N2 - Mode & Frequency Problem

— 14 June 2024, Patcharapong K.

Intro

In this problem, we were given the task of finding `{mode, freq}` by using “Divide and Conquer” (from now on, for conciseness, will be called DC) strategy. Aj.Nattee wanted us to do 3 DC variants based on division boundaries, which are the following:

1. Divide from the middle
2. Divide into left, middle, right
3. Divide into ranges of each value

We will be discuss each variant in the below sections.

C++ Headers/Typedefs

Since we will be typing lots of things like `pair<int,int>` , `vector<int>` , I try to typedef them into more readable type names. This way we can concentrate on just the algorithmic part, thus being more language-agnostic.

```
#include <algorithm>
#include <iostream>
#include <map>
#include <vector>

using namespace std;
typedef vector<int> Vector;
typedef map<int, int> Map;
typedef pair<int, int> Pair;
```

Parameters

Since we will be “cutting” our vector into halves, boundary variables should be required. Since a vector has two ends, left and right, we start with the function prototype as follows:

```
void solve(int left, int right); // returns void for now
```

We may be adding more parameters if needed, but “left” and “right” will have to be there.

Since we need to return two values, mode and frequencies, we might as well return a pair:

```
// typedef pair<int,int> Pair
Pair solve(int left, int right) {
    ...
    return {mode, freq}; // Construct a pair in place
}
```

We also need to define our right bound to be either ***exclusive*** or ***inclusive.*** In this case, I personally like a right-bound inclusive.

Basic Base Cases

Since the function should work on every instances, we start with the simple cases for all variants:

- Empty vector should return {-1, -1} to indicate nothingness
  - Can be checked by `if (left > right)`
    - If a vector is empty, solve(0,0-1) will be passed. Since 0>-1, {-1, -1} will be returned
    - This also helps the out of bound / invalid ranges as well
- A vector with one element should return {value, 1}
  - Can be checked with `if (left == right)`
- Invalid bounds such as negatives
  - Can be checked with `if (left < 0 || right < 0)`
  - ***Not implemented (assume the vector and the arguments are correct)***

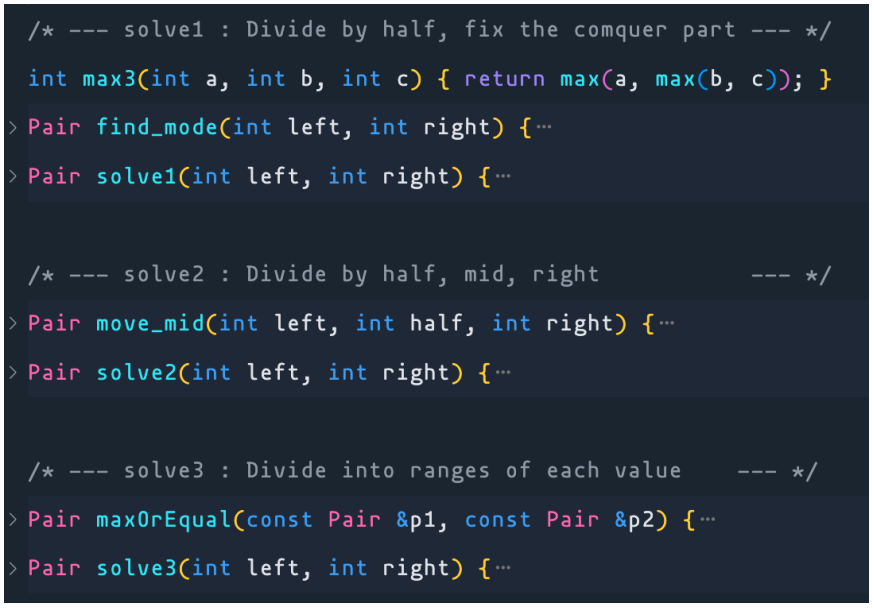
```
// typedef pair<int,int> Pair
Pair solve(int left, int right) {

    // Out of bound OR Empty Vector
    if (left > right) return {-1, -1};

    // One element
    if (left == right) return {v[left], 1};

    ...
}
```

▼ Overview & Helper functions



▼ Variant 1 - Divide from the middle

Helper Function	Description
max3(a, b, c)	returns max of three integers: (a, b, c)

