抓脑壳系列之分治回溯和递归

今日目标：

1：完成递归的相关面试题

2：能够说出分治，回溯的算法思想

3：完成分治回溯面试题

1、递归

111. 二叉树的最小深度

字节，腾讯，三星面试题，111. 二叉树的最小深度

递归解法：

一定要搞清楚题意：最小深度是从根节点**到最近叶子节点**的最短路径上的节点数量。

```
class Solution {
    public int minDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        //1.左孩子和有孩子都为空的情况，说明到达了叶子节点，直接返回1即可
        if(root.left == null && root.right == null) {
            return 1;
        }
        //如果左孩子为空
        if (root.left == null) {
            return minDepth(root.right) +1;
        }
        //如果右还在为空
        if (root.right == null) {
            return minDepth(root.left) +1;
        }
        //左右孩子都不为空
        return Math.min(minDepth(root.left),minDepth(root.right)) +1;
    }
}
```

非递归解法：**广度优先搜索BFS**

```
class Solution {
```

```
    //利用bfs
    public int minDepth(TreeNode root) {
        //特殊判断
        if (root ==null) {
            return 0;
        }

        int pathLevel = 1;
        //bfs需要借助队列(dfs借助递归)
        Queue<TreeNode> queue = new LinkedList();
        queue.offer(root);

        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i=0;i<size;i++) {
                TreeNode poll = queue.poll();
                if (poll.left==null && poll.right ==null) {
                    return pathLevel;
                }
                if (poll.left!=null) {
                    queue.offer(poll.left);
                }
                if (poll.right!=null) {
                    queue.offer(poll.right);
                }
            }
            pathLevel++;
        }
        return pathLevel;
    }
}
```

236. 二叉树的最近公共祖先

美团点评，亚马逊最近面试，236. 二叉树的最近公共祖先

> 参考精选题解：https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/solution/236-er-cha-shu-de-zui-jin-gong-gong-zu-xian-jian-j/

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        //递归终止条件
        if (root == null || root == p || root == q) { // 下探的过程中，遇到谁谁就是该子树的最近公共父节点，再没有往下查找的必要了，
            return root;
```

```
        }

        //分别在左子树和右子树中查找最近父节点
        TreeNode leftp = lowestCommonAncestor(root.left,p,q);
        TreeNode rightp = lowestCommonAncestor(root.right,p,q);
        //左子树中没有最近公共父节点,那肯定在右子树中，右子树中没有最近公共父节点的话那肯定在左子树中
        if (leftp == null) {
            return rightp;
        }
        if (rightp == null) {
            return leftp;
        }

        //如果左右子树中都没有找到最近公共父节点，证明p和q一个在左子树，一个在右子树，那当前节点，也就是左右子树的父节点就为最近公共父节点
        return root;


