Constructing an array of unique elements

Arjen Markus

January 29, 2024

Introduction

The Fortran programming problem I want to consider in this note is the following: given an array of values (or perhaps better: elements) construct a new array that only contains the unique elements of the original. For instance:
The given array contains A, F, B, C, C, A, D. Then we want an array only containing A, B, C, D and F as both A and C occur twice and the second occurrences can therefore be deleted.

This description is simple enough, but I want to be a bit more specific:

- The original array is not to be rearranged or changed in any way – we may have to use it later on again.
- All elements occupy the same amount of computer memory (for convenience) and can be compared for equality (to determine the uniqueness).
- The order in which the unique elements appear in the output array is of no importance. It might be the same order as in the original array, it may also be some sorted sequence. This makes it possible to sort a copy of the array and select the unique elements by passing once through that sorted array.

There is one more decision to be made: I want to analyse various implementations in detail and for that it might make a difference if the implementation results in a subroutine or a function. In the latter case, a function might return an (allocatable) array and the result needs to be copied into the left-hand variable:

```
integer              :: array(n)
integer, allocatable :: unique_elements(:)

interface
    function unique( array )
        integer, intent(in)  :: array(:)
        integer, allocatable :: unique(:)
    end function unique
```

```
    end interface

    unique_elements = unique(array)
```

The compiler might recognise that the left-hand variable, `unique_elements`, can adopt the allocated result directly (in pseudo code):

```
    tmp <- unique(array)
    call move_alloc( tmp, unique_elements)
```

instead of:

```
    tmp <- unique(array)
    allocate( unique_elements(size(tmp) )
    unique_elements = tmp
    deallocate( tmp )
```

Whether or not a compiler will actually do something like that to avoid the extra copying, is something for compiler writers to answer. Or for people who can read the assembly output. I want to avoid the question as much as possible and instead focus on what we can determine from the source code itself.

So, let us require the interface to be (just using the `integer` type as an example):

```
    interface
        subroutine unique( array, unique_elements )
            integer, intent(in)  :: array(:)
            integer, allocatable :: unique_elements(:)
        end subroutine unique
    end interface
```

That way the code will manipulate the variable that will hold the resulting arraty directly.

A straightforward implementation

Here is a first, straightforward, implementation that in a sense can be used as a benchmark. First we create an array that can hold any result – we do not know in advance how many unique elements there will be, but there will never be more than the size of the original array. We determine the unique elements and then create a new array with the right size to hold only those elements (defining the type of the elements in the array as `TYPE` to make the code generic):

```
subroutine unique( array, unique_elements )
    TYPE, intent(in)  :: array(:)
    TYPE, allocatable :: unique_elements(:)

    TYPE, allocatable :: work_array(:)
```

```
    integer           :: i, n

    if ( size(array) <= 1 ) then
        !
        ! Take care of the trivial cases
        !
        unique_elements = array
    else
        !
        ! Scan the elements and store the ones that we encounter
        ! for the first time.
        !
        work_array = array

        n = 1
        do i = 2,size(array)
            if ( all( work_array(i) /= work_array(1:n) ) ) then
               n = n + 1
               work_array(n) = work_array(i)
            endif
        enddo

        unique_elements = work_array(1:n)
    endif
end subroutine unique
```

The amount of work that has to be done to determine the result is measured in various quantities:

- The total amount of memory:
  - Define N to be the number of elements in the original array
  - Define M to be the number of unique elements
  - Extra memory in the work and result arrays: $N + M$ times the size of an element
- As for memory management, there are two allocations (`work_array` and `unique_elements`) and one deallocation (`work_array`).
- The number of comparisons. This is where it gets tricky:
  - If there is only one unique element in the original array, then `n` will remain 1 and the number of comparisons is $N - 1$.
  - If there are two unique elements, say the original array is `[1, 2, 2, 2, 2, 2, 2]`, then in the first step `n` becomes 2 and all remaining elements will be compared to two unique elements, `[1, 2]`. Hence there would $(N-1)+(N-2) = 2N-3$ comparisons. However,

if the original array is `[1, 1, 1, 1, 1, 1, 2]`, the second element is encountered only in the last step and therefore there would be only $N - 1$ comparisons.

- If all elements in the array are unique, then we get the maximum number of comparisons: $(N - 1)(N - 2)/2$.

