 2 个硬币，在这两种情况，后手也分别有两种取法，取 1 个或者取 2 个，所以在轮到先手下一次取硬币时，一共有四种情况，只有当情况 a 和 b 一定能赢，或者 c 和 d 一定能赢时，先手才能获得胜利，对应状态转移为 `f[i] = (f[n - 2] && f[n - 3]) || (f[n - 3]&&f[n - 4])` 。
3. 定义起点：初始化状态转移函数无法涉及的状态 `f[1] = true, f[2] = true, f[3] = false, f[4] = true` 。
4. 定义终点：为 `f[n]` 。



Java 实现

```
public class Solution {
    /**
     * @param n: an integer
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int n) {
        int[] dp = new int[n + 1];

        return MemorySearch(n, dp);
    }

    private boolean MemorySearch (int n, int[] dp) {  // 0 is em
pty, 1 is false, 2 is true
        if (dp[n] != 0) {
            if (dp[n] == 1) {
                return false;
            } else {
                return true;
            }
        }
```

```
        if (n <= 0) {
            dp[n] = 1;
        } else if (n == 1) {
            dp[n] = 2;
        } else if (n == 2) {
            dp[n] = 2;
        } else if (n == 3) {
            dp[n] = 1;
        } else {
            if ((MemorySearch(n - 2, dp) && MemorySearch(n - 3,
dp)) ||
                (MemorySearch(n - 3, dp) && MemorySearch(n - 4,
dp) )) {
                dp[n] = 2;
            } else {
                dp[n] = 1;
            }
        }

        if (dp[n] == 2) {
            return true;
        }
        return false;
    }
}
```

二、数学推理

可以通过观察找规律，只有 `n` 为 3 的倍数时，先手会输。

```
public class Solution {
    /**
     * @param n: an integer
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int n) {
        if (n <= 0) {
            return false;
        }

        if (n % 3 == 0) {
            return false;
        }
        return true;
    }

}
```

参考

1. Coins in a Line | 九章算法
2. Coins in a Line | lintcode题解

Coins in a Line II

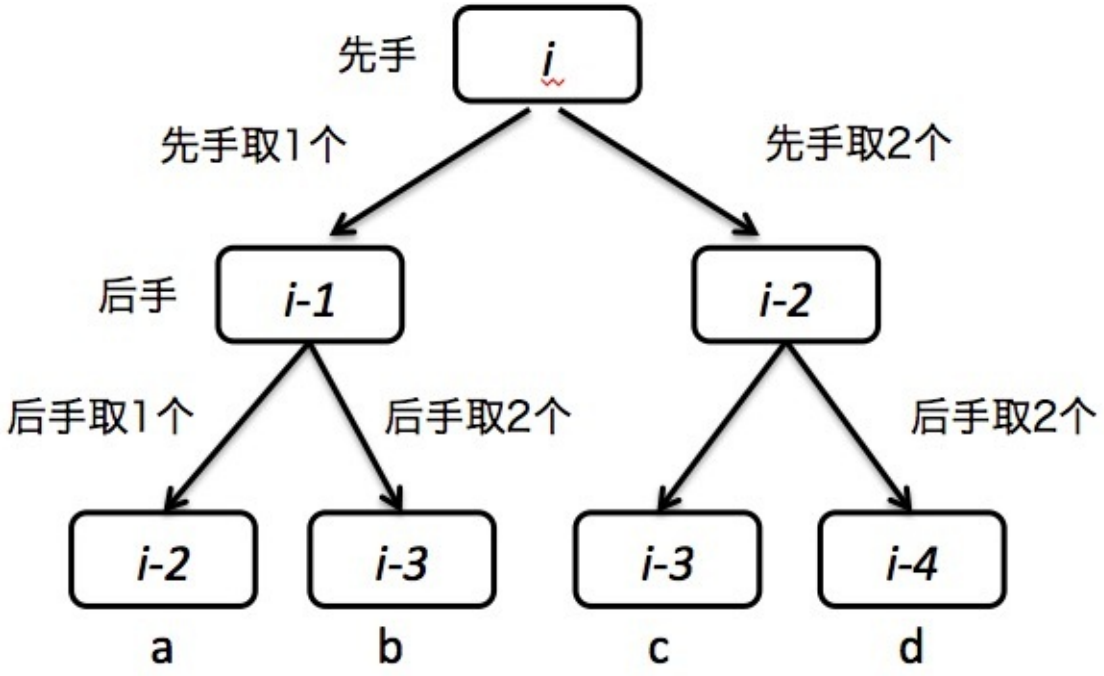
Coins in a Line II (leetcode lintcode)

```
Description
There are n coins with different value in a line.
Two players take turns to take one or two coins from left side u
ntil there are no more coins left.
The player who take the coins with the most value wins.

Could you please decide the first player will win or lose?
```

解题思路

一、记忆化搜索

1. 定义状态： `f[i]` 为现在还剩 `i` 个硬币，先手取硬币最终可得的最大价值。
2. 定义状态转移函数：依旧参考图示，在先手取过硬币之后，后手的策略一定是让先手取得尽可能少的硬币值，所以对应的，先手下一次取硬币面临的状态一定是 `a` 和 `b` 之间的最小值，或 `c` 和 `d` 之间的最小值。此时回过头来考虑先手在剩 `i` 个硬币时，选取策略是最大化硬币价值。所以对应状态转移方程为 `f[i] = Max{Min{f[i - 2], f[i - 3]} + values[n - i], Min{f[i - 3], f[i - 4]} + values[n - i] + values[n - i + 1]}` 。
3. 定义起点：初始化状态转移函数无法涉及的状态 `f[1], f[2], f[3], f[4]` 。
4. 定义终点：为 `f[n]` ，当先手取得的最大值比所有硬币总价值的一半大时胜利。



具体实现过程使用了记忆化搜索的方法。

Java 实现

```
public class Solution {
    /**
     * @param values: an array of integers
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int[] values) {
        int[] dp = new int[values.length + 1];
        boolean[] flag = new boolean[values.length + 1];

        int sum = 0;
        for (int now : values) {
            sum += now;
        }
        return sum < 2 * MemorySearch(values.length, dp, flag, v
alues);
    }

    private int MemorySearch (int n, int[] dp, boolean[] flag, i
nt[] values) {
        if (flag[n] == true) {
```

```
            return dp[n];
        }
        flag[n] = true;
        if (n == 0) {
            dp[n] = 0;
        } else if (n == 1) {
            dp[n] = values[values.length - 1];
        } else if (n == 2) {
            dp[n] = values[values.length - 1] + values[values.le
ngth - 2];
        } else if (n == 3) {
            dp[n] = values[values.length - 2] + values[values.le
ngth - 3];
        } else {
            dp[n] = Math.max(
                Math.min(MemorySearch(n-2, dp, flag, values), Me
morySearch(n-3, dp, flag, values)) + values[values.length-n],
                Math.min(MemorySearch(n-3, dp, flag, values), Me
morySearch(n-4, dp, flag, values)) + values[values.length-n] + v
alues[values.length - n + 1]
                );
        }

        return dp[n];
    }
}
```

二、动规

记忆化搜索实现起来比较麻烦，在网上看到另一种解法要简便很多。关键点在于状态的定义。

1. 定义状态：`f[i]` 表示先手从第 `i` 个硬币开始取，直到硬币取完，可得的最大价值。
2. 定义状态转移函数：当先手开始准备取第 `i` 个硬币时，有两种方法
   - 取一个硬币 `values[i]`。那么对手接着可以取一个 `values[i + 1]` 或者两个 `values[i + 1] + values[i + 2]`，那么先手下一次取硬币面临的是 `f[i + 2]` 或 `f[i + 3]`，而且是两者之中的最小值。
   - 取两个硬币 `values[i] + values[i + 1]`。同理对手接着可以取一个

或两个，先手下一次取硬币面临的是 `f[i + 3]` 或 `f[i + 4]` 中的最小值。
    - 所以对应状态转移方程为 `f[i] = Max{Min{f[i + 2], f[i + 3]} + values[i], Min{f[i + 3], f[i + 4]} + values[i] + values[i + 1]}` 。
3. 定义起点：初始化状态转移函数无法涉及的状态 `f[n], f[n - 1], f[n - 2], f[n - 3]` 。
4. 定义终点：为 `f[0]` ，当先手取得价值比后手多时胜利。

Java 实现

```
public class Solution {
    /**
     * @param values: an array of integers
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int[] values) {
        int len = values.length;
        if (len < 3) {
            return true;
        }
        // status
        int[] f = new int[len + 1];
        // initialize
        f[len] = 0;
        f[len - 1] = values[len - 1];
        f[len - 2] = values[len - 1] + values[len - 2];
        f[len - 3] = values[len - 2] + values[len - 3];
        for (int i = len - 4; i >= 0; i--) {
            f[i] = Math.max(
                Math.min(f[i + 2], f[i + 3]) + values[i],
                Math.min(f[i + 3], f[i + 4]) + values[i] + value
s[i + 1]
                );
        }

        int sum = 0;
        for (int i : values) {
            sum += i;
        }

        return f[0] > sum - f[0];
    }

}
```

参考

1. Coins in a Line II | 九章算法
2. lintcode ：Coins in Line II 硬币排成线 II | 水滴失船
3. Coins in a Line II | lintcode题解

Coins in a Line III

Coins in a Line III (lintcode)

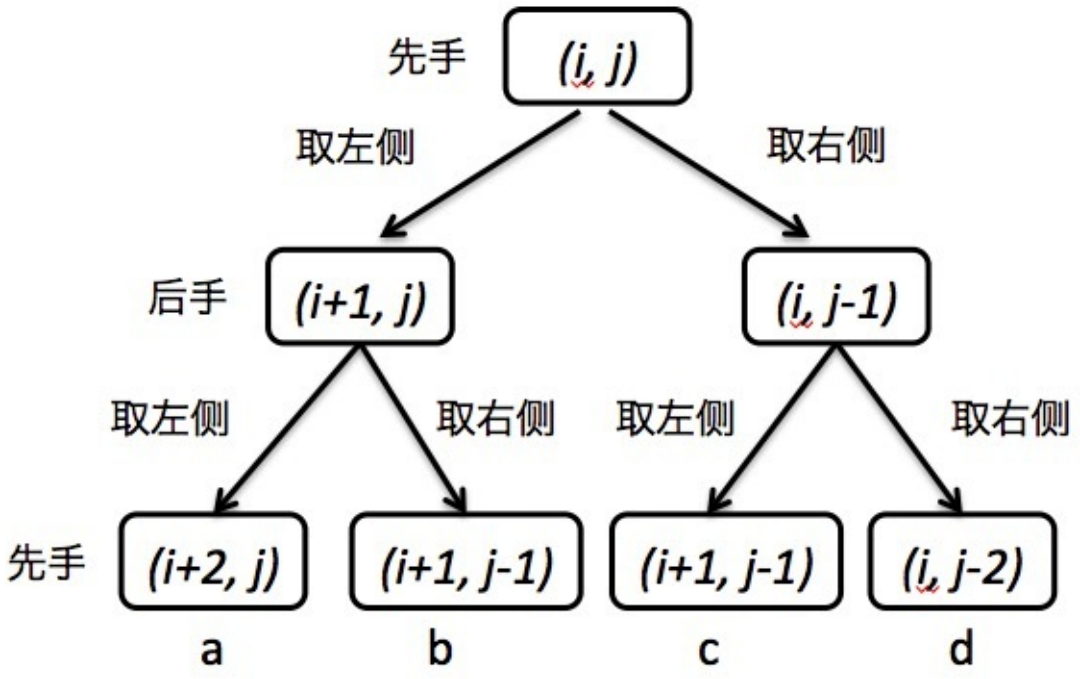
```
Description
There are n coins in a line.
Two players take turns to take a coin from one of the ends of th
e line
until there are no more coins left.
The player with the larger amount of money wins.
Could you please decide the first player will win or lose?

Example
Given array A = [3,2,2], return true.
Given array A = [1,2,4], return true.
Given array A = [1,20,4], return false.

Challenge
Follow Up Question:
If n is even.
Is there any hacky algorithm that can decide
whether first player will win or lose in O(1) memory and O(n) ti
me?
```

解题思路

使用动态规划（记忆化搜索）分析。

1. 定义状态：考虑到可以从两端选取硬币，所以要使用二维状态变量，以区分选取方向。定义 `f[i][j]` 为现在还剩第 `i` 个到第 `j` 个硬币，先手取硬币最终可得的最大价值。
2. 定义状态转移函数： 参考图示，剩余硬币为第 `i` 个到第 `j` 个时，先手有两种取法，
   - 取左侧硬币 `A[i]` 。后手对应两种取法，这之后先手面临的是 `f[i + 2][j]` 和 `f[i + 1][j - 1]` 中的最小值。
   - 取右侧硬币 `A[j]` 。后手对应两种取法，这之后先手面临的是 `f[i +`

`1][j - 1]` 和 `f[i][j - 2]` 中的最小值。
    - 所以对应状态转移方程为 `f[i][j] = Max{Min{f[i + 2][j], f[i + 1][j - 1]} + A[i], Min{f[i + 1][j - 1], f[i][j - 2]} + A[j]}` 。
3. 定义起点：初始化状态转移函数无法涉及的状态 `f[i, i] = A[i], f[i, i + 1] = Max{A[i], A[i + 1]}` 。
4. 定义终点：为 `f[0][n - 1]` 。



Java 实现

```
public class Solution {
    /**
     * @param values: an array of integers
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin (int[] values) {
        int n = values.length;
        int[][] f = new int[n + 1][n + 1];
        boolean[][] flag = new boolean[n + 1][n + 1];

        int sum = 0;
        for (int now : values) {
```

```
            sum += now;
        }

        return sum < 2 * MemorySearch(0, values.length - 1, f, f
lag, values);
    }

    private int MemorySearch (int left, int right, int[][] f, bo
olean[][] flag, int[] values) {
        if (flag[left][right]) {
            return f[left][right];
        }
        flag[left][right] = true;
        if (left > right) {
            f[left][right] = 0;
        } else if (left == right) {
            f[left][right] = values[left];
        } else if (left + 1 == right) {
            f[left][right] = Math.max(values[left], values[right
]);
        } else {
            int pick_left = Math.min(
                                MemorySearch(left + 2, right, f, fla
g, values),
                                MemorySearch(left + 1, right - 1, f,
 flag, values) + values[left]
                                );
            int pick_right = Math.min(
                                MemorySearch(left + 1, right - 1, f,
 flag, values),
                                MemorySearch(left, right - 2, f, fla
g, values) + values[right]
                                );
            f[left][right] = Math.max(pick_left, pick_right);
        }

        return f[left][right];
    }
}
```

Stone Game

Stone Game (lintcode)

```
Description
There is a stone game.
At the beginning of the game the player picks n piles of stones
in a line.
The goal is to merge the stones in one pile observing the follow
ing rules:
1. At each step of the game,the player can merge two adjacent pi
les to a new pile.
2. The score is the number of stones in the new pile.
You are to determine the minimum of the total score.

Example
For [4, 1, 1, 4], in the best solution, the total score is 18:
1. Merge second and third piles => [4, 2, 4], score +2
2. Merge the first two piles => [6, 4]，score +6
3. Merge the last two piles => [10], score +10

Other two examples:
[1, 1, 1, 1] return 8
[4, 4, 5, 9] return 43
```

解题思路

一、记忆化搜索

1. 定义状态：`f[i][j]` 表示把第 `i` 个到第 `j` 个石头合并在一起的最小分数值。
2. 定义状态转移函数：对 `f[i][j]` ，可以有任意划分，得到 `f[i][j] = f[i][k] + f[k+1][j] + sum[i][j], i<=k<j` ，只需遍历所有取值 k 求最小值即可。其中 `sum[i][j]` 表示第 `i` 个到第 `j` 个石头的总分值。
3. 定义起点：初始化状态转移函数无法涉及的状态 `f[i][i] = 0, 0 <= i < n}` 。

4. 定义终点：为 `f[0][n - 1]` 。

Java 实现

```
public class Solution {
    /**
     * @param A an integer array
     * @return an integer
     */
    public static int stoneGame(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

    int n = A.length;
    // status
    int[][] f = new int[n][n];
    int[][] visit = new int[n][n];

    // initialize
    for (int i = 0; i < n; i++) {
        f[i][i] = 0;
    }

    int[][] sum = new int[n][n];
    for (int i = 0; i < n; i++) {
        sum[i][i] = A[i];
        for (int j = i + 1; j < n; j++) {
            sum[i][j] = sum[i][j - 1] + A[j];
        }
    }

    return search(0, n - 1, f, visit, sum);
    }

    private static int search (int l, int r, int[][] f, int[][]
visit, int[][] sum) {
        if (visit[l][r] == 1) {
            return f[l][r];
        }
```

```
        if (l == r) {
            visit[l][r] = 1;
            return f[l][r];
        }

        f[l][r] = Integer.MAX_VALUE;
        for (int k = l; k < r; k++) {
            f[l][r] = Math.min(
                        f[l][r],
                        search(l, k, f, visit, sum) + search(k+1
, r, f, visit, sum) + sum[l][r]
                        );
        }
        visit[l][r] = 1;
        return f[l][r];
    }
}
```

二、动态规划

Java 实现

```
public class Solution {
    /**
     * @param A an integer array
     * @return an integer
     */
    public static int stoneGame(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        int n = A.length;
        // status
        int[][] f = new int[n][n];

        // initialize
        for (int i = 0; i < n; i++) {
```

```
            f[i][i] = 0;
        }

        int[][] sum = new int[n][n];
        for (int i = 0; i < n; i++) {
            sum[i][i] = A[i];
            for (int j = i + 1; j < n; j++) {
                sum[i][j] = sum[i][j - 1] + A[j];
            }
        }

        for (int delta = 1; delta < n; delta++) {
            for (int i = 0; i + delta < n; i++) {
                int j = i + delta;
                f[i][j] = Integer.MAX_VALUE;
                for (int k = i; k < j; k++) {
                    f[i][j] = Math.min(f[i][j], f[i][k] + f[k + 1
][j] + sum[i][j]);
                }
            }
        }

        return f[0][n - 1];
    }
}
```

参考

1. Stone Game 476 | LintCode题解

Scramble String

Scramble String (lintcode)

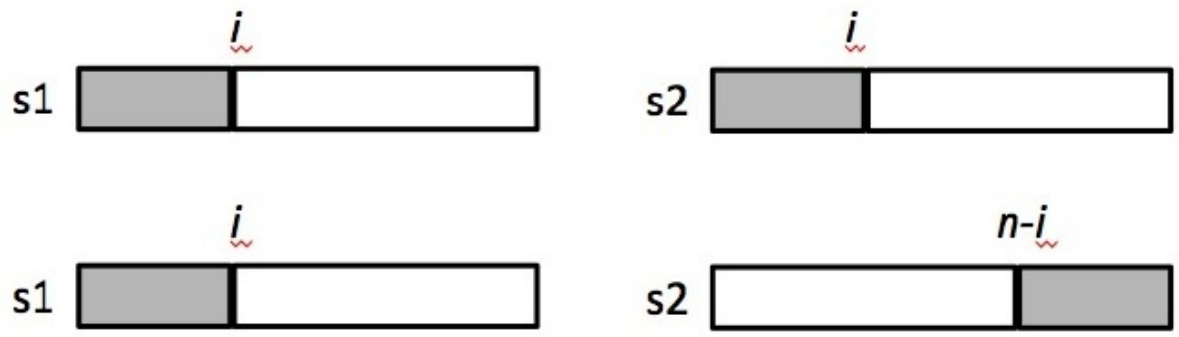
```
Description
Given a string s1,
we may represent it as a binary tree by partitioning it to two n
on-empty substrings recursively.

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 childre
n,
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" an
d "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.

Challenge
O(n3) time
```

解题思路

一、搜索

如图所示，分别对两个字符串进行搜索遍历对比。注意此处 `s2` 的划分方式。



Java 实现

```java
public class Solution {
    /**
     * @param s1 A string
     * @param s2 Another string
     * @return whether s2 is a scrambled string of s1
     */
    public boolean isScramble(String s1, String s2) {
        if (s1.length() != s2.length()) {
            return false;
```

```
        }

        // s1.length() == s2.length()
        if (s1.length() == 0 || s1.equals(s2)) {
            return true;
        }

        if (!isValid(s1, s2)) {
            return false;
        } // base case

        for (int i = 1; i < s1.length(); i++) {
            String s11 = s1.substring(0, i);
            String s12 = s1.substring(i, s1.length());
            String s21 = s2.substring(0, i);
            String s22 = s2.substring(i, s2.length());
            String s23 = s2.substring(0, s2.length() - i);
            String s24 = s2.substring(s2.length() - i, s2.length
());

            if (isScramble(s11, s21) && isScramble(s12, s22)) {
                return true;
            }

            if (isScramble(s11, s24) && isScramble(s12, s23)) {
                return true;
            }
        }

        return false;
    }

    private boolean isValid (String s1, String s2) {
        char[] arr1 = s1.toCharArray();
        char[] arr2 = s2.toCharArray();
        Arrays.sort(arr1);
        Arrays.sort(arr2);
        if (!(new String(arr1)).equals(new String(arr2))) {
            return false;
        }
```

```
            return true;
        }
    }
```

二、记忆化搜索

Java 实现

```
public class Solution {
    /**
     * @param s1 A string
     * @param s2 Another string
     * @return whether s2 is a scrambled string of s1
     */
    public boolean isScramble(String s1, String s2) {
        int len = s1.length();
        int[][][] visit = new int[len][len][len + 1];
        return checkScramble(s1, 0, s2, 0, len, visit);
    }

    private boolean checkScramble (String s1, int start1,
                                   String s2, int start2,
                                   int k, int[][][] visit) {
        if (visit[start1][start2][k] == 1) {
            return true;
        }
        if (visit[start1][start2][k] == -1) {
            return false;
        }

        if (s1.length() != s2.length()) {
            visit[start1][start2][k] = -1;
            return false;
        }
        if (s1.length() == 0 || s1.equals(s2)) {
            return true;
        }
        if (!isValid(s1, s2)) {
            visit[start1][start2][k] = -1;
```

```
            return false;
        }

        for (int i = 1; i < s1.length(); i++) {
            String s11 = s1.substring(0, i);
            String s12 = s1.substring(i, s1.length());
            String s21 = s2.substring(0, i);
            String s22 = s2.substring(i, s2.length());
            String s23 = s2.substring(0, s2.length() - i);
            String s24 = s2.substring(s2.length() - i, s2.length
());

            if (checkScramble(s11, start1, s21, start2, i, visit
) &&
                checkScramble(s12, start1+i, s22, start2+i, k-i,
 visit)) {
                visit[start1][start2][k] = 1;
                return true;
            }

            if (checkScramble(s11, start1, s24, start2+k-i, i, v
isit) &&
                checkScramble(s12, start1+i, s23, start2, k-i, v
isit)) {
                return true;
            }
        }
        visit[start1][start2][k] = -1;
        return false;
    }

    private boolean isValid (String s1, String s2) {
        char[] arr1 = s1.toCharArray();
        char[] arr2 = s2.toCharArray();
        Arrays.sort(arr1);
        Arrays.sort(arr2);
        if (!(new String(arr1)).equals(new String(arr2))) {
            return false;
        }
        return true;
```

```
        }
    }
```

