第零章 C++ 标准库及函数

容器

`map`容器

如：`map<char, int>mc;`

在`map`中，需要一个键和值，前者为键，后者为值

相关函数：

`map.find(value)`：在`map`容器中寻找`value`这个值，如果存在就返回`value`的位置，否则返回`end`(最后一个迭代器)

`map.count(value)`： 查找`map`中`value`是否存在，返回值`1`表示存在，`0`表示不存在

`vector`容器

如：`vector<int>vi;`

在`vector`容器中，支持初始化.

相关函数：

`vi.size()`：查看容器中的元素大小.

`vi.resize(const size_t New_size,const int &val)`：可以接受两个参数.(可用于拷贝)

成员：

`vi.capacity`：查看当前容器的大小.

算法函数

`accumulate()`求和算法，返回求的总和

如：`int sum = accumulate(nums.begin(), nums.end(), 0)`

先初始化为0，然后再计算从第一个下标开始到最后一个下标结束的所有元素的和

`copy()`可以复制一个数组到另外一个数组中

如：`copy(nums1.begin(), nums1.end(), nums2.begin()+k)`

意思是把`nums1`中的所有元素复制到`nums2`中`k`开始的到最后的位置

注意：copy函数不能增大容器的大小，如果要进行复制到原容器后面，需要先变大容器大小.

`count()`计数函数

如：`count(nums.begin(), nums.end(), value)`

在`nums`这个容器中寻找`value`这个值，返回查找到的*个数*

`exp()`函数

如：`exp(1)`，此函数接受一个参数，代表着`lnx`，其中`x`代表未知数

`exp(1)`=`2.71828`，以 e 为底的**对数函数**

`find()`从一个容器中寻找某一个元素

如：`find(vi.begin(), vi.end(),num)`

在`vi`这个容器中寻找`num`这个元素，如果存在就返回1，不存在就返回尾后迭代器

`earse()`删除容器中的元素(一般为容器的成员函数)

如：`nums.earse(nums.begin()+k, nums.end())`

删除从`k`开始的元素到最后的位置

`insert()`插入一段元素(一般为容器的成员函数)

如：`st.insert(nums.begin(), i)`

把一个元素`i`插入到容器的第一个位置(`i`的元素要和容器的类型相同)

`*max_element()`寻找最大值,并返回最大值

如：`*max_element(nums.begin(), nums.end());`

在这里，需要两个迭代器来计算哪一段的最大值

`*max_element(array, array+k);`

而在这里，传送的实参是一个数组，k 为下标，同样可以计算出一个数组中的最大值

`max()`比较大小函数

如：`max(a,b)`对`a`和`b`进行比较，返回大的值

C++11新特性：

`max({a,b,c})`对`{ }`里面的值进行比较，返回大的值

`min()`比较大小函数

和上面`max`函数相反，`min`函数是返回小的值，同时也满足*C++11的新特性*

`*min_element()`寻找最小值

用法和`*max_element()`相同，唯一不同的是返回值

`memset()`把指定的一块内层初始化为一个相同的值

原型：`extern void *memset(void *buffer, int c, int count)`

例：

```
int arr[2][4];
memset(arr,0,sizeof(array));
```

`reverse()`倒转函数

如：`reverse(st.begin(), st.end())`

可以把字符串、数组中的元素倒转

`sort()` 排序算法，无返回值

如：`sort(nums.begin(), nums.end())`

排序从第一个下标开始到最后以一个元素

`__gcd()` 求最大公约数

如：`__gcd(a,b)`；

在此函数中，a 和 b 的类型要一样，可以用的类型 `int` 和 `long long`

在头文件 `<algorithm>` 中

关于串的函数

在C + + 中，可以使用关于串的一些函数

`substr()` 得到字符串中某一段子串

如：`str.substr(0, 5)`

得到字符串 `str` 中从0开始到5的子串

`to_string()` 可以把其他类型转化成字符串类型

`itoi()&atoi()` 整形字符串(某进制)的相互转化

如：`itoi(num, str, 2)`

在上述中，`num` 为整形 `(int)`，`str` 为缓存区，用于保存转化后的字符串 `(char* Buffer)`，`2` 为要转化的进制数 `(int)`

如：`atoi(str.c_str())`

`str` 为 `string` 类型

```
c_str()函数返回一个指向正规C字符串的指针常量, 内容与本string串相同.
这是为了与c语言兼容，在c语言中没有string类型，故必须通过string类对象的成员函数c_str()把
string 对象转换成c中的字符串样式。
注意：一定要使用strcpy()函数 等来操作方法c_str()返回的指针
```

第一章 辅助算法

1.1. 快读和快输

- 快读算法的速度是普通输入算法的速度的20倍

```
/*
# input()
*/
int input(){
    char ch;
    int x=0,f=1;
    getchar(ch);
    while(x>'9'&&x<'0'){if(ch=='-')f=-1;ch=getchar();}
    while(x>='0'&&x<='9'){x=x*10+ch-'0';ch=gethcar();}
    return x*f;
}
```

- 快速输出

```
char f[100]; // the digits of the output number

void output(const int& x) {
    int tmp = x;
    //把负数情况列出来
    if (tmp < 0) {tmp = -tmp;putchar('-');}
    int s = 0;
    //把数字转化为字符，保存到字符串中.
    while (tmp > 0) {f[s++] = tmp%10 + '0';tmp /= 10;}
    //逐个输出字符
    while (s > 0) putchar(f[--s]);
}
```

1.2. 大数的基本运算

- 大数相加

  (1). 字符串形式

```
class Solution {
public:
    string addStrings(string num1, string num2) {
        string sum;
        int cur=0,i=num1.size()-1,j=num2.size()-1;
        /*
        *cur之所以不能等于0，是因为要实现进制，比如1和9相加
        */
        while(i>=0||j>=0||cur!=0){
            if(i>=0)cur+=num1[i--]-'0';
            if(j>=0)cur+=num2[j--]-'0';
            sum.insert(sum.begin(),cur%10+'0');
            cur/=10;
        }
        return sum;
    }
};
```

- 大数相乘
- 阶乘

  函数`pow()`的实现

  - 用一般方法实现`pow()`函数效率不是太高，使用*快速幂*的方法，可以提高很多
  - 快速幂的实现和二进制转为十进制差不多，如：1010的十进制为10
  - $10=1\times2^3+0\times2^2+0\times2^1+0\times2^0$

```
class Solution {
public:
    double myPow(double x, int n) {
        long long N=n;
        if(N<0){x=1/x;N=-N;}
        double xi=1;
        double xj=x;
        //核心代码如下：
        for(long long k=N;k;k/=2){
            if(k%2==1)xi*=xj;
            xj=xj*xj;
        }
        return xi;
    }
};
```

第二章 数据结构

2.1. 数组

2.1.1. 投票算法

- 投票算法模拟投票，例如：当有人选择候选者1号时，得到的票就会增加，而当有人选择后选择2号时，用1号得票表示就是1号的票减少，当1号得票为0时，1号失竞选资格，此时也可以代表2号的票超过1号的票.
- 投票算法还用于**括号深度**的计算

题号169

在一组数组中计算出出现次数大于数组大小1/2的数

```
    输入：[1 2 3 2 2 2]
    输出：2
在上述中，数组长度为6，即大于3的元素为2，所以输出2
```

代码：

```
    int majorityElement2(vector<int>& nums){
        int count_1=0;
        int num=nums[0];
        for(int i=1;i<nums.size();i++){
            if(count_1==0){
```

```
                count_1=1;
                num=nums[i];
            }
            if(num==nums[0])
                count_1++;
            else
                count_1--;
        }
        return num;
    }
```

2. 1. 2 双指针

- 通常双指针需要定义两个变量来模仿指针
- 常遇到的题型：
    1. 在同一个数组中，进行元素的变化

        如：在一个数组中寻找这个值，并在这个数组中删除这个值，最后返回数组的长度

例 2.2

在一组数组中，将所有为0的元素排到数组末尾，非0元素的顺序不变(尽量使用少的空间)

```
如：
    输入：[0 3 1 0 8 0]
    输出：[3 1 8 0 0 0]
```

```
    void moveZeroes(vector<int>& nums){
        int i=0,j=0;
        while(i<nums.size()&&j<nums.size()){
            if(nums[i]==0)
                i++;
            else{
                nums[j]=nums[i];
                j++;
                i++;
            }
        }
        for(int k=j;k<nums.size();k++){
            nums[k]=0;
        }
    }
```

2.1.3. 滑动窗口

- 在滑动窗口中，需要给窗口定一个界限
- 当滑动窗口中的元素大于窗口中的元素时需要删除某些元素

1248. 统计 [优美子数组]

