ASSIGNMENT SET I

Data Structures & Algorithms Lab

001910501090
RITABROTO GANGULY

First Semester
BCSE II Session 2020-21

Problem No.: 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:

Define functions for finding factorial of user input number. Try in both iterative and recursive methods

Structured Pseudocode:

Iterative:

```
fact(n):
    result = 1
while n!=0
    do
        result = result * n
n-- done
```

```
    return result
```

Recursive:

```
fact(n)
  if(n<=2)
    return n

return n*fact(n-1)
```

Results:

It is apparent that uint is capable of storing 10 digit numbers. So, results upto 12! are fine (12! is expected to be 9 digit as 12 and 11 ≈ 10 and results are growing by 1 digit). So overflow either occurs at n=13 or n=14 (number of digits do not increase at 14). As last two digits of 12! are 0, it should be so for 13!. So overflow occurs at 13!.

As for ulong, similar procedure if followed to find n at which overflow occurs. 20 digit numbers can be clearly stored in ulong. So 19! is correct as it is expected to be less than 20 digits. 20! can be quickly checked with eye and it is found to be correct. Last 4 digits of 20! are 0 but for n=21, the result does not have the same. So the overflow occurs at n=21.

Discussion:

uint or unsigned int is 32 bits long and is capable of storing integers 0 to 2^{32} -1. So it can bu used to correctly compute and store factorials of n only if $n! < 2^{32}$. Taking log on both sides with base 2, as 2 > 1, we have $\log_2(n!)$

< 32. From calculator, we have $\log_2(12!) \approx 28.835455234$ and $\log_2(13!) \approx 32.535894952$. Hence the uint variable was not able to store value of 13! correctly.

The above screenshots show that 12! can be stored in 32 bits while 13! overflows to the 33rd bit.

Similarly for ulong instead of 32, we have 64 bits and hence $n! < 2^{64}$. $\log_2(20!) \approx 61.077383921$ and $\log_2(21!) \approx 65.469701344$.

Files with commented source code:

1.c

Problem No.: 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:

Define functions for finding fibonacci of user input number. Try in both iterative and recursive methods

Structured Pseudocode:

Iterative:

```
fiboiter(n):
    if n<2
return n fi
    x0 = 0, x1=1
    for i = 0 to n-2
    do
```

```
x1 = x1 + x0
        x0 = x1 - x0
    done
return x1
```

Recursive:

```
fiborec(n):
    if n<2
return n else
        return fiborec(n-2) + fiborec(n-1)
    fi
```

Results:

As a fibonacci term is computed by adding previous two terms in the sequence, with the exception of term 1(1) and term 2(2), all terms < 2*previous term as the sequence is monotonic in nature. Upto 47, the result of the program increases as expected. 47^{th} fibonacci number is calculated to be 2971215073 which is 10 digited. Hence the next number is also expected to be 10 digited (as next term cannot be more than double and the sequence is monotonically increasing). But 48^{th} fibonacci number is calculated to be 9 digited by the program. Hence overflow must have occurred at n=48.

For ulong, slightly different approach is taken. First, the call to recursive function is commented because the number of calls is also a fibonacci sequence and grows roughly geometrically. First the csv is made and an estimate is taken of the point where overflow might have occured. From the estimate it is seen that number of digits gradually increase till 93 but at n=94, number of digits decreases. Hence overflow occurs at n=94 or n=93. We can easily see that n=93 has given correct answer from above picture. Hence overflow occurs at n=94.

Discussion:

As fibonacci term is calculated as *f(x) = f(x-2) + f(x-1) [base case: f(1)=1 f(0)=0]* in function fiborec(n), if number of numction calls is function (say g) of x, then *g(x) = g(x-2) + g(x-1) + 1 [base case: g(0)=g(1)=1]*. Hence number of function calls required to calculate nth fibonacci sequence recursively is greater than the result itself. The call stack (above main) will get to height x ∀ x>0 and 1 for x = 0. On the other hand, in iterative approach, number of function calls remains constant (1) and hence the call stack can only get to height 1.

