填空

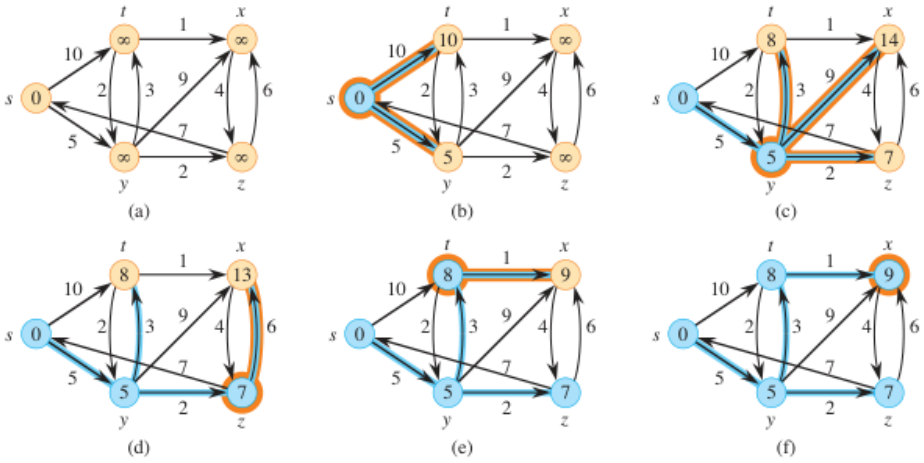
太多了老子不想打了

选择

同理，鬼知道考什么

小题可能涉及的考点

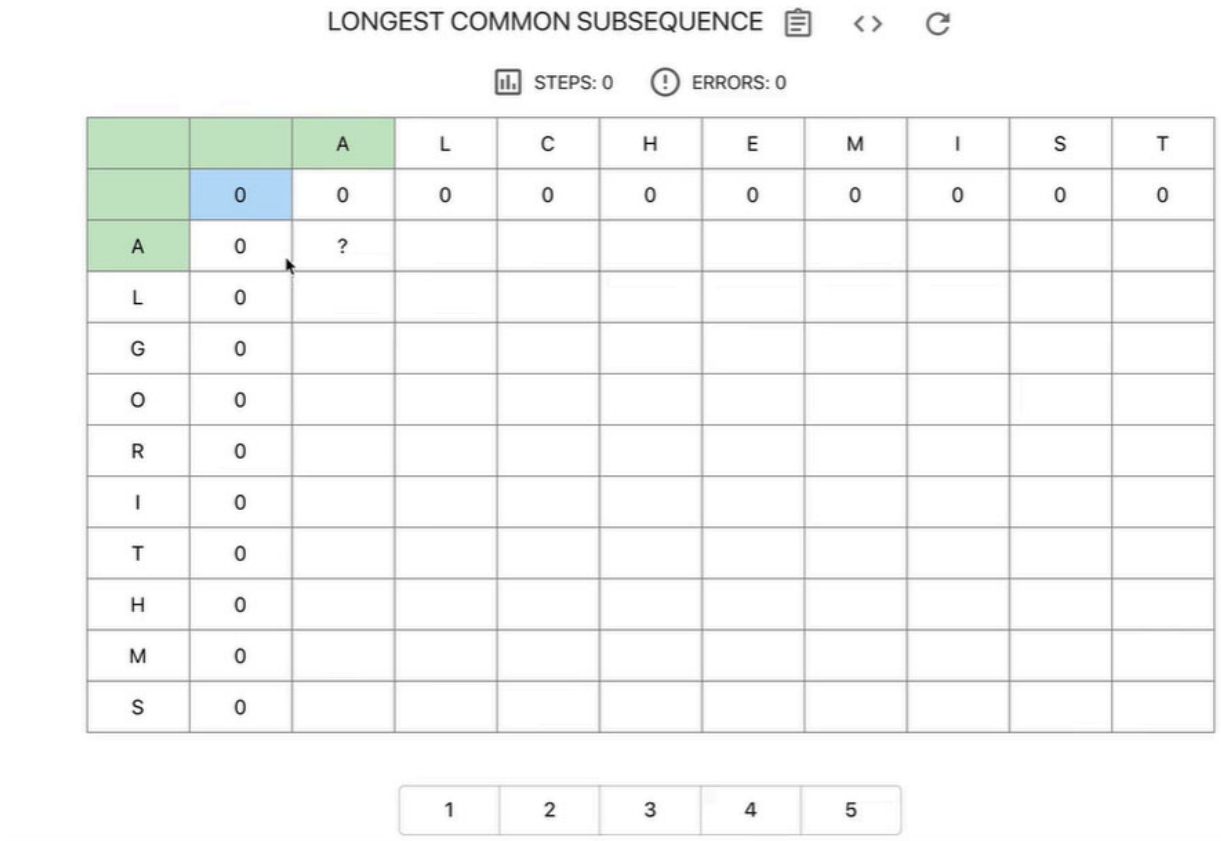
dijkstra



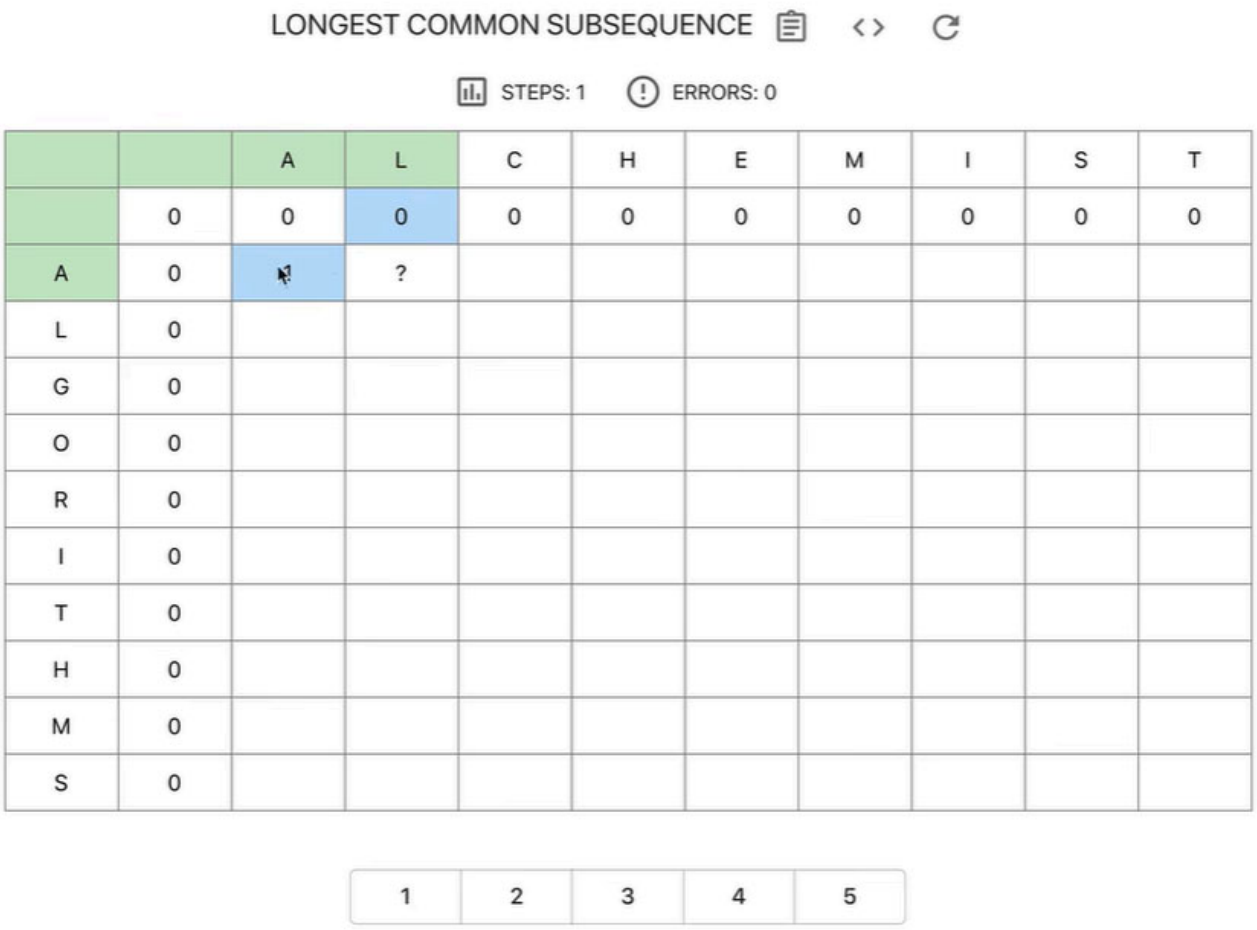
给定点置0，其余置无穷大。从0出发，节点值为出发点加上路径上的权重，找出最小的一个，设置节点的值。后面的都这么干，把所有点连完就好了

最长公共子序列：可以不连续，但是值顺序不能不同

dp表，最后找出表中最大值
值相同 找左上角+1



值不同 在相邻的两个格子里面找较大的




```
if (a === b) {
  table[row][col] = table[row - 1][col - 1] + 1;
} else {
  table[row][col] = Math.max(table[row][col - 1], table[row - 1][col]);
}
```

最长公共字串：要连续

还是建dp表，最后找出表中最大值
值不同置0
值相同找左上角+1，图中标出

LONGEST COMMON SUBSTRING

STEPS: 16 ERRORS: 0

		A	A	C	G	T	T	A	G
	0	0	0	0	0	0	0	0	0
A	0	1	1	0	0	0	0	1	0
C	0	0	0	2	0	0	0	0	0
G	0	?							
T	0								
A	0								

```
if (a === b) {
  table[row][col] = table[row - 1][col - 1] + 1;
} else {
  table[row][col] = 0;
}
```

