计算机算法设计与分析

081203M04001H

Chap 4 课程作业

2022 年 11 月 16 号

Professor: 卜东波

学生: 周胤昌
学号: 202228018670052
学院: 网络空间安全学院
所属专业: 网络空间安全
方向: 安全协议理论与技术

Problem 1

给定字符串 S, 由 "x" (x 代表坑) 和 "." (. 代表正常路) 组成. 填连续 k 个坑的成本时 $k+1$, 用预算 M 去尽可能的填坑, 请求出填完坑后正常路的最大长度 (可以非连续).

Solution: 该题是**20220806 微软笔试题 T1**, 其**贪心思路: 计算连续坑的大小, 从大到小进行贪心**. 原因为: 填 k 个坑的额外成本为 1, 连续填的坑越多, 则摊还成本越低 (即 $1/k$). 因此要**优先把最长的连续坑给填好, 直到预算用尽为止**. 算法可以描述为:

- 记录所有连续的坑长度并从大到小排序;
- 从大到小填坑, 先判断剩下的预算能否把当前这个长度为 k 的连续坑给填好, 又分为两种情况:
  - 钱够 (即 $M > k+1$), 则 $\text{ans} \leftarrow \text{ans} + k$ 且 $M \leftarrow M - (k+1)$, 继续循环;
  - 钱不够 (即 $M \le k+1$), 则 $\text{ans} \leftarrow \text{ans} + (M-1)$(直接把预算给用完), 并跳出循环;
- 最后加上本来就是正常路的个数 (即原本字符串 S 中 "." 的个数) 即可得到最终答案.

C++ 语言代码描述如下, 在 main 函数中的测试样例均已通过:

```cpp
int NormalRoad(string S, int M) { // S 是道路状况, M 为预算
    int res = 0; // 初始化正常道路的长度
    int L = S.size();
    vector<int> cnt; // 存储连续坑的长度
    for(int i = 0; i < L; i++) {
        if(S[i] == '.') {
            res++; // 碰到正常位置, 则正常道路长度自增一
        }
        else {
            int tmp = 0; // 用以记录当前连续坑的长度
            while(i < L && S[i] == 'x') {
                tmp++, i++; // 用 while 循环来寻找当前连续坑的末尾
            }
            cnt.push_back(tmp); // 记录该连续坑的长度
            i--; // while 循环结束时需要往回退一位, 否则就会跳过一个待遍历的位置从而导致漏解
        }
    }
    auto cmp = [](int a, int b){
        return a > b; // 从大到小排序也可直接:sort(cnt.begin(), cnt.end(), greater<int>())
    };
    sort(cnt.begin(), cnt.end(), cmp);
    for(auto len : cnt) { // 从大到小遍历坑长度
        if(M > len + 1) { // 预算足够填当前所在的连续坑 (长度)
            res += len; // 修复成正常道路, 则正常道路长度加上 len
            M -= (len + 1); // 扣去相应的填坑预算 (即 len + 1)
        }
        else { // 预算不够填当前所在的连续坑 (长度)
            res += (M - 1); // 预算只够修复长度为 M-1 的连续坑 (即当前连续坑的一个连续子列)
            break; // 预算用完了, 退出循环, 不用再遍历修坑了
        }
    }
    return res;
}
```

测试代码如下:

```cpp
#include <bits/stdc++.h> //万能头, 刷题可以用, 大型项目不要用
using namespace std;
int main() {
    vector<string> S = {
        "xxx...xxxx....xxxxx", "...xxx...x...xxx.", "...xxxxx", "x.x.xxx....x", "."
    };
    vector<int> M = {11, 7, 4, 14, 5};
    for(int i = 0; i < 5; i++) {
        cout << " 第" << i + 1 <<" 个样例的最大正常道路长度为" << NormalRoad(S[i], M[i]) <<
 endl;
    }
}
```

代码输出为:

```
开始运行...
第 1 个样例的最大正常道路长度为 16
第 2 个样例的最大正常道路长度为 15
第 3 个样例的最大正常道路长度为 6
第 4 个样例的最大正常道路长度为 12
第 5 个样例的最大正常道路长度为 1
运行结束.
```

- 对于第 1 个测试样例: 理论结果应为 7(原本的正常路) + 9(填坑形成的正常路) = 16;
- 对于第 2 个测试样例: 理论结果应为 10(原本的正常路) + 5(填坑形成的正常路) = 15;
- 对于第 3 个测试样例: 理论结果应为 3(原本的正常路) + 3(填坑形成的正常路) = 6;
- 对于第 4 个测试样例: 理论结果应为 6(原本的正常路) + 6(填坑形成的正常路) = 12;
- 对于第 5 个测试样例: 理论结果应为 1(原本的正常路) + 0(填坑形成的正常路) = 1.

