实验一 分治与递归

1.1 实验内容

设有 n 个互不相同的元素 $x_1,x_2,\cdots, x_n$，每个元素 x_i 带有一个权值 w_i，且 $\sum_{i=1}^{n} w_i = 1$。若元素 x_k 满足$\sum_{x_i<x_k} w_i \le \frac{1}{2}$ 且$\sum_{x_i>x_k} w_i \le \frac{1}{2}$，则称元素 x_k 为 $x_1,x_2,\cdots, x_n$ 的带权中位数。请编写一个算法，能够在最坏情况下用 O(n)时间找出 n 个元素的带权中位数。

1.2 实验思路

参考快速排序算法思路。每次从数组中随机选取一个作为 pivot，将原数组划分为小于 pivot 和大于 pivot 两部分。分别计算这两部分数组的权重之和，则我们要找的带权中位数必然位于权重之和大于 1/2 的那半边。理想情况下，每次都可以排除大约一半的元素。可以证明，这个算法的平均时间复杂度是 O(n)。然而在最坏情况下，该算法需要$\Omega(n^2)$的计算时间，不满足题目要求。

根据参考书[1]2.9 节线性时间选择的思路，我们可以通过选择“中点的中点”作为 pivot 来进行划分。已证明，这种方法可以确保每次划分都可以将子数组大小至少缩小到初始数组的η倍，$(0 < \eta < 1)$。由此可得算法的时间复杂度 $T(n) = O(n)$.

1.3 算法伪代码

Input: 给定数组 a[p:r], 相应的权值 w[p:r], 初始划分权值 k=0.5，初始值 p=0, r=n-1, n 为原数组大小;

Step1: if (size of a[p:r] < 5) then bubbleSort(a[p:r]); 然后直接遍历 a[p:r]找到带权中位数; else goto Step2

Step2: 将数组 a[p:r]划分为每组 5 个元素的子数组（最后一组可能不够 5 个），用 bubbleSort 找到每组的中位数，再从这些中位数中找出中位数作为 pivot;

Step3: 用以上 pivot 对 a[p:r]做 Partition; 形成两个子数组 a[p:x], a[x+1:r];

Step4: w1 = sum(w[p:x]); w2 = sum(w[x+1:r]); if w1 > 0.5 then 对第一个数组重复 Step2; else 按如下规则更新划分权值：k -= sum(w[p:x]), 并对第二个数组重复 Step2

Output: 带权中位数

1.4 算法源代码

```
package algorithmHomework;
import java.util.Random;
import java.util.Scanner;
/**
 * 分治算法
 * @author Yulong Yang
 *
```

```java
 */
public class Task1_final {
    public static void main(String[] args) {

        //测试实验一

        Scanner scanner = new Scanner(System.in);

        System.out.println("请输入数组大小: ");

        int n = scanner.nextInt();

        System.out.println("请输入数组: ");

        int [] a = new int[n];
        for (int i = 0; i < n; i++) {
            a[i] = scanner.nextInt();
        }

        System.out.println("请输入权重: ");

        double [] w = new double[n];
        for (int i = 0; i < n; i++) {
            w[i] = scanner.nextDouble();
        }
        int result = select(a, w, 0, n-1, 0.5);

        System.out.println("带权中位数是:" + result);


        //生成随机数以测试代码

//      Random r = new Random();

//      System.out.println("输入数组大小n: ");

//      Scanner scanner = new Scanner(System.in);
//      int n = scanner.nextInt();
//      int a[] = new int[n];
//      double w[] = new double[n];
//      for (int i = 0; i < n; i++) {
//          a[i] = r.nextInt();
//          w[i] = r.nextDouble();
//      }
//      double sum = 0;
//      for (int i = 0; i < n; i++)
//          sum += w[i];
//      for (int i = 0; i < n; i++)
//          w[i] /= sum;
//

//      System.out.println("生成的随机数组和权值分别是: ");
```

```
//     for (int i = 0; i < n; i++) {
//        System.out.print(a[i]);
//        System.out.print(' ');
//     }
//     System.out.println(' ');
//     for (int i = 0; i < n; i++) {
//        System.out.print(w[i]);
//        System.out.print(' ');
//     }
//
//     int result = select(a, w, 0, n-1, 0.5);
//     System.out.println("\n带权中位数是:" + result);
}

   public static int select(int a[], double w[], int p, int r,
double k) {
      /**
       * 实验一：带权中位数
       * 输入：数组a[p:r]，以及其权值w[p:r].包括边界。k是权值划分
边界，在本题中k=0.5
       */
      if (r-p < 5) {
         bubbleSort(a, w, p, r);  //将a带着w一起冒泡排序
         double weight = 0.0;
         for (int i = p; i <= r; i++) {
            weight += w[i];

            if (weight > k)
               return a[i];
         }
      }

      for (int i = 0; i <= (r-p-4)/5; i++) {
         int s = p + 5 * i;
         int t = s + 4;
         bubbleSort(a, w, s, t);
         swap(a, w, p+i, s+2);

//        double sumWeights = 0;
```

```
//        for (int index = s; index <= t; index++)
//           sumWeights += w[index];
//        double weights = 0;
//        for (int index = s; index <= t; index++) {
//           weights += w[index];
//           if (weights > sumWeights / 2)
//              swap(a, w, p+i, index);
//        }
      }

      bubbleSort(a, w, p, p+(r-p-4)/5);
      int x = a[p+(r-p+6)/10];
      int i = partition(a, w, p, r, x); // 以x为pivot划分数组
a[p:r]，并且返回划分完成后x的下标
      double weights = 0;
      for (int index = p; index <= i; index++) {
         weights += w[index];

      }
      if (weights > k)
         return select(a, w, p, i, k);
      else
         return select(a, w, i+1, r, k-weights);
   }

   public static void bubbleSort(int a[], double w[], int p,
int r) {
      /**
       * 冒泡排序

       * 输入：数组a，排序首地址p，排序尾地址r

       * 效果：将a[p:r]之间的元素(包括两个边界)按升序排好
       */
      for (int i = p; i <= r; i++) {
         for (int j = r; j > i; j--) {
            if (a[j] < a[j-1])
               swap(a, w, j, j-1);
         }
      }
   }
```

```
    public static void swap(int a[], double w[], int i, int j)
{
        /**
         * 将a[i]与a[j]交换
         */
        int temp = a[i];
        a[i] = a[j];
        a[j] =temp;

        double temp2 = w[i];
        w[i] = w[j];
        w[j] = temp2;
    }


    public static int partition(int a[], double w[], int r, int
p, int x) {
        int left = r, right = p;
        while (left < right) {
            if (a[left] < x)
                left++;
            if (a[right] > x)
                right--;
            swap(a, w, left, right);
        }
        return left;
    }
}
```

