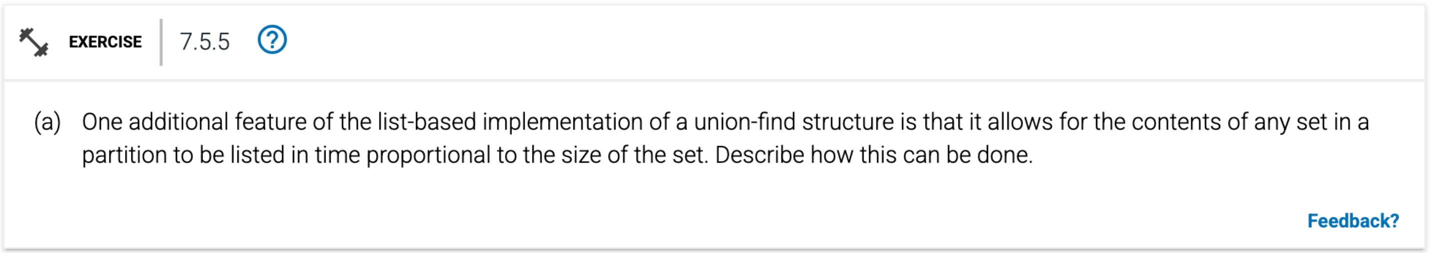
Homework 4 Solution-Komal Wavhal

Chapter 7 Exercises: Reinforcement-7.5.5, Creativity-7.5.9, Application-7.5.21
Chapter 8 Exercises: Creativity-8.5.12, Application-8.5.22, Application-8.5.23
Chapter 9 Exercises: Creativity-9.5.17, Application-9.5.24

Table of Contents

Reinforcement-7.5.5



EXERCISE | 7.5.5

(a) One additional feature of the list-based implementation of a union-find structure is that it allows for the contents of any set in a partition to be listed in time proportional to the size of the set. Describe how this can be done.

Feedback?

Solution

In the list-based implementation of a union-find (or disjoint-set) data structure, each set is represented as a linked list. Each element in the set contains a reference to both its next element and the head of the list, which represents the set. This structure allows for efficient listing of all elements in a set. Here's how this can be achieved:

How to List the Contents of a Set:

1. List Representation:
   - Each set in the union-find structure is represented as a singly linked list.
   - The head of the list serves as the representative of the set, which can be used to identify the set uniquely.
   - Each element in the list maintains a reference to the next element in the set, and every element point to the head of the list.
2. Tracking the Head:
   - To facilitate union operations efficiently, the linked list also keeps a pointer to the head of the set (representative).
   - Each node in the linked list has a reference to the head node, so accessing the head of any element is a quick operation.
3. Listing Elements in the Set:
   - To list all elements in a set, you first need to find the representative (head) of the set, which can be done using the find operation.
   - Once the head of the list is located, you can traverse the linked list starting from this head node.
   - By following the next pointers, you can iterate through all the elements in the set.
   - Since each set is a linked list, traversing the list takes time proportional to the size of the set, O (size of the set).

4. Union Operation Adjustment:
    - During the union operation, when two sets are merged, the smaller list is appended to the larger one.
    - This merging strategy helps keep the list traversal efficient and reduces the overall time complexity of operations.
5. **Example:**

    Suppose you have a union-find structure with three elements in one set:

    - Set: {1, 2, 3}
    - The linked list representation might look like:

    Head -> 1 -> 2 -> 3 -> None

The head of this list is the representative of the set (element 1).
To list all elements in this set, start at the head and follow the next pointers to traverse through 1, 2, and 3.
Complexity:

3. Finding the Head: The find operation takes O (1) time if each element maintains a reference to the head of the list.
4. Traversing the List: The traversal is straightforward and takes time proportional to the size of the set, i.e., O (size of the set).

Pseudocode:

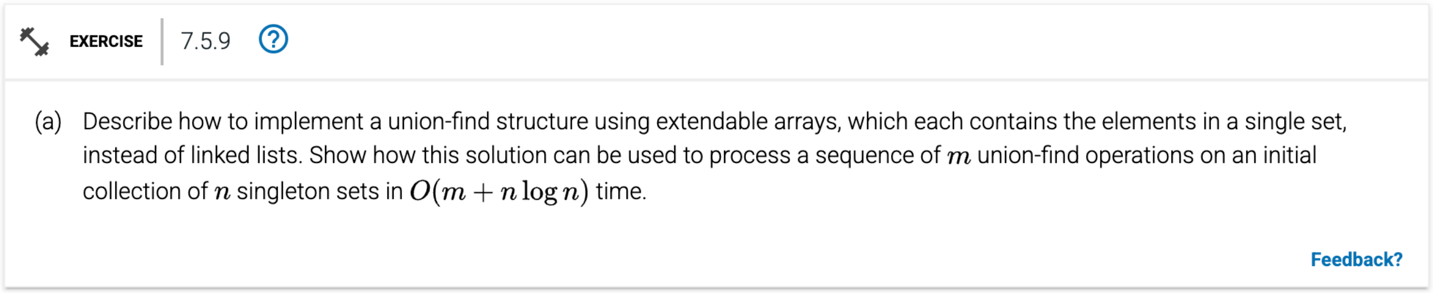
Algorithm ListContents(element)
Input: element - an element in the set we want to list
Output: Prints all elements in the set containing the input element

```
# Step 1: Find the head of the set to which the input element belongs head = element.head
# Step 2: Initialize a pointer to traverse the list current = head
# Step 3: Traverse the linked list and print each element while current is not None:
print(current.value) current = current.next
```

Conclusion:

This list-based union-find implementation allows for the listing of all elements in a set efficiently. By representing each set as a linked list and maintaining references to the head of the list, the structure supports listing all elements in a set in linear time relative to the set's size. This property is useful in applications where access to the full contents of a set is required.

