CSE 4404 Lab Solutions

CSE_4404_Lab_03_Task_1B.pdf
PDF

solve all the tasks

Here are the complete solutions for all three tasks from **CSE 4404 Lab 03: Divide & Conquer**:

Task 01: First and Last Occurrence Using Divide and Conquer (Binary Search)

```cpp
#include <iostream>
using namespace std;

int firstOccurrence(int a[], int n, int q) {
    int low = 0, high = n - 1, result = -1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (a[mid] == q) {
            result = mid;
            high = mid - 1; // Search on the left side
        } else if (a[mid] < q) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return result;
}

int lastOccurrence(int a[], int n, int q) {
    int low = 0, high = n - 1, result = -1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (a[mid] == q) {
            result = mid;
            low = mid + 1; // Search on the right side
        } else if (a[mid] < q) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return result;
}
```

Task 02: 2D Peak Finding (Binary Search by Column)

```cpp
#include <iostream>
#include <vector>
using namespace std;

pair<int, int> findPeak2D(vector<vector<int>> &a) {
    int n = a.size();
    int m = a[0].size();

    int left = 0, right = m - 1;

    while (left <= right) {
        int midCol = (left + right) / 2;
        int maxRow = 0;

        // Find global maximum in the middle column
        for (int i = 0; i < n; i++) {
            if (a[i][midCol] > a[maxRow][midCol]) {
                maxRow = i;
            }
        }

        int leftVal = (midCol - 1 >= 0) ? a[maxRow][midCol - 1] : -1e9;
        int rightVal = (midCol + 1 < m) ? a[maxRow][midCol + 1] : -1e9;

        if (a[maxRow][midCol] >= leftVal && a[maxRow][midCol] >= rightVal) {
            return {maxRow, midCol}; // Peak found
        } else if (leftVal > a[maxRow][midCol]) {
            right = midCol - 1;
        } else {
            left = midCol + 1;
        }
    }
    return {-1, -1}; // Should never reach here
}
```

Task 03: Majority Element Using Divide and Conquer

```cpp
#include <iostream>
#include <map>
using namespace std;

int countInRange(int a[], int left, int right, int val) {
    int count = 0;
    for (int i = left; i <= right; i++) {
        if (a[i] == val) count++;
    }
    return count;
}

int majorityElement(int a[], int left, int right) {
    if (left == right) return a[left];

    int mid = (left + right) / 2;
    int leftMajor = majorityElement(a, left, mid);
    int rightMajor = majorityElement(a, mid + 1, right);

    if (leftMajor == rightMajor) return leftMajor;

    int leftCount = countInRange(a, left, right, leftMajor);
    int rightCount = countInRange(a, left, right, rightMajor);

    int majorityThreshold = (right - left + 1) / 2;
    if (leftCount > majorityThreshold) return leftMajor;
    if (rightCount > majorityThreshold) return rightMajor;

    return -1; // No majority
}
```

Wrapper for Output in Task 3:

cpp

```
int main() {
    int n;
    cin >> n;
    int a[n];
    for (int i = 0; i < n; ++i) cin >> a[i];

    int res = majorityElement(a, 0, n - 1);
    if (res == -1) cout << "NO MAJORITY" << endl;
    else cout << res << endl;
}
```

Let me know if you'd like these combined in a single file or tested with sample inputs.

explain task 1 in details and easy to enderstand way by goint through an example test case

Sure! Let's go step-by-step through **Task 1: Finding First and Last Occurrence of an Element in a Sorted Array** using a **Divide and Conquer** approach (specifically, **Binary Search**).

🔍 Problem Summary

You are given:

- A **sorted array** `a[]` in ascending order.
- An **integer** `q`.
  Your job is to find:

- The **first** and **last** index (0-based) where `q` appears.

If `q` doesn't exist in the array, return `-1 -1`.

🔢 Sample Input

```ini
n = 10
a = [1, 2, 4, 4, 4, 5, 6, 7, 9, 10]
q = 4
```

✅ Expected Output

```
2 4
```

Because:

- First occurrence of 4 is at **index 2**
- Last occurrence of 4 is at **index 4**

🧠 Strategy: Binary Search

We'll use **two modified binary searches**:

1. One to find the **first occurrence**
2. One to find the **last occurrence**

Each search will take **O(log n)** time.

🔨 Step-by-Step Explanation

🔹 `firstOccurrence(a, n, q)`:

Binary search for the first `q`, but **keep going left** when you find `q`.