1.5 测试结果

测试点一：正确答案：591.

```
请输入数组大小:
10
请输入数组:
256 760 591 761 621 446 774 555 342 67
请输入权重:
0.1124 0.0250 0.1149 0.1637 0.1073 0.0689 0.1819 0.1218 0.0646 0.0394
带权中位数是:591
```

程序运行正确

测试点二：随机生成测试数据

```
输入数组大小n:
20
生成的随机数组和权值分别是:
-275290598 -1549867294 -420857853 -109248054 -59889593 1643365975
1868612198 -1794809413 -2134120633 -1773750043 -2020051294 -129433642 -
779483109 -1162285790 1554013829 -1785991055 1284287495 -1343857651
601409563 -1737117466
0.09656773144589867 0.046950984199436954 0.07856479691969345
0.061517083696472616 0.09657596561987604 0.05006905851403734
0.019385880956985953 0.06949921230701882 0.012433377060254656
0.011400760770300828 0.015413426947396303 0.01608756351168227
0.06389094485653934 0.009702421455657029 0.09095470236769912
0.04405999293998829 0.0784899415262453 0.08014506119093096
0.04432717466324082 0.013963919050644962
带权中位数是:-275290598
```

实验二 动态规划

2.1 实验内容

设有一个长度为 L 的钢条，在钢条上标有 n 个位置点（p1,p2,…,pn）。现在需要按钢条上标注的位置将钢条切割为 n+1 段，假定每次切割所需要的代价与所切割的钢条长度成正比。请编写一个算法，能够确定一个切割方案，使切割的总代价最小。方便起见，我们令每次的切割代价等于被切割钢条的长度。

2.2 实验思路

最优子结构性质：设对于长度为 L 的钢条，最优切割顺序是(k1, k2,…,kn). 其中第一个划分点 k1 将长度为 L 的钢条划分为长度为 L1 和长度为 L2 的两个子钢条。则(k2,…,kn)是将长度为 L1 的钢条和长度为 L2 的钢条最优切割的解。采用反证法证明如下：

长度为 L 的钢条的最小切割代价计算公式是：

$$cost(L)=cost(L_1)+cost(L_2)+L$$

若(k2,…,kn)不是将 L1 和 L2 最优分割的解，即存在另一组解(j2,j3,…jn)能够使 L 最优化分(使用这组解将钢条 L 划分为 L1',L2'，且划分代价是 cost(L1')+cost(L2'))，而且满足：

$$cost(L_1')+cost(L_2')>cost(L_1)+cost(L_2)$$

则显然用(j2,j3,…,jn)划分产生的最终代价 cost(L')有：

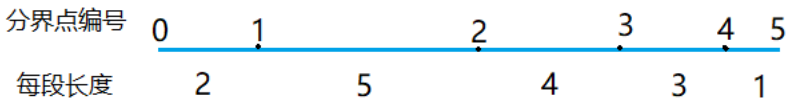
$$cost(L')=cost(L_1')+cost(L_2')+L>cost(L)$$

这说明(k1,j2,j3,…,jn)不是最优解，与假设矛盾，最优子结构性质得证。

动态规划算法设计：具有最优子结构性质的问题可以用动态规划(Dynamic Programming)算法进行求解。算法的具体设计如下。

状态表示：用一个二维数组 dp[n][n]表示问题求解过程中的所有状态。dp[i][j]

表示从分界点 i 到分界点 j 之间这段钢条的最小划分代价(i<j)。



状态转移方程：

$$dp[i][j]=\min\{dp[i][k]+dp[k][j]\}+sum(a[i:j-1]),\ \ for\ k=i+1\sim j-1$$

假如现在要求 dp[i][j]，也就是从 i 到 j 之间的这段钢条的最小划分代价。这部分代价可分为两部分：第一部分为可变部分，是本次划分结束后产生的两个子钢条的划分最小代价之和，随着划分点 k 的不同而变化；第二部分是固定部分，是切割 i~j 这段钢条时所产生的代价，等于钢条 i~j 的长度。

计算动态规划表时，一定要确保在 dp[i][j]开始计算时，计算所需要的表格单元值已经在之前的步骤中被计算了出来(也就是 dp[i][k], dp[k][j] for k = i+1~j-1). 这就需要我们为之设计计算 dp 表的迭代步骤：引入变量 gap = j – i，也就是钢条的分段数。显然，分段数少的钢条划分代价需要优先计算，这样就可以确保每次计算 dp[i][j]时所需要的值都已经求出。构造两层循环以计算 dp 表格：外层依据 gap 升序循环；内层依据钢条左分界点升序循环。

