CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2024**

Homework #2

Name(s): Raashi Maheshwari, Tanvi Yamarty

Homework Policy

- If you leave a question completely blank, you will receive 20% of the grade for that question. This however does not apply to the extra credit questions.
- You may also consult all the materials used in this course (lecture notes, textbook, slides, etc.) while writing your solution, but no other resources are allowed.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "sort the array in $\Theta(n \log n)$ time using merge sort".
- Remember to always **prove the correctness** of your algorithms and **analyze their running time** (or any other efficiency measure asked in the question). See "Practice Homework" for an example.
- The extra credit problems are generally more challenging than the standard problems you see in this course (including lectures, homeworks, and exams). As a general rule, only attempt to solve these problems if you enjoy them.
- **Groups:** You are allowed to form groups of size *two* or *three* students for solving each homework (you can also opt to do it alone if you prefer). The policy regarding groups is as follows:
  - You can pick different partners for different assignments (e.g., from HW1 to HW2) but for any single assignment (e.g., HW1), you have to use the same partners for all questions.
  - The members of each group only need to write down and submit a single assignment between them, and all of them will receive the same grade.
  - For submissions, only one member of the group submits the full solutions on Canvas and lists the name of their partners in the group. The other members of the group also need to submit a PDF on Canvas that contains only a single line, stating the name of their partner who has submitted the full solution on Canvas (Example: Say A, B, and C are in one group; A submits the whole assignment and writes down the names A, B, and C. B and C only submit a one-page PDF with a single line that says "See the solution of A").
  - You are allowed to discuss the questions with any of your classmates even if they are not in your group. **But each group must write their solutions independently.**

Problem 1. For this problem, we are assuming that arithmetic operations take $O(1)$ time. So addition, subtraction, division, and multiplication, are done in $O(1)$ time no matter how big the numbers are. (This is not exactly true in practice, but it's a reasonable approximation to reality.)

Now consider the following function SQRT(n)

- Input: a positive integer n
- Output: an integer k such that $k^2 = n$, or "No Solution" if none exists

Example: SQRT(9) should output 3, while SQRT(10) should output "no solution"

Write pseudocode for a recursive algorithm that computes SQRT(n) in time $O(\log(n))$. In addition to pseudocode, you should state the recursion formula for your algorithm. So all you need is pseudocode and the recursion formula, nothing else. You do NOT need to prove that this recursion formula solves to $T(n) = O(\log(n))$ **(15 points)**

Solution. In order to solve this problem, we can keep in mind one key thing: the square root of any number n should be between 1 and $\frac{n}{2}$. This is because if the square root value is greater than $\frac{n}{2}$, when we square that number, we will get a number greater than n, which does not work.

Pseudocode:

```
Function SquareRoot(n, lo, hi):

    # Base case: low is greater than high, there is no solution
    IF lo > hi THEN
        RETURN "no solution"

    # Calculate middle
    mid = (lo + hi) // 2

    # Check to see if the middle value squared is equal to n
    IF mid * mid == n THEN
        RETURN mid

    # Check to see if the middle value squared is less than n
    IF mid * mid < n THEN
        RETURN SQRT(n, mid + 1, hi) # check the lower half of the array

    ELSE # check to see if the middle value squared is greater than n
        RETURN SQRT(n , lo, mid - 1) # check the upper half of the array
```

The algorithm runs using the same principle behind binary search. The function takes in three parameters: n (the number we have to find the square root of), hi (the upper bound of the search range), and lo (the lower bound of the search range).

Our base case is the first IF statement that checks if lo is greater than hi. If it is, then our search has gone through all the values and has failed to find the square root, so the algorithm returns "no solution".

Our three recursive cases that the algorithm is checking are:

1. If $mid \times mid$ is equal to n, the function returns mid since it would be the square root.
2. If $mid \times mid$ is less than n, it searches the lower half of the range/array (from lo to $mid - 1$).
3. If $mid \times mid$ is greater than n, it searches the upper half of the range (from $mid + 1$ to hi).

The recursion formula for this would be:

$$T(n) = T\left(\frac{n}{2}\right) + O(1).$$

This is because each recursive call halves the space that we are searching ($T(n/2)$) and each operation within the search space (the multiplication operation) takes constant time to execute ($O(1)$). The recursion formula then simplifies to a time complexity of $O(\log n)$.

Problem 2. Suppose we have an array $A[1 : n]$ of n *distinct* numbers. For any element $A[i]$, we define the **rank** of $A[i]$, denoted by $rank(A[i])$, as the number of elements in A that are strictly smaller than $A[i]$ plus one; so $rank(A[i])$ is also the correct position of $A[i]$ in the sorted order of A.

Suppose we have an algorithm **magic-pivot** that given any array $B[1 : m]$ (for any $m > 0$), returns an index i such that $rank(B[i]) = m/2$ and has worst-case runtime of $O(m)$[1].

