SEASONS OF CODE
IIT BOMBAY
SUMMARY

Velagala Deeraj Sathvik Reddy
23B0906, Computer Science

June 2024

Summary of Learnings

During the first four weeks of the Summer of Science , I have gained a solid foundation in various essential data structures and algorithms. My problem-solving abilities have significantly improved, and I have learned to approach problems from multiple perspectives. Below are the explanations and theoretical code snippets for each topic covered.

Basics of Data Structures and Algorithms

A data structure is a way to store data in the memory of a computer. It is important to choose an appropriate data structure for a problem, because each data structure has its own advantages and disadvantages.
A set is a data structure that maintains a collection of elements. The basic operations of sets are element insertion, search and removal.
A map is a generalized array that consists of key-value-pairs. While the keys in an ordinary array are always the consecutive integers 0,1,...,n1, where n is the size of the array, the keys in a map can be of any data type and they do not have to be consecutive values. **Example Code:**

```
#include<bits/stdc++.h>
using namespace std;

void solve(){
    set<int> s; //declaration
    multiset<int> ms;
    s.insert(2);
    s.insert(3);
    s.insert(4);
    s.insert(4);
    ms={1,2,3,4,4,5,5,5,6}; //elements can be added by 'braces' or using '
        insert'

    cout<<"mscount4"<<ms.count(4)<<'\n'; //counting '4' in ms
    cout<<"scount4"<<s.count(4)<<'\n';//count will be 1 as a set is defined
    to have distinct elements

    ms.erase(ms.find(4)); //removes only one 4 instance
    cout<<"mscount4"<<ms.count(4)<<'\n';


    ms.erase(4); //erased 4 from ms
    cout<<"mscount4"<<ms.count(4)<<'\n';

    //set iterators
    auto it1 = s.begin();
    cout << *it1 << "\n";
```

```
    it1=s.end();
    auto it2 = s.find(2);
    if (it2 == s.end()) {
        cout<<"2 is found"<<'\n';// x is not found
    }

    //For example, the following code finds the element nearest to x:
    auto it = s.lower_bound(3);
    if (it == s.begin()) {
        cout << *it << "\n";
    }
    else if (it == s.end()) {
        it--;
        cout << *it << "\n";
    }
    else {
        int a = *it; it--;
        int b = *it;
        if (3-b < a-3) cout << b << "\n";
    else cout << a << "\n";
    }

    map<string,int> mp; //map is created with key-value pairs
    mp["apple"]=1;
    mp["banana"]=2;

    cout<<mp["cat"]<<'\n'; //default value is '0'

    if(mp.count("apple")){
        cout<<"exists!"<<'\n';
    } //count gives bool whether key exists or not

    for (auto x : mp) {
        cout << x.first << " " << x.second << "\n"; //outputs all key value
            pairs
    }
}

int main(){
    solve();
}
```

Recursion, Sorting

Recursion involves solving a problem where the solution depends on solutions to smaller scales of the same problem. Sorting algorithms arrange data elements in a specified order, such as numerical or lexicographical order.

Recursion:

Recursion is a method of solving problems by calling a function in itself. It allows complex problems to be broken down into simpler, more manageable sub-problems. A classic example of a recursive function is calculating the factorial of a number.

```
#include <iostream>
#include <vector>
using namespace std;

// Example of recursive function (factorial)
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

int main() {
```

```cpp
    // Example of recursive function call
    int fact = factorial(5);
    cout << "Factorial of 5: " << fact << endl;
    return 0;
}
```

Sorting Techniques:
Sorting techniques are used to create an order in a collection of elements. Some common sorting algorithms include Bubble Sort, Insertion Sort, and Merge Sort.

