> 2018年10月10日

这一节将以一个具体的算法题给出4种不同解法，分析各自的时间复杂度并比较其各自的运行性能。

给出两个求和公式，以下分析中会用到：

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \tag{1}$$

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \tag{2}$$

> **最大子序列和问题**

$A_1, A_2, A_3, \ldots, A_N$，求 $\sum_{k=i}^{j} A_k$的最大值。（为方便起见，若所有整数均为负数，则最大子序列和为0）。

例如：输入 $-2, 11, -4, 13, -5, -2$，其最大子序列和为 $11 + (-4) + 13 = 20$。

1，时间复杂度为 $O(N^3)$的解法

```java
    public static int maxSubSum1(int[] a) {
        int maxSum = 0;
        for (int i = 0; i < a.length; i++) {
            for (int j = i; j < a.length; j++) {
                int thisSum = 0;
                for (int k = i; k <= j; k++) {
                    thisSum += a[k];
                }
                if (thisSum > maxSum) {
                    maxSum = thisSum;
                }
            }
        }
        return maxSum;
    }
```

该种解法最简单暴力，定义子序列的起始位置为`i`，结束位置为`j`，假设数组a的长度为`N`，当$i=0$时，$j=0,1,2,3,\ldots,N-1$，共`N`种情况，当$i=1$时，$j=1,2,3,\ldots,N-1$，共`N-1`种情况，以此类推，当$i=N-1$时，$j=N-1$，仅此一种情况；将`i`与`j`之间的所有元素和记为`thisSum`，一旦`thisSum`的值比`maxSum`大，就更新`maxSum`的值为`thisSum`。

第一个循环大小为`N`，第二个循环大小为`N-i`，第三个循环大小为`j-i+1`，则总运行次数和为：$\sum_{i=0}^{N-1}\sum_{j=i}^{N-1}\sum_{k=i}^{j}1$ 首先有：$\sum_{k=i}^{j}1=j-i+1$ 接着：$\sum_{j=i}^{N-1}(j-i+1)=\frac{(N-i+1)(N-i)}{2}$ 那么：

$$\begin{aligned}\sum_{i=0}^{N-1}\frac{(N-i+1)(N-i)}{2}&=\sum_{i=1}^{N}\frac{(N-i+1)(N-i+2)}{2}\\&=\frac{1}{2}\sum_{i=1}^{N}i^2-(N+\frac{3}{2})\sum_{i=1}^{N}i+\frac{1}{2}(N^2+3N+2)\sum_{i=1}^{N}1\\&=\frac{1}{2}\frac{N(N+1)(2N+1)}{6}-(N+\frac{3}{2})\frac{N(N+1)}{2}+\frac{N^2+3N+2}{2}N\\&=\frac{N^3+3N^2+2N}{6}\end{aligned}$$

所以该种解法的时间复杂度为 $O(\frac{N^3+3N^2+2N}{6})=O(N^3)$

2，时间复杂度为 $O(N^2)$的解法

```java
    public static int maxSubSum2(int[] a) {
        int maxSum = 0;
        for (int i = 0; i < a.length; i++) {
            int thisSum = 0;
            for (int j = i; j < a.length; j++) {
                thisSum += a[j];
                if (thisSum > maxSum) {
                    maxSum = thisSum;
                }
            }
        }
        return maxSum;
    }
```

