Victoria University Of Bangladesh

Course title ~ CSE-108

Bachelor of Tourism & Hotel Management

Submitted By ~ Computer Fundamental and Programming Techniques
Name ~ Khan Sifat
Student ID: 1521530011
Batch - 53
Program - BTHM

1. (a)

De Morgan's Theorems are fundamental principles in Boolean algebra that describe the relationships between logical operations involving negation, conjunction (AND), and disjunction (OR). There are two De Morgan's Theorems:

1. **First De Morgan's Theorem**: It states that the negation of a conjunction (AND) is equivalent to the disjunction (OR) of the negations of the individual terms.

   Mathematically, it can be represented as:

   $\neg(A \land B) \equiv (\neg A) \lor (\neg B)$

2. **Second De Morgan's Theorem**: It states that the negation of a disjunction (OR) is equivalent to the conjunction (AND) of the negations of the individual terms.

   Mathematically, it can be represented as:

   $\neg(A \lor B) \equiv (\neg A) \land (\neg B)$

Now, let's prove each theorem using truth tables:

1. **Proof of First De Morgan's Theorem**:

A	B	A ∧ B	¬(A ∧ B)	¬A	¬B	(¬A) ∨ (¬B)
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

As we can see from the truth table, the columns for ¬(A ∧ B) and (¬A) ∨ (¬B) are identical, confirming the equivalence of ¬(A ∧ B) and (¬A) ∨ (¬B).

2. **Proof of Second De Morgan's Theorem**:

A	B	A ∨ B	¬(A ∨ B)	¬A	¬B	(¬A) ∧ (¬B)
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Similarly, the columns for ¬(A ∨ B) and (¬A) ∧ (¬B) are identical, confirming the equivalence of ¬(A ∨ B) and (¬A) ∧ (¬B).

These truth tables provide formal proofs of De Morgan's Theorems by demonstrating that the logical expressions on both sides of each equivalence have the same truth values for all possible combinations of input values (A and B).

2. (a)

Sure, here's a simple C program to determine whether a given year is a leap year or not:

```c
#include <stdio.h>

int main() { int year;

    // Input year from the user
    printf("Enter the year: ");
    scanf("%d", &year);

    // Check if the year is divisible by 4
    // If divisible by 100, also check if divisible by 400
    if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) {
        printf("%d is a leap year.\n", year);
    } else {
        printf("%d is not a leap year.\n", year);
    }

    return 0;
}
```

Explanation:
1. We prompt the user to input a year.
2. We use conditional statements to check if the given year satisfies the conditions for a leap year:
   - If the year is divisible by 4 and not divisible by 100, or
   - If the year is divisible by 400.
3. If the conditions are met, we print that the year is a leap year; otherwise, we print that it is not a leap year.

b)

Sure, here's an algorithm to find the biggest score among three given scores:

Algorithm to Find the Biggest Score:

1. Start
2. Input three scores: score1, score2, score3
3. Set max_score = score1
4. If score2 > max_score, set max_score = score2
5. If score3 > max_score, set max_score = score3
6. Print max_score as the biggest score
7. Stop

Flowchart:
```plaintext
  +--------+              +--------+
  | Start  |             |  End   |
  +---+----+              +----+---+
      |                    |
      |                    |
      v                    v
+-----+-----+               +--+--+
| Input     |              | Print|
| scores    |               | Max  |
+----+------+               +-+--+-+
     |                     ^  |
     v                     |  |
+----+------+                |  |
| max_score |<----------------+  |
+----+------+                  |
     |                       |
     v                       |
+----+------+                  |
|  score1  |                 |
|  score2  |                 |
|  score3  |                 |
+----------+                 |
                   |
    +---------+        |
    | score2 >|--------+
    | max     |
    +----+----+
         |
         |
         v
  +------+------+
  | max_score =|
  | score2     |
  +------+------+
         |
         |
         v
 +-------+-------+
 | score3 > max  |
 | _score        |
 +-------+-------+
         |
         |
         v
 +-------+-------+
 | max_score =   |
 | score3        |
 +---------------+
```