Bubble Sort:

Simple algorithms for sorting an array work in $O(n^2)$ time. Such algorithms are short and usually consist of two nested loops. A famous $O(n^2)$ time sorting algorithm is bubble sort where the elements "bubble" in the array according to their values.

```cpp
#include <bits/stdc++.h>
using namespace std;

void solve(){
    int n;
    cin >> n;
    int arr[n];
    int swap_count = 0; // Initialize swap counter to 0

    // Input the elements of the array and print them
    for(int i = 0; i < n; i++){
        cin >> arr[i];
        cout << arr[i] << " ";
    }
    cout << '\n';

    // Bubble Sort Algorithm
    // Outer loop runs n times
    for(int i = 0; i < n; i++){
        // Inner loop for each pass through the array
        for(int j = 0; j < n - 1; j++){
            // If the current element is greater than the next element, swap
               them
            if(arr[j] > arr[j + 1]){
                swap(arr[j], arr[j + 1]);
                swap_count++; // Increase swap counter
            }
        }
    }

    // Print the sorted array
    for(int i = 0; i < n; i++){
        cout << arr[i] << " ";
    }
    // Print the total number of swaps made
    cout << '\n' << "swap_count is: " << swap_count << '\n';
    cout << '\n';
}

int main(){
    int t;
    cin >> t;
    while(t--){
        solve();
    }
}
```

Insertion Sort:

Insertion Sort is a simple sorting algorithm that builds with a time complexity of $O(n^2$ in a worst possible case . It is much less efficient on large lists than more advanced algorithms such as merge sort.

```cpp
#include <bits/stdc++.h>
using namespace std;

void solve(){
    int n;  // Variable to store the number of elements in the array
    cin >> n;  // Input the number of elements

    vector<int> v(n, 0);  // Declare a vector of size n initialized to 0

    // Input the elements of the vector
    for(int i = 0; i < n; i++){
        cin >> v[i];
    }

    // Modified Insertion Sort Algorithm
    for(int i = 0; i < n - 1; i++){
        // Check if current element is greater than or eq to next element
        if(v[i] >= v[i + 1]) {
            int j = i + 1;  // Start checking from the next element
            // Move the element to its correct position by swapping
            while(j > 0){
                if(v[j] <= v[j - 1])
                    swap(v[j], v[j - 1]);  // Swap the elements if they are out
                        of order
                else
                    break;  // Break the loop if the order is correct
                j--;
            }
        }
    }

    // Print the sorted vector
    for(int i = 0; i < n; i++){
        cout << v[i] << " ";
    }
    cout << '\n';  // Print a newline character
}

int main(){
    int t;  // Variable to store the number of test cases
    cin >> t;  // Input the number of test cases
    while(t--);  // Loop through all test cases (Note: this loop is currently
        doing nothing)
}
```

Merge Sort:

It is possible to sort an array efficiently in O(nlogn) time using algorithms that are not limited to swapping consecutive elements. One such algorithm is merge sort, which is based on recursion. Merge sort sorts a subarray `array[a ...b]` as follows:

1. If $a = b$, do not do anything, because the subarray is already sorted.
2. Calculate the position of the middle element: $k = \left\lfloor \frac{a+b}{2} \right\rfloor$.
3. Recursively sort the subarray `array[a ...k]`.
4. Recursively sort the subarray `array[k + 1 ...b]`.

5. Merge the sorted subarrays `array[a ...k]` and `array[k + 1 ...b]` into a sorted subarray `array[a ...b]`.

```cpp
#include <bits/stdc++.h>
using namespace std;

// Function to merge two halves of the array
void merge(vector<int>& v, int l, int r) {
    if (l == r) return;  // Base case is if there is only one element, no need to sort

    int mid = (l + r) / 2;  // Find the mid-point of the current segment

    // Recursively sort left and right halves
    merge(v, l, mid);
    merge(v, mid + 1, r);

    // Temporary vector to store the merged result
    vector<int> v1 = v;

    int i = l, j = mid + 1, k = l;  // Pointers for the two halves and the merged array
    while (i <= mid && j <= r) {
        if (v1[i] <= v1[j])
            v[k++] = v1[i++];  // Copy the smaller element from the left half
        else
            v[k++] = v1[j++];  // Copy the smaller element from the right half
    }

    // Copy any remaining elements from the right half
    while (j <= r)
        v[k++] = v1[j++];

    // Copy any remaining elements from the left half
    while (i <= mid)
        v[k++] = v1[i++];
}

int main() {
    int t;  // Number of test cases
    cin >> t;  // Input the number of test cases
    while (t--) {
        int n;  // Number of elements in the array
        cin >> n;  // Input the number of elements
        int l = 0;  // Left index of the array
        int r = n - 1;  // Right index of the array
        vector<int> v(n, 0);  // Initialize a vector of size n with all elements 0
        for (int i = 0; i < n; i++) {
            cin >> v[i];  // Input the elements of the array
        }
        merge(v, l, r);  // Sort the array using merge sort
        for (int i = 0; i < n; i++) {
            cout << v[i] << " ";  // Output the sorted array
        }
        cout << '\n';  // Print a newline after each test case
    }
}
```

