数据结构基础

刘杰
人工智能学院
南开大学 Nankai University

第1章 绪论

数据结构的发展历史

- 20世纪40年代：只能直接对二进制数进行计算；
- 20世纪50年代：各种高级程序设计语言纷纷出现，所能描述的数据类型也逐渐增多；
- 20世纪60年代：美国计算机界出现了信息结构这一名称，后来改用数据结构；
- 1968年：由美国计算机协会（ACM）颁发了建议性的计算机教学计划，计划中规定数据结构作为独立的一门课程；

数据结构的发展历史

- 1968年，D. E. Knuth教授的巨著《计算机程序设计艺术》第一卷《基本算法》出版。
- 20世纪60年代末到70年代初：结构程序设计成为程序设计方法学的主要内容，人们对数据结构越来越重视，著名的计算机科学家N. Wirth写了《算法+数据结构=程序》；
- 20世纪80年代以后：抽象数据类型概念的引入，将数据类型和与之相结合的操作合为一体，而将操作的具体实现和它的定义分离开来，将数据结构的理论和实践提高到一个新的水平。

本课程的内容

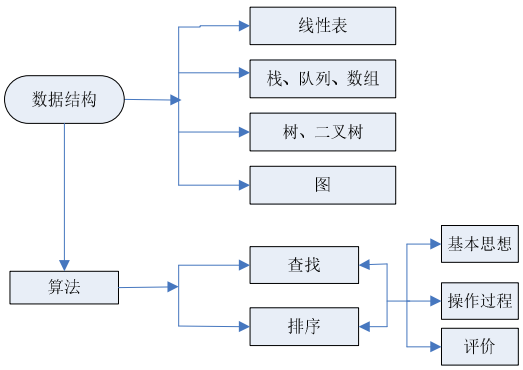
数据结构课程的内容体系：

过程 \ 方面	数据表示	数据处理
抽象	逻辑结构	基本运算
实现	存储结构	算法
评价	不同数据结构的比较和算法性能分析	

简单地说，本课程研究的内容包括以下三方面：

1. **数据的逻辑结构**
2. **数据的存储结构/物理结构**
3. **数据的运算**

本课程的内容——知识图



基本概念与术语

- **数据结构、算法与程序的关系：**

  算 法 + 数 据 结 构 = 程 序

- **数据结构**：(数据元素间的关系称为结构)，相互间存在一种或多种特定关系的数据元素的集合称为数据结构
  - 逻辑结构：元素间的逻辑关系，与计算机无关
  - 物理结构：把数据存储到计算机中，并具体体现数据之间的关系。简言之，是逻辑结构在计算机中的表示(包括数据元素和关系的表示)，同一种逻辑结构可以对应不同的物理结构

数据(Data)：

- 是计算机处理的信息的某种特定的符号表示形式。
- 所有能被输入到计算机中，且能被计算机处理的符号的总称。如：实数、整数、字符（串）、图形和声音等。
- 是计算机操作对象的集合。

基本概念和术语

在用高级程序语言编写的程序中，每个数据都应有一个所属的、确定的数据类型。

其实数据类型反映三个方面的内容：存储结构，取值范围和允许进行的操作。

基本概念和术语

抽象数据类型（ADT）：是指一个数学模型和定义在该模型上的一组操作。

栈的抽象数据类型描述

- 定义：元素类型为T的栈，由T的有限序列组成，对栈可以进行以下的操作：
  - 创建一个空栈；
  - 测试栈是否为空；
  - 压栈操作，即当栈不满时将一个新元素加入到栈顶部；
  - 出栈操作，即当栈不空时删除栈顶元素；
  - 取栈顶元素，即当栈不空时输出栈顶元素；

9

数据元素（Data Element)：

是数据（集合）中的一个“个体”

是数据结构中讨论的基本单位

不同场合也叫结点、顶点、记录

数据项（Data Item）：

是数据结构中讨论的最小单位

数据元素是由若干个数据项组成

例如：描述一个运动员的数据元素

姓名	俱乐部名称	出生日期			参加日期	职务	业绩
		年	月	日			

称之为组合项，其它为简单项

数据对象是性质相同的数据元素的集合

例如：学生成绩表是一个数据对象

学号	姓名	数学分析	普通物理	高等代数
20001	张三	90	56	89
20002	李四	80	87	67
20003	丁一	67	67	87
20004	马二	98	90	67
20005	王五	56	87	68

数据对象中的数据元素不会是孤立的，而是彼此相关的，这种彼此之间的关系称为“结构”。

是相互之间存在一种或多种关系的数据元素的集合

- 逻辑结构
  - 线性结构 (1:1的关系)
  - 非线性结构
    - 树形结构 (1:n的关系)
    - 图状结构 (m:n的关系)
    - 集合结构 (松散关系)
- 存储结构
  - 顺序存储
  - 链式存储
  - 索引存储
  - 散列存储
- 数据操作：创建、销毁、插入、删除、修改等

