第三章 贪心算法

文章目录

贪心算法

（1）含义

所谓贪心算法是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的仅是在某种意义上的局部最优解。虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解。如单源最短路经问题，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

贪心算法没有固定的算法框架，算法设计的关键是贪心策略的选择。必须注意的是，贪心算法不是对所有问题都能得到整体最优解，选择的贪心策略必须具备无后效性。所谓无后效性是指：“下一时刻的状态只与当前状态有关，而和当前状态之前的状态无关，当前状态是对以往决策的总结”。所以对所采用的贪心策略一定要仔细分析其是否满足无后效性。

> 遵循某种规则，根据当前状态不断贪心地选取当前最优策略的算法设计方法。
> 贪心算法是否可行可通过证明证明出来。
> 贪心算法没有模板和框架，更重要的是思想。

（2）基本要素

1. 贪心选择性质
   所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优解的选择，即贪心选择来达到。贪心策略所使用的前提是：局部最优策略能导致产生全局最优解。
   贪心算法通过解局部最优解策略来达到全局最优解。
2. 最优子结构性质
   当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。

（3）基本思路

1. 建立数学模型来描述问题。
2. 把求解的问题分成若干个子问题。
3. 对每一子问题求解，得到子问题的局部最优解。
4. 把子问题的局部最优解合成原来解问题的一个解。

从问题的某一个初始解出发逐步逼近给定的目标，以尽可能快的地求得更好的解。当达到算法中的某一步不能再继续前进时，算法停止。
不能再继续前进的算法存在的问题：

1. 不能保证求得的最后解是最佳的；
2. 不能用来求最大或最小解问题；
3. 只能求满足某些约束条件的可行解的范围。

（4）实现框架

```
从问题的某一初始解出发;
while （能朝给定总目标前进一步）
{
    利用可行的决策，求出可行解的一个解元素;
}
由所有解元素组合成问题的一个可行解;
```

（5）基本案例

例：贪心法找钱

题目描述

有1元、5元、10元、20元、100元、200元的钞票无穷多章。现使用这些钞票支付X元，最少需要多少张？

解题思路

根据当前需要支付的金额，尽可能多的使用面值较大的钞票。

为什么这么做一定是对的？

面额为1元、5元、10元、20元、100元、200元，任意面额是比自己小的面额的倍数关系。所以当使用一张较大面额钞票时，若使用较小面额钞票替换，则一定需要更多的其他面额的钞票。

故，当前最优解即为全局最优解，贪心成立！

（如果增加7元面额，贪心不成立！举反例：14=7+7优于14=10+1+1+1+1）

代码实现

```
#include<stdio.h>

int main(){
  const int RMB[]={200,100,20,10,5,1};//钞票金额
  const NUM = 6;//6种面值
  int X = 628;//需要支付金额为628
  int count = 0;
  for(int i=0;i<NUM;i++){
    int use = X/RMB[i];//需要面额为RMB[i]的钞票use张
    count += use;  //总计增加use张
```

```
    X = X - RMB[i]*use;//总额-使用RMB[i]已组成的金额，即剩余需要支付的金额
    printf("需要面额为%d的%d张",RMB[i],use);
    printf("剩余需要支付的金额%d.\n",X);
}
printf("总共需要%d张\n",count);
return 0;
}
```

leetcode

例1：分发饼干（455）

题目描述

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 gi ，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 sj 。如果 sj >= gi ，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。

注意：

你可以假设胃口值为正。

一个小朋友最多只能拥有一块饼干。

解题思路

让更多孩子得到满足，有如下规律：

1. 某个糖果如果不能满足某个孩子，则该糖果也一定不能满足需求因子更大的孩子
2. 某个孩子可以用更小的糖果满足，则没必要用更大的糖果满足，因为可以保留更大的糖果满足需求因子更大的孩子（贪心）
3. 孩子的需求因子更小则更容易被满足，故优先从需求因子小的孩子尝试，可以得到正确的结果。

故算法步骤为：

1. 对需求因子数组g与糖果大小数组s进行从小到大排序
2. 按照从小到大的顺序使用各糖果尝试是否可满足某个孩子，每个糖果只尝试1次，若尝试成功，则换下一个孩子尝试；直到发现没有更多孩子或没有更多糖果，循环结束

