1.

i. Using the source given in the assignment specification, we know that a permutation in which repetition is allowed can be calculated by doing , which is equal to . Therefore, where repetition is allowed and order matters, calculating the number of permutations of *n* things when we choose *r* of them can be easily done with a single calculation:

permutations ← n \*\* r

Although the exponent/power function exists in ADL and can be used in this calculation, it could also be done using a for loop producing the same result:

permutations ← 1

for i ← 1 to r by 1 do

permutations ← permutations \* n

end

However, we may as well just use the \*\* function as it removes the need for a loop and does the calculation on one line.

Written as an abstraction with the relevant declaration for permutations gives:

procedure permutationsWithRep(IN n, IN r, OUT permutations)

declare permutations

permutations ← n \*\* r

end

Since this abstraction is only returning a single value after doing a calculation, it could be written as a function rather than a procedure:

function permutationsWithRep(IN n, IN r)

declare permutations

permutations ← n \*\* r

return permutations

end

There is a tool on the website cited in the specification ([*http://www.mathsisfun.com/combinatorics/combinations-permutations-calculator.html*](http://www.mathsisfun.com/combinatorics/combinations-permutations-calculator.html)) that calculates the number of permutations given values for *n* and *r* and a set of conditions (Repetition allowed? Order matters?). I will use this and some values of *n* and *r* to test my algorithm.

Example: n = 3, r = 2

Calculator gives: 9

My algorithm gives: permutations ← 32 (= 9)

Example: n = 7, r = 3

Calculator gives: 343

My algorithm gives: permutations ← 73 (= 343)

Example: n = 10, r = 6

Calculator gives: 1000000

My algorithm gives: permutations ← 106 (= 1000000)

ii. When we decide that repetition is not allowed (but order still matters), calculating the number of permutations of *n* things when we choose *r* of them becomes more complex.

Again using the source given in the assignment specification, we know that the number of permutations given will be smaller than in the previous question, as the amount of choices we have decreases each time we choose something.

The formula for calculating the amount of permutations with the given conditions is . Doing the division and subtraction within this formula is easy, however we will need to deal with calculating factorials which will be a little more complicated.

For starters, we will deal with finding . We can store this value in a new variable so that it can be used more easily. This is just a simple subtraction:

declare difference

difference ← n – r

The next thing to do would be to calculate the sum of and , however this is something that we can calculate later and for now just treat is as an abstraction. I will define a new abstraction to handle the calculation of the factorials, and then will finish the rest of the original calculation first. Since I know that the abstraction will return a single value, I can use a function rather than a procedure:

function factorial(IN number)

declare numberFact

return numberFact

end

Now we can call the function and deal with the numbers as if the calculation has been done. We will need two new variables to hold the different factorials that we will have calculated, and then it will be just a simple division:

declare nFact, differenceFact

nFact ← factorial(IN n)

differenceFact ← factorial(IN difference)

permutations ← nFace / differenceFact

We can put all of these together to produce the complete abstraction to calculate the number of permutations:

procedure permutationsWithoutRep(IN n, IN r, OUT permutations)

declare nFact, difference, differenceFact

difference ← n - r

nFact ← factorial(IN n)

differenceFact ← factorial(IN difference)

permutations ← nFact / differenceFact

end

Now we need to deal with the function that calculates the factorials of the given numbers.

To calculate a number factorial, you simply do the number multiplied by every integer that comes before it in the set of positive integers. For example, . This can be done using a for loop with the index counting down and repeated multiplication. However, to keep in line with the general usage of for loops, it can also be done using the index to count up because multiplication is commutative ().

I will need a variable to hold the index/loop counter and another to hold the number factorial, followed by the for loop which will just multiply the numberFact variable by the loop counter:

declare count

numberFact ← 1

for count ← 1 to number by 1 do

numberFact ← numberFact \* count

end

To complete this function, I simply need to add the return statement to return the value of numberFact to the calling function. Here is the completed factorial function:

function factorial(IN number)

declare count

numberFact ← 1

for count ← 1 to number by 1 do

numberFact ← numberFact \* count

end

return numberFact

end

I will use the same calculator given in the previous answer to calculate how many permutations there are given values of *n* and *r*, and I will see what my algorithm would give. I will use the same values given in the previous question for ease.