- 算法的基本概念
- 算法的描述
- 算法分析

算法基本概念

- D.E.Knuth定义：一个算法，就是一个有穷规则的集合，其中之规则规定了一个解决某一特定类型的问题的运算序列；此外，它还应有五个重要特性：
  - 输入：有0或多个输入值；
  - 输出：有1或多个输出值；
  - 确定性：算法的每一步骤，必须是确切地定义的；
  - 有穷性：一个算法必须总是在执行有穷步之后结束；
  - 可行性（有效）：算法中要做的运算都是相当基本的，能够精确地进行的。

算法基本概念

算法设计的目标：

设计算法时，通常应考虑达到以下目标：

1. 正确性
2. 可读性
3. 健壮性
4. 高效率（时间与空间）

伪代码

- 不能够直接执行
- 只表示思想
- 可以由任意语言实现
- 示例：冒泡排序

伪代码

```
for i = [0, n)
    for j = [0, n - i - 1)
        if x[j] > x[j + 1]
            swap(x[j], x[j + 1])
        end
    end
end
```

```
def bubble(x):
    '''
    x: 待排序的列表
    '''
    n = len(x)
    for i in range(n):
        for j in range(n - i - 1):
            if x[j] > x[j + 1]:
                temp = x[j]
                x[j] = x[j + 1]
                x[j + 1] = temp
    return x
```

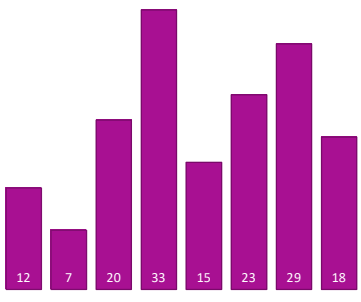
Python 实现

```
void bubble(int x[], int n) {
    /*
    x: 待排序的数组
    n: a中元素个数
    */
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n - i - 1; ++j)
            if (x[j] > x[j + 1]) {
                int temp = x[j];
                x[j] = x[j + 1];
                x[j + 1] = temp;
            }
}
```

C语言 实现

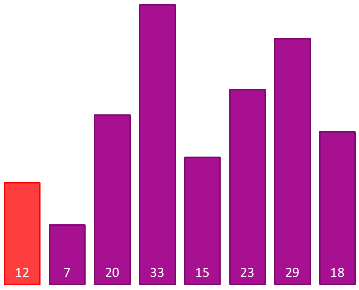
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



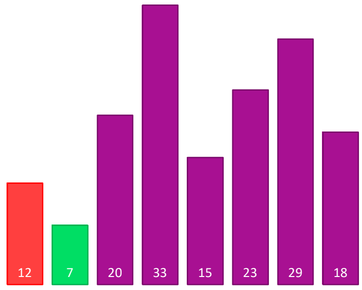
示例：冒泡排序

◦将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



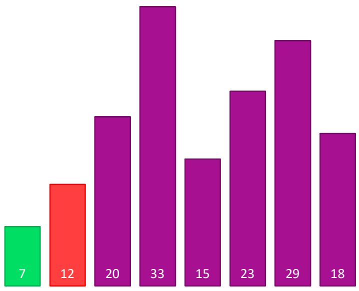
示例：冒泡排序

◦将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



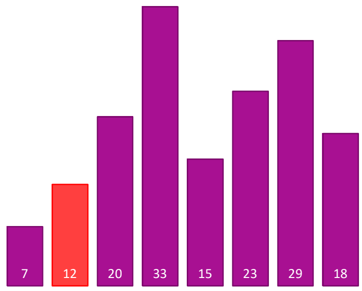
示例：冒泡排序

◦将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



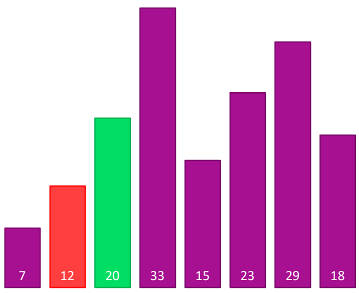
示例：冒泡排序

◦将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



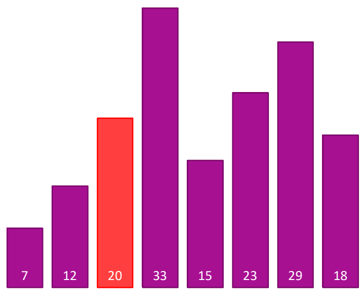
示例：冒泡排序

◦将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



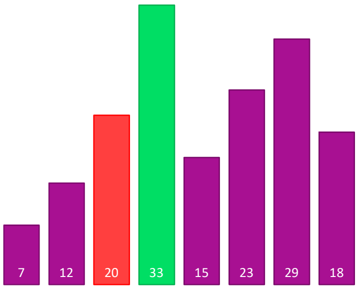
示例：冒泡排序

◦将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



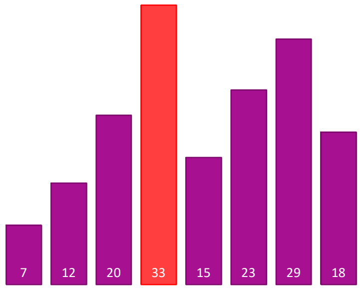
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



