三个基本算法

一、深度优先搜索

一种用于遍历或搜索树或图的算法。这个算法会尽可能深的搜索树的分支。当节点v的所在边都己被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。

public class TreeNode extends Object {

public final int val;

public TreeNode left = null;

public TreeNode right = null;

public TreeNode(int val) {

this.val = val;

}

@Override

public String toString() {

return "TR" + val;

}

@Override

public int hashCode() {

return val;

}

@Override

public boolean equals(Object obj) {

if (obj == null || !(obj instanceof TreeNode)) {

return false;

}

TreeNode tar = (TreeNode) obj;

return tar.val == this.val && tar.left == this.left && tar.right == this.right;

}

}

public class S102PrintTree {

public static void main(String[] a) {

TreeNode root = new TreeNode(3);

TreeNode l2\_1 = new TreeNode(9);

TreeNode l2\_2 = new TreeNode(20);

TreeNode l3\_1 = new TreeNode(15);

TreeNode l3\_2 = new TreeNode(7);

root.left = l2\_1;

root.right = l2\_2;

l2\_2.left = l3\_1;

l2\_2.right = l3\_2;

System.*out*.println(*dfs*(root));

}

private static List<List<Integer>> dfs(TreeNode root) {

List<List<Integer>> res = new ArrayList<>();

*dfsHelper*(root, 0, res);

return res;

}

private static void dfsHelper(TreeNode now, int lvl, List<List<Integer>> resArg) {

if (now == null) {

return;

}

while (resArg.size() <= lvl) {

resArg.add(new LinkedList<>());

}

resArg.get(lvl).add(now.val);

*dfsHelper*(now.left, lvl + 1, resArg);

*dfsHelper*(now.right, lvl + 1, resArg);

}

}

二、广度优先搜索

BFS是从[根节点](https://zh.wikipedia.org/wiki/%E6%A0%91_(%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84)#%E6%9C%AF%E8%AF%AD)开始，沿着树的宽度遍历树的[节点](https://zh.wikipedia.org/wiki/%E8%8A%82%E7%82%B9)。如果所有节点均被访问，则算法中止。

public static List<List<Integer>> bfs(TreeNode root) {

*maxQsize* = 0;

List<List<Integer>> res = new ArrayList<>();

if (root == null ) {

return res;

}

Queue<TreeNode> q = new LinkedList<>();

q.add(root);

Set<TreeNode> visited = new HashSet<>();

while (!q.isEmpty()) {

int levelSize = q.size();

if (levelSize > *maxQsize*) {

*maxQsize* = levelSize;

}

List<Integer> cur = new ArrayList<>();

for (int i = 0; i < levelSize; i++) {

TreeNode n = q.poll();

if (n.left != null && !visited.contains(n.left)) {

q.add(n.left);

}

if (n.right != null && !visited.contains(n.right)) {

q.add(n.right);

}

cur.add(n.val);

visited.add(n);

}

res.add(cur);

}

return res;

}

三、动态规划

1、基本概念

    动态规划过程是：每次决策依赖于当前状态，又随即引起状态的转移。一个决策序列就是在变化的状态中产生出来的，所以，这种多阶段最优化决策解决问题的过程就称为动态规划。

2、求解的基本步骤

     动态规划所处理的问题是一个多阶段决策问题，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态。这些决策形成了一个决策序列，同时确定了完成整个过程的一条活动路线(通常是求最优的活动路线)。如图所示。动态规划的设计都有着一定的模式，一般要经历以下几个步骤。

    初始状态→│决策１│→│决策２│→…→│决策ｎ│→结束状态

                      图1 动态规划决策过程示意图

    (1)划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。在划分阶段时，注意划分后的阶段一定要是有序的或者是可排序的，否则问题就无法求解。

    (2)确定状态和状态变量：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。

    (3)确定决策并写出状态转移方程：因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以如果确定了决策，状态转移方程也就可写出。但事实上常常是反过来做，根据相邻两个阶段的状态之间的关系来确定决策方法和状态转移方程。

    (4)寻找边界条件：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。

    一般，只要解决问题的阶段、状态和状态转移决策确定了，就可以写出状态转移方程（包括边界条件）。

实际应用中可以按以下几个简化的步骤进行设计：

    （1）分析最优解的性质，并刻画其结构特征。

    （2）递归的定义最优解。

    （3）以自底向上或自顶向下的记忆化方式（备忘录法）计算出最优值

    （4）根据计算最优值时得到的信息，构造问题的最优解

3、动态规划算法基本框架

for(j=1; j<=m; j=j+1) // 第一个阶段

  xn[j] = 初始值;

 for(i=n-1; i>=1; i=i-1)// 其他n-1个阶段

  for(j=1; j>=f(i); j=j+1)//f(i)与i有关的表达式

xi[j]=j=max（或min）{g(xi-1[j1:j2]), ......, g(xi-1[jk:jk+1])};

t = g(x1[j1:j2]); // 由子问题的最优解求解整个问题的最优解的方案

print(x1[j1]);

for(i=2; i<=n-1; i=i+1）

{

t = t-xi-1[ji];

for(j=1; j>=f(i); j=j+1)

if(t=xi[ji])

break;

}

心得：

算法是程序员的基本功，学习算法最重要的是学习解题时候的思考方式以启发具体问题的解决。