五种排序的时间复杂度

选择排序：$O(n^2)$

```
/* 选择排序 */
void selectionSort(int nums[], int n) {
    // 外循环：未排序区间为 [i, n-1]
    for (int i = 0; i < n - 1; i++) {
        // 内循环：找到未排序区间内的最小元素
        int k = i;
        for (int j = i + 1; j < n; j++) {
            if (nums[j] < nums[k])
                k = j; // 记录最小元素的索引
        }
        // 将该最小元素与未排序区间的首个元素交换
        int temp = nums[i];
        nums[i] = nums[k];
        nums[k] = temp;
    }
}
```

冒泡排序：$O(n^2)$

```
/* 冒泡排序 */
void bubbleSort(int nums[], int size) {
    // 外循环：未排序区间为 [0, i]
    for (int i = size - 1; i > 0; i--) {
        // 内循环：将未排序区间 [0, i] 中的最大元素交换至该区间的最右端
        for (int j = 0; j < i; j++) {
            if (nums[j] > nums[j + 1]) {
                int temp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = temp;
            }
        }
    }
}
```

插入排序：$O(n^2)$，完全有序达到最佳 $O(n)$

```
/* 插入排序 */
void insertionSort(int nums[], int size) {
    // 外循环：已排序区间为 [0, i-1]
    for (int i = 1; i < size; i++) {
        int base = nums[i], j = i - 1;
        // 内循环：将 base 插入到已排序区间 [0, i-1] 中的正确位置
        while (j >= 0 && nums[j] > base) {
            // 将 nums[j] 向右移动一位
            nums[j + 1] = nums[j];
            j--;
        }
        // 将 base 赋值到正确位置
        nums[j + 1] = base;
    }
}
```

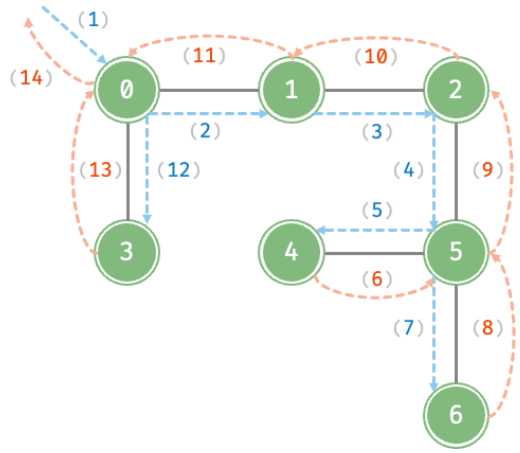
快速排序：$O(nlogn)$，最差 $O(n^2)$

```c
/* 元素交换 */
void swap(int nums[], int i, int j) {
    int tmp = nums[i];
    nums[i] = nums[j];
    nums[j] = tmp;
}

/* 哨兵划分 */
int partition(int nums[], int left, int right) {
    // 以 nums[left] 为基准数
    int i = left, j = right;
    while (i < j) {
        while (i < j && nums[j] >= nums[left]) {
            j--; // 从右向左找首个小于基准数的元素
        }
        while (i < j && nums[i] <= nums[left]) {
            i++; // 从左向右找首个大于基准数的元素
        }
        // 交换这两个元素
        swap(nums, i, j);
    }
    // 将基准数交换至两子数组的分界线
    swap(nums, i, left);
    // 返回基准数的索引
    return i;
}
```

归并排序：$O(nlogn)$

```c
/* 合并左子数组和右子数组 */
void merge(int *nums, int left, int mid, int right) {
    // 左子数组区间为 [left, mid], 右子数组区间为 [mid+1, right]
    // 创建一个临时数组 tmp ，用于存放合并后的结果
    int tmpSize = right - left + 1;
    int *tmp = (int *)malloc(tmpSize * sizeof(int));
    // 初始化左子数组和右子数组的起始索引
    int i = left, j = mid + 1, k = 0;
    // 当左右子数组都还有元素时，进行比较并将较小的元素复制到临时数组中
    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            tmp[k++] = nums[i++];
        } else {
            tmp[k++] = nums[j++];
        }
    }
    // 将左子数组和右子数组的剩余元素复制到临时数组中
    while (i <= mid) {
        tmp[k++] = nums[i++];
    }
    while (j <= right) {
        tmp[k++] = nums[j++];
    }
    // 将临时数组 tmp 中的元素复制回原数组 nums 的对应区间
    for (k = 0; k < tmpSize; ++k) {
        nums[left + k] = tmp[k];
    }
    // 释放内存
    free(tmp);
}

/* 归并排序 */
void mergeSort(int *nums, int left, int right) {
    // 终止条件
    if (left >= right)
        return; // 当子数组长度为 1 时终止递归
    // 划分阶段
    int mid = left + (right - left) / 2;    // 计算中点
    mergeSort(nums, left, mid);      // 递归左子数组
    mergeSort(nums, mid + 1, right); // 递归右子数组
    // 合并阶段
    merge(nums, left, mid, right);
}
```

dfs



图的深度优先遍历（DFS）

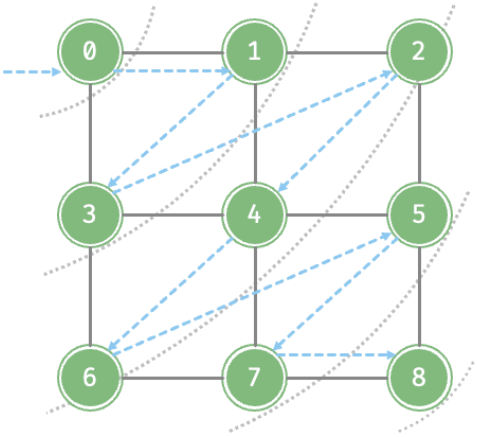
以顶点 0 为起始，
走到头才返回，再走到头才返回，
以此类推 … 直至完成遍历

遍历序列为

0, 1, 2, 5, 4, 6, 3

```c
/* 检查顶点是否已被访问 */
int isVisited(Vertex **res, int size, Vertex *vet) {
    // 遍历查找节点，使用 O(n) 时间
    for (int i = 0; i < size; i++) {
        if (res[i] == vet) {
            return 1;
        }
    }
    return 0;
}

/* 深度优先遍历辅助函数 */
void dfs(GraphAdjList *graph, Vertex **res, int *resSize, Vertex *vet) {
    // 记录访问顶点
    res[(*resSize)++] = vet;
    // 遍历该顶点的所有邻接顶点
    AdjListNode *node = findNode(graph, vet);
    while (node != NULL) {
        // 跳过已被访问的顶点
        if (!isVisited(res, *resSize, node->vertex)) {
            // 递归访问邻接顶点
            dfs(graph, res, resSize, node->vertex);
        }
        node = node->next;
    }
}

/* 深度优先遍历 */
// 使用邻接表来表示图，以便获取指定顶点的所有邻接顶点
void graphDFS(GraphAdjList *graph, Vertex *startVet, Vertex **res, int *resSize) {
    dfs(graph, res, resSize, startVet);
}
```