Example: if $B = [1, 7, 6, 3, 13, 4, 5, 11]$, then **magic-pivot**$(B)$ will return index 7 as rank of $B[7] = 5$ is 4 which is $m/2$ in this case.

(a) Use **magic-pivot** as a black-box to obtain a deterministic quick-sort algorithm with worst-case running time of $O(n \log n)$. **(10 points)**

(b) Use **magic-pivot** as a black-box to design an algorithm that given the array A and any integer $1 \leq r \leq n$, finds the element in A that has rank r in $O(n)$ time[2]. **(15 points)**

Solution. To solve this problem, we can use the concept of splitting the array into subarrays and using recursion.

Pseudocode for quick-sort algorithm using a magic pivot

```
FUNCTION QuickSort(A):
    IF length(A) <= 1 THEN
        RETURN A

    // Step 1: Use magic-pivot to find a pivot
    pivotIndex = magic-pivot(A)
    pivotValue = A[pivotIndex]

    // Step 2: Create subarrays
    leftArr = []
    middlePivotVal = [pivotValue]
    right = []

    // Step 3: Parition the array into the corresponding subarrays, appending it
    //to the left and right subarrays as needed
    FOR each element in A:
        IF element < pivotValue THEN
            leftArr.append(element)
        ELSE IF element > pivotValue THEN
            rightArr.append(element)

    // Step 4: Combine left subarray after running quicksort, the middle pivot value,
    //and the right subarray after running quicksort and return
    RETURN QuickSort(leftArr) + middlePivotVal + QuickSort(rightArr)
```

This method takes in an array A. It then finds the middle element in the array by using the concept of magic pivot and doing:

$$\text{pivot} = A[\text{magicPivot}(A)]$$

After finding the magic pivot value, it goes on to create three subarrays:

[1]Such an algorithm indeed exists, but its description is rather complicated and not relevant to us in this problem.

[2]Note that an algorithm with runtime $O(n \log n)$ follows immediately from part (a) – sort the array and return the element at position r. The goal however is to obtain an algorithm with runtime $O(n)$.

1. The middle value is stored in an array of size 1. This array essentially just stores the actual pivot value.
2. The right side of the array stores all the values that are greater than the pivot value. This is known as the right subarray.
3. The left side of the array stores the values from the original array that are less than the pivot value. This is known as the left subarray.

After creating the three subarrays, the algorithm proceeds to combine the results using QuickSort on the left and right subarrays and appending them with each other and the middle pivot value:

$$\text{QuickSort(left subarray)} + \text{middle pivot} + \text{QuickSort(right subarray)}.$$

Solution. To solve this problem, we can use the concept of splitting the array into subarrays and using recursion.

Pseudocode for a rank algorithm using a magic pivot

```
FUNCTION SearchElementWithRank(A, r, lo, hi):
    // Base case
    IF lo == hi THEN
        RETURN A[lo]  // The only element is the rank r

    //Create the sub-array for rank finding
    rankFindingArr = A[lo:hi]

    // Step 2: Use magic-pivot to find a pivot
    pivotIndex = magic-pivot(rankFindingArr)
    pivotValue = A[pivotIndex]

    //Determine the rank of the pivot
    rankPivot = rank(pivotValue)

    //Determine which side to search in
    //if its the middle, you can return the pivotValue
    //if rank of pivot is greater than r, search the lower half (the left subarray)
    //if rank of pivot is less than r, search the upper half (the right subarray)

    IF rankPivot == r THEN
        RETURN pivotValue
    ELSE IF rankPivot > r THEN
        // Search in the left side
        RETURN SearchElementWithRank(A, r, lo, pivotIndex - 1)
    ELSE
        // Search in the right side, adjust r
        RETURN SearchElementWithRank(A, r - rankPivot, pivotIndex + 1, hi)
```

This algorithm is using the following approach:

1. If the array A only has one element in it, you can return that singular element. This is the base case.

2. When there is more than one element in the array, you have to calculate the value of the pivot and get its rank. This step will be needed to determine the three recursive cases we focus on.
3. The next part of the algorithm is the three recursive cases:
    (a) If the rank of the pivot is equal to r, we can just return the pivot.
    (b) If the rank of the pivot is less than r, we need to search in the right/lower half and adjust the search parameters and r as needed.
    (c) If the rank of the pivot is greater than r, we need to search in the left/upper half and adjust the search parameters and r as needed.

Problem 3. Given an array A, we say that elements $A[i]$ and $A[j]$ are swapped if $j > i$ but $A[j] < A[i]$.

Example: If $A = [8, 5, 9, 7]$, then there are a total of 3 swapped pairs, namely 8 and 5; 8 and 7; and 9 and 7.

Example:: if $A = 2, 3, 4, 5, 1, 6, 7, 8$ then there are a total of 4 swapped pairs: 1 and 2; 1 and 5; 1 and 3; 1 and 4.

