End-Semester Project

operation on numbers

Basic operations on numbers

- ***isEven(int n):***

- Purpose: To check whether a number is even

- Input: An integer `n`

- Return value: Boolean (true for even, false for odd)

- Procedure:

- **Use modulo operator to check if number is divisible by 2**
- **Return true if remainder is 0, false otherwise**

- Example:`isEven(4) → true  isEven(7)` → `false`

- **longFactorial(int n):**

- Purpose: To calculate factorial of a number

- Input: non-negative integer `n`

- Output: integer factorial of `n`

- Procedure:

- **Initialize result variable `f` to 1**
- **Use for loop to multiply `f` by each number from 1 to `n`**
- **Return final factorial value**

-Example:

`longFactorial(5)` returns `120` (1 * 2 * 3 * 4 * 5)

`longFactorial(0)` returns `1

• *gcd(int n, int m):*

- Purpose: To calculate GCD using Euclidean algorithm

- Input: Two integers n and m

- Output: Integer representing the GCD

- Procedure:

  • Determine larger and smaller number

  • Perform iterative division to find remainder

  • Update numbers in iteration repeatedly until remainder is 0

- Return final divisor

- Example : gcd(48, 18` returns 6

• *sumOfDigits(int n):*

- Purpose: Find sum of digits of a number

- Input: Integer `n`

- Output: Integer sum of all digits

- Procedure:

- Initialize sum to 0
- Use while loop to extract each digit
- Use modulo to get rightmost digit
- Divide number by 10 to remove rightmost digit
- Add each digit to sum

-Example :sumOfDigits(9999) returns 36 .

• *isPalindrome(int n):*

- Purpose: Determine if number is palindrome

- Input: Integer `n`

- Return value: Boolean on whether it is a palindrome

- Procedure:

- Hold original number

- Reverse number digit by digit
- Compare reversed number with the original
- Example Input/Output:

isPalindrome(121)` returns `true`

- ***isPrime(int n):***

-Purpose: To check if number is prime

- Input: Integer n

- Output: Boolean to indicate whether it is a prime number or not

- Procedure:

• Find square root of the number

• Check divisibility up to the sqrt of number

- ***reverseNumber(int n):***

- Objective: Reverse the digits of a number

- Input: Integer `n`

- Output: Digits of the integer reversed

- Steps:

- Initialize reversed number to 0
- Extract rightmost digit
- Build reversed number by multiplying previous result by 10
- Remove rightmost digit from original number

- Example:

reverseNumber(100) returns 1

- ***numberPrime(int a):***

- Purpose: Check whether a number is prime

- Input: Integer `a`

- Output: Boolean on whether number is prime

-Procedure:

- Calculate square root of number
- Check divisibility up to square root

- Example: Completely needs rewrite.

Intermediate operations on numbers

- ### isPerfect(int num):

- **Purpose: To check if a given number is a perfect number.**
- **Input: The integer num.**
- **Return value: true if the number is a perfect number (i.e., the sum of its divisors equals the number); otherwise, false.**
- **Procedure:**
  - **Initialize sum to 0.**
  - **Loop through all integers from 1 to num / 2 to check if they are divisors.**
  - **If a divisor is found, add it to sum.**
  - **At the end, compare sum with num. If they are equal, return true; otherwise, return false.**
- **Example:**
  - **isPerfect(6) returns true because the divisors of 6 are 1, 2, and 3, and their sum is 6.**

- ### primeFactors(int num):

- **Purpose: To print the prime factors of a given number.**
- **Input: The integer num.**
- **Return value: None. The prime factors are printed.**
- **Procedure:**
  - **Start with the smallest prime factor, 2.**
  - **If num is divisible by i, print i and divide num by i until num becomes 1.**
  - **If num is not divisible by i, increment i and continue checking.**
- **Example:**
  - **primeFactors(24) prints 2 2 2 3, which are the prime factors of 24.**

- ### isAutomorphic(int num):

- **Purpose: To check if a given number is an automorphic number.**
- **Input: The integer num.**
- **Return value: true if the number is automorphic (i.e., its square ends in the same digits as the number); otherwise, false.**
- **Procedure:**
  - **Compute the square of num.**
  - **Compare the last digits of the square with the digits of num.**
  - **If they match, return true; otherwise, return false.**
- **Example:**
  - **isAutomorphic(25) returns true because $25^2 = 625$, which ends in 25.**

