

Assignment Three- Programming

1951976 李林飞

2021 年 6 月 12 日

目录

1 Problem 1	2
1.1 Question	2
1.2 算法思路	2
1.3 核心代码	2
1.4 效率分析	4
1.5 运行结果	4
2 Problem 2	5
2.1 Question	5
2.2 算法思路	5
2.3 核心代码	5
2.3.1 Interval 结构体	5
2.3.2 Interval 输入处理	5
2.3.3 主问题算法	7
2.3.4 主函数	8
2.4 效率分析	8
2.5 运行结果	8

1 Problem 1

1.1 Question

Given an integer n , return the count of all numbers with unique digits x , where $0 \leq n < 10$ and $0 \leq x < 10^n$. Please give a Dynamic Programming solution.

1.2 算法思路

根据组合数学知识，先考虑特定位数时满足条件的个数。其核心思想：从最高位开始，由于最高位不能为 0，故从是 9 个数 $\{1,2,3,4,5,6,7,8,9\}$ 中选一个数字放在最高位，有 9 种选法，次高位上可以为 0，再去了最高位的数，还有 9 种选法，再下一位则只有 8 种选法，依次类推..... 设 $f(k)$ 为 k 位数中满足条件的个数，则有：

$$f(1) = 10$$

$$f(2) = 9 * 9 = 81$$

...

$$f(k) = 9 * 9 * 8 * \dots * (9 - k + 2)$$

进一步可得递推关系：

$$f(k) = \begin{cases} 10, k = 1 \\ 81, k = 2 \\ f(k-1) * (9 - k + 2), k > 2 \end{cases}$$

所以该题的最终结果为：

$$CountUniqueDigits(n) = \begin{cases} 1, n = 0 \\ f(1) + f(2) + \dots + f(n) \end{cases}$$

1.3 核心代码

方法 1：动态规划

首先可以使用备忘录 (dp 数组) 存储 $f(k)$ 的结果，然后迭代求解，如下：

```

1 // 动态规划
2 int CountUniqueDigits_dpTable(int n)
3 {
4     /*备忘录*/
5     vector<int> memo(n+1,9);
6     /*边界条件*/
7     memo[0] = 1;
8     /*填充备忘录*/
9     for(int i = 2 ; i <= n; i++)
10    {
11        for(int j = 9; j >= 9-i+2; j--)
12        {
13            memo[i] *= j;
14        }
15    }
16
17    int res = 0;
18    for (int i = 0; i <=n; i++)
19    {
20        res += memo[i];
21    }
22
23    return res;
24 }

```

方法 2: 动态规划 + 状态压缩

又因为 $f(k)$ 只与前一项有关，所以可以用“状态压缩”来减小空间复杂度。代码如下：

```

1 // 动态规划+状态压缩
2 int CountUniqueDigits_dp(int n)
3 {
4     /*边界条件*/
5     if (n == 0)
6         return 1;
7     if (n == 1)
8         return 10;
9
10    /*计算*/
11    int res = 10;
12    int tmp = 9; /*第i位数的情况*/
13    for(int i = 2; i <= n; i++)
14    {
15        tmp = tmp * (9-i+2);
16        res += tmp;
17    }
18    return res;

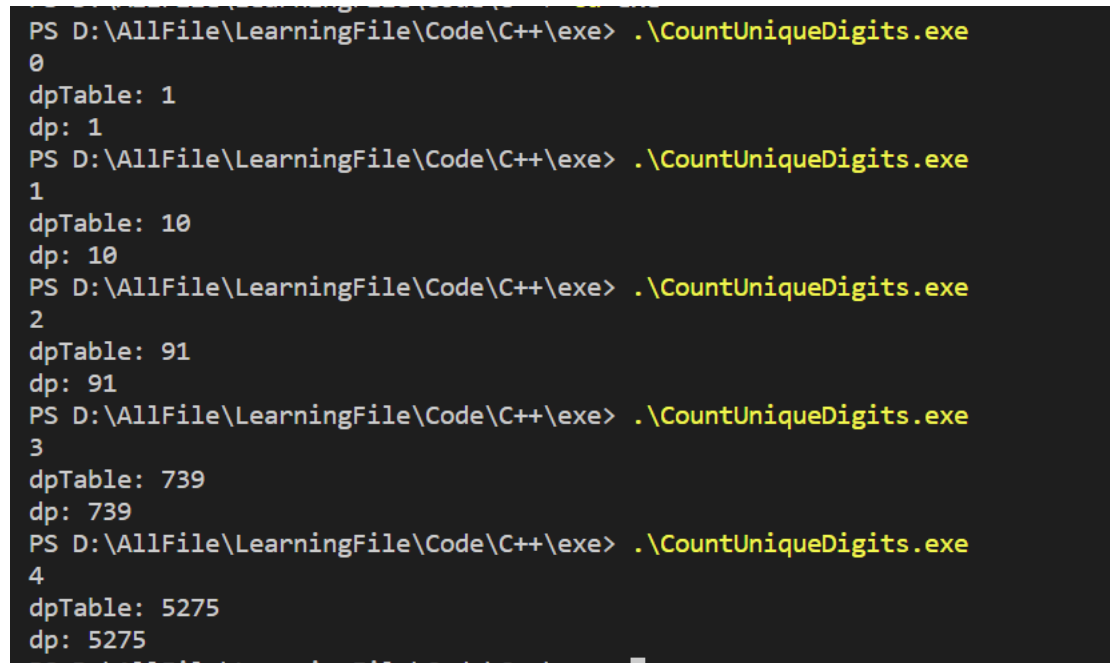
```

