COMP 352 – Assignment 2

DUE: MONDAY October 28, 2019

for INSTRUCTOR: Dr. Hakim Mellah

George Mavroeidis (40065356)

**Question 1:**

The two cases are similar in construction, but have certain differences in their implementation. The main idea is to have stack 1 start from the left and stack 2 from the right. We encounter 2 cases that have different advantages and limitations. In both cases, methods push(e), pop(), isEmpty() and size() must be separate for both stacks.

**Case 1:** Fairness in space - > Divide the array in half. If stack 1 is full, it reaches the second half. Array is full only when both stacks are full, meaning they both have reached the middle. **Note:** N is always even

Algorithm *Stack<E>(N)* {

**Input:** integer that determines size of array and position of stack 2. Final integer recommended

**Output:** it is a constructor. No output

ptr1 ← -1;

ptr2 ← N;

arr ← new E[N];

}

Algorithm size1() {

**Input:** no input

**Output:** integer that displays the size of stack 1

return ptr1 + 1

}

Algorithm size2() {

**Input:** no input

**Output:** integer that displays the size of stack 1

return arr.length – ptr2

}

Algorithm isEmpty1() {

**Input:** no input

**Output:** boolean that displays if stack 1 is empty

return ptr1 == -1

}

Algorithm isEmpty2() {

**Input:** no input

**Output:** boolean that displays if stack 2 is empty

return ptr2 == arr.length;

}

Algorithm isFull1() {

**Input:** no input

**Output:** boolean that displays if stack 1 is full

return ptr1 == (arr.length - 1) / 2

}

Algorithm isFull2() {

**Input:** no input

**Output:** boolean that displays if stack 2 is full

return ptr2 == arr.length / 2

}

Algorithm *push1(e)* {

**Input:** element of type E to be inserted in array for stack 1

**Output:** void

if **(isFull1() == true)**

throw new Exception

ptr1++

arr[ptr1] ← e

}

Algorithm *push2(e)* {

**Input:** element of type E to be inserted in array for stack 2

**Output:** void

if **(isFull2() == true)**

throw new Exception

ptr2--

arr[ptr2] ← e

}

Algorithm *pop1()* {

**Input:** no input

**Output:** last element removed from stack 1

if **(isEmpty1() == true)**

throw new Exception

E popped ← arr[ptr1]

arr[ptr1] ← null

ptr1--

return popped

}

Algorithm *pop2()* {

**Input:** no input

**Output:** last element removed from stack 2

if **(isEmpty2() == true)**

throw new Exception

E popped ← arr[ptr2]

arr[ptr2] ← null

ptr2++

return popped

}

**Case 2:** Space is critical - > There only limitation for each stack is if they meet one another or one reaches towards the other end. Array is full when they two stacks meet in no particular index. The methods remain mostly the same, but this time the capacity for each stack is reached until their pointers meet up or the array is full. There could be a case where Stack 1 fills the whole array and Stack 2 has no elements. The following code takes into account all cases.

For writing efficiency, the code is exactly the same with Case 1. Only difference is the methods isFull1() and isFull2():

Algorithm isFull1() {

**Input:** no input

**Output:** boolean that displays if stack 1 is full

return ptr1 == ptr2 -1

}

Algorithm isFull2() {

**Input:** no input

**Output:** boolean that displays if stack 2 is full

return ptr2 == ptr1 + 1

}

**Conclusion:**

The complexity of the whole algorithm for both cases is in O(1) and Ω(1). In general, both methods use variable assignment to check and calculate sizes, insert and remove elements from array. This is achieved due to the presence of pointers

**Question 2:**

Iteratively, we can use nested for loops for checking each element if it’s bigger than the other. We scan each item in the array and check with the rest which is bigger and which one not. We assume a stack constructor already exists with an array named *arr* and a pointer named *ptr*:

Algorithm **max()** {

**Input:** no input

**Output:** void

int current ← 0

int max ← 0

for **(int i ← 0; i < arr.length; i++)** {

current ← a[i]

for **(int j = i + 1; j < arr.length; j++)** {

if (current > a[j]

max ← current

}

}

return max;

}

Obviously, this is not efficient and does not take advantage of the Stack ADT. The time complexity is O(n2). This is because each element needs to scan the array again for possible larger values. Best case is still Ω(n2), because even though the maximum element can be the first one, we cannot verify it until we scan each and every item in the array.

