链滴

4 月 22 日 -5 月 7 日腾讯 nlp 算法实习面试题

作者：julyedu
原文链接：https://ld246.com/article/1621933713363
来源网站：链滴

评论有奖：评论区回复 “１１”，领取最新升级版《名企AI面试100题》电子书！！

本文目录：

答案解析：

问题8：介绍下bert位置编码和transformer的区别，哪个好，为什么?

Transformer解决并行计算问题的法宝，就是Positional Encoding，简单点理解就是，对于一句文本每一个词语都有上下文关系，而RNN类网络由于其迭代式结构，天然可以表达词语的上下文关系，但transformer模型没有循环神经网络的迭代结构，所以我们必须提供每个字的位置信息给transformer才能识别出语言中的顺序关系，为了解决这个问题，谷歌Transformer的作者提出了position encodig。

原版的Transformer中，谷歌对learned position embedding和sinusoidal position encoding进行对比实验，结果非常相近。而Sinusoidal encoding更简单、更高效、并可以扩展到比训练样本更长序列上，因此成为了Transformer的默认实现。对于机器翻译任务，encoder的核心是提取完整句子语义信息，它并不关注某个词的具体位置是什么，只需要将每个位置区分开（三角函数对相对位置有助）；而Bert模型对于序列标注类的下游任务，是要给出每个位置的预测结果的。

问题9：sigmod函数的缺点，为什么会产生梯度消失？不是以0为中心的话，为什么会收敛慢。

sigmoid函数特点：它能够把输入的连续实值变换为0和1之间的输出，特别的，如果是非常大的负数那么输出就是0；如果是非常大的正数，输出就是1。

缺点：缺点1：在深度神经网络中梯度反向传递时导致梯度消失，其中梯度爆炸发生的概率非常小，梯度消失发生的概率比较大。缺点2：Sigmoid 的 output不是0均值（即zero-centered）。缺点3：解析式中含有幂运算，计算机求解时相对来讲比较耗时。对于规模比较大的深度网络，这会较大地增训练时间。

问题10：Layer Normalization和Batch Normalization 的区别

Batch Normalization 是对这批样本的同一维度特征做归一化， Layer Normalization 是对这单个样的所有维度特征做归一化。

BatchNorm的缺点：

1、需要较大的batch以体现整体数据分布2、训练阶段需要保存每个batch的均值和方差，以求出整均值和方差在infrence阶段使用3、不适用于可变长序列的训练，如RNNLayer NormalizationLayer ormalization是一个独立于batch size的算法，所以无论一个batch样本数多少都不会影响参与LN计的数据量，从而解决BN的两个问题。

LN的做法是根据样本的特征数做归一化。LN不依赖于batch的大小和输入sequence的深度，因此可用于batch-size为1和RNN中对边长的输入sequence的normalize操作。但在大批量的样本训练时，果没BN好。实践证明，LN用于RNN进行Normalization时，取得了比BN更好的效果。但用于CNN，效果并不如BN明显。

问题11：Leetcode 8. 字符串转换整数 (atoi)，考虑科学计数法解析：

有三种方法：正常遍历、有限状态机和正则表达式，这里只提供正则表达式的参考答案。

s.lstrip()表示把开头的空格去掉

使用正则表达式：

^：匹配字符串开头

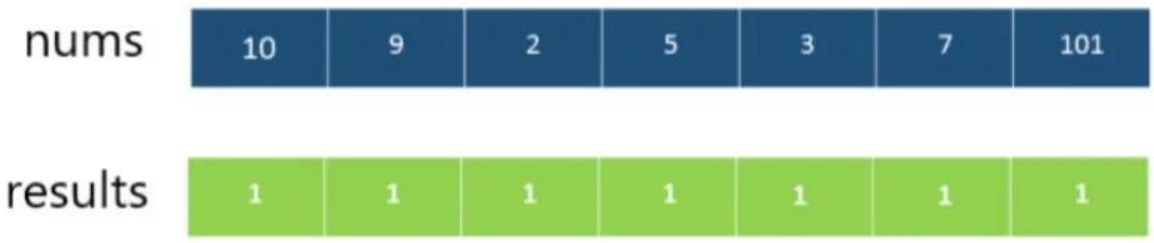
[+-]：代表一个+字符或-字符?：前面一个字符可有可无

\d：一个数字 +：前面一个字符的一个或多个max(min(数字, 2**31 - 1), -2**31) 用来防止结果越界

代码如下：

```python
class Solution:
    def myAtoi(self, s: str) -> int:
        return max(min(int(*re.findall('^[\+\-]?\d+', s.lstrip())), 2**31 - 1), -2**31)
```