Creativity-7.5.9


EXERCISE 7.5.9

(a) Describe how to implement a union-find structure using extendable arrays, which each contains the elements in a single set, instead of linked lists. Show how this solution can be used to process a sequence of m union-find operations on an initial collection of n singleton sets in $O(m + n \log n)$ time.

Feedback?

Solution

Implementing Union-Find Using Extendable Arrays:

In this approach, we use extendable arrays (or dynamic arrays) to represent each set in the union-find structure instead of using linked lists. Each set is stored in a dynamic array, which allows us to quickly access, traverse, and merge sets. Let's go through the steps to implement this method and explain how it can be used to process a sequence of union-find operations efficiently.

Data Structures:

Array Representation:

- Each set in the union-find structure is stored in a dynamic array. For a set Si , this array contains all the elements currently in that set.
- Each element maintains a reference (or index) to its set's representative array. This reference helps in finding which set an element belongs to.

Representative Array:

- Maintain a separate array `rep` where `rep[i]` points to the representative (or the head) of the array for the set containing element i. This is used to quickly find the representative of a set.

Steps for Union-Find Operations Using Extendable Arrays:

Union Operation:

The union operation combines two sets into a single set.

Algorithm Union(x, y)

Input: x, y - elements to be unified
Output: Combines the sets containing x and y

```
# Step 1: Find the representative (array) of both sets rep_x = Find(x)
rep_y = Find(y)

# If they are already in the same set, return if rep_x == rep_y:

return

# Step 2: Determine the smaller and larger sets if size(rep_x) < size(rep_y):

# Swap to always merge the smaller set into the larger one swap(rep_x, rep_y)

# Step 3: Merge the smaller set (rep_y) into the larger set (rep_x) for element in rep[rep_y]:

append element to rep[rep_x]

update rep[element] to rep_x # Update the representative reference

# Step 4: Clear the merged set's array (optional for memory management) clear(rep[rep_y])
```

Explanation: The union operation first finds the representatives (arrays) of the sets containing x and y. If they are already in the same set, the operation returns. Otherwise, it merges the smaller set into the larger set to keep the structure balanced. This helps optimize the overall time complexity of the union operations.

Find Operation:

The find operation returns the representative of the set containing an element.

Algorithm Find(x)

Input: x - the element whose set representative is to be found Output: Returns the representative of the set containing x return rep[x]

Explanation: The find operation simply accesses the rep array to return the representative of the set containing element x. This is a quick operation with constant time complexity O(1)

Complexity Analysis:

Union-Find Operations:

4. **Union:** In the worst case, merging two sets takes time proportional to the size of the

smaller set. However, since we always merge the smaller set into the larger set, this "union by size" technique ensures that each element can only be involved in O(logn) unions over the course of all operations. Therefore, the total time for all union operations is O(nlogn).

5. **Find:** Each find operation is O(1) because it directly accesses the rep array to find the set's representative.

Overall Time Complexity:

5. **Initialization:** Creating the initial n singleton sets takes O(n) time.
6. **Union-Find Operations:** Given a sequence of m union-find operations on n elements,

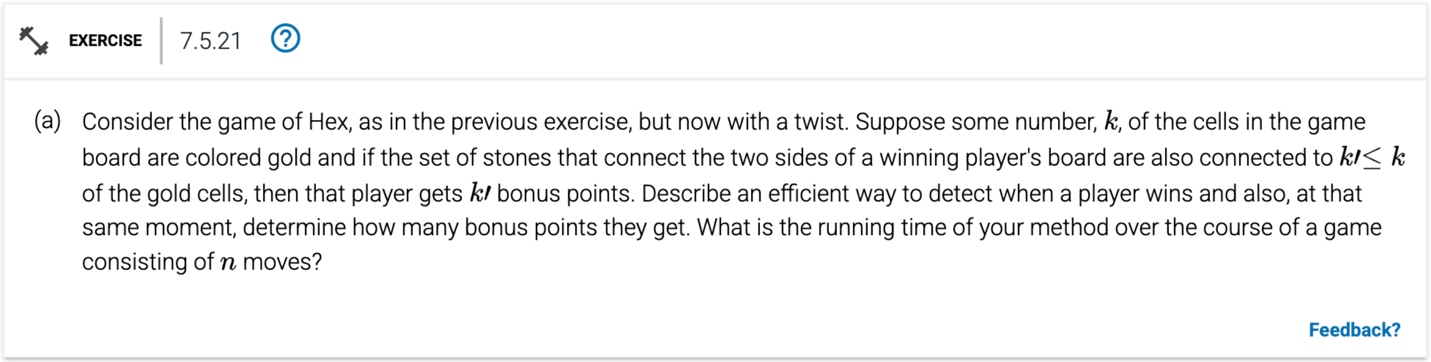
the overall time complexity for this method is O(nlogn+m). The O(nlogn) term arises from the union operations, and the O(m) term accounts for the m constant-time find operations.

Union by Size: By always merging the smaller set into the larger one, the structure remains balanced. This ensures that the total number of array accesses during union operations is logarithmic relative to the number of elements.
Direct Access with Arrays: Using extendable arrays allows direct access and traversal of sets, making both find and union operations efficient.

Conclusion:

By representing sets using **extendable arrays** and using a **union by size** strategy, the union-find structure can efficiently handle a sequence of union-find operations. The overall time complexity of O(nlogn+m) ensures that the operations are manageable even for large sets, providing an efficient solution for problems involving partition management.

