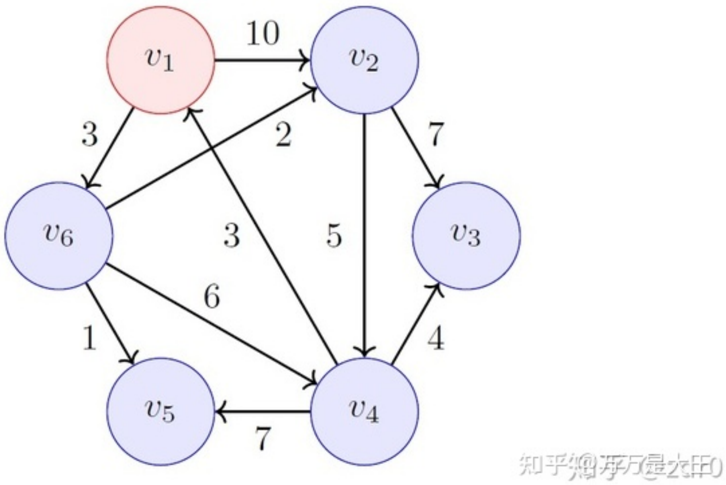
Dijkstra 算法



步骤	S	v2	v3	v4	v5	v6
1	v1	10	∞	∞	∞	3
2	v1——v6	5	∞	9	4	
3	v1——v6——v5	∞	∞	∞		
4	v1——v6——v2		12	10		
5	v1——v6——v4		13			
6	v1——v6——v2——v3					

【代码】

时间复杂度：O(V^2+E), V是顶点， E是边。

```
vector<int> Dijkstra(vector<vector<int>> matrix,int s) {    //matrix邻接矩阵，s源点
    int n = matrix.size();          //节点总数
    vector<int> dis(n,INF);         //distance记录最短路径长度，初始值无穷大（自定义常量）
    vector<int> par(n,-1);          //parent记录路径，初始值-1
    vector<int> vis(n,0);           //visited记录是否访问过，初始值为0
    dis[s] = 0;                             //设源点的距离为0
    int k = s;                              //k变量是当前访问的节点，初始为源点
    for (int i = 0; i < n; i++) {           //一次计算一个节点，循环n次
        vis[k] = 1;                         //将当前访问中的节点设为已访问过
        for (int j = 0; j < n; j++) {       //循环与当前节点相连的节点，更新距离值
            if (vis[j] == 0 && dis[k] + matrix[k][j] < dis[j]) {     //如果节点j没有被访问过且经过k再到j的距离小于直接到j的距离时
                dis[j] = dis[k] + matrix[k][j];                         //更新到达j点的距离
                par[j] = k;                                             //更新路径
}}
        k = 0;       //k清零
        for (int j = 0; j < n; j++) {       //扫描距离信息找到所有未选节点中距离最小的那个节点
            if (vis[j] == 0) {              //判断是否访问过
```

```
                if (dis[j] < dis[k] || k == 0) {//k==0证明第一次进入循环，k被赋值为第一次找到的未访问节点，或者找到距离最小的节点
                    k = j;                          //最后得到距离最小节点进入下一次循环
}}}       return par;}
```

Horspool 算法

Horspool算法是Boyer-Moore算法的简化版本，这也是一个空间换时间的典型例子。算法把模式P和文本T的开头字符对齐，从模式的最后一个字符开始比较，如果尝试比较失败了，它把模式向后移。每次尝试过程中比较是从右到左的。

移动距离的规则简化了很多，具体如下：

$$t(c)=\begin{cases}\text{模式的长度}m & \text{（如果c不包含在模式的前m-1个字符中）}\\ \text{模式前}m-1\text{个字符中最右边的}c\text{到模式最后一个字符的距离} & \text{（其他情况下）}\end{cases}$$

行1： 此时 c 是 A，最右开始，开局就匹配失败，c 没有出现在 BHELL 中，因此移动 6 位。

行2：此时 c 是 O，可见 LLO 匹配，但 A 和 E 匹配失败，然而 BH 没有 c，那么也移动 6 位。

							c						c						c		
...		B	H	E	L	L	A	B	H	A	L	L	O	B	H	E	L	L	O		...
1		B	H	E	L	L	O														
2								B	H	E	L	L	O								
3														B	H	E	L	L	O		

