北 京 林 业 大 学

2019 学年—2020 学年第 2 学期计算机算法设计与实践实验报告书

专业：计算机科学与技术(创新实验班)　班级：计创 18

姓　　名：连月菡　　学　　号：181002222

实验地点：家　　任课教师：王春玲

实验题目：实验 4 最大子段和问题

实验环境：Visual Studio 2019 Community

一、 实验目的

（1）深刻掌握动态规划法的设计思想并能熟练运用;

（2）理解这样一个观点：**同样的问题可以用不同的方法解决，一个好的算法是反复努力和重新修正的结果。**

二、 实验内容

给定由n个整数(可能有负整数)组成的序列$(a_1,a_2,\cdots,a_n)$，求该序列形如的子段和的最大值，当所有整数均为负整数时，其最大子段和为0。

三、 实验结果

主函数

```
1.  int main()
2.  {
3.      int num;
4.      int arr[200007];
5.      int ans;
6.      LARGE_INTEGER c1;//开始计时
7.      LARGE_INTEGER c2;//结束计时
8.      LARGE_INTEGER frequency;//计时器频率
9.      QueryPerformanceFrequency(&frequency);
10.     double quadpart = (double)frequency.QuadPart;
11.     scanf("%d", &num);
12.     for (int i = 0; i < num; ++i)
13.         scanf("%d ", &arr[i]);
14.     QueryPerformanceCounter(&c1);
15.     ans = zero(bf(arr, num));//01 蛮力法
16.     QueryPerformanceCounter(&c2);
```

```
17.     cout << "高精度计数器用时：
    " << (double)((c2.QuadPart - c1.QuadPart) * 1.0 / quadpart * 1.0) * 1000000
    << endl;
18.
19.     cout << "最大子段和是" << ans << endl;
20.     QueryPerformanceCounter(&c1);
21.     ans = zero(dc(arr, 0, num - 1));//02 分治法
22.     QueryPerformanceCounter(&c2);
23.     cout << "高精度计数器用时：
    " << (double)((c2.QuadPart - c1.QuadPart) * 1.0 / quadpart * 1.0) * 1000000
    << endl;
24.
25.     cout << "最大子段和是" << ans << endl;
26.     QueryPerformanceCounter(&c1);
27.     ans = zero(dp(arr, num));//03 动态规划法
28.     QueryPerformanceCounter(&c2);
29.     cout << "高精度计数器用时：
    " << (double)((c2.QuadPart - c1.QuadPart) * 1.0 / quadpart * 1.0) * 1000000
    << endl;
30.
31.     cout << "最大子段和是" << ans << endl;
32.     return 0;
33. }
```

（1） 分别用**蛮力法**、**分治法**和**动态规划法**设计最大子段和问题的算法；

蛮力法求解最大子段和问题的设计思想很简单，依次从第 1 个数开始计算长度为 1，2，…，n 的子段和，将这一阶段的最大子段和保存起来；再从第 2 个数开始计算长度为 1，2，…，n-1 的子段和，将这一阶段的最大子段和与前一阶段求得的最大子段和比较，取较大者保存起来；以此类推，最后保存的即是整个序列的最大子段和。

蛮力法

Input: Num (n 个数字)

Output:Maxium Subarray Sum (最大子段和)