边界条件：动态规划表格的边界情况是子钢条只剩下唯一一个划分点。因此边界条件是：

$$dp[i][i+2]=a[i][i+1],\ \ for\ i=0\sim n-1$$

复杂度分析：第一层 gap 循环需要迭代大约 n 次，第二层中，左分界点层循环需要迭代 n-gap 次，第三层计算最优分界点需要迭代 gap 次。时间复杂度为：

$$f(n)=O(\sum_{gap=3}^{n}gap(n-gap))=O(n^3)$$

算法所用空间主要是 dp[n][n]表格。空间复杂度为$O(n^2)$

划分点顺序记录方法：维护一个二维数组 point[i][j]，表示 i~j 这段钢条的最优化分点 k。

2.3 算法伪代码

注：如果输入格式改为第一行输入钢条长度 L 和分割点个数 n，第二行输入每个分割点的坐标，只需要把坐标先冒泡排序方法按升序排序，再计算出 a[]即可。整个算法的时间复杂度仍然是 $O(n^3)$

Input：划分点个数n, 钢条每段长度数组$a[n]$

*Step*1：填写动态规划表格的边界：$dp[i][i+2]=a[i][i+1],\ \ for\ i=0\sim n-1;$

*Step*2：迭代计算动态规划表格：

$for\ gap=3\sim n+1:$

$for\ start=0\sim n+1-gap:$

$dp[start][start+gap]=\min\{dp[start][k]+dp[k][start+gap]\}+$

$sum(a[i:j-1]),\ \ for\ k=start+1\sim start+gap-1$

在计算dp的同时更新point[i][j]的值

Output：$dp[0][n+1]$

2.4 算法源代码

```
package algorithmHomework;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;
import java.util.Scanner;
/**
 * 动态规划算法
 * @author Yulong Yang
 *
 */
public class Task2 {
   static int inf = Integer.MAX_VALUE;
   public static void main(String[] args) {
      //创建源
      File src = new File("C:\\Users\\lenovo\\eclipse-
workspace\\algorithmHomework\\src\\algorithmHomework\\exp2_in.
txt");
      Scanner scanner = null;
      try {
         scanner = new Scanner(src);
      } catch (FileNotFoundException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
      while (scanner.hasNext()) {
         int L = scanner.nextInt();
```

```
            int n = scanner.nextInt();

            int []a = new int[n+1];
            int []location = new int[n];

            for (int i = 0; i < n; i++)
               location[i] = scanner.nextInt();
            bubbleSort(location, 0, n-1);

            a[0] = location[0];
            for (int i = 1; i < n; i++)
               a[i] = location[i] - location[i-1];
            a[n] = L - location[n-1];



            //动态规划部分
            int [][]dp = new int[n+2][n+2];
            int [][]point = new int[n+2][n+2];
            for (int i = 0; i <= n-1; i++) {
               dp[i][i+2] = a[i] + a[i+1];
               point[i][i+2] = i+1;
            }
            for (int gap = 3; gap <= n+1; gap++) {
               for (int start = 0; start+gap <= n+1; start++) {
                  int new_cost = 0;
                  for (int i = start; i <= start+gap-1; i++)
                     new_cost += a[i];
                  int min_cost = inf;
                  for (int k = start+1; k <= start+gap-1; k++) {
                     int temp = dp[start][k] + dp[k][start+gap];
                     if (temp < min_cost)
                        point[start][start+gap] = k;
                     min_cost = temp < min_cost ? temp :
min_cost;
                  }
                  dp[start][start+gap] = min_cost + new_cost;
               }
            }
            System.out.println(dp[0][n+1]);

      }
//     //输入输出部分
```

```
//     Scanner scanner = new Scanner(System.in);
//     System.out.println("请输入钢条长度:");
//     int L = scanner.nextInt();
//     System.out.println("请输入切割点个数:");
//     int n = scanner.nextInt();
//     int []a = new int[n+1];
//     int []location = new int[n];
//     System.out.println("位置点坐标(共"+"n个数字):");
//     for (int i = 0; i < n; i++) {
//        location[i] = scanner.nextInt();
//     }
//     a[0] = location[0];
//     for (int i = 1; i < n; i++)
//        a[i] = location[i] - location[i-1];
//     a[n] = L - location[n-1];
//
//     //动态规划部分
//     int [][]dp = new int[n+2][n+2];
//     int [][]point = new int[n+2][n+2];
//     for (int i = 0; i <= n-1; i++) {
//        dp[i][i+2] = a[i] + a[i+1];
//        point[i][i+2] = i+1;
//     }
//     for (int gap = 3; gap <= n+1; gap++) {
//        for (int start = 0; start+gap <= n+1; start++) {
//           int new_cost = 0;
//           for (int i = start; i <= start+gap-1; i++)
//              new_cost += a[i];
//           int min_cost = inf;
//           for (int k = start+1; k <= start+gap-1; k++) {
//              int temp = dp[start][k] + dp[k][start+gap];
//              if (temp < min_cost)
//                 point[start][start+gap] = k;
//              min_cost = temp < min_cost ? temp : min_cost;
//           }
//           dp[start][start+gap] = min_cost + new_cost;
//        }
//     }
//     System.out.println("最小切割代价是:"+dp[0][n+1]);
```

```
//     System.out.println("动态规划表如下: ");
//     for (int i = 0; i < n+2; i++) {
//        for (int j = 0; j < n+2; j++) {
//           System.out.print(dp[i][j]);
//           System.out.print('\t');
//        }
//        System.out.print('\n');
//     }
//     System.out.println("最优切割顺序是:");
//     printPath(point, 0, n+1);
   }
   public static void printPath(int point[][], int start, int
end) {
      //递归输出最优解
      if (end-start <= 1)
         return;
      int k = point[start][end];
      System.out.print(k);
      System.out.print(' ');
      printPath(point, start, k);
      printPath(point, k, end);
   }
   public static void bubbleSort(int a[], int p, int r) {
      /**
       * 冒泡排序
       * 输入: 数组a, 排序首地址p, 排序尾地址r
       * 效果: 将a[p:r]之间的元素(包括两个边界)按升序排好
       */
      for (int i = p; i <= r; i++) {
         for (int j = r; j > i; j--) {
            if (a[j] < a[j-1])
               swap(a, j, j-1);
         }
      }
   }
   public static void swap(int a[], int i, int j) {
      /**
       * 将a[i]与a[j]交换
```

```
        */
        int temp = a[i];
        a[i] = a[j];
        a[j] =temp;
    }
}
```

