算法复杂度

无隅老师

算法复杂度

- 时间复杂度： 执行算法所需要的计算工作量
- 空间复杂度： 执行算法所需要的内存空间

Warmup ：从一个简单问题开始

- 从数组中查找一个数

- 如果数组有 10 个元素？
- 如果数组有 1000 个元素？
- 如果数组有 100000 个元素？

什么是时间复杂度

- 在计算机科学中，算法的时间复杂度是一个函数，它定量描述了该算法的运行时间。
- 以算法输入值规模n为自变量的函数：$T(n) = O(f(n))$

Big O, Big Theta and Big Omega

- 大 O 符号　　O　$<$
- 大 Ω 符号　　Ω　$>$
- 大 Θ 符号　　Θ　$=$

大 O 符号 Example

$T(n) = 4n^2 + 2n + 1$

（1） n=1时 $4n^2$ 项是2n 项的2倍大

（2） n=500时 $4n^2$ 项是2n 项的1000倍大

2n 项对表达式值的影响可忽略不计的。

结论： $T(n) \in O(n^2)$ 或 $T(n) = O(n^2)$

Q & A

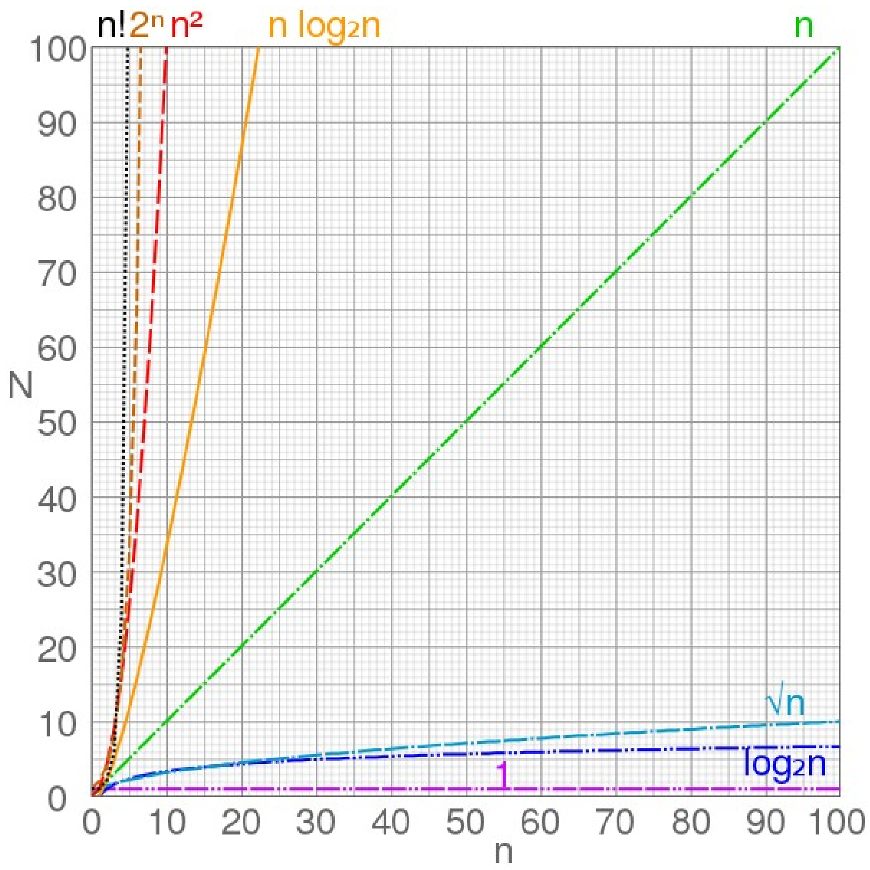
$$T(n) = 4n^2 + 2n + 1$$

$$T(n) \in O(n^3)$$

$$T(n) \in O(n^2)$$

如果同时都满足条件，我们需要给出更收敛的答案

时间复杂度比较



- $O(n^2 + n) \rightarrow O(n^2)$
- $O(logn + n) \rightarrow O(\mathrm{n})$
- $O(5 \ * 2^n + 1000n^{100}) \rightarrow O(2^n)$

$$\boxed{O(n!) > O(2^n) > O(n^3) > O(n^2) > O(nlogn) > O(n) > O(logn) > O(1)}$$

Best case, Worst case, Expected Case

- 从数组中查找一个数

- 最好情况： 目标值是数组第一个元素，$T(n) = O(1)$
- 最坏情况： 目标值是数组最后一个元素，$T(n) = O(n)$
- 期望情况（平均情况）： $T(n) = O(n)$

计算时间复杂度 – 一般问题

- 基本操作的时间复杂度
  - 丢弃常数项
  - 丢弃次要项
- 基本操作被执行了多少次（For ／ While 循环）
- 复合操作：加还是乘

基本操作的时间复杂度 - 丢弃常数项

```java
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int num : array) {
    if (num < min) min = num;
    if (num > min) max = num;
}
```

```java
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int num : array) {
    if (num < min) min = num;
}

for (int num : array) {}
    if (num > min) max = num;
}
```