在第一种解法中，拿掉最里面的那层循环，并稍做改动，就是现在的解法2。

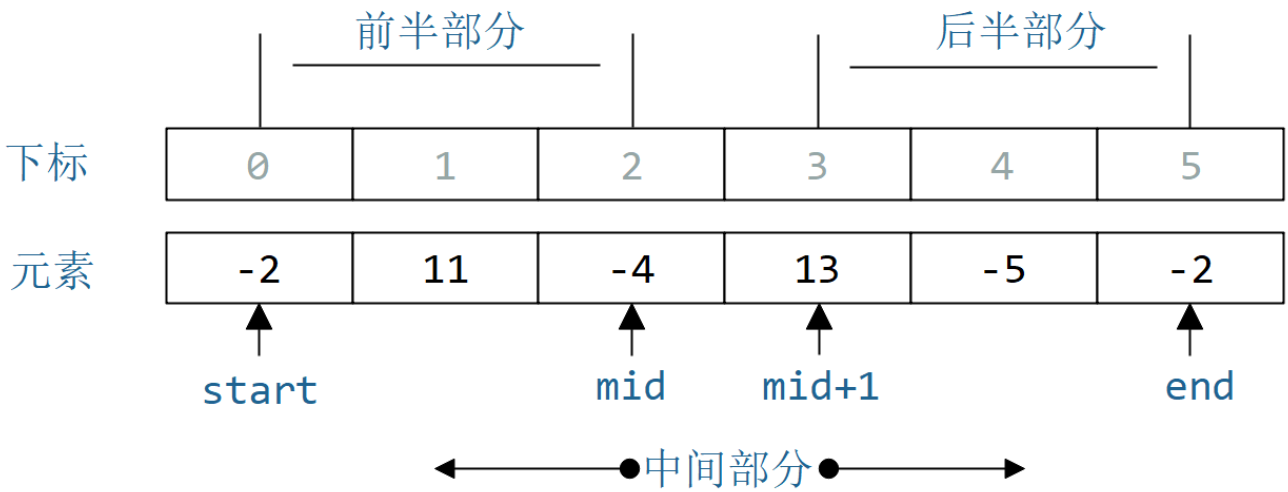
其中第一层循环大小为`N`，第二层循环为`N-i`，则总运行次数为：$\sum_{i=0}^{N-1}\sum_{j=i}^{N-1}1$ 其中：$\sum_{j=i}^{N-1}1=N-1-i+1=N-i$ 那么：

$$\begin{aligned}\sum_{i=0}^{N-1}(N-i) &= N\sum_{i=0}^{N-1}1-\sum_{i=0}^{N-1}i \\ &= N(N-1+1)-\frac{(N-1)N}{2} \\ &= \frac{N^2-N}{2}\end{aligned}$$

所以第二种解法的时间复杂度为 $O(\frac{N^2-N}{2})=O(N^2)$

3，时间复杂度为 $O(NlogN)$的解法

如下图所示，可以将数组分为三部分，分别为前中后三部分。



最大子序列和就可能出现在这三个部分中，其中 $mid=\frac{start+end}{2}=\frac{0+5}{2}=2$，前半部分是从`start`到`mid`这一部分的元素，即 $-2,11,-4$，所以该部分最大元素为`11`；后半部分是从`mid+1`到`end`这一部分的元素，即 $13,-5,-2$，所以该部分最大元素为`13`；而中间部分元素是以`mid`起始，分别向左和向右进行累加计算，分别求出其向左和向右部分的最大值，从`mid`向左得到其最大值：$-4+11=7$，而向右是从`mid+1`开始算起得到其最大值：13，最后将左右两部分和相加即为中间部分的最大值：$7+13=20$；比较前中后部分的最大值，发现中间部分的值`20`最大，所以该数组最大啊子序列和为`20`。

那么在程序中如何实现呢？这就要采用`分治策略`，将数组`a`分为前后两半子数组`b,c`，再将前半数组`b`分为前后两半子数组`d,e`，后半数组`c`分为前后两半子数组`f,g`，……，直到数组不能再分为止，此时子数组中就只有一个元素，一个元素就好判断了，该元素为正就直接把该元素值返回给上一级子数组，为负就返回0，然后回到上一级子数组，将之前返回的前后部分子数组的最大值与中间部分最大值进行比较，得出其最大值，接着将最大值返回其上一级子数组，直至回到原数组，这时原数组就得到了前后部分子数组的最大值，接着求出中间部分子数组的最大值并与前后部分进行比较即可得到整个数组的最大子序列和。

Talk is cheap, show code :

```
public static int maxSubSum3(int[] a) {
        return a.length > 0 ? maxSumRec(a, 0, a.length - 1) : 0;
    }

    private static int maxSumRec(int[] a, int left, int right) {
        if (left == right) {
            if (a[left] > 0) {
                return a[left];
            } else {
                return 0;
            }
        }

        int center = (left + right) / 2;
        int maxLeftSum = maxSumRec(a, left, center);
        int maxRightSum = maxSumRec(a, center + 1, right);

        int maxLeftBorderSum = 0;
        int leftBorderSum = 0;
        for (int i = center; i >= left; i--) {
            leftBorderSum += a[i];
            if (leftBorderSum > maxLeftBorderSum) {
                maxLeftBorderSum = leftBorderSum;
            }
        }

        int maxRightBorderSum = 0;
        int rightBorderSum = 0;
        for (int i = center + 1; i <= right; i++) {
            rightBorderSum += a[i];
            if (rightBorderSum > maxRightBorderSum) {
                maxRightBorderSum = rightBorderSum;
            }
        }

        return max3(maxLeftSum, maxRightSum,
                maxLeftBorderSum + maxRightBorderSum);
    }

    private static int max3(int a, int b, int c) {
```

```
41          return a > b ? a > c ? a : c : b > c ? b : c;
42      }
```

