
第三课 贪心算法

林沐

内容概述

1.7道经典贪心算法的相关题目

预备知识:贪心法找钱

例1:分糖果 (easy) (排序、贪心)

例2:摇摆序列 (medium) (贪心)

例3:移除K个数字 (medium) (栈、贪心)

例4-a:跳跃游戏 (medium) (贪心)

例4-b:跳跃游戏2 (hard) (贪心)

例5:射击气球 (medium) (排序、贪心)

例6:最优加油方法 (hard) (堆、贪心)

2.详细讲解题目解题方法、代码实现

预备知识:贪心法，钞票支付问题

有1元、2元、5元、10元、20元、50元、100元的钞票**无穷多张**。现使用这些钞票**支付** X 元，**最少**需要多少张？

例如， $X = 628$

最佳支付方法：

6张100块的，1张20块的，1张5块的，1张2块的，1张1块的；
共需要 $6+1+1+1+1=10$ 张。

直觉告诉我们：**尽可能多**的使用**面值较大**的钞票！

贪心法：遵循某种规律，不断**贪心**的选取**当前最优**策略的算法设计方法。

预备知识:分析

为何这么做**一定**是对的?

面额为1元、2元、5元、10元、20元、50元、100元，每个面额是比自己小的面额的**2倍或以上**。

所以当使用一张较大面额钞票时，若用**较小面额钞票**替换，需要**两张或更多**的钞票！

例如：

$$2 = 1 + 1$$

$$5 = 2 + 2 + 1$$

$$10 = 5 + 5$$

$$20 = 10 + 10$$

$$50 = 20 + 20 + 10$$

$$100 = 50 + 50$$

故，**当前最优解**即为**全局最优解**，贪心成立！

思考：如果增加**7元面额**，贪心**还成立吗**？

预备知识:实现

```
需要面额为100的6张, 剩余需要支付RMB 28.  
需要面额为50的0张, 剩余需要支付RMB 28.  
需要面额为20的1张, 剩余需要支付RMB 8.  
需要面额为10的0张, 剩余需要支付RMB 8.  
需要面额为5的1张, 剩余需要支付RMB 3.  
需要面额为2的1张, 剩余需要支付RMB 1.  
需要面额为1的1张, 剩余需要支付RMB 0.  
总共需要10张RMB  
请按任意键继续. . .
```

```
#include <stdio.h>
```

```
int main() { //人民币面额
    const int RMB[] = {100, 50, 20, 10, 5, 2, 1};
    const int NUM = 7; // 7种面额

    int X = 628; //用RMB最少多少张, 组成X?
    int count = 0;
    for (int i = 0; i < NUM; i++) {
        int use = X / RMB[i]; //需要面额为RMB[i]的use张
        count += use; //总计增加use张
        X = X - RMB[i] * use; //将总额减去使用RMB[i]已组成的金额
        printf("需要面额为%d的%d张, ", RMB[i], use);
        printf("剩余需要支付RMB %d.\n", X);
    }

    printf("总共需要%d张RMB\n", count);
    return 0;
}
```

例1:分糖果

已知一些**孩子**和一些**糖果**，每个孩子有**需求因子g**，每个糖果有**大小s**，当某个糖果的大小s \geq 某个孩子的需求因子g时，代表该糖果可以满足该孩子；求使用这些糖果，最多能满足多少孩子？(注意，某个孩子最多只能用**1个**糖果满足)

例如，需求因子数组g = [5, 10, 2, 9, 15, 9]；糖果大小数组s = [6, 1, 20, 3, 8]；**最多**可以满足3个孩子。

```
class Solution {  
public:                                     //孩子们的需求因子 g数组  
    int findContentChildren(std::vector<int>& g, std::vector<int>& s){  
    } //返回最多可以有多少孩子被满足      //糖果的大小 s数组  
};
```

选自 **LeetCode 455. Assign Cookies**

<https://leetcode.com/problems/assign-cookies/description/>

难度:**Easy**

例1:思考

例如，需求因子数组 $g = [5, 10, 2, 9, 15, 9]$ ；糖果大小数组 $s = [6, 1, 20, 3, 8]$ 。

为了**更明显**的判断某个孩子可以被某个糖果满足，对 g, s 进行**排序后观察**：

$g = [2, 5, 9, 9, 10, 15]$ ； $s = [1, 3, 6, 8, 20]$ 。

- 1.是否可以**直接暴力枚举**，对**每个**糖果都尝试**是否可以**满足某个孩子？
- 2.当某个孩子可以被**多个糖果**满足时，是否需要**优先**用某个糖果满足这个孩子？
- 3.当某个糖果可以满足**多个孩子**时，是否需要**优先**满足某个孩子？

例1:贪心规律

需求因子 数组 $g = [2, 5, 9, 9, 10, 15]$; **糖果大小** 数组 $s = [1, 3, 6, 8, 20]$ 。

核心目标: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。

如,

糖果1($s = 1$)不能满足孩子1($g = 2$), 则不能满足孩子2、孩子3、...、孩子7;

糖果2($s = 3$)不能满足孩子2($g = 5$), 则不能满足孩子3、孩子4、...、孩子7;

...

2.某个孩子可以用**更小**的糖果满足, 则没必要用**更大糖果**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!)**

如,

孩子1($g = 2$), 可以被糖果2($s = 3$)满足, 则没必要用糖果3、糖果4、糖果5满足;

孩子2($g = 5$), 可以被糖果3($s = 6$)满足, 则没必要用糖果4、糖果5满足;

...

3.孩子的需求因子更小则其更容易被满足, 故**优先**从**需求因子小的**孩子尝试, 用某个糖果满足一个较大需求因子的孩子或满足一个较小需求因子的孩子**效果是一样的**(最终满足的总量不变)。**(贪心!)**

例1:算法思路

- 1.对需求因子数组g与糖果大小数组s进行从小到大的排序。
- 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；直到发现没更多的孩子或者没更多的糖果，循环结束。

图1:

$g = 2, 5, 9, 9, 10, 15$

↑ $child = 0$

$s = 1, 3, 6, 8, 20$

↑ $cookie = 0 ; cookie++$

图3:

$g = 2, 5, 9, 9, 10, 15$

↑ $child = 1 ; child++$

$s = 1, 3, 6, 8, 20$

↑ $cookie = 2 ; cookie++$

图2:

$g = 2, 5, 9, 9, 10, 15$

↑ $child = 0 ; child++$

$s = 1, 3, 6, 8, 20$

↑ $cookie = 1 ; cookie++$

图4:

$g = 2, 5, 9, 9, 10, 15$

↑ $child = 2$

$s = 1, 3, 6, 8, 20$

↑ $cookie = 3 ; cookie++$

例1:算法思路

图5:

$g = 2, 5, 9, 9, 10, 15$
 ↑ $child = 2$; $child++$
 $s = 1, 3, 6, 8, 20$
 ↑ $cookie = 4$; $cookie++$

图6: 没糖果了!

$g = 2, 5, 9, 9, 10, 15$
 ↑ $child = 3$
 $s = 1, 3, 6, 8, 20$
 ↑ $cookie = 5$

最终:

使用糖果2($s=3$)满足孩子1($g=2$);

使用糖果3($s=6$)满足孩子2($g=5$);

使用糖果5($s=20$)满足孩子3($g=3$);

child变量即为**最后结果**! 糖果1与糖果4没有被使用到。

例1:实现，课堂练习

```
#include <vector>
#include <algorithm>
class Solution {
public:
    int findContentChildren(std::vector<int>& g, std::vector<int>& s) {
        std::sort(g.begin(), g.end());
        std::sort(s.begin(), s.end()); //对孩子的需求因子g与糖果大小s两数组排序
        int child = 0;
        int cookie = 0; //child代表已满足了几个孩子，cookie代表尝试了几个糖果
        while (1) {
            if (g[child] <= s[cookie]) { //当孩子孩子的满足因子小于或等于糖果大小时
                2
            }
            3
        }
        return child; //最终child即为得到满足的孩子的个数
    }
};
```

3分钟时间填写代码，
有问题随时提出！

例1:实现

```
#include <vector>
#include <algorithm>
class Solution {
public:
    int findContentChildren(std::vector<int>& g, std::vector<int>& s) {
        std::sort(g.begin(), g.end());
        std::sort(s.begin(), s.end()); //对孩子的需求因子g与糖果大小s两数组排序
        int child = 0;
        int cookie = 0; //child代表已满足了几个孩子，cookie代表尝试了几个糖果
        while (child < g.size() && cookie < s.size()) { //当孩子或糖果同时均未尝试完时
            if (g[child] <= s[cookie]) { //当孩子孩子的满足因子小于或等于糖果大小时
                child++; //该糖果满足了孩子，孩子指针child向后移动
            }
            cookie++; //无论成功或失败，每个糖果只尝试一次，cookie向后移动
        }
        return child; //最终child即为得到满足的孩子的个数
    }
};
```

例1:测试与leetcode提交结果

```
int main() {  
    Solution solve;  
    std::vector<int> g;  
    std::vector<int> s;  
    g.push_back(5);  
    g.push_back(10);  
    g.push_back(2);  
    g.push_back(9);  
    g.push_back(15);  
    g.push_back(9);  
    s.push_back(6);  
    s.push_back(1);  
    s.push_back(20);  
    s.push_back(3);  
    s.push_back(8);  
    printf("%d\n", solve.findContentChildren(g, s));  
    return 0;  
}
```

[Assign Cookies](#)

Submission Details

21 / 21 test cases passed.

Status: **Accepted**

Runtime: 46 ms

Submitted: 1 hour, 52 minutes ago

3
请按任意键继续 . . .

例2:摇摆序列

一个整数序列，如果两个相邻元素的差恰好正负(负正)**交替出现**，则该序列被称为**摇摆序列**。一个小于2个元素的序列**直接**为摇摆序列。

例如:

序列 [1, 7, 4, 9, 2, 5]，相邻元素的差 (6, -3, 5, -7, 3)，该序列为摇摆序列。

序列 [1,4,7,2,5] (3, 3, -5, 3)、 [1,7,4,5,5] (6, -3, 1, 0)不是摇摆序列。

给一个随机序列，求这个序列**满足摇摆序列**定义的**最长子序列**的长度。

例如:

输入[1,7,4,9,2,5]，结果为6；输入[1,17,5,10,13,15,10,5,16,8]，结果为7([1,17,10,13,10,16,8])；输入[1,2,3,4,5,6,7,8,9]，结果为2。

```
class Solution {
public:
    int wiggleMaxLength(std::vector<int>& nums) {
    }
};
```

选自 **LeetCode 376. Wiggle Subsequence**

<https://leetcode.com/problems/wiggle-subsequence/description/>

难度:**Medium**

例2:思考

[1, 17, 5, 10, 13, 15, 10, 5, 16, 8], **整体**不是摇摆序列:

观察该序列**前6位**: [1, 17, **5, 10, 13, 15**...]; **橙色**部分为上升段:

其中它有3个子序列是摇摆序列:

[1, 17, 5, 10, ...]

[1, 17, 5, 13, ...]

[1, 17, 5, 15, ...]

在不清楚原始序列的7位是什么的情况下, **只看前6位**, 摇摆子序列的**第四位**从**10, 13, 15**中**选择1个**数。

我们应该选择**哪个最好**呢?

思考**半分钟**!

例2:分析

摇摆子序列:[1, 17, 5, 10, ...]、[1, 17, 5, 13, ...]、[1, 17, 5, 15, ...]

原始序列: [1, 17, 5, 10, 13, 15, 10, 5, 16, 8]

第三位 ↓ 第七位 ↓

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++

摇摆子序列: [1, 17, 5, 10, 13, 15, ...]