给你一个整数数组 `nums` 和一个整数 `k`。

如果某个 连续 子数组中恰好有 `k` 个奇数数字，我们就认为这个子数组是「优美子数组」。

请返回这个数组中「优美子数组」的数目。

示例 1：

```
输入：nums = [1,1,2,1,1], k = 3
输出：2
解释：包含 3 个奇数的子数组是 [1,1,2,1] 和 [1,2,1,1] 。
```

示例 2：

```
输入：nums = [2,4,6], k = 1
输出：0
解释：数列中不包含任何奇数，所以不存在优美子数组。
```

示例 3：

```
输入：nums = [2,2,2,1,2,2,1,2,2,2], k = 2
输出：16
```

提示：

- 1 <= `nums.length` <= 50000
- 1 <= `nums[i]` <= 10^5
- 1 <= k <= `nums.length`

```
class Solution {
public:
    int numberOfSubarrays(vector<int>& nums, int k) {
        int ans=0,count=0;
        vector<int>vei;
        for(auto num:nums){
            ans++;
            if(num%2==1){
                vei.push_back(ans);
                ans=0;
            }
            if(vei.size()>=k)
                count+=vei[size(vei)-k];
        }
        return count;
    }
};
```

2.2. 栈和队列

- 栈
  - 指先进后出的一种线性结构.
  - 一般我们把插入和删除元素的一头叫做栈顶，第一个插入的元素叫做栈底.

    394. 字符串解

    给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: `k[encoded_string]`，表示其中方括号内部的 `encoded_string` 正好重复 k 次。注意 k 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 k ，例如不会出现像 3a 或 2[4] 的输入。

示例:

```
s = "3[a]2[bc]", 返回 "aaabcbc".
s = "3[a2[c]]", 返回 "accaccacc".
s = "2[abc]3[cd]ef", 返回 "abcabccdcdcdef".
```

```
class Solution {
public:
    string decodeString(string s) {
        stack<pair<int,string>>soo;
        int count=0;
        string res;
        for(char c:s){
            if(isdigit(c))
                count=count*10+(c-'0');
            else if(c=='['){
                soo.push({count,res});
                count=0;
                res="";
            }
            else if(c==']'){
                auto pai=soo.top();
                soo.pop();
                string tmp=res;
                for(int i=1;i<pai.first;i++)
                    res+=temp;
                res=p.second+res;
            }
            else
                res+=c;
        }
        return res;
    }
};
```

2.3. 树

2.3.1 二叉树

二叉树结构体模板如下:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
```

- **二叉树树的直径**

  543 树的直径

  给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过根结点。

  示例：
  给定二叉树

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

  返回 **3**, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

  注意：两结点之间的路径长度是以它们之间边的数目表示。

  a). 自上而下

  ```
  class Solution {
  public:
      int ans;
      int deepth(TreeNode *root){
          if(root==NULL)
              return 0;
          int L=deepth(root->left);
          int R=deepth(root->right);
          ans=max(ans,R+L+1);
          return max(L,R)+1;
      }
      int diameterOfBinaryTree(TreeNode* root) {
          int ans=1;
          deepth(root);
          return ans-1;
      }
  };
  ```

  b). 自下而上

2.3.3. 验证二叉搜索树（中序遍历）

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

- 节点的左子树只包含小于当前节点的数。
- 节点的右子树只包含大于当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1:

```
输入:
    2
   / \
  1   3
输出: true
```

示例 2:

```
输入:
    5
   / \
  1   4
     / \
    3   6
输出: false
解释: 输入为: [5,1,4,null,null,3,6]。
     根节点的值为 5 ，但是其右子节点值为 4 。
```

```
class Solution {
public:
    long pre=LONG_MIN;
    bool isValidBST(TreeNode* root) {
        if(root==null)
            return true;
        //访问左子树
        if(!(isValidBST(root->left)))
            return false;
        //与上一个节点进行比较，如果大于上衣个节点就满足 root>root->left->val
        if(root->val<=pre)
            return false;
        pre=root->val;
        return isValidBST(root->right);
    }
};
```

(1) 平衡二叉树

左右子树高度差的绝对值小于1的二叉树叫做平衡二叉树

- 判断是否为平衡二叉树

  110. 平衡二叉树

给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：

一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过1。

示例 1:

```
给定二叉树 [3,9,20,null,null,15,7]
    3
   / \
  9  20
    /  \
   15   7
返回 true 。
```

示例 2:

```
给定二叉树 [1,2,2,3,3,null,null,4,4]
       1
      / \
     2   2
    / \
   3   3
  / \
 4   4
 返回 false 。
```

2.3.2 哈希树

- 前缀树(字典树)

  208. 实现 `Trie` (前缀树)

  实现一个 `Trie`(前缀树)，包含`insert`,`search`, 和`startsWith` 这三个操作。

  示例:

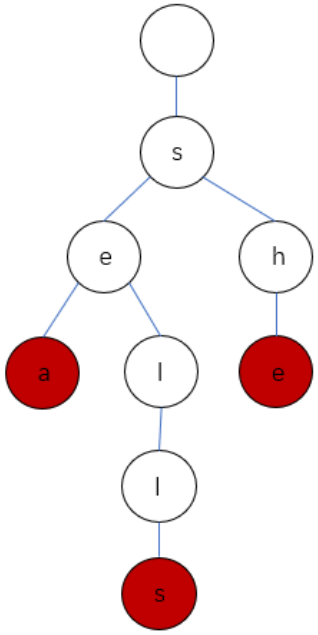
```
Trie trie = new Trie();

trie.insert("apple");
trie.search("apple");   // 返回 true
trie.search("app");     // 返回 false
trie.startsWith("app"); // 返回 true
trie.insert("app");
trie.search("app");     // 返回 true
```

  说明:

  你可以假设所有的输入都是由小写字母 a-z 构成的。

  保证所有输入均为非空字符串。

关于保存字符如 *图a* 示



图a

```
class Trie {
private:
    bool isempty;
    Trie *next[26];
public:
    /** Initialize your data structure here. */
    Trie():isempty(false),next{nullptr} {}

    /** Inserts a word into the trie. */
    void insert(string word) {
        Trie *root=this;
        for(const char w:word){
            if(root->next[w-'a']==nullptr)
                root->next[w-'a']=new Trie();
            root=root->next[w-'a'];
        }
        root->isempty=true;
    }

    /** Returns if the word is in the trie. */
    bool search(string word) {
        Trie *root=this;
        for(const char w:word){
            root=root->next[w-'a'];
            if(root==nullptr)
                return false;
        }
        return root->isempty;
    }

    /** Returns if there is any word in the trie that starts with the given
prefix. */
    bool startsWith(string prefix) {
```

```
        Trie *root=this;
        for(const char w:prefix){
            root=root->next[w-'a'];
            if(root==nullptr)
                return false;
        }
        return true;
    }
};

/**
 * Your Trie object will be instantiated and called as such:
 * Trie* obj = new Trie();
 * obj->insert(word);
 * bool param_2 = obj->search(word);
 * bool param_3 = obj->startsWith(prefix);
 */
```

2.4. 图

2.5. 搜索

2.5.1 深度优先搜索(`dfs`)

695. 岛屿的最大面积

给定一个包含了一些 0 和 1的非空二维数组 grid , 一个 岛屿 是由四个方向 (水平或垂直) 的 1 (代表土地) 构成的组合。你可以假设二维矩阵的四个边缘都被水包围着。

找到给定的二维数组中最大的岛屿面积。(如果没有岛屿，则返回面积为0。)

示例 1:

```
[[0,0,1,0,0,0,0,1,0,0,0,0,0],
 [0,0,0,0,0,0,0,1,1,1,0,0,0],
 [0,1,1,0,1,0,0,0,0,0,0,0,0],
 [0,1,0,0,1,1,0,0,1,0,1,0,0],
 [0,1,0,0,1,1,0,0,1,1,1,0,0],
 [0,0,0,0,0,0,0,0,0,0,1,0,0],
 [0,0,0,0,0,0,0,1,1,1,0,0,0],
 [0,0,0,0,0,0,0,1,1,0,0,0,0]]

 对于上面这个给定矩阵应返回 6。注意答案不应该是11，因为岛屿只能包含水平或垂直的四个方向的‘1’。
```

示例 2:

```
[[0,0,0,0,0,0,0,0]]
```

对于上面这个给定的矩阵, 返回 `0`。

注意: 给定的矩阵`grid` 的长度和宽度都不超过 50。