行1：此时 c 是 L，开局匹配失败，但 OHELL 中有 L，那么移动距离为 5 - 4 = 1 位

行2：此时 c 是 O，LLO 匹配，但 A 和 E 匹配失败，不过 OH 含有 c，移动 5 - 0 = 5 位

行3：此时 c 是 L，开局匹配失败，但 OHELL 中有 c，移动 5 - 4 = 1 位

行4：此时 c 是 O，LLO匹配成功，A 和 E 匹配失败，但 OH 有 c，移动 5 - 0 = 5 位。

							c	c					c	c					c		
...		_	B	H	A	L	L	O	B	H	A	L	L	O	H	E	L	L	O		...
1		O	H	E	L	L	O														
2			O	H	E	L	L	O													
3								O	H	E	L	L	O								
4									O	H	E	L	L	O							
5														O	H	E	L	L	O		

这个表也就是说除了搜索词的字母移动距离是 t[c] = **min**(搜索词长度 - 字母 c 位置)，其他都是移动搜索词长度的距离。

字符 c	A	B	C	D	E	F	....	R	...	Z	_
移动距离 $t(c)$	4	2	6	6	1	6	6	3	6	6	6

在特定文本中的实际查找是像下面这样的：

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R             B A R B E R
        B A R B E R         B A R B E R
          B A R B E R           B A R B E R
```

【代码】

