递归树：如何借助树来求解递归算法的时间复杂度

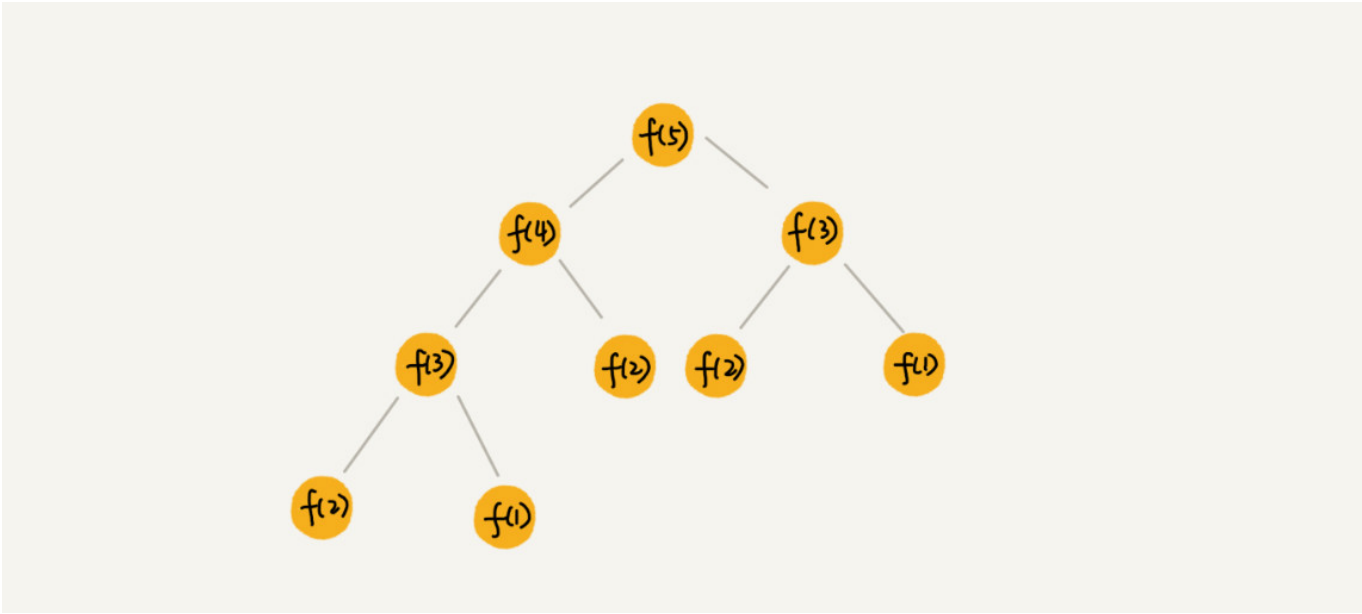
除了使用递推公式来分析算法的时间复杂度，另外还有一种方法，借助递归树来分析递归算法的时间复杂度。