It is possible to create a max() method or a Stack implementation in general, where the time Complexity is O(1) and Ω(1). The following steps give a general idea of the algorithm:

* Implement a Stack constructor with an original stack array and a tracker stack array
* First element is pushed into the main and tracker stack
* The next element pushed is compared with the last element of the tracker stack
  + If it is bigger, it is entered in the tracker stack
  + If not, the tracker element is duplicated in order to
* The *max()* method will return the last element of the tracker stack
* In general, the tracker stack is sorted

Algorithm *Stack<E>(N)* {

**Input:** integer that determines size of array

**Output:** it is a constructor. No output

ptr1 ← -1;

ptr2 ← -1;

main ← new E[N];

tracker ← new E[N];

}

Algorithm isMainFull() {

**Input:** no input

**Output:** boolean that displays if main stack is full. Main will always be equal/larger than tracker

return ptr1 == N

}

Algorithm isTrackerEmpty() {

**Input:** no input

**Output:** boolean that displays if tracker is empty

return ptr2 == -1

}

Algorithm *pushFirst(e)* {

**Input:** element of type E to be inserted in array for stack

**Output:** void

**Comment:** this method adds the first element to both arrays. The reason why it is separate is to

facilitate the process of initializing a stack

ptr1++

main[ptr1] ← e

if **(isTrackerEmpty() == true)** {

ptr2++

tracker[ptr2] ← e

}

}

Algorithm *push(e)* {

**Input:** element of type E to be inserted in array for stack

**Output:** void

**Comment:**

if **(isMainFull() == true)**

throw new Exception

ptr1++

main[ptr1] ← e

ptr2++

if **(main[ptr1] > tracker[ptr2])**

tracker[ptr2] ← e

else

tracker[ptr2] ← tracker[ptr2-1]

}

Algorithm *pop()* {

**Input:** no input

**Output:** last element removed from stack

**Comment:** the element popped in tracker is also the maximum element

if **(isEmpty1() == true)**

throw new Exception

E popped ← arr[ptr1]

main[ptr1] ← null

tracker[ptr2] ← null

ptr1--

ptr2--

return popped

}

Algorithm *max()* {

**Input:** no input

**Output:** element of type E that is located at the top of tracker

return tracker[ptr2]

}

**Conclusion:**

The goal is to keep the pointers of both arrays on the same level so we can track properly the maximum element. All methods use the Stack ADT and the time complexity is O(1) and Ω(1). This method works well for small data, because we have two arrays and the space required is larger. Basically, the only trade-off is the extra array, where the worst case is when the array is already sorted and each element pushed is added to the tracker. Another great feature of this implementation is the ability to track the maximum for every element added.

**Quick Demo Explanation:**

int N = 4;

Stack<Integer> stack = new Stack<Integer>(N);

stack.pushFirst(0); // Element pushed first in both main and tracker

stack.push(3); // Element is bigger than 0, so it is pushed on tracker as well

stack.push(2); // Element is smaller than 3, so 3 is pushed again on tracker

stack.max(); // returns 3, since it is the last element added to tracker

stack.push(10); // Element is bigger than 3, so it is pushed on tracker as well

stack.max(); returns 10, since it is the last element added to tracker

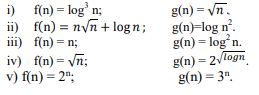
stack.push(5); // Element is smaller than 10, so 10 is pushed again on tracker

stack.max(); // still returns 10, since it is the last element added to tracker

stack.pop() // Pops 10 from both main and tracker

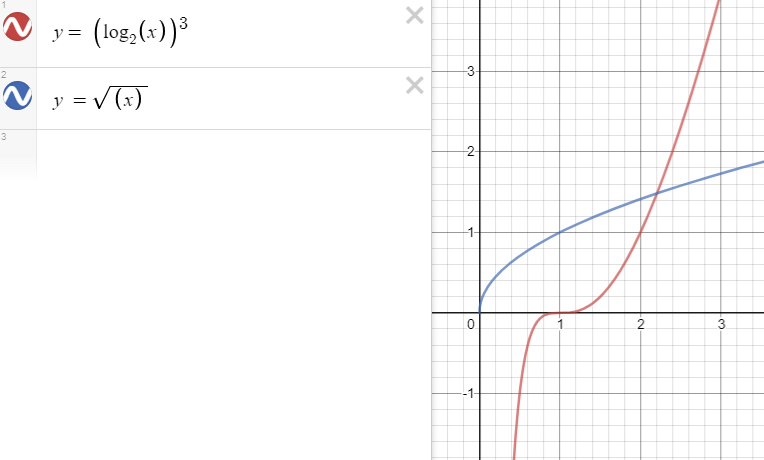
stack.max() // returns 10 again, still the maximum value