In other words the exact number of comparisons would depend on the exact content of the array, but the maximum number is uniquely defined. A typical behaviour, however, with a few duplicate elements, is difficult to determine. Therefore we will focus on the worst-case behaviour.

Note: In the calculations it is assumed that the `all` function examines all elements, whereas it can of course use shortcircuiting if it finds that the element is duplicated. The assumption makes the analysis simpler.

- Finaly, there are $3M$ assignments of the elements in question. If the elements occupy a large amount of space (or if there is something complicated regarding an assignment, like allocatable components in a derived type), then this might be an important measure in itself.

Yet another interesting measure is the number of lines of code. Leaving out the declarations and the comments, the implementation is 13 lines long.

Iterative solution with the "pack" function

Here is an alternative solution which uses the `pack` intrinsic function. It was published by @eelis (`https://fortran.discourse.group`) as a contribution to the *Advent of Code 2023* (`https://github.com/ettaka/aoc2023/blob/9edcf5f4387bad10d22f6153d339b0e7355f59af/7/aoc7_part1.f90`):

z

```
subroutine unique( array, unique_elements )
    TYPE, intent(in)  :: array(:)
    TYPE, allocatable :: unique_elements(:)

    TYPE, allocatable :: work_array(:)
    integer           :: i

    allocate( unique_elements(0) )
    work_array = array(:)

    do while ( size(work_array) > 0 )
        unique_elements = [unique_elements, work_array(1)]
        work_array = pack( work_array, &
                           mask=(work_array(:) /= work_array(1)) )
    end do
end subroutine unique
```

To analyse this algorithm we will also need to consider the cost of the `pack` function. For this a few assumptions have to be made as it may be implemented in different ways. Still, there is an advantage to using `pack` over the previous solution: the order in which duplicates appear in the original list of data elements is less important and any duplicates will be removed rightaway.

If we take the first special case again, only one unique element, then the first application of `pack` returns an empty array and we are done. If there are two unique elements like in `[1, 2, 2, 2, 2, 2, 2]`, then the first round would return a work array `[2, 2, 2, 2, 2, 2]` and the second round would return an empty work array.

If the original array is `[2, 2, 2, 2, 2, 2, 2, 1]`, then the first round returns a work array `[1]` and the second again an empty work array. However the `pack` function has been implemented, all elements will have to be compared to the first one, so that the number of comparisons is N for the case with only one unique element, $N+(N-1)$ for the case with the single value `1` at the start and $N+1$ for the case with the single value `1` at the end (or indeed not at the start).

The number of comparisons in this solution, assuming the worst case, where all elements are unique, is $N(N-1)/2$, which can be improved to $(N-1)(N-2)/2$ by changing the mask to: `work_array(2:) /= work_array(1)`.

The number of allocations, apart from the invocation of `pack`, and deallocations is:

- Two allocations for the initialisation of `unique_elements` and `work_array`.
- Two reallocations (assuming one allocation and one deallocation) for updating these two arrays. This happens in the loop, so $2M$ allocations and $2M$ deallocations.
- At the end of the routine, the work array has to be deallocated.
- The total number of allocations will be $2+2M$ and the number of deallocations $1+2M$.

The number of assignments to the array `unique_elements` is $M(M-1)$, as it is extended with each iteration of the loop. The number of assignments to the work array is much harder to determine: this depends on the presence of duplicated elements. In the worst case it will be $(N-1)(N-2)/2$.

The maximum amount of memory that is used at any one time is at the end of the first iteration: we have the longest working array then (the size equal to the size of the original array) and that will only diminish. The result array is the shortest then, but it cannot grow beyond the size of the original array. And the work array will be copied from the full copy of the original array, Hence: the extra memory is $2N$.

The length of the code (number of executable statements) is 6 – less than half of the very first solution.

Cost of the "pack" function

This leaves the cost of invoking the `pack` function. Let L be the size of the work array and K be the number of elements in the result. A smart compiler might avoid creating a mask array explicitly and implement it as:

```
j = 0
allocate( result(size(work_array) ) ! L == size(work_array)
do i = 1,size(work_array)
    if ( work_array(i) /= work_array(1) ) then
        j = j + 1
        result(j) = work_array(i)
    endif
end do

result = result(1:j) ! K == j
```

This solution uses L extra elements, $2K$ assignments (the factor 2 because of the truncation to the right size), two allocations and one deallocation.

