Class 1

Detailed Notes

Problem1 : Maximum Absolute Difference

Link: Maximum Absolute Difference

Solution:

```
int Solution::maxArr(vector<int> &A) {

    vector<int> B, C;

    for (int i = 0; i < A.size(); i++) {

        B.push_back(A[i] + i);

        C.push_back(A[i] - i);

    }

    int Bmax = B[0], Bmin = B[0], Cmax = C[0], Cmin = C[0];

    for (int i = 1; i < B.size(); i++) {

        Bmax = max(Bmax, B[i]);

        Bmin = min(Bmin, B[i]);

    }

    for (int i = 1; i < C.size(); i++) {

        Cmax = max(Cmax, C[i]);

        Cmin = min(Cmin, C[i]);

    }

    return max(Bmax - Bmin, Cmax - Cmin);

}
```

Explanation:

The goal is to maximize the value of |A[i] - A[j]| + |i - j|. By examining the expression, it can be reformed into four potential expressions based on the absolute values:

- A[i] + i - (A[j] + j)
- A[i] - i - (A[j] - j)
- - (A[i] + i - A[j] - j)
- - (A[i] - i - A[j] + j)

From this, it becomes clear that finding the maximum and minimum values for A[i] + i and A[i] - i separately will cover all cases, reducing the problem to computing these maxima and minima.

Why it works:

This method works because it consolidates the complexity of comparing every possible pair by transforming the problem into a simpler form where you only need to find extremes (maximum and minimum values) and then compute their differences. This exploits the distributive properties of maxima and minima over addition and subtraction, substantially simplifying the computation.

Time Complexity: O(n), where n is the length of the array. This is because the array is traversed a constant number of times (specifically, four times).

Space Complexity: O(n), for storing the transformed arrays B and C.

Problem 2: Find a Repeating and a Missing Number

Link: Find a Repeating and a Missing Number

Solution:

```
class Solution{

public:

    vector<int> findTwoElement(vector<int> arr, int n) {

        int r = -1, m = -1;

        for (int i = 0; i < n; i++) {

            if (arr[abs(arr[i]) - 1] < 0) {

                r = abs(arr[i]);

            } else {

                arr[abs(arr[i]) - 1] *= -1;

            }

        }

        for (int i = 0; i < n; i++) {

            if (arr[i] > 0) {

                m = i + 1;

            }

        }

        return vector<int>({r, m});
    }

};
```

Explanation:

The solution uses the properties of array indices and marking to find the missing and repeating numbers. By traversing the array and flipping the sign of the elements at the indices derived from the array values, the algorithm marks those elements as seen. If a number is found that points to an already negative value, it indicates that the number is a repeat. In the second pass, any index that has a positive number indicates that the number corresponding to that index is missing.

Why it works:

The approach exploits the fixed range of numbers 1 to n to use the array itself for marking without additional space. This clever use of marking ensures that we can track seen numbers and detect anomalies in a single scan, while the second scan identifies any numbers that were never marked.

Time Complexity: O(n)

Space Complexity: O(1) — aside from the input and the result, no extra space is used.

Problem 3: Next Permutation

Link: [Next Permutation on LeetCode]()

Solution:

```
class Solution {
public:
    void nextPermutation(vector<int>& nums) {
        if (nums.size() == 1) {
            return;
        }
        int p1 = nums.size() - 2;
        while (p1 >= 0 && nums[p1] >= nums[p1 + 1]) {
            p1--;
        }
        if (p1 == -1) {
            reverse(nums.begin(), nums.end());
            return;
        }
        int index = p1 + 1;
        for (int i = p1 + 2; i < nums.size(); i++) {
            if (nums[i] > nums[p1] && nums[i] < nums[index]) {
                index = i;
            }
        }
        swap(nums[p1], nums[index]);
        sort(nums.begin() + p1 + 1, nums.end());
    }
};
```

Explanation:

This function modifies the given sequence to the next permutation in lexicographic order. The algorithm works in three main steps:

1. Find the first decreasing element from the right: This element (p1) is the pivot to initiate the next higher permutation.
2. Find the smallest element on the right of the pivot that is greater than the pivot: This element will be swapped with the pivot to ensure the next sequence is the smallest greater permutation.
3. Sort the remaining sequence after the pivot position: This ensures the sequence is the next permutation that is the smallest.

Why it works:

The algorithm finds the next lexicographical permutation directly by determining the smallest possible increment. This is done by swapping and then rearranging the smallest possible part of the sequence, resulting in the next permutation without generating all permutations.

Time Complexity: O(n) - Each step of the algorithm (finding p1, finding the element to swap, sorting the tail) is linear in terms of the length of the sequence.

Space Complexity: O(1)- The space used does not depend on the input size, as all operations are done in-place.

Problem 4: Adding One to Number Represented as Array of Digits

Link: [Adding One on InterviewBit]

Solution:

```
vector<int> Solution::plusOne(vector<int> &A) {
    while (A.size() > 0 && A[0] == 0) {
        A.erase(A.begin());
    }
    int carry = 1;
    for (int i = A.size() - 1; i >= 0; i--) {
        A[i] += carry;
        carry = A[i] / 10;
        A[i] %= 10;
    }
    if (carry == 1) {
        A.insert(A.begin(), 1);
    }
    return A;
}
```

Explanation:

The function increments a large number represented as a vector of its digits. It starts by removing any leading zeros, then adds one to the number from the least significant digit, handling carry as it propagates towards the most significant digit. If there is still a carry after the last digit, a new digit is added at the beginning.

Why it works:

The algorithm correctly handles carries, which is crucial for arithmetic operations on numbers represented as digit arrays. It incrementally updates the array from the least to the most significant digit, ensuring that all intermediate states are correct.

Time Complexity: O(n), where n is the number of digits. Most of the time, only one pass through the array is needed.
Space Complexity: O(1), aside from the space needed for input.

Problem 5: Max Sum Contiguous Subarray

Link: [Max Sum Contiguous Subarray on InterviewBit]()

Solution:

```
int Solution::maxSubArray(const vector<int> &A) {
    int sum = 0, max_sum = INT_MIN;
    for (int i = 0; i < A.size(); i++) {
        sum += A[i];
        if (sum > max_sum) {
            max_sum = sum;
        }
        if (sum < 0) {
            sum = 0;
        }
    }
    return max_sum;
}
```

Explanation:
This function calculates the maximum sum of any contiguous subarray using Kadane’s Algorithm. It maintains a running sum (sum) and a maximum sum (max_sum). As it iterates over the array, it adds each element to the running sum. If sum becomes negative, it resets sum to zero since a negative sum would not contribute to a maximum sum of any subsequent subarray.

Why it works:
Kadane's Algorithm efficiently finds the maximum sum subarray by using the fact that any subarray whose sum is negative can never contribute to the maximum sum of any subsequent subarray starting after it.

Time Complexity: O(n)- Single traversal of the array.

Space Complexity: O(1)- Uses only a few extra variables, independent of input size.