计算机学院《算法设计与分析》第三次作业

20373363 李子涵

November 21, 2022

1 最长空位问题

给定一长度为 n 的 01 串 $S =< s_1, s_2, \cdots, s_n >$，仅有一次机会挑选其中两个元素 $s_i, s_j (1 \leq i, j \leq n)$ 并交换他们的位置。请设计算法求出交换之后的 S 中最多有几个连续的 0。

例如，串 $S =$ “10010101” 通过交换 s_4 和 s_7 可以变为 “10000111”，连续的 0 的数量为 4。

请先简要描述策略，然后写出伪代码，最后分析时间复杂度。

1.1 策略

有效的交换操作是将被 1 个 1 分割的两个全零串的 1 交换成 0。为了满足交换的有效性，需要考虑边界情况: 是否能找到多余的 0 来和 1 进行交换。如果存在两个全零串之外的多余的 0，则此时最大长度为两个零串的长度再加 1；如果两个被 1 个 1 分隔的全零串之外不存在多余的 0，则无法从外部找多多余 0，选取第一个串的第一个 0 和中间的 1 交换，最大长度为字符串中 0 的个数。

另外一种边界情况是，不进行交换操作时，也可能找到最长全零串作为答案。此时这种情况对应字符串中只存在一个全零串。

基于以上分析，考虑查找全零字符串的分隔符: 包括字符串的首尾以及中间出现的 1，即可找到包含 1 个 1 的两个全零串。具体算法是在字符串首尾各添加一个 1，即 $s_0 = 1, s_{n+1} = 1$，形成新字符串 S'。接着遍历字符串 S' 并记录所有 1 也就是分隔符的位置 $seprator[i]$（表示字符串 S' 中第 i 个分隔符 1 的下标是 $seprator[i]$）。

当 $seprator.length$ 大于 3 时，遍历字符串中的 1，计算 $seprator[i+1] - seprator[i-1] - 1$ 并记录最大值即为答案；此时得到的结果需要与字符串中 0 的总数比较，来判断是否出现上述的边界情况。

当 S' 中 1 的个数等于 2 时，说明原字符串 S 中全是 0，直接返回 n；

1.2 伪代码

Algorithm 1: 最长空位问题——贪心算法

```
Input: 正整数 n，为字符串 S 长度，长度为 n 的字符串 S[]
Output: 通过一次交换元素后 S 中最多连续 0 的数量
function main(n, S[]):
    // 通过在字符串S首尾分别增加分隔符1构造新字符串S'
    S' ← '1' + S + '1';
    // 初始化seprator数组，用于后续记录分隔符1的位置
    seprator[] ← [];
    seprator.length ← 0;
    // 在seprator数组记录字符串S'中分隔符1的位置
    for i ← 0 to n + 1 do
        if S'[i] = '1' then
            seprator[seprator.length] ← i;
            seprator.length ← seprator.length + 1;
        end
    end
    // 记录交换后最长全零串长度ans
    ans ← 0;
    if seprator.length  ==  2 then
        ans ← n;
    end
    else
        // 遍历计算交换后的最长长度
        for i ← 1 to  seprator.length − 2 do
            ans ← Max{ans, seprator[i + 1] − seprator[i − 1] − 1};
        end
    end
    ans = Min{n − seprator.length + 2, ans};
    return ans;
end
```

1.3 时间复杂度分析

由上述策略分析和伪代码知，遍历字符串寻找分隔符 S 时间复杂度为 $O(n)$，遍历分隔符数组 $seprator$ 时间复杂度为 $O(n)$，综上时间复杂度为 $O(n)$。

2 最大收益问题

某公司有一台机器，在每天结束时，该机器产生的收益为 X_1 元。在每天开始时，若当前剩余资金大于等于 U 元，则可以支付 U 元来升级该机器（每天最多只能升级一次）。从升级之日起，该机器每天可以多产出 X_2 元的收益。即是说，在执行 K 次升级之后，这台机器每天的产出为 $X_1 + K \times X_2$ 元。

