# Department of Computing

**Name: Mahum Samar**

**CMS ID : 290647**

**CS250: Data Structure and Algorithms**

**Class: BSCS 9B**

# lab 08:recursion

# Task 01

## Code:

|  |
| --- |
| #include <iostream> #include <string> #include <cstring>  using namespace std;  class ListNode { public:  int data;  ListNode \*next; }; class SinglyLinkedList {  *//class for the singly linked list* public:  ListNode \*headNode;  ListNode \*lastNode;    bool isEmpty()  {  *//method to check if the list is empty* return headNode == **NULL**;  }  SinglyLinkedList()  {  *//constructor* headNode = **NULL**;  lastNode = **NULL**;  }   void InsertNode(int x)  {  *//method to insert new node in the list* ListNode \*newNode = new ListNode();  newNode -> data = x;  if(isEmpty())  {  headNode = newNode;  lastNode = newNode;  }  else  {  lastNode -> next = newNode;  newNode -> next = **NULL**;  lastNode = newNode;  }   }   void InsertNodeReference(ListNode \*node)  {  *//method to insert the node by its reference int he list* if(isEmpty())  {  headNode = node;  lastNode = node;  }  else  {  lastNode -> next = node;  node -> next = **NULL**;  lastNode = node;  }  }   void PrintList(ListNode \*loc)  {  *//method for printing the given list* if(!isEmpty())  {  while(loc != **NULL**)  {  cout << loc -> data << " ";  loc = loc -> next;  }  cout << endl;  }  else  {  cout << "List is empty." << endl;  }   }  };  class RecursiveFunctions {  *//class containing all the recursive functions for the lab 08* public:  int RecursiveFactorial(int n)  {  *//task 1 method to find the factorial of the number recursively.* cout << "calling factorial ( " << n << " )" << endl;  *//base case* if(n == 0)  {  cout << "Base case. Returned 1." << endl;  return 1;  }  else  {  *//recursive case* int ans = n \* RecursiveFactorial(n-1);  cout << "factorial ( " << n << " ) ending. Answer = " << ans << endl;  return ans;  }  }   int RecursiveFibonacci(int n)  {  *//task 2 method to calculate the fibonacci number recursively* cout << "calling fibonacci ( " << n << " )" << endl;  *//base case* if(n == 0 || n == 1)  {  cout << "Base case. Returned 1." << endl;  return 1;  }  else  {  *//recursive case* int ans = RecursiveFibonacci(n-1) + RecursiveFibonacci(n-2);  cout << "Fibonacci ( " << n << " ) ending. Answer = " << ans << endl;  return ans;  }  }   void PrintFibonacci(int n)  {  *//method for printing the fibonacci series iteratively* int sum;  int num1 = 0;  int num2 = 1;  for(int i = 0 ; i < n ; i++)  {  *//base case* if(i == 0 || i == 1)  {  sum = i;  }  else  {  *//calculating next number of the series.* sum = num1 + num2;  num1 = num2;  num2 = sum;  }  cout << sum << " ";  }   }   int RecursiveSearch(int array[], int startingIndex, int lastIndex, int x)  {   *//task 3 method to search if the given number exists in the list or not.  //base case* cout << "Starting index : " << startingIndex << " Last index : " << lastIndex << endl;  if(startingIndex > lastIndex)  {  cout << "base case\n" << "Starting index : " << startingIndex << " Last index : " << lastIndex << endl;  return -1;  }  if(array[startingIndex] == x)  {  cout << "if value found\n" << "Starting index : " << startingIndex << " Last index : " << lastIndex << endl;  return startingIndex;  }  *//recursive call* return RecursiveSearch(array, startingIndex+1, lastIndex, x);    }   bool RecursivePalindrome(char str[], int start, int endIndex, int length)  {  *//task 4 method to check if the word is palindrome or not.* cout << "Starting index : " << start << " Last index : " << endIndex << endl;  *//base case* if(length == 0 || length == 1)  {  *//empty or single character.* return true;  }  if(start == endIndex)  {  cout << "base case:\nif middle character\n" << "Starting index : " << start << " Last index : " << endIndex << endl;  *//if middle character* return true;  }  if(str[start] != str[endIndex])  {  cout << "if the corresponding character not equal\n" << "Starting index : " << start << " Last index : " << endIndex << endl;  cout << str[start] << " not equal to " << str[endIndex]<< endl;  *//if corresponding character not equal* return false;  }  else  if(start < endIndex-1)  {  cout << "before recursive call. Starting index : " << start << " Last index : " << endIndex << endl;  *//recursive call* return RecursivePalindrome(str, start + 1, endIndex -1, length);  }  else  {  return false;  }   }   void RecursiveReversePrintSinglyLinkedList(ListNode \*loc)  {  *//task 5 method to print the linked list in reverse order by recursion.* if(loc != **NULL**)  {  cout << "node value passed to the recursion call: " << loc -> data << endl;  RecursiveReversePrintSinglyLinkedList(loc -> next);  cout << "Value returned after recursion call. ";  cout << loc -> data << endl;  }   }   ListNode\* RecursiveReverseSinglyLinkedList(ListNode \*loc, SinglyLinkedList \*singlyLinkedList)  {  *//task 6 method to reverse the singly linked list by recursion.* cout << "at node " << loc -> data << endl;  if(loc == **NULL**)  {  return **NULL**;  }  if(loc -> next == **NULL**)  {  singlyLinkedList -> headNode = loc;  cout << "base case:\nwhen at last node of the list. At node with data." << singlyLinkedList -> headNode -> data << endl;  return loc;  }   cout << "node passed in recursion call with the value of " << loc -> data << endl;  ListNode \*newNode = RecursiveReverseSinglyLinkedList(loc -> next,singlyLinkedList);   cout << "node returned after the recursive call with data." << newNode -> data << endl;  newNode -> next = loc;  loc -> next = **NULL**;  return loc;   }   ListNode\* RecursiveEvenOdd(ListNode \*odd, ListNode \*even, SinglyLinkedList \*singlyLinkedList2)  {  *//task 7 method to separate the even and odd positioned node.  //placing the even positioned nodes in reverse order at the front of the main list   //node to save the previous even node address so that it can be accessed when returned from recursion* ListNode \*evenPre = new ListNode();   *//base case* if(even -> next == **NULL**)  {  cout << "base case \nif next of even positioned node is null." << even -> data << endl;  singlyLinkedList2 -> InsertNodeReference(even);  odd -> next = **NULL**;  return even;  }  else  {  *//if there are more nodes in the list the even and odd nodes are incremented* odd -> next = even -> next;  odd = even -> next;   if(odd -> next == **NULL**)  {  *//if there are no more nodes after the odd node.* singlyLinkedList2 -> InsertNodeReference(even);  return even;  }   evenPre = even;  *//incrementing the even node.* even -> next = odd -> next;  even = odd -> next;   cout << "before recursion call\nodd node with value: " << odd -> data << " even node with value : " << even -> data << endl;  }   *//even node returned* ListNode \*node = RecursiveEvenOdd(odd, even, singlyLinkedList2);   cout << "passing previous even node in the even nodes list with value " << evenPre -> data << endl;  singlyLinkedList2 -> InsertNodeReference(evenPre);   return node;  }  }; |

