5-6：

> 设 G 是有 n 个结点的有向图，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。

题目分析

(1)：

证：对于前缀为 $x[1:i]$ 的旅行售货员回路，目前的花费 $currentcost = \sum_{j=2}^{i} a(x_{j-1}, x_j)$。

对于剩余的回路，假设每一步都尽可能走最小出边（能走小的就走小的），则每一步的花费 $cost[j] \geq min[x_j]$， 将剩余的花费求和，得 $\sum_{j=i}^{n} \geq \sum_{j=i}^{n} min(x_j)$，与 $currentcost$ 相加，即得：

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

Q.E.D.

(2)：

首先要根据 (1) 中的结论设计出上界函数，上界函数为：

$$MINCOST + ccost \leq bestc \tag{2}$$

其中，$MINCOST$ 为剩余未遍历结点的最小边权之和，如果每个结点均走最小边仍然不能优于当前最优解，则该子树内不会有最优解，应当剪枝。

代码实现

修改原本的程序，加入上界函数：

```
#include <iostream>
#include <ctime>
#include <windows.h> // 引入Sleep
using namespace std;

int n;   // 图G的结点数
int x[100];  // 当前的解空间
int bestx[100];  // 当前的最优解
float bestcost;  // 当前最优值
float ccost;  // 当前费用
float a[100][100];  // 图G的邻接矩阵
```

```cpp
float mincost[100];  // 对于每一个结点i，其最小出边的cost为mincost[i]
float MINCOST;  // 用于记录剩余结点的 mincost之和，一开始初始化为mincost[i]之和
const float MAX = 2000000.0;  // 无穷大，表示没有路径相连
// 变量定义部分

void backtrace(int i);
void init();

int main(){
    init();
    clock_t t_begin = clock();
    backtrace(2);
    clock_t t_end = clock();
    cout << "最短路径长为: " << bestcost << endl;
    cout << "最短路径为: ";
    for(int i = 1; i <= n; i++){
        cout << bestx[i] << " ";
    }
    cout << endl << "回溯耗时为: " << (t_end - t_begin) << "ms";
}

void init(){
    n = 10;
    for(int i = 1; i <= n; i++) x[i] = i, bestx[i] = i, mincost[i] = MAX;
    ccost = MINCOST = 0;
    bestcost = MAX;
    for(int i = 1; i <= n;i++){
        for(int j = 1; j <= n; j++)
            a[i][j] = MAX;
    }
    a[1][9] = a[9][1] = a[8][7] = a[7][8] = a[2][4] = a[4][2] = 5;
    a[4][10] = a[10][4] = a[3][6] = a[6][3] = a[1][8] = a[8][1] = 4;
    a[2][7] = a[7][2] = a[1][3] = a[3][1] = a[5][8] = a[8][5] = a[9][10] = a[10][9] = 6;
    a[4][5] = a[5][4] = a[8][9] = a[9][8] = 2;
    a[1][7] = a[7][1] = a[3][10] = a[10][3] = a[7][10] = a[10][7] = 3;
    a[3][8] = a[8][3] = a[5][6] = a[6][5] = 9;
    a[1][4] = a[4][1] = a[2][10] = a[10][2] = a[6][9] = a[9][6] = 8;
    a[6][7] = a[7][6] = 1;
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= n; j++){
            if(a[i][j] != MAX) mincost[i] = min(mincost[i], a[i][j]);
        }
        MINCOST += mincost[i];
```

```cpp
    }
}

void backtrace(int i) {
    if (i == n) {
        if (a[x[n-1]][x[n]] < MAX && a[x[n]][1] < MAX &&
           (ccost + a[x[n-1]][x[n]] + a[x[n]][1] < bestcost || bestcost == MAX)) {
            for (int j = 1; j <= n; j++) bestx[j] = x[j];
            bestcost = ccost + a[x[n - 1]][x[n]] + a[x[n]][1];
        }
    }
    else {
        for(int j = i; j <= n; j++) {
            if (a[x[i - 1]][x[j]] < MAX &&
               (bestcost == MAX || ccost + a[x[i - 1]][x[j]] < bestcost) &&
                    ccost + MINCOST < bestcost) {
                swap(x[i], x[j]);
                ccost += a[x[i - 1]][x[i]];
                MINCOST -= mincost[x[i]];
                backtrace(i + 1);
                Sleep(2);  // 每次回溯一次就要休眠2ms，为了对比不同算法之间的效率
                ccost -= a[x[i - 1]][x[i]];
                MINCOST += mincost[x[i]];
                swap(x[i], x[j]);
            }
        }
    }
}
```

样例输出

下面先给出没有使用上界函数优化的代码的输出，再给出5-5中的输出，最后给出使用上界函数的输出：

```
// 无上界函数
最短路径长为：39
最短路径为：1 7 6 3 10 2 4 5 8 9
回溯耗时为：4366ms
// 使用maxcost对bestc进行初始化
最短路径长为：39
最短路径为：1 7 6 3 10 2 4 5 8 9
回溯耗时为：4349ms
// 使用高效上界函数
最短路径长为：39
最短路径为：1 7 6 3 10 2 4 5 8 9
回溯耗时为：3018ms
```

算法分析

分析样例输出可得：

5-5中初始化bestc的算法尽管简化了代码，但并没有优化时间复杂度，效率与无上界函数的版本一致。

5-6中使用了高效的上界函数，效率有了显著的提高。

从算法复杂度分析，虽然本算法最坏情况下与无上界函数情况一致，均为$O(n!)$，但从输出可以看出，本算法效率远远优于之前的两种算法，其因就在于高效的上界函数剪去了杂枝，让解空间树不再庞大，从而减小了搜索的范围。