Application-7.5.21



EXERCISE | 7.5.21

(a) Consider the game of Hex, as in the previous exercise, but now with a twist. Suppose some number, k, of the cells in the game board are colored gold and if the set of stones that connect the two sides of a winning player's board are also connected to $k' \leq k$ of the gold cells, then that player gets k' bonus points. Describe an efficient way to detect when a player wins and also, at that same moment, determine how many bonus points they get. What is the running time of your method over the course of a game consisting of n moves?

Feedback?

Solution

In this modified version of the game of Hex, the objective is to efficiently detect a win and simultaneously calculate bonus points based on connections to gold cells. To tackle this problem, we need an approach that allows us to:

- Determine if a player has formed a path connecting their sides of the board.
- Check how many gold cells are connected to this path for the bonus point

calculation.

- Given the constraints, we should aim for the lowest possible running time throughout the course of the game.

Data Structures for the Solution:

1. Union-Find (Disjoint-Set) Structure:

   1. Use a union- find data structure with path compression and union by rank to track connected groups of stones for each player. This allows us to efficiently unite regions of stones and check if two regions are connected.

   1. Maintain two additional virtual nodes for each player:
      i. One for the left edge of the board.
      ii. One for the right edge of the board.
   2. The player wins when the virtual nodes on opposite sides of the board become connected in the union-find structure.

2. Gold Cell Tracking:
   1. Maintain an array gold_connected to keep track of whether each cell

is a gold cell or not.

   2. Maintain a hash table (or dictionary) connected_gold_count for each

union-find set to store the count of connected gold cells.

Step-by-Step Algorithm:

1. **Initialization**
2. Create a union-find data structure to handle the board’s cells, including two virtual nodes for each player.
3. Initialize the gold_connected array to mark the positions of all gold cells on the board.
4. Initialize a connected_gold_count hash table to store the count of gold cells connected to each union-find set.
5. **Processing Moves (Union and Bonus Points Calculation)**
6. For each move, perform the following steps:
   1. **PlacetheStone:**Placetheplayer'sstoneatthespecifiedcell.
   2. **Union Adjacent Stones:** Check all adjacent cells (up, down, left,

right, and diagonals) to see if they contain the current player's stones:

i. For each such adjacent cell, use the union operation to merge

the current cell with the adjacent cell in the union-find

structure.

ii. If the adjacent cell belongs to a different union-find set and is

connected to gold cells (tracked by connected_gold_count),

update the count for the new merged set.

3. **UpdateGoldCountforCurrentCell:**Ifthecurrentcellisagold

cell, update the gold cell count in the union-find set that now contains

this cell.

4. **Connect Virtual Nodes:** If the stone is placed on the board's left or

right edge, use the union operation to connect it to the corresponding

virtual node.

5. **CheckforWin:**Aftertheunionoperations,checkifthetwovirtual

nodes for the player are now connected:
i. If they are connected, the player has won the game.

ii. Retrieve the count of connected gold cells from connected_gold_count for the set containing the two virtual nodes to calculate the bonus points.

7. **Bonus Points Calculation**
8. 8. At the moment of victory, access the connected_gold_count for the union- find set containing the two virtual nodes to get the count of connected gold cells.
9. 9. The number of bonus points is equal to this count.

Running Time Analysis:

1. **Initialization:** Initializing the union-find structure and other arrays takes O(n) time, where n is the number of cells on the board.
2. **Processing Each Move:** For each move:

   1. Checking adjacent cells involves a constant number of operations (up to 6 in a

hexagonal grid).

   2. Each union and find operation in the union-find structure with path compression

and union by rank has an amortized time complexity of $O(\alpha(n))$, where α is the inverse Ackermann function, which grows very slowly and is almost constant for practical input sizes.

   3. Updating the connected_gold_count during union operations involves a constant amount of work since it requires merging information about the connected gold cells.

3. **Total Running Time:** For a game consisting of n moves, the overall running time is:$O(n\alpha(n))$ Since $\alpha(n)$ is a very slow-growing function, the time complexity is nearly linear in practice.

Pseudocode:

Algorithm Initialize(board_size, gold_cells)
Input: board_size - total number of cells on the board

gold_cells - list of positions of gold cells
Output: Initializes the union-find structure and tracking arrays

```
for i = 0 to board_size + 2 do
    parent[i] = i # Each cell is its own parent initially
    rank[i] = 0 # Initial rank of each cell is 0
    connected_gold_count[i] = 0 # No gold cells connected initially
    if i in gold_cells then
        gold[i] = True
    else
        gold[i] = False

# Virtual nodes for each player
virtual_node_left = board_size # Index for the left edge virtual node
virtual_node_right = board_size + 1 # Index for the right edge virtual node
```

```
Algorithm Find(x)
Input: x - cell to find the representative of
Output: Returns the representative (root) of the set containing x

if parent[x] != x then
    parent[x] = Find(parent[x]) # Path compression

return parent[x]
```

```
Algorithm Union(x, y)
Input: x, y - cells to unite
Output: Unites the sets containing x and y, updates gold cell count

rootX = Find(x)
rootY = Find(y)

if rootX == rootY then
    return # Already in the same set

# Union by rank
if rank[rootX] < rank[rootY] then
    parent[rootX] = rootY
    connected_gold_count[rootY] += connected_gold_count[rootX] # Merge gold cell count
elif rank[rootX] > rank[rootY] then
    parent[rootY] = rootX
    connected_gold_count[rootX] += connected_gold_count[rootY]
else
    parent[rootY] = rootX
    rank[rootX] += 1
    connected_gold_count[rootX] += connected_gold_count[rootY]
```