Another solution is to first construct a temporary logical array for the mask, count the number K of resulting elements and then loop over the work array:

```
allocate( mask(size(work_array)) )
... determine the mask and count the number of true values (K)

j = 0
allocate( result(K) )
do i = 1,size(work_array)
    if ( mask(i) ) then
        j = j + 1
        result(j) = work_array(i)
    endif
end do
```

This solution uses a logical array of L long, an array of K elements and K assignments only. Both in terms of memory and of assignments and allocations/deallocations, it is more efficient than the first solution.

In both cases the number of comparisons is L.

Total cost of the iterative "pack" solution

Combining these observations, we get:

- We need to store about $N+2M$ elements – `unique_elements`, `work_array` and the result of `pack()`.
- We have the following sets of assignments:
  - $\frac{1}{2}N^2$ for `unique_elements`.

– Roughly the same number for `work_array` and for the results of `pack()`.

This brings the total to: $\frac{3}{2}N^2$ assignments.

Recursive solution with the "pack" function

Yet another implementation uses recursion in stead of a loop to construct the result array (`https://github.com/wavebitscientific/functional-fortran/blob/master/src/functional.f90`; code adjusted for the nomenclature and layout used here):

```
subroutine unique( array, unique_elements )
    !! Returns a set given array 'array'.
    TYPE, dimension(:), intent(in)  :: array !! Input array
    TYPE, dimension(:), allocatable :: unique_elements

    unique_elements = set( array )

contains
pure recursive function set( x ) result(res)
    !! Returns a set given array 'x'.
    TYPE, dimension(:), intent(in)  :: x !! Input array
    TYPE, dimension(:), allocatable :: res

    if ( size(array) > 1) then
        res = [x(1), set( pack( x(2:), .not. x(2:) == x(1)) )]
    else
        res = x
    endif

end function set

end subroutine unique
```

Since the result of the function is recursively used in an expression, it is easier to leave it as a function. But to keep the interface uniform we need to wrap it in a subroutine.

The essential difference with the previous, iterative, solution is that during the recursion all earlier invocations stay alive (Fortran does not define *tail recursion* that would allow the program to be more efficient in terms of memory). So, each new recursion adds to the memory use: if the original array (the result of `pack`) is N large, then the maximum amount of extra memory is in the order of $N(N-1)$ if all elements are indeed unique. The memory is only released when the recursive calls return.

So, this solution uses more memory. Its advantage is, perhaps, elegance and it is slightly shorter, effectively only 5 lines.

Sorting the elements

If we do not care about the order of the elements in the original array, then a straightforward implementation could be to first sort the elements and then examine them:

```
subroutine unique( array, unique_elements )
    TYPE, intent(in)  :: array(:)
    TYPE, allocatable :: unique_elements(:)

    TYPE, allocatable :: work_array(:)
    integer           :: i, j

    work_array = array
    call sort( work_array )

    if ( size(work_array) > 0 ) then

        j = 1
        unique_elements = [work_array(1)]

        do i = 2,size(work_array)
            if ( work_array(i) /= unique_elements(j) ) then
                unique_elements = [unique_elements, work_array(i)]
                j               = j + 1
            endif
        end do
    else
        !
        ! Take care of the trivial case
        !
        allocate( unique_elements(0) )
    endif
end subroutine unique
```

The advantage of this method is that once we have the sorted array, we only need to scan it once and we need only a work array of N elements. Of course, there are drawbacks:

- The elements in the original array need to be "sortable", i.e. there is some method to determine if one is smaller than the other.
- The sorting takes time and possibly extra memory as well. Using a trivial *quicksort* implementation which does the sorting in place, we can reduce the amount of extra memory to a minimum – the algorithm is easiest to implement via recursion, so you do have that overhead.

Typically, a quicksort algorithm requires $O(Nlog_2N)$ steps (comparisons) and a similar number of actual rearrangements of the elements. When rearranging in place, you need three or four assignments to swap the elements. If the amount of work to swap the data is large, you could do this via pointers (swapping addresses) or via indirect addressing (sorting an index array instead of the elements themselves).

All in all:

- $M+1$ allocations and M deallocations. (If we use the strategy of the first implementation, this can be reduced to two allocations and one deallocation.)
- A work array of N elements.
- $O(Nlog_2N) + N$ comparisons.

Using a hash array

The code by John Burkhardt, `SET_THEORY` (`https://people.math.sc.edu/Burkardt/f_src/set_theory/set_theory.html`), assumes that the elements in the original array are all small integers, small in the sense that you can have an array in memory that indicates if the element is present. The indication consists of a single bit or a logical.

