算法入门与数学基础

Kareninahui

南京航空航天大学

2024-10-18

目录

- 什么是算法
- 求最小公约数的几种算法
- 评判算法的优劣
- 数的表示

什么是算法

定义

算法就是解决问题的一系列步骤。在计算机中，算法就是一组指令，每条指令告诉计算机该做什么。

基本特性

算法有几个重要的特性：

- **输入**：算法可以有零个或多个输入。
- **输出**：算法至少会有一个输出结果。
- **有穷性**：算法在执行有限的步骤后会结束，不会无限循环，每一步都能在合理时间内完成。
- **确定性**：算法的每一步都是明确的，不会有歧义。
- **可行性**：算法的每一步都是可执行的，也就是说，每一步都能通过有限次数的操作完成。

求最大公约数的几种算法

题目描述

定义两个正整数的最大公约数 $\gcd(a, b)$ 为最大的正整数 d，使得 d 可以同时整除 a 和 b。

例如，$\gcd(9, 12) = 3$，因为 $9 \div 3$ 和 $12 \div 3$ 的余数是 0，而无法找到一个比 3 更大的正整数满足要求。

现在给定两个正整数 a, b，要求出 $\gcd(a, b)$。

（1）直接枚举法

也就是从 $\min(a, b)$ 枚举到 1 直到找到第一个即是 a 的约数也是 b 的约数的数字为止

```
int gcd(int a, int b) {
    if (a < b) swap(a, b);
    for (int i = b; i > 0; i--) {
        if (a % i == 0 && b % i == 0) return i;
    }
}
```

（2）更相减损法

更相减损术是中国古代数学中的一种求最大公约数（GCD）的方法，它是基于“减而治之”的策略。

这种方法在《九章算术》中有记载，其基本原理是：两个正整数，一个减小，一个保持不变，用较大的数去减较小的数，然后再用较小的数与所得的差求最大公约数，

如此循环，直到两数相等，那么相等的数就是这两个数的最大公约数。

更相减损法的证明

更相减损法

```
int gcd(int a, int b) {

    while(a!=b){
        if(a>b) a=a-b;
        else b=b-a;
    }
    return a;
}
```

（3）欧几里得算法

欧几里得算法，也称为辗转相除法，是用来计算两个非负整数 a 和 b 的最大公约数（记作 $\gcd(a,b)$）的一种方法。其基本步骤是：用较大的数除以较小的数，然后再用除数除以上一次的余数，如此重复，直到余数为 0 时，最后的除数就是这两个数的最大公约数。

欧几里得算法证明

迭代写法

```
int gcd(int a, int b) {
    //若 a<b 第一次辗转相处刚好把a和b互换
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

递归写法:

```cpp
int gcd(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}
```

习题

- 课堂例题
- 最大公约数
- 最大公约数和最小公倍数
- 三角函数

评判算法的优劣

算法复杂度

同一问题可用不同算法解决，而一个算法的质量优劣将影响到算法乃至程序的效率。算法分析的目的在于选择合适算法和改进算法。

算法复杂度分为时间复杂度和空间复杂度。其作用: 时间复杂度是指执行算法所需要的计算工作量;而空间复杂度是指执行这个算法所需要的内存空间。

时间复杂度

定义

计算机科学中，算法的时间复杂度是一个函数，它定量描述了该算法的运行时间。这是一个关于代表算法输入值的字符串的长度的函数。时间复杂度常用 O 符号表述，记作

$$T(n) = O(f(n))$$

这个函数的低阶项和首项系数常常忽略，只保留最高阶项。

计算方式

1. 确定算法中的基本操作：基本操作通常是算法中执行次数最多的部分，通常是赋值、比较、算术运算等。
2. 找出基本操作的执行次数：通过分析代码结构（循环、递归等），找出基本操作执行的次数。
3. 忽略低阶项和常数项：只关注输入规模 n 变化时的增长情况，忽略低阶项和常数因子。
4. 用大 O 表示法描述增长率：使用 O 表示法来描述执行次数与输入规模之间的关系。

分析算法结构

考虑以下几种常见的算法结构:

顺序结构：基本操作的执行次数是累加的。
循环结构：基本操作的执行次数通常是循环次数乘以每次循环中的操作次数。
条件结构：条件语句本身不增加时间复杂度，但是条件内的操作需要考虑。
递归结构：相较于前三种，递归的时间复杂度分析稍复杂，递归的时间复杂度通常通过递归树或主定理来分析。

递归结构的时间复杂度分析-主定理

主定理（Master Theorem）用于分析分治算法的时间复杂度，适用于形如 $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ 的递归关系式，其中：

- $T(n)$：问题规模为 n 时的运行时间
- a：递归调用次数
- n/b：每次递归输入规模，$b > 1$
- $f(n)$：当前层的计算工作

主定理解决以下形式的递归关系：
$T(n) = aT\left(\frac{n}{b}\right) + f(n)$
其中 $a \geq 1$ 和 $b > 1$ 为常数，$f(n)$ 为给定函数。

计算基本操作的执行次数

以下是一些基本操作：

加减乘除：一般来说认为是 $O(1)$。

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

int main() {
    int a = 3, b = 5;
    int c = a + b, d = a * b, e = a - b, f = b / a;
    cout << c << " " << d << " " << e << " " << f << endl;
    return 0;
}
```