3.

a)

Certainly! Here are the basic logic gates along with their symbols and truth tables for two inputs:

1. **AND Gate**:

   - Symbol:
     ```
         -----
     A ---|     |
         | AND |--- Output (A AND B)
     B ---|     |
         -----
     ```
   - Truth Table:
     ```
     A | B | Output
     ----------
     0 | 0 |   0
     0 | 1 |   0
     1 | 0 |   0
     1 | 1 |   1
     ```

2. **OR Gate**:
   - Symbol:
     ```
         -----
     A ---|     |
         |  OR |--- Output (A OR B)
     B ---|     |
         -----
     ```
   - Truth Table:
     ```
     A | B | Output
     ----------
     0 | 0 |   0
     0 | 1 |   1
     1 | 0 |   1
     1 | 1 |   1
     ```

```

3. **NOT Gate**:
   - Symbol:
     ```

   A | NOT |--- Output (NOT A)

     ```
   - Truth Table:
     ```
     A | Output
     ----
     0 | 1
     1 | 0
     ~~~

The laws for getting output from the given input using these gates are as follows:

- **AND Gate**: The output is high (1) only when both inputs are high (1).
- **OR Gate**: The output is high (1) if any of the inputs are high (1).
- **NOT Gate**: The output is the inverse of the input.

These logic gates form the basis of digital circuit design and can be combined in various ways to perform complex logical operations.

B)

Sure, here is a simple C program that calculates the average of five numbers:

```c
#include <stdio.h>
     ~~~

```
int main() {
    float num1, num2, num3, num4, num5, average;

    // Input five numbers from the user
    printf("Enter five numbers:\n");
    scanf("%f %f %f %f %f", &num1, &num2, &num3, &num4, &num5);

    // Calculate the average
    average = (num1 + num2 + num3 + num4 + num5) / 5;

    // Output the average
    printf("The average of the five numbers is: %.2f\n", average);

    return 0;
}
```

Explanation

1. **Include necessary header files:**
   - `#include <stdio.h>` is included for standard input and output functions.

2. **Declare variables:**
   - `float num1, num2, num3, num4, num5, average;` declares variables to store the five numbers and the average. `float` is used to allow for decimal points.

3. **Prompt user for input:**
   - `printf("Enter five numbers:\n");` prompts the user to input five numbers.
   - `scanf("%f %f %f %f %f", &num1, &num 2, &num 3, &num 4, &num 5);` reads five floating-point numbers from the user and stores them in the respective variables.

4. **Calculate the average:**

   - `average = (num1 + num2 + num3 + num4 + num5) / 5;` computes the average by adding the five numbers and dividing by 5.

5. **Output the result:**
   - `printf("The average of the five numbers is: %.2f\n", average);` prints the average with two decimal places.

Compilation and Execution

To compile and run the program:

1. Save the code in a file, e.g., `average.c`.
2. Open a terminal or command prompt.
3. Navigate to the directory containing `average.c`.
4. Compile the program using a C compiler, e.g., `gcc average.c -o average`.
5. Run the executable, e.g., `./average` on Unix-like systems or `average.exe` on Windows.

The program will prompt you to enter five numbers, and then it will display the average of those numbers.

4.

A)

Algorithm to Find the Summation of Series 1 + 2 + 3 + ... + N

Step 1: Start

Step 2: Declare integer variables `N` and `sum`. Set `sum` to 0.

Step 3: Input the value of `N`.

Step 4: For `i` from 1 to `N`, do the following:

  - Add `i` to `sum`.

Step 5: End For loop.

Step 6: Output the value of `sum`.

Step 7: End

Flowchart to Find the Summation of Series 1 + 2 + 3 + ... + N

Below is a description of how the flowchart should be designed:

1. **Start**: Begin the process.
2. **Input N**: Get the value of `N` from the user.
3. **Initialize sum**: Set `sum` to 0.
4. **Initialize i**: Set `i` to 1.
5. **Check if i <= N**: If `i` is less than or equal to `N`, proceed to the next step. If not, go to step 9.
6. **Add i to sum**: Update `sum` by adding `i` to it.
7. **Increment i**: Increase the value of `i` by 1.
8. **Go back to step 5**: Repeat the loop until `i` is greater than `N`.
9. **Output sum**: Display the value of `sum`.
10. **End**: Terminate the process.

