1. [5 marks] **Divide and Conquer/Binary Search**. In the **FixedPoint** problem, the input is an array A of n integers **sorted in descending** order and the valid solution is the index $i : 1 \leq i \leq n$ such that $A[i] = i$ (this value of index i is called *fixed point*). If such value of i does not exist the valid solution is -1. Note that integers in array can be negative. For example, if the input array A=[9,8,3,1,-2,-10] the output is 3, as A[3]=3. If the input array A=[9,8,4,1,-2,-10] the output is -1.

   Give an efficient algorithm to find fixed point in the array. Your algorithm must run in time $\Theta(\log n)$.

   ### Algorithm

```
FixedPoint(A, n):
    left = 1
    right = n
    result = -1

    while left <= right:
        mid = floor((left + right)/2)

        if A[mid] == mid:
            return mid
        else if A[mid] > mid
            left = mid + 1
            // since array is sorted in descending order, and A[mid] > mid,
            // all elements to the left will also be greater than there indices
        else:
            right = mid - 1
            // element to the right will be less than their indices
    return -1
```

   ### Correctness

   The algorithm is based on the following observations:

   - since the array is sorrted in **deccending order**, if `A[mid] > mid` then for all indices `j < mid`, we have `A[j] >= A[mid] > mid >= j`, so no fixed point exists to the left.
   - similarly, if `A[mid] < mid`, then for indices `j > mid`, we have `A[j] <= A[mid] < mid <= j`, so a fixed point to the right doesn't exist.

   This allows half of the search space in eahc iteration to be eleminated.

   ### Time Complexity

   The algorithm performs a binary serach, dividing the search space in half at each step. Therefore, the final time complexity is $\boldsymbol{\Theta}$**(log n)**.

2. [7 marks] **Divide and Conquer**.
   Suppose you have an unsorted array $A[1..n]$ of elements. You can only perform equality tests on the elements (e.g. they are large GIFs). In particular this means that you cannot sort the array. You want to find (if it exists) a *majority* element, i.e., an element that appears more than half the time. For example in $[a, b, a]$ the majority element is a, but $[a, b, c]$ and $[a, a, b, c]$ have no majority element. Straightforward approach of checking if $A[i]$ is a majority element for all $i = 1, \dots, n$ will have run-time in $\Theta(n^2)$ which is too slow. Consider the following approach to finding a majority element: recursively find the majority element y in the first half of the array and the majority element z in the second half of the array, and combine results of recursive calls into the answer to the problem.

   1. Prove that if x is a majority element in A then it has to be a majority element in the first half of the array or the second half of the array (or both).
   2. Using observation of part (a) give a divide-and-conquer algorithm to find a majority element, that runs in time $O(n \log n)$. **Detailed pseudocode is required**. Be sure to argue correctness and analyze the run time. If given array has no majority element, return FAIL.

Part (a): Proof

Claim: If `x` is a majority element in ‘A1, then it must be a majority element in the first half or the second half or even bpth halfs.

Proof by contradiction:

First let `n` be the size of array `A`. Suppose `x` is the majority in array `A`, which means `x` appears more than `n/2` times in `A`. Next assume for contradictions that `x` is not a majority element in either half.

- the size of the first half is `n/2`, in which `x` appears at most `(n/2)/2` times
- the size of the second half is `n/2`, in which `x` appears at most `(n/2)/2` times

For even `n`: each half has size `n/2`, so `x` appearsat most `n/4 + n/4 = n/2` amount of times total.

For odd `n = 2k + 1`: first half is size `k`, second half is size `k+1`. So `x` appears at most `(k/2) + ((k+1)/2) <= k/2 + k/2 + 1 = k + 1/2 <= (2k+1)/2 = n/2` amount of times.

In both cases, `x` appears at most `n/2` times, which contradicts the assumption made that `x` is a majority element that appears more than `n/2` times. Therefore, `x` must be a majority element in at least one of the 2 halfs.

