# 算法导论

> **Tip**
> 获取本讲代码: https://e1psycongroo.github.io/NekoBytes-TheMissing/_site/resources/Lecture4/Codes/
>
> 一个可视化 C 语言语句的小工具, 请利用工具高效学习

## 基本语法回顾——以 2048 为例

> **Tip**
> 让我们先从屎山出发
>
> 过早的优化是万恶之源
>
> 先实现后完美

```c
/* Direction 为自定义 enum 类型，用于表示方向
 * 你可以简单将其理解为
 * Direction dir; <=> int dir;
 * #define UP 0
 * #define DOWN 1
 * #define LEFT 2
 * #define RIGHT 3
 * #define NONE 4
 */
enum Direction{UP, DOWN, LEFT, RIGHT, NONE};
// board 用于结构化存储 2048 的数据
#define BOARD_SIZE 4
int board[BOARD_SIZE][BOARD_SIZE];

/* run the game */
void run_game(void);
/* initialize the board */
void initialize_board(void);
/* generate randon number at board */
void generate_number(void);
/* get the direciton from keyboard input */
enum Direction choose_direction(void);
/* print the game board */
void print_board(void);

/* bool 类型由 stdbool.h 提供
 * stdbool.h 还提供了 true 和 false 两种布尔值
 * C 语言规定 0 值表示为 false，非 0 值表示为 true
 */
/* determine if the game is over */
bool is_finished(void);
/* move left */
bool move_and_merge_left(void);
/* move right */
bool move_and_merge_right(void);
/* move up */
bool move_and_merge_up(void);
```

```c
/* move down */
bool move_and_merge_down(void);

void run_game(void) {
  initialize_board();
  generate_number();
  print_board();
  enum Direction direction;
  bool generate_flag = true;
  do {
    if (generate_flag) {
      generate_number();
    }
    print_board();
    while ((direction = choose_direction()) == NONE) {
      continue;
    }
    switch (direction) {
    case UP:
      generate_flag = move_and_merge_up();
      break;
    case DOWN:
      generate_flag = move_and_merge_down();
      break;
    case LEFT:
      generate_flag = move_and_merge_left();
      break;
    case RIGHT:
      generate_flag = move_and_merge_right();
      break;
    }
  } while (!is_finished());
}
```

如果你对上述语句的语法感到陌生(除了 enum )，你应该回去阅读第二讲的课程内容，或者STFW, RTFM, ATFAI

## enum 枚举类型

枚举类型是独立的类型，其值为包含所有其显示命名的常量（枚举常量）的底层类型（例如默认的int）的值.

需要注意，枚举类型与一般的基本类型不同，枚举类型程序员通过 enum 关键字自定义的一种类型.

你可以通过下面这种方式使用枚举类型:

```c
// 定义枚举类型 color_t 该类型有三种取值: RED(0), GREEN(1), BLUE(2), 枚举常量是自动递增的
enum color_t {RED, GREEN, BLUE};
// 定义 color_t 类型的变量 color, 初始化为 RED(0)
enum color_t color = RED;

#include <stdio.h>

int main(void) {
  // 你可以通过'='在枚举常量后面指定枚举值对应的底层类型的值
```

```
  enum TV { FOX = 11, CNN = 25, ESPN = 15, HBO = 22, MAX = 30, NBC = 32 };
  printf("List of cable stations:\n");
  printf(" FOX: \t%2d\n", FOX);
  printf(" HBO: \t%2d\n", HBO);
  printf(" MAX: \t%2d\n", MAX);
}
```

```
List of cable stations:
  FOX:   11
  HBO:   22
  MAX:   30
```

一般枚举可以作为常量定义的一种方式, 可以愉快的与`switch`搭配使用，提升代码可读性.

## 2048 具体实现展开

```
/* 不言自明 */
void initialize_board(void) {
  /* 随机数种子初始化，用于确保 rand 能够正常使用 */
  srand(time(0));
  for (int i = 0; i < BOARD_SIZE; i++) {
    for (int j = 0; j < BOARD_SIZE; j++) {
      board[i][j] = 0;
    }
  }
}

/* 使用 for 循环，判断游戏是否终止:
 *  1. 出现 2048
 *  2. 只要游戏板被填满
 * 出于简化实现的目的，我们简单化了终止条件的判断，
 * 如果感兴趣你可以实现一个完整功能的is_finished,
 * 当然你可以需要为此修改你游戏运行的其他逻辑，
 * 比如打印(考虑不同位数数字的打印情况)
 */
bool is_finished(void) {
  bool flag_no_space = true;
  for (int i = 0; i < BOARD_SIZE; i++) {
    for (int j = 0; j < BOARD_SIZE; j++) {
      if (board[i][j] == 2048) {
        return true;
      }
      if (board[i][j] == 0) {
        flag_no_space = false;
      }
    }
  }
  return flag_no_space;
}

/* 随机生成 2/4 置于游戏板空白位置
 * 使用 for 循环将空白位置的坐标 (i, j) 记录到 empty_space
 * 使用 stdlib.h 提供的 rand 函数生成随机数
 * 表达式 (rand() % 2 + 1) * 2 用于随机生成 2/4
```

```c
 */
void generate_number(void) {
  int empty_space[BOARD_SIZE * BOARD_SIZE][2];
  int count = 0;

  for (int i = 0; i < BOARD_SIZE; i++) {
    for (int j = 0; j < BOARD_SIZE; j++) {
      if (board[i][j] == 0) {
        empty_space[count][0] = i;
        empty_space[count][1] = j;
        count++;
      }
    }
  }

  if (count > 0) {
    int index = rand() % count;
    int x = empty_space[index][0];
    int y = empty_space[index][1];
    board[x][y] = (rand() % 2 + 1) * 2;
  }
}

/* 不言自明 */
enum Direction choose_direction(void) {
  char dir = getchar();
  switch (dir) {
  case 'A':
  case 'a':
    return LEFT;
  case 'W':
  case 'w':
    return UP;
  case 'S':
  case 's':
    return DOWN;
  case 'D':
  case 'd':
    return RIGHT;
  default:
    return NONE;
  }
}

/* 打印游戏板画面
 * system: 用于在 C 语言程序中执行终端命令, clear 用于终端清屏(RTFM-mam system, man
clear)
 */
void print_board(void) {
  system("clear");
  printf("-------------------------\n");
  for (int i = 0; i < BOARD_SIZE; i++) {
    putchar('|');
    for (int j = 0; j < BOARD_SIZE; j++) {
      if (board[i][j] != 0) {
        printf("%5d|", board[i][j]);
```

```
      } else {
        printf("     |");
      }
    }
    putchar('\n');
  }
  printf("------------------------\n");
}

/* 辅助函数，不言自明 */
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

/* 左移动更新逻辑
 * 函数将操作拆分为两步，向左移动与向左合并
 * moved_flag: 用于判断是否进行移动，return给调用方，用于判断时候需要生成新的数字
 * 移动逻辑: for 循环检测空白就将右边的数交换过来
 * 合并逻辑: 循环检测两个相邻(无视空格)的数，如果相等就合并
 * 2048 的合并逻辑注释事项:
 * | 4 | 4 | 4 | 4 | 左合并的结果应该为 | 8 | 8 | 0 | 0 |
 */
bool move_and_merge_left(void) {
  bool moved_flag = false;
  for (int row = 0; row < BOARD_SIZE; row++) {
    for (int col = 0; col < BOARD_SIZE - 1; col++) {
      if (board[row][col] == 0) {
        for (int search_col = col + 1; search_col < BOARD_SIZE; search_col++) {
          if (board[row][search_col] != 0) {
            moved_flag = true;
            swap(&board[row][col], &board[row][search_col]);
            break;
          }
        }
      }

      int next_not_zero_col = col + 1;
      while (board[row][next_not_zero_col] == 0 &&
             next_not_zero_col < BOARD_SIZE)
        next_not_zero_col++;
      if (next_not_zero_col < BOARD_SIZE &&
          board[row][col] == board[row][next_not_zero_col]) {
        moved_flag = true;
        board[row][col] *= 2;
        board[row][next_not_zero_col] = 0;
      }
    }
  }
  return moved_flag;
}
/* 其余移动处理函数类似，不再在此展示 */
```

事实上一个最简单的应用程序: 2048, 已经包含了数据结构(数组)和基本的算法(循环迭代).

