回溯法

填空题会有代码填空，大题会手动回溯

1. 回溯法的算法框架

1.1. 问题的解空间

> 1 用回溯法解问题时，应明确定义**问题的解空间**
>
> 2 解空间的向量表示：问题的解向量，$(x_1, x_2, ..., x_n)$
>
> 1. **显约束**：对分量x_i的取值限定
> 2. **隐约束**：为满足问题的解，对不同分量之间施加的约束
> 3. **解空间**：即解向量满足**显式约束**条件的所有多元组

1.2. 回溯法思想

> 1 回溯法的基本做法：组织有序的，避免不必要搜索的穷举式搜索法
>
> 2 三类结点
>
> 1. 活结点：自生已生成，子节点还未全部生成
> 2. 扩展结点：正在生成子节点
> 3. 死结点：子节点全部生成完毕
>
> 3 两种问题状态生成法
>
> 1. 深度优先：
>   - 扩展结点R生成一个子节点C
>   - 将C作为新的扩展结点，完成对以C为根的子树的穷尽搜索
>   - 重新将R变回扩展结点，继续生成R的下一个子节点
> 2. 广度优先：在一个扩展结点变成死结点之前，它一直是扩展结点
>
> 4 回溯法如何搜索解空间树：
>
> 1. 按照深度优先策略，从根节点出发
> 2. 搜索至解空间树的任意一点时，判断该节点是否包含问题解
>   - 包含时：进入该子树，继续按深度优先策略搜索
>   - 不包含时：跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯
>
> 5 回溯法基本思想
>
> 1. 针对所给问题，定义问题的解空间，确定易于搜索的解空间结构
> 2. 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索
>
> 6 剪枝函数
>
> 1. 用**约束函数**在扩展结点处剪去不满足约束的子树
> 2. 用**限界函数**剪去得不到最优解的子树

1.3. 两种回溯

```
//t为递归深度
//f(n,t)表示在当前扩展结点处未搜索过的子树的起始编号
//g(n,t)为终止编号
//h(i)表示当前扩展结点处x[t]的第i个可选值
```

1.3.1. 递归回溯(背下来)

```
void Backtrack (int t)
{
 if (t>n) Output(x); //记录或输出可靠解x，x为数组
 else
 {
     for (int i=f(n,t); i<=g(n,t); i++)
     {
         x[t]=h(i);
         if (Constraint(t)&&Bound(t)) //剪枝
             Backtrack(t+1);
     }
 }
}
```

1.3.2. 迭代回溯(会填空即可)

```
void IterativeBacktrack ()
{
 int t=1;
 while (t>0)
 {
     if (f(n,t)<=g(n,t))
     {
         for (int i=f(n,t); i<=g(n,t); i++)
         {
             x[t]=h(i);
             if (Constraint(t)&&Bound(t))
             {
                 if (solution(t)) Output(x); //判断是否已得到可行解
                 else t++;
             }
         }
     }
     else t--;
 }
}
```

1.4. 子集树和排列树

1.4.1. 子集树

1 含义：要求从集合S找出满足某种性质的子集时，相应解空间树就是子集树

2 算法描述：时间复杂度为$\Omega(2^n)$

```
void Backtrack (int t)
{
 if (t>n) Output(x);
 else
 {
     for (int i=0; i<=1; i++)
     {
         x[t]=i;
         if (Constraint(t)&&Bound(t))
             Backtrack(t+1);
     }
 }
}
```

1.4.2. 排列数

1 含义：要求从集合S找出满足某种性质的元素的排列时，相应解空间树就是排列树

2 算法描述：时间复杂度为$\Omega(n!)$

```
void Backtrack (int t)
{
 if (t>n) Output(x);
 else
 {
     for (int i=t; i<=n; i++)
     {
         Swap(x[t], x[i]);
         if (Constraint(t)&&Bound(t))
             Backtrack(t+1);
         Swap(x[t], x[i]);
     }
 }
}
```