The goal for this problem is to write a recursive algorithm that given an array A determines the number of swapped pairs in the array in $O(n \log(n))$ time. The algorithm to do this is extremely similar to merge sort; in fact, the algorithm does all of merge sort (verbatim), plus a few extra things to keep track of the number of swaps. In particular, one side effect of the algorithm is to sort the array A.

what you need to do: In order to save you the trouble of retyping the pseudocode for merge sort,I have written the pseudocode for most of the algorithm. Your job is simply to fill in two lines. These are the key lines that keep track of the number of swaps.

Note that the algorithm below has an outer function MergeSortAndCountSwaps which I wrote entirely; nothing to fill in. But MergeSortAndCountSwaps then calls MergeAndCountSwapsBetween as a subroutine; in that subroutine you will have to fill in two lines.

The Algorithm MergeSortAndCountSwaps(A)
INPUT: array A with unique elements (no duplicates)
OUTPUT: pair (SortedA, S), where SortedA is the array A sorted in increasing order and S is the total number of swapped pairs in A.

- A1 $\leftarrow$ first half of A. So $A1 = A[0], ..., A[\frac{n}{2} - 1]$
- A2 $\leftarrow$ second half of A. So $A2 = A[\frac{n}{2}], ..., A[n - 1]$
- (SortedA1,S_1) $\leftarrow$ MergeSortAndCountSwaps(A1)
- (SortedA2,S_2) $\leftarrow$ MergeSortAndCountSwaps(A2)
- (SortedA, S_3) $\leftarrow$ MergeAndCountSwapsBetween(SortedA1,SortedA2)
- Return (SortedA, $S_1 + S_2 + S_3$).

We now need to define MergeAndCountSwapsBetween(A, B). This is the part where I will have you fill in a few lines. Note that MergeAndCountSwapsBetween(A, B) is very similar to Merge(A,B) from class, and in particular it is NOT a recursive function.

MergeAndCountSwapsBetween(A,B):

```
1. Initialize array $C$ of size $2n$
2. $i \leftarrow 0$
3. $j \leftarrow 0$
4. $S' \leftarrow 0$        Comment: $S'$ will count swaps; the answer lines below which you have to add will be
   responsible for changing $S'$.
5. While (i < n OR j < n)
    (a) if $j = n$ or A[i] < B[j] :
        • set next element of $C$ to $A[i]$
        • (formally, $C[i + j] \leftarrow A[i]$)
```

```
            • i ← i+1
       (b) if i = n or A[i] > B[j]:
            • set next element of $C$ to $B[j]$
            • j ← j+1
  6. output pair $(C, S')$.
```

Questions you need to answer:

- What goes in Answer 1? **S' ← S' + j**
- What goes in Answer 2? **S' ← (n - i)**

NOTE: please don't recopy all of the pseudocode above. **(20 points)**

Problem 4. Write an algorithm in pseudocode:

- Input: An array A with n distinct (non-equal) elements
- Output-1: numbers x and y in A that minimize $|x - y|$, where $|x - y|$ denotes absolute-value(x-y). (If there are multiple closest pairs, you only have to return one of them.)
- Output-2: a pair of numbers w and z in A that minimize $|w + z|$.

The run-time should be significantly better than $O(n^2)$. **(25 points)**

Solution. Output - 1: minimize $|x - y|$

```
FUNCTION MinimizeDifference(A):
    #Sort the array A
    Sort(A)

    #Initialize variables to track the minimum difference and the closest pair

  min_diff = $\infty$

    x = A[0]
    y = A[1]

    #Traverse the now sorted array to find the closest pair
    FOR i = 1 to n-1 DO
        diff = |A[i] - A[i-1]|

        IF diff < min_diff THEN
            min_diff = diff
            x = A[i-1]
            y = A[i]

    RETURN (x, y)

END FUNCTION
```

The first thing we did here was to sort the array, which takes a running time of $O(n \log n)$.

After the array is sorted, we initialized our variables. We use a really large value (such as infinity) to store the minimum difference and also initialize two other variables, x and y, to store the closest pair of elements found during the traversal.

Our algorithm uses a loop to continue traversing through all the pairs of consecutive elements in the sorted array. If any of the pairs returns a smaller difference than the min_diff, we update the values of the variables x and y.

Once the loop runs through all the elements, the x and y variables store the closest pair with the minimum difference, so we can just return the pair (x, y).

Our time complexity is $O(n \log n)$, as sorting the array takes $O(n \log n)$ and traversing the array to find the minimum difference pair takes $O(n)$.

Problem 5. Given an integer array, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum. Give a divide and conquer algorithm that runs much faster than $O(n^2)$. **(20 points)**

Solution. To solve this problem, as the question states, we can use the divide and conquer algorithm. This algorithm is also known as a form of a divide and conquer algorithm and is a very famous LeetCode algorithm as well.

A brute-force $O(n^2)$ algorithm involves having two nested loops and iterating over combinations of subarrays to get a maximum sum. We would use a 'maxSum' variable, and have the outer loop's pointer i represent the start of the subarray, and the inner loop's pointer j represent the end of the subarray. Every time we compute the sum of the subarray, we would make a comparison check to 'maxSum', which would be initialized to 0 initially.