显然上述 5 个测试样例都通过了, 再来分析一下算法的复杂度: 算法第一部分的 for 循环和 while 循环, 本质上就是对字符串 S 作一次遍历, 即消耗 $O(L)$ 的时间. 而最坏情况下, 连续坑的数目最多为 $\lfloor L/2 \rfloor$(即 S 形如"x.x.x.x. $\cdots$ x."), 也就是说 cnt 数组的长度最长为 $\lfloor L/2 \rfloor$. 于是排序需要消耗 $O\left(\frac{L}{2}\log\frac{L}{2}\right)$ 的时间. 算法最后对 cnt 数组做一次遍历, 循环内部操作消耗常数时间, 于是最坏情况下算法的最后部分消耗 $O\left(L/2\right)$ 的时间. 综合起来, 最坏情形下算法的总时间复杂度为

$$T\left(L\right) = O\left(L\right) + O\left(\frac{L}{2}\log\frac{L}{2}\right) + O\left(L/2\right) = O\left(L\log L\right)$$

算法需要 cnt 这个辅助数组, 其长度最长为 $L/2$, 其空间消耗为 $O(L/2)$, 而 (归并) 排序最多也需要消耗 $O(L/2)$ 的栈空间. 于是最坏情形下, 算法的总空间复杂度为

$$S\left(L\right) = O\left(\frac{L}{2}\right) + O\left(\frac{L}{2}\right) = O\left(L\right)$$

其中 L 为字符串 S 的长度.

Problem 2

nums 是由非负整数组成的数组, 请重新排列并拼接他们使得他们能够形成最大的整数.

Solution: 题目的本质其实就是让我们做一个排序, 使得拼接后的数字最大化. 那么我们就需要确定排序规则 (即贪心规则). 贪心策略 (**根据结果**来决定的排序规则) 为: 对于 nums 中的任意两个数 a, b, 如果拼接结果 $a||b$(|| 表示拼接的意思) 要比 $b||a$ 要大, 那么认为应该将 a 放在 b 前面. 接下来我们需要证明该贪心策略的正确性:

将经过上述贪心策略得到的贪心解记作 ans, 真实最优解记为 max. 而 ans 作为可行解, 显然有 ans $\leq$ max 的. 所以接下来只需证明 ans $\geq$ max, 我们采用反证法, 假设 ans $<$ max. 由于 ans 和 max 都是由同一批数组元素组成的, 于是 ans 中必然至少有一对数字 (比如 x 和 y, x 在 y 前面) 互换可以使得 ans 变大, 那么根据排序逻辑可知: x 所在的整体 (一个数, 可能不只有 x 一个数) 应该排在 y 所在的整体 (也是一个数, 可能不只有 y 一个数) 后面. 这就与我们是通过**根据结果的排序规则**所得到 ans 的过程是相矛盾的! 因此假设是错误的, 即 ans $\geq$ max, 即 ans $=$ max. 因此我们的贪心策略是正确的! 另外值得说明的细节是: 上述排序比较逻辑作用在 nums 上应具有**全序关系**, 具体证明详见**宫水三叶の相信科学系列**, 此处就不赘述了. 于是我们可以写出如下的 C++ 代码, 并已完全通过**LeetCode-T179**的所有测试样例:

```
1  class Solution {
2  public:
3      string largestNumber(vector<int>& nums) {
4          int n = nums.size();
5          vector<string> temp;
6          for(auto num : nums) {
7              temp.push_back(to_string(num));
8          }
9          auto cmp = [](string A, string B){
10             return A + B > B + A;
11         };
12         sort(temp.begin(), temp.end(), cmp);
13         string res;
14         for(auto x : temp) {
15             res += x;
16         }
17         return res[0] == '0' ? "0" : res; //若前导 res[0]=0, 那么后面一定都是 0, 即结果为 0
18     }
19 };
```