Even in the field of operations and comparisions, in recursive approach, at least g(x) comparisions are done to check if x is a base case or not. Apart from that, if h(x) is the number of operations required to calculate f(x), then *h(x) = 3 + h(x-2) + h(x-1) [base case: h(0)=h(1)=0]*. (3 because of 'n-1', 'n-2' and '+' in the return statement). However, in Iterative approach, there is 1 comparision and no operation for x<2 and n+1 comparisions, 2n assignments and 2n-2 operations.

Hence in all respects, the iterative approach is better here than the recursive one.

Files with commented source code:

2.c

Problem No.: 3
Problem Statement: Write programs for linear search and binary search for searching integers, floating point numbers and words in arrays of string types.

Solution Approach:

Macro functions for integer and floating point types, to make use of generic programming and a different function for char* type

Structured Pseudocode:

```
linsearch(arr[d],search_value):
    for i = 0 to d-1
    do
        if arr[i] = search_value
return i fi
    done
    return -1
sort(arr[d]):
    for i = 0 to d-2
do
pos = i
        for j = i+1 to d-1
        do
            if arr[j] < arr[pos]
                pos = j
fi done
        swap arr[pos], arr[j]
    done
```

```
binsearch(arr[d],search_value):

    l = 0, u = d-1

while l ≤ u

do
m = (l+u)/2

        if arr[m] = search_value

            return m

        elif arr[m] < search_value

            l = m+1

else
u = m-1

fi done

return -1
```

Results:

As seen all 6 of the programs correctly find search values.

To see if they really work, I prepared an input file with help of random words and numbers generated by q4 and q5. As sorting is not really part of binary search, the input files were sorted and the call to sort function was commented.

First, the input file was prepared so that the search value would be found.

There is barely any time difference in case of integers but there is significant time difference in case of words.

Now, the programs were tested with search values that were not to be found. There is significant difference in execution time in case of binary and linear search for words.

Discussion:

As above pictures show, binary search is more efficient than linear search. Algorithm for linear search, as the name suggests, has time complexity O(n) while algorithm for binary search has time complexity O(log n) where n is the array's length.

In linear search, the array is traversed from from start to end and each element is compared with search value. If a match is found, traversal is discontinued. Hence in best case (1^{st} element is the match), time complexity is O(1). In worst case, there is no match or the match is at last cell. Then the complexity is O(n) and n comparisions have to be performed. In average case, only half the array needs to be traversed and the complexity is still O(n) but only $\lceil n/2 \rceil$ comparisions have to be done.

In binary search, the array has to be sorted (without loss of generality, assumed ascending). The median element is picked and compared with the search value. If match is found, then search is stopped. If the median is greater (or lesser) than search value, binary search is performed on a sub array left (or right) of the median. In practice, lower and upper bound of indices of the current array or subarray are maintained. If the lower bound becomes more than upper bound, it means that the subarray in question has no valid indices and the search value has no match. As search progresses, with each the searching area becomes half of the previous one. Hence an array of n cells can be searched in $\lceil \log_2 n \rceil + 1$ steps. Hence the worst case complexity of this algorithm is O(log n). The best case, match in middle of the array, complexity is O(1). On average $\lceil (\log_2 n + 1)/2 \rceil$ steps have to be taken but the complexity remains O(log n).

However, while considering linear and binary search we must remember that in general, arrays to be searched are not sorted and binary search hence will place an overhead which can be avoided with linear searches. Even the fastest sorting algorithms have time complexity O(n*log n) in average cases which is worse than that of linear search.

Files with commented source code:

3.c

Problem No.: 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:

Generating 1,00,000 integers without between 1 and 1,00,000 will mean that each number in the domain gets generated exactly once. Hence generation need not be done. We only need to place the numbers in random order.

rand()%n will give an integer between 0 and n-1 including them in C.A domain of all possible values is prepared and an element is chosen with the method above with n as length of domain. Then the element is written and is deleted from the domain. This is done repeatedly till domain is empty.