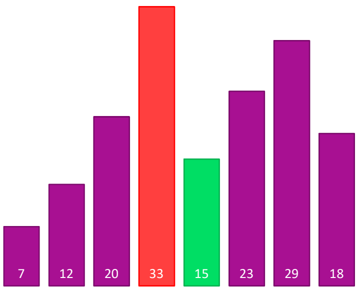
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



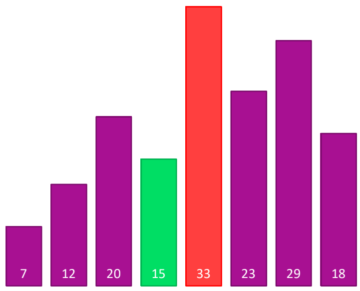
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



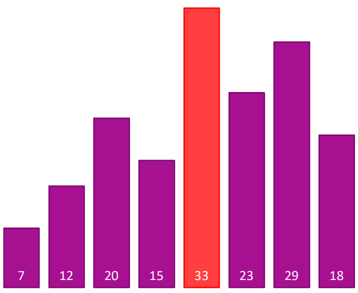
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



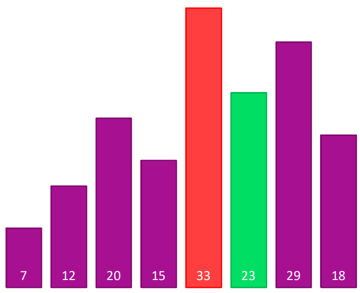
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



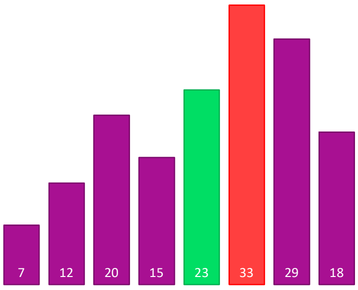
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



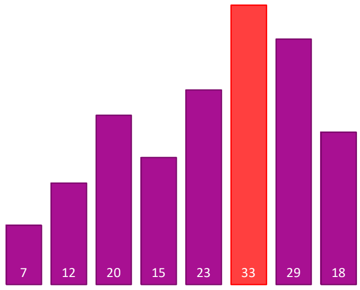
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



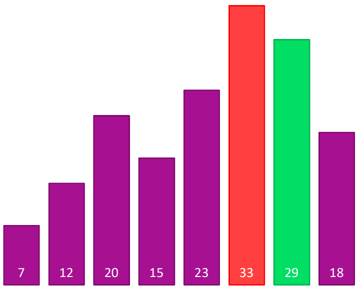
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



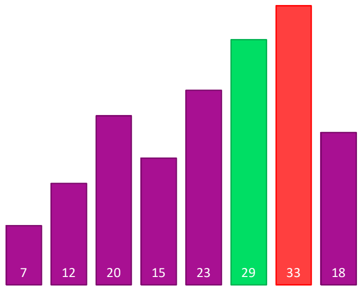
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



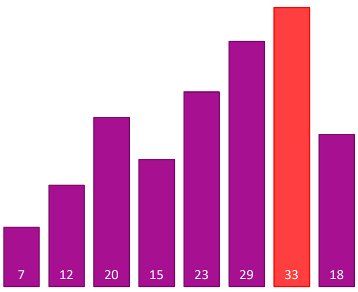
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



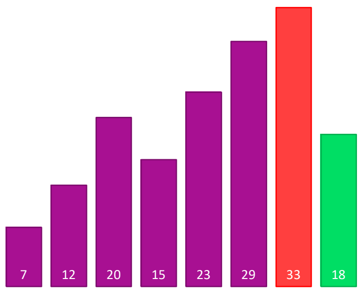
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



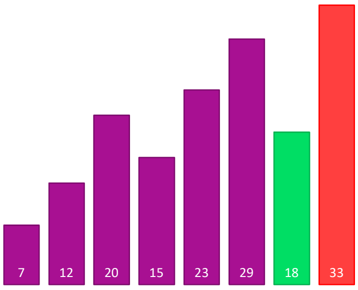
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



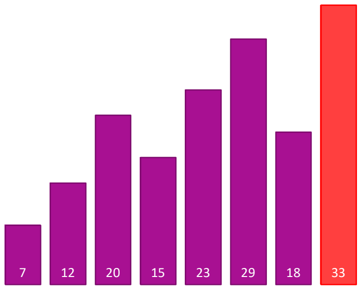
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



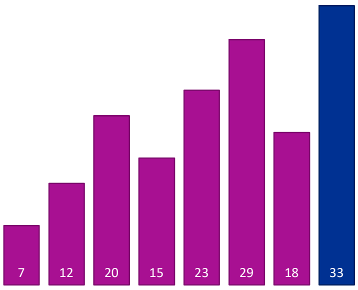
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



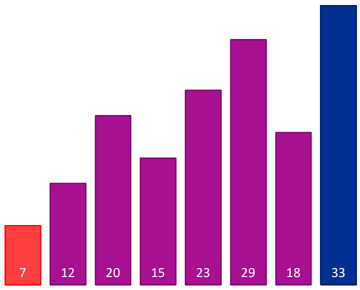
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