递归树与时间复杂度分析

递归的思想就是将大问题分解成小问题来求解，将小问题分解成小小问题来求解。

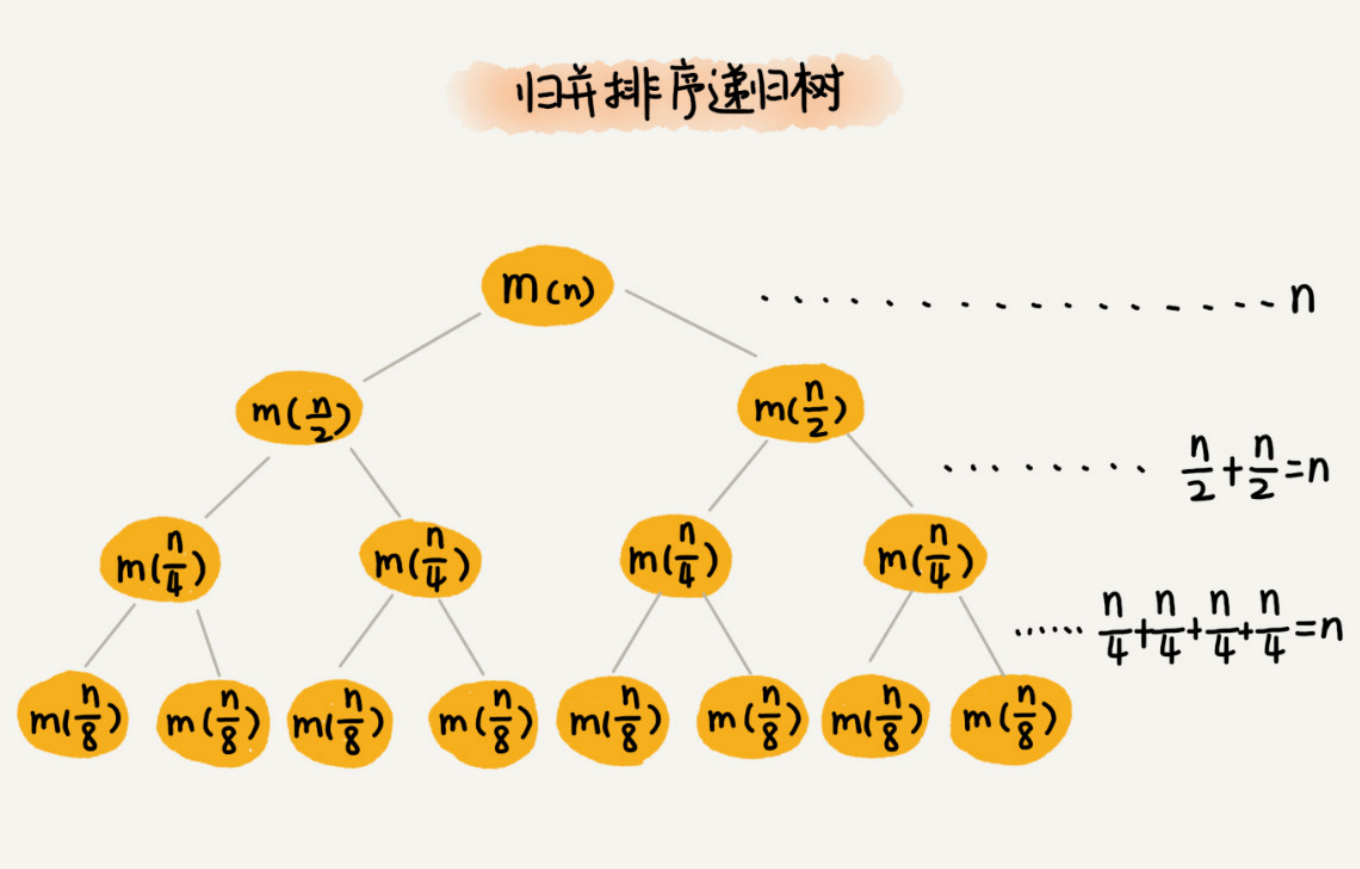
将这个一层一层的过程画成图就是一颗树，这棵树起一个名字就是递归树。

如图所示是一颗斐波那契数列的递归树：



如何使用递归树来求解时间复杂度

归并排序算法还记得吗？归并排序每次都会将数据规模一分为二，将归并排序画成递归树，如图：



以为每次分解的过程都是一分为二，所以代价很低，我们把时间上消耗记做常量1，归并排序中比较耗时的是归并操作，也就是将两个子数组合并成为一个大数组，但从上面的图中我们发现每一层的归并操作消耗的时间是一样的，记为n。

因此，此时只需要知道树的高度h就可以知道总的时间复杂度O(n * h);

从归并排序的原理来看归并排序树是一颗满二叉树。满二叉树的高度大约是logn（以2为底）。所以最终的复杂度就是O(n*logn);