2. 装载问题

2.1. 问题描述

1 有两艘船载重量为C_1，C_2

2 有n个集装箱，集装箱i的重量为w_i，满足$\sum_{i=1}^{n} w_i \leq C_1 + C_2$

3 设计一个方案：将所有集装箱都装入这两个船，并且第一艘轮船尽可能装满

这实质上等价于——选取全体集装箱的一个子集，该子集总重接近于C_1

2.2. 算法分析

1 可行性约束函数：$\sum_{i=1}^{n} w_i x_i \leq c_1$其中$x_i = 0/1$表示第$i$个集装箱是否放入第一个船中

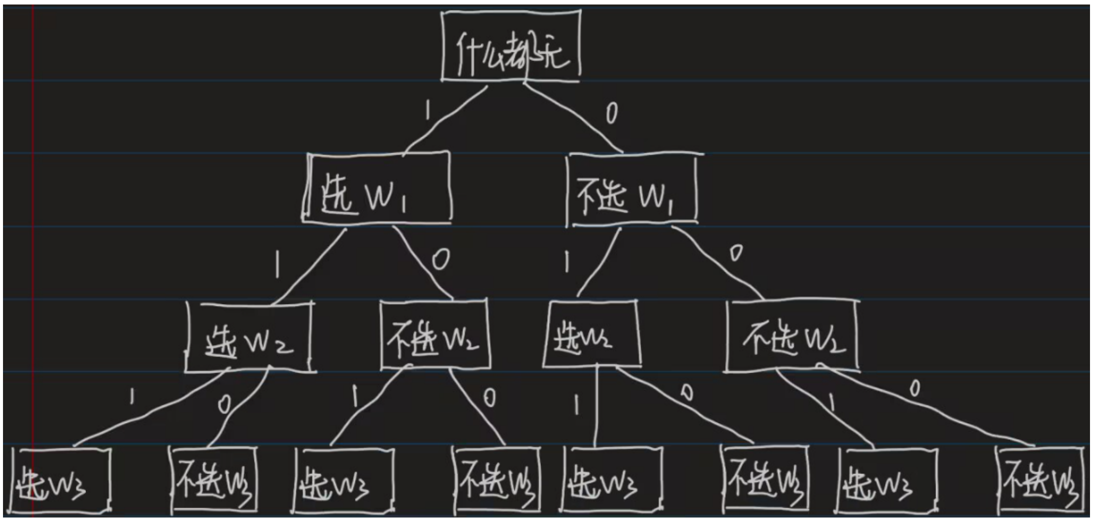
2 全局变量初始化

```
const int   = 100;  // 假设集装箱数量的最大值
int w[ ];           // 集装箱重量数组
int c;                  // 第1艘轮船的载重量
int n;                  // 集装箱数
int cw = 0;             // 当前载重量
int bestw = 0;          // 当前最优载重量
```

3 函数 `Backtrack`： 实现了回溯法的逻辑，用于搜索解决装载问题的最优解

```
void Backtrack(int i) //搜索第i层结点
{
 //在叶结点处，验证目前载重知否大于已知最优，然后更新后退出
 if(i>n)
 {
     if(cw>bestw) bestw=cw;
     return;
 }
 //非叶结点处继续递归
 //当前载重量+第i个集装箱重量仍小于载重限制时，装入该集装箱
 if(cw+w[i]<=c)
 {
     cw+=w[i];
     Backtrack(i+1); //递归调用以考虑下一个集装箱，即继续搜索下一层
     cw-=w[i]; //递归返回后撤销对cw的更新，即退出左子树
 }
 Backtrack(i+1); //无论是否选择集装箱i，都探索不选择i的情况，即进入右子树
}
/*在主函数中调用 Backtrack(1)，表示从第1层开始搜索*/
```

子集树：以三个集装箱w_1, w_2, w_3为例讲解以上代码，假如最后的结果是选择w_1, w_3



1. 起始：在根节点没选择任何集装箱， `cw=0`

2. 第一层决策：尝试选择`w1`(左子节点)，`cw + w1`小于载重限制，所以更新`cw`并进入下一层`Backtrack(2)`
3. 第二层决策：尝试选择`w2`(左子节点)，`cw + w2`小于载重限制，所以更新`cw`并进入下一层`Backtrack(3)`
4. 第三层决策：尝试选择`w3`(左子节点)，`cw + w3`大于载重限制，所以不进入左子节点，而是调用`Backtrack(3)`进入右子节点
5. 已进入叶节点：得到了一个解`x[3]=[1,1,0]`，记录$w_{best} = w_1 + w_2$
6. 开始回溯：回到第二层决策，撤销对`w2`的选择然后回到第一层决策
7. 然后开始探索右子节点(不选择`w2`)，以此类推

