CSE 595: Advanced Topics in Computer Science
Presentation 2 and 3

Zeeshan Shaikh

Department of Computer Science, Stonybrook University

06/10/2021

Topics for today's presentation

- Given an array of size n and a number k, find all elements that appear more than n/k times.

Problem 2: Given an array of size n and a number k, find all elements that appear more than n/k times

Understanding the problem:

- Given: an array and a random value "k".
- Task: To find all the elements such the frequency of the element is greater than the arrayLength/k.
- Example:
  exampleArray = [1, 2, 3, 4, 1, 1, 2, 2, 5]
  k = 4
  Answer = [1, 2]

Possible Approaches:

- Naive Approach: Calculate the frequency of all the elements by traversing the array
- Using a HashMap
- Alternate Method
- Boyer-Moore Voting Algorithm ($n/2$, $n/3$ and n/k generalization)

Overview of Complexity Analysis of the discussed problems.

Complexity Analysis for the algorithms			
Algorithm	Time	Space	Side Notes
Hashmap	$O(N)$	$O(N)$	Still requires to iterate through array
Alternate Method	$O(N * K)$	$O(K)$	Double loops increase time complexity
Boyer-Moore	$O(N)$	$O(1)$	Manually writing if-else conditions is tedious for $k > 3$

Problem 2: Hash method

Algorithm 1 Hash algorithm for n/k

```
Calculate the
for i from 0 to n − 1 do
    Add to the count of the element in the hashmap
end for
for Iterate over hashmap do
    if elementCount > ratio then
        save the corresponding element
    end if
end for
```

Time Complexity: $O(N)$
Space Complexity: $O(N)$

Problem 2: Alternate method

Algorithm 2 Alternate algorithm for n/k

define a structure of length k-1 to hold the element and count
if element of array is already present in the structure **then**
    Increase its count
else if element of array is not present in the structure **then**
    if there is space in structure **then**
        Add element and set count to 1
    else
        Reduce the count of all elements by 1
    end if
end if

Problem 2: Alternate algorithm Visualization

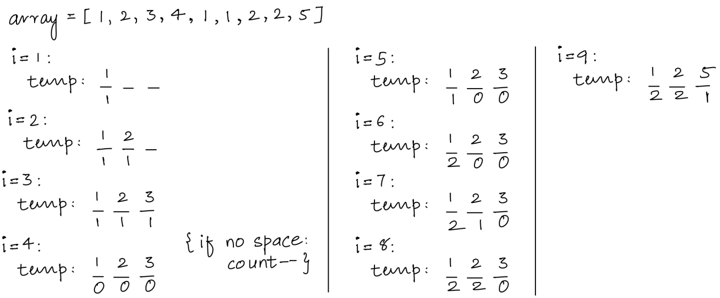


Figure 1: Structure element changes over the course of the algorithm

Time Complexity: $O(N * K)$
Space Complexity: $O(K)$

Introduction to Boyer-Moore Voting Algorithm

The usual variations of this problem, is to find the frequency of elements that occur more than n/3 times or n/4 time, with the constraint of linear time complexity and O(1) space.
LOGIC:

- There can be atmost 1 element with frequency more than n/2.
- There can be atmost 2 elements with frequency more than n/3.
- There can be atmost 3 elements with frequency more than n/4.
- There can be atmost k-1 elements with frequency more than n/k.

Problem 2: Boyer-Moore Voting Algorithm

Algorithm 3 Boyer-Moore Voting Algorithm for n/2

define variables to hold the element and count
if element == variable **then**
    Increase its count
else if $element != variable$ and $variablecount == 0$ **then**
    Add element and set count to 1
else
    Reduce the count by 1
end if

Problem 2: Alternate algorithm Visualization

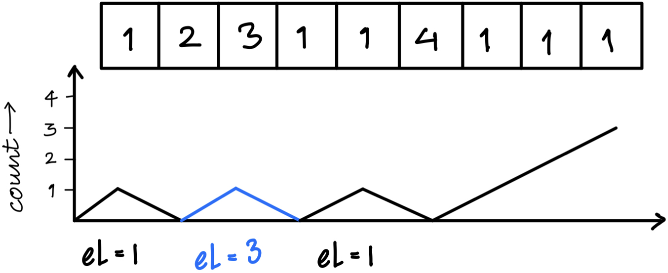


Figure 2: Boyer-Moore Voting Algorithm for n/3

Problem 2: Boyer-Moore Voting Algorithm

Algorithm 4 Boyer-Moore Voting Algorithm for n/3

```
define variables to hold the element and count
if element == variable1 OR element == variable2 then
    Increase its count
else if element != variable1 then
    if count == 0 then
        Add element and set count to 1
    else
        count − −
    end if
else if element != variable2 then
    if count == 0 then
        Add element and set count to 1
    else
        count − −
    end if
end if
```