In the first variant, we simply divide from the middle. The middle separation case is handled in the merge process instead. This variant allows for easy “divide” part:

```
int half = left + (right - left) / 2;

// Divide by half
Pair p1 = solve1(left, half);
Pair p2 = solve1(half + 1, right);
```

Now comes the challenging part, the “conquer” part

```
// Middle calculation
int mid_mode = v[half];
int mid_mode_count = 1;
int h1 = half, h2 = half;

while (h1 >= left  && v[h1] == mid_mode) --h1;
while (h2 <= right && v[h2] == mid_mode) ++h2;
```

```
mid_mode_count = h2 - h1 - 1;

// Returning the max frequency entry
int max_count = max3(p1.second, mid_mode_count, p2.secon
d);
if (max_count == p1.second)
    return {p1.first, p1.second};
if (max_count == mid_mode_count)
    return {mid_mode, mid_mode_count};
return {p2.first, p2.second};
```

The code above compares the left side and right side with the “middle calculation”. The middle calculation gets the count of the value in the middle. By adjusting (—h1, ++h2) the pointers, we can reach the boundaries of the value.

▼ Variant 1 - Full Code

```
Pair solve1(int left, int right) {
    if (v.size() == 0) return {-1, -1};
    if (left == right)
        return {v[left], 1};

    int half = left + (right - left) / 2;

    // Divide by half
    Pair p1 = solve1(left, half);
    Pair p2 = solve1(half + 1, right);

    // Conquer and fix the boundaries
    int mid_mode = v[half];
    int mid_mode_count = 1;
    if (half + 1 < right && v[half] == v[half + 1]) {
        int h1 = half;
        int h2 = half + 1;
        while (h1 >= 0 && v[h1] == v[half])
            --h1;
```

```
        while (h2 < v.size() && v[h2] == v[half])
            ++h2;
        mid_mode_count = h2 - h1 - 1;
    }
    int max_count = max3(p1.second, p2.second, mid_mo
de_count);
    if (max_count == p1.second)
        return {p1.first, p1.second};
    if (max_count == p2.second)
        return {p2.first, p2.second};
    return {mid_mode, mid_mode_count};
}
```

▼ Variant 2 - Divide into left, middle, right

Helper Function	Description
move_mid(left, half, right)	Returns a pair {leftBound, rightBound} of the middle values.

This time we got a harder “divide”, much simpler “conquer”, alternate to Variant 1. Since we need to divide into the additional middle part, the `move_mid` function is used to return the left and right bounds of the middle value. Suppose we have a vector

```
// Pos      0 1 2 3 4 5 6 7
Vector v = {1,1,2,2,2,2,5,5};
```

`move_mid` will return {2,5} which is the leftmost and rightmost bound of the middle value (val: 2). Here is what `move_mid` looks like:

```
Pair move_mid(int left, int half, int right) {
    int mid_entry = v[half];

    int half1 = half, half2 = half;
    while (half1 >= left  && v[half1] == mid_entry) --ha
lf1;
    while (half2 <= right && v[half2] == mid_entry) ++ha
lf2;
```

```
    return {half1 + 1, half2 - 1};
}
```

Now the recursive part becomes just calling on left and right, the middle part can be calculated directly.

```
Pair halves = move_mid(left, half, right);
int half1 = halves.first;
int half2 = halves.second;

Pair p1 = solve2(left, half1 - 1);
Pair p2 = {v[half], half2 - half1 + 1};
Pair p3 = solve2(half2 + 1, right);
```

Now comes the conquer part, we just find maximum of the three (left, middle, right). If the max frequency happened to be from p1, return a pair of p1. Same goes to p2 and p3. By having the if statement covering p1 first, we make sure the first occurrence of the highest frequency is returned.