    }
}
```

2、算法思想-分治，回溯

前言：

分治和回溯，从本质上来讲它算一种递归，只不过他是属于递归的一个细分类，所以我们可以任务分治回溯就是一种特殊的递归或者复杂度的递归。

> 因此考虑这种问题有一个要点：找重复性(重复子问题)。

2.1、分治

> 分治算法在维基百科上的定义为：在计算机科学中，**分治法**是建基于多项分支递归的一种很重要的算法范式。字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

通过维基百科的定义我们可以发现分治算法的核心就是**分而治之**，当然这个定义和递归有点类似，这里我们要说一下分治和递归的区别：**分治算法是一种处理问题的思想，递归是一种编程技巧**，当然了在实际情况中，分治算法大都采用递归来实现。

> 并且用递归实现分治算法的基本步骤为：
>
> 1：分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

> 2：解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题
>
> 3：合并：将各个子问题的解合并为原问题的解。

什么样的问题适合用分治算法去解决呢?

- 原问题与分解成的小问题具有相同的模式；
- 原问题分解成的子问题可以独立求解，子问题之间没有相关性，这一点是分治算法跟动态规划的明显区别，至于动态规划下一小节会详细讲解并对比这两种算法；
- 具有分解终止条件，也就是说，当问题足够小时，可以直接求解；
- 可以将子问题合并成原问题，并且合并操作的复杂度不能太高，否则就起不到减小算法总体复杂度的效果了，这也是能否使用分治法的关键特征

分治算法应用场景举例：

排序：后期要讲的很多排序算法有很多就利用了分治的思想，比如归并排序，快速排序。

海量数据处理：分治算法思想还经常用在海量数据处理的场景中。比如：给 10GB 的订单文件按照金额排序这样一个需求，看似是一个简单的排序问题，但是因为数据量大，有 10GB，而我们的机器的内存可能只有 2、3GB，总之就是小于订单文件的大小因而无法一次性加载到内存，所以基础的排序算法在这样的场景下无法使用。

> 要解决这种数据量大到内存装不下的问题，我们就可以利用分治的思想。我们可以将海量的数据集合根据某种方法，划分为几个小的数据集合，每个小的数据集合单独加载到内存来解决，然后再将小数据集合合并成大数据集合。实际上，利用这种分治的处理思路，不仅仅能克服内存的限制，还能利用多线程或者多机处理，加快处理的速度。
>
> 假设现在要给 10GB 的订单排序，我们就可以先扫描一遍订单，根据订单的金额，将10GB 的文件划分为几个金额区间。比如订单金额为 1 到 100 元的放到一个小文件，101 到 200 之间的放到另一个文件，以此类推。这样每个小文件都可以单独加载到内存排序，最后将这些有序的小文件合并，就是最终有序的 10GB 订单数据了。
>
> 如果订单数据存储在类似 GFS 这样的分布式系统上，当 10GB 的订单被划分成多个小文件的时候，每个文件可以并行加载到多台机器上处理，最后再将结果合并在一起，这样并行处理的速度也加快了很多。

分治代码模板：

```
private static int divide_conquer(Problem problem, ) {
  //问题终止条件
  if (problem == NULL) {
      //处理最小子问题的解
    int res = process_last_result();
    return res;
  }

    //将问题拆分成一个一个的重复子问题。
  subProblems = split_problem(problem)
  //下探到下一层求解子问题
  res0 = divide_conquer(subProblems[0])

  res1 = divide_conquer(subProblems[1])
```

```
    .....

    //将子问题的解合并变成最终问题的解
    result = process_result(res0, res1);


    //清理当前层状态等其他信息
      .......

    return result;
}
```

2.2、回溯

> 04年上映的《蝴蝶效应》，影片中讲述的是主人公为了实现自己的目标一直通过回退的方法回到童年，在一些重要的人生岔路口重新做出选择，最终实现整个美好人生的故事，当然了这只是电影，现实中人生是无法倒退的，但是这其中蕴含的就是思想就是我们要讲的回溯思想。

回溯算法实际上是一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。

回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的思想为回溯。

> 回溯法采用试错的思想，尝试分步的去解决一个问题。在分步解决问题的过程中，当它通过尝试发现现有的分步答案不能得到有效的正确答案时，它将取消上一步甚至上几步的计算（回退），再通过其他的可能的分步解答再次尝试寻找问题的解。

回溯法通常用最简单的递归方法来实现，在反复重复上面所讲的步骤后可能会出现以下两种情况：

1：找到了一个可能存在的正确答案

2：在尝试了所有可能的分步方法后宣告没有答案

在最坏情况下，回溯法会导致一次复杂度为指数时间的计算。

> 回溯算法是一种遍历算法，以 深度优先遍历 的方式尝试所有的可能性。有些教程上也叫「暴力搜索」。回溯算法是 有方向地 搜索，区别于多层循环实现的暴力法。

3、面试实战

50. Pow(x, n)

字节，快手最近面试题，50. Pow(x, n)

朴素解法：

```
class Solution {
    //暴力解法 两种情况n>=0;n<0;
    public double myPow(double x, int n) {
        double res = 1.0;
        int count = n > 0 ? n:-n;
        for (int i=0;i<count;i++) {
            res = res * x;
        }
        return n > 0 ? res:1/res;
    }
}
```

> 时间复杂度：O (n)

分治思想：`x^n = x^(n/2) * x^ (n/2)`