1. **子段长度从 1 开始,枚举出该长度的所有子数组并求其和,每次都和当前最大字段和进行比较。**
2. **从长度 1-n,不断更新当前最大字段和。**
3. **遍历结束时的当前最大子段和就是这个数组的最大子段和,若为结果小于 0,则最大子段和为 0。**

```
1.   int zero(int n) {//如果最大子段和小于 0,则返回 0;
```

```
2.      return n < 0 ? 0 : n;
3.  }
4.  int bf(int arr[], int x) {//BruteForce 01 蛮力法
5.      int ans = -1;//因为如果答案为负数,那么必为 0,如果遇到比负数大的子段和,则更新
6.      int size = 0;//目标子段长度
7.      for (int i = 1; i <= x; ++i)//从长度 1 到数组中的所有数字个数
8.      {
9.          size = i;//目标子段长度
10.         int nownum = 0;//当前计算的子段中的数字个数
11.         int nowsum = 0;//当前计算的子段的和
12.         int beg = 0;//从头开始遍历
13.         int j = beg;//j 作为起始下标
14.         for (; j < x; ++j)//从起始位置一直到最远位置(除非容量超过目前的目标子段长
    度  nownum>size)
15.         {
16.             nownum++;//当前子段长度 +1
17.             if (nownum > size)//容量超过目前的目标子段长度
18.             {
19.                 nowsum = 0; nownum = 0;//因为要重新计算子段和,所以当前和与当前子
    段长度清零
20.                 beg++;//数组后移,计算新的子段
21.                 j = beg;//因为 j 不会自己变更为起始坐标
22.                 continue;
23.             }
24.             nowsum += arr[j];//累加子段和
25.             if (nowsum > ans)//如果大于目前最大子段和,则赋值给 ans
26.                 ans = nowsum;//更新 ans 大小
27.         }
28.     }
29.     return ans;
30. }
```

分治法

分治法
Input: Num (n 个数字) **Output:Maxium Subarray Sum (最大子段和)** **1.将给定的数组分成两半。** **2.返回下面三个操作的最大值** **a.对左半部分进行同样的分半递归操作，得到左半部分的最大子段和。** **b.对右半部分进行同样的分半递归操作，得到右半部分的最大子段和。** **c.得到穿过数组重点的子数组的最大子段和：找到从中点开始并在中点左边某点处结束的最大和，然后找到从 mid+ 1 开始并以 mid+ 1 右边的求和点结束的最大和。最后，将两者合并,并返回。**

```
1.  int maxn(int a, int b) {//求两数中的较大数
2.      return a > b ? a : b;//若 a>b,则返回 a,反之返回 b
3.  }
4.
5.  int maxn(int a, int b, int c) {//求三个数中的最大数,重载
6.      return maxn(maxn(a, b), c);//调用上面的比较两个数的 max 函数
7.  }
8.
9.  int crossing(int arr[], int l, int mid, int later)//计算含有中点的最大子段和
10. {
11.     int nowsum = 0;//当前子段和
12.     int lsum = -1;//左侧最大子段和
13.     for (int i = mid; i >= l; --i)//从中间向左移动
14.     {
15.         nowsum += arr[i];//累加子段数字
16.         if (nowsum > lsum)//如果当前子段和大于用于记录左侧最大子段和的 lsum
17.             lsum = nowsum;//则赋值记录
18.     }
19.     nowsum = 0;//注意初始化当前子段和
20.     int rsum = -1;//右侧最大子段和
21.     for (int i = mid + 1; i <= later; ++i)//从中间向右移动
22.     {
23.         nowsum += arr[i];//累加子段数字
24.         if (nowsum > rsum)//如果当前子段和大于用于记录右侧最大子段和的 rsum
25.             rsum = nowsum;//则赋值记
26.     }
27.     return lsum + rsum;//左右最大子段和之和就是包含中点的最大子段和
28. }
29. int dc(int arr[], int l, int later) {//Divide and Conquer 02 分治法
30.     if (l == later)//首尾下标重合
31.         return arr[l];//返回该点值
32.     int mid = (l + later) / 2;//取中点
33.     return maxn(dc(arr, l, mid), dc(arr, mid + 1, later), crossing(arr, l, m
    id, later));
34. }
```

动态规划法求解最大子段和问题的关键是要确定**动态规划函数**。记

$$\mathbf{b[j]}=\max_{1\le i\le j}\left\{\sum_{k=i}^{j}a[k]\right\}\quad(1\le j\le n)$$

则 $$\max_{1\le i\le j\le n}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}\max_{1\le i\le j}\sum_{k=i}^{j}a[k]=\max_{1\le j\le n}b[j]$$

由 b[j]的定义，当 b[j-1]>0 时，b[j]=b[j-1]+a[j]，否则，b[j]=a[j]。可得如下递推式：

b[j]=max{ b[j-1]+a[j], a[j] }, 1≤*j*≤*n*

动态规划法

Input: Num (n 个数字)

Output:Maxium Subarray Sum (最大子段和)

