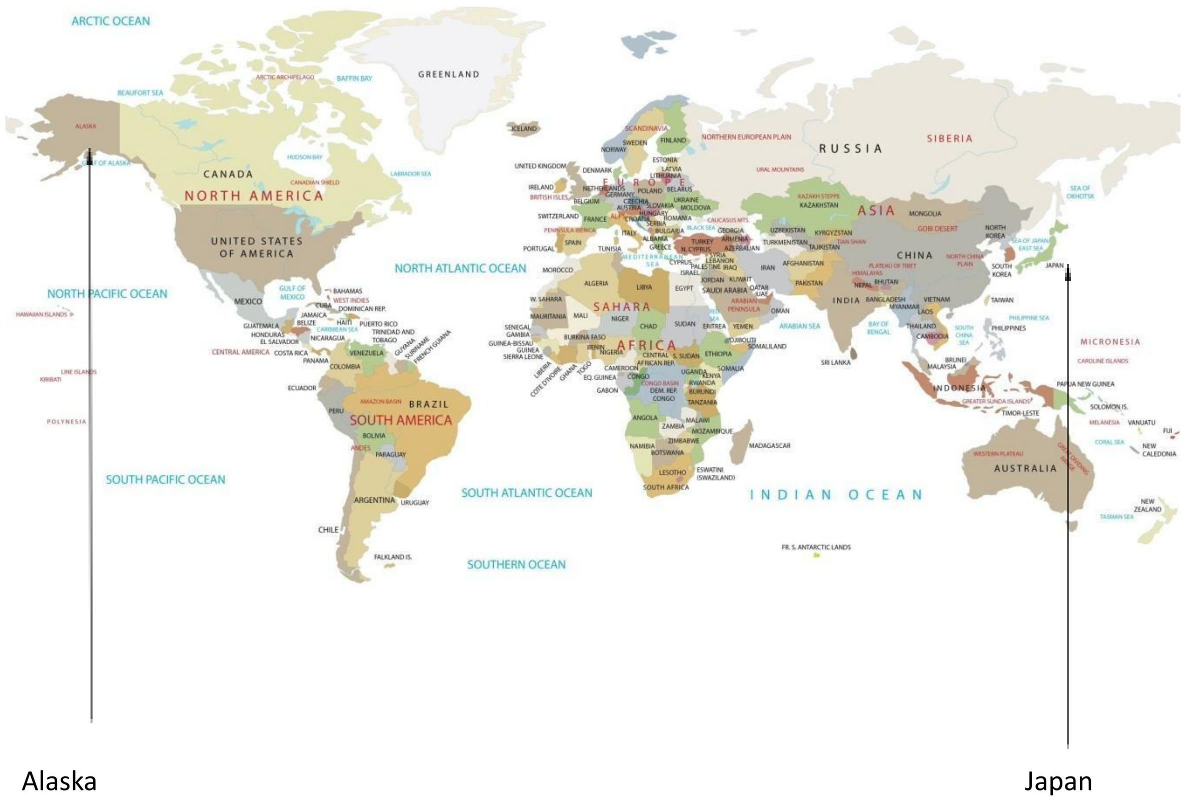
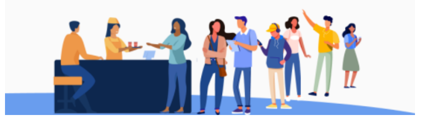
Chapter 2 Part 3: Circular Arrays

Introduction



Consider the 2D image of the world above. As it is 2D, you can use a 2D array to store information related to what unit of a land mass or sea area belongs where. Now what if I ask you how far Alaska is from Japan? If you say they are quite far by counting the number of indexes in-between the two locations, then you will be very wrong. Actually they are quite close as we know that the world is like a sphere – it is not flat. So the right-end wraps around and Join with the left-end. However, given scientists do not know any way to store circular objects such as a planet as it is in the computer memory, we need to use linear, rectangular, cubic, etc. arrays to represents circular objects of respective dimensions in our programs. The important aspect here is that we have to treat the indexes of these arrays in a special way so that indexes do not start from zero and finally end on a positive integer. Instead, they rotate within a range.



