**ADS 1 - Assignment 1 – Samuel Haque**

**Q1)**

1. The first question asked me to produce an algorithm to calculate the number of combinations when no repetition is allowed. Using the website provided in the footnote I realised to calculate the right values I needed to produce an algorithm that solved the equation:

n!

r! (n-r)!

Number of combinations when no repetition is allowed =

Now this equation made me realised that I needed to show myself a diagram that would help me visualise the inputs and what outputs should be return. Therefore I decided to draw a black box that looked like this (with “n” being the number of elements to choose from and “r” being the elements that are chosen):

OUT

IN

n!

r! (n-r)!

n

r

Calculate number of combinations when no repetition is allowed.

The black box allowed me to visualise what the user would input and what outputs should be produced however this also made me realise that to create this algorithm I would need to break down the black box above into smaller steps using stepwise refinement. Therefore I came to the conclusion of drawing the diagram below before I started to produce a suitable algorithm:

Calculate (n-r)!

Calculate n-r

Calculate n!

Calculate

n!

r! (n-r)!

Calculate number of combinations when no repetition is allowed.

Breaking the problem into steps made it easier for me to write a suitable algorithm and thus I produced this function that can be used to calculate the equation:

function calcNumbCombiNoRep (IN n, IN r)

define NMinusR, NMinusRFact, FactR, combiNoRep

n factorial(n)

NMinusR n – r

NMinusRFact factorial(NMinusRFact)

FactR factorial(r)

combiNoRep n div (FactR \* NMinusR)

return combiNoRep

end function

The function above starts with figuring out the factorial of n then calculates the factorial of n-r. Then calculates the factorial of r and puts all of these factorials into the equation: Finally the answer is returned.

n!

r! (n-r)!

1. In this question it I must use a similar method however instead the algorithm must calculate the number of combinations when repetition is allowed. This requires a different equation to be used:

(r+n-1)!

r! (n-1)!

Number of combinations when no repetition is allowed =

So to create an algorithm I first needed to create a black box to determine what inputs and outputs my algorithm should have. Here is the black box that I created:

IN

OUT

r

n

Calculate number of combinations when no repetition is allowed.

(r+n-1)!

r! (n-1)!

This black box helped me visualise the problem however it does not help me see the steps that need to be taken in order to produce the correct output. To show the individual steps that need to be taken I used step wise refinement and represented it in the diagram below:

Calculate

(r+n-1)!

r! (n-1)!

Calculate r+n-1

Calculate r!

Calculate (r+n-1)!

Calculate (n-1)!

Calculate number of combinations when no repetition is allowed.

Now after being able to see the steps required visually I formed this function below:

function calcNumbCombiWithRep (IN n, IN r)

define rAddNMinus1, factRAddNMinus1, factR, factNMinus1, combiWithRep

rAddNMinus1 (r+n)-1

factRAddNMinus1 factorial(rAddNMinus1)

factR factorial(r)

factNMinus1 factorial(n-1)

combiWithRep factRAddNMinus1 div factR \* factNMinus1

return combiWithRep

end function

The function produced first calculates the factorial of (r+n)-1, then calculates the factorial of r, and then calculates the factorial of n-1. Finally it puts these values onto the equation: and returns the answer produced.

(r+n-1)!

r! (n-1)!

**Q2)**

1. In this question I am asked to first find all the multiples of 3 and all the multiples of 5 under 1000 and then add them together. So first I need to determine the inputs required and what I should expect to be returned as an output. To display this I created a black box again. This is the black box I produced:

OUT

IN

Calculate the sum of the multiples of 3 and 5 under 1000

Sum of the multiples

size

After drawing up a black box I realised I must break this down into steps so I produced the following diagram below:

Calculate the sum of the multiples of 3 and 5 under 1000

If value is divisible by 3 or 5 add it to the sum of the other values that are divisible by 3 or 5

Check if the value divides by 5

Check if the value divides by 3

The diagram above allows me to see how I should create the algorithm with step wise refinement and therefore it puts me in the position to be able create the algorithm. This is done below:

function calcSumOfMultiples (IN size)

define sumOfMultiples, i

i 0

sumOfMultiples 0

for i to size do

if i mod 3 = 0 or i mod 5 = 0

then sumOfMultiples i + sumOfMultiples

end if

i i+1

end for

return sumOfMultiples

end function

Here I used a for loop to check if each number up to the number entered by the user is divisible by 3 or 5. To do this I used modulus to divide the number by 3or 5. This would then look at the remainder once it is divided and if the remainder is equal to zero it means that the number is a multiple of 3 or 5, as any multiple of 3 can be divided by 3 with no remainders, the same applies for multiples of 5.

