Experiment **#17**		Student ID	
Date		Student Name	[@KLWKS_BOT THANOS]

Experiment Title: Implementation of Programs on Comparison of time complexity for recursive algorithms.

Aim/Objective: To understand the concept and implementation of Basic programs on Comparison of time complexity for recursive alg.

Description: The students will understand and able to implement programs on Comparison of time complexity for recursive alg.

Pre-Requisites:

Knowledge: Comparison of time complexity for recursive alg. in C/C++/Python

Tools: Code Blocks/Eclipse IDE

Pre-Lab:

Given an integer n, return true if it is a power of two. Otherwise, return false.

An integer n is a power of two, if there exists an integer x such that n == 2^x.

Example 1:

Input: n = 1

Output: true

Explanation: 20 = 1

Example 2:

Input: n = 16

Output: true

Explanation: 24 = 16

Example 3:

Input: n = 3

Output: false

Constraints:

-231 <= n <= 231 – 1

- **Algorithm/Procedure/Program:**

Algorithm:

1. **Input**: Integer `n`.
2. If `n <= 0`, return `false`.
3. If `(n & (n - 1)) == 0`, return `true`; otherwise, return `false`.

Example:

- For `n = 16`: `(16 & 15) == 0` → return `true`.
- For `n = 3`: `(3 & 2) != 0` → return `false`.

Time Complexity: O(1)

```
#include <stdio.h>
#include <stdbool.h>


bool isPowerOfTwo(int n);

int main() {
   int n = 16;
   if (isPowerOfTwo(n)) {
      printf("%d is a power of two.\n", n);
   } else {
      printf("%d is not a power of two.\n", n);
   }
   return 0;
}

bool isPowerOfTwo(int n) {
   return n > 0 && (n & (n - 1)) == 0;
}
```

- **Data and results:**

Data:

Input: Integer `n`. Check if `n` is a power of two.

Result:

For `n = 16`, result is `true`. For `n = 3`, result is `false`.

- **Analysis and inferences:**

Analysis:

Time complexity is O(1) due to bitwise operations used.

Inferences:

The algorithm works efficiently for any valid integer input.

In-Lab:

A digit string is good if the digits (0-indexed) at even indices are even and the digits at odd indices are prime (2, 3, 5, or 7).

For example, "2582" is good because the digits (2 and 8) at even positions are even and the digits (5 and 2) at odd positions are prime. However, "3245" is not good because 3 is at an even index but is not even.

Given an integer n, return the total number of good digit strings of length n. Since the answer may be large, return it modulo 109 + 7.

A digit string is a string consisting of digits 0 through 9 that may contain leading zeros.

Example 1:

Input: n = 1

Output: 5

Explanation: The good numbers of length 1 are "0", "2", "4", "6", "8".

Example 2:

Input: n = 4

Output: 400

Example 3:

Input: n = 50

Output: 564908303

Constraints:

1 <= n <= 1015

- **Procedure/Program:**

```
#include <stdio.h>

#define MOD 1000000007

long long mod_exp(long long base, long long exp, long long mod) {
  long long result = 1;
  while (exp > 0) {
    if (exp % 2 == 1) {
```

```
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

long long count_good_digit_strings(long long n) {
    long long even_positions = (n + 1) / 2;
    long long odd_positions = n / 2;
    long long even_part = mod_exp(5, even_positions, MOD);
    long long odd_part = mod_exp(4, odd_positions, MOD);
    return (even_part * odd_part) % MOD;
}

int main() {
    long long n;
    scanf("%lld", &n);
    printf("%lld\n", count_good_digit_strings(n));
    return 0;
}
```

- **Data and Results:**

Data: Given an integer n, find number of good digit strings.

Result: For input $n = 4$, result is 400 modulo $10^9 + 7$.

- **Analysis and Inferences:**

Analysis: Efficient power calculation using modular exponentiation handles large n.

Inferences: Solution scales well for large n using $O(\log n)$.

Post -Lab:

The Fibonacci numbers, commonly denoted F(n) form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

F(0) = 0, F(1) = 1

F(n) = F(n - 1) + F(n - 2), for n > 1.

Given n, calculate F(n).

Example 1:

Input: n = 2

Output: 1

Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.

Example 2:

Input: n = 3

Output: 2

Explanation: F(3) = F(2) + F(1) = 1 + 1 = 2.

Example 3:

Input: n = 4

Output: 3

Explanation: F(4) = F(3) + F(2) = 2 + 1 = 3.

Constraints:

0 <= n <= 30

- **Procedure/Program:**

```
#include <stdio.h>

int fibonacci(int n) {
   if (n == 0) return 0;
   else if (n == 1) return 1;

   int a = 0, b = 1, temp;
   for (int i = 2; i <= n; i++) {
      temp = a + b;
      a = b;
```

```
        b = temp;
    }
    return b;
}

int main() {
    int n;
    scanf("%d", &n);
    printf("%d\n", fibonacci(n));
    return 0;
}
```

- **Data and Results:**

Data: Input: n = 2, n = 3, n = 4.

Result: Output: 1, 2, 3 for n = 2, 3, 4.

- **Analysis and Inferences:**

Analysis: Fibonacci calculation follows efficient iterative approach for correct results.

Inferences: Algorithm calculates Fibonacci numbers with optimal time complexity, O(n).

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What is a recursive algorithm, and how does it differ from an iterative algorithm?

- A recursive algorithm calls itself; iterative uses loops.

2. Compare the time complexity of a recursive algorithm with its iterative counterpart for the same problem.

- Recursion has higher overhead; iteration is more memory-efficient.

3. What is the significance of the base case in recursive algorithms when analyzing time complexity?

- The base case stops recursion, affecting the number of calls.

4. Can you provide an example of a problem where a recursive algorithm has a better time complexitycompared to an iterative algorithm?

- Fibonacci with memoization can be more efficient recursively.

Evaluator Remark (if Any):	**Marks Secured ___ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.