西安交通大學

算法设计与分析实验报告

一、 分治与递归

（1）问题描述：

设有 n 个互不相同的元素 x1,x2,…, xn，每个元素 xi 带有一个权值 wi，且$\sum_{i=1}^{n} w_i = 1$。若元素 xk 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 xk 为 x1,x2,…, xn 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。

（2）问题分析：

该问题与线性时间选择问题类似，要求在线性时间内选择出一个符合要求的数，排序顺序均按元素的大小顺序排序。最坏情况线性时间内解决该问题，意味不能进行整体排序，可以将 Select 算法的元素大小位次为依据选择，改进成以权重划分为依据，用改进的 Select 算法解决此问题。

（3）算法设计：

设带权k位数满足$\sum_{x_i<x_k} w_i \le k$ 且$\sum_{x_i>x_k} w_i \le k$。

与主教材类似，将n个输入元素划分成n/5个组，每组5个元素。用简单排序算法，将每组中的元素排好序，并取出每组的中位数，共n/5个。

select(aa a[], int p, int r, double k,int t)提供第t大数或带权k位数两种方式的select。递归调用select来找出这n/5个中位数的中位数，以这个元素作为划分基准，进行一次partion。之后判断枢纽的左边之和sum是否大于k（初始为0.5），若大于，则在右边寻找带权sum-k位数，否则在左边寻找带权k位数。

初始条件为select(a, 0, n - 1, 0.5,-1);

容易知道，在最坏情况下时间复杂度为$O(n)$)

（4）算法实现：

```
#define NN 100
#include<iostream>
using namespace std;
```

```
struct aa {
	float x, w;
};
bool operator == (const aa& A, const aa& B) {
	return A.x == B.x;
}
bool operator != (const aa& A, const aa& B) {
	return A.x != B.x;
}
bool operator < (const aa& A, const aa& B) {
	return A.x < B.x;
}
bool operator > (const aa& A, const aa& B) {
	return A.x > B.x;
}
void swap(aa* a, aa* b) {
	aa temp = *a;
	*b = *a;
	*a = temp;
	}

		void bubbleSort(aa a[], int p, int r)
{
	for (int i = p; i < r; i++)
	{
		for (int j = i + 1; j <= r; j++)
		{
			if (a[j] < a[i])
				swap(a[i], a[j]);
		}
	}
}

void bubble(aa a[], int p, int r) {
	if (r <= p)		return;
	for (int i = p; i < r; i++) {
		if (a[i] > a[r])
			swap(a[i], a[r]);
	}
}


int partition(aa a[], int i, int j, int k) {
	swap(a[i], a[k]);
```

```
	while (i != j) {
		while (a[i] < a[j])	j--;
		if (i != j) {
			swap(a[i], a[j]);
			i++;
		}
		while (a[i] < a[j])	i++;
		if (i != j) {
			swap(a[i], a[j]);
			j--;
		}
	}
	return i;
}

int select(aa a[], int p, int r, double k,int t) {
	double sum = 0;
	if (r - p <= 5) {
		bubbleSort(a, p, r);
		int i = p;
		while (sum + a[i].w <= k && i < r) {
			sum += a[i].w;
			i++;
		}
		return i;
	}

	for (int i = 0; i <= (r - p - 4) / 5; i++) {
		int s = p + 5 * i, n = s + 4;
		for (int j = 0; j < 3; j++) bubble(a, s, n - j);
		swap(a[p+i], a[s + 2]);
	}

	int pr = select(a, p, p + (r - p - 4) / 5,-1 ,(r - p + 6) / 10); //求解中位数
	pr = partition(a, p, r, pr);
	if (k != -1) {		//求解带权中位数
		sum = 0;
		for (int i = p; i < pr; i++)	sum += a[i].w;
		if (sum < k)
			return select(a, pr, r, k - sum, -1);
		else
				return select(a, p, pr, k, -1);
	}
	else {				//求解中位数
```

```
            if (k <= pr)return select(a, p, pr,-1 ,t);
            else return select(a, pr+1 + 1, r, -1,t -(pr - p + 1));
        }
}

int main()
{
        int  n;
        aa a[NN];
        cin >> n;
        for (int i = 0; i < n; i++) {
            cin >> a[i].x;
        }
        for (int i = 0; i < n; i++) {
            cin >> a[i].w;
        }
        n = select(a, 0, n - 1, 0.5,-1);
        cout << a[n].x;
     }
```