```
int Horspool(vector<char> & T,vector<char> & P)
{
    int n = T.size();
    int m = P.size();
    vector<int> table(96,m);//以字母表中可打印字符为索引的数组
    for(int i = 0;i < m - 1;i++)
    {
        table[P[i] - 32] = m - 1 - i;//模式串中每个字符的移动距离，从左至右扫描模式，相同字符的最后一次改写恰好是该字符在模式串的最右边
    }
    int i = m - 1;
    while(i <= n - 1)
    {
        int k = 0;
        while(k <= m - 1 && P[m - 1 - k] == T[i - k])  k++;
        if(k == m)  return i - m + 1;//匹配成功，返回索引
        else  i += table[T[i] - 32];//模式串向右移动
    }   return -1;//匹配失败
}
```

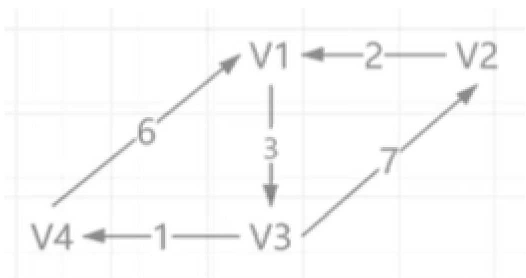
Floyd 算法

执行n次迪杰斯特拉算法，这样就可以求出每一对顶点间的最短路径。

核心原理是：对于图中每一个中间顶点 k，令任意顶点 i,j，都计算 i 到 j 的最短路径和 i 到 k + k 到 j 的路径，比较两者最小值并更新 i 到 j 的最短路径：

$$d[i][j] = min(d[i][j], d[i][k] + d[k][j])$$

【应试】



建立邻接矩阵 D0，首先选择 V1，完成 D0 -> D1，将 D0 中 V1 的行列以及对角线（永远都是 0 ）都原封不动转移到 D1，然后将此时 D1 中含有的 ∞ 对应的行列从 D0 也原封不动的搬过来，如下图：

D0

	V1	V2	V3	V4
V1	0_{11}	∞_{12}	3_{13}	∞_{14}
V2	2_{21}	0_{22}	∞_{23}	∞_{24}
V3	∞_{31}	7_{32}	0_{33}	1_{34}
V4	6_{41}	∞_{42}	∞_{43}	0_{44}

D1

	V1	V2	V3	V4
V1	0_{11}	∞_{12}	3_{13}	∞_{14}
V2	2_{21}	0_{22}		∞_{24}
V3	∞_{31}	7_{32}	0_{33}	1_{34}
V4	6_{41}	∞_{42}		0_{44}

空格处规则：在空格处做十字线，若字相交处值相加 小于 前驱 D 值，则更新为最小值，同时，最短路径更新为两个红字最短路径的并。

D1

	V1	V2	V3	V4
V1	0_{11}	∞_{12}	3_{13}	∞_{14}
V2	2_{21}	0_{22}	5_{213}	∞_{24}
V3	∞_{31}	7_{32}	0_{33}	1_{34}
V4	6_{41}	∞_{42}	9_{413}	0_{44}

D2

	V1	V2	V3	V4
V1	0_{11}	∞_{12}	3_{13}	∞_{14}
V2	2_{21}	0_{22}	5_{213}	∞_{24}
V3	9_{321}	7_{32}	0_{33}	1_{34}
V4	6_{41}	∞_{42}	9_{413}	0_{44}

这样一直到 D4，此时即可得两点间最短路径。

【代码】

执行的时间复杂度为O(n3)。

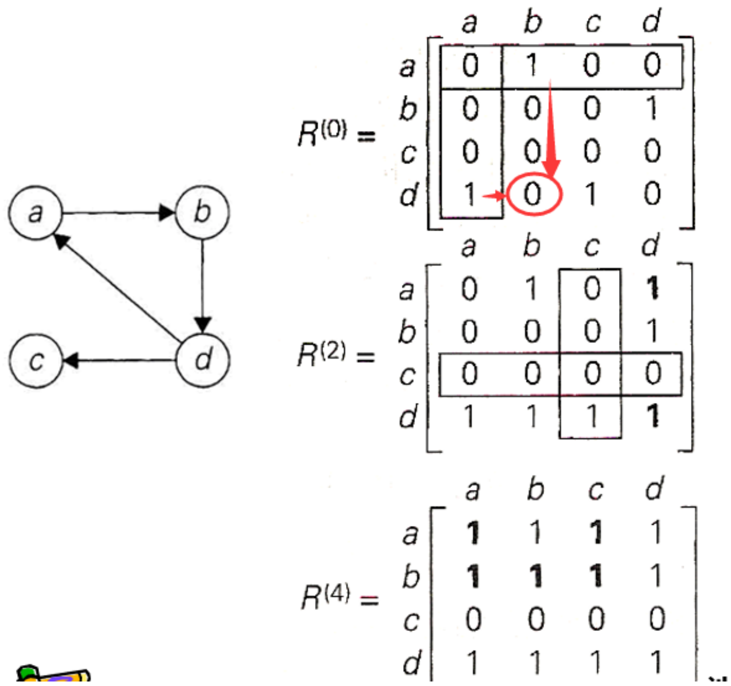
```
void ShortestPath_FLOYD(MGraph G, PathMatrix P[], DistancMatrix &D) {
    // 算法7.16
    // 用Floyd算法求有向网G中各对顶点v和w之间的最短路径P[v][w]及其
    // 带权长度D[v][w]。若P[v][w][u]为TRUE，则u是从v到w当前求得最
    // 短路径上的顶点。
    int v,w,u,i;
    for (v=0; v<G.vexnum; ++v)          // 各对结点之间初始已知路径及距离
        for (w=0; w<G.vexnum; ++w) {
            D[v][w] = G.arcs[v][w].adj;
            for (u=0; u<G.vexnum; ++u)  P[v][w][u] = FALSE;
            if (D[v][w] < INFINITY) {      // 从v到w有直接路径
                P[v][w][v] = P[v][w][w] = TRUE;
            }//if
        }//for
    for (u=0; u<G.vexnum; ++u)
        for (v=0; v<G.vexnum; ++v)
            for (w=0; w<G.vexnum; ++w)
                if (D[v][u]+D[u][w] < D[v][w]) {  // 从v经u到w的一条路径更短
                    D[v][w] = D[v][u]+D[u][w];
                    for (i=0; i<G.vexnum; ++i)
                        P[v][w][i] =(P[v][u][i] || P[u][w][i]);
                }//if
} // ShortestPath_FLOYD
```

Warshall 算法

传递闭包：就是 R 能构成传递关系的最小序偶集合。关系是序偶的集合，R 关系里边的元素就是序偶。

【应试】

同 Floyd 算法，首先建立 0-1 邻接矩阵 D0，然后选择 V1 结点的行列原封不动挪到 D1，然后，对于其他空格，如果其十字和 V1 结点的行列相交处都是 1，那么就是 1，否则就**保持原状**。这样依次到 D4，此时 R 的传递闭包是 D4 对应所有序偶对的集合关系。



【代码】

传统的求传递闭包算法，通过矩阵点乘的来迭代的方式得到传递关系闭包的集合。矩阵点乘的算法复杂度为O(n^3)，迭代次数为n-1次(得到R^n为结果),算法复杂度为O(n^4)。对于此类算法，为了找到某一关系(a,b)，要把其他的元素作为中间元素来判断是否存在传递关系。简言之，针对所求的关系，去遍历中间元素的关系去判断。

例如：a,b,c,d,e属于A集合，R为A的关系集合，为了找到(a,b)，需要把c,d,e作为中间元素

