**Name: - ARKAJYOTI NASKAR**

**Roll: - 002110501144**

**Class: - BCSE-II**

**sem: - first**

**session: - 2022-2023**

**Name: - Soham Chowdhury**

**Roll: - 002110501145**

**Class: - BCSE-II**

**Sem: - first**

**session: - 2022-2023**

Assignment 1

**Problem 1:-**

# Problem Statement

Write a program to compute the factorial of an integer n iteratively and recursively. Check when there is overflow in the result and change the data types for accommodating higher values of inputs.

# Solution Approach

At first, I wrote the code to get the factorial of an integer n. To accommodate large numbers, the datatype of variables has been changed to *“unsigned long long int”*. Then, by giving inputs we are checking whether there is overflow or not by comparing that value with the real answer.

# Pseudocode

*Iterative approach :*

n = <INPUT\_FROM\_USER>

result = 1

For i = 1 to i = N :

result = result \* i

Endfor

Output(result)

*Recursive approach :*

factorial(n)

Begin

If n == 0 return 1

else return n \* factorial(n-1)

Endif

End

# Results

|  |  |
| --- | --- |
| **Data Type** | **Overflow at n =** |
| *int* | 13 |
| *unsigned int* | 13 |
| *long int* | 21 |
| *unsigned long int* | 21 |
| *long long int* | 21 |
| *unsigned long long int* | 21 |

***\*\* The result is same for both iterative method and recursive method.***

**Problem 2: -**

# Problem Statement

Write a program to generate the nth Fibonacci number iteratively and recursively. Check when there is overflow in the result and change the data types for accommodating higher values of inputs. Plot the Fibonacci number vs n graph.

# Solution Approach

At first, I wrote the code to get the nth Fibonacci number. To accommodate large numbers, the datatype of variables has been changed to *“unsigned long long int”*. Then, by giving inputs we are checking whether there is overflow or not by comparing that value with the real answer. Then we do some changes in the program to print all Fibonacci numbers and plot those data in a graph.

# Pseudocode

*Iterative approach :*

n = <INPUT\_FROM\_USER>

a = 0 b = 1

For i = 0 to i = n :

tmp = a + b

a = b

b = tmp

Endfor

**Output**(nth fibonacci number is a)

*Recursive approach :*

fibonacci(n)

Begin

If n <= 1 return n

else return fibonacci(n-1) \* fibonacci(n-2)