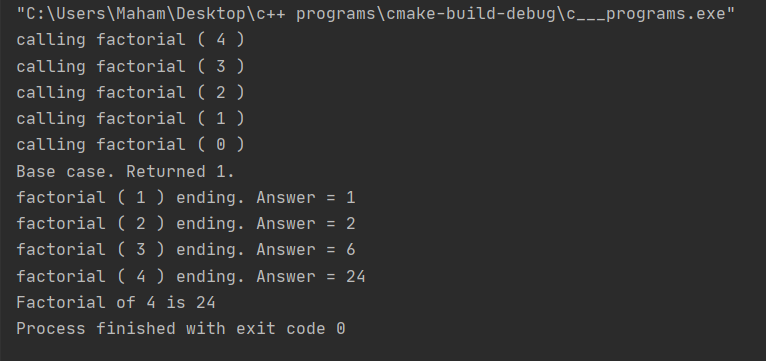
# SEPARATE MAIN METHOD FOR ALL TASK

# Task 01

## Code:

|  |
| --- |
| int main()  {  RecursiveFunctions\* recursiveFunctions = new RecursiveFunctions();  int n = 4;  int answer = recursiveFunctions->RecursiveFactorial(n);  cout << "Factorial of " << n << " is " << answer;  return 0;  } |