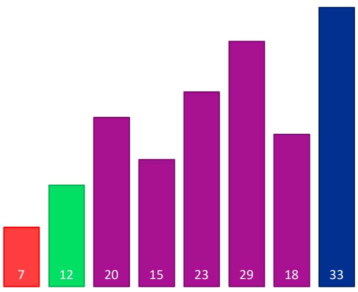
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



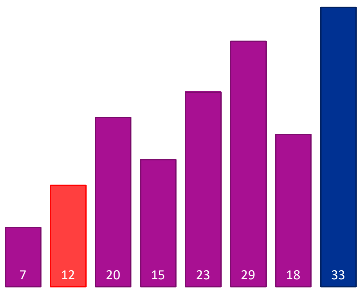
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



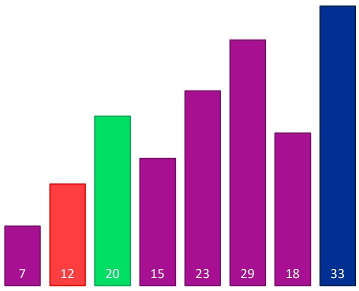
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



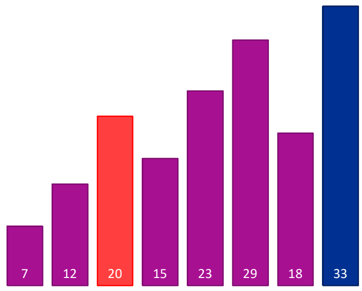
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



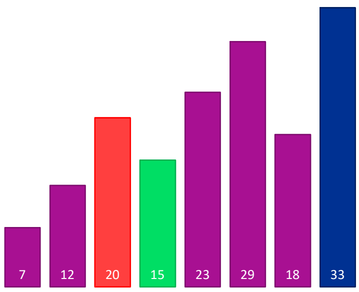
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



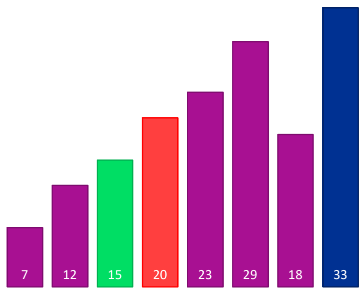
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



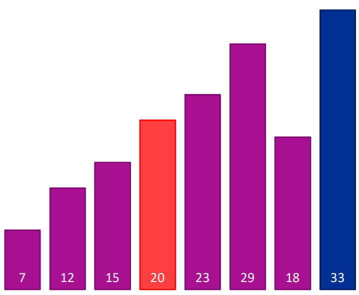
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



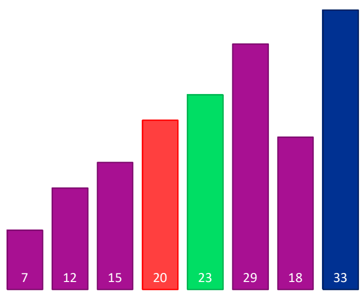
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



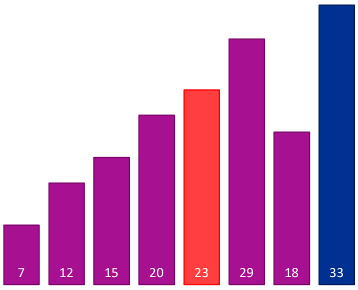
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



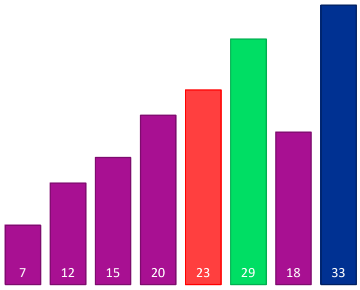
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



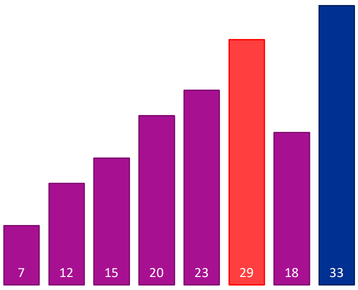
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



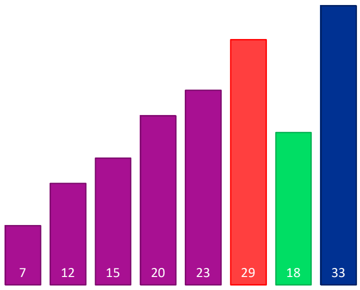
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



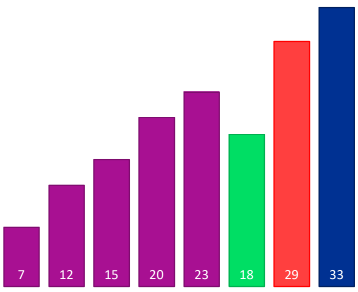
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



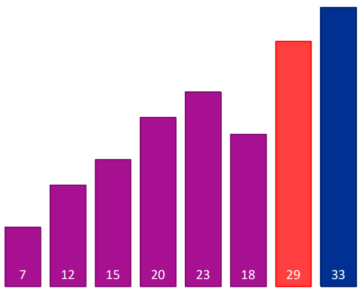
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



