

Assignment Three - Programming

1953871 邓泉

Problem 1

1 分析

- 1.1 问题重述

输入：非负整数 n

输出： $[0, 10^n)$ 范围内的各位上数值都不同的数字 x

- 1.2 参数分析

$$0 \leq n < 10$$

2 算法设计

- 2.1 算法思路

数字的基本元素是 $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$ 共10个数字。

$n = 0$	$[0]$	1
$n = 1$	$[0; 1, 2 \dots 9]$	$1 + 9$
$n = 2$	$[0; 1, 2 \dots 9; 10, 12 \dots, 98(\text{除去} 11, 22, \dots, 99)]$	$1 + 9 + 9 \times (10 - 1)$
$n = 3$	$[0; 1, 2 \dots 9; 10, 12 \dots, 98(\text{除去} 11, 22, \dots, 99); 100, 101 \dots 998(\text{除去} 111, 222 \dots 999; 11)]$	$1 + 9 + 9 \times (10 - 1) + 9 \times (10 - 1) \times (10 - 2)$
.	.	.
.	.	.

$n \geq 1$ 时， n 就表示数字 x 的最大位数。从 $n = 1$ 起，往后 n 每增加1，数字 x 所允许的最大位数就增加1，记 $dp(k)$ 表示 n 从 $k - 1$ 增加到 k 后所新增的数字个数。

$$dp(0) = 1$$

$$dp(1) = 9$$

考虑 n 从 $k - 1$ 到 k 时 ($2 \leq k < 10$)，新增数字必定是 k 位数，它们的生成可视为：在满足条件的 $dp(k - 1)$ 个 $k - 1$ 位数（即 n 从 $k - 2$ 到 $k - 1$ 时新增的数字）的末尾添加一个数字 ($0 \sim 9$)，形成 $dp(k - 1) \times 10$ 个 k 位数，再从中剔除有重复数位的数字。

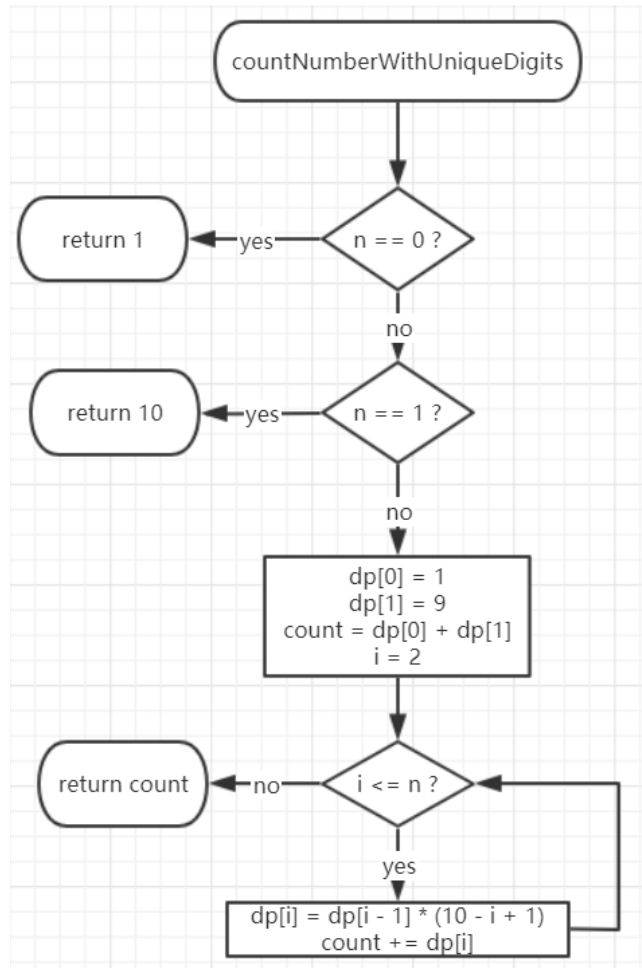
例如： n 从 1 到 2 时，先在数字 $1 \sim 9$ 末尾各添加一个数位 ($0 \sim 9$)，形成了 9×10 个两位数，再从中剔除掉 $11, 22, \dots, 99$ 等 9 个不满足条件的数字，最后得到了 $9 \times (10 - 1)$ 个两位数，即 $dp(2) = dp(1) \times (10 - 1)$ 。

不难发现, n 从 $k-1$ 到 k 时 ($2 \leq k < 10$), 先形成 $dp(k-1) \times 10$ 个 k 位数, 而在生成过程中: 对于每一个作为基础的 $k-1$ 位数, 会产生 $(k-1)$ 个重复的 k 位数, 因此共有 $dp(k-1) \times (k-1)$ 个 k 位数需要被舍弃, 所以有以下公式:

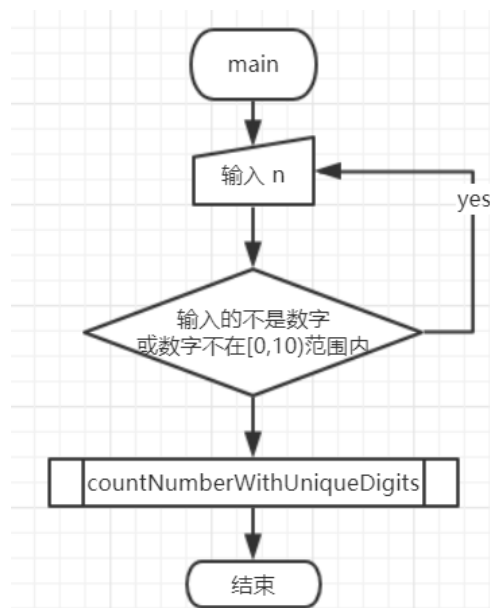
$$dp(n) = dp(n-1) * (10 - n + 1) \text{ for } 2 \leq n < 10$$

• 2.2 流程图

◦ 功能函数 `countNumberWithUniqueDigits`



◦ 主函数 `main`



• 2.3 实现

◦ 功能函数 `countNumberWithUniqueDigits`

```
// 自底向上的DP算法 动态规划
int countNumberWithUniqueDigits(int n)
{
    if (n == 0) {
        return 1;
    }
    else if (n == 1) {
        return 10;
    }
    int dp[10];
    dp[0] = 1;
    dp[1] = 9;
    int count = dp[0] + dp[1];
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] * (10 - i + 1);
        count += dp[i];
    }
    return count;
}
```

◦ 主函数 `main`

```
int main()
{
    int n;

    // 输入
    while (1) {
        cout << "n = ";
        cin >> n;
        if (!cin.good()) {
            cin.clear();
            cin.ignore(65535, '\n');
            cout << "输入非法, 请重新输入。" << endl;
            system("pause");
            system("cls");
        }
        else if (n < 0 || n > 10) {
            cout << "输入的数字不在[0,10]范围内, 请重新输入。" << endl;
            system("pause");
            system("cls");
        }
        else {
            break;
        }
    }

    // 输出
    cout << countNumberWithUniqueDigits(n) << endl;

    return 0;
}
```

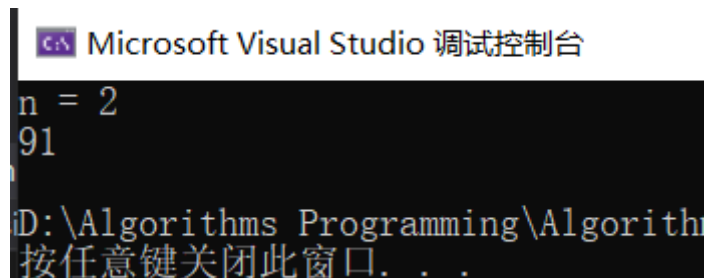
- 2.4 复杂度分析

时间复杂度	空间复杂度
$O(n)$	$O(n)$

3 测试

- 3.1 case 1


Input : 2
Output :



```
Microsoft Visual Studio 调试控制台
n = 2
91
D:\Algorithms Programming\Algorith
按任意键关闭此窗口. . .
```

- 3.2 case 2 (边界测试)

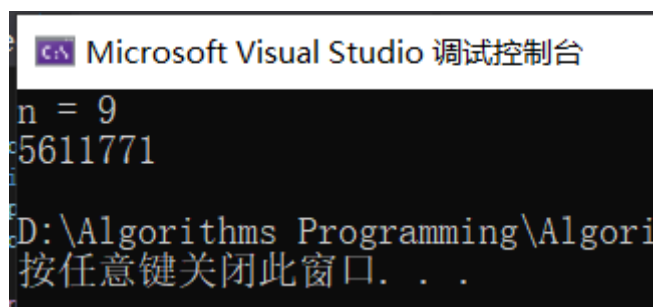
Input : 0
Output :



```
Microsoft Visual Studio 调试控制台
n = 0
1
D:\Algorithms Programming\Algori
按任意键关闭此窗口. . .
```

- 3.3 case 3 (边界测试)