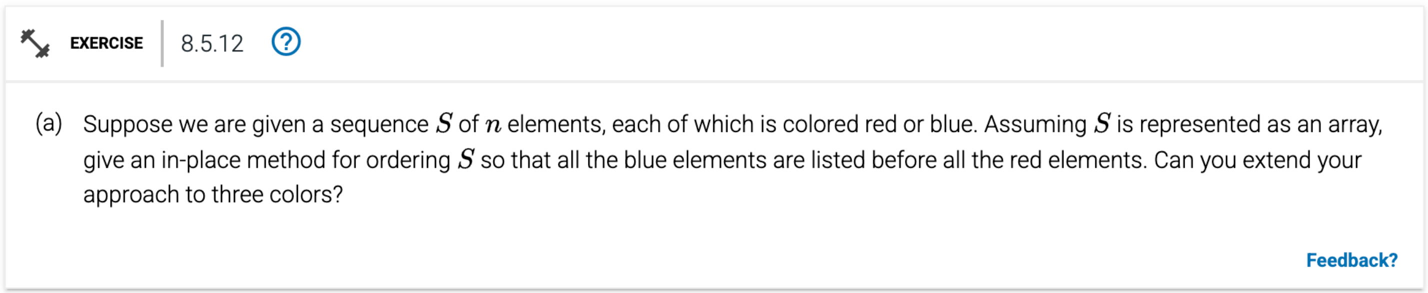
```
Algorithm PlaceStone(player, cell, board_size)
Input: player - the player placing the stone
       cell - the cell index where the stone is placed
       board_size - total number of cells on the board
Output: Places the stone, unites adjacent cells, and checks for win and bonus
```

```
points
# If the cell is a gold cell, update the gold count for this set

if gold[cell] == True then
root = Find(cell) connected_gold_count[root] += 1

# Check adjacent cells (up, down, left, right, diagonals) for each adjacent in AdjacentCells(cell) do

if adjacent is occupied by the current player's stone then Union(cell, adjacent)

# Connect to virtual nodes if the stone is on the board's edges if cell is on the player's left edge then

Union(cell, virtual_node_left)
if cell is on the player's right edge then

Union(cell, virtual_node_right)

# Check for win
if Find(virtual_node_left) == Find(virtual_node_right) then

# Player has won; calculate bonus points
root = Find(virtual_node_left)
bonus_points = connected_gold_count[root]
print("Player", player, "wins with", bonus_points, "bonus points!")

Algorithm AdjacentCells(cell)
Input: cell - the cell index to find adjacent cells for
Output: Returns a list of adjacent cell indices
# Return the valid neighboring cells of the current cell (up to 6 in a hexagonal

grid)
```

Conclusion:

By using a union-find data structure with path compression and union by rank, along with an additional mechanism to track connected gold cells, we can efficiently detect when a player wins and calculate their bonus points. The overall running time for processing all moves in the game is O(nα(n))O(nα(n)), which is optimal given the problem's constraints. This approach ensures both victory detection and bonus point calculation are handled efficiently throughout the game.

Creativity-8.5.12



EXERCISE | 8.5.12

(a) Suppose we are given a sequence S of n elements, each of which is colored red or blue. Assuming S is represented as an array, give an in-place method for ordering S so that all the blue elements are listed before all the red elements. Can you extend your approach to three colors?

Solution

Given a sequence SS of nn elements, each of which is colored red or blue, and assuming SS is represented as an array, we need to reorder SS in-place so that all blue elements come before all red elements. This is similar to the Dutch National Flag problem for two colors. The problem can also be extended to handle three colors efficiently.

(a) In-Place Method for Two Colors (Red and Blue)

We can solve this problem using a two-pointer approach. This method will run in O(n)O(n) time, using a single pass through the array, and requires no additional space.

Algorithm:

1. Use two pointers:
o left: Points to the position where the next blue element should be

placed.
o right: Traverses the array from the start to the end.

2. Traverse the array using the right pointer:
o If the element at S[right] is blue, swap it with the element at S[left]. o Increment left to move to the next position.
o Always increment right to continue scanning the entire array.

3. By the end of the traversal, all blue elements will be at the beginning, followed by all red elements.

Pseudocode:

```
Algorithm PartitionColors(S):
Input: S - an array of n elements, each colored 'red' or 'blue'
Output: Reorders the array so all 'blue' elements come before all 'red' elements

left = 0 # Pointer to place the next 'blue' element


for right = 0 to n - 1 do
if S[right] == 'blue' then
```

•

•

Swap S[left] with S[right] left = left + 1

Time Complexity: O(n)O(n), as we traverse the array once.
Space Complexity: O(1)O(1), since the rearrangement is done in-place without extra storage.

Extension to Three Colors

To extend this approach to handle three colors (e.g., 'blue', 'white', and 'red'), we can use the **Dutch National Flag algorithm**:

1. Use three pointers:

a. low:Markstheendofthe'blue'section.
b. mid: The current element being examined.
c. high:Marksthebeginningofthe'red'section.

2. The array is partitioned into three sections:

a. Fromthestarttolow-1:Allelementsare'blue'. b. From low to mid - 1: All elements are 'white'.
c. Frommidtohigh:Unprocessedelements.
d. From high + 1 to the end: All elements are 'red'.

3. The algorithm processes elements at mid:

1. IfS[mid]is'blue',swapitwithS[low]andincrement both low and mid.
2. If S[mid] is 'white', increment mid.
3. IfS[mid]is'red',swapitwithS[high]anddecrementhigh.

