目录

上午		
第一讲	数据结构和算法基础	复杂度、数组、链表、递归、尾递归
第二讲	排序与二分查找	排序特性、排序问题实战、二分查找
下午		
第三讲	动态数据结构	跳表、散列表、二叉树、堆
第四讲	字符串匹配算法	字符串算法回顾、Trie 树、实战
第五讲	图及相关算法	图的概念与存储、图相关算法介绍
第六讲	算法思想	贪心、分治、回溯、动态规划

第一讲：基础数据结构和编程技巧

王争

前 Google 工程师

目录

复杂度分析

复杂度分析

1. 最坏时间复杂度

2. 最好时间复杂度

3. 平均时间复杂度

4. 均摊时间复杂度

均摊复杂度分析

```
// array表示一个长度为 n 的数组
int[] array = new int[n];
int count = 0;

void insert(int val) {
   if (count == n) {
      int sum = 0;
      for (int i = 0; i < n; ++i) {
         sum = sum + array[i];
      }
      System.out.println(sum);
      count = 0;
   }

   array[count] = val;
   ++count;
}
```

insert() 在大部分情况下，时间复杂度都为 O(1)。只有个别情况下，复杂度才比较高，为 O(n)。

对于 insert() 函数来说，O(1) 时间复杂度的插入和 O(n) 时间复杂度的插入，出现的频率是非常有规律的，而且有一定的前后时序关系，一般都是一个 O(n) 插入之后，紧跟着 n-1 个 O(1) 的插入操作，循环往复。

O(1)	.....	O(1)	O(1)	O(n)	O(1)	.....	O(1)	O(1)	O(n)
O(2)	.....	O(2)	O(2)	O(1)	O(2)	.....	O(2)	O(2)	O(1)

均摊复杂度分析

均摊复杂度分析适用的场景总结：

对一个数据结构进行一组连续操作中，大部分情况下时间复杂度都很低，只有个别情况下时间复杂度比较高，而且这些操作之间存在前后连贯的时序关系。

这个时候，我们就可以将这一组操作放在一起分析，看是否能将较高时间复杂度的那次操作的耗时平摊到其他时间复杂度比较低的操作上。

在能够应用均摊时间复杂度分析的场合，一般均摊时间复杂度就等于最好情况时间复杂度。

数组

编程语言中的数组

数组中的数据，真的必须连续存储吗？真的必须类型相同吗？

JavaScript:

```
var arr = new Array(4,'hello', new Date());
```

Java:

```
int arr[][] = new int[3][];
arr[0] = new int[1];
arr[1] = new int[2];
arr[2] = new int[3];
```

编程语言中的数组 不等于 数据结构中的数组

编程语言中的 “数组” 并不完全等同于我们在讲数据结构和算法的时候提到的 “数组”。
编程语言在实现自己的 “数组” 类型的时候，并不是完全遵循数据结构 “数组” 的定义，
而是针对编程语言自身的特点做了调整。