1. **初始化 dp 数组 b,且 b[0]=0,其下标由 p 定位,p 初始化为 1**
2. **在 for 循环中,给定数组 num 的下标由 j 定位,j 初始化为 0,且 j∈[0,n-1]。**
   a.每次循环,j 和 p 都加一,代表数组向后移动。
   b.每次循环,计算上一步的结果 b[p-1]是否需要加上当前遍历到的给定数组下标指向的数值 n[j],如果加上更大,或者不加更大,都把更大值赋给当前步结果 b[p]。
   c.如果 b[p]大于 ans(answer,最大子段和),那么把 b[p]的值赋给 ans。

3.循环结束,返回 ans

```
1.  int b[2001];
2.  int dp(int n[], int x) {//Dynamic Programming 03 动态规划法
3.      memset(b, 0, sizeof(b));//初始化记录每步最大子段和的 dp 数组
4.      int ans = 0;//最大子段和
5.      int j = 0;//数组的下标 0~(num-1)
6.      int p = 1;//dp 数组的下标
7.      for (; j < x; ++j, ++p)//每次循环,下标后移
8.      {
9.          b[p] = max(b[p - 1] + n[j], n[j]);//由动态规划函数进行计算
10.         if (b[p] > ans) ans = b[p];//如果此步子段和大于最大子段和,则赋值记录
11.     }
12.     return ans;
13. }
```

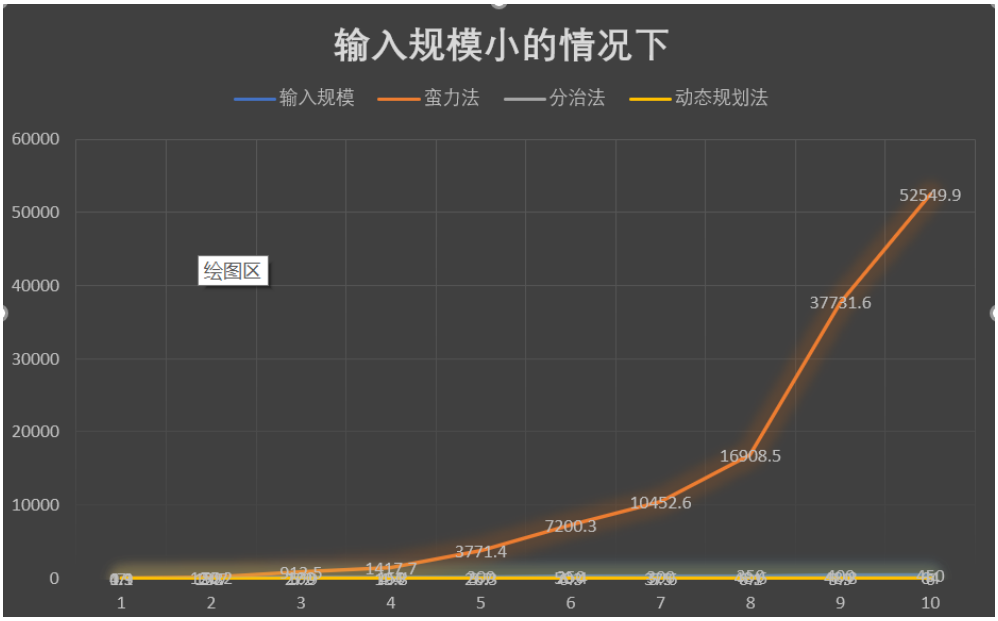
（2） 比较不同算法的时间性能；

蛮力法: O(n^2).
分治法: O(n log(n))
动态规划法:O(n)
由上可以看出，三种算法的时间效率高低顺序为 动态规划>分治>蛮力

```
Microsoft Visual Studio 调试控制台
7
2 -4 3 -1 2 -4 3
0
高精度计数器用时：0.9
最大子段和是4
高精度计数器用时：1.3
最大子段和是4
高精度计数器用时：4.1
最大子段和是4
```

