# 算法 - 算法分析

* [算法 - 算法分析](#算法---算法分析)
  + [数学模型](#数学模型)
    - [1. 近似](#1-近似)
    - [2. 增长数量级](#2-增长数量级)
    - [3. 内循环](#3-内循环)
    - [4. 成本模型](#4-成本模型)
  + [注意事项](#注意事项)
    - [1. 大常数](#1-大常数)
    - [2. 缓存](#2-缓存)
    - [3. 对最坏情况下的性能的保证](#3-对最坏情况下的性能的保证)
    - [4. 随机化算法](#4-随机化算法)
    - [5. 均摊分析](#5-均摊分析)
  + [ThreeSum](#threesum)
    - [1. ThreeSumSlow](#1-threesumslow)
    - [2. ThreeSumBinarySearch](#2-threesumbinarysearch)
    - [3. ThreeSumTwoPointer](#3-threesumtwopointer)
  + [倍率实验](#倍率实验)

## 数学模型

### 1. 近似

N3/6-N2/2+N/3 ~ N3/6。使用 ~f(N) 来表示所有随着 N 的增大除以 f(N) 的结果趋近于 1 的函数。

### 2. 增长数量级

N3/6-N2/2+N/3 的增长数量级为 O(N3)。增长数量级将算法与它的具体实现隔离开来，一个算法的增长数量级为 O(N3) 与它是否用 Java 实现，是否运行于特定计算机上无关。

### 3. 内循环

执行最频繁的指令决定了程序执行的总时间，把这些指令称为程序的内循环。

### 4. 成本模型

使用成本模型来评估算法，例如数组的访问次数就是一种成本模型。

## 注意事项

### 1. 大常数

在求近似时，如果低级项的常数系数很大，那么近似的结果是错误的。

### 2. 缓存

计算机系统会使用缓存技术来组织内存，访问数组相邻的元素会比访问不相邻的元素快很多。

### 3. 对最坏情况下的性能的保证

在核反应堆、心脏起搏器或者刹车控制器中的软件，最坏情况下的性能是十分重要的。

### 4. 随机化算法

通过打乱输入，去除算法对输入的依赖。

### 5. 均摊分析

将所有操作的总成本除于操作总数来将成本均摊。例如对一个空栈进行 N 次连续的 push() 调用需要访问数组的次数为 N+4+8+16+...+2N=5N-4（N 是向数组写入元素的次数，其余都是调整数组大小时进行复制需要的访问数组次数），均摊后访问数组的平均次数为常数。

## ThreeSum

ThreeSum 用于统计一个数组中和为 0 的三元组数量。

public interface ThreeSum {  
 int count(int[] nums);  
}

### 1. ThreeSumSlow

该算法的内循环为 if (nums[i] + nums[j] + nums[k] == 0) 语句，总共执行的次数为 N(N-1)(N-2) = N3/6-N2/2+N/3，因此它的近似执行次数为 ~N3/6，增长数量级为 O(N3)。

public class ThreeSumSlow implements ThreeSum {  
 @Override  
 public int count(int[] nums) {  
 int N = nums.length;  
 int cnt = 0;  
 for (int i = 0; i < N; i++) {  
 for (int j = i + 1; j < N; j++) {  
 for (int k = j + 1; k < N; k++) {  
 if (nums[i] + nums[j] + nums[k] == 0) {  
 cnt++;  
 }  
 }  
 }  
 }  
 return cnt;  
 }  
}

### 2. ThreeSumBinarySearch

将数组进行排序，对两个元素求和，并用二分查找方法查找是否存在该和的相反数，如果存在，就说明存在和为 0 的三元组。

应该注意的是，只有数组不含有相同元素才能使用这种解法，否则二分查找的结果会出错。

该方法可以将 ThreeSum 算法增长数量级降低为 O(N2logN)。

public class ThreeSumBinarySearch implements ThreeSum {  
  
 @Override  
 public int count(int[] nums) {  
 Arrays.sort(nums);  
 int N = nums.length;  
 int cnt = 0;  
 for (int i = 0; i < N; i++) {  
 for (int j = i + 1; j < N; j++) {  
 int target = -nums[i] - nums[j];  
 int index = BinarySearch.search(nums, target);  
 // 应该注意这里的下标必须大于 j，否则会重复统计。  
 if (index > j) {  
 cnt++;  
 }  
 }  
 }  
 return cnt;  
 }  
}

public class BinarySearch {  
  
 public static int search(int[] nums, int target) {  
 int l = 0, h = nums.length - 1;  
 while (l <= h) {  
 int m = l + (h - l) / 2;  
 if (target == nums[m]) {  
 return m;  
 } else if (target > nums[m]) {  
 l = m + 1;  
 } else {  
 h = m - 1;  
 }  
 }  
 return -1;  
 }  
}

### 3. ThreeSumTwoPointer

更有效的方法是先将数组排序，然后使用双指针进行查找，时间复杂度为 O(N2)。

同样不适用与数组存在重复元素的情况。

public class ThreeSumTwoPointer implements ThreeSum {  
  
 @Override  
 public int count(int[] nums) {  
 int N = nums.length;  
 int cnt = 0;  
 Arrays.sort(nums);  
 for (int i = 0; i < N - 2; i++) {  
 int l = i + 1, h = N - 1, target = -nums[i];  
 while (l < h) {  
 int sum = nums[l] + nums[h];  
 if (sum == target) {  
 cnt++;  
 l++;  
 h--;  
 } else if (sum < target) {  
 l++;  
 } else {  
 h--;  
 }  
 }  
 }  
 return cnt;  
 }  
}

## 倍率实验

如果 T(N) ~ aNblogN，那么 T(2N)/T(N) ~ 2b。

例如对于暴力的 ThreeSum 算法，近似时间为 ~N3/6。进行如下实验：多次运行该算法，每次取的 N 值为前一次的两倍，统计每次执行的时间，并统计本次运行时间与前一次运行时间的比值，得到如下结果：

|  |  |  |
| --- | --- | --- |
| N | Time(ms) | Ratio |
| 500 | 48 | / |
| 1000 | 320 | 6.7 |
| 2000 | 555 | 1.7 |
| 4000 | 4105 | 7.4 |
| 8000 | 33575 | 8.2 |
| 16000 | 268909 | 8.0 |

可以看到，T(2N)/T(N) ~ 23，因此可以确定 T(N) ~ aN3logN。

public class RatioTest {  
  
 public static void main(String[] args) {  
 int N = 500;  
 int loopTimes = 7;  
 double preTime = -1;  
 while (loopTimes-- > 0) {  
 int[] nums = new int[N];  
 StopWatch.start();  
 ThreeSum threeSum = new ThreeSumSlow();  
 int cnt = threeSum.count(nums);  
 System.out.println(cnt);  
 double elapsedTime = StopWatch.elapsedTime();  
 double ratio = preTime == -1 ? 0 : elapsedTime / preTime;  
 System.out.println(N + " " + elapsedTime + " " + ratio);  
 preTime = elapsedTime;  
 N \*= 2;  
 }  
 }  
}

public class StopWatch {  
  
 private static long start;  
  
  
 public static void start() {  
 start = System.currentTimeMillis();  
 }  
  
  
 public static double elapsedTime() {  
 long now = System.currentTimeMillis();  
 return (now - start) / 1000.0;  
 }  
}