Structured Pseudocode:

```
for i = 0 to 99,999
do
    domain[i] = i+1
done
for i = 1,00,000 to 1
do
    pos = random number between 0 and i (but not i)
    write domain[pos] to output file
    for j = pos+1 to i-1
    do
        domain[j-1] = domain[j]
    done
done
```

Results:

The program takes between 4.8 and 4.9 seconds to execute. 11

As seen in the picture above, 1,00,000 lines have been written to the output file.

Discussion:

In above algorithm, calculation of random position and writing of corresponding element are constant time tasks. Put in a loop with number of iterations linearly dependent on number of integers to be generated, the complexity of the algorithm would have been O(n) with n corresponding to number of integers to be generated.

But due to shifting of elements to reduce domain size, the complexity may worsen. In best case, last element is always selected to be written. In such case, only one shift needs to be performed in each iteration of outer loop. Hence complexity remains O(n). In worst case, first element is always selected to be written. Hence $\Sigma(i-1)$ for $i = 1$ to n shifts have to be perdormed. Hence complexity becomes $O(n^2)$. For average case, middle element is selected and $\Sigma\lfloor(i-1)/2\rfloor$ for $i = 1$ to n shifts have to be performed which decreases number of shifts to half but the complexity remains $O(n^2)$.

Other important part of this problem is to generate random numbers. In C, rand() used some algorithm on a number called seed to generate other number. These numbers are not truly random but only pseudorandom. If same seed is used each time, rand() will generate same sequence of numbers everytime. With srand(uint), we can set the seed for rand(). If this number is set to some constant value, or not set at all, in which it is 1 by default, whenever we run the program, same "random" numbers will be generated in the same order and thus in the case of this problem, same output file will be made.

The solution to this is to set the seed with some variable value. This can be done through user input. Other way is to set the seed with help of the time in real world. The time(time_t*) function in time.h helps us get number of seconds that have passed since 12 AM GMT 1st January, 1970. If we use

this value to set seed, we will get different sequences of numbers from rand(). Though still not random, this system is better than the previous one.

Files with commented source code:

4.c

Problem No.: 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:

Here the domain of possible combinations is 26^{10} which is far greater than 10^5 which was the number for integers between 1 and 1,00,000. $48 > \log_2(26^{10}) > 47$ so it is not possible to make a domain of all possible combinations practically. So, here a random 10 block lettered string is generated and it is checked if it has already been generated or not. If a match is found, the string is discarded and new string is created and the process is repeated till we get 1,00,000 unique strings.

Structured Pseudocode:

```
to_write[1,00,000]

for i = 0 to 99,999

do
do

flag = false

curr = generate_random_string()

for j = 0 to i-1

do

    flag = is_equal(curr,to_write[j])

    if flag = true
```

```
break fi
            done
        until flag = false
        to_write[i] = curr
done
for i = 0 to 99,999
do
        write to_write[i] to output file
done
is_equal(str1,str2):
        for i = 0 to 9
        do
            if str1[i]!=str2[i]
                return false
            fi
        done
        return true
generate_random_string():
        str = “”
for i = 0 to 9 do
            str[i] = random letter between ‘A’ and ‘Z’
        done
return str
```

Results:

The program takes well above 15 seconds to execute. This is significantly more than what was required by q4.

As seen above, the output file is generated properly.

Discussion:

The functions for generating random string and checking for equality take constant time and thus have time complexity O(1). The process of checking whether the genereated string is in the array of already generated string is a linear search with average time complexity O(n) where n is the number of strings we already have.

As this checking has to be done for each newly generated string, the overall time complexity becomes $O(n^2)$.

In this problem, another approach can be taken which is similar to that of q4. We can make a domain of 1,00,000 strings and then exhaust the domain bu writing its elements one by one. Moreover, if we somehow make the domain sorted, complexity for checking uniqueness of newly generated string will be reduced to O(log n) from O(n).