其中`center`为数组中间元素的下标，`maxLeftSum`和`maxRightSum`分别为数组前后部分的最大值，`maxLeftBorderSum`为中间部分向左计算的最大值，`maxRightBorderSum`为中间部分向右计算最大值；`maxLeftBorderSum + maxRightBorderSum`即为中间部分的最大值。

计算中间部分，即计算`maxLeftBorderSum`和`maxRightBorderSum`总花费时间为 N，而计算前后两半部分，即`maxLeftSum`和`maxRightSum`每个花费 $T(N/2)$个时间单元，则总共花费时间： $T(N) = 2T(N/2) + N$ 其中 $T(1) = 1$，则 $T(2) = 4 = 2 * 2$，$T(4) = 12 = 4 * 3$，$T(8) = 32 = 8 * 4$，$T(16) = 80 = 16 * 5$。

那么当 $N = 2^k$，则 $T(N) = N * (k + 1) = N(logN + 1)$，忽略低阶项，所以该方法的时间复杂度为：$O(NlogN)$。

4，时间复杂度为 $O(N)$的解法

```
1  public static int maxSubSum4(int[] a) {
2          int maxSum = 0;
3          int thisSum = 0;
4
5          for (int i = 0; i < a.length; i++) {
6              thisSum += a[i];
7
8              if (thisSum > maxSum) {
9                  maxSum = thisSum;
10             } else if (thisSum < 0) {
11                 thisSum = 0;
12             }
13         }
14
15         return maxSum;
16     }
```

此种方法将时间复杂度优化到了 $O(N)$，只需一轮循环即可找到最大子序列；其思路为：若当前子序列的和`thisSum`为负数，则将`thisSum`置为0，下一个数组元素作为新的子序列的起始位置，`thisSum`从该元素开始累加，直至找到最大子序列的和。