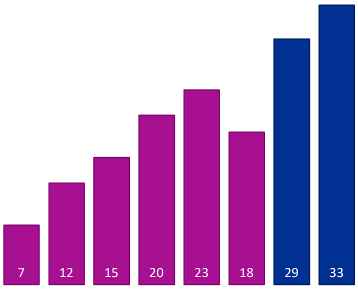
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



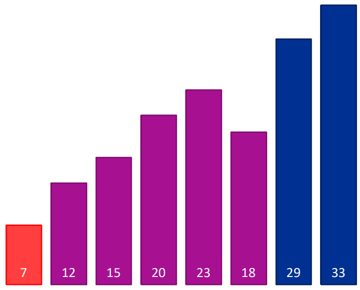
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



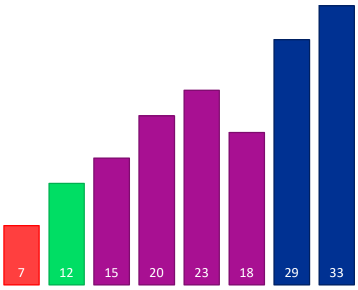
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



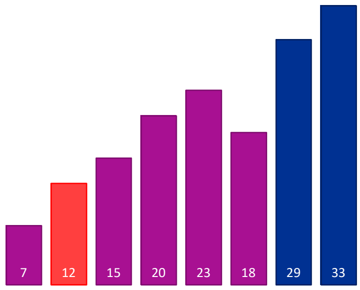
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



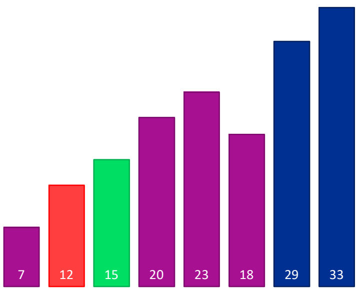
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



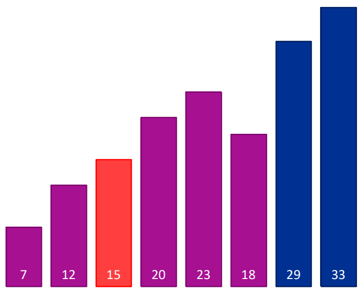
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



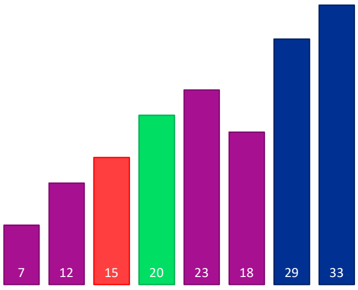
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



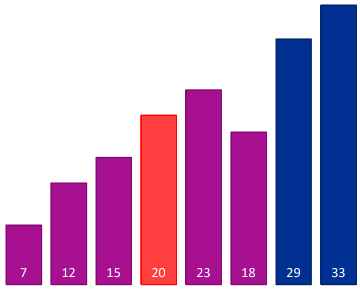
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



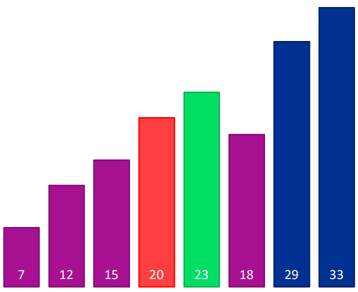
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



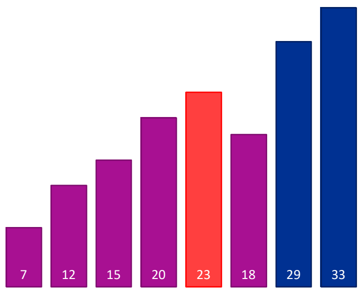
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



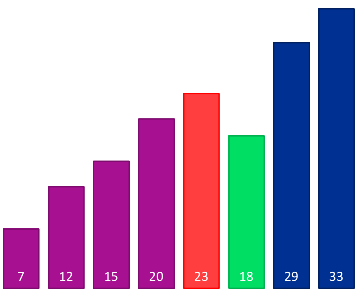
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



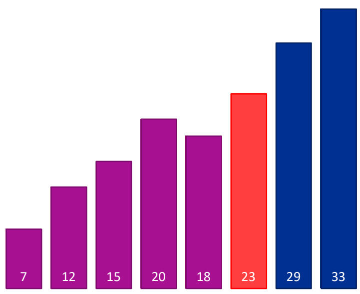
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



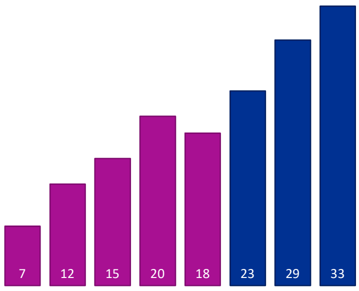
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