Let's go through the example:

- `low = 0`, `high = 9`
- Mid = (0+9)/2 = 4 → `a[4] = 4` → Match!
  - Save `result = 4`
  - Search **left** (`high = 3`)
- New mid = (0+3)/2 = 1 → `a[1] = 2` < 4 → go **right** (`low = 2`)
- New mid = (2+3)/2 = 2 → `a[2] = 4` → Match!
  - Save `result = 2`
  - Search **left** (`high = 1`) → end

✅ Final `result = 2`

🔹 `lastOccurrence(a, n, q)`:

Binary search for the last `q`, but **keep going right** when you find `q`.

Again, for the same example:

- `low = 0`, `high = 9`
- Mid = 4 → `a[4] = 4` → Match!
  - Save `result = 4`
  - Search **right** (`low = 5`)
- New mid = (5+9)/2 = 7 → `a[7] = 7` > 4 → go **left** (`high = 6`)
- New mid = (5+6)/2 = 5 → `a[5] = 5` > 4 → go **left** (`high = 4`) → end

✅ Final `result = 4`

🧑‍💻 Code Recap

```cpp
```

```
int firstOccurrence(int a[], int n, int q) {
    int low = 0, high = n - 1, result = -1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (a[mid] == q) {
            result = mid;
            high = mid - 1; // keep searching left
        } else if (a[mid] < q) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return result;
}

int lastOccurrence(int a[], int n, int q) {
    int low = 0, high = n - 1, result = -1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (a[mid] == q) {
            result = mid;
            low = mid + 1; // keep searching right
        } else if (a[mid] < q) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return result;
}
```

🧾 Final Output

For the input:

```
10
1 2 4 4 4 5 6 7 9 10
4
```

Call:

```cpp
cout << firstOccurrence(a, n, 4) << " " << lastOccurrence(a, n, 4);
```

Output:

```


2 4
```

Would you like me to write a full `main()` to test this?

explain task 2 in details and easy to enderstand way by goint through an visual 2d array example test case

Sure! Let’s break down **Task 2: 2D Peak Finding using Divide and Conquer** in an **easy and visual way** with an example.

🧠 What is a 2D Peak?

A **2D Peak** is an element in a 2D grid (matrix) that is **greater than or equal to all of its valid neighbors** (left, right, up, and down).

We’ll use a **divide-and-conquer strategy** that works like **binary search**, but on columns.

✅ Given Test Case

Here's the matrix from your sample input:

```pgsql
4 4       ← This means 4 rows and 4 columns
10  8 10 10
14 13 12 11
15  9 11 21
16 17 19 20
```

Let’s write this more clearly:

```markdown
Row\Col |  0   1   2   3
-------------------------
   0    | 10   8  10  10
   1    | 14  13  12  11
   2    | 15   9  11  21
   3    | 16  17  19  20
```

We need to find **any one element** that is a **peak**.

🧭 Algorithm: Binary Search on Columns

■ Step-by-step:

1. **Choose the middle column**:
   For 4 columns → middle is column 1 (0-based indexing: `(0+3)/2 = 1`)
2. **Find the global maximum in this column (1):**
   - `a[0][1] = 8`
   - `a[1][1] = 13`
   - `a[2][1] = 9`
   - `a[3][1] = 17` ⬅️ **Max**

   So, current candidate is `(3, 1) = 17`
3. **Check neighbors of (3, 1):**
   - **Left** = a[3][0] = 16 ✅ smaller

  - **Right** = a[3][2] = 19 ❌ **larger**

  Since **right neighbor is larger**, go **right → columns 2 to 3**

🔁 Next Iteration (columns 2 to 3)

- Now mid column = `(2+3)/2 = 2`
- Look for max in column 2:
  - `a[0][2] = 10`
  - `a[1][2] = 12`
  - `a[2][2] = 11`
  - `a[3][2] = 19` ⬅️ **Max**
- Candidate = `(3, 2) = 19`
- Check neighbors:
  - **Left** = 17 ✅ smaller
  - **Right** = 20 ❌ larger

Move right again → only column 3 left

■ Final Iteration (column 3)

- Mid column = 3
- Max in column 3:
  - `a[0][3] = 10`
  - `a[1][3] = 11`
  - `a[2][3] = 21` ⬅️ **Max**
  - `a[3][3] = 20`
- Candidate = `(2, 3) = 21`
- Neighbors:
  - **Left** = 11 ✅ smaller

- **Up** = 11 ✅ smaller
- **Down** = 20 ✅ smaller

✅ **21 is a 2D peak!**

🧾 Final Output:

```


2 3
```

(Position of 21 in the matrix)