该公司初始资金为 C 元，请你设计算法求出 n 天之后该公司拥有的总资金的最大值。

请先简要描述策略，然后写出伪代码，最后分析时间复杂度。

2.1 策略

对于第 $i(1 \le i \le n)$ 天，如果当天升级机器，需要付出成本 U 元，从第 i 天到第 n 天每天可以额外获得收益 X_2 元，因此对于第 i 天升级来说，后续从第 i 天到第 n 天产生的净收益为 $(n-i+1) \times X_2 - U$。

所以比较第 i 天升级的收益是否大于不升级的收益，只需要转换为比较 $(n-i+1) \times X_2 - U > 0$，因此可以做出决策第 i 天是否升级。

从 1 遍历到 n，对每天进行如上判断做出是否升级机器的决策并累加收益获得答案。

2.2 伪代码

Algorithm 2: 最大收益问题——贪心算法

Input: 初始资金 C 元，初始机器每天收益 X_1 元，升级机器成本 U 元，升级机器后每天额外收益 X_2 元，经营周期 n 天

Output: n 天后该公司拥有的总资金的最大值

```
function main(C, X_1, U, X_2, n):
    // 公司初始总资金为C
    total ← C;
    // 机器每天产生的初始收益为X_1
    daily ← X_1;
    // 遍历决策当天是否升级机器并记录总资金变化
    for i ← 1 to n do
        // 如果升级机器收益更多，则更新机器后续每天产生的收益，并且总资金减去升级成本
        if (n − i + 1) × X_2 − U > 0 then
            daily ← daily + X_2;
            total ← total − U;
        end
        total ← total + daily;
    end
    return total;
end
```

2.3 时间复杂度分析

由上述策略和伪代码知，遍历每一天的时间复杂度为 $O(n)$。

3 探险家分组问题

营地中共有 n 个探险家，第 i 个探险家的经验值为 $e_i(1 \le i \le n)$。现他们希望组成尽可能多的队伍前去探险。探险家组建队伍需满足如下规则：

1. 探险家可以不参加任何队伍，即留在营地；

2. 如果第 i 个探险家参加了某支队伍，那么该队伍的人数应不小于其经验值 e_i。

请设计一个尽可能高效的算法求出最多可组建几支队伍前去探险，并分析其时间复杂度。

例如有 $n=5$ 名探险家，其经验值分别为 $e=\{2,1,2,2,6\}$，则可组建 $(1,2),(2,2)$ 两支队伍，把经验值为 6 的探险家留在营地。

请先简要描述策略，然后写出伪代码，最后分析时间复杂度。

3.1 策略

由于仅要求队伍数量最多，并未要求冒险家的具体人数，所以可以使用贪心策略，将探险家经验值升序排序得到数组 $arr\_sorted\_by\_exp$；

遍历 $arr\_sorted\_by\_exp$ 开始组队，将当前遍历到的冒险家放入当前队伍。对于每一支队伍，队伍的总人数约束条件，应该总满足最晚被选进队伍的冒险家的经验值。则确定一支队伍成立，只需要当被选入队伍的冒险家的经验值等于当前队伍总人数，即可判断成立一支队伍，继续遍历按如上算法组建下一支队伍即可。

经过上述遍历，返回组成队伍数。

3.2 伪代码

Algorithm 3: 探险家分组问题——贪心算法

```
Input: 探险家数量 n，长度为 n 的数组 arr 表示探险家经验值
Output: 可以组成的最多队伍数
function main(n, arr):
    // 将arr按经验值升序归并排序获得有序数组
    arr_sorted_by_exp ← MergeSort(arr);
    // num表示当前队伍总人数
    num ← 0;
    // max表示当前队伍成立所需的总人数，被最晚选进队伍的冒险家的经验值更新
    max ← 0;
    // ans表示成立的队伍数
    ans ← 0;
    // 遍历arr_sorted_by_exp组建队伍
    for i ← 1 to n do
        // 当前冒险家加进当前队伍
        num ← num + 1;
        // 更新队伍的总人数约束，更新为当前被选进队伍的冒险家的经验值
        max ← Max{max, arr_sorted_by_exp[i]};
        // 如果已选择冒险家数等于组成队伍所需的总数，说明可以成立队伍
        if max = num then
            ans ← ans + 1;
            num, max ← 0, 0;
        end
    end
    return ans;
end
```

