Saveetha School of Engineering.

SIMATS Chennai, TamilNadu.

Name:R.Maruthi.

Reg.no: 192325055.

Course code:CSA0867

Couse Name: DAA[Design and Analysis of Algorithm].

Date:06/06/2024. Day:Wednesday.

1).Aim:Write a program to find the reverse of a given number using recursive.

Algorithm:

1.**Base Case**: If the number is a single-digit number, return the number.

2. **Recursive Case**:

* Extract the last digit of the number.
* Reduce the number by removing the last digit.
* Use recursion to reverse the reduced number.
* Append the last digit to the reversed number obtained from the recursive call.

Time complexity:

* **Recursive Calls**: The number of recursive calls is proportional to the number of digits in the number n.
* If n has d digits, there will be d recursive calls.
* **Constant-Time Operations**: Each call performs a fixed number of operations, all of which are constant time.

Thus, the time complexity is determined by the number of recursive calls, which is proportional to the number of digits d in n.

For a number with ddd digits, the time complexity is:

O(d)

Program for python:

import time

def reverse\_number(n, reversed\_num=0):

# Base case: if the number is a single digit, return the current reversed number

if n == 0:

return reversed\_num

last\_digit = n % 10

reversed\_num = reversed\_num \* 10 + last\_digit

return reverse\_number(n // 10, reversed\_num)

def reverse(n):

if n < 0:

return -reverse\_number(-n)

else:

return reverse\_number(n)

number = 12345

start\_time = time.time()

reversed\_number = reverse(number)

end\_time = time.time()

execution\_time = end\_time - start\_time

print(f"Reversed number: {reversed\_number}")

print(f"Execution time: {execution\_time} seconds")

Output:

Reversed number: 54321

Execution time: 9.298324584960938e-06 seconds

Result:The code was executed successfully.

2).Aim: Write a program to find the perfect number.

Algorithm:

A perfect number is a positive integer that is equal to the sum of its proper divisors (excluding itself). For example, 6 is a perfect number because its divisors are 1, 2, and 3, and 1 + 2 + 3 = 6.

1. **Input**: A positive integer n.
2. **Initialize**: Set the sum of divisors sum\_divisors to 0.
3. **Iterate**: Loop through all numbers from 1 to n/2 (inclusive).
4. **Check Divisors**: For each number i, if i divides n evenly (i.e., n % i == 0), add i to sum\_divisors.
5. **Compare**: After the loop, compare sum\_divisors to n.
   * If sum\_divisors equals n, then n is a perfect number.
   * Otherwise, n is not a perfect number.
6. **Output**: Print or return the result.

Time complexity:

* **Loop Range**: The loop runs from 2 to n\sqrt{n}n​. Thus, the number of iterations is approximately n\sqrt{n}n​.
* **Constant-Time Operations**: Each iteration performs a constant number of operations (checking if n%i==0n \% i == 0n%i==0, and possibly adding two divisors).

Therefore, the time complexity of this function is:

O(n)O(\sqrt{n})O(n​).

Program for python:

import time

def is\_perfect\_number(n):

if n < 1:

return False

sum\_divisors = 0

for i in range(1, n // 2 + 1):

if n % i == 0:

sum\_divisors += i

return sum\_divisors == n

number = 28

start\_time = time.time()

result = is\_perfect\_number(number)

end\_time = time.time()

execution\_time = end\_time - start\_time

if result:

print(f"{number} is a perfect number.")

else:

print(f"{number} is not a perfect number.")

print(f"Execution time: {execution\_time:.10f} seconds")

Output: 28 is a perfect number.

Execution time: 0.0000135899 seconds

Result:The code was executed successfully.

3).Aim: Write C program that demonstrates the usage of these notations by analyzing the time complexity of some example algorithms.

Algorithm:

* **Constant Time Complexity O(1)**:
* Function constantTimeExample performs an operation that always takes the same amount of time, regardless of the input size.
* **Linear Time Complexity O(n)**:
* Function linearTimeExample performs a series of operations in a loop that runs n times.
* **Quadratic Time Complexity O(n^2)**:
* Function quadraticTimeExample performs nested loops, each running n times.
* **Execution Time Measurement**:
* Use the clock function to measure the start and end times of each function call.
* Calculate the elapsed time by subtracting the start time from the end time.