线性搜索：在数组中查找一个元素，需要遍历整个数组，时间复杂度为 $O(n)$。

```cpp
int search(const int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) return i;
    }
    return -1;
}
```

二分搜索、快速幂：每次比较将搜索范围减半，时间复杂度为 $O(\log n)$。

```
int binarySearch(const int arr[], int left, int right, int target) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;
        }
        if (arr[mid] < target) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }
    return -1;
}
```

插入排序、冒泡排序：对于每个元素，可能需要遍历之前排序好的所有元素，时间复杂度为 $O(n^2)$。

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

找出最高阶项

对于以下表达式：
$3n^2 + 2n + 1$
最高阶项是 $3n^2$，因此我们可以忽略 $2n$ 和 1。

使用大 O 记号表示

根据上面的例子，我们可以将时间复杂度表示为：$O(n^2)$。

分析三种求最大公约数的算法的算法复杂度

方法一：枚举法

```
int gcd(int a, int b) {
    if (a < b) swap(a, b);
    for (int i = b; i > 0; i--) {
        if (a % i == 0 && b % i == 0) return i;
    }
}
```

该方法是循环结构，有一层 for 循环，可知时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

方法二：更相减损法

```
int gcd(int a, int b) {
    while(a!=b){
        if(a>b) a=a-b;
        else b=b-a;
    }
    return a;
}
```

可以很容易地看出其时间复杂度与 a/b 的大小有关，当 a/b 较小时其时间复杂度约为 $O(\log n)$；当其极大时（a 远大于 b ）时，由于时间与相减的次数挂钩，其时间复杂度会退化为 $O(n)$。空间复杂度为 $O(1)$。

方式三 辗转相除法

```
iint gcd(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}
```

欧几里得算法（也称为辗转相除法）的时间复杂度是 $O(\log(n))$，其中 a 和 b 是输入的两个非负整数。

欧几里得算法的时间复杂度的证明

数的表示

在数学中，我们常用十进制数，而在计算机中会接触到更多的进制。常见的进制有：

二进制（Binary）：基数为2，只使用0和1。例如，二进制的1010代表十进制的10。
八进制（Octal）：基数为8，使用0到7。例如，八进制的12代表十进制的10。
十进制（Decimal）：基数为10，使用0到9。例如，十进制的10就是10。
十六进制（Hexadecimal）：基数为16，使用0到9和A到F，其中A到F代表十进制的10到15。例如，十六进制的A代表十进制的10。

例题：10 进制转 x 进制

题目描述

给定一个十进制整数 n 和一个小整数 x。将整数 n 转为 x 进制。对于超过十进制的数码，用 `A`，`B` ... 表示。

输入格式

第一行一个整数 n;
第二行一个整数 x。

输出格式

输出仅包含一个整数，表示答案。

代码实现

```
string dict = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
string ten_to_x(int n, int x)  //十进制转 x 进制函数。
{
    string ans = "";
    while (n != 0) //模拟短除法。
    {
        ans += dict[n % x];
        n /= x;
    }
    string t = "";  //倒取余数。
    for (int i = ans.length()-1; i >= 0; i--) t += ans[i];
    return t;
}
```

易知时间复杂度为 $O(\log n)$，空间复杂度为 $O(1)$。

相关练习

十进制转x进制

x进制转十进制

x进制转y进制

进制习题

位运算

位运算是指按照二进制进行的运算，主要用于对整数的二进制位进行操作。在 C/C++ 语言中，位运算符提供了对整型数据进行高效操作的能力。

在 C 语言中，提供了 6 种位运算符，它们分别是按位与 `&`，按位或 `|`，按位异或 `^`，按位取反 `~`，左移 `<<` 和右移 `<<`。

这些运算符只能用整型操作数，也就是说只能用于带符号和不带符号的 `short`，`int`，`long`，`char` 类型。

按位运算

- 按位与 `&`：对两个操作数的每一位进行与操作，只有当两个对应的二进制位都为1时，结果才为1。
- 按位或 `|`：对两个操作数的每一位进行或操作，只要两个对应的二进制位有一个为1，结果就为1。
- 按位异或 `^`：对两个操作数的每一位进行异或操作，当两个对应的二进制位不同时，结果为1。
- 按位取反 `~`：对操作数的每一位进行取反操作，即1变为0，0变为1。

左移和右移

- 左移 `<<`：将操作数的所有二进制位向左移动指定的位数，右边补0。
- 右移 `>>`：将操作数的所有二进制位向右移动指定的位数，左边补符号位（对于有符号数）或0（对于无符号数）。

不难看出，左移n位相当于乘以2^n，右移n位相当于除以2^n。