CSE 595: Advanced Topics in Computer Science
Presentation 1

Zeeshan Shaikh

Department of Computer Science, Stonybrook University

05/30/2021

Topics for today's presentation

- Count quadruplets with sum K from given array.
- Given an array of size n and a number k, find all elements that appear more than n/k times.

Problem 1: Count quadruplets with sum K from given array.

Understanding the problem:

- Given: an array and a target sum.
- Task: To find all possible quadruplets in the array, sum of which is equal to the given target sum.
- Example:
  exampleArray = [1, 2, 3, 4, 5]
  targetSum = 10
  Answer = 1 [subset:[1,2,3,4] satisfies!]

Possible Approaches:

- Naive Approach: Calculate the sum of all 4-element-combinations and check if it's equal to the required sum.
- Using a HashMap:
  - One Pass HashMap
  - Two Pass HashMap
- Double Pointer Method

Problem 1: Naive Approach

Algorithm 1 Naive Approach Algorithm

```
for i = 1 to N − 3 do
    for j = i + 1 to N − 2 do
        for k = j + 1 to N − 1 do
            for l = k + 1 to N do
                sum = array[i] + array[j] + array[k] + array[l]
                if sum = targetSum then
                    Increase counter
                end if
            end for
        end for
    end for
end for
```

Problem 1: Naive Approach Visualization

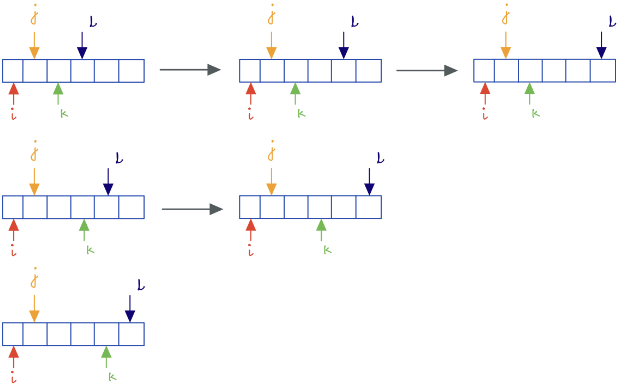

Figure 1: Movement of the iterators in the Naive Approach

Time Complexity: $O(N^4)$
Space Complexity: $O(N)$

Problem 1: HashMap Approach - Double Pass

Algorithm 2 Double Pass HashMap Algorithm

```
for i = 0 to N − 3 do
    for j = i + 1 to N − 2 do
        remainingSum = targetSum − (array[i] + array[j])
        for k = j + 1 to N do
            Create entries in the hashmap for values corresponding
            to array[k]
        end for
        for k = j + 1 to N − 1 do
            finalElement = remainingSum − array[k]
            if finalElement in hashmap then
                Increase counter
            end if
        end for
    end for
end for
```

Problem 1: Double Pass Hashmap Visualization

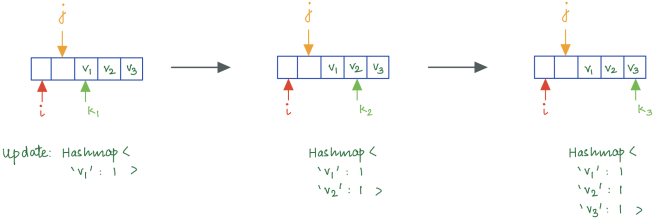

Figure 2: Movement of the iterators and update in the two-pass HashMap Approach

Problem 1: HashMap Approach - Single Pass

Algorithm 3 Single Pass HashMap Algorithm

```
for i = 0 to N − 1 do
    for j = i + 1 to N do
        tempSum = array[i] + array[j]
        remainingSum = targetSum − tempSum
        if tempSum < targetSum then
            counter += hashmap[remainingSum]
        end if
    end for
    for j = 0 to i do
        tempSum2 = array[i] + array[j]
        if tempSum2 < targetSum then
            Increase the counter of tempSum2 in hashmap
        end if
    end for
end for
```