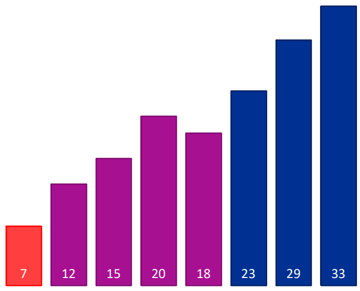
示例：冒泡排序

◦将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



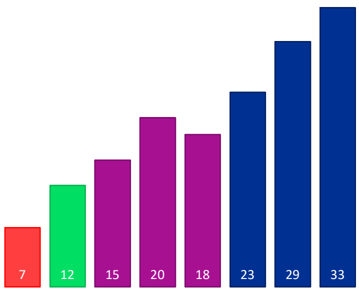
示例：冒泡排序

◦将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



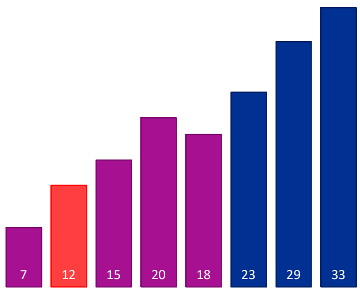
示例：冒泡排序

◦将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



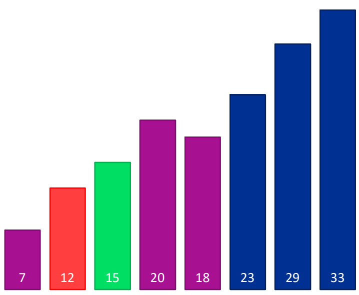
示例：冒泡排序

◦将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



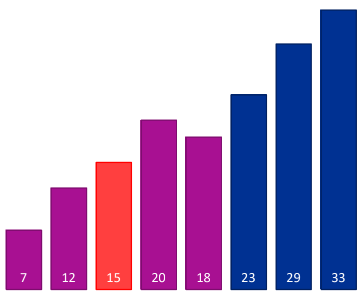
示例：冒泡排序

◦将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



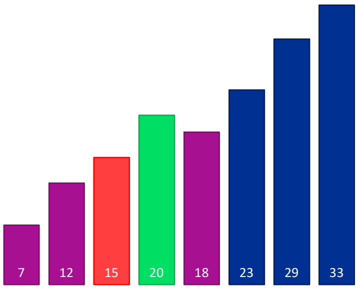
示例：冒泡排序

◦将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



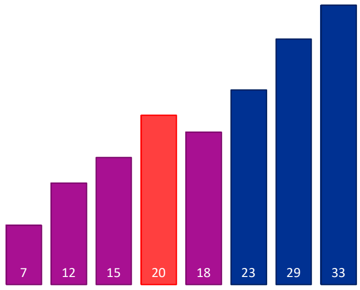
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



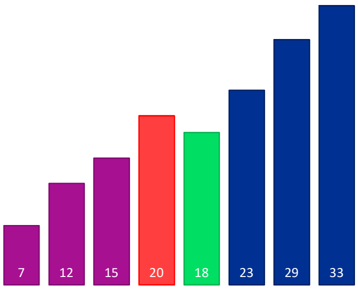
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



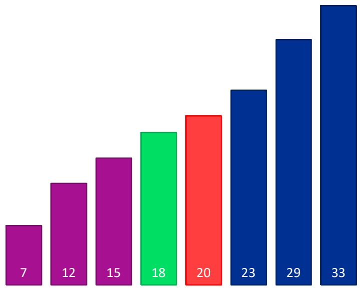
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



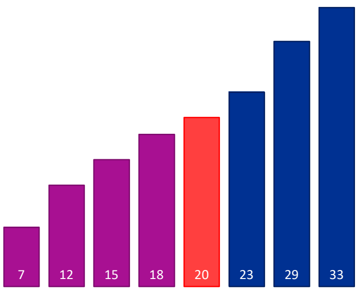
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



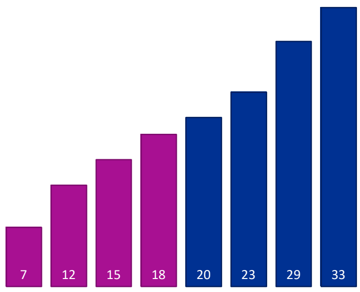
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



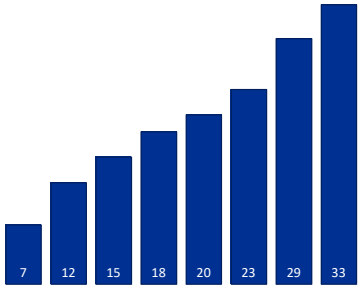
示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序



示例：冒泡排序

- 将数组[12, 7, 20, 33, 15, 23, 29, 18]按照递增的顺序排序

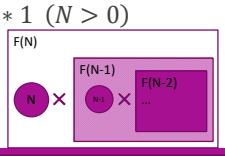


算法形式规范

- 函数的嵌套
  - 一个函数的函数体中包含一个或多个函数调用语句，即称为函数嵌套
    - 嵌套的含义是，如果函数A要调用函数B，也就是说，函数A的定义要依赖于函数B的定义。因此函数B的定义或函数B的原型必须出现在函数A的定义语句之前。
    - 另一方面，函数A调用函数B，在调用A的过程中，即执行A的函数体过程中，调用B，也就是中途把程序控制转到B的函数体，在执行结束后再返回到A的函数体中