1.4 效率分析

在“动态规划”方法中，计算 $f(k)$ 是一个两层循环，最坏情况下会计算 $9 * n$ 次，循环计算结果时是 $n + 1$ 次，所以时间复杂度为 $\Theta(n), f(k)$ 的结果需要数组存储，空间复杂度为 $\Theta(n)$ 。

在“动态规划 + 状态压缩”方法中，只要一次循环就可以了，时间复杂度为 $\Theta(n)$ ，空间复杂度为 $\Theta(1)$ 。

1.5 运行结果



```
PS D:\AllFile\LearningFile\Code\C++\exe> .\CountUniqueDigits.exe
0
dpTable: 1
dp: 1
PS D:\AllFile\LearningFile\Code\C++\exe> .\CountUniqueDigits.exe
1
dpTable: 10
dp: 10
PS D:\AllFile\LearningFile\Code\C++\exe> .\CountUniqueDigits.exe
2
dpTable: 91
dp: 91
PS D:\AllFile\LearningFile\Code\C++\exe> .\CountUniqueDigits.exe
3
dpTable: 739
dp: 739
PS D:\AllFile\LearningFile\Code\C++\exe> .\CountUniqueDigits.exe
4
dpTable: 5275
dp: 5275
```

图 1: 运行结果截图

2 Problem 2

2.1 Question

Given an array of intervals $intervals$ where $intervals = [start_i, end_i]$, return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

2.2 算法思路

采用贪心算法：为了计算最小的重叠区间的个数，可以将问题转换为将去区间放在数轴上，怎样放可以让数轴上容纳最多的不重叠的区间。为了确定一种这样的方法，我们可以先找第一个放在开头的位置，第一个的左边没有区间，所以只需要考虑其右侧是否存在重叠区间。假设我们将左端点最小的区间放在第一个，但如果该区间的右端点较大，则可能让一些小的区间无法放入，因此这种方案并不能保证放最多的区间。如果将右端点最小的区间放在第一个，由于其左边没有区间，故不需要考虑其是否会重叠，而如果存在多个右断点最小，考虑放第 2 个区间时只会考虑第 1 个右端点的值，因此此时选哪个都可以。同样的道理，当放第二个区间时，可能存在多个左端点比第一个右端点大的区间，但考虑到放第 3 个区间，因此还是要优先考虑右端点最小的区间，然后在挑选左端点也满足条件的区间..... 依次类推。

由上模拟分析我们可以得到两点重要的信息：第一，第一个区间是右端点最小的区间，如果存在多个，任意一个都可以；第二，对于放任意一个区间，我们总是会优先考虑右端点最小，然后再考虑其左端点是否会导致重叠。基于这两点，可以得到如下解法：先把所有区间按照右端点进行升序排序，选择最小的作为首个区间；然后遍历检查每一个区间，如果该区间的左端点大于等于上一区间的右端点，则不会重叠，将该区间放在“数轴”上，并更新“数轴”区间最右端的值；如果该区间的左端点小于当前区间的右端点值，说明存在重叠，不能加入“数轴”，重叠的区间数加 1。这种选择区间的方法就是“贪心”的思想。

2.3 核心代码

2.3.1 Interval 结构体

```
1 //Interval 结构体
```

```

2 struct Interval
3 {
4     int start; /*左端点*/
5     int end;   /*右端点*/
6 };