```
class Solution {
```

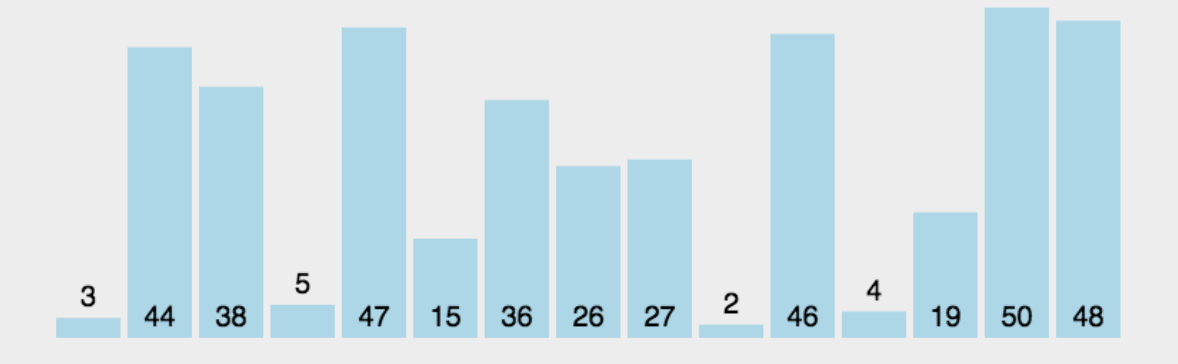
```
public:
    int dfs(vector<vector<int>>&grid,int i,int j){
        if(i<0||j<0||i==grid.size()||
           j==grid[0].size())
            return 0;
        if(grid[i][j]==1){
            grid[i][j]=0;
            //这段代码与树的深度搜索有相似之处
            return dfs(grid,i-1,j)+dfs(grid,i,j-1)
            +dfs(grid,i+1,j)+dfs(grid,i,j+1)+1;
        }
        return 0;
    }
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int maxarea=0;
        for(int i=0;i<grid.size();i++){
            for(int j=0;j<grid[0].size();j++){
                if(grid[i][j]==1)
                    maxarea=max(maxarea,dfs(grid,i,j));
            }
        }
        return maxarea;
    }
};
```

5.5.2 二分查找

2.6. 排序

2.6.1. 交换排序

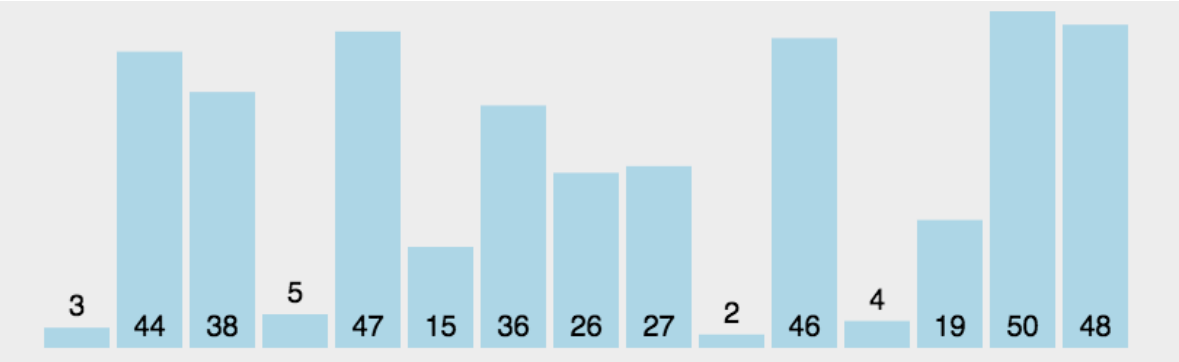
- 冒泡排序




```
    vector<int> sortArray(vector<int>& nums) {
        int len=nums.size()-1;
        for(int i=0;i<len;i++){
            for(int j=0;j<len-i;j++)
                if(nums[j]>nums[j+1])
                    swap(nums[j],nums[j+1]);
        }
        return nums;
    }
```

3.6.2. 选择排序

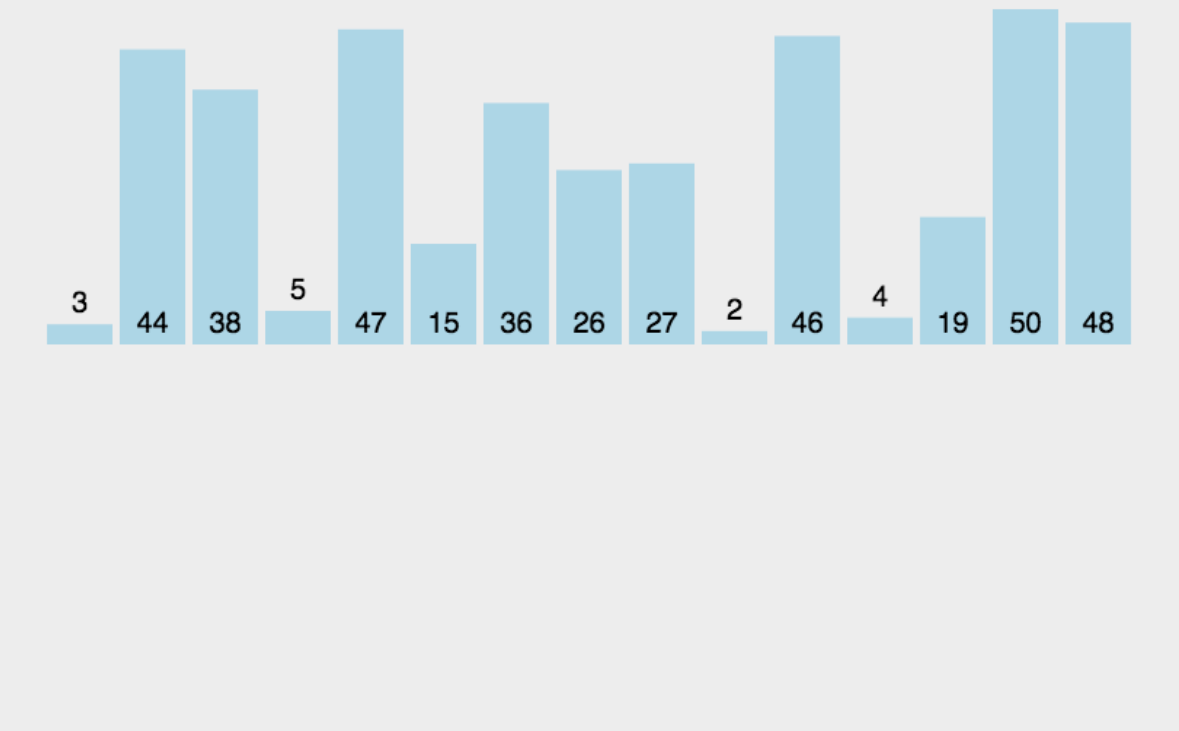
- 简单选择排序




```
class Solution {
public:
    vector<int> sortArray(vector<int>& nums) {
        int len=nums.size()-1;
        //temp用来记住下标
        int temp=0;
        for(int i=0;i<len;i++){
            temp=i;
            for(int j=i+1;j<=len;j++){
                if(nums[j]<nums[temp])
                    temp=j;
            }
            swap(nums[i],nums[temp]);
        }
        return nums;
    }
};
```

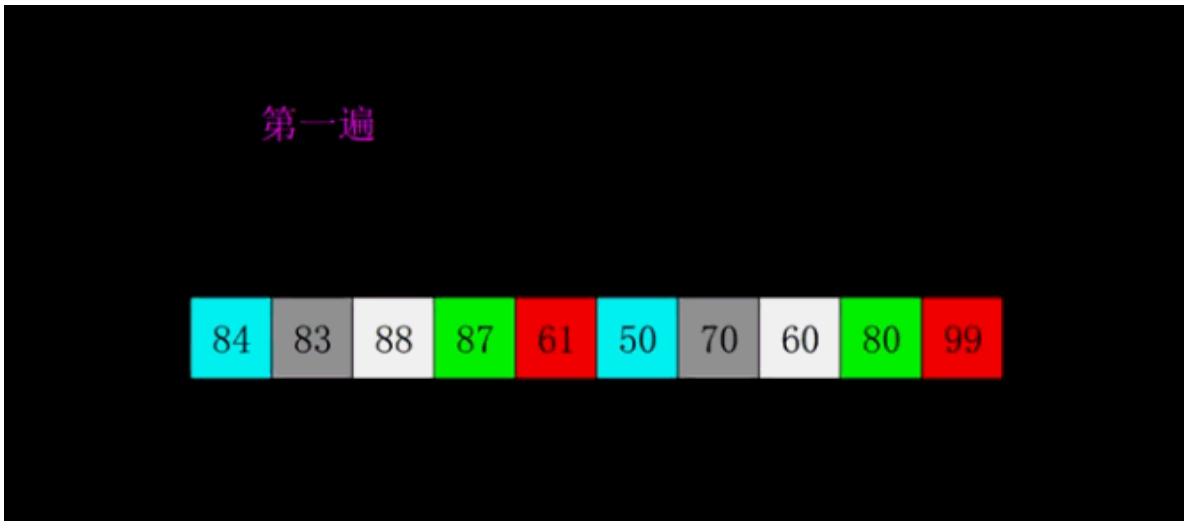
2.6.3. 插入排序

1. **简单插入排序**