Binary Search and Bit Manipulation

A general method for searching for an element in an array is to use a `for` loop that iterates through the elements of the array. For example, the following code searches for an element `x` in an array:

```
for (int i = 0; i < n; i++) {
    if (array[i] == x) {
        // x found at index i
    }
}
```

The time complexity of this approach is $O(n)$, because in the worst case, it is necessary to check all elements of the array. If the order of the elements is arbitrary, this is also the best possible approach, because there is no additional information available where in the array we should search for the element x. However, if the array is sorted, the situation is different. In this case it is possible to perform the search much faster, because the order of the elements in the array guides the search. The following binary search algorithm efficiently searches for an element in a sorted array in $O(\log n)$ time.

Binary Search implementation using two pointers:

Binary search works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

```
#include <bits/stdc++.h>
using namespace std;

void solve() {
    int n;
    cin >> n;  // Input the size of the array
    int arr[n];
    for (int i = 0; i < n; i++) {
        cin >> arr[i];  // Input the elements of the array
    }
    int x;
    cin >> x;  // Input the element to search for

    int a = 0, b = n - 1;  // Initialize pointers for the beginning and end of
       the array
    while (a <= b) {  // Loop until the search space is exhausted
        int mid = (a + b) / 2;  // Calculate the midpoint
        if (arr[mid] == x) {  // Check if the midpoint element is the target
            cout << "x found at index " << mid << '\n';  // Output the index if
                found
            return;  // Exit the function
        }
        if (arr[mid] >= x) {
            b = mid - 1;  // Adjust the search space to the left half
        } else {
            a = mid + 1;  // Adjust the search space to the right half
        }
    }
    // If the element is not found, output the elements at indices a and b (if
       they exist)
    if (a < n) cout << "Element at a: " << arr[a] << " ";
    if (b >= 0) cout << "Element at b: " << arr[b] << " ";
    cout << "x: " << x << '\n';  // Output the searched element
}

int main() {
    solve();  // Call the solve function
}
```

Binary Search implementation using for loop:

An alternative method to implement binary search is based on an efficient way to iterate through the elements of the array. The idea is to make jumps and slow the speed when we get closer to the target element.

The search goes through the array from left to right, and the initial jump length is n/2. At each step, the jump length will be halved: first n/4, then n/8,n/16, etc., until finally the length is 1. After the jumps, either the target element has been found or we know that it does not appear in the array.

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;

void solve() {
    int n;
    cin >> n;  // Input the size of the array
    int arr[n];
    for (int i = 0; i < n; i++) {
        cin >> arr[i];  // Input the elements of the array
    }
    int x;
    cin >> x;  // Input the element to search for

    int k = 0;  // Initialize k to 0, which will be used to find the position
        of x
    // b starts as half of the array size and reduces by half each iteration
    for (int b = n / 2; b >= 1; b /= 2) {
        while (k + b < n && arr[k + b] <= x) k += b;  // Adjust k by adding b
            if within bounds and arr[k + b] <= x
    }
    // Check if the element at index k is the target element
    if (arr[k] == x) {
        cout << "x found at index " << k << '\n';  // Output the index where x
            is found
    } else {
        cout << "x not found\n";  // Output if x is not found
    }
}