## output:

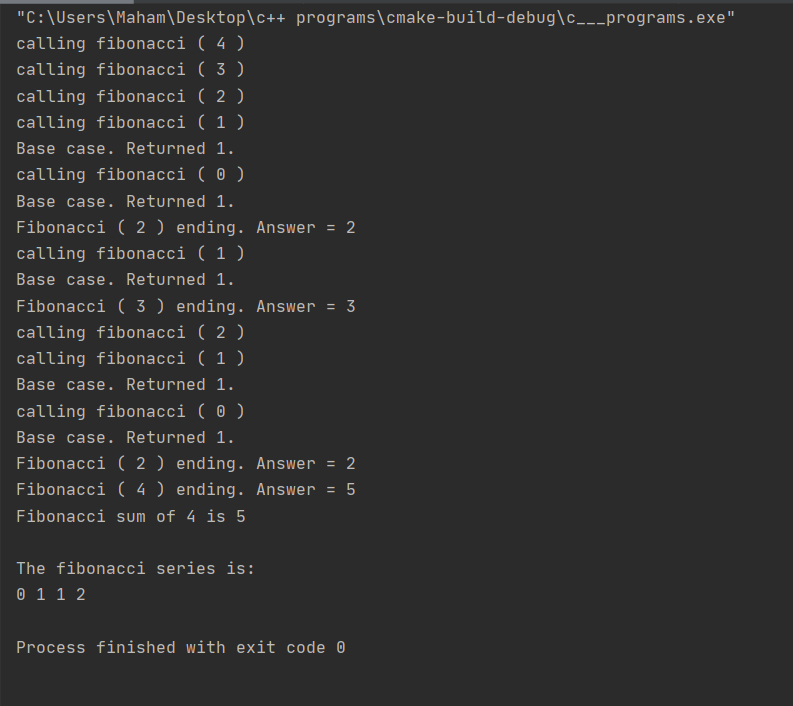


# Task 02

## Code:

|  |
| --- |
| int main()  {  RecursiveFunctions\* recursiveFunctions = new RecursiveFunctions();  int n = 4;  int answer = recursiveFunctions->RecursiveFibonacci(n);  cout << "Fibonacci sum of " << n << " is " << answer << endl;  cout << endl;  cout << "The fibonacci series is: " << endl;  recursiveFunctions->PrintFibonacci(n);  cout << endl;  return 0;  } |

## output:

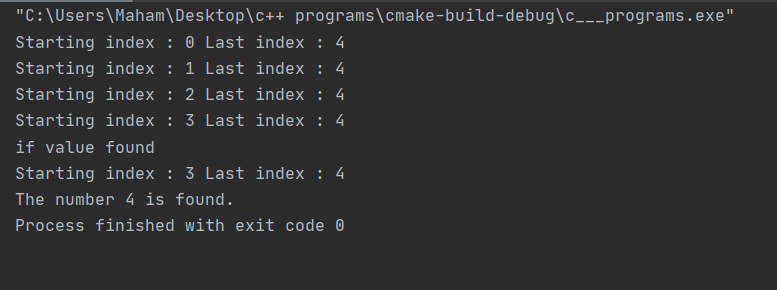


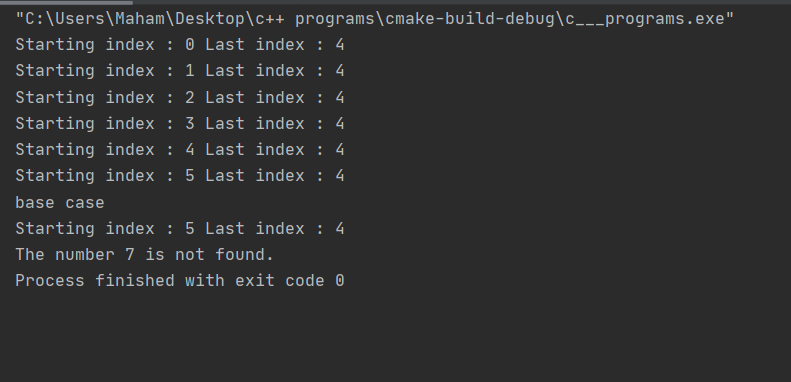
# Task 03

## Code:

|  |
| --- |
| int main()  {  RecursiveFunctions\* recursiveFunctions = new RecursiveFunctions();  //defining an array  int array[] = { 1,2,3,4,5 };  //getting number of elements of array.  int sizeOfArray = sizeof(array) / sizeof(array[0]);  int n = 4, i = 0;  //returning index or -1 based on the found result  int index = recursiveFunctions->RecursiveSearch(array, i, sizeOfArray - 1, n);  if (index != -1)  {  cout << "The number " << n << " is found.";  }  else  {  cout << "The number " << n << " is not found.";  }  return 0;  } |