However, for this divide and conquer algorithm, we can start by dividing the array into two parts (two halves), and then use recursion to find the maximum subarray that exists in the right half and also in the left half. We can also find the maximum sum of sub array that intersects at the midpoint of the array itself. And then every time, we can return the greatest sum out of the left sub array, the middle sub array, and the right sub array.

Let us examine the psuedo code for this algorithm in detail:

```
main:
int left=0
int right =array.length-1

maxSubArray(array, left, right)
    if right == left:
        return array[left]

    mid = (left + right) / 2

    maxOfLeft = maxSubArray(array, left, mid)
    maxOfRight = maxSubArray(array, mid + 1, right)
    maxOfMid = maxSumIntersect(array, left, right, mid)

    return max(maxOfLeft, maxOfRight, maxOfMid)
```

Now let's write the recursive function for the maximum sum of the middle (intersection) subarray:

```
maxSumIntersect(array, left, right, mid)
    sum = 0
    sumLeft = -infinity
    for i = mid downto left:
        sum = sum + array[i]
        if sum > sumLeft:
            sumLeft = sum

    sum = 0
    sumRight = -infinity
    for i = mid + 1 to right:
        sum = sum + array[i]
        if sum > sumRight:
            sumRight = sum
    return sumLeft + sumRight
```

This algorithm has a time complexity of $O(n \log n)$, because every time we are diving our space into half the array by using divide and conquer which reduces our run time exponentially. For finding the middle intersection sum we rely on linear time.

Problem 6. a) An **unimodal array** first increases to a peak and then decreases. Design a divide-and-conquer algorithm to find the maximum element of a unimodal array in $O(\log n)$ time. **(10 points)**

Solution. When we see a runtime of $O(\log n)$, most times, we think of the divide and conquer approach. To find the maximum element of a unimodal array in $O(\log n)$ time, we can implement binary search. The reason that binary search counts as a "divide and conquer" is because as you progress through the iterations, you will see that you are splitting the array into a subarray depending on if the target you are searching for is greater than or less than the middle value. The pseudo-code for the algorithm is listed below:

```
function findMax(array):
    leftIndex = 0
    rightIndex = array.length - 1

    while (leftIndex < rightIndex):
        middle = (leftIndex + rightIndex) // 2
        if (array[middle + 1] > array[middle]): # the array is still increasing
            leftIndex = middle + 1  # move search to the right of middle
        else:
            rightIndex = middle  # peak is to the left or at middle
    return array[leftIndex]
```

---

b) You are given an array $A[1..n]$ with the following properties:

- $A[1] > A[2]$ (the first element is greater than the second).
- $A[n-1] \leq A[n]$ (the second-to-last element is less than or equal to the last).

A **local minimum** is an element $A[x]$ such that:

$$A[x-1] \geq A[x] \leq A[x+1].$$

For example, in the array $A = [9,7,7,2,1,3,7,5,4,7,3,3,4,8,6,9,6]$, there are six local minima.

Design an $O(\log n)$-time algorithm to find a local minimum. **(10 points)**

Solution. In the previous step, we tried to find the maximum of a unimodal array, and now we are trying to find the "local minimum". Again, it is possible to use binary search to get an $O(\log n)$ approach to solve this question. We would be splitting our searching space by half for every iteration. Let us look at the algorithm below:

```
function minimum(array):
    leftIndex = 1
    rightIndex = array.length - 2
    // we use 1 as our starting and length - 2 as our ending indexes
    // question states that A[1] > A[2] and A[n-1] < A[n].

    while (leftIndex < rightIndex):
        middle = (rightIndex + leftIndex) // 2
        if (array[middle - 1] >= array[middle] and array[middle] <= array[middle + 1]):
            return array[middle]
        elif array[middle] > array[middle - 1]:
            rightIndex = middle - 1
        else:
            leftIndex = middle + 1
```

In this algorithm, we can see that we are trying to see where the local minimum occurs because we are looking for a "valley". We are seeing where the middle is between the lowest sequence of values and the highest sequence of values, and based on that, we are moving the search to the right or left.

---

c) An array $A[0..n-1]$ of distinct elements is **bitonic** if there exist unique indices i and j such that:

- $A[i-1] < A[i] > A[i+1]$ (a peak),
- $A[j-1] > A[j] < A[j+1]$ (a valley).

For example:

- $[4, 6, 9, 8, 7, 5, 1, 2, 3]$ is bitonic.
- $[3, 6, 9, 8, 7, 5, 1, 2, 4]$ is not bitonic.

