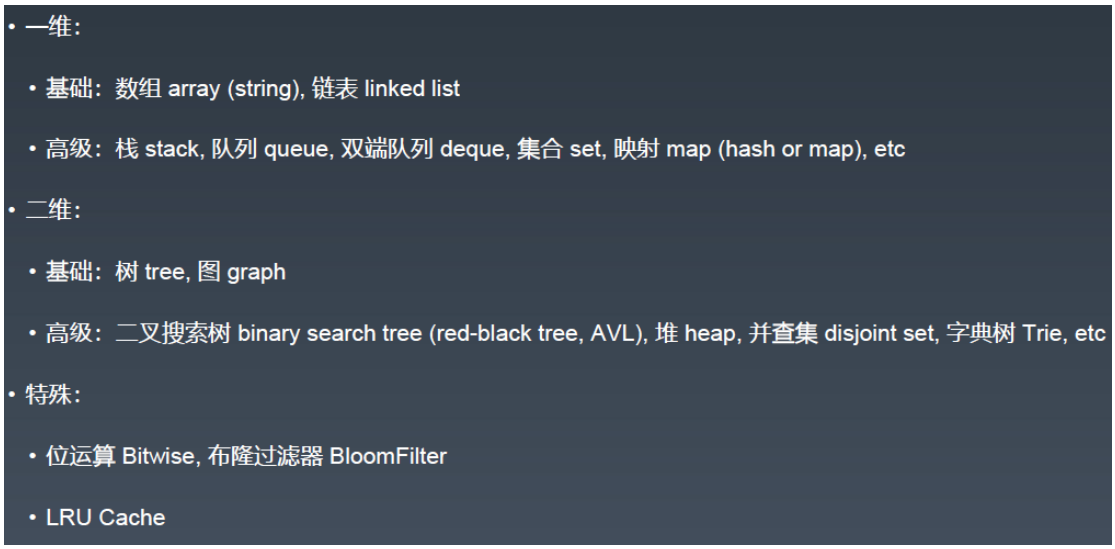
极客时间算法面试突击营 毕业总结

学号：G20210712010097

训练营的目的是帮助学员提前接触并掌握在面试时可能会遇到的算法题目。本总结的写作理念是从解题方法入手，归纳总结常见算法的解决方法并帮助强化记忆，供日后复习以及在面试前增强记忆。

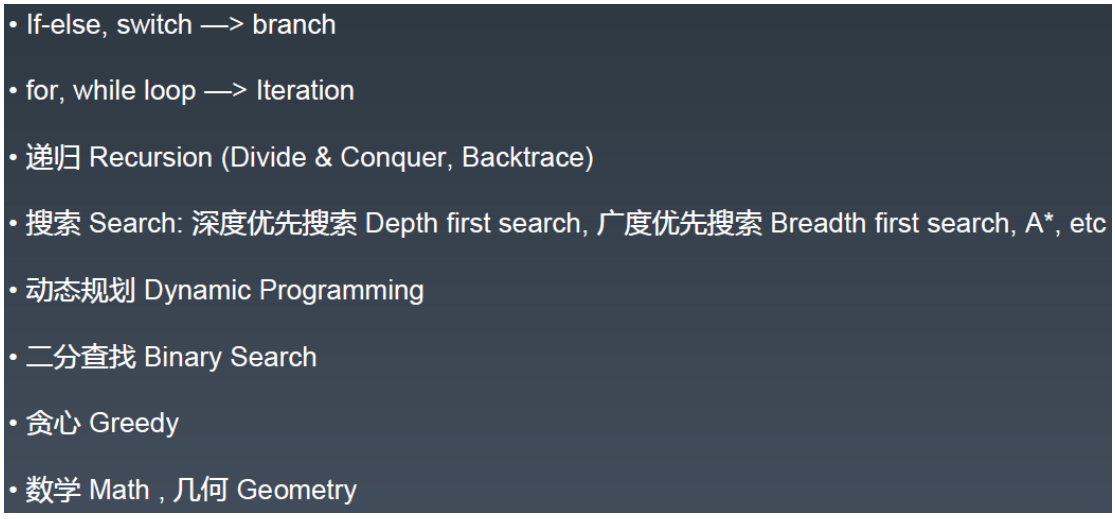
1. 常见的数据结构



提醒：

1. 在遇到链表类题目时，优先考虑递归的方法，因为链表本来就是用递归思想来定义的；
2. 在定义动态规划的 dp array 时，一定要记住，如果感觉一维数组不能帮助解决问题，要有升维的思想（一维到二维）；
3. 熟悉 Java 里面的 PriorityQueue 和 ArrayDeque。