（5）运行结果：

输入：10

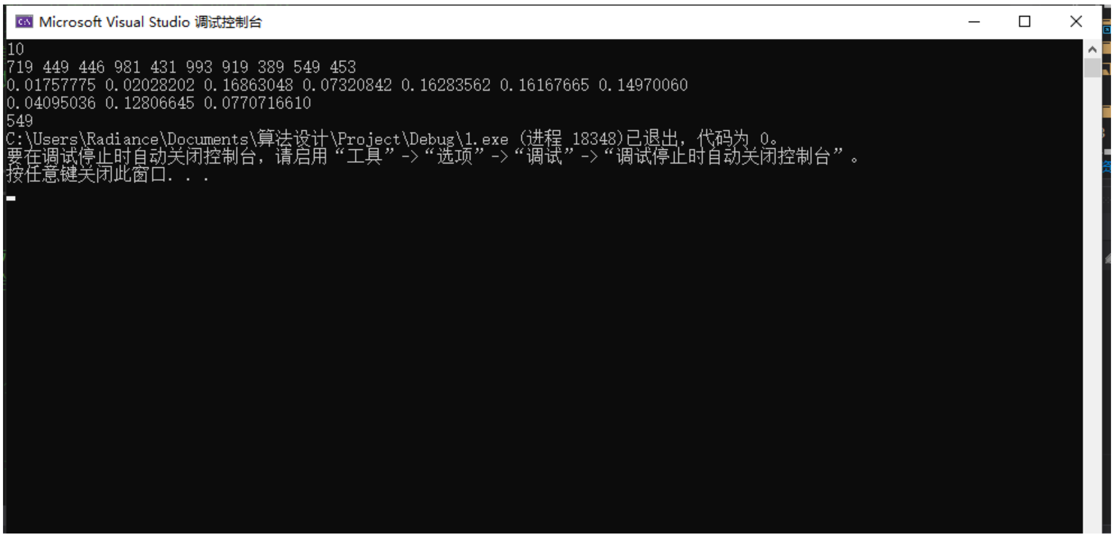
719 449 446 981 431 993 919 389 549 453

0.01757775　0.02028202　0.16863048　0.07320842　0.16283562

0.16167665 0.14970060

0.04095036 0.12806645 0.0770716610

输出：549



二、 动态规划

（1）实验要求：

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。

（2）问题分析：

在考虑某子段的切割方法时，会基于该子段的子段的最优切割进行，而两个不同段的切割可能包含相同子段切割问题，故该问题有最优子结构与重叠子问题，利用动态规划进行解决。

（3）算法设计：

将钢条的长度和标记位置保存在一位数组$x[\,]$中，其中$x[0]=0$，$x[n+1]=L$。将切割的最小代价结果保存在$dp[\,]$中，其中$dp[i][j]$是 i 到 j 的最小代价。通过递归的程序设计实现自底向上的计算每段切割的最小代价。

该算法的状态转移方程式：

$$dp[i][j]=\begin{cases}0, i+1=j\\ min\left(dp[i][k]+dp[k][j]\right)+x[j]-x[i], i+1<j\end{cases}$$

int solve(int i, int j)是解从第 i 个位置到第 j 个位置的最小代价：

若 dp[i][j]已经存在，则直接调用；若不存在则按状态转移方程式计算最优值。

先令 x[0] = 0;x[n + 1] = L;初始条件为 solve(0, n + 1);

