# 第九讲

*Algorithm Analysis*

薛浩

2023 年 4 月 20 日

www.stickmind.com

# 今日话题

**话题 1：编程基础** 初学编程的新手，一般应该熟练使用函数和库处理字符串相关的编程任务。

**话题 2：抽象数据类型的使用** 在尝试实现抽象数据类型之前，应该先熟练使用这些工具解决问题。

**话题 3：递归和算法分析** 递归是一种强有力的思想，一旦掌握就可以解决很多看起来非常难的问题。

**话题 4：类和内存管理** 使用 C++ 实现数据抽象之前，应先学习 C++ 的内存机制。

**话题 5：常见数据结构和算法** 在熟练使用抽象数据类型解决常见问题之后，学习如何实现它们是一件很自然的事情。

# 话题 3：递归和算法分析

递归是一种强有力的思想，一旦掌握就可以解决很多看起来非常难的问题。

- 递归过程
- 算法分析
- 递归回溯
- 排序算法

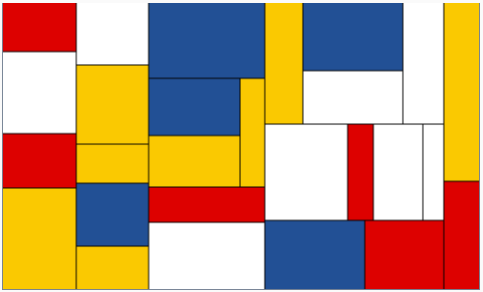


Figure 1: 递归艺术

# 如何分析程序的效率?

# 目录

# 复习：递归信任

# 递归信任 Recursive Leap of Faith

解决递归问题必须要保持整体观，你要相信这个过程是有效的。递归调用的内部操作与其他方法调用没有本质不同。

不要尝试追踪递归过程的细节！只有当你对这个过程有足够的信心，编写递归程序才会变得很自然。

在编写递归程序时，重要的是要相信！只要参数定义了一个更简单的子问题，任何递归调用都会返回正确的答案。

这种心理上的策略可以称为**递归信任**（Recursive Leap of Faith）。

# 识别递归问题

**问题**

求一个规模为 $N$ 的问题

**递归求解**

规模为 $N-1$ 的问题是否为同样的形式?

# 递归范式

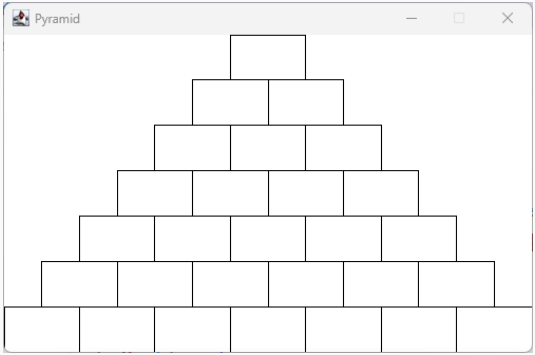
寻找递归解决方案主要是要弄清楚如何分解问题，使其符合递归范式。

```
if (问题最简单的形式) {
    无需递归，直接处理并返回结果
} else {
    将问题分解为一个或多个形式相同的子问题
    调用同样的函数或方法解决每一个子问题
    整合所有子问题的处理结果，并返回最终结果
}
```

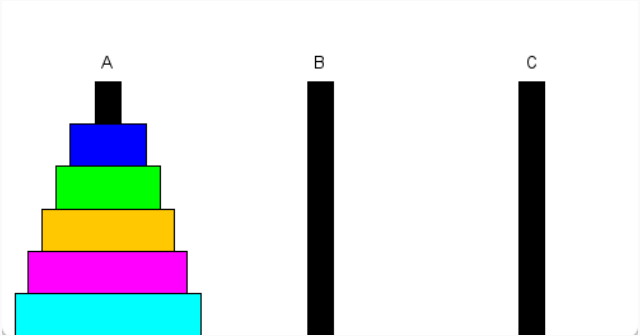
你必须做两件事：

- 识别无需递归处理的**最简单形式**（base case）
- 寻找一个**递归分解**（recursive decomposition）策略，可以将问题分解为相同形式、略微简单的子问题

