Maximilien Malderle

26562906

352 Assignment 2

Nr1.

a)

Algorithm findAbsolutePair(Arr, x)

Returns indices and values of matching pair

x = key board input

for loop1 -> 0 to smaller length -1

for loop2 -> loop1 +1 to smaller length

if (arr[loop1] == (arr[loop2] + x OR –x))

-> output indices loop1 and loop2 with associated arr values

b)

This design allows the program to search and compare every array cell with another, guaranteeing the delivery of every possibility. The down side is that the algorithm has to search every possible combination of cells of an array.

c)

Big O of this algorithm is n^2, because the worst case scenario has the algorithm executing 2 for loops which go through length-1 elements. (n-1)(n-1) closely approaches n^2.

d)

Big Omega is also n^2, because the algorithm has no choice but to execute every comparison to assure the completeness of the answer, therefore the best case resembles the worst case n^2.

Nr.2

a)

Algorithm findAbsolutePairStack(Arr, x)

Returns indices and values of matching pair

Quicksort the array.

X = from keyboard

For loop through array

-> add and subtract x from every number and push both to the stack

for loop through the array backwards

-> pop the array and binary search the associated number in the array

-> output the loop index along the returned position from the binary search along with the respective values.

b) The Reason behind this design is that all the possible values are stored in a stack, which can return those values in a constant time, then the array can be searched for those elements, this removes the constraint of the first design which forced the algorithm to iterate through every position to ensure the completeness.

c)

The big O of this algorithm can be derived from deconstruction the process. The worst case scenario for quick sort is n^2. Filling the stack is associated to looping the through the array of length n once, then operating two addition computation and two push operations. So this process requires 4n operations. The operation of searching the values goes through an array n, then perfoms 2 pops at O(1) each and 2 binary searches which with a worst case of log n. This results in 4n log n. The worst case of this algorithm is O(n^2+ 4n + 4n log n) - > O(n^2).

d)

The big omega on the other hand is much better than that of the first algorithm, the best case of a quick sort algorithm is n log n. The first loop remains constant at 4n. The second loop has a best case of Omega(4) per iteration. The best case for the algorithm approaches Omg(n log n + 4n + 4n) - > Omg( n log n).

e)

The Big O space complexity is n for the array, 2n for the stack, 1 for x, 1 for each binary search and n for the quick sort. This means that the algorithm operates with a space complexity of O(4n) which approaches O(n).

Nr.3

1. f(n) = log n^2 is O((log n)^2)
2. f(n) = n\*n^(1/2) log n is Omg(log n2)
3. c1 log^2 n <= n <= c2 log^2 n
4. f(n) = n^(1/2) is O(log 10) for c\*log10
5. f(n) = 2^(n!) is Omg(3^n)
6. fn = 2^(10\*n) is Omg(n^n)

Nr 4.

a)

Big O for this algorithm is n-1 + (n-2)^2 + n-1 this can be written down as O(n^2) this is because of the nested for loop in the middle of the algorithm. The big Omg of the algorithm is also n^2 because there are no conditions that allow the algorithm to break out of any of the loops.

b)

The first for loop creates an array of Zom = (0,0,0,0,0,0)

The double for loop produces Zom = (4,1,3,5,0,2)

The final loop produces M = (04,15,32,45, 71,98)

c)

This algorithm produces an array called Zom of numbers which will be used by the last loop to assign the spaces in M where values of A will be copied. So if position i is less than position j in array A then 1 is added to position j of array Zom, unless i is greater than j in which case i of Zom is incremented by 1. In the final loop depending on the iteration Zom will return a number which will be used by M as the address where A[i] is stored. Ultimately this algorithm sorts the array from smallest to largest.

d)

An improvement would be a quick sort algorithm, which could do the same thing with an average time of n log n instead of n^2.

e)

The space complexity can be reduced easily by swapping the values directly in the array instead of creating an hierarchy array to then use to create a new sorted array. This solution will subtract 2n from the space complexity of the algorithm.