Problem 2: Boyer-Moore Voting Algorithm for n/3

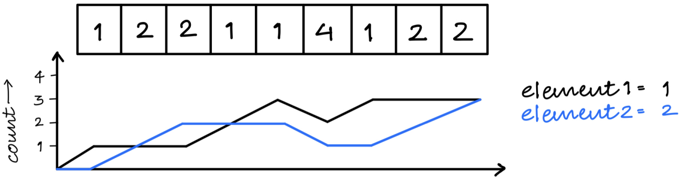


Figure 3: Element wise count changes over the course of the algorithm

Time Complexity: $O(N)$
Space Complexity: $O(1)$

Topics for today's presentation

- Trapping Rainwater

Problem 3: Trapping Rainwater

- Given: An Array with elements which represent the height of a bar.
- Task: To find the amount of water that can be trapped in between these bars.
- Example: $array = [3, 0, 2, 0, 3], answer = 7$

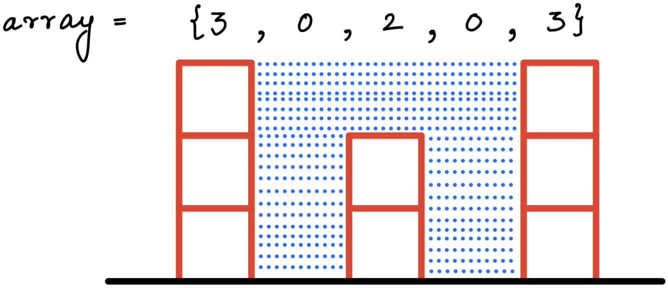


Figure 4: Visual representation of the problem

Overview of Complexity Analysis of the discussed problems.

Complexity Analysis			
Algorithm	Time	Space	Side Notes
Naive	$O(N^2)$	$O(1)$	Requires to iterate through the array, nested loops increase time complexity
Dynamic Prog.	$O(N)$	$O(N)$	Fastest method of the four
Stacks	$O(N^2)$	$O(N)$	Double loops increase time complexity
Two Pointer	$O(N)$	$O(1)$	Got to initialize or keep track of 4 variables

Problem 3: Real time problem run-time and space

Time Submitted	Status	Runtime	Memory	Language
06/07/2021 09:48	Accepted	12 ms	14.1 MB	cpp
06/07/2021 09:46	Accepted	4 ms	14.5 MB	cpp
06/07/2021 06:23	Accepted	0 ms	14.3 MB	cpp
06/07/2021 05:04	Accepted	496 ms	14.1 MB	cpp

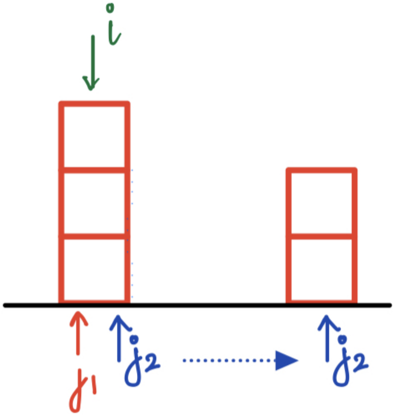
Problem 3: Naive Method

Algorithm 5 Naive algorithm

for i from 0 to $n-1$ **do**
    Initialize two variables to hold the maximum size on left and right side of the current element
    for j from i to 0 **do**
        leftMax = max(leftMax, currentElement)
    end for
    for j from i to $n-1$ **do**
        rightMax = max(rightMax, currentElement)
    end for
    $answer += min(leftMax, rightMax) - currentElement[i]$
end for

Time Complexity: $O(N^2)$
Space Complexity: $O(1)$

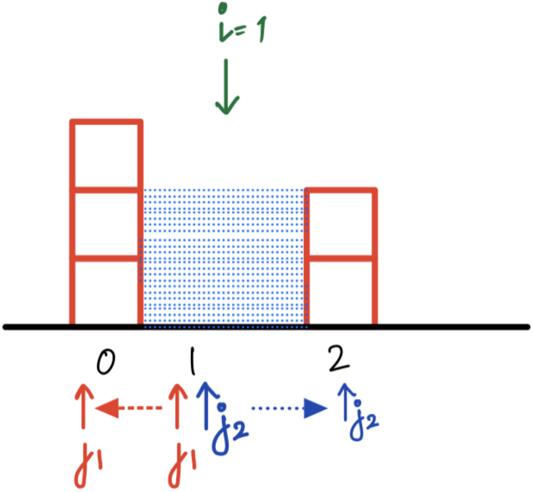
Problem 3: Naive algorithm Visualization



answer += min(left_max, right_max) - height[i]

answer += 3 - 3 = 0

Problem 3: Naive algorithm Visualization



answer += min(left_max, right_max) - height[i]