# 金字塔 Pyramid



# 汉诺塔 Hanoi



# 分治策略

# 练习：findInVector

处理一系列存储在 Vector 中的数值时，常见的操作是寻找某个特定值。

尝试实现以下函数：

```
bool findInVector(const Vector<int>& vec, int key);
```

该函数查找一个整型向量 vec，判断其是否包含匹配 key 的元素。如果有匹配的元素，返回 true；否则，返回 false。

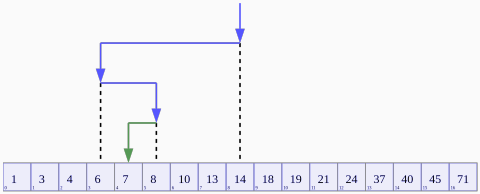
# 算法策略

**线性查找算法**（linear-search algorithm）必须依次检查每个元素，直到发现匹配的元素或者遍历完所有的元素为止。

```
procedure 线性查找 (vec, key)
    for vec 中每个 item do
        if item 等于 key then
            return 真
        end if
    end for
    return 假
end procedure
```

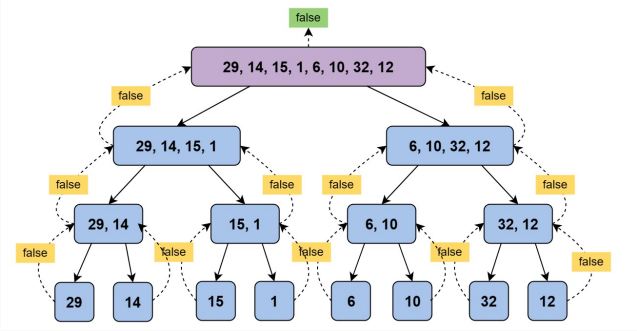
# 算法策略

**二分查找算法**（binary-search algorithm）是一种在**有序序列**中查找某一特定元素的搜索算法。每次利用中间元素将序列一分为二，从而缩小搜索范围，所以也称作折半搜索算法。

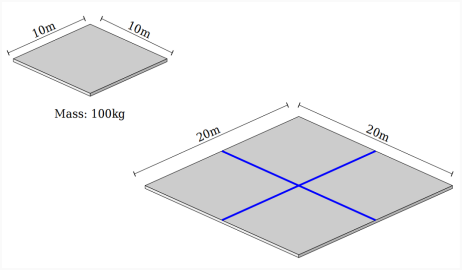


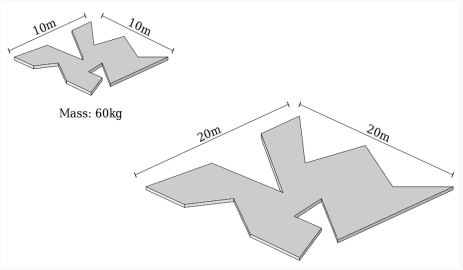
# 算法策略

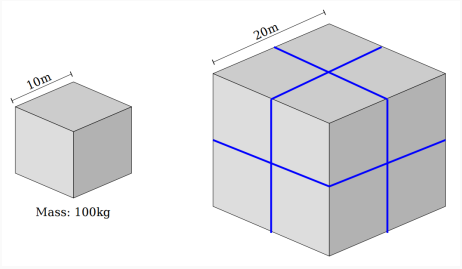
**分治策略**（divide-and-conquer algorithm）递归地将问题分为多个相同形式的子问题，直到最后的子问题可以简单地直接求解。二分查找算法是分治策略的典型实现，但缺点是序列必须要排序。

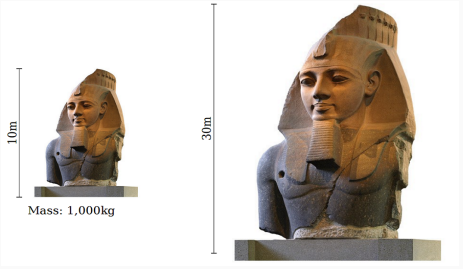


# 大 O 表示


10m
10m
Mass: 100kg
20m
20m


10m
10m
Mass: 60kg
20m
20m


10m
Mass: 100kg
20m


10m
Mass: 1,000kg
30m

*没有具体的公式的情况下，可以根据数量增长的速率，预估增长后的值。*

# 评估程序的效率

方法 1：作业 1 中，我们通过秒表计时来判断程序执行的快慢。

方法 2：如果每次操作都需要固定的时间，那么统计整体操作的次数也可以评估程序的效率。

```
double averageOf(const Vector<int> &vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }
    return total / vec.size();
}
```