2.5 测试结果

测试点一：单个样例测试

```
请输入钢条长度:
7
请输入切割点个数:
4
位置点坐标(共n个数字):
1 3 4 5
最小切割代价是:16
动态规划表如下:
0   0   3   7   10  16
0   0   0   3   6   12
0   0   0   0   2   6
0   0   0   0   0   3
0   0   0   0   0   0
0   0   0   0   0   0
最优切割顺序是:
    2 1 4 3
```

测试点二：批量数据测试

用实验给定的测试数据 exp2_in.txt 测试结果如下：

```
2
14
20
421
474
7232
5240
67145
724145
626176
```

结果与标准答案 exp2_out.txt 完全一致。

实验三 贪心算法

3.1 实验内容

有 n 个作业和 m 台相同设备，每个作业 i 可选择一台设备进行加工，加

工时间为 ti。每台机器一次只能加工一个作业。编写算法，实现对作业的调度，使得 n 个作业的等待时间和最小。

3.2 实验思路

最优子结构：以只有一台机器的情况为例。假设某个安排($a_1, a_2,\ldots a_n$)是使 n 个作业等待时间最短的安排，对应的完成时间是($t_1', t_2', \ldots, t_n'$). 则安排($a_1, a_2, \ldots, a_{n-1}$)必然也是使前 n-1 个作业等待时间最短的安排, 对应的完成时间是($t_1', t_2', \ldots, t_{n-1}'$). 则它们的等待时间分别是：

$$T(a_1,...,a_n) = t_1'(n-1) + t_2'(n-2) + ... + t_{n-1}'$$
$$T(a_1,...,a_{n-1}) = t_1'(n-2) + t_2'(n-3) + ... + t_{n-2}'$$

从上式可以看出，若$T(a_1,...,a_{n-1})$不是最优值，则$t_1'(n-1) + t_2'(n-2) + ... + t_{n-2}'$必然也不是最优值，从而$T(a_1,...,a_n)$也不是最优值。所以该问题具有最优子结构性质。

贪心策略：将 n 个作业按照耗时升序排列($t_1' < t_2' < \ldots < t_n'$)，依次用机器加工。一旦某机器空闲出来，就安排下一个作业在该机器上加工，直到全部作业完成为止。

贪心选择性质：在一台机器的情况下，本问题具有贪心选择性质。按照贪心选择性质所求得的最短等待时间为：

$$T(a_1,...,a_n) = t_1'(n-1) + t_2'(n-2) + ... + t_{n-1}'$$

若交换作业 a_i, a_j 的位置，使 $t_i > t_j$ ($i < j$) 则根据以下不等关系：（顺序和大于逆序和）

$$(n-i)t_i' + (n-j)t_j' < (n-i)t_j' + (n-j)t_i'$$

可知$T(a_1,...,a_i,...,a_j,...,a_n) < T(a_1,...,a_j,...,a_i,...,a_n)$因此一台机器时，该问题有贪心选择性质。

然而对于多台机器，贪心选择性质不一定成立。比如对于两台机器，8 个作业的消耗时间分别为：1，2，3，4，5，6，7，100. 则按照上述贪心策略，机器 A 上安排的作业消耗时间是：1，2，5，6，7；机器 B 上安排的作业消耗时间是：2，4，100. 交换消耗时间为 3 的作业和消耗时间为 4 的作业，将会得到更好的解。综上所述，以上贪心策略只能求的近似解。

3.3 算法伪代码

Input: 机器数量m, 样本数量n, 耗时$a[n]$;

Initialize: $waitTime = 0$(总的等待时间);

$Time\_\mathrm{Re}\min ds[i] = 0,\ for\ i = 0 \sim m-1$(每台机器的剩余时间列表)

*Step*1: *if* $m > n$: $waitTime = 0$; *goto Output*;

else: *goto Step*2

*Step*2: *quitSort*(a)

*Step*3: *for* $i = 0 \sim n-1$:

$waitTime+ = \min(Time\,\mathrm{Re}\min ds)$;

TimeReminds[i]-=min(TimeReminds); *for* $i = 0 \sim m-1$;

TimeReminds.appand(a[i])