实战一： 分析快速排序的时间复杂度

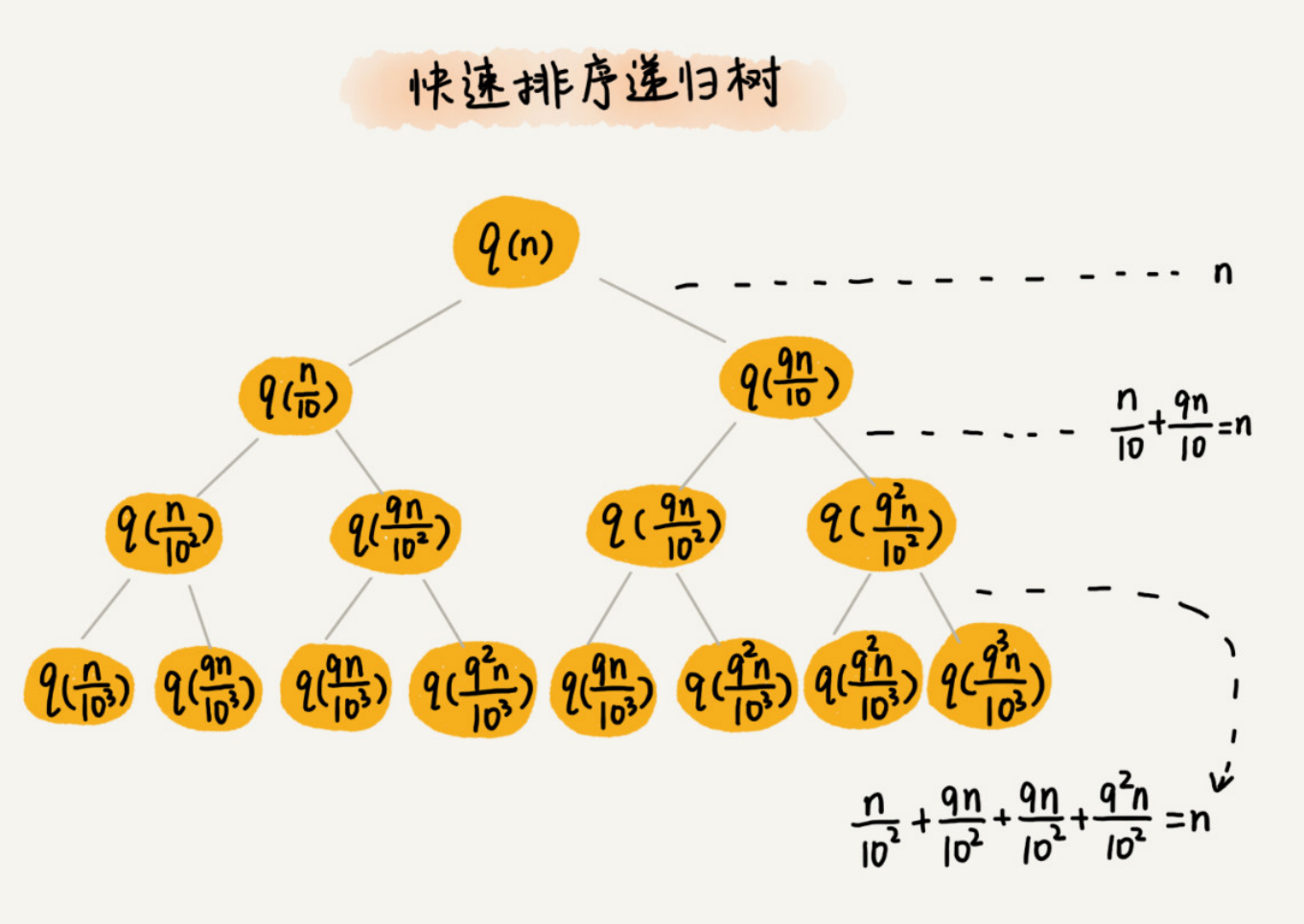
先来回忆一下使用递推公式的分析方法。

快速排序在最好的情况下，每次分区都能一分为二，这个时候用递推公式T(n) = 2T(n/2) + n; 很容易就可以推导出时间复杂度O(n*logn)。但是我们并不可能每次分区都这么幸运，正好一分为二。

假设每次分区的大小比例是1:k；当k = 9时，如果用地推公式的方法来求解时间复杂度的话，递推公式就写成T(n) = T(n/10) + T(9n/10) + n;