### 数据结构

数据结构是程序组织结构化数据的一种方式, 数组作为一种基础的数据结构被广泛应用.

想象一下 C 语言如果没有数组, 这意味着: 如果你需要表示一连串的数值, 那么你需要对所有的数值变量进行声明`int a1, a2, a3...;`.

常见的数据结构还有:

1. 栈
2. 队列
3. 链表
4. 树

目前你掌握到的数据结构只有数组, 但这就足够了, 让我们先把数据结构放到一边, 让我们先来探讨一下算法.

# 认识算法

## 什么是算法

> Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

算法是解决问题的工具, 算法需要处理满足问题约束条件下任意的输入组并给出对应的输出组.

具体而言, 算法是将输入变为输出的操作序列.

## 什么是算法要解决的问题

> The statement of the problem specifies in general terms the desired input/output relationship, typically of arbitrarily large size.

算法要解决的问题是对输入输出之间关系的抽象描述,这样的描述往往对任意输入规模的数据包括大规模数据都生效.

因而算法要解决的问题不是特定的实例, 而是广泛的抽象存在.

## 算法的需求

> When you design and analyze algorithms, you need to be able to describe how they operate and how to design them. You also need some mathematical tools to show that your algorithms do the right thing and do it efficiently.

在设计一个算法时, 你首先要对算法的行为进行精确的描述.

其次要通过数学证明等手段说明算法的正确性, 即可以通过给定输入得到正确的结果.

## 如何描述算法

算法是对如何解决问题的描述,描述算法就是描述如何去解决一个问题.

例如有人问你,如何从文泽路地铁站走到杭电生活区.

你回答道：“向北走,到达弗雷德广场后,向西走.”

可以说这也是一种算法,当然在实际生产活动中,我们遇到的问题往往更加抽象.

例如:

问题背景：给定一张网格图,向右为x轴正方向,向上为y轴正方向,在格点处可以上下左右移动至临近格点

问题输入：起点坐标(a,b)终点坐标(c,d)

问题输出：一种从起点前往终点的方案,以字符串表示操作序列(0:上 1:下 2:左 3:右)

这个问题非常简单,相信解决这个问题的算法已经在读者的大脑中呈现了,下面我将使用比较常见的方式来呈现这个算法.

## 文字描述

当前坐标与终点坐标重合时,算法结束

当前坐标位于终点坐标下方时,向上走

当前坐标位于终点坐标上方时,向下走

当前坐标位于终点坐标右侧时,向左走

当前坐标位于终点坐标左侧时,向右走

可以发现这样的描述相当冗长且不严谨,在实践中,除了向别人解释某种算法,我们不会使用这种方式.

例如经典的程序员笑话：女朋友说,下班回来带一个西瓜.如果看到番茄,就买两个.于是程序员买回来的是两个西瓜.

## 数学描述

设f(x,y)为从坐标(x,y)出发,到终点(c, d)的合法操作序列,类型为支持使用加法拼接的字符串

$$f(x,y)=\begin{cases}"0"+f(x-1,y), & x>c\\ "1"+f(x+1,y), & x<c\\ "2"+f(x,y-1), & y>d\\ "3"+f(x,y+1), & y<d\\ & (x,y)=(c,d)\end{cases}$$

这是较为常见的描述方式,实际的描述会比上文更加严谨,当算法描述到这一步时,离实现基本不远了

## 伪代码描述

字面意思, 通过类似代码的文字语言去描述算法执行的过程, 随着学习的语言越来越多, 你会发现在基本逻辑上他们基本是互通的.

```
函数 findPath(a, b, c, d):
    如果 (a, b) == (c, d):
        返回 ""  // 到达终点,返回空字符串

    操作序列 = ""

    如果 a > c:
        操作序列 += "0"  // 向上走
        返回 操作序列 + findPath(a - 1, b, c, d)

    如果 a < c:
        操作序列 += "1"  // 向下走
        返回 操作序列 + findPath(a + 1, b, c, d)
```

```
如果 b > d:
    操作序列 += "2"  // 向左走
    返回 操作序列 + findPath(a, b - 1, c, d)

如果 b < d:
    操作序列 += "3"  // 向右走
    返回 操作序列 + findPath(a, b + 1, c, d)
```

当算法完备到这个程度时, 只要你熟悉语法, 想必实现算法便不是什么难事, 这也便是为何这节课安排在week4的原因了

> 相信你已经学会C语言了, 让我们来解决这个问题试试吧

# 为什么我们需要算法

> Of course, computers may be fast, but they are not infinitely fast. Computing time is therefore a bounded resource, which makes it precious. Although the saying goes, Time is money, time is even more valuable than money: you can get back money after you spend it, but once time is spent, you can never get it back. Memory may be inexpensive, but it is neither infinite nor free. You should choose algorithms that use the resources of time and space efficiently

随着硬件的不断提升, 计算机的效率得到了显著的提高.

然而, 这不能改变我们对更优秀的算法的渴求, 因为计算机的资源终究是有限的.

一味的追求硬件上的提升, 而放弃算法上的精进, 首先会遇到严重的边际效应（就好像周长不变的情况下, 面积最大的长方形是正方形）

> 看看牙膏厂吧, 家人们, 如果提升硬件那么容易, 为啥还要挤牙膏

其次, 硬件和算法实际上是相辅相成的, 算法的演化往往会带动硬件厂商去制作更适配该算法的硬件

例如针对深度学习等算法对矢量计算的高度依赖, 现代CPU引入了向量化指令集, 也有GPU架构为了支持高效的矢量计算而设计（例如CUDA）

就我们身边而言, 你每次真金白银升级的硬件, 最后的结局往往是被软件开发者浪费, 说的就是浪费内存和电量微信, 以及之前每次更新不清除老数据浪费存储空间的〇神（听说现在改了, 不知道阿, 我不玩〇神）

> “无论硬件给了你多少性能, 软件都会把它拿走！”

# 算法举例——正整数的高精度乘法

## 问题背景

众所周知, C语言中的整数类型能存储的类型是有限制的

假设我现在希望得到1000位的正整数与1000位的正整数相乘的结果

我们能否设计一个相应的算法呢

## 问题描述

输入：正整数x,y

输出：x*y的结果

## 整体思路

简单的, 我们联想到小学生都会的竖式乘法

发现这个算法对数字的位数不存在限制

考虑在计算机中设计算法来模拟这一过程

## 拆解问题

1. 输入
2. 拆分数字到各位
3. 竖式乘法
4. 输出

**子问题A——拆分数字**

输入：正整数x

输出：从低位向高位顺序的各位数字组成的序列

思路：使用取余获取当前个位, 使用整除（C语言中整数对整数做除法自动舍去小数位）获取高位数字

伪代码:

```
函数 split_number(整数 x) 返回 数字序列:
    如果 x 等于 0:
        返回 {0}

    创建数字序列 arr

    // 将 x 的每一位数字存入数字序列 arr
    循环 当 x 不等于 0:
        将 x取余10 的值存入数字序列 arr
        将 x 设为 x整除10 的值

    返回 数字序列 arr
```

**子问题B——竖式乘法**

输入：拆分数字得到的序列x与y

输出：计算结果z

思路：创建序列a存储运算结果, 枚举两段序列的各个元素, 并将相乘的结果存入a, 最后对序列a进行进位处理

伪代码:

```
函数 multiply(数字序列 x, 数字序列 y) 返回 数字序列:
    创建数字序列 z, 长度为 x 的长度 + y 的长度

    // 初始化数字序列 z 的所有元素为 0
    循环 对于每个元素 i 从 0 到 z 的长度 - 1:
        设置 z[i] 为 0

    // 进行乘法运算
```

```
    循环 对于每个元素 i 从 0 到 x 的长度 - 1:
        循环 对于每个元素 j 从 0 到 y 的长度 - 1:
            z[i + j] += x[i] * y[j]

    // 处理进位
    循环 对于每个元素 i 从 0 到 z 的长度 - 2:
        z[i + 1] += z[i] 整除 10
        z[i] = z[i] 取余 10

    // 去除前导零
    循环 当 z 的最后一个元素为 0 且 z 的长度大于 1:
        数字序列z的长度 减少 1

    返回数字序列 z
```

事实上, 上述代码使用C去优美实现时, 我们需要结构体与malloc的帮助.