```

2.3.2 Interval 输入处理

如果是个位数，则通过 `isdigit()` 函数可以从 `string` 串中提取数字，但对于多位数，主要是向后查看是否有数字，如果有，则需要将数字连接成一个数，然后通过 C++11 提供的 `stoi()` 函数转换成数，存在 `list` 中，进一步将 `list` 中的数字分对存在在 `Interval` 类型的 `vector` 中。

```

1 // 提取数字
2 vector<Interval> string2Interval(string str)
3 {
4     char numarr[] = {'0','1','2','3','4','5','6','7','8','9'};
5     list<int> numlist;
6     int startIndex = 0;
7
8     while (startIndex != -1)
9     {
10         vector<char> tempnum;
11
12         startIndex = -1;
13
14         for (size_t i = 0; i < str.length(); i++)
15         {
16             for (size_t j = 0; j < (sizeof(numarr) / sizeof(numarr[0])); j++)
17             {
18                 if (str[i] == numarr[j])
19                 {
20                     startIndex = i;
21                     break;
22                 }
23             }
24
25             if (startIndex != -1)
26             {
27                 tempnum.push_back(str[startIndex]);
28                 int tempindex = 0;
29
30                 // 向下查找数据
31                 char tempchar = str[startIndex + (tempindex += 1)];
32

```

```

33     //表示为数字
34     while (int(tempchar - 48) >= 0 && int(tempchar - 48) <= 9)
35     {
36         tempnum.push_back(tempchar);
37         tempchar = str[startIndex + (tempindex += 1)];
38     }
39
40     //删除查询到的数据
41     str.erase(startIndex, tempindex);
42     break;
43 }
44 }
45
46 if (!tempnum.empty()) {
47     //cout << "tempnum : " << string(tempnum.begin(), tempnum.end()) <<
endl;
48     numlist.push_back(stoi(string(tempnum.begin(), tempnum.end())));
49 }
50 }
51
52 /*将list转换为Interval*/
53 vector<Interval> Intervals;
54 Interval tmp; /*临时Interval结构体*/
55 int count = 0;
56 for (list<int>::iterator it = numlist.begin(); it != numlist.end(); it
++)
57 {
58     if(count % 2 == 0)
59     {
60         tmp.start = *it;
61         count++;
62     }
63     else
64     {
65         tmp.end = *it;
66         Intervals.push_back(tmp);
67         count++;
68     }
69
70     //cout << *it << endl;
71 }
72
73 return Intervals;
74 }

```

2.3.3 主问题算法

先按右端点进行排序，然后遍历计算重复区间的个数：假设当前不连续区间的最右端为 `end`，对于新加入的区间，如果它的左端点大于等于 `end`，则说明此时依然不重叠，加入后更新 `end` 值；如果它的左端点小于 `end`，说明存在重叠区间，则重复区间数加 1。

```
1 // 按右端点值升序排序
2 bool endLessSort(Interval a, Interval b)
3 {
4     return a.end < b.end;
5 }
6
7 // 主算法
8 int nonOverlappingIntervals(vector<Interval> intervals)
9 {
10     /*按右断点值从小到大排序*/
11     sort(intervals.begin(), intervals.end(), endLessSort);
12     int len = intervals.size();
13
14     /*记录重复区间个数*/
15     int count = 0;
16
17     /*定义end，表示当前连续不重叠区间的最右端*/
18     int end = intervals[0].end;
19     for (int i = 1; i < len; ++i)
20     {
21         /*没有重叠，需要更新不重叠区间的右端点*/
22         if (end <= intervals[i].start)
23             end = intervals[i].end;
24         /*如果重叠，则count++*/
25         else
26             count++;
27     }
28     return count;
29 }
```

2.3.4 主函数

输入形如 `[[1, 2], [3, 4]]` 的字符串，然后转换为 `Interval` 类型后计算 `count` 并打印。

```
1 // main函数
2 int main()
3 {
```



```

4  /*输入字符串，形式为[[1,2],[3,4],[12,45]]*/
5  string s;
6  cin >> s;
7  cout << nonOverlappingIntervals(string2Interval(s)) << endl;
8  return 0;
9  }

```

2.4 效率分析

在遍历寻找重叠区间过程中循环了 n 次；在对所有区间的右断点进行升序排序时调用 `sort()` 函数，假设该函数以最好的性能进行排序，即时间复杂度为 $\Theta(n \log n)$ 。所以总的时间复杂度为 $\Theta(n \log n)$ 。

2.5 运行结果

```

PS D:\AllFile\LearningFile\Code\C++\exe> .\NonoverlappingIntervals.exe
[[1,2],[2,3],[3,4],[1,3]]
1
PS D:\AllFile\LearningFile\Code\C++\exe> .\NonoverlappingIntervals.exe
[[1,2],[1,2],[1,2]]
2
PS D:\AllFile\LearningFile\Code\C++\exe> .\NonoverlappingIntervals.exe
[[1,2],[2,3]]
0
PS D:\AllFile\LearningFile\Code\C++\exe> .\NonoverlappingIntervals.exe
[[1,2],[1,3],[2,3],[12,13],[3,5],[4,6],[7,10],[11,12],[9,13]]
3
PS D:\AllFile\LearningFile\Code\C++\exe> .\NonoverlappingIntervals.exe
[[1,1],[3,4],[1,2],[1,3],[2,3],[1,4]]
2

```

图 2: 运行结果截图