4. Repeat until mid is greater than high Pseudocode for Three Colors:

Algorithm PartitionThreeColors(S):
Input: S - an array of n elements, each colored 'blue', 'white', or 'red'
Output: Reorders the array so 'blue' elements come first, followed by 'white', then 'red'

```
low = 0 mid = 0 high = n - 1

while mid <= high do
if S[mid] == 'blue' then

Swap S[low] with S[mid] low = low + 1
mid = mid + 1

else if S[mid] == 'white' then mid = mid + 1

else if S[mid] == 'red' then Swap S[mid] with S[high] high = high – 1
```

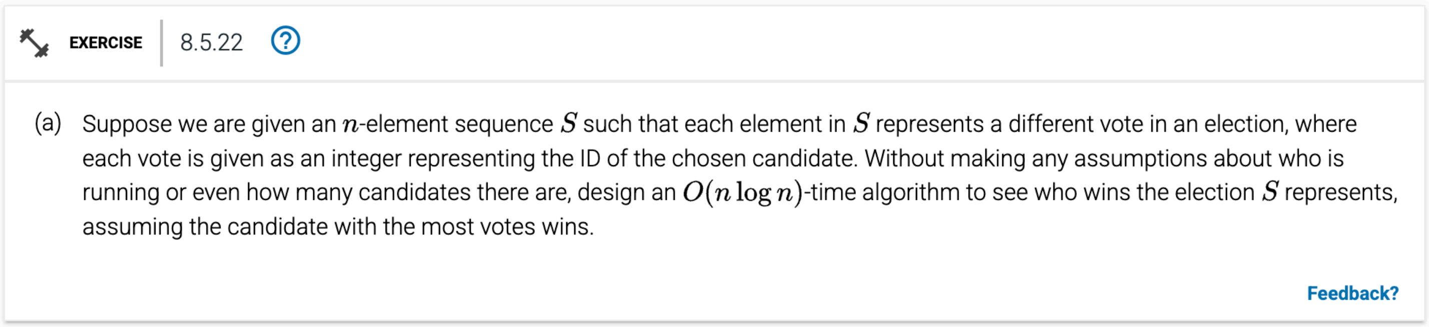
Time Complexity: O(n), as each element is processed at most once. **Space Complexity:** O(1), since the rearrangement is done in-place.

Conclusion

The in-place partitioning methods described above efficiently sort an array of colored elements in linear time. For the two-color problem (red and blue), the two- pointer approach ensures that all blue elements appear before all red elements in a single pass with O(n) time complexity and O(1) space complexity. This method is simple yet optimal for this specific partitioning task.

For the extended three-color problem, the Dutch National Flag algorithm provides a similarly efficient solution, allowing us to sort elements into three distinct groups in O(n) time while maintaining in-place sorting. Both algorithms are not only effective but also practical for scenarios where space efficiency is crucial. This makes them well-suited for real-world applications that require quick and memory- efficient sorting of categorical data.

Application-8.5.22



EXERCISE 8.5.22

(a) Suppose we are given an n-element sequence S such that each element in S represents a different vote in an election, where each vote is given as an integer representing the ID of the chosen candidate. Without making any assumptions about who is running or even how many candidates there are, design an $O(n \log n)$-time algorithm to see who wins the election S represents, assuming the candidate with the most votes wins.

Feedback?

Solution

To find the candidate with the most votes in an n-element sequence S where each element represents a vote for a candidate, we need an efficient algorithm. While the question asks for an O(nlogn) algorithm, it is possible to solve this problem in O(n) time using a more optimized approach. Let's explore both solutions.

Optimized O(n) Solution: Hash Map (Counting Frequencies)

The most efficient way to find the candidate with the most votes is to use a hash map (or dictionary) to count the frequency of each candidate’s ID in a single pass. This method uses additional space but achieves linear time complexity.

1. **Initialize a Hash Map:** Create an empty hash map (or dictionary) to store each candidate's ID as a key and their vote count as the value.

2. **Count Votes:** Iterate through the sequence SS:

   1. For each element (vote) in SS , use the candidate's ID to update their count
   2. in the hash map.
   3. If the candidate's ID is not already in the hash map, add it with a count of 1.
   4. If the candidate's ID is already present, increment their count by 1.

3. **Find the Maximum:** After processing all votes, iterate through the hash map to find the candidate ID with the highest vote count.

4. **Return the Winner:** Output the candidate ID that has the maximum votes.

Pseudocode:

Algorithm FindWinner(S):

```
Input: S - an array of n elements representing votes (candidate IDs) Output: The candidate ID with the most votes

# Step 1: Initialize an empty hash map to count votes vote_count = {}

# Step 2: Count votes for each candidate for each vote in S do

if vote in vote_count then vote_count[vote] = vote_count[vote] + 1

else

vote_count[vote] = 1

# Step 3: Find the candidate with the maximum votes max_votes = 0
winner = None
for candidate, count in vote_count.items() do

if count > max_votes then max_votes = count winner = candidate

# Step 4: Return the candidate with the most votes return winner
```

Complexity Analysis:

- Time Complexity: The algorithm makes two passes over the data:
- Counting votes in O(n) time.
- Finding the maximum in the hash map, which takes O(k) time, where kk is

the number of unique candidates. Since k≤n, this step is also O(n).

- Therefore, the overall time complexity is O(n).
- Space Complexity: The hash map requires O(k) space to store the vote

counts, where k is the number of unique candidates. In the worst case, k=n, so the space complexity is O(n).

O(nlogn) Solution: Sorting:

If we strictly follow the O(nlogn) constraint, we can use sorting. Here's how to do it:
1. **Sort the Sequence:** Sort the sequence SS using a comparison-based sorting

algorithm like merge sort or quicksort, which takes O(nlogn) time.
2. **Count Consecutive Elements:** Iterate through the sorted sequence to count

the number of times each candidate appears consecutively.
3. **Find the Maximum:** Keep track of the candidate with the maximum count

as you iterate through the sorted sequence.