这说明我们当前的语法还不足够实现所有算法, 我们仍需努力.

## 算法与数据结构

> A data structure is a way to storeand organize data in order to facilitate access and modifications. Using the appropriate data structure or structures is an important part of algorithm design. No single data structure works well for all purposes, and so you should know the strengths and limitations of several of them

在今后的算法学习中, 我们常需要数据结构来帮助算法的实现, 甚至很多算法是基于某种数据结构存在的（例如图论算法）

如何判断我的算法是否需要数据结构? 算法的雏形是处理问题的流程, 如果在该流程中, 存在与当前问题无关但难以解决的问题, 那么你的算法可能需要数据结构的帮助.

例如在设计算法的过程中, 我发现我需要某一个动态的元素集合中的最小值, 那么我便可以使用小根堆这一数据结构来满足的需求.

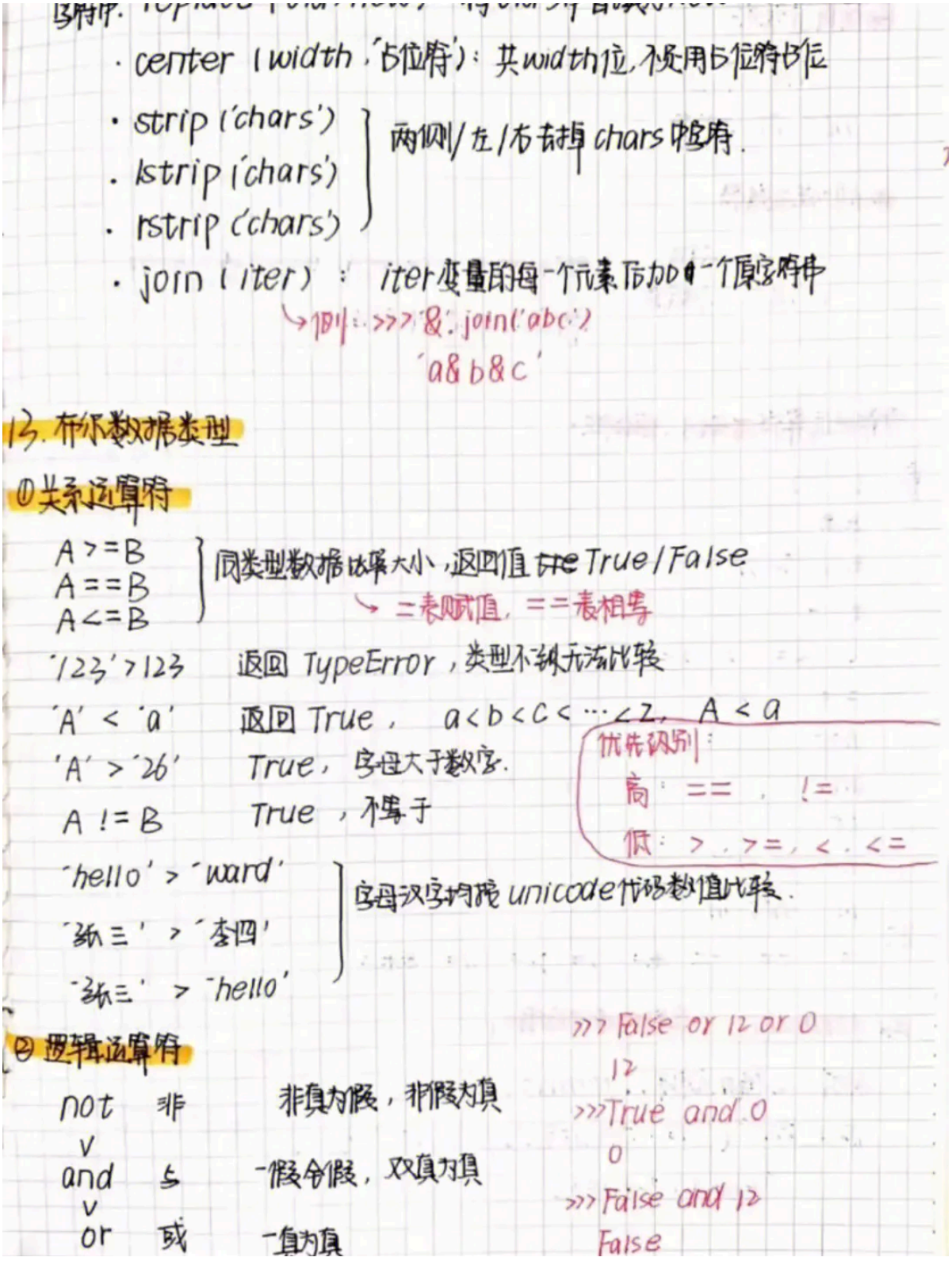
有人说, 程序 $=$ 算法 $+$ 数据结构.

算法就像骨, 是程序的支架, 构建了程序的功能性.数据结构就像肉, 为程序提供力量, 保证程序的功能得到正确的实现.

在日后学习算法的过程中, 希望同学们不要生搬硬套, 而是多去思考程序背后的原理.

学数据结构时将其应用到相关算法中加深理解; 学算法时运用学过的数据结构巩固记忆.

高效学习是破除无用内耗的良药.




> 不要这么学, 求你了...

# 认识函数

## 什么是函数

上面的伪代码在开头出现了函数的概念, 除此以外其余操作流程都在我们目前熟悉的知识体系内.

在之前的学习中, 我们的代码都写在main中, 似乎这就是代码唯一能存在的地方.

事实上, int main只是C语言中一种特殊的的函数, 被称为主函数, 我们也可以定义自己的函数并将代码写在其中.

在数学中, 函数在数学中, 函数是一个关系, 它将一个集合中的每个元素（称为自变量或输入）映射到另一个集合中的唯一元素（称为因变量或输出）.

函数通常用符号表示, 如f(x), 其中 f是函数名, x是自变量.

而在C语言中, 函数也可以是一种映射关系, 例如上文的两个函数的结果都仅与传入参数有关, 并且输入输出满足了某种关系.

但是更多时候C语言的函数更像一种过程, 仅仅只是对代码的封装, 可以等价的展开到引用函数处.

许多语言严格的将过程与函数严格的区分开来, 例如Pascal使用关键字function与procedure进行区分.

至于C, 你可以认为没有返回值的函数就是过程.

## 函数的定义

```
<return type> function_name(<type_1> parament_1,<type_2> parament_2 ...){
    do somthing...
    return something or nothing;
}
```

例如对于main函数而言, 其返回类型为int.

> **Caution**
> C标准规定其类型必须为int, 虽然事实上很多类型也能通过编译, 例如signed, 但是我们不推荐你这么做

当其返回值为0时, 表示程序正常结束, 当返回值不为0时, 表示程序没有正常结束

任何时候, 当你在当前函数的作用域中调用return, 都代表程序的结束.return的类型需要与函数声明一致.（如果函数类型为void, 仅需return;即可）

对于有返回值的函数, 如果在函数结束时仍没有执行return, **可能会引发UB**.