容易知道，该算法时间复杂度为$O\left(n^3\right)$。通过四边形不等式优化，算法时间复杂度可能可以降为$O\left(n^2\right)$

（4）算法实现：

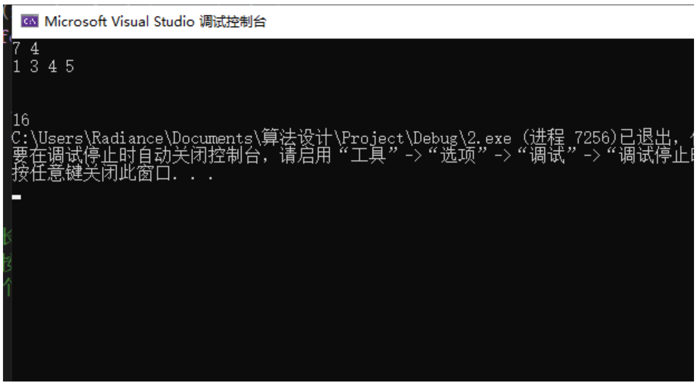
```
#include<iostream>
#define NN 100
using namespace std;
int L, n, x[NN], dp[NN][NN];

int solve(int i, int j) {
	int best;
	if (j == i + 1) return 0;
	if (dp[i][j] > 0) return dp[i][j];
	dp[i][j] =  solve(i, i + 1) + solve(i + 1, j)+ x[j] - x[i];
	for (int k = i + 1; k < j ; k++) {
		best =  solve(i, k) + solve(k, j)+ x[j] - x[i];
		if (best < dp[i][j])
			dp[i][j] = best;
	}
	return dp[i][j];
}

int main() {
	cin >> L>>n ;
	x[0] = 0;
	x[n + 1] = L;
	for (int i = 1; i < n+1; i++) {
		cin >> x[i];
	}

	for (int j = 0; j <= n + 1; j++)
		for (int i = 0; i <= n + 1; i++)
			dp[i][j] = 0;

	cout <<  " \n\n";
	cout << solve(0, n + 1);
}
```

（5）运行结果：

输入：7 4

1 3 4 5

输出：16



三、 贪心法

（1）实验要求：

设 T 为一带权树，树中的每个边的权都为整数。又设 S 为 T 的一个顶点的子集，从 T 中删除 S 中的所有结点，则得到一个森林，记为 T/S。如果 T/S 中所有树从根到叶子节点的路径长度都不超过 d，则称 T/S 是一

个 d 森林。设计一个算法求 T 的最小顶点集合 S，使 T/S 为一个 d 森林。

（2）程序思想：

该问题具有最优子结构性质和贪心选择性质，因此可以利用贪心算法进行解决，贪心算法时间复杂度较低。

（3）算法设计：

输入时，若出度为零的点标记为叶子结点，存入 leaf[]中，将出度存入 deg[]中，同时将每个节点的亲节点及长度分别存入 par[]和 plen[]中。

从叶子结点开始向上计算，如果正在计算的是根节点，则跳过该次计算。若父结点已经被删除则忽略；若父结点未被删除，则对父节点进行判断：如果当前节点步长加上到父节点的距离超过了父结点最大步长，若不超过 d 则更新父结点最大步长，否则删除父节点。接着令父节点的出度减去一，如果出度变为 0，则将其添加至新的叶子结点。反复计算

直至不再产生叶子结点。

该算法将遍历每个节点一次，因此时间复杂度为$O(n)$

（4）算法实现：

```
#include "iostream"
#include "fstream"
#include "queue"
#define NN 5000

using namespace std;
int num;

int par[NN];
int leaf[NN];
int deg[NN];
int plen[NN];
int del[NN];
int dist[NN];

int main()
{
  int nn;
  ifstream fin("exp3_in.txt");
  fin >> nn;

  for (int k = 0; k < nn; k++) {
    num = 0;
    memset(del, 0, sizeof(del));
    memset(dist, 0, sizeof(dist));
    memset(leaf, 0, sizeof(leaf));
    memset(par, 0, sizeof(par));
    memset(deg, 0, sizeof(deg));
    memset(plen, 0, sizeof(plen));
    int n, d;
    fin >> n >> d;
```