Pseudocode:

1. Algorithm FindWinner_Sorting(S):

```
Input: S - an array of n elements representing votes (candidate IDs) Output: The candidate ID with the most votes

# Step 1: Sort the sequence Sort(S)

# Step 2: Initialize variables to track the current count and maximum count max_votes = 0
current_count = 1
winner = S[0]

# Step 3: Iterate through the sorted sequence to count consecutive elements for i = 1 to n - 1 do
    if S[i] == S[i - 1] then
        current_count = current_count + 1
    else
        if current_count > max_votes then
            max_votes = current_count
            winner = S[i - 1]
        current_count = 1 # Reset count for new candidate

# Check the last candidate's count if current_count > max_votes then
    winner = S[n - 1]

# Step 4: Return the candidate with the most votes return winner
```

Complexity Analysis

- **Time Complexity:** Sorting the sequence takes O(nlogn). The second pass to

count consecutive elements runs in O(n). Therefore, the overall time

complexity is O(nlogn).

- **Space Complexity:** The sorting step typically requires O(n) additional

space, especially for algorithms like merge sort. Thus, the space complexity is O(n).

Conclusion

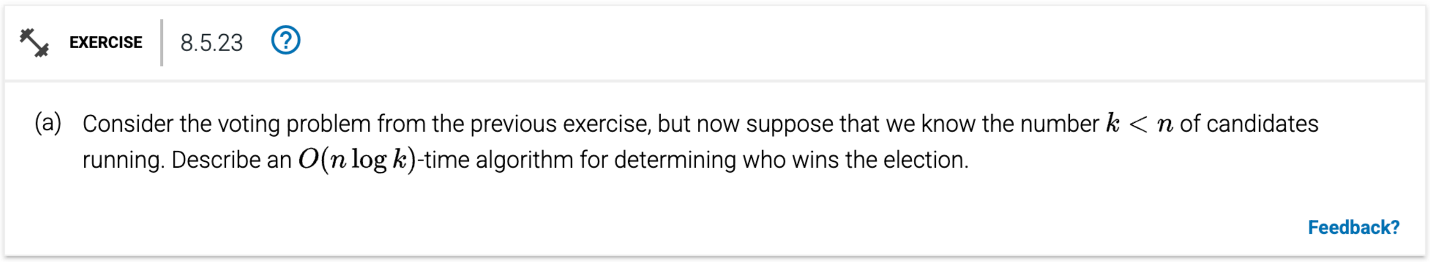
• The optimized **O(n) time algorithm** using a hash map is the most efficient

solution for finding the candidate with the most votes. It processes each vote

in linear time and directly counts frequencies, making it suitable for large input sizes.

• The **O(nlogn) time algorithm** using sorting is a valid approach if we must adhere strictly to the sorting constraint. However, the O(n) hash map solution is preferable in practical scenarios where we aim for the lowest running time.

Application-8.5.23



EXERCISE 8.5.23

(a) Consider the voting problem from the previous exercise, but now suppose that we know the number $k < n$ of candidates running. Describe an $O(n \log k)$-time algorithm for determining who wins the election.

Feedback?

Solution

Given that we now know there are k candidates running in the election and k<n, we can design a more efficient algorithm to find the candidate with the most votes in O(nlogk) time. By taking advantage of this information, we can use a more targeted counting approach than in the previous problem.

Efficient O(nlogk) Algorithm: Using a Min-Heap (Priority Queue):

Since we know there are only k candidates, we can utilize a min-heap (or priority queue) to keep track of vote counts efficiently. The key idea is to count the votes using a hash map and use the min-heap to maintain the candidates with the top vote counts as we process the sequence.

Step-by-Step Algorithm

1. **Initialize Data Structures:**

o Use a **hash map** (dictionary) vote_count to count the votes for each of the kk candidates.

o Use a **min-heap** of size kk to keep track of candidates and their counts as we process the votes.

2. **Count Votes:**

o Iterate through the sequence SS of n votes:

§ For each vote, update the candidate's count in the vote_count hash map.

§ If this candidate's count increases and qualifies it as one of the top kk candidates, add/update this candidate in the min-heap.

3. **Using the Min-Heap:**

o The min-heap helps to efficiently keep track of the candidate with the

minimum vote count among the top kk. If a candidate's vote count

exceeds the minimum in the heap, update the heap.

o This allows us to efficiently find the candidate with the most votes as

we go through the sequence. 4. **Determine the Winner:**

o After processing all votes, extract the candidate with the highest count from the min-heap.

Pseudocode:

```
Algorithm FindWinner(S, k):
Input: S - an array of n elements representing votes (candidate IDs)

k - the number of candidates (k < n) Output: The candidate ID with the most votes

# Step 1: Initialize a hash map to count votes vote_count = {}

# Step 2: Initialize an empty min-heap min_heap = []

# Step 3: Count votes for each candidate for each vote in S do

# Update the vote count in the hash map if vote in vote_count then

vote_count[vote] = vote_count[vote] + 1 else

vote_count[vote] = 1

# Step 4: Maintain the min-heap of size k if vote not in min_heap then

if size(min_heap) < k then
# Add new candidate to the heap
Insert min_heap with (vote_count[vote], vote)

else
# Check if the current candidate has more votes than the minimum in the

if vote_count[vote] > min_heap[0].count then ExtractMin(min_heap)
Insert min_heap with (vote_count[vote], vote)
```

```
# Step 5: Find the candidate with the maximum votes in the heap max_votes = 0
winner = None
for each (count, candidate) in min_heap do

if count > max_votes then max_votes = count winner = candidate

return winner
```

Explanation:

1. Hash Map for Counting: The hash map vote_count keeps track of the votes for each candidate in O(1) time per vote. This is necessary to update the counts efficiently as we iterate through the votes.
2. Min-Heap for Top Candidates: We use a min-heap of size k to keep track of the candidates with the highest counts. Inserting into the heap and extracting the minimum element both take O(logk) time.
3. Heap Maintenance: As we update the counts in the hash map, we use the min-heap to keep track of the candidates' counts efficiently, ensuring the heap size is always k. If a candidate's

count increases and it's not already in the heap, we check if it qualifies to be among the top k by comparing it to the smallest element in the heap.