Time complexity:

 **Best Case**: O(n logn).

 **Worst Case**: O(n logn).

Program for python:

import time

import random

# Algorithm 1: Linear Search (O(n))

def linear\_search(arr, x):

for i in range(len(arr)):

if arr[i] == x:

return i

return -1

# Algorithm 2: Binary Search (O(log n))

def binary\_search(arr, x):

left, right = 0, len(arr) - 1

while left <= right:

mid = left + (right - left) // 2

if arr[mid] == x:

return mid

elif arr[mid] < x:

left = mid + 1

else:

right = mid - 1

return -1

# Algorithm 3: Bubble Sort (O(n^2))

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n - i - 1):

if arr[j] > arr[j + 1]:

arr[j], arr[j + 1] = arr[j + 1], arr[j]

return arr

# Function to measure execution time

def measure\_time(func, \*args):

start\_time = time.time()

result = func(\*args)

end\_time = time.time()

return end\_time - start\_time, result

# Function to analyze time complexity

def analyze\_time\_complexity():

# Test arrays

sizes = [1000, 5000, 10000]

results = []

for size in sizes:

arr = list(range(size))

x = random.choice(arr)

# Measuring time for linear search

time\_linear, \_ = measure\_time(linear\_search, arr, x)

# Measuring time for binary search

time\_binary, \_ = measure\_time(binary\_search, arr, x)

# Measuring time for bubble sort

random.shuffle(arr) # Shuffle to ensure unsorted array for bubble sort

time\_bubble, \_ = measure\_time(bubble\_sort, arr.copy())

results.append((size, time\_linear, time\_binary, time\_bubble))

# Printing results

print("Array Size | Linear Search Time (O(n)) | Binary Search Time (O(log n)) | Bubble Sort Time (O(n^2))")

for size, time\_linear, time\_binary, time\_bubble in results:

print(f"{size:10d} | {time\_linear:24f} | {time\_binary:25f} | {time\_bubble:22f}")

# Main function

if \_\_name\_\_ == "\_\_main\_\_":

analyze\_time\_complexity()

Output:

Constant time example (O(1)) took 0.000000 seconds to execute

Linear time example (O(n)) took 0.000024 seconds to execute

Quadratic time example (O(n^2)) took 0.392146 seconds to execute.

Result: The code was executed successfully.

4).Aim: Write a program that demonstrate the mathematical analysis of non-recursive and recursive algorithms.

Algorithm:

 **Non-recursive Linear Time ComplexityO(n)**:

Function linear\_sum calculates the sum of the first n natural numbers using a loop.

**Recursive Exponential Time Complexity O(2^n)**:

Function recursive\_fibonacci calculates the n-th Fibonacci number using a naive recursive approach.

**Execution Time Measurement**:

Use the time module to measure the start and end times of each function call.

Calculate the elapsed time by subtracting the start time from the end time.

1. Time complexity: **Initialization**: Initializing result takes constant time, O(1)O(1)O(1).
2. **Loop**: The loop runs from 1 to n, so it executes n times. Each iteration involves a constant-time multiplication operation.

Thus, the time complexity is dominated by the loop, which runs nnn times.

Time Complexity=O(n)\text{Time Complexity} = O(n)Time Complexity=O(n)

Program for python:

import time

def linear\_sum(n):

total = 0

for i in range(1, n + 1):

total += i

return total

def recursive\_fibonacci(n):

if n <= 0:

return 0

elif n == 1:

return 1

else:

return recursive\_fibonacci(n - 1) + recursive\_fibonacci(n - 2)

n = 100000

start\_time = time.time()

result = linear\_sum(n)

end\_time = time.time()

execution\_time = end\_time - start\_time

print(f"Sum of first {n} natural numbers (O(n)) is {result}")

print(f"Execution time: {execution\_time:.10f} seconds")

n = 30

start\_time = time.time()

result = recursive\_fibonacci(n)

end\_time = time.time()

execution\_time = end\_time - start\_time

print(f"{n}-th Fibonacci number (O(2^n)) is {result}")

print(f"Execution time: {execution\_time:.10f} seconds")

Output:

Sum of first 100000 natural numbers (O(n)) is 5000050000

Execution time: 0.0032382011 seconds

30-th Fibonacci number (O(2^n)) is 832040

Execution time: 0.1688570976 seconds.