注意: 如果要排列拼接得到**最小的整数**, 那么只需要将排序规则 (贪心策略) 变更为: 若 $a||b$ 比 $b||a$ 更小, 则 a 排在 b 前面. 剩下的分析思路完全一致, 并且在上述 C++ 代码中的第 10 行改为 return $A + B < B + A$ 即可.

再来分析一下复杂度: 显然排序需要消耗 $O(n\log n)$ 的时间, 中间数组 temp 需要消耗 $O(n)$ 的空间, 其中 n 为 nums 的长度. 所以时间复杂度 $T(n) = O(n\log n)$, 空间复杂度 $S(n) = O(n)$.

Problem 3

一共有 n 个会议, 第 i 个会议的起始时间和结束时间分别为 s_i 和 e_i. 请设计一个贪心算法来找到能够安排所有会议的最少会议室数量.

Solution: 要求出安排所有会议所需的最少会议室数目, 即**只需求出同一时刻进行的会议数量的最大值**. 于是我们可以将问题转化为**上下车问题** (都是为了占用某种资源).

比如 interval $= [[0, 30], [5, 10], [15, 20]]$, 第一个人从 0 上车且 30 下车、第二个人从 5 上车且 10 下车. 问题就是: **在某个时刻, 车上最多能有多少人?(即最少需要多少个会议室)** 显然, 上车则人数自增 1; 下车则人数自减 1. 我们把 intervals 拆解一下 (目的在于避免引起某个时间点的状态混淆 (即上下车)), 例如

$$\text{上车} : [0, 1] , [5, 1] , [15, 1] , \quad \text{下车} : [10, -1] , [20, -1] , [30, -1]$$

然后按照时间从小到大进行排序 (关于排序规则更详细的说明请阅读下面的代码注释), 最后遍历排序后的 meetings 以迭代式更新求得车上人数的最大值.

算法具体的 C++ 代码如下所示, 并已完全通过**LeetCode-T253**的所有测试样例:

```cpp
class Solution {
public:
    int minMeetingRooms(vector<vector<int>>& intervals) {
        vector<vector<int>> meetings;
        for(auto interval : intervals) {
            meetings.push_back({interval[0], 1}); // 拆分 intervals 是为了
            meetings.push_back({interval[1], -1}); // 在某个时间点避免引起状态 (上下车) 混淆
        }
        auto cmp = [](vector<int> A, vector<int> B){
            if(A[0] == B[0]) {
                return A[1] < B[1]; // 碰到同一时刻有人上车且有人下车时需先下后上
                // 即如果时间相同, 结束会议排在开始会议前面, 也就是先结束这场会议再开始新的一场
            }
            else {
                return A[0] < B[0]; // 按时间从小到大排序
            }
        };
        sort(meetings.begin(), meetings.end(), cmp); // 为了方便后续按时间先后顺序进行扫描
        int cnt = 0, res = 0; // 用以对某时刻的车上人数进行统计
        for(auto meeting : meetings) {
            cnt += meeting[1];
            res = max(res, cnt); // 迭代式更新求得车上人数的最大值
        }
        return res;
    }
};
```

再来分析一下算法的复杂度: 算法中的主体就是排序, 耗时 $O(n\log n)$, 且算法需要辅助二维数组 meetings, 因此消耗 $O(n)$ 的空间. 因此, 算法的时空复杂度分别为 $T(n) = O(n\log n)$ 和 $S(n) = O(n)$, 其中 n 为会议的总数目.

Problem 4

N 是一个非负整数, 从其上移掉 k 位数字来获取新的数字, 请设计贪心算法来找到可能的最大值和最小值.