Endif

End

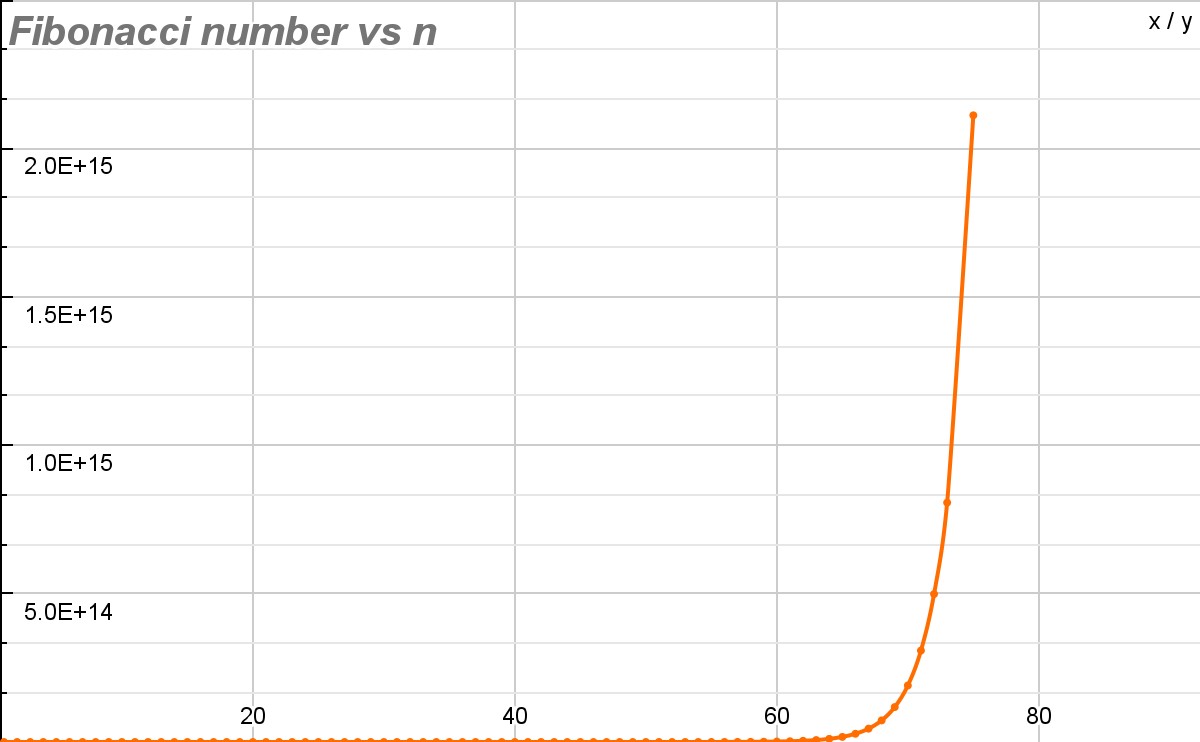
# Results: -

***By iterative method*** and using *“unsigned long long int”*, we get overflow at input 94. To calculate 93rd Fibonacci number it takes 0.000003 sec

***By recursive method*** and using *“unsigned long long int”*, we cannot determine where it actually overflows. Because as we are increasing the input the time is increasing too much. The time taken by the program for last few inputs are -

|  |  |
| --- | --- |
| Input | Time Taken [In sec] |
| 45 | 7.001000 |
| 46 | 13.023570 |
| 47 | 19. 804000 |
| 48 | 37. 818000 |
| 49 | 59. 102000 |
| 50 | 97.056203 |
| 51 | 160.024000 |
| 52 | 229.378500 |

The list of n and nth Fibonacci number has been calculated with the Fibonacci program [Iterative method]



**Problem 3:-**

# Problem Statement

Write programs for linear search and binary search for searching integers, floating point numbers and words in arrays of respective types.

# Solution Approach

In linear search we check the array elements sequentially to find our required one. Worst case *O(n)*. Binary search is only possible if the array is sorted. Here in every iteration, we divide the array in two equal parts and search for our required element in the required half. To search word we use character array instead of integer array.

# Structured Pseudocode

**Function linear\_search(\*array, size, value)** Begin

**found = 0**

for i = 0 to i = size-1:

**If array[i] = value: found = 1 break**

Endif

Endfor

**return found** End Function

**Function binary\_search(\*array, size, value)** Begin

**found = 0 low = 0 high = size - 1**

**while low <= high :**

**mid = (low + high) / 2**

If **array [mid] = value**: **found = 1 break** Endif

**Else if value > array[mid] : low = mid + 1** Endif

**Else high = mid - 1**

**return found**

Endwhile

End Function

**Limitations**

In binary search if someone gives an unsorted array as input the result will not be correct. Hence, we have to write a sort function so that the program runs for both sorted and unsorted array.

**Function sort\_array(\*array, size)** Begin

for i = 0 to i = size-1:

for j = i+1 to j = size-1:

If **array [i] > value**:

**temp = array[i]**

**array[i] = array[j]**

**array[j] = temp**

Endif

Endfor

Endfor

End Function

**Problem 4: -**

# Problem Statement

Write a program to generate 1,00,000 random integers between 1 and 1,00,000 without repetitions and store them in a file in character mode one number per line. Study and use the functions in C related to random numbers.

# Solution Approach

We take an array of length of 100000 and initialize the array elements from 1 to 100000 serially. We traverse the array from the last index and picked up an index randomly before the current index and swapped the current array element with the randomly picked index element and write the random number to file. This way we will get 100000 unique numbers from 1 to 100000 in a random order i.e. 100000 unique random number.

# Structured Pseudocode

**Function generateNumbers**(**lower**, **upper**, **filename[]**)Begin

**srand**(**time**(**NULL**))

**\*arr =** (**int\***)**calloc**(**upper+1**, **sizeof**(**int**)) **i=lower**

for temp = 0 to temp = upper-1: **arr[temp] = i, i++** Endfor

for i = upper-1 to i = 1: **r = rand**()**%i temp = arr[i]**

**arr[i] = arr[r] arr[r] = temp** Endfor

for i = upper-1 to i = 1:Write arr[i] in file

Endfor

End

# Results

It successfully writes all numbers from 1 to 1,00,000 in random\_numbers.txt without any repetition. Time taken : 0.019000s

**Problem 5: -**

# Problem Statement

Write a program to generate 1,00,000 random strings of capital letters of length 10 each, without repetitions and store them in a file in character mode one string per line.