Output: *waitTime*

注：为了降低寻找最小值的时间开销，我们为 TimeReminds 数组建立一个最小堆，每次用堆排序找出最小值。

复杂度分析：最小堆建立：$O(m\log m)$，耗时数组 a[n]快速排序：$O(n\log n)$，循环过程的开销：$O(n\log m)$。总的复杂度如下：

$$f(n) = O(n\log n + n + m\log m + n\log m) = O(n\log n)$$

3.4 算法源代码

```
// AlgorithmHomework.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。
//

#include <iostream>
#include <algorithm>
using namespace std;

class MinHeap
{
private:
    int length;
    int* a;
public:
    MinHeap(int *a, int length)
    {
        this->length = length;
        this->a = a;
        for (int i = this->length / 2; i >=0; i--)
            this->Reset(i);
```

```
    }
    void Reset(int i)
    {
        if (2*i+1 < this->length)
        {
            if (2*i+2 < this->length)
            {

                if (this->a[i] > this->a[2*i+1] || this->a[i] > this->a[2*i+2])
                {
                    if (this->a[2*i+1] > this->a[2*i+2])
                    {
                        int temp = this->a[i];
                        this->a[i] = this->a[2*i+2];
                        this->a[2*i+2] = temp;
                        this->Reset(2*i+2);
                    }
                    else
                    {
                        int temp = this->a[i];
                        this->a[i] = this->a[2*i+1];
                        this->a[2*i+1] = temp;
                        this->Reset(2*i+1);
                    }

                }
            }

            else if (this->a[i] > this->a[2*i+1])
            {
                int temp = this->a[i];
                this->a[i] = this->a[2*i+1];
                this->a[2*i+1] = temp;
            }
        }
    }

    void Display()
    {
        for (int i = 0; i < this->length; i++)
            cout << a[i] << " ";
        cout << endl;
    }
};
```

```
class JobNode
{
public:
    int id;   //作业编号
    int time;   //完成作业需要的时间
    JobNode()    //默认构造函数
    {

    }
    JobNode(int id, int time) //构造函数
    {
        this->id = id;
        this->time = time;
    }
};

class MachineNode
{
public:
    int id;    //机器编号
    int sumTime;   //已经安排在该机器上的作业总时间
    int numJob;   //已经安排在该机器上的作业数量
    int* jobID;   //已经安排在该机器上的作业编号
    MachineNode()
    {
        this->jobID = new int[100000];
    }
    MachineNode(int id, int sumTime = 0)
    {
        this->id = id;
        this->sumTime = sumTime;
        this->jobID = new int[100000];
    }

    void AddJob(JobNode job)   //将作业job安排在该机器上
    {
        this->jobID[this->numJob] = job.id;
        this->sumTime += job.time;
        this->numJob++;
    }

    void Display()
    {
```

```
        cout << "第" << this->id << "号机器上安排的作业如下：";
        for (int i = 0; i < this->numJob; i++)
            cout << this->jobID[i] << ' ';
        cout << '\n';
    }
};

bool cmpJob(JobNode a, JobNode b)
{
    /* 按照消耗时间的升序排列 */
    return a.time < b.time;
}

bool cmpMachine(MachineNode a, MachineNode b)
{
    // 按照已安排时间的升序排序
    return a.sumTime < b.sumTime;
}

bool cmpFinal(MachineNode a, MachineNode b)
{
    // 按照单个机器完成所有作业所需时间降序排列
    return a.sumTime > b.sumTime;
}

JobNode job[100000];
MachineNode machine[10000];
int n = 0;
int m = 0;
int main()
{
    freopen("exp3_in.txt", "r", stdin);
    while ((cin >> n) && (cin >> m))
    {
        int waitTime = 0;  //等待时间
        int TimeReminds[10000];  //每台机器的剩余时间
        for (int i = 0; i < 10000; i++)
            TimeReminds[i] = 0;

        for (int i = 0; i < m; i++)
            machine[i].id = i + 1;
        for (int i = 0; i < n; i++)
        {
            job[i].id = i + 1;
```

```
            cin >> job[i].time;
        }

        if (m >= n)  //机器比任务还多
            waitTime = 0;
        else
        {
            sort(job, job + n, cmpJob);
            MinHeap MyHeap(TimeReminds, m);

            for (int i = 0; i < n; i++)
            {
    //          sort(machine, machine + m, cmpMachine);
    //          sort(TimeReminds, TimeReminds + m);
                MyHeap.Reset(0);
    //          cout<<"first目前每台机器的剩余时间是：";
    //          for (int i = 0; i < m; i++)
    //              cout << TimeReminds[i] << ' ';
    //          cout << endl;
                waitTime += TimeReminds[0] * (n - i);
                for (int i = m - 1; i >= 0; i--)
                    TimeReminds[i] -= TimeReminds[0];

    //          cout<<"second目前每台机器的剩余时间是：";
    //          for (int i = 0; i < m; i++)
    //              cout << TimeReminds[i] << ' ';
    //          cout << endl;

    //          machine[0].AddJob(job[i]);
                TimeReminds[0] = job[i].time;
    //          cout<<"目前每台机器的剩余时间是：";
    //          for (int i = 0; i < m; i++)
    //              cout << TimeReminds[i] << ' ';
    //          cout << endl;
            }

    //      for (int i = 0; i < m; i++)
    //          machine[i].Display();
        }
        cout << waitTime << endl;
    }
    return 0;
}
```

3.5 测试结果

测试一：用示例数据 exp3_in.txt

```
1517
1519
179
525
6728
404
809
0
743
2057
160540
39657
41
68850
1560
29192
11713
84708
47780
568981
1270199
630801
137543
262597
147081
137952
1317810
996
164721
36120
1186079
1688293
401964
830001
620903
238286
870374
333
2699
68035
781494
5639218
67103
```

```
3663368
245587
12888843
5077423
129432
4904099
8530072


--------------------------------
Process exited after 10.13 seconds with return value 0
请按任意键继续. . .
```

输出结果与正确答案完全一样。

实验六 搜索算法

6.1 实验内容

给定 1 个 1000 行×20 列的 0-1 矩阵，对于该矩阵的任意 1 列，其中值为 1 的元素的数量不超过 10%。设有两个非空集合 A 和 B，每个集合由矩阵的若干列组成。集合 A 和 B 互斥是指对于矩阵的任意一行，同时满足下列 2 个条件:1）若 A 中有一个或多个元素在这一行上的值是 1，则 B 中的元素在这一行全部是 0；2）若 B 中有一个或多个元素在这一行上的值是 1，则 A 中的元素在这一行全部是 0。请你设计一个算法，找出一对互斥集合 A 和 B，使得 A 和 B 包含的列的总数最大。