## output:



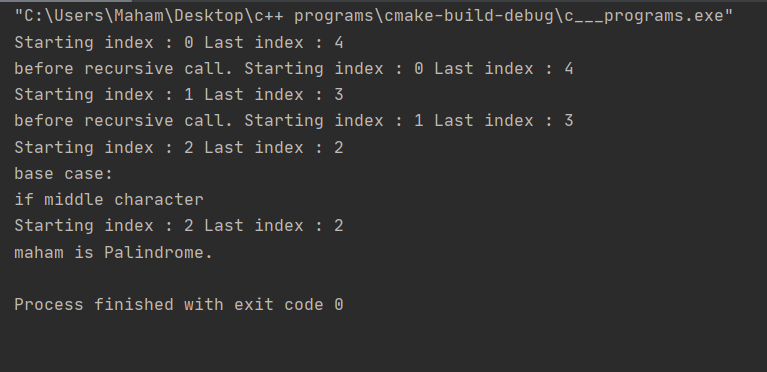


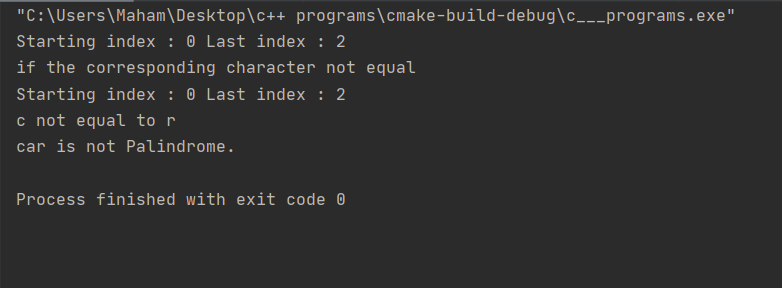
# Task 04

## Code:

|  |
| --- |
| int main()  {  RecursiveFunctions\* recursiveFunctions = new RecursiveFunctions();  //defining an array  char array[] = "maham";  //getting number of characters in the word.  int length = strlen(array);  //calling recursive function  bool result = recursiveFunctions->RecursivePalindrome(array, 0, length - 1, length);  if (result)  {  cout << array << " is Palindrome." << endl;  }  else  {  cout << array << " is not Palindrome." << endl;  }  return 0;  } |

## output:



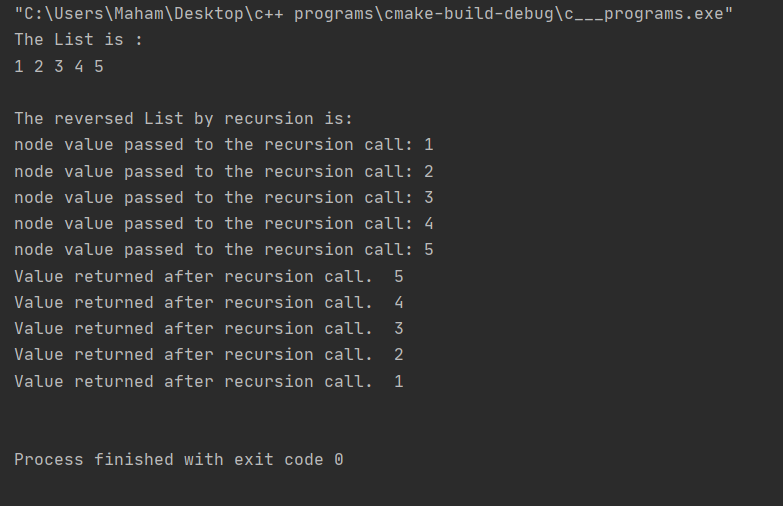


# Task 5

## Code:

|  |
| --- |
| int main()  {  RecursiveFunctions\* recursiveFunctions = new RecursiveFunctions();  SinglyLinkedList\* singlyLinkedList = new SinglyLinkedList();  for (int i = 1; i <= 5; i++)  {  singlyLinkedList->InsertNode(i);  }  cout << "The List is :" << endl;  singlyLinkedList->PrintList(singlyLinkedList->headNode);  cout << "\nThe reversed List by recursion is: " << endl;  recursiveFunctions->RecursiveReversePrintSinglyLinkedList(singlyLinkedList->headNode);  cout << endl;  return 0;  } |

