目录

数位 dp

1. 数位 dp 概述

数位动态规划（数位 DP）主要用于解决“在区间[1,r]这个范围内，满足某种约束的数字的数量、总和、平方“这一类问题，

针对这类问题，算法竞赛有两类写法，一种是记忆化搜索写法，一种是迭代写法。

在学习时，推荐大家使用记忆化搜索写法，原因如下：

记搜写法容易举一反三、易编码，预处理后的迭代写法，往往边界条件很多，状态转移方程容易写错或漏项，其边界容易漏判误判；且不同题目时，迭代写法的 DP 过程变化较大，而记搜的 dfs 框架则非常套路，容易举一反三。

1.1. 引入

数位 dp 有个通用的套路，就是先采用前缀和思想，将求解“[L,R]这个区间内的满足约束的数的数量”，转化为“[1,R]满足约束的数量 - 区间[1,L-1]满足约束的数量”。

所以我们最终要求解的问题通通转化为：[1,x]中满足约束的数量，或者[0,x]中的满足约束的数量（左边界取决于题目）。

然后将数字 x 拆分为一个个数位：

数位，如个位、十位、百位等，单个数码（比如十进制，此处就是指 0~9）在数 x 中所占据的一个位置

在代码中表现为：

a[1…len]：将数 x 分解为 R 进制（一般为十进制或者二进制），用数组存储，a[i]表示 x 在 R^(i−1)处的系数。

即 x 这个数的长度为 len，最高位上的数字为 a[len]，最低位上的数字为 a[1]

```
typedef long long LL;
LL solve(LL x)
{
    int len = 0;
    while(x > 0)
    {
        a[++ len] = x % 10;
        x /= 10;
    }
    return dfs(...); //记忆化搜索
}
```

思考：为什么低位数字存在低位？高位数字存在高位？

接下来考虑填数，高位往低位填，即 len->1，我们用一道例题一起练习一下。

1.2. 例题 烦人的数学作业（洛谷 P4999）

【问题描述】

求解区间[L,R]中所有数的数位和之和

数位和：一个数的所有数位上的数字加起来的和，比如 313 的数位和为 3+1+3=7

共 1≤T≤20 组数据，其中 1 <= L <= R <= 1e18。由于答案可能过大，最终答案 mod 1e9+7。

【问题分析】

既然叫做记忆化搜索，也就是就是层层深入，每一层搜索填写一个数，记忆化搜索函数 dfs 中，我们用形参 pos 来表示当前需要填写的位置

pos：int 型变量，表示当前枚举的位置，一般从高到低

我们需要计算的是[0,x]的所有数的数位和之和

假设 x=4132，我们用?来表示暂未填写的数位，则现在填数状态为????

我们第一步需填写第 len 位（最高位），但是很明显我们只能填写 0~4

1、 若填写大于 4 的数码的话，显然不在我们的枚举范围。比如 5???无论你怎么填写，也不可能在[0,4132]这个范围内
2、 若填写 0~3 的话，那就说明我们后面的数字可以任意填写，例如填写 3，区间[3000,3999]全部位于[0,4132]之中
3、 若填写 4 的话，则后续填写的数字还是会受到限制，比如下一位就不能超过 a3=1 了，否则就超出[0,4132]这个区间了

所以我们需记录一个变量 limit，表示当前数位是否可以任意填写，故在记忆化搜索函数 dfs 的常设定的形参加上 limit。

limit：bool 型变量，表示枚举的第 pos 位是否受到限制；

为 true 表示取的数不能大于 a[pos]，而只有在[pos+1,len]的位置上填写的数都等于 a[i]时该值才为 true

否则表示当前位没有限制，可以取到[0,R−1]，因为 R 进制的数中数位最多能取到的就是 R-1

当我们搜索到 pos=0 时，就表示所有数位都填写完毕了，每个“?”都替换成了具体的数字。显然这是一个递归边界，我们需要返回枚举填写的所有数位和 sum，故 dfs 函数的形参还需要添加：

sum：int 型变量，表示当前 len→(pos+1)的数位和

注：因为我们计算的时一个区间的答案，这个答案是总体的，并不需要方案，所以我们不用像某些搜索的题目里面用 b[1…len]数组记录下每个位置的选择，我们只需要记录最后有用的信息，在此处也就是 sum