贪心!!

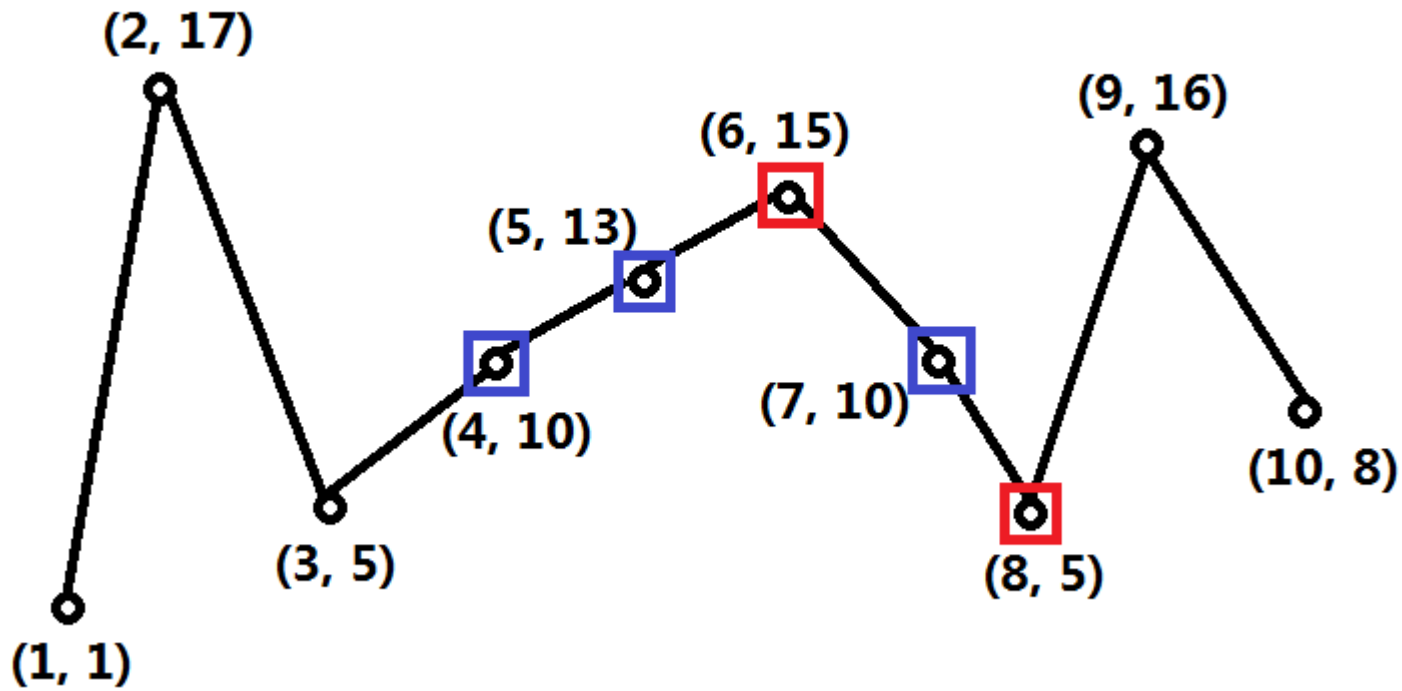
第4位选择最大的15, 更可能使得原始第7位成为摇摆序列的下一个元素

第三位 ↑ 其中的某一个 是第4位 ↑

例2:贪心规律

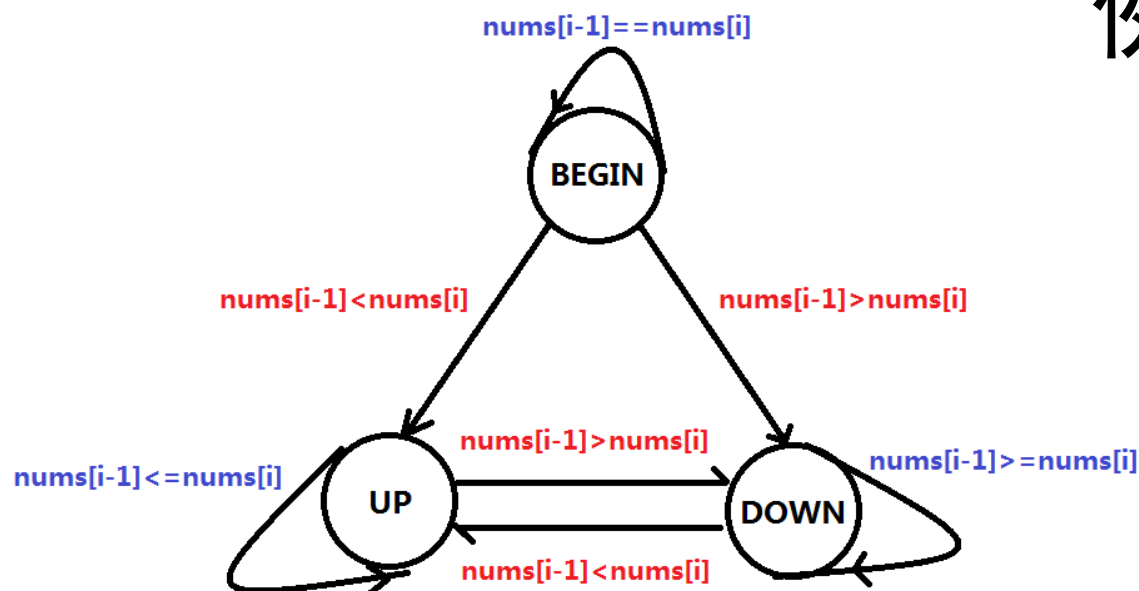
当序列有一段**连续的递增(或递减)**时，为形成**摇摆子序列**，我们只需要**保留**这段连续的递增(或递减)的**首尾元素**，这样**更可能**使得尾部的后一个元素成为摇摆子序列的下一个元素。

[1, 17, 5, 10, 13, 15, 10, 5, 16, 8]



当前扫描完成的摇摆子序列状态: BEGIN(初始), UP(上升), DOWN(下降)

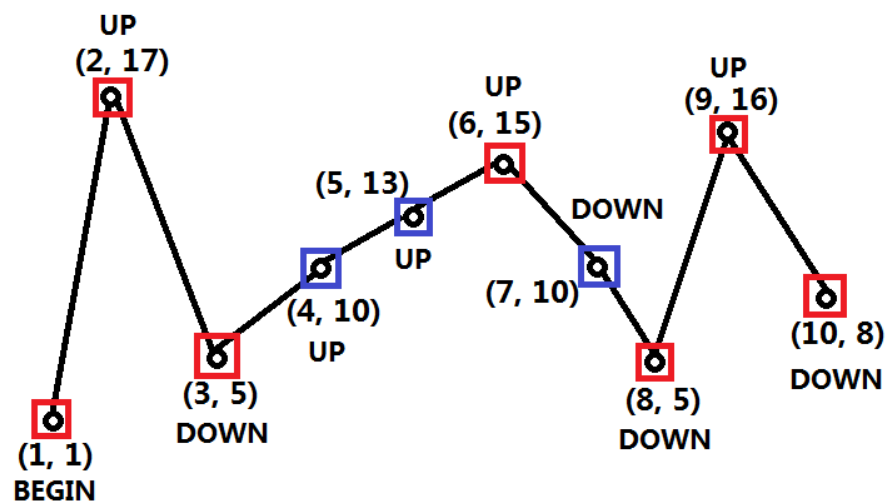
当状态转换时最长摇摆子序列长度 $\text{max_length}++$



$\text{nums}[i-1]$ 代表正在扫描的原始序列元素的前一个元素，同时是最长摇摆子序列的最后一个元素

$\text{nums}[i]$ 代表正在扫描的新元素

[1, 17, 5, 10, 13, 15, 10, 5, 16, 8]



例2: 算法思路

例2:实现，课堂练习

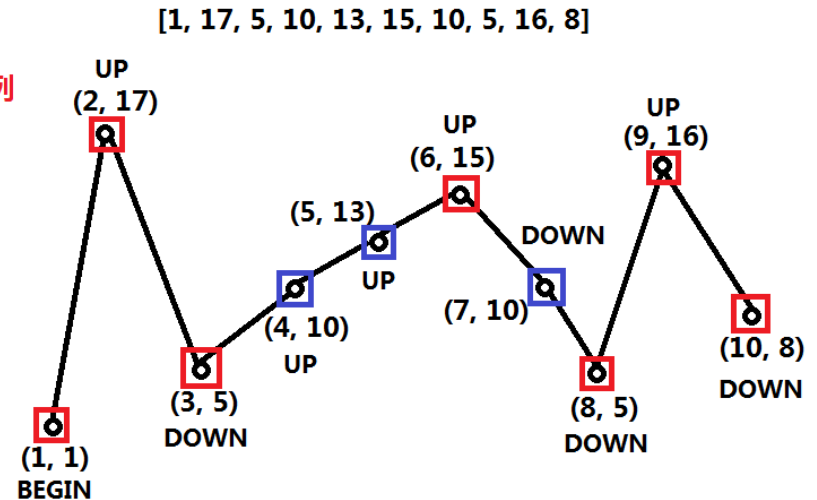
```
#include <vector>
class Solution {
public:
    int wiggleMaxLength(std::vector<int>& nums) {
        if (nums.size() < 2) {
            return nums.size(); //序列个数小于2时直接为摇摆序列
        }
        static const int BEGIN = 0;
        static const int UP = 1; //扫描序列时的三种状态
        static const int DOWN = 2;
        int STATE = BEGIN;
        int max_length = 1; //摇摆序列最大长度至少为1
        //从第二个元素开始扫描
        for (int i = 1; i < nums.size(); i++) {
            switch(STATE) {
                case BEGIN:
                    if (1) {
                        STATE = UP;
                        max_length++;
                    }
                    else if (nums[i-1] > nums[i]) {
                        2
                        max_length++;
                    }
                    break;
            }
        }
    }
};
```

```
        case UP:
            if (3) {
                STATE = DOWN;
                4
            }
            break;
        case DOWN:
            if (5) {
                STATE = UP;
                max_length++;
            }
            break;
    }
    return max_length;
};
```

3分钟时间填写代码，有问题随时提出！

例2:实现

```
#include <vector>
class Solution {
public:
    int wiggleMaxLength(std::vector<int>& nums) {
        if (nums.size() < 2) {
            return nums.size(); //序列个数小于2时直接为摇摆序列
        }
        static const int BEGIN = 0;
        static const int UP = 1; //扫描序列时的三种状态
        static const int DOWN = 2;
        int STATE = BEGIN;
        int max_length = 1; //摇摆序列最大长度至少为1
        //从第二个元素开始扫描
        for (int i = 1; i < nums.size(); i++) {
            switch(STATE) {
                case BEGIN:
                    if (nums[i-1] < nums[i]) {
                        STATE = UP;
                        max_length++;
                    }
                    else if (nums[i-1] > nums[i]) {
                        STATE = DOWN;
                        max_length++;
                    }
                    break;
                case UP:
                    if (nums[i-1] > nums[i]) {
                        STATE = DOWN;
                        max_length++;
                    }
                    break;
                case DOWN:
                    if (nums[i-1] < nums[i]) {
                        STATE = UP;
                        max_length++;
                    }
                    break;
            }
        }
        return max_length;
    }
};
```



```
case DOWN:
    if (nums[i-1] < nums[i]) {
        STATE = UP;
        max_length++;
    }
    break;
}
```

```
return max_length;
```

```
}; }
```

例2:测试与leetcode提交结果

```
int main() {  
    std::vector<int> nums;  
    nums.push_back(1);  
    nums.push_back(17);  
    nums.push_back(5);  
    nums.push_back(10);  
    nums.push_back(13);  
    nums.push_back(15);  
    nums.push_back(10);  
    nums.push_back(5);  
    nums.push_back(16);  
    nums.push_back(8);  
    Solution solve;  
    printf("%d\n", solve.wiggleMaxLength(nums));  
    return 0;  
}
```

Wiggle Subsequence

Submission Details

24 / 24 test cases passed.

