算法常用解题技巧

陈杉

(改编自赵耀老师的PPT)

算法常用技巧

- 空间换时间
- 二分搜索的妙用
- 规律类问题分析方法
- 大事化小
- 运算注意事项
- 对拍

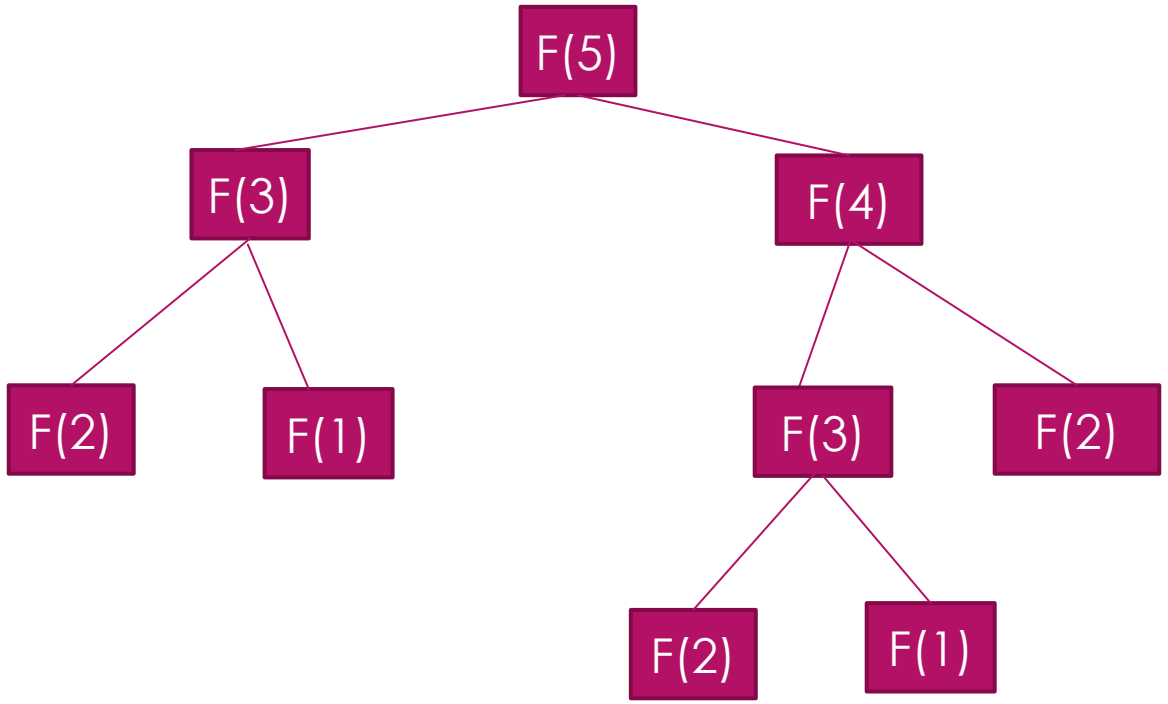
空间换时间

- **Fibonacci:**

F(1)=1，

F(2)=1，

F(n)=F(n-1)+F(n-2)（n>=3，n∈N*）



```
public static long fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

```
fibonacci(8)
   fibonacci(7)
      fibonacci(6)
         fibonacci(5)
            fibonacci(4)
               fibonacci(3)
                  fibonacci(2)
                     fibonacci(1)
                        return 1
                     fibonacci(0)
                        return 0
                     return 1
                  fibonacci(1)
                     return 1
                  return 2
               fibonacci(2)
                  fibonacci(1)
                     return 1
                  fibonacci(0)
                     return 0
                  return 1
               return 3
            fibonacci(3)
               fibonacci(2)
                  fibonacci(1)
                     return 1
                  fibonacci(0)
                     return 0
                  return 1
               fibonacci(1)
                  return 1
               return 2
            return 5
         fibonacci(4)
            fibonacci(3)
               fibonacci(2)
                .
                .
                .
```

过度冗余计算！

Memoization(记忆化)

记忆化是一种提高计算机程序执行速度的优化技术。通过存储大计算量函数的返回值，当这个结果再次被需要时将其从缓存提取，而不用再次计算来节省计算时间。记忆化是一种典型的在计算时间与空间之中取得平衡的方案。(wikipedia)

如何使用记忆化来避免过多的冗余计算?

Memoization Recursion