Example: n = 3 , r = 2

Calculator gives: 6

My algorithm gives: difference ← 3 – 2 (= 1)

nFact ← 1 \* 2 \* 3 (= 6)

differenceFact ← 1 (= 1)

permutations ← 6 / 1 (= 6)

Example: n = 7 , r = 3

Calculator gives: 210

My algorithm gives: difference ← 7 – 3 (= 4)

nFact ← 1 \* 2 \* … \* 6 \* 7 (= 5040)

differenceFact ← 1 \* 2 \* 3 \* 4 (= 24)

permutations ← 5040 / 24 (= 210)

Example: n = 10 , r = 6

Calculator gives: 151200

My algorithm gives: difference ← 10 – 6 (= 4)

nFact ← 1 \* 2 \* … \* 9 \* 10 (= 3628800)

differenceFact ← 1 \* 2 \* 3 \* 4 (= 24)

permutations ← 3628800 / 24 (= 151200)

2.

i. My original idea was to store each user as a single array holding all of their data, with each cell holding an individual data item (name, telephone number, father etc.) as follows:

studentName[name, telephoneNo, father, mother, siblings]

This system would work in theory, but would require a new array to be created every time there is a new student, so would not work well in practice.

Since this would not work, I realised I would need to create a series of data structures to store each data item separately:

name[], telephoneNo[], father[], mother[], siblings[]

This series of arrays means any new students can easily be added into the system and students can easily be removed by simply deleting their data from each array.

This system however would give us an issue when it gets to the siblings array, as students could have multiple or no siblings within the school. For example (using the data given in the question):

name[John Adams, Susan Bailey, Bruce Clark, …]

…

siblings[Jane, Willam, Donald, David, Lisa, …]

If the data in the siblings array was mapped one-to-one with the names of the students, then this system would provide limit each student to a maximum of one sibling, and would then report any other siblings as related to someone else. Reading from the example above, Jane would be the sibling of John Adams, Willam would be the sibling of Susan Bailey, and Donald would be the sibling of Bruce Clark, leaving David and Lisa to be siblings of other students in the school. We know however that this is not true, Jane, Willam and Donald are siblings of John Adams, Susan Bailey has no siblings, and David and Lisa are siblings of Bruce Clark. Therefore we need some extra information to be able to link these two arrays together.

The way I thought of solving this problem this would be to store the number of siblings each student has so that it can be used along with the student’s index in the name array to locate the position of their siblings in the siblings array.

Here is an example using the same data as above:

name[John Adams, Susan Bailey, Bruce Clark, …]

…

noOfSiblings[3, 0, 2, …]

siblings[Jane, Willam, Donald, David, Lisa, …]

Using this system, we know that John Adams has 3 siblings, and would therefore be the first 3 siblings in the array, Susan Bailey has 0 siblings and therefore has no siblings in the array, and Bruce Clark has 2 siblings so the next 2 siblings in the array are his. How exactly we would use this data to find the siblings will be discussed in the next part of the question.

Here is my final set of data structures, shown as text and diagrammatically, featuring example data taken from the question:

name[John Adams, Susan Bailey, Bruce Clark, …]

telephoneNo[01142253175, 01142256900, 0161248653, …]

father[Richard, Steven, XXXX, …]

mother[Mary, Sheila, Barbara, …]

noOfSiblings[3, 0, 2, …]

siblings[Jane, Willam, Donald, David, Lisa, …]

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| name |  | telephoneNo |  | father |  | mother |  | noOfSiblings |  | Siblings |
| John Adams |  | 01142253175 |  | Richard |  | Mary |  | 3 |  | Jane |
| Susan Bailey |  | 01142256900 |  | Steven |  | Sheila |  | 0 |  | Willam |
| Bruce Clark |  | 0161248653 |  | XXXX |  | Barbara |  | 2 |  | Donald |
| … |  | … |  | … |  | … |  | … |  | David |
| … |  | … |  | … |  | … |  | … |  | Lisa |
|  |  |  |  |  |  |  |  |  |  | … |

ii. To find the siblings of a student in this system, we would first need to know the index of their name in the name array. To do this I will use a while loop to search through the name array in order, checking if the name entered matches a name in the array. We do not need to check the validity of their input, so we can assume that the name they have entered exists in the array and therefore do not have to handle the case of if the name does not match.