However if we generate unique strings ang then add them to the domain at correct position, adding it will have complexity O(n) with insertion sort like method and the overall complexity will remain $O(n^2)$. Moreover, due to so many operations due to the colossal size of the array, the time taken will also be very large.

Another way out is to generate strings roughly in order. We can generate strings such that of 1,00,000 generated strigs, first 5,000 have their first letter ‘A’, next 5,000 have their first letter ‘B’ and so on. Though there will be no improvement in time complexity, the number of comparisions and shifts need to be done will be reduced 20 folds (with these particular numbers). And as we are randomly chosing elements from the domain, this order will not be reflected in the output file.

Files with commented source code:

5.c

Problem No.: 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:

This is a simple file operation problem. First each roll and name is read from given file and stored in an array. Then array is sorted according to roll numbers. The Sorted array is written into a file. The shortest and longest name are found while writing to the output file and are displayed.

Structured Pseudocode:

```
struct students{roll; name;};

int[] smallest(students stds[],int count){//finding the smallest name(s)

  int name_indexes[90]

  int index

  int min = 101

  for(int i=0;i<count;i++)

    int x = length of stds[i].name

    if(x<min) //if name length is smaller than the current smallest

      index = 1 //come back to the 1st index

      min = x

      name_indexes[index++] = i

    else if(x==min)//if name length is equal to the current smallest

      name_indexes[index++] = I //add behind the current name in the array

  For loop done

  name_indexes[0] = index //store array length in 0th index
```

```
  return name_indexes
}

int[] largest(students stds[],int count){//finding the largest name(s)
  int name_indexes[90]
  int index
  int max = -1
  for(int i=0;i<count;i++)
     int x = length of stds[i].name
     if(x>max) //if name length is larger than the current largest{
       index = 1 //come back to the 1st index
       max = x
       name_indexes[index++] = i
     else if(x==max) //if name length is equal to the current largest
       name_indexes[index++] = i //add behind the current name in the array
 For loop done
  name_indexes[0] = index //store array length in 0th index
  return name_indexes
}

main(){
 students stds[90]
 Open file to read from with pointer ptr
 Open file to write into with pointer ptr2
```

```
 int index = 0
 while(index<90 && !feof(ptr)) //read the file and store in array
     stds[index].roll = roll
     stds[index++].name = name
While loop done
 sort(stds) // sort stds according to roll
 int s[] = smallest(stds,index) //smallest names
 int l[] = largest(stds,index) //largest names
}
```

Results:

The program takes less than 0.01 second to execute. It can be seen that no record was left out or was duplicated.

Discussion:

Taking number of records as n, reading of reacords with arranging them in a list together has complexity $O(n^2)$. Writing sorted records has linear complexity as writing a record is a constant time task.

One should be careful while handling files in C. The format string used to read a line should match the lines in file exactly. While writing also only one format should be used. Otherwise it becomes difficult to read.

Files with commented source code:

6.c

Problem No.: 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 other four digit prime numbers keeping Li fixed.

Solution Approach:

ulong (64-bit) is used to store large integers. As rand() only generates a 31 bit number (the MSB is always 0), it has to be used twice to get actually long numbers. P is taken as a array of uint of length 8 and initialised with pre-known values. Remainders are calculated and a csv file is made.

Structured Pseudocode:

```
L[n]
integer pointer x
x = addr(L[0])
for i = 0 to 2n-1
do
    *x = random integer
x++ done
open output_csv_file as out
write header to out
for i = 0 to n-1
do
    write L[i] to out
for j = 0 to 7 do
        write ‘,’ to out
        write L[i]%P[j] to out
    done
done
```

Results:

The program takes nearly 0.1 second when working with 1,00,000 long series. The csv file has 1,00,001 lines (1 line for header) and is about 5.5 MB in size. If length of series is increase 10 folds, size of output file also increases 10 folds. However, when length of series is increased from 10^6 to 10^7, the execution time increases 20 folds.