3.3 时间复杂度分析

对冒险家数组归并排序时间复杂度为 $O(nlogn)$，遍历 $arr\_sorted\_by\_exp$ 时间复杂度为 $O(n)$，综上时间复杂度为 $O(nlogn)$。

4 分店选址问题

某奶茶品牌想在全国投资开分店来扩大规模提高影响力，通过向新老顾客发放问卷的形式调研产生了 n 个备选地址，并实地考察到了两组数据 $flow$ 和 $cost$，其中 $flow[i]$ 表示第 i 个备选地址的人流量，

$cost[i]$ 表示在该地址开店所需的最低资金。由于当前总资金有限，该奶茶品牌希望根据这两组数据，从 n 个备选地址中挑选出 k 个组成最终分店名单，使得能够满足在以下约束条件的前提下尽可能降低总投资成本。

1. 对每个被选中的地址，应当按照其人流量与其他 $k-1$ 个被选中地址人流量的比例来投入资金；
2. 被选中的每个地址的投入资金都不得低于其所需的最低资金。

请设计一个算法求满足上述条件的前提下，该奶茶品牌需要投入的总资金的最小值。

请先简要描述策略，然后写出伪代码，最后分析时间复杂度。

4.1 策略

记单位人流量投入资金为 $w_i = \frac{cost[i]}{flow[i]}$。

分析条件一可知每个分店的单位人流量投入资金 w_i 相同，且应该为所选择的 k 个分店中单位人流量投入资金 $w_i = \frac{cost[i]}{flow[i]}$ 最大的。故所选择的 k 个分店总成本为 $ans = w_{max} \times flow_{sum}$。

因此，贪心策略先计算单位人流量投入资金 w_i, 将 n 个元组 (i, w_i) 记作数组 $flow\_sorted\_by\_w[n]$。对其按单位人流量投入资金 $flow\_sorted\_by\_w[i].w$ 升序归并排序得到数组 $flow\_sorted\_by\_w[n]$。

对于每种方案的 $w_{max} \times flow_{sum}$，$w_{max}$ 只会出现在 $flow\_sorted\_by\_w[k...n].w$ 中，故剪枝后只需要遍历 $[k...n]$ 个元素，对每个 $flow\_sorted\_by\_w[i].w$ 作为 w_{max}，只需要在 $flow\_sorted\_by\_w[1...i-1]$ 中求 $flow_{sum}$ 的最小值并相乘即可，遍历时维护乘积最小值作为最小费用结果。求 $flow\_sorted\_by\_w[1...i-1]$ 中 $flow_{sum}$ 和的最小值采用优先队列维护，可以化简到 $O(logn)$ 的插入元素和删除最大值的复杂度。

4.2 伪代码

Algorithm 4: 分店选址问题——贪心算法

Input: 备选地址数量 n，长度为 n 的人流量数组 $flow[\ ]$，长度为 n 的所需最低资金数组 $cost[\ ]$，所需分店数 k

Output: 选择 k 个分店需要投入的总资金的最小值

```
function main(n, flow[ ], cost[ ], k):
    // 将(flow[i], cost[i]/flow[i])升序排序获得有序数组 flow_sorted_by_w，数组元素记作(flow, w)
    flow_sorted_by_w ← MergeSort(flow[i], cost[i]/flow[i]);
    // 初始化分店方案总人流量
    flow_sum ← 0;
    // 将前k − 1个单位人流量投入资金最小的分店加入优先队列pq以初始化分店方案
    pq = new PriorityQueue();
    for i ← 1 to k − 1 do
        // 将分店人流量加入优先队列
        pq.offer(flow_sorted_by_w[i].flow);
        // 更新当前总人流量
        flow_sum ← flow_sum + flow_sorted_by_w[i].flow;
    end
    ans ← +∞;
    // 从第k个分店开始遍历flow_sorted_by_w，以其作为最大单位人流量投入资金
    for i ← k to n do
        // 获取当前分店单位人流量投入资金
        w_max ← flow_sorted_by_w[i].w;
        // 将当前分店加入方案并更新总人流量
        pq.offer(flow_sorted_by_w[i].flow);
        flow_sum ← flow_sum + flow_sorted_by_w[i].flow;
        // 计算当前方案投资成本并更新答案
        ans ← Min{ans, flow_sum × w_max};
        // 移除当前分店方案中人流量最大的店铺以维持当前分店方案中为前i个分店中人流量最小的
        flow_max ← pq.poll();
        flow_sum ← flow_sum − flow_max;
    end
    return ans;
end
```

