算法分析第五章

1. 最大子段和问题：给定整数序列 $a_1, a_2, \cdots, a_n$，求该序列形如 $\sum_{k=i}^{j} a_k$ 的子段和的最大值： $\max\left\{0, \max_{1\le i\le j\le n} \sum_{k=i}^{j} a_k\right\}$

1) 已知一个简单算法如下：

```
int Maxsum(int n,int a,int& besti,int& bestj)
{
    int sum = 0;
    for(int i=1;i<=n;i++){
      int suma = 0;
     for(int j=i;j<=n;j++){

        suma + = a[j];
        if(suma > sum){
          sum = suma;
          besti = i;
          bestj = j;
          }
       }
    }
    return sum;
}
```

试分析该算法的时间复杂性。

2) 试用分治算法解最大子段和问题，并分析算法的时间复杂性。

3) 试说明最大子段和问题具有最优子结构性质，并设计一个动态规划算法解最大子段和问题。分析算法的时间复杂度。（提示：令 $b(j) = \max_{1\le i\le j\le n} \sum_{k=i}^{j} a_k$，$j = 1, 2, \cdots, n$）。

1）时间复杂度为 O(n^2)

2）使用前缀和+递归分治算法，算法复杂度为 O(nlogn)，

[i ,j]范围内的最大字段和 为 max([i, mid], [mid+1, j], 包含 mid 的跨越两段的最大字段和)

关键代码如下，代码文件见 Maxsum.py

```
def max_crossing_sum(sum_arr, i, mid, j):
    left_sum = 0
    right_sum = 0
    for k in range(i, mid):
        if i == 0:
            if k == i:
```

```
                    left_sum = sum_arr[k]
                else:
                    left_sum = max(left_sum, sum_arr[k])
            else:
                if k == i:
                    left_sum = sum_arr[k] - sum_arr[i-1]
                else:
                    left_sum = max(left_sum, sum_arr[k] - sum_arr[i-1])
    for k in range(mid, j+1):
        if mid == 0:
            if k == mid:
                right_sum = sum_arr[k]
            else:
                right_sum = max(right_sum, sum_arr[k])
        else:
            if k == mid:
                right_sum = sum_arr[k] - sum_arr[mid-1]
            else:
                right_sum = max(right_sum, sum_arr[k] - sum_arr[mid-1])
    return left_sum + right_sum

# 分治
def max_sum(arr, i, j, sum_arr):
    if i == j:
        return arr[i]
    else:
        mid = (i + j) // 2
        left_sum = max_sum(arr, i, mid, sum_arr)
        right_sum = max_sum(arr, mid+1, j, sum_arr)
        cross_sum = max_crossing_sum(sum_arr, i, mid, j)
        return max(left_sum, right_sum, cross_sum)


if __name__ == '__main__':
    arr = [1, 2, -4, 4, -1, 6, -9, 8, 9]
    sum_arr = [arr[0]]
    # 前缀和
    for i in range(1, len(arr)):
        sum_arr.append(arr[i] + sum_arr[i-1])
    max_sum_value = max_sum(arr, 0, len(arr)-1, sum_arr)
```

3）最大字段和具有最优子结构：

$$dp[i] = \max (dp[i-1] + a[i] , a[i])$$

dp[i]表示以第 i 个数为结尾的子段的最大字段和

可以论证：dp[i]考虑了以 i 为结尾的，i 以及之前任意位置开头的字段

时间复杂度为 O(n)，代码文件见 MaxsumN.py

```python
arr = [1, 2, -4, 4, -1, 6, -9, 8, 9]

dp = [0] * len(arr)

for i in range(len(arr)):
    dp[i] = max(dp[i-1] + arr[i], arr[i])

print(max(dp))
```

2.（双机调度问题）用两台处理机 A 和 B 处理 n 个作业。设第 i 个作业交给机器 A 处理时所需要的时间是 a_i，若由机器 B 来处理，则所需要的时间是 b_i。现在要求每个作业只能由一台机器处理，每台机器都不能同时处理两个作业。设计一个动态规划算法，使得这两台机器处理完这 n 个作业的时间最短（从任何一台机器开工到最后一台机器停工的总的时间）。以下面的例子说明你的算法：

$$n = 6,(a_1,a_2,a_3,a_4,a_5,a_6) = (2,5,7,10,5,2),(b_1,b_2,b_3,b_4,b_5,b_6) = (3,8,4,11,3,4)$$