Suppose we have [1,1,2,5,5]. If the p1 checking comes behind p3 and p2, it will answer {mode:5, freq:2} instead of the intended {mode:1, freq:2}.

```
int max_count = max3(p1.second, p2.second, p3.second);
if (max_count == p1.second) return {p1.first, p1.secon
d};
if (max_count == p2.second) return {p2.first, p2.secon
d};
return {p3.first, p3.second};
```

▼ Full Code - Variant 2

```
Pair move_mid(int left, int half, int right) {
    int mid_entry = v[half];
    int half1 = half, half2 = half;
    while (half1 >= left && v[half1] == mid_entry) {
        --half1;
    }
    while (half2 <= right && v[half2] == mid_entry) {
```

```
        ++half2;
    }
    return {half1 + 1, half2 - 1};
}

Pair solve2(int left, int right) {
    if (left > right) return {-1, -1};
    if (left == right) return {v[left], 1};
    int half = left + (right - left) / 2;

    // Divide into left, half, mid
    Pair halves = move_mid(left, half, right);
    int half1 = halves.first;
    int half2 = halves.second;

    Pair p1 = solve2(left, half1 - 1);
    Pair p2 = {v[half], half2 - half1 + 1};
    Pair p3 = solve2(half2 + 1, right);

    // Conquer
    int max_count = max3(p1.second, p2.second, p3.sec
ond);
    if (max_count == p1.second) return {p1.first, p1.
second};
    if (max_count == p2.second) return {p2.first, p2.
second};
    return {p3.first, p3.second};
}
```

▼ Variant 3 - Divide into ranges of each value

Helper Function	Description
maxOrEqual(pair1, pair2)	Returns pair1 if its value is more than or equal to pair2

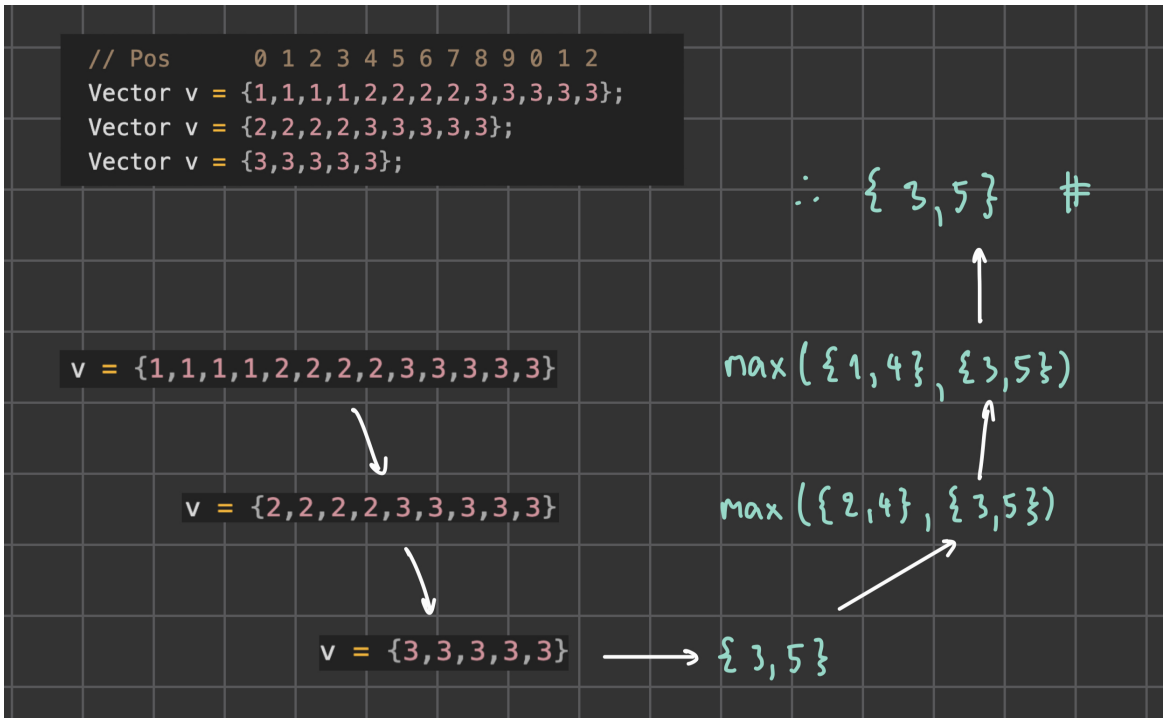
Here comes the last variant, this time we jump from one value range to another directly. Suppose we have the following vector:

```
// Pos      0 1 2 3 4 5 6 7 8 9 0 1 2
Vector v = {1,1,1,1,2,2,2,2,3,3,3,3,3};
```

Each call, will “divide” the vector into:

```
// Pos      0 1 2 3 4 5 6 7 8 9 0 1 2
Vector v = {1,1,1,1,2,2,2,2,3,3,3,3,3};
Vector v = {2,2,2,2,3,3,3,3,3};
Vector v = {3,3,3,3,3};
```

We then work our way backwards from the deepest recursive call.