参考

1. Scramble String | 九章算法

Min Stack

Min Stack (leetcode lintcode)

```
Description
Implement a stack with min() function, which will return the sma
llest number in the stack.
It should support push, pop and min operation all in O(1) cost.

Notice
min operation will never be called if there is no number in the
stack.

Example
push(1)
pop()   // return 1
push(2)
push(3)
min()   // return 2
push(1)
min()   // return 1
```

解题思路

题目要求是 `O(1)` 时间的 `min` 操作，对空间复杂度则没有要求，考虑使用两个栈，其中一个栈用作正常用处，另一个栈则同步存储对应的最小值。

注意和栈相关的操作：

> 1. 判断是否相等：`stack.empty()`。
> 2. 查看栈顶元素：`stack.peek()`。
> 3. 判断栈顶元素是否相等：`equals()`。

易错点

> 1. 注意考察存储最小值的栈为空的时候。

Java 实现

```
public class MinStack {

    private Stack<Integer> nums;
    private Stack<Integer> mins;

    public MinStack() {
        // do initialize if necessary
        nums = new Stack<Integer>();
        mins = new Stack<Integer>();
    }

    public void push(int number) {
        // write your code here
        nums.push(number);
        if (mins.isEmpty()) {
            mins.push(number);
        } else {
            mins.push(Math.min(number, mins.peek()));
        }
    }

    public int pop() {
        // write your code here
        mins.pop();
        return nums.pop();
    }

    public int min() {
        // write your code here
        return mins.peek();
    }
}
```

优化解法：

只有在进栈元素小于等于当前最小值时，才将其加入最小值栈；进栈元素大于当前最大值时，最小值栈无需操作。出栈时需要做同样的判断。

其中，进栈元素等于最小值时，为避免出栈时出现问题，故需要对最小值栈进行操作。

Java 实现

```
public class MinStack {

    private Stack<Integer> nums;
    private Stack<Integer> mins;

    public MinStack() {
        // do initialize if necessary
        nums = new Stack<Integer>();
        mins = new Stack<Integer>();
    }

    public void push(int number) {
        // write your code here
        nums.push(number);
        if (mins.isEmpty()) {
            mins.push(number);
        } else if (number <= mins.peek()) {
            mins.push(number);
        }
    }

    public int pop() {
        // write your code here
        // attention for diffrence between .equals() and ==
        // also attention for .peek() and pop()
        if (nums.peek().equals(mins.peek())) {
            mins.pop();
        }
        return nums.pop();
    }

    public int min() {
        // write your code here
        return mins.peek();
    }
}
```

参考

1. Min Stack | 九章算法

Implement Queue by Two Stacks

Implement Queue by Two Stacks (leetcode lintcode)

```
Description
As the title described, you should only use two stacks to implem
ent a queue's actions.
The queue should support push(element), pop() and top()
where pop is pop the first(a.k.a front) element in the queue.
Both pop and top methods should return the value of first elemen
t.

Example
push(1)
pop()     // return 1
push(2)
push(3)
top()     // return 2
pop()     // return 2

Challenge
implement it by two stacks, do not use any other data structure
and push,
pop and top should be O(1) by AVERAGE.
```

解题思路

经典题目，使用两个栈实现队列。进入队列时将元素压入 `stack1` ，出队列时从 `stack2` 取出，其中 `stack2` 中的元素是将 `stack1` 依次放入，所以取出时的顺序符合先进先出。

Java 实现

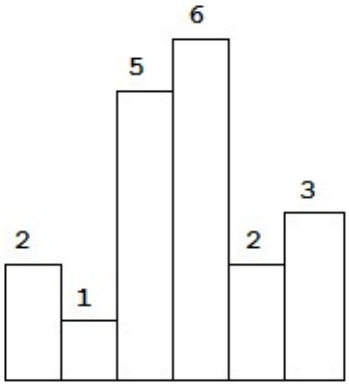
```
public class Queue {
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

    public Queue() {
       // do initialization if necessary
       stack1 = new Stack<Integer>();
       stack2 = new Stack<Integer>();
    }

    public void push(int element) {
        // write your code here
        stack1.push(element);
    }

    public int pop() {
        // write your code here
        if (stack2.isEmpty()) {
            stack2Tostack1();
        }
        return stack2.pop();
    }

    public int top() {
        // write your code here
        if (stack2.isEmpty()) {
            stack2Tostack1();
        }
        return stack2.peek();
    }

    public void stack2Tostack1 () {
        while (!stack1.isEmpty()) {
            stack2.push(stack1.pop());
        }
    }
}
```

参考

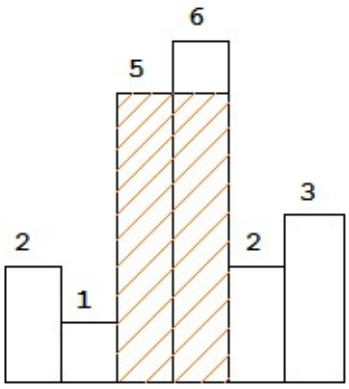
Largest Rectangle in Histogram

Largest Rectangle in Histogram (leetcode lintcode)

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




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



```
The largest rectangle is shown in the shaded area, which has are
a = 10 unit.

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

解题思路

一、遍历法

遍历所有可能的起点、终点，及对应宽度内高度的最小值，得到所有可能取得的矩形面积值，取最大值。

算法复杂度

- 时间复杂度：由于双重循环，所以复杂度为 `O(n^2)` 。

Java 实现

```
public class Solution {
    /**
     * @param height: A list of integer
     * @return: The area of largest rectangle in the histogram
     */
    public int largestRectangleArea(int[] height) {
        // write your code here
        if (height == null || height.length == 0) {
            return 0;
        }
        int n = height.length;
        int max = 0;
        for (int i = 0; i < n; i++) {
            int H = height[i];
            int W = 0;
            for (int j = i; j < n; j++) {
                H = Math.min(H, height[j]);
                W = i - j + 1;
                max = Math.max(max, H * W);
            }
        }

        return max;
    }
}
```

二、递增栈

思路：

对于每一个高度，只要知道它左边第一个比它小的数，它右边第一个比它小的数，那么就能求出这个高度对应的最大矩形的面积。这种情况需要使用辅助栈。

实现方法：

使用辅助栈，存储当前数组元素序号，每次比较栈顶值与当前元素的大小，如果当前值大于栈顶值，入栈。如果栈顶值小于当前值，弹出栈顶值，并计算栈顶值高度对应的面积。

由于栈顶值一定比栈中相邻值大，所以栈顶值高度为 `height(stack.pop())` ，而宽度为 `i - stack.peek() - 1` ，之所以要 `-1` 是因为栈顶值已弹出。当所有元素都已完成入栈，并且全部完成出栈，那么最后一个出栈的元素值对应的长度是直方图长度 `height.length` 。因为这是所有高度中的最小值，所以对应矩形宽度为直方图的宽度。

算法复杂度

- 时间复杂度：一次遍历，为 `O(n)` 。

Java 实现

```
public class Solution {
    /**
     * @param height: A list of integer
     * @return: The area of largest rectangle in the histogram
     */
    public int largestRectangleArea(int[] height) {
        // write your code here
        if (height == null || height.length == 0) {
            return 0;
        }

        Stack<Integer> stack = new Stack<Integer>();
        int max = 0;
        // when i == height.length. all valid i has been pushed
into the stack
        // deal with the values in stack
        for (int i = 0; i <= height.length; i++) {
            int curt = (i == height.length) ? -1 : height[i];
            while (!stack.isEmpty() && curt <= height[stack.peek
()]) {
                int H = height[stack.pop()];
                int W = stack.isEmpty() ? i : i - stack.peek() -
1;
                max = Math.max(max, H * W);
            }
            stack.push(i);
        }

        return max;
    }
}
```

参考

1. Largest Rectangle in Histogram 解题报告 | 水中的鱼
2. LeetCode: Largest Rectangle in Histogram 解题报告 | Yu's graden
3. Largest Rectangle in Histogram | 九章算法

Max Tree

Max Tree (leetcode lintcode)

```
Given an integer array with no duplicates. A max tree building o
n this array is defined as follow:
- The root is the maximum number in the array.
- The left subtree and right subtree are the max trees of the su
barray divided by the root number.
- Construct the max tree by the given array.

Example
Given [2, 5, 6, 0, 3, 1], the max tree constructed by this array
 is:
    6
   / \
  5   3
 /   / \
2   0   1

Challenge
O(n) time and memory.
```

解题思路

如果是自顶向下，按照 Max Tree 的定义构造，那么时间复杂度至少是 `O(nlogn)` 。查找最大值的时间复杂度是 `O(n)` ，如果最大值刚好可以将数组分为两部分，那么复杂度递归关系如下 `T(n) = 2 * T(n / 2) + O(n)` 。最坏的情况是数组是降序/升序，时间复杂度为 `O(n^2)` 。

考虑自底向上的方法。对一个数，考察其父亲结点是谁，它是左儿子还是右儿子。对于数 `i` ，寻找左边第一个比它大的数 `x` ，和右边第一个比它大的数 `y` ，如果 `x > y` 那么 `i` 是 `y` 的左儿子，否则是 `i` 是 `x` 的右儿子。可以用反证法证明。

具体实现使用一个降序栈。

- 将数组按从左到右顺序迭代，当处理一个新的结点 `curt` 时，所有在栈中的结点全部都在其左边，因此需要判断 `curt` 和栈中结点的关系（是 `curt` 的左儿子或者左父亲）。
- 当栈顶结点值大于当前结点值时，将当前结点设为栈顶结点的右儿子，进栈；当栈顶结点值小于当前结点值时，出栈，将其设置为当前结点的左儿子。
- 重复以上步骤，并返回栈底元素，即为最大数（根结点）。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param A: Given an integer array with no duplicates.
     * @return: The root of max tree.
     */
    public TreeNode maxTree(int[] A) {
        if (A == null || A.length == 0) {
            return null;
        }
        LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
        for (int i = 0; i < A.length; i++) {
            TreeNode curt = new TreeNode(A[i]);
            // use while to find the max left node of curt
            while (!stack.isEmpty() && curt.val > stack.peek().v
al) {
                curt.left = stack.pop();
            }
            if (!stack.isEmpty()) {
                stack.peek().right = curt;
```

```
            }
            stack.push(curt);
        }
        TreeNode result = new TreeNode(0);
        while (!stack.isEmpty()) {
            result = stack.pop();
        }
        return result;
    }
}
```

参考

1. Max Tree – Lintcode Java | WELKIN LAN

Rehashing

Rehashing (leetcode lintcode)

```
Description
The size of the hash table is not determinate at the very beginn
ing.
If the total size of keys is too large (e.g. size >= capacity /
10),
we should double the size of the hash table and rehash every key
s.
Say you have a hash table looks like below:

size=3, capacity=4
[null, 21, 14, null]
       ↓    ↓
       9   null
       ↓
      null

The hash function is:
int hashcode(int key, int capacity) {
    return key % capacity;
}

here we have three numbers, 9, 14 and 21,
where 21 and 9 share the same position as they all have the same
 hashcode 1 (21 % 4 = 9 % 4 = 1).
We store them in the hash table by linked list.

rehashing this hash table, double the capacity, you will get:
size=3, capacity=8

index:   0    1    2    3     4    5    6   7
hash : [null, 9, null, null, null, 21, 14, null]
Given the original hash table, return the new hash table after r
ehashing .
```

```
Notice

For negative integer in hash table, the position can be calculat
ed as follow:

- C++/Java: if you directly calculate -4 % 3 you will get -1.
  You can use function: a % b = (a % b + b) % b to make it is a
non negative integer.
- Python: you can directly use -1 % 3, you will get 2 automatica
lly.

Example
Given [null, 21->9->null, 14->null, null],
return [null, 9->null, null, null, null, 21->null, 14->null, nul
l]
```

解题思路

依次从原 hash 表中取出结点，重新进行 hash 映射，添加到扩展后的 hash 表即可。需要注意，如何向链表中添加结点，这里要分两种情况，当链表为空时，当链表不为空时。此外链表结点的赋值操作也要注意： `newhashTable[j] = new ListNode(hashTable[i].val);` 。

易错点：

1. 向链表数组中的一个链表 `result[j]` 中插入结点时，当链表为空时，插入操作需要使用 `result[j] = new ListNode(value)` ；如果先赋值 `ListNode tmp = result[j]` ，再添加结点 `tmp = new ListNode(value)` ，无法正常添加结点。

   当链表非空时，则需要先赋值 `ListNode tmp = result[j]` ，在 `tmp.next == null` 时再添加结点 `tmp.next = new ListNode(value)` 。如果直接使用 `result[j].next = new ListNode(value)` 赋值在某些测试例会出错。这部分的具体原理目前还不是很明白。

Java 实现

```
/**
 * Definition for ListNode
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param hashTable: A list of The first node of linked list
     * @return: A list of The first node of linked list which ha
ve twice size
     */
    public ListNode[] rehashing(ListNode[] hashTable) {
        if (hashTable == null || hashTable.length == 0) {
            return null;
        }
        int len = hashTable.length;
        int newLen = len * 2;
        ListNode[] result = new ListNode[newLen];

        for (int i = 0; i < len; i++) {
            ListNode curt = hashTable[i];
            while (curt != null) {
                int j = (curt.val % newLen + newLen) % newLen;

                if (result[j] == null) {
                    result[j] = new ListNode(curt.val);
                } else {
                    ListNode dummy = result[j];
                    while (dummy.next != null) {
                        dummy = dummy.next;
                    }
                    dummy.next = new ListNode(curt.val);
                }
```

```
                curt = curt.next;
            }
        }

        return result;
    }
};
```

参考

LRU Cache

LRU Cache (leetcode lintcode)

```
Description
Design and implement a data structure for Least Recently Used (L
RU) 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 alre
ady present.
When the cache reached its capacity,
it should invalidate the least recently used item before inserti
ng a new item.
```

解题思路

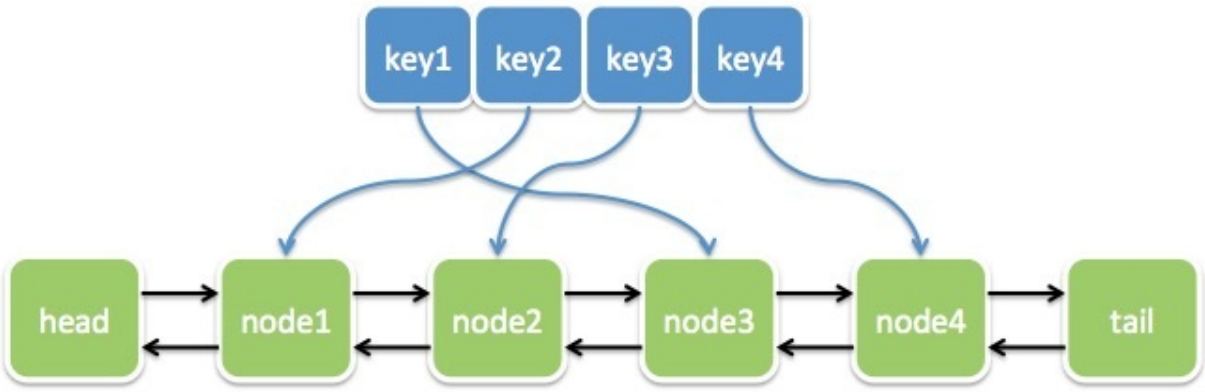
Least Recently Used Cache 的基本思想是“最近用到的数据被重用的概率比较早用到的大得多”。

对于 Cache ，如果要求查找复杂度为 `O(1)` ，那么需要使用 HashMap 保存 key 和 value。

而对于 Cache 的插入和删除操作，也需要复杂度为 `O(1)` ，使用双向链表存储 Cache。从头到尾依次按照从旧到新的顺序存储 Cache ，对于任一结点，如果被访问了，将其移至尾部，这样最近被使用过的内容向链表尾部移动，而未使用内容则向链表头部移动；如 Cache 已满，则删除头部结点（近期未使用），在尾部插入新结点。

用到的两个数据结构：

1. HashMap：保存 key 和 value。
2. Doubly Linked List：保存对象新旧程度的队列。



易错点

1. 双向链表的头尾哨兵结点在初始化时要连在一起。
2. 在涉及删除和添加结点时，要同时对 HashMap 和 LinkedList 进行删除/添加操作。

Java 实现

```
public class Solution {

    private class Node {
        int key, value;
        Node next, prev;
        public Node (int key, int value) {
            this.key = key;
            this.value = value;
            this.next = null;
            this.prev = null;
        }
    }

    private int capacity;
    private HashMap<Integer, Node> hash = new HashMap<Integer, N
ode>();
    private Node head = new Node(-1, -1);
    private Node tail = new Node(-1, -1);

    // @param capacity, an integer
    public Solution(int capacity) {
        // write your code here
```

```
        this.capacity = capacity;
        tail.prev = head;
        head.next = tail;
    }

    // @return an integer
    public int get(int key) {
        // write your code here
        if (!hash.containsKey(key)) {
            return -1;
        }

        // remove current
        Node current = hash.get(key);
        current.prev.next = current.next;
        current.next.prev = current.prev;

        // move current to tail
        move_to_tail(current);

        return hash.get(key).value;
    }

    // @param key, an integer
    // @param value, an integer
    // @return nothing
    public void set(int key, int value) {
        // write your code here
        if (get(key) != -1) {
            hash.get(key).value = value;
            return;
        }

        if (hash.size() == capacity) {
            hash.remove(head.next.key);
            head.next = head.next.next;
            head.next.prev = head;
        }

        Node insert = new Node(key, value);
```

```
            hash.put(key, insert);
            move_to_tail(insert);
        }

        private void move_to_tail (Node current) {
            current.prev = tail.prev;
            tail.prev = current;
            current.prev.next = current;
            current.next = tail;
        }
    }
```

参考

1. LRU Cache | 九章算法
2. [LeetCode] LRU Cache, Solution | 水中的鱼
3. LRU Cache leetcode java | 爱做饭的小莹子
4. LeetCode – LRU Cache (Java) | ProgramCreek

Data Stream Median

Data Stream Median (leetcode lintcode)

```
Description
Numbers keep coming, return the median of numbers at every time
a new number added.

Clarification
What's the definition of Median?
- Median is the number that in the middle of a sorted array.
If there are n numbers in a sorted array A, the median is A[(n -
 1) / 2].
For example, if A=[1,2,3], median is 2. If A=[1,19], median is 1
.

Example
For numbers coming list: [1, 2, 3, 4, 5], return [1, 1, 2, 2, 3]
.
For numbers coming list: [4, 5, 1, 3, 2, 6, 0], return [4, 4, 4,
 3, 3, 3, 3].
For numbers coming list: [2, 20, 100], return [2, 2, 20].

Challenge
Total run time in O(nlogn).
```

解题思路

一、优先队列/堆

- 使用两个堆 `Heap` 和一个变量 `median` ， `median` 存储当前的中位数，最大堆 `maxHeap` 存储小于 `median` 的数，最小堆 `minHeap` 存储大于 `median` 的数。
- 新插入数值比 `median` 小，放入 `maxHeap` ；比 `median` 大，放入 `minHeap` 。
- 比较 `maxHeap` 和 `minHeap` 的大小，本解法中其实是比较

`maxHeap.size() + 1` 和 `minHeap` 的大小。
  - 如果 `maxHeap.size() > minHeap.size()` ，那么将 `median` 放入 `minHeap` ，取出 `maxHeap` 最大值作为当前 `median` 。
  - 如果 `maxHeap.size() + 1 < minHeap.size()` ，那么将 `median` 放入 `maxHeap` ，取出 `minHeap` 最小值作为当前 `median` 。

Java 中的 `PriorityQueue` 默认是最小堆，堆顶元素是最小值，所以处理最大堆时取了负数。

Java 实现

```
public class Solution {
    /**
     * @param nums: A list of integers.
     * @return: the median of numbers
     */
    public int[] medianII(int[] nums) {
        int[] results = new int[nums.length];
        if (nums == null || nums.length == 0) {
            return results;
        }

        int median = nums[0];
        results[0] = median;
        PriorityQueue<Integer> maxHeap = new PriorityQueue<Integ
er>();
        PriorityQueue<Integer> minHeap = new PriorityQueue<Integ
er>();

        for (int i = 1; i < nums.length; i++) {
            if (nums[i] < median) {
                maxHeap.add(-nums[i]);
            } else {
                minHeap.add(nums[i]);
            }

            if (maxHeap.size() > minHeap.size()) {
                minHeap.add(median);
                median = -maxHeap.peek();
                maxHeap.remove();
```

```
            } else if (maxHeap.size() + 1 < minHeap.size()) {
                maxHeap.add(-median);
                median = minHeap.peek();
                minHeap.remove();
            }

            results[i] = median;
        }

        return results;
    }
}
```

二、优先队列/堆 II

和方法一的实现思路类似，具体实现细节有所不同。不再单独使用变量存储 median ，维持 maxHeap 和 minHeap 的大小，maxHeap 可以比 minHeap 多一个元素， maxHeap 堆顶即为中位数 median 。

本方法的实现更加直观。

Java 实现

```
public class Solution {
    /**
     * @param nums: A list of integers.
     * @return: the median of numbers
     */
    public int[] medianII(int[] nums) {
        int[] results = new int[nums.length];
        if (nums == null || nums.length == 0) {
            return results;
        }

        Comparator<Integer> revComp = new Comparator<Integer>()
{
            public int compare(Integer left, Integer right) {
                return right - left;
            }
        };
```

```
        PriorityQueue<Integer> minHeap = new PriorityQueue<Integ
er>(nums.length);
        PriorityQueue<Integer> maxHeap = new PriorityQueue<Integ
er>(nums.length, revComp);

        results[0] = nums[0];
        maxHeap.add(nums[0]);

        for (int i = 1; i < nums.length; i++) {
            maxHeap.add(nums[i]);
            minHeap.add(maxHeap.poll());

            if (minHeap.size() > maxHeap.size()) {
                maxHeap.add(minHeap.poll());
            }

            results[i] = maxHeap.peek();
        }

        return results;
    }

}
```

参考

1. Data Stream Median | 九章算法
2. lintcode 1: Data Stream Median | 西施豆腐渣
3. Data Stream Median – Lintcode Java | WELKIN LAN

Longest Consecutive Sequence

Longest Consecutive Sequence (leetcode lintcode)

```
Description
Given an unsorted array of integers, find the length of the long
est consecutive elements sequence.

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

Example
Given [100, 4, 200, 1, 3, 2],
The longest consecutive elements sequence is [1, 2, 3, 4]. Retur
n its length: 4.
```

解题思路

题目为求最长的连续序列，容易想到动态规划，但本题是一个特例，并不是考察动规，而是考察数据结构。

如果不考虑算法复杂度，直观的想法是先将数组排序，然后遍历寻找，然而基于比较的排序复杂度至少是 `O(nlogn)` ，不能满足题目对时间复杂度的要求。考虑使用哈希表。

- 将无序数组中的每个数存入 `HashSet` 中。
- 对于序列中的任一个数 `num[i]` ，
  - 判断 `num[i - 1], num[i - 2], num[i - 3]...` 是否在序列中（循环），搜索到的数从 `HashSet` 中删除以避免重复搜索。
  - 判断 `num[i + 1], num[i + 2], num[i + 3]...` 是否在序列中（循环），搜索到的数从 `HashSet` 中删除以避免重复搜索。
  - 进而找到整个连续序列。

算法复杂度

- 时间复杂度： `HashSet` 的操作 `add(), remove(), contains()` 的时间复

杂度为 `O(1)` ，每个数的只有一次 `add(), remove()` 操作，所以复杂度为 `O(n)` 。

易错点

> 1. 求最长长度的时候，注意 up 和 down 都是不满足条件的边界。

Java 实现

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return an integer
     */
    public int longestConsecutive(int[] num) {
        if (num == null || num.length == 0) {
            return 0;
        }

        HashSet<Integer> set = new HashSet<Integer>();
        for (int i = 0; i < num.length; i++) {
            set.add(num[i]);
        }

        int longest = 0;
        for (int i = 0; i < num.length; i++) {
            int down = num[i] - 1;
            while (set.contains(down)) {
                set.remove(down);
                down--;
            }

            int up = num[i] + 1;
            while (set.contains(up)) {
                set.remove(up);
                up++;
            }
            // between down and up are the longest consecutive s
equence
            longest = Math.max(longest, up - down - 1);
        }

        return longest;
    }
}
```

参考

1. [LeetCode] Longest Consecutive Sequence | 喜刷刷

Subarray Sum

Subarray Sum (leetcode lintcode)

```
Description
Given an integer array, find a subarray where the sum of numbers
 is zero.
Your code should return the index of the first number and the in
dex of the last number.