```
class Solution {
public:
    vector<int> sortArray(vector<int>& nums) {
        int In=0,Out=0;
        for(int i=1;i<nums.size();i++){
            In=i-1;
            Out=nums[i];
            while(In>=0&&nums[In]>Out){
                nums[In+1]=nums[In];
                --In;
            }
            nums[In+1]=Out;
        }
        return nums;
    }
};
```

2. **希尔排序**

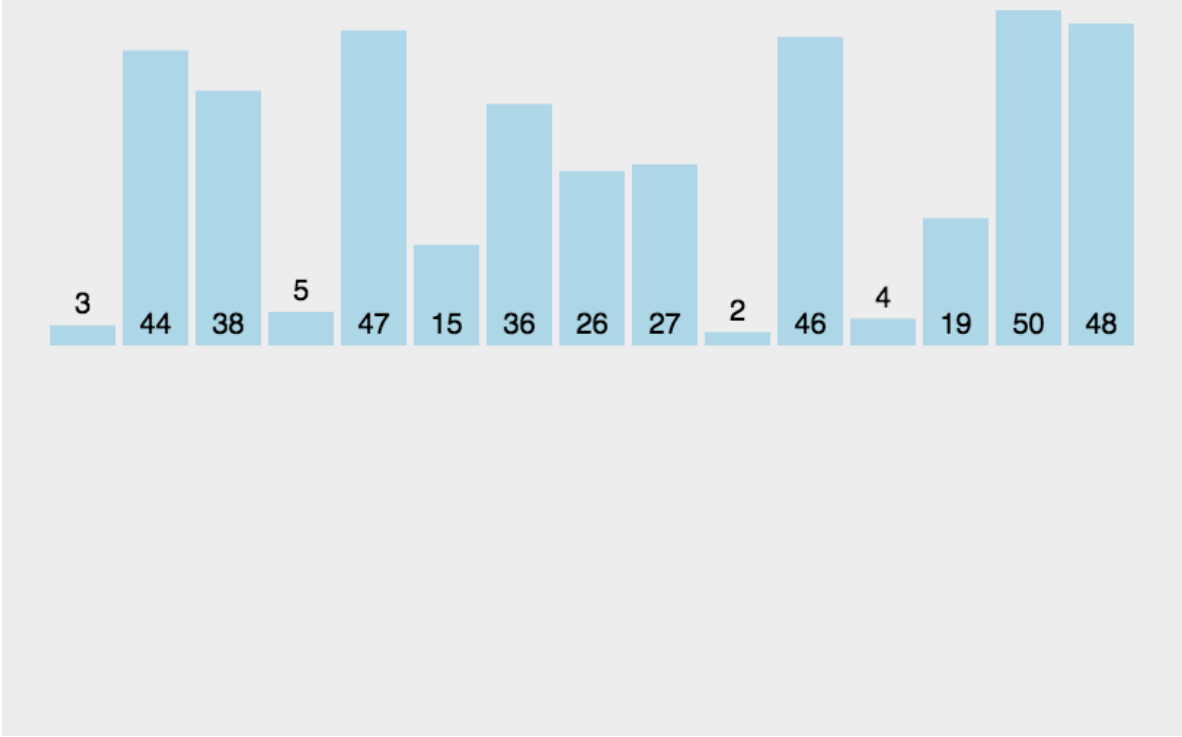


原始的希尔排序如下：

```
class Solution {
public:
    vector<int> sortArray(vector<int>& nums) {
        for(int len=nums.size();len>0;len/=2){
            for(int i=len;i<nums.size();i++){
                int In=i-len;
                int Out=nums[i];
                while(In>=0&&nums[In]>Out){
                    nums[In+len]=nums[In];
                    In-=len;
                }
                nums[In+len]=Out;
            }
        }
        return nums;
    }
};
```

2.6.4. 归并排序

- 归并排序采用**分治**和**递归**的思想
- `LeetCode`上归并相关题型（21. 合并两个有序链表，912. 排序数组）



2.6.5. 基数排序

第三章 数学

6.1. 动态规划

- 贪心算法是动态规划中特殊的一种
- 贪心算法必须无后效性，就是以前的过程对后来的状态不产生影响

a 常见`dp`

6.1.1 插入数字

给定一个数组和一个数 n，在数组中只存在0和1，现在要插入n 个数字1到数组中，插入规则如下：

不能把数字1插入到原数组中数字1旁边

求是否还能插入数字1，如果能插入返回true，否则返回false

```
输入：[1,0,0,0,1] 1
输出：true
```

```
bool canPlaceFlowers(vector<int>& flowerbed, int n){
    int count=0;
```

```
    flowerbed.insert(flowerbed.begin(),0);
    flowerbed.insert(flowerbed,end(),0);
    //也可以用 flowerbed.push_back(0);
    for(int i=1;i<flowerbed.size()-1;i++){
        if(flowerbed[i]==0&&flowerbed[i-1]==0&&flowerbed[i+1]==0){
            flowerbed[i]=1;//插入1
            count++;
        }
    }
    return n<=count;
    //return n<=count?1:0;
}
```

300 最长上升子序

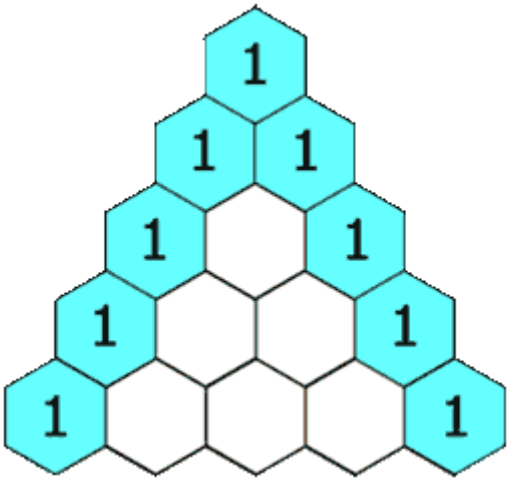
给定一个无序的整数数组，找到其中最长上升子序列的长度。

```
输入: [1,3,5,6], 5
输出: 2
```

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        int len=nums.size();
        if(len==0)
            return 0;
        vector<int>dp(len,0);
        for(int i=0;i<len;i++){
            dp[i]=1;
            for(int j=0;j<i;j++){
                if(nums[i]>nums[j])
                    dp[i]=max(dp[i],dp[j]+1);
                    /*
                    *寻找dp[i]以前的所有值比当前值小时，+1后做对比，然后更新dp[i]
                    *如果没有出现，dp[i]就为1
                    */
            }
        }
        return *max_element(dp.begin(),dp.end());
    }
}
```

118. 杨辉三角

给定一个非负整数 *numRows*，生成杨辉三角的前 *numRows* 行。



在杨辉三角中，每个数是它左上方和右上方的数的和。

示例:

```
输入: 5
输出:
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

```
class Solution {
public:
    vector<vector<int>> generate(int numRows) {
        vector<vector<int>>vi(numRows);
        if(numRows==0)
            return vi;
        //主要代码，在这里采取了动态规划的思想
        for(int i=0;i<numRows;i++){
            for(int j=0;j<=i;j++){
                if(j==0||i==j)
                    vi[i].push_back(1);
                else
                    vi[i].push_back(vi[i-1][j]+vi[i-1][j-1]);
            }
        }
        return vi;
    }
};
```

746. 使用最小花费爬楼梯

- 记录型问题
    - 相识题目{213. 打家劫舍 |面试题 17.16. 按摩师}

数组的每个索引做为一个阶梯，第 i个阶梯对应着一个非负数的体力花费值 costi。

每当你爬上一个阶梯你都要花费对应的体力花费值，然后你可以选择继续爬一个阶梯或者爬两个阶梯。

您需要找到达到楼层顶部的最低花费。在开始时，你可以选择从索引为 0 或 1 的元素作为初始阶梯。

```
示例 1:

输入: cost = [10, 15, 20]
输出: 15
解释: 最低花费是从cost[1]开始，然后走两步即可到阶梯顶，一共花费15。
 示例 2:

输入: cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]
输出: 6
解释: 最低花费方式是从cost[0]开始，逐个经过那些1，跳过cost[3]，一共花费6。
```

注意：

cost 的长度将会在 [2, 1000]。
每一个 cost[i] 将会是一个Integer类型，范围为 [0, 999]。