为保证输出结果唯一，每个集合的输出按升序排列，如果存在并列的情况,则依次采用以下策略：

1. A 和 B 元素个数差的绝对值最小
2. A 的个数要大于 B 的个数
3. A 的元素和要小于 B 的元素和

6.2 实验思路

设原集合为 S。两个列向量互斥当且仅当这两个列向量的内积为 0.集合 A, B 互斥当且仅当 $\forall x \in A, y \in B$，x 与 y 的内积为 0（正交）。本实验中使用的是深度优先搜索算法(DFS)。依次考虑编号为 0~19 的 20 个向量，每一个向量都有三种情况：加入 A，加入 B，两个都不加入。若某向量 a 得以加入 A，则 a 必然与 B 中的每一个向量都正交。(如果 B 是空集,则任意向量 a 都与之正交)。反之亦然。由于本题目不具有贪心选择性质，因此如果 a 虽然满足条件，但是放弃将 a 加入 A 或 B，反而有可能得到最优解。最坏情况下搜索的时间复杂度是 $O(3^n)$ (n 是列向量个数，本题 n=20). 如果考虑以下剪枝策略，则复杂度可以优化。

在每一步中，我们只需要考虑新加入的向量 a 与集合 A,B 的互斥关系即可。

1. 约束函数：如果向量 a 与 S 中的任意其他元素都互斥（在搜索之前可以提前判定），则 a 既可以加入 A，又可以加入 B；

2. 约束函数：如果向量 a 既与 A 互斥，又与 B 互斥，则 a 既可以加入 A，

又可以加入 B，还可以放弃 a；

3. 约束函数：如果向量 a 与 A 互斥但与 B 不互斥，则 a 可以加入 B 或者放弃 a；

4. 约束函数：如果向量 a 与 B 互斥但与 A 不互斥，则 a 可以加入 A 或者放弃 a；

5. 约束函数：如果向量 a 与 A 和 B 都不互斥，则 a 只能放弃；

6. 约束函数：如果全部向量都已经考虑完毕，A 或 B 仍然是空集，则该解不满足条件；

7. 限界函数：如果将剩下未考虑的全部向量都计算在内，当前划分方法仍然不足以更新已知最优解，则放弃继续搜索，直接回溯。

需要注意的是，本实验还存在无解的情况：没有任何一个向量能和其他向量正交，因此任何一个向量都不能加入 A 集合或 B 集合。这种情况只需要考察每一个向量与其他向量的正交性就可以在搜索之前提前排除。

6.3 算法伪代码

主程序伪代码如下：

Input：原始矩阵*data*[20][20]

*Step*1：构造*exclusive*[20][20]表示列向量之间的正交关系：

for each exclusive[i][j]：

exclusive[i][j] = 0表示向量i, j不互斥，反之互斥

*Step*2：判断无解情况：*exclusive*元素全为0则无解

*Step*3：构造*product*[20]用于记录与所有其他向量互斥的向量：

product[i] = 1表示i向量与其他向量互斥

*Step*4：深度优先搜索*search*($data, a, b, exclusive, product$)

Output：划分集合a, b

搜索函数 search 的伪代码如下：

Input：当前向量下标i；集合a, b；互斥矩阵*exclusive*，*product*

*Step*1(限界策略)：如果剩余元素个数已不足以更新最优值，结束该分支

*Step*2(终止条件)：*if* 所有向量均已考虑完毕：按照题目所给规则进行最优值更新

*Step*3(扩展子节点)：按照前述的剪枝规则扩展子节点：

例如，若向量a与A互斥但不与B互斥，则

search($i+1$, *temp_a*, *b*, *exclusive*, *product*)

search($i+1$, *a*, *b*, *exclusive*, *product*)