4️⃣ 复杂度分析：每个结点处花费$O(1)$，结点个数为$O(2^n)$，故时间复杂度为$O(2^n)$

2.3. 上界函数：剪去不含最优解的子树

```
int r = 0;// 剩余集装箱重量

void Backtrack(int i)
{
 if(i > n)
 {
     if(cw > bestw) bestw = cw;
     return;
 }
 r -= w[i];  // 更新剩余集装箱重量
 if(cw + w[i] <= c)
 {
     cw += w[i];
     Backtrack(i + 1);
     cw -= w[i];  // 回溯
 }
 if(cw + r > bestw) Backtrack(i + 1);  // 检查是否有必要继续探索当前分支
 r += w[i];  // 回溯，恢复剩余集装箱重量
}
```

1️⃣ 较原函数的改进

1. 新增一变量`r`：表示剩余集装箱重量
2. 将原有`if`外的`Backtrack(i+1)`改为上界函数：

```
if(cw+r>bestw) {Backtrack(i+1);}
r+=w[i];
```

2️⃣ 上界函数的含义：当前载重`cw`+剩余集装箱重量`r`，都要小于当前最优的时候，也没必要再继续了，直接吐出当前选择的集装箱重量，然后开始回溯

2.4. 构造最优解

1️⃣ 新增数据结构

1. `int* x`记录从根至当前结点的路径
2. `int* bestx`记录当前最优解的路径

2 修正后的代码：`bestx`可能被更新$O(2n)$次，故算法的时间复杂性为$O(n2^n)$

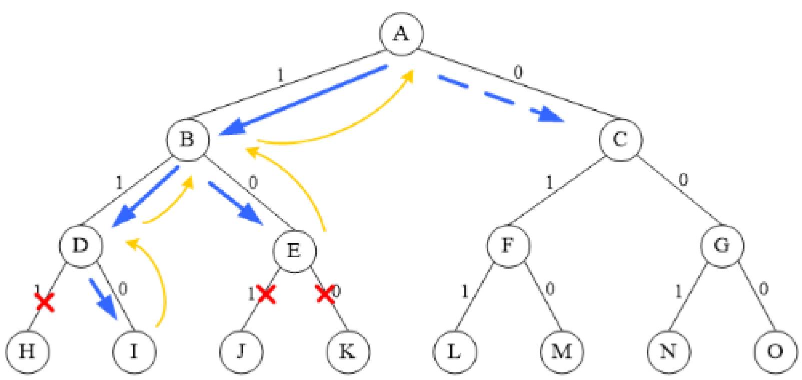
```
void backTrack(int i)// 搜索第i层结点
{
    if (i > n) // 到达叶结点
    {
        if (cw > bestw) //找到了更优的解
        {
            for (int j = 1; j <= n; j++) bestx[j] = x[j];//最优解路径设为当前路径
            bestw = cw;//更新最优解的值
        }
        return;
    }

    r -= w[i];//先选定当前集装箱，减去剩余集装箱重量，再通过以下来查看可行不可行
    if (cw + w[i] <= c)//加上当前集装箱后，不超过最大限制
    {
        x[i] = 1;//记录下所走过的当前路径
        cw += w[i];//加上目前装在后得到的重量
        backTrack(i + 1);//进入下一层
        cw -= w[i];//退出左子树
    }
    if (cw + r > bestw)//当所有剩下集装箱质量+当前质量都小于目前最优秀时，不再装下去
    {
        x[i] = 0;//不装第i个集装箱，路径上也不标识
        backTrack(i + 1);//进入右子树
    }
    r += w[i];
}
```

2.5. 迭代回溯(非递归形式，会填空即可)