上述部分就是普通的 dfs 搜索，其过程就像是一棵“满多叉树”，时间复杂度最坏为 O(10^len)，其中 len 最大值为 19（1e18 有 19 位），这显然不是我们能接受的复杂度。

而动态规划就是减少冗余的重复计算，也就是我们将这棵“满多叉树”中相同的子树部分给删除掉，从而来优化时空复杂度。

设状态 f[pos][sum]表示：

位置[pos+1,len]都已填写完毕，且这些数位之和为 sum 的情况下，数位[1,pos]任意填写（即 limit 为 false）

f[pos][sum]为满足约束的所有数的数位和之和。

对于[1,pos]来说他们根本就不知道前面填写了什么，他们只关心[pos+1,len]的数位和

所以如果填写状态为 03??、12??、21??、30??时，其实对于后面来说都是一样，我们只需要搜索出 03??的结果，另外几个的被搜索到的时候就可以直接查表返回答案。

故我们可以大致写出记忆化搜索的代码了：

其中 f 数组初始化均为−1，而~f[pos][num]是按位取反操作，当值等于-1 时返回 0，也就是判断值是否为-1

```
LL dfs(int pos, bool limit, int sum)
{
    if(!pos) //递归边界
        return sum;
    if(!limit && ~f[pos][sum]) //没限制并且 dp 值已搜索过
        return f[pos][sum];
    int up = limit ? a[pos] : 9;
    LL res = 0;
    for(int i = 0; i <= up; i ++)
        res = (res + dfs(pos - 1, limit && i == up, sum + i)) % md;
    if(!limit) //记搜，可复用
        f[pos][sum] = res;
    return res;
}
```

然后在 solve 中调用 dfs(len,true,0)即可。

我们可以从 limit 中知道，当前位置能否任意取，从而知道当前位置的上界 up 是多少，故：

up = limit ? a[pos] : 9;

我们知道只有当[pos+1,len]都取到 a[i]时，limit 才会为 true（注意初值就是 true，因为最高位一开始不能乱取），所以我们搜索的时候可以一路按位与过去：

limit && i == a[pos] // limit && i == up

那么，问题来了

问题 1：怎么证明这个记忆化搜索的优化了原先的时间复杂度？

问题 2：为什么状态需要设置为不受 limit 限制的前提下？

我们来解决问题 1：怎么证明这个记忆化搜索的优化了原先的时间复杂度？

先考虑 f 的状态数量，取最大值 len=19，sum=9×18+1=163

即 18 个 9 的时候 sum 最大（≤1e18）

为故总的状态数为 19×163=3097，是一个非常少的状态数

考虑每个状态被访问了多少次显然不好直接计算，我们考虑一个状态计算值的时候，需访问多少个状态，容易得到为 10 个状态，故如果只想要计算出 f 的时间复杂度就是 O(len×sum×R)，也就是 3097×10 = 30979。

接下来我们来看问题 2：为什么状态需要设置为不受 limit 限制的前提下？

多组数据的话，这个 f 数组是可以复用的。

从问题 1 我们可以得知，limit 为 true 的情况只有一条链，在 f 中记录毫无意义，因为这些状态根本不会被重复访问。并且，多组数据时我们还需要清除掉这个 limit 为 true 的数组，因为它们对于的 a[1…len]不相同，没法复用，也就是浪费时空复杂度。