同样main函数也可以有传入参数, 根据 C 标准, 其传入参数要么为空, 要么为(int argc, char *argv[]), 或者其他由实现定义

使用该参数的main函数编译生成的可执行文件在执行时可以接受命令行参数

让我们来实践一下:

1. 在你的工作目录中创建文件example.c
2. 写入:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf("Number of arguments: %d\n", argc);
    for (int i = 0; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }
    return 0;
}
```

3. 目录下命令行输入编译指令：gcc -o example example.c
4. 目录下命令行输入运行指令：./example arg1 arg2 arg3

## 函数的作用

在实际的编程中, 你的函数可能有以下作用:

- 对经常复用的代码的封装, 使你的程序主体更加简洁.
- 依旧是对代码的封装, 但是传入参数与传出参数之间存在关联性, 作为一种问题解决方案的抽象.
- 作为解决问题的一种不可代替的手段, 例如递归函数.

## 函数的抽象意义

当函数作为处理问题的工具时

对于函数的使用者, 函数的抽象含义为:

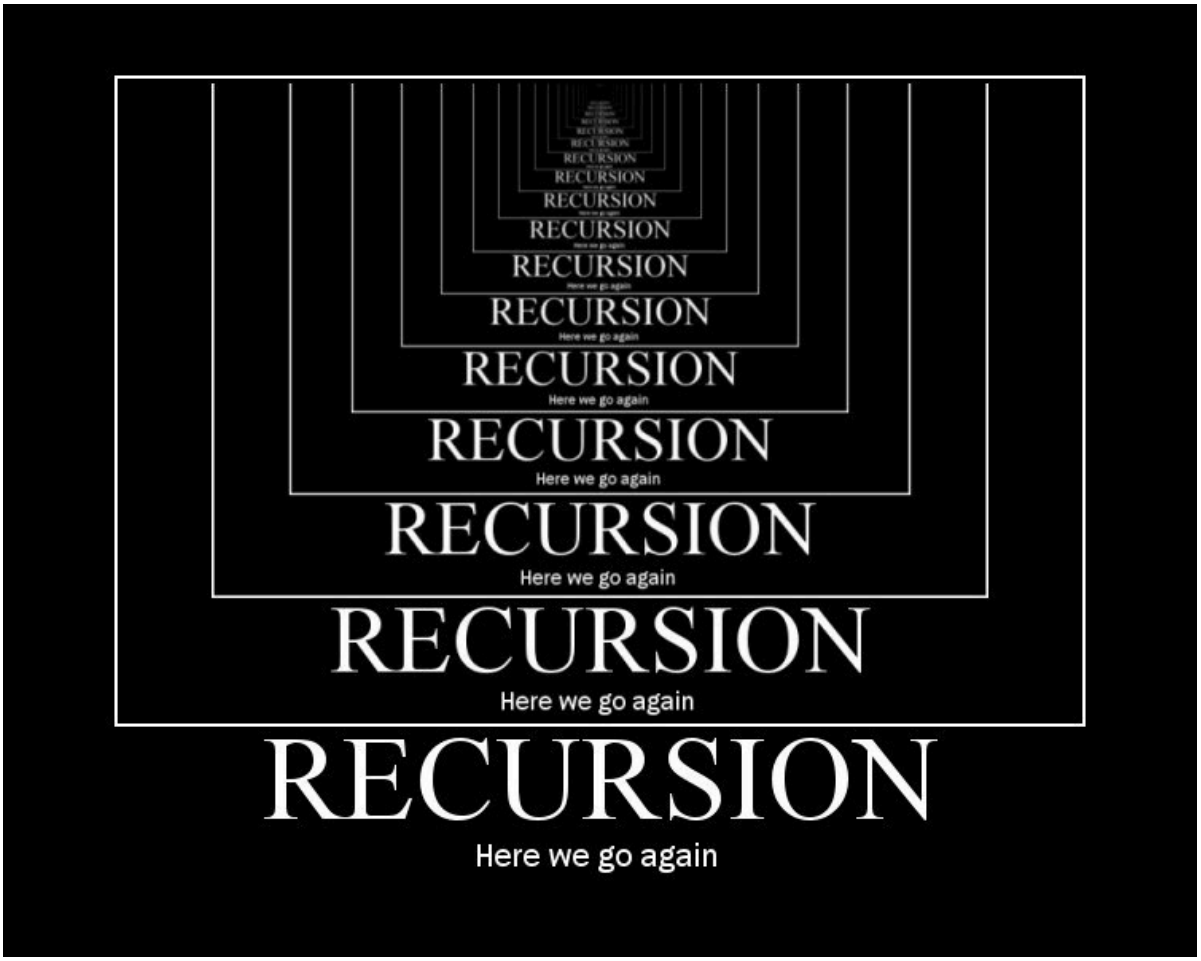
- 在调用该函数后, 便会给出符合函数功能的结果
- 函数的使用者不需要清楚函数内部发生了什么, 而是将其当作黑盒使用
- 例如printf是C标准库的一个函数, 当我们使用它时, 不需要了解它是如何实现的

对于函数的维护者, 函数的抽象含义为:

- 在维护该函数时, 其不需要了解使用者是在什么环境下调用该函数, 只需要关注参数即可
- 函数的工作区域仅在函数内部, 与外部作用域无关, 互不干扰

可以说, 计算机科学中的抽象是一种信息分层, 对象主体只需要在所在的信息层面工作, 这保证了信息的安全性和有效性

# 认识递归



# 什么是递归

我们常在主函数中调用各种函数

显然, 我们也可以在函数中调用函数——包括自己

函数调用自己的行为被泛称为递归, 例如:

```
void f() {
  printf("人类的本质是{");
  f();
  printf("}");
}
```

> 试着运行该函数

这是一个无限递归的函数, 它将迅速地耗尽栈空间, 并引发stack overflow的错误

从程序的角度出发, 这段程序的操作序列就像:

```
进入f():
  输出 "人类的本质是{"
  调用f()
  进入f'(): //使用‘区分函数的作用域，每个函数的作用域都是独立的
    输出 "人类的本质是{"
    调用f()
    进入f''(): //使用‘区分函数的作用域，每个函数的作用域都是独立的
      输出 "人类的本质是{"
      调用f()
      进入f''(): //使用‘区分函数的作用域，每个函数的作用域都是独立的
        输出 "人类的本质是{"
        调用f()
        ...
```

# Extra：递归与栈

众所周知, 函数保有其自身的局部变量, 当函数被调用时, 内存会为函数调用开辟一片空间用于存储函数的局部数据, 并在函数返回后将空间清除.

考虑一个函数调用情况: main -> f -> g -> h

那么对应的返回情况为: exit <- main <- f <- g <- h

可以看到, 对函数的处理具有`先进后出`的性质: 最先调用的函数在最后返回.

我们一般将拥有`先进后出`性质的数据结构称为: **栈**.

内存中用于程序执行存储函数局部数据的内存部分也就被我们称为**栈内存**（简称**栈**）.

显而易见, 无限递归的函数将迅速消耗你的栈空间并被操作系统检测引发错误.

# 如何设计递归算法

在实际生产中, 我们自然不希望我们的程序是无限递归的.

考虑到递归产生的资源损耗, 我们希望解决问题时递归的层数越少越好, 恰好能解决问题就行.

仍旧以斐波那契数列为例.

回想之前提及过的函数的抽象含义：作为使用者, 我们假设函数会返回期望的结果而不在意其实现.

考虑到斐波那契数列的每一项都等于前两项之和, 我们得到以下代码：

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

考虑函数的抽象含义, 我们认为f(n-2)与f(n-1)就是前两项

但考虑到n=1与n=2时, 项数的值是已知的, 于是我们修改代码:

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

啊, 多么的简洁, 这便是递归的魅力, 或许你还不明白发生了什么, 让我们从算法设计者的角度来重新审视这个问题

在算法导论中, 作者将设计递归算法的过程总结为三步：

1. 拆解：将原问题拆解为多个子问题, 子问题与原问题为同类型的问题且规模更小
2. 求解：将子问题视作新的原问题并重复上一步, 若无法重复则说明子问题是不可拆解的原子问题, 直接尝试求解
3. 合并：将子问题的解对原问题的解的贡献进行合并, 得到原问题的解

下文详细展开这三步在斐波那契数列问题中是如何体现的

### 拆解

当n>2时, 求解f(n)即求解f(n-2)与f(n-1)

子问题与原问题为同一类型问题且规模更小, 符合要求

### 求解

当n<=2时, 直接求解

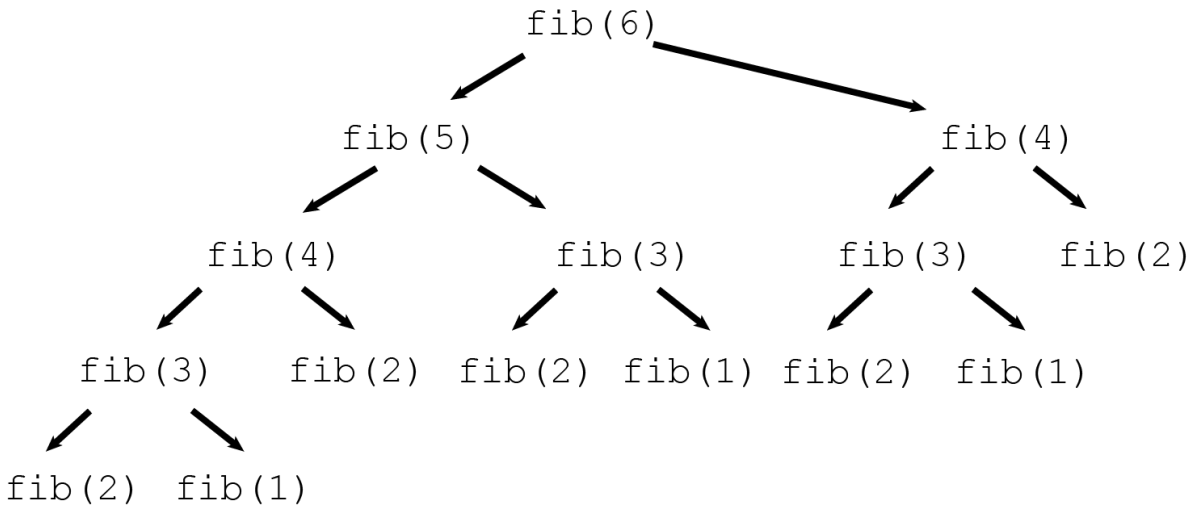
当n>2时, 继续拆解问题

### 合并

原问题的解等于子问题的解的和

## 递归树(选读)

让我们将递归调用的过程看作一棵树：




在树中, 图中的单向箭头被称为单向边

在单向边连接的两个点的相对关系中:

- 单向边的起始节点被称为父亲
- 单向边的中止节点被称为孩子
- 没有孩子的节点被称为叶子

观察上图, 我们发现所有的叶子节点都是不可分割的原子问题, 可以直接求解

假设PC指针在这颗"函数树"上移动, 则其的行为逻辑为:

1. 当前节点为原子问题时, 直接求解（**求解**）, 然后将指针向父亲节点移动（**回溯**）
2. 当前节点的孩子仍未求解时, 移动PC指针（**递归调用**）, 优先求解子问题**(拆分)**
3. 当前节点的孩子都得解时, 合并孩子解得到父亲解（**合并**）, 并将指针向父亲节点移动（**回溯**）

## 递归的作用

我们发现, 递归算法并非不可替代：无论是我们手动模拟堆栈的行为, 或是提前建立递归树在上面进行迭代

然而, 递归有着以下优点:

1. 简洁优雅：参数通过系统隐式传递, 无需程序员考虑空间分配, 无需大量临时变量存储数据, 用抽象壁垒守护程序员的心智.
2. 舒适自然：许多问题天然的具有可递归的性质, 例如树的遍历问题等.
3. 赏心悦目：递归代码普遍较短, 让人感受人类的智慧.

## 递归典范——归并排序

排序作为计算机科学的基本问题, 对于不同的应用场景都有不同的适用算法, 下文将介绍一种对递归进行典型利用的算法——归并排序

### 问题描述

输入：n个数字的序列(a1,a2...,an)

输出：输入序列的重排(a1',a2'...an'), 满足a1'<=a2'<=...<=an'

## 使用分治算法的设计思路解决问题

### 合并

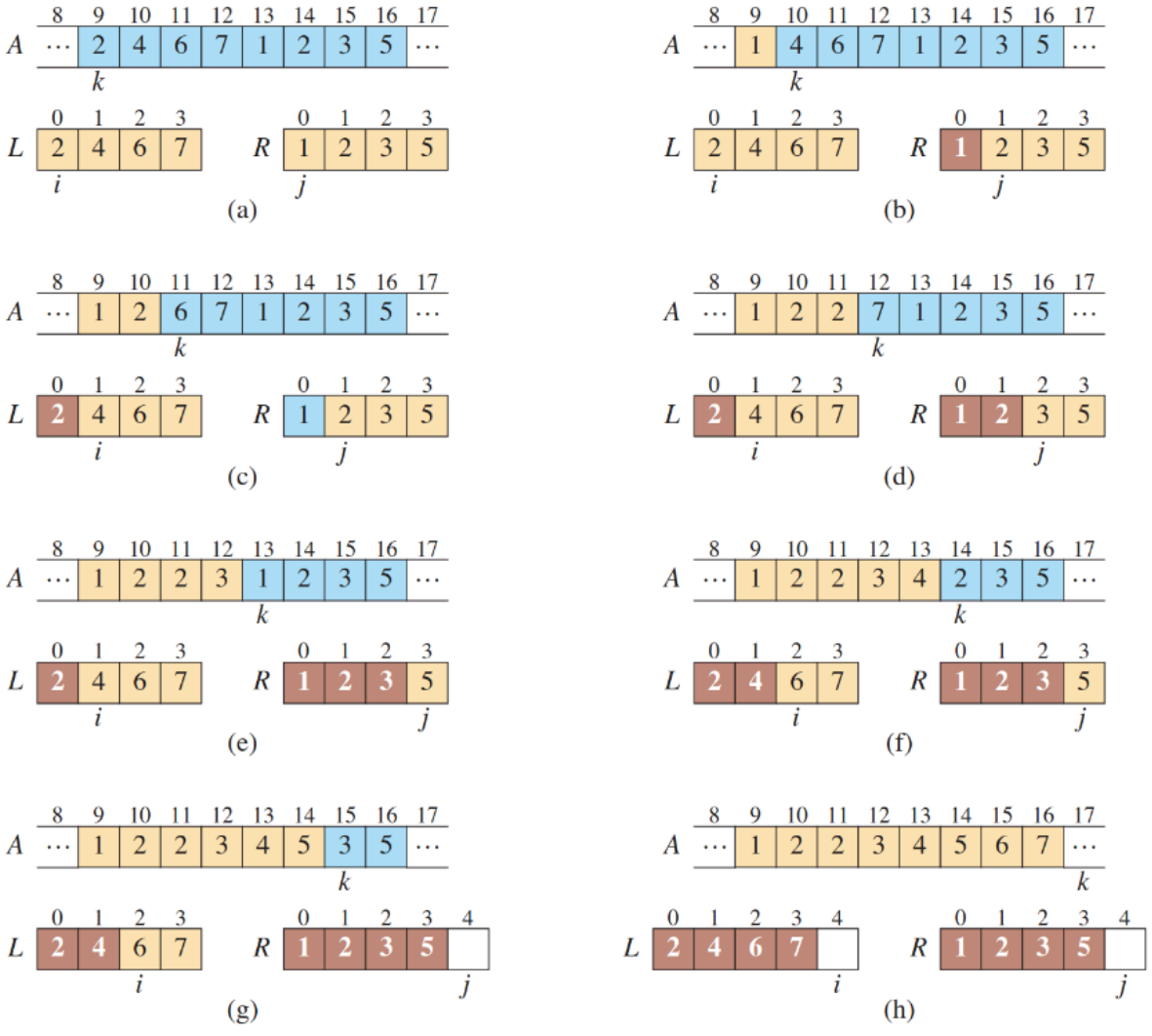
对于这个问题, 我们反过来思考, 对于怎样的子问题, 我们可以通过子问题得到的解得到原问题的解

假设我们现在有两段已经排序好的序列[l...r] [p..q], 我们能否在较短的时间内将两段序列中的所有数字排序成一个新序列呢?

考虑算法:

- 在两段序列非空时, 比较开头元素, 将较小的元素放到新序列尾部
- 有一段序列为空时, 将非空序列的所有元素依序放到新序列尾部
- 当两段序列皆空时, 算法结束

实际运行效果如图所示:




### 拆解

在知道什么样的子问题可以求解原问题后, 将原问题拆成子问题便简单了, 只要将原序列拆成恰好连续的两部分即可

考虑到复杂度问题, 我们将序列对半分, 原因在之后解释

称当前需要排序的序列为[l...r]

则取l,r中点mid, 将序列拆为: [l...mid] [mid+1...r]

**求解**

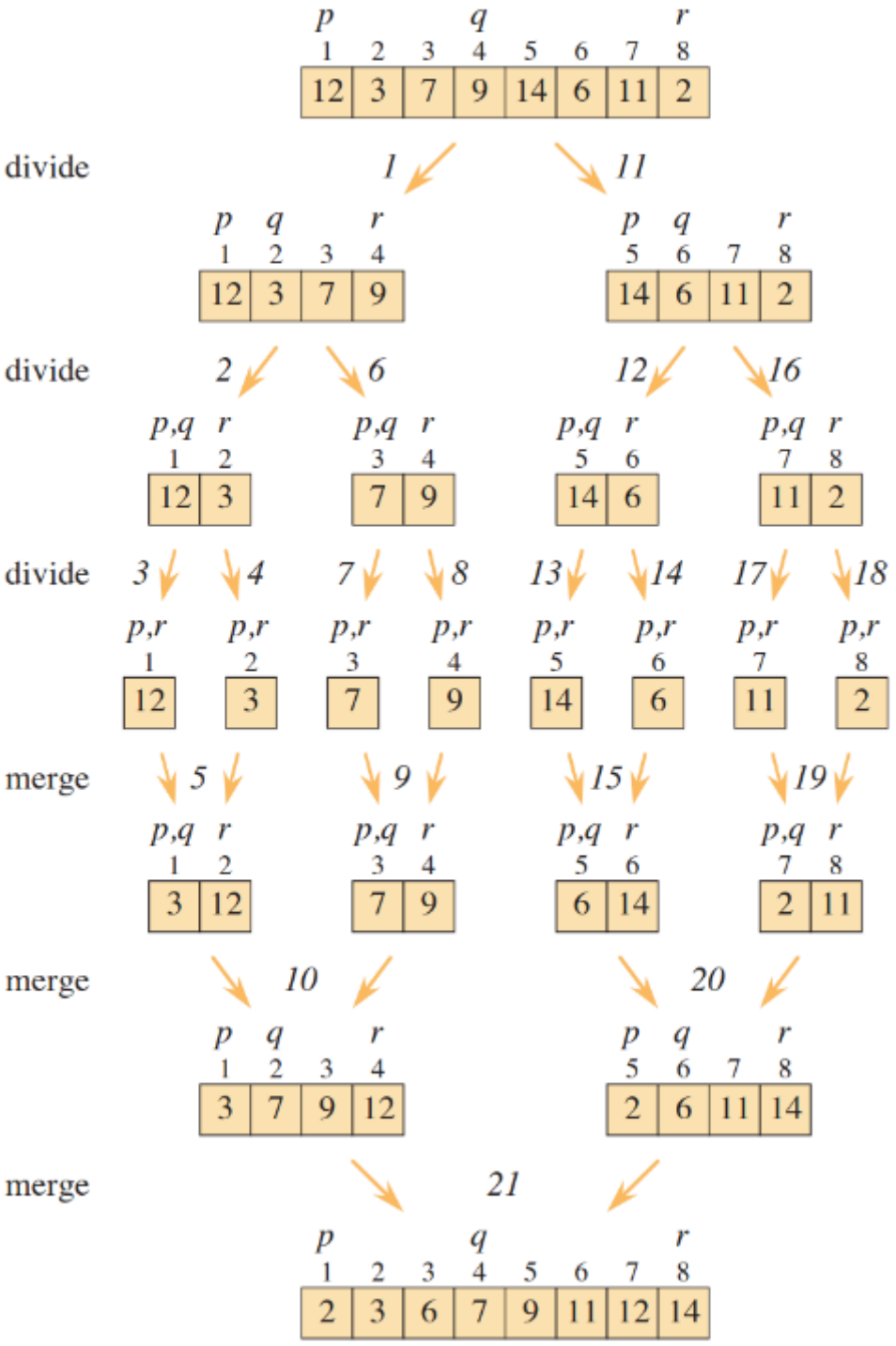
当前序列为空或只有一个元素时, 则当前序列无需排序, 直接回溯

反之, 将当前问题进行分解

## 伪代码

