M1932 Algorithms Outline

- 03 | 复杂度分析（上）：如何分析、统计算法的执行效率和资源消耗？
- 04 | 复杂度分析（下）：浅析最好、最坏、平均、均摊时间复杂度
- 05 | 数组：为什么很多编程语言中数组都从0开始编号？


复杂度
分析上
快省
NO.3

Big O. Lv.5. I

渐进时间复杂度.
时间复杂度.
Asymptotic time complexity

• 复杂度分析? 利用统计, 监控等事后统计法?

环境因素: 同样代码, 不同机器测试结果不同.

数据因素: 数据规模大小, 已排序.

不依赖环境, 测试数据, 估算执行效率的方法.

• Big O 表示法.

```
int sum = 0;
```

读数据 - 运算 - 写数据. 每行执行时间相同 unit - time.

代码执行时间 $T(n)$ 与每行执行次数成正比.

$T(n) = O(f(n))$ n是数据规模. $f(n)$ 每行代码执行次数总和.

O表示 $T(n)$ 和 $f(n)$ 成正比.

```
int cal (int n) {
  int sum=0; int i=0; int j=0;  -> 3
  for( ; i<=n; ++i) { j=1;      -> 2n
    for( ; j<=n; ++j) {
      sum = sum + i*j;          -> 2n^2
```

$T(n) = 2n^2 + 2n + 3 = O(2n^2 + 2n + 3) = O(n^2)$

低阶, 常量, 系数, 对数据规模可忽略.

代码执行时间随数据规模变化趋势.

• 时间复杂度分析

1. 循环次数最多的一段代码。

```
for (; i<=n; ++i) {
    sum = sum+i;
}
```

$\Rightarrow O(n)$.

低阶，常量，系数 忽略。

2. 加法原则：总复杂度是量级最大那段代码复杂度。

```
for (; p<100; ++p) {
    sum1 = sum_1 + p;
}
```

→ 不管 100/1000/10000 都是常数。

```
for (; q<n; ++q) {
    sum2 = sum_2 + q;
}
```

$\rightarrow T_1(n) = 2n = O(n)$.

```
for (; i<=n; ++i) {
    j=1
    for (; j<=n; ++j) {
        sum_3 = sum_3 + i*j;
    }
}
```

$> 2n$ $\rightarrow T_2(n) = 2n^2 + 2n = O(n^2)$.

$> 2n^2$

$T_1(n) = O(f(n))$ $T_2(n) = O(g(n))$

$\Rightarrow T(n) = T_1(n) + T_2(n)$

$= \max(O(f(n)), O(g(n)))$

$= O(\max(f(n), g(n)))$

$= O(n^2)$

3. 乘法法则：嵌套代码的复杂度等内外代码复杂度的乘积.

```
for(; i<n; ++i){
    ret = ret + f(i);      > T1(n)=n
}
int f(int n){
    ...
    for(; i<n; ++i){       > T2(n)=n.
        sum = sum + i;
    }
}
```

$\Rightarrow T(n) = T_1(n) * T_2(n)$ 多练

$= O(n*n) = O(n^2)$.

• 几种常见复杂度. 多项式量级.（非多项 NP）.

1. O(1). 无循环，无递归，上万行代码 也是 O(1).

乘法. $O(n\log n)$

归并. 快排.

2. $O(\log n)$ $O(n\log n)$.

```
while (i<=n){
    i = i*2;
}
```

$\Rightarrow 2^0\ 2^1\ 2^2\ 2^3 \ldots 2^k \ldots 2^n = n$. $2^x = n$ $x = \log_2 n$

$\Rightarrow O(\log_2 n) \Rightarrow O(\log n)$

eg: $O(\log_3 n) \Rightarrow \log_3 2 * \log_2 n \Rightarrow O(C\log_2 n) \Rightarrow O(\log n)$

3. $O(m+n)$ $O(m*n)$.

```
for (; i<m; ++i) {
    sum1 = sum1 + i;
}
for (; j<n; ++j) {
    sum2 = sum2 + j;
}
```

$O(m)$

$O(n)$.

量级不同. $O(m+n)$. 加法

$O(m*n)$ 乘法.