# Solution Approach

We make a function to generate random strings. We take an 2D array of 100000 rows and 10 columns. It is used to store the random strings that have been written to file. We also take an integer variable “no\_of\_strings\_written” to hold how many strings already have been written to file. We will run a while loop until no\_of\_strings\_written becomes 100000. In each loop we will generate a random string and check in an array that the new string exists or not. If not exists we store that in an array and write in file and increase no\_of\_strings\_written by 1.

# Structured Pseudocode

random\_string(char \*str, int size ) Begin

charset[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

If size != 0 :

--size for i = 0 to i = size: key = rand() % (sizeof charset - 1) str[i] = charset[key]

Endfor

Str[size] = '\0'

Endif

End

no\_of\_strings\_written = 0 arr[100000][10\*sizeof(char)] random\_str[10\*sizeof(char)]

While n!=100000: random\_string(random\_str,100) present = 0

For i = 0 to i = no\_of\_strings\_written:

if random\_str == arr[i]:

present = 1 break

Endif Endfor

If present == 0:

Put random\_str in arr[no\_of\_strings\_written] Write random\_str in file no\_of\_strings\_written++ Endif

Endwhile

# Results

It successfully writes all numbers from 1 to 1,00,000 in random\_numbers.txt without any repetition. Time taken : 13.636000s

**Problem 6:-**

# Problem Statement

Store the names of your classmates according to roll numbers in a text file one name per line. Write a program to find out from the file, the smallest and largest names and their lengths in number of characters.

Write a function to sort the names alphabetically and store in a second file.

# Solution Approach

We will first read the total number of names in the “list\_of\_names.txt” file. Then will allocate memory to the array and store all the names from file to array. Then we will assign the first name of the list to the shortest and longest name. Then, we will iterate over the array and by comparing length with respect to shortest and longest names and will update those values and print those, at last we will sort the array and save sorted names in another file

# Pseudocode

Function **store\_name\_in\_file\_sorted**(names, n , filename):

For i = 0 to i < n:

For j = i+1 to j < n: temp = names[i],names[i] = names[j] ,names[j] = temp Endfor Endfor

outputfile = open(filename)

for i = 0 to i < n:

writeLine(outputfile, names[i]) Endfor end function

firstfile = open "list\_of\_names.txt" totalnames = 0

tmp = read(firstfile) list\_of\_names = []

While tmp != endOfFile:

if tmp == "\n":

totalnames++ Endif

tmp = read(firstfile) Endwhile

current\_line = 0

While current\_line < totalnames:

list\_of\_names[current\_line] = readLine(firstfile) Endwhile

largest\_name = list\_of\_names[0] shortest\_name list\_of\_names[0]

For i =0 to i < totalnames:

if Len list\_of\_names[i] > Len largest\_name:

largest\_name = list\_of\_names[i] Endif

if Len list\_of\_names[i] < Len shortest\_name:

shortest\_name = list\_of\_names[i] Endif Endfor

print "Largest name : " largest\_name print "Shortest name : " shortest\_name store\_name\_in\_file\_sorte(list\_of\_names,totalnames, "output.txt")

# Results :

In the terminal we get the largest name and shortest name with their length. In the output.txt file we will get all the names in sorted order.

**Problem 7: -**

# Problem Statement

Take a four-digit prime number P. Generate a series of large integers L and for each member Li compute the remainder Ri after dividing Li by P. Tabulate Li and Ri. Repeat for seven others four-digit prime numbers keeping Li fixed.

# Solution Approach

We make 2 functions. 1 For generating random numbers and another to check if the number is prime or not. We take a variable and initialize to 1000. Now we will check for the prime number and if it is prime number we will enter the loop else we will increase the variable by 1. In the loop, we will generate huge random numbers [Li] and will divide that by prime number and store the random number[Li] and the remainder [Ri] in an array. After completing 7 loops , we will print all the prime numbers and their L & R list.