Design an $O(\log n)$-time algorithm to find the smallest element in a bitonic array. **(10 points)**

Solution. Following the same divide and conquer logic from part a and b, we can apply the binary search algorithm for this as well. Let us examine the pseudo code of the algorithm below:

```
function findBitonic(array):
    leftIndex = 0
    rightIndex = array.length - 1

    while (leftIndex < rightIndex):
        middle = (leftIndex + rightIndex) / 2
        if (array[middle] < array[middle + 1] and array[middle - 1] > array[middle]):
            return array[middle]
        else if array[middle + 1] < array[middle]:
            leftIndex = middle + 1
        else:
            rightIndex = middle
    return array[leftIndex]
```

Problem 7. Given an array A of length n, and two indices i and j with $i \leq j$, define $\text{Prod}(A, i, j)$ as the product of elements from $A[i]$ to $A[j]$, i.e., $A[i] \times A[i+1] \times \cdots \times A[j]$. If $i = j$, then $\text{Prod}(i, j) = A[i]$.

For example, if $A = [3, 0.1, 5, 20, 4]$, then $\text{Prod}(A, 1, 3) = 0.1 \times 5 \times 20 = 10$.

Now, consider the following problem MaxProd(A):

- **Input**: An array A of length n where all elements are positive, but may include fractional values (e.g., $A[i] = 0.1$ is valid).
- **Output**: The maximum possible value of $\text{Prod}(A, i, j)$ for some subarray $A[i..j]$.

For example, if $A = [3, 0.2, 4, 0.5, 1, 4, 0.3, 2]$, the maximum product is 8, which is achieved by the subarray $[4, 0.5, 1, 4]$.

a) Write pseudocode for a recursive algorithm MaxProd(A) that solves the problem in $O(n \log n)$ time. You only need to provide the pseudocode and the recurrence relation for the algorithm. **(10 points)**

Hint: The recurrence formula is similar to one we've used before, so you don't need to prove that it solves to $T(n) = O(n \log n)$.

Note: While there are faster, non-recursive algorithms for this problem, you must use a recursive algorithm for this homework.

Solution. We can use a divide and conquer algorithm similar to how we did for problem 5. The way we would do this is by splitting the array into two subarrays (or halves) and then using recursion to find the maximum product of the right subarray (right half), left subarray (left half), and the middle/intersection. This problem is very similar to the algorithm (finding the max subarray sum) that we did in question 5. Let us look at the pseudocode for this algorithm below:

```
startIndex = 0
endIndex = array.length - 1

MaxProductLandR(array, startIndex, endIndex):
        if startIndex = endIndex:
            return array[startIndex]
            // this is the base case for recursion
        middle = (endIndex + startIndex) / 2
        // find the midpoint of array
        rightMaxProd = MaxProductLandR(array, middle + 1, end) //right half recursion
        leftMaxProd = MaxProductLandR(array, startIndex, mid) //left half recursion
        midIntersectMax = intersectMax(A, startIndex, endIndex, middle)
        return max(rightMaxProd, leftMaxProd, midIntersectMax) //gets the biggest product
       // subarray out of the right, middle, and left subarrays

intersectMax(A, start, mid, end):
    leftVal = 1
    leftMaxProd = MINIMUM_VALUE
    rightVal = 1
    rightMaxProd = MINIMUM_VALUE

    for i = mid to start at index 0:
            leftVal = leftVal * array[i]
            leftMaxProd = max(leftMaxProd, leftVal)

    for i = mid + 1 to end at array(array length - 1):
```

```
                rightVal = rightVal * array[i]
                rightMaxProd = max(rightMaxProd, rightVal)

        //product of left and right subarrays gets us product of middle subarray
        return rightMaxProd * leftMaxProd
```

Using the pseudocode, we can see that the recurrence relation is $2T(n/2) + O(n)$. This is because in every recursive step, we are splitting the array into left and right subarrays, giving us $n/2$. We multiply by 2 because we make two recursive calls for the left, and right. So we essentially have two subproblems of $n/2$ as we discussed in class. Then in our `intersectMid` function, we are traversing the left and right subarrays to get us the product of the middle subarray, which gets us an $O(n)$ time.

---

b) Now, write pseudocode for a recursive algorithm that solves the problem in $O(n)$ time. As in Part 1, provide the pseudocode and the recurrence relation for the algorithm. **(15 points)**

Hint: Similar to the Max Profit problem covered in class, solve a helper problem MaxProdX(i, j), which gives additional information beyond just the maximum product.

If you're confident with the $O(n)$ algorithm, you can directly write a single solution for both Part 1 and Part 2. However, if you're unsure, first implement the $O(n \log n)$ solution for Part 2, then write the separate $O(n)$ algorithm.