```
class Solution {
public:
    int minCostClimbingStairs(vector<int>& cost) {
        int len=cost.size();
        for(int i=2;i<len;i++){
            cost[i]+=min(cost[i-1],cost[i-2]);
        }
        return min(cost[len-1],cost[len-2]);
    }
};
```

983. 最低票价

在一个火车旅行很受欢迎的国度，你提前一年计划了一些火车旅行。在接下来的一年里，你要旅行的日子将以一个名为 days 的数组给出。每一项是一个从 1 到 365 的整数。

火车票有三种不同的销售方式：

一张为期一天的通行证售价为 costs[0] 美元；
一张为期七天的通行证售价为 costs[1] 美元；
一张为期三十天的通行证售价为 costs[2] 美元。
通行证允许数天无限制的旅行。 例如，如果我们在第 2 天获得一张为期 7 天的通行证，那么我们可以连着旅行 7 天：第 2 天、第 3 天、第 4 天、第 5 天、第 6 天、第 7 天和第 8 天。

返回你想要完成在给定的列表 days 中列出的每一天的旅行所需要的最低消费。

示例 1：

```
输入：days = [1,4,6,7,8,20], costs = [2,7,15]
输出：11
解释：
例如，这里有一种购买通行证的方法，可以让你完成你的旅行计划：
在第 1 天，你花了 costs[0] = $2 买了一张为期 1 天的通行证，它将在第 1 天生效。
在第 3 天，你花了 costs[1] = $7 买了一张为期 7 天的通行证，它将在第 3, 4, ..., 9 天生效。
在第 20 天，你花了 costs[0] = $2 买了一张为期 1 天的通行证，它将在第 20 天生效。
你总共花了 $11，并完成了你计划的每一天旅行。
```

示例 2：

```
输入：days = [1,2,3,4,5,6,7,8,9,10,30,31], costs = [2,7,15]
输出：17
解释：
例如，这里有一种购买通行证的方法，可以让你完成你的旅行计划：
在第 1 天，你花了 costs[2] = $15 买了一张为期 30 天的通行证，它将在第 1, 2, ..., 30 天生效。
在第 31 天，你花了 costs[0] = $2 买了一张为期 1 天的通行证，它将在第 31 天生效。
你总共花了 $17，并完成了你计划的每一天旅行。
```

提示：

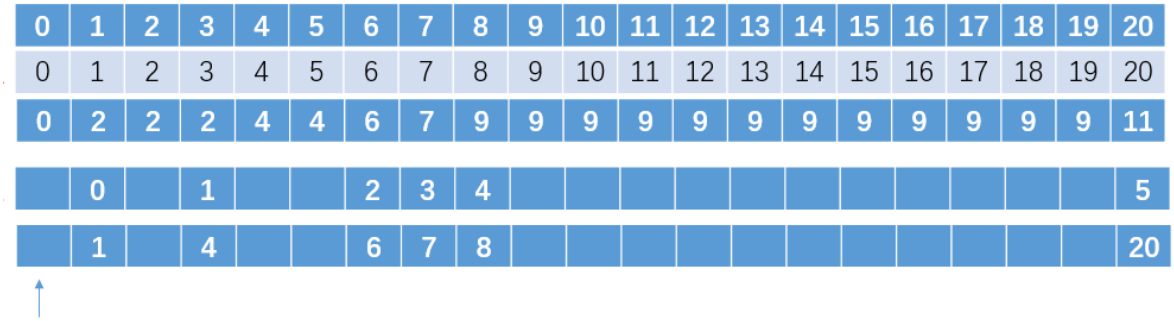
1 <= `days.length` <= 365

1 <= `days[i]` <= 365

`days` 按顺序严格递增

`costs.length` == 3

1 <= `costs[i]` <= 1000




```cpp
class Solution {
public:
    int mincostTickets(vector<int>& days, vector<int>& costs) {
        //用一个数组来存储最小的花费
        vector<int>dp(days.back()+1);
        for(int i=0;i<dp.size();i++)
            dp[i]=i;
        int days_idx=0;
        for(int i=1;i<dp.size();i++){
            if(i!=days[days_idx])
                dp[i]=dp[i-1];
            else{
                dp[i]=min({dp[max(0,i-1)]+costs[0],
                           dp[max(0,i-7)]+costs[1],
                           dp[max(0,i-30)]+costs[2]});
                days_idx++;
            }
```

```
        }
        return dp.back();
    }
};
```

5. 最长回文子串

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例 1：

```
输入: "babad"
输出: "bab"
注意: "aba" 也是一个有效答案。
```

示例 2：

```
输入: "cbbd"
输出: "bb"
```

```
```

b 状态压缩

入门题型

1371. 每个元音包含偶数次的最长子字符串

给你一个字符串 s ，请你返回满足以下条件的最长子字符串的长度：每个元音字母，即 'a'，'e'，'i'，'o'，'u' ，在子字符串中都恰好出现了偶数次。

示例 1：

```
输入：s = "eleetminicoworoep"
输出：13
解释：最长子字符串是 "leetminicowor" ，它包含 e，i，o 各 2 个，以及 0 个 a，u 。
```

示例 2：

```
输入：s = "leetcodeisgreat"
输出：5
解释：最长子字符串是 "leetc" ，其中包含 2 个 e 。
```

```
class Solution {
public:
```

```
    string sub(string &s,int x,int y){
        string ss;
        for(int i=x;i<y;i++){
            ss.push_back(s[i]);
        }
        return ss;
    }
    string longestPalindrome(string s) {
        string origin=s;
        reverse(s.begin(),s.end());
        int len=s.size(),MaxLen=0,MaxEnd=0;
        vector<vector<int>>dp(len,vector<int>(len));
        for(int i=0;i<len;i++){
            for(int j=0;j<len;j++){
                if(s[i]==origin[j]){
                    if(i==0||j==0)
                        dp[i][j]=1;
                    else
                        dp[i][j]=dp[i-1][j-1]+1;
                }
                if(dp[i][j]>MaxLen){
                    int before_index=len-1-j;
                    if(before_index+dp[i][j]-1==i){
                        MaxLen=dp[i][j];
                        MaxEnd=i;
                    }
                }
            }
        }
        return sub(s,MaxEnd-MaxLen+1,MaxLen+1);
    }
};
```

示例 3：

```
输入：s = "bcbcbc"
输出：6
解释：这个示例中，字符串 "bcbcbc" 本身就是最长的，因为所有的元音 a，e，i，o，u 都出现了 0 次。
```

提示：

- `1 <= s.length <= 5 x 10^5`
- `s` 只包含小写英文字母。

```
class Solution {
public:
    int findTheLongestSubstring(string s) {
        vector<int>pos(1<<5,-1);
        pos[0]=0;
        int ans=0,status=0;
        for(int i=0;i<s.size();i++){
            if(s[i]=='a')
                status^=1<<0;
            else if(s[i]=='e')
```

```
                status^=1<<1;
            else if(s[i]=='i')
                status^=1<<2;
            else if(s[i]=='o')
                status^=1<<3;
            else if(s[i]=='u')
                status^=1<<4;
            if(~pos[status])
                ans=max(ans,i+1-pos[status]);
            else
                pos[status]=i+1;
        }
        return ans;
    }
};
```

6.2. 数论

6.2.1. 欧几里得算法

最大公约数缩写 `gcd`

欧几里得算法又叫做**辗转相除法**

欧几里得算法是求两个数的最大公约数的算法

在算法中具体原来连接如右[点击链接]

例6.1.1

```
    给我们两个数1998和615
    ~1998/615=3(余 152)
    ~615/152=4(余 7)
    最后计算到没有余数时，那个数就是我们要求的最大公约数
```

算法实现：

```
方法1：
    int gcd(int a,int b){
        //如果b==0时，刚好是最后一次，就可以得到结果a
        return b>0?gcd(b,a%b):a;
    }
方法2：
    int gcd(int a,int b){
        while(b){
            b=a%b;
            a=b;
        }
        return a;
    }
方法3：
    int gcd(int a,int b){
        if(b)
            while((a%b)&&(b%a));
        return a+b;
```

```
    }
方法4：
    int gcd(int a,int b){
        while(b^=a^=b^=a%=b);
        return a;
    }
```

1071 求两个字符串的最大公约数

如：对于字符串 `S` 和 `T`，只有在 `S = T + ... + T`（`T` 与自身连接 1 次或多次）时，我们才认定“`T` 能除尽 "`S`”。返回最长字符串 `X`，要求满足 `X` 能除尽 `str1` 且 `X` 能除尽 `str2`。

