1. **A permutation perm of n + 1 integers of all the integers in the range [0, n] can be represented as a string s of length n where:**

* **s[i] == 'I' if perm[i] < perm[i + 1], and**
* **s[i] == 'D' if perm[i] > perm[i + 1].**

**Given a string s, reconstruct the permutation perm and return it. If there are multiple valid permutations perm, return any of them.**

**Example 1:**

**Input: s = "IDID"**

**Output:**

**[0,4,1,3,2]**

Ans: To reconstruct the permutation `perm` from the given string `s`, we can follow the following approach:

1. Initialize an empty list `perm` to store the reconstructed permutation.

2. Initialize a variable `n` to the length of the string `s`.

3. Initialize two variables `low` and `high` to 0 and `n` respectively. These variables will represent the range of available integers to choose from.

4. Iterate through each character `ch` in the string `s`:

- If `ch` is 'I':

- Append `low` to `perm`.

- Increment `low` by 1.

- If `ch` is 'D':

- Append `high` to `perm`.

- Decrement `high` by 1.

5. After the loop ends, append either `low` or `high` to `perm`. Both values would be the same at this point.

6. Return the reconstructed permutation `perm`.

Here's the implementation of this algorithm in Python:

```python

def reconstructPermutation(s):

perm = []

n = len(s)

low, high = 0, n

for ch in s:

if ch == 'I':

perm.append(low)

low += 1

else:

perm.append(high)

high -= 1

perm.append(low) # or perm.append(high)

return perm

```

Using this function, we can reconstruct the permutation for the example you provided:

```python

s = "IDID"

result = reconstructPermutation(s)

print(result)

```

Output:

```

[0, 4, 1, 3, 2]

```

Note that there can be multiple valid permutations for a given string `s`, and this algorithm will return any of them.

**4. Given a binary array nums, return *the maximum length of a contiguous subarray with an equal number of* 0 *and* 1.**

**Example 1:**

**Input:** nums = [0,1]

**Output:** 2

**Explanation:**

[0, 1] is the longest contiguous subarray with an equal number of 0 and 1.

Ans: To find the maximum length of a contiguous subarray with an equal number of 0 and 1 in a binary array, you can use the following approach:

1. Initialize a variable `maxLen` to store the maximum length found so far and a variable `count` to keep track of the cumulative count of 0s and 1s.

2. Create a dictionary `countDict` to store the cumulative count as the key and the corresponding index as the value. Initialize `countDict` with a key-value pair (0, -1) to handle the case when the subarray starts from the beginning.

3. Traverse the binary array `nums` from left to right and for each index `i`, do the following:

- If `nums[i]` is 0, increment `count` by 1. Otherwise, decrement `count` by 1.

- If `count` is already in `countDict`, calculate the length of the subarray as `i - countDict[count]` and update `maxLen` if necessary.

- If `count` is not in `countDict`, add it to `countDict` with the current index `i` as the value.

4. Return `maxLen` as the result.

Here is the implementation of the above algorithm in Python:

```python

def findMaxLength(nums):

maxLen = 0

count = 0

countDict = {0: -1} # Initial count with index -1

for i in range(len(nums)):

count += 1 if nums[i] == 1 else -1

if count in countDict:

subarrayLen = i - countDict[count]

maxLen = max(maxLen, subarrayLen)

else:

countDict[count] = i

return maxLen

```

Let's test the example you provided:

```python

nums = [0, 1]

print(findMaxLength(nums))

```

Output:

```

2

```

The function returns 2, indicating that the longest contiguous subarray with an equal number of 0 and 1 in the given `nums` array is [0, 1].

**5**. **The product sum of two equal-length arrays a and b is equal to the sum of a[i] \* b[i] for all 0 <= i < a.length (0-indexed).**

* **For example, if a = [1,2,3,4] and b = [5,2,3,1], the product sum would be 1*5 + 2*2 + 3*3 + 4*1 = 22.**

**Given two arrays nums1 and nums2 of length n, return *the minimum product sum if you are allowed to rearrange the order of the elements in* nums1.**

**Example 1:**

**Input: nums1 = [5,3,4,2], nums2 = [4,2,2,5]**

**Output: 40**

**Explanation:**

**We can rearrange nums1 to become [3,5,4,2]. The product sum of [3,5,4,2] and [4,2,2,5] is 3*4 + 5*2 + 4*2 + 2*5 = 40.**

**Ans:** To find the minimum product sum when rearranging the order of elements in `nums1`, you can use the following approach:

1. Sort both `nums1` and `nums2` arrays in non-decreasing order.

2. Initialize a variable `minSum` to store the minimum product sum.

3. Traverse `nums1` from left to right and `nums2` from right to left simultaneously.

- Multiply the current elements from `nums1` and `nums2` and add the product to `minSum`.

4. Return `minSum` as the result.

Here is the implementation of the above algorithm in Python:

```python

def minProductSum(nums1, nums2):

nums1.sort()

nums2.sort(reverse=True)

minSum = 0

for i in range(len(nums1)):

minSum += nums1[i] \* nums2[i]

return minSum

```

Let's test the example you provided:

```python

nums1 = [5, 3, 4, 2]

nums2 = [4, 2, 2, 5]

print(minProductSum(nums1, nums2))

```

Output:

```

40

```

The function returns 40, indicating that the minimum product sum when rearranging the elements of `nums1` to [3, 5, 4, 2] and multiplying it with `nums2` [4, 2, 2, 5] is 40.

**6**. **An integer array original is transformed into a doubled array changed by appending twice the value of every element in original, and then randomly shuffling the resulting array.**

**Given an array changed, return original *if* changed *is a doubled array. If* changed *is not a doubled array, return an empty array. The elements in* original *may be returned in any order*.**

**Example 1:**

**Input: changed = [1,3,4,2,6,8]**

**Output: [1,3,4]**

**Explanation: One possible original array could be [1,3,4]:**

* **Twice the value of 1 is 1 \* 2 = 2.**
* **Twice the value of 3 is 3 \* 2 = 6.**
* **Twice the value of 4 is 4 \* 2 = 8.**

**Other original arrays could be [4,3,1] or [3,1,4].**

**Ans:** To solve this problem, you can use the following approach:

1. Initialize an empty dictionary called `countDict` to keep track of the count of each element in the `changed` array.

2. Traverse the `changed` array and for each element, do the following:

- Increment the count of the current element in `countDict` by 1.

3. Initialize an empty list called `original`.

4. Traverse the `changed` array again and for each element, do the following:

- If the current element divided by 2 exists in `countDict` and its count is greater than 0, decrement the count of the current element divided by 2 in `countDict` by 1.

- Otherwise, return an empty list since the `changed` array is not a doubled array.

5. Append the current element divided by 2 to the `original` list.

6. Return the `original` list as the result.

Here is the implementation of the above algorithm in Python:

```python

from collections import defaultdict

def findOriginalArray(changed):

countDict = defaultdict(int)

for num in changed:

countDict[num] += 1

original = []

for num in changed:

if countDict[num] > 0:

if countDict[num \* 2] > 0:

countDict[num \* 2] -= 1

else:

return []

original.append(num)

return original

```

Let's test the example you provided:

```python

changed = [1, 3, 4, 2, 6, 8]

print(findOriginalArray(changed))

```

Output:

```

[1, 3, 4]

```

The function returns `[1, 3, 4]`, which is one possible original array that satisfies the given conditions.