Part 2:

Recursive Pseudo:

Function: call(int[] arr, int pos, int[] arr2)

Returns Boolean

if (pos == arr.length-1)

-> return true

end if

if (pos > arr.length-1 or pos < 0)

-> return false

end if

if (arr[pos] mod 2 == 0)

- > move = arr[pos]/2

else

-> move = arr[pos]/2 +1

end if

if (pos + move < arr.length AND pos – move > 0)

->if (arr2[pos +move] = 2)

-> return false

end if

else -> arr[pos +move] += 1 and return call(arr,pos,arr2)

end if

else if (pos + move == arr.length)

-> return true

end if

if (pos + move < arr.length AND pos – move > 0)

-> if(arr2[pos-move] == 2)

-> return false

end if

else -> arr2[pos –move] += 1 and return call(arr,pos,count)

end if

else if (pos + move == arr.length)

-> return true

end if

Stack call(int[] arr)

Returns boolean

While(loop == 0)

if (pos == arr.length -1)

-> loop = -1 and t = true and break

end if

if (pos > arr.length-1 or pos < 0)

-> loop = -1 and t = flase and break

end if

if (arr[pos] mod 2 == 0)

- > move = arr[pos]/2

else

-> move = arr[pos]/2 +1

end if

if (pos + move < arr.length AND pos – move > 0)

-> if(arr2[pos+move] == 2)

-> return false and loop = -1

end if

else -> arr2[pos + move] += 1 and stack.push(pos + move)

end if

if (pos + move < arr.length AND pos – move > 0)

-> if(arr2[pos+move] == 2)

-> return false and loop = -1

end if

else -> arr2[pos – move] +=1 and stack.push(pos + move)

end if

if(stack is not empty)

-> pos = pop stack

else

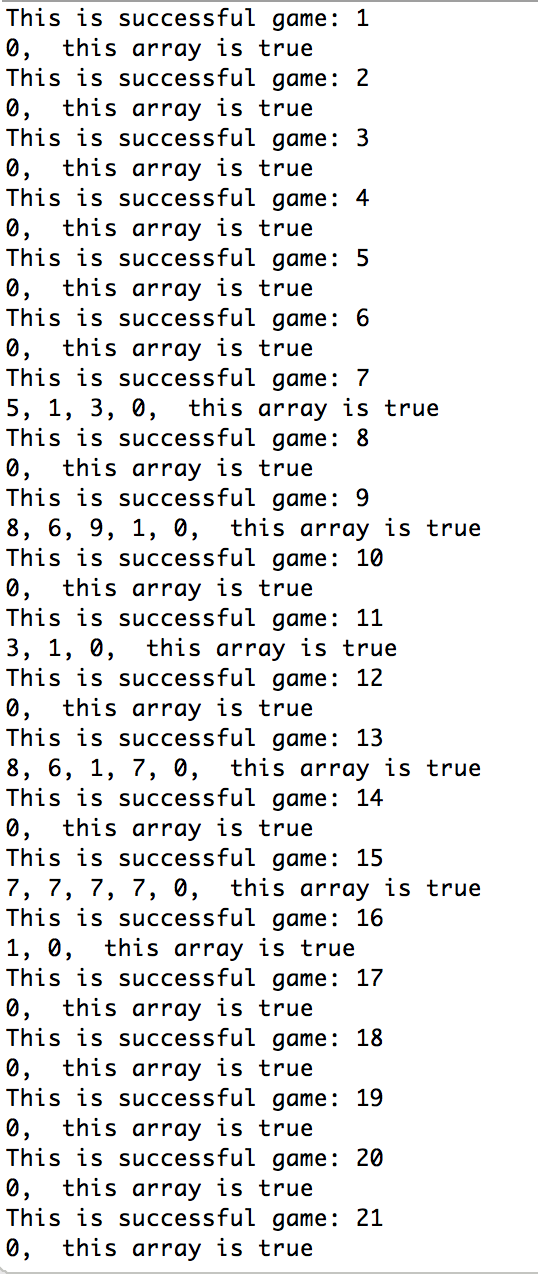
-> loop = -1 and t = false and break;