1 条件：$n=3, c_1=c_2=50, w=[10,40,40]$

2 子集树以及查找&回溯路径



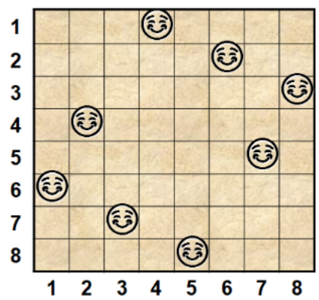
3 代码实现：复杂度为$O(2^n)$

```
template<typename Type>
Type MaxLoading(Type w[],Type c,int n,int bestx[])
{
 for(int j=1;j<=n;j++) r+=w[j];
```

```
    while(true) //搜索子树
 {
     //进入左子树，条件为真，则一直往左搜索
     while(i<=n&&cw+w[i]<=c) {r-=w[i];cw+=w[i];x[i]=1;i++;}
     //到达叶结点，更新最佳路径和最优解
     if(i>n)
     {
         for(int j=1;j<=n;j++) bestx[j]=x[j];
         bestw=cw;
     }
     //左子树走不下去了时，就进入右子树，吐出之前在左子树选择的集装箱重量，删除路径
     else {r-=w[i];x[i]=0;i++;}
     //剪枝回溯，依然是上界函数
        while(cw+r<=bestw)
     {
         i--;
         while(i>0&&!x[i]) {r+=w[i];i--;} //从右子树返回
         if(i==0) {delete[]x;return bestw;} //如返回到根，则结束
         //进入右子树
         x[i]=0;cw-=w[i];i++;
     }
 }
}
```

3. n皇后问题

3.1. 问题描述

> 在$n\times n$棋盘上放置皇后，使得同一行&&同一列&&同一对角线，都只有一个皇后
>
> 
>

3.2. 算法设计

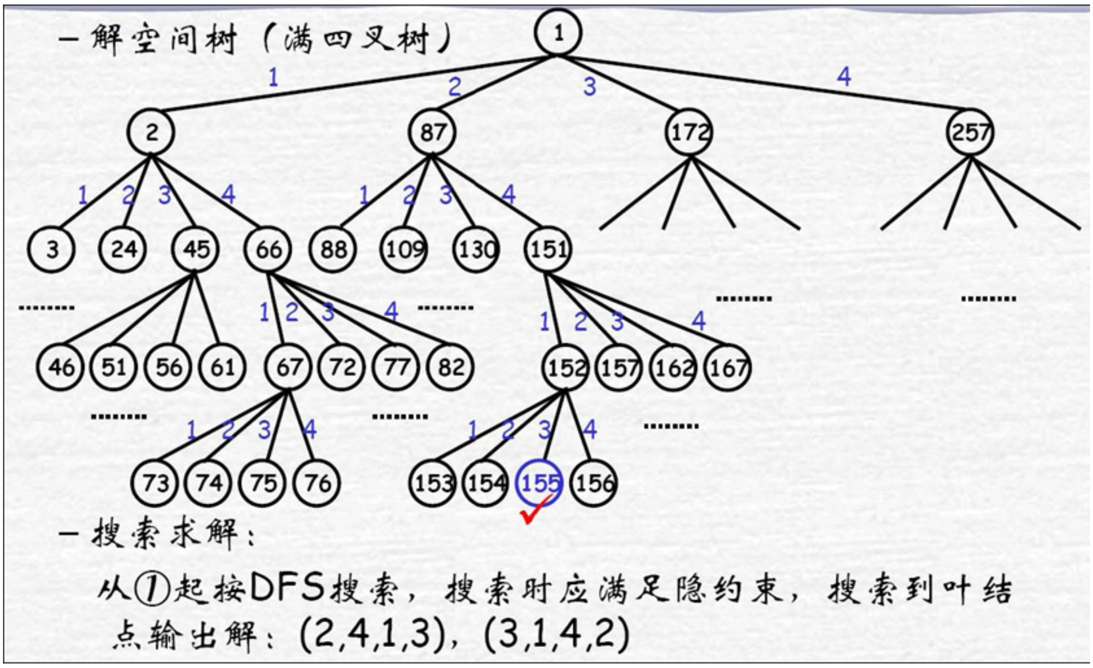
> **1** 解向量：$(x_1,x_2,\ldots,x_n)$，x_i表示皇后位于棋盘的(i,x_i)处，上图的解向量为$(4,6,8,2,7,1,3,5)$
>
> **2** 约束
>
> 1. 显约束：$i=1,2,\ldots,n$且$x_i=1,2,\ldots,n$
> 2. 隐约束：不同列$x_i\neq x_j$且不处于同一正反对角线$|i-j|\neq|x_i-x_j|$
>
> **3** 解空间：由解向量一行行构成的矩阵，用完全n叉树表示
>
> **4** 修饰：用可行性约束`Place()`剪去不满足行/列/对角线约束的子树
>
> **5** 有关数据结构：