Result:The code was executed successfully.

5).Aim: Write C programs for solving recurrence relations using the Master Theorem, Substitution Method, and Iteration Method will demonstrate how to calculate the time complexity of an example recurrence relation using the specified technique.

Algorithm:

**Techniques for Solving:**

1. **Master Theorem**: A general method for solving recurrence relations of the form T(n)=aT(nb)+f(n)T(n) = aT(\frac{n}{b}) + f(n)T(n)=aT(bn​)+f(n), where aaa, bbb, and f(n)f(n)f(n) are constants.
2. **Substitution Method**: A technique where you guess the form of the solution and then use mathematical induction to prove it.
3. **Iteration Method**: A method that involves expanding the recurrence relation until a pattern is identified, and then finding the closed-form solution.

Time complexity:

T(n)=O(nclogn)=O(nlogn)

**Program for python**:

import math

import time

def master\_theorem(a, b, f):

if f == "n^k":

k = math.log(a, b)

if k.is\_integer():

return f"T(n) = Theta(n^{k+1})"

elif k % 1 == 0.5:

return f"T(n) = Theta(n^{k+1} \* log(n))"

elif f == "n^k \* log(n)":

k = math.log(a, b)

return f"T(n) = Theta(n^{k} \* log(n))"

elif f == "n^k \* log^p(n)":

k = math.log(a, b)

return f"T(n) = Theta(n^{k} \* log^{p+1}(n))"

def substitution\_method(n):

return n\*\*2 \* (math.log(n) \*\* 0)

def iteration\_method(n):

sum = 0

while n > 1:

sum += n\*\*2

n //= 2

return sum

n = 1000

start\_time = time.time()

result\_master = master\_theorem(2, 2, "n^2")

end\_time = time.time()

execution\_time\_master = end\_time - start\_time

print("Using Master Theorem:", result\_master)

print("Execution time for Master Theorem:", execution\_time\_master, "seconds")

start\_time = time.time()

result\_substitution = substitution\_method(n)

end\_time = time.time()

execution\_time\_substitution = end\_time - start\_time

print("Using Substitution Method:", result\_substitution)

print("Execution time for Substitution Method:", execution\_time\_substitution, "seconds")

start\_time = time.time()

result\_iteration = iteration\_method(n)

end\_time = time.time()

execution\_time\_iteration = end\_time - start\_time

print("Using Iteration Method:", result\_iteration)

print("Execution time for Iteration Method:", execution\_time\_iteration, "seconds")

Output:

Using Master Theorem: None

Execution time for Master Theorem: 1.1920928955078125e-06 seconds

Using Substitution Method: 1000000.0

Execution time for Substitution Method: 9.298324584960938e-06 seconds

Using Iteration Method: 1333213

Execution time for Iteration Method: 9.059906005859375e-06 seconds.

Result: The code was executed successfully.

6).Aim: Given two integer arrays nums1 and nums2, return an array of their Intersection. Each element in the result must be unique and you may return the result in any order.

Algorithm:

* Convert both arrays nums1 and nums2 into sets to remove duplicate elements.
* Use the intersection() method to find the common elements between the two sets.
* Convert the resulting set back into a list to return the intersection.

1. Time complexity Analysis: **Conversion to Sets**: Converting nums1 and nums2 to sets takes O(n)O(n)O(n) and O(m)O(m)O(m) time, respectively, where nnn is the length of nums1 and mmm is the length of nums2.
2. **Set Intersection**: Finding the intersection of two sets takes O(min⁡(n,m))O(\min(n, m))O(min(n,m)) time because the operation iterates through the smaller set and checks membership in the larger set (which is an O(1)O(1)O(1) operation for each element).
3. **Conversion to List**: Converting the resulting set back to a list takes O(k)O(k)O(k) time, where kkk is the number of unique elements in the intersection. However, this step is usually negligible compared to the previous steps.