2. 常见的算法



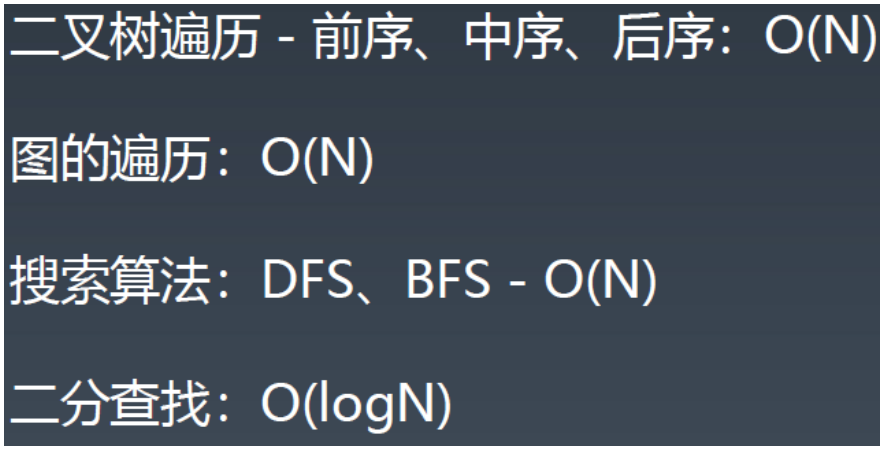
看起来很复杂的代码逻辑，都是由选择，循环和递归三种语法组成，因为计算机只懂得这几种语法逻辑。

在写 for 循环时要注意边界条件，比如说 dp[0]是否可以提前定义，以及是否可以循环到最后一个值 array.length。

3. 注意算法的时间复杂度！目标是通过改变编程方法来不断降低算法的时间复杂度！

对数时间复杂度会比线性快很多!

注意下列算法的时间复杂度:

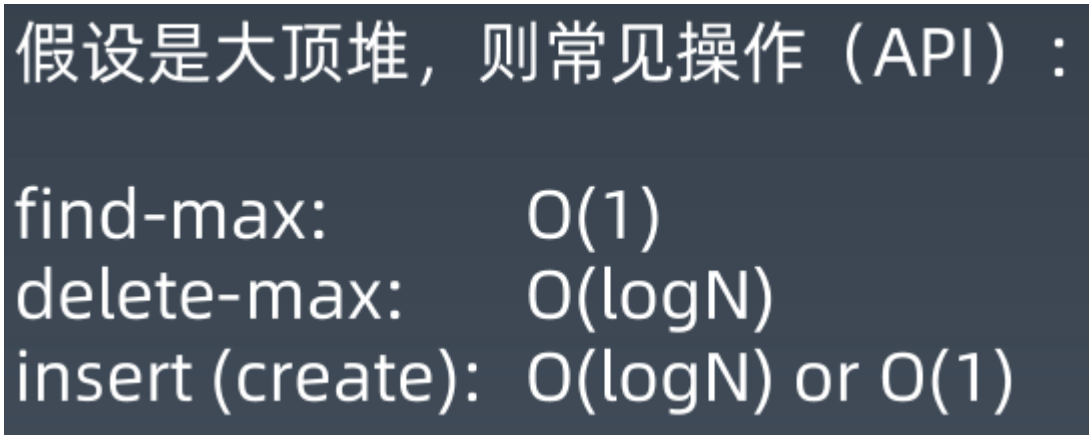


4. 算法题里有一大类，是对链表的操作，比如增加、删除、查找节点，或者反转链表。比如反转链表，可以用迭代，或者递归的方法:

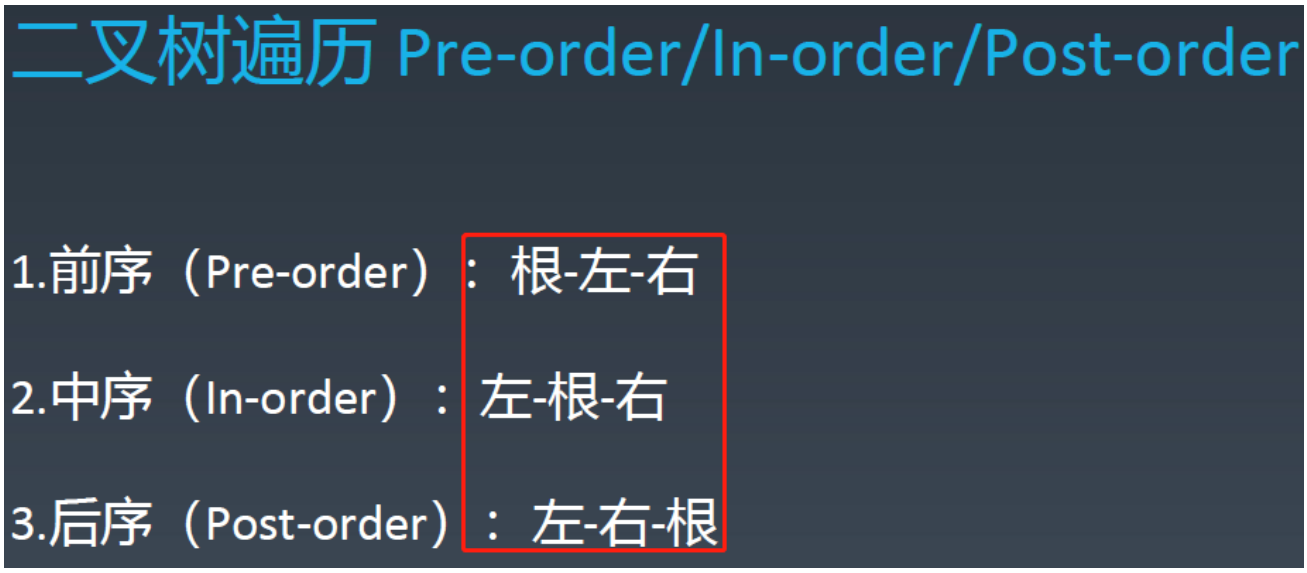
```
// 方法一 迭代
private ListNode iterationM(ListNode head){
    ListNode x= null,y=head;
    while(y!=null){
        ListNode t = y.next;
        y.next = x;
        x=y;
        y=t; // 这样就可以交换x和y的位置
    }
    return x;
}

// 方法二 递归 (找最小子问题)
private ListNode recursionM(ListNode head){
    // 结束条件
    if(head == null || head.next == null) return head;
    // 处理这一层
    ListNode res = recursionM(head.next);
    head.next.next = head; // 翻转
    head.next = null; // 它自己指向空
    // 进入下一层
    // 重置相关变量
    return res;
}
```

5. 栈和队列的数据结构，也很有用，常常跟 DFS 以及 BFS 结合。
   栈（Stack）：First-in Last-out
   队列（Queue）：First-in First-out, Last-in Last-out
   Deque: Double-ended Queue（两端都可以进出）
   Java 官方推荐，直接用 Deque 来替代 Stack：
   Deque<Integer> stack = new ArrayDeque<Integer>();
   Queue 和 Deque 的继承子类：LinkedList<E>
   Queue 的继承子类：PriorityQueue<E>
6. 哈希表 -- **记录在表中的位置和其关键字之间存在着一种确定的关系，所以查找非常快！**
   查找，增加，删除的时间复杂度都是 O(1)。
7. 堆：可以迅速找到一堆数中的最大或者最小值（大顶堆或小顶堆）的数据结构。



8. 二叉树的前、中、后序遍历（前中后是以根节点为基准确定的）



   树的解题方法一般都是递归，因为树本身就是靠递归来定义的。
9. 递归代码模板
   9.1 先写终止条件
   9.2 再处理当前层
   9.3 可能要把当前层的结果，带入到下一层
   9.4 看看是否需要 reset 一些条件（dfs 的时候常用）

Java 代码模版

```java
public void recur(int level, int param) {

    // terminator
    if (level > MAX_LEVEL) {
      // process result
      return;
    }

    // process current logic
    process(level, param);

    // drill down
    recur( level: level + 1, newParam);

    // restore current status

}
```

1. 不要人肉进行递归（最大误区）
2. 找到最近最简方法，将其拆解成可重复解决的问题（重复子问题）
3. 数学归纳法思维

找到重复子问题--关键思维点!