Flowchart

Here is a visual representation of the flowchart:

```
 +---------+
 |  Start  |
 +---------+
      |
      v
 +---------+
 | Input N |
 +---------+
      |
      v
 +-----------+
 | sum = 0   |
 +-----------+
      |
      v
 +-----------+
 | i = 1     |
 +-----------+
      |
      v
 +-----------------+
 |  i <= N ?       |
 +-----------------+
      |  No
      v
      |
    Yes
      v
 +-----------------+
 | sum = sum + i   |
 +-----------------+
      |
      v
 +-----------+
 | i = i + 1 |
 +-----------+
      |
      v
 +-----------------+
 |  i <= N ?       |
 +-----------------+
      |  No
      v
      |
    Yes
      v
 +-----------------+
 | Output sum      |
 +-----------------+
      |
      v
 +---------+
 |   End   |
 +---------+
```

Explanation of Flowchart

1. **Start**: The algorithm begins.
2. **Input N**: The user provides the value of `N`.
3. **Initialize sum**: Initialize `sum` to 0 to start the accumulation process.
4. **Initialize i**: Set the loop counter `i` to 1.
5. **Check condition (i <= N)**: Check if the current value of `i` is less than or equal to `N`.
   - **If true**: Continue to the next step.
   - **If false**: Go to step 9 to output the result.
6. **Add i to sum**: Add the current value of `i` to `sum`.
7. **Increment i**: Increase `i` by 1.
8. **Repeat the loop**: Return to step 5 to check the condition again.
9. **Output sum**: Display the final value of `sum`.
10. **End**: The algorithm ends.

This algorithm and flowchart provide a structured approach to compute the summation of the series from 1 to `N`.

B)

Certainly! Below is a C program that calculates the summation of the series \(1 + 2 + 3 + \ldots + N\) based on the algorithm and flowchart described:

```c
#include <stdio.h>

int main() {
    int N, sum = 0;

    // Input the value of N from the user
    printf("Enter the value of N: ");
    scanf("%d", &N);

    // Calculate the summation from 1 to N
    for (int i = 1; i <= N; i++) {
        sum += i; // Add i to sum
    }

    // Output the result
    printf("The summation of the series 1 + 2 + 3 + ... + %d is: %d\n", N,
sum);

    return 0;
}
```

Explanation

1. **Include necessary header files:**
   - `#include <stdio.h>` is included for standard input and output functions.

2. **Declare variables:**

   - `int N, sum = 0;` declares the integer variables `N` for the input number and `sum` to store the summation result, initializing `sum` to 0.

3. **Input the value of N:**
   - `printf("Enter the value of N: ");` prompts the user to enter a number.
   - `scanf("%d", &N);` reads the integer value input by the user and stores it in `N`.

4. **Calculate the summation:**
   - `for (int i = 1; i <= N; i++)` initializes the loop variable `i` to 1 and continues to loop as long as `i` is less than or equal to `N`, incrementing `i` by 1 each iteration.
   - `sum += i;` adds the current value of `i` to `sum`.

5. **Output the result:**
   - `printf("The summation of the series 1 + 2 + 3 + ... + %d is: %d\n", N, sum);` prints the final value of `sum`.

Compilation and Execution

To compile and run the program:

1. Save the code in a file, e.g., `summation.c`.
2. Open a terminal or command prompt.
3. Navigate to the directory containing `summation.c`.
4. Compile the program using a C compiler, e.g., `gcc summation.c -o summation`.
5. Run the executable, e.g., `./summation` on Unix-like systems or `summation.exe` on Windows.

The program will prompt you to enter the value of `N`, then it will calculate and display the summation of the series from 1 to `N`.