1. Explain the difference between an array size and capacity [0.1 pts]

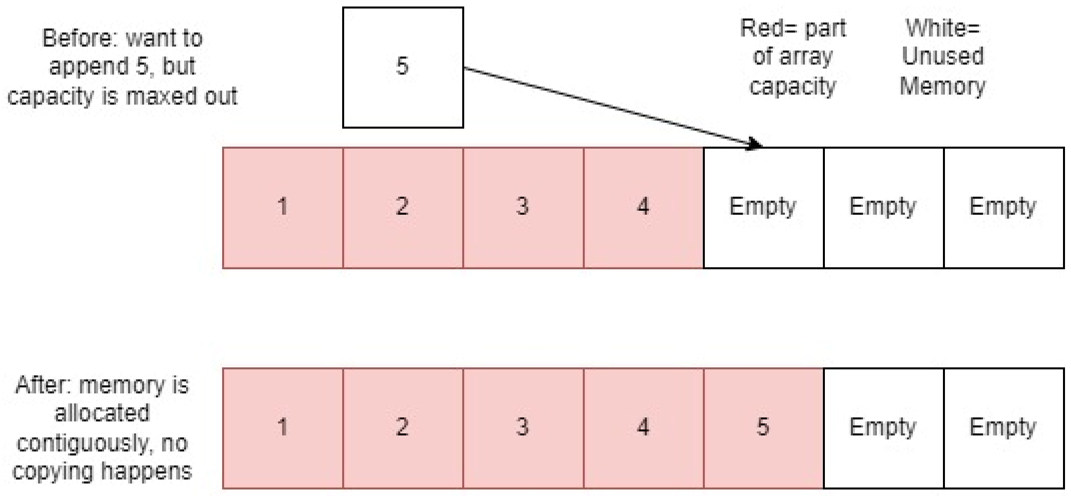
An array's size is the current number of elements stored in the array. Capacity is different because it is the amount of memory allocated for the array. This means that capacity is always greater than or equal to size. To increase size past the current capacity, capacity must first grow in some way.

2. What happens when an array needs to grow beyond its current capacity? Explain and produce a diagram showing the memory layout before and after expansion

We need to increase the array's capacity to grow the array further. There are two cases; for both let's consider an array with both capacity and size of four elements, and let's also suppose that we need to add an extra element at the end (i.e. append a value).

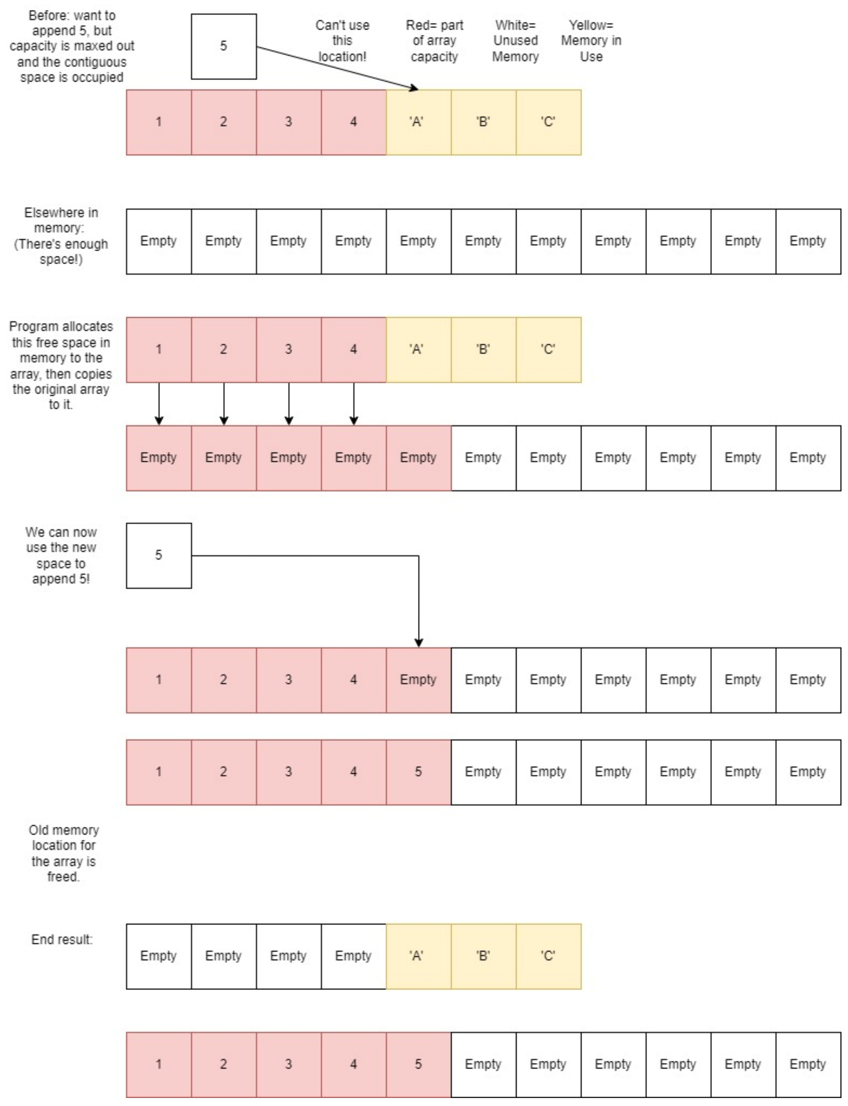
2.1First, consider the case where there is space in memory after the end of the array [0.1 pts]