1. `Backtrack(i)`：搜索解空间中第i层子树
2. `sum`：记录当前已找到的可行方案数

3.3. 简化：四皇后问题

1 解空间的完全四叉树表示

1. 根：搜索的初始状态
2. 第$i \to i+1$层结点连线上的数字：i行的皇后所摆放的列数



2 全局变量及其初始化

```
int n;     // 皇后个数
int[] x;   // 当前解，即x[i]表示第i行皇后所在的列数
long sum;  // 当前已找到的可行方案数，初始化为0
```

3 检查第k行的放置是否合法

```
bool place(int k)
{
    //遍历之前所有k→k-1已放置的皇后
    //如果当前k行的对角线冲突|k-i|=|x[k]-x[i]|，或者列冲突x[i]=x[k]，就返回非法
    for (int i = 1; i < k; i++)
    {
        if (Math.abs(k-i)==Math.abs(x[k]-x[i])||x[i]==x[k]) {return false;}
    }
    return true;
}
```

4 递归回溯

```
void backTrack(int t) //考虑第t行的皇后
{
    if (t > n) sum++; //已经来到叶结点了，意味找到一个解了，可行方案数+1
```

```
    else
    {
        //遍历i=1→n列，现尝试将第t行皇后放在第i列，然后检查其合法性
        //若放置合法，就递归调用，放置第t+1行(下一行)的皇后
        for (int i = 1; i <= n; i++)
        {
            x[t] = i;
            if (place(t)) backTrack(t + 1);
            //从backTrack(t + 1)返回后，会回溯，然后尝试下一个i值
        }
    }
}
```

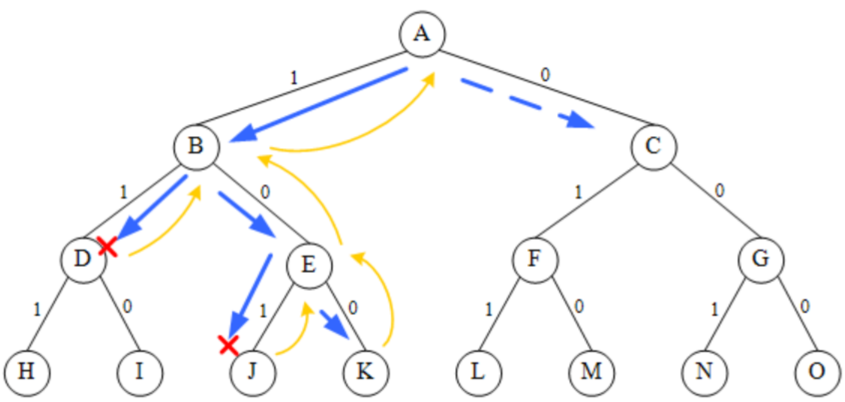
5 非递归回溯

```
void backTrack(int t)
{
    int k = 1; //k表示正在处理的行
    while (k > 0) //回到第一行前一直循环
    {
        x[k] += 1; //尝试将第k行的皇后移动到下一列
        //放在n列范围内&&放置不合法时，尝试下一列
        while ((x[k] <= n) && !place(k)) {x[k] += 1;}
        if (x[k] <= n)  //放在n列范围内
        {
            if (k == n) {sum++;} //已放n行，来到了叶节点了，可行方案数+1
            else {k++;x[k] = 0;} //不足n行，放下一行，一行又从第0列的下列开始试放
        }
        else {k--;} //第k行无法放置，则回溯，重新放置上一行
    }
}
```

4. 0-1背包问题

4.1. 算法描述

1 解空间：用子集树表示，每一层代表一个物品，放入左转为1，不放入右转为2



2 可行性约束函数：$\sum_{i=1}^{n} w_i x_i \leq c_1$

3 上界约束：r为剩余物品价值总和，cp为当前最优价值，当$cp + r \leq bestp$时剪去左子树

4.2. 回溯算法$O(n2^n)$

1 变量

```
struct Object
{
 int ID;  // 物品编号
 float d;  // 单位重量价值
};
int n;      // 物品数
int *w;     // 物品重量数组
int *p;     // 物品价值数组
int cw;     // 当前重量
int cp;     // 当前价值
int bestp; // 当前最优价值
int c;      // 背包容量
```