return

6.4 算法源代码

本题用 python 3 编写，所用编辑环境是 jupyter notebook

```
#!/usr/bin/env python
# coding: utf-8
```

```
# In[1]:


import numpy as np
import pandas as pd
import copy


# In[2]:


raw_data = np.loadtxt('exp6_in.txt', dtype=np.int32)
raw_data


# In[3]:


def isEclusive(exclusive,Set,i):
    for item in Set:
        if exclusive[i][item] == 0:
            return False
    return True


# In[4]:


def search(i, a, b, exclusive, product):
    '''
    data 表示原始矩阵, i 是当前所在的列下标, a 和 b 是两个集合，value 是当前两个集合的元素个数总和, exclusive 是互斥值矩阵
    '''

    global best_a, best_b
    if len(a) + len(b) + len(product) - i < len(best_a) + len(best_b):     # 限界策略
        return

    if i > len(data) - 1:
        if len(a) > 0 and len(b) > 0 and len(a)+len(b) > len(best_a)+len(best_b):
            best_a = copy.deepcopy(a)
            best_b = copy.deepcopy(b)
            return
```

```
        elif len(a) > 0 and len(b) > 0 and len(a)+len(b) == len(best_a)+len(best_b):
            if abs(len(a) - len(b)) < abs(len(best_a) - len(best_b)):
                best_a = copy.deepcopy(a)
                best_b = copy.deepcopy(b)
                return
            elif abs(len(a) - len(b)) == abs(len(best_a) - len(best_b)):
                if len(a) > len(best_a):
                    best_a = copy.deepcopy(a)
                    best_b = copy.deepcopy(b)
                    return
                elif len(a) == len(b):
                    if sum(a) < sum(best_a):
                        best_a = copy.deepcopy(a)
                        best_b = copy.deepcopy(b)
                        return
                    else:
                        return
                else:
                    return
            else:
                return
        else:
            return

    else:
        temp_b = copy.deepcopy(b)
        temp_a = copy.deepcopy(a)
        temp_b.append(i)
        temp_a.append(i)
        if product[i] == 1:     # 当前向量与其他向量全部正交
            search(i+1, temp_a, b, exclusive, product)
            search(i+1, a, temp_b, exclusive, product)
            search(i+1, a, b, exclusive, product)
            return

        if isEclusive(exclusive, a, i) and isEclusive(exclusive, b, i):
            search(i+1, temp_a, b, exclusive, product)
            search(i+1, a, temp_b, exclusive, product)
            search(i+1, a, b, exclusive, product)
            return

        elif isEclusive(exclusive, b, i):
            search(i+1, temp_a, b, exclusive, product)
            search(i+1, a, b, exclusive, product)
```

```
            return

        elif isEclusive(exclusive, a, i):
            search(i+1, a, temp_b, exclusive, product)
            search(i+1, a,b, exclusive,product)
            return
        else:
            search(i+1, a, b, exclusive,product)
            return


# In[5]:


col_size = 20   # 原矩阵列的长度
for rd in range(10):
    print('-------------------------\nrd = ', rd+1)   ############
    best_a = []
    best_b = []
    exclusive = np.zeros([col_size,col_size],dtype=np.int)    # 元素值为 1 互斥，0
表示不互斥 exclusive 应该是对称矩阵
    data = np.array(raw_data[1000*rd:1000*(rd+1)]).T.tolist()
    for i in range(col_size):
        for j in range(col_size):
            if np.inner(data[i],data[j]) > 0:
                exclusive[i][j] = 0
            else:
                exclusive[i][j] = 1
    if sum(sum(exclusive)) == 0:    # 判掉无解情况
        print(best_a)
        print(best_b)
        continue

    product = sum(exclusive)
    for i in range(len(product)):
        if product[i] >= col_size-1:
            product[i] = 1    # 表示与其他所有元素互斥
        else:
            product[i] = 0

    a = []    # a, b 存储列向量的下标
    b = []

    search(0,a,b,exclusive,product)
```

```
    print(best_a)
    print(best_b)
```

6.5 测试结果

使用实验提供的样例测试结果如下（注：以实验报告打包文件中的测试输入数据为准）。为了方便查看，我加入了一些分割线和标识文字。rd 代表测试轮数。紧接着两行列表分别代表 A 和 B。第十轮是两个空列表，代表无解。

```
------------------------
rd =  1
[0, 1, 2, 3, 4, 5, 7, 8, 9, 11]
[6, 10, 12, 13, 14, 15, 16, 17, 18, 19]
------------------------
rd =  2
[1, 2, 3, 4, 5, 6, 7, 10, 12, 13, 14, 16, 17, 18, 19]
[0, 8, 9, 11, 15]
------------------------
rd =  3
[1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 2]
------------------------
rd =  4
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 17, 18, 19]
[16]
------------------------
rd =  5
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[9]
------------------------
rd =  6
[0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 19]
[1, 9, 18]
------------------------
rd =  7
[0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 19]
[17]
------------------------
rd =  8
[0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19]
[8, 12]
------------------------
rd =  9
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 18, 19]
[14]
------------------------
```

```
rd =  10
[]
[]
```

实验七 选做题

7.1 实验内容

给定 1 个 1000 行×20 列的 0-1 矩阵，对于该矩阵的任意 1 列，其中值为 1 的元素的数量不超过 10%。设有两个非空集合 A 和 B，每个集合由矩阵的若干列组成。集合 A 和 B 互斥是指对于矩阵的任意一行，同时满足下列 2 个条件：1）若 A 中有一个或多个元素在这一行上的值是 1，则 B 中的元素在这一行全部是 0；2）若 B 中有一个或多个元素在这一行上的值是 1，则 A 中的元素在这一行全部是 0。请你设计一个算法，找出集合 A、B 和 C，满足：1）A、B、C 两两互斥，且 2）A、B 和 C 包含的列的总数最大。

7.2 实验思路

本题目的解法和第六题完全一样，只需要对搜索规则稍作修改即可。具体修改的地方如下。

1. 约束函数：如果当前向量与其他向量均正交，则它既可以放入 A，或者 B，或者 C；

2. 约束函数：如果当前向量与当前的集合 A,B,C 均正交，则它可以放入 A，或者 B，或者 C，也可以放弃放入任何集合；

3. 约束函数：如果当前向量与 A 和 B 正交，则它可以放入 C，也可以放弃放入任何集合，其他两种情况也类似；

4. 约束函数：如果当前向量与 A,B,C 均不正交，则只能被放弃；

5. 限界函数：如果将剩余结点全部加上都不足以更新当前最优值，则放弃继续搜索。

7.3 实验源代码

本实验使用的语言是 Python 3，编辑环境是 jupyter notebook。

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import pandas as pd
import copy


# In[2]:
```

```
raw_data = np.loadtxt('exp7_in.txt', dtype=np.int32)
raw_data


# In[3]:


def isEclusive(exclusive,Set,i):
    for item in Set:
        if exclusive[i][item] == 0:
            return False
    return True