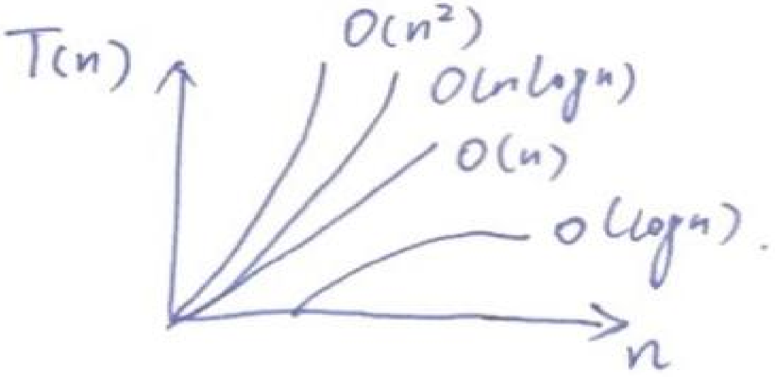
• 空间复杂度.

`int[] a = new int[n];` $O(n)$.空间复杂度.

掌握. $O(1)$ $O(n)$ $O(n^2)$.

• 小结.

从低到高阶复杂度.



$T(n)$: $O(1)$. $O(\log n)$. $O(n)$. $O(n\log n)$. $O(n^2)$.

复杂度量级（按数量级递增）

- 常量阶 $O(1)$
- 对数阶 $O(\log n)$
- 线性阶 $O(n)$
- 线性对数阶 $O(n\log n)$
- 平方阶 $O(n^2)$ 、立方阶 $O(n^3)$ …… K次方阶 $O(n^k)$

- 指数阶 $O(2^n)$
- 阶乘阶 $O(n!)$

非多项式量级


"best、worst、average、amortized"
NO.4
复杂度
分析下

- **最好、最坏均摊复杂度**

```
// n 表示数组 array 的长度
int find(int[] array, int n, int x) {
  int i = 0;
  int pos = -1;
  for (; i < n; ++i) {
    if (array[i] == x) pos = i;
  }
  return pos;
}
```

时间复杂度为 O(n)

```
// n 表示数组 array 的长度
int find(int[] array, int n, int x) {
  int i = 0;
  int pos = -1;
  for (; i < n; ++i) {
    if (array[i] == x) {
      pos = i;
      break;
    }
  }
  return pos;
}
```

如果第一个就找到了想要的值，时间复杂度为 O(1)
最坏的情况是遍历了所以数据后才找到，时间复杂度为 O(n)

所以最好复杂度为 O(1)，最坏复杂度为 O(n)

- # 平均时间复杂度

等差数列：等差数列是指从第二项起，每一项与它的前一项的差等于同一个常数的一种数列

$$1 + 2 + 3 + 4 + 5 \ldots + n = \frac{n(n+1)}{2}$$

```
// n 表示数组 array 的长度
int find(int[] array, int n, int x) {
  int i = 0;
  int pos = -1;
  for (; i < n; ++i) {
    if (array[i] == x) {
       pos = i;
       break;
    }
  }
  return pos;
}
```

对于上面的例子要查找的变量 x 在数组中的位置，有 n+1 种情况：在数组的 0 ~ n-1 位置中和不在数组中。

$$\frac{1+2+3+\cdots+n+n}{n+1} = \frac{n(n+3)}{2(n+1)}$$

推导过程：$\frac{n(n+1)}{2} + \frac{\mathrm{n}}{n+1} = \frac{n(n+3)}{2(n+1)}$

上面复杂度分析没有考虑到所有情况

在数组和不在数组的概率是一样的为：1/2

在0~n-1位置的概率也是一样的为：1/n

复杂度分析则为：

$$1\times\frac{1}{2n}+2\times\frac{1}{2n}+3\times\frac{1}{2n}+\cdots+n\times\frac{1}{2n}+n\times\frac{1}{2}$$

$$=\frac{3n+1}{4}$$

推导过程：

$$\frac{n(n+1)}{2}\,x\,\frac{1}{2n}+\frac{n}{2}=\frac{3n^2+n}{4n}=\frac{3n+1}{4}$$

此值为加权平均值（期望值）

去掉低阶，系数，常数项，最终加权平均复杂度（期望平均复杂度）为：O(n)