Solution. In part A, we implemented a divide and conquer approach to solve this question, but now we can use something similar to Kadane's Algorithm for a linear run time to solve this question. We are still implementing recursion, so it is imperative that we have a base case, which will be seeing if the array is empty. Let's look at the pseudocode below:

```
main:


maxProduct = array[0]
minProduct = array[0]
resultProduct = array[0]
i = 1  // start at the 1st index, because
we know the maxProduct of the first element will just be itself

maxProductSubArray(array, i, maxProduct, minProduct, resultProduct):
     if array.length = i:
            return resultProduct //this is the base case.

    //current value that we're looking at
     int current = array[i]

     maxProdNow = max(maxProduct * current, current, minProduct * current)
     minProdNow = min(maxProduct * current, current, minProduct * current)
     resultProduct = max(resultProduct, maxProdNow)

    return maxProductSubArray(array, i + 1, maxProdNow, minProdNow, resultProduct)
```

The recurrence relation for this is: $T(n) = T(n-1) + O(1)$. $O(1)$ represents the constant time for calculating `maxProdNow` and `minProdNow`. $n-1$ represents all the time for the recursive calls we must make for the $n-1$ elements of the array, since we know that the maxProduct of the first element will just be itself.

Problem 8. Suppose we have an array $A[1:n]$ which consists of numbers $\{1,\ldots,n\}$ written in some arbitrary order (this means that A is a *permutation* of the set $\{1,\ldots,n\}$). Our goal in this problem is to design a very fast randomized algorithm that can find an index i in this array such that $A[i] \mod 5 = 0$, i.e., $A[i]$ is divisible by five. For simplicity, in the following, we assume that n itself is a multiple of 5 and is at least 5 (so a correct answer always exist).

For instance, if $n = 10$ and the array is $A = [8,7,2,5,4,6,3,1,10,9]$, we can output either of indices 4 or 9.

(a) Suppose we sample an index i from $\{1,\ldots,n\}$ uniformly at random. What is the probability that i is a correct answer, i.e., $A[i] \mod 5 = 0$? **(5 points)**

Solution. Since we know that A is a permutation of the set $\{1,\ldots,n\}$ and n is a multiple of 5, we can claim that exactly one-fifth of the elements in A are divisible by 5.

The probability can be calculated by the following formula:

$$\text{Probability} = \frac{\text{Number of favorable outcomes}}{\text{Total number of possible outcomes}}$$

In this case, the number of favorable outcomes is $\frac{n}{5}$ since there are $\frac{n}{5}$ numbers in the set A that are divisible by 5. The total number of possible outcomes is n, which is the total number of elements in the set.

Thus, we end up with the following formula for probability:

$$\text{Probability} = \frac{\frac{n}{5}}{n} = \frac{1}{5}.$$

The probability that index i is a correct answer, where $A[i] \mod 5 = 0$, would be $\frac{1}{5}$.

(b) Suppose we sample m indices from $\{1,\ldots,n\}$ uniformly at random and with repetition. What is the probability that none of these indices is a correct answer? **(5 points)**

Solution. The probability that none of the indices is a correct answer can be found by using our answer to the previous part. We know that the probability that a given index is **correct** is $\frac{1}{5}$, hence the probability that a given index is **not correct** would be $1 - \frac{1}{5} = \frac{4}{5}$.

If we are sampling m number of indices uniformly at random with repetition, we can raise the probability of one index **not being correct** to m sample indices.

The probability that none of the m sampled indices is a correct answer would then be:

$$\left(\frac{4}{5}\right)^m$$

Now, consider the following simple algorithm for this problem:

Find-Index-1$(A[1:n])$**:**

- Let $i = 1$. While $A[i] \mod 5 \neq 0$, sample $i \in \{1,\ldots,n\}$ uniformly at random. Output i.

The proof of correctness of this algorithm is straightforward and we skip it in this question.

(c) What is the worst-case **expected** running time of **Find-Index-1**($A[1:n]$)? Remember to prove your answer formally. **(7 points)**

Solution. The worst-case **expected** running time of **Find-Index-1**($A[1:n]$) would be $O(1)$, or constant running time.

The algorithm works by continuously sampling an index (until an index that satisfies the condition is found) from the set at random and checking to see if $A[i] \mod 5 = 0$.

We can represent this as the random variable X, which is the number of iterations/checks you need to perform before you find an index that satisfies the condition $A[i] \mod 5 = 0$. X can be treated as a geometric random variable. From our previous parts, we know that the probability of getting a success in a trial (finding an index that is divisible by 5) is $\frac{1}{5}$.

We know that the number of expected iterations needed for a success in a geometric random variable is $\frac{1}{p}$, where p is the success probability. In our case, as we found previously, $p = \frac{1}{5}$.

Thus, we can claim that the expected number of iterations we would need is:

$$E[X] = \frac{1}{p} = \frac{1}{0.2} = 5.$$

Each iteration takes constant time, so the expected running time of the algorithm can be calculated as:

$$O(1) \times E[X] = O(1) \times 5 = O(5) = O(1).$$

The problem with **Find-Index-1** is that in the worst-case (and not in expectation), it may actually never terminate! For this reason, let us consider a simple modification to this algorithm as follows.

Find-Index-2($A[1:n]$)**:**

- For $j = 1$ to n:
  - Sample $i \in \{1, \ldots, n\}$ uniformly at random and if $A[i] \mod 5 = 0$, output i and terminate; otherwise, continue.
- If the for-loop never terminated, go over the array A one element at a time to find an index i with $A[i] \mod 5 = 0$ and output it as the answer.

