**CSC 505, Spring 2018 Homework #1**

**1)**

**(a) If f(n) is O(g(n)) then g(n) is O(f(n))**

Sol : FALSE

Counter example :

Let f(n) = 5n2 + 3n

G(n) = n2

5n2 + 3n <= 5n2 + 3n2

For n>=3

5n2 + 3n <= 8n2

**f(n) is O(g(n)).**

**Now, considering g(n) is O(f(n)),**

**We get**

**N2 <= c (5n2 + 3n)**

**Divide the equation by n2**

**1<=c \* (5 +3/n)**

**(b) If f(n) and g(n) are both ≥ 1 for sufficiently large n, and f(n) is O(g(n)), then lg f(n) is O(lg g(n)).**

Sol : TRUE

Given : f(n) =O(g(n))

🡪f(n) <=c \* g(n)

* Multiply by log on both sides of equation
* Log f(n) <=log(c\* g(n))
* Log f(n) <=logc + log(g(n)) [ log(a\*b) = loga+logb ]
* Log f(n) <=logc + log(g(n)) <= c’ \* log(g(n))

if we consider a new constant c’ and multiply with log(g(n)), it’s value will be greater than a constant c added to log(g(n))

🡪 log(f(n)) <= c’ \* log(g(n))

Hence prooved.

**(c) If f(n) is O(g(n)) then g(n) is Ω(f(n)).**

Sol : TRUE

🡪Given : f(n) <=c1 \* g(n)

🡪Divide both sides by g(n)

* f(n)/g(n) <=c1
* now divide both sides by c1
* 1/c1 \* f(n) <=g(n)
* g(n) >=1/c1 \* f(n)
* Since 1/c1 is also constant, we can replace it with c2
* g(n) >=c2 \* f(n)
* This is the definition of g(n) is Ω(f(n)).
* Hence prooved

**(d) f(n) is Θ(f(n/2)).**

Sol : False

Counter example :

To prove that f(n) is Θ(f(n/2)), we need to prove that f(n) is O (f(n/2)) and f(n) is Ω (f(n/2)).

First taking Big O:

Consider f(n) = ey.

* f(n/2) = ey/2
* Assume it holds for Big O
* ey <=c ey/2
* Divide both sides by ey/2
* ey / ey/2 < = c \* ey/2 / ey/2
* ey/2 <= c
* Here we cannot bound value for c.
* Thus that f(n) is not O (f(n/2))
* Therefore f(n) is not Θ(f(n/2)).

**(e) If g(n) is o(f(n)) then f(n) + g(n) is Θ(f(n))**

Sol : TRUE

Given :

G(n) < c1 \* f(n)

🡪 divide the equation by f(n)

🡪**g(n)/f(n) < c1**

To prove **f(n) + g(n) is Θ(f(n)),** we need to prove

f(n) + g(n) is O(f(n)) and f(n) + g(n) is Ω (f(n))

* f(n) + g(n) <= c2 \* f(n)
* divide the whole equation by f(n)
* 1 + g(n)/f(n) <= c2
* 1 + (c3) <=c2 where (c3 <c1.) from above equation which is in bold
* We can see that there is bound for the value c2.
* The value of c2 should be greater than or equal to some constant
* Thus f(n) + g(n) is O(f(n))

Now consider f(n) + g(n) is Ω (f(n))

* F(n) + g(n) >=c4 \* f(n)
* Divide the equation by f(n)
* 1+ g(n)/f(n) >=c4
* 1+ c3 >=c4 from above equation which is in bold
* Here also we have a bound for c4.
* It should be smaller than some constant

Therefore f(n) + g(n) is Ω (f(n))

f(n) + g(n) is Θ(f(n)).

Hence prooved.

--------------------------------------------------------------------------------------------------------------------------------------

**2)**

**(a) Let f(n) = 2n3 + 7n2 prove that f(n) ∈ O(n3).**

Sol :

Method 1 :

Let’s suppose : 2n3 + 7n2 <=2n3 + 7n3  for all n>=7

* 2n3 + 7n2 <=9n3
* 2n3 + 7n2  ∈ O(n3)

For a constant c, where c=9 in this above case.

Method 2:

There exists c,n such that

* 2n3 + 7n2 <=cn3  for n>n0.
* Divide both sides by n3
* 2 +7/n =c
* If n=1 : 2+7=9
* C=9.
* Since we have found a bound for the value c, the above statement is correct.
* Thus 2n3 + 7n2 <=9n3

**(b) Let f(n) = 3n3 − 5n2 and prove that f(n) ∈ Ω(n3 ).**

Sol : Method 1:

There exists c, no such that

3n3 -5n2 >=c \* n3  for n>n0.