```
Matrix getTranstiveClosure(Matrix matrixA,int matrix_n){
    for(int num=1;num<matrix_n;num++){//R^n,迭代n-1次按照C语言，0号位置为数组第一位
        for(int i=0;i<matrix_n;i++){//开始矩阵布尔积运算
            for(int j=0;j<matrix_n;j++){
                for(int k=0;k<matrix_n;k++){
                    if(matrixA.a[i][j]==0)//优化或运算
                    matrixA.a[i][j]=(matrixA.a[i][k]&matrixA.a[k][j]);}}}}
    return matrixA;
}//M=getTranstiveClosure(matrixA);来得到传递闭包矩阵
```

Warshall算法的算法复杂度为O(n^3)，其巧妙之处就在于无需矩阵的迭代，通过固定中间元素来进行判断关系，并对中间元素逐次遍历，并对传递关系进行迭代，需要遍历的中间元素为n个，所求矩阵遍历操作为n^2，在n规模足够大的时候warshall算法能体现出优越性。对于同样矩阵上的一个所求关系的元素，一般算法需要的操作为n*(n-1)，而warshall算法仅仅需要n次。

例如：a,b,c,d,e∈A，关系集合R，若(a,b)(b,c)(c,d)(d,e)∈R，先把a作为中间元素。

```
Matrix Warshall(Matrix matrixB,int matrix_n){
    for(int k=0;k<matrix_n;k++){//K值选择中间量的元素来补全传递关系的路线
        for(int i=0;i<matrix_n;i++){//遍历新矩阵
            for(int j=0;j<matrix_n;j++){
                if(matrixB.a[i][j]==0){//计算 与
                    matrixB.a[i][j]=matrixB.a[i][k]&&matrixB.a[k][j];
                }//如(a→b)&(b→c)⟶(a→c)的关系
            }}}}
```

LU 分解

我们定义 Ax = b 中，A = LU，其中，L 为初等变换矩阵的逆（单位下三角矩阵），U 为高斯消元后的 A（上三角矩阵）。

即： Pn....P1A = U，P是一系列初等行变换。定义 Pn...P1 为 L-1，故 L-1A = U，然后 A = LU。

那么，Ax = b，可转化为 LUx = b，因此，设 Ux = y，解 Ly = b，然后根据 y，解 Ux = y ，求得 x。

【例题 1】

$$A=\begin{bmatrix}2 & -1 & 1\\ 4 & 1 & -1\\ 1 & 1 & 1\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 1/2 & 1/2 & 1\end{bmatrix}\times\begin{bmatrix}2 & -1 & 1\\ 0 & 3 & -3\\ 0 & 0 & 2\end{bmatrix}=LU$$

【例题 2】：（A如下图，b 是 [1,2,3,4]）

$$A=\begin{bmatrix}1&1&1&1\\1&2&1&2\\1&2&2&3\\1&2&2&4\end{bmatrix}\Rightarrow\begin{bmatrix}1&1&1&1\\0&1&0&1\\0&1&1&2\\0&1&1&3\end{bmatrix}\Rightarrow\begin{bmatrix}1&1&1&1\\0&1&0&1\\0&0&1&1\\0&0&1&2\end{bmatrix}\Rightarrow\begin{bmatrix}1&1&1&1\\0&1&0&1\\0&0&1&1\\0&0&0&1\end{bmatrix}\ U$$

$$L^{-1}\text{为}\left(\begin{bmatrix}1&0&0&0\\-1&1&0&0\\-1&0&1&0\\-1&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&-1&1&0\\0&-1&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&-1&1\end{bmatrix}\right)^{-1}$$

$$L=\begin{bmatrix}1&0&0&0\\1&1&0&0\\1&1&1&0\\1&1&1&1\end{bmatrix}\Rightarrow\begin{bmatrix}1&0&0&0\\1&1&0&0\\1&1&1&0\\1&1&1&1\end{bmatrix}\ b=\begin{bmatrix}1\\2\\3\\4\end{bmatrix}\ \text{先求Ly=b}\Rightarrow y=\begin{bmatrix}1\\1\\1\\1\end{bmatrix}$$

$$\Rightarrow\begin{bmatrix}1&1&1&1\\0&1&0&1\\0&0&1&1\\0&0&0&1\end{bmatrix}x=\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\Rightarrow x=[0,0,0,1]\ \text{再求Ux=y}$$

因为有三重嵌套的循环，所有LU分解的复杂度是 $O(n^3)$ ，其中 加法操作$\sim \frac{1}{3}n^3$ ，乘法操作 $\sim \frac{1}{3}n^3$ ，一共 $\sim \frac{2}{3}n^3$

总体来看，使用LU分解求解线性系统的计算量：

- 第一步：LU分解： $\sim \frac{2}{3}n^3$ （计算量最大的部分）
- 第二步：求解 $Ly = b$： $\sim n^2$
- 第三步：求解 $Ux = y$： $\sim n^2$