```
输入：
    str1 = "ABCABC", str2 = "ABC"
输出：
    "ABC"
```

```
class Solution {
public:
    inline int gcd(int &a,int &b) {
        while(b){
            b=a%b;
            b=a;
        }
        return a;
    }
    string gcdOfStrings(string str1, string str2) {
        if(str1+str2!=str2+str1)
            return "";
        return str1.substr(0,gcd(str1.size(),str2.size()));
    }
};
```

3.3.5. 贝祖定理

- 裴蜀定理（或贝祖定理）
- 若a,b是整数,且 `gcd(a,b)=d`，那么对于任意的整数 `x`,`y`,`ax+by` 都一定是 `d` 的倍数，特别地，一定存在整数x,y，使 `ax+by=d` 成立。

365. 水壶问题

有两个容量分别为 *x*升 和 *y*升 的水壶以及无限多的水。请判断能否通过使用这两个水壶，从而可以得到恰好 *z*升 的水？

如果可以，最后请用以上水壶中的一或两个来盛放取得的 *z*升水。

你允许：

- 装满任意一个水壶
- 清空任意一个水壶
- 从一个水壶向另外一个水壶倒水，直到装满或者倒空

示例 1:

```
输入: x = 3, y = 5, z = 4
输出: True
```

示例 2:

```
输入: x = 2, y = 6, z = 5
输出: False
```

```
class Solution {
public:
    int gcd(int x,int y){return y>0?gcd(y,x%y):x;}
    bool canMeasureWater(int x, int y, int z) {
        if(x==0||y==0)
            return x+y==0||z==0;
        if(x+y<z)
            return false;
        return z%gcd(x,y)==0;
    }
};
```

6.2.2. 回文

409. 最长回文串

在构造过程中，请注意区分大小写。比如 `"Aa"` 不能当做一个回文字符串。

注意:
假设字符串的长度不会超过 1010。

示例 1:

```
输入:
"abccccdd"

输出:
7

解释:
我们可以构造的最长的回文串是"dccaccd", 它的长度是 7。
```

```
class Solution {
public:
    int longestPalindrome(string s) {
        unordered_map<char,int>umc;
        int v=0,ans=0;
        //当作计数器，把每一个字符出现的次数记录下来
        for(char c:s)
            ++umc[c];
        for(auto d:umc){
            v=d.second;
            //每次能计数的只能是偶数
            ans+=v/2*2;
            //有且只能存在一个奇数
            if(ans%2==0&&v%2==1)
                ans++;
```

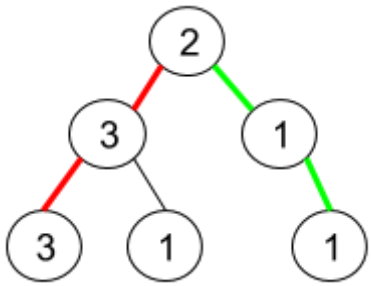
```
        }
        return ans;
    }
};
```

1457. 二叉树中的伪回文路径

给你一棵二叉树，每个节点的值为 1 到 9 。我们称二叉树中的一条路径是 「伪回文」的，当它满足：路径经过的所有节点值的排列中，存在一个回文序列。

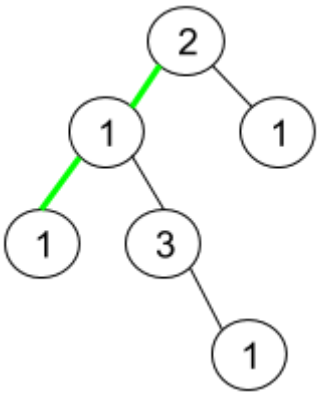
请你返回从根到叶子节点的所有路径中**伪回文**路径的数目。

示例 1：




```
输入：root = [2,3,1,3,1,null,1]
输出：2
解释：上图为给定的二叉树。总共有 3 条从根到叶子的路径：红色路径 [2,3,3] ，绿色路径 [2,1,1] 和路径 [2,3,1] 。
      在这些路径中，只有红色和绿色的路径是伪回文路径，因为红色路径 [2,3,3] 存在回文排列 [3,2,3] ，绿色路径 [2,1,1] 存在回文排列 [1,2,1] 。
```

示例 2：




```
输入：root = [2,1,1,1,3,null,null,null,null,null,1]
输出：1
解释：上图为给定二叉树。总共有 3 条从根到叶子的路径：绿色路径 [2,1,1] ，路径 [2,1,3,1] 和路径 [2,1] 。
      这些路径中只有绿色路径是伪回文路径，因为 [2,1,1] 存在回文排列 [1,2,1] 。
```

示例 3：

```
输入：root = [9]
输出：1
```

提示：

给定二叉树的节点数目在 1 到 10^5 之间。
节点值在 1 到 9 之间。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    int res=0;
    void dfs(TreeNode* root,int count){
        if(root->NULL)return;
        count^=(1<<res);
        if(root->left==null&&root->right==null){
            if()
        }
        dfs(root->left);
        dfs(root->right);
    }
    int pseudoPalindromicPaths (TreeNode* root) {
        dfs(root,0);
        return res;
    }
};
```

6.2.4. 同余定理

【同余定理】

给定一个正整数`m(m>0)`，如果两个整数`a`和`b`满足`a-b`能够被`m`整除，即`(a-b)/m`得到一个整数，那么就称整数`a`与`b`对模`m`同余，记作*`a≡b(mod m)`*。对模m同余是整数的一个等价关系。

例：`m=5，a=22，b=2，`那么有`(a-b)/5=4`，22 ≡ 2 (mod 5).

1. 当`(a+b)/m`时，余数为`b`.

2. 当`a/m`时，必为整除.

性质：

（1）若a≡0(mod m)，则m|a；(整除)

（2）a≡b(mod m)等价于`a`与`b`分别用`m`去除，余数相同.

974. 和可被 K 整除的子数组

给定一个整数数组 A，返回其中元素之和可被 K 整除的（连续、非空）子数组的数目。

示例：

```
输入：A = [4,5,0,-2,-3,1], K = 5
输出：7
解释：
有 7 个子数组满足其元素之和可被 K = 5 整除：
[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]
```

提示：

1 <= `A.length` <= 30000
-10000 <= `A[i]` <= 10000
2 <= `K` <= 10000

```
class Solution {
public:
    int subarraysDivByK(vector<int>& A, int K) {
        int sum=0,ans=0;
        unordered_map<int,int>umap={{0,1}};
        for(int a:A){
            sum+=a;
            //同余
            int mod=(sum%K+K)%K;
            if(umap.count(mod))
                ans+=umap[mod];
            ++umap[mod];
        }
        return ans;
    }
};
```

6.4. 其他

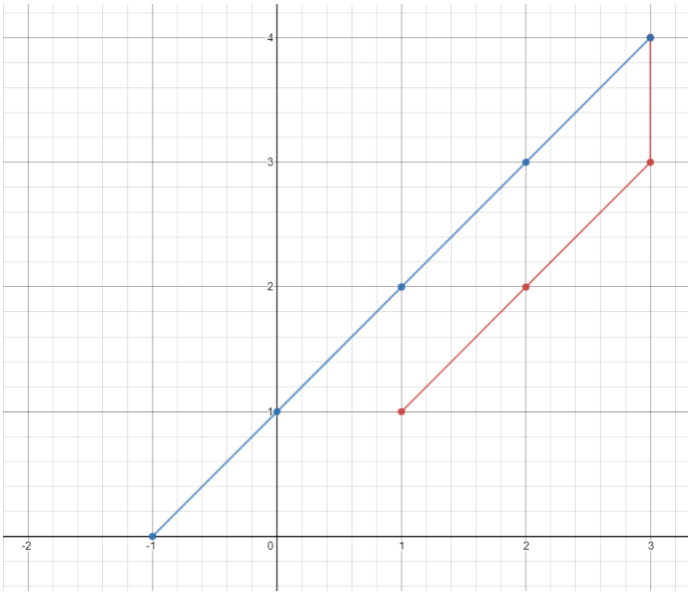
6.3.1. 切比雪夫距离

- 切比雪夫距离一般是在一个二维坐标中，两个坐标差的绝对值的最大值
    - ans=max(abs(`x2-x1`)，abs(`y2-y1`));

题号1266 最小时间

平面上有n个点，用坐标(x,y)表示，求访问这些点的最小时间

```
* 按顺序访问
* 每秒按水平、竖直或跨对角线移动
```



```
int minTimeToVisitAllPoints(vector<vector<int>>& points) {
    int x1=points[0][0],y1=points[0][1];
    int x2,y2;
    int second=0;
    for(int i=1;i<points.size();i++){
        x2=points[i][0],y2=points[i][1];
        second+=max(abs(x2-x1),abs(y2-y1));
        x1=x2;
        y1=y2;
    }
    return second;
}
```