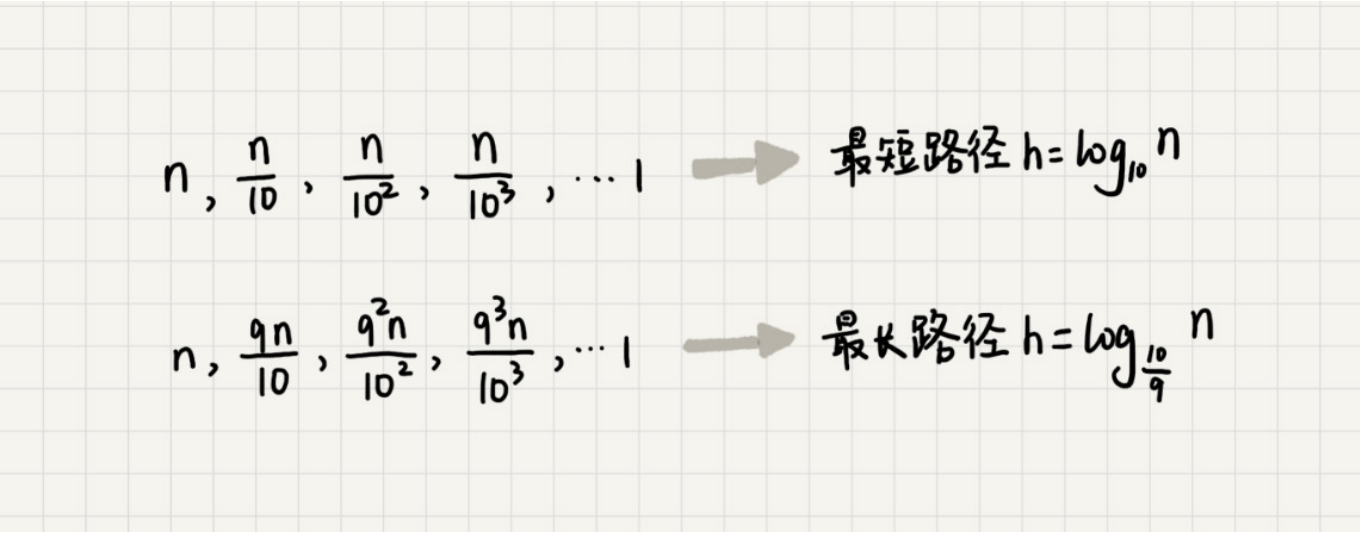
如果使用递归树来分析快速排序的平均情况时间复杂度，是不是比较简单呢?

当k = 9时，画成递归树如图所示：



快速排序过程中，每个分区都要遍历待分区的所有数据，所以每一层分区操作所遍历的数据的个数之和就是n，我们只要求出递归树的高度h，这个快排过程的数据个数就是h * n，也就是说，时间复杂度就是O(h * n).

但是快速排序结束的条件是待排序的小区间大小为1，也就是说叶子节点里的数据规模是1.从根节点n到叶子结点1，递归树中最短的路径每次都乘以1/10，最长的一个路径每次都乘以9/10.通过计算，我们可以得到最短路径是logn （以10为底）；最长路径是logn（以10/9为底）。



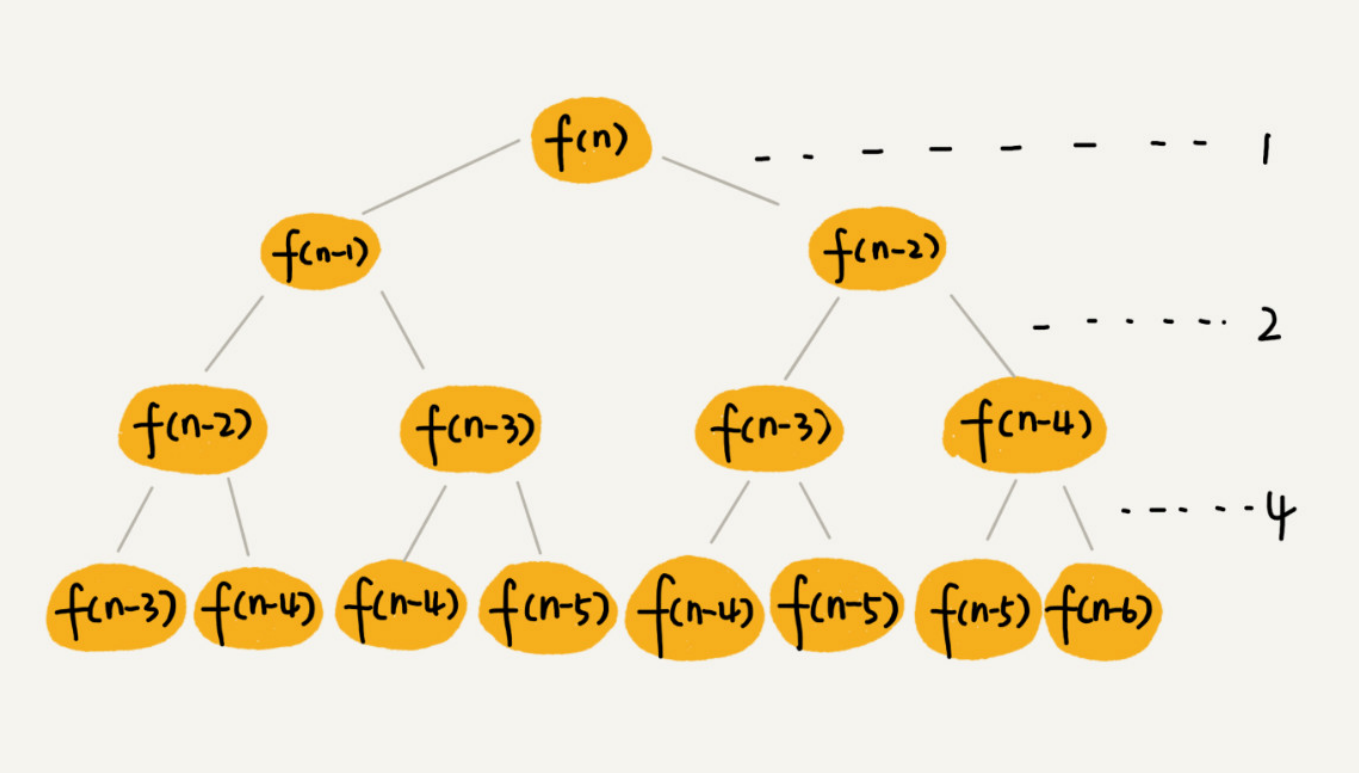
遍历数据总数介于n * 最长路径和 n * 最短路径之间。根据大O表示法，对数复杂度的底数不管是多少，我们统一写成logn，所以快排时间复杂度仍然是O(nlogn).