• 均摊时间复杂度

复杂度分析：摊还分析（平摊分析）

```
// array 表示一个长度为 n 的数组
// 代码中的 array.length 就等于 n
int[] array = new int[n];
int count = 0;

void insert(int val) {
    if (count == array.length) {
        int sum = 0;
        for (int i = 0; i < array.length; ++i) {
            sum = sum + array[i];
        }
        array[0] = sum;
        count = 1;
    }

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

$$1\times\frac{1}{n+1}+1\times\frac{1}{n+1}+\cdots+1\times\frac{1}{n+1}+n\times\frac{1}{n+1}=O(1)$$

清空数组后额外要插入一次sum，所以是 n+1

每次出现概率都是1/(n+1)

插入数据都是 1，最后遍历求和为 n

所以如上式，最后均摊复杂度为 O(1)

- **区别**

1. find() 在极端情况下为 O(1), insert() 在大多数情况下为 O(1) , 少数的极端情况下是 O(n)

2. insert() 比 find() 出现频率很有规律，都是 O(n) 插入后，接着 n-1 次 O(1) 操作

• 思考

```
// 全局变量，大小为 10 的数组 array，长度 len，下标 i。
int array[] = new int[10];

int len = 10;
int i = 0;

// 往数组中添加一个元素
void add(int element) {
   if (i >= len) { // 数组空间不够了
     // 重新申请一个 2 倍大小的数组空间
     int new_array[] = new int[len*2];
     // 把原来 array 数组中的数据依次 copy 到 new_array