故你不会在任何数位 dp 的题目中将 limit 作为 dp 的一维状态。

完整代码如下：

```
#include <bits/stdc++.h>
using namespace std;

typedef long long LL;
constexpr int N = 20, md = 1e9 + 7;
int a[N];
LL f[N][9 * 18 + 5];
// f[pos][sum]：在[pos + 1, len]中的数位和为 sum，[1, pos]的数位任意填。满足这样约束的数的总和
// 总共最多 19 * (9 * 18)个状态
LL dfs(int pos, bool limit, int sum)
{
    if(!pos) //递归边界
        return sum;
    if(!limit && ~f[pos][sum]) //没限制并且 dp 值已搜索过
        return f[pos][sum];
    int up = limit ? a[pos] : 9;
    LL res = 0;
    for(int i = 0; i <= up; i ++)
```

```
        res = (res + dfs(pos - 1, limit && i == up, sum + i)) % md;
    if(!limit) //记搜，可复用
        f[pos][sum] = res;
    return res;
}

LL solve(LL x)
{
    int len = 0;
    while(x > 0)
    {
        a[++ len] = x % 10;
        x /= 10;
    }
    return dfs(len, true, 0); //初始状态可以理解为 len 之前全部卡前导 0 的上
}
int T;
int main()
{
    memset(f, -1, sizeof f); //可复用的，多组样例都可使用
    cin >> T;
    while(T --)
    {
        LL l, r;
        cin >> l >> r;
        LL ans = (solve(r) - solve(l - 1) + md) % md;
        if(ans < 0)
            ans += md;
        cout << ans << '\n';
    }
}
```

1.3. 常用形参总结

在上述例题 0 中，我们已经知道了数位 dp 的过程，我们往往会把约束设置为形参和动态规划的状态。

以下为记忆化搜索函数 dfs 的常设定的形参

pos：int 型变量，表示当前枚举的位置，一般从高到低。

limit：bool 型变量，表示枚举的第 pos 位是否受到限制，

为 true 表示取的数不能大于 a[pos]，而只有在[pos+1, len]的位置上填写的数都等于 a[i]时该值才为 true

否则表示当前位没有限制，可以取到[0, R−1]，因为 R 进制的数中数位最多能取到的就是 R−1

last：int 型变量，表示上一位（第 pos+1 位）填写的值

往往用于约束了相邻数位之间的关系的题目

lead0：bool 型变量，表示是否有前导零，即在 len→(pos+1)这些位置是不是都是前导零

基于常识，我们往往默认一个数没有前导零，也就是最高位不能为 0，即不会写为 000123，而是写为 123

只有没有前导零的时候，才能计算 0 的贡献。

那么前导零何时跟答案有关？

统计 0 的出现次数、相邻数字的差值、以最高位为起点确定的奇偶位。

sum：int 型变量，表示当前 len→(pos+1)的数位和

r：int 型变量，表示整个数前缀取模某个数 m 的余数

该参数一般会用在：约束中出现了能被 m 整除

当然也可以拓展为数位和取模的结果

st：int 型变量，用于状态压缩

对一个集合的数在数位上的出现次数的奇偶性有要求时，其二进制形式就可以表示每个数出现的奇偶性

2. 基础题目

2.1. 洛谷 P2602 数字计数

【问题描述】

给定两个正整数 a 和 b，求在[a,b]中的所有整数中，每个数码(digit)各出现了多少次。其中 1 <= a <= b <= 1e12

【样例输入】

```
1 99
```

【样例输出】

```
9 20 20 20 20 20 20 20 20 20
```

【参考代码】

```
#include <bits/stdc++.h>
using namespace std;

typedef long long LL;
constexpr int N = 14;
int a[N];
int digit;//当前需要统计的数字
// f[pos][cnt][0/1]: 当最高位已经填了（没有前导 0 时）在[pos + 1, len]中 digit 填了 cnt 个，[1, pos]任意填，digit 出现的次数
// 第三维 0 表示 digit=0,第三维为 1 表示 digit!=1
// 当 limit=false 时，容易知道填 1-9 的数量是相同的
LL f[N][N][2];

LL dfs(int pos, bool limit, bool lead0, int cnt)
{
    if (!pos) // 递归边界
        return cnt;
    auto &now = f[pos][cnt][digit != 0];
    if (!limit && !lead0 && ~now) // 没限制并且 dp 值已搜索过
        return now;
    int up = limit ? a[pos] : 9;
    LL res = 0;
    for(int i = 0; i <= up; i ++)
    {
        //填 0 的时候需注意是否为前导
        //前导零不算入 0 的个数
        int tmp = cnt + (i == digit);
        if(lead0 && digit == 0 && i == 0)
            tmp = 0;
        res += dfs(pos - 1, limit && i == up, lead0 && i == 0, tmp);
    }
    if (!limit && !lead0) //无限制并且没有前导 0
```

```
        now = res;
    return res;
}

LL ans[10];
void solve(LL x, int f){
    int len = 0;
    while (x > 0)
    {
        a[++len] = x % 10;
        x /= 10;
    }
    for(int i = 0; i <= 9; i ++)
    {
        digit = i;
        ans[i] += f * dfs(len, true, true, 0);
    }
}

int main()
{
    memset(f, -1, sizeof f);
    LL l, r;
    cin >> l >> r;
    solve(r, 1); //加上[1, r]
    solve(l - 1, -1); //扣除掉[1, l-1]
    for(int i = 0; i <= 9; i ++)
        cout << ans[i] << ' ';
}
```