Notice
There is at least one subarray that it's sum equals to zero.

Example
Given [-3, 1, 2, -3, 4], return [0, 2] or [1, 3].
```

解题思路

一、遍历

使用双重循环依次把所有子序列全部遍历一遍，时间复杂度是 `O(n^2)` 。

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the fir
st number
     *          and the index of the last number
     */
    public ArrayList<Integer> subarraySum(int[] nums) {
        // write your code here
        ArrayList<Integer> results = new ArrayList<Integer>();
        if (nums == null || nums.length == 0) {
            return results;
        }

        for (int i = 0; i < nums.length; i++) {
            int sum = 0;
            for (int j = i ; j < nums.length; j++) {
                sum += nums[j];
                if (sum == 0) {
                    results.add(i);
                    results.add(j);
                    return results;
                }
            }
        }

        return results;
    }
}
```

二、HashMap

从数组第一个数开始求和 sum，使用 HashMap 记录当前 index 和 sum ，当出现相同的 sum 值时，说明 index1 + 1 到 index2 是一个和为零的子序列。

需要添加一个 index = -1 的虚拟结点，以确保数组第一个数为零的情况可以被记录。

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the fir
st number
     *          and the index of the last number
     */
    public ArrayList<Integer> subarraySum(int[] nums) {
        // write your code here
        ArrayList<Integer> results = new ArrayList<Integer>();
        if (nums == null || nums.length == 0) {
            return results;
        }

        HashMap<Integer, Integer> map = new HashMap<Integer, Int
eger>();

        map.put(0, -1);

        int sum = 0;
        for (int i = 0; i < nums.length; i++) {
            sum += nums[i];

            if (map.containsKey(sum)) {
                results.add(map.get(sum) + 1);
                results.add(i);
                return results;
            }
            map.put(sum, i);
        }

        return results;
    }
}
```

参考

1. Lintcode: Subarray Sum 解题报告 | Yu's garden

Anagrams

Anagrams (leetcode lintcode)

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

Notice
All inputs will be in lower-case

Example
Given ["lint", "intl", "inlt", "code"], return ["lint", "inlt",
"intl"].
Given ["ab", "ba", "cd", "dc", "e"], return ["ab", "ba", "cd", "
dc"].

What is Anagram?
- Two strings are anagram if they can be the same after change t
he order of characters.
```

解题思路

一、HashMap

- 对每个字符串 `string` 进行排序得到 `string'` 。
- 然后在 `HashMap` 中存储 `string' <—> List<string>` 。
  - 没有对应排序字符串 `string'` ，新建 `List` ，存储当前配对 。
  - 存在对应 `string'` ，将原始 `string` 加入对应链表。
- 最后输出所有满足条件的 `anagrams` 。

算法复杂度

- 时间复杂度：在 `for` 循环中有字符串排序，所以是 `N * L * O(logL)` ，其中 `N` 是字符串数组长度， `L` 是最长字符串的长度。

易错点

1. Java 中对字符串 `String` 排序，需要先转化为 `char` 类型数组，用 `Arrays.sort()` 排序后再转成字符串。

Java 实现

```
public class Solution {
    /**
     * @param strs: A list of strings
     * @return: A list of strings
     */
    public List<String> anagrams(String[] strs) {
        List<String> results = new ArrayList<String>();
        if (strs == null) {
            return results;
        }

        HashMap<String, List<String>> map = new HashMap<String,
List<String>>();
        for (int i = 0; i < strs.length; i++) {
            String s = strs[i];
            char[] chars = s.toCharArray();

            Arrays.sort(chars);
            String sSort = new String(chars);

            if (map.containsKey(sSort)) {
                map.get(sSort).add(s);
            } else {
                List<String> list = new ArrayList<String>();
                list.add(s);
                map.put(sSort, list);
            }
        }

        for (List<String> list : map.values()) {
            if (list.size() > 1) {
                result.addAll(list);
            }
```

```
        }

        return results;

    }
}
```

二、HashMap 2

思路同方法一类似，只是对 `string` 排序进行了优化，改为了对每个 `string` 求哈希值。

算法复杂度

- 时间复杂度：求哈希值函数的复杂度是 `O(L)` ，`L` 为最长字符串的长度。所以算法复杂度优化为 `N * O(L)` ，`N` 为字符串数组长度。

Java 实现

```
public class Solution {
    /**
     * @param strs: A list of strings
     * @return: A list of strings
     */
    public List<String> anagrams(String[] strs) {
        List<String> results = new ArrayList<String>();
        if (strs == null) {
            return results;
        }

        HashMap<Integer, ArrayList<String>> map = new HashMap<In
teger, ArrayList<String>>();

        for (String str : strs) {
            int[] count = new int[26];
            for (int i = 0; i < str.length(); i++) {
                count[str.charAt(i) - 'a']++;
            }

            int hash = getHash(count);
```

```
            if (!map.containsKey(hash)) {
                map.put(hash, new ArrayList<String>());
            }
            map.get(hash).add(str);
        }

        for (ArrayList<String> tmp : map.values()) {
            if (tmp.size() > 1) {
                results.addAll(tmp);
            }
        }

        return results;
    }

    private int getHash(int[] count) {
        int hash = 0;
        int a = 378551;
        int b = 63689;
        for (int num : count) {
            hash = hash * a + num;
            a = a * b;
        }
        return hash;
    }
}
```

参考

1. Anagrams.java | shawnfan/LintCode
2. Anagrams | 九章算法

Heapify

Heapify (leetcode lintcode)

```
Description
Given an integer array, heapify it into a min-heap array.
For a heap array A, A[0] is the root of heap, and for each A[i],

A[i * 2 + 1] is the left child of A[i] and A[i * 2 + 2] is the r
ight child of A[i].

Clarification
- What is heap?
- Heap is a data structure, which usually have three methods: pu
sh, pop and top.
where "push" add a new element the heap, "pop" delete the minimu
m/maximum element in the heap,
"top" return the minimum/maximum element.

- What is heapify?
- Convert an unordered integer array into a heap array. If it is
 min-heap,
for each element A[i], we will get A[i * 2 + 1] >= A[i] and A[i
* 2 + 2] >= A[i].

- What if there is a lot of solutions?
- Return any of them.

Example
Given [3,2,1,4,5], return [1,2,3,4,5] or any legal heap array.

Challenge
O(n) time complexity
```

解题方法

构造堆的基本操作有两种：`siftdown` 和 `siftup` 。两种操作在本质上是相同的，处理不符合堆定义的结点直至满足规则。以构造最小堆为例：

- `siftup` ：当前结点比父结点大，两者交换，迭代操作直至当前结点小于其父结点。
- `siftdown` ：当前结点比子结点大，同较小的儿子结点交换，迭代操作直至当前结点小于两个儿子结点。

对一个结点进行 `siftup` 或 `siftdown` 时涉及的实际操作数，与该结点需要移动的距离正相关。

- 对 `siftup` 是结点从树底层开始向上移动的距离，所以构造完成时树顶端结点的移动代价较高，自底向上 `siftup` 构造堆时，大量的叶子结点需要移动约为 `logn` 的距离。
- 对 `siftdown` 是结点从树顶端开始向下移动的距离，所以构造完成时叶子节点的移动代价较高，自顶向下 `siftdown` 构造堆时，只有根结点等少数几个结点需要移动约为 `logn` 的长距离。
- 不难发现，两种方式构造堆所需要的操作数是不同的。

假设树的高度为 `h = logn` ，那么在最坏情况下，自顶向下 `siftdown` 构造堆所需要的操作数为：

```
sum1 = (0 * n/2) + (1 * n/4) + (2 * n/8) + ... + (h * 1)
```

使用泰勒级数对其进行近似可以得到时间复杂度最坏为 `O(n)` 。具体证明过程请参考链接。

同样假设在最坏情况下，自底向上 `siftup` 构造堆所需要的操作数为：

```
sum2 = (h * n/2) + ((h-1) * n/4) + ((h-2)*n/8) + ... + (0 * 1)
```

`siftup` 所需要的操作更多，其中第一项 `h * n/2 = 1/2 * nlogn` ，因此其时间复杂度最差为 `O(nlogn)` 。

以上内容其实会引出一个问题：

> 如果构造堆的时间复杂度为 `O(n)` ，那么堆排序的时间复杂度为什么是 `O(nlogn)` 呢？

具体分析可参考本文提供的参考链接。

易错点

> 1. 在对数组遍历处理时，注意 `i` 的取值范围和取值顺序，尤其是 `siftdown` 实现。
> 2. 在 `siftup` 和 `siftdown` 函数中，循环终止的条件有两个，一个是自变量 `k` 的取值范围，另一个是满足最小堆的特性。

一、自底向上 `siftup`

根据最小堆的定义，对数组中的每个元素，依次进行 `siftup` 操作。

- `siftup` ：当前结点比父结点小，两者交换，迭代操作直至当前结点大于其父结点。

Java 实现

```
public class Solution {
    /**
     * @param A: Given an integer array
     * @return: void
     */
    public void heapify(int[] A) {
        if (A == null || A.length == 0) {
            return;
        }

        for (int i = 0; i < A.length; i++) {
            siftup(A, i);
        }
    }

    private void siftup (int[] A, int k) {
        while (k != 0) {
            int father = (k - 1) / 2;
            if (A[k] > A[father]) {
                break;
            }
            int temp = A[k];
            A[k] = A[father];
            A[father] = temp;

            k = father;
        }
    }
}
```

二、自顶向下 `siftdown`

对数组中的每个元素，依次进行 `siftdown` 操作，可参考这个演示 demo。

- `siftdown` ：当前结点比父结点大，同较小的儿子结点交换，迭代操作直至当前结点大于两个儿子结点。

注：具体实现时需要注意索引的边界问题。

Java 实现

```
public class Solution {
    /**
     * @param A: Given an integer array
     * @return: void
     */
    public void heapify(int[] A) {
        if (A == null || A.length == 0) {
            return;
        }
        // the heap array start at index 0, not index 1 (i = A.l
ength / 2)
        for (int i = A.length / 2 - 1; i >= 0; i--) {
            siftdown(A, i);
        }
    }

    private void siftdown (int[] A, int k) {
        while (k < A.length) {
            int smallest = k;
            if (k * 2 + 1 < A.length && A[k * 2 + 1] < A[smalles
t]) {
                smallest = k * 2 + 1;
            }
            if (k * 2 + 2 < A.length && A[k * 2 + 2] < A[smalles
t]) {
                smallest = k * 2 + 2;
            }
            if (smallest == k) {
                break;
            }

            int tmp = A[smallest];
            A[smallest] = A[k];
            A[k] = tmp;

            k = smallest;
        }
    }
```

```
}
```

参考

1. Heapify | 九章算法
2. How can building a heap be O(n) time complexity? | StackOverFlow
3. Heapify | LintCode & LeetCode 题解分析

Word Search II

Word Search II (leetcode lintcode)

```
Description
Given a matrix of lower alphabets and a dictionary.
Find all words in the dictionary that can be found in the matrix
.
A word can start from any position in the matrix and go left/rig
ht/up/down to the adjacent position.

Example
Given matrix:
doaf
agai
dcan
and dictionary:
{"dog", "dad", "dgdg", "can", "again"}
return {"dog", "dad", "can", "again"}

dog:
doaf
agai
dcan

dad:
doaf
agai
dcan

can:
doaf
agai
dcan

again:
doaf
agai
dcan

Challenge
Using trie to implement your algorithm.
```

解题思路

暴力搜索每一个单词会导致大量的重复比较，本题目主要考察单词查找树 Trie 的使用。

- 将字典中的单词依次存储在单词查找树 Trie 中
- 使用 DFS 在 board 中查找，利用 Trie 进行剪枝
- 将查找到的单词放入列表中并返回

Java 实现

```
public class Solution {
    /**
     * @param board: A list of lists of character
     * @param words: A list of string
     * @return: A list of string
     */
    public ArrayList<String> wordSearchII(char[][] board, ArrayL
ist<String> words) {
        ArrayList<String> ans = new ArrayList<String>();

        TrieTree tree = new TrieTree(new TrieNode());
        for (String word : words) {
            tree.insert(word);
        }

        String res = "";
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
                search(board, i, j, tree.root, ans, res);
            }
        }
        return ans;
    }

    class TrieNode {
        String s;
        boolean isString;
        HashMap<Character, TrieNode> subtree;
        public TrieNode() {
```

```
            // TODO Auto-generated constructor stub
            isString = false;
            subtree = new HashMap<Character, TrieNode>();
            s = "";
        }
    };

    class TrieTree {
        TrieNode root;
        public TrieTree (TrieNode TrieNode) {
            root = TrieNode;
        }
        public void insert (String s) {
            TrieNode now = root;
            for (int i = 0; i < s.length(); i++) {
                if (!now.subtree.containsKey(s.charAt(i))) {
                    now.subtree.put(s.charAt(i), new TrieNode())
;
                }
                now = now.subtree.get(s.charAt(i));
            }
            now.s = s;
            now.isString = true;
        }

        public boolean find (String s) {
            TrieNode now = root;
            for (int i = 0; i < s.length(); i++) {
                if (!now.subtree.containsKey(s.charAt(i))) {
                    return false;
                }
                now = now.subtree.get(s.charAt(i));
            }
            return now.isString;
        }
    };

    public int []dx = {1, 0, -1, 0};
    public int []dy = {0, 1, 0, -1};
```

```
    public void search (char[][] board,
                        int x, int y,
                        TrieNode root,
                        ArrayList<String> ans,
                        String res) {
        if (root.isString == true) {
            if (!ans.contains(root.s)) {
                ans.add(root.s);
            }
        }

        if (x < 0 || x >= board.length ||
            y < 0 || y >= board[0].length ||
            board[x][y] == 0 || root == null) {
            return;
        }

        if (root.subtree.containsKey(board[x][y])) {
            for (int i = 0; i < 4; i++) {
                char now = board[x][y];
                board[x][y] = 0;
                search(board, x + dx[i], y + dy[i], root.subtree
.get(now), ans, res);
                board[x][y] = now;
            }
        }
    }
}
```

参考

1. Word Search II | 九章算法

Single Number

Single Number (leetcode lintcode)

```
Description
Given 2*n + 1 numbers, every numbers occurs twice except one, fi
nd it.

Example
Given [1,2,2,1,3,4,3], return 4

Challenge
One-pass, constant extra space.
```

解题思路

使用位操作，任意两个相同的数做异或运算（不进位加法），结果为 0，也即 `A ^ A = 0` ，而 `A ^ 0 = A` 。

Java 实现

```
public class Solution {
    /**
      *@param A : an integer array
      *return : a integer
      */
    public int singleNumber(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        int res = A[0];
        for (int i = 1; i < A.length; i++) {
            res = res ^ A[i];
        }

        return res;
    }
}
```

Single Number II

Single Number II (leetcode lintcode)

```
Description
Given 3*n + 1 numbers, every numbers occurs triple times except
one, find it.

Example
Given [1,1,2,3,3,3,2,2,4,1] return 4

Challenge
One-pass, constant extra space.
```

解题思路

一、位运算

可以参考 Single Number 的解决思路，采用位运算。由于所有数字都出现奇数次，所以无法直接使用异或操作。考虑到计算机使用二进制存储数字，可以建立一个32位的数字，统计每一位1出现的次数，如果一个整数出现了三次，那么三个0或者三个1对3取余都为0，对每个数的对应位都加起来对3取余，剩下的就是 Single Number 。

Java 实现

```
public class Solution {
    /**
     * @param A : An integer array
     * @return : An integer
     */
    public int singleNumberII(int[] A) {
        if (A == null || A.length == 0) {
            return -1;
        }

        int result = 0;
        int[] bits = new int[32];
        for (int i = 0; i < 32; i++) {
            for (int j = 0; j < A.length; j++) {
                bits[i] += A[j] >> i & 1;
                bits[i] %= 3;
            }
            result |= (bits[i] << i);
        }

        return result;
    }
}
```

参考

1. [LeetCode] Single Number II 单独的数字之二 | Grandyang

Single Number III

Single Number III (leetcode lintcode)

```
Description
Given 2*n + 2 numbers, every numbers occurs twice except two, fi
nd them.

Example
Given [1,2,2,3,4,4,5,3] return 1 and 5

Challenge
O(n) time, O(1) extra space.
```

解题思路

由于有两个不同的 single number A 和 B ，所以如果直接对所有数求异或，得到的结果是 A ^ B ，无法直接区分两者。但是由于 A 和 B 不相等，所以异或结果一定不为 0 ，可以找到两者不相同的某一位，通过按位与操作将数组分为两部分，这样 A 和 B 就被划分到不同的子数组中，再分别进行异或操作即可得到两者。

易错点

1. `List<Integer>` 需要初始化为 `ArrayList<Integer>` ，不能直接初始化为 `List<Integer>` ，否则会报错 `List is abstract; cannot be instantiated List results = new List();` 。

Java 实现

```
public class Solution {
    /**
     * @param A : An integer array
     * @return : Two integers
     */
    public List<Integer> singleNumberIII(int[] A) {
        if (A == null || A.length == 0) {
            return null;
        }

        int xor = A[0];
        for (int i = 1; i < A.length; i++) {
            xor ^= A[i];
        }

        int lastBit = xor - ((xor - 1) & xor);

        int a1 = 0, a2 = 0;
        for (int i = 0; i < A.length; i++) {
            if ((A[i] & lastBit) == 0) {
                a1 ^= A[i];
            } else {
                a2 ^= A[i];
            }
        }

        List<Integer> results = new ArrayList<Integer>();
        results.add(a1);
        results.add(a2);
        return results;
    }
}
```

参考

1. [Leetcode] Single Number III, Solution | 水中的鱼

Majority Number

Majority Number (leetcode lintcode)

```
Description
Given an array of integers,
the majority number is the number that occurs more than half of
the size of the array.
Find it.

Notice
You may assume that the array is non-empty and the majority numb
er always exist in the array.

Example
Given [1, 1, 1, 1, 2, 2, 2], return 1

Challenge
O(n) time and O(1) extra space
```

解题思路

如果不考虑 `O(1)` 额外空间的限制，可以使用哈希表记录次数，然后判断次数大于一半的即可。

在本题要求下，使用抵消法。具体实现如下：

- 把链表中的数字视为议案，不断对某个议案进行投票
- 如果下一个提案同当前提案相同，那么投票数加一
- 如果下一个提案不同，那么投票数减一
- 如果投票数归零，那么将当前提案置为投票提案

在确定存在 Majority Number 的时候，该方法最后得到的就是 Majority Number 。

Java 实现

```
public class Solution {
    /**
     * @param nums: a list of integers
     * @return: find a  majority number
     */
    public int majorityNumber(ArrayList<Integer> nums) {
        if (nums == null || nums.size() == 0) {
            return -1;
        }

        int count = 0, candidate = -1;
        for (int i = 0; i < nums.size(); i++) {
            if (count == 0) {
                candidate = nums.get(i);
                count++;
            } else if (candidate == nums.get(i)) {
                count++;
            } else {
                count--;
            }
        }

        return candidate;
    }
}
```

参考

1. Lintcode: Majority Number 解题报告 | Yu's garden

Majority Number II

Majority Number II (leetcode lintcode)

```
Description
Given an array of integers,
the majority number is the number that occurs more than 1/3 of t
he size of the array.
Find it.

Notice
There is only one majority number in the array.

Example
Given [1, 2, 1, 2, 1, 3, 3], return 1.

Challenge
O(n) time and O(1) extra space.
```

解题思路

在 Majority Number I 中我们的做法是抵消两个不同的数，以此类推，在本题中，当遇到三个不同的数时进行抵消，由于 Majority Number 超过 1/3 ，所以最终一定会留下来。

我们需要保存两个数 `can1` 和 `can2` 及对应的次数 `c1` 和 `c2`

- 如果 `c1 == 0` ，将当前值赋给 `can1` ，初始化 `c1` 。
- 如果 `c2 == 0` ，将当前值赋给 `can2` ，初始化 `c2` 。
- 如果当前值等于 `can1` ，增加 `c1` 。
- 如果当前值等于 `can2` ，增加 `c2` 。
- 否则，也就是出现了第三个不同的值，减少 `c1` 和 `c2` 。
- 最后，检查剩下的两个候选值。

之所以最后还要检查候选值，是因为在抵消之后，Majority Number 对应的次数不一定是最多的，比如序列 `[1 1 1 1 2 3 2 3 4 4 4]`，抵消后 4 出现的次数比 1 1多。

易错点：

1. 在 `count1 == 0` 时，未对其进行初始化 `count1 = 1` 。
2. 在对两个候选值进行投票时，要先判断当前值是否等于任一候选值。如果先判断 `count1, count2` 是否为零，那么遇到两个重复数值时会被错认为两个不同的候选值。
3. 由于抵消之后 Majority Number 对应的数量不一定是最多的，所以需要重新进行计数。

Java 实现

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: The majority number that occurs more than 1/3
     */
    public int majorityNumber(ArrayList<Integer> nums) {
        if (nums == null || nums.size() == 0) {
            return -1;
        }

        int candidate1 = 0, candidate2 = 0;
        int count1 = 0, count2 = 0;
        for (int i = 0; i < nums.size(); i++) {
            if (nums.get(i) == candidate1) {
                count1++;
            } else if (nums.get(i) == candidate2) {
                count2++;
            } else if (count1 == 0) {
                candidate1 = nums.get(i);
                count1 = 1;
            } else if (count2 == 0) {
                candidate2 = nums.get(i);
                count2 = 1;
            } else {
                count1--;
```

```
                count2--;
            }
        }

        count1 = count2 = 0;
        for (int i = 0; i < nums.size(); i++) {
            if (nums.get(i) == candidate1) {
                count1++;
            } else if (nums.get(i) == candidate2) {
                count2++;
            }
        }
        return count1 > count2 ? candidate1 : candidate2;
    }
}
```

参考

1. Lintcode: Majority Number II 解题报告 | Yu's garden

Majority Number III

Majority Number III (leetcode lintcode)

```
Description
Given an array of integers and a number k,
the majority number is the number that occurs more than 1/k of t
he size of the array.
Find it.