- ### sumDivisors(int num):

- **Purpose: To calculate the sum of divisors of a given number.**
- **Input: The integer num.**
- **Return value: The sum of divisors of num.**
- **Procedure:**
  - **Loop through all integers from 1 to num - 1.**
  - **If an integer divides num, add it to the sum.**
  - **Return the total sum of divisors.**
- **Example:**
  - **sumDivisors(6) returns 6 because the divisors of 6 are 1, 2, and 3, and their sum is 6.**

- ## isArmstrong(int num):

- **Purpose: To check if a given number is an Armstrong number.**
- **Input: The integer num.**
- **Return value: true if the number is an Armstrong number (i.e., the sum of its digits each raised to the power of the number of digits equals the number); otherwise, false.**
- **Procedure:**
  - **Find the number of digits in num.**
  - **For each digit, raise it to the power of the number of digits and sum the results.**
  - **If the sum equals the original number, return true; otherwise, return false.**
- **Example:**
  - **isArmstrong(153) returns true because $1^3 + 5^3 + 3^3 = 153$.**

- ## isMagic(int num):

- **Purpose: To check if a given number is a magic number.**
- **Input: The integer num.**
- **Return value: true if the number is a magic number (i.e., the sum of its digits eventually reduces to 1); otherwise, false.**
- **Procedure:**
  - **Repeatedly sum the digits of the number until a single-digit result is obtained.**
  - **If the result is 1, return true; otherwise, return false.**
- **Example:**
  - **isMagic(19) returns true because 1 + 9 = 10, and 1 + 0 = 1.**

- ## fibonacciSeries(int n):

- **Purpose: To print the Fibonacci series up to the nth term.**
- **Input: The integer n.**
- **Return value: None. The Fibonacci series up to the nth term is printed.**
- **Procedure:**
  - **Start with the first two terms, 0 and 1.**
  - **For each subsequent term, add the two previous terms.**
  - **Print each term as it is generated.**
- **Example:**
  - **fibonacciSeries(5) prints 0 1 1 2 3.**

Advanced operations on numbers

- ## *isSmith(int n):*

- Objective: Check if number is a Smith number

- Input: integer n

- Output: string declaring Smith number

- Steps:

- Compute sum of digits of the number
- Calculate sum of the digits of all the prime factors
- Compare results

- Example: Checks if the sum of the digits is equal to the sum of prime factor digits

• *pascalTriangle(int n):*

- Objective:Prints Pascal's Triangle

- Input: The number of rows `n`

- Output: The printed triangle

- Notes:

- Nested loops
- Binomial coefficients
- Prints with spacing

- Example: Prints triangle with `n` rows

• *catalanNumber(int n):*

- Objective: Calculate nth Catalan number

- Input: Integer `n`

- Output: Catalan number

- Steps:

- Use iterative calculation
- Multiply and divide to compute

- Example:

`catalanNumber(4)` computes 4th Catalan number

• *combination(int n, int k):*

- Purpose: Compute combinations (nCk)

- Input:

`n`: Total items

`k`: Items to choose

- Output: Combinations count

- Procedure:

- Handle edge cases
- Optimize by choosing smaller subset
- Iterative calculation

- Example:

`combination(5,2)` calculates ways to choose 2 from 5

• *bellNumber(int n):*

- Objective: To calculate nth Bell number

- Input: Integer `n`

- Output: Bell number

- Steps:

- Dynamic programming approach (Dynamic programming is a computer programming technique where an algorithmic problem is first broken down into sub-problems, the results are saved, and then the sub-problems are optimized to find the overall solution — which usually has to do with finding the maximum and minimum range of the algorithmic query).
- Combination used
- Bell numbers are generated in an iterative manner

- Example:

Compute total number of ways to partition `n` elements

• *isHarshednumber(int a):*

- Purpose: To check a given number is a Harshad number

- Input: The integer number

- Return value:** `true`, if the number is Harshad; otherwise `false`

- Procedure:

- Make input number is positive
- Check if original number is Harshed number i.e. divisible by sum of its digit.