```
函数 merge_sort(数组 arr):
    如果 arr 的长度小于等于 1:
        返回 arr

    // 找到中间索引
    mid = arr.length / 2

    // 递归地对左右子数组进行归并排序
    左半部分 = merge_sort(arr[0:mid])
    右半部分 = merge_sort(arr[mid:arr.length])

    // 合并已排序的子数组
    返回 merge(左半部分, 右半部分)

函数 merge(数组 left, 数组 right):
    创建一个空数组 result
    初始化指针 i 和 j 为 0

    // 合并两个已排序的数组
    循环 当 i < left.length 且 j < right.length:
        如果 left[i] 小于等于 right[j]:
            将 left[i] 添加到 result
            i 增加 1
        否则:
            将 right[j] 添加到 result
            j 增加 1

    // 添加剩余的元素
    循环 当 i < left.length:
        将 left[i] 添加到 result
        i 增加 1

    循环 当 j < right.length:
        将 right[j] 添加到 result
        j 增加 1

    返回 result
```

**图示**




# 简要认识时间复杂度(选读)

我们常说算法的效率很重要, 算法的效率可以分为时间效率和空间效率

在描述问题时, 我们常用字母n描述问题的规模

朴素的想法是, 对于什么规模的问题, 以该算法实际使用了多少运行时间来衡量算法的快慢

但这样的想法有着诸多缺点, 比如：用户的硬件并不相同, 以什么硬件的运行时间为基准; 问题输入的数据有近乎无穷的可能, 以什么样的数据下的运行时间为基准.

为此, 时间复杂度的概念被引入了.

算法的时间复杂度是指算法运行时间随着输入规模增长而变化的趋势.

它并非衡量算法运行时间的绝对值（因为这取决于硬件、编译器等因素）, 而是一个渐进的、相对的衡量标准, 描述了算法运行时间对输入规模的依赖关系.

我们通常用大O符号（Big O notation）来表示时间复杂度.

大O符号表示的是算法运行时间的上界, 也就是最坏情况下的时间复杂度.

它忽略常数因子和低阶项, 只关注最高阶项.

例如O（1）是指该算法与输入规模无关, 为常数复杂度

O（n）指该算法时间复杂度与输入规模成正比

假设某算法的时间复杂度可以用多项式：$t(x)=x^2+x+1$表示, 再设多项式$f(x)=x^2$, 不难发现

$x=10$时, $t(x)=111, f(x)=100, \frac{t(x)}{f(x)}=1.11$

$x=10000$时, $t(x)=100010001, f(x)=100000000, \frac{t(x)}{f(x)}=1.00010001$

当输入规模足够大时, 最高阶项对运行时间的影响是主要的.

## 归并排序与冒泡排序的算法复杂度比较

对于初学者而言, 第一个接触的排序算法通常是冒泡排序, 它的代码如下:

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

显然, 这个算法有一个由两个循环组成的嵌套循环, 每个循环的时间复杂度都与n相关, 所以该算法是$O(n^2)$的存在的

反观归并排序, 他的代码可能为:

```
void merge_sort(int *arr, int l, int r) {
  if (l >= r)
    return;
  int mid = (l + r) / 2;
  merge_sort(arr, l, mid);
  merge_sort(arr, mid + 1, r);
  int buffer[r-l+1];
  int len = 0;
  int i = l, j = mid + 1;
  while (i <= mid && j <= r) {
    buffer[len++] = arr[i] < arr[j] ? arr[i++] : arr[j++];
  }
  while (i <= mid) {
    buffer[len++] = arr[i++];
```