6.3. 2. 曼哈顿距离

6.3.3. 汉明距离

汉明距离的特性：

对于固定的长度 n，汉明距离是该长度字符向量空间上的度量，很显然它满足非负、唯一及对称性，并且可以很容易地通过完全归纳法证明它满足三角不等式。

如果把a和b两个单词看作是向量空间中的元素，则它们之间的汉明距离等于它们汉明重量的差a-b。如果是二进制字符串a和b，汉明距离等于它们汉明重量的和a+b或者a和b汉明重量的异或a XOR b。汉明距离也等于一个n维的超立方体上两个顶点间的曼哈顿距离，n指的是单词的长度。

给予两个任何的字码，10001001和10110001，即可决定有多少个相对位是不一样的。在此例中，有三个位不同。要决定有多少个位不同，只需将exclusive OR运算加诸于两个字码就可以，并在结果中计算有多个为1的位。例如：

10001001

`Xor` 10110001

00111000

两个字码中不同位值的数目称为汉明距离(`Hamming distance`)。它的重要性在于如果有两个字码的汉明距离为d的话，就需要d的单一位错误已将其中一个字码转换为另一个。

461. 汉明距离

在`d(x, y)`中，寻找`x`到`y`的二进制位的距离：

示例:

```
输入: x = 1, y = 4

输出: 2

解释:
1   (0 0 0 1)
4   (0 1 0 0)
       ↑   ↑

上面的箭头指出了对应二进制位不同的位置。
```

普通解法:

```
class Solution {
public:
    int hammingDistance(int x, int y) {
        int z=x^y;
        int cot=0;
        while(z){
            //也可以用cot+=z&1来实现，这样就不需要if判断了
            if(z%2==1)
                cot++;
            z>>=1;
        }
        return cot;
    }
};
```

布赖恩·克尼根算法:

```
class Solution {
public:
    int hammingDistance(int x, int y) {
        int z=x^y;
        int cot=0;
        while(z){
            //最主要的是下面这一个语句
            z=z&(z-1);
            cot++;
        }
        return cont;
    }
}
```

6.3.4. 等差数列

- 等差数列通项公式、求和公式:

  $a_n=a_1+(n-1)\times d$

  $S_n=na_1+[n(n-1)/2]d \ \ (n\in N^*)$

  $S_n=n(a_1+a_n)\times d/2$

- 等差中项的关系：

  $a_n=a_m+(n-m)d$

268 缺失的数字

在一组数组中，找出缺失的数字：

```
输入：[3,0,1]
输出：2

缺失的数字为2
```

```
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        return nums.size()*(nums.size()+1)/2
            -accumulate(nums.begin(),nums.end(),0);
    }
}
```

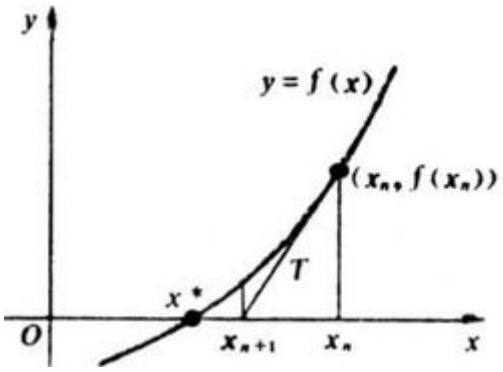
3.3.6 弦图

- 常用算法：最大优势算法

3.3.7. 牛顿求解法

- 参考公式：

  $X_{n+1}=X_n - F(X_n)/F'(X_n)$



69. x 的平方根

实现 `int sqrt(int x)` 函数。

计算并返回 x 的平方根，其中 x 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

示例 1:

```
输入：4
输出：2
```

示例 2:

输入：8
输出：2
说明：8 的平方根是 2.82842...,
由于返回类型是整数，小数部分将被舍去。

应用牛顿法解平方根问题，可令 $f(x)=x^2-a$

由 $f(x)\approx f(x_0)+(x-x_0)f'(x_0)$ 记忆方法：用斜率定义 $f'(x_0)=\frac{f(x)-f(x_0)}{x-x_0}$

令 $f(x)=0$，得 $f(x_0)+(x-x_0)f'(x_0)=0$

$$x=x_0-\frac{f(x_0)}{f'(x_0)}=x_0-\frac{x_0^2-a}{2x_0}=\frac{2x_0^2-x_0^2+a}{2x_0}$$

$$=\frac{1}{2}\frac{x_0^2+a}{x_0}=\frac{1}{2}(x_0+\frac{a}{x_0})$$

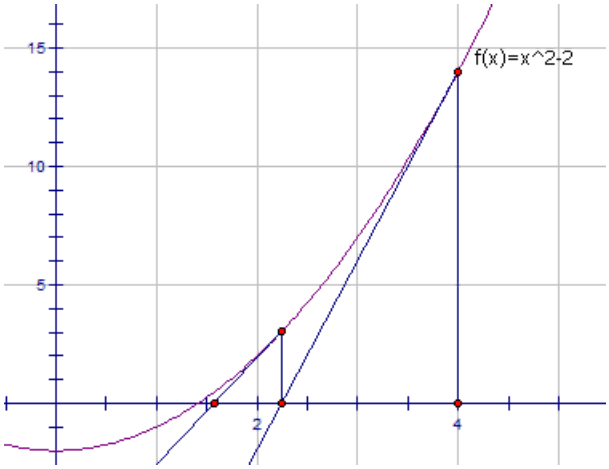
故迭代公式为 $x_0=\frac{1}{2}(x_0+\frac{a}{x_0})$

代码编写：

$cur=1$ 初值，可任意选取

如果初值选择负数，有可能得到负数平方根

> (4 + 2/ 4) / 2 = 2.25
>
> (2.25 + 2/ 2.25) / 2 = 1.56944..
>
> (1.56944..+ 2/1.56944..) / 2 = 1.42189..
>
> (1.42189..+ 2/1.42189..) / 2 = 1.41423..
>
> ....



♦ 设一个数为 x，我们可以有 $x_1=(x+a/x)$，一直循环计算下去，最后 x_1 无限接近于 x，就是我们要找的结果了.

```
class Solution {
public:
    int mySqrt(int a) {
        //注意在此处需要用 long 类型
        long x=a;
        while(x*x>a){
            x=(x+a/x)/2;
        }
        return int(x);
    }
};
```

第四章 未分类

4.1. hash表

- **1160. 拼写单词**

给你一份『词汇表』（字符串数组） words 和一张『字母表』（字符串） `chars`。

假如你可以用 `chars` 中的『字母』（字符）拼写出 `words` 中的某个『单词』（字符串），那么我们就认为你掌握了这个单词。

注意：每次拼写时，`chars` 中的每个字母都只能用一次。

返回词汇表 `words` 中你掌握的所有单词的 长度之和。

示例 1：

```
输入：words = ["cat","bt","hat","tree"], chars = "atach"
输出：6
解释：
可以形成字符串 "cat" 和 "hat"，所以答案是 3 + 3 = 6。
```

示例 2：

```
输入：words = ["hello","world","leetcode"], chars = "welldonehoneyr"
输出：10
解释：
可以形成字符串 "hello" 和 "world"，所以答案是 5 + 5 = 10。
```

提示：

1 <= `words.length` <= 1000
1 <= `words[i].length, chars.length` <= 100
所有字符串中都仅包含小写英文字母

```
class Solution {
public:
    int countCharacters(vector<string>& words, string chars) {
        //字典中的字符出现的次数记录到map中
        unordered_map<char,int>char_cont;
        for(char e:chars)
            char_cont[e]++;
```

```
        bool br=true;
        int ans=0;
        for(string stc:words){
            //将每个单词中每个字母的出现次数记录到map中
            unordered_map<char,int>word_cont;
            for(char c:stc)
                word_cont[c]++;
            br=true;
            for(char c:stc){
                //比较在这个单词中是否有字母出现次数大于字典中的字母数
                if(char_cont[c]<word_cont[c]){
                    br=false;
                    break;
                }
            }
            if(br)
                ans+=stc.size();
        }
        return ans;
    }
};
```

4.2. 串

4.2.1 串的匹配算法

- `BF` 算法
- `KMP`算法

4.3 时间计算

- 计算时间间隔
- 给出年月日，计算是周几

4.4. 位运算

- 异或运算`^`
  - 异或（xor）是一个数学运算符。它应用于逻辑运算。异或的数学符号为“⊕”，计算机符号为“`xor`”，常用"`^`"表示。其运算法则为：

    $a \oplus b = (\neg a \wedge b) \vee (a \wedge \neg b)$.
  - 异或也叫**半加运算**，其运算法则相当于不带进位的二进制加法：二进制下用`1`表示真，`0`表示假，则异或的运算法则为：`0⊕0=0`，`1⊕0=1`，`0⊕1=1`，`1⊕1=0`（同为0，异为1），这些法则与加法是相同的，只是不带进位，所以异或常被认作不进位加法.