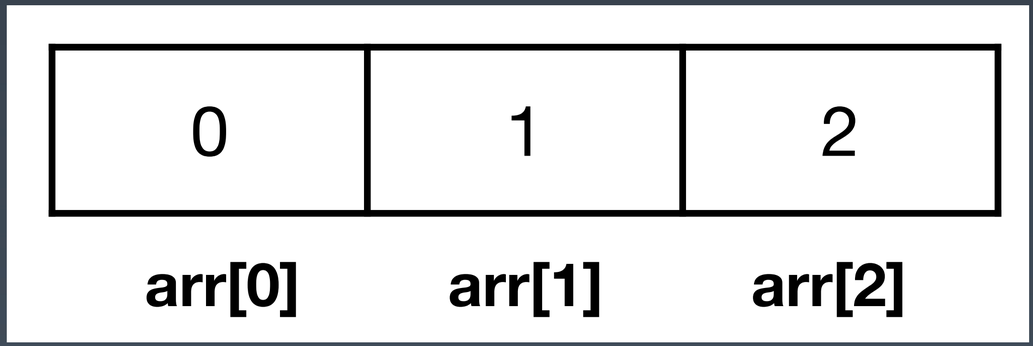
1. C/C++ 中数组的实现方式

2. Java 中数组的实现方式

3. JavaScript 中数组的实现方式

C/C++ 中数组的实现方式：基本数据类型数组

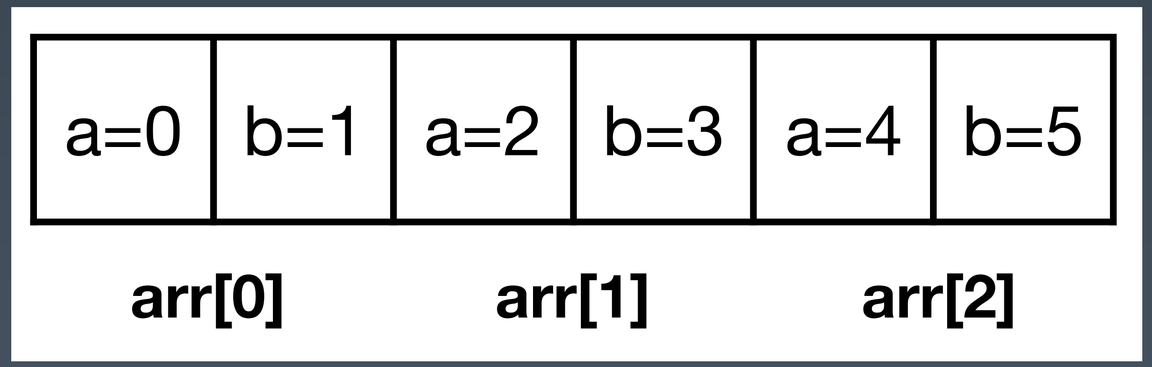
```
int arr[3];
arr[0] = 0;
arr[1] = 1;
arr[2] = 2;
```



C/C++ 中数组的实现方式：对象数组

```
struct Dog {
  char a;
  char b;
};

struct Dog arr[3];
arr[0].a = '0'; arr[0].b = '1'; // 为了节省页面，放到了一行里
arr[1].a = '2'; arr[1].b = '3';
arr[2].a = '4'; arr[2].b = '5';
```



C/C++ 中数组的实现方式：二维数组

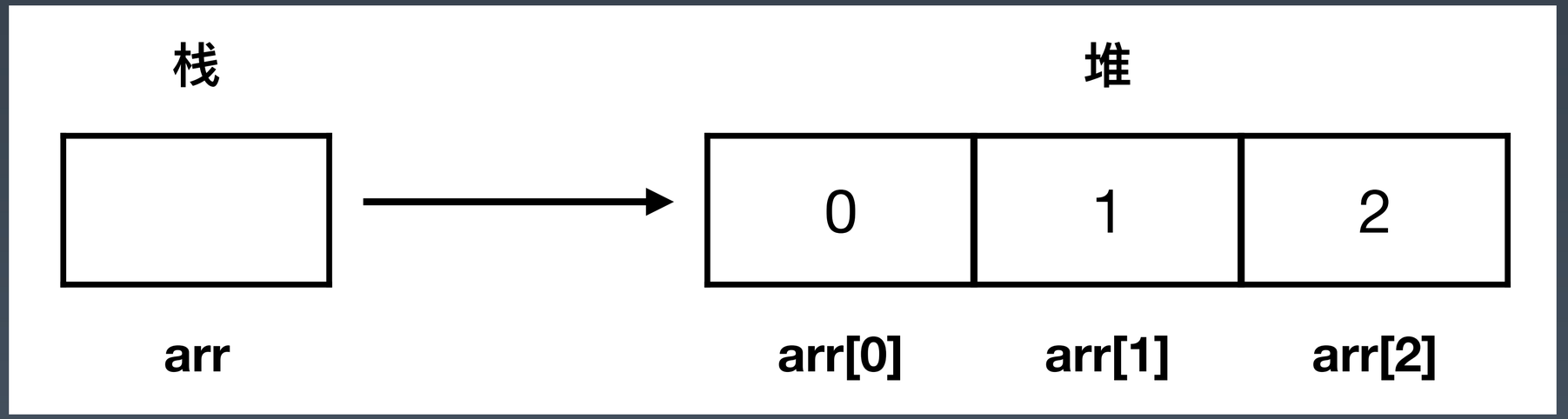
```
struct Dog {
  char a;
  char b;
};

struct Dog arr[3][2];
```


```
address_arr[i][j] = base_address + (i* m + j) * size_type
```