**Question 3:**

****

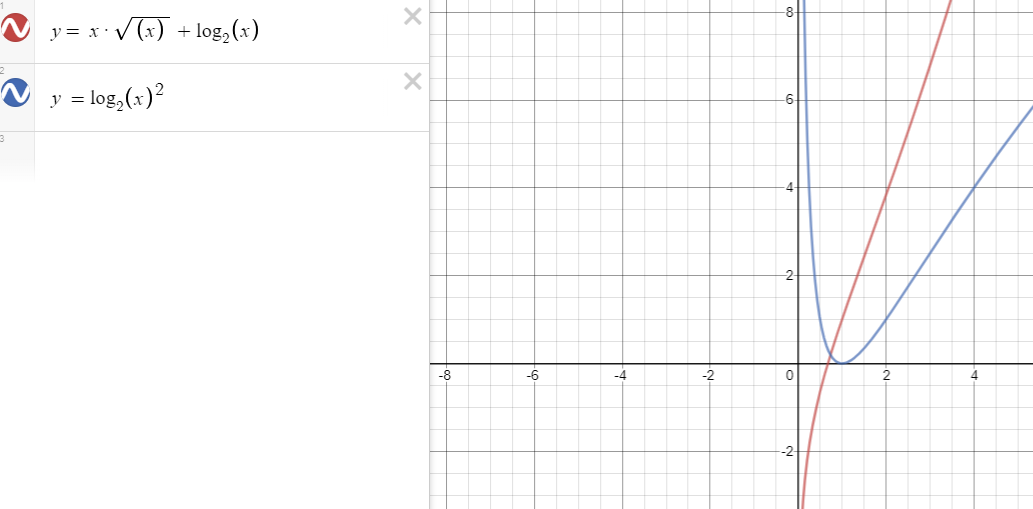
1. **Results:**

* **O(g(n))?** No – f(n) <=/= c\*g(n)
* **Ω(g(n))?** Yes – f(n) >= c\*g(n)
* **Θ(g(n))?** No – O(g(n)) does not work

****

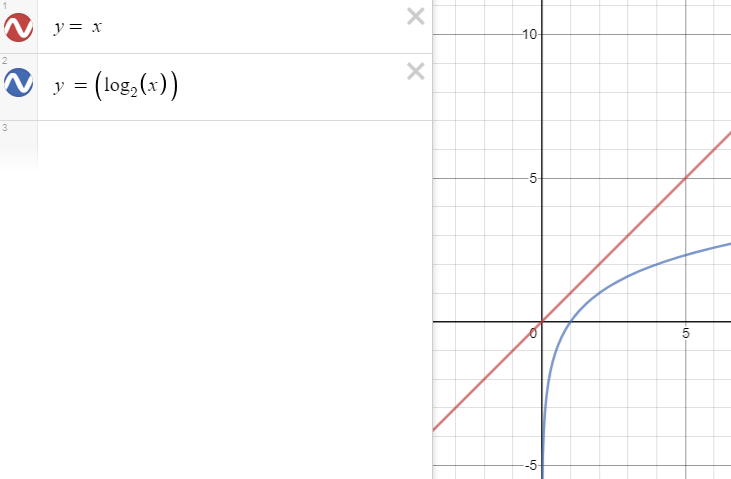
1. **Results:**

* **O(g(n))?** No – f(n) <=/= c\*g(n)
* **Ω(g(n))?** Yes – f(n) >= c\*g(n)
* **Θ(g(n))?** No – O(g(n)) does not work



1. **Results:**

* **O(g(n))?** No – f(n) <=/= c\*g(n)
* **Ω(g(n))?** Yes – f(n) >= c\*g(n)
* **Θ(g(n))?** No – O(g(n)) does not work