^(异或运算)	00(0)	01(1)	10(2)	11(3)	100(4)
00(0)	00^00=00(0)	00^01=01(1)	00^10=10(2)	00^11=11(3)	000^100=100(4)

136. 只出现一次的数字

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

示例 1:

```
输入: [2,2,1]
输出: 1
```

示例 2:

```
输入: [4,1,2,1,2]
输出: 4
```

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int a=0;
        for(int num:nums){
            a^=num;
        }
        return a;
    }
};
```

231. 2的幂

给定一个整数，编写一个函数来判断它是否是 2 的幂次方。

示例 1:

```
输入: 1
输出: true
解释: 20 = 1
```

示例 2:

```
输入: 16
输出: true
解释: 24 = 16
```

示例 3:

```
输入: 218
输出: false
```

```cpp
class Solution {
public:
    bool isPowerOfTwo(int n) {
        return n>0&&(n&(n-1)==0);
    }
};
```

·	2^0	2^1	2^2	2^3	2^4
二进制	1	0010	0100	1000	00010000

- 移位运算(`<<`和`>>`)
  - 右移运算符`>>`

>>	1011	10111	1010001
1	110	1011	101000
2	10	101	10100

  - 左移运算符

1290. 二进制链表转整数

给你一个单链表的引用结点 head。链表中每个结点的值不是 0 就是 1。已知此链表是一个整数数字的二进制表示形式。

请你返回该链表所表示数字的 十进制值 。

示例 1：

```
输入：head = [1,0,1]
输出：5
解释：二进制数 (101) 转化为十进制数 (5)
```

示例 2：

```
输入：head = [0]
输出：0
```

示例 3：

```
输入：head = [1]
输出：1
```

示例 4：

```
输入：head = [1,0,0,1,0,0,1,1,1,0,0,0,0,0,0]
输出：18880
```

示例 5：

```
输入：head = [0,0]
输出：0
```

提示：

链表不为空。
链表的结点总数不超过 30。
每个结点的值不是 0 就是 1。

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    int getDecimalValue(ListNode* head) {
        int ans=0;
        while(head){
            ans=(ans<<1)+head->val;
        }
        return ans;
    }
};
```

- 原地算法

  289 生命游戏

  根据 百度百科 ，生命游戏，简称为生命，是英国数学家约翰·何顿·康威在 1970 年发明的细胞自动机。

  给定一个包含 m × n 个格子的面板，每一个格子都可以看成是一个细胞。每个细胞都具有一个初始状态：1 即为活细胞（live），或 0 即为死细胞（dead）。每个细胞与其八个相邻位置（水平，垂直，对角线）的细胞都遵循以下四条生存定律：

  如果活细胞周围八个位置的活细胞数少于两个，则该位置活细胞死亡；
  如果活细胞周围八个位置有两个或三个活细胞，则该位置活细胞仍然存活；
  如果活细胞周围八个位置有超过三个活细胞，则该位置活细胞死亡；
  如果死细胞周围正好有三个活细胞，则该位置死细胞复活；
  根据当前状态，写一个函数来计算面板上所有细胞的下一个（一次更新后的）状态。下一个状态是通过将上述规则同时应用于当前状态下的每个细胞所形成的，其中细胞的出生和死亡是同时发生的。

  示例：

```
输入：
[
  [0,1,0],
  [0,0,1],
  [1,1,1],
  [0,0,0]
]
输出：
[
  [0,0,0],
  [1,0,1],
  [0,1,1],
  [0,1,0]
]
```

3.5 贪心算法

思想：贪心算法的基本思路是从问题的某一个初始解出发一步一步地进行，根据某个优化测度，每一步都要确保能获得局部最优解。

每一步只考虑一个数据，他的选取应该满足局部优化的条件。

若下一个数据和部分最优解连在一起不再是可行解时，就不把该数据添加到部分解中，直到把所有数据枚举完，或者不能再添加算法停止 。

即从初始解出发，每一步都要考虑最优解

55. 跳跃游戏

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1:

```
输入: [2,3,1,1,4]
输出: true
解释: 我们可以先跳 1 步，从位置 0 到达 位置 1, 然后再从位置 1 跳 3 步到达最后一个位置。
```

示例 2:

```
输入: [3,2,1,0,4]
输出: false
解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。
```

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int result=0;
        for(int i=0;i<nums.size();i++){
            if(i>result)return false;
            result=max(result,nums[i]+i);
        }
        return true;
    }
};
```

面试题 08.11. 硬币

给定数量不限的硬币，币值为25分、10分、5分和1分，编写代码计算n分有几种表示法。(结果可能会很大，你需要将结果模上1000000007)

示例1:

```
 输入: n = 5
 输出: 2
 解释: 有两种方式可以凑成总金额:
5=5
5=1+1+1+1+1
```

示例2:

```
 输入: n = 10
 输出: 4
 解释: 有四种方式可以凑成总金额:
10=10
10=5+5
10=5+1+1+1+1+1
10=1+1+1+1+1+1+1+1+1+1
```

```
class Solution {
public:
    int waysToChange(int n) {
        int max_len=1000000007;
        vector<int>dp(n+1);
        vector<int>coins={1,5,10,25};
        dp[0]=1;
        for(int coin:coins){
            for(int i=coin;i<=n;i++){
                dp[i]=(dp[i]+dp[i-coin])%max_len;
            }
        }
        return dp.back();
    }
};
```

1029. 两地调度

公司计划面试 `2N` 人。第 i 人飞往 A 市的费用为 `costs[i][0]`，飞往 B 市的费用为`costs[i][1]`。

返回将每个人都飞到某座城市的最低费用，要求每个城市都有 `N` 人抵达。

示例：

```
输入：[[10,20],[30,200],[400,50],[30,20]]
输出：110
解释：
第一个人去 A 市，费用为 10。
第二个人去 A 市，费用为 30。
第三个人去 B 市，费用为 50。
第四个人去 B 市，费用为 20。

最低总费用为 10 + 30 + 50 + 20 = 110，每个城市都有一半的人在面试。
```

提示：

1 <= `costs.length` <= 100
`costs.length` 为偶数
1 <= `costs[i][0], costs[i]`[1] <= 1000

```
//思路：先全到 A 地，然后减去到 B 地的额外消费
class Solution {
public:
    int twoCitySchedCost(vector<vector<int>>& costs) {
        vector<int>dox;//每一次到A地或B地的差
        int sum=0;
        for(auto val:costs){
            dox.push_back(val[1]-val[0]);
            sum+=val[0];//每次都到达A地时需要的总费用
        }
        sort(dox.begin(),dox.end());
        for(int i=0;i<dox.size()/2;i++){
            sum+=dox[i];
        }
        return sum;
    }
};
```

3.6. 回溯算法

[百度回溯算法]

基本思想：从一条路往前走，能进则进，不能进则退回来，换一条路再试。

46. 全排列

给定一个 没有重复 数字的序列，返回其所有可能的全排列。

示例:

```
输入: [1,2,3]
输出:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

```
class Solution {
public:
    void depth(vector<vector<int>>&pre,vector<int>nums,int first,int len){
        if(first==len){
            pre.push_back(nums);
            return;
        }
        for(int i=first;i<len;i++){
            swap(nums[first],nums[i]);
            depth(pre,nums,first+1,len);
            swap(nums[first],nums[i]);
        }
    }
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>>pre;
        depth(pre,nums,0,nums.size());
        return pre;
    }
};
```

784. 字母大小写全排列

给定一个字符串S，通过将字符串S中的每个字母转变大小写，我们可以获得一个新的字符串。返回所有可能得到的字符串集合。

示例:

```
输入: S = "a1b2"
输出: ["a1b2", "a1B2", "A1b2", "A1B2"]

输入: S = "3z4"
输出: ["3z4", "3Z4"]

输入: S = "12345"
输出: ["12345"]
```

注意:

S 的长度不超过12。
S 仅由数字和字母组成。

新知识点：大小写转化：S[index]=S[index]^32.

```cpp
class Solution {
public:
    void dfs(string &S,vector<string>&pre,int index){
        if(index==S.size()){
            pre.push_back(S);
            return;
        }
        dfs(S,pre,index+1);
        if(isalpha(S[index])){
            S[index]^=32;
            dfs(S,pre,index+1);
        }
    }
    vector<string> letterCasePermutation(string S) {
        vector<string>pre;
        dfs(S,pre,0);
        return pre;
    }
};
```