算法设计与分析实验报告

实验题目：最大字段和三种实现算法的时间复杂度分析

姓名：陈俊卉

学号：2020212256

日期：2022.9.27

一、实验环境

(列出实验的操作环境，如操作系统，编程语言版本等，更多信息可视各自实际情况填写)
① 操作系统：windows 10
② 编程语言：c++
③ 编程工具：vscode 及其组件

二、实验内容

具体要求请参照实验指导书中各实验的“实验内容及要求”部分。
(示例：1. 描述你对实验算法的设计思路;2. 给出算法关键部分的说明以及代码实现截图；3. 对测试数据、测试程序(没有要求则非必须)进行说明，如测试覆盖程度，最好最坏平均三种情况等等，并给出测试结果截图等信息)

1.算法的设计与实现

(1) 暴力枚举 [O(N^3)]

① 实验代码：

```
int MaxSubsequenceSum1(const int A[],int N) {
    /* ((O(1)+O(j-i+1)+O(2))*O(N-i))*N + 1 */
    int ThisSum,MaxSum,i,j,k;
    MaxSum = 0;                          /* O(1) */
    for (i = 0; i < N; i++) {            /* ((O(1)+O(j-i+1)+O(2))*O(N-i))*N */
        for (j = i; j < N; j++) {        /* (O(1)+O(j-i+1)+O(2))*O(N-i) */
            ThisSum = 0;                 /* O(1) */
            for (k = i; k <= j; k++) {   /* O(j-i+1) */
                ThisSum += A[k];
            }
            if(ThisSum > MaxSum)         /* each loop: O(1)+O(1) */
                MaxSum = ThisSum;        /* each loop: O(1) */
        }
    }
    return MaxSum;
}
/* j-i+1<N,N-i<N, so  ((O(1)+O(j-i+1)+O(2))*O(N-i))*N + 1 <= O(N)*O(N)*O(N) +1 =
O(N^3)    */
```

② 原理：

枚举所有的子列，并将所有的子列和与 MaxSum 比较。

(2) 暴力枚举 [O(N^2)]

① 实验代码：

```
int MaxSubsequenceSum2(const int A[],int N)  /* O(N^2) */
{
    int ThisSum,MaxSum,i,j;
    MaxSum = 0;
    for(i = 0;i < N; i++)                    /* (O(1)+(O(1)+O(1)+O(1))*(N-i))*N */
    {
        ThisSum = 0;                         /* O(1) */
        for(j = i;j < N;j++)                 /* (O(1)+O(1)+O(1))*(N-i) */
        {
            ThisSum += A[j];                 /* O(1) */
            if(ThisSum > MaxSum)             /* each loop: O(1)+O(1) */
            {
                MaxSum = ThisSum;            /* each loop: O(1) */
            }
        }
    }
    return MaxSum;
}
```

② 原理：

与(1)类似，枚举所有的子列，并将所有的子列和与 MaxSum 比较。但优化在：子列和其实可以从第二个循环里直接累加而来，而不需要像①一样每一次都重新进行遍历相加。

(3) 分治 [O(NlogN)]

① 实验代码：

```
int MaxSubsequenceSum3(const int A[],int left,int right) // T(n) = T(n/2) + n
=>   O(nlogn)
{
    int leftmax;
    int rightmax;
    int search_left_max;
    int search_left = 0;
```

```
int search_right_max;
int search_right = 0;
int midmax;
int mid = (left + right) /2;
int ThisSum;
// finished condition
if (left == right)
{
    return A[left];
}
else
{
    // maxsum through mid
    // search left
    for(int i = mid;i >= left; i--)
    {
        search_left += A[i];
        if(i == mid)
        {
            search_left_max = search_left;
        }
        else{
            if(search_left > search_left_max)
            {
                search_left_max = search_left;
            }
        }
    }

    // search right
    for(int i = mid + 1; i<=right; i++)
    {
        search_right += A[i];
        if(i == mid + 1)
        {
            search_right_max = search_right;
        }
        else{
            if(search_right > search_right_max)
            {
                search_right_max = search_right;
            }
        }
    }
```

```
        midmax = search_left_max + search_right_max;

        // maxsum for left
        leftmax = MaxSubsequenceSum3(A,left,mid);
        // maxsum for right
        rightmax = MaxSubsequenceSum3(A,mid+1,right);

        return fmax(fmax(leftmax,rightmax),midmax);

    }
}
```