- Example:

isHarshednumber(18)` returns `true` since 18 ÷ (1+8) = 2

• *primeFactors(int a):*

- Objective: To calculate the sum of the digits of prime factors

- Input: Integer `a`

- Output: Sum of digits of prime factors

- Steps:

- Handle even factors first
- Iterate through odd factors
- Add sum of digits for each prime factor

- Example:

For `primeFactors(24)`, would sum digits of 2, 2, 2, 3

• *binaryConversion(int dec):*

- **Objective: To convert a decimal number to its binary equivalent.**
- **Input: Integer dec.**
- **Output: Binary equivalent of the input decimal number.**
- **Steps:**
  1. **Initialize bin to 0 and i to 0 for storing the binary result and power of 10, respectively.**
  2. **Perform modulo operation to get the remainder when divided by 2 (binary digit).**
  3. **Multiply the remainder by the power of 10 and add to bin.**
  4. **Divide the number by 2 to continue the process.**
  5. **Increment the power (i) for the next binary digit.**
  6. **Repeat until the number becomes 0.**
- **Example:**
  For binaryConversion(10), the output would be 1010.

• *isNarcissistic(int num):*

- **Objective: To check whether a number is a narcissistic number (Armstrong number).**
- **Input: Integer num.**
- **Output: Boolean (true if the number is narcissistic, otherwise false).**
- **Steps:**
  1. **Determine the number of digits (power) in the number.**
  2. **For each digit, raise it to the power of the total digits and add to sum.**
  3. **Compare the sum with the original number.**
  4. **Return true if they are equal; otherwise, return false.**
- **Example:**
  For isNarcissistic(153), the output would be true ($1^3 + 5^3 + 3^3 = 153$).

- ## *sqrtApprox(int num):*

- **Objective: To approximate the square root of a number using the Newton-Raphson method.**
- **Input: Integer num.**
- **Output: Double value approximating the square root of the number.**
- **Steps:**
  1. **Initialize x0 with the input number and a small threshold (0.00001).**
  2. **Use the formula x=x0+numx02x = \frac{x_0 + \frac{\text{num}}{x_0}}{2} iteratively to refine the approximation.**
  3. **Stop the loop when the difference between two successive approximations is smaller than the threshold.**
  4. **Return the approximate square root.**
- **Example:**
  For sqrtApprox(16), the output would be 4.0.

- ## *power(int base, int exp):*

- **Objective: To calculate the result of raising a base to an exponent.**
- **Input: Integer base and exp.**
- **Output: Result of baseexp\text{base}^\text{exp}.**
- **Steps:**
  1. **Initialize result to 1.**
  2. **Multiply result by the base in a loop running exp times.**
  3. **Return the final result.**
- **Example:**
  For power(2, 3), the output would be 8.

- ## *isHappy(int num):*

- **Objective: To check whether a number is a "happy number".**
- **Input: Integer num.**
- **Output: Boolean (true if the number is happy, otherwise false).**
- **Steps:**
  1. **Continuously replace the number with the sum of the squares of its digits.**
  2. **Stop when the number becomes 1 (happy) or less than 7 (unhappy).**
  3. **Return true if the number is 1; otherwise, return false.**
- **Example:**
  For isHappy(19), the output would be true ($1^2 + 9^2 = 82 \rightarrow 8^2 + 2^2 = 68 \rightarrow \ldots \rightarrow 1$).

- ## *isAbundant(int num):*

- **Objective: To check whether a number is an "abundant number".**
- **Input: Integer num.**
- **Output: Boolean (true if the number is abundant, otherwise false).**
- **Steps:**
  1. **Calculate the sum of all proper divisors (excluding the number itself).**
  2. **If the sum is greater than the number, return true; otherwise, return false.**
- **Example:**
  For isAbundant(12), the output would be true (divisors: 1, 2, 3, 4, 6 → sum = 16 > 12).

- ## *isDeficient(int num):*

- **Objective: To check whether a number is a "deficient number".**
- **Input: Integer num.**
- **Output: Boolean (true if the number is deficient, otherwise false).**
- **Steps:**
  1. **Calculate the sum of all proper divisors (excluding the number itself).**
  2. **If the sum is less than the number, return true; otherwise, return false.**
- **Example:**
  For isDeficient(8), the output would be true (divisors: 1, 2, 4 → sum = 7 < 8).