Status: **Accepted**

Runtime: 0 ms

Submitted: 6 minutes ago

?
请按任意键继续. . .

例3:移除K个数字

已知一个使用**字符串**表示的**非负整数num**，将num中的**k个数字**移除，求移除k个数字后，可以获得的**最小的**可能的新数字。

输入：num = “1432219”，k = 3

在**去掉3个数字**后得到的很多很多可能里，如1432、4322、2219、**1219**、1229...；
去掉数字4、3、2得到的1219最小！

选自 **LeetCode 402. Remove K Digits**

<https://leetcode.com/problems/remove-k-digits/description/>

难度:**Medium**

例3:题目提示与要求

```
#include <string>
```

```
class Solution {  
public:
```

字符串表示的整数

```
    std::string removeKdigits (std::string num, int k) {  
    }  
};
```

返回的结果，用字符串表示

题目提示:

- 1.输入的num, 字符串长度 ≤ 10002 , 并且字符串长度 $\geq k$
- 2.输入的num字符串不会以任何数量的0字符开头

例3:思考与分析

假设 $\text{num} = 1432219$; $k = 1$,
分别去掉1, 4, 3, 2, 2, 1, 9 得到数字:

1) ~~1~~432219 -> 432219

2) ~~1~~432219 -> **132219** **最优解**

3) ~~1~~432219 -> 142219

4) ~~1~~432219 -> 143219

5) ~~1~~432219 -> 143219

6) ~~1~~432219 -> 143229

7) ~~1~~432219 -> 143221

分析与常识:

若去掉某一位数字, 为了使得到的新数字最小, 需要**尽可能**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3位**最小...

例如:

一个4位数“1***”, 一定比任何“8***”、“8***”、...、“2***”**小!**

一个4位数若**最高位确定**, 如“51***”一定比任何“59***”、“58***”、...、“52***”**小!**

一个4位数若**最高、次高位确定**, 如“531***”一定比任何“539***”、“538***”、...、“532***”**小!**

例3:贪心规律

对于最高位"1":

若去掉: ~~1~~432219 -> 4*****

若保留: 1 $\boxed{432219}$ -> 1*****

则去掉方框中的某个数字

由于 $4***** > 1*****$, 则应该保留最高位"1"

对于次高位"4":

若去掉: ~~14~~32219 -> 13*****

若保留: 14 $\boxed{32219}$ -> 14*****

则去掉方框中的某个数字

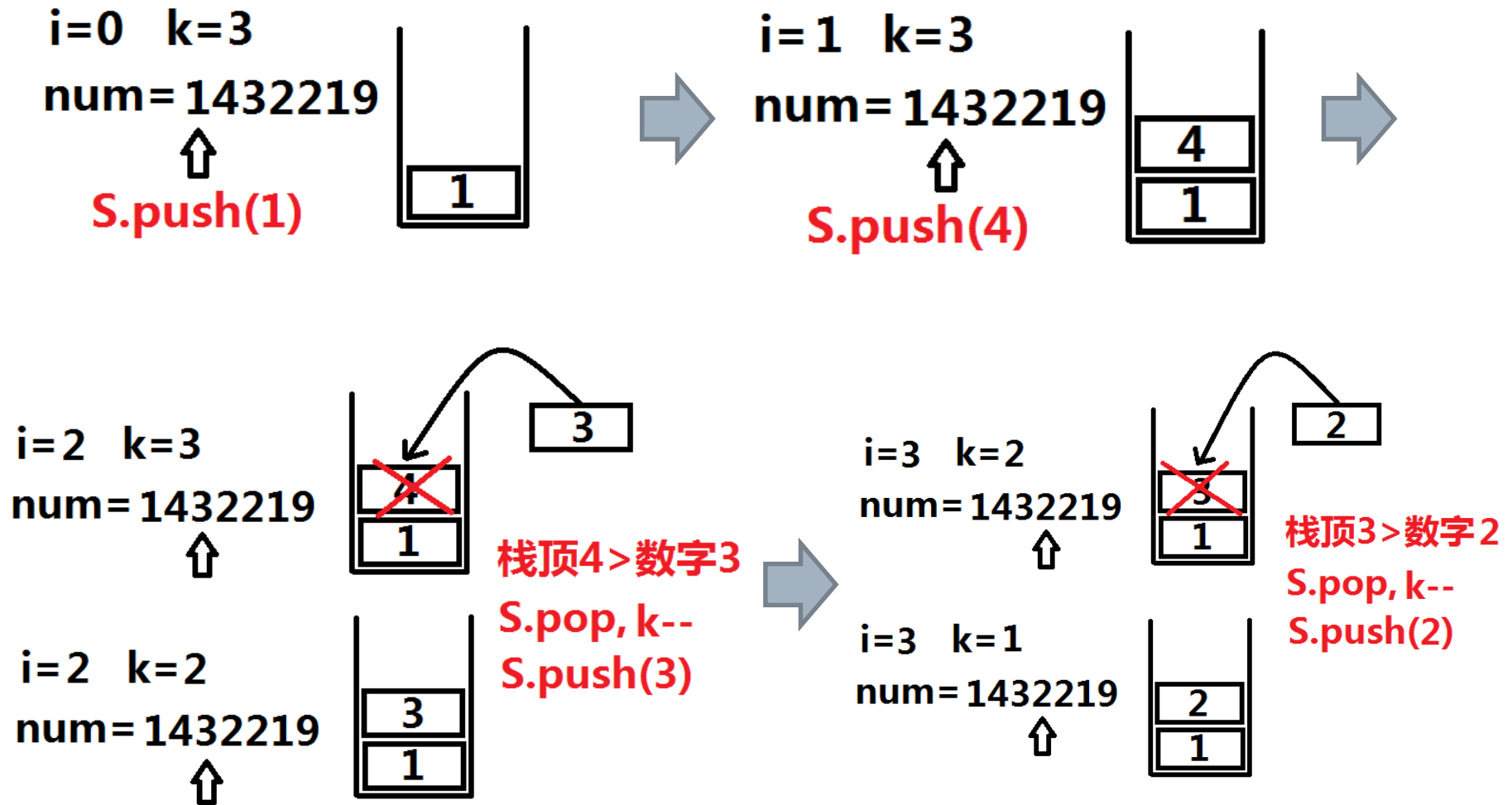
由于 $13***** < 14*****$, 则应去掉次高位"4"

从高位向低位遍历, 如果对应的数字大于下一位数字, 则把该位数字去掉, 得到的数字最小!

最暴力的方法:

去掉k个数字, 即从最高位遍历k次。

例3:使用栈来优化，存储结果(1)

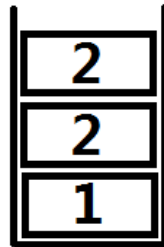


例3:使用栈来优化，存储结果(2)

i=4 k=1

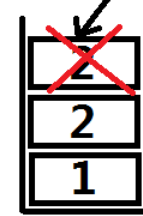
num=1432219

S.push(2) ↑



i=5 k=1

num=1432219



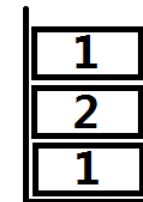
栈顶2>数字1

S.pop, k--

S.push(1)

i=5 k=0

num=1432219



i=6 k=0

num=1432219

S.push(9) ↑



栈底遍历至栈顶，即为最终结果“1219”

例3:思考如下问题

1.当所有数字都扫描完成后，**k仍然>0**，应该做怎样的处理？
例如: $\text{num} = 12345$ ， $k = 3$ 时。

2.当数字中**有0出现**时，应该有怎样的特殊处理？
例如: $\text{num} = 100200$ ， $k = 1$ 时。

3.如何将最后结果存储为**字符串**并返回？

例3:实现，课堂练习

```
class Solution {
public:
    std::string removeKdigits(std::string num, int k) {
        std::vector<int> S; //使用vector当作栈(因为vector可以遍历)
        std::string result = ""; //存储最终结果的字符串
        for (int i = 0; i < num.length(); i++) { //从最高位循环扫描数字num
            int number = num[i] - '0'; //将字符型的num转化为整数使用
            while(S.size() != 0 && 1 && k > 0) {
                //弹出栈顶元素 S.pop_back(); //当栈不空，栈顶元素大于数number，仍然可以删除数字时while循环继续
                2
            }
            if ( 3 ) {
                S.push_back(number); //将数字number压入栈中
            }
        }
        while(S.size() != 0 && k > 0) { //如果栈不空，且仍然可以删除数字
            4
            k--; //删除数字后更新K
        }
        //将栈中的元素从头遍历，存储至result
        for (int i = 0; i < S.size(); i++) {
            5
        }
        if (result == "") {
            result = "0"; //如果result为空，result即为0
        }
        return result;
    }
};
```

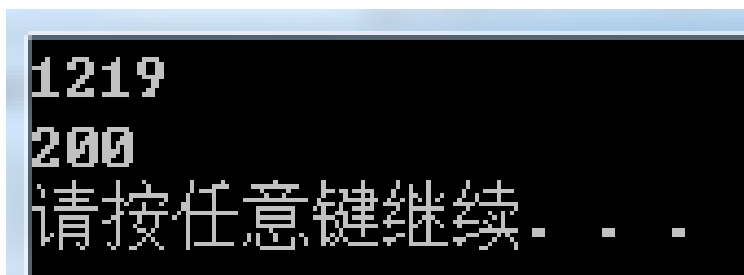
5分钟时间填写代码，
有问题随时提出！

例3:实现

```
class Solution {
public:
    std::string removeKdigits(std::string num, int k) {
        std::vector<int> S; //使用vector当作占(因为vector可以遍历)
        std::string result = ""; //存储最终结果的字符串
        for (int i = 0; i < num.length(); i++) { //从最高位循环扫描数字num
            int number = num[i] - '0'; //将字符型的num转化为整数使用
            while (S.size() != 0 && 1 S[S.size()-1] > number && k > 0) {
                //弹出栈顶元素 S.pop_back(); //当栈不空, 栈顶元素大于数number, 仍然可以删
                2 k--; //除数字时while循环继续
            }
            3 if (number != 0 || S.size() != 0) {
                S.push_back(number); //将数字number压入栈中
            }
        }
        while (S.size() != 0 && k > 0) { //如果栈不空, 且仍然可以删除数字
            4 S.pop_back();
            k--; //删除数字后更新K
        }
        //将栈中的元素从头遍历, 存储至result
        for (int i = 0; i < S.size(); i++) {
            5 result.append(1, '0' + S[i]);
        }
        if (result == "") {
            result = "0"; //如果result为空, result即为0
        }
        return result;
    }
};
```

例3:测试与leetcode提交结果

```
int main() {  
    Solution solve;  
    std::string result = solve.removeKdigits("1432219", 3);  
    printf("%s\n", result.c_str());  
    std::string result2 = solve.removeKdigits("100200", 1);  
    printf("%s\n", result2.c_str());  
    return 0;  
}
```



```
1219  
200  
请按任意键继续. . .
```

[Remove K Digits](#)

Submission Details

33 / 33 test cases passed.