Discussion:

Generation of random large integers has time complexity O(n). The loop for calculating remainders of L[i] with elements of P iterates exaclty 8 times for each L[i]. Hence it has complexity O(1). Hence calculation of remainders and writing of the results also has linear complexity O(n) where n is the length of the series L. Hence overall time complexity is also O(n).

The space complexity is also linear (as is evident from previous pictures). If binary file had been used or the lines had been formatted character by character, the space complexity would be perfectly linear.

Files with commented source code:

7.c

Problem No.: 8

Problem Statement: Convert your Name and Surname into large integers by juxtaposing integer ASCII codes for alphabet. Print the corresponding converted integer. Cut the large integers into two halves and add the two halves. Compute the remainder after dividing the by the prime numbers P in problem 7.

Solution Approach:

A character array can represent a number. Fill arrays with digits of ASCII values of characters. After accepting the name and surname, start filling corresponding numbers from least significant to most significant digit with help of ASCII values of the characters. Find mid point of the number with help of len. Make new numbers (halves of original).

Addition is done by adding individual digits of two numbers from least significant to most significant digit while keeping track of carry forward. Remainder is calculated by carrying down digits from most to least significant digit and finding remainder of carried down value. The addition and division algorithms resemble the way we humans do them by hand.

Structured Pseudocode:

```
int temp_index = 0

void fill_temp(int temp[], int num){//helper function to fill up temp array in reverse of num

 temp_index = 0

 while(num>0)

    int d = num%10

    temp[temp_index++] = d

    num/=10

 While loop done

}


void print(int a[],int size){//print number

 int begin = 0

 if(a[0]==0)

   begin = 1

 for(int i=begin;i<size;i++)

    print(a[i])
```

```
 print("\n")
}
void add(int a[], int b[], int size, int c[]){//addition on the numbers stored digit by digit in arrays
 int carry = 0
 for(int i = size-1;i>=0;i--)
    int sum = a[i] + b[i] + carry
    if(sum<10)
      c[i+1] = sum
      carry = 0
    else
      c[i+1] = sum%10
      carry = 1
 For loop done
 c[0] = carry
}

void divide(int s[],int size){//division on the numbers stored digit by digit in arrays
 int begin = 0
 if(s[0]==0)
   begin = 1
 int dividend = 0
 while(begin<size)
      dividend = dividend*10+s[begin]
      if(dividend>=prime)
        dividend = dividend % prime
```

```
      begin++
While loop done


 print(dividend);
}


main(){
Input a number a char array c
 int size = length of c
 int nums[3*size]
 int j = 0
 for(int i=0;i<size;i++)
    int num = c[i]
    if(num!=32) //if char is not a space
     int temp_size = 2 //digits in ascii value may be 2
     if(num>99) if more than 99
       temp_size = 3 //digits in ascii value is 3
       int temp[temp_size]
       fill_temp(temp,num) //stores digits of num in temp array
     fi
       while(temp_index>0)
           nums[j++] = temp[—temp_index] //store from temp to nums in reverse
 for loop done

 int half1_size = j/2 //actual size of half1
 int half2_size = j - half1_size
```

```
int half1[half2_size] //half2_size used for uniformity
int half2[half2_size]
int diff = half2_size - half1_size
for(int x = 0;x<half2_size;x++) //fill half1 with first half of nums
  if(x<diff)
    half1[x] = 0 //fill the front with 0s if half1_size is less than half2_size
  else
    half1[x] = nums[x-diff]

for(int y = 0;y<half2_size;y++) //fill half2 with second half of nums
  half2[y] = nums[half1_size + y]

Do addition, division and print operations
}
```

Results:

Enter a name for converion to number: rg dgr8
name-->number = 1141031001031145 6

first half = 11410310

second half = 10311456

sum = 021721766

Required remainder = 14

The outputs are thus correct.

Discussion:

Time Complexity

O(n)

Files with commented source code:

8.c