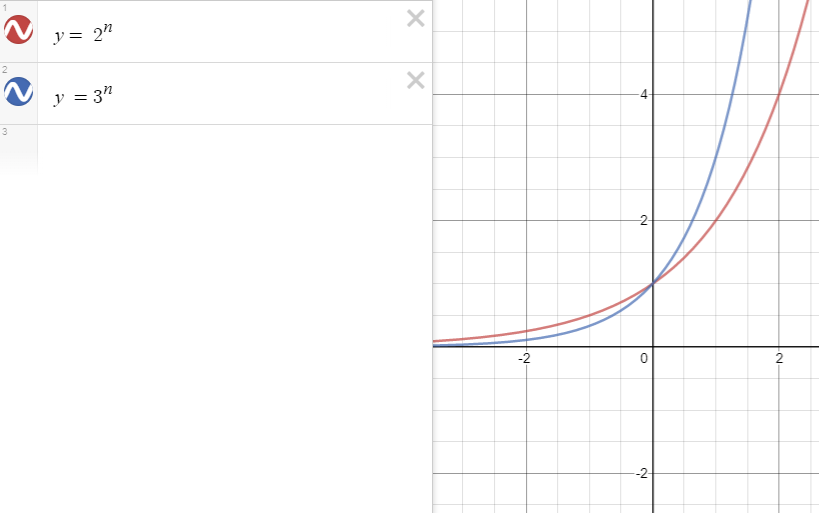
1. **Results:**

* **O(g(n))?** No – f(n) <=/= c\*g(n)
* **Ω(g(n))?** Yes – f(n) >= c\*g(n)
* **Θ(g(n))?** No – O(g(n)) does not work



1. **Results:**

* **O(g(n))?** No – f(n) <=/= c\*g(n)
* **Ω(g(n))?** Yes – f(n) >= c\*g(n)
* **Θ(g(n))?** No – O(g(n)) does not work



**Question 4:**

Algorithm *Stack<E>(N)* {

**Input:** integer that determines size of array for the stack. Final integer recommended

**Output:** it is a constructor. No output

ptr ← -1;

arr ← new int[N];

}

Algorithm size() {

**Input:** no input

**Output:** integer that displays the size of stack

return ptr + 1

}

Algorithm isEmpty() {

**Input:** no input

**Output:** boolean that displays if stack is empty

return ptr == -1

}

Algorithm isFull() {

**Input:** no input

**Output:** boolean that displays if stack is full

return ptr == arr.length

}

Algorithm *push(e)* {

**Input:** element of type E to be inserted in array for stack

**Output:** void

if **(isFull() == true)**

throw new Exception

ptr++

arr[ptr] ← e

}

Algorithm *pop()* {

**Input:** no input

**Output:** last element removed from stack

if **(isEmpty() == true)**

throw new Exception

E popped ← arr[ptr]

arr[ptr1] ← null

ptr1--

return popped

}

As we see so far, the Stack API stays the same as we have seen in a normal stack implementation. Every method keeps time complexity of O(1) and Ω(1). The only difference is one extra method that uses the stack methods to remove the duplicates of the array. This method will be implemented on the same file the array of integers is initialized.

The nested loop scans each item and compares it with each previous item to check if a duplicate is found. If yes, it is ignored. If not, the item is pushed into the stack. After the removal, the array is popped into a new array *newA* with possibly a modified size.

Algorithm removeDuplicates(int[] A) {

**Input:** Array of integers

**Output:** Array of integers, possibly modifying the input array

A[0] = push(arr[0])

for **(int i = 1; i < A.length; i++)** {

boolean isDouble ← false

for **(int j = 0; j < i; i++)** {

if **(A[i] == A[j])** {

isDouble ← true

break

}

}

if **(isDouble == false)**

arr.push(A[i])

}

int[] newA ← new int[arr.size()]

for **(int i = newA.length-1; i >= 0; i--)**

newA[i] ← arr.pop()

return newA

}

Due to the nested loops, we get time complexities of **O(n2)**. Big-Omega is **Ω(n)**. This is the best case, being that an array contains the same element for all indices, or the array only has 2 elements. The internal for loop only iterates once and breaks after the first element is compared. Also, when we create the new modified array, it only pops one element, meaning the final array will have one single element.

The space complexity in Big-O is **O(n)**, because in the worst case is when there are no duplicate elements and the size of the modified array depends on the size of the input array.