Module 4

Introduction to Memorization

Example 1: Fibonacci Sequence

Recursion without Memorization

```
#include <stdio.h>

int fibonacci(int n) {
  if (n <= 1) {
    return n;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
  int n = 10;
  printf("Fibonacci number at position %d is %d\n", n, fibonacci(n));
  return 0;
}
```

Memorized Fibonacci

```
#include <stdio.h>

#define MAX 1000

int fibonacci(int n, int memo[]) {
  if (memo[n] != -1) {
    return memo[n];
  }
  if (n <= 1) {
    memo[n] = n;
  } else {
    memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
  }
  return memo[n];
}

int main() {
  int n = 10;
  int memo[MAX];
  for (int i = 0; i < MAX; i++) {
    memo[i] = -1;
  }
  printf("Fibonacci number at position %d is %d\n", n, fibonacci(n, memo));
  return 0;
}
```

Example 2: Factorial Calculation
Factorial without Memorization

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

```
int factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

int main() {
    int n = 5;
    printf("Factorial of %d is %d\n", n, factorial(n));
    return 0;
}
```

Memorized Factorial

```
#include <stdio.h>

#define MAX 1000

int factorial(int n, int memo[]) {
    if (memo[n] != -1) {
        return memo[n];
    }
    if (n <= 1) {
        memo[n] = 1;
    } else {
        memo[n] = n * factorial(n - 1, memo);
    }
    return memo[n];
}

int main() {
    int n = 5;
    int memo[MAX];
    for (int i = 0; i < MAX; i++) {
        memo[i] = -1;
    }
    printf("Factorial of %d is %d\n", n, factorial(n, memo));
    return 0;
}
```

Example 3: Climbing Stairs
Climbing Stairs without Memorization

```
#include <stdio.h>

int climbStairs(int n) {
    if (n <= 1) {
        return 1;
    }
    return climbStairs(n - 1) + climbStairs(n - 2);
```

```c
}

int main() {
    int n = 5;
    printf("Number of ways to climb %d stairs is %d\n", n, climbStairs(n));
    return 0;
}
```

Memorized Climbing Stairs

```c
#include <stdio.h>

#define MAX 1000

int climbStairs(int n, int memo[]) {
    if (memo[n] != -1) {
        return memo[n];
    }
    if (n <= 1) {
        memo[n] = 1;
    } else {
        memo[n] = climbStairs(n - 1, memo) + climbStairs(n - 2, memo);
    }
    return memo[n];
}

int main() {
    int n = 5;
    int memo[MAX];
    for (int i = 0; i < MAX; i++) {
        memo[i] = -1;
    }
    printf("Number of ways to climb %d stairs is %d\n", n, climbStairs(n, memo));
    return 0;
}
```

Dynamic Programming

1. Fibonacci using Recursion

```c
#include <stdio.h>

// Function to calculate Fibonacci number using recursion
int fibonacci(int n) {
    if (n <= 1) {
        return n; // Base case: F(0) = 0, F(1) = 1
    }
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursive call
}

int main() {
    int n = 10;
    printf("Fibonacci number at position %d is %d\n", n, fibonacci(n));
    return 0;
}
```

2. Fibonacci using Memoization

```
#include <stdio.h>

#define MAX 1000

// Function to calculate Fibonacci number using memoization
int fibonacci(int n, int memo[]) {
    if (memo[n] != -1) {
        return memo[n]; // Return the stored result if it exists
    }
    if (n <= 1) {
        memo[n] = n; // Base case: F(0) = 0, F(1) = 1
    } else {
        memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo); // Recursive call with
memoization
    }
    return memo[n];
}

int main() {
    int n = 10;
    int memo[MAX];
    for (int i = 0; i < MAX; i++) {
        memo[i] = -1; // Initialize memo array with -1
    }
    printf("Fibonacci number at position %d is %d\n", n, fibonacci(n, memo));
    return 0;
}
```

3. Fibonacci using Dynamic Programming (Tabulation)

```
#include <stdio.h>

// Function to calculate Fibonacci number using dynamic programming (tabulation)
int fibonacci(int n) {
    if (n <= 1) {
        return n; // Base case: F(0) = 0, F(1) = 1
    }
    int dp[n + 1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2]; // Fill the dp array from bottom-up
    }
    return dp[n];
}

int main() {
    int n = 10;
    printf("Fibonacci number at position %d is %d\n", n, fibonacci(n));
    return 0;
}
```