bfs



图的广度优先遍历（BFS）

以顶点 0 为起始，
由近及远、层层扩张地访问顶点

遍历序列为

0, 1, 3, 2, 4, 6, 5, 7, 8

www.hello-algo.com

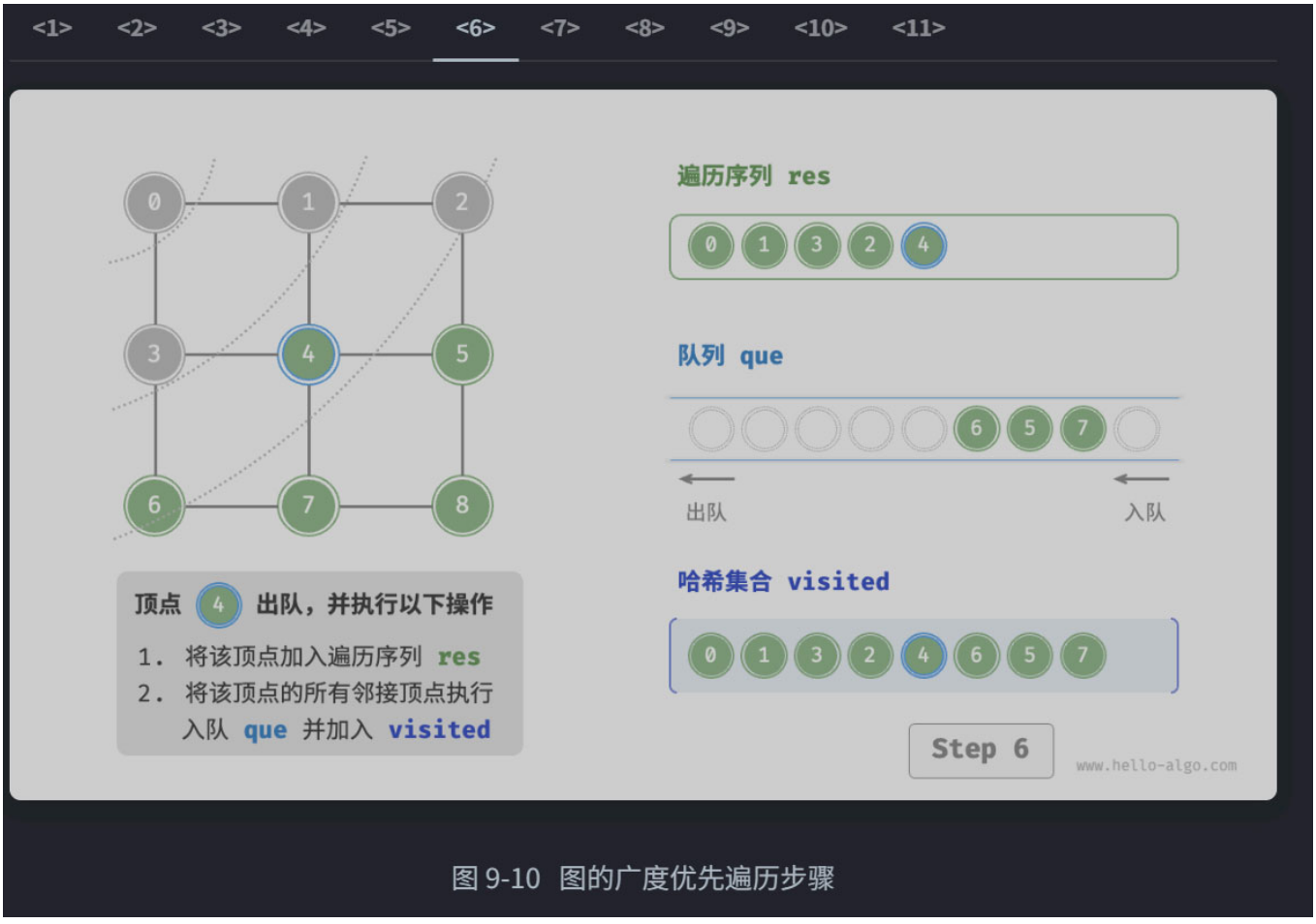


图 9-10 图的广度优先遍历步骤

```c
/* 节点队列结构体 */
typedef struct {
    Vertex *vertices[MAX_SIZE];
    int front, rear, size;
} Queue;

/* 构造函数 */
Queue *newQueue() {
    Queue *q = (Queue *)malloc(sizeof(Queue));
    q->front = q->rear = q->size = 0;
    return q;
}

/* 判断队列是否为空 */
int isEmpty(Queue *q) {
    return q->size == 0;
}

/* 入队操作 */
void enqueue(Queue *q, Vertex *vet) {
    q->vertices[q->rear] = vet;
    q->rear = (q->rear + 1) % MAX_SIZE;
    q->size++;
}

/* 出队操作 */
Vertex *dequeue(Queue *q) {
    Vertex *vet = q->vertices[q->front];
    q->front = (q->front + 1) % MAX_SIZE;
    q->size--;
    return vet;
}

/* 检查顶点是否已被访问 */
int isVisited(Vertex **visited, int size, Vertex *vet) {
    // 遍历查找节点，使用 O(n) 时间
    for (int i = 0; i < size; i++) {
        if (visited[i] == vet)
            return 1;
    }
    return 0;
}

/* 广度优先遍历 */
// 使用邻接表来表示图，以便获取指定顶点的所有邻接顶点
void graphBFS(GraphAdjList *graph, Vertex *startVet, Vertex **res, int *resSize, Vertex **visited, int *visitedSiz
    // 队列用于实现 BFS
    Queue *queue = newQueue();
    enqueue(queue, startVet);
    visited[(*visitedSize)++] = startVet;
    // 以顶点 vet 为起点，循环直至访问完所有顶点
    while (!isEmpty(queue)) {
        Vertex *vet = dequeue(queue); // 队首顶点出队
```

```
            res[(*resSize)++] = vet;      // 记录访问顶点
            // 遍历该顶点的所有邻接顶点
            AdjListNode *node = findNode(graph, vet);
            while (node != NULL) {
                // 跳过已被访问的顶点
                if (!isVisited(visited, *visitedSize, node->vertex)) {
                    enqueue(queue, node->vertex);             // 只入队未访问的顶点
                    visited[(*visitedSize)++] = node->vertex; // 标记该顶点已被访问
                }
                node = node->next;
            }
        }
        // 释放内存
        free(queue);
    }
```

回溯法的定义

主要思想是每次只构造解的一个分量，然后按照下面的方法来评估这个部分构造解。
如果一个部分构造解可以进一步构造而不会违反问题的约束，我们就接受对解的下一个分量所做的第一个合法选择。
如果无法对下一分量进行合法的选择，就不必对剩下的任何分量再做任何选择了。
在这种情况下，该算法进行回溯，把部分构造解的最后一个分量替换为它的下一个选择。

好多斐波那契

递归

```
def Fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```

迭代

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$

```
def Fibonacci_iteration(n):
    a, b = 0, 1
    if n == 0:
        return a
    for _ in xrange(n - 1):
        a, b = b, a + b
    return b
```

公式

对于斐波那契数列这个常见的递推数列，其第 n 项的值的通项公式如下：

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

```python
def Fibonacci_formula(n):
    root_five = 5**0.5
    result = (((1 + root_five) / 2)**n - ((1 - root_five) / 2)**n) / root_five
    return int(result)
```

该方法虽然看起来高效，但是由于涉及大量浮点运算，在n增大时浮点误差不断增大会导致返回结果不正确甚至数据溢出。

简答 30(3 道)

动态规划算法的定义(基本思想)

基本概念

动态规划是一种将复杂问题分解成更小的子问题，并通过存储子问题的结果来避免重复计算，从而提高效率的算法设计技巧。动态规划通常用于解决具有重叠子问题和最优子结构性质的问题。

原理

动态规划的核心思想是通过构建一个表格来存储子问题的解，从而避免重复计算。它通常采用自底向上的方式来逐步求解子问题，最终得到原问题的最优解。

动态规划的**步骤**通常包括：

1. 定义状态：确定用于描述子问题的状态变量。
2. 状态转移方程：找出子问题之间的关系，并用状态转移方程表示。
3. 边界条件：明确初始状态的解。
4. 计算最优解：按照状态转移方程，自底向上计算各子问题的解，最终得到原问题的解。

分治法的定义(基本思想)

分治算法是一种将原问题分解为若干个规模较小且结构与原问题相似的子问题，递归地解决这些子问题，然后将子问题的解合并得到原问题的解的方法。

原理

分治算法的基本步骤包括：

1. 分解：将问题分解成若干个子问题。
2. 解决：递归地解决这些子问题。
3. 合并：将子问题的解合并成原问题的解。
   分治算法适用于问题可以分解成若干规模较小的子问题，并且这些子问题之间相互独立的问题。

回溯法的定义(基本思想)

基本概念

回溯算法是一种系统地搜索问题的解空间的方法，主要用于寻找所有可能的解。它通过逐步构建解决方案，并在发现当前部分解决方案不满足问题的约束时，回溯到上一步重新尝试。

原理

回溯算法的基本思想是：

1. 选择：选择一个可能的解。
2. 约束检查：检查当前选择是否满足问题的约束。
3. 递归：对剩余问题递归求解。
4. 回溯：如果当前选择不满足约束或无法得到最终解，回溯并尝试其他选择。
   回溯算法通常用于解决组合问题、排列问题、子集问题等。

贪心算法的定义(基本思想)

基本概念

贪心算法是一种在每一步选择中都采取局部最优选择，希望通过这些局部最优选择最终达到全局最优的一种算法设计策略。贪心算法适用于具有贪心选择性质的问题，即局部最优选择能够导致全局最优解。

原理

贪心算法的基本原理是：

1. 选择性原则：在当前状态下做出局部最优选择。
2. 不可回溯性：一旦做出选择，不能回溯。
   贪心算法通常不需要存储所有子问题的结果，只需要逐步做出选择并计算当前的结果。

它们各个算法之间的相同与不同

自己看去

计算题 30(3道)

贪心算法(prim和kruskal)(一个A卷一个B卷)

prim

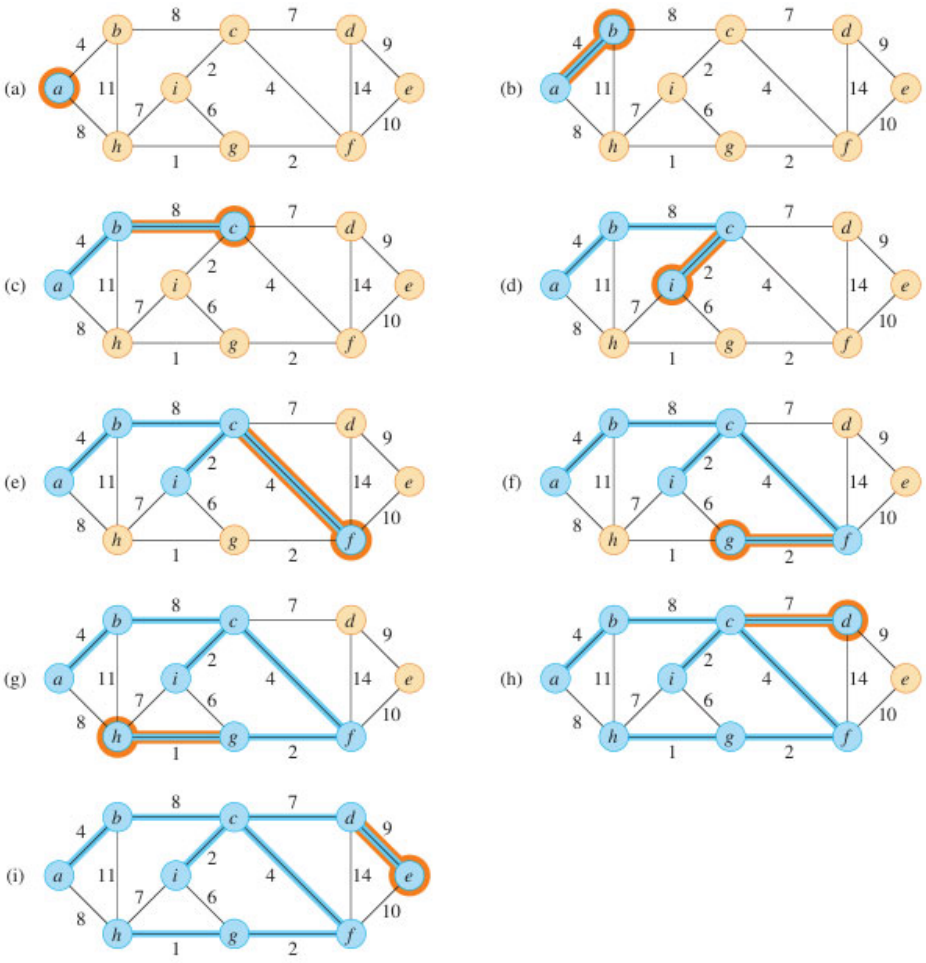


Figure 21.5 The execution of Prim's algorithm on the graph from Figure 21.1. The root vertex is a. Blue vertices and edges belong to the tree being grown, and tan vertices have yet to be added to the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. The edge and vertex added to the tree are highlighted in orange. In the second step (part (c)), for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

该算法的基本思想是从一个结点开始，不断加点（而不是 Kruskal 算法的加边）。
简单讲：随便选个点，然后找和它相连的所有边里权重最小的，把另外一个端点加进来，再找和这俩点相连的所有边里权重最小的，再加入下一个点，以此类推，记得不要成圈。

暴力：$O(n^2 + m)$。
二叉堆：$O((n + m) \log n)$。
Fib 堆：$O(n \log n + m)$。
arbitrary node：任意节点
伪代码:

```
1   Input. The nodes of the graph V ; the function g(u, v) which
    means the weight of the edge (u, v); the function adj(v) which
    means the nodes adjacent to v.
2   Output. The sum of weights of the MST of the input graph.
3   Method.
4   result ← 0
5   choose an arbitrary node in V to be the root
6   dis(root) ← 0
7   for each node v ∈ (V − {root})
8       dis(v) ← ∞
9   rest ← V
10  while rest ≠ ∅
11      cur ← the node with the minimum dis in rest
12      result ← result + dis(cur)
13      rest ← rest − {cur}
14      for each node v ∈ adj(cur)
15          dis(v) ← min(dis(v), g(cur, v))
16  return result
```

具体来说，每次要选择距离最小的一个结点，以及用新的边更新其他结点的距离。

其实跟 Dijkstra 算法一样，每次找到距离最小的一个点，可以暴力找也可以用堆维护。

堆优化的方式类似 Dijkstra 的堆优化，但如果使用二叉堆等不支持 O(1) decrease-key 的堆，复杂度就不优于 Kruskal，常数也比 Kruskal 大。所以，一般情况下都使用 Kruskal 算法，在稠密图尤其是完全图上，暴力 Prim 的复杂度比 Kruskal 优，但 不一定 实际跑得更快。

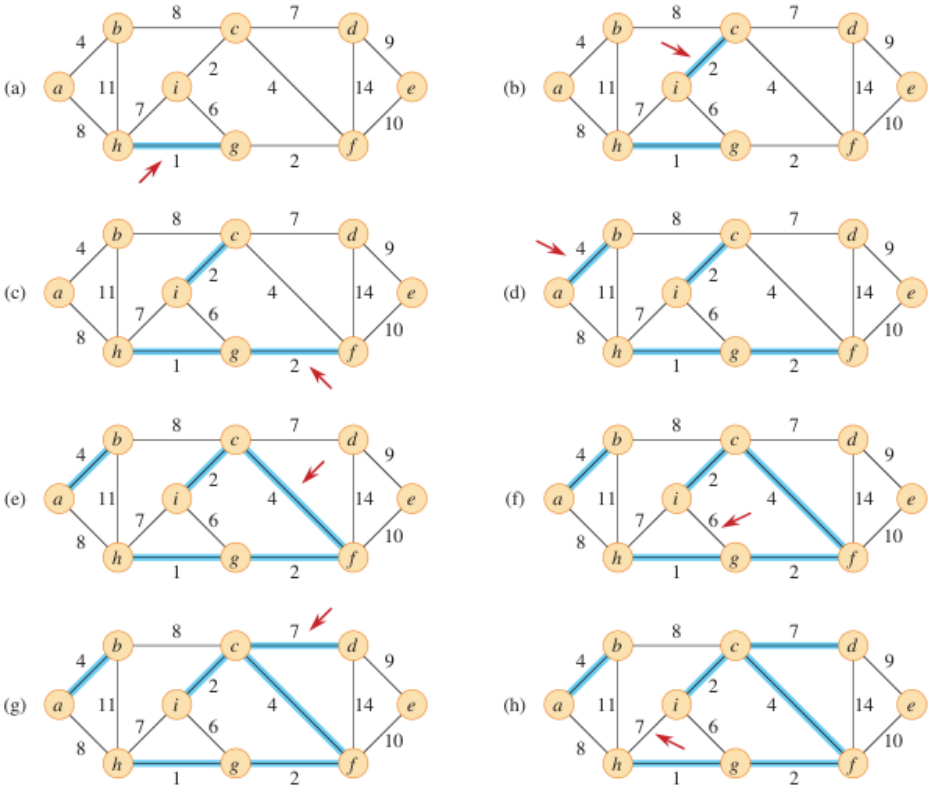


Figure 21.4 The execution of Kruskal's algorithm on the graph from Figure 21.1. Blue edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. A red arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Kruskal's algorithm

Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) with the lowest weight. Let C_1 and C_2 denote the two trees that are connected by (u, v). Since (u, v) must be a light edge connecting C_1 to some other tree, Corollary 21.2 implies

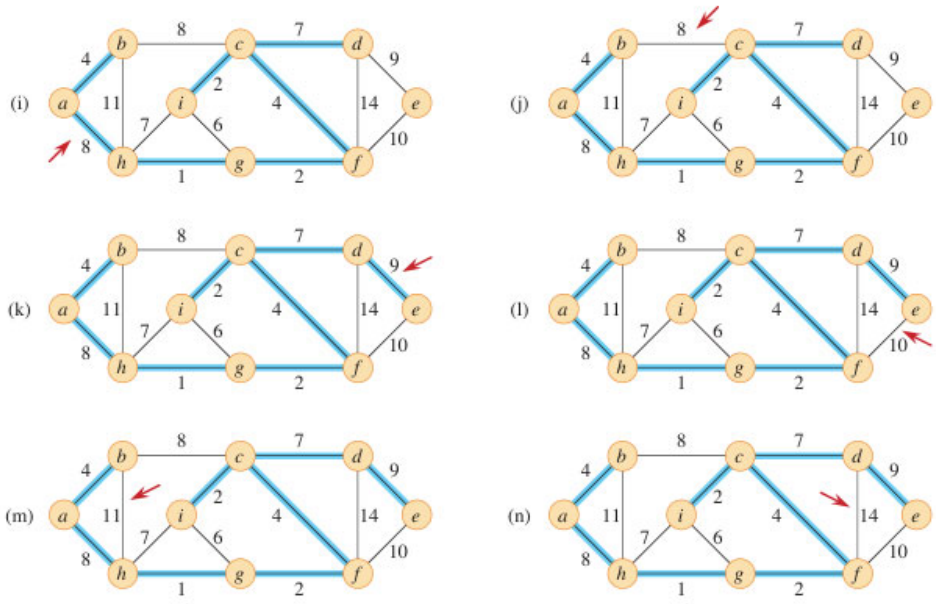


Figure 21.4, continued Further steps in the execution of Kruskal's algorithm.

该算法的基本思想是从小到大加入边，是个贪心算法。
简单讲：按边的权值从小到大往MST里加，且不能成圈

伪代码：

```
1   Input. The edges of the graph e, where each element in e is (u, v, w)
     denoting that there is an edge between u and v weighted w.
2   Output. The edges of the MST of the input graph.
3   Method.
4   result ← ∅
5   sort e into nondecreasing order by weight w
6   for each (u, v, w) in the sorted e
7         if u and v are not connected in the union-find set
8               connect u and v in the union-find set
9               result ← result ⋃ {(u, v, w)}
10  return result
```

算法虽简单，但需要相应的数据结构来支持……具体来说，维护一个森林，查询两个结点是否在同一棵树中，连接两棵树。

抽象一点地说，维护一堆 集合，查询两个元素是否属于同一集合，合并两个集合。

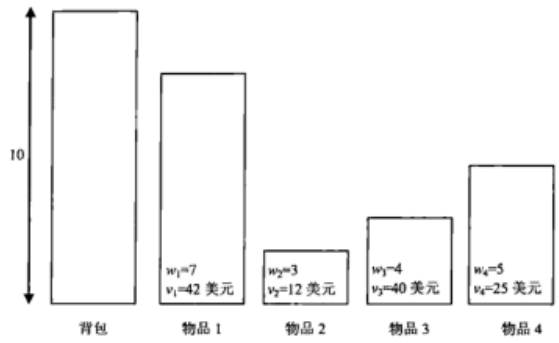
其中，查询两点是否连通和连接两点可以使用并查集维护。

如果使用 $O(m \log m)$ 的排序算法，并且使用 $O(m\alpha(m,n))$ 或 $O(m \log n)$ 的并查集，就可以得到时间复杂度为 $O(m \log m)$ 的 Kruskal 算法。

背包问题(蛮力法，动态规划，回溯法)

蛮力解01背包

简单概括：穷举
看图就懂了



(a)

子 集	总 重 量	总价值/美元
∅	0	0
{1}	7	42
{2}	3	12
{3}	4	40
{4}	5	25
{1, 2}	10	54
{1, 3}	11	不可行
{1, 4}	12	不可行
{2, 3}	7	52
{2, 4}	8	37
{3, 4}	**9**	**65**
{1, 2, 3}	14	不可行
{1, 2, 4}	15	不可行
{1, 3, 4}	16	不可行
{2, 3, 4}	12	不可行
{1, 2, 3, 4}	19	不可行

(b)

图 3.8 (a)背包问题的一个实例，(b)用穷举查找求得的解(最优的选择用粗体字表示)

学术一点：（其实也不怎么学术

蛮力法解决0-1背包问题的思路是:枚举所有可能的物品组合，计算它们的总重量和总价值，找出符合条件的最优解。具体实现方法如下

1. 枚举所有可能的物品组合，可以使用二进制数表示。例如，对于n个物品，可以用一个n位的二进制数表示一个物品组合，其中第i为1表示选择第i个物品，为0表示不选择。
2. 对于每个物品组合，计算它们的总重量和总价值，
3. 如果总重量不超过C，比较总价值与当前最优解的价值，更新最优解。
4. 重复步骤1~3，直到枚举完所有可能的物品组合。

代码实现：（找不到伪代码了

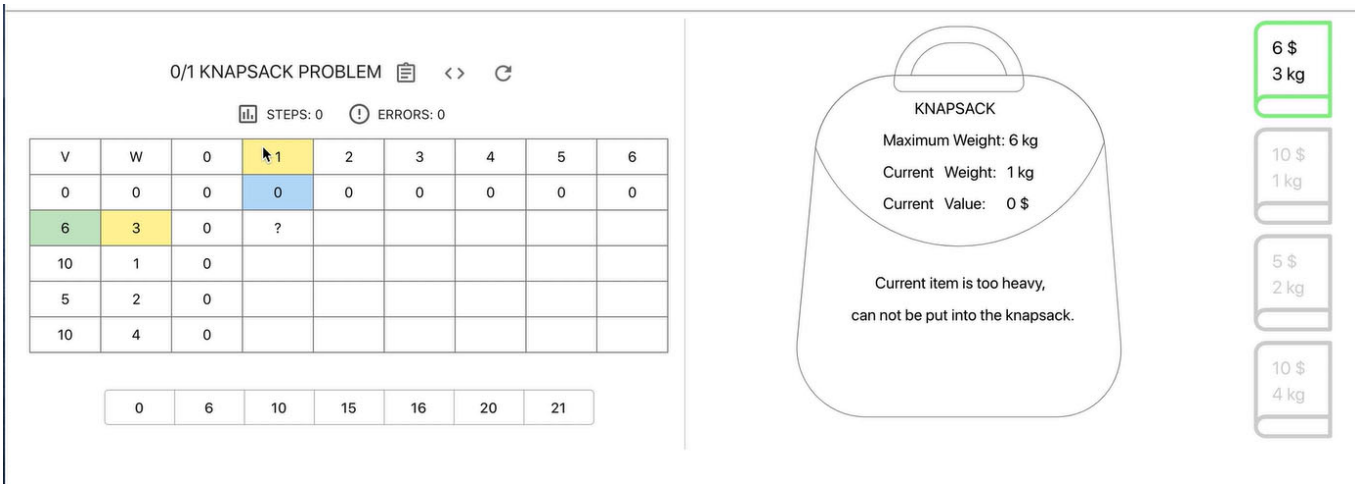
```
#include <stdio.h>
#define MAX_N 100
#define MAX_c 1000