4.3 时间复杂度分析

对分店进行按单位人流量投入资金归并排序的时间复杂度为 $O(nlogn)$,遍历数组 $flow\_sorted\_by\_w$ 时间复杂度为 $O(n)$，维护优先队列时间复杂度为 $O(logn)$。综上所述，本题时间复杂度为 $O(nlogn)$ 综上本题时间复杂度为 $O(nlogn)$。

5 交通建设问题

现有 n 个城市，初始时任意两个城市之间均不可互。现有两种交通建设方案：

1. 花费 $c_{i,j}$ 的代价，在城市 i 和城市 j 之间建设一条道路，可使这两个城市互相可达。
2. 花费 a_i 的代价，在城市 i 建设一个机场，可以使得其与其他所有建设了机场的城市互相可达。

只要两个城市 i 和 j 之间存在一条可达的路径，则这两个城市也互相可达。请设计一个尽可能高效的算法求出所有城市之间互相可达所需的最小花费。

请先简要描述策略，然后写出伪代码，最后分析时间复杂度。

5.1 策略

分析城市互相可达的两种策略：

1. 全图由道路相连成一张连通图。最小花费对应的连通图是一棵最小生成树。

2. 全图分为若干个由道路相连的连通图，每张连通图是一棵最小生成树。每个连通图之间由两连通图的机场相连，保证所有连通图之间可达，且最小花费对应每个连通图有且仅有一个建设机场成本最低的城市。

以下分析贪心算法：

1. 对于上述第一种策略，考虑不建设机场时的最低成本，直接使用 Prim 算法求解全图最小生成树即可。

2. 对于上述二种策略，考虑建若干个机场的情况，可以设一个额外的城市作为机场 $airport$，从而把点 (城市机场费用) 具有的权重转化为边权，即令城市 i 到附加城市 $airport$ 的距离为 a_i，即可使用 Prim 算法求解最小生成树，得到建设机场时的最低成本。

比较两种策略的最低成本，以最小值作为全局最低成本。

5.2 伪代码

Algorithm 5: 交通建设问题——贪心算法

Input: 城市数量 n，大小为 $n \times n$ 的二维数组 $c[\ ][\ ]$ 表示两个城市之间建设道路成本，长度为 n 的数组 $a[\ ]$ 表示建设机场成本

Output: 使所有城市之间互相可达所需的最小花费

function $main(n, c[\ ][\ ], a[\ ])$**:**

  // 初始化城市点集和道路花费的边集合

  $V \leftarrow \{1 \cdots n\}$;

  $E \leftarrow \{< u, v, w > |u, v, \in V, u \neq v, w = c[u][v]\}$;

  // 使用 Prim 最小生成树求解不建设机场时的全图连通图边集

  $T \leftarrow Prim(V, E)$;

  // 计算不建设机场时最小生成树所得边集的最小建设成本

  $cost\_without\_airport \leftarrow \sum_{t \in T} c[t.u][t.v]$;

  // 点集追加额外城市的机场结点$airport$

  $airport \leftarrow n + 1$;

  $V' \leftarrow V \cup \{airport\}$;

  // 将各个城市机场费用转化为与$airport$的边权a_i加入边集

  $E' \leftarrow E \cup \{< u, airport, w > |u \in V, w = a[u]\}$;

  // 使用 Prim 最小生成树求解建设机场时的全图连通图边集

  $T' \leftarrow Prim(V', E')$;

  // 计算建设机场时的最小建设成本

  $cost\_with\_airport \leftarrow \sum_{t \in T' c[t.u][t.v]}$;

  // 初始化答案变量

  $ans \leftarrow 0$;

  // 比较获得最低成本

  $ans \leftarrow Min\{cost\_without\_airport, cost\_with\_airport\}$;

  return ans;

end

5.3 时间复杂度分析

由上述策略和伪代码分析，两次 $Prim$ 算法求解最小生成树，点的规模是 n，边的规模是 n^2，因此总时间复杂度为 $O(n^2)$。