程序代码

```
    public int findContentChildren(int[] g, int[] s) {
        //g[]为孩子对糖果的需求因子，s[]为糖果的大小因子
            Arrays.sort(g);
        Arrays.sort(s);//对糖果的需求因子和糖果的大小因子进行排序

        int child = 0;//孩子指针，表示满足了child个孩子的需求
        int cookie = 0;//糖果指针，表示尝试了cookie个糖果

        //贪心思想：
        //因为某个孩子如果可以用更小的糖果满足，则没必要用更大的糖果满足。更大的糖果应该满足需求因子更大的孩子。
        //算法思路：
        //1. 对需求因子数组g与糖果大小数组s进行从小到大排序
        //2. 按照从小到大的顺序使用各糖果尝试是否可满足某个孩子，每个糖果只尝试1次，若尝试成功，则换下一个孩子尝试；直到发现没有更多孩子或没有更多糖果，循环结束
        //（孩子的需求数组可按由小到大的次序依此被满足，但糖果不一定由小到大依次满足）
        while(child<g.length && cookie<s.length) {
            //当糖果和孩子同时均未尝试完时
                if(g[child]<=s[cookie]) {
                    //当糖果的大小能够满足孩子的需求时，对下一个孩子进行尝试（因为如果不满足该孩子，则不可能满足需求更高的孩子）
                    child++;
                }
                cookie++;//无论成功失败，每个糖果只尝试一次。尝试下一个糖果
        }

        return child;//返回child即最终满足的孩子个数
    }
```

例2：摇摆排序（376）

题目描述

如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为摆动序列。第一个差（如果存在的话）可能是正数或负数。少于两个元素的序列也是摆动序列。

例如， [1,7,4,9,2,5] 是一个摆动序列，因为差值 (6,-3,5,-7,3) 是正负交替出现的。相反, [1,4,7,2,5] 和 [1,7,4,5,5] 不是摆动序列，第一个序列是因为它的前两个差值都是正数，第二个序列是因为它的最后一个差值为零。

给定一个整数序列，返回作为摆动序列的最长子序列的长度。 通过从原始序列中删除一些（也可以不删除）元素来获得子序列，剩下的元素保持其原始顺序。

```
示例 1:
输入: [1,7,4,9,2,5]
输出: 6
解释: 整个序列均为摆动序列。

示例 2:
输入: [1,17,5,10,13,15,10,5,16,8]
输出: 7
解释: 这个序列包含几个长度为 7 摆动序列，其中一个可为[1,17,10,13,10,16,8]。

示例 3:
输入: [1,2,3,4,5,6,7,8,9]
输出: 2
```

解题思路

贪心思想：当序列有一段连续的 递增（递减）时，为了形成摇摆子序列，只需要保留这段连续的递增（递减）的首位元素，这样更可能使尾部的后一个元素成为摇摆子序列下一个元素。

选择贪心元素目标：成为摇摆子序列的下一个元素的概率更大，摇摆子序列长度++。

程序代码

```
    public int wiggleMaxLength(int[] nums) {
         //贪心规律
             //当序列有一段连续的递增（或递减）时，为形成摇摆子序列，我们只需要保留这段连续的递增（或者递减）的首尾元素
             //这样更可能使尾部后一个元素成为摇摆子序列的下一个元素
             if(nums == null || nums.length == 0)return 0;//序列为空
             if(nums.length < 2)return nums.length;//序列个数小于2时直接为摇摆序列

             int maxLength = 1;//摆动子序列长度至少为1