int n,c;

int W[MAX N],V[MAX N];
int best_v;// 当前最优解的价值

void dfs(int i,int cw,int cv){
    if(i == n){ // 已经枚举完所有物品
        if(cw<=C && cv > best_v){
            best_v= cv;// 更新最优解
        }
        return;
    }
    dfs(i+1, cw, cv);//不选第i个物品
    dfs(i + 1, cw + w[i], cv + v[i]);// 选第i个物品
}

int main(){
    scanf("%d%d", &n, &c);
    for(int i=0; i<n; i++){
        scanf("%d%d" &w[i],&v[i]);
    }
    best_v=0;
    dfs(0, 0, 0);
    printf("%d\n",best v);
    return 0;
}
```

伪代码自己脑补下吧（

时间复杂度为O(nlogn)，其中n为物品数量

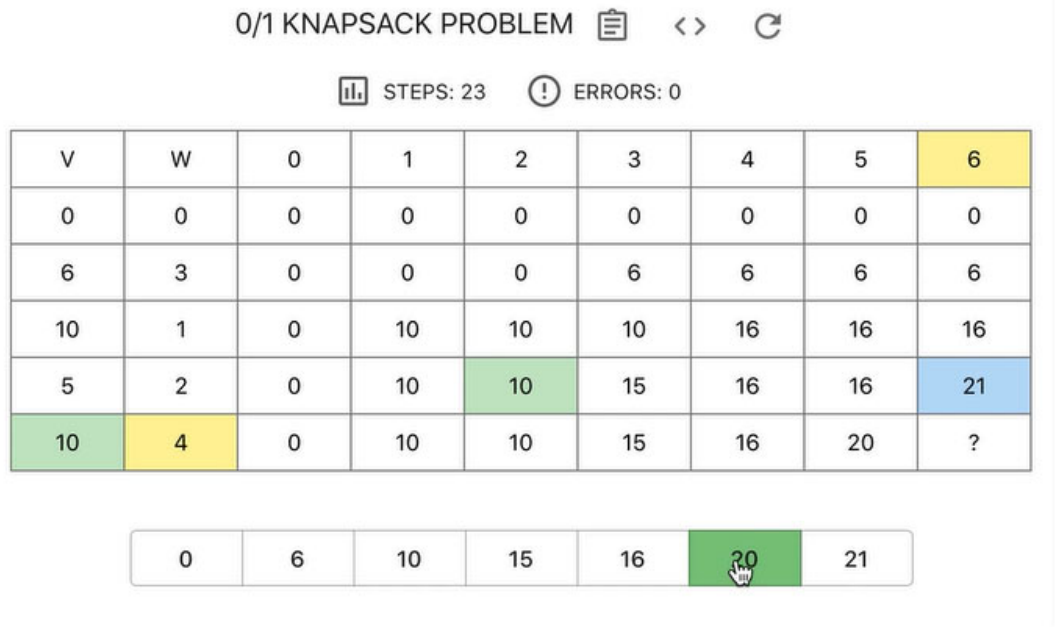
动态规划

背包最大承受重量有限制，大小无限制
0-1：物品不能重复使用，不能分割
b站课程：
dp表
整个表格第一行为背包的承受重量，从0到最大，表示每列中背包的最大承受重量。



如果放不下，就取上面正对着的那个格子的值

如果放得下：
俩绿格子加起来和蓝格子（正上方的）比大小，问号处放较大的那个数
一个绿格子取当前物品的价值
另外一个绿格子意思是：当背包最大承受重量为4，并且商品未放入背包前，背包的最优解。4 - 3 = 1且现在在w=3那行，上面一行为w=0，故取[0, 1]
每次都取[当前行的上一行, (当前背包最大承受重量-当前物品重量)]作为绿格子（我也不知道叫啥



FORMULA

```
if (itemWeight > currentWeight) {
  table[row][col] = table[row - 1][col];
} else {
  table[row][col] = Math.max(
    table[row - 1][col],
    table[row - 1][currentWeight - itemWeight] + itemValue
  );
}
```

网站：hello-algo

Question

给定 n 个物品，第 i 个物品的重量为 $wgt[i-1]$、价值为 $val[i-1]$，和一个容量为 cap 的背包。每个物品只能选择一次，问在限定背包容量下能放入物品的最大价值。

观察图 14-17，由于物品编号 i 从 1 开始计数，数组索引从 0 开始计数，因此物品 i 对应重量 $wgt[i-1]$ 和价值 $val[i-1]$。

动态规划实质上就是在状态转移中填充 dp 表的过程，代码如下所示：

