在分支限界法中每个活节点只有一次机会成为扩展节点，一旦遍历到这个点就一次性产生所有子节点，分支限界法与回溯算法不同之处在于前者使用的遍历算法是广度优先搜索，回溯算法所要得到的是所有答案，而分支限界法只要最优或其中一个答案。

分支限界法主要运用了队列和优先队列。

单源最短路

运用了分治限界的单元最短路算法原理是通过堆优化令查找节点更快。即为优化版dijkstra。

复杂度为$O(m+n)logn$
详情见贪心部分堆优化版dijkstra。

补充思路：
从起点开始，将起点作为扩展节点入队。并将所有点按照到目前集合的距离作为优先级将他们入队，距离越短优先级越高。
接着不断从堆中取出最近的点，加入集合，并用这个点更新到其他点的距离。当我们的目标点被加入集合之后，就结束。

一点复杂度分析：复杂度主要来自这几个方面：对于每一个扩展节点（或者说每一层），有这么几步操作：

- 找到最近的点（堆优化之下找到最近的点复杂度为$O（1）$）
- 将他加入集合中($O(1)$)
- 用新的点更新到其他点的距离
  (本质上遍历的是所有边。边数是m，每次修改堆中的数需要的时间为$logn$，因此总共的复杂度为$mlogn$)
- 使用stl建堆过程复杂度$O(nlogn)$

总共复杂度为$O((m+n)logn)$

装载问题

优先队列版

本质就是扩展出一系列的情况。我们首先构造队列，队列中相当于存放了各种情况，例如：
我们选取/不选取第i个点，等等。每一个节点的值就是我们进行到这一层时，取得的目前装载最大值。
在每一层节点之后我们加上-1，这是在告诉程序：如果你在队列中提出了-1，说明你把这一层已经遍历完了。如果队列为空，说明活节点扩展完毕，退出；反之继续搜索。

> 有一批共n个集装箱要装上艘载重量为c的轮船，其中集装箱i的重量为wi。找出一种最优装载方案，将轮船尽可能装满，即在装载体积不受限制的情况下，将尽可能重的集装箱装上轮船。

```
# include<bits/stdc++.h>
using namespace std;
int answer=-0x3f3f3f3f;
queue<int>q;
int n;
int c;
int w[1000005];

void bfs()
{
        int i=0;
        q.push(-1);
        q.push(w[0]);
        q.push(0);

        while(true)
        {
                auto t =q.front();
                q.pop();

                if(t==-1)
                {
                        if(q.size()==0)
                        {
                                return;
                        }

                        else
                        {
                                q.push(-1);
                                i++;
                                continue;
                        }
                }

                if(i==n)
                {
                        answer=max(answer,t);
                        continue;
                }
                if(t+w[i]<=c)
                {
                        q.push(t+w[i]);
```

```
                        q.push(t);
                }
                else
                q.push(t);

        }
}
int main (void)
{
        cin>>n>>c;
        for(int i=0;i<n;i++)
        cin>>w[i];
        bfs();
        cout<<answer<<endl;

}
```

改进策略：
每次进入左子树时，我们就可以直接更新answer的值。进入右子树时，如果当前节点的权值加上剩余所有货物的权值依然小于答案，我们就不用遍历了，直接剪枝。

```
# include<bits/stdc++.h>
using namespace std;
int answer=-0x3f3f3f3f;
queue<int>q;
int n;
int c;
int w[1000005];
int remain;
void bfs()
{
        int i=0;
        q.push(-1);
        q.push(w[0]);
        q.push(0);

        while(true)
        {
                auto t =q.front();
                q.pop();

                if(t==-1)
                {
```

```
                        if(q.size()==0)
                        {
                                return;
                        }

                        else
                        {
                                q.push(-1);
                                i++;
                                remain-=w[i];

                                continue;
                        }
                }

                if(i==n)
                {
                        answer=max(answer,t);
                        continue;
                }
                if(t+w[i]<=c)
                {
                        q.push(t+w[i]);
                        q.push(t);
                        answer=max(answer,t+w[i]);
                }
                else
                {
                        if(t+remain<answer)
                        continue;

                q.push(t);
                }
        }
}
int main (void)
{
        cin>>n>>c;
        for(int i=0;i<n;i++)
        cin>>w[i];
        for(int i=1;i<n;i++)
        remain+=w[i];
```

```
        bfs();
        cout<<answer<<endl;

}
```

优先队列版

思路与上相同，只不过我们利用一个大顶堆存放状态。每个节点的优先级就是从根节点出发到这个节点的载重加上剩余所有货物的质量。我们容易知道一旦遍历到了叶子结点就找到了答案。