```

```
        //采用状态机的设计思想，扫描。扫描序列共有3种状态。
        int state = 0;//state表示当前状态，初始状态为BEGIN
        final int BEGIN = 0;//初始状态
        final int UP = 1;//上升状态
        final int DOWN = 2;//下降状态



        for(int i=1;i<nums.length;i++) {//从第2个元素开始扫描
            switch(state) {
            //state表示当前状态，switch表示进行状态转换
            case BEGIN:
                if(nums[i-1]<nums[i]) {
                    state = UP;
                    maxLength++;
                }
                else if(nums[i-1]>nums[i]) {
                    state = DOWN;
                    maxLength++;
                }
                break;
            case UP:
                //由于保留递增区间的末端元素，故只需要在状态转换时加入摇摆子序列。
                if(nums[i-1]>nums[i]) {
                    state = DOWN;
                    maxLength++;
                }
                break;
            case DOWN:
                //由于保留递减区间的末端元素，故只需要在状态转换时加入摇摆子序列。
                if(nums[i-1]<nums[i]) {
                    state = UP;
                    maxLength++;
                }
                break;
            }
        }
        return maxLength;
    }
```

动态规划

```
    final int BEGIN = 0;
    final int UP = 1;
    final int DOWN = 2;

    public int wiggleMaxLength(int[] nums) {
        if(nums == null || nums.length == 0)return 0;
        if(nums.length == 1)return 1;
       int[] pre = new int[nums.length];
       pre[0] = -1;
       int pre_idx = 0;
       int status = BEGIN;
       // 构造可构成摆动数组的前一个数组的末尾元素
       // 为状态开始改变的起始元素
       for(int i=1;i<nums.length;i++) {
                switch(status) {
                case BEGIN:
                    if(nums[i] > nums[i-1]) status = UP;
                    else status = DOWN;
                    break;
                case UP:
                    if(nums[i] < nums[i-1]) {status = DOWN;pre_idx = i-1;}
                    break;
                case DOWN:
                    if(nums[i] > nums[i-1]) {status = UP;pre_idx = i-1;}
                    break;
                }
                pre[i] = pre_idx;
       }
       return constructWiggleArray(nums,pre);
    }

    public int constructWiggleArray(int[] nums,int[] pre) {
        int[] dp = new int[nums.length];    // dp[i]表示长度为i的数组最长摆动序列的长度
            dp[0] = 1;
            for(int i=1;i<nums.length;i++) {
                if(nums[i] == nums[i-1])dp[i] = dp[i-1];    // 特殊:连续重复数字
                // 对于第i个元素，存在选/不选两种可能
                // 若选择，则选择上一个可构成摆动序列的数组最长长度+1；不选择，则是长度为i-1的数组最长长度
                else dp[i] = Integer.max(dp[i-1], dp[pre[i]]+1);
            }

            return dp[nums.length-1];
    }
```

例3：移除K个数字（402）

题目描述

给定一个以字符串表示的非负整数 num，移除这个数中的 k 位数字，使得剩下的数字最小。

注意:

num 的长度小于 10002 且 ≥ k。

num 不会包含任何前导零。

```
示例 1 :
输入: num = "1432219", k = 3
输出: "1219"
解释: 移除掉三个数字 4, 3, 和 2 形成一个新的最小的数字 1219。
示例 2 :
```

```
输入: num = "10200", k = 1
输出: "200"
解释: 移掉首位的 1 剩下的数字为 200. 注意输出不能有任何前导零。
示例 3 :
输入: num = "10", k = 2
输出: "0"
解释: 从原数字移除所有的数字，剩余为空就是0。
```