Java 中数组的实现方式：基本数据类型数组

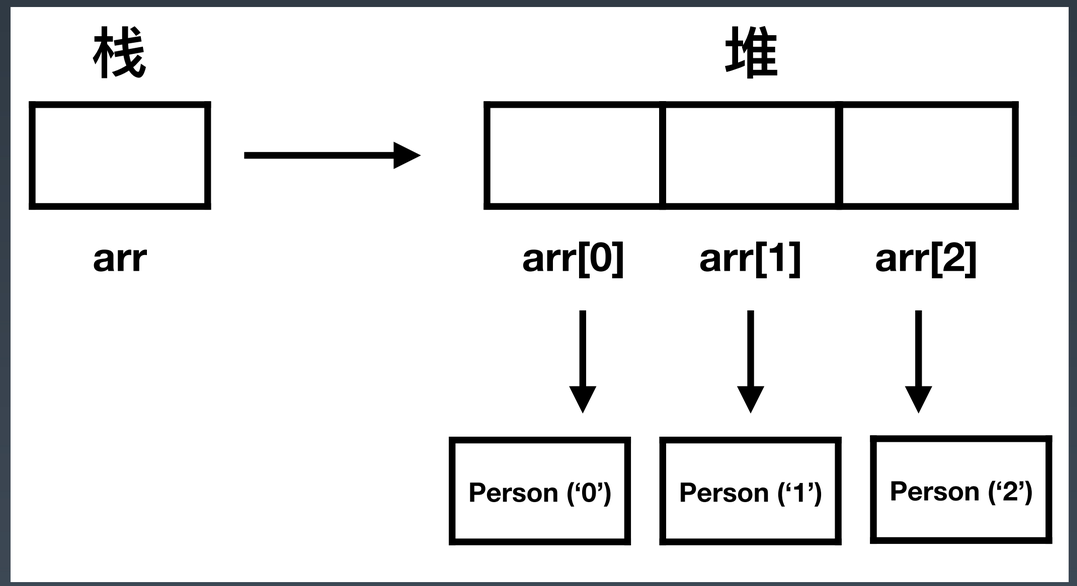
```
int arr[] = new int[3];
arr[0] = 0;
arr[1] = 1;
arr[2] = 2;
```