Runtime: 3 ms

Status: **Accepted**

Submitted: 0 minutes ago

课间休息10分钟

有问题提出！

例4-a:跳跃游戏

一个数组存储了**非负整型数据**，数组中的第*i*个元素 $a[i]$ ，代表了可以从数组第*i*个位置**最多**向前跳跃 $a[i]$ 步；已知数组**各元素**的情况下，求是否可以从数组的**第0个**位置**跳跃**到数组的**最后一个**元素的位置？

例如：

$nums = [2, 3, 1, 1, 4]$ ，可以从 $nums[0] = 2$ 跳跃至 $nums[4] = 4$ ；

$nums = [3, 2, 1, 0, 4]$ ，不可以从 $nums[0] = 3$ 跳跃至 $nums[4] = 4$ 。

```
class Solution {  
public:  
    bool canJump(std::vector<int>& nums) {  
    }  
};
```

选自 **LeetCode 55. Jump Game**

<https://leetcode.com/problems/jump-game/description/>

难度:**Medium**

例4-a:思考

从第 i 个位置，**最远**可跳 $\text{nums}[i]$ 步：

$\text{nums} = [2, 3, 1, 1, 4, \dots]$

确实最终是否可以跳至最后一个位置，**最困难**的地方在于：

无法**直观的**观察出从第0个位置开始依次向后的**跳跃方式**。

例如：

从第0位置，可以跳跃至第1位置或第2位置；

从第1位置，可以跳跃至第2、第3、第4位置；

从第2位置，可以跳跃至第3位置；

...

最初在位置0时，应该**如何选择**，跳至第1位置还是第2位置？

若选择跳至了第1个位置，**之后**又该跳跃至第2、第3、第4个位置的哪一个？

思考**半分钟**！

例4-a:分析

从第i个位置，**最远**可跳nums[i]步：

nums = [2, 3, 1, 1, 4, ...];

从第i个位置，**最远可跳至**第index[i]个位置：

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

index = [2, 4, 3, 4, 8, ...]

若从第0位置**最远**可以跳至第i个位置；

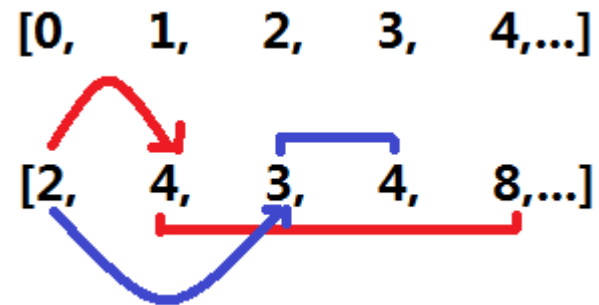
则从第0位置**也一定**可以跳至：

第1个位置、第2个位置、...、第i-1个位置。

从第0个位置，应该跳至第1、第2、...、第i-1、第i个位置中的**哪个**？

应该跳至第1、2、...、i-1、i位置中，**又可向前跳至**更最远位置

(即index[1]、index[2]、...、index[i-1]、...、index[i]最大的那个)的位置！**(贪心)**



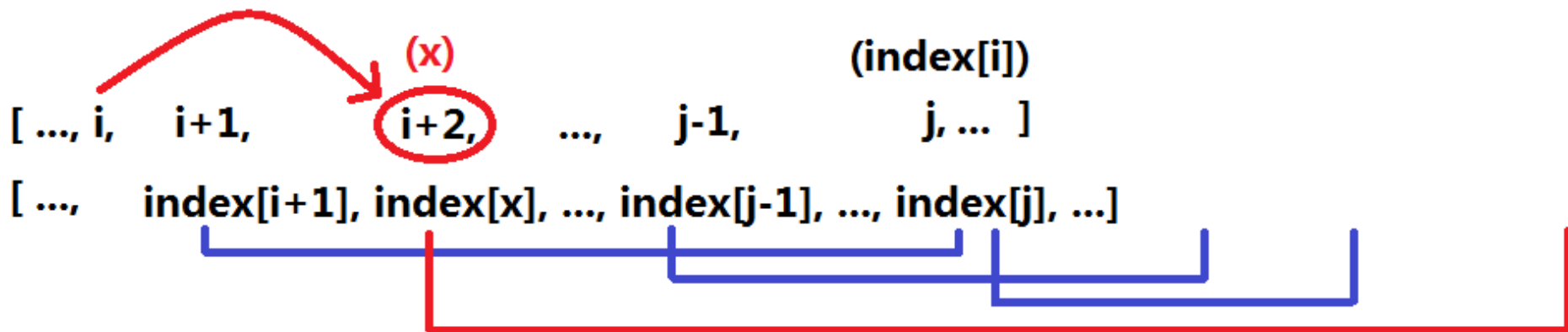
例4-a:贪心规律

若此时处在第 i 位置，该位置**最远**可以跳至第 j 位置($\text{index}[i]$)，故第 i 位置**还可跳至**：第 $i+1$ 、 $i+2$ 、 \dots 、 $j-1$ 、 j 位置；

从第 i 位应**跳至**第 $i+1$ 、 $i+2$ 、 \dots 、 $j-1$ 、 j 位中可以跳的**更远位置的位置**，即 $\text{index}[i+1]$ 、 $\text{index}[i+2]$ 、 \dots 、 $\text{index}[j-1]$ 、 $\text{index}[j]$ **最大的**那个！

原因：

假设该位置为 x ， $\text{index}[x]$ 最大，故从位置 x 出发，可以跳至 $i+1$ 、 $i+2$ 、 \dots 、 $j-1$ 、 j **所有位置可以达到的位置**；所以跳至位置 x **最理想**。



例4-a:算法思路

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: $\text{index}[i] = \text{nums}[i] + i$;

2.初始化:

1)设置变量jump代表**当前所处**的位置, 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, 最远可到达的位置**, 初始化为index[0]。

3.利用jump扫描index数组, **直到**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。

位置: [0, 1, 2, 3, 4]

最远跳跃nums: [2, 3, 1, 1, 4]

最远达到的位置index: [2, 4, 3, 4, 8]

max_index = 4

index = [2, 4, 3, 4, 8, 1, 2, 3, 4 ...]



jump = 2

例4-a: 算法思路

位置: [0, 1, 2, 3, 4, 5, ...]

最远跳跃nums: [2, 3, 1, 1, 4, 1, ...]

最远达到的位置index: [2, 4, 3, 4, 8, 6, ...]

图1: [2, 4, 3, 4, 8, 6, ...]



jump = 0

max_index = 2

图2: [2, 4, 3, 4, 8, 6, ...]



jump = 1

max_index = 4

图3: [2, 4, 3, 4, 8, 6, ...]



jump = 2

max_index = 4

图4: [2, 4, 3, 4, 8, 6, ...]



jump = 3

max_index = 4

图5: [2, 4, 3, 4, 8, 6, ...]



jump = 4

max_index = 8

图6: [2, 4, 3, 4, 8, 6, ...]



jump = 5

max_index = 8

例4-a:实现，课堂练习

```
#include <vector>
class Solution {
public:
    bool canJump(std::vector<int>& nums) {
        std::vector<int> index; //最远可跳至的位置
        for (int i = 0; i < nums.size(); i++) {
            1 //计算index数组
        }
        int jump = 0; //初始化jump与max_index
        int max_index = index[0];
        while (2) {
            if (max_index < index[jump]) {
                3
            }
            4
        }
        if (5) {
            return true;
        }
        return false;
    }
};
```

5分钟时间填写代码，
有问题随时提出！

例4-a:实现

```
#include <vector>
class Solution {
public:
    bool canJump(std::vector<int>& nums) {
        std::vector<int> index; //最远可跳至的位置
        for (int i = 0; i < nums.size(); i++) {
            index.push_back(i + nums[i]); //计算index数组
        }
        int jump = 0; //初始化jump与max_index //直到jump跳至数组尾部或
        int max_index = index[0]; //jump超越了当前可以跳的最远位置
        while (jump < index.size() && jump <= max_index) {
            if (max_index < index[jump]) {
                max_index = index[jump]; //如果当前可以跳的更远，则更新max_index
            }
            jump++; //扫描jump
        }
        if (jump == index.size()) { //若jump达到数组尾，则返回真
            return true;
        }
        return false; //否则返回假
    }
};
```


例4-a:测试与leetcode提交结果

Jump Game

Submission Details

```
int main() {  
    std::vector<int> nums;  
    nums.push_back(2);  
    nums.push_back(3);  
    nums.push_back(1);  
    nums.push_back(1);  
    nums.push_back(4);  
    Solution solve;  
    printf("%d\n", solve.canJump(nums));  
    return 0;  
}
```

75 / 75 test cases passed.

Status: **Accepted**

Runtime: 12 ms

Submitted: 0 minutes ago

```
1  
请按任意键继续. . .
```

例4-b:跳跃游戏2

一个数组存储了**非负整型数据**，数组中的第i个元素a[i]，代表了可以从数组第i个位置**最多**向前跳跃a[i]步；已知数组**各元素**的情况下，**确认**可以从第0位置跳跃到数组最后一个位置，求**最少**需要**跳跃**几次？

例如：

nums = [2, 3, 1, 1, 4]，从第0位置跳到第1位置，从第1位置跳至最后一个位置。

```
class Solution {  
public:  
    int jump(std::vector<int>& nums) {  
    }  
};
```