```
  }
  while (j <= r) {
    buffer[len++] = arr[j++];
  }
  memcpy(arr + l, buffer, sizeof(int) * len);
  return;
}
```

函数主体有三个循环, 皆与n相关, 而且函数自己还会调用自己, 复杂度看起来比冒泡排序还高

然而, 让我们以n=10000的数据进行一次测试, 却得到结果:

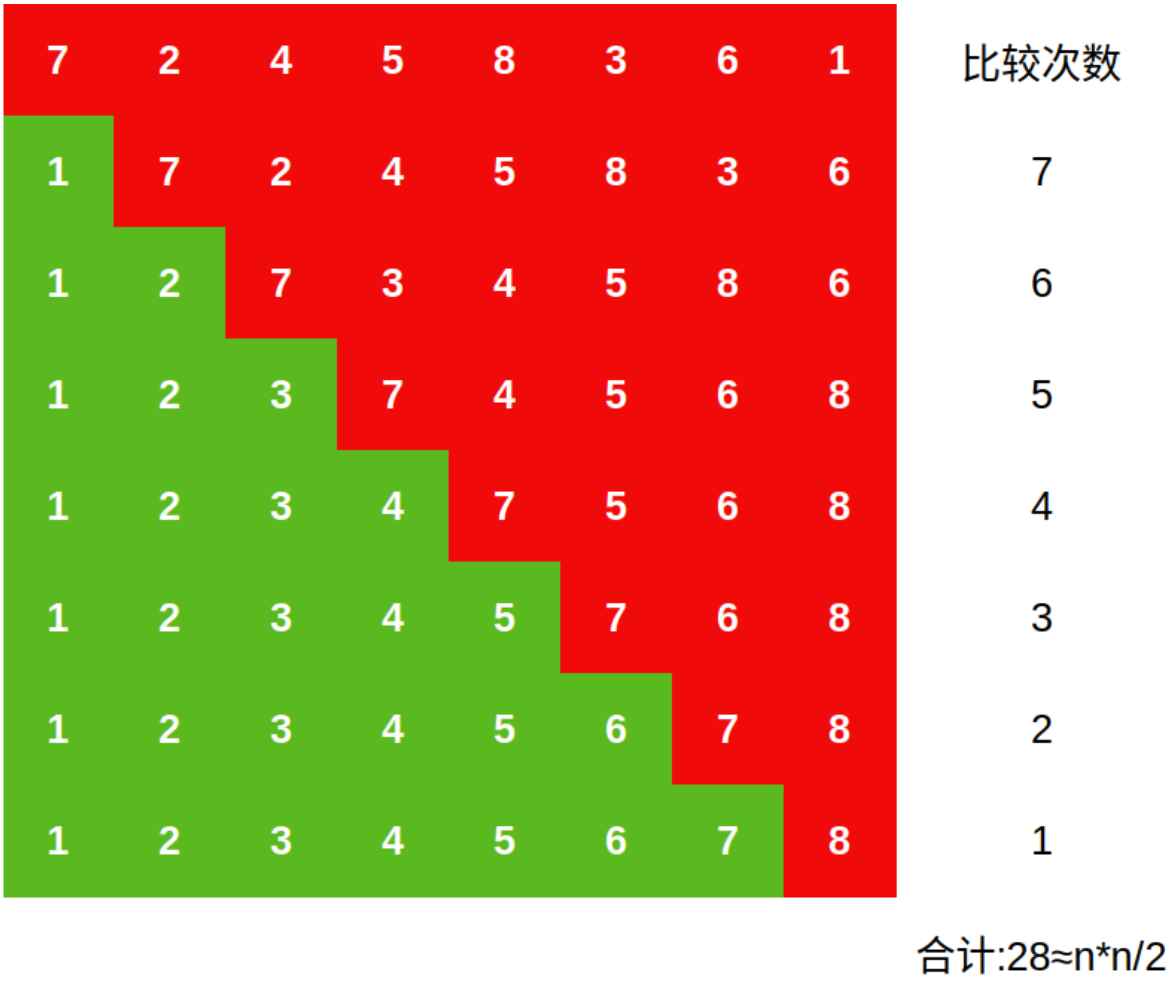
```
mergesort cost 0.002185 second on datasize[10000]
bubblesort cost 0.290680 second on datasize[10000]
```

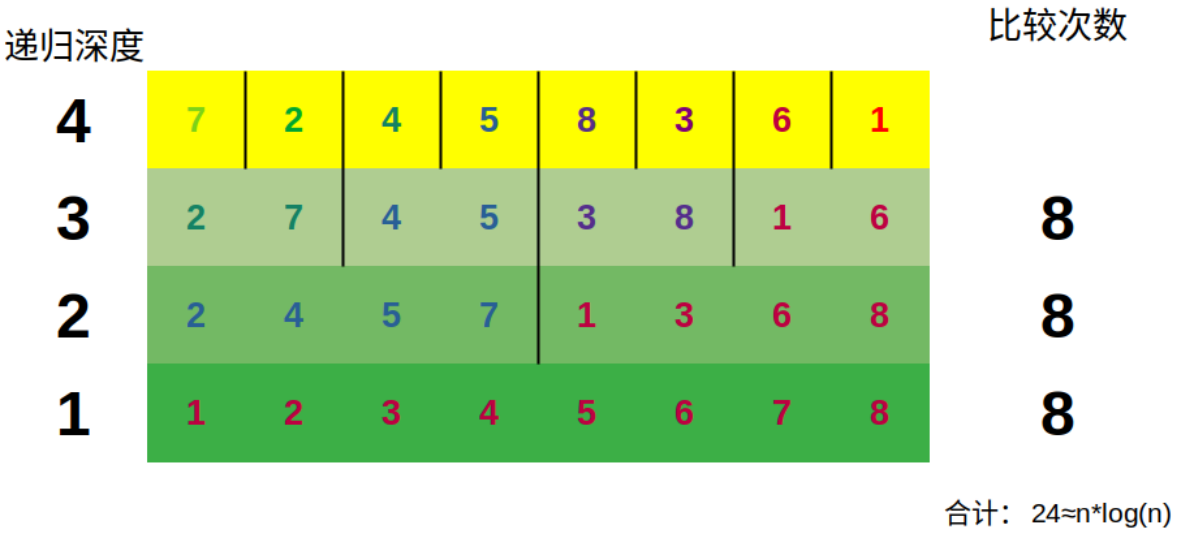
> 测试代码放在仓库里了, 快去自己试试吧

先说结论, 归并排序的时间复杂度为$O(nlog(n))$

出于这门课的性质, 详细的数学证明不再展开, 读者只需通过下方图片感性理解即可

出于代码实现的不同, 我们此处仅比较排序算法中最关键的一步——比较, 此处将比较次数作为唯一的判断依据






观察上图, 不难发现, 冒泡排序的操作次数近似于三角形的面积, 所以冒泡排序是与$n^2$相关的

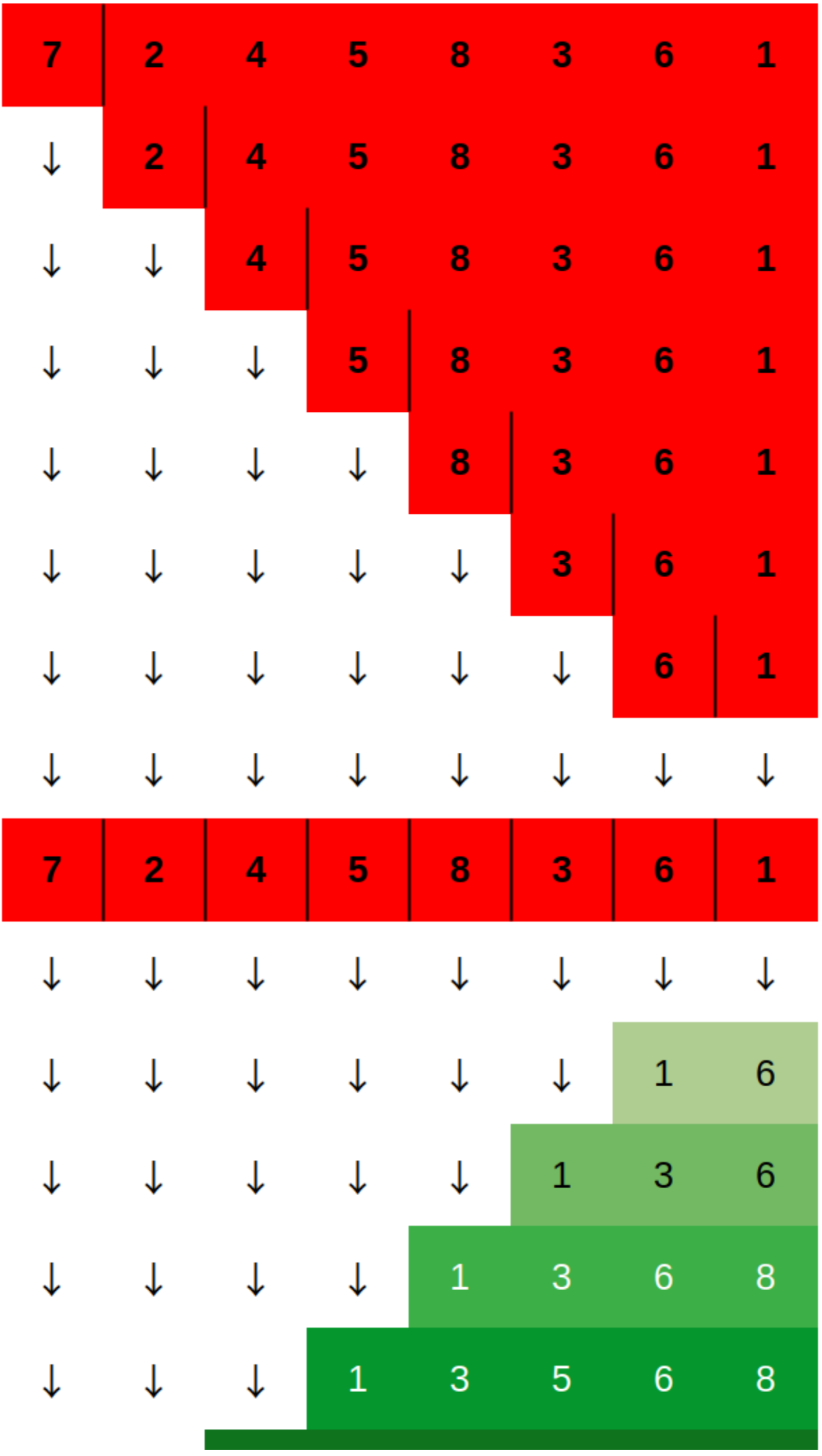
虽然归并排序的操作次数貌似是一个矩形, 但是其长度与$n$相关, 其高度实际上是与$log(n)$相关的(可通过二叉树相关知识证明)

所以归并排序在时间复杂度上是极其优越的, 其算法本身和算法的思维被广泛应用到今后的学习中

算法的复杂度不是肉眼能看出的所以然, 而是需要数学工具去进行分析的

感兴趣的同学可以自行阅读算法导论原书

# 为什么归并排序时选择将序列对半分


7 2 4 5 8 3 6 1