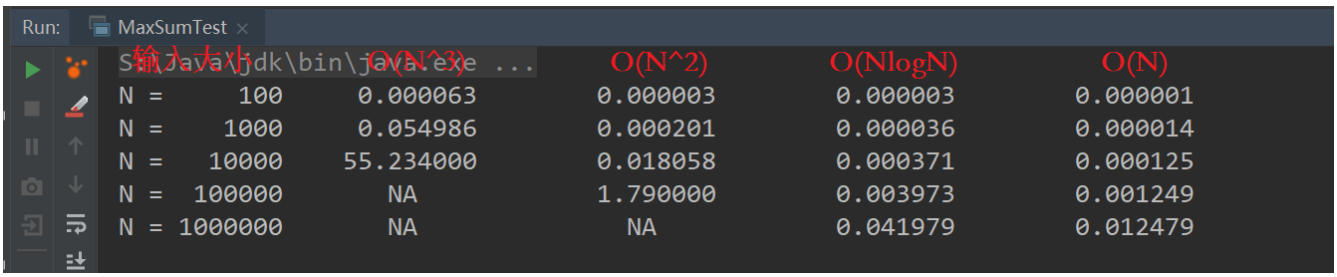
5，对比分析

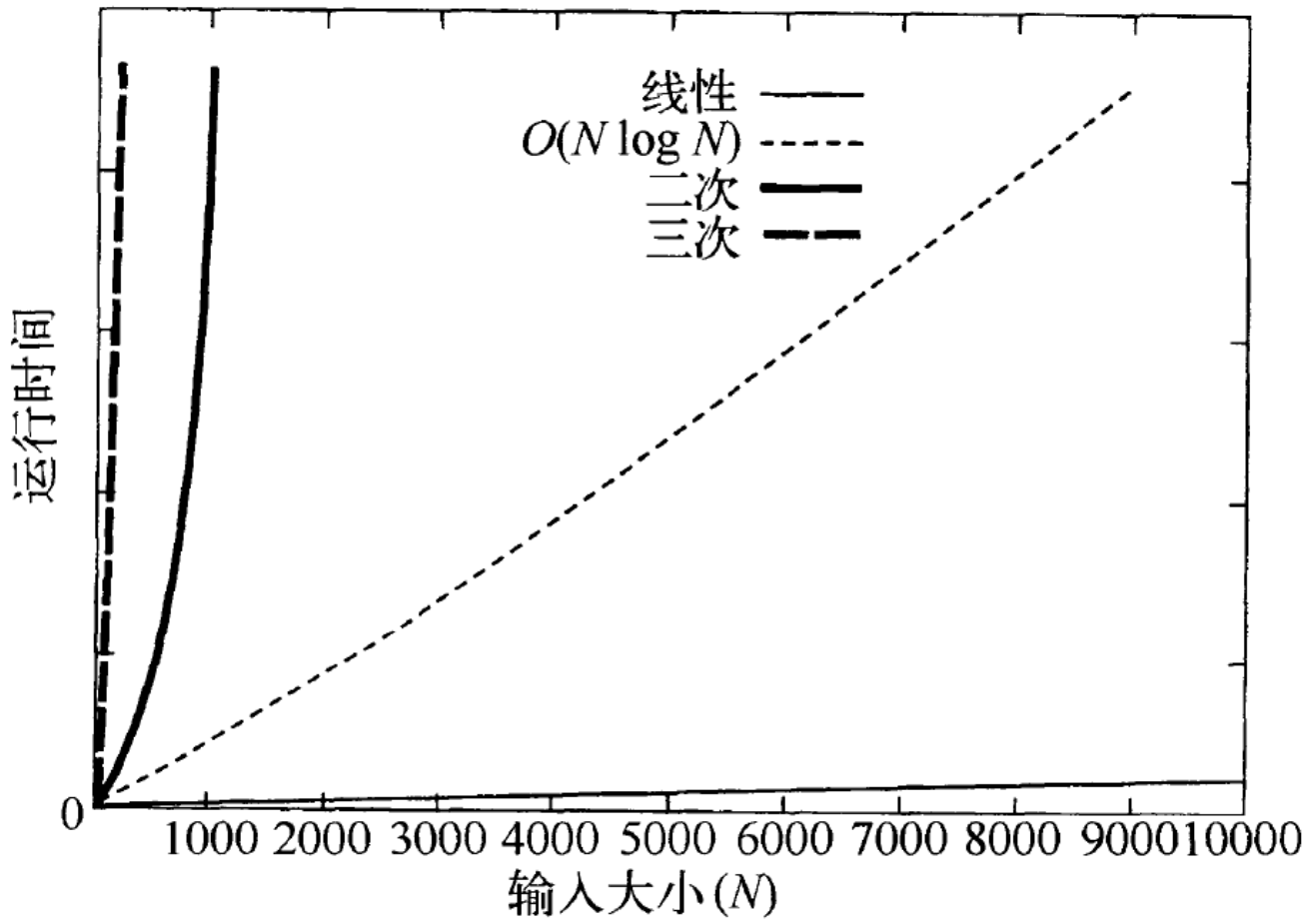
使用下面代码测试上述4中解法所消耗的时间：

```
public static void getTimingInfo(int n, int alg) {
        int[] test = new int[n];
        Random rand = new Random();

        long startTime = System.currentTimeMillis();
        long totalTime = 0;

        int i;
        for (i = 0; totalTime < 4000; i++) {
            for (int j = 0; j < test.length; j++) {
                test[j] = rand.nextInt(100) - 50;
            }
            switch (alg) {
                case 1:
                    maxSubSum1(test);
                    break;
                case 2:
                    maxSubSum2(test);
                    break;
                case 3:
                    maxSubSum3(test);
                    break;
                case 4:
                    maxSubSum4(test);
                    break;
                default:
            }

            totalTime = System.currentTimeMillis() - startTime;
        }
        System.out.print(String.format("\t%12.6f",
                (totalTime * 1000 / i) / (double) 1000000));
    }

    public static void main(String[] args) {
        for (int n = 100; n <= 1000000; n *= 10) {
            System.out.print(String.format("N = %7d", n));

            for (int alg = 1; alg <= 4; alg++) {
```

```
40                  if ((alg == 1 && n > 50000) || (alg == 2 && n >
    500000)) {
41                      System.out.print("\t      NA    ");
42                      continue;
43                  }
44                  getTimingInfo(n, alg);
45              }
46              System.out.println();
47          }
48      }
```

运行结果如下图，当预测时间过长，将其设为`NA`，从图中可以看出，不同时间复杂度的程序虽然得出的结果是一样的，但运行性能相差巨大，犹如波音与摩拜的差别。





> 总结：以后写代码之前要多思考，避免一上来就暴力求解，造成巨大的性能开销，应尽量将程序优化到线性阶或线性对数阶以内。