```
class Solution {
    //分治 x^n = x^(n/2) * x^ (n/2)
    public double myPow(double x, int n) {
        if (n<0) {
            return 1/recurPow(x,-n); // 2^(-10) = 1 / 2^10
        }else {
            return recurPow(x,n);
        }
    }

    public double recurPow(double x,int n) {
        //终止：最小的子问题
        if (n==0) {
            return 1.0;
        }
        if (n==1) {
            return x;
        }
        // 处理当前层逻辑，下探到下一层

        //大问题分解为小的子问题
        double subRes = recurPow(x,n/2);


        //合并子问题的解
        if (n%2==0) {
            return subRes * subRes;// x^n = x^(n/2) * x^ (n/2)
        }else {
            return subRes * subRes * x;
        }
    }

}
```

时间复杂度：O（logn）

扩展题目：69. x 的平方根

牛顿迭代法：http://www.matrix67.com/blog/archives/361

牛顿迭代法代码：http://www.voidcn.com/article/p-eudisdmk-zm.html

46. 全排列

字节，华为，腾讯，美团点评最近面试题，46. 全排列

我们在高中的时候就做过排列组合的数学题，我们也知道 `n` 个不重复的数，全排列共有 `n!` 个。

那么我们当时是怎么穷举全排列的呢?

比方说给三个数 `[1,2,3]`，你肯定不会无规律地乱穷举，一般是这样：

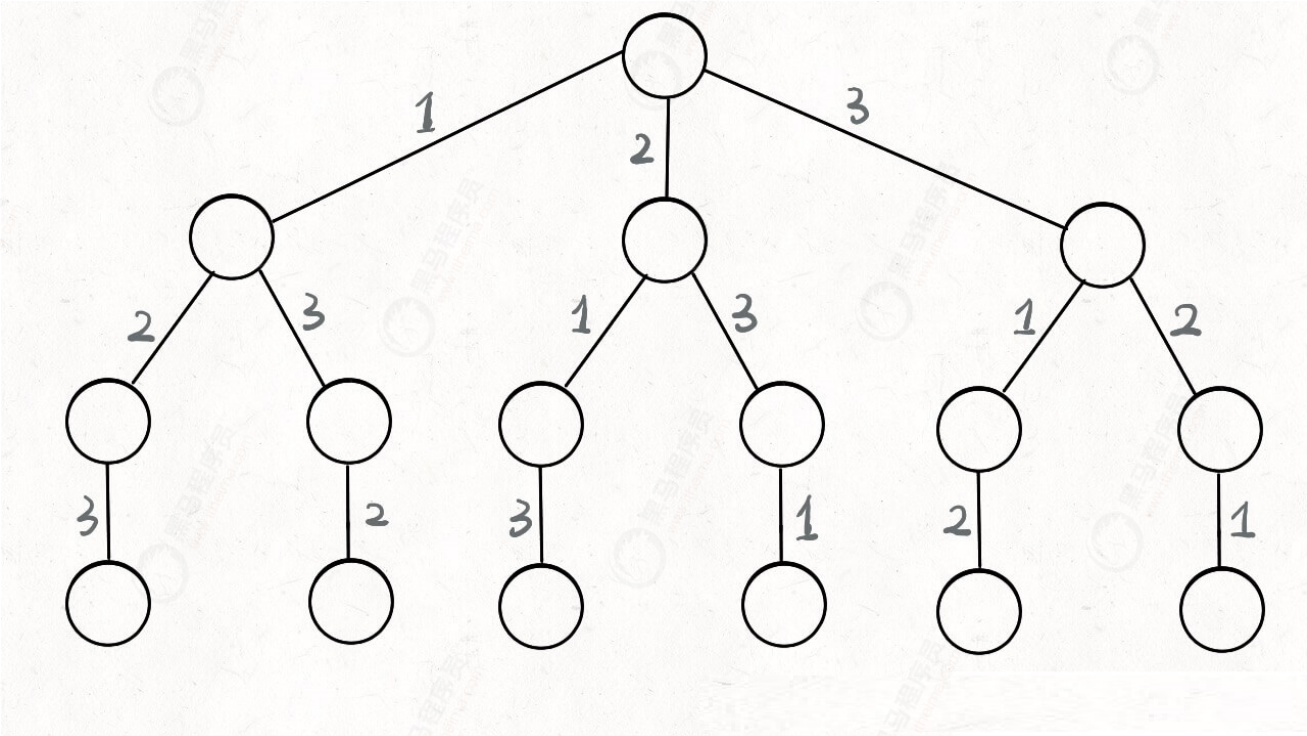
1：固定第一位为 1，然后第二位可以是 2，那么第三位只能是 3；然后可以把第二位变成 3，第三位就只能是 2 了；

2：固定第一位为2，然后再穷举后两位.....

3：固定第一位为3，然后再穷举后两位.....

其实这就是回溯算法。

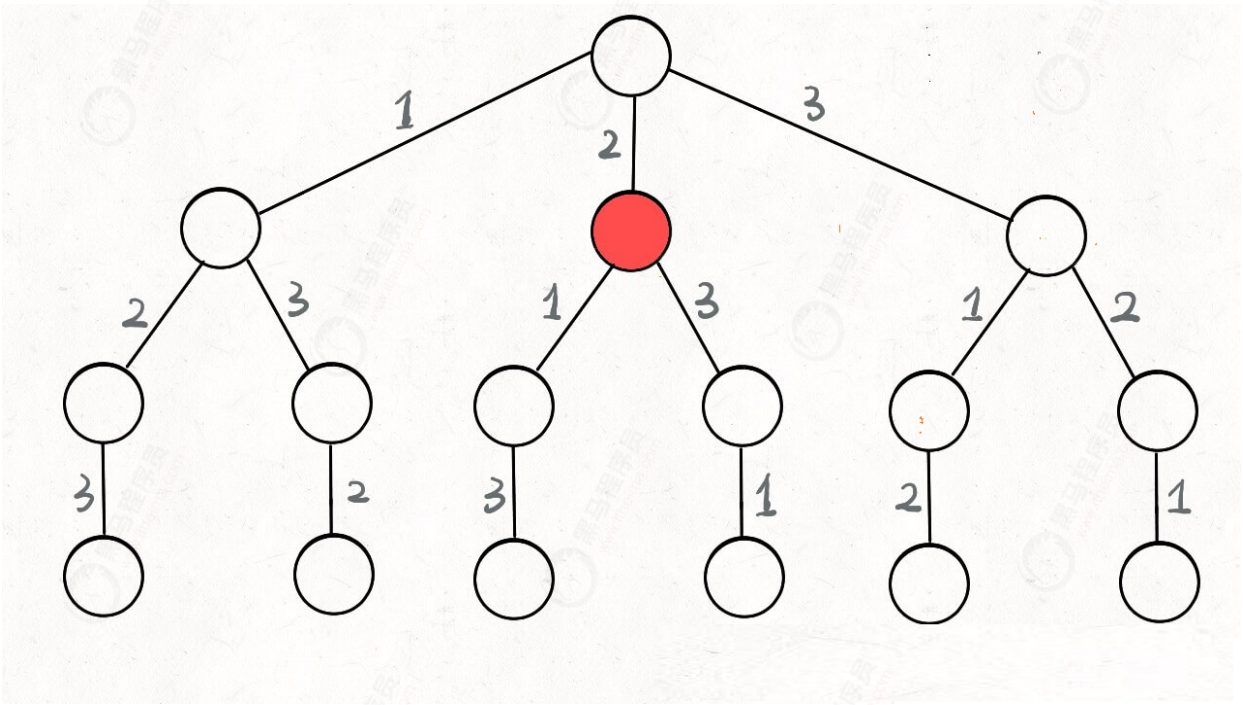
我们把刚刚穷举的过程画下来，会发现是一棵树