2 计算到了当前节点，能最终装入的物品价值的最大值，以确定是否继续探索当前分支

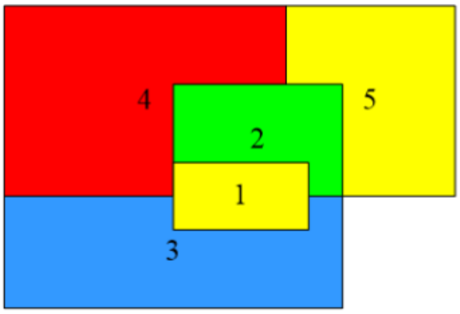
```
int Bound(int i) //考察第i件物品
{
 int cleft = c - cw;  // 剩余的背包容量
 int b = cp;  // 当前价值
 while (i <= n && w[i] <= cleft) //检查剩余的物品是否可以完整放入背包
 //是的话，就将物品全部放入，更新剩余容量和当前价值，然后i++考察下一个物品
 {
     cleft -= w[i];
     b += p[i];
     i++;
 }
 //如果循环结束后，还有物品未放进背包，就取一定比例的物品放入
 if (i <= n) {b += p[i] * cleft / w[i];}
 return b;  // 返回上界(理论上能装入的最大价值)
}
```

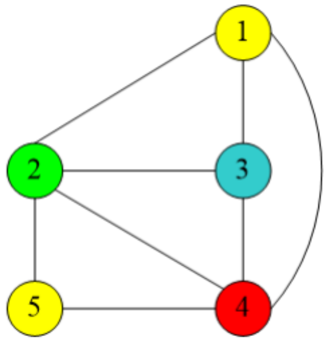
3 递归回溯

```
void Backtrack(int i)
{
 if (i > n) {bestp = cp;return;}
 if (cw + w[i] <= c)//进入左子树
 {
     cw += w[i];
     cp += p[i];
     Backtrack(i + 1);
     cw -= w[i];
     cp -= p[i];
 }
 if (Bound(i + 1) > bestp)//剪掉左子树，进入右子树
 {
     Backtrack(i + 1);
 }
}
```

5. 图的m着色问题

5.1. 问题描述





1 给定无向连通图G，和m种颜色，每个顶点一种颜色，相邻两顶点颜色各不相同

2 图的色数：最少需要多少种颜色，就能使相邻顶点颜色各不相同

3 可着色优化问题：求m的最小值问题

5.2. 算法设计

1 用邻接矩阵A，表示无向连通图

2 解向量：$(x_1, x_2, \ldots, x_n)$，其中x_i表示顶点i所表示的颜色(颜色种类编号为值)

3 可行性约束函数：顶点i与已着色的相邻顶点颜色不重复

5.3. 算法实现$O(nm^n)$

1 变量定义

```
int n;    //图的顶点数
int m;    //可用颜色数
int **a;  //图的邻接矩阵
int *x;   //当前解，x[i]表示i结点的颜色
long sum; //当前已找到的可m着色方案数
```

2 检查顶点k的着色是否与其相邻顶点的着色冲突

```
bool Color::OK(int k)
{
 for(int j=1;j<=n;j++)//遍历所有顶点
 {
     //某一结点与k结点相邻(邻接矩阵项为1)，并且颜色相同，则返回false
     if((a[k][j]==1)&&(x[j]==x[k]))
         return false;
 }
 return true;
}
```

3 递归回溯

```
void Color::Backtrack(int t)//考虑子集树的第t层，也就是第t个结点
```

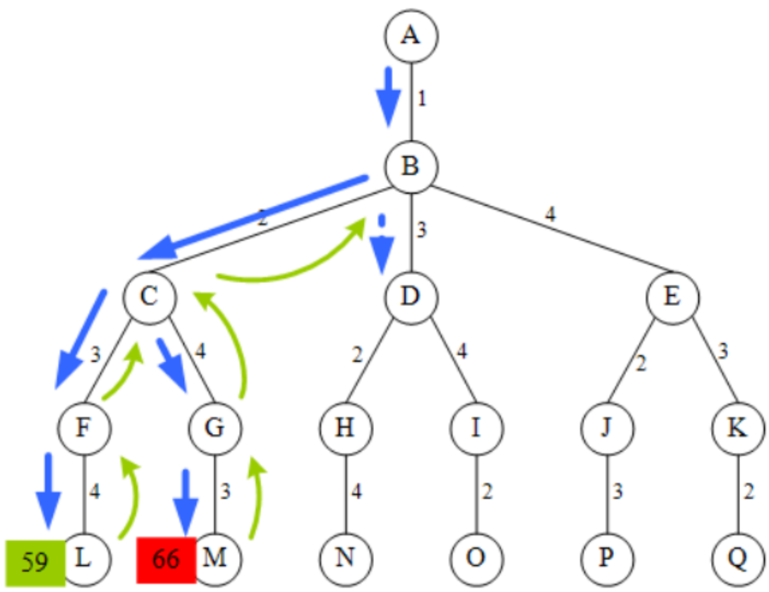
```
{
 if(t>n)//来到叶节点了，则可行解树+1，然后答应出可行解(每个结点的颜色)
 {
     sum++;
     for(int i=1;i<=n;i++)
         cout<<x[i]<<' ';
     cout<<endl;
 }
 else
 {
     for(int i=1;i<=m;i++) //遍历m种颜色，考察每种颜色是否适用当前t结点
     {
         x[t]=i; //对顶点t使用颜色i
         if(OK(t)) Backtrack(t+1);//如果可行，就递归进入下一层考察下一个t+1结点
         x[t]=0; //开始回溯，取消之前的颜色分配，然后回退
     }
 }
}
```