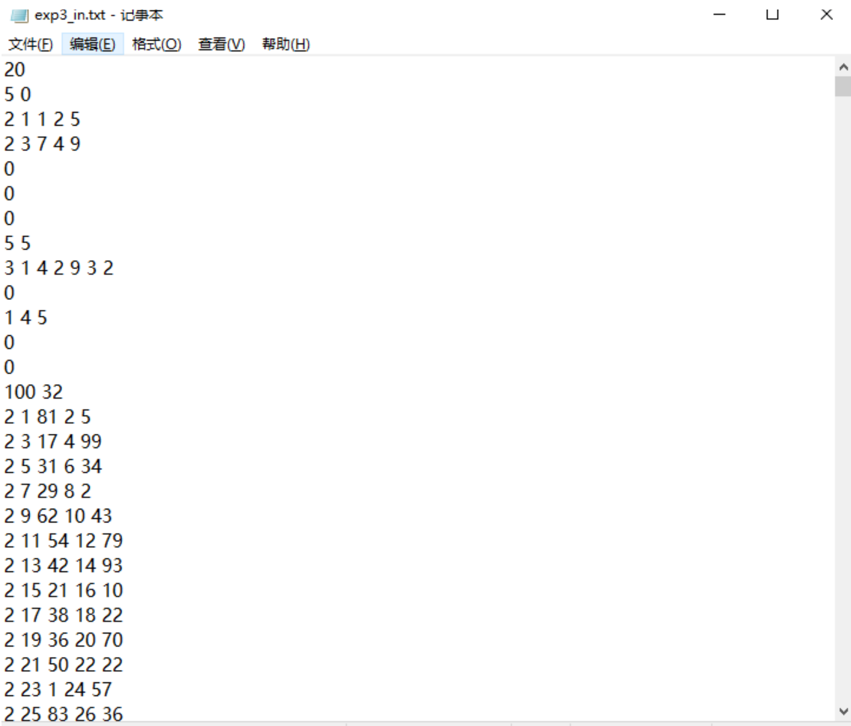
```
    int node, len, sum = 0;;
    for (int i = 0; i < n; i++)
    {
        fin >> deg[i];
        if (deg[i] == 0) {
            num++;
            leaf[num - 1] = i;
        }
        else
            for (int j = 0; j < deg[i]; j++)
            {
                fin >> node >> len;
                par[node] = i;
                plen[node] = len;
            }
    }

    for (int i = 0; i < num; i++)
    {
        if (leaf[i] != 0)
        {
            int parent = par[leaf[i]];
            int len = plen[leaf[i]];
            if (del[parent] == 0) {
                if (dist[leaf[i]] + len > dist[parent]) {
                    if (dist[leaf[i]] + len > d)
                    {
                        del[parent] = 1;
                        parent = par[parent];
                        sum++;
                    }
                    else
                        dist[parent] = len + dist[leaf[i]];
                }
            }
            deg[parent]--;
            if (deg[parent] == 0) {
                num++;
                leaf[num - 1] = parent;
            }
        }
    }
    cout << sum << endl;
}
```

```
    fin.close();
    return 0;
}
```

（5）运行结果：

输入：共 20000 行数据如下图

输出：共 20 组数据如下图




```
20
5 0
2 1 1 2 5
2 3 7 4 9
0
0
0
5 5
3 1 4 2 9 3 2
0
1 4 5
0
0
100 32
2 1 81 2 5
2 3 17 4 99
2 5 31 6 34
2 7 29 8 2
2 9 62 10 43
2 11 54 12 79
2 13 42 14 93
2 15 21 16 10
2 17 38 18 22
2 19 36 20 70
2 21 50 22 22
2 23 1 24 57
2 25 83 26 36
```

四、 回溯/分支界限法

（1）实验要求：

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。

（2）问题分析：

采用回溯法， 每列可以加入 A，加入 B，或丢弃，可以分为加入三个子树中。当不满足界限函数或约束函数时，回溯原先结点。当到达叶节点时，判断是否储存最佳状态。

每一次加入列时都计算列与矩阵是否冲突，时间复杂度太高。因此可提前计算每两列间是否冲突并储存，当新加入列时可与矩阵所含列依次比较，大幅度缩小了时间复杂度。

（3）算法设计：

输入时，将第 i 列的非零元素位置储存在 vector[i]中。所有列输入完毕后，利用二重循环，判断所有的列之间是否有冲突，存入 conf[i][j]中（其中 conf[i][j]=1 表示第 i 列和第 j 列冲突）。