选自 **LeetCode 45. Jump Game II**

<https://leetcode.com/problems/jump-game-ii/description/>

难度:**Hard**

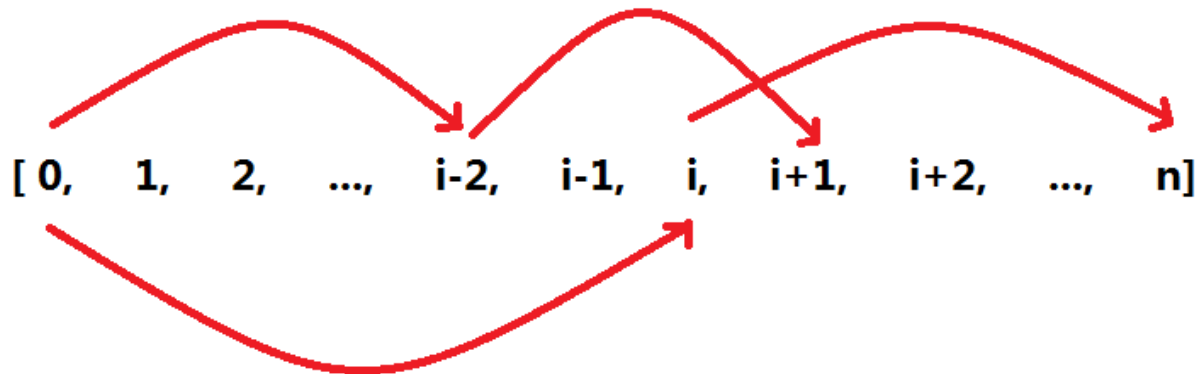
例4-b:思考

从第0位置，最少需要跳几次达到最后一个位置？

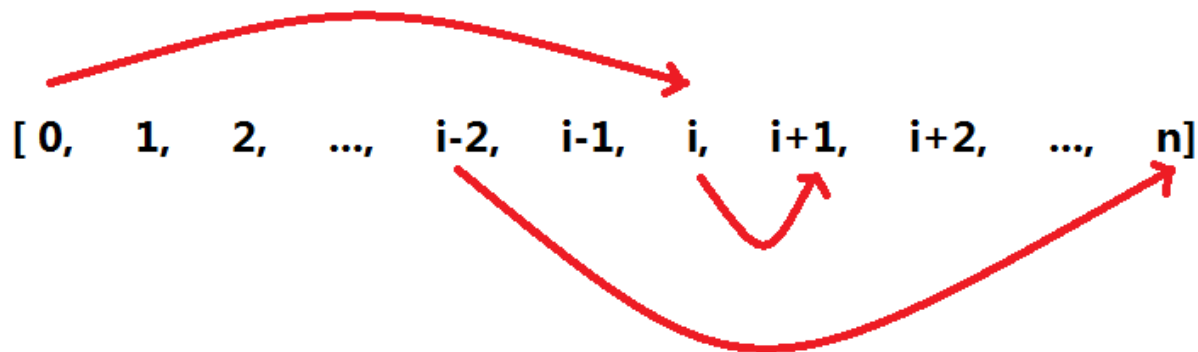
如果希望最少跳跃达到终点，则需要明确何时进行跳跃是最合适的。

例如，在到达位置*i*前：

如果提早跳跃，则可能增加跳跃次数：



如果在位置*i*前都未跳跃，则可能无法到达位置*i*后的地方：



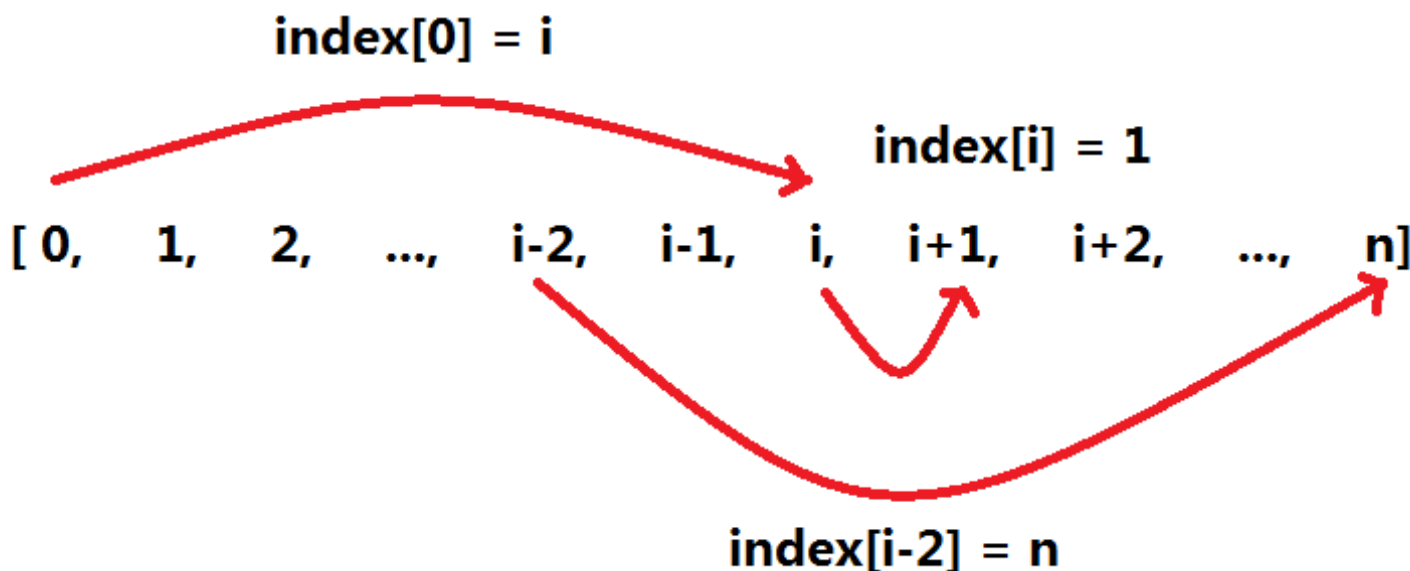
思考**半分钟**！

例4-b:贪心规律

在**到达某点前**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！

结论:在**无法到达更远**的地方时，在这之前**应该跳到**一个可以到达更远位置的位置！

$\text{index}[i-2]$ 是 $\text{index}[0,1,\dots,i]$ 中最大的



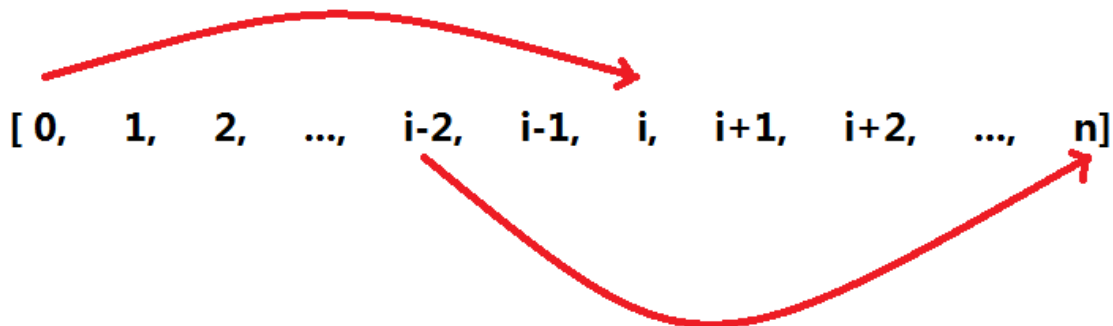
例4-b:算法思路

1. 设置 `current_max_index` 为 **当前可达到的最远位置**;
2. 设置 `pre_max_max_index` 为在 **遍历** 各个位置的过程中, 各个位置 **可达到的最远位置**;
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_max_index`
5. 遍历过程中, 若 `nums[i] + i` (`index[i]`) **更大**, 则更新 `pre_max_max_index` = `nums[i] + i`。

`current_max_index = nums[0]`

`pre_max_max_index = nums[i-2] + i - 2`

`jump_min = 1`



`current_max_index = nums[i-2] + i - 2`

`pre_max_max_index` = 无所谓

`jump_min = 2`

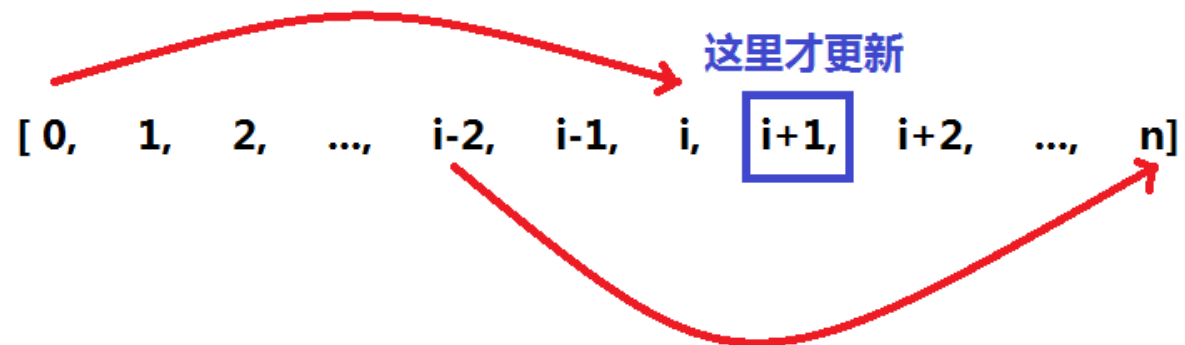
例4-b:实现， 课堂练习

```
#include <vector>
class Solution {
public:
    int jump(std::vector<int>& nums) {
        if (nums.size() < 2) {
            return 0; //如果数组长度小于2，则说明不用跳跃，返回0
        }
        int current_max_index = nums[0]; //当前可达到的最远位置
        int pre_max_max_index = nums[0]; //遍历各个位置过程中，可达到的最远位置
        int jump_min = 1;
        for (int i = 1; i < nums.size(); i++) {
            if (1) { //当??时，需要更新??
                jump_min++;
                2
            }
            if (pre_max_max_index < nums[i] + i) {
                3 //更新pre_max_max_index
            }
        }
        return jump_min;
    }
};
```

3分钟时间填写代码，
有问题随时提出！

例4-b:实现

```
#include <vector>
class Solution {
public:
    int jump(std::vector<int>& nums) {
        if (nums.size() < 2) {
            return 0; //如果数组长度小于2，则说明不用跳跃，返回0
        }
        int current_max_index = nums[0]; //当前可达到的最远位置
        int pre_max_max_index = nums[0]; //遍历各个位置过程中，可达到的最远位置
        int jump_min = 1;
        for (int i = 1; i < nums.size(); i++) {
            if (i > current_max_index) { //若无法再向前移动移动了，才进行跳跃
                jump_min++; //即更新当前可达到的最远位置
                current_max_index = pre_max_max_index;
            }
            if (pre_max_max_index < nums[i] + i) {
                pre_max_max_index = nums[i] + i; //更新pre_max_max_index
            }
        }
        return jump_min;
    }
};
```



例4-b:测试与leetcode提交结果

Jump Game II

Submission Details

92 / 92 test cases passed.

Status: **Accepted**

Runtime: 15 ms

Submitted: 0 minutes ago

```
int main() {  
    std::vector<int> nums;  
    nums.push_back(2);  
    nums.push_back(3);  
    nums.push_back(1);  
    nums.push_back(1);  
    nums.push_back(4);  
    Solution solve;  
    printf("%d\n", solve.jump(nums));  
    return 0;  
}
```

2

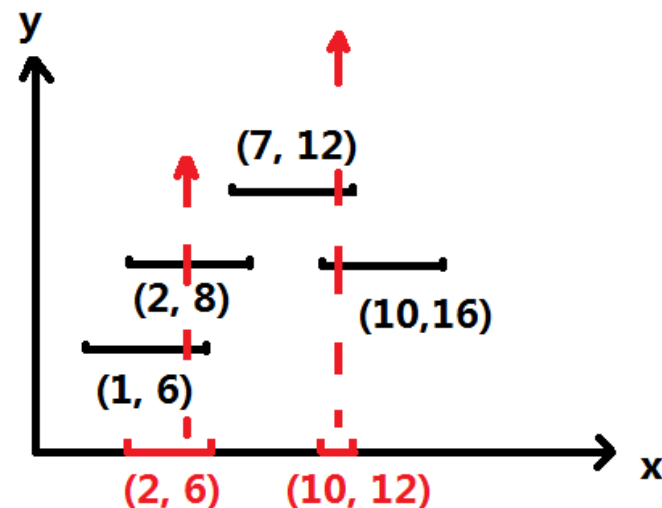
请按任意键继续. . .