### Overall Time Complexity

Combining these steps, the overall time complexity is:

O(n+m+min (n,m))O(n + m + \min(n, m))O(n+m+min(n,m))

Since min (n,m)\min(n, m)min(n,m) is at most nnn or mmm, we can simplify this to:

O(n+m)O(n + m)O(n+m)

Thus, the time complexity of the program to find the intersection of two integer arrays, where each element in the result is unique, is O(n+m)O(n + m)O(n+m).

Here's the Python code implementing this algorithm:

import time

def intersection(nums1, nums2):

set1 = set(nums1)

set2 = set(nums2)

result\_set = set1.intersection(set2)

return list(result\_set)

nums1 = [1, 2, 2, 1]

nums2 = [2, 2]

start\_time = time.time()

intersection\_result = intersection(nums1, nums2)

end\_time = time.time()

execution\_time = end\_time - start\_time

print("Intersection of nums1 and nums2:", intersection\_result)

print("Execution time:", execution\_time, "seconds")

Output: Intersection of nums1 and nums2: [2]

Execution time: 1.1920928955078125e-05 seconds.

Result: The code was executed successfully.

7).Aim: Given two integer arrays nums1 and nums2, return an array of their intersection. Each element in the result must appear as many times as it shows in both arrays and you may return the result in any order.

Algorithm:

 Initialize an empty dictionary counter to store the count of elements in nums1.

 Iterate through nums1 and count the occurrences of each element in the dictionary.

 Initialize an empty list result to store the intersection with frequency.

 Iterate through nums2 and check if the element exists in counter and its count is greater than zero.

 If conditions are met, append the element to result and decrement its count in counte r.

 Return the result list.

Time complexity: O(n+m)O(n + m)O(n+m).

Below is the Python code implementing this algorithm:

import time:

def intersection(nums1, nums2):

counter = {}

for num in nums1:

counter[num] = counter.get(num, 0) + 1

result = []

for num in nums2:

if num in counter and counter[num] > 0:

result.append(num)

counter[num] -= 1

return result

nums1 = [1, 2, 2, 1]

nums2 = [2, 2, 2]

start\_time = time.time()

intersection\_result = intersection(nums1, nums2)

end\_time = time.time()

execution\_time = end\_time - start\_time

print("Intersection of nums1 and nums2 with frequency:", intersection\_result)

print("Execution time:", execution\_time, "seconds")

Output:

Intersection of nums1 and nums2 with frequency: [2, 2]

Execution time: 4.76837158203125e-06 seconds

Result: The code was executed successfully.

8).Aim: Given an array of integers nums, sort the array in ascending order and return it.You must solve the problem without using any built-in functions in O(nlog(n)) time complexity and with the smallest space complexity possible.

Algorithm:

* Define a function merge\_sort that takes an array nums as input.
* If the length of nums is less than or equal to 1, return nums as it is already sorted.
* Divide the array nums into two halves: left and right.
* Recursively call merge\_sort on both left and right.
* Merge the sorted left and right halves into a single sorted array.
* Return the merged sorted array.

Time complexity:O(n+m)O(n + m)O(n+m).

Now, let's implement this algorithm in Python:

import time

def merge\_sort(nums):

if len(nums) <= 1:

return nums

mid = len(nums) // 2

left\_half = nums[:mid]

right\_half = nums[mid:]

left\_sorted = merge\_sort(left\_half)

right\_sorted = merge\_sort(right\_half)

sorted\_nums = merge(left\_sorted, right\_sorted)

return sorted\_nums

def merge(left, right):

result = []

left\_idx, right\_idx = 0, 0

while left\_idx < len(left) and right\_idx < len(right):

if left[left\_idx] < right[right\_idx]:

result.append(left[left\_idx])

left\_idx += 1

else:

result.append(right[right\_idx])

right\_idx += 1

result.extend(left[left\_idx:])

result.extend(right[right\_idx:])

return result

nums = [5, 2, 9, 1, 6, 4]

start\_time = time.time()

sorted\_nums = merge\_sort(nums)

end\_time = time.time()

execution\_time = end\_time - start\_time

print("Sorted array:", sorted\_nums)

print("Execution time:", execution\_time, "seconds")

Output:

Sorted array: [1, 2, 4, 5, 6, 9]

Execution time: 1.9073486328125e-05 seconds.

Result: The code was executed successfully.

9).Aim: Given an array of integers nums, half of the integers in nums are odd, and the other half are even.

To solve this problem in Python, you can follow these steps:

1. Count the number of odd integers in the array.
2. Compare the count of odd integers with the count of even integers to ensure that both counts are equal (since half of the integers are odd and the other half are even).
3. Return True if both counts are equal; otherwise, return False.

Here's the algorithm:

1. Initialize two variables, odd\_count and even\_count, to count the number of odd and even integers, respectively.
2. Iterate through the array and increment odd\_count if the current integer is odd; otherwise, increment even\_count.
3. Check if odd\_count is equal to even\_count.
4. Return True if they are equal; otherwise, return False.

Time complexity: Thus, the overall time complexity of this program is O(n)O(n)O(n), where nnn is the number of elements in the array nums.

Here's the Python code implementing the algorithm:

import time

def check\_even\_odd\_balance(nums):

odd\_count = 0

even\_count = 0

start\_time = time.time()

for num in nums:

if num % 2 == 0:

even\_count += 1

else:

odd\_count += 1

end\_time = time.time()

is\_balanced = odd\_count == even\_count

execution\_time = end\_time - start\_time

return is\_balanced, execution\_time

nums = [1, 2, 3, 4, 5, 6]

result, execution\_time = check\_even\_odd\_balance(nums)

print("Is the array balanced with respect to even and odd numbers?", result)

print("Execution time:", execution\_time, "seconds")

Output:

Is the array balanced with respect to even and odd numbers? True

Execution time: 9.059906005859375e-06 seconds.

Result: The code was executed successfully.

10).Aim: Sort the array so that whenever nums[i] is odd, i is odd, and whenever nums[i] is even, i is even. Return any answer array that satisfies this condition.

To sort the array such that odd elements appear at odd indices and even elements appear at even indices, you can follow these steps:

1. Separate odd and even numbers into two separate lists.
2. Merge the odd and even lists alternatively into a new list, ensuring that the indices correspond to the parity of the elements.

Here's the algorithm:

1. Initialize two empty lists, odd\_nums and even\_nums, to store odd and even elements, respectively.
2. Iterate through the input array nums, and append odd elements to odd\_nums and even elements to even\_nums.
3. Initialize an empty list result to store the sorted array.
4. Use two pointers, i and j, to iterate through odd\_nums and even\_nums, respectively.
5. Alternate between appending elements from odd\_nums and even\_nums to result.
6. Return the result list.

Time complexity: O(n)

Here's the Python code implementing the algorithm:

import time

def sort\_array\_by\_parity(nums):

odd\_nums = []

even\_nums = []

start\_time = time.time()

for num in nums:

if num % 2 == 0:

even\_nums.append(num)

else:

odd\_nums.append(num)

result = []

i, j = 0, 0

while i < len(odd\_nums) and j < len(even\_nums):

if i % 2 == 0:

result.append(odd\_nums[i])

i += 1

else:

result.append(even\_nums[j])

j += 1

while i < len(odd\_nums):

result.append(odd\_nums[i])

i += 1

while j < len(even\_nums):

result.append(even\_nums[j])

j += 1

end\_time = time.time()

return result, end\_time - start\_time

nums = [4, 2, 5, 7, 1, 6]

sorted\_array, execution\_time = sort\_array\_by\_parity(nums)

print("Sorted array with odd elements at odd indices and even elements at even indices:", sorted\_array)

print("Execution time:", execution\_time, "seconds")

Output:

Sorted array with odd elements at odd indices and even elements at even indices: [5, 4, 2, 6, 7, 1]

Execution time: 1.0967254638671875e-05 seconds.

Result: The code was executed successfully.