状态树的第 i 层代表第 i 列，每个非叶子结点有 3 个孩子结点，左子树该列表示加入 A，中子树表示加入 B，右子树表示丢弃

界限函数剪枝判断:

1）如果 A 的大小加剩余所有列的个数仍小于 B 的大小，剪枝

2）如果 A 的大小加 B 的大小加剩余所有列的个数小于最佳元素个数，剪枝

3）如果 A 的大小加 B 的大小加剩余所有列的个数等于最佳元素个数，但是差的绝对值到最后必然大于最佳元素差，剪枝

若新列加入在 A 中，则在 B 中判断该列是否与 B 中某一列有冲突。反之亦然。

当不满足界限函数或约束函数时，回溯。当到达叶节点时，用 update 函数判断是否储存最佳状态。

（4）算法实现：

```
#define ROW 1000
#define COL 20

#include<iostream>
#include<vector>
#include <fstream>
using namespace std;

ifstream fin("exp4_in.txt");
int best_len = 0, best_dif = COL;
bool conf[COL][COL];
vector<int> A, B;
vector<int> x[COL];
vector<int> bestA, bestB;
```

```
void update() {
   best_len = A.size() + B.size();
   best_dif = A.size() - B.size();
   bestA.assign(A.begin(), A.end());
   bestB.assign(B.begin(), B.end());
}
bool bound(int i) {
   if (A.size()+ (COL - i) < B.size())
      return false;
   if (A.size() + B.size() + (COL - i) < best_len)
      return false;

   if (A.size() + B.size() + (COL - i) == best_len) {
      if (A.size() - B.size() >= best_dif+ (COL - i))
         return false;}

}

bool feasible(int i, int flag) {
   vector<int> S;
   if (flag == 2)return true;
   if (flag == 0) S= B;
   if (flag == 1) S = A;
   for (auto col : S) {
      if (conf[i][col])
         return false;
   }
   return true;
}
void backtrack(int i, int flag)
{
   if (i > COL - 1) {   // 到达叶结点
      if (A.empty() || B.empty())
         return;
      if (feasible(i, flag)) {
         if ((A.size() + B.size() > best_len))
            update();
         else if ((A.size() + B.size() == best_len))
            if (A.size() - B.size() < best_dif)
               update();
         return;
      }
      return;
```

```
    }

    if (feasible(i, flag) && bound(i)) {
        if (flag == 0)
            A.push_back(i);
        else if (flag == 1)
            B.push_back(i);
        backtrack(i + 1, 0);
        backtrack(i + 1, 1);
        backtrack(i + 1, 2);
        if (flag == 0)
            A.pop_back();
        else if (flag == 1)
            B.pop_back();
    }

}
bool judge(int i, int j) {
    for (auto col : x[i]) {
        for (auto dol : x[j]) {
            if (col == dol)
                return true;
        }
    }
    return false;
}

int main() {
    int temp,n;
    fin >> n;
    for (int k = 0; k < n; k++) {
        for (int j = 0; j < ROW; j++) {  //第 j 行
            for (int i = 0; i < COL; i++) {
                fin >> temp;
                if (temp)
                    x[i].push_back(j);
            }
        }
        for (int i = 0; i < COL; i++) {
            for (int j = i; j < COL; j++) {
                conf[i][j] = judge(i, j);
                conf[j][i] = conf[i][j];
            }
        }
```

```
        backtrack(0, 0);
        backtrack(0, 1);
        backtrack(0, 2);

        for (auto col : bestA) {
            cout << col << " ";
        }
        cout << endl;
        for (auto col : bestB) {
            cout << col << " ";
        }
        cout << endl;
        A.clear();
        B.clear();
        bestA.clear();
        bestB.clear();
        for (int j = 0; j < COL; j++) {
            x[j].clear();
        }
        best_len = 0; best_dif = COL;
        memset(conf, 0, sizeof(conf));
    }
}
```

（5）运行结果：

输入：共 20000 行数据如下图

输出：共 20 组数据如下图

```
20
0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0
0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0
0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
```