# In[4]:


def search(data, i, a, b, c, exclusive, product):
    '''
    data 表示原始矩阵, i 是当前所在的列下标, a 和 b 是两个集合，value 是当前两个集合的元素个数总和, exclusive 是互斥值矩阵
    '''

    global best_a, best_b, best_c
    if len(a) + len(b) + len(c) + len(product) - i < len(best_a) + len(best_b) + len(best_c):    # 限界策略
        return

    if i > len(data) - 1:
        if len(a) > 0 and len(b) > 0 and len(c) > 0 and len(a)+len(b)+len(c) > len(best_a)+len(best_b)+len(best_c):
            best_a = copy.deepcopy(a)
            best_b = copy.deepcopy(b)
            best_c = copy.deepcopy(c)
            return
        elif len(a) > 0 and len(b) > 0 and len(c) > 0 and len(a)+len(b)+len(c) == len(best_a)+len(best_b)+len(best_c):
            if abs(len(a) - len(b)) + abs(len(b) - len(c)) < abs(len(best_a) - len(best_b)) + abs(len(best_b) - len(best_c)):
                best_a = copy.deepcopy(a)
                best_b = copy.deepcopy(b)
                best_c = copy.deepcopy(c)
                return
            elif abs(len(a) - len(b)) + abs(len(b) - len(c)) == abs(len(best_a) -
```

```
len(best_b)) + abs(len(best_b) - len(best_c)):
                    if sum(a) < sum(best_a):
                        best_a = copy.deepcopy(a)
                        best_b = copy.deepcopy(b)
                        best_c = copy.deepcopy(c)
                        return
                    elif sum(a) == sum(best_a):
                        if sum(b) < sum(best_b):
                            best_a = copy.deepcopy(a)
                            best_b = copy.deepcopy(b)
                            best_c = copy.deepcopy(c)
                            return
                        elif sum(b) == sum(best_b):
                            if sum(c) < sum(best_c):
                                best_a = copy.deepcopy(a)
                                best_b = copy.deepcopy(b)
                                best_c = copy.deepcopy(c)
                                return
                            else:
                                return
                        else:
                            return
                    else:
                        return
                else:
                    return
            else:
                return

        else:
            temp_b = copy.deepcopy(b)
            temp_a = copy.deepcopy(a)
            temp_c = copy.deepcopy(c)
            temp_b.append(i)
            temp_a.append(i)
            temp_c.append(i)
            if product[i] == 1:     # 当前向量与其他向量全部正交
                search(data, i+1, temp_a, b, c, exclusive, product)
                search(data, i+1, a, temp_b, c, exclusive, product)
                search(data, i+1, a, b, temp_c, exclusive, product)
                return

            if isEclusive(exclusive, a, i) and isEclusive(exclusive, b, i) and
isEclusive(exclusive, c, i):
```

```
                search(data, i+1, temp_a, b, c, exclusive, product)
                search(data, i+1, a, temp_b, c, exclusive, product)
                search(data, i+1, a, b, temp_c, exclusive, product)
                search(data, i+1, a, b, c, exclusive, product)
                return
        if isEclusive(exclusive, a, i) and isEclusive(exclusive, b, i):
                search(data, i+1, a, b, temp_c, exclusive, product)
                search(data, i+1, a, b, c, exclusive, product)
                return
        elif isEclusive(exclusive, a, i) and isEclusive(exclusive, c, i):
                search(data, i+1, a, temp_b, c, exclusive, product)
                search(data, i+1, a, b, c, exclusive, product)
                return
        elif isEclusive(exclusive, b, i) and isEclusive(exclusive, c, i):
                search(data, i+1, temp_a, b, c, exclusive, product)
                search(data, i+1, a, b, c, exclusive, product)
                return
        else:
                search(data, i+1, a, b, c, exclusive, product)
                return


# In[5]:


col_size = 20   # 原矩阵列的长度
for rd in range(10):
#         if rd != 3:
#                 continue
        print('-------------------------\nrd = ', rd+1)    ############
        best_a = []
        best_b = []
        best_c = []
        exclusive = np.zeros([col_size,col_size],dtype=np.int)     # 元素值为 1 互斥，0
表示不互斥  exclusive 应该是对称矩阵
        data = np.array(raw_data[1000*rd:1000*(rd+1)]).T.tolist()
        for i in range(col_size):
                for j in range(col_size):
                        if np.inner(data[i],data[j]) > 0:
                                exclusive[i][j] = 0
                        else:
                                exclusive[i][j] = 1
        if sum(sum(exclusive)) == 0:     # 判掉无解情况
                print(best_a)
```

```
            print(best_b)
            print(best_c)
            continue

        product = sum(exclusive)
        for i in range(len(product)):
            if product[i] >= col_size-1:
                product[i] = 1      # 表示与其他所有元素互斥
            else:
                product[i] = 0

        a = []      # a, b 存储列向量的下标
        b = []
        c = []

        search(data,0,a,b,c,exclusive,product)
        mutual = []
        for i in range(len(product)):    #####
            if product[i] == 1:
                mutual.append(i)
        print(best_a)
        print(best_b)
        print(best_c)


# In[ ]:
```

7.4 测试结果

（注：以实验报告打包文件中的测试输入数据为准）

```
-------------------------
rd =  1
[0, 1, 2, 4]
[13, 17, 18, 19]
[3, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16]
-------------------------
rd =  2
[8, 9, 14, 15]
[16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13]
-------------------------
```

```
rd =  3
[5, 8, 15]
[16, 17, 18, 19]
[0, 1, 2, 3, 4, 6, 7, 9, 10, 11, 12, 13, 14]
-------------------------
rd =  4
[15, 16]
[17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
-------------------------
rd =  5
[12, 15, 16]
[17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14]
-------------------------
rd =  6
[15, 16]
[17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
-------------------------
rd =  7
[15, 16]
[17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
-------------------------
rd =  8
[15, 16]
[17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
-------------------------
rd =  9
[5, 6, 15]
[16, 17, 18, 19]
[0, 1, 2, 3, 4, 7, 8, 9, 10, 11, 12, 13, 14]
-------------------------
rd =  10
[16, 17]
[18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

参考文献

[1] 王晓东 算法设计与分析（第四版）北京：清华大学出版社 2018