Another, common use of circular addressing is queue management with limited amount of resource. Consider a restaurant having a waiting room overseeing the dining place. The number of seats inside the dining place remains fixed so is the capacity of the waiting room. If the dining place is full, you can only let some new diners enter the dining place if some come out. You let them enter the dining place in the order they appeared in the waiting room. If even the waiting room is filled with waiting customers, you sincerely apologize to more incoming customers and ask them to come later. You realize the capacities of both rooms are like linear circular arrays. We often call such arrays, buffers, in real life.

Allocating a circular array, regardless of the number of dimensions it has, is the same as allocating a regular array. You just have to provide the length of all dimensions when creating the array. The only, interesting thing is how you index the array elements. This is what you will learn next. To keep our discussion simple, we will use a linear circular array to explain things. You should be able to use multidimensional circular arrays once you understand how linear circular arrays work.

Indexing a Circular Array

The key thing you have to remember about indexing a circular array is that: you need to do a modulus of any index by the length of the array to find the actual location of the element at a certain index of the circular array in the linear array you used to store it.

Let me explain how this is done. Assume the capacity/length of a circular array, named *buffer*, is 10. Suppose, you want to read the element at index 9, then you do the following:

a = buffer[9 % 10]

You may think this does not make a lot of sense as 9%10 = 9. But, what if you ask what is the next index after 9? You do:

a = buffer[(9 + 1) % 10]

Now you understand the trick, right? As you know (9+10)%10 = 0, the first element of the linear element is the next element to the last element of the circular array. This works for going leftwards also. For example, if you ask for the element that is two steps left of the element at index zero; then you do the following:

a = buffer[(0 - 2) %10]

The answer is (0-2)%10 = -2%10 = 8. The magic lies with the modulus operator that does the index rotation. The capacity of the modulus operator to rotate numbers within a range is extremely useful in other areas also, as you will learn in future courses on cryptography and security.

Iterating/Listing the Elements of a Circular Array

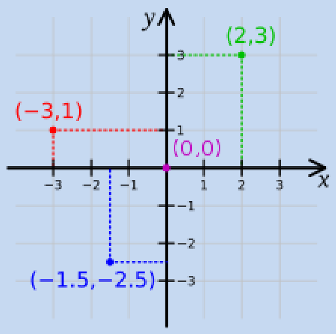
Given a circular array's indexes wraps around at the end, you can iterate over the elements from any starting index. Suppose you are asked to write a code to list all elements of a circular array starting from an input index. Using the Python syntax, you can do this as follows:

```
def listElements(startPos, array):
    size = len(array)
    for i in range(size):
        index = (startPos + i) % size
        print(array[index])
```

You can do reverse iteration, shifting, rotating, insert, remove, etc. following the same process. The only rule is -- as you know already – just do a modulus by the dimension length for any index.

The modulus operator is very useful for indexing even when the underlying multidimensional collection object is not circular. For example, consider representing a 2D grid as a coordinate system. The center of the coordinate system is at <0, 0>. However, we would want the vertical and horizontal axes to extend towards both negative and the positive directions. Suppose the grid extends from –N to +N horizontally and –M to +M vertically. Then to find the element at any coordinate *(i,j)* we will do the following:

element = grid[(i % 2M][j % 2N].



Since, the indexes of a coordinate system do not wrap around, we need to check if *i* and *j* are within the valid range if there is any chance of invalid index input. That is the only difference.

You understand the circular array is basically a mechanism of indexing circular objects. There is no distinction between the underlying storage data structure for circular and regular arrays. Often time, you need to combine circular indexing with regular indexing. For example, if you need to do a computation about a cylindrical object then only one of its three dimensions is circular.