```
算法 LU Decomposition
//输入矩阵A[1…n,1…n],并且A的所有顺序主子式都不为0
//输出下三角矩阵L和上三角矩阵U
L←n阶zero矩阵  //初始化一个n阶zero矩阵
U←n阶zero矩阵
for i ← 1 to n do
    L[i][i] ← 1 //L的主对角线元素全为1
        if i==0 do  //分别利用A的第一行元素第一列元素，确定L的第一列元素和U的第一行元素
        U[0][0] ← A[0][0]
        for j ← 1 to n do
            U[0][j] ←A[0][j]
            L[j][0] ←A[j][0]/U[0][0]
    else
        for j ← i to n do  //计算矩阵U
            for k ← 1 to i-1 do
                temp ←L[i][k] * U[k][j]
            U[i][j] ←A[i][j]-temp
        for j ← i+1 to n do  //计算矩阵L
            for k ← 1 to i-1 do
                temp ←L[j][k] * U[k][i]
            L[j][i] ← (A[j][i] - temp)/U[i][i]
```

背包问题

【0-1背包问题】

背包最大重量为 5。问背包能背的物品最大价值是多少?

- **不放物品i**：由dp[i - 1][j]推出，即背包容量为j，里面不放物品i的最大价值，此时dp[i][j]就是dp[i - 1][j]。(其实就是当物品i的重量大于背包j的重量时，物品i无法放进背包中，所以被背包内的价值依然和前面相同。)
- **放物品i**：由dp[i - 1][j - weight[i]]推出，dp[i - 1][j - weight[i]] 为背包容量为j - weight[i]的时候不放物品i的最大价值，那么dp[i - 1][j - weight[i]] + value[i] （物品i的价值），就是背包放物品i得到的最大价值

DP 转移方程：dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]).

第 0 列，即背包容量为 0 时，背包物品的最大价值，自然就是 0.

第 1 行，即背包容量为 j 时，背包物品的最大价值。当 j >= weight[0]时，dp[0][j] 应该是value[0]，因为背包容量放足够放编号 1 物品，否则就是 0，因为无法放物品。

		承重量 j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	37

比方说：dp[3][4] = max（dp[2][4] , dp[2][4-3]+20)

【代码】

- 二维常规

```
void zero_one_bag_problem() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    int bagweight = 4;
    // 二维数组
    vector<vector<int>> dp(weight.size(), vector<int>(bagweight + 1, 0));
    // 初始化
    for (int j = weight[0]; j <= bagweight; j++) {
        dp[0][j] = value[0];
    }
    // weight数组的大小 就是物品个数
    for(int i = 1; i < weight.size(); i++) { // 遍历物品
        for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
            if (j < weight[i]) dp[i][j] = dp[i - 1][j];
            else dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);

        }
    }
    cout << dp[weight.size() - 1][bagweight] << endl;
}
```

- 滑动压缩

```
void zero_one_bag_problem() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    int bagWeight = 4;

    // 初始化
    vector<int> dp(bagWeight + 1, 0);
    for(int i = 0; i < weight.size(); i++) { // 遍历物品
        for(int j = bagWeight; j >= weight[i]; j--) { // 遍历背包容量
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
        }
    }
    cout << dp[bagWeight] << endl;
}
```

- 零钱兑换

生成行数为 N、列数为 aim+1 的矩阵 dp，dp[i][j]的含义是在使用 arr[0..i]货币的情况下，组成钱数 j 有 多少种方法。dp[i][j]的值求法如下：

1．对于矩阵 dp 第一列的值 dp[..][0]，表示组成钱数为 0 的方法数，很明显是 1 种，也就是 不使用任何货币。所以 dp 第一列的值统一设置为 1。

2．对于矩阵 dp 第一行的值 dp[0][..]，表示只能使用 arr[0]这一种货币的情况下，组成钱的 方法数，比如，arr[0]==5 时，能组成的钱数只有 0，5，10，15，…。所以，令 dp[0][k*arr[0]]=1 （0≤k×arr[0]≤aim，k 为非负整数）。

3．除第一行和第一列的其他位置，记为位置(i,j)。dp[i][j]的值是以下几个值的累加。

完全不用 arr[i]货币，只使用 arr[0..i-1]货币时，方法数为 dp[i-1][j]。 - 用 1 张 arr[i]货币，剩下的钱用 arr[0..i-1]货币组成时，方法数为 dp[i-1][j-arr[i]]。 - 用 2 张 arr[i]货币，剩下的钱用 arr[0..i-1]货币组成时，方法数为 dp[i-1][j-2*arr[i]]。 k 张 arr[i]货币，剩下的钱用 arr[0..i-1]货币组成时，方法数为 dp[i-1][j-karr[i]]。 j-karr[i]>=0，k 为非负整数。

4．最终 dp[N-1][aim]的值就是最终结果。