Module 5

1. LCS using Recursion

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

int LCS_Recursion(char *X, char *Y, int m, int n) {
    if (m == 0 || n == 0)
        return 0;
    if (X[m-1] == Y[n-1])
        return 1 + LCS_Recursion(X, Y, m-1, n-1);
    else
        return (LCS_Recursion(X, Y, m, n-1) > LCS_Recursion(X, Y, m-1, n)) ?
                LCS_Recursion(X, Y, m, n-1) : LCS_Recursion(X, Y, m-1, n);
}

int main() {
```

```c
    char X[] = "ad";
    char Y[] = "abcd";
    int m = strlen(X);
    int n = strlen(Y);
    printf("Length of LCS (Recursion) is %d\n", LCS_Recursion(X, Y, m, n));
    return 0;
}
```

2. LCS using Memoization

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

int LCS_Memoization(char *X, char *Y, int m, int n, int memo[][n+1]) {
    if (m == 0 || n == 0)
        return 0;
    if (memo[m][n] != -1)
        return memo[m][n];
    if (X[m-1] == Y[n-1])
        memo[m][n] = 1 + LCS_Memoization(X, Y, m-1, n-1, memo);
    else
        memo[m][n] = (LCS_Memoization(X, Y, m, n-1, memo) > LCS_Memoization(X, Y, m-1, n, 
memo)) ?
                     LCS_Memoization(X, Y, m, n-1, memo) : LCS_Memoization(X, Y, m-1, n, memo);
    return memo[m][n];
}

int main() {
    char X[] = "ad";
    char Y[] = "abcd";
    int m = strlen(X);
    int n = strlen(Y);
    int memo[m+1][n+1];
    for (int i = 0; i <= m; i++)
        for (int j = 0; j <= n; j++)
            memo[i][j] = -1;
    printf("Length of LCS (Memoization) is %d\n", LCS_Memoization(X, Y, m, n, memo));
    return 0;
}
```

3. LCS using Dynamic Programming (Tabulation)

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

// Function to calculate LCS using dynamic programming (tabulation)
#include <stdio.h>
#include <string.h>

int LCS_Tabulation(char *X, char *Y, int m, int n) {
    int L[m+1][n+1];
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (X[i-1] == Y[j-1])
                L[i][j] = L[i-1][j-1] + 1;
            else
                L[i][j] = (L[i-1][j] > L[i][j-1]) ? L[i-1][j] : L[i][j-1];
        }
    }
    return L[m][n];
}

int main() {
    char X[] = "ad";
    char Y[] = "abcd";
    int m = strlen(X);
    int n = strlen(Y);
    printf("Length of LCS (Tabulation) is %d\n", LCS_Tabulation(X, Y, m, n));
    return 0;
}
```

1. Edit Distance using Recursion

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

// Function to find minimum of three numbers
int min(int x, int y, int z) {
    return (x < y) ? ((x < z) ? x : z) : ((y < z) ? y : z);
}

// Function to calculate Edit Distance using recursion
int editDistance(char* str1, char* str2, int m, int n) {
    if (m == 0) return n;
    if (n == 0) return m;

    if (str1[m - 1] == str2[n - 1]) {
        return editDistance(str1, str2, m - 1, n - 1);
    }

    return 1 + min(editDistance(str1, str2, m - 1, n),
               editDistance(str1, str2, m, n - 1),
               editDistance(str1, str2, m - 1, n - 1));
}

int main() {
    char str1[] = "sunday";
    char str2[] = "saturday";
    printf("Minimum edit distance is %d\n", editDistance(str1, str2, strlen(str1), strlen(str2)));
    return 0;
}
```

2. Edit Distance using Memoization

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

#define MAX 1000

// Function to find minimum of three numbers
int min(int x, int y, int z) {
  return (x < y) ? ((x < z) ? x : z) : ((y < z) ? y : z);
}

// Function to calculate Edit Distance using memoization
int editDistance(char* str1, char* str2, int m, int n, int memo[MAX][MAX]) {
  if (m == 0) return n;
  if (n == 0) return m;

  if (memo[m][n] != -1) {
    return memo[m][n]; // Return the stored result if it exists
  }

  if (str1[m - 1] == str2[n - 1]) {
    memo[m][n] = editDistance(str1, str2, m - 1, n - 1, memo);
  } else {
    memo[m][n] = 1 + min(editDistance(str1, str2, m - 1, n, memo),
               editDistance(str1, str2, m, n - 1, memo),
               editDistance(str1, str2, m - 1, n - 1, memo));
  }
  return memo[m][n];
}