（提示：类似于 0/1 背包问题的点对表示，可以采用三元组 (T_A,T_B,δ) 表示调度状态，其中，T_A、T_B 分别表示机器 A，B 已占用的加工时间，δ 是一个整数，用以表示当前调度方案中安排给机器 A 的作业情况，例如，将作业 i 安排给机器 A，则 δ 增加 2^{i-1}。剩下来的工作就是计算调度的状态集并进行化简：

$$S_0 = \{(0,0,0)\}, S_i = \left((a_i,0,2^{i-1}) + S_{i-1}\right) \cup \left((0,b_i,0) + S_{i-1}\right),\ i = 1,2,\cdots,n$$

期间的状态集化简可根据某个设定的阈值进行）。

dp[i][sum_a]表示第 i 个作业在 A 机器处理时间为 sum_a 的时候 B 机器处理时间的最小值

则有状态转移方程：

dp[i][sum_a] = min (dp[i-1][sum_a-value(Ai)] , dp[i-1][sum_a] + value(Bi)])

这个方程分别考虑第 i 个作业放在 A 机器和 B 机器上的情况

根据例子：

dp 数组赋值为 inf

```
PS D:\研究生\算法分析与设计\作业程序> & D:/ProgramFilesFolder/05-Anaconda3/python.exe d:/研究生/算法分析与设计/作业程序/TwoTasksDP.py
dp[0][0]=0
dp[1][0]=3 dp[1][2]=0
dp[2][0]=11 dp[2][2]=8 dp[2][5]=3 dp[2][7]=0
dp[3][0]=15 dp[3][2]=12 dp[3][5]=7 dp[3][7]=4 dp[3][9]=8 dp[3][12]=3 dp[3][14]=0
dp[4][0]=26 dp[4][2]=23 dp[4][5]=18 dp[4][7]=15 dp[4][9]=19 dp[4][10]=15 dp[4][12]=12 dp[4][14]=11 dp[4][15]=7 dp[4][17]=4 dp[4][19]=8 dp[4][22]=3 dp[4][24]=0
dp[5][0]=29 dp[5][2]=26 dp[5][5]=21 dp[5][7]=18 dp[5][9]=22 dp[5][10]=18 dp[5][12]=15 dp[5][14]=14 dp[5][15]=10 dp[5][17]=7 dp[5][19]=11 dp[5][20]=7 dp[5][22]=4 dp[5][24
]=3 dp[5][27]=3 dp[5][29]=0
dp[6][2]=29 dp[6][4]=26 dp[6][5]=25 dp[6][7]=21 dp[6][9]=18 dp[6][10]=22 dp[6][11]=22 dp[6][12]=18 dp[6][14]=15 dp[6][15]=14 dp[6][16]=14 dp[6][17]=10 dp[6][19]=7 dp[6][
20]=11 dp[6][21]=11 dp[6][22]=7 dp[6][24]=4 dp[6][26]=3 dp[6][27]=7 dp[6][29]=3 dp[6][31]=0
The minimum sum of two tasks is: 15
```

最小值为 15，代码文件见 TwoTasksDP.py

3．考虑下面特殊的整数线性规划问题

$$\max \sum_{i=1}^{n} c_i x_i$$

$$\sum_{i=1}^{n} a_i x_i \le b, \quad x_i \in \{0,1,2\}, 1 \le i \le n$$

试设计一个解此问题的动态规划算法，并分析算法的时间复杂度。

在设计算法时可以优先考虑 m_i，也就是先判断背包剩下的容量能不能放进去 c_i，若可以再判断能否使 $p_i = 1$，若可以则就再放入一个 c_i

递推式：

$$m(k,x) = \begin{cases} -\infty & x < 0 \\ m(k-1,x) & 0 \le x < w_k \\ \max\{m(k-1,x),\ m(k-1,x-w_k)+p_k\} & w_k \le x < 2w_k \\ \max\{m(k-1,x),\ m(k-1,x-w_k)+p_k, m(k-1,x-2w_k)+2p_k\} & x \ge 2w_k \end{cases}$$

时间复杂度为 O(n * b) ，代码文件见 IntegerManage.py