6. TSP问题

6.1. 问题描述

> 给定一个完全图，每边有一个长度，求一条路径，满足
>
> 1 经过所有顶点正好一次
>
> 2 路径封闭
>
> 3 满足以上两个条件，然后路径长度最小

6.2. 算法描述

> 1 有关数据结构
>
> 1. 用子集树表示子空间：现选定1，站在1处选定2/3/4，再站在2处选定3/4(选过1了不能重复)....
>
> 
>

2. `x[]`：表示当前解，对应的排列树中根到叶的路径，将其初始化为最无脑的`x=[1,2,...,n]`
3. `x[1:i]`：从结点`x[1]`开始，依次经过`x[2],x[3],...,x[i-1]`，直到结点`x[i]`的路径总和

2 递归算法设计

1. `i=n`时：当前扩展结点为叶节点的父节点，当$x[n-1] \to x[n]$和$x[n] \to x[1]$两边都存在时，则找到一条回路了，根据此条回路长度更新当前最优解
2. `i<n`时：当前扩展结点在$i-1$层，图中应该要有`x[i-1]`到`x[i]`的边，检查`x[1:i]`费用是是否已经超过最优值
    - 如果已经超过了就没必要继续了，直接剪去子树

6.3. 算法实现：递归回溯

1 定义变量

```
int n;       // 图的顶点数
int x[];     // 当前解，根到叶的路径
int bestx[];// 当前最优解
int a[][];   // 邻接矩阵，a[i][j]=NO_EDGE表示i→j的结点不存在
          // 此外a[i][j]的值表示i→j间的距离
int cc;      // 当前费用
int bestc;   // 当前最优值
```

2 递归回溯

```
void backTrack(int i)
{
 if (i == n) //当来到了叶子结点
 {
     //x[n-1]→x[n]和x[n]→x[1]之间都有边，即能形成闭环
     bool close = a[x[n - 1]][x[n]] != NO_EDGE && a[x[n]][1] != NO_EDGE;
     //计算整个环路的总距离
     int cost = cc + a[x[n - 1]][x[n]] + a[x[n]][1];
     //该环路距离小于当前最优，代表着有更优解
     bool better = cost < bestc;
     //此外还有一种可能就是之前并未找到任何一个环路，所以现在不论什么环路都是最优的
     bool no_edge = (bestc == NO_EDGE);
     //形成环路且(环路为更优解OR是找到的第一个环路)，那么就更新最优解
     if (close && (better || no_edge))
     {
         //更新最优路径为当前路径
         for (int j = 1; j <= n; j++) {bestx[j] = x[j];}
         //更新最优值，即最短路径
         bestc = cc + a[x[n - 1]][x[n]] + a[x[n]][1];
     }
 }
 else
 {
     for (int j = i; j <= n; j++)
```

```
        {
            bool have_edge = a[x[i - 1]][x[j]] != NO_EDGE;//i-1点到j点有边连接
            bool better = cc + a[x[i - 1]][x[j]];//当前长度+该新边长>最优解
            //执行递归&回溯，为什么要交换？待弄清楚
            if (have_edge&&(better||bestc == NO_EDGE))
            {
                swap(x[i], x[j]);
                cc += a[x[i - 1]][x[i]];
                backTrack(i + 1);
                cc -= a[x[i - 1]][x[i]];
                swap(x[i], x[j]);
            }
        }
  }
}
```