Problem 1: Single Pass Hashmap Visualization

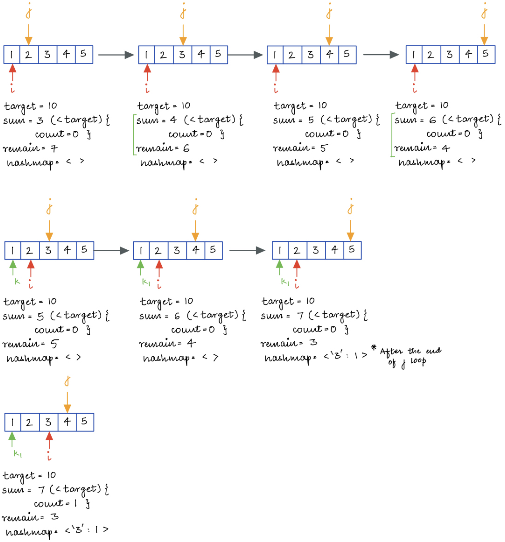

Figure 3: Movement of the iterators and update in the single-pass HashMap Approach

Complexity Analysis for Hashmaps

- Two Pass: Time Complexity: $O(N)$
  Space Complexity: $O(N)$
- One Pass: Time Complexity: (Worst case: linear, Average case: constant) $O(N)$
  Space Complexity: $O(N)$

Introducing the twoSum problem and solution using pointer approach

Algorithm 4 Basic two pointer algorithm

Initiate two pointers such that:
$low = 0, high = lengthArray - 1$
while $high > low$ **do**
    $sum = array[low] + array[high]$
    if sum = targetSum **then**
        increment counter
        increment low
        $high = lengthArray - 1$
    else if $sum > target$ **then**
        decrement high
    else
        increment low
    end if
end while

Pointer Solution Vizualisation

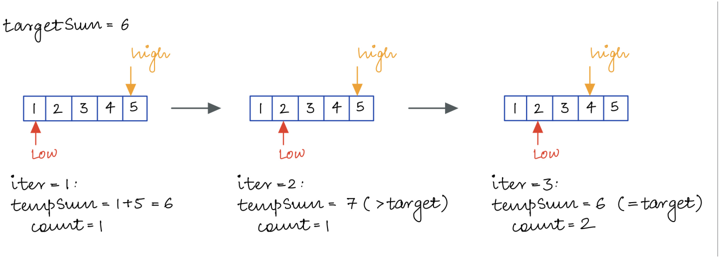

Figure 4: Pointer movement over the course of the algorithm

Time Complexity: $O(N^4)$
Space Complexity: $O(N)$

Can we generalize the two pointer algorithm?

- The two point algorithm is a very efficient algorithm to solve the two-sum problem.
- The question now is, can we generalize it for 3-sum, 4-sum ... k-sum algorithms?
- We can do so by converting any k-sum to a 2sum algorithm with k-2 loops.
- For example: If we take the 3-sum problem, we can fix the first element and then treat the remaining array as a 2-sum problem and apply the algorithm discussed in the previous slide.
- How would it look for 4-sum problem then?

Problem 1: Two pointer method

Algorithm 5 Two pointer algorithm for 4-sum

```
for i = 0 to arrayLength − 3 do
    for j = i + 1 to arrayLength − 2 do
        remain = targetSum − (array[i] + array[j])
        low = i + 1, high = lengthArray − 1
        while high > low do
            sum = array[low] + array[high]
            if sum = remain then
                increment counter, low
                high = lengthArray − 1
            else if sum > remain then
                decrement high
            else
                increment low
            end if
        end while
    end for
end for
```

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

Problem 2: Hash method

Algorithm 6 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 7 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

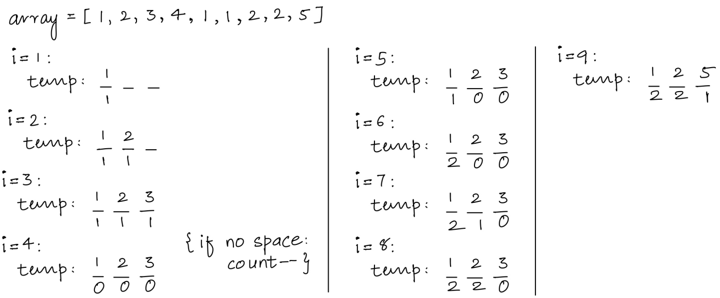

Figure 5: Structure element changes over the course of the algorithm

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

Thankyou!