基本操作的时间复杂度 - 丢弃次要项

- $O(n^2 + n) \to O(n^2)$
- $O(logn + n) \to O(\mathrm{n})$
- $O(5 \ * 2^n + 1000n^{100}) \to O(2^n)$

复合操作：加还是乘

- 假设算法有两步，每一步的时间复杂度为 O(A)， O(B)

  计算时间复杂度时什么时候该将两步的时间复杂度相加，什么时候该相乘

```
for (int a : arrA) {
    print(a);
}

for (int b : arrB) {
    print(b);
}
```

```
for (in a : arrA) {
    for (int b :arrB) {
        print(a + "," + b);
    }
}
```

O(A + B)

O(A * B)

结论： 1. 先做 A，然后做完 A 后， 再做 B， 应该将这两件事的时间复杂度相加

2. 每一次做 A 的时候都需要将 B 全部做一遍， 应该将这两件事的时间复杂度相乘

例题

```java
void foo(int[] array) {
    int sum = 0;
    int product = 1;
    for (int i = 0; i < array.length; i++) {
        sum += array[i];
    }

    for (int i = 0; i < array.length; i++) {
        product *= array[i]
    }
    System.out.println(sum + ", " + product);
}
```

例题

```java
void printPairs(int[] array) {
    for (int i =  0; i < array.length; i++) {
        for (int j = 0; j < array.length; j++) {
            System.out.println(array[i] + ", " + array[j]);
        }
    }
}
```

例题

```
void printPairs(int[] array) {
    for (int i =  0; i < array.length; i++) {
        for (int j =  i + 1; j < array.length; j++) {
            System.out.println(array[i] + ", " + array[j]);
        }
    }
}
```

时间复杂度计算 - 递归问题

什么是递归： 在数学与计算机科学中，是指在函数的定义中使用函数自身的方法。递归一词还较常用于描述以自相似方法重复事物的过程。

```
int calculate(int n) {
    if (n <= 0) {
        return 1;
    }
    return calculate(n - 1) + calculate(n - 1);
}
```

经验性结论：
递归问题的时间复杂度通常（并不总是）看起来形如 $O(\text{branches}^{\text{depth}})$
其中 branches 指递归分支的总数，
depth 指递归调用深度

例题

```
int sum(Node node) {
    if (node == null) {
        return 0;
    }

    return  sum(node.left) + node.value + sum(node.right);
}
```

例题

```java
void permutation(String str) {
    permutation(str, "");
}

void permutation(String str, Stirng prefix) {
    if (str.length() == 0) {
        System.out.println(prefix);
    } else {
        for (int i = 0; i < str.length(); i++) {
            String rem = str.substring(0, i) + str.substring(i + 1);
            permutation(rem, prefix + str.charAt(i));
        }
    }
}
```

例题

(1)

```java
int fib(int n) {
    if (n <= 0) return 0;
    else if (n == 1) return 1;
    return fib(n - 1) + fib(n - 2);
}
```

(2)

```java
void allFib(int n) {
    for (int i = 0; i < n; i++) {
        System.out.println(i + ": " + fib(i));
    }
}

int fib(int n) {
    if (n <= 0) return 0;
    else if (n == 1) return 1;
    return fib(n - 1) + fib(n - 2);
}
```

(3)

```java
void allFib(int n) {
    int[] memo = new int[n + 1];
    for (int i = 0; i < n; i++) {
        System.out.println(i + ": " + fib(i, memo));
    }
}

int fib(int n) {
    if (n <= 0) return 0;
    else if (n == 1) return 1;
    else if (memo[n] > 0) return memo[n];

    memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
    return memo[n];
}
```

时间复杂度计算 – 主定理

- $T(n) = aT\left(\frac{n}{b}\right) + n^c$
- 比较与 c 的大小
- 比较$\log_b a$与c的大小
- $if\ log_b a > c{:}\, T(n) = nlog_b a$
- $if\ log_b a = c{:}\, T(n) = n^c logn$
- $if\ log_b a < c{:}\, T(n) = n^c$

主定理应用举例

- 二分搜索 $T(n) = T\left(\frac{n}{2}\right) + m$
- $a = 1, b = 2, c = 0,$

  $log_b a = c = 0$

  符合第二种情况 --> $T(n) = n^c logn = logn$

- $T(n) = O(logn)$

算法空间复杂度

- 时间复杂度不是衡量算法的唯一指标，有时候还需要考虑空间复杂度
- 创建长度为 n 的数组，需要 O(n) 的空间，创建一个 m * n 的二维数组，需要 O(n^2) 的空间
- 注意与时间复杂度的区别／联系

总结

- 算法复杂度
  - 时间复杂度
  - 空间复杂度
- 时间复杂度的概念
  - 概念
  - 3 种记号 -> 大 O 符号
  - 最好／最坏／平均情况
- 时间复杂度的计算
  - 一般问题

    3 大原则：丢弃常数项，丢弃次要项，复合操作
  - 递归问题
  - 主定理
- 空间复杂度