To find the name using a while loop, we will need a few things. First we will need the name of the student, second we will need an index variable to store their position in the array, next we will need the size of the name array to make sure we don’t increment the index past the size of the array (this will be included as an input to the abstraction), and finally we will need a Boolean variable to use to leave the loop once the name has been found:

declare found, nameIndex, i

We will need this while loop to run from the first name in the array up to the last, comparing the name of the student we want to the names in the array. We will need an if statement to check if the name matches that in the row, and if it does we should set the Boolean value to true so we can leave the loop. If the name in the current index does not match the name, we will want to increment the loop counter to try the next name. The completed while and nested if statement would look like this:

i ← 1

while (i =< nameSize AND NOT found)

if (studentName = name[i])

found ← true

nameIndex ← i

else

i ← i + 1

endif

end

Since a student may have many or no siblings within the school, the names of siblings in the siblings array will not be at the same index as the student’s name in the name array. To find the index of their siblings’ name(s) in the siblings array, we would need to know how many siblings are before it in the array. This can be calculated by adding up the number of siblings that are stored before the desired student’s data in the noOfSiblings array. Since this would give the number of siblings before the student’s sibling(s) in the siblings array, we need to add one to get to the student’s sibling(s) we are looking for. This calculation can be done with a for loop. We now know the index of the student’s name in the name array, and since there is a one-to-one relationship between the name array and the noOfSiblings array, we can count up to that index to find the amount of siblings that come before the student’s we are after:

siblingIndex ← 1

for j ← 1 to (nameIndex - 1) by 1 do

siblingIndex ← siblingIndex + noSiblings[j]

end

siblingIndex was set to 1 initially to handle that addition of one that I discussed.

We can then print out the names of the student’s siblings by going to the siblingIndex in the siblings array and printing each name until we have printed as many names as the student has siblings. Again, this can be done with a for loop. We will need a variable to store the number of siblings the student has that will be obtained from the noOfSiblings array, and two variables to use as indexes, one to count from 1 to the number of siblings, and another to combine the index of the first sibling with the index of the amount of siblings:

declare numberSiblings, k, l

numberSiblings ← noSiblings[nameIndex]

for k ← 1 to numberSiblings by 1 do

l ← k + siblingIndex

print(siblings[l])

end

With all of the required parts of the algorithm done, I can put it together to make my final algorithm as an abstraction:

prodecure findSiblings(IN studentName, IN name[], IN nameSize, IN noSiblings[], IN siblings[])

declare found, nameIndex, siblingIndex, numberSiblings, i, j, k, l

i ← 1

while (i =< nameSize AND NOT found)

if (studentName = name[i])

found ← true

nameIndex ← i

else

i ← i + 1

endif

end

siblingIndex ← 1

for j ← 1 to (nameIndex - 1) by 1 do

siblingIndex ← siblingIndex + noSiblings[i]

end

numberSiblings ← noSiblings[nameIndex]

for k ← 1 to numberSiblings by 1 do

l ← k + siblingIndex

print(siblings[l])

end

end

3.

i. Method 1: Removing duplicates from the original array

Since the array that we use as an input contains sorted integers, to determine if a number has a duplicate we need only compare it to the next number in the array.

To do this, I can use a for loop to go through the array one by one and an if statement to compare one value in the array to the next, such as the following:

for i ← 1 to size – 1 by 1 do

if (array[i] = array[i + 1])

We will deal with what happens in the if shortly.

We only want to search from 1 to size - 1 because we are comparing one value with the next, therefore once the for loop reaches size - 1, the value in array[size – 1] will be compared with the value contained in array[size]. We can also use separate variables to store size - 1 and i + 1 just to make the algorithm cleaner since we will use them multiple times:

end ← size - 1

j ← i + 1

If a duplicate entry is found in the algorithm, we want to remove that duplicate value. The easiest way to do this would be to ‘shift’ every later value in the array back one cell, thus removing the duplicate, and then decreasing the size of the array by 1.

