**CS 2210a**

**Assignment 1**

**Vivian Lam**

**Data Structures and Algorithms**

**R. Solis-Oba**

**October 1, 2015**

**QUESTION 1:**

f(n) ≤ c∙g(n) Ɐ n ≥ n0

By the definition of big Oh, we need to find constants c > 0 and n0 ≥1 such that

≤ c∙1 Ɐ n ≥ n0

≤ c Ɐ n ≥ n0

Since the right hand side of the inequality needs to be positive choose c= 1

≤ 1 Ɐ n ≥ n0

Note that ≤ 1 for all values n ≥ 1, so choose n0 =1

As we have found constant values c= 1 and n0 = 1 that make the inequality true ∴ we have proven that is O(1)

**QUESTION 2:**

f(n) ≤ c∙g(n) Ɐ n ≥ n0

proof by contradiction: By the definition of big Oh, we need to find constants c > 0 and n0 ≥1 such that

n ≤ c∙ Ɐ n ≥ n0

Divide both sides by to isolate *c* ≤ c Ɐ n ≥ n0≤ c Ɐ n ≥ n0

We cannot find such constants that satisfies the inequality. The function *n* is not bounded above by for any value c>0 and n0≥1. The inequality does not hold for all *n* sufficiently large with a fixed constant *c*∴ *n* is not O()

**QUESTION 3:**

f(n) ≤ c’∙g(n) Ɐ n ≥ n0’

First inequality: by the definition of big Oh, we need to find constants c’ > 0 and n0’ ≥1 such that

f(n) - c’∙g(n) ≤ 0 Ɐ n ≥ n0’

Second inequality: by the definition of big Oh, we need to find constants c > 0 and n0 ≥1 such that

k(f(n) +g(n)) ≤ c∙g(n) Ɐ n ≥ n0

Distribute *k*

k∙f(n) +k∙g(n) ≤ c∙g(n) Ɐ n ≥ n0

Since the right hand side of the inequality needs to be positive we can use the first inequality to choose c = (1+c’k+k)

k∙f(n) +k∙g(n) ≤ (1+c’k+k)∙g(n) Ɐ n ≥ n0

Expand

k∙f(n) +k∙g(n) ≤ g(n) +c’k∙g(n) +k∙g(n) Ɐ n ≥ n0

Move c’k∙g(n) to the other side

k∙f(n) - c’k∙g(n) +k∙g(n) ≤ g(n) + k∙g(n) Ɐ n ≥ n0

Cancel k∙g(n) from both sides

k∙f(n) - c’k∙g(n) ≤ g(n) Ɐ n ≥ n0

*k* is an arbitrary constant and does not matter. We know from the other inequality that f(n) - c’∙g(n) ≤ 0

0 ≤ g(n) Ɐ n ≥ n0

As we have found constant values c = (1+c’k+k) and n0 = 1 (the values for c’ and n0’ do not matter to much as they are already confirmed valid with the first inequality) that make the inequality true ∴ we have proven that k(f(n) +g(n)) is O(g(n))

**QUESTION 4:**

**Algorithm:** Balanced Array Check(*A*, *n*)

**Input:** Array *A*, and number *n* of elements in *A*

**Output:** Boolean *balancedStatus*, such that it’s *true* if *A* is balanced and *false* otherwise.

*i*←0

*j*←0

*balancedSoFar*←false

*balancedStatus*←true

**If** (*n*%2=0) **{**

**While** **(**(*i* ≤ *n*-1) & (*balancedStatus* = true)**)**{

*j*←0

*balancedSoFar*←false

**While** **(***j* ≤ *n*-1**)**{

**If** (A[*j*] = - A[*i*]){

*balancedSoFar*←true

}

*j*++

}

**If** (*balancedSoFar* = false){

*balancedStatus*←false

}

*i*++

}

**return** *balancedStatus*

**}**

**Else** **return** false **{**

**}**

**Proof of Correctness for Balanced Array Check**

**a) Algorithm Balanced Array Check terminates after a finite amount of time**

There are many instances in the above algorithm that ensures it terminates. To prove this, the algorithm first checks if the amount of elements in the array is an even number with the command **If n%2=0**. If the array has an odd number of elements it will return **false** and terminate immediately, otherwise, the algorithm will continue. Observe that the variables *i* and *j* are initialized to be zero and the Boolean variable *balancedSoFar* is set to false and *balancedStatus* is set to true. The variables *i* and *j* are used to iterate and compare elements in the array*.* These variables exist to iterate through the array and terminate the algorithm under certain conditions.

The variable *i* initially takes value zero and after each iteration of the outer **while** loop the value of *i* increases by 1. The condition of the outer **while** loop, **i ≤ n-1**,limits the number of iterations so that the loop will end as soon as *i* > *n*.

The variable *j* initially takes value zero and after each iteration of the inner **while** loop the value of *j* increases by 1. The condition of the inner **while** loop, **j ≤ n-1**,limits the number of iterations so that the loop will end as soon as *j* > *n*.