例5:射击气球

已知在一个平面上有**一定数量**的气球，平面可以看作一个**坐标系**，在平面的**x轴**的不同位置安排弓箭手向**y轴方向**射箭，弓箭可以向y轴走**无穷远**；给定气球的宽度 $x_{start} \leq x \leq x_{end}$ ，问**至少**需要多少弓箭手，将**全部**气球打爆？

例如: 四个气球: $[[10,16], [2,8], [1,6], [7,12]]$ ，至少需要2个弓箭手。

```
class Solution {  
public:  
    int findMinArrowShots(  
        std::vector<std::pair<int, int> >& points) {  
    }  
};
```

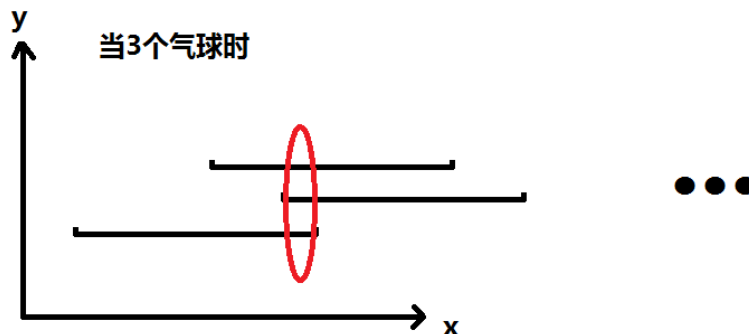
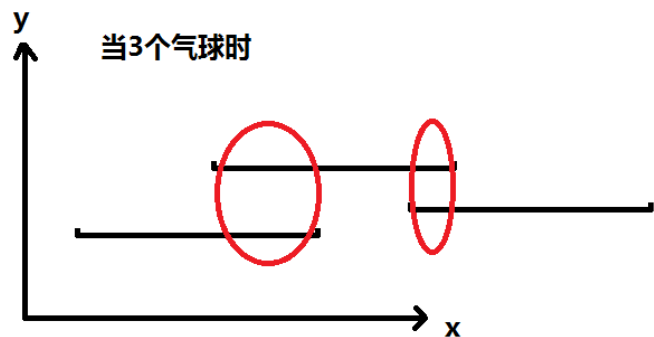
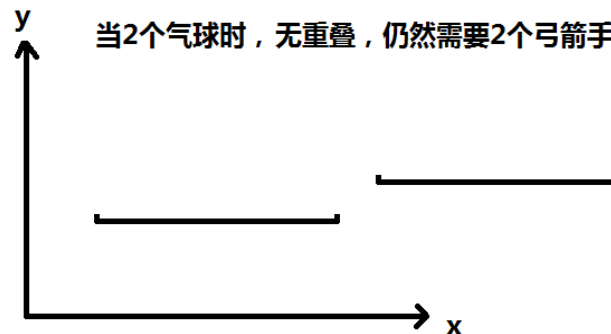
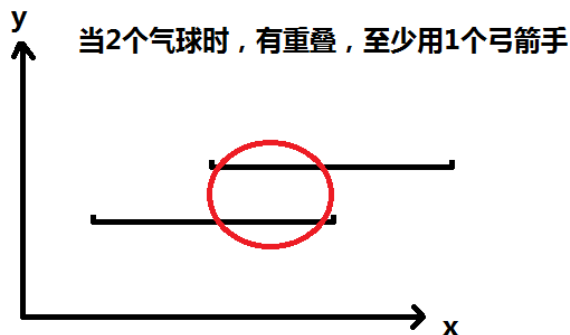
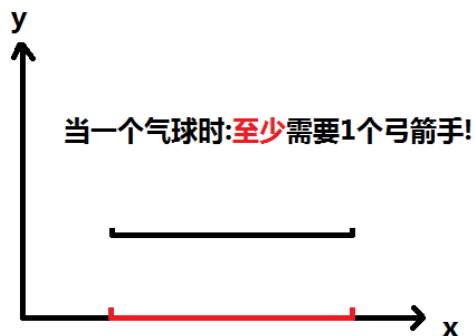


选自 **LeetCode 452. Minimum Number of Arrows to Burst Balloons**

<https://leetcode.com/problems/minimum-number-of-arrows-to-burst-balloons/description/>

难度:**Medium**

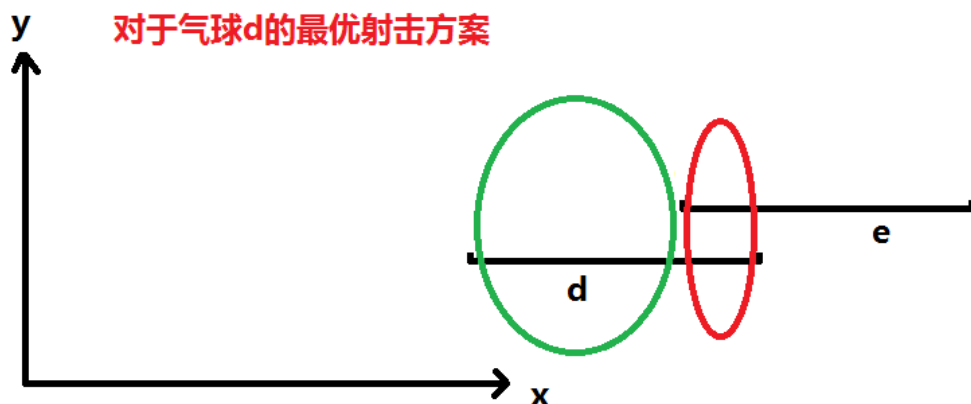
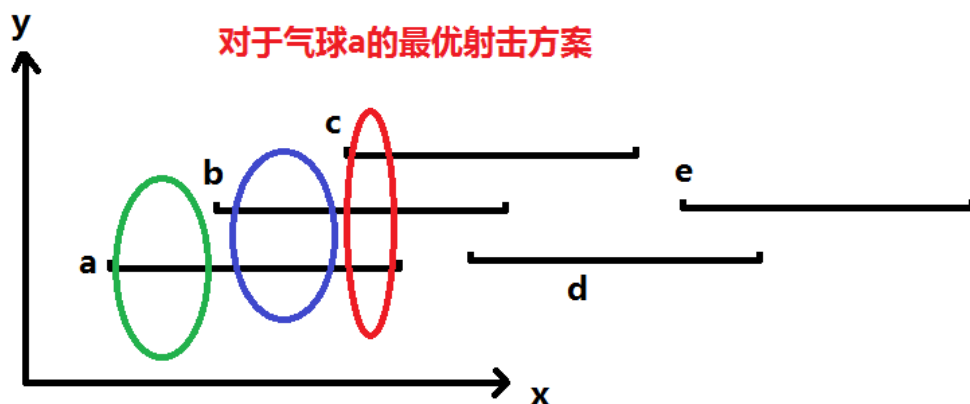
例5:思考



所以,我们应该用怎样的策略**安排弓箭手呢?**
思考**半分钟!**

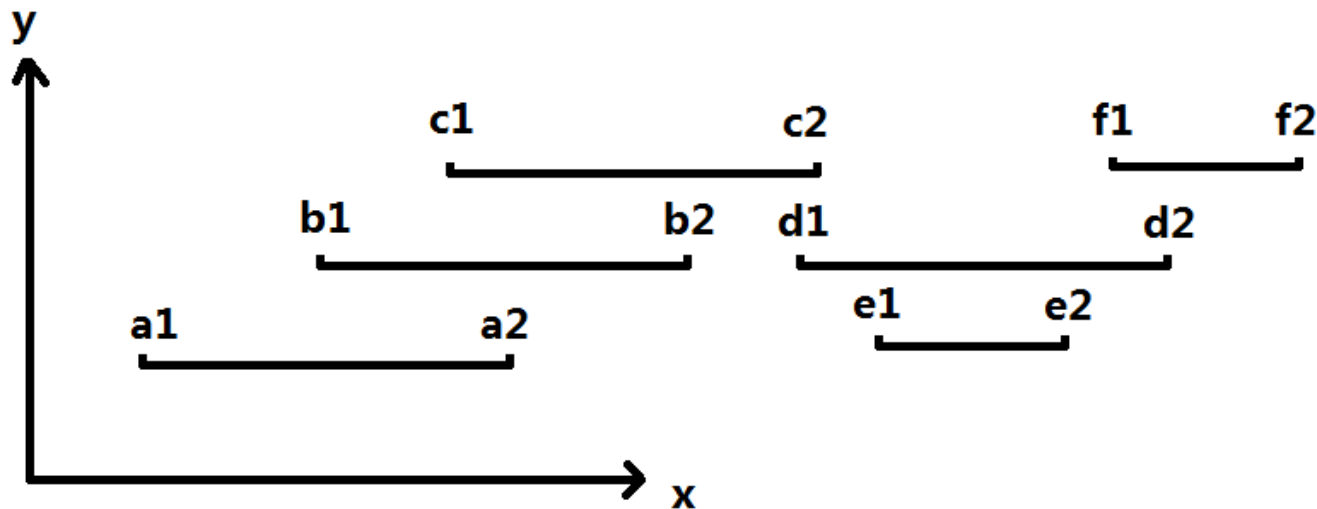
例5:贪心规律

- 1.对于某个气球，**至少需要**使用**1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！**(贪心!)**



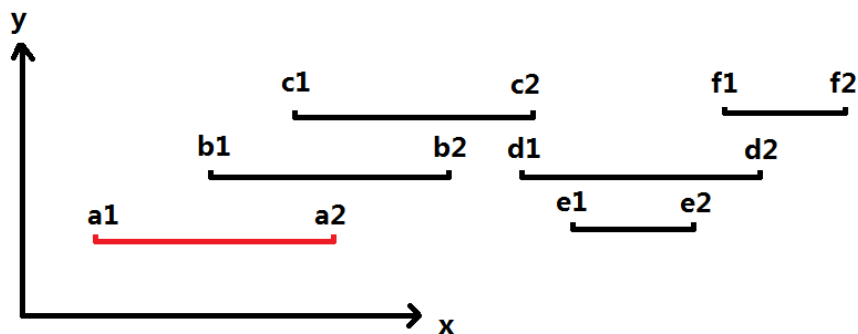
例5:算法思路

- 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(保证射击区间可以将新气球也击穿)。
- 3.如果新的气球没办法被击穿了，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)，随后继续遍历气球数组。

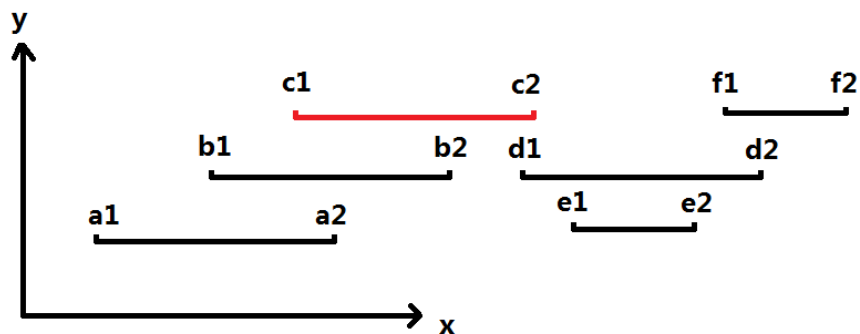


例5:算法思路

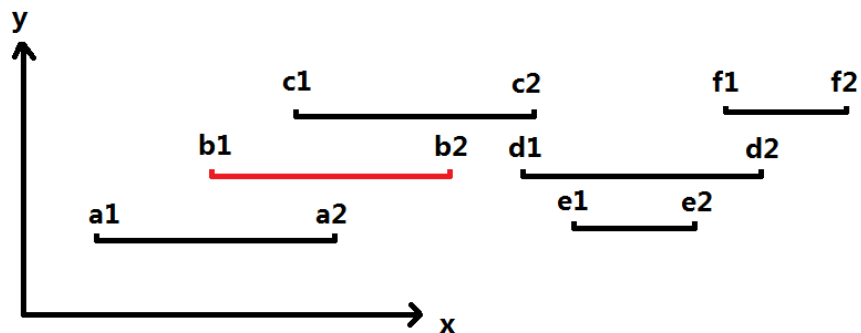
$i=0$ shoot_num = 1 射击区间 $[a1, a2]$



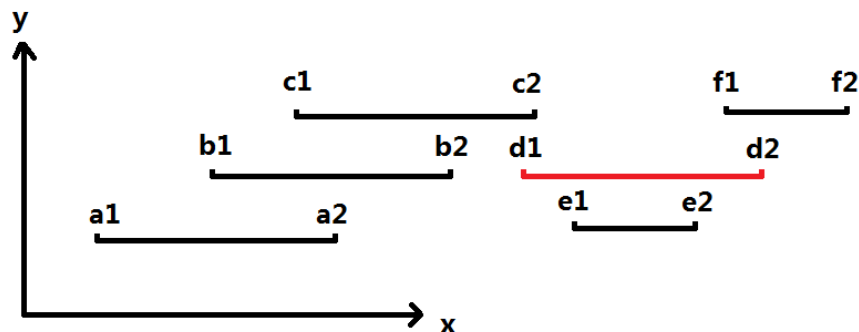
$i=2$ shoot_num = 1 射击区间 $[c1, a2]$



$i=1$ shoot_num = 1 射击区间 $[b1, a2]$

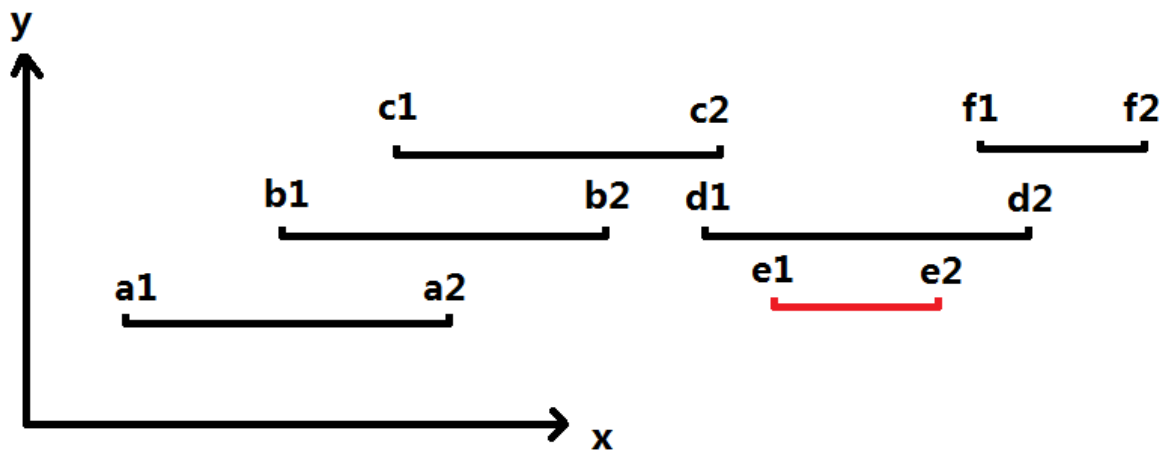


$i=3$ shoot_num = 2 射击区间 $[d1, d2]$

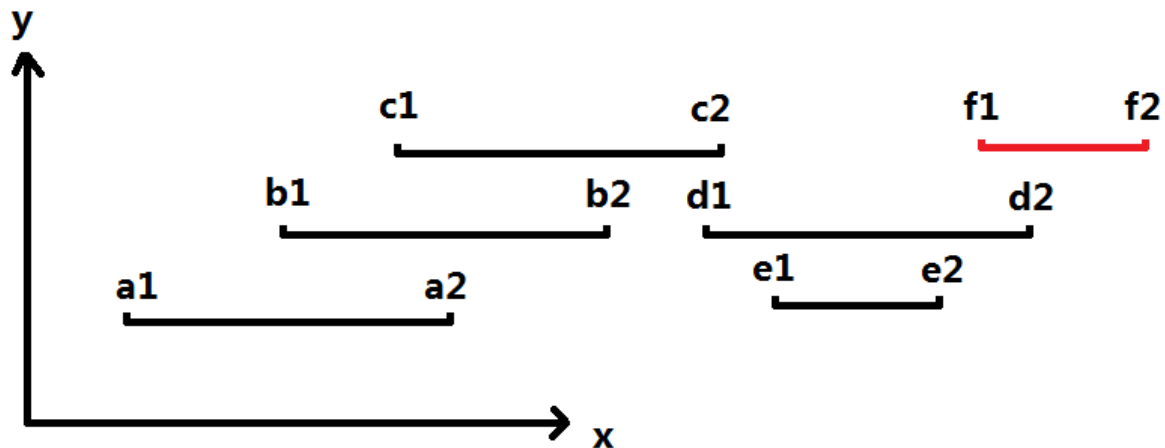


例5:算法思路

$i = 4$ $\text{shoot_num} = 2$ 射击区间 $[e1, e2]$



$i = 5$ $\text{shoot_num} = 3$ 射击区间 $[f1, f2]$



例5:实现, 课堂练习

```
#include <algorithm>
#include <vector>
```

```
bool cmp(const std::pair<int, int> &a, const std::pair<int, int> &b) {
    return a.first < b.first; //无需考虑左端点相同时的排序
}
```

```
class Solution {
public:
```

```
int findMinArrowShots(std::vector<std::pair<int, int> >& points) {
    if (points.size() == 0) { //传入的数据可能为空, 直接返回0
        return 0;
    }
```

```
    std::sort(points.begin(), points.end(), cmp); //对气球按照左端点从小到大排序
```

```
    int shoot_num = 1; //初始化弓箭手数量为1
```

```
    int shoot_begin = points[0].first;
```

```
    int shoot_end = points[0].second; //初始化射击区间, 即为第一个气球的两端点
```

```
    for (int i = 1; i < points.size(); i++) {
```

```
        if ( 1 ) {
```

//当?情况下, 更新当前的射击区间

```
            shoot_begin = points[i].first;
```

//更新的射击区间左端点即为新气球左端点

```
            if (shoot_end > points[i].second) {
```

```
                2
```

//当射击区间右端点大于新气球右端点时

```
            }
```

```
        }
```

```
        else {
```

//再保证当前气球被射穿的前提下, 射击区间不能再更

```
            3
```

新了, 需要增加一个新的射击区间了

```
            shoot_begin = points[i].first;
```

```
            shoot_end = points[i].second;
```

```
        }
```

```
    }
```

```
    return shoot_num;
```

```
}
```

```
};
```

3分钟时间填写代码, **有**
问题随时提出!

例5:实现

```
#include <algorithm>
#include <vector>

bool cmp(const std::pair<int, int> &a, const std::pair<int, int> &b) {
    return a.first < b.first; //无需考虑左端点相同时的排序
}

class Solution {
public:
    int findMinArrowShots(std::vector<std::pair<int, int> >& points) {
        if (points.size() == 0) { //传入的数据可能为空，直接返回0
            return 0;
        }
        std::sort(points.begin(), points.end(), cmp); //对气球按照左端点从小到大排序

        int shoot_num = 1; //初始化弓箭手数量为1
        int shoot_begin = points[0].first; //初始化射击区间，即为第一个气球的两端点
        int shoot_end = points[0].second;

        for (int i = 1; i < points.size(); i++) {
            if (points[i].first <= shoot_end) {
                shoot_begin = points[i].first;
                if (shoot_end > points[i].second) {
                    shoot_end = points[i].second;
                }
            }
            else {
                shoot_num++;
                shoot_begin = points[i].first;
                shoot_end = points[i].second;
            }
        }
        return shoot_num;
    }
};
```

Diagram illustrating the greedy algorithm for the minimum number of arrows to burst balloons:

- The first interval is defined by `shoot_begin` and `shoot_end`.
- The second interval is defined by `point[i].first` and `point[i].second`.
- Red arrows indicate the logic: if the first point of the second interval is less than or equal to the current `shoot_end`, then the second interval is contained within the current one, and we update `shoot_end` to the maximum of the two.
- If not, a new arrow is needed, and both `shoot_begin` and `shoot_end` are updated to the values of the new interval.

例5:测试与leetcode提交结果

Minimum Number of Arrows to Burst Balloons

Submission Details

43 / 43 test cases passed.

Status: **Accepted**

Runtime: 85 ms

Submitted: 0 minutes ago

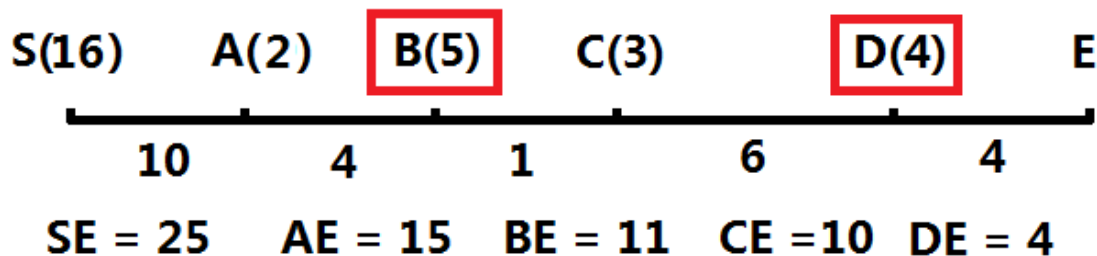
```
int main() {  
    std::vector<std::pair<int, int> > points;  
    points.push_back(std::make_pair(10, 16));  
    points.push_back(std::make_pair(2, 8));  
    points.push_back(std::make_pair(1, 6));  
    points.push_back(std::make_pair(7, 12));  
    Solution solve;  
    printf("%d\n", solve.findMinArrowShots(points));  
    return 0;  
}
```

2
请按任意键继续. . .

例6:最优加油方法

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**，起点位置至终点的**距离L**与起始时刻油箱中**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)

```
int get_minimum_stop(int L, int P, //L为起点到终点的距离, P为起点初始的汽油量
    std::vector<std::pair<int, int> > &stop) { // pair<加油站至终点的距离, 加油站汽油量>
```



选自 **poj 2431 Expedition**

<http://poj.org/problem?id=2431>

难度:**Hard**

4个加油站A, B, C, D, 从S到E**至少**加2次油

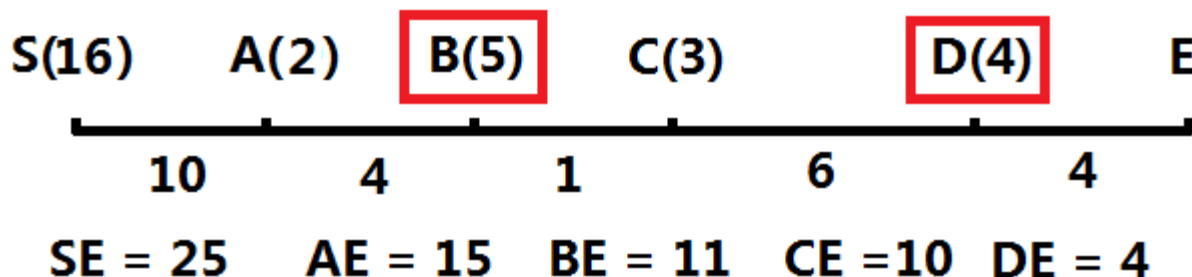
例6:思考

汽车经过n个加油站，对于这n个加油站，应该在**哪个**加油站**停下**来加油，最终既能**到达终点**，又使得加油**次数最少**？

若**按顺序遍历**加油站，则面临：

如果在某个加油站**停下**来加油，**可能**是**没必要**的，有可能不进行这次加油**也能**到达终点；

如果在某个加油站**不停下**来加油，**可能**由于汽油不够而**无法到达**终点或者后面要**更多次**的加油才能到达终点。



思考**半分钟**。

4个加油站A, B, C, D，从S到E**至少**加2次油

例6:贪心规律

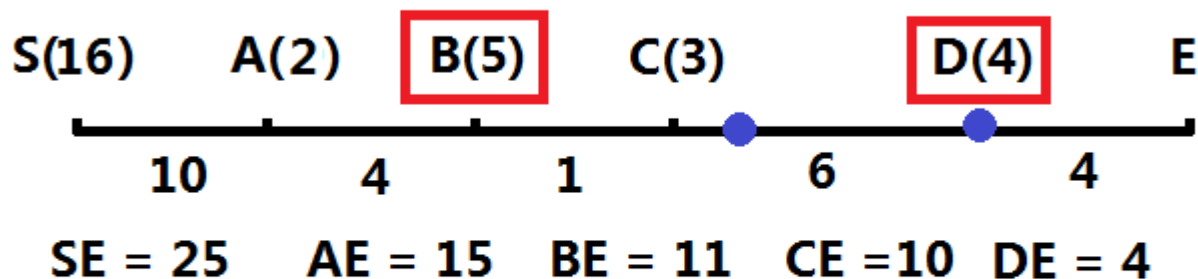
何时加油最合适?