2.2. 洛谷 P2657 Windy 数

【问题描述】
Windy 定义了一种 Windy 数。
不含前导零且相邻两个数字之差至少为 2 的正整数被称为 Windy 数。
Windy 想知道，在 A 和 B 之间，包括 A 和 B，总共有多少个 Windy 数？
【输入】
输出文件包含两个数，A B。
【输出】
输出文件包含一个整数。
【样例输入 1】

```
1 10
```

【样例输出 1】

```
9
```

【样例输入 2】

```
25 50
```

【样例输出 2】

```
20
```

【参考代码】

```
#include <bits/stdc++.h>
using namespace std;
```

```
typedef long long LL;
constexpr int N = 11, INF = 1e9;
int a[N];
// f[pos][last]: 长度为pos+1的以last开头的windy数的数量
// 相当于[1, pos]没有填, 第pos+1位填了last
int f[N][10];

int dfs(int pos, bool limit, bool lead0, int last)
{
    if (!pos) // 递归边界
        return 1;
    if (!limit && last != INF && ~f[pos][last]) // 没限制并且dp值已搜索过, 并且last填了
        return f[pos][last];
    int up = limit ? a[pos] : 9;
    int res = 0;
    for(int i = 0; i <= up; i ++)
    {
        //填0的时候需注意是否为前导
        if(lead0) // 如果是前导0表示还没有开始填数, 则需要让last依旧不约束下一个数
            res = res + dfs(pos - 1, limit && i == up, lead0 && i == 0, i == 0 ? last : i);
        else
            if (abs(last - i) >= 2) // 与上一个数差值至少为2
                res = res + dfs(pos - 1, limit && i == up, false, i);
    }
    if (!limit && last != INF) // 记搜, 并且last填了
        f[pos][last] = res;
    return res;
}

int solve(LL x)
{
    int len = 0;
    while (x > 0)
    {
        a[++len] = x % 10;
        x /= 10;
    }
    return dfs(len, true, true, INF); // last设置为一个没有约束下一个数位的数
}

int main()
{
    memset(f, -1, sizeof f);
    int l, r;
    cin >> l >> r;
    LL ans = solve(r) - solve(l - 1);
    cout << ans << '\n';
}
```

2.3. 洛谷 P4317 花神的数学题

【问题描述】

设 sum(i)表示 i 的二进制表示中 1 的个数。给出一个正整 1≤N≤1e15 ，花神要问你∏（i=1~N）=sum(i) ，也就是 sum(1)~sum(N)的乘积，答案取模 10000007

【样例输入】

```
3
```

【样例输出】

```
2
```

【参考代码】

```
#include <bits/stdc++.h>
using namespace std;

typedef long long LL;
constexpr int N = 60, md = 10000007;

int a[60];

// f[pos][cnt] 表示在[pos + 1, len]中已经填写了 cnt 个 1，[1, pos]任意填写数，所有合法方案的乘积
LL f[N][N];
LL dfs(int pos, int limit, int cnt)
{
    if(!pos)
        return max(cnt, 1); //当枚举到 0 的时候，得返回否则会让所有答案都为 0
    if(!limit && ~f[pos][cnt])
        return f[pos][cnt];
    LL res = 1;
    int up = limit ? a[pos] : 1;
    for(int i = 0; i <= up; i ++)
        res = res * dfs(pos - 1, limit && i == up, cnt + (i == 1)) % md;
        //本题算的是乘积
    if(!limit)
        f[pos][cnt] = res;
    return res;
}

LL solve(LL x)
{
    int len = 0;
    while(x > 0)
    {
        a[ ++ len] = x % 2;
        x /= 2;
    }
    return dfs(len, true, 0);
}

int main()
{
    memset(f, -1, sizeof f);
    LL n;
```

```
    cin >> n;
    cout << solve(n);
}
```