# Pseudocode

Function **is\_prime**(n): for i = 2 to i <= n:

if n%i == 0 : Return 0

Endif

Endfor

Return 1 end function

P = 1000

loop = 7

While loop--:

While is\_prime(P) == 0 :

P++ Endwhile print "The 4 digit number is " + P

For i =0 to i< 100:

L[i]=RANDOM\_INT R[i]=L[i]%P print"Li "+L[i]+"Ri"+R[i]Endfor

P++; Endwhile

**Problem 8:-**

# Problem Statement

Convert your Name and Surname into large integers by juxtaposing integer ASCII codes for the alphabet.

Print the corresponding converted integer. Cut the large integers into two halves and add the two halves. Compute the remainder after dividing them by the prime numbers P in problem 7.

# Solution Approach

We will take the full name of the user by input and will store that in a variable. Then we convert all alphabets of the string one by one and store that in another array. Then we will take two strings s1 and s2 and divide the previous string in two parts and store them in s1 and s2. Then s1 and s2 will be converted to integer and will sum up [take it as S]. Then we will run a loop in search of 7 prime numbers greater than 1000. And will divide S with every prime number and print the result in the terminal.

# Structured Pseudocode

Function is\_prime(n): for i = 2 to i <= n:

if n%i == 0 : Return 0 Endif Endfor

Return 1 end function

P = 1000 loop = 7

name = input "Enter your name" len = Len(name) tmp = 0

for i = 0 to i < len:

arr[i] = STRING\_TO\_INT name[i] Endfor

s1\_len = len(arr)/2 s2\_len = len(arr) - s1\_len for i = 0 to i < s1\_len:

s1\_len[i] = arr[i] Endfor

for i = 0 to i < s2\_len:

s2\_len[i] = arr[s1\_len+i] Endfor

While loop--:

While is\_prime(P) == 0 :

P++ Endwhile

print "Prime number" + P + "Remainder" + (STRING\_TO\_INT s1 +

STRING\_TO\_INT s2)%P

P++

Endwhile

**Problem 9:-**

**Problem Statement**

 Write a program to check whether a given number is perfect or not. If it’s a perfect number add 97 to each of its digits. Now consider the result as an ASCII series. Convert the same into equivalent character sequence.

**Solution Approach**

 At first we take input from the user. It is checked whether it’s a perfect number or not. If no then the control is returned to the *main()* otherwise 97 is added to each digit using a function and the corresponding character sequence is hence generated and returned to the main() as the final output.

**Structured Pseudocode**

*isPerfect():* Checks for a perfect number by the following method🡪

for i=1 to i<n:

  if n%i==0:

  sum+=i

  endif

endfor

If the sum equals *n* then it returns 1 else 0.

*convertToArray():* We start the conversion from the end of the number since we will use % operation which extracts the last digit, then 97 is added to it to get the ASCII values and stored in an array after it has been typecasted to char explicitly. This array is then printed as the final output.

*main():* Takes input from the user and helps in executing the functions in the program.

**Results**

if the number isn't perfect it tells us and does not show the equivalent character sequence. else it just gives us the equivalent character sequence which was obtained by adding 97 to the ascii values to the characters(given by user) and then converting it back to characters.

**Problem 10:-**

**Problem Statement**

Write a program to find the maximum number of pieces that can be formed by making *N* cuts , where *N* is a positive integer.

**Solution Approach**

Let f(n) denote the maximum number of pieces that can be obtained by making n cuts. Trivially, f(0) = 1 , As there'd be only 1 piece without any cut. Similarly, f(1) = 2, Proceeding in similar fashion we can deduce the recursive nature of the function. The function can be represented recursively as : f(n) = n + f(n-1)

Hence a simple solution based on the above formula can run in O(n).

We can optimize above formula. We now know , f(n) = n + f(n-1) .

Expanding f(n-1) and so on we have , f(n) = n + n-1 + n-2 + ...... + 1 + f(0)

which gives, f(n) = (n\*(n+1))/2 + 1

**Structured Pseudocode**

*CircleHelper():*returns the Maximum number of pieces  for n cuts is when the piece is a circle.