算法形式规范

- 递归
  - 一般来说，一个函数或表达式在定义时又用到它自身，就说这个定义是递归的。
  - 就算法来说，如果一个算法的实现步骤中又需要调用它自己，就称这个算法是递归的。
  - 示例：
    - 整数N的阶乘是N的函数F(N)，它的定义是：
      - $F(0) = 1$
      - $F(N) = N * (N-1) * (N-2) * \ldots * 1 \quad (N > 0)$
    - 以递归的形式描述为：
      - $F(N) = N * F(N-1) \quad (N > 0)$



算法形式规范

```
long rfact(int n) {            // 计算n的阶乘
  assert ((n>=0) && (n<=12));
      // assert是一个宏，
      //当其后的逻辑表达式为真时，
      //继续执行下一条语句，否则退出执行
  if (n<=1) return 1;
  return  n * rfact(n-1);  // 递归调用
}
```

递归示例：折半查找（二分查找）

```
//递归法
int IterBiSearch(int data[], const int x, int beg, int last)
{
   int mid;
   if(beg <= last)
   {
   mid = (beg + last) / 2;
   if (x == data[mid])
      return mid;
   else if (x < data[mid])
      return IterBiSearch(data, x, beg, mid - 1);
   else if (x > data[mid])
      return IterBiSearch(data, x, mid + 1, last);
   }
   return -1;
}
```

算法

- 算法与问题
  - 计算（Computation）学科或称计算机科学把问题作为自己的研究对象
  - 用计算机来解决问题（Problem solving）称为问题求解
  - 算法要解决的是一类问题，算法的执行则针对的是问题的实例。每一个问题都可以看作是该问题的所有实例（instance）的集合
  - 一个问题的一个实例就是为计算该问题所需的一组输入
- 算法与程序
  - 程序(Program)可以用来描述算法，同一个算法可以用不同的语言编写的程序来描述。
  - 程序不一定是算法

性能分析

- 规范
- 对错
- 文档
- 逻辑
- 易懂
- 内存使用（空间）
- 运行时间（时间）

算法分析→算法复杂度的评判

算法复杂度 { 时间复杂度 / 空间复杂度

与算法执行时间相关的因素：

1. 算法选用的策略
2. 问题的规模
3. 编写程序的语言
4. 编译程序产生的机器代码的质量
5. 计算机执行指令的硬件速度
6. 程序运行的软件环境

有些算法在规模相同的情况之下，其语句频度会因输入的数据值或输入的数据顺序不同而不同，则时间复杂度也会不同，为此有最好、最坏和平均时间复杂度之分。

时间代价

- 评估算法的时间代价是算法分析的核心。算法的时间代价的大小用算法的时间复杂度(Time Complexity)来度量
- 但要考虑到以下因素：
  - 用来运行算法的计算机的性能的差别；
  - 算法运行的软件平台和描述语言的差别；
  - 算法所解的问题是多种多样的；
  - 同一问题的算法对不同的实例，所花的时间开销也可能有很大的差别

算法复杂性度量的基本操作

- 算法的时间代价的度量不应依赖于算法运行的硬件和软件平台，因此不能从下面几个方面来度量时间代价：
  - 算法运行的实际执行时间；
  - 运行过程中所执行的指令条数；
  - 运行过程中程序循环的次数
- 基本操作是指算法运行中起主要作用且花费最多时间的操作
- 例如：
  - 两个实数矩阵的乘法问题中，矩阵的实数元素之间的数乘是基本操作；
  - 对N个整数进行排序的算法中，整数间的比较是基本操作

问题实例长度

- 问题实例长度：算法运行的时间（或空间）代价还与问题实例长度，即输入规模有关，这称为问题实例长度。 问题实例长度是指作为该问题的一个实例的输入规模大小。
- 例如：
  - 排序问题：问题实例长度是待排序元素序列的长度n；
  - 矩阵乘积：问题实例长度是矩阵（指n阶方阵）的阶数n；
  - 图的最短路径问题：图G = <V,E>的顶点数 n =|V| 和边数 m =|E| 是问题实例长度；
  - 字符串匹配问题：文本T的长度n，亦可再加上样本P的长度m为问题实例长度。

时间代价

语句	程序步数
float sum (**float** list [], **int** n)	0
{	0
float tempsum = 0;	1
int i = 0;	1
for (i = 0; i < n; i++)	n+1
tempsum += list[i];	n
return tempsum;	1
}	0
程序总步数	2n+4

时间代价

语句	程序步数
void add (**int** a[][MAX_SIZE] …)	0
{	0
int i, j;	1
for (i = 0; i < rows; i++)	rows + 1
for (j = 0; j < cols; j++)	rows * (cols + 1)
c[i][j] = a[i][j] + b[i][j];	rows * cols
}	0
程序总步数	2rows*cols + 2rows + 1

时间代价

- 前面实现的冒泡排序的时间代价是多少呢？

空间代价

- 衡量算法优劣的一重要因素。算法的空间代价的大小用算法的空间复杂度（Space Complexity）来度量
- 空间复杂度的分析类似于时间复杂度的分析，其有关的概念和方法几乎是平行的。
- 固定和变长空间