Java 中数组的实现方式：对象数组

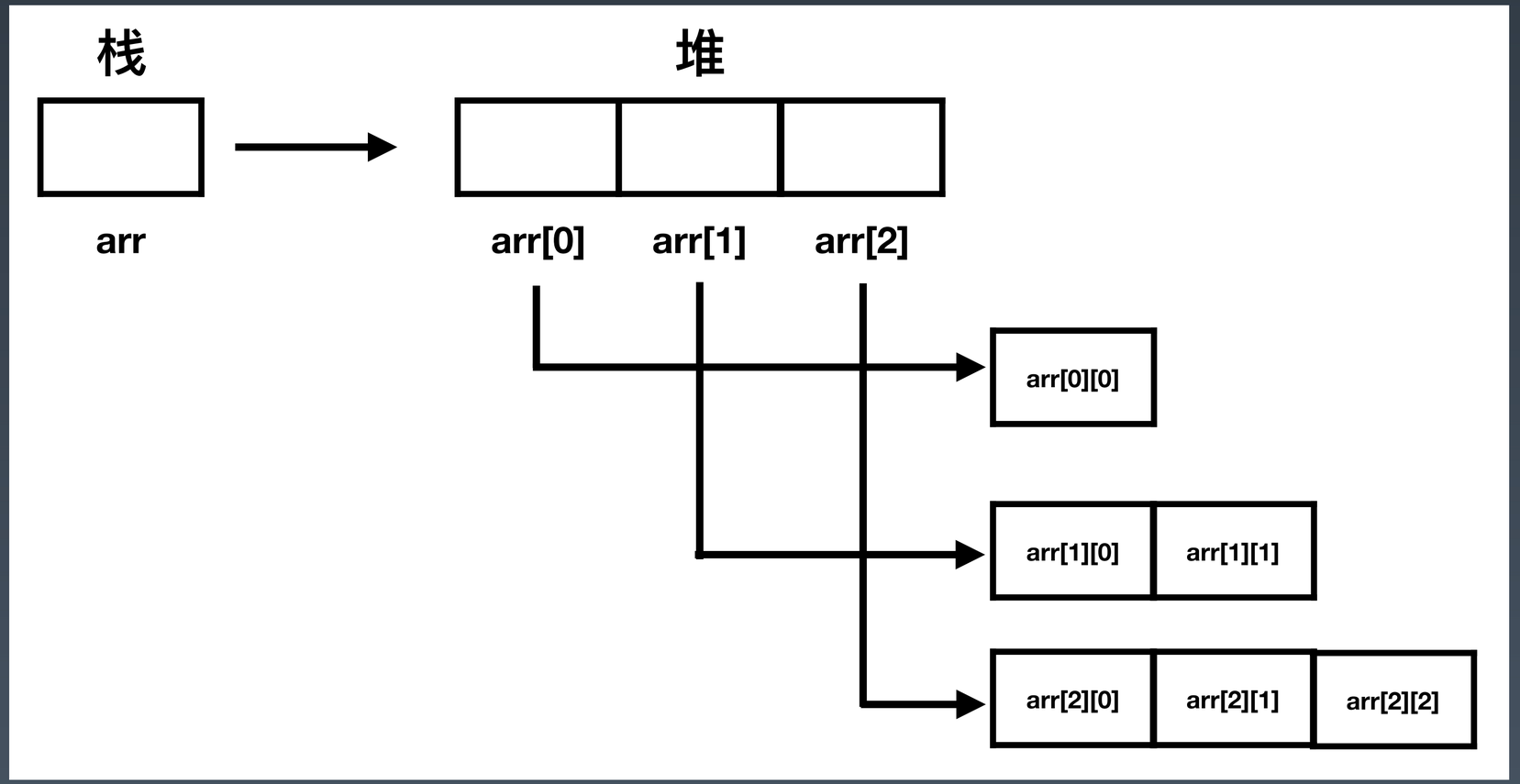
```
class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
}

Person arr[] = new Person[3];
arr[0] = new Person("0");
arr[1] = new Person("1");
arr[2] = new Person("2");
```



Java 中数组的实现方式：二维数组

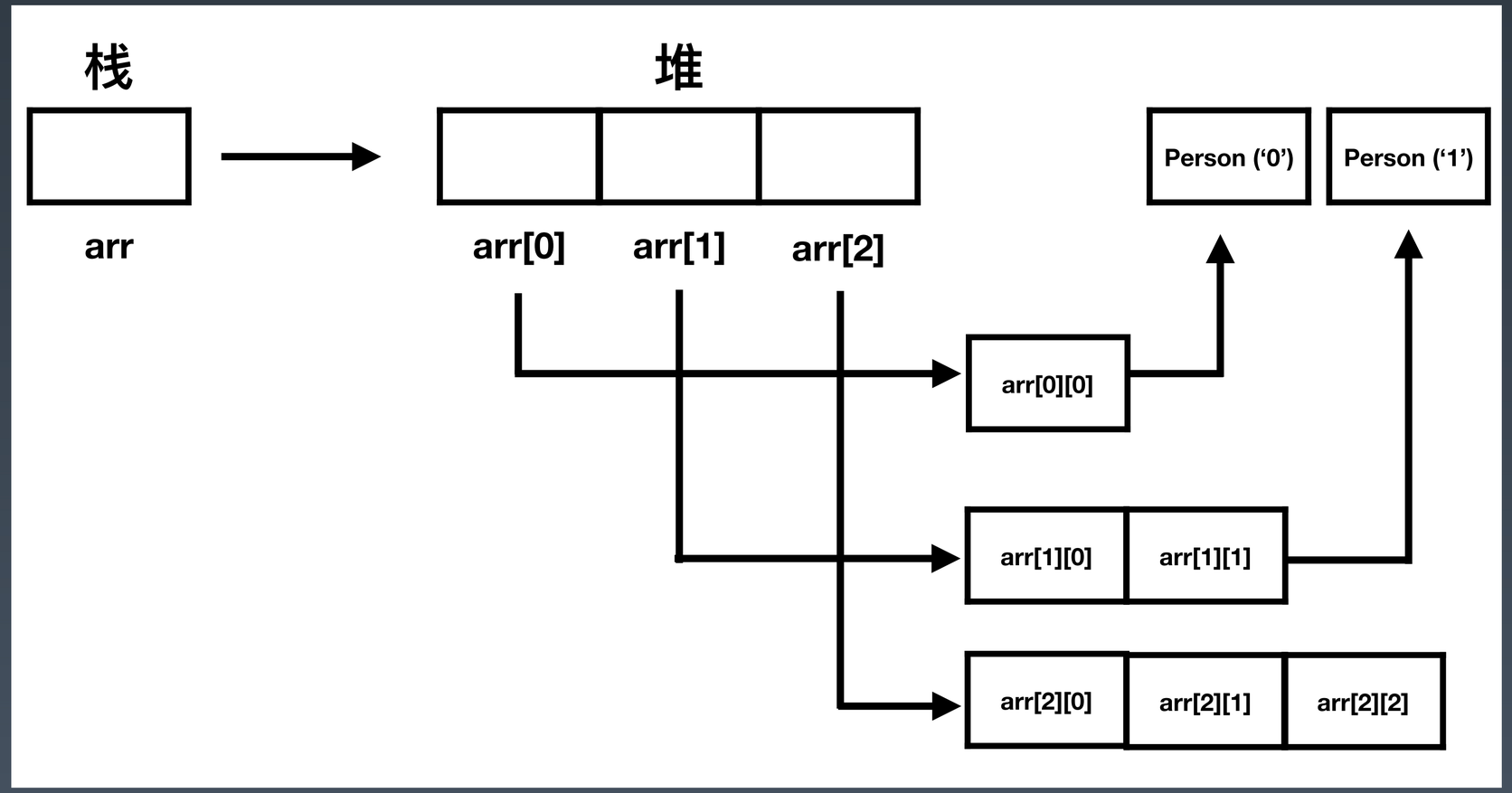
```
int arr[][] = new int[3][];
arr[0] = new int[1];
arr[1] = new int[2];
arr[2] = new int[3];
```



Java 中数组的实现方式：二维数组

```
Person arr[][] = new
Person[3][];
arr[0] = new Person[1];
arr[1] = new Person[2];
arr[2] = new Person[3];

arr[0][0] = new
Person("0");
arr[1][1] = new
Person("1");
```



JavaScript 中数组的实现方式

如果数组中存储的是相同类型的数据，那 JavaScript 就真的用数据结构中的数组来实现。也就是说，会分配一块连续的内存空间来存储数据。

如果数组中存储的是非相同类型的数据，那 JavaScript 就用类似散列表的结构来存储数据。也就是说，数据并不是连续存储在内存中的。这也是 JavaScript 数组支持存储不同类型数据的原因。

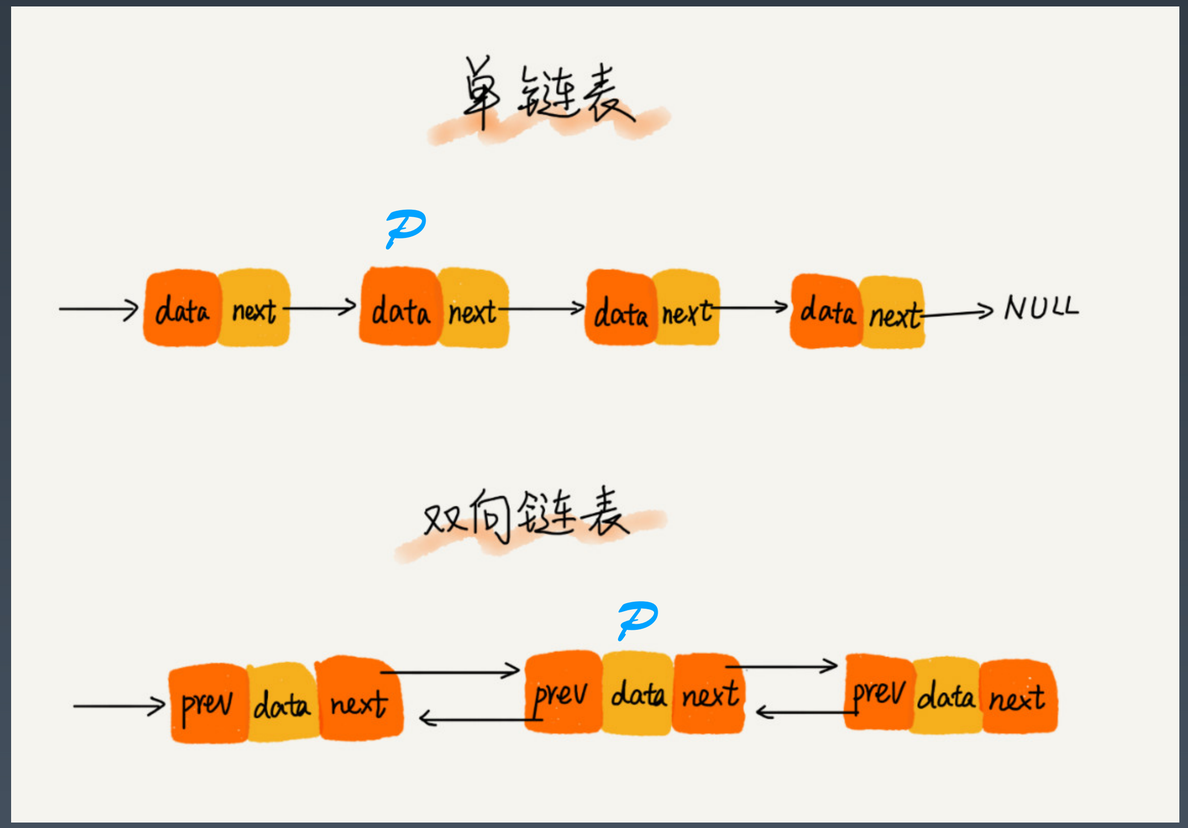
如果你往一个存储了相同类型数据的数组中，插入一个不同类型的数据，那 JavaScript 会将底层的存储结构，从数组变成散列表。

ArrayBuffer: 标准的数据结构数组的实现

链表

单链表 vs 双链表的操作时间复杂度

	单链表	双向链表
void remove(int dataValue);	O(n)	O(n)
void remove(Node *p);	O(n)	O(1)
void remove(Node *prev);	O(1)	O(1)
void insertBefore(Node *p, int dataValue);	??	??
void insertAfter(Node *p, int dataValue);	??	??



递归

递归时间复杂度分析

1. 递推公式

2. 递归树

递归时间复杂度分析：递推公式

归并排序算法递推公式：

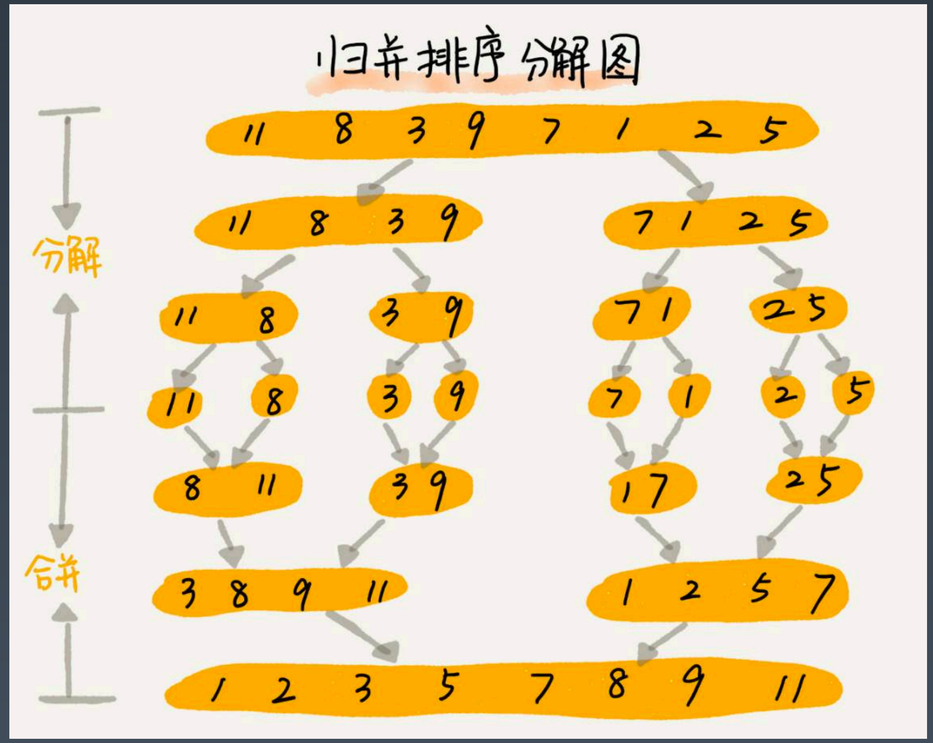
msort(0…n) = merge(**msort(0…n/2)**, **msort(n/2+1…n)**)