If the value that is being checked is a multiple, it is added to the sum of the previous multiples. This check is done until the all values are checked.

1. This next problem is asking me to create an algorithm that find the factorial of a number then adds the digits within the factorial together. This I believe may be broken down into two black boxes as it would be simpler to understand and write an algorithm for.

The first black box would be used to figure out the factorial of the number entered and would look like this (n being the number entered into the algorithm):

OUT

IN

n!

n

Calculate the factorial of the number entered

The next black box would use the output from the black box above and add the individual digits of the factorial together The black box for this will look like this:

OUT

IN

Calculate the sum of the digits within the factorial

Sum of the digits within n!

n!

Now the inputs and outputs that are expected to be used and produced are shown, I need to break the black boxes into steps. The first black box is pretty simple as there is only really one step. Therefore the steps will look like this:

Calculate the factorial of the number entered.

Calculate n!

The second black box will have a few more steps to include within the algorithm. These are the steps that will need to be taken in order to produce the right algorithm:

Calculate the sum of the digits within the factorial

Find the unit digit of n!

Make the factorial smaller by 1 digit

Do this until all of the digits are added together

Add this unit digit to the sum of the rest of the digits

Now the algorithm can be written up. I produced the following algorithm:

function calcFactorial (IN n)

define factOfN

factOfN factorial(n)

end function

The function above calculates the factorial of n and sets it to a variable that could be used in the next function.

function addDigits (IN factofN)

define sumOfFact, digit

while factOfN>0

Digit = factOfN mod 10

sumOfFact = sumOfFact + Digit

factOfN = factOfN div 10

end while

return sumOfFact

end function

The function above uses a while loop that calculates the modulus of the factorial of n. This would divide n by to and sets the “digits” variable to the unit value. For example 345 would be divided by 10 to be 34.5 and then the “digits” variable is to 5. This value will be added to the sum of all of the other digits and then the original “factOfN” variable is divided by 10. For example 345 would become 34. This is then entered into the while loop again until the “factOfN” variable becomes less than 0. At that point it is not entered into the loop. For example 0.3 would not be entered into the while loop. Finally the total sum of the digits within the factorial is returned.

**Q3)**

For this problem I needed to produce black boxes once again to make sure the correct inputs are being entered and the correct outputs are produced. The black boxes used are shown below:

OUT

IN

Calculate the total amount of numbers that are prime and within the array.

Total number of prime numbers within the array.

Numbers[]

size

OUT

IN

True/ false depending on if the number is a prime or not

potentialPrime

Check if the number is prime

The second black box will be used as an abstraction within the first function.

The black boxes showed me that two functions will be used as an abstraction but for the correct algorithm to be created by reducing the mistakes I must create diagrams that will show the steps that will be taken to produce the correct algorithm. These diagrams are shown below:

If they are prime then they will add to the total sum of prime numbers.

Check each number using the second function to see whether they are prime or not

Calculate the total amount of numbers that are prime and within the array.

Divide the number in the array by every number between 1 and one number less than its self

Find the modulus of these calculations. If the number is prime there should be a value being returned. However if there isn’t a value after the modulus is figured out then the number has completely been divided by another number and therefore isn’t prime.

Check if the number is prime

Now the steps have been refined, they can be implemented into algorithms. Previously I mentioned that two functions will be used and abstraction will take place this can be seen below:

function totalNumOfPrimes (IN numbers[],IN size )

declare isPrime , potentialPrime, i

for i to size by 1 do

potentialPrime numbers[i]

isPrime isAPrimeNumber(potentialPrime)

if (isPrime = true)

then

primeNumCounter primeNumCounter+1

end if

return the primeNumCounter

end for

The first function is the function that calculates the total number of primes within the array. This is done by using an abstraction using the function below to check if the number is a prime or not. The function above uses an if statement to check if the value within the array is prime. If so then it is added to a counter that adds up all the prime numbers that are found within the array. Finally the counter of all of the prime numbers within the array is returned

function isAPrimeNumber (IN potentialPrime)

declare potentialPrime, i

potentialPrime true

i 1

for i to potentialPrime-1 by 1 do

if (potentialPrime mod i = 0) then

return false

end if

end for

end function

This function above is used as abstraction in the previous function. It does this via a for loop and a nested if statement. The if statement states that if the value within the array is divisible by the number in the counter and returns no remainders then the it is not a prime number. The for loop is used so that the value within the array is divided by every number from 1 to 1 number below the value of the number within the array.