Notice
There is only one majority number in the array.

Example
Given [3,1,2,3,2,3,3,4,4,4] and k=3, return 3.

Challenge
O(n) time and O(k) extra space
```

解题思路

在 `HashMap<key, value>` 中维护 `k - 1` 个 `candidate` ，`key` 为数字，`value` 为其出现次数。

- 在 `candidate` 个数小于 `k` 时，将数字及其出现次数依次存入 `HashMap` 或者进行更新。
- 在 `candidate` 个数大于等于 `k` 时，将 `k - 1` 个数的出现次数减 `1` ，并记录次数为 `0` 的数字，将其从 `HashMap` 删除。
- 完成它上述步骤后，在所有数字中，统计出现次数最多的数字，即为结果。

HashMap 的遍历方法

- 如果只需要访问 key

```
Map<String, Object> map = ...;

for (String key : map.keySet()) {
    // ...
}
```

- 如果只需要访问 value

```
for (Object value : map.values()) {
    // ...
}
```

- 如果 key 和 value 都需要访问

```
for (Map.Entry<String, Object> entry : map.entrySet()) {
    String key = entry.getKey();
    Object value = entry.getValue();
    // ...
}
```

易错点

1. 当 HashMap 中元素达到 `k` 个时，需要将所有的 `value` 值减一，此时不能直接删除对应的 `key`，推测和 HashMap 的遍历机制有关。需要记录 `value` 为零的 `key`，接下来删除。

Java 实现

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @param k: As described
     * @return: The majority number
     */
    public int majorityNumber(ArrayList<Integer> nums, int k) {
        if (nums == null || nums.size() == 0) {
            return -1;
        }
```

```
        // count at most k keys
        HashMap<Integer, Integer> counters = new HashMap<Integer
, Integer>();
        for (Integer i : nums) {
            if (!counters.containsKey(i)) {
                counters.put(i, 1);
            } else {
                counters.put(i, counters.get(i) + 1);
            }

            if (counters.size() >= k) {
                removeKey(counters);
            }
        }

        // corner case
        if (counters.size() == 0) {
            return Integer.MIN_VALUE;
        }

        // recalculate counters
        for (Integer i : counters.keySet()) {
            counters.put(i, 0);
        }
        for (Integer i : nums) {
            if (counters.containsKey(i)) {
                counters.put(i, counters.get(i) + 1);
            }
        }

        // find the max key
        int maxCounter = 0, maxKey = 0;
        for (Integer i : counters.keySet()) {
            if (counters.get(i) > maxCounter) {
                maxCounter = counters.get(i);
                maxKey = i;
            }
        }
```

```
        return maxKey;
    }

    private void removeKey (HashMap<Integer, Integer> counters)
{
        Set<Integer> keySet = counters.keySet();
        List<Integer> removeList = new ArrayList<>();
        for (Integer key : keySet) {
            counters.put(key, counters.get(key) - 1);
            if (counters.get(key) == 0) {
                removeList.add(key);
            }
        }
        for (Integer key : removeList) {
            counters.remove(key);
        }
    }
}
```

参考

1. Majority Number III | 九章算法
2. Iterate through a HashMap [duplicate] | stackoverflow

Fast Power

Fast Power (leetcode lintcode)

```
Description
Calculate the a^n % b where a, b and n are all 32bit integers.

Example
For 231 % 3 = 2
For 1001000 % 1000 = 0

Challenge
O(logn)
```

解题思路

本题目主要考察快速幂的实现，以及 int 类型数据溢出的处理。

快速幂实现的思路是 `a ^ n = (a ^ n/2) * (a ^ n/2)` ，这里需要注意的是当 `n` 为奇数时需要再多乘一个 `a` 。在具体实现时，对于 `n == 0` 和 `n == 1` 这两种情况也需要单独处理。

题目中提到 `a` 和 `b` 都是 `32` 位整数，所以在计算过程中中间变量使用 `long` 类型存储以免溢出。另，在本题中假定 `a` 和 `b` 都是正数。

本题目还用到了整数取模操作的以下特点： `(a * b) % p =((a % p) * (b % p)) % p` 。

算法复杂度

- 时间复杂度： `O(logn)` 。

易错点

> 1. 注意对 int 类型数据溢出的处理。

Java 实现

```
class Solution {
    /*
     * @param a, b, n: 32bit integers
     * @return: An integer
     */
    public int fastPower(int a, int b, int n) {
        if (n == 1) {
            return a % b;
        }
        if (n == 0) {
            return 1 % b;
        }

        long product = fastPower(a, b, n / 2);
        product = (product * product) % b;
        if (n % 2 == 1) {
            product = (product * a) % b;
        }

        return (int) product;
    }

};
```

参考

1. Lintcode: Fast Power 解题报告 | Yu's garden

Sqrt(x)

Sqrt(x) (lintcode)

```
Description
Implement int sqrt(int x).
Compute and return the square root of x.

Example
sqrt(3) = 1
sqrt(4) = 2
sqrt(5) = 2
sqrt(10) = 3

Challenge
O(log(x))
```

解题思路

一、二分法

将该题转化为寻找第一个平方不大于目标值 `x` 的数，即可使用二分法。为了防止溢出，需使用 `long` 类型作为中间变量。

Java 实现

```
class Solution {
    /**
     * @param x: An integer
     * @return: The sqrt of x
     */
    public int sqrt(int x) {
        if (x < 0) {
            return -1;
        }
        if (x == 0) {
            return 0;
        }
        // find the last number which square of it <= x
        long start = 1, end = x;
        while (start + 1 < end) {
            long mid = start + (end - start) / 2;
            if (mid * mid <= x) {
                start = mid;
            } else {
                end = mid;
            }
        }

        if (end * end <= x) {
            return (int) end;
        }
        return (int) start;
    }
}
```

二、牛顿法

牛顿法偏数学计算，是一种在实数域/复数域近似求解方程的方法，使用函数 `f(x)` 的泰勒级数的前面几项寻找方程 `f(x) = 0` 的根，具体原理不在此详述，感兴趣同学可以查阅参考链接。这里直接给出求解的迭代公式，其中 `f'(x)` 是函数 `f(x)` 的导数。

```
x_(n+1) = x_n - f(x_n)/f'(x_n)
```

具体到本题目中，对应函数为 `f(y)` （考虑到 `x` 是常数，用 `y` 作自变量），有如下推导：

```
y_(n+1) = y_n - f(y_n) / f'(y_n)
        = y_n - ((y_n)^2 - x)/2y_n
        = ((y_n)^2 + x)/2y_n
        = (y_n + x/y_n) / 2
```

Java 实现

```
class Solution {
    /**
     * @param x: An integer
     * @return: The sqrt of x
     */
    public int sqrt(int x) {
        if (x == 0) {
            return 0;
        }

        double lastY = 0;
        double y = 1;
        while (y != lastY) {
            lastY = y;
            y = (y + x / y) / 2;
        }

        return (int) y;
    }
}
```

参考

1. Sqrt(x) | 九章算法
2. Sqrt(x) — LeetCode | Code Ganker
3. 牛顿法 | 维基百科

Trailing Zeros

Trailing Zeros (leetcode lintcode)

```
Description
Write an algorithm which computes the number of trailing zeros i
n n factorial.

Example
11! = 39916800, so the out should be 2

Challenge
O(log N) time
```

解题思路

正整数 n 的阶乘有多少个尾随零，可以用以下公式求解：

$$f(n) = \sum_{i=1}^{k} \left\lfloor \frac{n}{5^i} \right\rfloor = \left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{5^2} \right\rfloor + \left\lfloor \frac{n}{5^3} \right\rfloor + \cdots + \left\lfloor \frac{n}{5^k} \right\rfloor,$$

其中， `k` 需要满足 `5^(k+1) > n` 。

可以进一步简化为更容易实现的以下形式：

定义 `q_i = n / (5^i)` ，初始状态为 `q_0 = n` ，迭代公式为 `q_i+1 = q_i / 5` 。

Java 实现

```
class Solution {
    /*
     * param n: As desciption
     * return: An integer, denote the number of trailing zeros i
n n!
     */
    public long trailingZeros(long n) {
        if (n <= 0) {
            return 0;
        }

        long sum = 0;
        while (n != 0) {
            sum += n / 5;
            n /= 5;
        }
        return sum;
    }
}
```

参考

1.Trailing Zeros | 九章算法

2.Trailing Zeros | Wikipedia

O(1) Check Power of 2

O(1) Check Power of 2 (leetcode lintcode)

```
Description
Using O(1) time to check whether an integer n is a power of 2.

Example
For n=4, return true;
For n=5, return false;

Challenge
O(1) time
```

解题思路

如果一个数 `n` 是 `2` 的整数幂，那么以二进制表示的话只有一位是 `1` ，其余每一位都是 `0` ，而 `n - 1` 则全是 `1` ，且比 `n` 少一位最高位，两者相与为零。

以 `n = 4` 为例

```
n   : 4 ---> 0100
n-1 : 3 ---> 0011
```

Java 实现

```
class Solution {
    /*
     * @param n: An integer
     * @return: True or false
     */
    public boolean checkPowerOf2(int n) {
        if (n <= 0) {
            return false;
        }

        return (n & (n - 1)) == 0;
    }
};
```

参考

1. O(1) Check Power of 2 | 数据结构与算法/leetcode/lintcode题解

Digit Counts

Digit Counts (lintcode)

```
Description
Count the number of k's between 0 and n. k can be 0 - 9.

Example
if n = 12, k = 1 in
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
we have FIVE 1's (1, 10, 11, 12)
```

解题思路

依次判断每个数字含多少个 `k` ，注意特殊情况是 `k == 0` 。

```
class Solution {
    /*
     * param k : As description.
     * param n : As description.
     * return: An integer denote the count of digit k in 1..n
     */
    public int digitCounts(int k, int n) {
        // write your code here
        if (k < 0 || k > 9 || n < 0) {
            return 0;
        }

        int count = 0;
        for (int i = 0; i <= n; i++) {
            count += helper(k, i);
        }
        return count;
    }

    private int helper(int k, int n) {
        if (k == 0 && n == 0) {
            return 1;
        }
        int num = 0;
        while (n != 0) {
            if (n % 10 == k) {
                num++;
            }
            n = n / 10;
        }
        return num;
    }
};
```

Ugly Number II

Ugly Number II (lintcode)

```
Description
Ugly number is a number that only have factors 2, 3 and 5.
Design an algorithm to find the nth ugly number.
The first 10 ugly numbers are 1, 2, 3, 4, 5, 6, 8, 9, 10, 12...

Notice
Note that 1 is typically treated as an ugly number.

Example
If n=9, return 10.
```

解题思路

根据丑数的定义依次从小到大生成 `n` 个丑数，用三个指针 `p2, p3, p5` 分别保存乘以 `2, 3, 5` 以后大于当前最大丑数的最小数，三个新的丑数中的最小值即为下一个丑数。

Java 实现

```
class Solution {
    /**
     * @param n an integer
     * @return the nth prime number as description.
     */
    public int nthUglyNumber(int n) {
        // Write your code here
        if (n <= 0) {
            return 0;
        }

        int[] list = new int[n];
        list[0] = 1;
        int p2 = 0, p3 = 0, p5 = 0;


        for (int i = 1; i < n; i++) {
            while (list[p2] * 2 <= list[i - 1]) {
                p2++;
            }
            while (list[p3] * 3 <= list[i - 1]) {
                p3++;
            }
            while (list[p5] * 5 <= list[i - 1]) {
                p5++;
            }

            list[i] = Math.min(list[p2] * 2, Math.min(list[p3] *
3, list[p5] * 5));
        }

        return list[n - 1];
    }

};
```

Count 1 in Binary

Count 1 in Binary (lintcode)

```
Description
Count how many 1 in binary representation of a 32-bit integer.

Example
Given 32, return 1
Given 5, return 2
Given 1023, return 9

Challenge
If the integer is n bits with m 1 bits. Can you do it in O(m) ti
me?
```

解题思路

将 `num` 和 `num - 1` 求与操作，可消除最右边一位 `1`，如此反复操作，可以在 `O(m)` 时间完成计数。

Java 实现

```java
public class Solution {
    /**
     * @param num: an integer
     * @return: an integer, the number of ones in num
     */
    public int countOnes(int num) {
        // write your code here
        int count = 0;
        while (num != 0) {
            num = num & (num - 1);
            count++;
        }
        return count;
    }
};
```

Quick Sort

Quick Sort

Java 实现

```
public class Solution {
    /**
     * @param A an integer array
     * @return void
     */
    public void sortIntegers2(int[] A) {
        quickSort(A, 0, A.length - 1);
    }

    private void quickSort(int[] A, int start, int end) {
        if (start >= end) {
            return;
        }

        int left = start, right = end;
        // key point 1: pivot is the value, not the index
        int pivot = A[(start + end) / 2];

        // key point 2: every time you compare left & right, it
should be
        // left <= right not left < right
        while (left <= right) {
            // key point 3: A[left] < pivot not A[left] <= pivot
            while (left <= right && A[left] < pivot) {
                left++;
            }
            // key point 3: A[right] > pivot not A[right] >= piv
ot
            while (left <= right && A[right] > pivot) {
                right--;
            }
            if (left <= right) {
```

```
                int temp = A[left];
                A[left] = A[right];
                A[right] = temp;

                left++;
                right--;
            }
        }

        quickSort(A, start, right);
        quickSort(A, left, end);
    }
}
```

Reverse Pairs

Reverse Pairs (lintcode)

```
Description
For an array A, if i < j, and A [i] > A [j], called (A [i], A [j
]) is a reverse pair.
return total of reverse pairs in A.

Example
Given A = [2, 4, 1, 3, 5] , (2, 1), (4, 1), (4, 3) are reverse p
airs. return 3
```

解题思路

使用归并排序的思路，这里需要注意的是怎么对逆序对进行计数。

Java 实现

```java
public class Solution {
    /**
     * @param A an array
     * @return total of reverse pairs
     */
    public long reversePairs(int[] A) {
        // Write your code here
        if (A == null || A.length == 0) {
            return 0;
        }
        return mergeSort(A, 0, A.length - 1);
    }

    private long mergeSort(int[] A, int start, int end) {
        if (start >= end) {
            return 0;
        }
```

```
        long sum = 0;
        int mid = start + (end - start) / 2;
        sum += mergeSort(A, start, mid);
        sum += mergeSort(A, mid + 1, end);
        sum += merge(A, start, mid, end);

        return sum;
    }

    private long merge(int[] A, int start, int mid, int end) {
        int[] temp = new int[A.length];
        int left = start;
        int right = mid + 1;
        int index = start;
        long sum = 0;

        while (left <= mid && right <= end) {
            if (A[left] <= A[right]) {
                temp[index++] = A[left++];
            } else {
                temp[index++] = A[right++];
                sum += mid - left + 1;
            }
        }
        while (left <= mid) {
            temp[index++] = A[left++];
        }
        while (right <= end) {
            temp[index++] = A[right++];
        }

        for(int i = start; i <= end; i++) {
            A[i] = temp[i];
        }

        return sum;
    }
}
```

Merge Sorted Array

Merge Sorted Array (leetcode lintcode)

```
Description
Given two sorted integer arrays A and B, merge B into A as one s
orted array.

Notice
You may assume that A has enough space (size that is greater or
equal to m + n) to hold additional elements from B.
The number of elements initialized in A and B are m and n respec
tively.

Example
A = [1, 2, 3, empty, empty], B = [4, 5]
After merge, A will be filled as [1, 2, 3, 4, 5]
```

解题思路

如果从头开始比较，在合并过程中数组 A 会涉及大量的移动操作。考虑到数组 A 中有足够的空间，可以从数组末段开始比较，以从大到小的顺序合并两个数组。需要注意 A 中原有元素可能已全部移至尾部，B 中还有元素。

算法复杂度：

- 时间复杂度： `O(m + n)` 。
- 空间复杂度： `O(1)` 。

Java 实现

```
class Solution {
    /**
     * @param A: sorted integer array A which has m elements,
     *           but size of A is m+n
     * @param B: sorted integer array B which has n elements
     * @return: void
     */
    public void mergeSortedArray(int[] A, int m, int[] B, int n)
{
        if (A == null || A.length == 0 ||
            B == null || B.length == 0) {
            return;
        }

        int i = m - 1;
        int j = n - 1;
        int k = m + n - 1;
        // compare the elements in A and B from the bottom
        while (i >= 0 && j >= 0) {
            if (A[i] > B[j]) {
                A[k--] = A[i--];
            } else {
                A[k--] = B[j--];
            }
        }
        // elements in A have been moved to bottom while B still
 has elements
        while (j >= 0) {
            A[k--] = B[j--];
        }
    }
}
```

参考

1. Merge Sorted Array | 九章算法

Merge Two Sorted Arrays

Merge Two Sorted Arrays (leetcode lintcode)

```
Description
Merge two given sorted integer array A and B into a new sorted i
nteger array.

Example
A=[1,2,3,4]
B=[2,4,5,6]
return [1,2,2,3,4,4,5,6]

Challenge
How can you optimize your algorithm if one array is very large a
nd the other is very small?
```

解题思路

没什么特别的，比较、合并即可。需要考虑一个数组元素已全部取出，而另外一个数组还有剩余的情况。

Java 实现：

```
class Solution {
    /**
     * @param A and B: sorted integer array A and B.
     * @return: A new sorted integer array
     */
    public int[] mergeSortedArray(int[] A, int[] B) {
        if (A == null || A.length == 0) {
            return B;
        }
        if (B == null || B.length == 0) {
            return A;
        }

        int M = A.length;
        int N = B.length;
        int[] C = new int [M + N];
        int i = 0, j = 0, k = 0;
        while (i < M && j < N) {
            if (A[i] <= B[j]) {
                C[k++] = A[i++];
            } else {
                C[k++] = B[j++];
            }
        }
        while (i < M) {
            C[k++] = A[i++];
        }
        while (j < N) {
            C[k++] = B[j++];
        }
        return C;
    }
}
```

Recover Rotated Sorted Array

Recover Rotated Sorted Array (leetcode lintcode)

```
Description
Given a rotated sorted array, recover it to sorted array in-plac
e.

Clarification
What is rotated array?
For example, the orginal array is [1,2,3,4],
The rotated array of it can be [1,2,3,4], [2,3,4,1], [3,4,1,2],
[4,1,2,3]

Example
[4, 5, 1, 2, 3] -> [1, 2, 3, 4, 5]

Challenge
In-place, O(1) extra space and O(n) time.
```

解题思路

本题目用到一个小技巧叫做“三步翻转法”，以题目中的数组为例。

```
原始数组：     [4, 5, 1, 2, 3]
先找到转折点： [4, 5] [1, 2, 3]
第一步翻转：    [5, 4]
第二步翻转：           [3, 2, 1]
第三步翻转：    [1, 2, 3, 4, 5]
```

算法复杂度

- 时间复杂度：对整个数组做翻转，复杂度为 `O(n)` 。
- 空间复杂度： `O(1)` 。

易错点

> 1. ArrayList 的相关操作。设置第 `i` 个元素的值， `nums.set(i, value)` 。
> 2. 需要考虑输入数组没有翻转的情况，将三步翻转放在 `for` 循环中可以避免单独处理该情况，也即如果循环判断所有元素都小于下一个元素，那么不会进行任何操作。

Java 实现：

```
public class Solution {
    /**
     * @param nums: The rotated sorted array
     * @return: void
     */
    public void recoverRotatedSortedArray(ArrayList<Integer> num
s) {
        if (nums == null || nums.size() == 0) {
            return;
        }

        for (int i = 0; i < nums.size() - 1; i++) {
            if (nums.get(i) > nums.get(i + 1)) {
                reverse(nums, 0, i);
                reverse(nums, i + 1, nums.size() - 1);
                reverse(nums, 0, nums.size() - 1);
                return;
            }
        }

    }

    private void reverse (ArrayList<Integer> nums, int start, int
 end) {
        for (int i = start, j = end; i < j; i++, j--) {
            int temp = nums.get(i);
            nums.set(i, nums.get(j));
            nums.set(j, temp);
        }
    }
}
```

参考

1. Recover Rotated Sorted Array | 九章算法

Remove Duplicates from Sorted Array

Remove Duplicates from Sorted Array (leetcode lintcode)

```
Description
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.

Example
Given input array A = [1,1,2],
Your function should return length = 2, and A is now [1,2].
```

解题思路

本题目使用双指针（数组下标），一个指针 `curr` 遍历数组，另一个指针 `prev` 记录最后一个不重复元素的位置。以图示说明。

比如输入数组是 [1, 1, 2, 2, 3, 3, 4]
双指针的初始状态如下，上面的指针 prev 记录不重复元素位置，下面指针 curr 遍历数组：

```
|
v
1  --> 1 --> 2 --> 2 --> 3 --> 3 --> 4
       ^
       |
```

比较两个指针指向元素，如果相等，则 prev 不移动， curr 继续遍历。

```
|
v
1  --> 1 --> 2 --> 2 --> 3 --> 3 --> 4
            ^
            |
```

如果两个指针指向元素不等，那么将 prev 右移一位，将 curr 指向元素拷贝至 prev 处，然后 curr 右移一位。

```
       |
       v
1  --> 2 --> 2 --> 2 --> 3 --> 3 --> 4
                  ^
                  |
```

以此类推即可。

算法复杂度

- 时间复杂度：遍历数组，需要 `O(n)` 。
- 空间复杂度： `O(1)` 。

易错点

1. 在数组的不同元素之间赋值时，注意查看下标是否越界，谨慎使用 `num[++prev] = nums[curr++];` 这种表达。
2. 题目最后要求返回的是不同元素的个数，所以最后的下标需要加一。

Java 实现

```java
public class Solution {
    /**
     * @param A: a array of integers
     * @return : return an integer
     */
    public int removeDuplicates(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int prev = 0;
        int curr = 1;
        while (curr < nums.length) {
            if (nums[curr] == nums[prev]) {
                curr++;
            } else {
                prev++;
                nums[prev] = nums[curr];
                curr++;
            }
        }
        return prev + 1;
    }
}
```

另一种形式的 Java 实现

```
public class Solution {
    public int removeDuplicates(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        int prev = 0;
        for (int curr = 0; curr < A.length; curr++) {
            if (A[curr] != A[prev]) {
                A[++prev] = A[curr];
            }
        }
        return prev + 1;
    }
}
```

Remove Duplicates from Sorted Array II

Remove Duplicates from Sorted Array II (leetcode lintcode)

```
Description
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]
.
```

解题思路

参考题目 Remove Duplicates from Sorted Array 的思路，用指针 `curr` 遍历数组，指针 `prev` 记录最后一个不重复元素的位置。考虑到允许同一个元素出现两次，那么 `curr` 指向元素除了与 `prev` 指向元素比较，还需要和 `prev - 1` 指向元素比较。