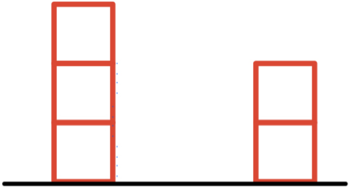
answer += 2 - 0 = 2

Problem 3: Dynamic Programming method

Algorithm 6 Dynamic Programming algorithm

Initialize two variables to hold the maximum size on left and right side of the current element
for i from 1 to $n-1$ **do**
    $leftMax[i] = max(leftMax[i-1], currentElement[i])$
end for
for j from $n-2$ to 0 **do**
    $rightMax[i] = max(rightMax[i+1], currentElement[i])$
end for
for j from 0 to $n-1$ **do**
    $answer+ = min(leftMax[i], rightMax[i]) - currentElement[i]$
end for

Problem 3: Dynamic Programming algorithm visualization



Init:

left_max[0] = array[0] = 3

right_max[2] = array[2] = 2

left_max[1] = max(3,0) = 3 & right_max[1] = max(2,0) = 2

left_max[2] = max(3,2) = 3 right_max[0] = max(2,3) = 3

Figure 5: Structure element changes over the course of the algorithm

Problem 3: Dynamic Programming algorithm visualization

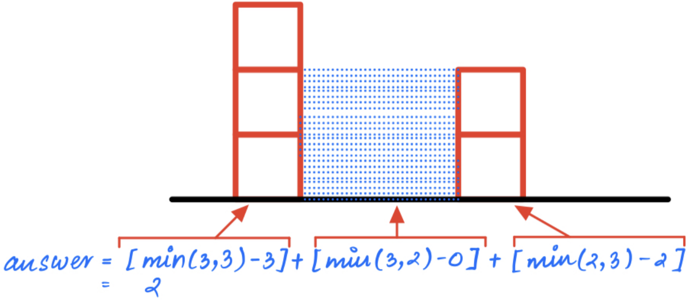


Figure 6: Structure element changes over the course of the algorithm

Problem 3: Stack method

Algorithm 7 Algorithm using Stack

for i from 0 to $n-1$ **do**
  while $(!stack.empty()$ **and** $currentElement > stack.top())$ **do**
    Store index of top element and pop
    $distance = currentTop - currentElement$
    $boundingHeight = min(currentTop, previousTop) - currentElement$
    $answer = boundingHeight * distance$
  end while
end for

What's the intuition behind Stack?

- Instead of storing the largest bar upto an index, we can use stack to store the bars that are bounded by longer bars
- If we encounter a bar $<=$ bar on top of the stack, we know it is bounded and hence add it to the stack.
- If we encounter a bar $>$ bar on top, we know it bounds the top as well the bar next in the stack.

Problem 3: Two Pointer method

Algorithm 8 Algorithm using two pointers

Initialize 4 variables to hold the two pointers and the max values for left and right respectively
while *leftIndex* $<=$ *rightIndex* **do**
    if *rightMax* $<=$ *leftMax* **then**
        $output+ = max(0, rightMax - arr[rightIndex])$
        Update rightMax and rightIndex(-1)
    else
        $output+ = max(0, leftMax - arr[leftIndex])$
        Update leftMax and leftIndex(+1)
    end if
end while