```c
/* 0-1 背包：动态规划 */
int knapsackDP(int wgt[], int val[], int cap, int wgtSize) {
    int n = wgtSize;
    // 初始化 dp 表
    int **dp = malloc((n + 1) * sizeof(int *));
    for (int i = 0; i <= n; i++) {
        dp[i] = calloc(cap + 1, sizeof(int));
    }
    // 状态转移
    for (int i = 1; i <= n; i++) {
        for (int c = 1; c <= cap; c++) {
            if (wgt[i - 1] > c) {
                // 若超过背包容量，则不选物品 i
                dp[i][c] = dp[i - 1][c];
            } else {
                // 不选和选物品 i 这两种方案的较大值
                dp[i][c] = myMax(dp[i - 1][c], dp[i - 1][c - wgt[i - 1]] + val[i - 1]);
            }
        }
    }
    int res = dp[n][cap];
    // 释放内存
    for (int i = 0; i <= n; i++) {
        free(dp[i]);
    }
    return res;
}
```

创建dp表见 https://www.hello-algo.com/chapter_dynamic_programming/knapsack_problem/#3

随便截两张意思一下

重量 wgt	价值 val
wgt[i-1]	val[i-1]
1	5
2	11
3	15

i \ c	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0

物品数量 **n** = 3　　背包容量 **cap** = 4

初始化尺寸为（**n** + 1）×（**cap** + 1）的 dp 矩阵

重量 wgt	价值 val
wgt[i-1]	val[i-1]
1	5
2	11
3	15

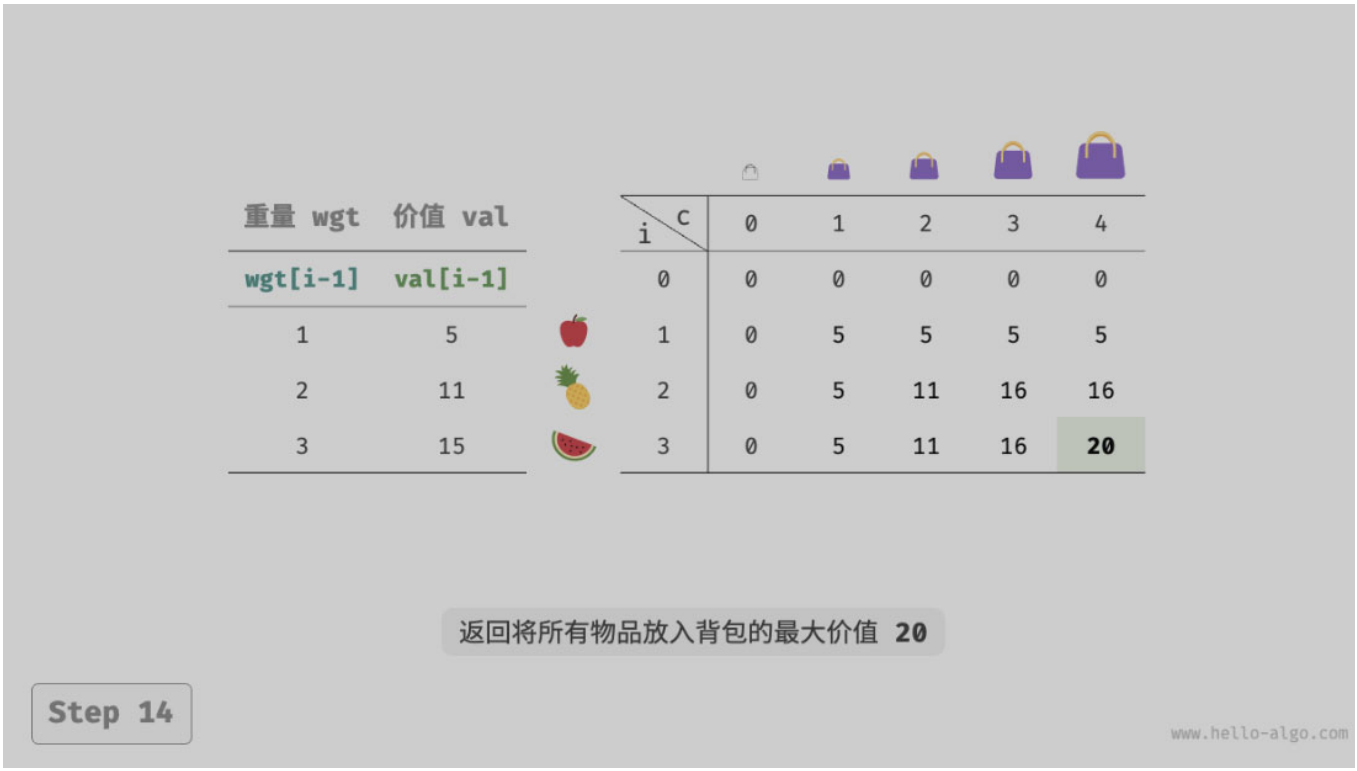
i \ c	0	1	2	3	4
0	0	0	0	0	0
1	**0**	5	5	5	5
2	0	5	**11**	0	0
3	0	0	0	0	0

+11

状态转移：

`dp[i, c]` = max(`dp[i - 1, c]` , **`dp[i - 1, c - wgt[i-1]] + val[i-1]`**)

Step 7



时间复杂度为O(NW)，其中N 表示物品数量，W 表示背包的容量

回溯

还是dfs

给定n种物品和一个背包。物品i的重量是w_i ,其价值为v_i,背包的容量为W。一种物品要不全部装入背包，要不不装入背包，不允许部分物品装入的情况。装入背包的物品的总重量不能超过背包的容量，在这种情况下，问如何选择装入背包的物品，使得装入背包的物品的总价值最大？需要采用回溯的方法进行问题的求解。

分析：

（1）问题的解空间：
将物品装入背包，有且仅有两个状态。第i种物品对应$(x_1, x_2, ..., x_n)$,其中x_i可以取0或1，分别代表不放入背包和放入背包。解空间有2^n种可能解，也就是n个元素组成的集合的所有子集的个数。采用一颗满二叉树来讲解空间组织起来，解空间树的深度为问题的规模n。

（2）约束条件：

$$\sum_{i=1}^{n} w_i x_i \leq W$$

（3）限界条件：
0-1背包问题的可行解不止一个，而目标是找到总价值最大的可行解。因此需要设置限界条件来加速找出最优解的速度。如果当前是第t个物体，那么1-t物体的状态都已经被确定下来，剩下就是t+1~n物体的状态，采用贪心算法计算当前剩余物品所能产生的最大价值是否大于最优解，如果小于最优解，那么被剪枝掉。

采用回溯法进行问题的求解，也就是**具有约束函数/限界条件的深度优先搜索**算法。

采用回溯法特有框架：

```
回溯算法()
    如果到达边界：
        记录当前的结果，进行处理
    如果没有到达边界：
        如果满足限界条件：（左子树）
            进行处理
            进行下一层的递归求解
            将处理回退到处理之前
        如果不满足限界条件：（右子树）
            进行下一层递归处理
```

伪代码描述：

递归式回溯算法：

```
BACKTRACK-REC(t)     //t为扩展结点在树中所处的层次
     if t > n                                        //已到叶子结点，输出结果
       OUTPUT(x)
       //检查扩展结点的每个分支。s(n,t)与e(n,t)分别为当前扩展结点处未搜索过
       //的子树的起始编号和终止编号
     else
       for i from s(n,t) to e(n,t)
          x[t] = h(i)          //  h[i]: 在当前扩展结点处 x[t]的第i个可选值
          if CONSTRAINT(t) && BOUND(t) //约束函数与限界函数
              BACKTRACK-REC(t+1) //进入t+1层搜索
```

迭代式回溯算法：

