1) a) findSum(array, target) {
        loop through all array positions {
            if the index (target - value) exists in new array {
                store the current index, the value (index) stored in the (target - value) position, the value at current index and (target - value)
            }
            if value is not already in the new array {
                add the index of the value at the index in the new array of the value (like a hash table)
            }
        }
        return all the stored solutions
    }

    b) The motivation was to find a way to do the algorithm without looping the array more than once and therefore keeping the complexity at O(n). This meant that I would need to store all the values that I had come across and have an easy and quick (no more loops) to find those values afterwards. A hash table-like solution was a clear choice for this and the algorithm was therefore based around that data structure.

    c) The Big-O complexity is O(n) : (n*(C+C)) -›› (n). All array elements are gone through once and a constant number of operations is done on each

    d) The Big-O complexity is O(n) : (n*(C)) -›› (n). Since the algorithm is required to find all pairs that sum to the target, it will always loop n times and will not stop when a match is found.

2) a) findSum(array, target) {
        loop through all array positions {
            loop through stack {
                if a stack_value of (target - value) is found {
                    store current index, the stack_length - stack_index, the value at current index and the value at stack_index
                }
            }
            add the value of this index to the stack
        }
    }

    b) Since a stack can only be accessed one element at a time, it was unavoidable to have to loop through it and my previous approach could not be applied to this data structure.

    c) The Big-O complexity is O(n^2) : (n*(n*(C))) -›› (n^2). The algorithm needs to loop over each element for each element therefore resulting in this complexity.

    d) The Big-O complexity is O(n^2) : (n*(n*(C))) -›› (n^2). Similarly to last, the complexity stays the same since all pairs need to be found.

3) a) oddFirst(array) {
        while flag is true {
            make flag false
            loop through all elements in the array {
                if the value is even {
                    swap it with the value after it
                    make flag true
                }
            }
        }
    }

    b) I could place each element from the array to either the even or the odd array and concatenate them at the end.

4) a)

    b) [F, A, K, S, Q, null, L, null, null, Y, P, null, M, E, null, R, D, null, null, null, U, null, null, null, null]

5) a) findDepth(tree) {
        if tree is empty {
            return 0
        }
        else if tree has one value {
            return 1
        }
        else {
            recursive call on left branch
            recursive call on right branch
            return the biggest depth from the right or left branch +1
        }
    }

Programming Questions

Node List pseudo code description:

toString :	toString() {
	loop through all array pos until last {
	print value
	}
	}
	complexity: 1) O(n)
	2) O(n) (no array manipulation)

fits :	fits() {
	return true if number of values + 1 isn't 80% of total
	}
	complexity: 1) O(1)
	2) O(1) (no array manipulation)

expand :	expand() {
	if rule is double, return new array with length *2
	else, return new array with length +10
	}
	complexity: 1) O(1)
	2) O(1) (no array manipulation)

first :	first() {
	if array is non empty, return Position at index 0
	else, exception
	}
	complexity: 1) O(1)
	2) O(1) (no array manipulation)

last :	last() {
	if array is non empty, return Position at last index
	else, exception
	}
	complexity: 1) O(1)
	2) O(1) (no array manipulation)

prev :	prev() {
	for all values in array {
	if Position is found, return the one before it
	}
	throw exception if not found
	}
	complexity: 1) O(n)
	2) O(n) (no array manipulation)

next :	next() {
	for all values in array {
	if Position is found, return the one after it
	}
	throw exception if not found
	}
	complexity: 1) O(n)
	2) O(n) (no array manipulation)

set :	set() {
	for all values in array {
	if Position is found, replace it with a new one
	}
	throw exception if not found
	}

addFirst :	addFirst() {
	check if array can fit one more value (fits()) {
	create new array of same size as current
	} else {
	create a new array using expand()
	}
	set index 0 of new array to a new Position
	for all values in original array {
	place the value to index +1 in new array
	}
	return the new Position
	}
	complexity: 1) O(n)
	2) O(n) (worst case, array size *= 2)

addLast :	addLast() {
	check if array can't fit one more value (fits()) {
	create new array using expand()
	for all values in original array {
	place the value in same index in new array
	}
	}
	set the last element +1 to a new Position
	return the new Position
	}
	complexity: 1) O(1)
	2) O(n)

addBefore :	addBefore() {
	check if array can fit one more value (fits()) {
	create new array of same size as current
	} else {
	create a new array using expand()
	}
	for all values in original array {
	if the Position is found, also add the new Position
	place the value in the same index in original array
	}
	if Position is not found, exception
	return the new Position
	}
	complexity: 1) O(n)
	2) O(n)

addAfter :	addAfter() {
	check if array can fit one more value (fits()) {
	create new array of same size as current
	} else {
	create a new array using expand()
	}
	for all values in original array {
	place the value in the same index in original array
	if the Position is found, also add the new Position
	}
	if Position is not found, exception
	return the new Position
	}
	complexity: 1) O(n)
	2) O(n)

delete :	delete() {
	create new array of the same size as original array
	for all values in original array {
	if the index Position is the one to delete, skip it
	place values of the same index in the new array
	}
	if Position was not found, exception
	}
	complexity: 1) O(n)
	2) O(n)

swap :	swap() {
	store element of p1
	set element of p1 to element of p2
	set element of p2 to stored value
	}
	complexity: 1) O(1)
	2) O(1) (no array manipulation)

truncate :	truncate() {
	create new array of values +1 length
	for all values in original array {
	place values of the same index in new array
	}
	}
	complexity: 1) O(n)
	2) O(n)

setExpRule :	setExpansionRule() {
	if rule is char is recognized, change it
	}
	complexity: 1) O(1)
	2) O(1) (no array manipulation)

The truncate() method is not very useful most of the time. Not only does it need to copy the array to make it smaller, the next element to be added to the Node List will also require the copying of the entire array because it would otherwise not fit. The only reason that the truncate method should be used is after a lot of data has been deleted and a very large chunk of the array is not being used and just takes up a lot of memory.

The expansion rule of doubling the size will most likely provide better performance since it will need to increase its size a lot less than it would incrementing the space by 10 each time. This is especially true when a lot of Positions are being added to the Node List since for example, to get to 1000 Positions in size, doubling will get there in 10 doubling operations (starting at size 2) and the incrementing solution will need 100 calls.
When both cases are tested however, the difference becomes very apparent. When adding 10 000 Positions with addLast() (a method that runs in O(1) if the array doesn't need to be resized), the doubling technique was performing 100x better than the incrementing solution. However, when using the addFirst() method which runs for O(n) for all cases, the difference between the resizing technique (which was a few hundred milliseconds) becomes almost insignificant since the total execution time went up to ~27 seconds for both.

The complexities would definitely change, but not all of them. for example, any method where it was needed to look through all array elements would remain at O(n). However, for others, like addFirst() it would go from O(n) to O(1) and for swap() it would jump from O(1) to O(n).

The complexities would also change in the same way as when the manipulation of the underlying structure is not accounted for. The improvement when using a linked list structure is especially visible in methods like addAfter(), where although the complexity will technically remain at O(n), the program can stop looking for matches after finding one and therefore possibly end very quickly. In that example, an array based structure will always stay at O(n) even if the Position object is found very early in execution.