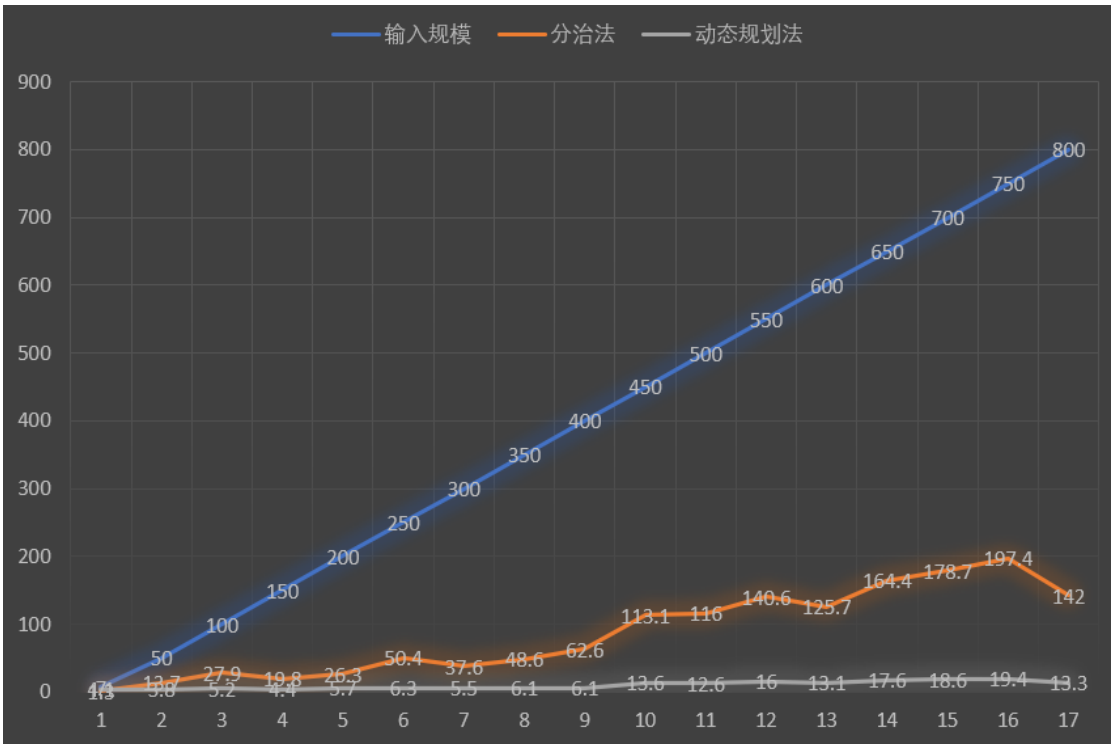
从上到下,依次是蛮力法,分治法,和动态规划法得到的答案和时间。
多次实验,得到以下数据:

输入规模	7	50	100	150	200	250	300	350	400	450
蛮力法	0.9	120.2	912.5	1417.7	3771.4	7200.3	10453	16908.5	37731.6	52550
分治法	1.3	13.7	27.9	19.8	26.3	50.4	37.6	48.6	48.8	64
动态规划法	4.1	3.8	5.2	4.4	5.7	6.3	5.5	6.1	5.9	6



在输入规模小的情况下可以看出蛮力法的用时增长速度快,但是看不出分治法和动态规划法的区别。

输入规模	7	50	100	150	200	250	300	350	400	450	500	550	600	650	700	750	800
分治法	1.3	13.7	27.9	19.8	26.3	50.4	37.6	48.6	62.6	113.1	116	141	125.7	164.4	178.7	197.4	142
动态规划法	4.1	3.8	5.2	4.4	5.7	6.3	5.5	6.1	6.1	13.6	12.6	16	13.1	17.6	18.6	19.4	13.3



在输入规模较大的情况下可以看出分治法的用时符合 nlogn 的时间复杂度,动态规划法时间复杂度更小,增长更缓慢。

（3） 给出测试数据，并写出程序文档。
Github_测试用例_链接

四、 实验中存在的问题及解决办法

Q1:VS2019 的 Terminal 输入超过一定长度的输入数据,会 Runtime Error
A1:使用 freopen 等函数进行重定向,从文件里读入输入的大规模数据。