```
public int coins3(int[] arr, int aim) {
    if (arr == null || arr.length == 0 || aim < 0) {
        return 0;
    }
        int[][] dp = new int[arr.length][aim + 1];
        for (int i = 0; i < arr.length; i++) {
                dp[i][0] = 1;
        }
        for (int j = 1; arr[0] * j <= aim; j++) {
                dp[0][arr[0] * j] = 1;
        }
        int num = 0;
        for (int i = 1; i < arr.length; i++) {
                for (int j = 1; j <= aim; j++) {
                    num = 0;
                    for (int k = 0; j - arr[i] * k >= 0; k++) {
                            num += dp[i - 1][j - arr[i] * k];
                    }
                dp[i][j] = num;
            }
        }
        return dp[arr.length - 1][aim];
}
```

- 完全背包

有N件物品和一个最多能背重量为W的背包。第i件物品的重量是weight[i]，得到的价值是value[i] 。**每件物品都有无限个（也就是可以放入背包多次）**，求解将哪些物品装入背包里物品价值总和最大。

```
// 先遍历背包，再遍历物品
void test_CompletePack() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    int bagWeight = 4;

    vector<int> dp(bagWeight + 1, 0);

    for(int j = 0; j <= bagWeight; j++) { // 遍历背包容量
        for(int i = 0; i < weight.size(); i++) { // 遍历物品
            if (j - weight[i] >= 0) dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
        }
    }
    cout << dp[bagWeight] << endl;
```

```
}
```

最优二叉搜索树

如果一棵最优二叉查找树T有一棵包含关键字ki,..,kj的子树T'，那么这可子树T'对于关键字Ki,...,kj和虚拟键di-1,...dj的子问题也必定是最优的。

根据最优子结构，寻找最优解：

给定关键字ki,...,kj，假设kr(i<=r<=j)是包含这些键的一棵最优子树的根。其左子树包含关键字ki,...,kr-1和虚拟键di-1,...,dr-1，右子树包含关键字kr+1,...,kj和虚拟键dr,...dj。我们检查所有的候选根kr,就保证可以找到一棵最优二叉查找树。

递归解：

定义e[i,j]为包含关键字ki,...,kj的最优二叉查找树的期望代价，最终要计算的是e[1,n]。

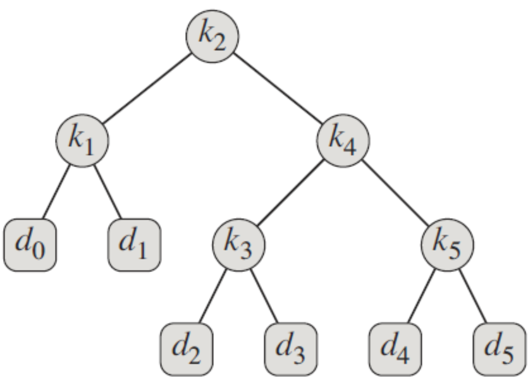
当j = i - 1时，此时子树中只有虚拟键，期望搜索代价为e[i,i - 1] = qi-1.

当j >= i时，需要从ki,...,kj中选择一个根kr，然后分别构造其左子树和右子树。下面需要计算以kr为根的树的期望搜索代价。然后选择导致最小期望搜索代价的kr做根。

现在需要考虑的是，当一棵树成为一个节点的子树时，期望搜索代价怎么变化？子树中每个节点深度都增加1.期望搜索代价增加量为子树中所有概率的总和。

已知 命中结点 p 和失败结点 q 的概率分布，求出最优二叉搜索树的 BST。

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10



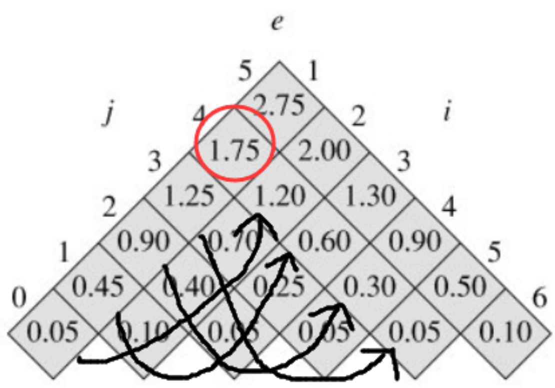
node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

建立dp主表 e，i 和 j 的个数就是 q 的个数，其中 j 从左下 0 开始，i 从右上 1 开始，最后一行先把 q 填进去。

然后按照从左到右，从下到上的原则，结合 dp 表 w。计算规则：

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 , \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j . \end{cases}$$

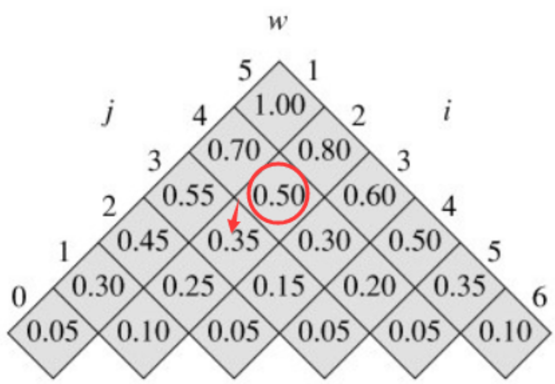
比方说，e[1][4] = 所有线的 min(线两边的数相加 + w[1][4])



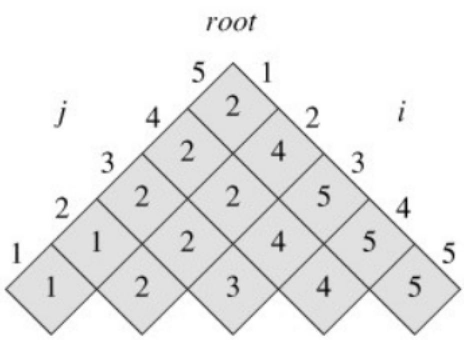
dp 表 w 和 e 一样，首先最后一行把 q 都填进去。然后，从左下到右上，计算方式是这样的：

w[i][j] = w[i][j - 1] + p[j] + q[j]

比方说 w[2][4] = w[2][3] + p[4] + q[4] = 0.35 + 0.10 + 0.15 = 0.5



然后建立根表



不过 PPT 里的更简单，只需要遵循

$$\text{当}1 \le i \le j \le n\text{时，}\quad C[i,j] = \min_{i \le k \le j}\{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s$$

即可，并不需要建立 w 表。由于没有失败概率，e 表的最后一行皆为 0 ，倒数第二行填充成功概率。根表就是 e 表求解min值时对应的子树分割点 k。

【代码】

