第一章：绪论

1. 什么是数据结构？

1.1 数据结构概述

数据（data）是指能输入计算机并能被计算机程序识别和处理的符号。我们可以将数据分为两大类，一类是整数、小数等数值数据，另一类是文字、声音和图像等非数值数据。在生活中，学生的成绩、身高和体重等是数据，学生的照片、指纹和语音指令等也都是数据。

数据元素（data element）是数据的基本单位，在计算机程序中通常作为一个整体来进行考虑和处理。数据元素由任意多个**数据项（data item）**组成，数据项是构成数据元素不可分割的最小单位，性质相同的数据元素通常具有相同个数和相同类型的数据项组成。在不同的应用场合中，我们把数据元素又称为元素、结点、顶点等等。

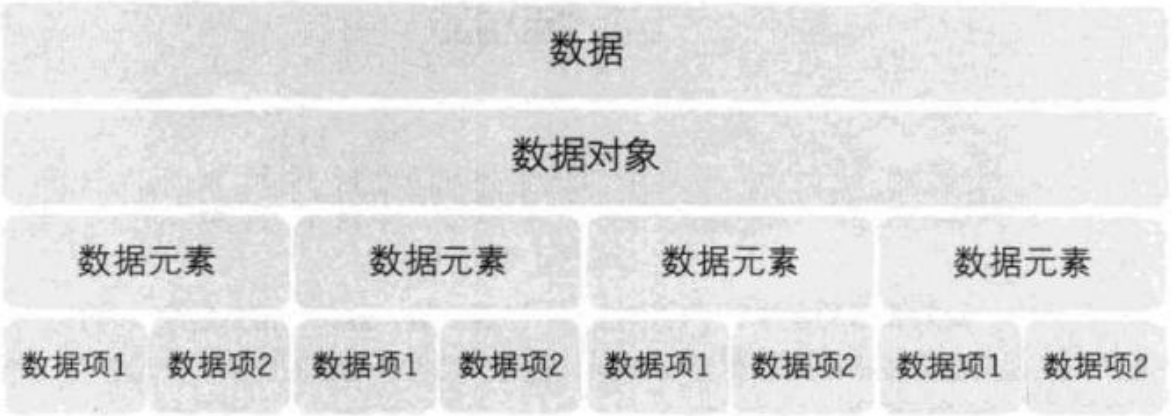
例如，以学籍登记表为例，每个学生的档案就是一个数据元素，而档案中的学号、姓名、出生日期等都是数据项，如下图所示。

数据元素 → 003 王五 女 1998/11/27 团员

数据项 → 1998/11/27

学号	姓名	性别	出生日期	政治面貌
001	张三	男	1999/09/02	团员
002	李四	女	2000/03/25	党员
003	王五	女	1998/11/27	团员
004	赵六	男	1999/07/09	党员

数据对象（data object）是性质相同的数据元素的集合，是数据的子集。例如学籍登记表中的所有学生的档案集合就是数据对象。数据、数据对象、数据元素和数据项的关系图如下：



数据结构（data structure）就是指相互之间存在一种或多种特定关系的数据元素的集合。或者说，数据结构是带结构的数据元素集合。在任何问题中，数据元素之间都不是孤立存在的，它们之间存在着某种关系，数据元素相互之间的关系我们就称之为结构（structure）。