int main() {
  char str1[] = "sunday";
  char str2[] = "saturday";
  int memo[MAX][MAX];
  for (int i = 0; i < MAX; i++) {
    for (int j = 0; j < MAX; j++) {
      memo[i][j] = -1; // Initialize memo array with -1
    }
  }
  printf("Minimum edit distance is %d\n", editDistance(str1, str2, strlen(str1), strlen(str2),
memo));
  return 0;
}
```

3. Edit Distance using Dynamic Programming (Tabulation)

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

// Function to find minimum of three numbers
int min(int x, int y, int z) {
  return (x < y) ? ((x < z) ? x : z) : ((y < z) ? y : z);
}
```

```
// Function to calculate Edit Distance using dynamic programming (tabulation)
int editDistance(char* str1, char* str2, int m, int n) {
    int dp[m + 1][n + 1];

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0) {
                dp[i][j] = j; // If first string is empty, insert all characters of second string
            } else if (j == 0) {
                dp[i][j] = i; // If second string is empty, remove all characters of first string
            } else if (str1[i - 1] == str2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1]; // If characters are the same, no operation needed
            } else {
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]); // Min of insert, delete, replace
            }
        }
    }

    return dp[m][n];
}

int main() {
    char str1[] = "sunday";
    char str2[] = "saturday";
    printf("Minimum edit distance is %d\n", editDistance(str1, str2, strlen(str1), strlen(str2)));
    return 0;
}
```

0/1 Knapsack Problem

1. Knapsack using Recursion

```
#include <stdio.h>

// Function to calculate the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to solve the Knapsack problem using recursion
int knapsack(int W, int wt[], int val[], int n) {
    if (n == 0 || W == 0) {
        return 0;
    }
    if (wt[n - 1] > W) {
        return knapsack(W, wt, val, n - 1);
    } else {
        return max(val[n - 1] + knapsack(W - wt[n - 1], wt, val, n - 1), knapsack(W, wt, val, n - 1));
    }
}
```

```
int main() {
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
    printf("Maximum value in Knapsack = %d\n", knapsack(W, wt, val, n));
    return 0;
}
```

2. Knapsack using Memoization

```
#include <stdio.h>

#define MAX 1000

// Function to calculate the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to solve the Knapsack problem using memoization
int knapsack(int W, int wt[], int val[], int n, int memo[MAX][MAX]) {
    if (n == 0 || W == 0) {
        return 0;
    }
    if (memo[n][W] != -1) {
        return memo[n][W];
    }
    if (wt[n - 1] > W) {
        memo[n][W] = knapsack(W, wt, val, n - 1, memo);
    } else {
        memo[n][W] = max(val[n - 1] + knapsack(W - wt[n - 1], wt, val, n - 1, memo), knapsack(W,
wt, val, n - 1, memo));
    }
    return memo[n][W];
}

int main() {
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
    int memo[MAX][MAX];
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            memo[i][j] = -1; // Initialize memo array with -1
        }
    }
    printf("Maximum value in Knapsack = %d\n", knapsack(W, wt, val, n, memo));
    return 0;
}
```

3. Knapsack using Dynamic Programming (Tabulation)

```
#include <stdio.h>

// Function to calculate the maximum of two integers
int max(int a, int b) {
  return (a > b) ? a : b;
}

// Function to solve the Knapsack problem using dynamic programming (tabulation)
int knapsack(int val[], int wt[], int n, int W) {
  int dp[n + 1][W + 1];

  for (int i = 0; i <= n; i++) {
    for (int w = 0; w <= W; w++) {
      if (i == 0 || w == 0) {
        dp[i][w] = 0; // Base case: If no items or no capacity, value is 0
      } else if (wt[i - 1] <= w) {
        dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]); // Include or exclude the item
      } else {
        dp[i][w] = dp[i - 1][w]; // Exclude the item
      }
    }
  }

  return dp[n][W];
}

int main() {
  int val[] = {60, 100, 120};
  int wt[] = {10, 20, 30};
  int W = 50;
  int n = sizeof(val) / sizeof(val[0]);

  printf("Maximum value in Knapsack = %d\n", knapsack(val, wt, n, W));
  return 0;
}
```

Dynamic Programming Interview Problems

Problem 1: Climbing Stairs (Easy)
You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?
Dynamic Programming Solution

```
#include <stdio.h>

int climbStairs(int n) {
  if (n <= 1) {
    return 1;
  }
  int dp[n + 1];
  dp[0] = 1;
  dp[1] = 1;
  for (int i = 2; i <= n; i++) {
    dp[i] = dp[i - 1] + dp[i - 2];
  }
  return dp[n];
}

int main() {
  int n = 10;
  printf("Number of ways to climb %d stairs is %d\n", n, climbStairs(n));
  return 0;
}
```