解题思路

贪心思想：

若去掉某一位数字，为了使得到的新数字最小，需尽可能地让得到的新数字优先最高位最小，其次次高位最小，再其次第三位最小

故应从高位向低位遍历。如果对应的数字大于下一个数字，则把该位数字去掉，得到的数字最小。若去掉K个数字，可从最高位开始遍历，选择最小的数字；共需遍历K次。

可使用栈存储最终结果或删除工作。从高位向低位遍历nums，如果遍历的数字大于栈顶元素，则将数字push入栈；如果小于栈顶元素，则进行弹栈操作，直到

1. 栈为空
2. 不能再删除数字（k=0）
3. 栈顶小于当前元素

程序代码

```
public String removeKdigits(String num, int k) {
     //贪心思想：
     //若去掉某一位数字，为了使得到的新数字最小，需尽可能地让得到的新数字优先最高位最小，其次次高位最小，再其次第三位最小
     //故应从高位向低位遍历。如果对应的数字大于下一个数字，则把该位数字去掉，得到的数字最小。
     //若去掉K个数字，可从最高位开始遍历，选择最小的数字；共需遍历K次
     //可使用栈存储最终结果或删除工作。从高位向低位遍历nums
     //如果遍历的数字大于栈顶元素，则将数字push入栈；如果小于栈顶元素，则进行弹栈操作，直到
     //（1）栈为空（2）不能再删除数字（k=0）（3）栈顶小于当前元素
     //注意两种特殊情况（1）原数组中含有0（2）遍历完后仍可删除
     Stack<Integer> s = new Stack<Integer>();
     Stack<Integer> r = new Stack<Integer>();
     String result = "";
         for(int i=0;i<num.length();i++) {
             //从最高位开始对字符串中数字遍历
             Integer number = Integer.parseInt(num.charAt(i)+"");
             while(s.isEmpty()==false && number<s.peek() && k>0) {
                 //当栈不为空 && 当前遍历元素<栈顶元素 && 仍然可移除元素
                 s.pop();//弹出栈顶元素
                 k--;    //可删除元素减一
             }
             //否则则可将当前遍历元素入栈
             //含有0的情况：注意当栈为空时不能将0元素入栈
             if(!(s.size()==0 && number == 0)) {
                 s.push(number);
             }
         }

         //如果遍历完所有元素，且仍存在可移除元素，则从栈顶开始一一移除
         while(s.isEmpty()==false && k>0) {
             s.pop();
             k--;
         }
         //栈中元素转化为字符串，若移除所有元素（栈为空），则返回"0"
         if(s.isEmpty())result = "0";
         while(!s.isEmpty()) {
             Integer ele = s.pop();
             r.push(ele);
         }
         while(!r.isEmpty()) {
             Integer ele = r.pop();
             result+= ele + "";
         }

         return result;
    }
```

例4a：跳跃游戏（55）

题目描述

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: 从位置 0 到 1 跳 1 步, 然后跳 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。
```

解题思路

（1）贪心思想

若此时处在第i个位置，该位置最远可以跳至第j位置（j=index[i]），故第i位置还可跳至：第i+1、i+2…j-1、j位置。应选择从第i位置应跳至i+1、i+2、…、j-1、j位中可以跳的更远位置的位置，即index[i+1]、index[i+2]、…index[j]最大的那个。

因此，假设该位置为x，index[x]最大，故从位置x触发，可以跳至i+1…j中所有位置可以达的位置，故跳至位置x最理想。

简言之，选择可以跳至最远位置的位置。

（2）解题步骤

1. 求从第i位置最远可跳至第index[i]位置；根据从第i位置最远可跳nums[i]步：index[i] = nums[i]+i;
2. 初始化：
   （1）设置变量jump代表当前所处位置，初始化为0
   （2）设置变量max_index代表从第0位置至第jump位置这个过程中，最远可到达位置，初始化为index[0]
3. 利用jump扫描index数组，知道jump达到index数组尾部或jump超过max_index，扫描过程中，更新max_index

4. 若最终jump为数组长度，则返回true，否则返回false

程序代码

```
public boolean canJump(int[] nums) {
        //贪心思想：
        //若此时处在第i个位置，该位置最远可以跳至第j位置（j=index[i]），故第i位置还可跳至：第i+1、i+2..j-1、j位置
        //从第i位置应跳至i+1、i+2、...、j-1、j位中可以跳的更远位置的位置，即index[i+1]、index[i+2]、..index[j]最大的那个
        //因此，假设该位置为x，index[x]最大，故从位置x触发，可以跳至i+1...j中所有位置可以达的位置，故跳至位置x最理想。
    //简言之，选择可以跳至最远位置的位置。

        //最远跳跃nums数组存储第i个位置可以跳nums[i]步
        //最远达到位置index数组表示nums数组中从第i个位置开始跳，最远可以跳index[i]步：index[i] = nums[i]+1;
    int[] index = new int[nums.length];
    for(int i=0;i<nums.length;i++)
            index[i] = nums[i]+i;

    int jump = 0;//jump表示当前所处位置，初始化为0
    int max_index = index[0];//max_index表示从第0位置至第jump位置中可到达的最远位置，初始化为index[0]

