A1_Report_123090612

Q1 Array Problem

①What are the possible solutions for the problem?

I don't think there's much else to solve for such a simple question dealing with numbers. It's enough to process the input numbers from top to bottom as required by the question, and there is no need for a “complicated” solution.

②How do you solve this problem?

1. ***Initialization***
    1. ***Accept the input data and convert it to the data type I need for later use***

```
n, m, P = map(int, input().split())
a_list = list(map(int, input().split()))
```

- Use the `split()` method to split the input string into multiple substrings by space, returning a list containing those substrings.
- `map(int, ...)` Generates an integer iterator by passing each string element of the resulting list in turn to the int function.
- The property that the `input()` function automatically handles newlines is utilized here. Each `input()` call reads a line of input until the user presses Enter (which is a line feed).
- The second line uses the same idea to populate the `a_list` with the input after processing the data type.

2. ***Preparation***

```
total_sum = sum(a_list)
counts = {}
count_zero = 0
```

- Calculate and initialize the sum of all elements in `a_list`, stored in the variable `total_sum`, which is subsequently modified.
- `counts` is a dictionary of the number of occurrences of each number (after taking absolute values).
- `count_zero` is used to count the number of elements in the list that have a value of zero.

3. ***Count the frequency of numbers and the number of zeros***

```
for val in a_list:
    abs_val = abs(val)
    if abs_val == 0:
        count_zero += 1
    else:
        counts[abs_val] = counts.get(abs_val, 0) + 1
```

- There's an important operation here, which is to take the elements in `a_list` to their absolute values.

4. ***Count how many different numbers there are in the initial case***

```
max_distinct = 0
    if count_zero > 0:
        max_distinct += 1
    for count_v in counts.values():
        max_distinct += min(count_v, 2)
```

- 0 will have at most 1 occurrence.
- Other numbers with the same absolute value will have at most 2 occurrences.

2. Implementation of 3 operations
    - None of the regular operations that deal with inputs are much to explain.
    - In the process of updating data, it's essential to update `counts`, `count_zero`, and `max_distinct` as well.
    - When calculating the sum, do not sum after all the numbers have been updated. It's more efficient to use the difference between one number and the old one (in the same position) as soon as a number is updated to calculate the new sum.

③Why is your solution better than others?

For question 1, I don't know whether my solution is better than other students', and I myself can't come up with any different ways of solution. I can only explain how I improved my program.
My improvements are mainly in the part about finding the number of numbers with different values (aka operation 3). Here is my initial solution (using operation 3 as an example).

```
elif ops[0] == '3':
    max_distinct = 0
    if count_zero > 0:
        max_distinct += 1
    for count_v in counts.values():
        max_distinct += min(count_v, 2)
    print(max_distinct)
```

To address the inefficiencies here, I made improvements in two places, the update operation and the query operation, respectively.

- Update operation
    - The improved program updates the `max_distinct` variable in real time on each update, in addition to updating the counter, based on the change between the old and new values. This way, only the count of the affected value needs to be adjusted at each update, instead of traversing the entire counter.
      The process is as follows:
    - If the value being updated (the old value) has an absolute value of 0 or some positive number, the program decreases the value of `max_distinct` accordingly after the old value is removed from the array.
    - Then `max_distinct` is updated as soon as the new value is added to the array, depending on the new value.
      Because each update affects only one value, program 1 avoids a global recalculation of `max_distinct`. Therefore, in scenarios with a large number of update and query operations, the new program will have a much lower time complexity than the original program.
- Distinct value queries
    - When querying for “distinct value”, the original program needed to go through the entire `counts` dictionary and recalculate the number of distinct absolute values, but the new program doesn't need to do that anymore.

Q2 List

①What are the possible solutions for the problem?

I came up with 3 solutions to this problem. They don't differ much in terms of accepting the data and other prep, the main difference is in the idea of testing whether the inputs are valid. Here is a brief explanation of the three different ideas.
`(Regarding prep: in the first and second solutions, I labeled all the elements with their positions in the original input, so that in subsequent operations I could disregard the order of segmentation and only consider the result. But I later realized that this did not seem correct.)`

1. Classification is based on the difference in the number of child intervals contained in the parent interval.

1. If the parent interval contains only 1 sub-interval, then either of the following cases need to be satisfied, otherwise the input will be treated as invalid. `(This method of determination was eventually found to be problematic.)`
    - Either the sub-interval can completely cover the parent interval (with the same endpoints)
    - Or the left endpoint of the sub-interval minus one equals the left endpoint of the parent interval, or the right endpoint of the sub-interval plus one equals the right endpoint of the parent interval.
2. If the parent interval contains 2 sub-intervals, then it's necessary to satisfy that the two sub-intervals can be merged to form the parent interval.
3. If the parent interval contains ≥ 2 child intervals in the final result, then the input is determined to be invalid directly.