     for (int j = 0; j < len; ++j) {
       new_array[j] = array[j];
     }
     // new_array 复制给 array，array 现在大小就是 2 倍 len 了
     array = new_array;
     len = 2 * len;
   }
   // 将 element 放到下标为 i 的位置，下标 i 加一
   array[i] = element;
   ++i;
}
```

时间负责度为:

O(1)

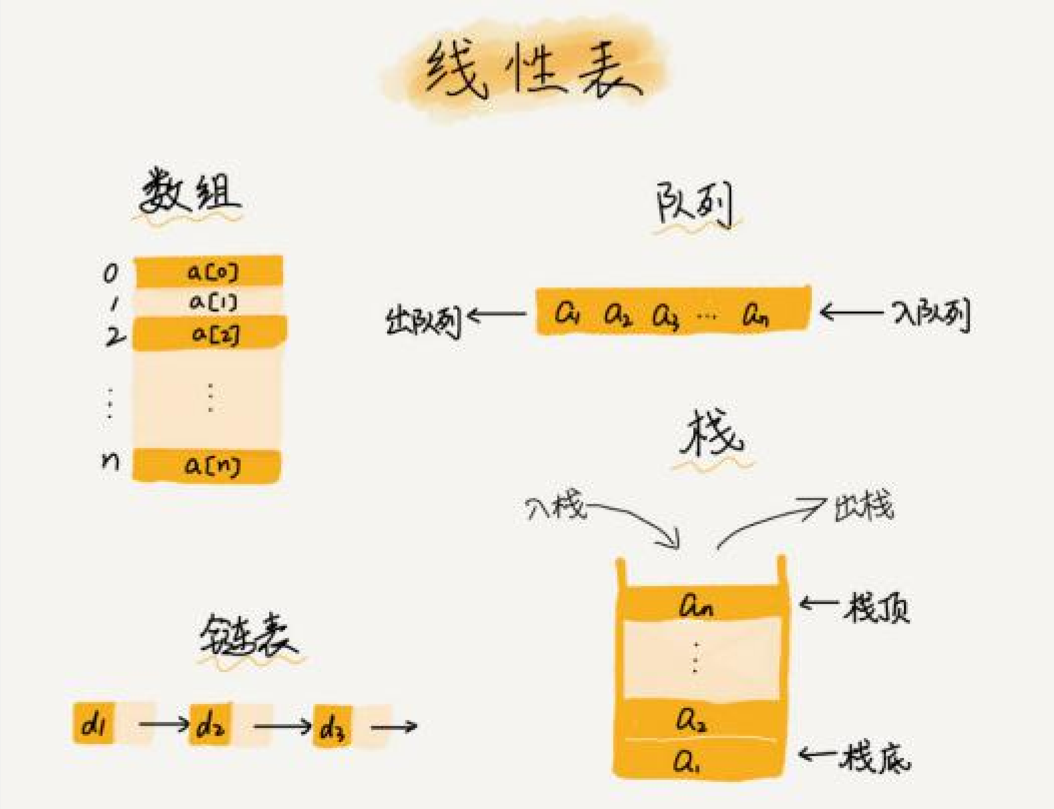
空间复杂度为：

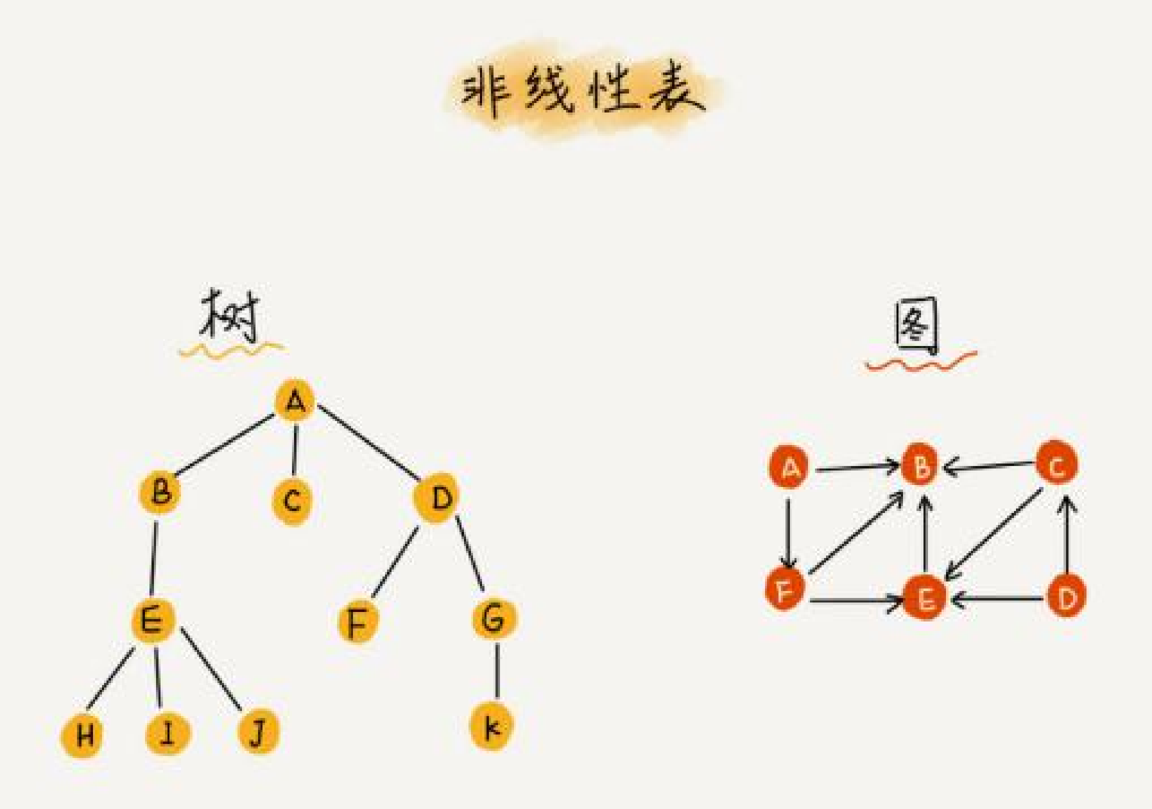
O(logn)


Array
数组
No.5

- **数组**

是一种线性表数据结构。它用一组连续的内存空间，来存储一组具有相同类型的数据。





- **随机访问**

创建一个大小为10的int数组，内存地址连续

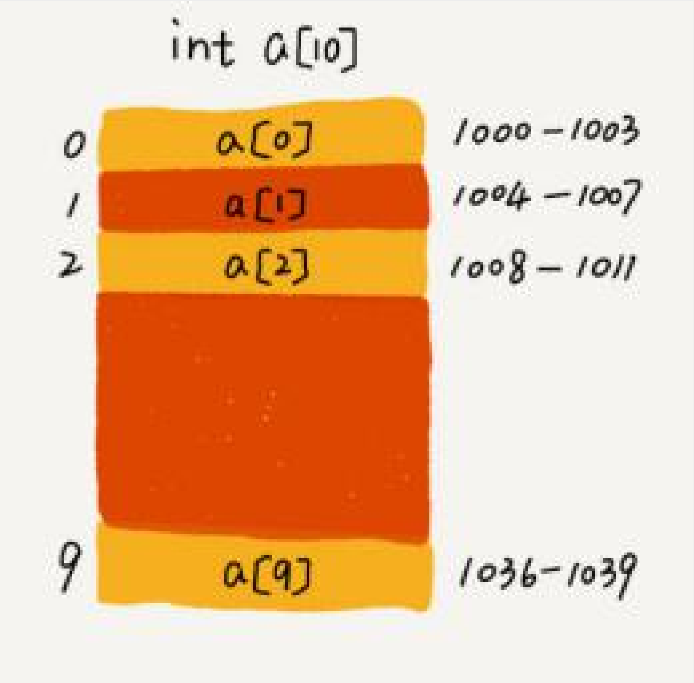
寻址公式：

a[i]_address = base_address + i * data_type_size

面试说：

链表插入删除快，查找慢；数组插入删除慢，~~查找快~~

数组随机访问的时间复杂度为 O(1)



- **低效地插入**

如果数组是有序的，有规律的

在尾部插入时间复杂为 O(1)

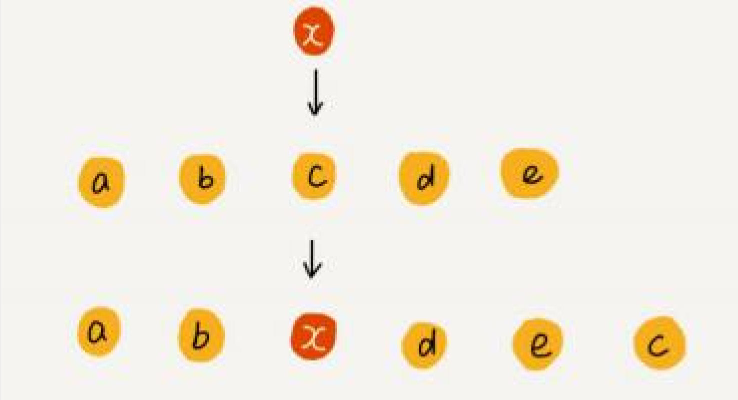
在头部插入，因为涉及到数据搬移，时间复杂度为 O(n)

每个位置插入概率是一样的，平均复杂度为 O(n)

如果数组无序，无规则

可以将需要插入位置的值放到尾部，时间复杂度为 O(1)

插入需要插入的值就为 O(1)



- **低效地删除**

按常规操作删除数组中的元素

同上述插入的时间复杂度

通过标记清楚算法

如果要删除某些元素，可以先标记并不删除

当插入数据时，数组空间不足，进行一次删操作，避免数据搬移

JVM 标记清除垃圾回收算法的核心思想

- **警惕数组越界**

第三行判断条件=3会导致数组访问越界

C语言不会像Java帮你进行越界判断