🧑‍💻 C++ Code Summary

```cpp
pair<int, int> findPeak2D(vector<vector<int>> &a) {
    int n = a.size();
    int m = a[0].size();
    int left = 0, right = m - 1;

    while (left <= right) {
        int midCol = (left + right) / 2;

        // Step 1: Find max in midCol
        int maxRow = 0;
        for (int i = 0; i < n; ++i) {
            if (a[i][midCol] > a[maxRow][midCol]) {
                maxRow = i;
            }
        }

        int current = a[maxRow][midCol];
        int leftVal = (midCol > 0) ? a[maxRow][midCol - 1] : -1e9;
        int rightVal = (midCol < m - 1) ? a[maxRow][midCol + 1] : -1e9;
        // Step 2: Check if it's a peak
        if (current >= leftVal && current >= rightVal) {
            return {maxRow, midCol};
        } else if (rightVal > current) {
            left = midCol + 1;
        } else {
            right = midCol - 1;
        }
    }
    return {-1, -1}; // should not happen
}
```

🧠 Summary:

- Divide and conquer works like **binary search on columns**.
- At each step:
    - Find max in middle column.
    - Check its neighbors.
    - Decide whether to move left or right.
- Final time complexity: **O(n log m)**

Would you like to try another example visually or test it in code?

explain task 3 in details and easy to enderstand way by goint through an example test case

Absolutely! Let’s walk through **Task 3: Finding the Majority Element using Divide and Conquer** with a clear and detailed explanation, step-by-step.

🧠 What’s the Problem?

Given an array `a[]` of size `n`, find the **majority element**—the element that appears **more than n/2 times**.

🔴 If no such element exists, output: `NO MAJORITY`

✅ Sample Input

```ini
n = 7
a = [2, 2, 1, 1, 2, 2, 2]
```

Expected Output:

```
2
```

Because:

- 2 appears **5 times**
- 5 > 7/2 = 3.5 → ✅ more than half

🔧 Approach: Divide and Conquer

This is like a **recursive majority vote** system:

1. **Divide** the array into two halves.
2. **Recursively find majority** element in each half.
3. If both halves agree → that's the majority.
4. If they disagree → count both in the whole segment and pick the one that appears more than n/2 times.
5. If no one does → no majority.

🔁 Step-by-Step Breakdown

Input: `[2, 2, 1, 1, 2, 2, 2]`

1. Split into left and right:

   ```ini
   left = [2, 2, 1]      → index 0 to 2
   right = [1, 2, 2, 2]  → index 3 to 6
   ```

🔽 First Recursive Call: `majorityElement(0, 2)`

Split again:

- left = [2] → majority = 2
- right = [2, 1] → split again

Right side of [2, 1]:

- left = [2] → majority = 2
- right = [1] → majority = 1
- count both in [2, 1]:
  - 2 appears once
  - 1 appears once → no one is majority

So, right half returns ❌ "no majority"

Now combine:

- left majority = 2
- right = none
  → check if 2 is majority in [2, 2, 1]:
  - 2 appears 2 times
  - Total elements = 3
  - 2 > 3/2 → ✅ majority = 2

🔽 Second Recursive Call: `majorityElement(3, 6)`

Input: `[1, 2, 2, 2]`

Split:

- left = [1, 2]
- right = [2, 2]

Left:

- [1] → 1
- [2] → 2 → no majority

Check count in [1, 2]:

- no one is majority → ❌

Right = [2, 2] → both are 2 → majority = 2

Now combine [1, 2, 2, 2]:

- candidate = 2
- 2 appears 3 times
- 3 > 4/2 → ✅ majority = 2

🧮 Final Step:

Left = 2
Right = 2
Both agree → ✅ majority = 2

✅ Output: `2`

🧑‍💻 Code Recap

```cpp
```

```
int countInRange(int a[], int left, int right, int val) {
    int count = 0;
    for (int i = left; i <= right; i++)
        if (a[i] == val) count++;
    return count;
}

int majorityElement(int a[], int left, int right) {
    if (left == right) return a[left];

    int mid = (left + right) / 2;
    int leftMajor = majorityElement(a, left, mid);
    int rightMajor = majorityElement(a, mid + 1, right);

    if (leftMajor == rightMajor) return leftMajor;

    int leftCount = countInRange(a, left, right, leftMajor);
    int rightCount = countInRange(a, left, right, rightMajor);
    int size = right - left + 1;

    if (leftCount > size / 2) return leftMajor;
    if (rightCount > size / 2) return rightMajor;

    return -1; // No majority
}
```

🖨️ Wrapper with Output:

cpp

```cpp
int main() {
    int n;
    cin >> n;
    int a[n];
    for (int i = 0; i < n; ++i) cin >> a[i];

    int result = majorityElement(a, 0, n - 1);
    if (result == -1)
        cout << "NO MAJORITY" << endl;
    else
```

```
        cout << result << endl;
}
```

🧠 Summary

- Time complexity: `O(n log n)`
- Divide array like mergesort.
- Combine answers carefully.
- Check actual frequency only when needed.

Would you like to test with a "NO MAJORITY" example next?