Since we are comparing the max between `pair.second` , we can either use the custom comparator which is too complicated to write. The alternative way I did was to create the `maxOrEqual` function to return the first pair if it’s greater or equal to the second pair.

> *Why equals as well?*

Since the answer will propagate back to the left side (front) of the vector, if we have same frequency pairs e.g., {1,5} and {3,5}:

- with equals, returns {1,5}

- without equals, returns {3,5}

To ensure the first occurrence, we need to include the equal case as well.

```
Pair maxOrEqual(const Pair &p1, const Pair &p2) {
    if (p1.second >= p2.second) {
        return p1;
    }
    return p2;
}
```

Now comes the problem of how do we get the bounds of the current value? We could jump one by one. However, since the data is already sorted, we could use the help of `<algorithm>`'s header. The `upper_bound()` function it is.

```
// parameters: left, right
// basic base cases not included here
auto nextPos_it = upper_bound(v.begin(), v.end(), v[lef
t]);
int nextPos = nextPos_it - v.begin();
// terminating case
if (nextPos == right) return {v[left], nextPos - left};

return maxOrEqual(solve3(left, nextPos), solve3(nextPos,
right));
```

We divide by calling solve on different parameters and conquer by finding their max, simple!

▼ Full Code - Variant 3

```
Pair maxOrEqual(const Pair &p1, const Pair &p2) {
    if (p1.second >= p2.second) {
        return p1;
    }
    return p2;
}
```

```
Pair solve3(int left, int right) {
    if (left > right) return {-1, -1};
    if (left == right) return {v[left], 1};
    auto nextPos_it = upper_bound(v.begin(), v.end(),
v[left]);
    int nextPos = nextPos_it - v.begin();
    if (nextPos == right) return {v[left], nextPos -
left};
    return maxOrEqual(solve3(left, nextPos), solve3(n
extPos, right));
}
```

▼ Testing

Here are the test cases used in checking the code.

```
vector<Vector> test_cases = {
    {},                                 // 1 Empty vector
    {1},                                // 2 Single element
    {1, 1, 1, 1, 1},                    // 3 All elements the same
    {1, 2},                             // 4 Two different elements
    {2, 2, 2, 2, 2, 3, 3, 3, 3},        // 5 Two different elements with more
    duplicates
    {1, 2, 3, 4, 5},                    // 6 Increasing sequence
    {1, 1, 2, 5, 5},                    // 7 NEW
    {1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3},  // 8 Mixed elements
    {-3, -2, -1, 0, 1, 2, 3},           // 9 Vector with negative numbers
    {1, 1, 2, 3, 3, 4, 5, 5, 6}         // 10 Vector with duplicates and unique
    elements
};

vector<Pair> expected_results = {
    {-1, -1},  // 1 Empty vector
    {1, 1},    // 2 Single element
    {1, 5},    // 3 All elements the same
    {1, 1},    // 4 Two different elements
    {2, 5},    // 5 Two different elements with more duplicates
    {1, 1},    // 6 Increasing sequence
    {1, 2},    // 7 NEW
    {2, 4},    // 8 Mixed elements
    {-3, 1},   // 9 Vector with negative numbers
    {1, 2}     // 10 Vector with duplicates and unique elements
};
```

💡 Key Takeaways

- Make sure the simpler cases always work first, then build the complex case from there.
    - Make sure that {} (empty vector) works
    - Make sure that {1} (one element vector) works
    - Make sure that {1,1,1...,1} (one value, many duplicates) works

    - Make sure that {1,1,1...,1,2,2,...,2} (two values, many duplicates) works
    - Then, we make sure that the general cases works.
- When moving the pointer/iterator around, always “guard” the boundaries
    - When decreasing, `--half1` , make sure that half1 doesn’t go out of the left bound
    - When increasing, `++half2` , make sure that half2 doesn’t go out of the right bound
- Careful of the “off-by-one” problem.