Algorithms and Data Structures I – Assignment 1

1. Permutations
2. To start tackling the problem of calculating the number of permutations when repetition is allowed I decided to treat the whole problem as a black box so I could figure what are the inputs and outputs of my algorithm.



Once I had figured out what the inputs and outputs of my algorithm were I then researched how to calculate the number permutations when repletion is allowed and found that the following formula gave the number of permutations:

With this formula in mind, I then broke down my black box using a stepwise refinement tree.



Thinking about my stepwise refinement tree I realised the problem was quite simple and therefore I went on to write my ADL algorithm solution to the problem.

Procedure CalcPermutationsRepAllowed(IN n, IN r, OUT p)

p ← n\*\*r

end

1. As with the first problem I started by looking at the problem as a black box so I could determine the inputs and outputs for my algorithm.



With the parameters of my algorithm sorted I then researched how to calculate the number of permutations were repetition is not allowed. From my research, I found that the number of permutations could be calculated using the following formula:

Using the formula, I then broke down my initial black box into induvial steps using a stepwise refinement tree.



From the stepwise refinement tree, I decided to treat to the calculation of the factorials as a separate abstraction and therefore decided to produce another black box diagram to figure out the required parameters.



Knowing the parameters for the calculate factorial procedure I went on to write the ADL algorithm for calculating the factorial of a given number.

Procedure Factorial(IN number, OUT numberFactorial)

numberFactorial ← 1

For count ← 2 to number by 1 do

numberFactorial ← numberFactorial \* count

end

end

With my procedure for calculating the factorial I could then go on to write the ADL algorithm for calculating the number of permutations.

Procedure CalcPermutationsRepNotAllowed(IN n, IN r, OUT p)

Declare n\_fact, n\_minus\_r\_fact

Call Factorial(n,n\_fact)

Call Factorial((n-r),n\_minus\_r\_fact)

P ← n\_fact / n\_minus\_r\_fact

End

1. Data Structure Design and Manipulation

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Student Name** |  | **Telephone number** |  | **Father** |  | **Mother** |
| Adams, John |  | 01142253175 |  | Richard |  | Mary |
| Bailey, Susan |  | 01142256900 |  | Steven |  | Sheila |
| Clark, Bruce |  | 0161248653 |  | XXXX |  | Barbara |
| ….. |  | ….. |  | ….. |  | ….. |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sibling 1** |  | **Sibling 2** |  | **Sibling 3** |  | **Sibling 4** |
| Jane |  | Willam |  | Donald |  | XXXX |
| XXXX |  | XXXX |  | XXXX |  | XXXX |
| David |  | Lisa |  | XXXX |  | XXXX |
| ….. |  | ….. |  | ….. |  | ….. |

Each table above represents a single dimension array, going from 1 to a chosen size, where the index of an element in any of the arrays can be mapped to the corresponding elements in the other arrays. For example, if a student’s name is at position 1 of the student name array their telephone number, father, mother, and siblings will also be at position 1 in their corresponding arrays. Due to the correlation between each array the size of each array must be the same, for example if the student array has 200 positions available all the of other arrays must also have 200 positions available. One limitation of this method of implementation is the maximum number of siblings is fixed depending on how many individual arrays are dedicated to each sibling, in this case I chose 4 arrays as this seemed like a sensible upper bound as it is unlikely to have more than 5 children from the same family going to the same school at the same time.

1. To start solving the problem I decide to treat the entire solution as a black box so I could determine the inputs and outputs of my algorithm.



Whilst deriving my black box diagram for this problem I concluded that my algorithm would not necessarily have any outputs as the printing of the sibling names to the screen is something that can be handle internally by my algorithm. With the parameters for my algorithm sorted I broke the problem down into separate parts using a stepwise refinement tree.



From my stepwise refinement tree, I decided to treat the finding of the position of the student’s name as a separate abstraction and therefore produced another black box diagram to analyse its parameters.



With the parameters of the abstraction determined and thinking further about how I might write this algorithm I concluded that I was ready to produce an ADL procedure for this abstraction.

Procedure FindPositionOfStudent(IN name, IN array[], IN size, OUT position)  
   
 declare found

Position ← 0

found ← false

while (found = false and position ≤ size)

position ← position + 1

if array(position) = name then

found ← true

end

end

if found = false then

position ← 0

end

end

The above ADL procedure will return the position of a given student’s name in the Student Name array or will return 0 if the student’s name was not found.

With my search algorithm complete, I concluded that print of the sibling names given their position was a simple task and therefore I decide I was ready to write my ADL solution to the problem.

Procedure PrintSiblings(IN name, IN students[], IN siblings1[], IN siblings2[], IN siblings3[], IN siblings4[], IN size)

declare position

call FindPositionOfStudents(name,students,size,position)

if position = 0 then

print(“The student was not found”)

else

print(siblings1(position))

print(siblings2(position))

print(siblings3(position))

print(siblings4(position))  
end

end

1. One-Dimensional Array Problems
2. As with all the previous problems I started by treating the entire solution as a black box so I could determine the inputs and outputs of my solution.



Knowing the parameters of my algorithm I then went on to break the problem down into smaller parts using a stepwise refinement tree.



Whilst I was creating the stepwise refinement tree I realised the problem could be tackled by making a new array containing all the occurrences of unique numbers as opposed to removing the duplicates.

From my stepwise refinement tree the problem appeared rather complexed and therefore I decide to treat the scanning for unique numbers an abstraction and therefore I produced a black box diagram to decide its parameters.



With the parameters of my abstraction decided upon I went ahead and wrote the ADL procedure for this abstraction.