按照视点不同，数据结构分为数据的逻辑结构和数据的存储结构。

1.2 数据的逻辑结构

数据的逻辑结构反映的是数据元素之间的逻辑关系，其中的逻辑关系是指数据元素之间的前后间关系，而与他们在计算机中的存储位置无关。

- **划分方式一**

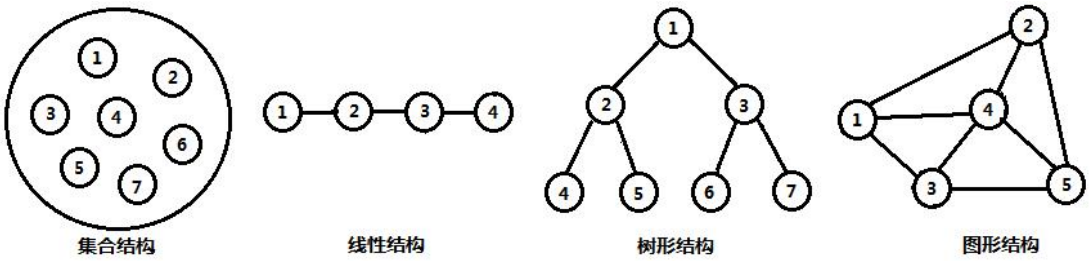
按照四类的基础逻辑结构划分，数据的逻辑结构包括：

(1) 集合结构：数据结构中的元素之间除了在“同属一个集合”的关系外，别无其它关系；

(2) 线性结构：数据结构中的元素存在“一对一”的线性关系，例如冰糖葫芦；

(3) 树形结构：数据结构中的元素存在“一对多”的层次关系，例如公司组织架构；

(4) 图形结构或网状结构：数据结构中的元素存在“多对多”的任意关系，例如地图。



- **划分方式二**

按照线性和非线性划分，数据的逻辑结构包括：

(1) 线性结构

在线性结构中，有且仅有一个开始和终端结点，并且所有节点都最多有一个直接前驱和一个直接后继。也就是，数据元素之间存在“一对一”的关系。

常见的线性结构有：数组、链表、队列、栈等，后面我们会详细的讲解。

(2) 非线性结构

简单地说，非线性结构就是表中各个结点之间具有多个对应关系(即一对多关系和多对多关系)，在非线性结构的一个结点可能有多个直接前驱结点和多个直接后继结点。

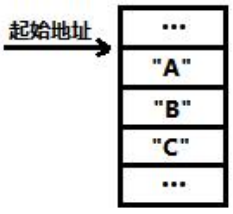
常见的非线性结构有：二维数组、多维数组、树结构和图结构等，后面我们会详细的讲解。

1.3 数据的物理结构

数据的逻辑结构在计算机存储空间中的存放形式称为数据的物理结构，或称为数据的存储结构。常见的存储结构有：顺序存储结构、链式存储结构、索引存储结构和散列存储结构。

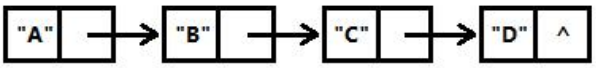
- **顺序存储结构**

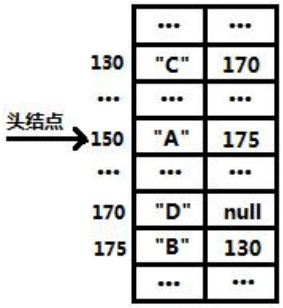
用一组连续的存储空间单元来依次的存储数据元素，数据元素之间的逻辑关系由存储位置来表示。例如：在 java 语言中，数组采用的就是顺序存储结构。



- **链式存储结构**

用一组任意的存储单元来存储数据元素，通过保存地址的方式找到相关联的数据元素，数据元素之间的逻辑关系用引用（指针）来表示。例如：数据结构中链表采用的就是链式存储结构。





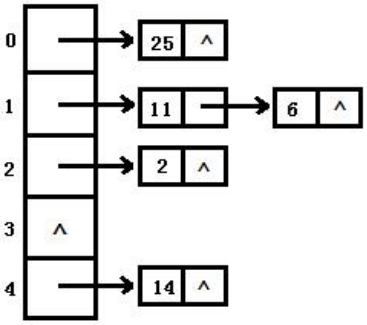
- **索引存储结构**

除建立存储结点信息外，还建立附加的索引表来标识结点的地址。例如：图书目录、字典的目录、通讯目录等等。

- **散列存储结构**

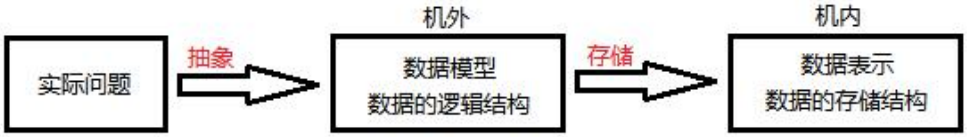
根据结点的关键字直接计算出该结点的存储地址。例如：java 集合中的 HashSet 和 HashMap 采用的都是散列存储结构，一种神奇的结构，添加、查询速度快。

key	25	11	2	6	14
key%5	0	1	2	1	4



- **逻辑结构和物理结构总结**

逻辑结构是从具体问题抽象出来的数据模型，是面向问题的，反应了数据结构的关联方式或邻接关系；物理结构，指的就是逻辑结构在计算机中的存储形式，是面向计算机的，其目标是将数据及逻辑关系存储到计算机中。