- ***fibonacciSeries(int n):***

- **Objective: To print the first n terms of the Fibonacci series.**
- **Input: Integer n.**
- **Output: Prints the Fibonacci series up to the nth term.**
- **Steps:**
  1. **Initialize the first two terms (0 and 1).**
  2. **Print the first two terms.**
  3. **Use a loop to calculate subsequent terms as the sum of the previous two.**
  4. **Print each term until the nth term is reached.**
- **Example:**
  For fibonacciSeries(5), the output would be 0 1 1 2 3.

Operations on arrays

Basic operations on Arrays

- ***initializeArray(int array1[], int size):***

- Purpose: Take array elements as input from the user

- Input: Array and its size

- Output: Void (it populates the array)

- Procedure:

- **Initialize loop counter i**
- **Start loop from 0 to size-1**
- **Print prompt message for user input**
- **Use scanf() to read integer value**
- **Store input value in corresponding array index**
- **Repeat for all array positions**

- Example: Fills array with user integers

- ***printArray(int array[], int size):***

- Objective: Display array elements

- Input: Array and its size

- Output: Prints array elements

- Steps:

- **Initialize loop counter `i`**
- **Print header message**
- **Start loop from 0 to `size-1`**
- **Print each array element**

- Example: Prints out the array's contents

- ***findMax(int arr[], int size):***

- Purpose: Find the maximum value

-Input: An array and its size

- Output: Maximum value

- Procedure:

- **Declare the variable `max` equal to the first element in array**
- **Declare the loop counter variable `i`**
- **Start loop from second element (index 1)**
- **Compare each subsequent element with current `max`**
- **if element larger than `max`, update `max`**
- **Continue until end of array**
- **Return final `max` value**

- Example: Returns largest array element

- ***findMin(int arr[], int size) :***

- Problem: Minimum element

- Input: Array and its size

- Output: Minimum value

- Steps :

- **Initialize the value of `min` with first element of array**
- **Initialize the loop counter variable `i`**

- Enter the loop at the second element (index 1)
- For every other element in array, compare with current `min`
- If the element smaller than `min`, update `min`
- Do until end of array
- Return final `min` value

- Example: Returns smallest array element

- ***sumArray(int arr[], int size):***

- Objective: Calculate array sum

- Input: Array and its size

- Output: Total sum of elements

- Detailed Steps:

- Initialize `sum` variable to zero
- Initialize loop counter `i`
- Start loop from 0 to `size-1`
- Add each array element to `sum`
- Repeat for all elements in the array
- Compute and return `sum` total

- Example: Calculate sum of array elements

- ***averrageArray(int arr[], int size):***

- Purpose: Find the average of the array

- Input: Array and its size

- Output: Average of elements

- Procedure:

- Declare and initialize `sum` variable to zero
- Initialize loop counter `i`
- Loop starts from 0 to `size-1`
- Add each array element to `sum`
- Calculate average by dividing `sum` by `size`
- Return average value

- Example: Calculates mean of array elements

- ***isStored(int array[], int size):***

- Purpose: To check whether array is sorted in ascending order

- Input: Array and its size

- Output: Boolean on sorted status

- Procedure:

- Initialize loop counter `i`
- Iterate from first to second-to-last element
- Compare current element with next element
- If current element > next element, return false
- If loop completes finding unsorted pair, return true

- Example: Checks whether array is ascending

Intermediate Array Functions

- *reverseArray(int array[], int size):*

- Purpose: To print array in reverse order

- Input: Array and its size

- Output: Prints reversed array

- Steps:

Print header message

Initialize loop counter `i`

Start loop from last index to first index

Print each element in reverse order

- Example: Prints array elements backward

- *countEvenOdd(int array[], int* counteven, int* countodd, int size):*

- Purpose: To Count Even and Odd Numbers in Array

- Input: Array, pointers for count and array size

- Output: Prints count of even and odd numbers

- Steps:

- Initialize even and odd counters to zero
- Initialize Loop counter `i`
- Use loop to traverse through an entire array
- Check each element for divisibility by 2
- Increment even or odd counter
- Print the count of even and odd numbers

- Example: Counting of even and odd elements

• *secondLargest(int arr[], int size):*

- Objective: Finding of second largest element

- Input: Array and its size

- Output: Second largest element

- Steps:

- Compare first two elements
- Set initial max and second largest
- Iterate from third element
- Update max and second largest if necessary
- Handle the case for all unique and repeated elements
- Return second largest value

- Example: Finds second highest number

• *elementFrequency(int array[], int size):*

- Purpose: To count the frequency of elements (0-9)

- Input: Array and its size

- Output: Prints frequency of each number

- Steps:

- Loop through numbers 0-9
- Frequency counter initialization
- Check each number within array
- Count occurrences
- If number exists print frequency

- Example: Displays frequency of occurrence of each digit

• *removeDuplicates(int array[], int size):*

- Objective: Removing duplicate elements of the array

- Input: Array and its size

- Output: Prints an array without duplicate

- Step Description:

  • Make use of nested loops for comparisons between all elements

  • At the occurrence of a duplicate, shift the next elements

  • Decrease the size of the array

  • Adjusting Loop counters

  • Prints the modified array

- Example: Removes all repeated elements

• *binarySearch(int arr[], int size, int target):*

- Purpose: Search sorted array using binary search

- Input: Sorted array, size, target value

- Output: Index of target or -1 if not found

- Detailed Steps:

- Set initial start and end indices
- Calculate midpoint
- Compare midpoint value with target
- Adjust search range based on comparison
- Repeat until target found or search space exhausted

- Example: Efficiently finds element in sorted array

• *linearSearch(int arr[], int size, int target):*

- Objective: Search array sequentially

- Input: Array, size, target value

- Output: Index of target or -1 if not found

- **Steps:**

Loop through the elements of the array

At each element, compare with the target

If the target is found, return index

Repeat until the end of the array is reached

- **Example: Checks each element for target**

- ***leftShift(int arr[], int size, int rotations):***

- **Purpose: Left shift the elements in an array**

- **Input: Array, size, rotation count**

- **Output: The content of the shifted array**

- **Steps:**

- **Rotate for specified count**
- **Hold the last element**
- **Right shift all elements**
- **Move stored element to front**
- **Print array modified**

- **Example: Circularly shifts elements of the array**

Storing Arrays Functions

- ***bubbleSort(int arr[], int size):***

- **Purpose: To sort array using bubble sort algorithm**

- **Input: Array in unsorted manner and size**

- **Output: Array in sorted manner and total swaps**

- **Steps:**

- **Iterate through array multiple passes**
- **Compare adjacent elements**
- **Swap if out of order**

- Track swaps and sorting status
- Print sorted array and swap count

- Example: Sorts array by repeatedly swapping adjacent elements

• *selectionSort(int arr[], int size):*

- Purpose: To sort an array using selection sort

- Input: Unsorted array and size

- Output: Sorted array

- Procedure:

- Minimum element in unsorted portion
- Swap minimum with first unsorted element
- Repeat for subsequent array sections
- Print final sorted array

- Example: Finds and places smallest elements sequentially

• *insertionSort(int arr[], int size):*

- Purpose: To sort array using insertion sort

- Input: Unsorted array and size

- Output: Sorted array

- Steps:

- Start from second element
- Comparison with previously placed elements
- Element insertion at the correct position
- Shift of other elements
- Printing of sorted array

-Example: It builds the sorted array one element at a time

• *merge(int arr[], int start, int mid, int end):*

Purpose: Merging of two sorted subarrays

- Input: Array, subarray boundaries

- Output: Merged sorted subarray

- Steps:

- Create temporary subarrays
- Copy elements to temp arrays
- Merge elements in sorted order

- Handle remaining elements

- Example: Merges two sorted regions of array

• *mergeSort1(int arr[], int lb, int ub):*

- Purpose: To perform a recursive merge sort

- Input: An array and an upper and lower bounds

- Output: The sorted array

- Procedure:

- Divide the array into two halves
- Recursively sort the subarrays
- Merge the sorted subarrays

- Example: This implements divide-and-conquer sorting

• *mergeSort2(int arr[], int lb, int ub):*

- Purpose: To print the merge-sorted array

- Input: The array and bounds

- Output: Printing of the sorted array

-Steps:

- Call the merge sort function
- Print the sorted array that results

- Example: prints out array after sorting via merge sort

• *partition(int arr[], int lb, int ub):*

- Purpose: Partition array for quick sort

- Input: Array, lower and upper bounds

- Output: Partition index

- Steps:

- Choose the first element as the pivot
- Rearrange the array around the pivot

- Swap the pivot to the correct position

- Example: Prepares array for quick sort

• *quickSort(int arr[], int lb, int ub):*

- Purpose: Implementation of Quick Sort using recursion

- Input: Array, lower and upper bounds

- Output: Array after sorting

- Procedure:

- Partitioning the array
- Recursive sorting of sub-arrays
- Employ Divide and Conquer strategy

- Example: It will sort an array efficiently using the partitioning concept.

• *finalSort(int arr[], int size):*

- Purpose: Quick sort and print result

- Input: Array and size

- Output: Sorted array

- Steps:

- Set array bounds
- Call quick sort
- Print sorted array

- Example: Quick sort of entire array

Advanced Array Functions

• *findmissingnumber(int arr[], int size):*

- Purpose:To find missing number in the sequence

- Input: Array and size

- Output: Missing Number

- Steps:

Calculating expected sum of the sequence

Calculate actual sum of the array

Subtract

- Example: Finds the missing number in the given continuous sequence

• *findPairs(int arr[], int size, int sum):*

- Purpose: Find Pairs that Equal Target Sum

- Input: Array, size, target sum

- Output: Pairs Printed; No Pairs Found

- Steps:

- Check for pairs using nested loops
- Printing pairs with target sum
- Handling case when no pairs will have target sum

- Example: To find all pairs of numbers whose sum is equal to the target sum

• *findsubarraywithsum(int arr[], int size, int sum):*

- Purpose: Finding a subarray whose sum is equal to the target sum

- Input: Array, size, target sum

- Output: Subarray/no subarray

- Steps:

- Add to current sum
- Adjust window boundaries
- Print matching subarray or failure message

- Example: Finds contiguous section of array that matches the sum

• *rearrangeAlternatePositiveNegative(int arr[], int size):*

- Purpose: To restructure array with alternating signs

- Input: Array and size

- Output: Array with rearranged elements

- Steps:

- Separate positive and negative elements
- Interleave positive and negative numbers
- Remaining elements arrangement
- Print rearranged array

- Example : Re-arrange array in such way that all positive and negative numbers are alternate

• *findmajorityelement(int arr[], int size):*

- Purpose: To find majority element that occurs more than n/2 times.

-Input : Array and size

- Output: Majority element or -1

- Steps:

- Count occurrences of each element
- Check if count exceeds half array size
- Return first majority element found
- Return -1 if no majority element

- Example: Finds most frequent element

• *longestIncreasingSubsequence(int arr[], int size) :*

- Purpose: to find the length of the longest increasing subsequence

- Input:An array and size

- Output: Length of the longest increasing subsequence

- Steps:

- Temporary array initialization
- Compute increasing subsequence lengths
- Find maximum length
- Return the longest subsequence length

- Example: Prints longest increasing sequence

- ***findDuplicates(int arr[], int size):***

- Purpose: Find out count of all duplicate elements

- Input: Array and size

- Output: Prints all information about duplicates

- Steps:

- **Use the visited array to keep track of which element is scanned or not**
- **For each unique element, count its occurrence**
- **Print duplicate details**
- **Handle case of no duplicates**

- Example: Reports repeated elements

- ***findUnion(int arr1[], int size1, int arr2[], int size2):***

- Objective: To create union of two arrays

- Input: Two arrays and their sizes

- Output: Printed union of arrays

- Steps:

- **Merges first array**
- **Adds unique elements from second array**
- **Prints out resulting union array**

- Example: Merges arrays without duplicates

- ***findIntersection(int arr1[], int size1, int arr2[], int size2):***

- Objective: Find common elements between arrays

- Input: Two arrays and their sizes

- Output: Printed intersection

- Steps:

- **Compare elements of first array**
- **Check to the second array**
- **Print common elements**

- Example: Finds common elements in both arrays

- ***THE MEMBERS OF THE GROUP:***
- ***Gahlouz Tinhinen A3.***
- ***Mekci Raihan B4.***

- ***THE NAME OF SUPERVISING TEACHER:***
- **Berghouth.**