The variables *balancedSoFar* and *balancedStatus* help to ensure that the outer **while** loop will terminate if the array is unbalanced. This is possible because if the array is unbalanced, that is, if there is no additive inverse in the array, then the **If (A[j] = - A[i])** statement is ignored and the value of *balancedSoFar* remains false. Since *balancedSoFar* is false it will satisfy the **If (*balancedSoFar* = false)** statement and will set the value of *balancedStatus* to be false. This will then violate the outer loop’s **(*balancedStatus* = true)** condition, causing it to terminate.

**b) Algorithm Balanced Array Check outputs the correct answer**

This algorithm returns a Boolean value regarding whether the inputted array is balanced or not. The algorithm first checks if the amount of elements in the array is an even number because if the array has an odd number of elements then one of the elements will not have a respective additive inverse. If the array contains an odd number of elements then the algorithm will return false because the array is not balanced. If the array contains an even number of elements then the algorithm continues.

In the third last line the algorithm returns the Boolean value of *balancedStatus*. This value is dependent on whether an additive inverse of an element in the array is found. To prove that this algorithm returns true if the array is balanced, note that in the beginning *balancedStatus* takes the value true, *balancedSoFar* takes the value false and with each iteration of the outer while loop *balancedSoFar* is reset to be false. This is because *balancedSoFar* assumes that the array is currently unbalanced until the algorithm finds the additive inverse for the current element *i*. If an additive inverse is found then *balancedSoFar* will take the value true which will then ignore the **If (*balancedSoFar* = false)** statement. The algorithm will traverse through the array and as long as it finds an additive inverse in each iteration for *i* then *balancedSoFar* will take the value true, **If (*balancedSoFar* = false)** is ignored and *balancedStatus* remains and returns true.

To prove that the algorithm returns false if the array is unbalanced, consider the above except an additive inverse for the current element *i* is not found. If an additive inverse is not in the array then the statement **If (A[*j*] = - A[*i*])** is ignored and *balancedSoFar* remains false. Since *balancedSoFar* is false it satisfies the **If (*balancedSoFar* = false)** statement and will change the value of *balancedStatus* to be false and at the end the algorithm will return false.

***ii.* Worst Case and Complexity for Balanced Array Check**

The worst case for the Balanced Array Check algorithm is if the array is balanced and the algorithm traverses through the entire array. The time complexity for the worst case is *O(n2)*. This was found by dividing the algorithm into four sections, each with a constant, *k*, representing the number of primitive operations performed in that section. The constant *k1*represents the number of operations in the beginning of the algorithm where variables are initialized. *k1* also covers the outer-most **if** statement **(n%2=0)** and the **return *balancedStatus***statement as these commands will always occur and will only be performed once. *k4*covers the **return** command for the outer-most **else** statement. The reason this section is separate from *k1* is because the algorithm may or may not perform the **return** statement in *k4*. In the worst case scenario *k1*is performed and *k4* is not. The outer **while** loop performs *k2n* number of operations because it traverses *n* amount of times. The inner **while** loop performs *k3n* number of operations because it traverses *n* amount of times. Considering all of this, the total number of operations is *k1 + k2n(k3n) + k4*. This equation has an order of 2 and is thus equivalent to *O(n2).*

*i*←0

*j*←0

Total: *k1 + k2n(k3n) + k4 = O(n2)*

*k1*

*balancedSoFar*←false

*balancedStatus*←true

**If** (*n*%2=0) **{**

**While** **(**(*i* ≤ *n*-1) & (*balancedStatus* = true)**)**{

*j*←0

*balancedSoFar*←false

**While** **(***j* ≤ *n*-1**)**{

*(k3n)*

**If** (A[*j*] = - A[*i*]){

*balancedSoFar*←true

}

*j*++

}

*k2n*

**If** (*balancedSoFar* = false){

*balancedStatus*←false

}

*i*++

}

*k1*

**return** *balancedStatus*

**}**

**Else** **return** false **{**

*k4*

**}**

**QUESTION 5:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Size of input n=** | **Running time of Linear Search** | **Running time of Quadratic Search** | **Running time of Factorial Search** |
| **5** | 697 ns | 1393 ns | 570063 ns |
| **8** | 1915 ns | 2069 ns | 58833290 ns |
| **9** | 1111 ns | 2161 ns | 265365200 ns |
| **10** | 1061 ns | 2689 ns | 2626409549 ns |
| **11** | 1126 ns | 3260 ns | 27946354587 ns |
| **12** | 1194 ns | 3720 ns | 452328178671 ns |
| **100** | 2435 ns | 10508 ns |  |
| **1000** | 24480 ns | 639485 ns |  |
| **2000** | 25880 ns | 2225023 ns |  |
| **10 000** | 38648 ns | 14148922 ns |  |