第五章

1.

问题描述：子集和问题的一个实例为$<S, t>$。其中，$S=\{x_1, x_2, \cdots, x_n\}$是一个正整数的集合，$c$是一个正整数。子集和问题判定是否存在$S$的一个子集$S1$，使得$\sum_{x\in S1} x = c$。试设计一个解子集和问题的回溯法。

算法设计：对于给定的正整数的集合$S=\{x_1, x_2, \cdots, x_n\}$和正整数$c$，计算$S$的一个子集$S_1$，使得$\sum_{x\in S_1} x = c$。

递归回溯寻找子集，分为该数字在这个子集中和不在这个子集中两种情况。递归也是通过这两种情况进行的，注意“剪枝”，题目可能存在多个解，然而数据偏弱，所以只要从第一个数据进行递归就能得到正解。

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#define maxn 11234

using namespace std;

/**
*n 集合大小
*k 目标值
*s 集合
*sum 当前以累加值
*num 当前子集的大小。
*flag 标记是否已经找到子集。
*f 记录当前子集。
*/
int s[maxn], k, n, sum, num, flag, f[maxn];

void slove(int i){
    //如果已经找到合适的子集，结束递归。
    if(flag){
        return;
    }
    int j;
    //累加。
    sum += s[i];
    //当前数据进入子集。
    f[num++] = s[i];
    //大于目标值，，剪枝，结束递归。
    if(sum > k)
        return;
    //已经找到合适的子集。
```

```
    else if(sum == k){
        flag = 1;
        return;
    }
    //继续还没达到目标值，继续累加。
    for(j=i+1; j<n; j++){
        slove(j);
        if(!flag){
            //说明当前数据不适合进入子集，回溯。
            sum -= s[j];
            num--;
        }
        else{
            return;
        }
    }
}

int main(){
    int i;
    scanf("%d%d",&n,&k);
    sum = 0;
    //计算子集所有数据相加能否达到目标值，没有这一步骤会超时。
    for(i=0; i<n; i++){
        scanf("%d",&s[i]);
        sum += s[i];
    }
    if(sum < k){
        printf("No Solution!\n");
        return 0;
    }
    sum = num = flag = 0;
    //开始进入递归。
    for(i=0; i<n; i++){
        slove(i);
        if(!flag){
            sum -= s[i];
            num--;
        }
        else{
            break;
        }
    }
    if(flag){
```

```
        for(i=0; i<num; i++){
            printf("%d%c", f[i], i==num-1 ? '\n' : ' ');
        }
    }
    else{
        printf("No Solution!\n");
    }
    return 0;
}
```

2.

问题描述：设有 n 件工作分配给 n 个人。将工作 i 分配给第 j 个人所需的费用为 c_{ij}。试设计一个算法，为每个人都分配 1 件不同的工作，并使总费用达到最小。

算法设计：设计一个算法，对于给定的工作费用，计算最佳工作分配方案，使总费用达到最小。

```
#include<iostream>
#include<climits>
using namespace std;

int work[100];// 工作数组，用于存储工作编号
int work_fee[100][100];// 每个工人对应的每个工作的费用
int n, minsum= INT_MAX, newsum=0;// n 表示工人个数，minsum 表示最小费用和，newsum 表示最新费用和

void backTrack(int k) {
    int number; // 工作编号
    if (k > n) {// 到达叶子节点
        if (newsum < minsum) { // 当最新费用小于当前最小费用时，更新最小费用
            minsum = newsum;
        }
        return;
    }

    else {// 未到达叶子节点，继续
        for (int i = k; i <= n; i++) {
            number = work[i]; // 工作编号
            newsum = newsum + work_fee[k][number];// 将当前工人 k 的 number 号工作加入最新工作费用
            swap(work[k], work[i]);// 交换两个位置上的工作编号
            if (newsum < minsum) { //当最新费用小于当前最小费用，继续下一个数
                backTrack(k + 1);
            }
            swap(work[k], work[i]);// 还原之前交换的工作编号
            newsum = newsum - work_fee[k][number];// 减去之前加入的数
        }
    }
}

int main() {
    cin >> n; // 输入工人数
    for (int i = 1; i <=n; i++) { // 初始化工作费用
        for (int j = 1; j <= n; j++) {
            cin >> work_fee[i][j];
```