**Function CircleHelper():**

**return (1+n\*(n+1)/2)**

**Results**

we get the maximum number of pieces by performing the given number of cuts.

**Problem 11:-**

**Problem Statement**

 Given a string and a string dictionary, write a program to find the longest string in the dictionary that can be formed by deleting certain characters of the given string. If there are more than one possible results return the longest word else return an empty string if none matches.

**Solution Approach**

 We get the lengths of the given string and then go on comparing with the words of the dictionary. If a certain character will go unmatched then that word is skipped and we again restart checking for the given string for another word from the dictionary. In this way if a certain word can be totally found then we check if that is the longest word that can be formed out of the words in the dictionary and accordingly the result is printed.

**Structured Pseudocode:**

*delete():* It is the function that settles the problem by checking the words from the dictionary and matching it with the given string(the one input by the user).

The major functioning works as follows🡪

while sp < slen && dp < dlen:

    while sp < slen && dp < dlen && s[sp] != dictionary[i][dp]:

            sp++

            end while

            sp++

            dp++

        endwhile

        if dp == dlen:

            if maxlen < dlen:

                maxlen = dlen

                maxind = i

            end if

        endif

    endwhile

    if maxind >= 0:

        return dictionary[maxind]

    endif

    return ""

Hence we get the required output.

*main():* Function that deals with the user and also provides the output to the user.

**Results**

 we get the longest string in the dictionary which can be obtained by deleting some characters from the string from the user.

**Problem 12:-**

**Problem Statement**

Given two strings(s and t) of lowercase alphabet and a value M. write a program to detect if two strings are M-anagrams of each other or not.

Two strings are called M-anagrams if the below conditions are true.

1.Both have the same number of characters.

2.Two strings can become anagram by changing at most M characters in a string. If the result is true , further find the minimum number of operations required to convert s to t.

The list of permitted operations are:

1.Insert a character at any position of the string.

2.Remove any character from the string.

3.Replace any character from the string with any other character.

4.Shift any character to it's left or right position

**Solution Approach**

 We perform the different operations using different functions. One of them checks for M-Anagrams, the other performs the 4 said operations and there is another function that finds out the minimum number of steps required.

**Structured Pseudocode**

*swap():* It helps us to to do the required swapping operation without increasing the complexity of the operational function which thus helps in smooth running of the code.

Swapping occurs as🡪

char c = s[i]

s[i] = s[j]

s[j] = c

*minOperationToConvert():* For the base cases it checks if s1 and s are having the same required length then we have to work as🡪

if i == l1:

return i-j

    endif

    else if(j == l2):

return j-i

    end elseif

Now, we intend to perform the operations as allowed by the problem.

Right shift operation is performed as🡪

If j+1<=l2  && s1[i] == s2[j+1]:

swap s1,i,i+1

a = 1 + minOperationToConvert(s1,s2,i+1,j+1,l1,l2)

swap s1,i,i+1

    endif

Left shift operation is performed as🡪

else if  i+1<=l1  && s1[i+1] == s2[j]:

swap s1,j,j+1

b = 1 + minOperationToConvert(s1,s2,i+1,j+1,l1,l2)

swap s1,j,j+1

    end elseif

Deletion operation🡪

c  = 1 + minOperationToConvert(s1,s2,i+1,j,l1,l2)

Insertion operation🡪

d  = 1 + minOperationToConvert(s1,s2,i,j+1,l1,l2)

Replacement operation🡪

e  = 1 + minOperationToConvert(s1,s2,i+1,j+1,l1,l2)

Finally, the minimum number of operations required will always be the operations needed the least number of times. Hence we can make use of min() function to get the required ans as🡪

int ans = min of a,b,c,d,e.

is\_main\_anagram(): In this function we get the length of the strings s1 and s2. Then we have two arrays that stores the frequency of characters in the two strings. Now we try to find the number of different characters present in the two strings such that if🡪

if diff <= M:

        return 1

else return 0.

Thus we get M-Anagram checked.

*main():* Takes in the two input from the user s1,s2 and also the value of M required for checking M-Anagrams. Now if we can reach the desired solution it is printed otherwise we output it is not M-Anagram.

**Results:**

returns true depending on if the given strings are M anagrams of each other or not, if they are M anagrams of each other then the number of operations required to convert one string to another is given.