Input : 9
Output : 5611771



```
Microsoft Visual Studio 调试控制台
n = 9
5611771
D:\Algorithms Programming\Algori
按任意键关闭此窗口. . .
```

- 3.4 case 4 (错误测试)

Input : 10

Output : (提示输入错误, 按任意键后即可重新输入)

```
D:\Algorithms Programming\Algorithms Programming
n = 10
输入的数字不在[0, 10) 范围内, 请重新输入。
请按任意键继续. . .
```

(按任意键后...)

```
D:\Algorithms Programming\Algorit
n = _
```

Problem 2

1 分析

- 1.1 问题重述

给定了一系列区间interval, 每个区间有一个首元素start和尾元素end, 并且 $end > start$, 我们需要做这样的事情: 在这一系列区间中, 删除某些区间使得剩余的区间不存在重叠的情况, 我们需要保证所删除的区间数目最少。计算让这些区间互不重叠所需要移除区间的最少个数。起止相连不算重叠, 例如[1, 2]和[2, 3]这样存在边界接触的不认为是重叠。

输入: 一个数组, 数组由多个长度固定为 2 的数组组成, 表示区间的开始和结尾。

输出: 一个整数, 表示需要移除的区间数量。

2 算法设计

- 2.1 算法思路

题目给出了一系列interval, 这些interval不一定按照某种顺序排列, 关键词提到了最小删除interval的数量, 很容易想到贪心算法。因此, 我设计的贪心算法目的是: 在当前情况下, 能保证当前已经访问过的interval的集合都是不重叠的, 并且删除了最小数目的重叠interval。

于是, 自然地想到了以下两个思路:

- 思路1: 按start升序排列 (错误)

将所有的区间interval根据start元素升序排列, 在start相同的时候, 根据end升序排列。这样的序列我们进行这样的操作: 从第二个interval开始, 当后面的元素start小于前一个元素的end, 说明就会发生重叠, 于是将其删除, 并且计数器count记录一次, 反之则代表不会重叠。以此类推, 对intervals遍历一次后就会把重复的interval删去。

此法看起来有道理，但实际是错误的，考虑以下的情况：

```
Input : [[1,2], [3,4], [5,6], [2,9]]
Sorted : [[1,2], [2,9], [3,4], [5,6]]

Expected Output : 1    // only remove [2, 9]
Actually Output : 2    // remove [3,4] and [5,6]
```

