4-5.最优装载问题：

若使用贪心算法求解两艘船的最优装载问题，则第一艘船的装载方案只有一种最优解，即将物品按从轻到重的顺序进行装载，但这样并不一定能将第一艘船装满，可能会造成空间的浪费。例如：物品重量为{3，3，3，5，5，6}，两艘船的载重量分别为 10、15。若使用贪心算法，则会将重量为 3、3、3 的物品装上第一艘船，空间浪费为 1，此时第二艘船只能再装下两个物品，总共装下了 5 个物品。实际上，可以把重量为 5、5 的物品装上第一艘船，其余物品装上第二艘船，此时刚好全部装下，共 6 个物品。因此，若将最优装载问题的贪心算法推广到 2 艘船的情形，无法产生最优解。

4-12.最大权值达到最小的生成树：

假设T是图G的一棵最小生成树，T'是图G的一棵使最大权值达到最小的生成树，但不是最小生成树。e是T中的最大权边，e'是T'中的最大权边，且$\omega(e') < \omega(e)$。将e从T中删去后，T将分为两个连通分支。此时，一定有T'中的边e''连接这两个连通分支，否则T'将不是连通的。将e''加入T的这两个连通分支将得到一棵新的生成树$T'' = T - e + e''$。由于e'是T'中的最大权边，故$\omega(e'') \le \omega(e') < \omega(e)$，从而有$\omega(T'') = \omega(T) - \omega(e) + \omega(e'') < \omega(T)$。这与$T$是最小生成树矛盾，故$T'$就是最小生成树。

因此，可以使用 Prim 算法构造 G 的最大权值达到最小的生成树。

算法描述如下：

```
void prim(int n, edge **e) {
    T = 空集;
    S = { 1 };
    while ( S != V ) {
        ( i, j ) = i ∈ S and j ∈ V-S 的最小权边;
        T = T ∪ { ( i, j ) };
        S = S ∪ { j };
    }
}
```

4-14.整数权值的 Dijkstra 算法：

由于各边的权值为 0~N-1 的整数（N 为非负整数），因此图中任意一个点到源点的距离 dist 值不会超过 N*n，从而可以建立一个长度为 N*n 的队列数组，数组中的每个队列存储着“dist 值为该队列下标值”的顶点。刚开始先将源点入第 0 个队列，然后从左向右线性扫描队列数组，此时最先遇到的非空队列存储着 dist 值最小的顶点，根据贪心算法的思想，将该顶点放入集合 S 中，再把 V-S 中与该顶点相邻的顶点入相应队列，不断进行上述过程直至计算出全部的 dist 值。

最多会扫描一遍队列数组，复杂度为 O(N*n)。此外，最多会有 e 次入、出队列操作，复杂度为 O(e)。故总的时间复杂度为 O(N*n+e)。

代码如下：

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

```
const int N = 101;    //权值上限
const int n = 5;      //顶点个数

//邻接矩阵（-1表示边不存在）
const int edge[n][n] = {
    {0, 10, -1, 30, 100},
    {-1, 0, 50, -1, -1},
    {-1, -1, 0, -1, 10},
    {-1, -1, 20, 0, 60},
    {-1, -1, -1, -1, 0}
};

void dijkstra(const int& source) {
    //处理错误输入
    if (source <= 0 || source > n) {
        cout << "False input!" << endl;
        return;
    }

    //初始化辅助数组
    int dist[n];
    for (int i = 0; i < n; i++)
        dist[i] = -1;
    bool visited[n] = { false };
    queue<int> bucket[N * n];

    bucket[0].push(source - 1);
    int count = 0;
    //线性扫描队列数组
    for (int i = 0; i < N * n; i++) {
        while (!bucket[i].empty()) {
            //出队列
            int v = bucket[i].front();
            bucket[i].pop();
            if (visited[v])
                continue;
            //获取最短路径
            dist[v] = i;
            visited[v] = true;
            count++;
            if (count == n)
                break;
            //将未访问的相邻顶点入队列
```

```
            for (int j = 0; j < n; j++) {
                if (!visited[j] && edge[v][j] > 0)
                    bucket[i + edge[v][j]].push(j);
            }
        }
        if (count == n)
            break;
    }

    //输出
    for (int i = 0; i < n; i++)
        cout << "dist[" << i + 1 << "]:" << dist[i] << endl;
}

//测试程序
int main(void) {
    cout << "source point(from 1-" << n << "):";
    int s;
    cin >> s;
    dijkstra(s);
    return 0;
}
```