T(n) = 2*T(n/2) + n; n>1

T(1) = C; n=1 时，只需要常量级的执行时间，所以表示为 C。

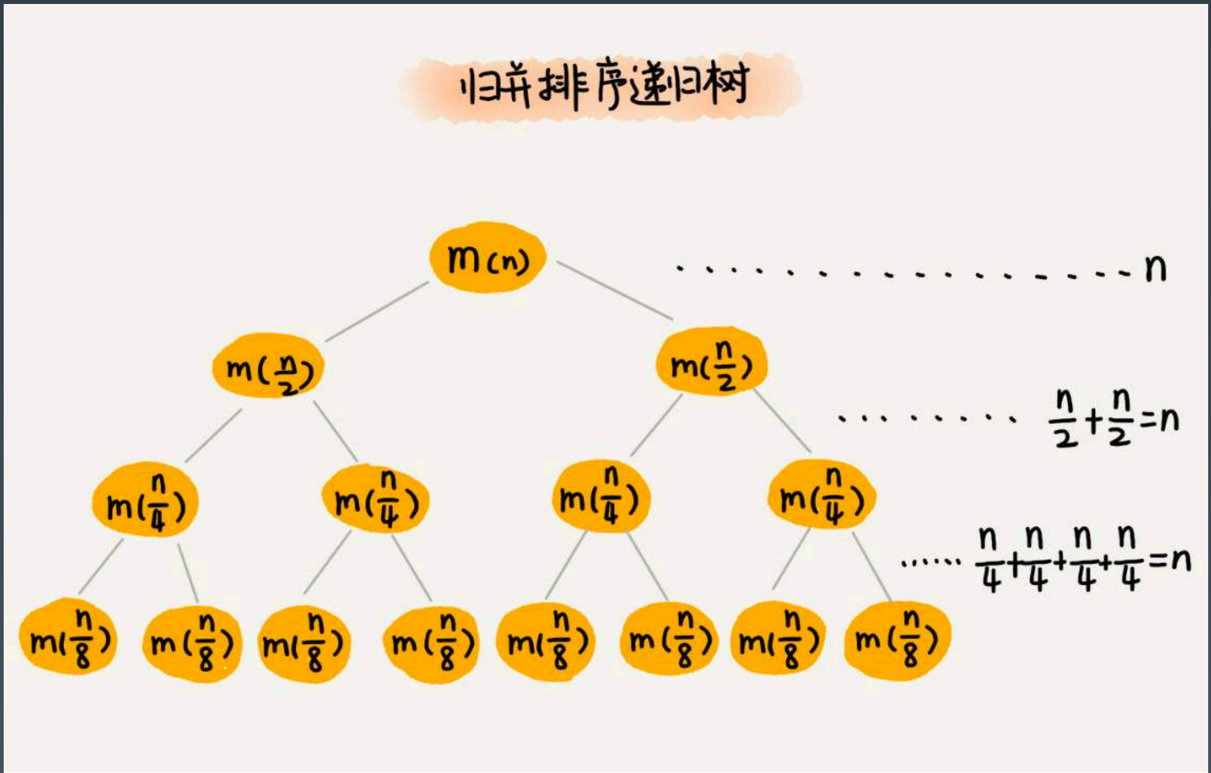
```
T(n) = 2*T(n/2) + n
     = 2*(2*T(n/4) + n/2) + n = 4*T(n/4) + 2*n
     = 4*(2*T(n/8) + n/4) + 2*n = 8*T(n/8) + 3*n
     = 8*(2*T(n/16) + n/8) + 3*n = 16*T(n/16) + 4*n
     .......
     = 2^k * T(n/2^k) + k * n
     ……
```

当 n/2^k=1, T(n/2^k)=T(1), k=log2(n) T(n)=C*n+n*log2(n)