考虑排序数组的非降特性，所以 `nums[curr] == nums[prev - 1]` 时，也必然有 `nums[prev] == nums[prev]` ，所以，实际上只需要比较 `nums[curr]` 和 `nums[prev - 1]` 即可。

算法复杂度

- 时间复杂度：遍历数组，需要 `O(n)` 。
- 空间复杂度： `O(1)` 。

Java 实现

```
public class Solution {
    /**
     * @param A: a array of integers
     * @return : return an integer
     */
    public int removeDuplicates(int[] nums) {
        if (nums.length < 3) {
            return nums.length;
        }
        int prev = 1;
        int curr = 2;

        while (curr < nums.length) {
            if (nums[curr] == nums[prev] && nums[curr] == nums[p
rev - 1]) {
                curr++;
            } else {
                prev++;
                nums[prev] = nums[curr];
                curr++;
            }
        }
        return prev + 1;
    }
}
```

参考 Remove Duplicates from Sorted Array 题目的解法二，有以下 Java 实现

```
public class Solution {
    /**
     * @param A: a array of integers
     * @return : return an integer
     */
    public int removeDuplicates(int[] nums) {
        // write your code here
        if (nums.length < 3) {
            return nums.length;
        }
        int prev = 1;      // point to previous
        int curr = 2;      // point to current
        for (; curr < nums.length; curr++) {
            if (/* nums[curr] != nums[prev]  ||*/ nums[curr] !=
nums[prev - 1]) {
                nums[++prev] = nums[curr];
            }
        }

        return prev + 1;
    }
}
```

九章算法还提供了另一种更通用的解法

```
public class Solution {
    /**
     * @param A: a array of integers
     * @return : return an integer
     */
     // 该解法较为通用
    public int removeDuplicates(int[] nums) {
        // write your code here
        if(nums == null)
            return 0;
        int cur = 0;
        int i ,j;
        for(i = 0; i < nums.length;){
            int now = nums[i];
            for( j = i; j < nums.length; j++){
                if(nums[j] != now)
                    break;
                if(j-i < 2)     // 可根据题目要求，对重复出现最大次数进行修改。
                    nums[cur++] = now;
            }
            i = j;
        }
        return cur;
    }
}
```

参考

1. Remove Duplicates from Sorted Array II | 九章算法

Minimum Subarray

Minimum Subarray (leetcode lintcode)

```
Description
Given an array of integers, find the subarray with smallest sum.
Return the sum of the subarray.

Notice
The subarray should contain one integer at least.

Example
For [1, -1, -2, 1], return -3.
```

解题思路

一、前缀和

参考 Maximum Subarray 的前缀和方法。

Java 实现

```java
public class Solution {
    /**
     * @param nums: a list of integers
     * @return: A integer indicate the sum of minimum subarray
     */
    public int minSubArray(ArrayList<Integer> nums) {
        if (nums == null || nums.size() == 0) {
            return 0;
        }

        int curSum = 0;
        int minSum = Integer.MAX_VALUE;
        int maxSum = 0;
        for (int i = 0; i < nums.size(); i++) {
            curSum += nums.get(i);
            minSum = Math.min(minSum, curSum - maxSum);
            maxSum = Math.max(maxSum, curSum);
        }

        return minSum;
    }
}
```

二、动规

1. 定义状态：定义一维状态变量 `f[i]` ，表示以第 `i` 个数字 `nums[i]` 为结尾的子数组的最小和。
2. 定义状态转移函数：对于当前状态 `f[i]` ，其上一个状态为 `f[i - 1]` ，
   - 当 `f[i - 1] >= 0` 时，舍弃之前的部分，`f[i]` 取第 `i` 个数字 `nums[i]` 即可。
   - 当 `f[i - 1] < 0` 时，连续求和即可。
   - 根据上述讨论可得 `f[i] = f[i - 1] >= 0 ? nums[i] : f[i - 1] + nums[i]` （或者直接比较两者大小取较小值即可）。考虑到当前状态只取决于上一个状态，所以使用一个变量保存状态可以节省内存开销。
3. 定义起点：
   - `f[0] = 0` 。
4. 定义终点：最终结果即为 `f[i], i = 0,1,...n` 的最小值。可使用一个全局

变量保存局部变量的最小值。

Java 实现

```
public class Solution {
    /**
     * @param nums: a list of integers
     * @return: A integer indicate the sum of minimum subarray
     */
    public int minSubArray(ArrayList<Integer> nums) {
        if (nums == null || nums.size() == 0) {
            return 0;
        }

        int curMin = nums.get(0);
        int minRes = nums.get(0);

        for (int i = 1; i < nums.size(); i++) {
            curMin = Math.min(nums.get(i), curMin + nums.get(i))
;
            minRes = Math.min(curMin, minRes);
        }

        return minRes;
    }
}
```

参考

1. LintCode-Minimum Subarray | LiBlog

Maximum Subarray Difference

Maximum Subarray Difference (leetcode lintcode)

```
Description
Given an array with integers.
Find two non-overlapping subarrays A and B, which |SUM(A) - SUM(
B)| is the largest.
Return the largest difference.

Notice
The subarray should contain at least one number

Example
For [1, 2, -3, 1], return 6.

Challenge
O(n) time and O(n) space.
```

解题思路

本次可参考题目 Maximum Subarray I / II 的做法，使用隔板划分为左右两个子数组，然后依次求取左边数组的最大子数组、最小子数组，右边数组的最大子数组、最小子数组，然后依次比较其差值的绝对值。

这道题目就是体力活。

易错点：

1. 在求局部的最小值 `minSum` 时，比较的是当前的数字 `nums[i]` ，以及 `minSum' + nums[i]` 。比如 `minSum'` 为正数时，可以舍弃。

Java 实现

```
public class Solution {
    /**
     * @param nums: A list of integers
```

```
     * @return: An integer indicate the value of maximum differe
nce between two
     *          Subarrays
     */
    public int maxDiffSubArrays(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int len = nums.length;

        int minSum, maxSum;
        int[] minResLeft = new int[len];
        int[] maxResLeft = new int[len];
        minSum = maxSum = minResLeft[0] = maxResLeft[0] = nums[0
];

        for (int i = 1; i < len; i++) {
            minSum = Math.min(nums[i], minSum + nums[i]);
            minResLeft[i] = Math.min(minResLeft[i - 1], minSum);
            maxSum = Math.max(nums[i], maxSum + nums[i]);
            maxResLeft[i] = Math.max(maxResLeft[i - 1], maxSum);
        }

        int[] minResRight = new int[len];
        int[] maxResRight = new int[len];
        minSum = maxSum = minResRight[len - 1] = maxResRight[len
 - 1] = nums[len - 1];

        for (int i = len - 2; i >= 0; i--) {
            minSum = Math.min(nums[i], minSum + nums[i]);
            minResRight[i] = Math.min(minResRight[i + 1], minSum
);
            maxSum = Math.max(nums[i], maxSum + nums[i]);
            maxResRight[i] = Math.max(maxResRight[i + 1], maxSum
);
        }

        int result = 0;
        for (int i = 0; i < len - 1; i++) {
```

```
            result = Math.max(result, Math.max(Math.abs(minResLe
ft[i] - maxResRight[i + 1]), Math.abs(maxResLeft[i] - minResRigh
t[i + 1])));
        }

        return result;
    }
}
```

参考

Subarray Sum Closest

Subarray Sum Closest (leetcode lintcode)

```
Description
Given an integer array, find a subarray with sum closest to zero
.
Return the indexes of the first number and last number.

Example
Given [-3, 1, 1, -3, 5], return [0, 2], [1, 3], [1, 1], [2, 2] o
r [0, 4].

Challenge
O(nlogn) time
```

解题思路

题目要求的是和的绝对值最小的子数组，结合数字前缀和，子数组和的绝对值最小，等价于两个前缀和的差（的绝对值）最小，结合题目对时间复杂度的要求，对前缀和进行排序，然后寻找差值最小的前缀和即可。

- 建立 `Pair` 数据结构，分别存储索引 `i` 及其对应的前缀和 `sum[i - 1]` 。
- 遍历求数组所有的前缀和，并将其及对应索引加入 `Pair` 。
- 对 `Pair` 中的前缀和排序，此处需定义比较运算符。
- 寻找差值最小的前缀和，并存储对应的索引值。此处需注意索引的大小可能是颠倒的。

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the fir
st number
     *          and the index of the last number
     */
```

```
    class Pair {
        int sum;
        int index;
        public Pair (int sum, int index) {
            this.sum = sum;
            this.index = index;
        }
    }

    public int[] subarraySumClosest(int[] nums) {
        int[] res = new int[2];
        if (nums == null || nums.length == 0) {
            return res;
        }

        int len = nums.length;
        // corner case : if there is only one element
        if (len == 1) {
            res[0] = res[1] = 0;
            return res;
        }

        Pair[] sums = new Pair[len + 1];
        int prev = 0;
        sums[0] = new Pair(0, 0);
        for (int i = 1; i <= len; i++) {
            sums[i] = new Pair(prev + nums[i - 1], i);
            prev = sums[i].sum;
        }

        Arrays.sort(sums, new Comparator<Pair> () {
            public int compare (Pair a, Pair b) {
                return a.sum - b.sum;
            }
        });

        int ans = Integer.MAX_VALUE;
        for (int i = 1; i <= len; i++) {
            if (ans > sums[i].sum - sums[i - 1].sum) {
```

```
                ans = sums[i].sum - sums[i - 1].sum;
                int[] temp = new int[]{sums[i].index - 1, sums[i
 - 1].index - 1};
                Arrays.sort(temp);
                res[0] = temp[0] + 1;
                res[1] = temp[1];
            }
        }

        return res;
    }
}
```

参考

1. Subarray Sum Closest | 九章算法

Median

Median (leetcode lintcode)

```
Given a unsorted array with integers, find the median of it.
A median is the middle number of the array after it is sorted.
If there are even numbers in the array, return the N/2-th number
 after sorted.

Example
Given [4, 5, 1, 2, 3], return 3.
Given [7, 9, 4, 5], return 5.

Challenge
O(n) time.
```

解题思路

最简单的方法是先对数组排序，然后取中位数。快排和归并排序等基于比较的排序算法的时间复杂度为 `O(logn)`，桶排序、计数排序、基数排序等线性排序算法对数据有一定限制，且空间复杂度较高。

由于只需要找出中位数即可，所以可以参考快速排序中的 partition 操作，根据 pivot 元素将数组划分为左小右大的两部分，然后对包含中位数的部分继续查找即可。当 pivot 元素的下标等于数组长度的一半时，即为中位数。

易错点

> 1. 一个数组的中位数位置 `k = (length - 1)/2`。

Java 实现

```
public class Solution {
    /**
     * @param nums: A list of integers.
     * @return: An integer denotes the middle number of the arra
y.
```

```
     */
    public int median(int[] nums) {
        if (nums == null || nums.length == 0) {
            return -1;
        }
        return helper(nums, 0, nums.length - 1, (nums.length - 1
)/2);
    }

    private int helper(int[] nums, int start, int end, int mid)
{
        if (start >= end) {
            return nums[end];
        }
        int m = start;
        for (int i = start + 1; i < end + 1; i++) {
            if (nums[i] < nums[start]) {
                m++;
                swap(nums, i, m);
            }
        }
        swap(nums, start, m);

        if (m == mid) {
            return nums[m];
        } else if (m  > mid) {
            return helper(nums, start, m - 1, mid);
        } else {
            return helper(nums, m + 1, end, mid);
        }
    }

    private void swap(int[] nums, int i, int j) {
        int tmp = nums[i];
        nums[i] = nums[j];
        nums[j] = tmp;
    }
}
```

参考

1. Median | 数据结构与算法/leetcode/lintcode题解

Kth Largest Element

Kth Largest Element (leetcode lintcode)

```
Description
Find K-th largest element in an array.

Notice
You can swap elements in the array

Example
In array [9,3,2,4,8], the 3rd largest element is 4.
In array [1,2,3,4,5], the 1st largest element is 5, 2nd largest element is 4,
3rd largest element is 3 and etc.

Challenge
O(n) time, O(1) extra memory.
```

解题思路

参考题目 Median 的思路。

Java 实现

```java
class Solution {
    /*
     * @param k : description of k
     * @param nums : array of nums
     * @return: description of return
     */
    public int kthLargestElement(int k, int[] nums) {
        // write your code here
        if (nums == null || nums.length == 0 ||
            k <= 0 || k > nums.length) {
            return -1;
        }
```

```
        return helper(nums, 0, nums.length - 1, k - 1);
    }

    private int helper(int[] nums, int start, int end, int k) {
        if (start >= end) {
            return nums[end];
        }

        int prev = start;
        int curt = start + 1;
        int pivot = nums[start];
        for( ; curt <= end; curt++) {
            if (nums[curt] > pivot) {
                prev++;
                swap(nums, prev, curt);
            }
        }
        swap(nums, prev, start);
        if (prev == k) {
            return nums[prev];
        } else if (prev > k) {
            return helper(nums, start, prev - 1, k);
        } else {
            return helper(nums, prev + 1, end, k);
        }
    }

    private void swap(int[] A, int i, int j) {
        int temp = A[i];
        A[i] = A[j];
        A[j] = temp;
    }
};
```

Kth Smallest Number in Sorted Matrix

Kth Smallest Number in Sorted Matrix (lintcode)

```
Description
Find the kth smallest number in at row and column sorted matrix.

Example
Given k = 4 and a matrix:
[
  [1 ,5 ,7],
  [3 ,7 ,8],
  [4 ,8 ,9],
]
return 5

Challenge
Solve it in O(k log n) time where n is the bigger one between ro
w size and column size.
```

解题思路

然后依次将每个数组的最小值放入一个最小堆，从最小堆中取出的第 `k` 个数字就是所求结果。

Java 实现

```java
public class Solution {
    /**
     * @param matrix: a matrix of integers
     * @param k: an integer
     * @return: the kth smallest number in the matrix
     */
    class Node {
        int val, row, col;
        public Node (int val, int row, int col) {
            this.val = val;
```

```
            this.row = row;
            this.col = col;
        }
    }

    public int kthSmallest(int[][] matrix, int k) {
        // write your code here
        if (matrix == null || matrix.length == 0) {
            return -1;
        }

        int n = matrix.length;
        Queue<Node> heap = new PriorityQueue<Node>(n, new Compar
ator<Node>(){
            public int compare(Node a, Node b) {
                return a.val - b.val;
            }
        });

        for (int i = 0; i < n; i++) {
            if (matrix[i].length > 0) {
                int row = i;
                int col = 0;
                int val = matrix[row][col];
                heap.add(new Node(val, row, col));
            }
        }

        for (int i = 0; i < k; i++) {
            Node temp = heap.poll();
            int val = temp.val;
            int row = temp.row;
            int col = temp.col;

            if (i == k - 1) {
                return val;
            }
            if (col < matrix[row].length - 1) {
                col++;
                val = matrix[row][col];
```

```
                    heap.add(new Node(val, row, col));
                }
            }

            return -1;
        }
    }
```

Kth Largest in N Arrays

Kth Largest in N Arrays (lintcode)

```
Description
Find K-th largest element in N arrays.

Notice
You can swap elements in the array

Example
In n=2 arrays [[9,3,2,4,7],[1,2,3,4,8]], the 3rd largest element
 is 7.
In n=2 arrays [[9,3,2,4,8],[1,2,3,4,2]], the 1st largest element
 is 9, 2nd largest element is 8,
3rd largest element is 7 and etc.
```

解题思路

首先将每个一维数组排序，然后依次将每个数组的最大值放入一个最大堆，从最大堆中取出的第 `k` 个数字就是所求结果。题目的关键是在从堆中取出数字后还要将对应一维数组的最大值加入堆，需要新建一个 `Node` 类，来记录数组数值，数组行，数组列。

Java 实现

```java
class Node {
    public int value, row_id, col_id;
    public Node(int _v, int _row, int _col) {
        this.value = _v;
        this.row_id = _row;
        this.col_id = _col;
    }
}
```

```
public class Solution {
    /**
     * @param arrays a list of array
     * @param k an integer
     * @return an integer, K-th largest element in N arrays
     */
    public int KthInArrays(int[][] arrays, int k) {
        // Write your code here
        if (arrays == null || arrays.length == 0 ) {
            return -1;
        }
        int m = arrays.length;

        PriorityQueue<Node> heap =  new PriorityQueue<Node>(k, n
ew Comparator<Node>(){
            public int compare(Node a, Node b) {
                return b.value - a.value;
            }
        });

        for (int i = 0; i < m; i++) {
            Arrays.sort(arrays[i]);
            if (arrays[i].length > 0) {
                int row_id = i;
                int col_id = arrays[i].length - 1;
                int value = arrays[i][col_id];
                heap.add(new Node(value, row_id, col_id));
            }
        }

        for (int i = 0; i < k; i++) {
            Node temp = heap.poll();
            int row_id = temp.row_id;
            int col_id = temp.col_id;
            int value = temp.value;

            if (i == k - 1) {
                return value;
            }
```

```
            if (col_id > 0) {
                col_id--;
                value = arrays[row_id][col_id];
                heap.add(new Node(value, row_id, col_id));
            }
        }

        return -1;
    }
}
```

Spiral Matrix II

Spiral Matrix II (lintcode)

```
Description
Given an integer n, generate a square matrix filled with element
s from 1 to n^2 in spiral order.

Notice
Example
Given n = 3,

You should return the following matrix:
[
  [ 1, 2, 3 ],
  [ 8, 9, 4 ],
  [ 7, 6, 5 ]
]
```

解题思路

矩阵的顺时针遍历，在这里注意有实现的小技巧，像剥洋葱一样遍历矩阵，每次遍历最外围的元素，每次遍历的元素个数都是当前行数/列数减一，在遍历最外围元素后对矩阵的长宽、起始点坐标进行更新。

注意特殊情况，只有一行或一列。

Java 实现

```java
public class Solution {
    /**
     * @param n an integer
     * @return a square matrix
     */
    public int[][] generateMatrix(int n) {
        // Write your code here
        if (n < 0) {
```

```
            return null;
        }

        int num = 1;
        int[][] ans = new int[n][n];
        int x = 0, y = 0;

        while (n > 0) {
            if (n == 1) {
                ans[x][y] = num;
                break;
            }

            for (int i = 0; i < n - 1; i++) {
                ans[x][y++] = num++;
            }
            for (int i = 0; i < n - 1; i++) {
                ans[x++][y] = num++;
            }
            for (int i = 0; i < n - 1; i++) {
                ans[x][y--] = num++;
            }
            for (int i = 0; i < n - 1; i++) {
                ans[x--][y] = num++;
            }
            x++;
            y++;
            n = n - 2;
        }
        return ans;
    }
}
```

Partition Array

Partition Array (leetcode lintcode)

```
Description
Given an array nums of integers and an int k, partition the arra
y
(i.e move the elements in "nums") such that:
All elements < k are moved to the left
All elements >= k are moved to the right
Return the partitioning index, i.e the first index i nums[i] >=
k.

Notice
You should do really partition in array nums instead of
just counting the numbers of integers smaller than k.
If all elements in nums are smaller than k, then return nums.len
gth

Example
If nums = [3,2,2,1] and k=2, a valid answer is 1.

Challenge
Can you partition the array in-place and in O(n)?
```

解题思路

一、反方向双指针

本题目和快速排序是同一类题目，使用指向首尾的两个指针 `left` 和 `right` 向中间遍历，当 `left` 指向对象大于等于 `k` ，`right` 指向对象小于 `k` 时，交换两者，直至 `left > right` 。操作过程要时刻注意两个指针的相对位置，确保不越界。

易错点

1. 在移动过程要确保两根指针的相对位置，不会让数组越界。
2. 两个特殊情况需考虑，数组全部小于 `k` ，数组全部大于 `k` 。

Java 实现

```
public class Solution {
    /**
     *@param nums: The integer array you should partition
     *@param k: As description
     *return: The index after partition
     */
    public int partitionArray(int[] nums, int k) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int left = 0;
        int right = nums.length - 1;

        while (left <= right) {
            while (left <= right && nums[left] < k) {
                left++;
            }
            while (left <= right && nums[right] >= k) {
                right--;
            }

            if (left <= right) {
                int temp = nums[left];
                nums[left] = nums[right];
                nums[right] = temp;
                left++;
                right--;
            }
        }

        return left;
    }
}
```

二、同方向双指针

使用指针 `prev` 记录最后一个小于 `k` 的元素索引，`curt` 遍历数组寻找小于 `k` 的元素。

```
public class Solution {
    /**
     *@param nums: The integer array you should partition
     *@param k: As description
     *return: The index after partition
     */
    public int partitionArray(int[] nums, int k) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        int prev = -1;
        int curt = 0;
        for ( ; curt < nums.length; curt++) {
            if (nums[curt] < k) {
                prev++;
                swap(nums, prev, curt);
            }
        }

        return prev + 1;
    }

    private void swap(int[] nums, int start, int end) {
        int tmp = nums[start];
        nums[start] = nums[end];
        nums[end] = tmp;
    }
}
```

Two Sum

Two Sum (leetcode lintcode)

```
Description
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 suc
h 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.

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

xample
numbers=[2, 7, 11, 15], target=9
return [1, 2]

Challenge
Either of the following solutions are acceptable:
O(n) Space, O(nlogn) Time
O(n) Space, O(n) Time
```

解题思路

一、哈希表

依次将数组元素及对应索引放入哈希表，并判断哈希表中是否存在目标值与数字元素的差值，找到后返回对应索引即可。

算法复杂度

- 时间复杂度：`O(n)`。
- 空间复杂度：`O(n)`。

易错点

> 1. 题目要求返回的是元素的自然顺序，而非从零开始的数组索引，故最终返回的索引要在数组索引基础上引加一。

Java 实现

```
public class Solution {
    /*
     * @param numbers : An array of Integer
     * @param target : target = numbers[index1] + numbers[index2
]
     * @return : [index1 + 1, index2 + 1] (index1 < index2)
     */

    public int[] twoSum(int[] numbers, int target) {
        int[] results = new int[2];
        if (numbers == null || numbers.length == 0) {
            return results;
        }

        HashMap<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < numbers.length; i++) {
            if (map.get(numbers[i]) != null) {
                results[0] = map.get(numbers[i]) + 1;
                results[1] = i + 1;
                return results;
            }
            map.put(target - numbers[i], i);
        }

        return results;
    }
}
```

二、双指针

- 建立 `Pair` 数据结构，分别存储索引 `i` 及其对应的数字元素 `numbers[i - 1]` 。

- 对 `Pair` 中的数字元素排序，此处需定义比较运算符。
- 使用双指针，分别从 `Pair` 的开头和结尾向中间查找。对两个指针对应元素求和。
  - 和大于目标值，说明结尾元素较大，排除掉。
  - 和小于目标值，说明开头元素较小，排除掉。
  - 和等于目标值，返回其对应的索引。

算法复杂度

- 时间复杂度：使用基于比较的排序，所以是 `O(nlogn)` 。
- 空间复杂度：`O(n)` 。

易错点

> 1. `Pair` 结构的声明、赋值。
> 2. 比较运算符的重构。

Java 实现

```
public class Solution {
    /*
     * @param numbers : An array of Integer
     * @param target : target = numbers[index1] + numbers[index2
]
     * @return : [index1 + 1, index2 + 1] (index1 < index2)
     */
    public class Pair {
        int value;
        int index;
        private Pair (int value, int index) {
            this.value = value;
            this.index = index;
        }
    }

    public int[] twoSum(int[] A, int target) {
        int[] ans = new int[2];
        if (A == null || A.length < 2) {
            return ans;
        }
```

```
        Pair[] nums = new Pair[A.length];
        for (int i = 0; i < A.length; i++) {
            nums[i] = new Pair(A[i], i + 1);
        }

        Arrays.sort(nums, new Comparator<Pair>() {
            public int compare (Pair a, Pair b) {
                return a.value - b.value;
            }
        });

        int left = 0, right = A.length - 1;
        while (left < right) {
            int sum = nums[left].value + nums[right].value;
            if (sum > target) {
                right--;
            } else if (sum < target) {
                left++;
            } else {
                ans[0] = Math.min(nums[left].index, nums[right].
index);
                ans[1] = Math.max(nums[left].index, nums[right].
index);
                return ans;
            }
        }
        return ans;
    }
}
```

参考

1. Two Sum | 九章算法

3 Sum

3 Sum (leetcode lintcode)

```
Description
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 zer
o.