# 大 O 表示 Big-O Notation

**大 O 表示法**（Big-O Notation）是表示时间复杂度最常见的方式，由德国数学家 Paul Bachmann 于 1892 年创立。

大 O 表示法由字母 $O$ 和一个公式组成，该公式对函数的运行时间进行了定性评估，传统上表示为 $N$。例如，可以用 $O(N)$ 表示线性搜索的时间复杂度。

大 O 表示法旨在提供定性评估，括号内的公式要求尽可能简单，通常会作以下简化：

- 消除随 N 变大后，对运行时间影响不显著的项
- 消除任何常量系数

例如，$O(\frac{Nx(N-1)}{2})$ 简化为 $O(N^2)$

# 算法分析

# 练习：printStar

```
void printStar(int n) {
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < m; col++) {
            cout << "*";
        }
        cout << endl
    }
}
```

# 练习：containsAlpha

```
bool containsAlpha(const string &s) {
    for (int i = 0; i < s.length(); i++) {
        if (isalpha(s[i])) {
            return true;
        }
    }
    return false;
}
```

# 练习：printStar

```
void printStar(int n) {
    for (int row = 0; row < n; row++) {
        if (row == 0 || row == n - 1) {
            for (int col = 0; col < n; col++) {
                cout << '*';
            }
            cout << endl;
        } else {
            cout << '*' << endl;
        }
    }
}
```

# 练习：avareed

```
void avareed(int n) {
    for (int i = 0; i < n; i++) {
        if (i >= n / 4 && i < 3 * n / 4) {
            for (int j = 0; j < n; j++) {
                cout << '*';
            }
        } else {
            cout << '?';
        }
    }
}
```

# 练习：findInVector

```
bool findInVector(const Vector<int>& vec, int key) {
    if (vec.size() == 1) {
        return (key == vec[0]) ? true : false;
    } else {
        if (vec[0] == key) {
            return true;
        } else {
            Vector<int> rest = vec.subList(1);
            return findInVector(rest, key);
        }
    }
}
```

# 练习：findInVectorDivide

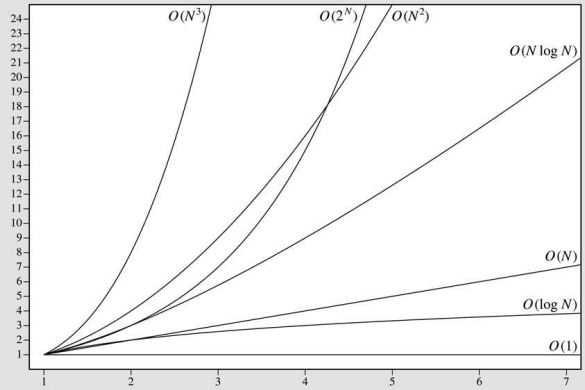
```
bool findInVectorDivide(const Vector<int>& vec, int key) {
    if (vec.size() == 1) {
        return (key == vec[0]) ? true : false;
    }

    int mid = vec.size() / 2;
    Vector<int> firstPart = vec.subList(0, mid);
    Vector<int> secondPart = vec.subList(mid);

    bool result1 = findInVectorDivide(firstPart, key);
    bool result2 = findInVectorDivide(secondPart, key);
    return (result1 || result2) ? true : false;
}
```

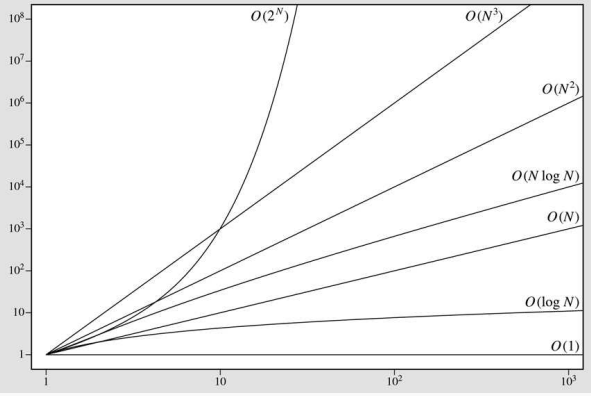
# 标准算法复杂度

**FIGURE 10-7** Growth characteristics of the standard complexity classes: linear plot



# 标准算法复杂度

**FIGURE 10-8** Growth characteristics of the standard complexity classes: logarithmic plot



# 如何分析程序的效率?

# 问题?