int main() {
    solve();  // Call the solve function
}
```

Sliding Window, Two-Pointer, and Greedy Algorithms

Sliding window and two-pointer techniques are used to solve problems involving subarrays or subsequences.
A greedy algorithm constructs a solution to the problem by always making a choice that looks the best at the moment. A greedy algorithm never takes back its choices, but directly constructs the final solution. For this reason, greedy algorithms are usually very efficient.

Example Code:

Sliding window is a common technique for problems involving subarrays. Greedy algorithms make the most optimal choice at each step, aiming for a globally optimal solution.

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Example of sliding window technique (Maximum sum subarray of size k)
int maxSumSubarray(vector<int>& arr, int k) {
    int n = arr.size();
    int Sum = 0;
    for (int i = 0; i < k; ++i)
        Sum += arr[i];
    int maxSum = Sum;
    for (int i = k; i < n; ++i) {
        Sum += arr[i] - arr[i - k];
        maxSum = max(maxSum, Sum);
    }
    return maxSum;
}

// Example of greedy algorithm (Minimum coins to make change)
int minCoins(vector<int>& coins, int amount) {
```

```
    sort(coins.begin(), coins.end(), greater<int>());
    int count = 0;
    for (int coin : coins) {
        count += amount / coin;
        amount %= coin;
    }
    return count;
}

int main() {
    // Example of sliding window technique call
    vector<int> nums = {4, 2, 1, 7, 8, 1, 2, 8, 1, 0};
    int k = 3;
    int maxSum = maxSumSubarray(nums, k);
    cout << "Maximum sum of subarray of size " << k << ": " << maxSum << endl;

    // Example of greedy algorithm call
    vector<int> coins = {1, 2, 5, 10};
    int amount = 18;
    int minCoinCount = minCoins(coins, amount);
    cout << "Minimum coins to make change for " << amount << ": " <<
       minCoinCount << endl;

    return 0;
}
```

Greedy Algorithms - Application Example

The following code is an application of Kadane's Algorithm, which is a classic example of a greedy approach used to find the maximum sum subarray in a given array.

```
// Including the necessary libraries
#include<bits/stdc++.h>
using namespace std;
typedef long long int lli;

void solve(){
    int n;
    cin >> n; // Reading the size of the array
    lli a[n];
    lli sum;   // This will store the maximum subarray sum found so far
    lli sum1 = 0; // This will store the current subarray sum
    int head = 0; // This will track the starting index of the current subarray

    // Reading the array elements
    for(int i = 0; i < n; i++){
        cin >> a[i];

        // Initialize sum with the first element
        if(i == 0) sum = a[i];

        // Update the current subarray sum
        sum1 = sum1 + a[i];

        // If the current subarray sum becomes negative and we're not at the
           last element,
        // reset the current subarray starting point to the next element and
           reset sum1
        if(sum1 <= 0 && i < n-1) {
            head = i + 1;
            sum1 = 0;
        }
```

```
        // If the current subarray sum is greater than the maximum sum found so
            far,
        // update the maximum sum
        else if(sum1 > sum) sum = sum1;
    }

    // Print the maximum subarray sum found
    cout << sum;
}

int main(){
    solve();
}
```

Explanation and Greedy Algorithm Application

Kadane's Algorithm is a prime example of a greedy approach. The algorithm iteratively processes each element in the array and makes a local decision that eventually leads to a globally optimal solution.

- **Initialization**:
  - `sum` is initialized to the first element of the array to keep track of the maximum subarray sum found so far.
  - `sum1` is initialized to 0 and will accumulate the sum of the current subarray.
  - `head` is used to mark the start of the current subarray.
- **Iteration**:
  - For each element in the array, `sum1` is updated by adding the current element (`a[i]`).
  - If `sum1` becomes negative (and we are not at the last element), it means including the current subarray would not help in finding the maximum sum, so `sum1` is reset to 0, and the starting point of the subarray (`head`) is updated to the next element.
  - If `sum1` is greater than the `sum`, it means the current subarray has the largest sum found so far, so `sum` is updated to `sum1`.
- **Result**:
  - The value in `sum` at the end of the iteration is the maximum subarray sum.

Greedy Choice: The greedy choice here is that at each step, the algorithm decides whether to continue with the current subarray or start a new subarray based on the sum becoming negative. This local decision ensures that the algorithm efficiently finds the maximum subarray sum.

