算法复杂度的计算

1. O(大O符号)：上界

定义：若存在两个正的常数 c 和 n0，对于任意 $n \geq n0$，都有 $T(n) \leq cf(n)$,则称 $T(n) = O(f(n))$ (或称算法在 $O(f(n))$ 中)。

大 O 符号用来描述增长率的上限,表示 T(n)的增长最多像 f(n)增长的那样快,也就是说, 当输入规模为 n时, 算法消耗时间的最大值,这个上限的阶越低, 结果就越有价值。上界是对算法效率的一种承诺。

2. Θ ：对于存在大于0的常数c1、c2和非负的整数n0，以及足够大的n，对于所有的 $n \geq n0$ 来说，有 $c1g(n) <= f(n) <= c2g(n)$ 。表示紧确界。
3. Ω：$f(n) = \Omega(g(n))$，当且仅当存在正的常数c和n0，使得对于所有 $n >= n0$，有 $f(n) >= cg(n)$ 。下界

递归与分治

1. Ackerman 函数：增长速度极快

   其中 ackerman 函数就是用递归定义的。

2. 全排列问题

   n 个数集合的全排列问题可以分解为n 个n-1 个数的全排列问题（一个全排列的第一个数只有 n 中可能性）

3. 汉若塔（hanoi）问题

   对于n 个盘子的问题，你可以先把 n-1 个盘子移到另一个柱子上，然后把最底下的一盘子移到空闲的盘子上，在执行一次把 n-1 个盘子移到非空闲的柱子上，就完成了任务。

4. 霍纳规则：

   既然是最优的算法规则，我们可以根据它的计算过程来分析为何它会简化原来的计算过程。一般多项式的表达式如下：

   $P(n) = \sum_{k=0}^{n} a_k x_k = a_0 + x(a_1 + x(a_2 + \ldots + x(a_n - 1 + xa_n)))$

   一般的计算方法就是 n 次循环，然后每个内层进行计算 k 次幂(计算机采取多次相乘计算)

   ```
   int sum = 0;
   for(int i=0;i<n;i++)
     sum += sum + a[k]*x[k];
   ```

   时间复杂度是 $\Theta(n^2)$

   而采取霍纳规则计算可以得到 $\Theta(n)$ 复杂度。

```
int horner(int *a, int n,int x)
{
    int ax = a[n] * x + a[n - 1];
    for(int i = n - 2; i >= 0; i--)
    {
        ax = ax * x + a[i];
    }
    return ax;
}
```

5. 逆序对数目

   在一个数组 A 中，如果存在 $i < j, A[i] > A[j]$ ，那么称 $A[i]$ 和 $A[j]$ 是一对逆序对。

   现在给定一个任意排序的数组 A，如何计算这个数组所有的逆序对数，要求复杂度是 $\Theta(nlog(n))$。

   可以想象插入排序的过程，就是把相邻的逆序对进行调整，我们想到可以用排序的方法对逆序对进行统计，其中合并排序这样的算法就可以达到要求的复杂度。

递归容易用数学归纳法进行证明，但是在规模太大的时候运行的效率太低。

主要的思想就是 **分解-> 解决-> 合并**。分治算法的模式都是一样的，我们可以从数学上进行进行抽象

$$T(n) = \begin{cases} \Theta(1) \, if \, n < c \\ aT(\frac{n}{b}) + D(n) + C(n) \end{cases}$$

具体求解递归的方法

- 代入法：猜测解的形式，通过数学归纳进行证明。假设 n-1 的情况满足猜测的解，证明n 的情况也能得到同样的结果。这里对于边界的情况没有数学上的证明那么严格，如果只是有限的的n =1,2..不成立的话，只需要证明对于 $n > n_0$ 之后假设成立即可。
- 递归树法：转换为一棵树，代入法需要一个好的猜测，所以很多时候很难找到。

  每次拆分成子问题都会有一个代价（比如说合并排序中，n 规模的问题拆分成 n/2 的问题，需要一次合并的过程，合并就是一次拆分代价），我们把树的一个点值设为拆分一定规模问题的代价，把整个拆分的递归树画出来，先对每一层的节点的代价求和，然后要知道树的深度，在对整棵树的代价求和。
- 主方法

$$\begin{gathered} T(n) = aT(\frac{n}{b}) + f(n) \\ \text{其中} b > 1, \text{问题的规模才能减小} \\ \text{可以得到公式：} \\ \text{需要比较两个函数的增长率：} f(n) \text{和} \, n^{log_b a} \\ 1.\text{如果} \, f(n) > n^{log_b a}, \text{那么} \\ T(n) = \Theta(f(n)) \\ 2.\text{如果} \, f(n) < n^{log_b a}, \text{那么} \\ T(n) = \Theta(n^{log_b^a}) \\ 3.\text{如果} \, f(n) == n^{log_b a}, \text{那么} \\ T(n) = \Theta(n^{log_b^a} log(n)) \end{gathered}$$