```
const int MaxVal = 9999;
const int n = 5;
//搜索到根节点和虚拟键的概率
double p[n + 1] = {-1,0.15,0.1,0.05,0.1,0.2};
double q[n + 1] = {0.05,0.1,0.05,0.05,0.05,0.1};
```

```
int root[n + 1][n + 1];//记录根节点
double w[n + 2][n + 2];//子树概率总和
double e[n + 2][n + 2];//子树期望代价

void optimalBST(double *p,double *q,int n)
{
    //初始化只包括虚拟键的子树
    for (int i = 1;i <= n + 1;++i)
    {
        w[i][i - 1] = q[i - 1];
        e[i][i - 1] = q[i - 1];
    }
    //由下到上，由左到右逐步计算
    for (int len = 1;len <= n;++len)
    {
        for (int i = 1;i <= n - len + 1;++i)
        {
            int j = i + len - 1;
            e[i][j] = MaxVal;
            w[i][j] = w[i][j - 1] + p[j] + q[j];
            //求取最小代价的子树的根
            for (int k = i;k <= j;++k)
            {
                double temp = e[i][k - 1] + e[k + 1][j] + w[i][j];
                if (temp < e[i][j])
                {
                    e[i][j] = temp;
                    root[i][j] = k;
                }
            }}}}

//输出最优二叉查找树所有子树的根
void printRoot()
{
    cout << "各子树的根：" << endl;
    for (int i = 1;i <= n;++i)
    {
        for (int j = 1;j <= n;++j)
        {  cout << root[i][j] << " ";  }
        cout << endl;
    }
    cout << endl;
}

//打印最优二叉查找树的结构
//打印出[i,j]子树，它是根r的左子树和右子树
void printOptimalBST(int i,int j,int r)
{
    int rootChild = root[i][j];//子树根节点
    if (rootChild == root[1][n])
    {
        //输出整棵树的根
        cout << "k" << rootChild << "是根" << endl;
        printOptimalBST(i,rootChild - 1,rootChild);
        printOptimalBST(rootChild + 1,j,rootChild);
        return;
    }
```