② 原理：

使用分治思想：但对于每一段分治子列，其子列有三种可能：左侧子列，右侧子列与跨越 mid 的子列。其中左侧子列和右侧子列的 maxsum 可以通过分治递归计算得出，而跨越 mid 的子列需要在递归返回时使用 for 循环从中间开始向左向右延伸求得此种子列最大值。于是我们可以得出时间复杂度的递推公式：T(N) = T(N/2) + N. 经过计算得出复杂度为 O(NlogN).

(4) 动态规划 O(N)

①实验代码：

```
int MaxSubsequenceSum4(const int A[],int N) { /* O(1)+O(N)*(O(1)+O(1)+O(1))  */
    int ThisSum,MaxSum,j;
    ThisSum = MaxSum = 0;           /* O(1) */
    for (j = 0; j < N; j++) {       /* total: O(N)*(O(1)+O(1)+O(1)) */
        ThisSum += A[j];           /* each loop: O(1)      ; total: O(N)*O(1) */

        /* if-else part total for each loop:   O(1)+O(1)     ; total:
O(N)*(O(1)+O(1))   */

        if(ThisSum > MaxSum) {       /* each loop: O(1)+O(1) ; total: O(N)*(O(1)+O(1))
*/
            MaxSum = ThisSum ;       /* each loop: O(1)      ; total: O(N)*O(1) */
        } else if (ThisSum < 0) {    /* each loop: O(1)+O(1) ; total: O(N)*(O(1)+O(1))
*/
            ThisSum = 0;             /* each loop: O(1)      ; total: O(N)*O(1) */
        }
    }
    return MaxSum;
}
/*  O(1)+O(N)*(O(1)+O(1)+O(1)) = O(N)  */
```

② 原理：

只考虑累加得到正数。若累加后得到负数，则直接舍弃(ThisSum = 0)，而 MaxSum 会被保留。

但这是以长度为 0 也作为子列作为条件的。即若所有的数都是负数，其子列长度取 0，最大和取 0。同样地，ppt 所提供的暴力枚举也存在这个问题。其原因是 MaxSum 初始化应该为最小负数，而不应该是 0。

若考虑子列长度至少为 1，则代码应该如下（同样为 O(N)）：

```
int MaxSubsequenceSum5(const int A[],int N) {
    int dp[N];
    int MaxSum = A[0];
    dp[0] = A[0];
    for (int i = 1;i < N; i++) {
        dp[i] = fmax(dp[i-1] + A[i] , A[i]);
        MaxSum = fmax(MaxSum, dp[i]);
    }
    return MaxSum;
}

/*  O(N)  */
```

dp 数组内表示**以该下标为结尾的子列和的最大值**，事实上并不需要这个 dp 数组，只需要 ThisSum 表示即可：

```
int MaxSubsequenceSum6(const int A[],int N) {
    int MaxSum = A[0];
    int ThisSum = A[0];
    for (int i = 1;i < N; i++) {
        ThisSum = fmax(ThisSum + A[i] , A[i]);
        MaxSum = fmax(MaxSum, ThisSum);
    }
    return MaxSum;
}

/*  O(N)  */
```

2. 测试

测试结果如下图所示。

N = 10：

```
create pseudo-random list:
-59 76 3 69 -26 -54 -79 -88 -72 43
Sum1 result:148
Sum1 running time:0s
Sum2 result:148
Sum2 running time:0s
Sum3 result:148
Sum3 running time:0s
Sum4 result:148
Sum4 running time:0s
Sum5 result:148
Sum5 running time:0s
Sum6 result:148
Sum6 running time:0s
```