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

Example
For example, given array S = {-1 0 1 2 -1 -4}, A solution set is
:
(-1, 0, 1)
(-1, -1, 2)
```

解题思路

一、双指针

三个数之和为零 `a + b + c = 0` ，很容易转化为两个数求和等于一个固定值 `b + c = -a` ，由此本题可转化为题目 two sum 。由于数组中元素可能重复，所以对数组排序后处理会比较方便，而且排序不会增加算法整体的复杂度。在排序后，使用双指针从开头和结尾向中间移动寻找，注意重复元素的处理。

算法复杂度

- 时间复杂度：排序为 `O(nlogn)` ，两重循环寻找三个数为 `O(n^2)` ，所以时间复杂度为 `O(n^2)` 。
- 空间复杂度： `O(n)` 。

易错点

1. 重复元素的处理。
2. 双指针的边界条件限定。

Java 实现

```
public class Solution {
    /**
     * @param numbers : Give an array numbers of n integer
     * @return : Find all unique triplets in the array which gives the sum of zero.
     */
    public ArrayList<ArrayList<Integer>> threeSum(int[] numbers) {
        ArrayList<ArrayList<Integer>> results = new ArrayList<ArrayList<Integer>>();
        if (numbers == null || numbers.length < 3) {
            return results;
        }

        Arrays.sort(numbers);
        for (int i = 0; i < numbers.length - 2; i++) {
            if (i != 0 && numbers[i] == numbers[i - 1]) {
                continue;   // to skip duplicate numbers. e.g. [0,0,0,0]
            }

            int left = i + 1;
            int right = numbers.length - 1;
            while (left < right) {
                int sum = numbers[left] + numbers[right] + numbers[i];
                if (sum == 0) {
                    ArrayList<Integer> tmp = new ArrayList<Integer>();

                    tmp.add(numbers[i]);
                    tmp.add(numbers[left]);
                    tmp.add(numbers[right]);
                    results.add(tmp);
```

```
                    left++;
                    right--;
                    while (left < right && numbers[left] == numb
ers[left - 1]) {
                        left++; // to skip duplicates
                    }
                    while (left < right && numbers[right] == num
bers[right + 1]) {
                        right--;
                    }
                } else if (sum < 0) {
                    left++;
                } else {
                    right--;
                }
            }
        }

        return results;
    }
}
```

参考

1. 3 Sum | 九章算法

3 Sum Closest

3 Sum Closest (leetcode lintcode)

```
Description
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.

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

Example
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).

Challenge
O(n^2) time, O(1) extra space
```

解题思路

一、双指针

本题目的解决方法可参考 3 Sum ，使用双指针，由于不要求返回每个数字的索引，复杂度降低很多，只需要遍历所有三个数字的和，并找到最接近 target 值的和即可。

易错点

> 1. 记得要给数组排序。

Java 实现

```
public class Solution {
    /**
     * @param numbers: Give an array numbers of n integer
     * @param target : An integer
     * @return : return the sum of the three integers, the sum c
losest target.
     */
    public int threeSumClosest(int[] numbers, int target) {
        if (numbers == null || numbers.length < 3) {
            return -1;
        }

        int len = numbers.length;
        int result = Integer.MAX_VALUE;

        Arrays.sort(numbers);
        for (int i = 0; i < len - 2; i++) {
            int left = i + 1;
            int right = len - 1;
            while (left < right) {
                int sum = numbers[left] + numbers[right] + numbe
rs[i];
                result = Math.abs(sum - target) < Math.abs(resul
t - target) ? sum : result;

                if (sum > target) {
                    right--;
                } else {
                    left++;
                }
            }
        }

        return result;
    }
}
```

4 Sum

4 Sum (leetcode lintcode)

```
Description
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.

Notice
Elements in a quadruplet (a,b,c,d) must be in non-descending ord
er. (ie, a ≤ b ≤ c ≤ d)
The solution set must not contain duplicate quadruplets.

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)
```

解题思路

本题目可参考 3 Sum 的解题思路，将 4 Sum 降为 3 Sum 问题，再降为 2 Sum 问题，仍是使用双指针实现。

需要注意的是，数组中有重复元素，所以在遍历时要略过重复元素。

Java 实现

```
public class Solution {
    /**
     * @param numbers : Give an array numbersbers of n integer
     * @param target : you need to find four elements that's sum
 of target
```

```
     * @return : Find all unique quadruplets in the array which
gives the sum of
     *           zero.
     */
    public ArrayList<ArrayList<Integer>> fourSum(int[] numbers,
int target) {
        ArrayList<ArrayList<Integer>> results = new ArrayList<Ar
rayList<Integer>>();
        if (numbers == null || numbers.length < 4) {
            return results;
        }

        int len = numbers.length;
        Arrays.sort(numbers);
        for (int i = 0; i < len - 3; i++) {
            if (i != 0 && numbers[i] == numbers[i - 1]) {
                continue;
            }
            for (int j = i + 1; j < len - 2; j++) {
                if (j != i + 1 && numbers[j] == numbers[j - 1])
{
                    continue;
                }

                int left = j + 1;
                int right = len - 1;
                while (left < right) {
                    int sum = numbers[left] + numbers[right] + n
umbers[i] + numbers[j];
                    if (sum == target) {
                        ArrayList<Integer> tmp = new ArrayList<I
nteger>();

                        tmp.add(numbers[i]);
                        tmp.add(numbers[j]);
                        tmp.add(numbers[left]);
                        tmp.add(numbers[right]);
                        results.add(tmp);
                        left++;
                        right--;
                        while (left < right && numbers[left] ==
```

```
numbers[left - 1]) {
                            left++;
                        }
                        while (left < right && numbers[right] ==
 numbers[right + 1]) {
                            right--;
                        }
                    } else if (sum < target) {
                        left++;
                    } else {
                        right--;
                    }
                }
            }
        }

        return results;
    }
}
```

考虑推广到更普遍的 k-sum 问题，需要使用递归，将问题逐渐降维。

以下实现还有 bug，尚未弄明白哪里有问题。

```
public class Solution {
    /**
     * @param numbers : Give an array numbersbers of n integer
     * @param target : you need to find four elements that's sum
 of target
     * @return : Find all unique quadruplets in the array which
gives the sum of
     *           zero.
     */
    public ArrayList<ArrayList<Integer>> fourSum(int[] A, int ta
rget) {
        ArrayList<ArrayList<Integer>> ans = new ArrayList<ArrayL
ist<Integer>>();
        /*if (A == null || A.length < 4) {
            return ans;
        }*/
```

```
        int n = A.length;
        Arrays.sort(A);

        ArrayList<Integer> sol = new ArrayList<Integer>();
        kSum(A, 0, n - 1, target, 4, ans, sol);
        return ans;
    }

    private void kSum(int[] A, int start, int end,
                      int target, int k,
                      ArrayList<ArrayList<Integer>> ans,
                      ArrayList<Integer> sol) {
        if (k < 0) {
            return;
        }
        if (k == 1) {
            for (int i = start; i <= end; i++) {
                if (A[i] == target) {
                    sol.add(target);
                    ans.add(sol);
                    sol.remove(target);
                    return;
                }
            }
        }
        if (k == 2) {
            twoSum(A, start, end, target, ans, sol);
        }

        for (int i = start; i <= end - k + 1; i++) {
            if (i > start && A[i] == A[i - 1]) {
                continue;
            }
            sol.add(A[i]);
            kSum(A, i + 1, end, target - A[i], k - 1, ans, sol);
            sol.remove(A[i]);
        }

    }
```

```
    private void twoSum(int[] A, int start, int end,
                        int target,
                        ArrayList<ArrayList<Integer>> ans,
                        ArrayList<Integer> sol) {
        while (start < end) {
            int sum = A[start] + A[end];
            if (sum == target) {
                sol.add(A[start]);
                sol.add(A[end]);
                ans.add(sol);
                sol.remove(A[start]);
                sol.remove(A[end]);
                start++;
                end--;
                while (start < end && A[start] == A[start - 1])
{
                    start++;
                }
                while (start < end && A[end] == A[end + 1]) {
                    end--;
                }
            } else if (sum < target) {
                start++;
            } else {
                end--;
            }
        }
    }
}
```

参考

1. [LeetCode] 4Sum |喜刷刷

Two Sum II

Two Sum II (lintcode)

```
Description
Given an array of integers, find how many pairs in the array suc
h that their sum is bigger than a specific target number.
Please return the number of pairs.

Example
Given numbers = [2, 7, 11, 15], target = 24. Return 1. (11 + 15
is the only pair)

Challenge
Do it in O(1) extra space and O(nlogn) time.
```

解题思路

使用对撞型双指针，考虑到原始数组可能是乱序，需要先对数组进行排序。

```java
public class Solution {
    /**
     * @param nums: an array of integer
     * @param target: an integer
     * @return: an integer
     */
    public int twoSum2(int[] nums, int target) {
        // Write your code here
        if (nums == null || nums.length < 2) {
            return 0;
        }
        Arrays.sort(nums);

        int ans = 0;
        int start = 0;
        int end = nums.length - 1;
        while (start < end) {
            int sum = nums[start] + nums[end];
            if (sum > target) {
                ans += end - start;
                end--;
            } else {
                start++;
            }
        }
        return ans;
    }
}
```

Triangle Count

Triangle Count (lintcode)

```
Description
Given an array of integers, how many three numbers can be found
in the array,
so that we can build an triangle whose three edges length is the
 three numbers that we find?

Example
Given array S = [3,4,6,7], return 3. They are:
[3,4,6]
[3,6,7]
[4,6,7]

Given array S = [4,4,4,4], return 4. They are:
[4(1),4(2),4(3)]
[4(1),4(2),4(4)]
[4(1),4(3),4(4)]
[4(2),4(3),4(4)]
```

解题思路

此题目可视为 Two Sum II 的 follow up，三角形的三条边满足——任意两边之和大于第三边，在将数组排序后 `[…a, b, c, ...]`，不难得出 `b + c > a` 和 `a + c > b` 都是必然成立的，只要判断 `a + b > c` 即可。

易错点：

> 1. 记得先给数组排序。

Java 实现

```java
public class Solution {
    /**
     * @param S: A list of integers
     * @return: An integer
     */
    public int triangleCount(int s[]) {
        // write your code here
        if (s == null || s.length < 3) {
            return 0;
        }
        Arrays.sort(s);

        int ans = 0;
        for (int i = 2; i < s.length; i++) {
            int start = 0;
            int end = i - 1;
            while (start < end) {
                int sum = s[start] + s[end];
                if (sum > s[i]) {
                    ans += end - start;
                    end--;
                } else {
                    start++;
                }
            }
        }
        return ans;
    }
}
```

Sort Letters by Case

Sort Letters by Case (leetcode lintcode)

```
Description
Given a string which contains only letters. Sort it by lower cas
e first and upper case second.

Notice
It's NOT necessary to keep the original order of lower-case lett
ers and upper case letters.

Example
For "abAcD", a reasonable answer is "acbAD"

Challenge
Do it in one-pass and in-place.
```

解题思路

思路与 Partition Array 相同，使用指向数组首尾的双指针，向中间遍历。

易错点

1. `Character.isLowerCase()` 函数的使用。

Java 实现

```
public class Solution {
    /**
     *@param chars: The letter array you should sort by Case
     *@return: void
     */
    public void sortLetters(char[] chars) {
        if (chars == null || chars.length == 0) {
            return;
        }

        int left = 0;
        int right = chars.length - 1;
        while (left <= right) {
            while (left <= right && Character.isLowerCase(chars[
left])) {
                left++;
            }
            while (left <= right && Character.isUpperCase(chars[
right])) {
                right--;
            }

            if (left <= right) {
                char temp = chars[left];
                chars[left] = chars[right];
                chars[right] = temp;
            }
        }
        return;
    }
}
```

Sort Colors

Sort Colors (leetcode lintcode)

```
Description
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 colo
r red,
white, and blue respectively.

Notice
You are not suppose to use the library's sort function for this
problem.
You should do it in-place (sort numbers in the original array).

Example
Given [1, 0, 1, 2], sort it in-place to [0, 1, 1, 2].

Challenge
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 foll
owed by 2's.
Could you come up with an one-pass algorithm using only constant
 space?
```

解题思路

本题目“是 E. W. Dijkstra 的 `荷兰国旗问题` 引发的一道经典编程练习，因为这就好像用三种可能的主键值将数组排序一样，这三种主键值对应着荷兰国旗上的三种颜色”。

Dijkstra 采用的解法为“三向切分”法，“从左向右遍历数组一次，维护一个指针 `left` 使得 `nums[0, 1, …, left-1]` 中的元素都小于 `1`，一个指针 `right` 使得 `nums[right+1, … , len-1]` 中的元素都大于 `1`，一个指针 `i` 使得 `nums[left, …, i-1]` 中的元素全部等于 `1`，`nums[i, … right]` 中的元素尚未确定”。开始时 `i` 和 `left` 相等，对 `nums[i]` 进行三向比较处理以下情况：

- `nums[i] < 1`，交换 `nums[left]` 和 `nums[i]`，将 `left` 和 `i` 分别加一；
- `nums[i] > 1`，交换 `nums[right]` 和 `nums[i]`，将 `right` 减一；
- `nums[i] == 1`，将 `i` 加一。

借用 [LeetCode] Sort Colors 中的图来说明会更清楚

```
0......0   1......1   x1 x2 .... xm   2.....2
           |           |            |
         left          i          right
```

易错点

> 1. 循环结束的条件是 `i > right`，这是因为当 `i == right` 时，两者共同指向的对象是未经比较的，有可能是 `0`，这样就需要交换。

Java 实现

```
Description
class Solution {
    /**
     * @param nums: A list of integer which is 0, 1 or 2
     * @return: nothing
     */
    public void sortColors(int[] nums) {
        if (nums == null || nums.length == 0) {
            return;
        }

        int left = 0;
        int right = nums.length - 1;
        int i = left;
        while (i <= right) {
            if (nums[i] < 1) {
                exch(nums, left, i);
                left++;
                i++;
            } else if (nums[i] > 1) {
                exch(nums, i, right);
                right--;
            } else {
                i++;
            }
        }

        return;
    }

    private void exch(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

参考

1. 《算法（第4版）》 2.3 快速排序 / 2.3.3.2 三取样切分
2. [LeetCode] Sort Colors | 喜刷刷

Longest Substring Without Repeating Characters

Longest Substring Without Repeating Characters (lintcode)

```
Description
Given a string, find the length of the longest substring without
 repeating characters.

Example
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.

Challenge
O(n) time
```

解题思路

解法一：标记数组

题目求解的是无重复字母的最长子串，考虑到字母可以使用 ASCII （ 7 bits）表示，可以建立一个数组保存已访问过的字符，然后使用指针 `i` 表示起始位置，`j` 表示当前位置，当 `j` 指向的字符已经被访问时，向前移动 `i` 。

Java 实现

```
public class Solution {
    /**
     * @param s: a string
     * @return: an integer
     */
    public int lengthOfLongestSubstring(String s) {
        // write your code here
        if (s == null || s.length() == 0) {
            return 0;
        }
        // map from character's ASCII to its last occured index
        int[] map = new int[256];
        int i = 0;
        int j = 0;
        int ans = 0;
        for (i = 0; i < s.length(); i++) {
            while (j < s.length() && map[s.charAt(j)] == 0) {
                map[s.charAt(j)] = 1;
                ans = Math.max(ans, j-i+1);
                j++;
            }
            map[s.charAt(i)] = 0;
         }

         return ans;
    }
}
```

解法二：HashSet

相比解法一，稍微有点繁琐。

```
public class Solution {
    /**
     * @param s: a string
     * @return: an integer
     */
    public int lengthOfLongestSubstring(String s) {
        // write your code here
        if (s == null || s.length() == 0) {
            return 0;
        }

        int start = 0;
        int max = 0;

        HashSet<Character> set = new HashSet<Character>();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (!s.contains(c)) {
                set.add(c);
                max = Math.max(max, i - start + 1);
            } else {
                for (int j = start; j < i; j++) {
                    set.remove(s.charAt(j));
                    if (s.charAt(j) == c) {
                        start = j + 1;
                        break;
                    }
                }
            }
            set.add(c);
        }

        return max;
    }
}
```

参考

1. Longest Substring Without Repeating Characters | 九章算法

Clone Graph

Clone Graph (leetcode lintcode)

```
Description
Clone an undirected graph. Each node in the graph contains a lab
el and a list of its neighbors.

How we serialize an undirected graph:
Nodes are labeled uniquely.
We use # as a separator for each node, and ,
as a separator for node label and each neighbor 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 thr
ee parts as separated by #.
- First node is labeled as 0. Connect node 0 to both nodes 1 and
 2.
- Second node is labeled as 1. Connect node 1 to node 2.
- 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:
   1
  / \
 /   \
0 --- 2
     / \
     \_/

Example
return a deep copied graph.
```

解题思路

首先复习一下宽度优先搜索（BFS, Breadth-first Search）的原理：从树或图的某个结点开始，先遍历该结点的所有相邻结点，然后再遍历下一层相邻结点。BFS 的实现通常需要一个辅助队列 queue 保存已遍历结点。

伪代码如下：

```
procedure BFS(G,v) is
      create a queue Q
      create a set V
      add v to V
      enqueue v onto Q
      while Q is not empty loop
         t ← Q.dequeue()
         if t is what we are looking for then
            return t
        end if
        for all edges e in G.adjacentEdges(t) loop
           u ← G.adjacentVertex(t,e)
           if u is not in V then
               add u to V
               enqueue u onto Q
           end if
        end loop
     end loop
     return none
 end BFS
```

一、BFS

由于题目只给出了一个结点，所以要使用 BFS 遍历无向图找到所有结点，和遍历树不同的是，需要使用 `HashSet` 去除重复结点。

- 找到无向图的所有结点。
- 依次拷贝结点（主要是结点值），使用 `HashMap<oldNode, newNode>` 。
- 依次拷贝每个结点的邻居结点。

其实分解后每个步骤都比较容易实现，需要对 `HashSet` 、 `HashMap` 、 `Queue` 的操作比较熟悉。

Java 实现

```
/**
 * Definition for undirected graph.
 * class UndirectedGraphNode {
 *     int label;
 *     ArrayList<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) {
 *         label = x;
 *         neighbors = new ArrayList<UndirectedGraphNode>();
 *      }
 * };
 */
public class Solution {
    /**
     * @param node: A undirected graph node
     * @return: A undirected graph node
     */
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode no
de) {
        if (node == null) {
            return node;
        }

        // use bfs algorithm to traverse the graph and get all n
odes.
        ArrayList<UndirectedGraphNode> nodes = getNodes(node);

        // copy nodes, store the old -> new mapping information
in a hash map
        HashMap<UndirectedGraphNode, UndirectedGraphNode> mappin
g = new HashMap<>();
        for (UndirectedGraphNode oldNode : nodes) {
            mapping.put(oldNode, new UndirectedGraphNode(oldNode
.label));
        }

        // copy neighbors (edges)
        for (UndirectedGraphNode oldNode : nodes) {
            UndirectedGraphNode newNode = mapping.get(oldNode);
            for (UndirectedGraphNode neighbor : oldNode.neighbor
```

```
s) {
                UndirectedGraphNode newNeighbor = mapping.get(ne
ighbor);
                newNode.neighbors.add(newNeighbor);
            }
        }

        return mapping.get(node);
    }

    private ArrayList<UndirectedGraphNode> getNodes(UndirectedGr
aphNode node) {
        Queue<UndirectedGraphNode> queue = new LinkedList<Undire
ctedGraphNode>();
        HashSet<UndirectedGraphNode> set = new HashSet<Undirecte
dGraphNode>();

        queue.offer(node);
        set.add(node);
        while (!queue.isEmpty()) {
            UndirectedGraphNode head = queue.poll();
            for (UndirectedGraphNode neighbor : head.neighbors)
{
                if (!set.contains(neighbor)) {
                    set.add(neighbor);
                    queue.offer(neighbor);
                }
            }
        }

        return new ArrayList<UndirectedGraphNode>(set);
    }
}
```

二、BFS II

可以在 BFS 遍历所有结点时完成拷贝，这样可以将方法一中的三个步骤简化为两个步骤，具体参考以下程序。

Java 实现

```
/**
 * Definition for undirected graph.
 * class UndirectedGraphNode {
 *     int label;
 *     ArrayList<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) {
 *         label = x;
 *         neighbors = new ArrayList<UndirectedGraphNode>();
 *     }
 * };
 */
public class Solution {
    /**
     * @param node: A undirected graph node
     * @return: A undirected graph node
     */
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode no
de) {
        if (node == null) {
            return node;
        }

        ArrayList<UndirectedGraphNode> nodes = new ArrayList<>()
;
        HashMap<UndirectedGraphNode, UndirectedGraphNode> map =
new HashMap<>();

        // clone nodes
        nodes.add(node);
        map.put(node, new UndirectedGraphNode(node.label));

        int start = 0;
        while (start < nodes.size()) {
            UndirectedGraphNode head = nodes.get(start++);
            for (UndirectedGraphNode neigh : head.neighbors) {
                if (!map.containsKey(neigh)) {
                    map.put(neigh, new UndirectedGraphNode(neigh
.label));
                    nodes.add(neigh);
```

```
                }
            }
        }

        // clone neighbors
        for (UndirectedGraphNode oldNode : nodes) {
            UndirectedGraphNode newNode = map.get(oldNode);
            for (UndirectedGraphNode oldNeigh : oldNode.neighbor
s) {
                newNode.neighbors.add(map.get(oldNeigh));
            }
        }

        return map.get(node);
    }
}
```