3. 强化练习

3.1. 洛谷 P6218 Round Numbers S

【问题描述】

如果一个正整数的二进制表示中，0 的数目不小于 1 的数目，那么它就被称为「圆数」。

例如，9 的二进制表示为 1001，其中有 2 个 0 与 2 个 1。因此，9 是一个「圆数」。

请你计算，区间 [l,r] 中有多少个「圆数」。

对于 100% 的数据，1≤l,r≤2×1e9。

【输入】

一行，两个整数 l,r。

【输出】

一行，一个整数，表示区间 [l,r] 中「圆数」的个数。

【输入样例】

```
2 12
```

【输出样例】

```
6
```

【算法分析】

本题的约束条件，显然为 0 的数量 num0 和 1 的数量 num1

[1,pos]中并不关心前面是怎么填写的，只关心前面填了多少个 1 和 0

故我们设状态 f[pos][num0][num1]：

在[pos+1,len]中已经使用了 num0 个 0，num1 个 1，[1,pos]任意填。

满足这样约束的圆数的数量

当然注意递归边界时，需要判断之前填写的是不是圆数，而不是直接返回 1

```
    if (!pos) // 递归边界
        return num0 >= num1;
```

另外，既然涉及到了 0 的个数，显然我们就应该计算前面是否为前导零 lead0，否则就会误把前导零也计入 0 的个数

【参考代码】

```
#include <bits/stdc++.h>
using namespace std;

typedef long long LL;
constexpr int N = 32;
int a[N];
int f[N][N][N];
// f[pos][num0][num1]
// 在[pos + 1, len]中已经使用了 num0 个 0，num1 个 1，[1, pos]任意填。满足这样约束的圆数的数量

int dfs(int pos, bool limit, bool lead0, int num0, int num1)
{// 前导 0，二进制中 0 的个数（非前导 0），二进制中 1 的个数
    if (!pos) // 递归边界
        return num0 >= num1;
    auto &now = f[pos][num0][num1];
    if (!limit && ~now) // 没限制并且 dp 值已搜索过
        return now;
```

```
    int up = limit ? a[pos] : 1;
    int res = 0;
    for(int i = 0; i <= up; i ++)
    {
        bool suf0 = lead0 && i == 0; // 后继状态的前导零
        int s0 = suf0 ? 0 : num0 + (i == 0);
        int s1 = suf0 ? 0 : num1 + (i == 1);
        res = res + dfs(pos - 1, limit && i == up, suf0, s0, s1);
    }
    if (!limit) // 记搜，可复用
        now = res;
    return res;
}
// 实际上统计的为[0, x]的圆数个数
int solve(int x)
{
    int len = 0;
    while (x > 0)
    { // 转化为二进制形式
        a[++len] = x % 2;
        x /= 2;
    }
    return dfs(len, true, true, 0, 0);
}
int main()
{
    memset(f, -1, sizeof f); // 可复用的，多组样例都可使用
    int l, r;
    cin >> l >> r;
    int ans = solve(r) - solve(l - 1);
    cout << ans << '\n';
}
```

3.2. P4124 手机号码

【问题描述】

人们选择手机号码时都希望号码好记、吉利。比如号码中含有几位相邻的相同数字、不含谐音不吉利的数字等。手机运营商在发行新号码时也会考虑这些因素，从号段中选取含有某些特征的号码单独出售。为了便于前期规划，运营商希望开发一个工具来自动统计号段中满足特征的号码数量。