N = 100：

```
create pseudo-random list:
-59 76 3 69 -26 -54 -79 -88 -72 43 -23 -95 66 44 12 -11 81 -17 -97 -91 -70 32 -17 53 -9 21 35 -77 -80 97 -80 -82 -2 -19 -40 -87 40 69 51 -55 61 -7
 67 83 86 -59 -34 -27 13 -66 -80 7 0 95 38 96 -20 -7 93 75 -45 -80 -20 -67 -13 57 96 -2 69 36 10 -13 -89 -4 -51 -26 32 47 -28 31 32 61 65 -45 -37
82 42 40 -38 78 -40 -22 4 -33 10 -69 -62 -28 -85 -41
Sum1 result:739
Sum1 running time:0.000996s
Sum2 result:739
Sum2 running time:0s
Sum3 result:739
Sum3 running time:0s
Sum4 result:739
Sum4 running time:0s
Sum5 result:739
Sum5 running time:0s
Sum6 result:739
Sum6 running time:0s
```

N = 1000：

```
Sum1 result:1502
Sum1 running time:0.422641s
Sum2 result:1502
Sum2 running time:0.001992s
Sum3 result:1502
Sum3 running time:0s
Sum4 result:1502
Sum4 running time:0s
Sum5 result:1502
Sum5 running time:0s
Sum6 result:1502
Sum6 running time:0s
```

N = 10000：（算法一不参与测试）

```
Sum2 result:8808
Sum2 running time:0.161569s
Sum3 result:8808
Sum3 running time:0.000997s
Sum4 result:8808
Sum4 running time:0s
Sum5 result:8808
Sum5 running time:0.000998s
Sum6 result:8808
Sum6 running time:0s
```

N = 100000：(算法一不参与测试)

```
Sum2 result:17847
Sum2 running time:10.8486s
Sum3 result:17847
Sum3 running time:0.010004s
Sum4 result:17847
Sum4 running time:0s
Sum5 result:17847
Sum5 running time:0.00298s
Sum6 result:17847
Sum6 running time:0.000997s
```

制表如下（以本机测试为准）：

Algorithm		1	2	3	4	5	6
Time		O(N^3)	O(N^2)	O(NlogN)	O(N)	O(N)	O(N)
Input size	N=10	0	0	0	0	0	0
	N=100	0.000996	0	0	0	0	0
	N=1000	0.422641	0.001992	0	0	0	0
	N=10000	NA	0.161569	0.000997	0	0.000998	0
	N=100000	NA	10.8486	0.0010004	0	0.00298	0.000997

三、出现问题及解决

(列出你在实验中遇到了哪些问题以及是如何解决的)

问题一：使用 clock 精度不够，最多只精确到毫秒，难以看出算法区别。

解决方案：使用 c++的 chrono 库解决。Chrono 库提供了 microsecond 级别的时间准确度，从而能够更加精确地比较算法之间的区别。

问题二：Sum5 中 O(N)比其他 O(N)的代码慢。

原因可能是数组的写入写出需要的时间比变量的写入时间长。

问题三：ppt 所提供的算法代码：

ppt 算法代码是以长度为 0 也作为子列作为条件的。即若所有的数都是负数，按照算法，其最大子列长度取 0，最大和取 0。而笔者认为子列的最小长度应该为 1 才合理。这也在笔者所写的 O(NlogN)代码和修改后的 O(N)代码有所体现。而若答案为正数，二者输出是一样的。

若仍采用 ppt 的代码，应该将 MaxSum 初始化为最小负数，而不应该是 0。

四、总结

(对所实现算法的总结评价，如时间复杂度，空间复杂度，是否有能够进一步提升的空间，不同实现之间的比较，不同情况下的效率，通过实验对此算法的认识与理解等等)

通过本次实验，我了解了最大字段和的暴力枚举、递归和动态规划算法，并且对他们进行了时间复杂度分析，进一步通过测试得到了实验数据，并对实验数据与理论不合理的地方进行了探讨并得出结论。

此外，还对 ppt 上的算法进行分析并提出了理解的不同之处，并加以修改得到新的代码。

此次实验令我对算法以及算法的测试有了具体的认知，这对我今后的算法分析学习有着莫大的帮助。