三、BFS III

相较于方法二，可以在使用 BFS 遍历结点时，同时拷贝结点及结点关系。

Java 实现

```
/**
 * Definition for undirected graph.
 * class UndirectedGraphNode {
 *     int label;
 *     ArrayList<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) {
 *         label = x;
 *         neighbors = new ArrayList<UndirectedGraphNode>();
 *     }
 * };
 */
public class Solution {
    /**
     * @param node: A undirected graph node
     * @return: A undirected graph node
     */
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode no
```

```
de) {
        if (node == null) {
            return node;
        }
        Queue<UndirectedGraphNode> q = new LinkedList<Undirected
GraphNode>();
        HashMap<UndirectedGraphNode, UndirectedGraphNode> map =
new HashMap<>();

        map.put(node, new UndirectedGraphNode(node.label));
        q.add(node);

        while (!q.isEmpty()) {
            UndirectedGraphNode curt = q.poll();
            for (UndirectedGraphNode neighbor : curt.neighbors)
{
                if (!map.containsKey(neighbor)) {
                    // clone nodes
                    map.put(neighbor, new UndirectedGraphNode(ne
ighbor.label));
                    q.add(neighbor);
                }
                // clone neighbors
                map.get(curt).neighbors.add(map.get(neighbor));
            }
        }
        return map.get(node);
    }

}
```

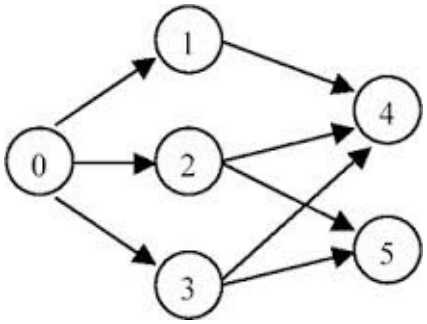
参考

1. Clone Graph | 九章算法
2. Clone Graph leetcode java（DFS and BFS 基础） | 爱做饭的小莹子

Topological Sorting

Topological Sorting (leetcode lintcode)

```
Description
Given an directed graph, a topological order of the graph nodes
is defined as follow:
- For each directed edge A -> B in graph, A must before B in the
 order list.
- The first node in the order can be any node in the graph with
no nodes direct to it.
Find any topological order for the given graph.

Notice
You can assume that there is at least one topological order in t
he graph.

Clarification
Learn more about representation of graphs

Example
For graph as follow:
```



```
The topological order can be:

[0, 1, 2, 3, 4, 5]
[0, 2, 3, 1, 5, 4]
...

Challenge
Can you do it in both BFS and DFS?
```

解题思路

一、BFS

- 将有父结点的结点，及其父结点个数（入度）作为 `key` 和 `value` 存入 `HashMap` 中。
- 找到入度为零的结点，放入队列。
- 将队列中结点取出，将其子结点入度分别减一，更新后入度为零的结点放入队列。

复杂度分析

以 `V` 表示顶点数，`E` 表示有向图中边的数量。

- 时间复杂度：获得结点入度为 `O(V + E)` ，寻找入度为 0 的结点 `O(V)` ，BFS 遍历 `O(V + E)` ，故总的时间复杂度为 `O(V + E)` 。
- 空间复杂度：使用哈希表存储结点，故空间复杂度为 `O(V)` 。

Java 实现

```
/**
 * Definition for Directed graph.
 * class DirectedGraphNode {
 *     int label;
 *     ArrayList<DirectedGraphNode> neighbors;
 *     DirectedGraphNode(int x) {
 *         label = x;
 *         neighbors = new ArrayList<DirectedGraphNode>();
 *     }
 * };
```

```
 */
public class Solution {
    /**
     * @param graph: A list of Directed graph node
     * @return: Any topological order for the given graph.
     */
    public ArrayList<DirectedGraphNode> topSort(ArrayList<Direct
edGraphNode> graph) {
        ArrayList<DirectedGraphNode> results = new ArrayList<Dir
ectedGraphNode>();
        if (graph == null) {
            return results;
        }

        // Map.Entry = <Node N, numbers of parents(in-degree) fo
r N>
        HashMap<DirectedGraphNode, Integer> map = new HashMap<>(
);
        for (DirectedGraphNode node : graph) {
            for (DirectedGraphNode neighbor : node.neighbors) {
                if (map.containsKey(neighbor)) {
                    map.put(neighbor, map.get(neighbor) + 1);
                } else {
                    map.put(neighbor, 1);
                }
            }
        }

        // add nodes without parents to the queue
        Queue<DirectedGraphNode> queue = new LinkedList<Directed
GraphNode>();
        for (DirectedGraphNode node : graph) {
            if (!map.containsKey(node)) {
                queue.offer(node);
                results.add(node);
            }
        }

        // poll a node from the queue and update # parents (-1)
for its children
```

```
        while (!queue.isEmpty()) {
            DirectedGraphNode node = queue.poll();
            for (DirectedGraphNode n : node.neighbors) {
                map.put(n, map.get(n) - 1);
                //add nodes without parents to the queue
                if (map.get(n) == 0) {
                    results.add(n);
                    queue.offer(n);
                }
            }
        }
        return results;
    }
}
```

二、DFS

参考

1. Topological Sorting | 九章算法
2. Topological Sorting | 数据结构与算法/leetcode/lintcode题解

Subsets

Subsets (leetcode lintcode)

```
Given a set of distinct integers, return all possible subsets.

Notice
- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

Example
If S = [1,2,3], a solution is:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]

Challenge
Can you do it in both recursively and iteratively?
```

解题思路

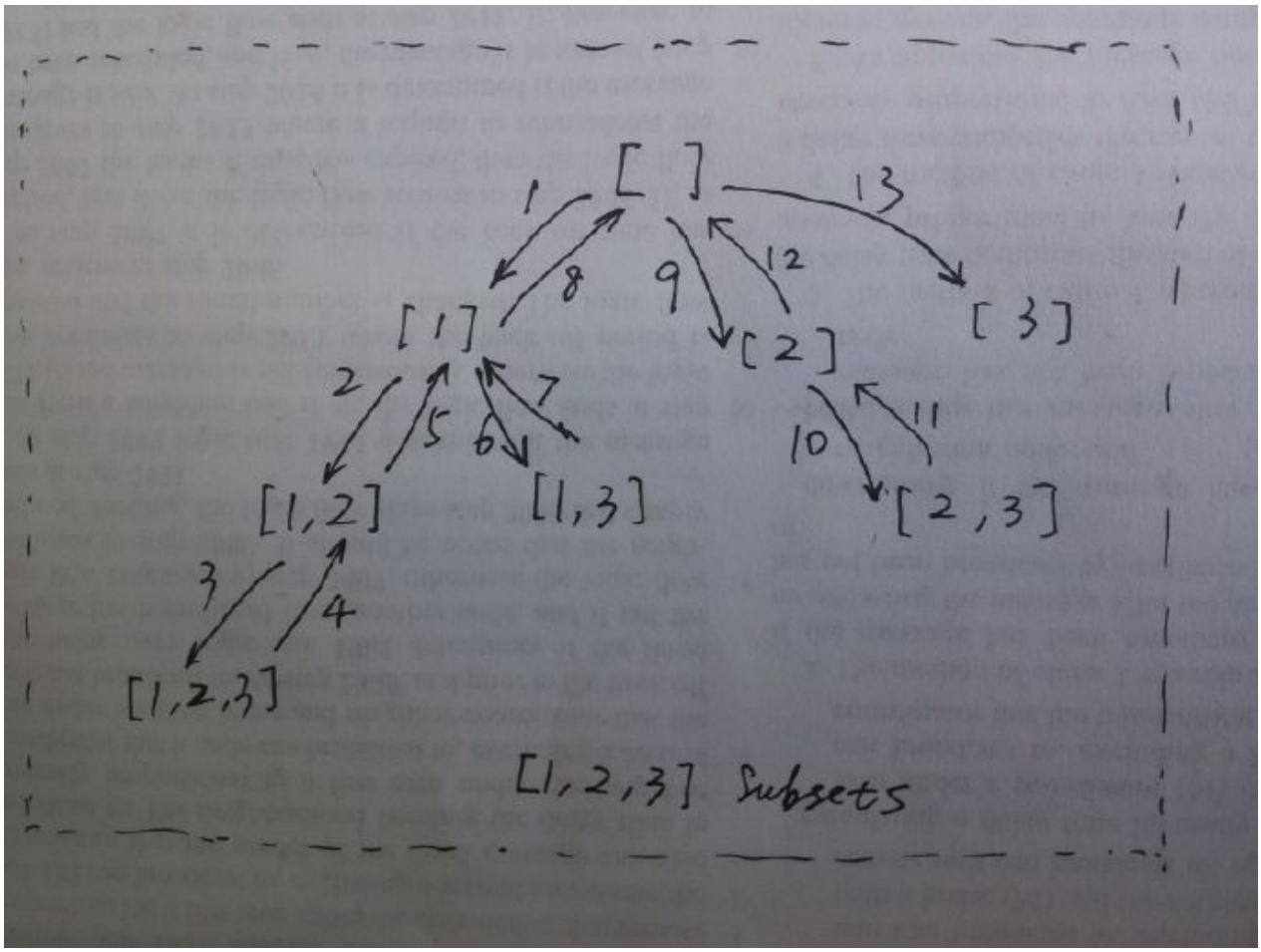
一、DFS

要求一是子集中元素为升序，故先对原数组进行排序。要求二是子集不能重复，于是将原题转化为数学中的组合问题，使用深度优先搜索（ `DFS` ）进行穷举求解。

借用参考链接 1 中的解释，具体穷举过程可以用图示和函数运行的堆栈图理解，以数组 `[1, 2, 3]` 为例进行分析，下图所示为 `list` 及 `result` 动态变化的过程，箭头向下表示 `list.add` 及 `result.add` 操作，箭头向上表

示 `list.remove` 操作。为了确保所有的情况都能够遍历到，在 `list` 加入一个元素后，还需要删除该元素以恢复原状态。

函数 `dfs(result, list, nums, 0)` 则表示将以 `list` 开头的所有组合全部加入 `result` 。当 `list` 是 `[1]` 时，对应图中步骤 2~7 ，依次将 `[1, 2]`, `[1, 2, 3]`, `[1, 3]` 全部添加到 `result` 中。



算法复杂度

- 时间复杂度：在本题中解的个数为 `2^n` ，产生一个解的复杂度最坏可以认为是 `n` ，故算法时间复杂度的上界可以认为是 `O(n* 2^n)` 。
- 空间复杂度：使用临时空间 `list` 保存中间结果， `list` 最大长度为数组长度，故空间复杂度近似为 `O(n)` 。

易错点

> 1. 记得给原始数组排序，以确保最终子数组是非降的。

Java 实现：

```
class Solution {
    /**
     * @param nums: A set of numbers.
     * @return: A list of lists. All valid subsets.
     */
    public ArrayList<ArrayList<Integer>> subsets(int[] nums) {

        ArrayList<ArrayList<Integer>> result = new ArrayList<Arr
ayList<Integer>>();
        // 注意：此处int型数组取长度无需括号()
          if (nums == null || nums.length == 0) {
            result.add(new ArrayList<Integer>());  // when nums
= [], return [[]]
            return result;
        }
        ArrayList<Integer> list = new ArrayList<Integer>();
        Arrays.sort(nums);

        dfs(result, list, nums, 0);
        return result;
    }

    private void dfs(ArrayList<ArrayList<Integer>> result,
                     ArrayList<Integer> list, int[] nums, int po
s) {

        result.add(new ArrayList<Integer>(list));

        for (int i = pos; i < nums.length; i++) {
            list.add(nums[i]);
            dfs(result, list, nums, i + 1);
            list.remove(list.size() - 1);   //移除最后一个元素
        }

    }
}
```

二、Bit Map

一共 `2^n` 个子集，每个子集对应 `0 … 2^n - 1` 之间的一个二进制整数，该整数一共 `n` 个 `bit` 位，用第 `i` 个 `bit` 位的取值 `1` 或 `0` 表示 `nums[i]` 在或不在集合中，我们只需遍历完所有的数字——对应所有的 `bit` 位可能性（外循环），然后转化为对应的数字集合——判断数字每一个 `bit` 的取值（内循环）——即可。

```
以nums = [1,2,3]举例
0 -> 000 -> null                      -> []
1 -> 001 -> nums[0]                   -> [1]
2 -> 010 -> nums[1]                   -> [2]
3 -> 011 -> nums[0],nums[1]           -> [1,2]
4 -> 100 -> nums[2]                   -> [3]
5 -> 101 -> nums[1],nums[2]           -> [1,3]
6 -> 110 -> nums[1],nums[2]           -> [2,3]
7 -> 111 -> nums[0],nums[1],nums[2] -> [1,2,3]
```

Java 实现：

```
class Solution {
    /**
     * @param nums: A set of numbers.
     * @return: A list of lists. All valid subsets.
     */
    public ArrayList<ArrayList<Integer>> subsets(int[] nums) {
        // write your code here
        if (nums == null || nums.length == 0) {
            return new ArrayList<ArrayList<Integer>>();
        }

        ArrayList<ArrayList<Integer>> result = new ArrayList<Arr
ayList<Integer>>();
        int n =  nums.length;
        Arrays.sort(nums);  // 注意函数使用是Arrays.sort()
        // 1 << n = 100...0 , i的取值范围是0 ~ 2^n -1
        // i用来遍历所有的比特位可能性
        for (int i = 0; i < (1 << n); i++) {
            ArrayList<Integer> subset = new ArrayList<Integer>()
;
            for (int j = 0; j < n; j++) {
                // j的取值范围是0 ~ n
                // j用来判断n个bit位的取值是否为1
                if ((i & (1 << j)) != 0) {
                    subset.add(nums[j]);
                }
            } // for j
            result.add(subset);
        } // for i
        return result;
    }

}
```

参考

1. Subsets - 子集 | 数据结构与算法/leetcode/lintcode题解
2. Subsets | 九章算法

3. LeetCode: Subsets 解题报告 | Yu's garden

Subsets II

Subsets II (leetcode lintcode)

```
Description
Given a list of numbers that may has duplicate numbers, return a
ll possible subsets

Notice
- Each element in a subset must be in non-descending order.
- The ordering between two subsets is free.
- The solution set must not contain duplicate subsets.

Example
If S = [1,2,2], a solution is:
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]

Challenge
Can you do it in both recursively and iteratively?
```

解题思路

一、DFS

题目与 Subsets 有关，解题思路可以参考 Subsets 的解法，但是需要考虑如何删除重复的子集。

以 `{1, 2(1), 2(2), 2(3)}` 为例，`{1, 2(1)}` 和 `{1, 2(2)}` 重复，`{1, 2(1), 2(2)}` 和 `{1, 2(2), 2(3)}` 重复。可以看到，对于重复元素，我们只关心取了几个，不关心取哪几个。

解决方案是从重复元素的第一个持续向下添加列表，而不能取第二个或之后的重复元素。换句话讲就是重复的元素必须连续选取。该方法很巧妙，建议使用图示以及函数运行的堆栈图理解。

算法复杂度

- 时间复杂度：在本题中解的个数为 `2^n` ，产生一个解的复杂度最坏可以认为是 `n` ，故算法时间复杂度的上界可以认为是 `O(n* 2^n)` 。
- 空间复杂度：使用临时空间 `list` 保存中间结果，`list` 最大长度为数组长度，故空间复杂度近似为 `O(n)` 。

Java 实现：

```
class Solution {
    /**
     * @param S: A set of numbers.
     * @return: A list of lists. All valid subsets.
     */
    public ArrayList<ArrayList<Integer>> subsetsWithDup(ArrayLis
t<Integer> S) {
        // write your code here
        ArrayList<ArrayList<Integer>> result = new ArrayList<Arr
ayList<Integer>>();
        if (S == null || S.size() == 0) {
            return result;
        }

        Collections.sort(S);
        ArrayList<Integer> list = new ArrayList<Integer>();

        dfs(result, list, S, 0);
        return result;
    }

    private void dfs(ArrayList<ArrayList<Integer>> result,
                     ArrayList<Integer> list, ArrayList<Integer>
```

```
  S, int pos) {

        result.add(new ArrayList(list));

        for (int i = pos; i < S.size(); i++) {
            // 遇到重复的元素（已排序），必须从第一个开始连续取
            // 使用i-1是为了防止索引越界
            if (i != pos && S.get(i) == S.get(i - 1)) {
                continue;
            }
            list.add(S.get(i));
            dfs(result, list, S, i + 1);
            list.remove(list.size() - 1);
        }

    }
}
```

参考

1. Unique Subsets | 数据结构与算法/leetcode/lintcode题解
2. Subsets II | 九章算法

Permutations

Permutations (leetcode lintcode)

```
Description
Given a list of numbers, return all possible permutations.

Notice
You can assume that there is no duplicate numbers in the list.

Example
For nums = [1,2,3], the permutations are:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]

Challenge
Do it without recursion.
```

解题思路

一、DFS

`Permutation` 要求遍历得到所有长度为 `n` 的排列，与 `Subsets` 的实现相比较，有两点不一样的地方：一是只有在长度为 `n` 时才会记录；二是不再要求升序，所有排序都必须遍历到。

算法复杂度

- 时间复杂度：一般而言，搜索问题的复杂度可以这么计算 `O(解的个数 * 产生解的复杂度)`。在本题中解的个数为 `n!`，产生一个解的复杂度最坏可以认为

是 `n`，故算法渐进性能的上界可以认为是 `O(n*n!)`。这里的时间复杂度并未考虑查找 `list` 中是否包含重复元素的 `contain` 操作。
- 空间复杂度：使用临时空间 `list` 保存中间结果，`list` 最大长度为数组长度，故空间复杂度近似为 `O(n)` 。

易错点：

> 1. 在往 `result` 中添加结果时要新建一个 `list` 的拷贝（深拷贝），这是因为 Java 中所有对象都是引用，如果添加的是 `list`，在主程序中修改 `list` 时， `result` 中的 `list` 对象也会被修改。

Java 实现

```
public class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>
>();
        if (nums == null || nums.length == 0) {
            result.add(new ArrayList<Integer>());  // when nums
= [], return [[]]
            return result;
        }

        List<Integer> list = new ArrayList<Integer>();
        // 把以 list 开头的所有排列全部放到 result 中
        // 把以空{}为开头的所有排列全部放到 result 中
        // 把以{1},{2},{3}为开头的所有排列全部放到 result 中...
        dfs(result, list, nums);
        return result;
    }

    private void dfs (List<List<Integer>> result,
                      List<Integer> list,
                      int[] nums) {
        if (list.size() == nums.length) {
            result.add(new ArrayList<Integer>(list));
            return;
        }

        for (int i = 0; i < nums.length; i++) {
            if (list.contains(nums[i])) {
                continue;
            }
            list.add(nums[i]);
            dfs(result, list, nums);
            list.remove(list.size() - 1);
        }
    }
}
```

二、插入法（非递归）

以题目中的输入数组 `[1, 2, 3]` 举例说明：

- 当只有 `1` 时候：`[1]`
- 当加入 `2` 以后：`[2, 1], [1, 2]`
- 当加入 `3` 以后：`[3, 2, 1], [2, 3, 1], [2, 1, 3]; [3, 1, 2], [1, 3, 2], [1, 2, 3]`

易错点：

> 1. 在新建 `List` 类型变量时要使用 `ArrayList` 或其他类型初始化，在赋值时同样。

Java 实现

```
public class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();

        // start from an empty list
        result.add(new ArrayList<Integer>());

        for (int i = 0; i < nums.length; i++) {
            // list of list in current iteration of the array num

            List<List<Integer>> current = new ArrayList<>();

            for (List<Integer> list : result) {
                // # of locations to insert is largest index + 1
                for (int j = 0; j < list.size() + 1; j++) {
                    // + add nums[i] to different locations
                    list.add(j, nums[i]);
                    current.add(new ArrayList<Integer>(list));

                    // - remove nums[i] add
                    list.remove(j);
                }
            }
            // carefull for the copy
            result = new ArrayList<>(current);
        }
        return result;
    }
}
```

参考

1. Permutations | 数据结构与算法/leetcode/lintcode题解
2. Permutations | 九章算法
3. permutation 和subset的时间复杂度 | 九章问答
4. LeetCode – Permutations (Java) | Program Creek

Permutations II

Permutations II (leetcode lintcode)

```
Description
Given a list of numbers with duplicate number in it. Find all un
ique permutations.

Example
For numbers [1,2,2] the unique permutations are:
[
  [1,2,2],
  [2,1,2],
  [2,2,1]
]

Challenge
Using recursion to do it is acceptable. If you can do it without
 recursion, that would be great!
```

解题思路

在 Permutation 题目的基础上，增加了对重复元素的处理，此处可参考 Subsets II 题目中对重复元素的处理。即，最终要达到的目的是，出现重复的数字，原来排在前面的，最终结果中也排在前面，没有顺序上的变化也就不会产生重复的子集。

由于重复元素的存在，且每次遍历都会从第一个元素开始，所以需要给每个数组元素增加一个标志位，表示是否已经加入到 `list` 中。两种情况需要剪枝，即略过不符合要求的结果。一是该元素已经在 `list` 中，通过标志位判断；二是该元素和前一个元素相等，但前一个元素不在 `list` 中，如果加入该元素会打乱重复元素的选取顺序。

Java 代码

```
class Solution {
    /**
```

```
     * @param nums: A list of integers.
     * @return: A list of unique permutations.
     */
    public List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>
>();
        if (nums == null || nums.length == 0) {
            result.add(new ArrayList<Integer>());
            return result;
        }

        Arrays.sort(nums);
        ArrayList<Integer> sol = new ArrayList<Integer>();
        boolean[] visited = new boolean[nums.length];
        dfs(result, sol, visited, nums);

        return result;
    }

    private void dfs(List<List<Integer>> result,
                     ArrayList<Integer> sol,
                     boolean[] visited,
                     int[] nums) {
        if (sol.size() == nums.length ) {
            result.add(new ArrayList<Integer>(sol));
            return;
        }

        for (int i = 0; i < nums.length; i++) {
            // 相同的数字，原来排在前面的，在结果中也应该排在前面，这样就
保证了唯一性
            // 所以当前面的数字未使用时，后面的数字也不应被使用
            if (visited[i] || (i != 0 && nums[i] == nums[i - 1]
&& !visited[i - 1])) {
                continue;
            }
            visited[i] = true;
            sol.add(nums[i]);
            dfs(result, sol, visited, nums);
            sol.remove(sol.size() - 1);
```

```
                visited[i] = false;
            }
        }
    }
```

参考

1. Permutations II | 九章算法

Binary Tree Path Sum

Binary Tree Path Sum (lintcode)

```
Description
Given a binary tree, find all paths that sum of the nodes in the
 path equals to a given number target.
A valid path is from root node to any of the leaf nodes.

Example
Given a binary tree, and target = 5:
     1
    / \
   2   4
  / \
 2   3

return
[
  [1, 2, 2],
  [1, 4]
]
```

解题思路

跟 Permutation 题目类似，需要注意的是每次进入缩小规模都需要考虑两种情况——左子树和右子树，所以传递路径参数时需要新建对象，以免发生混乱。

Java 实现

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
```

```
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param root the root of binary tree
     * @param target an integer
     * @return all valid paths
     */
    public List<List<Integer>> binaryTreePathSum(TreeNode root,
int target) {
        // Write your code here
        List<List<Integer>> result = new ArrayList<List<Integer>
>();
        if (root == null) {
            return result;
        }
        List<Integer> list = new ArrayList<Integer>();
        helper(result, list, root, target);
        return result;
    }

    private void helper(List<List<Integer>> result, List<Integer
> list,
                        TreeNode root, int target) {
        if (root == null ) {
            return;
        }

        list.add(root.val);
        if (target == root.val && root.left == null && root.righ
t == null) {
            result.add(new ArrayList<Integer>(list));
            return;
        }
        List<Integer> listLeft = new ArrayList<Integer>(list);
        List<Integer> listRight = new ArrayList<Integer>(list);
        helper(result, listLeft, root.left, target - root.val);
        helper(result, listRight, root.right, target - root.val)
```

```
;
            list.remove(list.size() - 1);
        }
    }
```

N-Queens

N-Queens (leetcode lintcode)

```
Description
The n-queens puzzle is the problem of placing n queens on an n×n
 chessboard
such that no two queens attack each other.
Given an integer n, return all distinct solutions to the n-queen
s puzzle.
Each solution contains a distinct board configuration of the n-q
ueens' placement,
where 'Q' and '.' both indicate a queen and an empty space respe
ctively.