一般来说，一种数据结构可以用多种存储结构来存储，而采用不同的数据存储结构，其数据的处理效率往往不同。

2. 什么是算法？

2.1 算法的简介

- **算法的定义**

从生活的角度上来讲，算法就是解决问题的方法，现实生活中关于算法的示例不胜枚举，例如做一道菜的步骤、一个安装旋转座椅的操作说明等等。

从计算机的角度来讲，算法是对特定问题求解步骤的一种描述，是指令的有限序列。简单的说，算法就是计算机解题的过程。

- **算法的五大特性**

(1) 输入性：一个算法有 0 个或多个输入，以刻画运算对象的初始情况，所谓 0 个输入是指算法本身定出了初始条件。

(2) 输出性：一个算法有一个或多个输出，以反映对输入数据加工后的结果。没有输出的算法是毫无意义的。

(3) 可行性：算法是可行的，即算法中的每一条指令都是可以实现的，均能在有限时间内完成。

(4) 有穷性：算法执行的指令个数是有限的，必须能在执行有限个指令后终止。

(5) 确定性：算法对于特定的合法输入，对应的输出是唯一的。也就是，对于相同的输入肯定会得出相同的结果输出。

- **评价算法优劣的依据**

举例：如何计算 1+2+3+...+100 的结果？
算法 1：通过循环，依次累加来实现。耗费时间
算法 2：使用递归来实现。耗费内存
算法 3：高斯解法，首尾相加*50。

同一问题可用不同的算法来解决，而一个算法的质量优劣将影响到算法乃至程序的效率。因此，我们学习算法目的在于选择合适算法和改进算法，一个算法的评价主要从时间复杂度和空间复杂度来考虑。

(1) 时间复杂度：评估执行程序所需的时间，可以估算出程序对处理器的使用程度。

(2) 空间复杂度：评估执行程序所需的存储空间，可以估算出程序对计算机内存的使用程度。

2.2 数据结构和算法的关系

两者既有联系又有区别，联系是“程序=数据结构+算法”。数据结构是算法实现的基础，算法总是要依赖某种数据结构来实现的，本质上算法的操作对象就是数据结构。区别是数据结构关注的是数据的逻辑结构、存储结构相关的基本操作，而算法更多的是关注如何在数据结构的基础上解决实际问题。

所以说，算法是编程思想，数据结构则是这些思想的基础，高效的程序需要在数据结构的基础上设计和选择算法。

3. 算法的时间复杂度

3.1 时间复杂度的计算

- **如何衡量算法的执行时间？**

衡量算法的执行时间，通常有两种方法。

(1) 事后统计的方法

这种方法理论上是可行的，但不是一个最好的解决方案。该方法有两个缺陷：（一）要想对设计的算法的运行性能进行评测，必须先依据问题编写出相应的算法并实际运行。（二）所得算法的执行时间依赖于计算机的硬件、软件等环境因素，有时容易掩盖算法本身的优劣。

因为事后统计方法更多的依赖于计算机的硬件、软件等环境因素，有时容易掩盖算法本身的优劣。因此，我们更多采用事前估算的方法来衡量算法的执行时间。

(2) 事前估算的方法

在编写程序前，通过分析某个算法的时间复杂度来判断哪个算法更优。

- **时间频度的介绍**

一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。一个算法中的语句执行次数称为语句频度或时间频度，记为 T(n)。

示例：计算以下代码的时间频度 T(n)

```
public void test(int n) {
    int sum = 0;
    for(int i = 0; i < n; i++) {
        sum += i;
    }
}
```

在时间频度 T(n)中，n 称为问题的规模，当 n 不断变化时，时间频度 T(n)也会不断变化。如果我们想知道它变化时呈现什么规律。为此，我们引入时间复杂度概念。

- **时间复杂度的定义**

一般情况下，算法中基本操作重复执行的次数是问题规模 n 的某个函数，用 T(n)表示，若有某个辅助函数 f(n)，使得当 n 趋近于无穷大时，T(n)/f(n)的极限值为不等于零的常数，则称 f(n)是 T(n)的同数量级函数。记作 T(n)=O(f(n))，我们也称 O(f(n))为算法的渐进时间复杂度，简称时间复杂度。

- **时间复杂度的计算步骤**

(1) 计算基本操作的执行次数 T(n)

在做算法分析时，一般默认考虑最坏的情况。

(2) 计算 T(n)的数量级 f(n)

求 T(n)的数量级 f(n)，只需要将 T(n)做两个操作：（一）忽略常数项、低次幂项和最高次幂项的系数。（二）f(n)=(T(n)的数量级)。例如，在 T(n)=4n^2+2n+2 中，T(n)的数量级函数 f(n)=n^2。

计算 T(n)的数量级 f(n)，我们只要保证 T(n)中的最高次幂正确即可，可以忽略所有常数项、低次幂项和最高次幂的系数。这样能够简化算法分析，将注意力集中在最重要的一点上：增长率。

(3) 用大 O 表示时间复杂度

当 n 趋近于无穷大时，如果 T(n)/f(n)的极限值为不等于零的常数，则称 f(n)是 T(n)的同数量级函数。记作 T(n)=O(f(n))。例如，在 T(n)=4n^2+2n+2 中，就有 f(n)=n^2，使得 T(n)/f(n)的极限值为 4，也就是得到时间复杂度为 O(n^2)。

切记，时间频度不相同时，但是时间复杂度有可能相同，如 T(n)=n^2+3n+4 与 T(n)=4n^2+2n+1 它们的时间频度不同，但时间复杂度相同，都为 O(n^2)。

3.2 常见的时间复杂度介绍

常见的时间复杂度有：常数阶 O(1)，对数阶 O(log2n)，线性阶 O(n)，线性对数阶 O(nlog2n)，平方阶 O(n^2)，立方阶 O(n^3)，指数阶 O(2^n)和阶乘阶 O(n!)。

接下来，我们就来学习这些常见的时间复杂度。

- **常数阶 O(1)**

无论代码执行了多少行，只要是没有循环等复杂结构，那这个代码的时间复杂度就都是 O(1)。

```
int num1 = 3, num2 = 5;
int temp = num1;
num1 = num2;
num2 = temp;
System.out.println("num1:" + num1 + " num2:" + num2);
```

在上述代码中，没有循环等复杂结构，它消耗的时间并不随着某个变量的增长而增长，那么无论这类代码有多长，即使有几万几十万行，都可以用 O(1)来表示它的时间复杂度。

- **对数阶 O(log2n)**

O(log2n)指的就是：在循环中，每趟循环执行完毕后，循环变量都放大两倍。

```
int n = 1024;
for(int i = 1; i < n; i *= 2) {
    System.out.println("hello whsxt");
```

```
}
```

推算过程：假设该循环的执行次数为 x 次（也就是 i 的取值为 2^x），就满足了循环的结束条件，即满足了 2^x 等于 n，通过数学公式转换后，即得到了 x = log2n，也就是说最多循环 log2n 次以后，这个代码就结束了，因此这个代码的时间复杂度为：O(log2n) 。

同理 如果每趟循环执行完毕后 循环变量都放大 3 倍 那么该代码的时间复杂度为 :O(log3n) 。

- **线性阶 O(n)**

```
int n = 100;
for(int i = 0; i < n; i++) {
    System.out.println("hello whsxt");
}
```

在上述代码中，for 循环会执行 n 趟，因此它消耗的时间是随着 n 的变化而变化的，因此这类代码都可以用 O(n)来表示它的时间复杂度。

- **线性对数阶 O(nlog2n)**

```
int n = 100;
for(int i = 1; i <= n; i++) {
    for(int j = 1; j <= n; j *= 2) {
        System.out.println("hello whsxt");
    }
}
```

线性对数阶 O(nlog2n) 其实非常容易理解，将时间复杂度为 O(log2n)的代码循环 n 遍的话，那么它的时间复杂度就是 n*O(log2n)，也就是了 O(nlog2n)。

- **平方阶 O(n^2)**

```
int n = 100;
for(int i = 1; i <= n; i++) {
    for(int j = 1; j <= n; j++) {
        System.out.println("hello whsxt");
    }
}
```

外层 i 的循环执行一次，内层 j 的循环就要执行 n 次。因为外层执行 n 次，那么总的就需要执行 n*n 次，也就是需要执行 n^2 次。因此这个代码的时间复杂度为：O（n^2）。

平方阶的另外一个例子：

```
int n = 100;
for(int i = 1; i <= n; i++) {
    for(int j = i; j <= n; j++) {
        System.out.println("hello whsxt");
    }
}
```

当 i=1 的时候，内侧循环执行 n 次，当 i=2 的时候，内侧循环执行（n-1）次，……一直这样子下去就可以构造出一个等差数列：n+（n-1）+（n-2）+……+2+1 ≈ (n^2)/2。根据大 O 表示法，去掉最高次幂的系数，就可以得到时间复杂度为：O（n^2）。

同理，立方阶 O(n^3)，参考上面的 O(n^2)去理解，也就是需要用到 3 层循环。

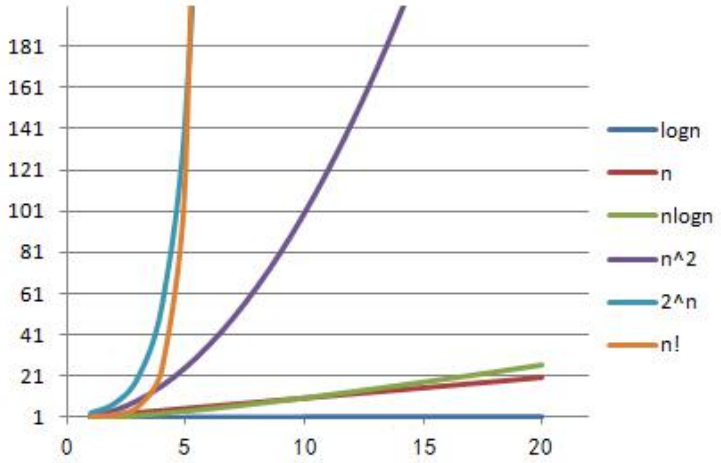
- **指数阶 O(2^n)和阶乘阶 O(n!)**

指数阶 O(2^n)指的就是：当 n 为 10 的时候，需要执行 2^10 次。

阶乘阶 O(n!)指的就是：当 n 为 10 的时候，需要执行 10*9*8*...*2*1 次。

➢ **常见的时间复杂度耗时比较**

算法的时间复杂度是衡量一个算法好坏的重要指标。一般情况下，随着规模 n 的增大，T(n)的增长较慢的算法为最优算法。



其中 x 轴代表 n 值，y 轴代表 T(n)值。T(n)值随着 n 的值的变化而变化，其中可以看出 O(n!)和 O(2^n)随着 n 值的增大，它们的 T(n)值上升幅度非常大，而 O(logn)、O(n)、O(nlogn)、O(n^2)随着 n 值的增大，T(n)值上升幅度相对较小。

常用的时间复杂度按照耗费的时间从小到大依次是：O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(n^3) < O(2^n) < O(n!)。

3.3 最好、最坏和平均时间复杂度

➢ **最好和最坏时间复杂度**

最好情况时间复杂度就是在最理想的情况下，执行这段代码的时间复杂度。

最坏情况时间复杂度就是在最糟糕的情况下，执行这段代码的时间复杂度。

来看看下面这段代码：

```
/**
 * 获取元素 element 在数组 arr 中的索引值
 * @return 如果 element 在数组中存在，则返回对应的索引，否则返回-1
 */
public int find(int[] arr, int element) {
    for(int i = 0; i < arr.length; i++) {
        if(arr[i] == element)
            return i;
    }
    return -1;
}
```

因为元素 element 在数组中的位置是不确定的，有可能数组中的第一个元素就是 element，那就就意味着只需循环一次即可，其时间复杂度就是 O(1)；如果数组中不存在元素 element 或者是数组中的最后一个元素才是 element，那就需要遍历整个数组，时间复杂度就是 O(n)。

在这里，O(1) 就是最好情况时间复杂度，O(n) 就是最坏情况时间复杂度。

- **平均情况的时间复杂度**

借助上面的例子继续来分析，要查找的元素 element 在数组中的位置，有 n+1 种情况：在数组的[0，n-1]索引位置中和不在数组中。

在这里我们引入概率论的相关知识，假设元素 element 在数组中与不在数组中的概率各为 1/2，并且假设出现在索引[0, n-1]这 n 个位置的概率都是 1/n。根据概率乘法法则，要查找的元素 element 出现在[0, n-1]中任意位置的概率就是 1/2n。到这里 ,就可以得到这样的计算公式 :1*(1/2n) + 2*(1/2n) + 3*(1/2n) + …… + n*(1/2n) + n*(1/2) = (3n + 1)/4。得到的这个值就是概率论中的加权平均值，也叫做期望值。根据这个加权平均值，去掉常数项、低次幂项和最高次幂项的系数，我们得到的平均时间复杂度也是 O(n)。

所以，平均时间复杂度就是：加权平均时间复杂度（亦称为期望时间复杂度）。

- **最好、最坏和平均时间复杂度总结**

算法中，如果不做特别的说明，我们讨论的时间复杂度均是最坏情况下的时间复杂度。这样做的原因是：最坏情况下的时间复杂度是算法在任何输入实例上运行时间的界限，这就保证了算法的运行时间不会比最坏情况更长。

4. 算法的空间复杂度

4.1 算法的空间复杂度介绍

算法在运行过程中所需的存储空间包括：（1）输入输出数据占用的空间；（2）算法本身占用的空间；（3）执行算法所需的辅助空间。其中输入和输出数据占用的空间取决于问题，与算法无关；算法本身占用的空间虽然与算法相关，但是一般其大小是固定的。所以，算法的空间复杂度(Space Complexity)是指算法在执行过程中需要的辅助空间，也就是除算法本身和输入输出数据占用的空间外，算法零时开辟的存储空间。

如果算法所需的辅助存储空间的数量是问题规模 n 的函数，通常计作：S(n)=O(f(n))。其中，n 为问题的规模，分析方法与算法的时间复杂度类似。

案例一：计算嵌套循环执行次数

```
public static int method(int n) {
    int i = 0, j = 0, count = 0;
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            count++;
        }
     }
    return count;
}
```

在以上代码中，由于算法中临时变量的个数与问题规模 n 无关，所以空间复杂度均为 O(1)。

案例二：计算 1+2+3+...+n 的结果

```
public static int sum(int n) {
    if(n == 1)
        return 1;
    else
        return n + sum(n - 1);
}
```

以上的案例采用了递归，每次调用本身都要分配空间，所以空间复杂度为 O(n)。

在做算法分析时，因为时间复杂度要比空间复杂度更容易出问题，所以一般情况下我们更多对时间复杂度进行研究。 从用户使用体验上看，更看重的程序执行的速度。一些缓存产品（redis）和算法（基数排序）本质就是使用空间来换时间。

另外，一般面试或者工作的时候没有特别说明的话，算法复杂度就是指时间复杂度。

4.2 时间复杂度和空间复杂度总结

举例：金融港到关谷广场有很多条路。如果选择路程近的，那么可能会堵车耗时（时间复杂度的性能变差）；如果选择耗时短的，那么可能会绕路（空间复杂度的性能变差）。那么到底选择走哪一条路，就需要根据实际情况来综合考虑了。

编程算法中亦是如此，时间复杂度和空间复杂度往往是相互影响的。当追求一个较好的时间复杂度时，可能会使空间复杂度的性能变差，即可能导致占用较多的存储空间；相反的当追求一个较好的空间复杂度时，就可能会使时间复杂度的性能变差，即可能导致占用较长的运行时间。

因此，当设计一个算法（特别是大型算法）时，要综合考虑算法的各项性能，算法的使用频率，算法处理的数据量的大小，算法描述语言的特性，算法运行的机器系统环境等各方面因素，才能够设计出比较好的算法。