**1.大数与密码学**

调用pow函数计算p的1/n次幂即可

2.两数组之间的距离值

在这个代码中，calculateDistance 函数接受两个整数数组 arr1 和 arr2 以及一个整数 d 作为输入，并返回两个数组之间的距离值。它首先初始化一个计数器 distance 为 0，然后遍历 arr1 中的每个元素 num1。对于每个 num1，它遍历 arr2 中的每个元素 num2，并检查 std::abs(num1 - num2) 是否小于或等于 d。如果是，则设置 isWithinRange 为 true 并跳出内层循环。如果 arr2 中没有元素与 num1 的差的绝对值小于或等于 d，则 isWithinRange 保持为 false，并且在外部循环中增加 distance 计数器。最后，函数返回 distance。

这个实现的时间复杂度是 O(n\*m)，其中 n 是 arr1 的长度，m 是 arr2 的长度。如果数组很大，这个实现可能会很慢。为了优化性能，可以使用排序和二分查找等算法来减少内层循环的复杂度，但这取决于具体的数据分布和查询需求。

3.奇怪的子序列

首先题目中的条件是含有位运算的，我们就可以从这里开始入手优化。

因为题目中给的条件是 a i&a i−1 =0，所以这就证明了只要两个数之间有一个二进制位上相同的 1 就可以让长度加一，那么我们就可以枚举每一个二进制位，找出他们与运算后是否为 0，如果不是 0 就让长度加一，我们的一个状态转移方程就出来了：

Max=max(dp[c]+1,Max)

其中 dp[c] 是二进制第 c 位时最大的长度，当前状态下满足条件就加一，然后与最大长度 Max 取最大值。

需要注意的一个地方就是我们在状态转移后需要每次更新一下 dp[c] ，因为当前满足条件的最长长度已经找出来是 Max 了，所以所有当前满足条件的二进制位的最大长度都变为 1，那么就又有一个转移：

dp[c]=Max

最后再统计一下所有更新后 Max 的最大值就好了。

4.火星上的比赛

声明并初始化两个字符数组a和b，用于存储输入的火星数字。

使用memset函数将数组中的每个元素初始化为一个特殊字符（这里用'x'），便于后续计算长度。

通过cin从标准输入读取两个火星数字。

数字倒置与长度计算：

由于字符串在数组中的存储是正向的，而计算需要从最低位开始，因此需要将数组中的数字倒置。

使用循环遍历数组，计算每个火星数字的长度，并在遍历过程中将数字倒置。

相加操作：

声明一个字符数组sum用于存储相加的结果。

初始化进位值add为'0'。

使用循环遍历倒置后的a和b数组，进行逐位相加操作。

将字符数字转换为整数进行计算。

计算当前位的和以及进位值。

根据和的值，转换为相应的字符并存入sum数组。

如果最高位有进位，需要增加sum数组的长度。

去除前导零：

在输出结果前，需要去除sum数组中的前导零。

使用循环从数组末尾开始向前遍历，找到第一个非零字符的位置，确定输出长度的起始位置。

输出结果：

从确定的位置开始，逆序输出sum数组中的字符，即为两个火星数字的和。

清理：

在输出结束后，使用memset函数将a和b数组重置为初始状态，为可能的后续操作做准备。

算法的主要特点是：

通过倒置数组，将字符串的读取顺序与数学计算的顺序对齐，简化了计算过程。在相加过程中，通过转换字符为整数进行计算，然后再转换回字符，实现了以20为基数的加法运算。特别注意了最高位进位和去除前导零的处理，确保输出的正确性。

5.电力建设者小陈

这个问题实际上是一个最小生成树问题。因为我们需要连接所有村庄，并确保连接的总费用最小，这符合最小生成树问题的定义。在最小生成树问题中，常用的算法有Prim算法和Kruskal算法。由于题目中并没有限制边的数量，我们可以选择Kruskal算法来解决这个问题。

Kruskal算法的基本思路是从小到大选择边，如果这条边连接的两个顶点不在同一个集合中（即它们还没有被连接起来），那么就选择这条边，并将两个顶点合并到同一个集合中。重复这个过程，直到所有的顶点都在同一个集合中，或者已经没有边可以选择为止。

我们可以将每个村庄看作一个顶点，村庄之间的距离和高度差平方和看作边的权重。然后，我们使用Kruskal算法找到最小生成树，这个最小生成树的权重和就是我们需要的最小费用。

首先，我们需要将输入的村庄信息（坐标和高度）转化为边的信息，即计算每对村庄之间的距离和高度差平方和，作为边的权重。

然后，我们对所有的边按照权重进行排序，从小到大选择边。

在选择边的过程中，我们使用并查集来判断这条边连接的两个村庄是否已经在同一个集合中（即是否已经连通）。

如果两个村庄不在同一个集合中，我们选择这条边，并将两个村庄合并到同一个集合中。同时，我们累加这条边的权重到总费用中。

重复步骤3和4，直到所有的村庄都在同一个集合中，或者已经没有边可以选择为止。

最后，输出总费用，保留两位小数。

由于题目中村庄的数量n最大为1000，边的数量最大为n\*(n-1)/2，因此我们需要使用高效的排序算法对边进行排序，以避免超时。在实际编程中，可以使用STL中的sort函数进行排序。此外，由于需要频繁地合并和查询集合，我们可以使用并查集（Union-Find）数据结构来优化这个过程。