Part (b): Algorithm

```
FindMajority(A, left , right):
    // base case
    if left == right:
        return A[left]

    // divide
    mid = floor((left + right)/2)
    // combine
    if y == z:
        return y

    // count the amount of occurances of y and z in A[left...right]
    count_y = 0
    count_z = 0
    for i = left to right:
        if A[i] == y:
            count_y = count_y + 1
        if A[i] == z:
            count_z = count_z + 1

    n = right - left + 1
    if count_y > n/2:
        return y
    else if count_z > n/2:
        return z
    else:
        return FAIL

Main Algorithm:
    result = FindMajority(A, 1, n):
        if result == FAIL:
            return FAIL

        // verify if the result is actually a majority element
        count = 0
        for i = 1 to n:
            if A[i] == result:
                count += 1

        if count > n/2:
            return result
        else:
            return FAIL
```

Correctness

The algorithm is correct because:

1. **Base case:** A single element is a trivial majority element of a size-1 array.
2. **Recursive case:** By part (a), if a majority element exists in `A[left...right]`, it must be a majority in the left half or the right half. So now the only candidates are `y` and `z`.
3. **Combine step:** Now we check if `y` or `z` is a majority in the full range. If neither is, then there is no majority element that exists in this range.
4. The final verification ensures that the returned element is a truly majority element in the entire array.

Time Complexity

The recurrance relation is:

- `T(n) = 2T(n/2) + O(n)` where the `O(n)` term comes from counting the occurrences in the combine step. By the master Theorem(case 2): `T(n) = O(n log n)`

---

3. [8 marks] **Greedy algorithm**.
Here is a special "matching" problem: given a set A of n numbers and a set B of n numbers, form pairs (a_1, b_1), ..., (a_n, b_n), with $\{a_1, \dots, a_n\} = A$ and $\{b_1, \dots, b_n\} = B$, so that the following cost function is minimized

$$\sum_{i=1}^{n} (a_i - b_i)^2 .$$

Consider the following "greedy" strategy to solve this problem: pick the pair (a, b) with the smallest difference $|a - b|$ ($a \in A, b \in B$). Then remove a from A and b from B, and repeat.

Give a counterexample showing that this strategy is incorrect, i.e., it can sometimes give a non-optimal solution. **Show your work!**

Give another greedy strategy that will solve this special matching problem. Provide pseudo-code of your algorithm that takes lists A and B and returns minimal possible value of the cost function. Prove that it always returns an optimal solution. Analyze running time.

Counterexample for the First Strategy

Counterexample:

- `A = {1, 2, 10}`
- `B = {3, 9, 11}`

Greedy:

1. Smallest: |10 - 11| = 1, pick (10, 11)
2. Remaining: A = {1, 2}, B = {3, 9}. Smallest: |2 - 3| = 1, pick (2, 3)
3. Pick (1, 9) Cost = (10 - 11)^2 + (2 - 3)^2 + (1 - 9)^2 = 1 + 1 + 64 = **66**

Optimal (sort and match): Array A sorted: {1, 2, 10}, array B sorted: {3, 9, 11} Pairs: (1, 3), (2, 9), (10, 11) Cost = (1 - 3)^2 + (2 - 9)^2 + (10 - 11)^2 = 4 + 49 + 1 = **54**

Since 66 > 54 the greedy algorithm is not optimal.

Correct Greedy Algorithm

Algorithm:

```
OptimalMatching(A, B, n):
    Sort A in acsending order
    Sort B in acsending order
    cost = 0

    for i = 1 to n:
        cost = cost + (A[i] + B[i])^2
    return cost
```

Proof of correctness:

Claim: Sorting both arrays to match the elements and pairing corresponding elements would causes the cost function to be minimized.

Proof using exchange arguments: Let A' and B' be the sorted versions of A and B sorted into acsending order. Now suppose there exists an optimal matching that differs from the original sorted matching. Because in any optimal match, there exists indices where j > i such that A'[i] is matched with B'[k] and A'[j] is matched with B'[m] where k > m.