Problem 2: House Robber (Medium)
You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, and you cannot rob two adjacent houses. Determine the maximum amount of money you can rob tonight without alerting the police.
Dynamic Programming Solution

```
#include <stdio.h>

int max(int a, int b) {
  return (a > b) ? a : b;
}

int rob(int* nums, int numsSize) {
  if (numsSize == 0) return 0;
  if (numsSize == 1) return nums[0];

  int dp[numsSize];
  dp[0] = nums[0];
  dp[1] = max(nums[0], nums[1]);

  for (int i = 2; i < numsSize; i++) {
    dp[i] = max(dp[i - 1], nums[i] + dp[i - 2]);
  }

  return dp[numsSize - 1];
}

int main() {
  int nums[] = {1, 2, 3, 1};
  int numsSize = sizeof(nums) / sizeof(nums[0]);
  printf("Maximum amount of money that can be robbed is %d\n", rob(nums, numsSize));
  return 0;
}
```

Problem 3: Coin Change (Medium)
Given an integer array coin representing coins of different denominations and an integer amount representing a total amount of money, return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

Dynamic Programming Solution

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

int coinChange(int* coins, int coinsSize, int amount) {
  int dp[amount + 1];
  for (int i = 0; i <= amount; i++) {
    dp[i] = amount + 1;
  }
  dp[0] = 0;

  for (int i = 1; i <= amount; i++) {
    for (int j = 0; j < coinsSize; j++) {
      if (coins[j] <= i) {
        dp[i] = dp[i] < (dp[i - coins[j]] + 1) ? dp[i] : (dp[i - coins[j]] + 1);
      }
    }
  }

  return dp[amount] > amount ? -1 : dp[amount];
}

int main() {
  int coins[] = {1, 2, 5};
  int coinsSize = sizeof(coins) / sizeof(coins[0]);
  int amount = 11;
  printf("Fewest number of coins needed to make up %d is %d\n", amount, coinChange(coins,
coinsSize, amount));
  return 0;
}
```

Problem 4: Longest Increasing Subsequence (Medium)
Given an integer array nums, return the length of the longest strictly increasing subsequence.
Dynamic Programming Solution

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

```
int lengthOfLIS(int* nums, int numsSize) {
    if (numsSize == 0) return 0;

    int dp[numsSize];
    for (int i = 0; i < numsSize; i++) {
        dp[i] = 1;
    }

    int maxLength = 1;
    for (int i = 1; i < numsSize; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j]) {
                dp[i] = dp[i] > (dp[j] + 1) ? dp[i] : (dp[j] + 1);
            }
        }
        maxLength = maxLength > dp[i] ? maxLength : dp[i];
    }

    return maxLength;
}

int main() {
    int nums[] = {10, 9, 2, 5, 3, 7, 101, 18};
    int numsSize = sizeof(nums) / sizeof(nums[0]);
    printf("Length of the longest increasing subsequence is %d\n", lengthOfLIS(nums, numsSize));
    return 0;
}
```

Problem 5: Subset Sum Problem (Medium)
Problem Statement

Given a set of positive integers and an integer sum, determine if there is a subset of the given set with a sum equal to the given sum.
Dynamic Programming Solution

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

bool isSubsetSum(int set[], int n, int sum) {
    bool subset[n + 1][sum + 1];

    for (int i = 0; i <= n; i++) {
        subset[i][0] = true;
    }

    for (int i = 1; i <= sum; i++) {
        subset[0][i] = false;
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (j < set[i - 1]) {
```

```
            subset[i][j] = subset[i - 1][j];
         } else {
            subset[i][j] = subset[i - 1][j] || subset[i - 1][j - set[i - 1]];
         }
      }
   }

   return subset[n][sum];
}

int main() {
   int set[] = {3, 34, 4, 12, 5, 2};
   int sum = 9;
   int n = sizeof(set) / sizeof(set[0]);
   if (isSubsetSum(set, n, sum) == true)
      printf("Found a subset with given sum\n");
   else
      printf("No subset with given sum\n");
   return 0;
}
```

Problem 6: Maximum Subarray Sum (Easy)
Problem Statement
Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

```
#include <stdio.h>

int max(int a, int b) {
   return (a > b) ? a : b;
}

int maxSubArray(int* nums, int numsSize) {
   int max_so_far = nums[0];
   int curr_max = nums[0];

   for (int i = 1; i < numsSize; i++) {
      curr_max = max(nums[i], curr_max + nums[i]);
      max_so_far = max(max_so_far, curr_max);
   }

   return max_so

_far;
}
int main() {
   int nums[] = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
   int numsSize = sizeof(nums) / sizeof(nums[0]);
   printf("Maximum subarray sum is %d\n", maxSubArray(nums, numsSize));
   return 0;
}
```