10. 分治(Divide & Conquer)！回溯(Backtracking)！

分治代码模板:

```python
def divide_conquer(problem, param1, param2, ...):
  # recursion terminator
  if problem is None:
    print_result
    return
  # prepare data
  data = prepare_data(problem)
  subproblems = split_problem(problem, data)
  # conquer subproblems
  subresult1 = self.divide_conquer(subproblems[0], p1, ...)
  subresult2 = self.divide_conquer(subproblems[1], p1, ...)
  subresult3 = self.divide_conquer(subproblems[2], p1, ...)
  ...
  # process and generate the final result
  result = process_result(subresult1, subresult2, subresult3, …)

  # revert the current level states
```

还是与递归类似的!

回溯法采用试错的思想，它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的解答的时候，它将取消上一步甚至是上几步的计算，再通过其它的可能的分步解答再次尝试寻找问题的答案。 通常是用最简单的递归方法来实现

11. **深度优先搜索（Depth First Search）和广度优先搜索（Breadth First Search）**

dfs 示例代码：

```
def dfs(node):

    if node in visited:
        # already visited
        return

    visited.add(node)

    # process current node
    # ... # logic here
    dfs(node.left)
    dfs(node.right)
```

一般都要记录下已经访问过的！

```
visited = set()

def dfs(node, visited):
    if node in visited: # terminator
        # already visited
        return

    visited.add(node)

    # process current node here.
    ...
    for next_node in node.children():
        if not next_node in visited:
            dfs(next node, visited)
```

递归写法，需要定义已经访问的集合，已经访问过的元素要放入集合。

```
def DFS(self, tree):

    if tree.root is None:
        return []

    visited, stack = [], [tree.root]

    while stack:
        node = stack.pop()
        visited.add(node)

        process (node)
        nodes = generate_related_nodes(node)
        stack.push(nodes)

    # other processing work
    ...
```

这个是非递归写法！

bfs 示例代码：

```
def BFS(graph, start, end):

    queue = []
    queue.append([start])
    visited.add(start)

    while queue:
        node = queue.pop()
        visited.add(node)

        process(node)
        nodes = generate_related_nodes(node)
        queue.push(nodes)

    # other processing work
    ...
```

需要用到queue这种结构！

12. 贪心算法！

贪心算法 Greedy

不看之前的运算结果，不回退！

贪心算法是一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是全局最好或最优的算法。

贪心法可以解决一些最优化问题，如：求图中的最小生成树、求哈夫曼编码等。然而对于工程和生活中的问题，贪心法一般不能得到我们所要求的答案。

适用于解决一些对结果要求不是那么精确的问题！

适用贪心算法的场景

简单地说，问题能够分解成子问题来解决，子问题的最优解能递推到最终问题的最优解。这种子问题最优解称为最优子结构。

13. 二分查找！

1. 目标函数单调性（单调递增或者递减）

2. 存在上下界（bounded）

3. 能够通过索引访问（index accessible)

这三点是能使用二分查找算法的前提！

```
left, right = 0, len(array) - 1
while left <= right:
    mid = (left + right) / 2
    if array[mid] == target:
        # find the target!!
        break or return result
    elif array[mid] < target:
        left = mid + 1
    else:
        right = mid - 1
```

二分查找代码模板

14. 动态规划

分治 + 回溯 + 递归 + 动态规划

这些方法是一定要掌握的！

递归代码模版

```
public void recur(int level, int param) {

    // terminator
    if (level > MAX_LEVEL) {
      // process result
      return;
    }

    // process current logic
    process(level, param);

    // drill down
    recur( level: level + 1, newParam);

    // restore current status

}
```

1. 人肉递归低效、很累

2. 找到最近最简方法，将其拆解成可重复解决的问题

3. 数学归纳法思维（抵制人肉递归的诱惑）

本质：寻找重复性 —> 计算机指令集

动态规划 和 递归或者分治 没有根本上的区别（关键看有无最优的子结构）

共性：找到重复子问题

差异性：最优子结构、中途可以**淘汰**次优解

状态转移方程（DP 方程）

opt[i , j] = opt[i + 1, j] + opt[i, j + 1]

完整逻辑：

if a[i, j] = ‘空地’:

  opt[i , j] = opt[i + 1, j] + opt[i, j + 1]

else:

  opt[i , j] = 0

时刻注意要有“升维”的思想！

动态规划关键点

1. 最优子结构 opt[n] = best_of(opt[n-1], opt[n-2], ...)
2. 储存中间状态：opt[i]
3. 递推公式（美其名曰：状态转移方程或者 DP 方程）

   Fib: opt[i] = opt[n-1] + opt[n-2]