6.交叠链表 II（题解不通过）

本题要求合并两个链表，合并的方式是将两个链表的节点交替取出，直到其中一个链表遍历完为止。如果某个链表还有剩余节点，则将剩余节点直接添加到新链表的末尾。

为了完成这个任务，我们可以遵循以下步骤：

创建新链表的头节点LL，头节点不存储数据，只是为了方便后续操作。

创建两个指针p1和p2，分别指向两个输入链表的第一个有效节点（即头节点后的第一个节点）。

使用一个指针r来构建新链表，初始时指向新链表的头节点LL。

进入循环，每次循环中：

将p1所指向的节点的值添加到新链表中，并将r移动到新添加的节点上。

将p2所指向的节点的值添加到新链表中，并将r移动到新添加的节点上。

将p1和p2分别向前移动一个节点。

循环直到p1或p2其中一个为NULL，即一个链表遍历完为止。

如果p1不为NULL（即链表1还有剩余节点），则将链表1的剩余节点依次添加到新链表的末尾。

如果p2不为NULL（即链表2还有剩余节点），则将链表2的剩余节点依次添加到新链表的末尾。

最后，将新链表的最后一个节点的next指针置为NULL，表示链表结束。

返回新链表的头节点LL。

算法采用了双指针法，通过交替取出两个链表的节点来构建新链表。算法的时间复杂度为O(n+m)，其中n和m分别为两个输入链表的长度，因为每个节点只被访问和处理一次。空间复杂度也为O(n+m)，因为需要为新链表分配节点空间来存储合并后的值。

算法的关键在于正确处理两个链表长度不同的情况，确保一个链表遍历完后，另一个链表的剩余节点能够正确地添加到新链表的末尾。通过循环和条件判断，算法能够优雅地处理这种情况，并返回合并后的链表。

此外，算法中还使用了额外的辅助函数CreateList\_R来创建链表，以及showList来显示链表的值

7.食材运输

本题的主要目标是制定一个运输方案，使得所有酒店的等待时间的最大值最小。考虑到每辆车从哪个检查点出发以及每辆车的运输路线，我们需要寻找一个最优的解。

预处理与状态定义：

首先，我们读取每个酒店对每种食材的需求情况，将其转换为二进制状态，存储在need数组中。

然后，我们读取双向道路的信息，将其存储在邻接表和权重数组中。

定义g[i][j]表示当只考虑食材集合j时，从酒店i出发并遍历所有需要该集合食材的酒店，最后返回酒店i所需的最短时间。

定义dp[i][j]表示当考虑食材集合i，并设置j个检查点时，所有酒店等待时间的最大值的最小值。

计算g数组：

对于每种食材，我们计算g数组。具体地，我们先标记需要当前食材的酒店，并计算从任意酒店出发遍历所有需要该食材的酒店并返回的最短时间（使用DFS实现）。

同时，我们计算从任意酒店出发到最远需要当前食材的酒店的时间，从而得到g[i][j]的值。

动态规划计算dp数组：

初始化dp[0][1] = 0，表示当没有食材需要考虑时，只需要一个检查点，等待时间为0。

对于每种食材，我们已经计算了g数组，现在我们可以利用g数组来计算dp数组。

对于每个食材集合i和检查点数量j，我们尝试将集合i分割为两个子集s和i-s，并计算只考虑子集s时一个检查点的等待时间最大值，以及只考虑子集i-s时j-1个检查点的等待时间最大值。取两者的最大值作为当前方案的等待时间，并更新dp[i][j]。

输出结果：

最后，输出dp[(1<<k)-1][m]，即考虑所有食材并设置m个检查点时，所有酒店等待时间的最大值的最小值。

算法总结：

主要使用了动态规划的思想来解决问题。首先，我们预处理了食材需求和道路信息，并定义了状态数组g和dp。

然后，我们计算了g数组，它代表了只考虑某种食材集合时，从某个酒店出发并遍历所有需要该集合食材的酒店所需的最短时间。

接下来，我们使用动态规划来计算dp数组，通过考虑不同的食材集合分割和检查点数量，来找到最优的运输方案。

最后，我们输出了最优方案下所有酒店等待时间的最大值的最小值。

为了优化计算，我们在计算g数组时使用了DFS来遍历所有需要当前食材的酒店，并计算了从任意酒店出发到最远需要当前食材的酒店的时间。

8.链表的中位数

明确问题的要求，找出有序链表的中位数。如果链表长度是奇数，则中位数就是中间那个数；如果是偶数，则中位数是中间两个数的平均值。

选择算法：由于链表是有序的，我们可以利用快慢指针的技术来找到链表的中点，而不需要遍历整个链表两次或者将链表元素存储到数组中。快慢指针是一种常用的链表算法，其中快指针每次移动两个节点，慢指针每次移动一个节点。当快指针到达链表末尾时，慢指针刚好在链表的中间位置。