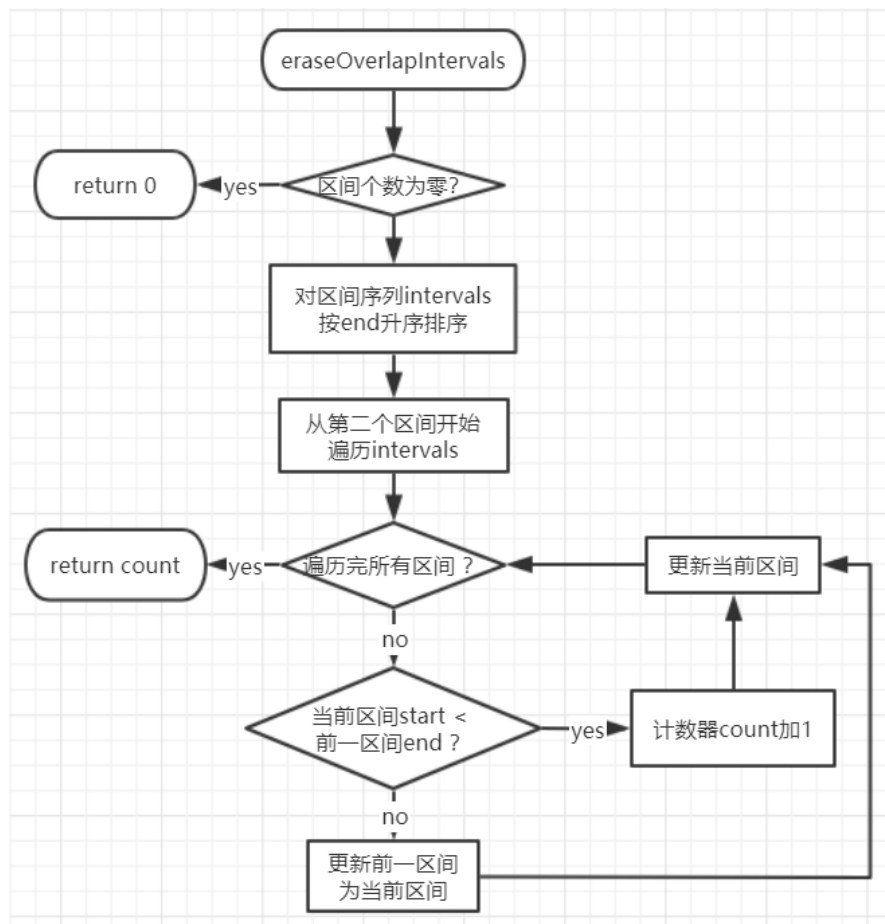
思路2：按end升序排列（正确）

在思路1的基础上稍作修改，将所有的interval根据end元素升序排列，在end相同的时候，根据start升序排列。这样的序列保证了end从最小开始，只需判断每一个的interval的start是否会超过end即可，遍历算法与上一个方法类似。

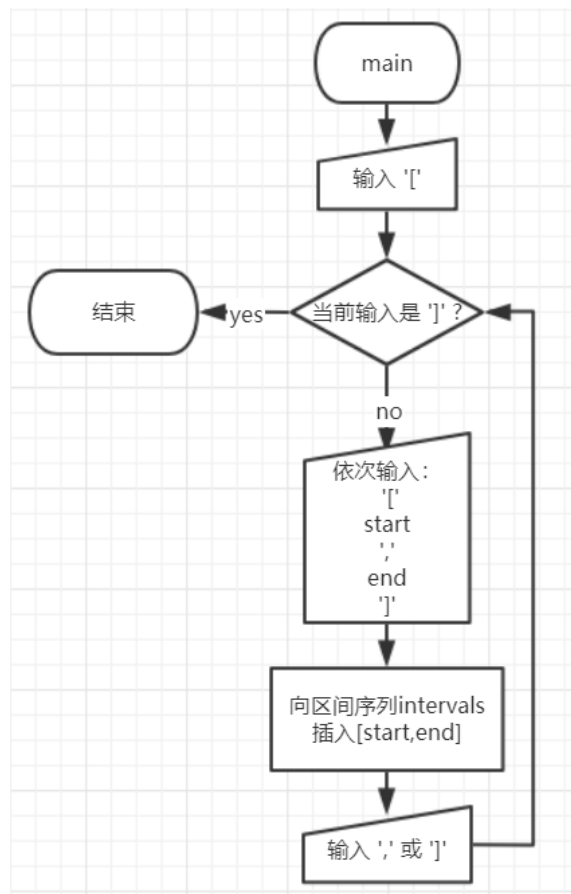
这种算法的好处在于避免了这样的情况（也即思路1错误的原因）：保留了end过大的interval，甚至将很多start排在后面的区间包含了进去，导致不得不舍弃后面更多数量的interval。在思路2下，会删除这个end大的区间，这样才能保证最小删除区间的数目。

2.2 流程图

功能函数 `eraseOverlapIntervals`



主函数 `main`



• 2.3 实现

◦ 自定义类型 `Interval` (区间)

```

struct Interval {
    int start;
    int end;
    Interval() : start(0), end(0) {}
    Interval(int s, int e) : start(s), end(e) {}
};
  
```

◦ 功能函数 `eraseOverlapIntervals`

```

int eraseOverlapIntervals(vector<Interval>& intervals)
{
    int count = 0;
    if (intervals.size() == 0) {
        return 0;
    }
    // 排序
    sort(intervals.begin(), intervals.end(), CompareInterval);
    int current_end = intervals[0].end;
    for (int i = 1; i < (int)intervals.size(); i++) {
        if (intervals[i].start < current_end) {
            count++;
        }
        else {
            current_end = intervals[i].end;
        }
    }
    return count;
}
  
```

```
}
```

o 主函数 main

```
int main()
{
    vector<Interval> intervals;
    // 输入 形如[[1,2],[2,3],[3,4],[1,3]]
    char ch;
    int s, e;
    cout << "intervals = ";
    cin >> ch; // 开始: 输入[
    while (ch != ']') {
        cin >> ch; // [
        cin >> s;
        cin >> ch; // ,
        cin >> e;
        cin >> ch; // ]
        intervals.push_back(Interval(s, e));
        cin >> ch; // , 继续; ] 结束
    }
    // 输出
    cout << eraseOverlapIntervals(intervals) << endl;

    return 0;
}
```

• 2.4 函数 eraseOverlapIntervals 复杂度分析

输入区间的个数记作 N 。eraseOverlapIntervals函数空间复杂度主要来源于vector容器以及sort排序可能产生的空间开销；时间复杂度主要来源于：对intervals的排序 + 遍历排序后的intervals，容易得到遍历过程的时间复杂度是 $O(N)$ 。而sort排序的时间复杂度和额外的空间复杂度则需要做一个更详细的讨论。