工具需要检测的号码特征有两个：号码中要出现至少 3 个相邻的相同数字；号码中不能同时出现 8 和 4。号码必须同时包含两个特征才满足条件。满足条件的号码例如：13000988721、23333333333、14444101000。而不满足条件的号码例如：1015400080、10010012022。

手机号码一定是 11 位数，前不含前导的 0。工具接收两个数 L 和 R，自动统计出 [L,R] 区间内所有满足条件的号码数量。L 和 R 也是 11 位的手机号码。

数据范围：1e10≤L≤R<1e11。

【输入】

输入文件内容只有一行，为空格分隔的 2 个正整数 L,R。

【输出】

输出文件内容只有一行，为 1 个整数，表示满足条件的手机号数量。

【输入样例】

```
12121284000 12121285550
```

【输出样例】

```
5
```

【样例解释】

满足条件的号码： 12121285000、 12121285111、 12121285222、 12121285333、 12121285550。

【算法分析】

对于不能同时出现 4 和 8 这个约束，我们需设参数 have4 表示 4 是否出现过，以及 have8 表示 8 是否出现过。

涉及到相邻数的约束，我们就不得不设置上一个数 last 这个参数，但是要至少三个相同数字相邻，所以我们设参数 last1 表示前一位（第 pos+1 位）填写的数字，last2 表示前两位（第 pos+2 位）填写的数字，然后枚举当前位置 pos 的值，就可以判断三者是否相同，这个结果记录到形参 same 当中。

故我们可以得到 dp 的状态：f[pos][last1][last2][same][have4][have8]

same 表示在[pos+1, len]中是否已经出现了至少连续三个相同数字

last1, last2 表示第 pos+1, pos+2 位填写的分别是什么数字

have4 表示[pos+1, len]中是否出现了 4

have8 表示[pos+1, len]中是否出现了 8

[1, pos]任意填写数

满足以上约束的所有合法方案的数量

值得注意的是，根据题目描述，本题的最高位数字的枚举需要从 1 开始

【参考代码】

```
#include <bits/stdc++.h>
using namespace std;

typedef long long LL;
constexpr int N = 12;

int a[N];

// f[pos][last1][last2][same][have4][have8] 表示在[pos + 1, len]中是否已经出现了至少连续三个相同数字(same)
// last1, last2 表示第 pos + 1, pos + 2 位填写的分别是什么数字
// have4 表示[pos + 1, len]中是否出现了 4
// have8 表示[pos + 1, len]中是否出现了 8
// [1, pos]任意填写数，所有合法方案的数量

LL f[N][11][11][2][2][2];
int len;

LL dfs(int pos, int limit, int last1, int last2, bool same, bool have4, bool have8)
{
    if(!pos)
        return same && !(have4 && have8) ? 1 : 0;
    auto &now = f[pos][last1][last2][same][have4][have8];
    if(!limit && ~now)
        return now;
    LL res = 0;
```

```
    int up = limit ? a[pos] : 9;
    int down = pos == len ? 1 : 0; //最高位只能从 1 开始
    for(int i = down; i <= up; i ++)
    {
        bool tmp = same || (last1 == i && last2 == i);//same 新值
        if(i == 4 && !have8) //没出现 8，才能写 4
            res += dfs(pos - 1, limit && i == up, i, last1, tmp, true, false);
        else if(i == 8 && !have4)//没出现 4，才能写 8
            res += dfs(pos - 1, limit && i == up, i, last1, tmp, false, true);
        else if(i != 4 && i != 8)
            res += dfs(pos - 1, limit && i == up, i, last1, tmp, have4, have8);
    }
    if(!limit)
        now = res;
    return res;
}

LL solve(LL x)
{
    if(x < 1e10) //不是手机号
        return 0;
    len = 0;
    while(x > 0)
    {
        a[ ++ len] = x % 10;
        x /= 10;
    }
    return dfs(len, true, 10, 10, false, false, false);
    // last 初值 10 为了不让跟后面数字相同
}

int main()
{
    memset(f, -1, sizeof f);
    LL l, r;
    cin >> l >> r;
    cout << solve(r) - solve(l - 1);
}
```