空间代价

```
float abc(float a, float b, float c) {
   return a + b + b * c + (a + b - c) / (a + b) + 4.0;
}

float sum(float list[], int n) {
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
      tempsum += list[i];
   return tempsum;
}
```

空间代价

```
float rsum(float list[], int n) {
  if (n) return rsum(list, n - 1) + list[n - 1];
  return 0;
}
```

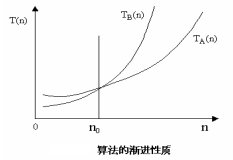
类别	名称	字节数
参量：数组指针	list[]	4
参量：整数	n	4
返回地址：（内部使用）		4
每次递归所需空间总和		**12**

复杂度函数及其渐进性质

- 算法的时间（或空间）代价由该算法用于问题长度为n的实例所需要的基本操作次数来刻画。
- 一般一个算法的时间复杂度用函数T(n)表示，空间复杂度用S(n)表示。
- 算法的比较是根据复杂度函数的渐进性质的比较进行的。

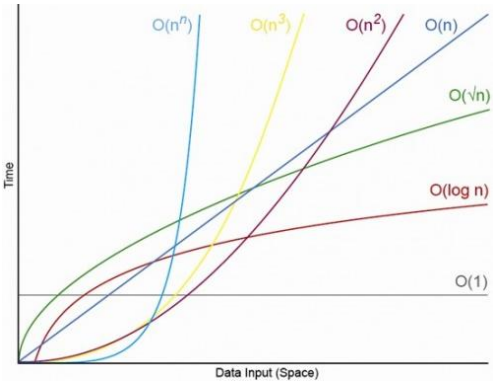
复杂度函数及其渐进性质

- 例如：算法A和算法B，其时间复杂度函数为$T_A(n)$和$T_B(n)$，在下图中，虽然当$n < n_0$时$T_B(n) < T_A(n)$，但在$n > n_0$时，或说n充分大时$(n \to \infty)$，有$T_A(n) < T_B(n)$，因此认为算法A优于算法B。



算法的渐进性质

复杂度



最坏情形和最好情形

例：插入排序算法 Insertsort

n = 10 时，有三种输入：

- 正序： 1 2 3 4 5 6 7 8 9 10　9次比较
- 逆序： 10 9 8 7 6 5 4 3 2 1　45次比较
- 随机： 3 7 4 5 10 2 9 6 1 8　28次比较

同一算法，相同的问题长度，不同的输入，其时间代价一般不同。

因此在实际的算法分析中，复杂度函数值T(n)**不是唯一**的，在大多数情况下取其**最大值**，即最坏情形的时间（空间）复杂度。

最坏情形和最好情形

语句	程序步数
void bubble (**int** x[], **int** n)	0
{ // 冒泡排序	0
for (**int** i = 0; i < n; ++i)	n+1
for (**int** j = 0; j < n - i - 1; ++j)	n(n+1)/2
if (x[j] > x[j + 1])	n(n-1)/2
{	0
int temp = x[j];	[0, n(n-1)/2]
x[j] = x[j + 1];	[0, n(n-1)/2]
x[j + 1] = temp;	[0, n(n-1)/2]
}	0
}	0
程序总步数	$[n^2+n+1, (5/2)n^2-(1/2)n+1]$

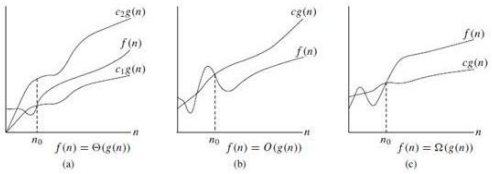
渐近性定义

- 对于算法的复杂度的研究主要是侧重在其渐进性质，即当问题长度n比较大的情形。
- 为了简化算法复杂度分析的方法，往往只需计算当问题规模较大时算法的渐进复杂度的阶。
- 称（复杂度）函数f(n)是O(g(n))的，即$f(n) = O(g(n))$，如果存在常数$c > 0$与n_0，当$n \geq n_0$时有$f(n) \leq cg(n)$。
  - 例如：$T_1(n) = (n + 1) / 2 = O(n)$，$T_2(n) = 3n^2 + 4n + 5 = O(n^2)$

渐近性定义

- 称（复杂度）函数f(n)是Ω(g(n))的，即$f(n) = \Omega(g(n))$，如果存在常数$c > 0$与n_0，当$n \geq n_0$时有$f(n) \geq cg(n)$。
  - 例如：$T1(n) = (n + 1) / 2 = \Omega(n)$，$T2(n) = 3n^2 + 4n + 5 = \Omega(n^2)$
- 称（复杂度）函数f(n)是θ(g(n))的，即$f(n) = \theta(g(n))$，如果存在常数$c_1, c_2 > 0$与n_0，当$n \geq n_0$时有$c_1g(n) \geq f(n) \geq c_2g(n)$。
  - 例如：$T1(n) = (n + 1) / 2 = \theta(n)$，$T2(n) = 3n^2 + 4n + 5 = \theta(n^2)$

渐近性定义



显然，如果$f(n) = O(g(n))$且$f(n) = \Omega(g(n))$，则$f(n) = \theta(g(n))$。