By using Kadane's Algorithm, this solution efficiently finds the maximum sum subarray in linear time (`O(n)`), which is optimal for this problem.

Binary Search Application - Example

The following code implements a binary search algorithm to guess a number within a given range using the `guessNumber` function.
Leetcode- Guess number problem

```
// Including necessary libraries
#include<bits/stdc++.h>
using namespace std;

/**
 * Forward declaration of guess API.
 * @param  num   your guess
 * @return       -1 if num is higher than the picked number
```

```
 *                  1 if num is lower than the picked number
 *                  otherwise return 0
 */
int guess(int num);

class Solution {
public:
    // Function to guess the number
    int guessNumber(int n) {
        int x = 0; // Starting guess
        for (int i = n; i > 0; i /= 2) {
            // Check if current guess is correct
            if (guess(x + i) == 0)
                return x + i;
            // Adjust guess based on guess result
            while (guess(x + i) != -1)
                x += i;
        }
        return x;
    }
};

int main() {
    Solution sol;
    int n = 100; // Example range limit
    int guessed_number = sol.guessNumber(n);
    cout << "Guessed Number: " << guessed_number << endl;
    return 0;
}
```

Explanation:

The provided code uses the application of binary search in the context of guessing a number. Here's an explantion of how it works:

- **Binary Search Approach**:
  - The function `guessNumber` uses binary search to guess the correct number within the range from 1 to `n`.
  - It initializes `x` to 0 as the starting guess.
  - For each iteration, it sets `i` to half of the current range (`n`), aiming to narrow down the search space.
  - It calls the `guess` function with `x + i`:
    - If `guess(x + i)` returns 0, it means `x + i` is the correct number, and the function returns it.
    - If `guess(x + i)` returns -1, it adjusts `x` downwards by adding `i` (moving to the lower half of the search space).
  - The loop continues until the correct number is guessed, and the function returns the guessed number.
- **Complexity**:
  - The algorithm runs in `O(log n)` time complexity, as it halves the search space in each iteration.

This example demonstrates how binary search can efficiently find the target number within a sorted range, leveraging the property of the search space halving at each step.

Two-Pointers Algorithm Application - Example

The following code implements the `isSubsequence` function, which checks if string `s` is a subsequence of string `t`. It utilizes the two-pointers technique to efficiently match characters in both strings.

```cpp
// Including necessary libraries
#include<bits/stdc++.h>
using namespace std;

class Solution {
public:
    // Function to check if s is a subsequence of t
    bool isSubsequence(string s, string t) {
        int i = 0, j = 0;
        int m = s.size(), n = t.size();

        // Traverse both strings with two pointers
        while (i < m && j < n) {
            if (s[i] == t[j]) {
                cout << "matched with " << s[i] << " at " << j << '\n';
                i++; // Move pointer in s to next character
            }
            j++; // Always move pointer in t to next character
        }

        // If all characters of s are matched, s is a subsequence of t
        return i == m;
    }
};

int main() {
    Solution sol;
    string s = "abc";
    string t = "ahbgdc";
    bool isSubseq = sol.isSubsequence(s, t);
    cout << "Is s a subsequence of t? " << (isSubseq ? "Yes" : "No") << endl;
    return 0;
}
```

Explanation and Two-Pointers Algorithm Application

The provided code demonstrates the application of the two-pointers algorithm to check if string `s` is a subsequence of string `t`. Here's a breakdown of how it works:

- **Two-Pointers Approach**:
  - Initialize two pointers `i` and `j` to traverse strings `s` and `t`, respectively.
  - Iterate through both strings:
    - If characters `s[i]` and `t[j]` match, increment `i` (move to the next character in `s`).
    - Always increment `j` (move to the next character in `t`).
  - If `i` reaches the end of `s` (`i == m`), then all characters of `s` have been found in order in `t`, confirming `s` as a subsequence of `t`.
  - Return `true` if `s` is a subsequence of `t`, otherwise return `false`.

This example illustrates how the two-pointers technique efficiently checks for subsequences by maintaining two pointers that traverse through the strings while ensuring that characters match in the correct order.