## output:

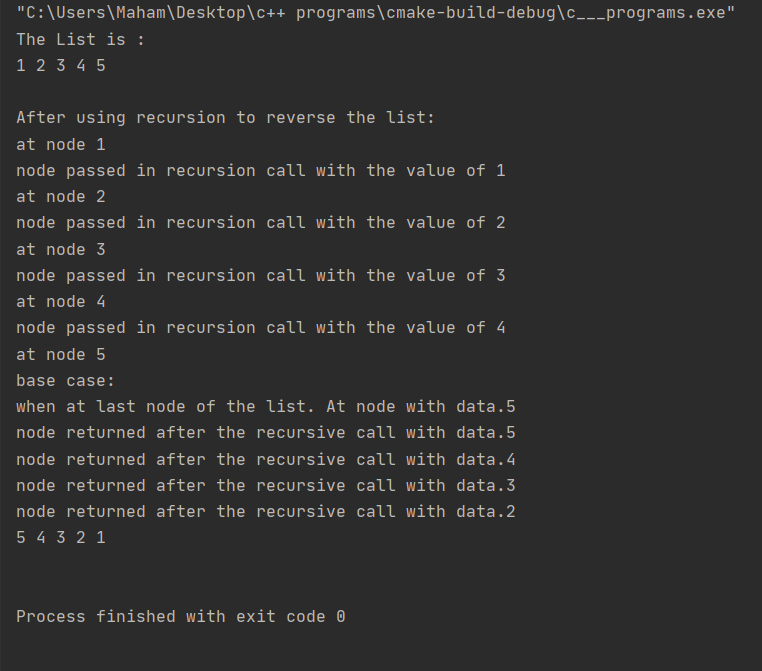


# Task 06

## Code:

|  |
| --- |
| int main()  {  RecursiveFunctions\* recursiveFunctions = new RecursiveFunctions();  SinglyLinkedList\* singlyLinkedList = new SinglyLinkedList();  for (int i = 1; i <= 5; i++)  {  singlyLinkedList->InsertNode(i);  }  cout << "The List is :" << endl;  singlyLinkedList->PrintList(singlyLinkedList->headNode);  cout << "\nAfter using recursion to reverse the list: " << endl;  ListNode\* temp = singlyLinkedList->headNode;  recursiveFunctions->RecursiveReverseSinglyLinkedList(singlyLinkedList->headNode, singlyLinkedList);  singlyLinkedList->lastNode = temp;  singlyLinkedList->PrintList(singlyLinkedList->headNode);  cout << endl;  return 0;  } |

## output:



# Task 07

## Code:

|  |
| --- |
| int main()  {  RecursiveFunctions\* recursiveFunctions = new RecursiveFunctions();  //main singly linked list  SinglyLinkedList\* singlyLinkedList = new SinglyLinkedList();  //linked list for storing the even nodes in the reverse order.  SinglyLinkedList\* singlyLinkedList1 = new SinglyLinkedList();  for (int i = 1; i <= 10; i++)  {  singlyLinkedList->InsertNode(i);  }  cout << "The List is :" << endl;  singlyLinkedList->PrintList(singlyLinkedList->headNode);  cout << "\nAfter using recursion to reverse the list: " << endl;  if (singlyLinkedList->isEmpty())  {  cout << "List is empty." << endl;  }  else  {  //if the list is not empty then the recursive function is called.  recursiveFunctions->RecursiveEvenOdd(singlyLinkedList->headNode, singlyLinkedList->headNode->next, singlyLinkedList1);  //attaching the even linked list at the start of main linked list.  singlyLinkedList1->lastNode->next = singlyLinkedList->headNode;  singlyLinkedList->headNode = singlyLinkedList1->headNode;  singlyLinkedList->PrintList(singlyLinkedList->headNode);  }  cout << endl;  return 0;  } |

## output:

