贪心算法

1. 假设你有一个数组prices，长度为n，其中prices[i]是股票在第i天的价格，请根据这个价格数组，返回买卖股票能获得的最大收益

   (1) 你可以买入一次股票和卖出一次股票，并非每天都可以买入或卖出一次，总共只能买入和卖出一次，且买入必须在卖出的前面的某一天

   (2) 如果不能获取到任何利润，请返回0

   (3) 假设买入卖出均无手续费

示例：

```
输入：[8,9,2,5,4,7,1]
返回值：5
说明：在第3天(股票价格 = 2)的时候买入，在第6天(股票价格 = 7)的时候卖出，最大利润 = 7-2 = 5 ，不能选择在第2天买入，第3天卖出，这样就亏损7了；同时，你也不能在买入前卖出股票。
```

思路：遍历价格数组一遍，记录历史最低点，然后在每一天考虑这么一个问题：如果我是在历史最低点买进的，那么我今天卖出能赚多少钱？当考虑完所有天数之时，我们就得到了最好的答案。

代码：

```
int maxProfit(vector<int> &prices)
{
    int n = prices.size();
    int min_num = prices[0], maxD = 0;
    for (int i = 1; i < n; i++)
    {
        if (prices[i] < min_num)
            min_num = prices[i];
        if (maxD < prices[i] - min_num)
            maxD = prices[i] - min_num;
    }
    return maxD;
}
```

2. 假设你有一个数组prices，长度为n，其中prices[i]是某只股票在第i天的价格，请根据这个价格数组，返回买卖股票能获得的最大收益
   1. 你可以多次买卖该只股票，但是再次购买前必须卖出之前的股票
   2. 如果不能获取收益，请返回0
   3. 假设买入卖出均无手续费

要求：空间复杂度 O(n)，时间复杂度 O(n)

进阶：空间复杂度 O(1)，时间复杂度 O(n)

示例：

```
输入：[8,9,2,5,4,7,1]
返回值：7
说明：在第1天(股票价格=8)买入，第2天(股票价格=9)卖出，获利9-8=1
      在第3天(股票价格=2)买入，第4天(股票价格=5)卖出，获利5-2=3
      在第5天(股票价格=4)买入，第6天(股票价格=7)卖出，获利7-4=3
      总获利1+3+3=7，返回7
```

思想：由于不限制交易次数，只要今天股价比昨天高，就交易。

代码：

```
int maxProfit(vector<int> &prices)
{
    int n = prices.size();
    int res = 0, diff;
    for (int i = 1; i < n; i++)
    {
        diff = prices[i] - prices[i - 1];
        if (diff > 0)
            res += diff;
    }
    return res;
}
```

3. 在一条环路上有 n 个加油站，其中第 i 个加油站有 gas[i] 升油，假设汽车油箱容量无限，从第 i 个加油站驶往第 (i+1)%n 个加油站需要花费 cost[i] 升油。

   请问能否绕环路行驶一周，如果可以则返回出发的加油站编号，如果不能，则返回 -1。

   题目数据可以保证最多有一个答案。

示例：

```
输入：[1,2,3,4,5],[3,4,5,1,2]
返回值：3
说明：只能从下标为 3 的加油站开始完成 （即第四个加油站）
```

思想：我们首先检查第 0 个加油站，并试图判断能否环绕一周；如果不能，就从第一个无法到达的加油站开始继续检查。

解题思路

- 定义一个rest记录汽车油箱剩余油量。定义一个sum表示恰好走完一圈后剩余油量。
- 遍历所有站点，更新rest和sum。如果剩余油量小于0，则跟换起始站点，重置rest。
- 最后如果sum大于等于0，说明总有一个起始站点合法，返回对应起始站点，否则，返回-1。

代码：

```
int gasStation(vector<int> &gas, vector<int> &cost)
{
    int n = gas.size(); // 加油站个数
    int rest = 0;       //汽车油箱剩余油量
    // start指向起始加油站，sum表示恰好走玩一圈剩余油量
    int start = 0, sum = 0;
    for (int i = 0; i < n; i++)
    {
        rest += gas[i] - cost[i];
        sum += gas[i] - cost[i];
        if (rest < 0)
        {
            start = (i + 1) % n;
            rest = 0;
        }
    }
    if (sum >= 0)
        return start;
    else
        return -1;
}
```

4. 给定一个以字符串表示的数字 num 和一个数字 k ，从 num 中移除 k 位数字，使得剩下的数字最小。如果可以删除全部数字，则结果为 0。

   1.num仅有数字组成

   2.num是合法的数字，不含前导0

   3.删除之后的num，请去掉前导0（不算在移除次数中）

示例：

```
输入："1432219",3
返回值："1219"
说明：移除 4 3 2 后剩下 1219
```

思想：每次都需要从头遍历，如果当前数字比其下一位大，则删除当前数字

代码：

```
char *removeKnums(char *num, int k)
{
    int len = strlen(num), i, j;
    for (i = 0; i < k; i++)
    {
        for (j = 0; j < len - 1; j++)
        {
            // 删除第一个较大的元素
            if (num[j] > num[j + 1])
            {
```

```
                while (j < len - 1)
                {
                    num[j] = num[j + 1];
                    j++;
                }
                num[j] = '\0';
                len--;
                break;
            }
        }
        if (j == len - 1) //每次必须删除一位
        {
            num[j] = '\0';
            len--;
        }
    }
    if (len > 1)
    {
        // 去掉前导括号
        for (i = 0, j = 0; i < len; i++)
        {
            if (num[i] != '0')
                num[j++] = num[i];
            else
                break;
        }
        num[j] = '\0';
    }
    return num;
}
```

5. 假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。

   对每个孩子 i，都有一个胃口值 g[i]，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j，都有一个尺寸 s[j] 。如果 s[j] >= g[i]，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

示例：

```
输入: g = [1,2,3], s = [1,1]
输出: 1
解释:
你有三个孩子和两块小饼干，3个孩子的胃口值分别是：1,2,3。
虽然你有两块小饼干，由于他们的尺寸都是1，你只能让胃口值是1的孩子满足。
所以你应该输出1。
```

思想：为了尽可能满足最多数量的孩子，从贪心的角度考虑，应该按照孩子的胃口从小到大的顺序依次满足每个孩子，且对于每个孩子，应该选择可以满足这个孩子的胃口且尺寸最小的饼干

代码：

```
int findcount(vector<int> g, vector<int> s)
{
    sort(g.begin(), g.end());
    sort(s.begin(), s.end());
    int m = g.size(), n = s.size();
    int i = 0, j = 0, count = 0;
    while (i < m && j < n)
    {
        if (g[i] > s[j])
            j++;
        else
        {
            i++;
            j++;
            count++;
        }
    }
    return count;
}
```

6. 给你一个整数数组 nums ，判断这个数组中是否存在长度为 3 的递增子序列。

   如果存在这样的三元组下标 (i, j, k) 且满足 i < j < k ，使得 nums[i] < nums[j] < nums[k] ，返回 true ；否则，返回 false 。

示例：

```
输入：nums = [2,1,5,0,4,6]
输出：true
解释：三元组 (3, 4, 5) 满足题意，因为 nums[3] == 0 < nums[4] == 4 < nums[5] == 6
```

思想：从左到右遍历数组nums，遍历过程中维护两个变量first 和second，分别表示递增的三元子序列中的第一个数和第二个数，任何时候都有 first<second。

初始时first=nums[0]，second=+∞。对于 1≤i<n，当遍历到下标 i时，令 num=nums[i]，进行如下操作：

1. 如果num>second，则找到了一个递增的三元子序列，返回 true；
2. 否则，如果 num>first，则将 second 的值更新为 num；
3. 否则，将 first 的值更新为 num。

如果遍历结束时没有找到递增的三元子序列，返回false。

上述做法的贪心思想是：为了找到递增的三元子序列，first 和second 应该尽可能地小，此时找到递增的三元子序列的可能性更大。

代码：

```
bool increasingTriplet(vector<int> &nums)
{
    int n = nums.size();
    if (n < 3)

        return false;
```

```
    int first = nums[0], second = INT_MAX, num;
    for (int i = 1; i < n; i++)
    {
        num = nums[i];
        if (num > second)
            return true;
        else if (num > first)
            second = num;
        else
        {
            first = num;
            second = INT_MAX;
        }

    }
    return false;
}
```

7. 给定一个包含非负整数的数组 `nums` ，返回其中可以组成三角形三条边的三元组个数。

示例：

```
输入: nums = [2,2,3,4]
输出: 3
解释:有效的组合是:
    2,3,4 (使用第一个 2)
    2,3,4 (使用第二个 2)
    2,2,3
```

思想：对于正整数 a, b, ca,b,c，它们可以作为三角形的三条边，当且仅当：

a+b>c a+c>b b+c>a

均成立。如果我们将三条边进行升序排序，使它们满足a≤b≤c，那么 a + c > b和 b + c > a一定成立，我们只需要保证 a + b > c。

因此，我们可以将数组nums 进行升序排序，随后使用二重循环枚举 a 和 b。设 a=nums[i], b=nums[j]，为了防止重复统计答案，我们需要保证 i < j。剩余的边 c 需要满足c<nums[i]+nums[j]，我们可以在[j+1,n−1] 的下标范围内使用二分查找（其中 n 是数组nums 的长度），找出最大的满足nums[k]<nums[i]+nums[j] 的下标 k，这样一来，在[j+1, k] 范围内的下标都可以作为边 c 的下标，我们将该范围的长度 k - j累加入答案。当枚举完成后，我们返回累加的答案即可。

代码：

```
int triangleNumber(vector<int> &nums)
{
    int n = nums.size();
    sort(nums.begin(), nums.end());
    int ans = 0, left, right, k;
    for (int i = 0; i < n - 2; i++)
    {

        for (int j = i + 1; j < n - 1; j++)
```

```
        {
            left = j + 1;
            right = n - 1;
            k = j;
            while (left <= right)
            {
                int mid = (left + right) / 2;
                if (nums[mid] < nums[i] + nums[j])
                {
                    k = mid;
                    left = mid + 1;
                }
                else
                    right = mid - 1;
            }
            ans += k - j;
        }
    }
    return ans;
}
```

8. 给定一个区间的集合 intervals ，其中 intervals[i] = [$start_i$, end_i] 。返回 需要移除区间的最小数量，使剩余区间互不重叠 。

示例：

```
输入: intervals = [[1,2],[2,3],[3,4],[1,3]]
输出: 1
解释: 移除 [1,3] 后，剩下的区间没有重叠。
```

思想：我们可以不断地寻找右端点在首个区间右端点左侧的新区间，将首个区间替换成该区间。那么当我们无法替换时，首个区间就是所有可以选择的区间中右端点最小的那个区间。因此我们将所有区间按照右端点从小到大进行排序，那么排完序之后的首个区间，就是我们选择的首个区间。

如果有多个区间的右端点都同样最小怎么办？由于我们选择的是首个区间，因此在左侧不会有其它的区间，那么左端点在何处是不重要的，我们只要任意选择一个右端点最小的区间即可。

当确定了首个区间之后，所有与首个区间不重合的区间就组成了一个规模更小的子问题。由于我们已经在初始时将所有区间按照右端点排好序了，因此对于这个子问题，我们无需再次进行排序，只要找出其中与首个区间不重合并且右端点最小的区间即可。用相同的方法，我们可以依次确定后续的所有区间。

代码：

```
bool cmp(vector<int> a, vector<int> b)
{
    return b[1] > a[1];
}
int eraseOverlapIntervals(vector<vector<int>> &intervals)
{
    int n = intervals.size();
    if (n == 0)
        return 0;
```

```
    int right = intervals[0][1];
    int ans = 1;
    for (int i = 1; i < n; i++)
    {
        if (intervals[i][0] >= right)
        {
            ans++;
            right = intervals[i][1];
        }
    }
    return n - ans;
}
```

回溯法

1. 给一个01矩阵，1代表是陆地，0代表海洋， 如果两个1相邻，那么这两个1属于同一个岛。我们只考虑上下左右为相邻。

   岛屿: 相邻陆地可以组成一个岛屿（相邻:上下左右） 找到最大岛屿面积， 如果没有岛屿，返回0。

   例如：

   输入

   [

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

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

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

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

   [0,0,1,1,1]

   ]

   对应的输出为3

   (注：存储的01数据其实是数字0,1)

示例

```
输入：[[1,1,0,0,0],[0,1,0,1,1],[0,0,0,1,1],[0,0,0,0,0],[0,0,1,1,1]]
返回值：4
```

思路：

矩阵中多处聚集着1，要想统计1聚集的堆数而不重复统计，那我们可以考虑每次找到一堆相邻的1，就将其全部改成0，而将所有相邻的1改成0的步骤又可以使用深度优先搜索（dfs）：当我们遇到矩阵的某个元素为1时，统计首先将其置为了0，然后查看与它相邻的上下左右四个方向，如果这四个方向任意相邻元素为1，则进入该元素，进入该元素之后我们发现又回到了刚刚的子问题，又是把这一片相邻区域的1全部置为0，因此可以用递归实现。

代码：

```
int dfs(vector<vector<int>> &grid, int x, int y)
{
    int m = grid.size(), n = grid[0].size();
    if (x < 0 || x == m || y < 0 || y == n || grid[x][y] != 1)
        return 0;
    grid[x][y] = 0; // 将所有遍历到的点都修改成0
    int ans = 1;
    ans += dfs(grid, x, y - 1);
    ans += dfs(grid, x, y + 1);
    ans += dfs(grid, x - 1, y);
    ans += dfs(grid, x + 1, y);
    return ans;
}
int solve(vector<vector<int>> &grid)
{
    int ans = 0;
    int m = grid.size(), n = grid[0].size();
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            // 遍历到1的位置，进行深度优先搜索
            if (grid[i][j] == 1)
            {
                ans = max(ans, dfs(grid, i, j));
            }
        }
    }
    return ans;
}
```

2. 给出n对括号，请编写一个函数来生成所有的由n对括号组成的合法组合。

示例：

```
输入：n = 3
输出：["((()))","(()())","(())()","()(())","()()()"]
```

思想：相当于一共n个左括号和n个右括号，可以给我们使用，我们需要依次组装这些括号。每当我们使用一个左括号之后，就剩下n-1个左括号和n个右括号给我们使用，结果拼在使用的左括号之后就行了，因此后者就是一个子问题，可以使用递归：

- **终止条件：** 左右括号都使用了n个，将结果加入数组。
- **返回值：** 每一级向上一级返回后续组装后的字符串，即子问题中搭配出来的括号序列。
- **本级任务：** 每一级就是保证左括号还有剩余的情况下，使用一次左括号进入子问题，或者右括号还有剩余且右括号使用次数少于左括号的情况下使用一次右括号进入子问题。

但是这样递归不能保证括号一定合法，我们需要保证左括号出现的次数比右括号多时我们再使用右括号就一定能保证括号合法了，因此每次需要检查左括号和右括号的使用次数。

代码：

```
void backtrack(vector<char>&cur, int k, int open, int close, int n)
{
    if (k== n * 2)
    {
        for(int i=0;i<2*n;i++)
            cout<<cur[i];
        cout<<endl;
        return;
    }
    if (open < n)
    {
        cur[k]='(';
        backtrack(cur, k+1, open + 1, close, n);
    }
    if (close < open)
    {
        cur[k]=')';
        backtrack(cur, k+1, open, close+1, n);
    }
}
void Parenthesis(int n)
{
    vector<char>cur(n);
    backtrack(cur, 0, 0, 0, 3);
}


// 使用容器
void backtrack(vector<string> &ans, string &cur, int open, int close, int n)
{
    if (cur.size() == n * 2)
    {
        ans.push_back(cur);
        return;
    }
    if (open < n)
    {
        cur.push_back('(');
        backtrack(ans, cur, open + 1, close, n);
        cur.pop_back();
    }
    if (close < open)
    {
        cur.push_back(')');
        backtrack(ans, cur, open, close + 1, n);
        cur.pop_back();
    }
}
vector<string> Parenthesis(int n)
{
    vector<string> result;
    string cur;
    backtrack(result, cur, 0, 0, n);
```

```
        return result;
    }
```

3. 给定一个不含重复数字的数组 `nums` ，返回其 所有可能的全排列 。你可以 按任意顺序 返回答案

示例：

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

思路：全排列就是对数组元素交换位置，使每一种排列都可能出现。 那我们考虑递归的几个条件：

- **终止条件：** 要交换位置的下标到了数组末尾，没有可交换的了，那这就构成了一个排列情况，可以加入输出数组。
- **返回值：** 每一级的子问题应该把什么东西传递给父问题呢，这个题中我们是交换数组元素位置，前面已经确定好位置的元素就是我们返还给父问题的结果，后续递归下去会逐渐把整个数组位置都确定，形成一种排列情况。
- **本级任务：** 每一级需要做的就是遍历从它开始的后续元素，每一级就与它交换一次位置。

代码：

```
void backtrack(vector<vector<int>> &res, vector<int> &output, int first, int len)
{
    // 所有数都填完了
    if (first == len)
    {
        res.push_back(output);
        return;
    }
    for (int i = first; i < len; ++i)
    {
        // 动态维护数组
        swap(output[i], output[first]);
        // 继续递归填下一个数
        backtrack(res, output, first + 1, len);
        // 撤销操作
        swap(output[i], output[first]);
    }
}
vector<vector<int>> permute(vector<int> &nums)
{
    vector<vector<int>> res;
    int n=nums.size();
    backtrack(res, nums, 0, n);
    return res;
}
```

4. 给定一个可包含重复数字的序列 `nums` ，按任意顺序 返回所有不重复的全排列。

示例：

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

思想：与上题的区别主要是对于重复元素的处理，跳过已经与开头交换过的元素即可

代码：

```
// 判断重复元素
bool ok(vector<int> &output, int first, int i)
{
    if (i > first)
    {
        for (int t = first; t < i; t++)
            if (output[t] == output[i])
                return false;
        return true;
    }
    else
        return true;
}
void backtrack(vector<vector<int>> &res, vector<int> &output, int first, int len)
{
    // 所有数都填完了
    if (first == len)
    {
        res.emplace_back(output);
        return;
    }
    for (int i = first; i < len; ++i)
    {
        if (ok(output, first, i))
        {
            // 动态维护数组
            swap(output[i], output[first]);
            // 继续递归填下一个数
            backtrack(res, output, first + 1, len);
            // 撤销操作
            swap(output[i], output[first]);
        }
    }
}
vector<vector<int>> permuteUnique(vector<int> &nums)
{
    vector<vector<int>> res;
    int n = nums.size();
    backtrack(res, nums, 0, n);
    return res;
}
```

5. N 皇后问题是指在 n * n 的棋盘上要摆 n 个皇后， 要求：任何两个皇后不同行，不同列也不在同一条斜线上， 求给一个整数 n ，返回 n 皇后的摆法数。

示例：

```
输入：8
返回值：92
```

思想：n个皇后，不同行不同列，那么肯定棋盘每行都会有一个皇后，每列都会有一个皇后。如果我们确定了第一个皇后的行号与列号，则相当于接下来在n-1行中查找n-1个皇后，这就是一个子问题，因此使用递归：

- **终止条件：** 当最后一行都被选择了位置，说明n个皇后位置齐了，增加一种方案数返回。
- **返回值：** 每一级要将选中的位置及方案数返回。
- **本级任务：** 每一级其实就是在该行选择一列作为该行皇后的位置，遍历所有的列选择一个符合条件的位置加入数组，然后进入下一级。