Again, we skip the proof of correctness of this algorithm.

(d) What is the **worst-case running time** of **Find-Index-2**($A[1:n]$)? What about its worst-case **expected** running time? Remember to prove your answer formally. **(8 points)**

Solution. Worst-case running time:
For the worst-case running time, the for loop would run for all n number of iterations but would never end up reaching the correct index. If this happens and all iterations are completed, the algorithm would continue and go over the array once more, but this time, it would be sequential. We are guaranteed that at least one number in A is divisible by 5, so we can say that the sequential search *will* terminate after at most n steps.

In this case, the worst-case running time would be $O(n) + O(n)$, which simplifies to a running time of $O(n)$.

Worst-case expected running time:
In order to get the worst-case expected running time, we can start off by calculating the expected number of iterations that the for loop will go through before finding an index i where $A[i] \mod 5 = 0$.

As seen in previous parts, the probability of getting an index where $A[i] \mod 5 = 0$ is $\frac{1}{5}$ and the expected number of iterations it would take before finding such an index is 5.

However, this algorithm may not have to go through all the iterations and may terminate early. In this case, the worst case would be the expected number of checks that you would need to find the correct index. This would give the for-loop an expected running time of $O(5)$, which can be considered as constant running time.

In the case that the algorithm's for-loop fails to find the correct index in the first few iterations of the loop, the sequential scan will go through and find the correct index. This would take $O(n)$, however, the probability of this happening is extremely low. We can claim that the expected running time of the algorithm would be the expected running time of the for-loop, which, as previously mentioned, is constant at $O(5)$.

In conclusion:

- The worst-case running time of the algorithm would be $O(n)$.
- The worst-case expected running time would be $O(5)$, which simplifies to $O(1)$.

Problem 9. Given an array A of length n, an element x is *frequent* if it appears at least $n/4$ times. There can be between 0 and 4 frequent elements.

The array A consists of incomparable objects, meaning that you can only check equality (i.e., $A[i] == A[j]$), and there is no notion of ordering or sorting.

Part 1: Write pseudocode for the algorithm `IsFrequent(A, x)` that checks if x is frequent in A. The algorithm should run in $O(n)$ time. You may not use dictionaries or hash functions. **(10 points)**

Hint: This is a short, non-recursive solution.

Solution. The pseudocode for this problem is as follows:

```
Function ifFrequent(A, x):
count = 0
n = length(A)



for i = 0 to i = n-1:
    if A[i] == x:
        count++


boolean frequent = (count >= n/4)
return frequent
```

This algorithm takes linear time $O(n)$ to run because it is sequentially going through every element of the array (there are n elements in the array) and each operation takes constant time.

Part 2 : Write pseudocode for a recursive algorithm that finds all frequent elements in A, or returns "no solution" if none exist. The algorithm should run in $O(n \log n)$ time. Provide the recurrence formula for your algorithm. **(20 points)**

Hint: Reuse `IsFrequent` from Part 1.

Note: You may not use dictionaries, hash functions, or sorting algorithms for this problem. Recursive solutions are required for Part 2.

Solution. The recursive solution for the algorithm is as follows:

```
Function FindFrequentElements(A):
n = length(A)


# Base Case: return no solution if the array does not have enough elements


    if n < 4:
        return "no solution"


# Divide the array into a left half and a right half
middle = n // 2
leftSide = A[1:middle]
rightSide = A[middle+1:n]
```

# Initialize a results array to store the list of frequent elements
resultArr = []

# Use recursion to check the left and right sides
leftFrequentElements = `FindFrequentElements(leftSide)`
rightFrequentElements = `FindFrequentElements(rightSide)`

# Check if the elements from the left and right half can qualify as frequent for the entire array

```
    for every x in leftSide:
        if IsFrequent(A, x):
            resultArr.append(x)

    for every y in rightSide:
        if IsFrequent(A, y) AND y not in resultArr:
            resultArr.append(y)
```

# If after traversing the entire array, no frequent elements are found: return ”no solution”

```
    if resultArr.size == 0:
        return "no solution"
```

# Else, return the frequent elements found
```
return resultArr
```

Recurrence Formula

The first part of the algorithm involves dividing the array. We split the array into two halves, giving us two subproblems of size $\frac{n}{2}$.

The second part of the algorithm is checking the frequency. We recursively call the method `IsFrequent` to check if each element is frequent in the entire array, and we do this for every element returned by the recursive calls.

Thus, we can express the recurrence as:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Since this is a standard recurrence for a divide-and-conquer algorithm, we can use the Master Theorem to solve the recurrence. By applying the Master Theorem, we find that the running time is:

$$T(n) = O(n \log n)$$

Problem 10. You are given two strings A and B of length n, each consisting of possibly repeated characters from an alphabet of size M. Two strings A and B are said to be **anagrams** of one another if it is possible to reorder the characters in A to obtain string B (without removing or adding any characters). For instance, the strings "ELEVEN PLUS TWO" and "TWELVE PLUS ONE" are anagrams of one another (think of empty-space also as a character in the alphabet).