    //利用jump遍历index数组，直到jump遍历完index数组（能够到达最后一个位置）
    //或者jump>max_index（即当前所处位置超过了可跳跃最大位置，故该位置及其之后的位置不可达）
    //若jump>max_index && jump<index.length则说明可跳跃的最大位置<数组长度，因此无法到达数组最后一个位置
    while(jump<index.length && jump<=max_index) {
            if(max_index < index[jump])max_index = index[jump];//若有可以跳跃更远位置，则更新max_index数组
            jump++;//继续遍历
        }
        if(jump == index.length) {
            //jump遍历完所有元素，则说明可以到达最后一个位置
            return true;
        }else {
            return false;
        }
    }
```

例4b：跳跃游戏2（45）

题目描述

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。你的目标是使用最少的跳跃次数到达数组的最后一个位置。

```
示例:
输入: [2,3,1,1,4]
输出: 2
解释: 跳到最后一个位置的最小跳跃数是 2。
从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。
说明:
假设你总是可以到达数组的最后一个位置。
```

解题思路

贪心思想

在到达某点之前若一直不跳跃，发现从该点不能跳到更远的地方了，在这之前肯定有次必要的跳跃。因此，如果希望最少跳跃达到终点，则需要明确何时进行跳跃最合适。

因为假设总是可以到达数组的最后一个位置。因此如果无法达到更远的位置时，在此之前一定可以跳到一个到达更远位置的位置。

算法步骤

1. 设置current_max_index为当前可到达最远位置
2. 设置pre_max_index为在遍历各个位置的过程中，各个位置可到达的最远位置
3. 设置jump_min为最少跳跃的次数
4. 利用i遍历nums数组，若i超过current_max_index，jump_min加1，current_max_index = pre_max_index
5. 遍历过程中，若nums[i]+i(index[i])更大，需更新pre_max_index = index[i]

程序代码

```
public int jump(int[] nums) {
     //贪心思想：
         //在到达某点之前若一直不跳跃，发现从该点不能跳到更远的地方了，在这之前肯定有次必要的跳跃。
         //因此，如果希望最少跳跃达到终点，则需要明确何时进行跳跃最合适。
         //因为假设总是可以到达数组的最后一个位置。因此如果无法达到更远的位置时，在此之前一定可以跳到一个到达更远位置的位置。
         //因此，遍历nums中每一个元素（假设不跳跃），并记录所有遍历位置中可到达的最大位置，
         //若遍历的位置i超过当前可以到达的最远位置，则将最少跳跃的次数++，并将当前可以到达位置更新为遍历过位置中最远位置
         if(nums.length<2)return 0;//若数组长度<2，说明不用跳跃，返回0
         int current_max_index = nums[0];//当前可到达最远距离，初始化为nums[0]
         int pre_max_index = nums[0];//遍历过位置可到达最远距离，初始化为nums[0]
         int jump_min = 1;//最少跳跃次数，初始化为1
         int[] index = new int[nums.length];//最远距离数组，存储nums中每个位置最远可跳跃的位置
         for(int i=0;i<nums.length;i++)
             index[i] = nums[i]+i;

         for(int i=0;i<nums.length;i++) {
             //遍历nums数组
             if(i>current_max_index) {
                 //如果遍历位置i超过可以到达最远位置，则在该位置之前一定可以到达一个更远的位置（基础：一定可以到达终点）
                 //故应该选择这个更远的位置开始跳跃，并更新可以到达最远位置

                 jump_min++;//需要进行一次跳跃
                 current_max_index = pre_max_index;//选择遍历位置中可跳跃的最远位置
             }
             //遍历过程中更新pre_max_index
             if(pre_max_index<index[i])
                 pre_max_index = nums[i] + i;
         }

         return jump_min;
    }
```

例5：射击气球（452）

题目描述

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以y坐标并不重要，因此只要知道开始和结束的x坐标就足够了。开始坐标总是小于结束坐标。平面内最多存在104个气球。

一支弓箭可以沿着x轴从不同点完全垂直地射出。在坐标x处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足 xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

```
Example:
输入:
[[10,16], [2,8], [1,6], [7,12]]
输出:
2
解释:
对于该样例，我们可以在x = 6（射爆[2,8],[1,6]两个气球）和 x = 11（射爆另外两个气球）。
```

解题思路

贪心规律

1. 对于某个气球，至少需要使用一只弓箭将它击穿
2. 在这只气球将其击穿的同时，尽可能击穿其他更多的气球

算法思路

1. 对各个气球进行排序，按照气球的左端点从小到大排序
2. 遍历气球数组，同时维护一个射击区间，在满足可以将当前气球射穿的情况下，尽可能击穿更多的气球，每击穿一个新的气球，更新一次射击区间（保证射击可以将新气球也击穿）
3. 如果新的气球没办法被击穿，则需要增加一名弓箭手，即维护一个新的射击区间（将该气球击穿），然后继续遍历气球数组。

程序代码

```
    public int findMinArrowShots(int[][] points) {
        //贪心规律：
            //1.对于某个气球，至少需要使用一只弓箭将它击穿
            //2.在这只气球将其击穿的同时，尽可能击穿其他更多的气球
            //算法思路：
            //1.对各个气球进行排序，按照气球的左端点从小到大排序
            //2.遍历气球数组，同时维护一个射击区间，在满足可以将当前气球射穿的情况下，尽可能击穿更多的气球，每击穿一个新的气球，更新一次射击区间
            //（保证射击可以将新气球也击穿）
            //3.如果新的气球没办法被击穿，则需要增加一名弓箭手，即维护一个新的射击区间（将该气球击穿），然后继续遍历气球数组。
            if(points.length == 0) return 0;//传入数据为空，直接返回0
            //将传入数据列表化
            List<Point> pointList = new ArrayList();
            for(int i=0;i<points.length;i++) {
                Point p = new Point(points[i][0],points[i][1]);
                pointList.add(p);
            }
            //将气球按照左端点的大小进行排序
            Collections.sort(pointList, new Comparator<Point>(){
                @Override
                public int compare(Point p1, Point p2) {
                    return p1.start - p2.start;
                    }
                });
            int shot_num = 1;//射击次数初始为1
            int shot_begin = pointList.get(0).start;//射击区间起点初始为第一个气球数组的起点
            int shot_end = pointList.get(0).end;//射击区间终点初始为第一个气球数组的终点

            for(int i=0;i<pointList.size();i++) {
                //遍历气球数组
                if(pointList.get(i).start<=shot_end) {
                    //当前气球数组起点在射击区间终点内，则说明可以射击当前气球，则可更新射击区间（取可射击到当前气球的区间交集）
                    shot_begin = pointList.get(i).start;
                    if(shot_end>pointList.get(i).end)shot_end = pointList.get(i).end;
                }else {
                    //当前气球数组起点在射击区间终点外，没有重合，则说明无法射击当前气球，则不能更新射击区间
                    //为了使当前气球被射穿，应该增加射击次数，增加一个新的射击区间
                    shot_num++;
                    shot_begin = pointList.get(i).start;
                    shot_end = pointList.get(i).end;
                }
            }
            return shot_num;
    }

    //气球端点
    public class Point{
            public Integer start;
            public Integer end;

            Point(Integer start,Integer end){
                this.start = start;
                this.end = end;
            }
    }
```

例6：最优加油方法

题目描述

已知一条公路上，有一个起点与一个终点，这之间有n个加油站。

已知从这n个加油站到终点的距离d与各个加油站可以加油的量l，起点位置至终点位置的距离L与起始时刻邮箱中汽油量P;

假设使用1个单位的汽油即走1个单位的距离，油箱没有上限，最少加几次油，可以从起点开至终点？（如果无法到达终点，返回-1）

解题思路

解题思路同前跳跃问题II——选择最合适的点进行跳跃（无法继续前进时，选择之前能够跳跃的最远距离的点进行跳跃）

本题应该解决问题：如何选择最优的加油站进行加油？

1. 何时加油？油用光时加油最合适。
2. 选择哪个加油站加油？选择加油量最多的加油站加油最合适

贪心思想

在油即将用光时，选择油量最多的加油站加油。这样能保证既能到达终点，且加油的次数最少

算法思路

1. 设置一个最大堆，用来存储经过的加油站的汽油量。
2. 按照从起点至终点的方向，遍历各个加油站之间的距离。
3. 每次需要走两个加油站之间的距离d，如果发现油不够走距离d时，从最大堆中取出一个油量添加，直到可以足够走距离d。（贪心思想）
4. 如果把最大堆的汽油都添加仍不够行进距离d，则无法达到终点
5. 当前油量减少d，并将当前加油站油量添加至最大堆

程序代码