2 4 5 8 3 6 1
4 5 8 3 6 1
5 8 3 6 1
8 3 6 1
3 6 1
6 1
7 2 4 5 8 3 6 1
1 6
1 3 6
1 3 6 8
1 3 5 6 8


观察上图, 我们发现当归并排序使用最极端的"一九分"时, 其复杂度劣化到$n^2$级别

考虑到递归对计算机资源的大量开销, 该程序的效率甚至可能还不如冒泡排序, 真是触目惊心！！！

在之后的学习中我们可能会接触插入排序这一算法, 学习过后可以发现此处的归并排序实际劣化为与插入排序"等价"的算法

计算分治算法复杂度中常用的数学方法有主函数定理等

对于归并排序的复杂度证明便可以使用主函数定理

感兴趣的同学可以查阅：https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

## 时间复杂度与实际效率

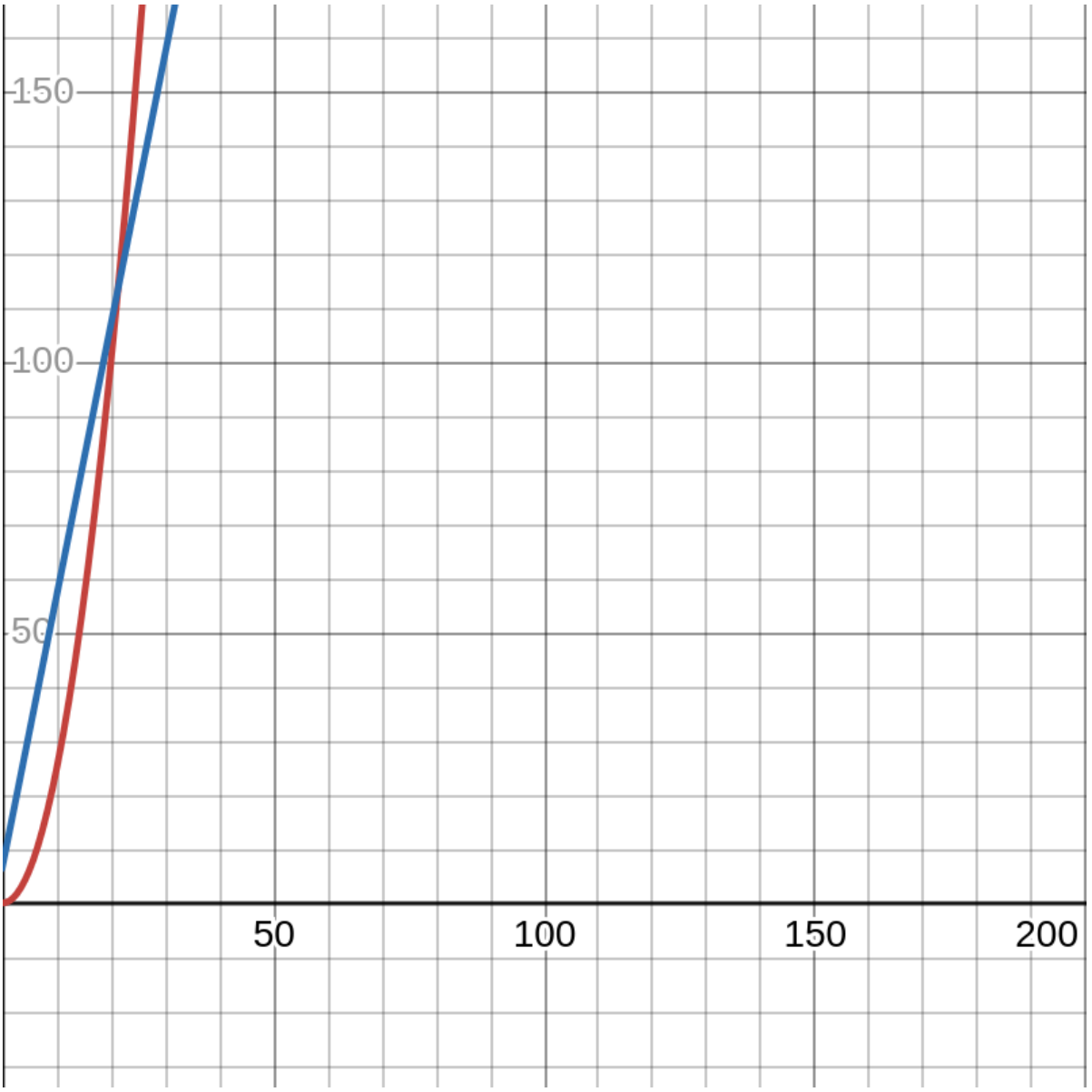
事实上, 算法的时间复杂度有时候与实际效率并不一致.

从数学的角度思考, 在变量较小时, 常数对函数的大小影响极大, 当变量趋于无穷时, 函数间的差异才是最显著的

例如, 假设算法A与算法B的实际算法操作次数为$f_A(n)=n^2/4$和$f_B(n)=5n+7$

显然, 二者的时间复杂度为$O(n^2)$和$O(n)$

然而实际上, 如下图所示, 在数据较小时, 后者显著优于前者.




这说明常数对算法实际效率的影响是巨大的, 在日后的编程中我们需要注意这点.

# 我想学算法

1. 经典书籍: 算法导论
2. Stanford的算法入门(C++实现): CS 106B, 宝宝巴士
3. UCB的算法入门(Java实现): CS 61B, 偏工程
4. MIT的典中典: MIT 6.006, MIT 6.046, 偏理论
5. Coursera: Algorithm I, Algorithm II, 高质量网课