递归时间复杂度分析：递归树

msort(0…n) = merge(**msort(0…n/2)**, **msort(n/2+1…n)**)



归并排序每次会将数据规模一分为二。

归并排序包含两部分操作：分解和合并。

分解的时间消耗记作常量 1。

归并算法中比较耗时的是归并操作，也就是把两个子数组合并为大数组。每一层归并操作消耗的时间总和是一样的，跟要排序的数据规模有关。我们把每一层归并操作消耗的时间记作 n。

假设树的高度是 h，用高度 h 乘以每一层的时间消耗 n，得到总的时间复杂度 O(n*h)。

归并排序递归树是一棵满二叉树。满二叉树的高度大约是 log2(n)。

所以，归并排序递归实现的时间复杂度就是 O(nlogn)。

尾递归

递归导致堆栈溢出产生原因

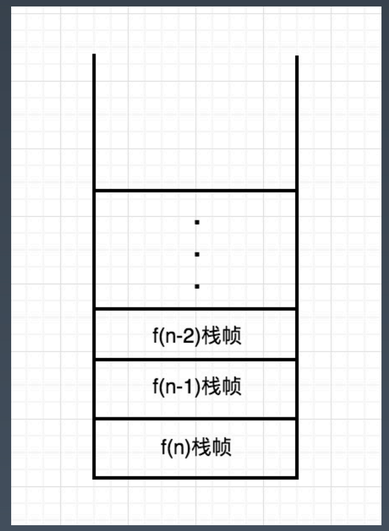
Exception in thread "main" java.lang.StackOverflowError

函数调用会使用栈来保存临时变量。每调用一个函数，都会将临时变量封装为栈帧压入内存栈，等函数执行完成返回时才出栈。系统栈或者虚拟机栈空间一般都不大。如果递归求解的数据规模很大，调用层次很深，一直压入栈，就会有堆栈溢出的风险。

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

f(n) 的计算依赖 n 局部变量和 f(n-1) 的返回值

栈帧中的信息：局部变量、返回地址等



如何避免递归导致堆栈溢出?

1. 限制递归调用的最大深度递

```
int depth = 0;

int f(int n) {
  ++depth;
  if (depth > 1000) throw
exception;

  if (n <= 1) return 1;
  return n * f(n-1);
}
```

2. 将递归代码转化成非递归代码

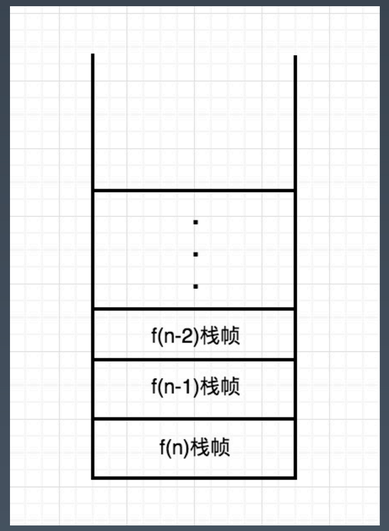
```
int f(int n) {
  int result = 1;
  for (int i = 2; i <= n; ++i) {
     result = result * i;
  }
  return result;
}
```

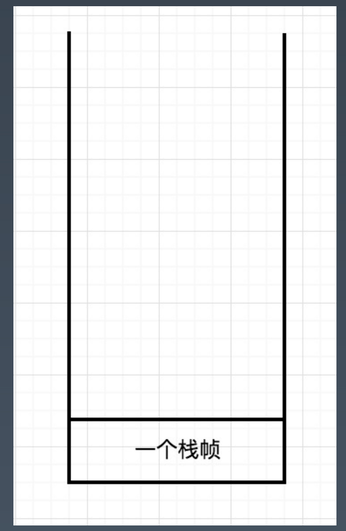
如何避免递归导致堆栈溢出?

3. 将递归改造成尾递归

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

```
int f(int n, int res) {
     if (n <= 1) return res;
     return f(n - 1, n * res);
}
```





关于尾递归的一些解释

1. 并不是所有的递归都可以改成尾递归
2. 能改成尾递归的代码都可以改成迭代
3. 尾递归并不一定能避免堆栈溢出
4. 尾递归的可读性很差，不推荐使用

THANKS!