布线问题

bfs。从起点开始一圈一圈往外搜，当我们第一次搜索到了目的地时，就搜到了最短路径。如果要求最优解，开设pair的二维数组，分别记录这个点的前驱。

```
# include<bits/stdc++.h>
using namespace std;
queue<pair<int,int>>q;
pair<int,int>pre[1005][1005];
int arr[1005][1005];
int px[4]={-1,1,0,0};
int py[4]={0,0,-1,1};
int n;
int dis[1005][1005];

void bfs()
{
        memset(dis,-1,sizeof dis);

        q.push({0,0});
        dis[0][0]=0;

        pre[0][0]={-1,-1};

        while(true)
        {
                auto t=q.front();
                q.pop();
                for(int i=0;i<4;i++)
                {
                        int tempox=t.first+px[i];
                        int tempoy=t.second+py[i];
                        if(arr[tempox][tempoy]==0&&dis[tempox][tempoy]==-1)
```

```
                        {
                                dis[tempox][tempoy]=dis[t.first][t.second]+1;
                                pre[tempox][tempoy]={t.first,t.second};
                                q.push({tempox,tempoy});
                        }
                }
                if(dis[n-1][n-1]!=-1)
                break;

        }
}
int main (void)
{
        cin>>n;
        for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
        cin>>arr[i][j];
        int now1=n-1;
        int now2=n-1;
        bfs();



        while(1)
        {
                if(now1==-1)
                break;
                cout<<pre[now1][now2].first<<" "<<pre[now1][now2].second<<endl;
                now1=pre[now1][now2].first;
                now2=pre[now1][now2].second;

        }
}
```

0-1背包问题

和之前的装载问题本质一样。只不过在此限制条件是重量，所要找的最佳目标评价指标是价值。我们可以使用存放pair的队列进行处理。

旅行售货员问题

使用堆优化版的dijkstra算法。
选取一个起点，建堆，小顶堆优先级是当前路径费用加上剩余所有顶点的最小出边之和。选取扩展节点

之后扩展出其所有孩子节点。当遍历到叶子节点时更新最优费用。如果孩子节点的费用下限还是高于当前最优费用，那么就不将孩子节点入队（做剪枝）。

另一种更常用的方式（考试不考）
spfa+dfs

首先用n次spfa求出所有点的单源最短路，再用dfs遍历得出最短的路径。
复杂度$O(n!+nm)$（前者往往远远小于$n!$）

重庆城里有 n 个车站，m 条 **双向** 公路连接其中的某些车站。

每两个车站最多用一条公路连接，从任何一个车站出发都可以经过一条或者多条公路到达其他车站，但不同的路径需要花费的时间可能不同。

在一条路径上花费的时间等于路径上所有公路需要的时间之和。

佳佳的家在车站 1，他有五个亲戚，分别住在车站 a,b,c,d,e。

过年了，他需要从自己的家出发，拜访每个亲戚（顺序任意），给他们送去节日的祝福。

怎样走，才需要最少的时间?

输入格式

第一行：包含两个整数 n,m，分别表示车站数目和公路数目。

第二行：包含五个整数 a,b,c,d,e，分别表示五个亲戚所在车站编号。

以下 m 行，每行三个整数 x,y,t，表示公路连接的两个车站编号和时间。

输出格式

输出仅一行，包含一个整数 T，表示最少的总时间。

```
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <queue>

using namespace std;

typedef pair<int, int> PII;

const int N = 50010, M = 200010, INF = 0x3f3f3f3f;

int n, m;
int h[N], e[M], w[M], ne[M], idx;
int q[N], dist[6][N];
```

```
int source[6];
bool st[N];

void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;
}

void dijkstra(int start, int dist[])
{
    memset(dist, 0x3f, N * 4);
    dist[start] = 0;
    memset(st, 0, sizeof st);

    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, start});

    while (heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second;
        if (st[ver]) continue;
        st[ver] = true;

        for (int i = h[ver]; ~i; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[ver] + w[i])
            {
                dist[j] = dist[ver] + w[i];
                heap.push({dist[j], j});
            }
        }
    }
}

int dfs(int u, int start, int distance)
{
    if (u > 5) return distance;

    int res = INF;
```

```
    for (int i = 1; i <= 5; i ++ )
        if (!st[i])
        {
            int next = source[i];
            st[i] = true;
            res = min(res, dfs(u + 1, i, distance + dist[start][next]));
            st[i] = false;
        }

    return res;
}

int main()
{
    scanf("%d%d", &n, &m);
    source[0] = 1;
    for (int i = 1; i <= 5; i ++ ) scanf("%d", &source[i]);

    memset(h, -1, sizeof h);
    while (m -- )
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c), add(b, a, c);
    }

    for (int i = 0; i < 6; i ++ ) dijkstra(source[i], dist[i]);

    memset(st, 0, sizeof st);
    printf("%d\n", dfs(1, 0, 0));

    return 0;
}
```