```c
1 int main(int argc, char* argv[]){
2     int i = 0;
3     int arr[3] = {0};
4     for(; i<=3; i++){
5         arr[i] = 0;
6         printf("hello world\n");
7     }
8     return 0;
9 }
```

栈是由高到低位增长的

i和数组的数据从高位地址到低位地址依次是：

i, a[2], a[1], a[0]

a[3]通过寻址公式，计算得到地址正好是i的存储地址，所以a[3]=0，就相当于i=0

- **容器能否完全替代数组**

区别	ArrayList	Array
扩容	增加，删除，扩容都已封装，动态扩容为原来1.5倍	需要重新申请空间并将数据拷贝到新空间，再插入
数据类型	无法添加基本数据类型，需要拆箱装箱，损耗性能	任意类型
多维数组	嵌套显示 ArrayList<ArrayList > array	Object[][] array
用途	业务开发省时省力，性能损耗很小	底层开发

- **解答开篇，为什么数组下标是从0开始**

下标确切说是 offset 偏移量，如果0开始则

a[k]_address = base_address + k * type_size

如果是1开始，则

a[k]_address = base_address + (k-1)*type_size

减少一次CPU减法操作。当然也不是绝对所有语言都是0开始

- **思考1**

你理解的标记清除垃圾回收算法？

采用可达性算法分析来判断对象是否存活，并标记阶段，遍历GC_ROOTS标记可达对象为存活对象（还是标记要被清理的对象），在标记完成后进行GC清理操作。标记清理算法容易产生大量不连续的内存碎片，当清理完后由于碎片化严重还是无法分配内存空间则需要重新触发GC操作。所以一般使用在老年代。

所以还有标记-整理算法

- **思考2**

二维数组的内存寻址公式？

a[i][j]_address = base_address + (i * 一维数组大小 + j) * type_size