In ADL, this could also be done with a for loop, starting from the duplicate value and going to the new value end (size - 1), copying the value in one cell to the previous cell:

for i ← j to end by 1 do

array[i] ← array[i + 1]

end

We need to use a new index variable as i is being used in the original for loop, and can also use a new variable for the new index + 1, again to make the algorithm clearer.

k ← j (the new index)

l ← k + 1

After ‘shifting’ all of the data back one cell, we need to decrease the size of the array by 1 to indicate that we’ve removed a duplicate value:

size ← size - 1

Now that we have a method to remove duplicate values, we need to compare our original i value to the new integer that occupies the following cell. To do this, we could add another for loop which would basically repeat the same steps, or we could just decrement the index value so that on the next loop of the for loop, it will be checking the same index against the new value in the next cell. This can be done easily:

i ← i - 1

Putting these together with the relevant declarations and ends results in the following algorithm that I have written as an abstraction to be my final answer:

procedure removeDuplicatesExisting(IN array[], IN size, OUT array[], OUT size[])

declare i, j, k, l, end

end ← size - 1

for i ← 1 to end by 1 do

j ← i + 1

if (array[i] = array[j])

for k ← j to end by 1 do

l ← k + 1

array[k] ← array[l]

end

size ← size – 1

i ← i - 1

endif

end

end

Method 2: Creating a new array that contains no duplicates.

For starters, we will need to declare this new array as it is something that will be created within the abstraction:

declare array2[]

As with method 1, we know that the array we are given contains sorted integers, so we only need to compare each number in the array to the next to know if it is a duplicate.

Again, I will be using a for loop to go through the array, and will also be using a variable named end (this time end1 to denote that it relates to array1) to store the value of size - 1 (size1, again for array1) and another variable named j1 to store i1 + 1:

end1 ← size1 – 1

for i1 ← 1 to end1 by 1 do

j1 ← i1 + 1

Unlike the last method however, we don’t need to operate if one value is the same as the next, we only need to do something if the values are different.

In this method, we will copy over the value into the second array only if the next value is not the same as the previous one. We also need to copy over the first value from the first array to the second before we enter the for loop as there is no value before the first value that it can be a duplicate of:

array2[1] ← array1[1]

if (array1[i1] != array1[j1])

Now we need to consider what we want happen in this if statement.

This if statement checks if a value is the same as the one that comes after it, or we could also view it as it checks if a value is the same as the one that comes before it. The reason I have thought about it in this way because we have already copied over the first value in array1, and therefore to carry on we need to compare that to the next value to decide it we will copy over the next value. Since we are going to be copying over the second value, it makes more sense to think about comparing that value to the first rather than the other way around.

So now that we know we need to copy over the second value (or the value referenced by j1) we need to know where to copy that value to. We will need an index for the second array, a starting position for this index, something to increment the index counter, and a new variable to hold the size of this new array.

To determine the size of this array, we can just use the index variable for the array as that will point to the highest value in the array. Because of this, we will only want to increment the index counter when we are copying a value into the array, we don’t want to pre-increment this counter for possible future values as then it won’t be exactly indicative of the size and means we will need to do an additional calculation to get the size of the array later. We already copied the first value from array1 into array2, so we should start the index at 1 to indicate that it currently holds only 1 value:

declare size2, i2

i2 ← 1

i2 ← i2 + 1

size2 ← i2

Now that we have the necessary variables set up for array2, we need to know what to copy into it. Our if statement checks if j1 is not equal to i1, so we will want to copy over j1 into array2 if it turns out to be unique:

array2[i2] ← array1[j1]

Now that we have decided what we want to copy over and have written the statements that will do that, we can put all of this together into an abstraction to show the final algorithm:

procedure removeDuplicatesNew(IN array1[], IN size1, OUT array2[], OUT size2)

declare array2[], size2, i1, i2, j1, end1

end1 ← size1 - 1

array2[1] ← array1[1]

i2 ← 1

for i1 ← 1 to end1 by 1 do

j1 ← i1 + 1

if (array1[i1] != array1[j1])

i2 ← i2 + 1

array2[i2] ← array1[j1]

endif

size2 ← i2

end

end

ii. The biggest problem with creating an algorithm for this question is the fact that there is no limit to the size of the array, so there could be hundreds of different elements in the array that each have hundreds of occurrences which would all need to be checked and counted.

I think a way to combat this problem would be to first find the individual elements in the array, and to then find out how many times each element occurs.

When it comes to finding the individual elements in the array, I will make a call to the second algorithm that I created in Question 3 Part i, removeDuplicatesNew. While I could use the first algorithm I created, I think using the algorithm created in my second method would be better as it isn’t destructive and therefore leaves the original array intact. To call this abstraction, I will use the following statement:

CALL removeDuplicatesNew(array1[], size1, array2[], size2)

The next part of the problem is finding out how many occurrences of each element are in the array. I think that to do this I will have to use one or more for loops, although I think there is a good chance that it will need to be a pair of nested for loops.