```
public class Fab {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        int memo[] = new int[n + 1];
        System.out.println(fibonacci(n, memo));
    }
    public static int fibonacci(int n, int memo[]) {
        if (n == 1) {
            return 1;
        }
        if (n == 2) {
            return 2;
        }
        if (memo[n] > 0) {
            return memo[n];
        }
        memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
        return memo[n];
    }
}
```

每个步骤的结果存储在memo数组中，每次再次调用函数时，我们直接从memo数组返回结果。

在memo数组的帮助下，我们得到了一个修复后的递归树，它的大小减少到n。

动态规划

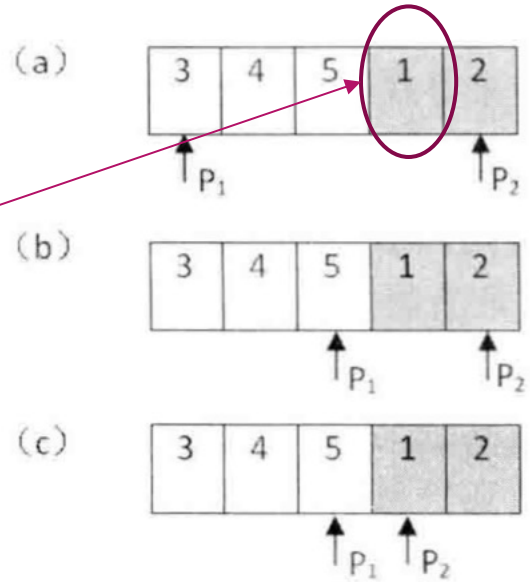
后续课程将学习这类算法

二分搜索的妙用

- 通常的应用：在递增或递减序列中找目标值
- 还可用于二分最优解：比如经典的割绳子问题 （**LeetCode 1891. 割绳子**），有n条绳子，长度不等，需要将所给的绳子能切割成k条长度均为L的绳子，求L最大能达到多少。绳子长度 L变长，那么得到的绳子的根数 k 不会变多，具有单调性。

单调序列二分搜索的扩展应用：
把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。例如，数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。

旋转之后的数组实际上可以划分为两个排序的子数组，而且前面子数组的元素都大于或者等于后面子数组的元素。最小的元素刚好是这两个子数组的分界线。



Mid > p1?
Mid < p2?
什么时候循环结束？

规律类问题的分析方法

- 可以先从小规模样例入手，或者先去掉一个限制简化问题，之后再去解决原问题

例题

- Lanran想要举办一场**最多有3个问题**的编程比赛。他有一个题目库，每个问题都有自己的难度等级。竞赛选题有一些要求：所选题目的难度不能相互被对方整除，而Lanran希望所选问题的难度之和尽可能大。请输出最大可能的难度和。

解题思路

根据题意，选题难度显然不能重复，可以得到难度递降序列a[]，例如：30 15 12 10 …

1. **3个最大的数都不能互相整除：**返回a[0]+a[1]+a[2]
2. **a[0]不能被a[1]整除，但a[0]或a[1]能被a[2]整除：**a[0] ≥ 2a[2] => a[0]+a[1] ≥ a[1]+a[2]+a[3]

   所以a[0]必选，类似可得a[1]也必选，然后遍历序列的其余数，直到找到不整除a[0]和a[1]的最大的数。
3. **a[0]能被a[1]整除：**
   i. **a[0]/a[1] ≥ 3** => 任意i ≥ 1, a[0]/a[i] ≥ 3 (a[1]和后面任何数字的组合都不可能超过a[0])

   所以a[0]必选，然后遍历序列的其余部分，以找到不整除a[0]且满足约束的最大的其他2个数。

   ii. **a[0]/a[1] = 2**

   当 a[0]/a[2] = 3, a[0]/a[3] = 5 时，观察到 a[0] < a[1]+a[2]+a[3]，例如：30 15 10 6 …

   当 a[0]/a[2] = 3, a[0]/a[3] < 6 时，观察到 a[0] > a[1]+a[2]+a[3]，例如：30 15 10 4 …

   没有观察到有效规律，但不难看出a[0]和a[1]必须选一个，所以：

   - 选择a[0]，然后遍历序列的其余部分，以找到不整除a[0]且满足约束的最大的其他至多2个数。
   - 选择a[1]，然后遍历序列的其余部分，以找到不整除a[1]且满足约束的最大的其他至多2个数。

大事化小

- 贪心、分治和动态规划都体现了将大的问题分解或转化成小问题的分析思路
- 这也是本学期学习的重点内容

运算注意事项

- 需要快速得到数组中某一连续段的和（乘积，异或），都可以用前缀和（类似）方法
- 乘法和加法，要考虑会不会可能有整数溢出；取模要考虑负数情况和除法情况；除法要考虑除数是否为0

前缀和、积、异或

- 前缀和

Sum(i,j) = sum[j]-sum[i-1]

- 前缀积

Product(i,j) = Product[j] / Product[i-1]

- 前缀异或

xor(i,j) = xor[j]^xor[i-1]

模运算

在算法题目中，有时候我们会遇到超过long(C++: long long)范围的数据，这个时候题目往往会要求我们输出答案对某个质数取模的结果。这个时候就涉及到模运算。

模运算与基本四则运算比较类似（除法除外）。

(a + b) % n = (a % n + b % n) % n

(a - b) % n = (a % n - b % n) % n

(a * b) % n = (a % n * b % n) % n

a ^ b % p = ((a % p)^b) % p

模运算常见错误