Now Consider the following swap matches:

- Original cost contribution: (A'[i] - B'[k])^2 + (A'[j] - B'[m])^2
- After swap: (A'[i] - B'[m])^2 + (A'[i] - B'[k])^2
  Expanding Further:
- Original: A'[i]^2 - 2A'[i]B'[k] + B'[k]^2 + A'[j]^2 - 2A'[j]B'[m] + B'[m]^2
- After swap: A'[i]^2 - 2A'[i]B'[m] + B'[m]^2 + A'[j]^2 - 2A'[j]B'[k] + B'[k]^2
  Therefore since A'[i] <= A'[j] and B'[m] <= B'[k].

Difference (original - Swap):
-2A'[i]B'[k] - 2A'[j]B'[m] + 2A'[i]B'[m] + 2A'[j]B'[k] = 2(A'[j]B'[k] - A'[i]B'[k] - A'[j]B'[m] + A'[i]B'[m]) = 2(A'[j] - A'[i])(B'[k] - B'[m]) >= 0

Since both are non-negative factors, cost isn't increased by swaping. By repeatedly applying such swaps, we can transform any optimal solution into sorted matching without increasing cost. therefore, the sorted matching is optimal.

Running Time

- Sorting A: O(n log n)
- Sorting B: O(n log n)
- Computing cost: O(n)

Total: O(n log n)

---

4. [10 marks] **Greedy algorithm**.
Suppose we would like to buy n items from SuperCheapStore (SCS); all items are currently priced at 1 CAD. Unfortunately there is no delivery and we have to deliver them home. The bad news are:
– we can fit only 1 item in our truck;
– it takes one day to drive home and back.
There is even worse news – SCS charges us for the storage of undelivered items and the charge for storage of item i growth exponentially as original price times factor $c_i > 1$ each day. It means that if item i is picked up d days from now, the charge will be $1 \cdot c_i^d$ CAD. In which order should we pick up our items from SCS so that total amount of charges is as small as possible.

Develop a greedy algorithm to solve this problem assuming that $c_i \neq c_j$ for $i \neq j$. The input to your algorithm is $c_1, c_2, \ldots, c_n$, the output is permutation of indices $\{1, 2, \ldots, n\}$ that corresponds to the order of delivery giving minimal storage cost. Prove that your algorithm gives an optimal solution. What is the running time of your algorithm?

Greedy Algorithm

Strategy: Pick up the items in decreasing orderof their c_i growth factors. Which is, pick up the largest value c first, then second largest value c, and so on.

```
OptimalPickup(C[1...n]):
    // sort items by growth factor in descending order then
    // return the permutation(order of pickup).
    Sort items so that c[1] >= C[2] >= ... >= c[n]

    return [1, 2, 3, ..., n]
```

Proof of Correctness

First consider the function f(x) = x^d where d is the number of the current day and x > 1 is the growth factor.

Next for any 2 items with c_a < c_b and days i < j:

- if we pick a for day i and b for day j: cost = c_a^i + c_b^j
- if we pick b for day i and a for day j: cost = c_b^i + c_a^j

The second is the better option: c_b^i + c_a^j < c_a^i + c_b^j Equivalently: c_b^i - c_a^i < c_b^j - c_a^j

Since 1 < c_a < c_b and i < j, the expontential groth function grows faster for largers bases and larger exponents. The difference is c_b^i - c_a^i grows smaller than c_b^j - c_a^j because i < j.

More roughly: c_b^j - c_a^j = c_b^i * c_b^(j - i) - c_a^i * c_a^(j - i)

Since c_a < c_b and i < j, we have c_b^(j- i) > c_a^(j - i), and the gapincreases exponentially. Therefore puuting larger growth factors earlier minimizes the total cost.

Running Time

- Sorting: O(n log n)
- Outputting permutation: O(n)

Total: O(n log n)