First, I will need a loop to go through the individual elements in the array so that we can use them to make a comparison and begin counting. I will also use a variable to store the current element that we are counting, as this will make the algorithm easier to read and understand than if I just compared it against the array every time. The first loop would need to go through the list of individual elements and then store the current element in a variable:

declare currentElement

for i ← 1 to size2 by 1 do

currentElement ← array2[i]

The next loop would need to count the number of occurrences of currentElement in the original array. To do this, I will create a new array to store the number of occurrences, so there will be a one-to-one relationship with this array and the array that stores the individual elements:

declare count[]

Every value in the array would need to be initialised to 1 since there is definitely at least 1 occurrence of each element in the array. This should be done within the first for loop since it needs to be done for each value of i:

count[i] ← 1

The behaviour of the for loop itself will need to increment the count array at the index of the current element if there is a matching element in the original array. To do this it will need to go from 1 to the size of the original array using a new index, and would then require an if statement to compare the value stored in currentElement with values in the original array. This would be done as follows:

for j ← 1 to size1 by 1 do

if (currentElement = array1[j])

count[i] ← count[i] + 1

endif

end

Now that we have a system to find the individual elements, and another to count them, we need something to find which of the elements occur most frequently. Although it would be easiest to print the answer to this rather than store it, that’s not what the question asks.

Since there could be multiple elements that occur the same amount of times, we will need to store these elements in an array and have a variable to store the size of this array. And since the most frequent element(s) will occur the same amount of times, we only need a single variable to store the number of occurrences. These new data elements could be as follows:

declare mostFrequent[], mostFreqSize, occurrences

To determine which element(s) occur the most, it will first be easier to find the highest number of occurrences and then use that to find the elements that have that many occurrences. Finding the highest number of occurrences is easy, this just requires us to skim through the count array and set the occurrences variable to the highest amount in the array.

We can initialise the value of occurrences to the first value in the count array because this will give us a value to make comparisons to. Since we will be looking for the highest value in the array, we can pick any value in the array as a starting point and just compare values to it to see if we can find one higher. It makes sense to start with the first value however:

occurrences ← count[1]

As with a lot of this problem, it will be best done with a for loop (and again a new index), and an if statement nested inside to make comparisons:

for k ← 1 to size2 by 1 do

if (count[k] > occurrences)

occurrences ← count[k]

endif

end

Now that we know the highest number of occurrences, we just need to know which elements occur this many times, and we then want to copy these elements into the mostFrequent array. Since I stored the individual elements and their frequency into two separate arrays that have a one-to-one relationship, if we find the location of the highest number in the count array, we know that the element that occurs that many times will be at the exact same index in array2.

To complete this algorithm, we will need a final for loop (and new index) to search through the count array, an if to check for the value occurrences, and then we will need to copy values into the mostFrequent array. We will need an index for this array too, since we will need to know what cell to copy the elements into. This index for the mostFrequent array can use the variable mostFreqSize since the highest position in the array will also be the size of the array:

declare l

mostFreqSize ← 0

for l ← 1 to size2 by 1 do

if (count[l] = occurrences)

mostFreqSize ← mostFreqSize + 1

mostFrequent[mostFreqSize] ← array2[l])

endif

end

If we combine this slew of components together, we get what should be the final algorithm which will take an array and size as inputs, and will output an array which contains the most frequently occurring element(s), a variable that contains the size of this array, and a variable that contains the amount of occurrences of the most frequently occurring element(s):

procedure findMostFrequent(IN array1[], IN size1, OUT mostFrequent[], OUT mostFreqSize, OUT occurrences)

declare i, j, k, l, currentElement, array2[], size2, count[], mostFrequent[], mostFreqSize, occurrences

CALL removeDuplicatesNew(array1[], size1, array2[], size2)

for i ← 1 to size2 by 1 do

currentElement ← array2[i]

count[i] ← 1

for j ← 1 to size1 by 1 do

if (currentElement = array1[j])

count[i] ← count[i] + 1

endif

end

end

occurrences ← count[1]

for k ← 1 to size2 by 1 do

if (count[k] > occurrences)

occurrences ← count[k]

endif

end

mostFreqSize ← 0

for l ← 1 to size2 by 1 do

if (count[l] = occurrences)

mostFreqSize ← mostFreqSize + 1

mostFrequent[mostFreqSize] ← array2[l])

endif

end

end