Solution: 对于两个长度相同的数字序列, **最左边不同的数字就足以决定这两个数字的大小关系!** 比如 $A = 1a$xxx, $B = 1b$xxx, 如果 $a > b$, 那么 $A > B$ (即贪心正确性的理论依据). 因此, 我们若想使得剩下的数字最小, 那么贪心规则就是**保证靠前的数字尽可能地小 (即剩下的数字序列尽可能地是一个递增序列)**. 于是我们需要构造一个单调 (递增) 栈来维护当前的答案序列, 并考虑从左向右地遍历数字序列来构造最后的答案. 具体思路如下:

- 从左到右遍历所有数字, 判断是否需要用栈来保留;
- 不断地去比较当前遍历元素与栈顶元素比较: 若栈顶元素大, 那么栈顶元素需要被弹出;
  - 如果删除 (弹出) 次数 k 用光了, 那么提前终止遍历;
  - 如果遍历完成但是删除次数 k 没用完, 于是我们需要删除末尾的元素来把次数用光;
  - 但是上述两点的代码实现起来略显复杂. 因为"删除"和"保留"都是互补操作, 因此我们在遍历完成时 (**即整个数字序列都经过了单调递增栈的过滤**) 只需要截取 (保留) 前 $n-k$ 个元素即可, 这样代码实现会方便一些;
- 再去除前导零, 即可得到最终结果 (如果最终序列为空, 则返回 0).

该贪心算法的 C++ 代码如下所示, 并已完全通过**LeetCode-T402**的所有测试样例:

```
1  class Solution {
2  public:
3      string removeKdigits(string num, int k) {
4          int n = num.size(), m = n - k;
5          string res;
6          for(char digit : num) {
7              //维持单调递增栈, 才能保证是最小数
8              //注意单调递增栈的入栈方法 (体现为 while 后两个循环条件)
9              while(k > 0 && res.empty() == false && res.back() > digit) {
10                 res.pop_back(); //即进行删除操作
11                 k--; //消耗一次删除操作
12             }
13             res.push_back(digit); //将 digit 入栈
14         }
15         res.resize(m); //此时删除操作可能没被用完但是结束遍历了, 这时只需截取前 n-k 个即可
16         while(res.empty() == false && res[0] == '0') {
17             res.erase(res.begin()); //抹去前导零 (但不占用删除次数)
18         }
19         return (res.empty() == true) ? "0" : res;
20     }
21 };
```

若想通过删除 k 位来得到最大值, 则我们只需要维护一个**单调递减栈**, 剩余思路与上述算法完全类似. 具体做法为: 只需在上述 C++ 代码的第 9 行中最后一个循环条件改为 res.back() < digit, 就可以得到删除 k 位后的最大值. 算法的时间复杂度: 由于每个数字最多仅会入栈出栈一次 (即 for 内部的 while 循环), 所以整个 for 循环的时间复杂度为 $O(n)$, 抹去前导零的消耗 $O(n)$ 的时间, 于是总时间复杂度为 $T(n) = O(n)$. 空间复杂度: 需要消耗 $O(n)$ 的栈空间, 因此 $S(n) = O(n)$, 其中 n 为 num 的位数.

Problem 5

给定一个非负整数组成的数组 nums, 你最初位于数组的第一个下标, 并且每个数组元素代表你在该位置可以跳跃的最大长度. 判断你是否可以到达最后一个下标.

Solution: 贪心思路是, 尽可能到达最远位置. 对于能达到的最远位置, 那么一定能到达它前面的所有位置 (**原因是可以根据能到达的最远位置和目标位置的距离, 调整各次跳跃的长度即可**). 也就是说: 如果最远位置的索引大于等于 $n-1$(n 为数组长度), 那么一定能够到达最后一个下标 (即 $n-1$); 如果最远位置的索引小于 $n-1$, 那么一定到达不了最后一个下标 (至此, 贪心的正确性证明就完毕了). **类比便可有更一般的结论: 对于当前索引 i 是否能够到达, 即只需判断当前情形下的最远位置是否** $\geq i$. 于是问题就转换为求解**能到达的最远位置**! 于是思路为: 初始化最远位置为 0, 然后依次遍历数组, 如果当前位置能到达并且当前位置 + 跳数 >(当前的) 最远位置, 就更新能到达的最远位置. 数组遍历完毕后, 即可求出最终的 (能到达的) 最远位置. 具体的 C++ 代码如下所示, 并已完全通过**LeetCode-T55**的所有测试样例:

```cpp
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int n = nums.size();
        int max_index = 0;
        for(int i = 0; i < n; i++) {
            /* 如果能够到达位置 i 且能跳得更远 */
            if(max_index >= i && i + nums[i] > max_index) {
                max_index = i + nums[i]; //那就更新最远能到达位置
            }
        }
        return max_index >= n - 1; //判断是否能到达最后一个下标
    }
};
```