4. Finding the Winner: After processing all votes, the heap contains the candidates with the highest counts, and we select the one with the maximum count.

Complexity Analysis

1. **Counting Votes:** Processing each vote and updating the hash map

takes O(1) time for each of the n votes, totaling O(n).

2. **Heap Operations:** Each insertion and extraction in the heap takes O(logk). Since we perform this operation for each of the n votes, this step takes O(nlogk) time in total.

3. **Overall Time Complexity:** The dominating term is O(nlogk).
4. **Space Complexity:** The hash map uses O(k) space to store counts for up

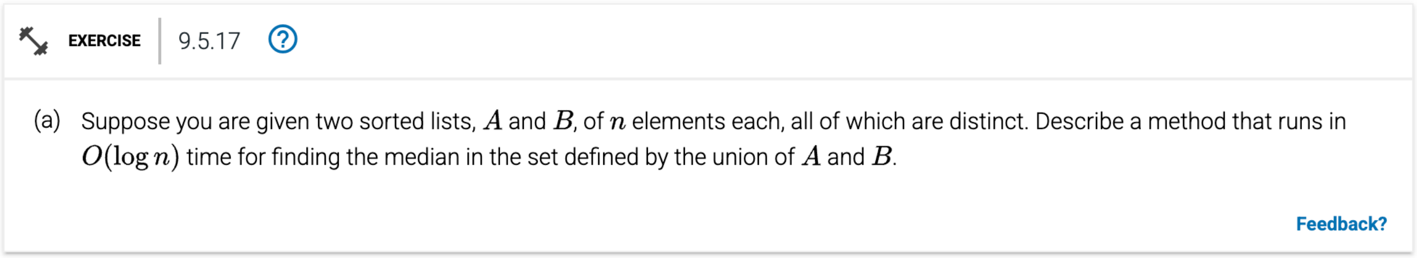
to k candidates, and the heap uses O(k) space as well.

Conclusion:

This algorithm efficiently finds the candidate with the most votes in O(nlogk) time by utilizing a hash map for counting votes and a min-heap to track the candidates with the highest counts. This approach is particularly useful when the number of candidates k is significantly smaller than the number of votes n. By combining these data structures, we ensure an optimal solution tailored to the given constraints of the problem.

Chapter 9

Creativity-9.5.17



EXERCISE 9.5.17

(a) Suppose you are given two sorted lists, A and B, of n elements each, all of which are distinct. Describe a method that runs in $O(\log n)$ time for finding the median in the set defined by the union of A and B.

Feedback?

Solution

To find the median of the union of two sorted lists A and B, each containing n distinct elements, in O(logn)time, we can leverage the properties of sorted arrays. The idea is to use a binary search approach on one of the arrays to determine the median position efficiently.

Explanation of the Problem

- Given:
  Two sorted lists A and B, each of size n. All elements are distinct.

- Goal:
  Find the median of the combined set A∪B. Since the total number of elements is 2n, the median will be the average of the n-th and (n+1)-th smallest elements in the merged list.

Approach: Binary Search Method
The key idea is to use binary search to find the correct partition point in both arrays where the median resides. We can perform the binary search on one of the arrays (say, A) and use properties of sorted arrays to deduce the correct partition in the other array B.
Step-by-Step Algorithm

1. Initialize Search Space:
Perform binary search on array A. Let the search range for indices in A be from 0 to n (inclusive).

2. Binary Search Loop:
While searching, consider a partition in A at index i (where i ranges from 0 to n) and a corresponding partition in B at index ` j = n - i). The index j ensures that the left half of the merged array contains exactly n elements when combining elements from both partitions.

o The elements on the left side of the partition are:
§ LeftA=A[i−1] (if i>0; otherwise, −∞if i=0) § LeftB=B[j−1](if j>0; otherwise, −−∞ if j=0)

o The elements on the right side of the partition are: § RightA=A[i] (if i<n; otherwise, ∞ if i=n)
§ RightB=B[j] (if j<n; otherwise, ∞ if j=n)

3. Check Partition Validity:
o The partition is valid if:

§ LeftA≤RightB and LeftB≤RightA.
o If the partition is valid, the median can be calculated as:

§ If n is even, the median is:median=max(LeftA,LeftB)+min(RightA,RightB)2

o If LeftA>RightB, it means we have partitioned too far into A. Move the search space to the left by setting high = i - 1.

o If LeftB>RightA, it means we need to move the partition in A to the right by setting low = i + 1.

4. Binary Search Termination:
o The binary search continues until the correct partition is found. Since

we are performing binary search on a range of size n, the time

complexity is O(logn).