Your goal is to design an algorithm that decides if a given pair of strings are anagrams of one another. You may assume that all characters in the alphabet can represented by distinct integers in the set $\{1, 2, \ldots, M\}$, and that you can convert any character to its integer representation in $O(1)$ time (note that M can be much larger than n).

(a) Design an algorithm for this problem with worst-case runtime of $O(n + M)$. **(15 points)**

(b) Design a randomized algorithm for this problem with worst-case expected runtime of $O(n)$. **(10 points)**

Solution. For Part A: We can use a hashmap to keep track of the frequencies of our characters. The frequencies of the characters are correspondent to the number of times they occur in the string. The steps go as follows: 1) Iterate through string A, incrementing hash maps with (key,value) pair where key represents character and value represents the number of time it occurs. 2) Now, iterate through the second string a decrememnt the characters' frequency every time as they occur in the same hash map. 3) After iterating through the hashmap, see if the values for all the characters in the hashmap are equal to 0. If they are, then this means that A and B are anagrams of each other because the same letters occurred in the same frequencies so that when you add them up and subtract them, they cancel out to get to 0. If they aren't 0, then this means A and B are not anagrams of each other.

For Part B: 1) Make a hashmap with 27 keys including the space character and the 26 alphabetical letters. 2) Use the random math function to get a random value and store it for each character (or key) in the hashmap.. 3) Iterate through string A, and find the corresponding value for each character in the hashmap, and make a product counter to multiply these values together and store them. 4) Do the same as step 3 for B. 5) If the productCounter for A and B are equivalent, then it is indicate of being perfect anagrams. If not, then they are not perfect anagrams.

Problem 11. Consider the following more complex variant of the Tower of Hanoi puzzle The puzzle has a row of k pegs, numbered from 1 to k. In a single turn, you are allowed to move the smallest disk on peg i to either peg $i-1$ or peg $i+1$, for any index i; as usual, you are not allowed to place a bigger disk on a smaller disk. Your mission is to move a stack of n disks from peg 1 to peg k.

Describe a recursive algorithm for the case $k = n + 1$ that requires at most $O\left(n^2\right)$ moves. **(20 points)**

Solution. Because this is a recursive problem, we always must start with a base case. In this scenario, our base case would be when n=1. This means that when we are trying to move n (1) disk, we can move it from P(1) to P(n+1). Now, if we were to think of the recursive solution, then we have to think about the recursive step. In this case, for the recursive step, we need to realize that for every iteration, we are trying to move the highest n-1 disks into P(n+1) all the way from P(1).

This gets us the recurrence formula 2*T(n-1)+1, and this is because we have 2 subproblems and n-1 work being done for every recursive call. Then we have a constant time added when we are trying to merge.

Let us design the alogirthm: **Algorithm: BMOVE(n, start, end)**

1. **Base case:** If $n = 0$, return.
2. **If** $n = 1$, loop through the pegs until you reach return.
3. Make a recursive call: BMOVE$(n-1, \text{start}, \text{end})$
4. BMOVE$(n-1, \text{start}, \text{end-1})$ // Calls the recursive function to move $n-1$ disks.
5. Indicate that a move is being made from the start peg to the end peg.
6. BMOVE$(n-1, \text{end-1}, \text{end})$

Challenge Yourself. You are given an array A of non-negative integers. Every element except one has a duplicate. Design an algorithm that finds the element with no duplicate in $O(n)$ time (You are promised that there is exactly one element with no duplicate). **(25 points)**

Example. A couple of examples for this problem:

1. $A = [4, 3, 1, 1, 4]$
   Output: 3
   Explanation: 3 is the only number that does not have a duplicate
2. $A = [3, 1, 4, 5, 1, 4, 3]$
   Output: 5
   Explanation: 5 is the only number that does not have a duplicate

Solution. This problem can be solved in O(n) time by utilizing hash maps. Firstly, you would traverse through the array and store each element in a hash table. We would use the element as the key and the count of each element as the corresponding value in the hash table.

The second time around, you would traverse the array again, looking specifically for the element with the value (count) of 1 in the hash table.

This algorithm would give you a running time of O(n).

The pseudocode for the algorithm is as follows:

```
def FindUnique(A):
    # Step 1: Initialize an empty hash table
    freq = {}

    # Step 2: Traverse the array to hash map with the key as the element and the value as the count
    for num in A:
        if num in freq:
            freq[num] += 1    # Increment the count if element is already in the hash table
        else:
            freq[num] = 1     # Initialize the count to 1 if it's the first occurrence

    # Step 3: Traverse the array again to find the element with count 1
    for num in A:
        if freq[num] == 1:
            return num        # Return the element with frequency 1

    # We do not have to account for the fact that no such element is found because the problem
    clarifies that we will always have exactly one element that does not have a duplicate
    in the array.
```