显然算法的时间复杂度为 $T(n) = O(n)$, 空间复杂度为 $S(n) = O(1)$, 其中 n 为数组的长度. 其实题目还可以添加条件变成一个新题目 (**可能出成考题**): 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, 假设你总是可以到达数组的最后一个位置. 思路就是挨着跳 (具体说明请看下述代码), 对应的 C++ 代码如下所示, 并已完全通过**LeetCode-T45**的所有测试样例:

```cpp
int jump(vector<int>& nums) {
    /* 若想以最少的跳跃次数到达终点, 只需每一步都尽可能跳的最远 (即贪心规则)*/
    /* 如果所有跳跃完成以后, 导致最后越过了终点, 那么只需适当调整 */
    /* 最后一次跳跃距离即可使得最后一次跳跃正好落在终点 (即贪心正确性的简单说明)*/
    int step = 0;    // 记录最小跳跃次数
    int Rborder = 0; // 记录当前可跳范围的右边界
    int maxPos = 0;  // 记录各个遍历点所对应的最远跳跃位置
    for (int i = 0; i < nums.size() - 1; i++) {
        maxPos = max(maxPos, nums[i] + i); // 遍历数组, 迭代式更新 (当前可跳范围内) 各遍历点的最远跳跃位置
        if (i == Rborder) { // 此时才能真正确定 (当前可跳范围内) 遍历点的最远跳跃位置, 并决定跳到哪一个遍历点
            step++; // 跳跃到下一个起跳点 (即决定跳到的遍历点)
            Rborder = maxPos; // 更新下一个起跳点对应可跳范围的右边界
        }
    }
    return step;
}
```

Problem 6

给定一个非负整数 k, 你**最多**可以交换该数字中的任意两位. 请设计一个贪心算法来找到你能得到的最大值.

Solution: 贪心思路是: **将最大且最靠右的数字, 与最靠左且小于它的数交换 (如果存在)**, 就能得到最大结果. 具体的正确性如下分析: 设整数 num **从右向左**的数字分别为 $(d_0, d_1, \cdots, d_{n-1})$, 则此时有 $\text{num} = \sum\limits_{i=0}^{n-1} d_i \times 10^i$. 假设我们交换第 j 位和第 k 位上的数字 $(0 \le j < k \le n-1)$, 则交换后的值 val 如下:

$$\begin{aligned} \text{val} &= \underbrace{\sum_{i=0}^{n-1} d_i \times 10^i - d_j \times 10^j - d_k \times 10^k}_{\text{没交换的位数对应的和}} + \underbrace{d_j \times 10^k + d_k \times 10^j}_{\text{交换的位数得到的值}} \\ &= \underbrace{\sum_{i=0}^{n-1} d_i \times 10^i}_{\text{num}} + \left(d_j - d_k\right) \times \left(10^k - 10^j\right) \end{aligned}$$

因此, 要想使得 val 值相较于 num 值增长的更多, 我们需要做到:

- 最优的交换一定需要满足 $d_j > d_k$;
- 最右边越大的数字与最左边较小的数字进行交换, 这样产生的整数才能保证越大.

因此我们可以这样做: 维护一个 maxIdx, 在对数字从右向左遍历时, 如果当前遍历到的数字 (即 charArray[i]) 大于 charArray[maxIdx], 记录并更新**目前已遍历完的数字中**的最大值 (即 charArray[maxIdx]) 的索引; 如果 charArray[i] < charArray[maxIdx], 此时记录并更新当前**可以考虑交换**的数对 (i, maxIdx). 因为要得到尽可能靠左边且满足要求 (即 charArray[i] < charArray[maxIdx]) 的数对 (即可行解), 所以要对整个数字数组**遍历到底 (即贪心)** 以求得最优解. 具体实现的 C++ 代码见如下 (已完全通过**LeetCode-T670**的所有测试样例, 算法的时空复杂度显然均为 $O(n) = O(\log(\text{num}))$:

```cpp
class Solution {
public:
    int maximumSwap(int num) {
        string charArray = to_string(num);
        int n = charArray.size();
        int maxIdx = n - 1, idx1 = -1, idx2 = -1;
        for(int i = n - 1; i >= 0; i--) {
            if(charArray[i] > charArray[maxIdx]) {
                maxIdx = i; //将最靠后的数定为候选数，若它之前出现了更大的数，则更新候选数
            }
            if(charArray[i] < charArray[maxIdx]) {
                idx1 = i, idx2 = maxIdx; //更新可行解，并继续向左遍历以求得最优解
            }
        }
        if(idx1 != -1) {  //说明 idx1 更新了，即结束循环后找到了最优可行解
            swap(charArray[idx1], charArray[idx2]);
            return stoi(charArray);
        }
        else return num; //若 idx1 一直没有更新，则说明数字位从左向右单调递减，则 num 即为所求
    }
};
```

Problem 7

给定一个整数数组, 将其划分为 k 个子集, 找到一个使得所有子集的极差之和最大化的划分方案.

Solution: 由于一个集合的极差只跟他的最大值和最小值有关, 因此若想使得所有子集的极差之和最大, 我们就要使得极差项尽可能的多且极差值尽可能的大. 于是我们的贪心规则为: 将**数组从小到大进行排序**, 最前面和最后面的元素构成一个子集, 然后倒数第二个元素和正数第二个元素构成一个子集, 依此类推并形成 $k-1$ 对数 (即 $k-1$ 个子集). 最后在原数组中把前面的 $k-1$ 个子集刨掉来作为最后一个子集 (即中间那一堆数构成最后一个子集). 这样的流程下来, 我们能够保障极差项尽可能的多 (体现为一对一对的做出子集), 并且能使得各个极差的数值尽可能的大 (这是由排序来保证的), 因此能够使得所有子集的极差和最大化. 算法的 C++ 代码如下所示:

```cpp
#include <bits/stdc++.h> //万能头, 刷题可以用, 大型项目不要用
using namespace std;
int MaxRangeSum(vector<int>& nums, int k) { // 贪心: 排序 + 对撞双指针
    int n = nums.size();
    if(n == k) return 0; // 处理边界条件
    else if(n < k || k <= 0) return -1; // 输入不合法时返回-1
    int res = 0, i = 0, j = n - 1;
    sort(nums.begin(), nums.end()); // 理由在解答里已陈述
    while(k > 0 && i < j) {
        res += nums[j--] - nums[i++];
        k--, n -= 2; // 要做的子集个数自减 1, 集合元素个数自减 2
        if(k == 1) { // 此时中间那一堆数构成最后一个子集
            res += nums[j] - nums[i];
            break;
        }
        else if(n == k) break; // 此时只需要将剩下的每个元素都单独作为 1 个子集
    }
    return res;
}
int main() {
    vector<vector<int>> mat = {
        {}, {1}, {5, -1}, {6, -7, 8}, {5, -2, 15, 20}, {12, -2, 22, 13, -5, 20}
    };
    vector<int> K = {0, 1, 1, 2, 3, 4};
    for(int i = 0; i < 6; i++) {
        cout << " 第" << i + 1 <<" 个测试样例的结果为" << MaxRangeSum(mat[i], K[i]) << endl;
    }
}
```

代码输出为 (可见算法目前来讲是没有遇到 bug 的):

```
开始运行...
第 1 个测试样例的结果为 0
第 2 个测试样例的结果为 0
第 3 个测试样例的结果为 6
第 4 个测试样例的结果为 15
第 5 个测试样例的结果为 22
第 6 个测试样例的结果为 49
运行结束.
```

算法的时空复杂度即为 $T(n) = \underbrace{O(n\log n)}_{\text{排序}} + \underbrace{O(n)}_{\text{while 循环}} = O(n\log n)$, $S(n) = \underbrace{O(n)}_{\text{排序}}$.