Problem 3: Two Pointer algorithm visualization

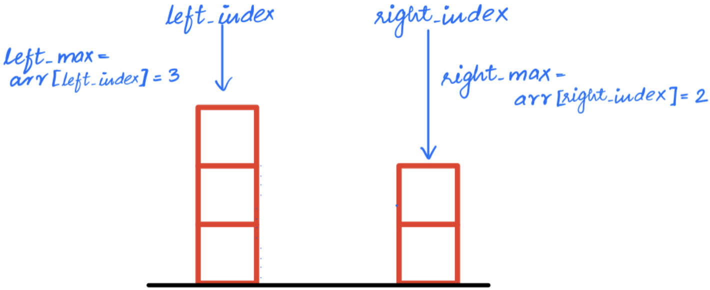


Figure 7: Structure element changes over the course of the algorithm

Problem 3: Two Pointer algorithm visualization

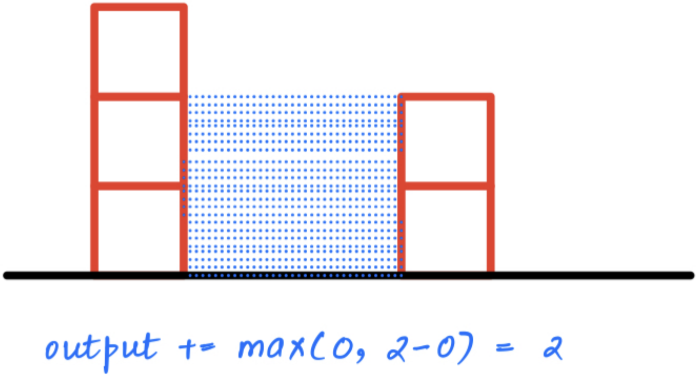
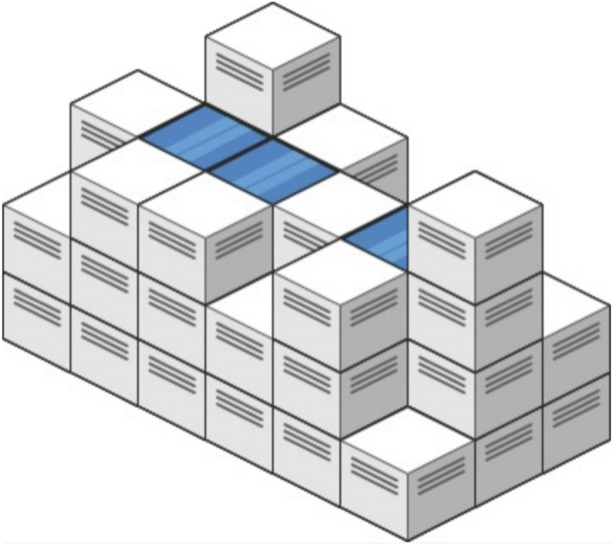


Figure 8: Structure element changes over the course of the algorithm

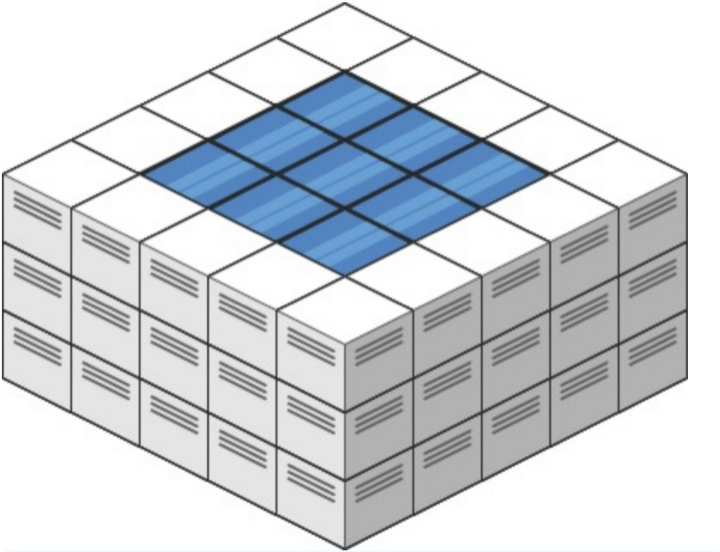
Extending this problem to 3D

- Can we extend this problem to 3D?
- The 3D problem requires an in-depth analysis of the problem.
- Let's see what it looks like

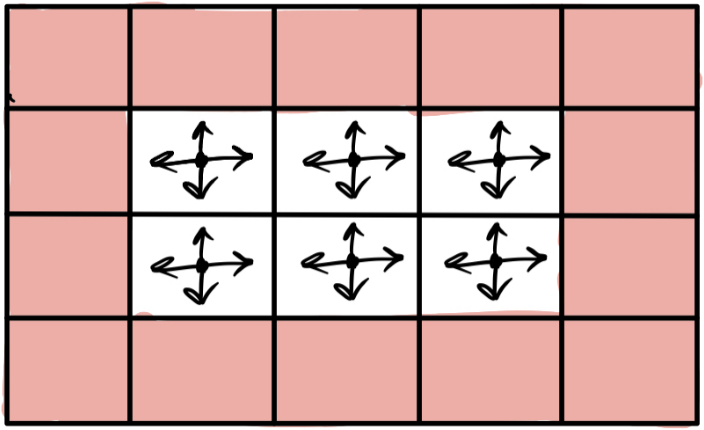
Problem 3: Visualization of the problem in 3D



Problem 3: Visualization of the problem in 3D



Problem 3: Intuitive understanding of the problem



- As discussed earlier, the amount of water that a cell can hold depends on the height of the smallest bar as its immediate neighbour i.e min(height of 4 neighbours).

Problem 3: Solving it in 3D

Algorithm 9 3D solution using MinHeap/Priority Queue

```
Initialize a heap and add the elements of the edge to the heap
while heap!empty do
    if currentHeight > maxHeight then
        Add the element to the heap
    else
        answer+ = maxHeight − currentHeight
        Loop over the neighbours to see if they're encountered before
        if !encountered then
            Recursive call
        end if
    end if
end while
```

Thankyou!