Pseudocode

Algorithm FindMedianSortedArrays(A, B): Input: A - sorted array of size n

B - sorted array of size n
Output: Median of the combined set A ∪ B

low = 0 high = n

while low <= high do i = (low + high) / 2 j=n-i

LeftA = A[i-1] if i > 0 else -∞ RightA = A[i] if i < n else ∞ LeftB = B[j-1] if j > 0 else -∞ RightB = B[j] if j < n else ∞

# Check if we have found the correct partition if LeftA <= RightB and LeftB <= RightA then

# If even number of total elements

return (max(LeftA, LeftB) + min(RightA, RightB)) / 2 elif LeftA > RightB then

high=i-1 #MovethepartitioninAtotheleft else

low=i+1 #MovethepartitioninAtotheright

Complexity Analysis

- Time Complexity: The binary search on array A has a time complexity

of O(logn).

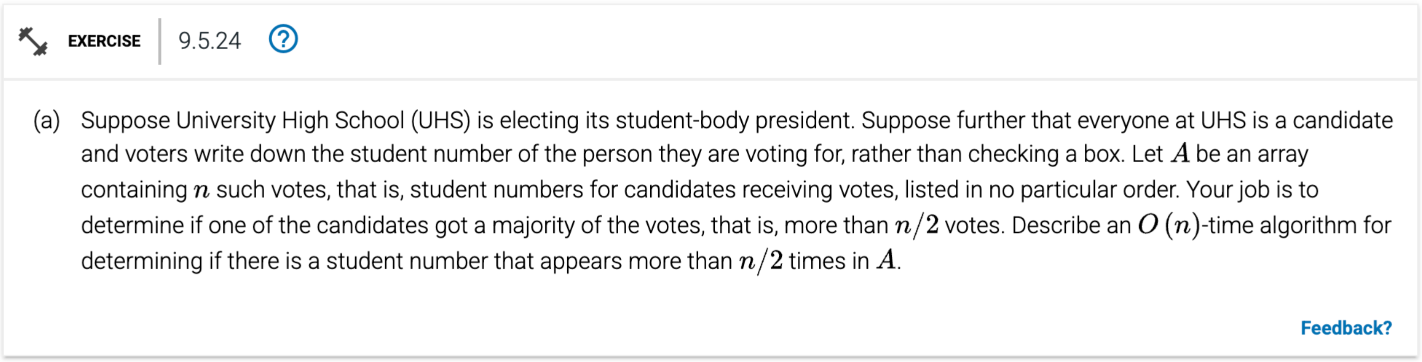
- Space Complexity: The algorithm uses a constant amount of extra

space, O(1).

Conclusion

• This binary search approach efficiently finds the median of the union of two sorted lists in O(logn) time. By leveraging the properties of sorted arrays, we avoid directly merging the lists, which would take O(n) time. This method ensures an optimal solution with the lowest possible running time given the problem's constraints.

Application-9.5.24


EXERCISE 9.5.24

(a) Suppose University High School (UHS) is electing its student-body president. Suppose further that everyone at UHS is a candidate and voters write down the student number of the person they are voting for, rather than checking a box. Let A be an array containing n such votes, that is, student numbers for candidates receiving votes, listed in no particular order. Your job is to determine if one of the candidates got a majority of the votes, that is, more than $n/2$ votes. Describe an $O(n)$-time algorithm for determining if there is a student number that appears more than $n/2$ times in A.

Feedback?

Solution

To find if there is a student number in the array A that appears more than n/2 times in O(n) time, we can use the **Boyer-Moore Voting Algorithm**. This algorithm allows us to identify a potential majority element in linear time and constant space. After identifying the candidate, we perform a second pass through the array to verify if it actually occurs more than n/2 times.

Boyer-Moore Voting Algorithm

This algorithm is designed to efficiently find the majority element in an array. It operates in two phases:

1. **Candidate Identification:** Find a potential candidate for the majority element.
2. **Verification:** Verify if the identified candidate is indeed the majority element.

Step-by-Step Algorithm:

1.Candidate Identification • Initialize two variables:

o candidate: To keep track of the potential majority candidate.

o count: To track the count of the current candidate. • Iterate through the array:

o If count is zero, set the current element as the candidate and set count to 1.

o If the current element is the same as candidate, increment count.
o If the current element is different from candidate, decrement count.

• At the end of this phase, candidate will hold the element that is potentially the majority element (if one exists).

2. Verification

- Initialize count to 0.
- Iterate through the array again to count the occurrences of the candidate.
- If the count is greater than n/2, the candidate is the majority element.
- Otherwise, no majority element exists.

Pseudocode:

```
Algorithm FindMajorityElement(A):
Input: A - an array of n elements (student numbers)
Output: The majority student number if it exists, otherwise None

# Step 1: Find the candidate for majority element
candidate = None
count = 0

for each student in A do
    if count == 0 then
        candidate = student
        count = 1
    else if student == candidate then
        count = count + 1
    else
        count = count - 1
```

```
# Step 2: Verify the candidate count = 0
for each student in A do
if student == candidate then count = count + 1
if count > n / 2 then return candidate
else
return None
```

Explanation

1. **Candidate Identification:** In the first pass, the algorithm uses a voting mechanism to identify a potential majority candidate. When count is zero, it picks the current element as the new candidate. It then adjusts the count based on whether subsequent elements match the current candidate.

2. **Verification:** The second pass is used to count the occurrences of the candidate to confirm if it appears more than n/2 times.

Complexity Analysis

- **Time Complexity:** The algorithm performs two passes through the array,

each taking O(n) time. Thus, the overall time complexity is O(n).

- **Space Complexity:** The algorithm uses a constant amount of extra space for

the candidate and count variables, so the space complexity is O(1).

1. **Conclusion**

The Boyer-Moore Voting Algorithm efficiently finds the majority element in an array in linear time and constant space. After identifying a potential majority candidate in the first pass, a second pass is used to confirm if it actually appears more than n/2 times. This approach guarantees an optimal solution with the lowest running time of O(n), making it suitable for large input sizes.