STL的sort()算法，数据量大时采用**Quick Sort**，分段递归排序，一旦分段后的数据量小于某个门槛，为避免Quick Sort的递归调用带来过大的额外负荷，就改用**Insertion Sort**。如果递归层次过深，还会改用**Heap Sort**。

实际上，STL sort算法是以上三种算法的综合，被称作Introspective Sorting(内省式排序)。sort函数一开始就判断序列大小，通过个数检验之后，再检测分割层次，若分割层次超过指定值，就改用Heap sort。都通过了这些校验之后，便进入与Quick Sort完全相同的程序。

1. N 很小，sort算法视作Insertion Sort(平均时间复杂度 $O(N^2)$ ，辅助存储 $O(1)$)

时间复杂度	空间复杂度
$O(N^2)$	$O(N)$

2. sort算法视作Heap Sort(平均时间复杂度 $O(N \log_2 N)$ ，辅助存储 $O(1)$)

时间复杂度	空间复杂度
$O(N \log_2 N)$	$O(N)$

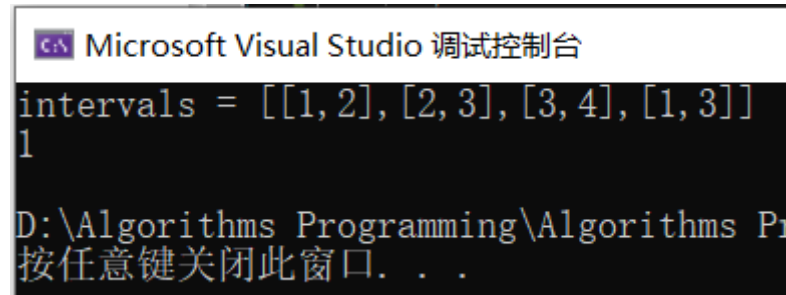
3. N 足够大，sort算法视作Quick Sort(平均时间复杂度 $O(N \log_2 N)$ ，辅助存储 $O(N \log_2 N)$)

时间复杂度	空间复杂度
$O(N \log_2 N)$	$O(N \log_2 N)$

3 测试

• 3.1 case 1

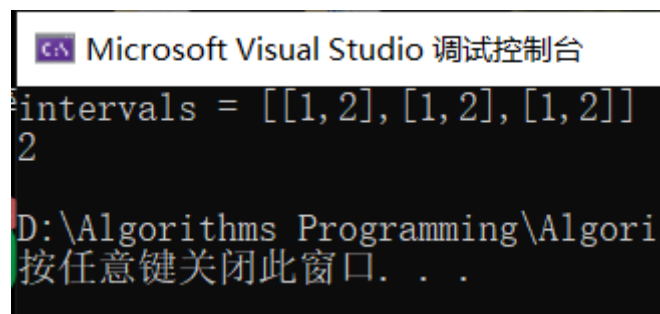
Input : `[[1,2],[2,3],[3,4],[1,3]]`
Output : 1



```
Microsoft Visual Studio 调试控制台
intervals = [[1,2],[2,3],[3,4],[1,3]]
1
D:\Algorithms Programming\Algorithms Pr
按任意键关闭此窗口. . .
```

• 3.2 case 2

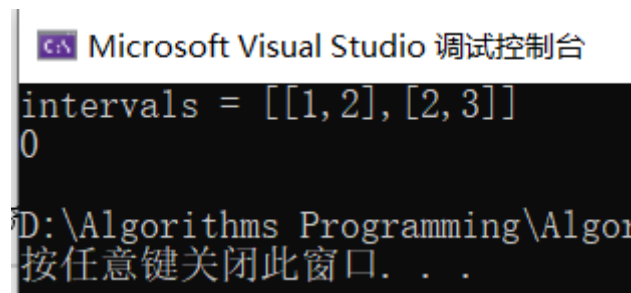
Input : `[[1,2],[1,2],[1,2]]`
Output : 2



```
Microsoft Visual Studio 调试控制台
intervals = [[1,2],[1,2],[1,2]]
2
D:\Algorithms Programming\Algori
按任意键关闭此窗口. . .
```

• 3.3 case 3

Input : `[[1,2],[2,3]]`
Output : 0



```
Microsoft Visual Studio 调试控制台
intervals = [[1,2],[2,3]]
0
D:\Algorithms Programming\Algor
按任意键关闭此窗口. . .
```