```
#include <iostream>
using namespace std;

using ll = long long;
const ll mod = 998244353;

int main() {

    ll a = 1e10, b = 1e10, c;
    c = a * b; //可能溢出
    c %= mod;
    // 正确写法：c = (a % mod) * (b * mod) % mod;

    return 0;
}
```

模运算常见错误

```
#include <iostream>
using namespace std;

using ll = long long;
const ll mod = 998244353;

int main() {

    ll a = 2, b = 9, c;
    c = (a - b) % mod;//可能为负
    // 正确写法: c = (a - b + mod) % mod;

    return 0;
}
```

模运算逆元

在模运算中是不能直接使用除法的。所以，我们引入乘法逆元的概念：

当p是质数，在mod p意义下，如果 a * a' = 1 (mod p) 或 a' * a = 1 (mod p) ，那么我们就说 a' 是 a 的逆元。当然，反过来， a 也是 a' 的逆元。

乘法逆元的性质：

存在唯一性

求解乘法逆元的方法：

费马小定理　扩展欧几里得

欧拉筛　　线性递推

模运算除法需要求逆

```
#include <iostream>
using namespace std;

using ll = long long;
const ll mod = 998244353;

int inv[20];

int main() {

    ll a = 10, b = 5, c;
    c = (a / b) % mod //模运算除法需要求除数的逆元inv[b]
    // 正确写法: c = a * inv[b] % mod;

    return 0;
}
```

对拍

- 为了验证某解题程序是否正确，另外写一个暴力求解该题目的程序，然后生成一些测试数据，看同样的数据，两个程序输出的结果是否相同，不同意味着被对拍的程序有问题。
- 鼓励大家自己写暴力程序、数据生成器、对拍脚本，自行发现算法问题

以下博客讲得挺详细的，可供大家参考：

https://blog.csdn.net/Njhemu/article/details/99539576

https://blog.csdn.net/weixin_30835649/article/details/98410133

调试

- 使用GDB打断点单步调试和监控变量
- 打印中间变量
- 小黄鸭调试法（解释自己的程序给小黄鸭或者其他同学听）

常数优化

- 卡常数，又称底层常数优化，特指在OI/ACM-ICPC等算法竞赛中针对程序基本操作进行的底层优化，一般在对程序性能要求较为严苛的题目或是在算法已经达到理论最优时间复杂度时使用。算法课不会出现必须卡常的题目，但以防万一还是需要知道一些比较有常用的优化方法。
- 用cin与cout记得关闭同步 std::ios::sync_with_stdio(false)
- 多用inline函数
- 在熟练的前提下用指针，会比用数组少一次寻址

STL库

- STL 即标准模板库（Standard Template Library），是 C++ 标准库的一部分，里面包含了一些模板化的通用的数据结构和算法
- 常用STL容器：vector：变长数组，(unordered_)set：集合，(unordered_)map：映射，priority_queue：优先队列（堆）…
- 常用STL算法：find：顺序查找，reverse：反转数组字符串；unique：数组去重；binary_search(lower_bound)：二分查找；next_permutation：将当前排列更改为 全排列中的下一个排列…
- 只要题目没有说明不能用，用STL库可以大大提高编码效率和减少错误概率。

参考资料

- https://oi-wiki.org/ 介绍算法竞赛中会用到的算法，工具等
- https://en.cppreference.com/ C++语法，标准库参考资料