实战二：分析斐波那契数列的时间复杂度

斐波那契数列代码：

```
int f(int n) {
    if (n == 1) return 1;
    if (n == 2) return 2;
    return f(n-1) + f(n-2);
}
```

将上面代码画成递归树为：



根据步长，如果每次都是-1的话最长路径就是n；每次数据规模都是-2的话，最短路径就是n/2。

每次分解之后的合并操作只需要一次加法运算，我们把这次加法运算的时间消耗记做1.所以从上往下第一层总时间消耗是1，第二层是2，第三层是2^2。以此类推是2^k - 1;那整个算法的消耗时间是每一层的消耗之和。

如果路径长度都为n，那这个总和就是2^n-1； 如果路径长度都为n/2，那这个总和就是2^(n/2)-1;

这算法的时间复杂度就介于O(2^n)和O(2^(n/2))之间。

实战三：分析全排列的时间复杂度

将n个数据的所有排列都找出来。

如何实现呢？ 如果我们确定了最后一位数据，那就变成了求剩下n-1个数据的排列问题。而最后一位数据可以是n个数据中任意一个，因此他的取值就有n种情况。所以n个数据的排列问题，就变成了n个n-1个数据的排列问题。

递推公式是：

假设数组中存储的是1，2， 3...n。
f(1,2,...n) = {最后一位是1，f(n-1)} + {最后一位是2，f(n-1)} +...+{最后一位是n，f(n-1)}。