只要从根遍历这棵树，记录路径上的数字，其实就是所有的全排列结果。

我们把这棵树称为回溯算法的：**决策树**（递归状态树）

为什么说这是决策树呢，因为你在每个节点上其实都在做决策。比如说你站在下图的红色节点上：



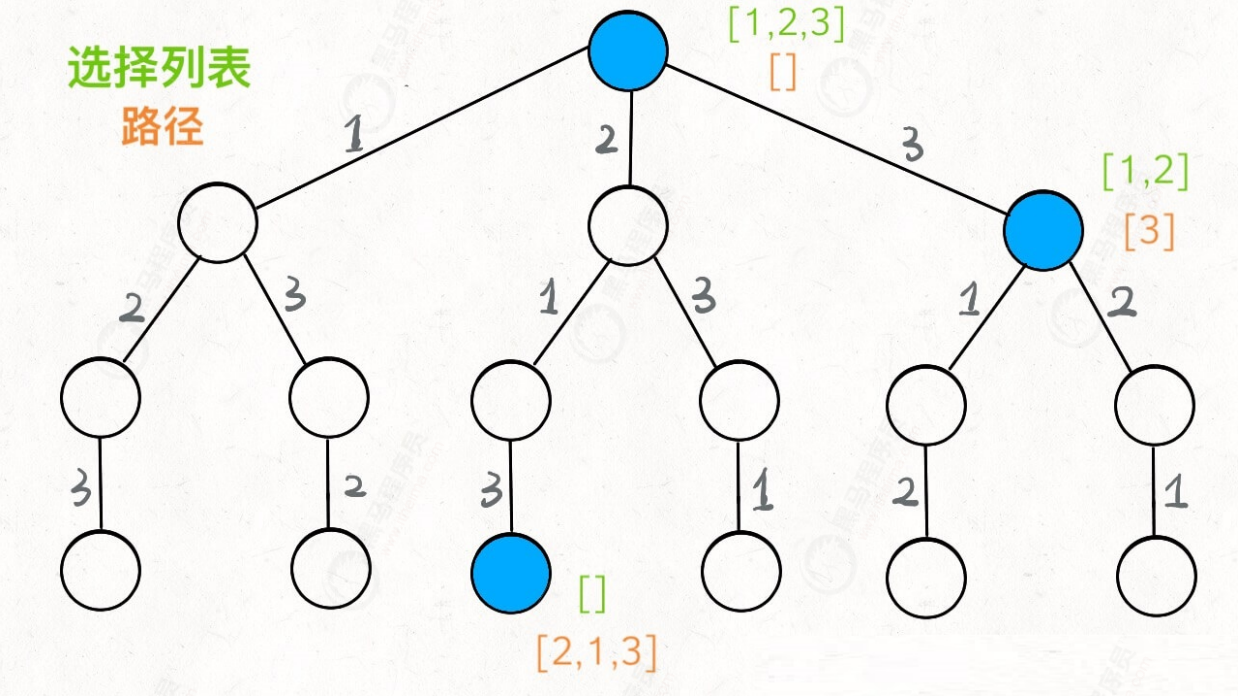
现在就需要做决策，可以选择 1 那条树枝，也可以选择 3 那条树枝。不选2的原因是因为 2 这个树枝在之前已经选择过了，而全排列是不允许重复使用的。

解决一个回溯问题，实际上就是一个决策树的遍历过程。只需要思考 3 个问题：

1、路径：用于记录我们已经做出的选择，走过的节点等等，比如`[2]`。

2、选择列表：表示当前可以做的选择，比如`[1,3]`

3、结束条件：也就是到达决策树底层，无法再做选择的条件。就是遍历到树的底层，在这里就是选择列表为空的时候（终止条件）

> 把路径和选择列表作为决策树上每个节点的属性，比如下图列出了几个节点的属性：
>
> 
>

回溯算法的代码框架如下：

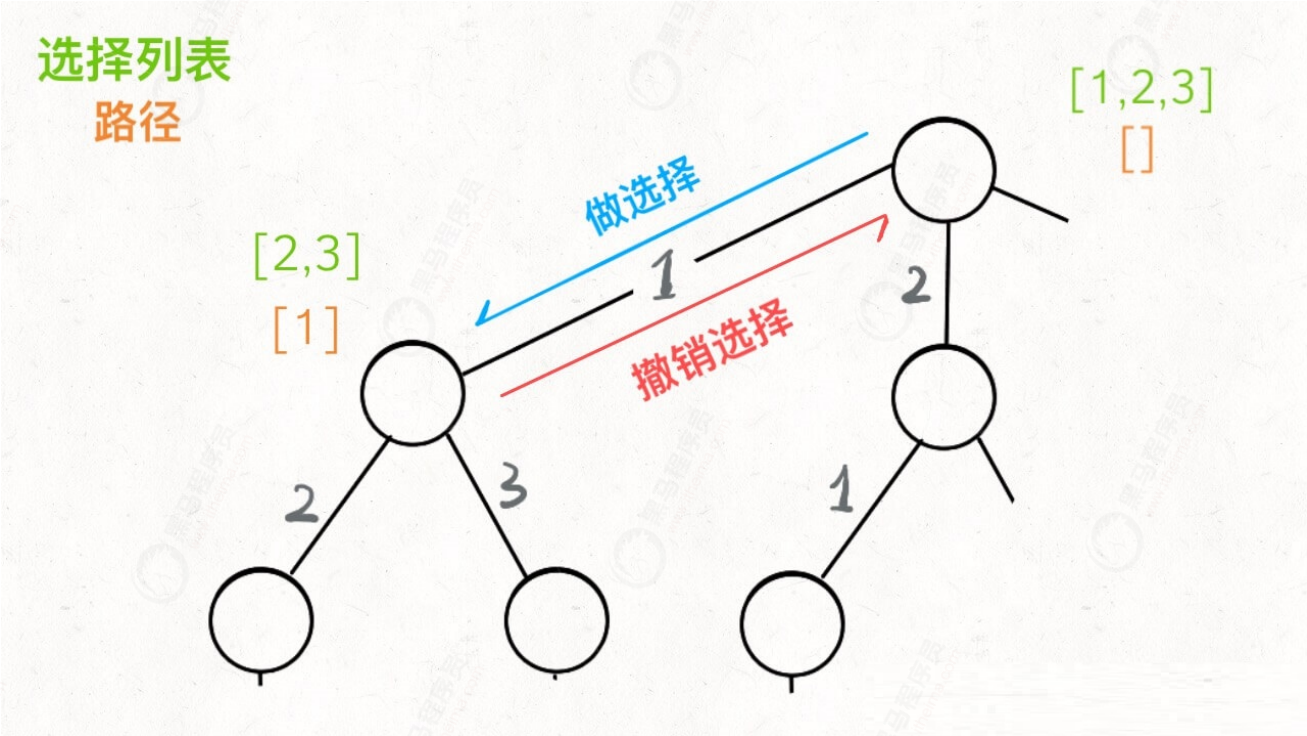
```
result = []
def backtrack(路径, 选择列表):
    if 满足结束条件:
        result.add(路径)
        return;