具体做法：

- step 1：对于第一行，皇后可能出现在该行的任意一列，我们用一个数组pos记录皇后出现的位置，下标为行号，元素值为列号。
- step 2：如果皇后出现在第一列，那么第一行的皇后位置就确定了，接下来递归地在剩余的n-1行中找n-1个皇后的位置。
- step 3：每个子问题检查是否符合条件，我们可以对比所有已经记录的行，对其记录的列号查看与当前行列号的关系：即是否同行、同列或是同一对角线。

代码：

```
bool Place(vector<int> &pos, int x, int y)
{
    // 遍历所有已经记录的行
    for (int i = 0; i < x; i++)
    {
        // 不能同行同列同斜线
        if (y == pos[i] || abs(x - i) == abs(y - pos[i]))
            return false;
    }
    return true;
}
void Backtrack(vector<int> &pos, int n, int x, int &res)
{
    if (x == n)
    {
        res++;
        return;
    }
    // 遍历所有列
    for (int i = 0; i < n; i++)
    {
        if (Place(pos, x, i))
        {
            pos[x] = i;

            Backtrack(pos, n, x + 1, res);
```

```
        }
    }
}
int Nqueen(int n)
{
    int res = 0;
    vector<int> pos(n, 0);
    Backtrack(pos, n, 0, res);
    return res;
}


//若题目要求输出所有解决方案呢
bool Place(vector<string> &ans, int k, int t)
{
    int n = ans.size();
    for (int i = 0; i < k; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            if (ans[i][j] == 'Q')
            {
                if ((abs(k - i) == abs(t - j)) || j == t)
                    return false;
            }
        }
    }
    return true;
}
void backtrack(vector<vector<string>> &Queen, vector<string> ans, int k)
{
    int n = ans.size();
    if (k == n)
    {
        Queen.push_back(ans);
        return;
    }
    for (int i = 0; i < n; ++i)
    {
        if (Place(ans, k, i))
        {
            ans[k][i] = 'Q';
            backtrack(Queen, ans, k + 1);
            ans[k][i] = '.';
        }
    }
}
vector<vector<string>> solveNQueens(int n)
{
    vector<vector<string>> Queen;
    vector<string> ans(n, string(n, '.'));
    backtrack(Queen, ans, 0);

    return Queen;
```

```
}
```

6. 给你一个 无重复元素 的整数数组 nums和一个目标整数 target ， 找出nums中可以使数字和为目标数 target 的 所有 不同组合 ， 并以列表形式返回。你可以按 任意顺序 返回这些组合。

   nums中的 同一个 数字可以 无限制重复被选取 。如果至少一个数字的被选数量不同，则两种组合是不同的。

   示例：

```
输入：candidates = [2,3,6,7], target = 7
输出：[[2,2,3],[7]]
解释：
2 和 3 可以形成一组候选，2 + 2 + 3 = 7 。注意 2 可以使用多次。
7 也是一个候选， 7 = 7 。
仅有这两种组合。
```

   思想：如果当前这个数小于当前剩余数，此时有两个选择，取或者不取，如果当前剩余数等于0，则输出

   代码：

```
void dfs(vector<int> &nums, vector<vector<int>> &ans, vector<int> &combine, int target, int
idx)
{
    if (idx == nums.size())
        return;
    if (target == 0)
    {
        ans.push_back(combine);
        return;
    }
    // 直接跳过
    dfs(nums, ans, combine, target, idx + 1);
    if (target >= nums[idx])
    {
        combine.push_back(nums[idx]);
        target -= nums[idx];
        dfs(nums, ans, combine, target, idx);
        combine.pop_back();
    }
}
vector<vector<int>> combinationSum(vector<int> nums, int target)
{
    vector<vector<int>> ans;
    vector<int> combine;
    dfs(nums, ans, combine, target, 0);
    return ans;
}
```