In this case, we only need to allocate the extra space right next to where the array is located. This means that the data doesn't need to be moved or copied elsewhere. Moreover, this can be repeated indefinitely for following append operations until the next space in memory is already occupied.


Before: want to append 5, but capacity is maxed out
5
Red= part of array capacity
White= Unused Memory
1
2
3
4
Empty
Empty
Empty
After: memory is allocated contiguously, no copying happens
1
2
3
4
5
Empty
Empty

2.2 Then, consider the case where the memory after the end of the array is occupied by another variable. What happens in that case? [0.2 pts]

This time we can't use the memory next to the end of the array to store the new value. This is because that memory is already in use, and we should never write over it. To avoid this, we will need to find another space in memory that can hold the entire array and an extra element, all contiguously. Therefore, if we have an array of four elements, we need to find a space that can hold at least five elements in a row. After said space is found, the entire array is copied in order, and after that the new element can finally be appended. Finally, the space that was used previously for the array is freed so that other variables may access that location in memory.


Before: want to append 5, but capacity is maxed out and the contiguous space is occupied
5
Can't use this location!
Red= part of array capacity
White= Unused Memory
Yellow= Memory in Use
1 2 3 4 'A' 'B' 'C'
Elsewhere in memory: (There's enough space!)
Empty Empty Empty Empty Empty Empty Empty Empty Empty Empty Empty
Program allocates this free space in memory to the array, then copies the original array to it.
1 2 3 4 'A' 'B' 'C'
Empty Empty Empty Empty Empty Empty Empty Empty Empty Empty Empty
We can now use the new space to append 5!
5
1 2 3 4 Empty Empty Empty Empty Empty Empty Empty
1 2 3 4 5 Empty Empty Empty Empty Empty Empty
Old memory location for the array is freed.
End result:
Empty Empty Empty Empty 'A' 'B' 'C'
1 2 3 4 5 Empty Empty Empty Empty Empty Empty

3. Discuss one or more techniques real-world array implementations
use to amortize the cost of array expansion [0.1 pts]

Because the previously discussed method of reallocating memory to hold the increased capacity takes a lot of time, especially with larger arrays or with arrays of more complex types, it is better to increase the capacity by a larger amount than just the number of elements to be appended. That way, in the case that more elements are appended, we don’t need to reallocate and copy the array to a new location in memory, and we can just follow the first case. To choose how many elements to allocate, many programming languages use a growth factor, which multiplies the current capacity to obtain the number of spaces to allocate. For example, if the current capacity is 20 and we have a growth factor of 1.5, we have that 20 * 1.5 = 30, and so when the array’s capacity is expanded, it will be expanded to 30 elements rather than 21 elements. Because the growth factor approach wastes memory to favour performance, a growth factor of 1.5 is considered to be aggressive.