问题12：最长递增子序列



如上图所示，用nums表示原数组，results[i]表示截止到nums[i] 当前最长递增子序列长度为results[i，初始值均为1；

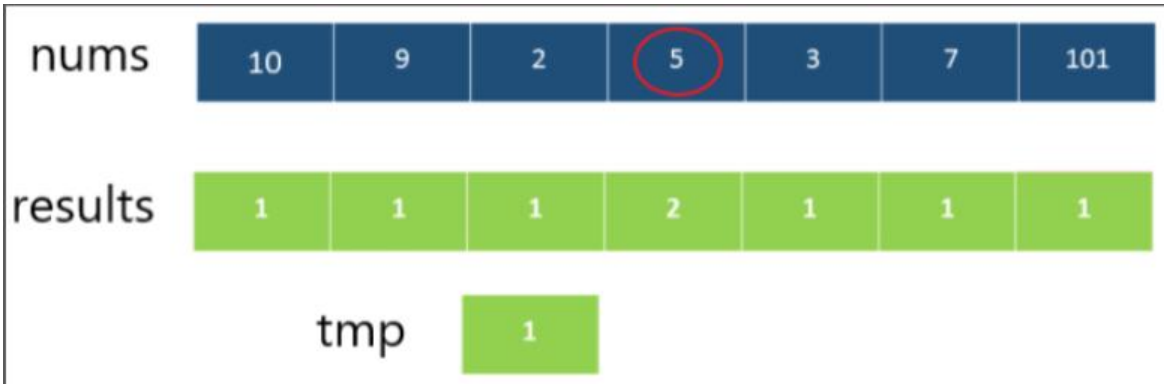
因为要找处递增序列，所以我们只需要找出nums[i]>nums[j]（其中j < i）的数，并将对应的results[j保存在tmp临时列表中，然后找出最长的那个序列将nums[i]附加其后面；示例：假设nums[i] = 5, i = 3

更新results[i]时只需要考虑前面小于results[i]的项，所以results[0-2]均为1；

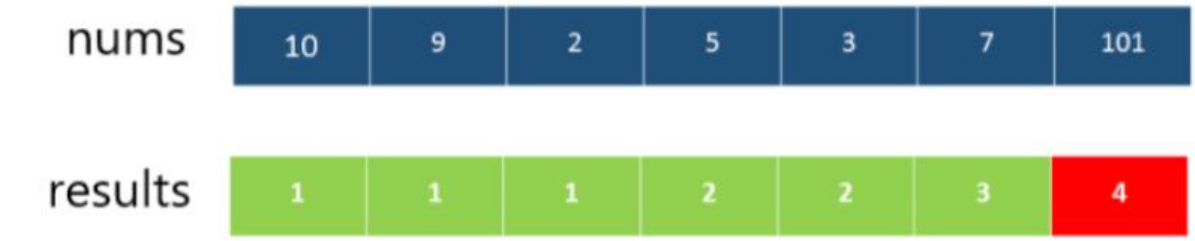
此时nums[i]前面小于5的项有nums[2]，需要将results[2]保存报tmp中，然后选取tmp中最大的数值

加1后赋值给results[i], 即results[5] = 2

tmp是临时列表更新results之后需要清空；



根据上述流程更新所有results[i]，即可得到下图：



此时我们只需要找出results中最大的那个值，即为最长递增子序列的长度。

参考代码如下：

```python
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        # 特殊情况判断
        if len(nums) == 0: return 0

        results = [1]*len(nums)
        for i in range(1, len(nums)):
            tmp = []
            for j in range(i):
                if nums[i] > nums[j]:
                    tmp.append(results[j])
            if tmp: results[i] = max(tmp) + 1
        return max(results)
```

问题13：全排列

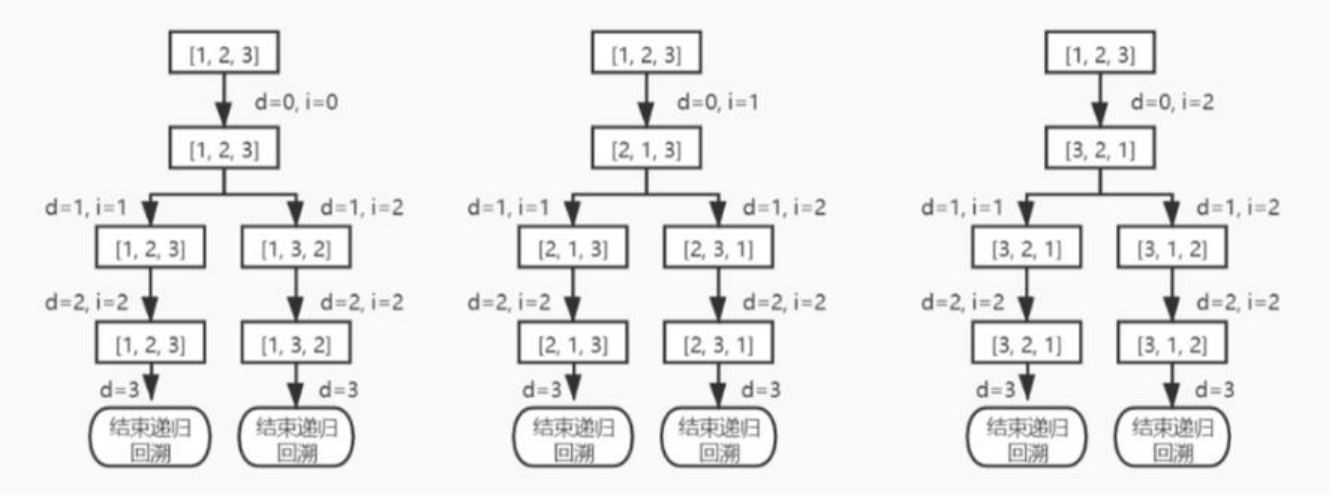
解析1：定义一个长度为len(nums)的空表output，从左往右一次填入nums中的数字，并且每个数只使用一次。可以枚举所有困难，从左往右每一个位置都一次填入一个数，用used表记录nums[i]是否填入output中，如果nums[i]木有填入output中则填入并标记，当output的长度为len(nums)时，得一个满足条件的结果，将output存入最终结果。然后将output回溯到未添加nums[i]状态，used回溯

未标记nums[i]状态，继续下一次尝试。

```python
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        # 特殊判断
        if len(nums) == 0: return nums
        result = []
        depth, n, output, used = 0, len(nums), [], [False]*len(nums)
        self.backtrack(depth, n, output, used, nums, result)
        return result

    def backtrack(self, depth, n, output, used, nums, result):
        # 递归终止条件:
        if depth == n:
            result.append(output[:])
        else:
            # 遍历数组中的每一个数
            for i in range(0, n):
                # 如果该数字已使用(已在output中)， 则进入下一个数字
                if used[i]: continue
                # 该数字还未出现在output中，则添加并标记
                output.append(nums[i])
                used[i] = True
                # 递归进入下一层，选择下一个数字
                self.backtrack(depth+1, n, output, used, nums, result)
                # 回溯操作：去除添加、去除标记
                output.pop()
                used[i] = False
```

解析2：在解析1中，我们使用了两个辅助空间，output和used；为节省空间，我们可以除掉他们；们可以将给定的nums数组分成左右两个部分，用n表示数组个数，depth表示当前需要填充的位置，边表示已经填好的内容[0~~depth)，右边表示待填充的内容[depth~~n)，假设前填充的位置是i，填充后为了保持上述结构，需要nums[i]和nums[depth]交换，在完成一次填充后回溯过程，还需要再次交换，保持原有内容。整个流程如下图所示：



原文链接：4 月 22 日 -5 月 7 日腾讯 nlp 算法实习面试题

参考代码：

```python
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        if len(nums) == 0 : return nums
        ret = []
        depth, n = 0, len(nums)
        self.backtrack(nums, ret, depth, n)
        return ret
    # 递归
    def backtrack(self, nums, ret, depth, n):
        # 递归终止条件
        if depth == n:
            ret.append(nums[:])
        else:
            # depth表示nums中第depth位置及之后的内容是需要确认的
            for i in range(depth, n):
                nums[depth], nums[i] = nums[i], nums[depth]
                # 递归
                self.backtrack(nums, ret, depth+1, n)
                # 递归之后回溯得到原始数组
                nums[depth], nums[i] = nums[i], nums[depth]
```

问题14：合并两个有序数组并去重

```python
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        """
        Do not return anything, modify nums1 in-place instead.
        """
        # 方法1:  直接合并，然后排序
        # nums1[m:] = nums2
        # nums1.sort()

        # 方法2：双指针法
        # 因为两个列表均已是有序，所以可以比较nums1和nums2的首端元素，将较小者插入到前面
        # p1:nums1的指针  p2:nums2的指针
        p1, p2 = 0, 0
        nums1_ = [nums1[i] for i in range(m)]

        i = 0
        while p1 <= m and p2 <= n:
            # 判断nums1是否已遍历完
            if p1 == m:
                nums1[m+p2:] = nums2[p2:]
                break
            # 判断nums2是否已遍历完
            elif p2 == n:
                nums1[n+p1:] = nums1_[p1:]
                break

            elif nums1_[p1] <= nums2[p2]:
                nums1[i] = nums1_[p1]
                p1, i = p1+1, i+1
            else:
                nums1[i] = nums2[p2]
                p2, i = p2+1, i+1

```