    for 选择 in 选择列表:
        // 做选择
        在选择列表中做出一个选择
        路径.add(选择)

        // 下探到下一层
        backtrack(路径, 选择列表)

        // 撤销选择
        路径.remove(选择)
        将该选择再加入选择列表
```

其核心就是 for 循环里面的递归，在递归调用之前「做选择」，在递归调用之后「撤销选择」。



下面我们来实现46，全排列代码

回溯+剪枝

```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> res = new ArrayList();
        //记录走过的路径
        Deque<Integer> path = new ArrayDeque();

        //记录是否已经选择过了,因为回溯要回到过去重新选择,必须要知道之前已经选择过哪些，因为可能重新选择的路径是之前已经选择过的,因此用一个boolean数组来进行剪枝,减少一些操作
        boolean[] visited = new boolean[nums.length];

        backTrack(nums,1,path,visited,res);
        return res;
    }

    public void backTrack(int[] nums,int level,Deque<Integer> path,boolean[] visited,List<List<Integer>> res) {
        //终止条件
        if (level > nums.length) {
            res.add(new ArrayList(path));
            return;
        }
        //从可选择列表中依次选择，可选择列表就是nums中的数据
        for (int i=0;i< nums.length;i++) {
            if (!visited[i]) { //剪枝，过滤掉不合法的结果(已经选择过的就不再选了)
                //做出选择
```

```
                path.addLast(nums[i]);
                visited[i] = true;

                //下探到下一层
                backTrack(nums,level+1,path,visited,res);

                //撤销本层选择,回溯发生的地点
                path.removeLast();
                visited[i] = false;
            }
        }

    }

}
```

注意这里的剪枝条件：如何过滤掉已经选择过的?