```
    public int getMinimumStop(int L,int P,int[][] stations) {
            //解题思路同前跳跃问题II——选择最合适的点进行跳跃（无法继续前进时，选择之前能够跳跃的最远距离的点进行跳跃）
            //本题应该解决问题：如何选择最优的加油站进行加油？1、何时加油？油用光时加油最合适。2、选择哪个加油站加油？选择加油量最多的加油站加油最合适
            //贪心思想：
            //在油即将用光时，选择油量最多的加油站加油。这样能保证既能到达终点，且加油的次数最少
            //算法思路：
            //1、设置一个最大堆，用来存储经过的加油站的汽油量。
            //2、按照从起点至终点的方向，遍历各个加油站之间的距离。
            //3、每次需要走两个加油站之间的距离d，如果发现油不够走距离d时，从最大堆中取出一个油量添加，直到可以足够走距离d。（贪心思想）
            //4、如果把最大堆的汽油都添加仍不够行进距离d，则无法达到终点
            //5、当前油量减少d，并将当前加油站油量添加至最大堆

            //L为起始点到终点距离，P为起点初始汽油量
            //List<Station>stations为n个加油站的信息，每个加油站stations，其中stations[i][0]（加油站至终点站距离），stations[i][1]（加油站汽油量）
            //创建一个最大堆，用来存储经过的加油站的汽油量（java大顶堆定义格式，PriorityQueue默认为最小堆，若为最大堆则需复写comparator）
            PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(stations.length,new Comparator<Integer>(){
                @Override
                public int compare(Integer a,Integer b){
                    return b-a;
                }
            });
            int minStop = 0;//初始化最小加油次数，默认为0
            int currentL = L;//currentL为当前位置与终点的距离，初始为L
            int currentP = P;//currentP为当前油量，初始为P

            //将传入数据列表化
            List<Station> stationList = new ArrayList();
            Station end_station = new Station(0,0);//终点站
            stationList.add(end_station);//将终点站加入，用来判断是否能到达终点
            for(int i=0;i<stations.length;i++) {
                Station s = new Station(stations[i][0],stations[i][1]);
                stationList.add(s);
            }
            //将加油站按照与终点的距离由远到近（由大到小）进行排序
            Collections.sort(stationList, new Comparator<Station>(){
                @Override
                public int compare(Station s1, Station s2) {
                    return s2.distance - s1.distance;
                    }
                });

            for(int i=0;i<stationList.size();i++) {
                //遍历各个加油站
                int d = currentL - stationList.get(i).distance;//当前位置与下一个加油站位置
                while(currentP < d && !maxHeap.isEmpty()) {
                    //若当前油量无法行驶到下一个加油站，且仍可以加油（最大堆不为空），则从最大堆不断取出加油站进行加油。
                    //直到油量满足或者最大堆为空
                    currentP += maxHeap.poll();
                    minStop++;
                }
                if(currentP < d && maxHeap.isEmpty())//无法到达下一个加油站且无法加油，说明到达终点前无法继续前进，返回-1
                    return -1;
                //前进到下一个加油站，将这个加油站加入最大堆，并更新当前油量和当前距离
                maxHeap.add(stationList.get(i).cusume);
                currentP  = currentP - d;
                currentL  = stationList.get(i).distance;
            }

            return minStop;
    }
    //加油站类
    public class Station{
            public Integer distance;//加油站至终点的距离
            public Integer cusume;//加油站所加汽油量