The elegance of this representation is that adding an element has effect only once. The bit or the logical will then be set and you cannot have duplicates. The drawback is that you cannot use this representation for real numbers or strings beyond, say, two or three characters, because the number of possible values will exceed the memory of the computer. However, you could use a hashtable to store the elements, with the advantage that storing the same value twice is easily detectable, as the hash key will then already exist.

It is difficult to examine the performance of such a hashtable solution in all generality: there are many different possibilities for constructing a hashtable and many design choices. For instance: how long should the hashtable be? What is the cost of calculating the hash key?

The code below illustrates the `SET_THEORY` approach in a rather naïve way. For simplicity, it is limited to integers – otherwise a hashkey function would have been necessary:

```
subroutine unique( array, unique_elements )
    integer, intent(in)               :: array(:)
    integer, intent(out), allocatable :: unique_elements(:)

    logical, allocatable              :: work_array(:)

    integer                           :: i, min_value, max_value
```

```
    min_value = minval( array )
    max_value = maxval( array )

    allocate( work_array(min_value:max_value) )
    work_array = .false.

    ! Use the input array as index
    work_array(array) = .true.

    unique_elements = pack( [ (i, i = min_value, max_value)], &
                            work_array )
end subroutine unique
```

Some observations:

- The code assumes that the range of values is small enough that the work array fits into the memory. So, the success of the routine depends on the actual data that are passed.
- There are $2N$ comparisons to determine the bounds of the work array and N for the `pack()` function.
- Total memory to be used: $N + M$ for the temporary array used by `pack()` and `unique_elements`, R logicals for the work array, where R is the range of the values.
- All comparisons and assignments are for intrinsic types.
- The order in the original array is not preserved.

All in all, this implementation requires 6 statements without any loops or recursion.

Orderpack's solution

A last solution that I would like to mention here is that of `ORDRPACK` by Michel Olagnon and refactored and "fpm-ized" by John Urban (`https://github.com/urbanjost/M_orderpack`). The `unique` subroutine in this package uses a merge-sort algorithm to sort the data elements by rank and marking equal elements by the same rank. Then compressing the original array in place, it returns a changed array with the first M elements being unique.

The implementation is more extensive than the examples shown here. A quick examination of the code shows that it requires an array of N integers for storing the ranks and an array of N logicals to store whether the value has already been seen or not.

If you were to use it with the current API, where you get a new array, then a wrapper like this would be required:

```
subroutine unique( array, unique_elements )
    use M_orderpack, only: unique => unique_order

    TYPE, intent(in)                :: array
    TYPE, intent(out), allocatable :: unique_elements

    TYPE, allocatable               :: work_array
    integer                         :: number_unique

    work_array = array

    call unique_order( work_array, number_unique )

    unique_elements = work_array(1:number_unique)
end subroutine unique
```

While the `ORDRPACK` routine does preserve the order of the elements as they appear in the original array, our requirement is that the original array is left intact. Hence the wrapper must allocate a copy, increasing the required amount of memory.

Conclusion

A seemingly simple algorithm as the construction of an array of unique elements from a given array that may contain duplicates can be implemented in very different ways with very different costs. It is difficult to determine an "optimal" one, as the total amount of memory, the number of assignments allocations, deallocations and comparisons differ completely and the actual cost depends not only on the number of data elements but also on the precise order in which they appear and on the level of duplication.

Still, some general advice may be formulated:

- If the assignment or the comparison of the data elements is costly (each element takes a much larger chunk of memory than the typical 4 or 8 bytes for intrinsic types), it may be useful to use an index array instead of assigning the data elements in the intermediate steps or to use a hash key for the comparison instead of comparing the data elements directly.
- If you expect a lot of duplicates, that is: $M \ll N$, it may be more efficiently to use the `pack()` solution, as you get rid of the duplicates fast.
- If, on the other hand, there are only a few duplicates, then the first, straightforward, solution seems more appropriate.
- While elegant, recursive solutions are more costly in terms of memory usage than iterative solutions.

- If the amount of data is "small" or the construction of such an array of unique elements is not on the critical path, use any solution that ensures a correct result.

Disclaimer: The cost estimates presented here may be inaccurate, so consider them to be, well, estimates only. It might have been better to use the O-notation, but that would hide the proportionality constants and these are at least as interesting as the dependency on the various size parameters – N, M, etc.