```
BACKTRACK-ITE()
    t = 1              //t为扩展结点在树中所处的层次
    while t > 0:
      if s(n,t) <= e(n,t)
        for i from s(n,t) to e(n,t)
          do x[t] = h(i)    //h[i]: 在当前扩展结点处x[t]的第i个可选值
            if CONSTRINT(t) && BOUND(t)  //满足约束限界条件
              if t > n       //已到叶子结点，输出结果
                OUTPUT(x);
              else
                t ++         //前进到更深层搜索
      else
        t --                //回溯到上一层的活结点
```

时间复杂度：

$$T(n) = O(2n) + O(n2n) + O(nlog(n)) = O(n2^n)$$

空间复杂度：

$$O(nlog(n))$$

warshall和floyd(一个A卷一个B卷)

warshall算法

求传递闭包
笑死，离散刚考过

简单说就是写出R(0)，推出一堆矩阵R(n)

计算R(n)的方法：找出该矩阵的第n列有1的行，将该矩阵的第n行加到这些行上且保持这些行位置不变。

n为图中节点数。

加法：这里是or运算，具体为1+1=1，0+1=1，0+0=0

作为例子，图 8.13 告诉我们如何对图 8.11 中的有向图应用 Warshall 算法。

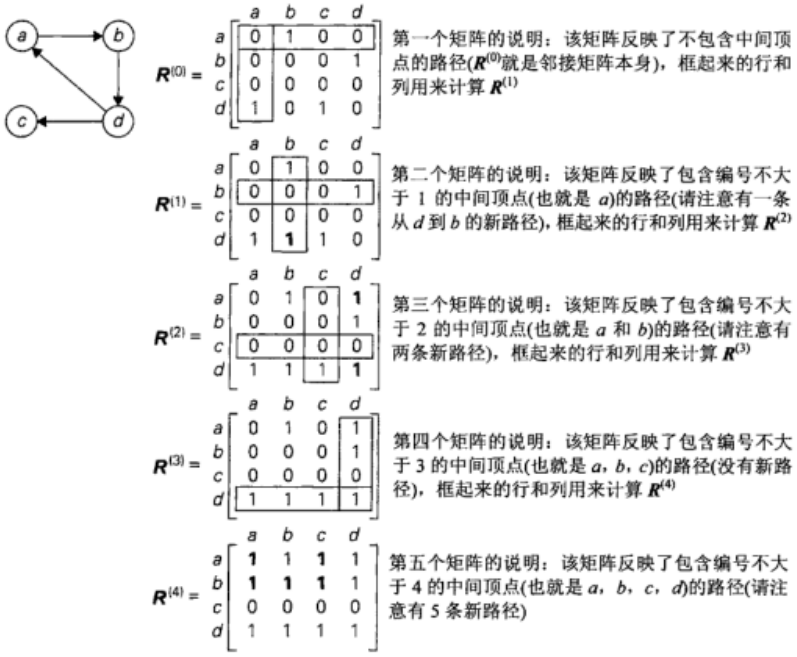


图 8.13　对图中的有向图应用 Warshall 算法，新的路径用粗体字表示

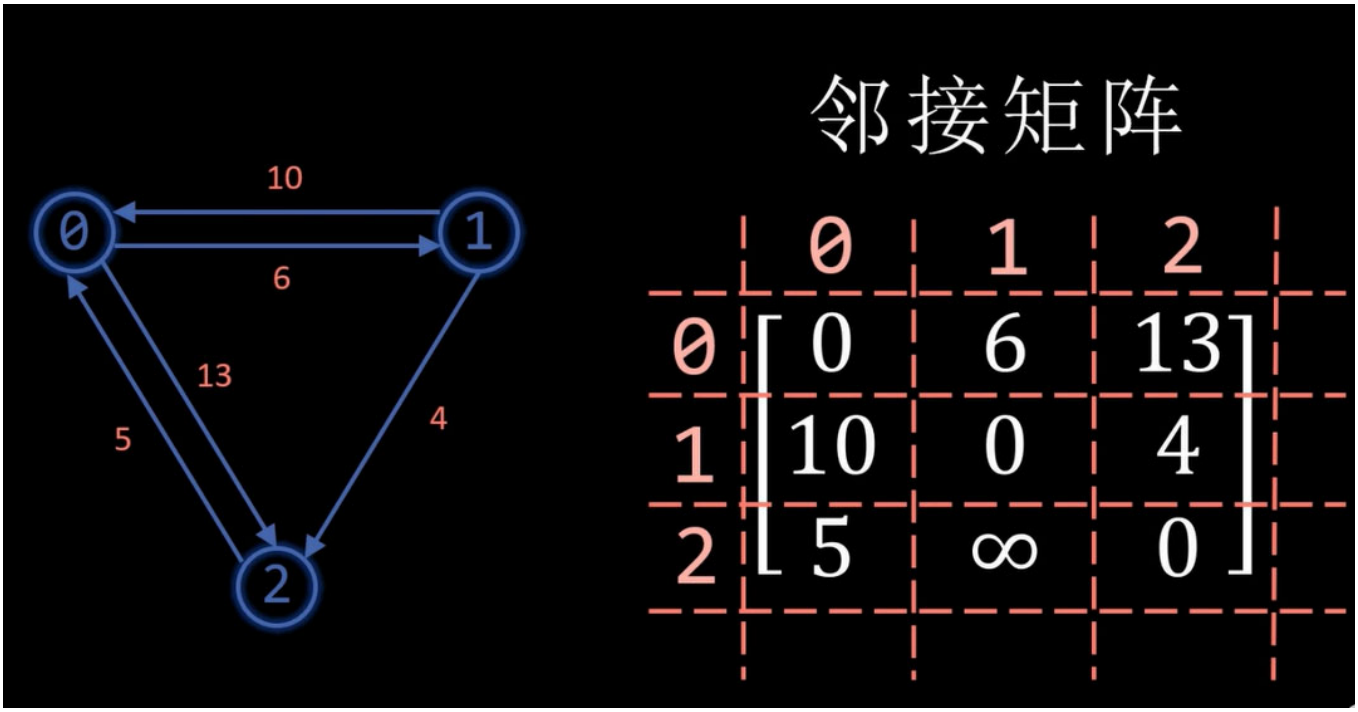
代码：（csdn抄的，自己编伪代码吧

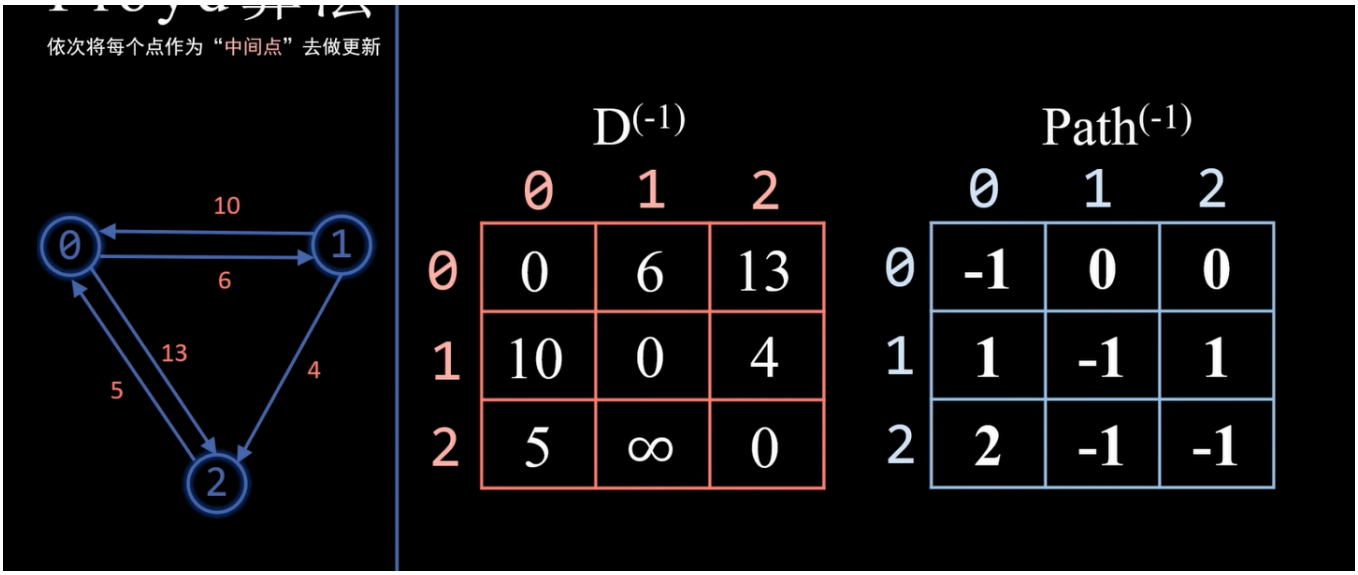
```
void Warshall(int nearArra[ROWS][COLUMNS]) {
        //Warshall算法求运用邻接矩阵求图中所有可能的通路（即可达矩阵）
        //参数 ：  图的邻接矩阵
        for (int i = 0; i < COLUMNS; i++) {
                for (int j = 0; j < ROWS; j++) {
                        if (0 != nearArra[j][i]) {
                                //若此位置是1，此位置对应的列序号的行（即i）加到当前位置的行（即j）上
                                for (int k = 0; k < COLUMNS; k++) {
                                        nearArra[j][k] += nearArra[i][k];    //nearArra是前面定义过的图的矩阵
                                }
                                }
                        }
                }
        }
```

floyd算法

求任意两点间最短路径，也能算有向图的传递闭包

例1：





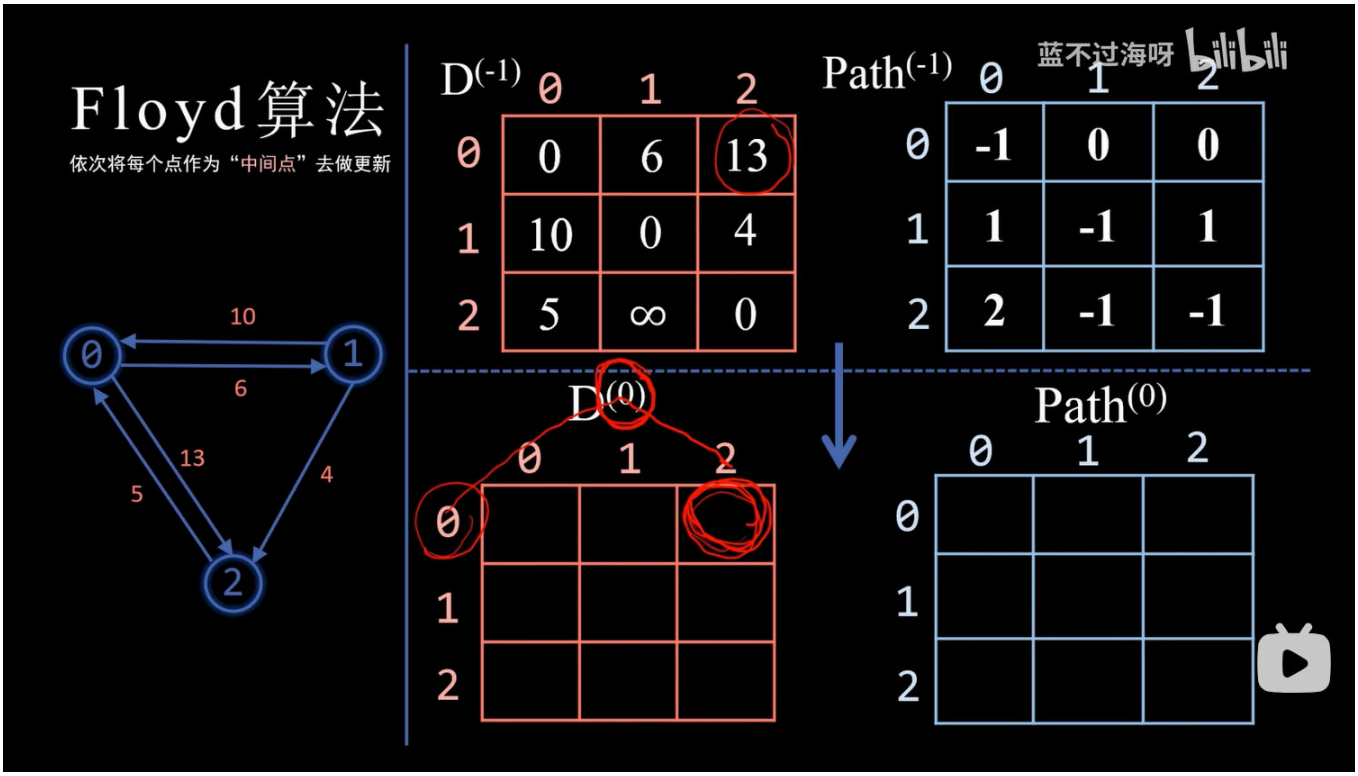
D：保存任意两点间最短路径的长度

Path：保存任意两点间的最短路径本身，为方便会存终点的前驱点(如图中[0,2]=0，因为0->2有边，终点为0，终点的前驱点就为2)

上标-1代表初始

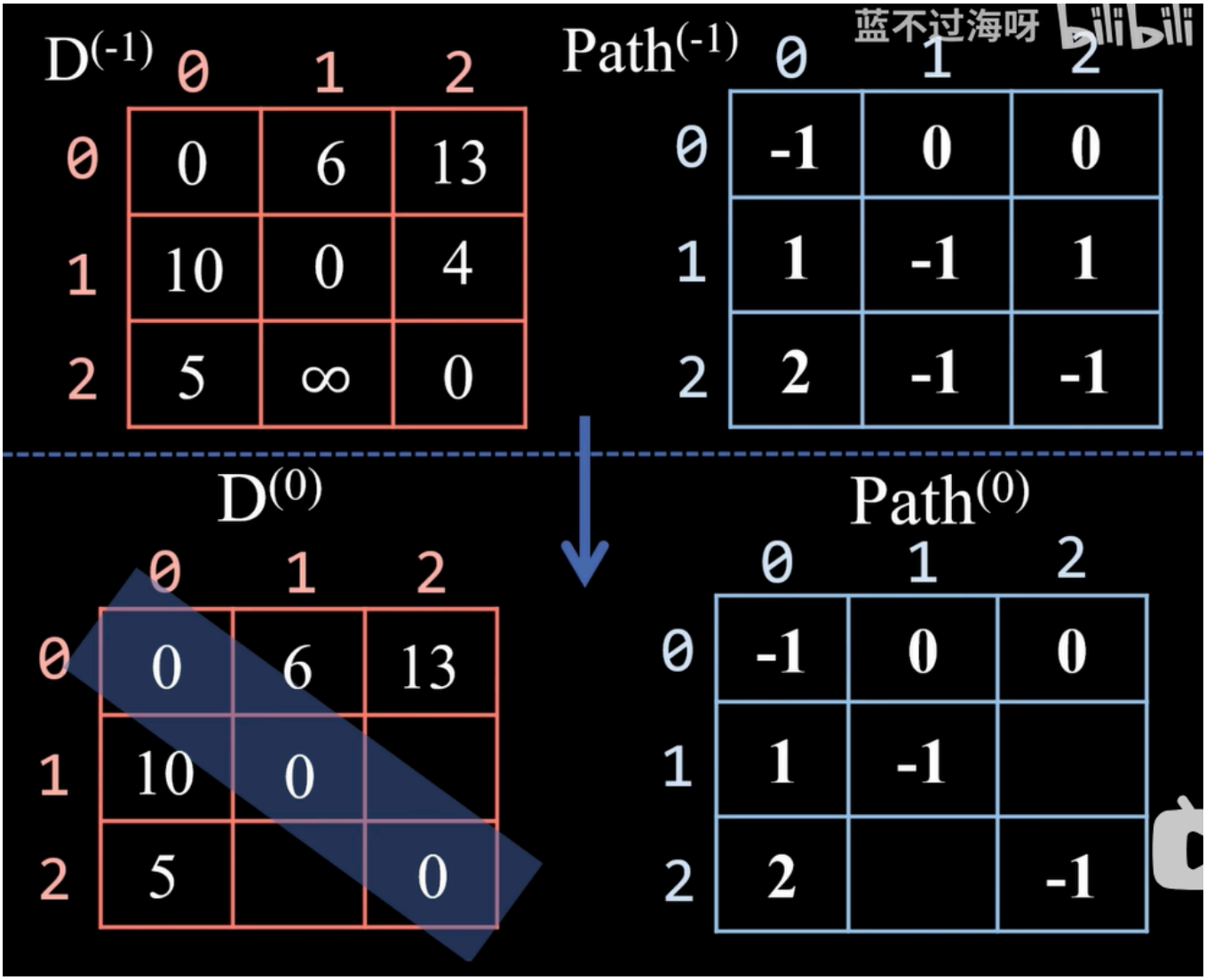
path中的-1表示不存在

D中长度不存在则置为无穷大

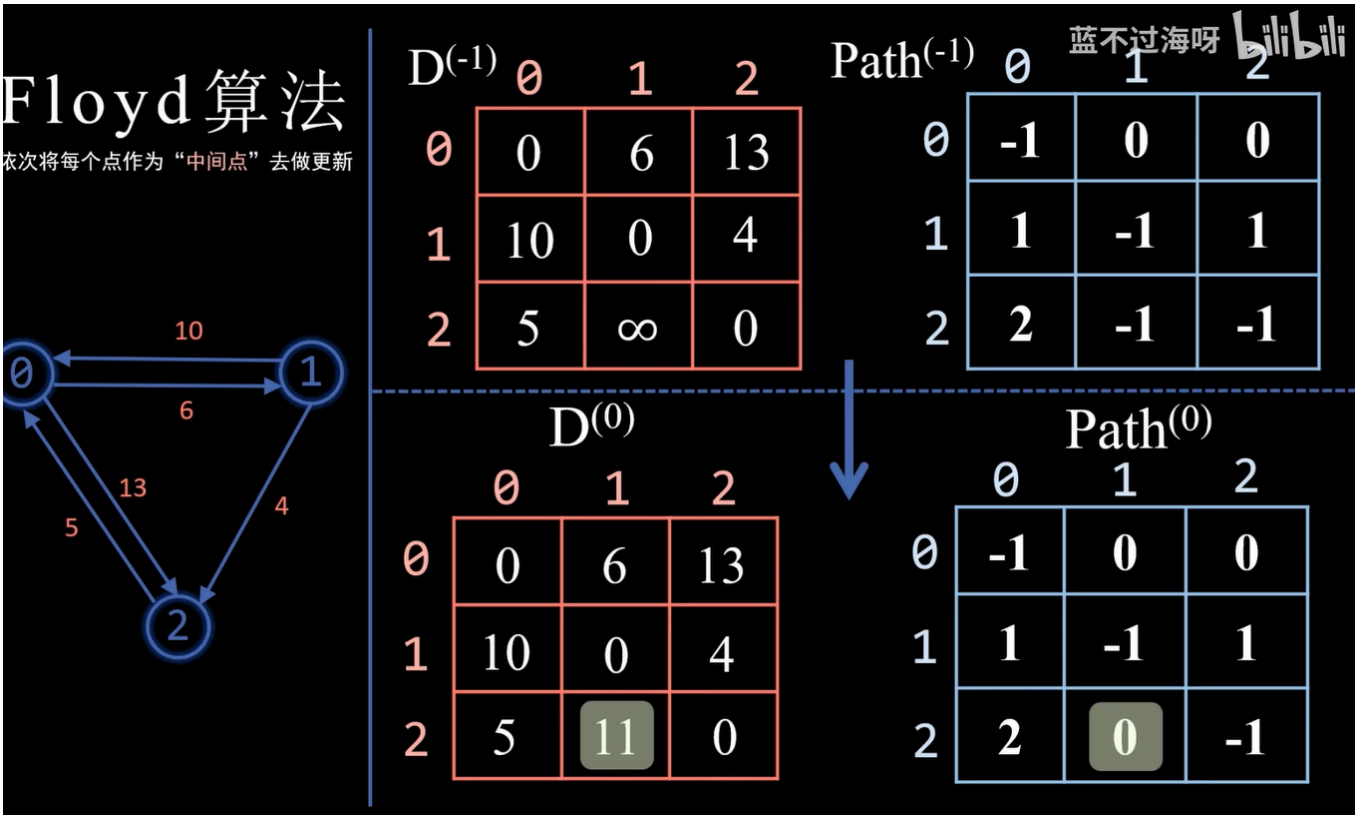


上图开始更新两个表。表上标代表作为中间点的节点，如果路径长度比之前记录的短，则更新

例：图中圈出的格子代表0**通过0**到2，也就是0到2.
由此可知，当上标为n时，表中的行n和列n都不需要改变。如下图。本例中对角线同理，不需要更改

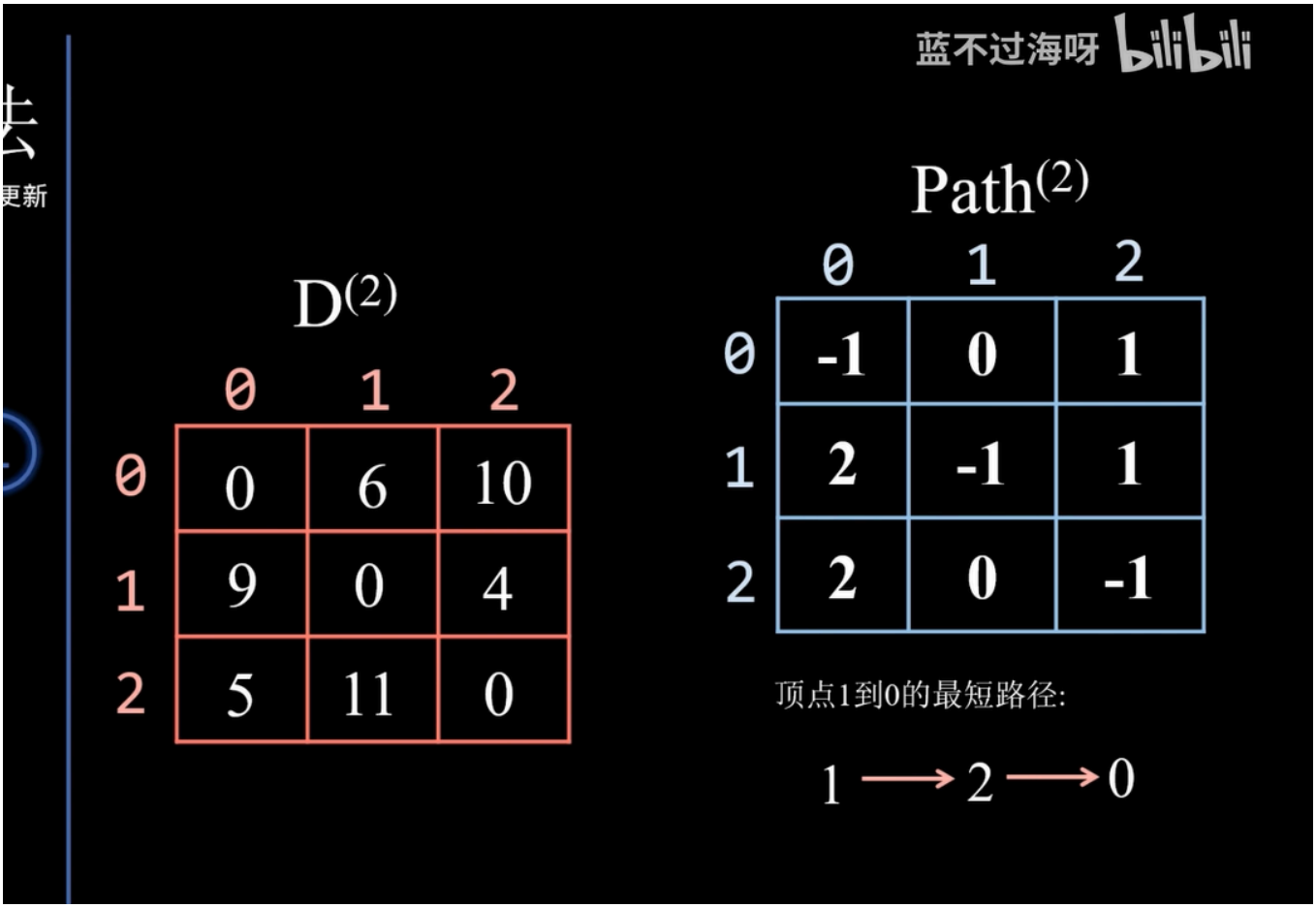

D(-1)
0 1 2
0 0 6 13
1 10 0 4
2 5 ∞ 0
Path(-1)
0 1 2
0 -1 0 0
1 1 -1 1
2 2 -1 -1
蓝不过海呀
D(0)
0 1 2
0 0 6 13
1 10 0
2 5 0
Path(0)
0 1 2
0 -1 0 0
1 1 -1
2 2 -1

更新完的数组如下：



后面同理。
D数组观察最短路径长度比较方便

Path推最短路径：



图中记录的是终点的前一个点

如上图，求1->0的最短路径，则推出2->0（查表得，终点是0，表中记录为终点的前驱点），再看1怎么到2，查表得1->2得出1->2->0