```
    if (j < i - 1)
    {  return;   }
    else if (j == i - 1)//遇到虚拟键
    {
        if (j < r)
        {   cout << "d" << j << "是" << "k" << r << "的左孩子" << endl;  }
        else   cout << "d" << j << "是" << "k" << r << "的右孩子" << endl;
        return;
    }else//遇到内部结点
    {
        if (rootChild < r)
        {   cout << "k" << rootChild << "是" << "k" << r << "的左孩子" << endl;  }
        else
            cout << "k" << rootChild << "是" << "k" << r << "的右孩子" << endl;  }
    printOptimalBST(i,rootChild - 1,rootChild);
    printOptimalBST(rootChild + 1,j,rootChild);
}
int main()
{
    optimalBST(p,q,n);
    printRoot();
    cout << "最优二叉树结构：" << endl;
    printOptimalBST(1,n,-1);
}
```

十大排序总结

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

- 插入排序

  最好情况: 即该数据已经有序，我们不需要移动任何元素。于是我们需要从头到尾遍历整个数组中的元素O(n).

最坏情况: 即数组中的元素刚好是倒序的，每次插入时都需要和已排序区间中所有元素进行比较，并移动元素。因此最坏情况下的时间复杂度是O(n^2).

平均时间复杂度:类似我们在一个数组中插入一个元素那样，该算法的平均时间复杂度为O(n^2).

- 选择排序

  最好情况，最坏情况：都需要遍历未排序区间，找到最小元素。所以都为O(n^2).因此，平均复杂度也为O(n^2).

- 冒泡排序

  最好情况:我们只需要进行一次冒泡操作，没有任何元素发生交换，此时就可以结束程序，所以最好情况时间复杂度是O(n).

  最坏情况: 要排序的数据完全倒序排列的，我们需要进行n次冒泡操作，每次冒泡时间复杂度为O(n),所以最坏情况时间复杂度为O(n^2)。

  平均复杂度：O(n^2)

- 归并排序

  归并排序的递推公式为T(n)=2*T(n/2)+n

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right)+n \\
&= 2\left[2T\left(\frac{n}{2^2}\right)+\frac{n}{2}\right]+n = 2^2T\left(\frac{n}{2^2}\right)+2n \\
&= 2^2\left[2T\left(\frac{n}{2^3}\right)+\frac{n}{2^2}\right]+2n = 2^3T\left(\frac{n}{2^3}\right)+3n \\
&= \cdots \\
&= 2^kT\left(\frac{n}{2^k}\right)+kn \\
&\frac{n}{2^k}=1 \Rightarrow 2^k=n \Rightarrow k=\log_2 n \\
T(1)=1 &\Rightarrow T(n)=n+n\log_2 n = n\log n
\end{aligned}
$$

- 快速排序

  快排的时间复杂度也可以像归并排序那样用递推公式计算出来。如果每次分区都刚好把数组分成两个大小一样的区间，那么它的时间复杂度也为O(nlogn).但是如果遇到最坏情况下，该算法可能退化成O(n^2).

- 计数排序

  计数算法的时间复杂度为O(n+k),由于我们需要分配额外的数组空间，空间复杂度也为O(n+k),即不是原地排序算法.

- 桶排序

  假设我们需要排序的数组元素有n个，同时用m个桶来存储我们的数据。那么平均每个桶的元素个数为k = n/m个.如果在桶内我们使用快速排序，那么时间复杂度为klogk,总的时间复杂度即为nlog(n/m).如果桶的数量接近元素的数量，桶排序的时间复杂度就是O(n) 了。但是如果运气不好，所有的元素都到了一个桶了，那么它的时间复杂度就退化成 O(nlogn) 了。

- 基数排序

  基数排序的时间复杂度是O(k*n),其中n是排序的元素个数，k是元素中最大元素的位数。因此，基数算法也是线性的时间复杂度，但是由于k取决于数字的位数，所以在某些情况下该算法不一定优于O(nlogn).

- 堆排序

  我们已经知道，包含n个元素的完整二叉树的高度为logn.

而当我们使用堆化函数，对某个元素进行维护时，我们需要继续将元素与其左，右子元素进行比较，并将其向下推移，直到其两个子元素均小于其大小。在最坏的情况下，我们需要将元素从根移动到叶子节点，进行多次logn的比较和交换。在建堆阶段，我们对n/2元素执行此操作，因此建堆的最坏情况复杂度为n/2*log(n) ~ nlogn。

在排序步骤中，我们将根元素与最后一个元素交换，并堆放根元素。对于每个元素，这又需要花费logn最长时间，因为我们可能需要将元素从根一直带到最远的叶子上。由于我们重复了n次，因此堆排序步骤也是nlogn。

因此，堆排序在所有情况下，即最好最坏以及平均情况下的时间复杂度均为O(nlogn).