Example
There exist two distinct solutions to the 4-queens puzzle:
[
  // Solution 1
  [".Q..",
   "...Q",
   "Q...",
   "..Q."
  ],
  // Solution 2
  ["..Q.",
   "Q...",
   "...Q",
   ".Q.."
  ]
]

Challenge
Can you do it without recursion?
```

解题思路

在国际象棋中，皇后是战斗力最强的一个棋子，可以横、竖、斜走，吃子和走法相同。

本题目的问题是如何在一个 `n * n` 的棋盘上面摆放 `n` 个皇后，使得任何一个皇后都无法直接吃掉其他的皇后？根据国际象棋的规则，要达到要求，任意两个皇后都不能处在同一行、同一列、同一斜线上面。

以 `4 * 4` 棋盘为例，将解决方案中每一行皇后所在列放入一维数组中，两个解决方案分别为 `[2, 4, 1, 3]` 和 `[3, 1, 4, 2]` ，数组索引表示皇后所在行，数组元素表示皇后所在列，不难发现，这其实是 `[1, 2, 3, 4]` 全排列的一部分。所以问题可以转化为①求解所有全排列；②判断全排列是否满足规则要求。

全排列问题使用 `DFS` 搜索即可，我们来看一下如何判断摆放位置是否符合要求。

- 一、不能在同一行，考虑到一维数组索引表示行，所以这个条件是必然满足的；
- 二、不能在同一列，需要将已存入数组的每一个元素值与即将存入的元素比较，两者不同时才可以存入；
- 三、不能在同一斜线，分为两种，左上到右下，右上到左下，如下图所示，两种情况下“斜率”的绝对值都是 `1` ，以此作为判断依据。（注：这里的斜率并不是在常规的笛卡尔坐标系）

1, 1	**1, 2**	**1, 3**	**1, 4**
2, 1	**2, 2**	**2, 3**	**2, 4**
3, 1	**3, 2**	**3, 3**	**3, 4**
4, 1	**4, 2**	**4, 3**	**4, 4**

Java 实现

```
class Solution {
    /**
     * Get all distinct N-Queen solutions
     * @param n: The number of queens
     * @return: All distinct solutions
     * For example, A string '...Q' shows a queen on forth posit
ion
     */
    ArrayList<ArrayList<String>> solveNQueens(int n) {
        ArrayList<ArrayList<String>> result = new ArrayList<>();
```

```
        if (n <= 0) {
            return result;
        }

        search(n, new ArrayList<Integer>(), result);
        return result;
    }

    private void search (int n,
                        ArrayList<Integer> cols,
                        ArrayList<ArrayList<String>> result) {
        if (cols.size() == n) {
            result.add(drawChessboard(cols));
            return;
        }
        // cols: index i is row-coordinate && cols.get(i) is col
umn-coordinate
        // for col : the coordinate is (cols.size(), col)
        // col: [0...n-1] form a permutation of column
        for (int col = 0; col < n; col++) {
            if (!isValid(cols, col)) {
                continue;
            }
            cols.add(col);
            search(n, cols, result);
            cols.remove(cols.size() - 1);
        }
    }

    private ArrayList<String> drawChessboard (ArrayList<Integer>
 cols) {
        ArrayList<String> chessboard = new ArrayList<>();
        for (int i = 0; i < cols.size(); i++) {
            StringBuilder cb = new StringBuilder();
            for (int j = 0; j < cols.size(); j++) {
                if (j == cols.get(i)) {
                    cb.append('Q');
                } else {
                    cb.append('.');
                }
```

```
            }
            chessboard.add(cb.toString());
        }
        return chessboard;
    }

    private boolean isValid (ArrayList<Integer> cols, int col) {
        int row = cols.size();
        // queen's coodinate in cols [i, cols.get(i)] vs queen t
o be added [row, col]
        for (int i = 0; i < row; i++) {
            // same column
            if (cols.get(i) == col) {
                return false;
            }
            // left-top to right-bottom && right-top to left-bot
tom
            if (Math.abs(i - row) == Math.abs(cols.get(i) - col)
) {
                return false;
            }
        }
        return true;
    }

}
```

参考

1. N-Queens | 九章算法
2. N-Queens leetcode java | 爱做饭的小莹子

N-Queens II

N-Queens II (leetcode lintcode)

```
Description
Follow up for N-Queens problem.
Now, instead outputting board configurations, return the total n
umber of distinct solutions.

Example
For n=4, there are 2 distinct solutions.
```

解题思路

实现思路参考问题 N-Queens ，不必输出具体结果，只需计算结果数量即可，这里需要一个全局变量。

Java 实现

```java
class Solution {
    /**
     * Calculate the total number of distinct N-Queen solutions.
     * @param n: The number of queens.
     * @return: The total number of distinct solutions.
     */
    public int result;
    public int totalNQueens(int n) {
        if (n <= 0) {
            return 0;
        }

        result = 0;
        search(n, new ArrayList<Integer>());
        return result;
    }

    private void search (int n, ArrayList<Integer> cols) {
```

```
        if (cols.size() == n) {
            result++;
            return;
        }
        // cols: index i is row-coordinate && cols.get(i) is col
umn-coordinate
        // for col : the coordinate is (cols.size(), col)
        // col: [0...n-1] form a permutation of column
        for (int col = 0; col < n; col++) {
            if (!isValid(cols, col)) {
                continue;
            }

            cols.add(col);
            search(n, cols);
            cols.remove(cols.size() - 1);
        }
    }

    private boolean isValid (ArrayList<Integer> cols, int col) {
        int row = cols.size();
        // queen‘s coodinate in cols [i, cols.get(i)] vs queen t
o be added [row, col]
        for (int i = 0; i < row; i++) {
            // same column
            if (cols.get(i) == col) {
                return false;
            }
            // same left-top to right-bottom && left-bottom to r
ight-top
            if (Math.abs(i - row) == Math.abs(cols.get(i) - col)
) {
                return false;
            }
        }
        return true;
    }
};
```

Palindrome Partitioning

Palindrome Partitioning (leetcode lintcode)

```
Description
Given a string s, partition s such that every substring of the p
artition is a palindrome.
Return all possible palindrome partitioning of s.

Example
Given s = "aab", return:
[
  ["aa","b"],
  ["a","a","b"]
]
```

解题思路

一般而言，如果题目要求所有的组合、排列、结果，都是用搜索（DFS + backtracking）来做。先来分析一下题目可能的解有多少个，如果字符串 `s` 共有 `n` 个字符，那么 `n` 个字符之间可以放置 `n - 1` 块隔板，每块隔板可能放也可能不放，所以可得到一共 `2 ^ (n-1)` 种划分情况，接下来要做的是判断哪些划分结果符合回文。

> s = " x1 | x2 | x3 | ... | x(n-1) | xn "

算法复杂度

- 时间复杂度：最多有 `2^(n-1)` 个结果，遍历得到每个结果的最坏时间开销是 `O(n)` ，所以是 `O(n * 2^(n-1))` 。

Java 实现

```
public class Solution {
    /**
     * @param s: A string
     * @return: A list of lists of string
```

```
     */
    public List<List<String>> partition(String s) {
        List<List<String>> result = new ArrayList<>();
        if (s == null) {
            return result;
        }

        List<String> path = new ArrayList<String>();
        helper(s, path, 0, result);

        return result;
    }

    private void helper(String s,
                        List<String> path,
                        int pos,
                        List<List<String>> result) {
        if (pos == s.length()) {
            result.add(new ArrayList<String>(path));
            return;
        }

        for (int i = pos; i < s.length(); i++) {
            String prefix = s.substring(pos, i + 1);
            if (!isPalindrome(prefix)) {
                continue;
            }

            path.add(prefix);
            helper(s, path, i + 1, result);
            path.remove(path.size() - 1);
        }
    }

    private boolean isPalindrome(String s) {
        int start = 0;
        int end = s.length() - 1;
        while (start < end) {
            if (s.charAt(start) != s.charAt(end)) {
                return false;
```

```
                }

                start++;
                end--;
            }
            return true;
        }
    }
```

参考

1. Palindrome Partitioning | 九章算法

Combinations

Combinations (leetcode lintcode)

```
Description
Given two integers n and k, return all possible combinations of
k numbers out of 1 ... n.

Example
For example,
If n = 4 and k = 2, a solution is:
[[2,4],[3,4],[2,3],[1,2],[1,3],[1,4]]
```

解题思路

Subsets 的变形题，找到所有长度为 `k` 的子集。

Java 实现

```
public class Solution {
    /**
     * @param n: Given the range of numbers
     * @param k: Given the numbers of combinations
     * @return: All the combinations of k numbers out of 1..n
     */
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> result = new ArrayList<List<Integer>
>();
        if (n < 1 || k < 1 || n < k) {
            return result;
        }

        List<Integer> path = new ArrayList<Integer>();
        dfs(result, path, n, k, 1);
        return result;
    }

    private void dfs(List<List<Integer>> result, List<Integer> p
ath,
                     int n, int k, int pos) {
        if (path.size() == k) {
            result.add(new ArrayList<Integer>(path));
            return;
        }

        for (int i = pos; i <= n; i++) {
            path.add(i);
            dfs(result, path, n, k, i + 1);
            path.remove(path.size() - 1);
        }
    }
}
```

Combination Sum

Combination Sum (leetcode lintcode)

```
Description
Given a set of candidate numbers (C) and a target number (T),
find all unique combinations in C where the candidate numbers su
ms to T.
The same repeated number may be chosen from C unlimited number o
f times.

For example, given candidate set 2,3,6,7 and target 7,
A solution set is:
[7]
[2, 2, 3]

Notice
All numbers (including target) will be positive integers.
Elements in a combination (a1, a2, … , ak) must be in non-descen
ding order. (ie, a1 ≤ a2 ≤ … ≤ ak).
The solution set must not contain duplicate combinations.

Example
given candidate set 2,3,6,7 and target 7,
A solution set is:
[7]
[2, 2, 3]
```

解题思路

本题与 Subsets 不同之处在于同一个数组元素可以重复使用，以 `[2, 3, 6, 7]` 举例，第一个放入 `list` 中的是首元素 `2` ，下一次递归调用仍将 `2` 放入 `list` 中，直至 `list` 的和大于目标值，然后依次取出 `list` 队尾的 `2` ，开始将 `3` 放入，以此类推。

易错点

> 1. 原始数组中可能有重复元素，所以需要先将原始数组排序，在函数中增加去重处理。

Java 实现

```
public class Solution {
    /**
     * @param candidates: A list of integers
     * @param target:An integer
     * @return: A list of lists of integers
     */
    public List<List<Integer>> combinationSum(int[] candidates,
int target) {
        List<List<Integer>> result = new ArrayList<>();
        if (candidates == null || candidates.length == 0 || targ
et <= 0) {
            return result;
        }

        List<Integer> list = new ArrayList<>();
        Arrays.sort(candidates);
        search(candidates, target, 0, list, result);
        return result;
    }

    private void search (int[] candidates,
                        int target,
                        int pos,
                        List<Integer> list,
                        List<List<Integer>> result) {

        int count = sum(list);
        if (count == target) {
            result.add(new ArrayList<Integer>(list));
            return;
        } else if (count > target) {
            return;
        }

        for (int i = pos; i < candidates.length; i++) {
```

```
            if (i != pos && candidates[i] == candidates[i - 1])
{
                continue;
            }
            list.add(candidates[i]);
            search(candidates, target, i, list, result);
            list.remove(list.size() - 1);
        }
    }

    private int sum(List<Integer> list) {
        if (list == null) {
            return 0;
        }

        int count = 0;
        for (int i = 0; i < list.size(); i++) {
            count += list.get(i);
        }

        return count;
    }
}
```

九章算法提供的解答将 `list` 求和操作融合在递归函数中，感觉更简洁，供参考。

Java 实现

```
public class Solution {
    /**
     * @param candidates: A list of integers
     * @param target:An integer
     * @return: A list of lists of integers
     */
    public List<List<Integer>> combinationSum(int[] candidates,
int target) {
        List<List<Integer>> result = new ArrayList<>();
        if (candidates == null || candidates.length == 0 || targ
et <= 0) {
```

```
            return result;
        }

        List<Integer> list = new ArrayList<>();
        Arrays.sort(candidates);
        search(candidates, target, 0, list, result);
        return result;
    }

    private void search (int[] candidates,
                        int target,
                        int pos,
                        List<Integer> list,
                        List<List<Integer>> result) {
        if (target == 0) {
            result.add(new ArrayList<Integer>(list));
            return;
        }

        for (int i = pos; i < candidates.length; i++) {
            if (candidates[i] > target) {
                break;
            }

            if (i != pos && candidates[i] == candidates[i - 1])
{
                continue;
            }

            list.add(candidates[i]);
            search(candidates, target - candidates[i], i, list,
result);
            list.remove(list.size() - 1);
        }
    }
}
```

参考

1. Combination Sum | 九章算法

Combination Sum II

Combination Sum II (leetcode lintcode)

```
Description
Given a collection of candidate numbers (C) and a target number
(T),
find all unique combinations in C where the candidate numbers su
ms to T.
Each number in C may only be used once in the combination.

Notice
- All numbers (including target) will be positive integers.
- Elements in a combination (a1, a2, … , ak) must be in non-desc
ending order.
  (ie, a1 ≤ a2 ≤ … ≤ ak).
- The solution set must not contain duplicate combinations.

Example
Given candidate set [10,1,6,7,2,1,5] and target 8,
A solution set is:
[
  [1,7],
  [1,2,5],
  [2,6],
  [1,1,6]
]
```

解题思路

参考 Combination Sum 的解题思路，区别在于不能重复选取数组元素，所以在搜索时下一层递归的起始位置加一。

Java 实现

```
public class Solution {
    /**
```

```
     * @param num: Given the candidate numbers
     * @param target: Given the target number
     * @return: All the combinations that sum to target
     */
    public List<List<Integer>> combinationSum2(int[] num, int ta
rget) {
        List<List<Integer>> result = new ArrayList<List<Integer>
>();
        if (num == null || num.length == 0 || target <= 0) {
            //result.add(new ArrayList<Integer>());
            return result;
        }

        Arrays.sort(num);
        List<Integer> path = new ArrayList<Integer>();
        dfs(result, path, num, target, 0);
        return result;
    }

    private void dfs(List<List<Integer>> result, List<Integer> p
ath,
                     int[] num, int target, int pos) {
        if (target == 0) {
            result.add(new ArrayList<Integer>(path));
        }

        for (int i = pos; i < num.length; i++) {
            if (num[i] > target) {
                break;
            }
            if (i != pos && num[i] == num[i - 1]) {
                continue;
            }
            path.add(num[i]);
            dfs(result, path, num, target - num[i], i + 1);
            path.remove(path.size() - 1);
        }
    }
}
```

Combination Sum III

Combination Sum III (leetcode)

```
Description
Find all possible combinations of k numbers that add up to a num
ber n,
given that only numbers from 1 to 9 can be used and each combina
tion should be a unique set of numbers.

Example 1:
Input: k = 3, n = 7
Output:
[[1,2,4]]

Example 2:
Input: k = 3, n = 9
Output:
[[1,2,6], [1,3,5], [2,3,4]]
```

解题思路

这道题相当于 Combination Sum II 的 follow up，在其基础上还需要考虑子集中的和是否等于目标值。

Java 实现

```
public class Solution {
    public List<List<Integer>> combinationSum3(int k, int n) {
        List<List<Integer>> result = new ArrayList<List<Integer>
>();
        if (n < 1 || k < 1) {
            return result;
        }
        List<Integer> path = new ArrayList<Integer>();
        dfs(result, path, n, k, 1);
        return result;
    }

    private void dfs(List<List<Integer>> result, List<Integer> p
ath,
                     int n, int k, int pos) {
        if (path.size() == k && n == 0) {
            result.add(new ArrayList<Integer>(path));
            return;
        } else if (path.size() == k && n != 0) {
            return;
        }
        for (int i = pos; i <= 9; i++) {
            if (i > n) {
                break;
            }
            path.add(i);
            dfs(result, path, n - i, k, i + 1);
            path.remove(path.size() - 1);
        }
    }
}
```

Letter Combinations of a Phone Number

Letter Combinations of a Phone Number (leetcode lintcode)

```
Description
Given a digit string excluded 01,
return all possible letter combinations that the number could re
present.
A mapping of digit to letters (just like on the telephone button
s) is given below.
```


```
Notice
Although the above answer is in lexicographical order, your answ
er could be in any order you want.

Example
Given "23"
Return ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]
```

解题思路

和 Subsets、Permutations 的实现类似，通过遍历找到所有可能的组合。

需要注意的地方有：

1. 建立一个数字到字母的对照表。
2. 注意 String 、 StringBuilder 相关的函数及操作。
    - String ：
        - 第 `i` 个字符 `s.charAt(i)` ；

        - 字符串相等 `s.equals(t)` ；
        - 空字符串 `s.isEmpty()` ；
        - 字符串长度 `s.length()` ；
        - 子字符串（索引范围 `start ~ end-1` ） `s.substring(start, end)` ；
        - 转换为字符数组 `s.toCharArray()` 。
    - StringBuilder ：
        - 添加字符或字符串 `sb.append(s)` ；
        - 第 `i` 个字符 `sb.charAt(i)` ；
        - 删除第 `i` 个字符 `sb.deleteCharAt(i)` ；
        - 字符串大小 `sb.length()` ；
        - 转化为 `string` 类型 `sb.toString()` 。
3. `digits` 字符串中的数字需要递增遍历，每个数字对应的字母需要从头遍历。

Java 实现

```
public class Solution {
    /**
     * @param digits A digital string
     * @return all posible letter combinations
     */
    public ArrayList<String> letterCombinations(String digits) {
        ArrayList<String> result = new ArrayList<>();
        if (digits == null || digits.length() == 0) {
            return result;
        }

        String[] dict = {"abc", "def", "ghi", "jkl", "mno", "pqr
s", "tuv", "wxyz"};

        StringBuilder path = new StringBuilder();

        search(digits, path, 0, dict, result);
        return result;
    }

    private void search (String digits,
                        StringBuilder path,
```

```
                        int pos,
                        String[] dict,
                        ArrayList<String> result) {

        if (path.length() == digits.length()) {
            result.add(path.toString());
            return;
        }

        String letter = dict[digits.charAt(pos) - '0' - 2];
        for (int j = 0; j < letter.length(); j++) {
            path.append(letter.charAt(j));
            search(digits, path, pos + 1, dict, result);
            path.deleteCharAt(path.length() - 1);
        }

    }
}
```

参考

1. [LeetCode] Letter Combinations of a Phone Number (Java) | Life In Code

Word Ladder

Word Ladder (leetcode lintcode)

```
Description
Given two words (start and end), and a dictionary,
find the length of shortest transformation sequence from start t
o end, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the dictionary

Notice
- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.

Example
Given:
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
As one shortest transformation is "hit" -> "hot" -> "dot" -> "do
g" -> "cog",
return its length 5.
```

解题思路

本题目主要考察图的宽度优先搜索（ BFS ），我们来思考一下问题如何抽象。

1. 将每个单词看成无权图中的一个结点。
2. 如果单词 `s1` 改变一个字符可以变成 `s2` （ `s1` 和 s2 均在字典中），则 `s1` 和 `s2` 之间有连接（双向连接，权重相等皆为 `1` ）
3. 给定 `s1` 和 `s2` ，问题求解的是图中从 `s1` 到 `s2` 的最短路径长度。

在使用 BFS 求最短路径时，有几个关键步骤：

1. 如何找到与当前结点相邻的所有结点。

   有两个策略可以选择：

   - 遍历整个字典，将其中每个单词与当前单词比较，判断是否只差一个字符。时间复杂度为 `O(n * w)` ， `n` 为字典中单词数量， `w` 为单词长度。
   - 遍历当前单词的每个字符 `x` ，将其改为 `a~z` 中除 `x` 外的任意一个，判断字典中是否包含新单词。时间复杂度为 `O(26 * w)` ， `w` 为单词长度。

   一个常见的英语单词的长度一般小于 100 ，而字典中的单词数则往往成千上万，所以第二种策略相对较优。

2. 如何标记一个结点已经被访问过，以避免重复访问。

   可以将访问过的单词从字典中删除，或使用哈希表保存已访问过的单词。

3. 一旦 BFS 找到目标单词，如何回溯找到路径？

易错点

- `String` 变量可以直接使用 `char[]` 进行初始化。

Java 实现

```
public class Solution {
    /**
      * @param start, a string
      * @param end, a string
      * @param dict, a set of string
      * @return an integer
      */
    public int ladderLength(String start, String end, Set<String
> dict) {
        if (start == null || end == null ||
            start.length() == 0 || end.length() == 0) {
            return 0;
        }
        assert(start.length() == end.length());
        if (dict == null || dict.size() == 0) {
            return 0;
```

```
        }

        if (start.equals(end)) {
            return 1;
        }

        dict.add(start);
        dict.add(end);

        HashSet<String> hash = new HashSet<String>();
        Queue<String> queue = new LinkedList<String>();
        queue.offer(start);
        hash.add(start);

        int length = 1;
        while (!queue.isEmpty()) {
            length++;
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                String word = queue.poll();
                for (String nextWord: getNextWords(word, dict))
{
                    if (hash.contains(nextWord)) {
                        continue;
                    }
                    if (nextWord.equals(end)){
                        return length;
                    }
                    hash.add(nextWord);
                    queue.offer(nextWord);
                }
            }
        }
        return 0;
    }

    // get connections with given word.
    // for example, given word = 'hot', dict = {'hot', 'hit', 'h
og'}
    // it will return ['hit', 'hog']
```

```
    private ArrayList<String> getNextWords(String word, Set<Stri
ng> dict) {
        ArrayList<String> nextWords = new ArrayList<String>();
        for (char c = 'a'; c <= 'z'; c++) {
            for (int i = 0; i < word.length(); i++) {
                if (c == word.charAt(i)) {
                    continue;
                }
                String nextWord = replace(word, i, c);
                if (dict.contains(nextWord)) {
                    nextWords.add(nextWord);
                }
            }
        }
        return nextWords;
    }

    // replace character of a string at given index to a given c
haracter
    // return a new string
    private String replace(String s, int index, char c) {
        char[] chars = s.toCharArray();
        chars[index] = c;
        return new String(chars);
    }
}
```

参考

1. Word Ladder | 九章算法
2. [LeetCode] Word Ladder I, II | 喜刷刷