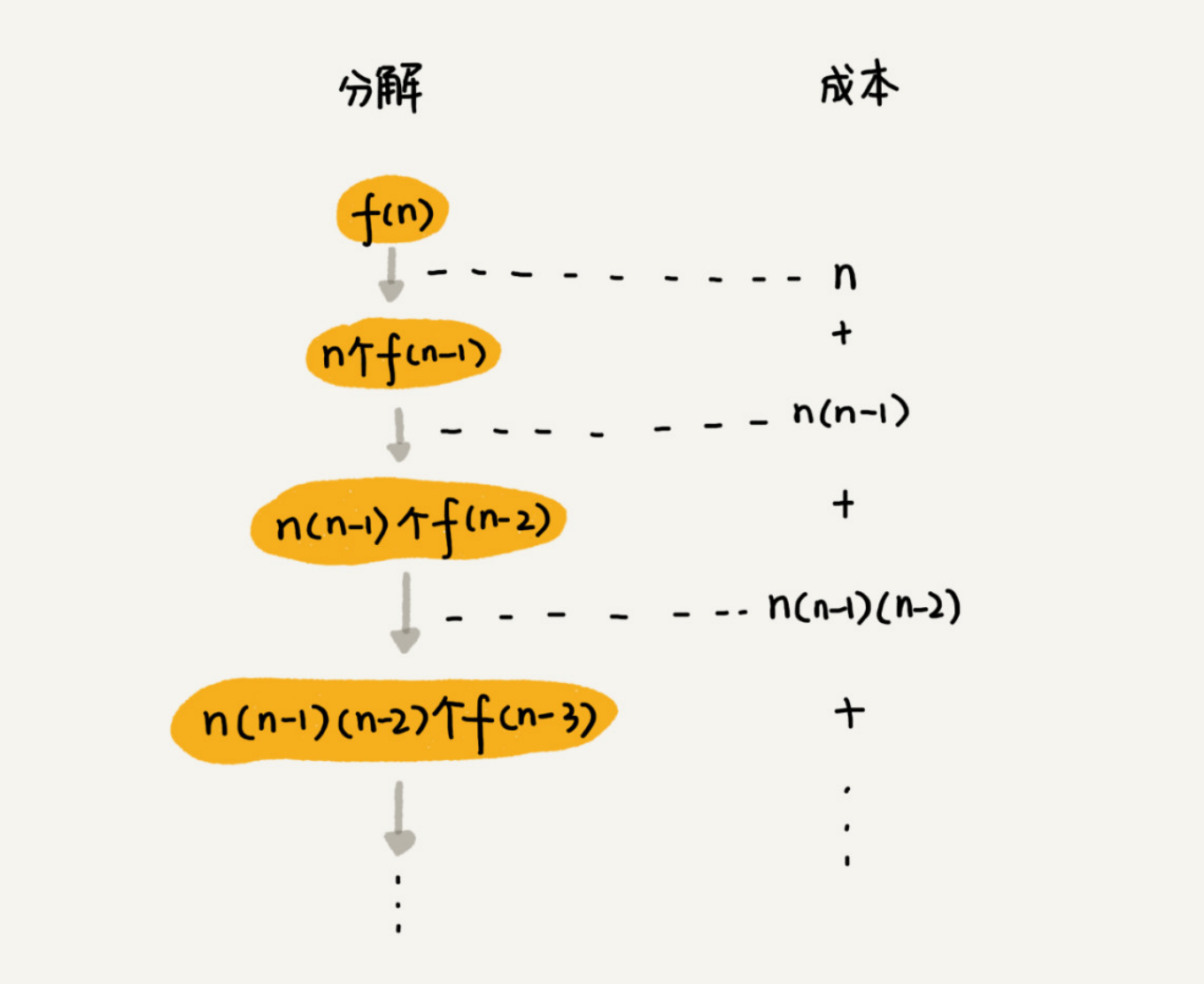
写成代码为：

```
/**
 * int[] a = a = {1,2,3,4};printPermutations(a,4,4)
 * k表示要处理的子数据的数据个数
 */
public void printPermutations(int[] data,int n,int k){
    if(k == 1) {
        for(int i = 0; i < n; ++i){
            System.out.print(data[i] + " ");
        }
        System.out.println();
    }

    for(int i = 0; i < k; ++i){
        int tmp = data[i];
        data[i] = data[k - 1];
        data[k - 1] = tmp;

        printPermutations(data,n,k-1);

        tmp = data[i];
        data[i] = data[k - 1];
        data[k - 1] = tmp;
    }
}
```



第一层分解有n次交换操作，第二层有n个节点，每个节点分解需要n-1次交换，所以第二层总的交换次数是n*(n-1)，第三层有n*(n-1)个节点，每个节点分解需要n-2次操作，第三层总的交换次数是n*(n-1)*(n-2)*。 *一次类推，第k层的总的交换次数是*n(n-1)*(n-2)*...*(n-k+1)。

最后每一层的交换次数之和就是总的交换次数。整体的求和比较复杂，但是整体最后求和肯定小 O(n*n!),也一定大于O(n!)。虽然我们没法知道非常精确的时间复杂度，但是这样一个范围已经让我们知道全排列的时间复杂度是非常高的了。

内容小结

有些代码比较适合，用递推公式来分析，比如归并排序的时间复杂度，快速排序的最好情况时间复杂度，有些比较适合采用递归树来分析，比如快速排序的平均时间复杂度。而有些可能两个都不怎么适合，比如二叉树的前中后序遍历。

课后思考

1 个细胞的生命周期是 3 小时，1 小时分裂一次。求 n 小时后，容器内有多少细胞？请你用已经学过的递归时间复杂度的分析方法，分析一下这个递归问题的时间复杂度。

假设细胞分离之后再死亡， n从第0个小时开始计算： n = 0: f(n) = 1; n = 1: f(1) = 2 * f(0); n = 2: f(2) = 2 * f(1); n = 3: f(3) = 2 * f(2) - f(0); n = 4: f(4) = 2 * f(3) - f(1); 以此类推；第k个小时的时候： n = k: f(k) = 2 * f(k - 1) - f(k - 3);