```
        }
    }
    for (int i = 1; i <= n; i++) {// 初始化工作编号
        work[i] = i;
    }
    backTrack(1);// 回溯查找
    cout << minsum << endl; // 输出最小费用
    return 0;
}
```

5-6 设 G 是有 n 个顶点的有向图，从顶点 i 发出的边的最小费用记为 $\min(i)$。

（1）证明图 G 的所有前缀为 $x[1:i]$ 的旅行售货员回路的费用至少为 $\sum_{j=2}^{i} a(x_{j-1}, x_j) + \sum_{j=i}^{n} \min(x_j)$，其中 $a(u,v)$ 是边 (u,v) 的费用。

（2）利用上述结论设计一个高效的上界函数，重写旅行售货员问题的回溯法，并与教材中的算法进行比较。

5-6.旅行售货员问题的上界函数：

（1）证明：前缀为$x[1:i]$的旅行售货员回路可表示为n个顶点的一个排列$(x[1], x[2], \cdots, x[i], \pi(i+1), \pi(i+2), \cdots, \pi(n))$。则该回路的费用为：

$$\hbar\ (i) = \sum_{j=2}^{i} a(x_{j-1}, x_j) + a(x_i, \pi(i+1)) + \sum_{j=i+1}^{n} a(\pi(j), \pi(j \bmod n + 1))$$

$$\geq \sum_{j=2}^{i} a(x_{j-1}, x_j) + \min(x_i) + \sum_{j=i+1}^{n} \min\ (\pi(j))$$

$$= \sum_{j=2}^{i} a(x_{j-1}, x_j) + \sum_{j=i}^{n} \min\ (x_j)$$

（2）假设当前最优值为 $\min\ \_cost$，当前费用为$cost$，前缀为$x[1:i-1]$的旅行售货员回路进入以$x[i]$为根结点的子树时，需满足$\hbar\ (i) < \min\ _cost$，否则无法得到比 $\min\ _cost$更优的解。因此，可以设计一个上界函数为：$min\_cost > (cost + a(x_{i-1}, x_i) + \sum_{j=i}^{n} \min\ (x_j))$。而教材中算法的上界函数为：$min\_cost > (cost + a(x_{i-1}, x_i))$，显然我们设计的上界函数更加高效。在下面代码的测试用例中，若使用教材中的上界函数，backtrack 函数需执行 19 次，而对于新的上界函数则只需执行 11 次，效率显著提高。但在最坏情况下，backtrack 函数仍需更新当前最优解$O((n-1)!)$次。

代码如下：

```
#include <iostream>
#include <cfloat>
using namespace std;

const int n = 5;            //图G的顶点个数
int x[n + 1];                   //当前解
int best_x[n + 1];          //当前最优解
float cost = 0;             //当前费用
float min_cost = FLT_MAX;//当前最优值
float min_x[n + 1];         //从顶点i发出的边的最小费用

int count = 0;              //记录递归次数

//邻接矩阵
float a[n + 1][n + 1] = {
```

```
        0, 0, 0, 0, 0, 0,
        0, -1, 5, 61, 34, 12,
        0, 57, -1, 43, 20, 7,
        0, 39, 42, -1, 8, 21,
        0, 6, 50, 42, -1, 8,
        0, 41, 26, 10, 35, -1
};

//求从各顶点发出的边的最小费用
void x_min(void) {
    for (int i = 1; i <= n; i++) {
        min_x[i] = FLT_MAX;
        for (int j = 1; j <= n; j++) {
            if (j != i && a[i][j] < min_x[i])
                min_x[i] = a[i][j];
        }
    }
}

//约束函数
bool constraint(int t) {
    return a[x[t - 1]][x[t]] > 0;
}

//限界函数
bool bound(int t) {
    float sum_min = 0;
    for (int i = t; i <= n; i++)
        sum_min += min_x[x[i]];
    return min_cost > (cost + a[x[t - 1]][x[t]] + sum_min);
}

/*
//教材上的限界函数
bool bound(int t) {
    return min_cost > (cost + a[x[t - 1]][x[t]]);
}
*/

void backtrack(int t) {
    ::count++;
    if (t == n) {
        if (a[x[n - 1]][x[n]] > 0 && a[x[n]][1] > 0 && min_cost > cost + a[x[n - 1]][x[n]]
+ a[x[n]][1]) {
```

```cpp
            for (int i = 1; i <= n; i++)
                best_x[i] = x[i];
            min_cost = cost + a[x[n - 1]][x[n]] + a[x[n]][1];
        }
    }
    else {
        for (int i = t; i <= n; i++) {
            //是否可以进入x[t]子树
            if (constraint(t) && bound(t)) {
                swap(x[t], x[i]);
                cost += a[x[t - 1]][x[t]];
                backtrack(t + 1);
                cost -= a[x[t - 1]][x[t]];
                swap(x[t], x[i]);
            }
        }
    }
}

float tsp(void) {
    for (int i = 1; i <= n; i++)
        x[i] = i;
    x_min();
    backtrack(2);
    return min_cost;
}

//测试程序
int main(void) {
    cout << "最小费用: " << tsp() << endl;
    cout << "路径: ";
    for (int i = 1; i <= n; i++)
        cout << best_x[i] << "->";
    cout << best_x[1] << endl;
    cout << "递归次数: " << ::count << endl;

    return 0;
}
```