end if

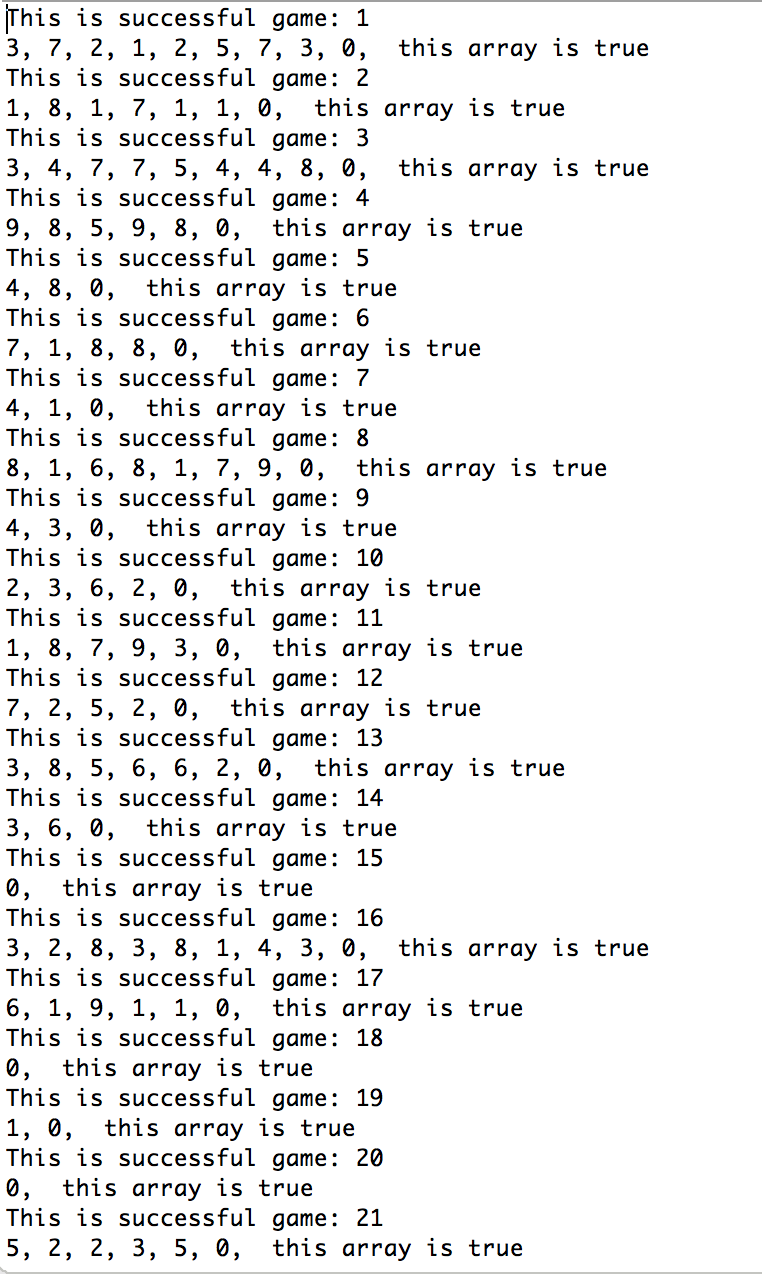
end while

return t

1. The time complexity of the recursive function is O(n+1) which is the worst case for both algorithms, because an unsolvable array will produce that many options and the combination of all possibilities will be tried to solve an array. The best case Omg(1) is the case where the array is solved at the first position. The space complexity is linear for the stack version, because all that is needed is the game array, the stack and the tracking array all of size n, so the space complexity is O(n). But is O(2n^2) for the recursive version, due to the amount of calls made to solve the worst case is n while always passing 2 arrays in the function.
2. This function is binary recursive, meaning there are two recursive calls being made, one for when the program travels right and another when it moves left. This affects the time complexity and space complexity, because if the program reaches a cell that can generate movement in 2 directions, then the function will automatically call both options, not knowing which one leads to the end more efficiently. This doubles the operations necessary when looking for a successful path and doubles the amount of memory required to keep track of both arrays.
3. I chose a stack because each time a move is possible it pushes the value and pops it at the end. This pop leads to another route, if unsuccessful all the pushed operations will run out, leaving the one at the break point creating a new path. This would be unsuccessful with a queue, because it would load all the next possibilities after the two first options. This means that the implementation would be unsuccessful, because when a split is created it would return right and then left then it would store the two options for each of the chosen paths after the previous right and left options. This means the queue would be unable of delivering information allowing the program to travel the path.
4. Recursive Function Output:



Output for Stack Implementation:



1. In this case the algorithms detect unsolvable arrays by having a duplicate array arr2, which increments a cell each time the algorithm visits the cell in the game array. Because you can select the array cell by reference we can easily update the associated spot by one. Then we implement a test if (arr2 == 2) this means that the cell has been visited before indicating a cycle, meaning not one cell brings the user to the desired end path. The time complexity is constant O(1) and space complexity of this implementation for the stack version is O(n) and for the recursive version O(n^2).