            Station(Integer distance,Integer cusume){
                this.distance = distance;
                this.cusume = cusume;
            }
    }
```

剑指offer

例1：剪绳子（66）

题目描述

给你一根长度为n的绳子，请把绳子剪成m段（m、n都是整数，n>1并且m>1），每段绳子的长度记为k[0],k[1],...,k[m]。请问k[0]xk[1]x...xk[m]可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

输入描述:

输入一个数n，意义见题面。（2 <= n <= 60）

输出描述:

输出答案。

程序代码

```java
// 66. 剪绳子
// 给你一根长度为n的绳子，请把绳子剪成m段（m、n都是整数，n>1并且m>1），
// 每段绳子的长度记为k[0],k[1],...,k[m]。请问k[0]xk[1]x...xk[m]可能的最大乘积是多少？
// 例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。
List<Integer> maxMultiList = new ArrayList<Integer>();  // 最大乘积列表
public int cutRope(int target) {
        // 1. 长度为target的绳子可分为1...n段
        // 2. 分别对1...n段绳子求能得到最大乘积的结果
        //    根据数学计算，可知，当a+b+c+..和相等(=target)的情况下，最大乘积即a,b,c..间差值最小的情况
        // 3. 比较所有分段方式的最大乘积，得到绳子减法的最大乘积
    for(int i=1;i<target;i++)
        getMaxMultiOfMPart(target,i+1);
    Collections.sort(maxMultiList);
    if(maxMultiList!=null && maxMultiList.size()>0)return maxMultiList.get(maxMultiList.size()-1);
    return 0;
}

public void getMaxMultiOfMPart(int n,int m) {
        // 求长度为n，分成m段的绳子的最大乘积
        int[] ropePart = new int[m];
        int multiResult = 1;
        int i = 0;          // i表示第i+1段绳子的长度
        while(i != m) {
            ropePart[i] = n/(m-i);
            n -= n/(m-i);
            i++;
        }
        for(i=0;i<m;i++)multiResult *= ropePart[i];
        maxMultiList.add(multiResult);
    }

    public static void main(String[] args) {
        Greedy greedy = new Greedy();
        System.out.println(greedy.cutRope(2));
    }
```

2019校招

例1：安置路灯（3）

题目描述

小Q正在给一条长度为n的道路设计路灯安置方案。
为了让问题更简单,小Q把道路视为n个方格,需要照亮的地方用’.‘表示, 不需要照亮的障碍物格子用’X’表示。
小Q现在要在道路上设置一些路灯, 对于安置在pos位置的路灯, 这盏路灯可以照亮pos - 1, pos, pos + 1这三个位置。
小Q希望能安置尽量少的路灯照亮所有’.‘区域, 希望你能帮他计算一下最少需要多少盏路灯。
输入描述:
输入的第一行包含一个正整数t(1 <= t <= 1000), 表示测试用例数
接下来每两行一个测试数据, 第一行一个正整数n(1 <= n <= 1000),表示道路的长度。
第二行一个字符串s表示道路的构造,只包含’.‘和’X’。
输出描述:
对于每个测试用例, 输出一个正整数表示最少需要多少盏路灯。

程序代码

```java
// 3. 安置路灯（贪心）
// 每遇见一个.（位置i）则在位置i+1放置路灯
// 贪心思想：尽可能地多照亮后面的道路
public void putLightInRoadTest() {
    Scanner sc = new Scanner(System.in);
    int t = sc.nextInt();   // 共有t个测试样例

    while(t-->0) {
            int n = sc.nextInt();   // 道路的长度
            String s = sc.next();   // 道路的构造
            putLightInRoad(n,s);
    }
}

public void putLightInRoad(int n,String s) {
    // 长度为n的道路s最少需要路灯数目
    int lightNum = 0;
    char[] road = s.toCharArray();
    int idx = 0;        //  从首字符开始遍历道路字符串
    while(idx<n) {
        if(road[idx] == '.') {
            road[idx] = 'X';
            if(idx+1<n)road[idx+1] = 'X';
            if(idx+2<n)road[idx+2] = 'X';
            lightNum++;
            idx = idx + 2;
        }
        else idx++;
    }
    System.out.println(lightNum);
}
```