2. I find that after correct cuts, the resulting interval (expressed in terms of position), assuming that after sorting this is interval `i`, and the left and right endpoints of this interval `i` are `i_a` and `i_b`. Then:
    - Either `i_a` = `(i-1)_a`
    - Either `i_a` = `(i+1)_a`
    - Either `i_b` = `(i-1)_b`
    - Either `i_b` = `(i+1)_b`
      However, I realized that although this method can effectively deal with most of the cut cases, there are still a few cases that cannot be effectively recognized by this method, such as the following input:

```
12 5
1 2 3 4 5 6 7 8 9 10 11 12
1 5 5 1 9
12 12 9 4 12
```

3. I completely gave up the bottom-up approach and returned to the simplest and most natural way of thinking described in the question: let the left and right endpoints of the parent interval be `l_pre` and `r_pre`, respectively, and let the left and right endpoints of the newly split interval be `l` and `r`, respectively, and check whether `l_pre` ≤ `l` ≤ `r` ≤ `r_pre`. If the condition is met, split the parent interval, and update `l_pre` and `r_pre`.
    - **This method eventually worked!**

②How do you solve this problem?

1. ***Define a function that checks whether a child interval can be contained within a parent interval.***

```python
def check_interval(l_pre, r_pre, l, r) -> tuple[int, int]:
    if (l == l_pre) :
        if (l < r <= r_pre):
            return (r + 1, r_pre)
    elif (r == r_pre):
        if (l_pre <= l < r):
            return (l_pre, l - 1)
    return None
```

2. ***Initialization, accept the input data***

```python
n, q = map(int, input().split())
permut = list(map(int, input().split()))
ls = list(map(int, input().split()))
rs = list(map(int, input().split()))
```

3. ***Construct a value-to-index mapping, making it easy to use the elements' positions in the original arrangement directly in subsequent operations instead of using their values.***

```python
val_to_idx = [0] * (n + 1)

for idx, val in enumerate(permut):
```

```
    val_to_idx[val] = idx

ls = [val_to_idx[val] for val in ls]
rs = [val_to_idx[val] for val in rs]
```

4. ***Verify the validity of the interval (the most important part)***

```
is_valid = True

if ls[0] != 0 or rs[0] != n - 1:
    is_valid = False
else:
    leaf_intervals= set([(ls[0], rs[0])])
    for l, r in zip(ls[1:], rs[1:]):
        is_father_found = False
        for (l_pre, r_pre) in leaf_intervals:
            if (l_pre <= l <= r <= r_pre):
                is_father_found = True
                break

        if not is_father_found:
            is_valid = False
            break

        leaf_intervals.remove((l_pre, r_pre))
        another_leaf = check_interval(l_pre, r_pre, l, r)
        if another_leaf is not None:
            leaf_intervals.add((l, r))
            leaf_intervals.add(another_leaf)
        else:
            is_valid = False
            break
```

- First test whether the interval of the first query (which is the largest one) covers the whole arrangement. If not, then it's directly output as illegal.
- Next, for each interval queried, iterate through the leaf intervals that currently exist (i.e., intervals that have not been further partitioned) to see if any of these intervals can contain the current query interval.
  - If a parent interval is found, the program sets `is_father_found` to `True`, indicating that a parent interval has been found that can hold the query interval, and jumps out of the loop.
  - If, after traversing all the leaf intervals, no parent interval that matches the condition can be found, consider it as an invalid structure and directly set `is_valid` to `False`.
- If a parent interval is found, it means that this parent interval will be split, so the program will first remove this parent interval from `leaf_intervals`.
- Next, call the previously defined `check_interval` function to see if the parent interval can be split into two new subintervals. One subinterval is the current query interval `(l, r)`, and the other is the remainder of the parent interval.
  - If `check_interval` returns another valid interval, the split was successful and the program adds the new two subintervals to the `leaf_intervals` collection. This updates the current set of leaf intervals.
  - If `check_interval` finds that it cannot effectively split the parent interval into two parts, the program sets `is_valid` to `False`, indicating that the structure is not legal, and stops further processing.

③Why is your solution better than others?

- This is the most natural solution, following a top-down approach rather than a bottom-up approach, avoiding getting bogged down in discussing overly complex cases
- No need to use overly complex data structures