油用光的时候加油最合适!

在哪个加油站加油最合适?

在油量最多的加油站加油最合适!

在蓝色的位置，汽油会被用光



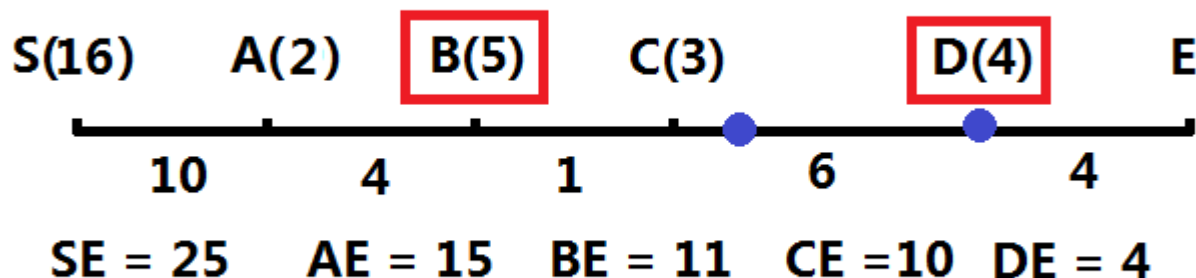
在红色的位置加油最合适，既能到达终点，加油的次数又少

4个加油站A, B, C, D，从S到E至少加2次油

例6: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**各个加油站之间与加油站到终点距离。
3. 每次**需要**走两个加油站之间的距离 d ，如果发现汽油不够走距离 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 。
4. 如果把最大堆的汽油都添加**仍然不够**行进距离 d ，则**无法达到**终点。
5. 当前油量**减少** d 。
6. 将当前加油站油量添加至**最大堆**。

在蓝色的位置，汽油会被用光



在红色的位置加油最合适，既能到达终点，加油的次数又少

例6:算法思路

图1:

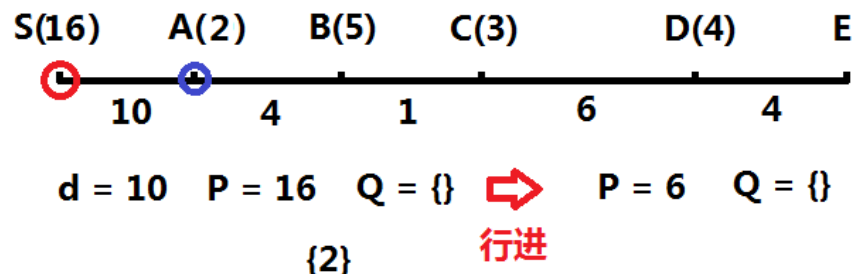


图2:

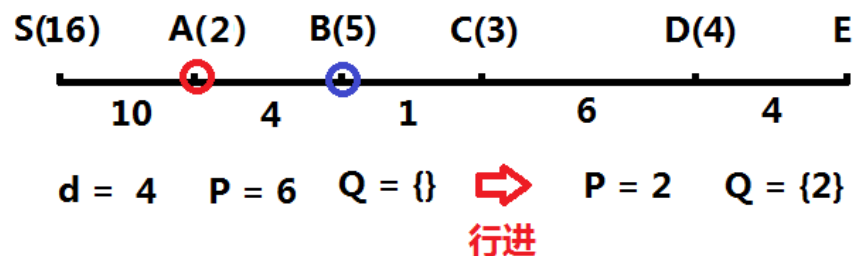


图3:

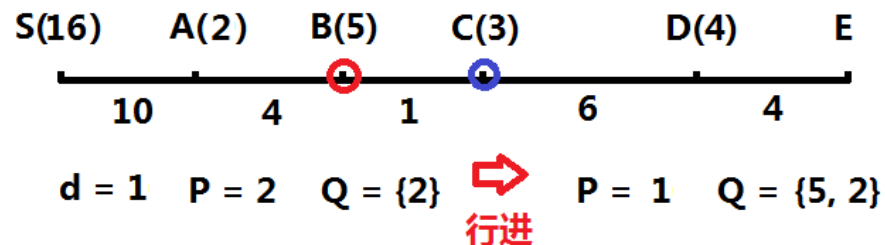


图4:

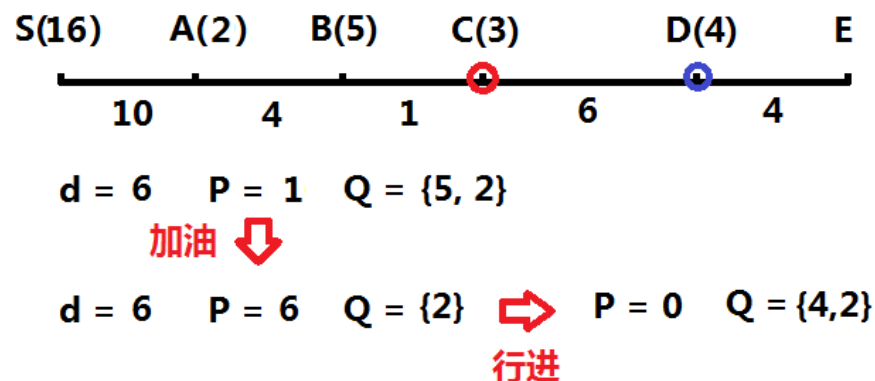
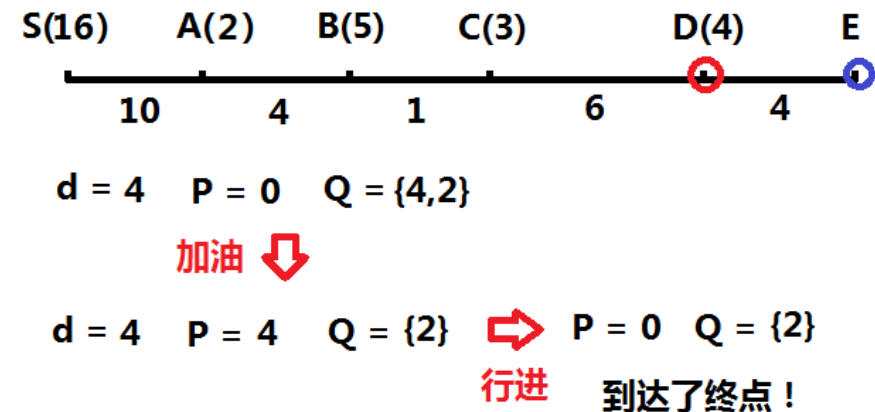


图5:



例6:实现，课堂练习

```
#include <vector>
#include <algorithm>
#include <queue>

bool cmp(const std::pair<int, int> &a, const std::pair<int, int> &b) {
    return a.first > b.first;
}

int get_minimum_stop(int L, int P, //L为起点到终点的距离, P为起点初始的汽油量
    std::vector<std::pair<int, int> > &stop) { // pair<加油站至终点的距离, 加油站汽油量>

    std::priority_queue<int> Q; //存储油量的最大堆
    int result = 0; //记录加过几次油的变量
    stop.push_back(std::make_pair(0, 0)); //将重点作为一个停靠点, 添加至stop数组
    std::sort(stop.begin(), stop.end(), cmp); //以停靠点至终点距离 从大到小 进行排序
    for (int i = 0; i < stop.size(); i++) { //遍历各个停靠点
        int dis = L - stop[i].first;
        while (1) { //当前要走的距离即为当前距终点距离
            // L 减去 下一个停靠点至终点距离
            2
            Q.pop();
            3
        }
        if (4) {
            return -1;
        }
        5
        L = stop[i].first; //更新L为当前停靠点至终点距离
        Q.push(stop[i].second); //将当前停靠点的汽油量添加至最大堆
    }
    return result;
}
```

例6:实现

```
#include <vector>
#include <algorithm>
#include <queue>
```

```
bool cmp(const std::pair<int, int> &a, const std::pair<int, int> &b) {
    return a.first > b.first;
}
```

```
int get_minimum_stop(int L, int P, //L为起点到终点的距离, P为起点初始的汽油量
    std::vector<std::pair<int, int> > &stop) { // pair<加油站至终点的距离, 加油站汽油量>
```

```
    std::priority_queue<int> Q; //存储油量的最大堆
```

```
    int result = 0; //记录加过几次油的变量
```

```
    stop.push_back(std::make_pair(0, 0)); //将重点作为一个停靠点, 添加至stop数组
```

```
    std::sort(stop.begin(), stop.end(), cmp); //以停靠点至终点距离 从大到小 进行排序
```

```
    for (int i = 0; i < stop.size(); i++) { //遍历各个停靠点
```

```
        int dis = L - stop[i].first;
```

```
        while (!Q.empty() && P < dis) {
```

//当前要走的距离即为当前距终点距离

L 减去 下一个停靠点至终点距离

```
            P += Q.top();
```

```
            Q.pop();
```

```
            result++;
```

```
        } if (Q.empty() && P < dis) {
```

```
            return -1;
```

```
        }
```

```
            P = P - dis;
```

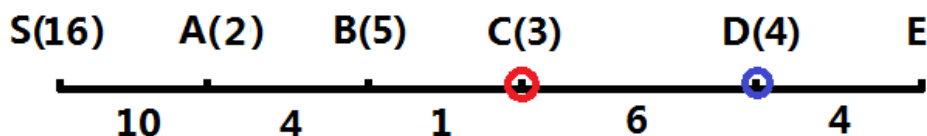
```
            L = stop[i].first;
```

```
            Q.push(stop[i].second); //更新L为当前停靠点至终点距离
```

//将当前停靠点的汽油量添加至最大堆

```
    }
    return result;
```

```
}
```



d = 6 P = 1 Q = {5, 2}

加油 ↓

d = 6 P = 6 Q = {2}



行进

P = 0 Q = {4, 2}

例6:poj测试与提交

Problem	Result	Memory	Time	Language	Code Length
2431	Accepted	412K	47MS	C++	999B

```
int main() {
    std::vector<std::pair<int, int> > stop;
    int N;
    int L;
    int P;
    int distance;
    int fuel;
    scanf("%d", &N);
    for (int i = 0; i < N; i++) {
        scanf("%d %d", &distance, &fuel);
        stop.push_back(std::make_pair(distance, fuel));
    }
    scanf("%d %d", &L, &P);
    printf("%d\n", get_minimum_stop(L, P, stop));
    return 0;
}
```

Sample Input

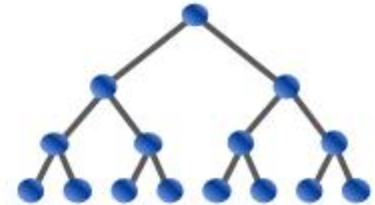
```
4
4 4
5 2
11 5
15 10
25 10
```

Sample Output

```
2
```

USACO介绍

USA Computing Olympiad

[OVERVIEW](#)[TRAINING](#)[CONTESTS](#)[HISTORY](#)[STAFF](#)[RESOURCES](#)

TRAINING PAGES NOW SUPPORT PYTHON

For those who prefer coding in Python, our training pages now support submission of programs written in Python 2.7.

2017 USA TEAM ANNOUNCED



2016-2017 SCHEDULE

Dec 16-19: First Contest
Jan 13-16: Second Contest
Feb 10-13: Third Contest
Mar 10-13: US Open
May 25-Jun 3: Training Camp
Jul 28-Aug 4: IOI 2017

<http://www.usaco.org/>

结束

非常感谢大家！

林沐