处理细节：在使用快慢指针时，我们还需要考虑如何处理链表长度为偶数的情况。在这种情况下，当快指针到达链表末尾时，慢指针会指向中间两个节点的后一个。为了计算这两个节点的平均值，我们需要保留对慢指针前一个节点的引用。

计算和返回结果：最后，根据链表长度是奇数还是偶数，计算中位数（单个节点的值或两个节点值的平均值），并将其作为浮点数返回。需要注意的是，如果计算平均值，应确保结果为两位小数。

算法总结：

算法：快慢指针技术

时间复杂度：O(n)，其中n是链表的长度。快指针和慢指针都只需要遍历链表一次。

空间复杂度：O(1)。该算法只需要常数级别的额外空间来存储快慢指针和前一个节点的引用。

适用性：该算法特别适用于有序链表，因为它依赖于链表的有序性来直接找到中位数，而无需对链表进行排序或将其元素存储到数组中。

关键点：

使用快慢指针找到链表的中点。

记录慢指针的前一个节点，以便在链表长度为偶数时能够计算中间两个节点的平均值。

根据链表长度是奇数还是偶数，返回单个节点的值或两个节点值的平均值作为中位数。

1. 拼多多周年庆

本题的主要目标是最大化布置花灯的道路长度，但同时要满足一些约束条件。我们可以将这个问题看作是一个树形动态规划问题，其中树的每个节点代表一个城市，而树的边代表道路。由于每个城市（除了首都）最多只有两条道路通向子城市，我们可以考虑针对每个节点进行状态转移。

对于每个节点，我们需要考虑它是否布置花灯，以及如果它布置花灯，其所有子节点中哪些也布置花灯。同时，我们需要记录从根节点到当前节点的路径长度，以确保不超过总长度上限。

输入处理：

读取总长度上限m和城市的个数n。

根据输入的城市和道路信息，构建树的结构，并保存每个城市的父节点和到父节点的距离。

查找根节点（首都）。

递归函数func：

这个函数是核心部分，用于计算以当前节点为根的子树中，可以布置花灯的最大道路长度。

函数首先初始化一个结果集ret，用于保存可能的道路长度。

对于当前节点，如果它的子节点到它的距离总和不超过m，则考虑将其子节点都布置花灯，并更新结果集。

对于每个子节点，递归调用func函数，获取该子节点为根的子树中可能布置花灯的道路长度，并保存到rets数组中。

遍历所有子节点，两两组合它们的结果，加上当前节点到父节点的距离，看是否可以等于m，如果可以，则更新结果集。

最后，对结果集进行排序并返回。

主函数main：

读取输入并构建树结构。

找到根节点（首都）。

调用递归函数func，传入根节点和总长度上限m。

输出结果集中最大的道路长度。

10.查找专家

初始化：首先，我们创建一个数据结构来跟踪哪些专家知道秘密。由于我们需要快速地检查某个专家是否知道秘密，所以选择unordered\_set（无序集合）作为数据结构。最初，专家0和firstPerson都知道秘密，因此他们被添加到集合中。

会议排序：我们将所有会议按照时间顺序排序。这可以通过使用优先队列（最小堆）实现，其中会议按照时间顺序排列。这样做的目的是为了模拟秘密在时间上的传播。

处理会议：我们按照时间顺序处理每个会议。对于每个会议，我们检查参会的两个专家是否都知道秘密。如果其中只有一个知道秘密，那么秘密将被传播给不知道秘密的专家，我们将其添加到知道秘密的专家集合中。

重新加入会议：当一个专家新知道秘密后，我们需要考虑这个新知情的专家在接下来的会议中可能会继续传播秘密。因此，我们将所有涉及新知道秘密的专家的会议（且时间晚于当前处理的会议）重新加入优先队列中。

输出结果：处理完所有会议后，集合中剩下的就是所有知道秘密的专家。我们将这些专家的编号输出即可。

算法总结：

数据结构选择：为了高效处理，选择了unordered\_set来跟踪知道秘密的专家，以便快速检查某个专家是否知道秘密。同时，使用优先队列（最小堆）来按时间顺序处理会议。

时间复杂度：算法的时间复杂度主要取决于会议的数量和重新加入队列的会议数量。在最坏的情况下，每个会议都可能被多次加入和处理，但由于每个会议只会被处理一次且重新加入队列的会议数量是有限的（受会议总数限制），所以总体时间复杂度仍然是可接受的。

空间复杂度：算法使用了unordered\_set和优先队列来存储数据和辅助处理，空间复杂度主要取决于会议的数量和专家的数量。

关键点：算法的关键在于正确处理秘密的传播和会议的重新加入。当一个专家新知道秘密时，必须考虑他/她在未来的会议中可能继续传播秘密的情况。

优化：如果输入数据很大，可以考虑使用更高效的数据结构或并行处理来优化算法性能。此外，对于重新加入队列的会议，可以通过更精细的管理来减少不必要的重复处理。