   二维路径：opt[i,j] = opt[i+1][j] + opt[i][j+1] (且判断a[i,j]是否空地)

动态规划小结

像计算机一样去思考问题！

1. 打破自己的思维惯性，形成机器思维
2. 理解复杂逻辑的关键
3. 也是职业进阶的要点要领

15. 位运算

- 位运算符
- 算数移位与逻辑移位
- 位运算的应用

异或：相同为 0，不同为 1。也可用“不进位加法”来理解。

异或操作的一些特点：

```
x ^ 0 = x
x ^ 1s = ~x     // 注意 1s = ~0
x ^ (~x) = 1s
x ^ x = 0
c = a ^ b  =>  a ^ c = b, b ^ c = a    // 交换两个数
a ^ b ^ c = a ^ (b ^ c) = (a ^ b) ^ c    // associative
```

指定位置的位运算

1. 将 x 最右边的 n 位清零：x & (~0 << n)
2. 获取 x 的第 n 位值（0 或者 1）：(x >> n) & 1
3. 获取 x 的第 n 位的幂值：x & (1 << n)
4. 仅将第 n 位置为 1：x | (1 << n)
5. 仅将第 n 位置为 0：x & (~ (1 << n))
6. 将 x 最高位至第 n 位（含）清零：x & ((1 << n) - 1)

实战位运算要点

- 判断奇偶：
  x % 2 == 1 —> (x & 1) == 1
  x % 2 == 0 —> (x & 1) == 0
- x >> 1 —> x / 2.
  即：x = x / 2; —> x = x >> 1;
  mid = (left + right) / 2; —> mid = (left + right) >> 1;
- X = X & (X-1) 清零最低位的 1
- X & -X => 得到最低位的 1
- X & ~X => 0

位运算符	含义
<<	左移
>>	右移
|	按位或
&	按位与
~	按位取反
^	按位异或

16. 排序（掌握两种高级排序!）

- 快速排序（Quick Sort） O(N*LogN)

  数组取标杆 pivot，将小元素放 pivot左边，大元素放右侧，然后依次对右边和右边的子数组继续快排；以达到整个序列有序。

- 归并排序（Merge Sort）— 分治 O(N*LogN)

  1. 把长度为n的输入序列分成两个长度为n/2的子序列；
  2. 对这两个子序列分别采用归并排序；
  3. 将两个排序好的子序列合并成一个最终的排序序列。

快排代码 - Java

```java
public static void quickSort(int[] array, int begin, int end) {
    if (end <= begin) return;
    int pivot = partition(array, begin, end);
    quickSort(array, begin, pivot - 1);
    quickSort(array, pivot + 1, end);
}

static int partition(int[] a, int begin, int end) {
    // pivot: 标杆位置, counter: 小于pivot的元素的个数
    int pivot = end, counter = begin;
    for (int i = begin; i < end; i++) {
        if (a[i] < a[pivot]) {
            int temp = a[counter]; a[counter] = a[i]; a[i] = temp;
            counter++;
        }
    }
    int temp = a[pivot]; a[pivot] = a[counter]; a[counter] = temp;
    return counter;
}
```

一开始标杆的位置是可以随便选的!

调用方式: `quickSort(a, 0, a.length - 1)`

归并排序代码 - Java

```java
public static void mergeSort(int[] array, int left, int right) {

    if (right <= left) return;
    int mid = (left + right) >> 1; // (left + right) / 2

    mergeSort(array, left, mid);
    mergeSort(array, mid + 1, right);
    merge(array, left, mid, right);

}
```

```java
public static void merge(int[] arr, int left, int mid, int right) {

    int[] temp = new int[right - left + 1]; // 中间数组
    int i = left, j = mid + 1, k = 0;

    while (i <= mid && j <= right) {
        temp[k++] = arr[i] <= arr[j] ? arr[i++] : arr[j++];
    }

    while (i <= mid)   temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];

    for (int p = 0; p < temp.length; p++) {
        arr[left + p] = temp[p];
    }
    // 也可以用 System.arraycopy(a, start1, b, start2, length)
}
```

注意这个写法！一定要有最后这个循环赋值！

归并：先排序左右子数组，然后合并两个有序子数组

归并排序 和 快速排序 步骤相反！

快排：先调配出左右子数组，然后对于左右子数组进行排序

17. 对字符串的处理！

    注意：在 Java 里，字符串是 immutable！

    通过大量做相关题目来掌握这一知识点！