Procedure GetUniqueNumbers(IN numbers[], IN size, OUT uniqueCount, OUT uniqueArray)

uniqueCount ← 0

for index ← 1 to (size – 1) by 1 do

if (numbers(index) ≠ numbers(index + 1) then

uniqueCount ← uniqueCount + 1

uniqueArray(uniqueCount) ← numbers(index)

end

end

if numbers(size) ≠ numbers(size -1) then

uniqueCount ← uniqueCount + 1

uniqueArray(uniqueCount) ← numbers(size)

end

end

The if statement after the for loop in the above ADL procedure is used to check if the final value in the array of numbers is unique as this case cannot be check by the for loop as this will result in the index being outside the boundary of the array.

After producing the abstraction for separating the unique numbers I decided the task of putting the unique numbers into a more compact array was a simple enough task to not require another abstraction and therefore I was ready to produce my ADL solution. However, whilst creating my initial black box for this problem I felt that unlike the other problems this one should be a function as it will return an array of a smaller size to the one that was inputted.

Function RemoveDuplicates(IN numbers[], IN size)

Declare uniqueArray[1..size], uniqueCount  
Call GetUniqueNumbers(numbers,size,uniqueCount,uniqueArray)

Declare uniqueArraySmall[1..uniqueCount]  
  
For index ← 1 to uniqueCount by 1 do  
 uniqueArraySmall(index) ← uniqueArray(index)  
end

Return uniqueArraySmall

end

My final solution uses a for loop to put the unique numbers into a smaller array that is the perfect size to fit all the unique numbers. The function then returns this final array of unique numbers.

1. As with all the previous problems I started by treating the entire problem as a black box so I could determine its parameters.



Once I had determined the parameters of my solution I then broke the black box down into separate parts using a stepwise refinement tree.



After coming up with my stepwise refinement tree I realised I could reuse the remove duplicates algorithm from part i to get all the unique elements out of the array; however, this required the array to be sorted prior and therefore I needed to add an abstraction to make a sorted copy of the initial array. To do this I first created a black box diagram for the sorting abstraction to find out its parameters.



With the parameters of my sorting algorithm determined I decided to go ahead a produce the ADL algorithm for this abstraction.

Procedure SortArray(IN elements[], IN Size, OUT sortedArray[])

declare prevElements[1..size], prevElementsIndex, count, new, sortedCount

sortedCount ← 0

prevElementsCounter ← 0

for elementIndex ← 1 to size by 1 do

count ← 1

new ← true

while (count ≤ prevElementsCounter AND new = true) do

if elements(elementIndex) = prevElements(count) then

new ← false

end

end

if new = true then

prevElementsCounter ← prevElementsCounter + 1

prevElements(prevElementsCounter) ← elements(elementIndex)

for elementIndex2 ← 1 to size by 1 do

if elements(elementIndex) = elements(elementIndex2) then

sortedCount ← sortedCount + 1

sortedArray(sortedCount) ← elements(elementIndex)

end

end

end

end

end

The above algorithm uses the main for loop to scan the entire array, first it checks if the element is different to the elements it has checked previously. If this is true it then adds the current element to the list of previous elements for the next iteration and then scans the entire array looking for elements that are the same as the current element and adds them to the sorted array.

The sorted array is not necessarily sorted in any real order bar the fact that all items that are the same will end up next to each other which allows the remove duplicates algorithm from part i to remove all the duplicates from the sorted array resulting in an array that only contains unique elements. However, remove duplicates is a function which only returns the array without its size and therefore to overcome this issue, I will reuse the Get Unique Numbers procedure from part i in the main body of my algorithm and repeat the code for removing the empty space from the array resulting in the algorithm have both access to the array without duplicates and its size.

Finally, with all my abstractions in place I produced the following ADL function that when given an array of elements and its size will return the most frequent element in the array.

Function MostFrequent(IN elements, IN size)

delcare sortedArray[1..size],uniqueCount,uniqueArray[1..size]

call SortArray(elements,size,sortedArray)

call GetUniqueNumbers(numbers,size,uniqueCount,uniqueArray)

Declare uniqueArraySmall[1..uniqueCount],occurences[1..uniqueCount]

For uniqueArrayIndex ← 1 to uniqueCount by 1 do

uniqueArraySmall(uniqueArrayIndex) ← uniqueArray(uniqueArrayIndex)

end

for uniqueArrayIndex ← 1 to uniqueCount by 1 do

occurences(uniqueArrayIndex) ← 0

for elementsIndex ← 1 to size by 1 do

if uniqueArraySmall(uniqueArrayIndex) = elements(elementsIndex) then

occurences(uniqueArrayIndex) ← occurences(uniqueArrayIndex) + 1

end

end

end

declare mostFrequentIndex, currentMostFrequent

currentMostFrequent ← 0

mostFrequentIndex ← 0

for occurencesIndex ← 1 to uniqueCount by 1 do

if occurences(occurencesIndex) > currentMostFrequent then

mostFrequentIndex ← occurencesIndex

end

end

return uniqueArraySmall(mostFrequentIndex)

end

(unfortunately, word has broken most of the indentation here)

The function above first creates a sorted copy of the initial array and then creates a copy of sorted array without any duplicates; it then takes this array and produces a copy without any unoccupied cells. This array now contains all the unique elements from the initial array and nothing else. The algorithm then scans the initial array looking for how many times each unique element occurs and stores the number of times an element occurs in its correlating position in the occurrences array. Finally, the algorithm finds the highest value in the occurrences array and keeps track of its position in the array which is then used to return the element it correlates to in the unique elements array.