Divide both sides of equation by n3

* 3 -5/n <=c
* If n=5 :
* 3-1 <=c
* c>=2.
* There fore, we have found a bound for the value of c.
* Thus,
* f(n) ∈ Ω(n3 ) where f(n) = 3n3 − 5n2.

Method 2 :

Let 3n3 -5n2  >=3n3 -n3 for n>=5

* 3n3 – 5n2 >= 2n3
* Here we can write it as
* 3n3  -5n2 >=cn3  where c=2
* Since we have found a bound for c, this statement is true.

Thus f(n) ∈ Ω(n3 ).

Where f(n) = 3n3 − 5n2.

**(c) Let f(n) = 8n3 + 4n2 and prove that f(n) ∈ O(n4). Note that the exponent on n is 4**

Sol : Method 1:

There exists c, n such that

Let 8n3 + 4n2  <=cn4  for n>n0

Divide the equation by n4

8/n + 4/n2 <=c

Put n=2, we get

4+1 <=c

* C>=5
* Therefore we have found a bound for the value c.
* Thus this statement is valid.
* Hence f(n) ∈ O(n4) where f(n) = 8n3 + 4n2

Method 2 :

Let 8n3 + 4n2 <=8n4 + 4n4 for n>=1

Ex : put n=1.5

8\*3.375 + 4(2.25) <=8 (5.0625) + 4(5.0625)

27 +10 <=60.75

We can see that 37<=60.47

Thus we can write it as

8n3 + 4n2 <=12n4

* 8n3 + 4n2  ∈ O(n4)
* Hence proved.

Q3 )

1. **T(n) = 15T(n/4) + n2**

**Sol :**

A=15

B=4

F(n) = n2

Nlogb(a) = nlog 4 (15) =  n1.95

Now compare f(n) with Nlogb(a)

We can take case 3 of Master’s theorem,

If f(n) = Ω(nlogb(a) - €) , then T(n) = Ɵ(f(n))

Here we have f(n) =n2 and Nlogb(a) = n1.95

When we compare both, if f(n) is larger, the Master’s theorem says that T(n) = Ɵ(f(n)).

Thus we can say that case 3 applies and T(n) = Ɵ(n2).

1. **T(n) = 2T(n/2) + n lg 2n**

Sol :

A=2

B=2

F(n)= n lg2n

Nlog b (a) =n log 2(2) =n

Here we cannot compare F(n) and Nlog b (a).

So Master’s theorem cannot be applied.

The alternate method is tree/levels method.

Let T(1) =c where c is a constant

* n/2k =1
* cost/instance when there the instant size is 1 is c

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| level | No of instances | Instant size | Cost/instances | Total cost |
| 0 | 1 | n | nlog2n | N log2n |
| 1 | 21 | n/21 | n/2 log2 n/2 | Nlog2 n/21 |
| 2 | 22 | n/22 | n/22 log2 n/22 | N log2 n/22 |
| … | … | … | … | … |
| … | … | … | … | … |
| i | 2i | n/2i | n/2i log2 n/2i | N log2 n/2i |
| k | 2k | n/2k | c | C \* 2k |

c \* 2k +∑ i=0k-1 n log2 n/2i

c \* 2k + n[ ∑ i=0k-1 log2 n – log22i ]

c \* 2k + n[ (K \* log2 n) - ∑ i=0k-1  i2 ]

c \* 2k + n[ (K \* log2 n) – ( 0 + 1 + 22 + 32 + …(k-1) 2 ) ]

c \* 2k + n[ (K \* log2 n) –{ (k-1) k ( 2k-1) /6 } ]

c \* 2k + nk log2n – n[ (2k3 – k2  -2k2 – k)/6]

c \* 2k + nk log2n – n[ (2k3 – 3k2 – k)/6]

c \* 2k + nk log2n –n [ ( 2log3n – 3 log2n –log n)/6]

Arranging in descending order of terms, we have

n/3 log3n + + nk log2n + n/2 log2 n + n/6 \* (logn) + c \* 2k

Therefore the highest order term is nlog3n

T(n) = O(n log3 n)

**(c) T(n) = 4T(n/2) + n2 lg lg n**

**A= 4**

**B=2**

**F(n) = n2 lg lg n**

Nlog b (a) =n log 2(4) =n2

Now we cannot compare **F(n) with** Nlog b (a)

So Master’s theorem cannot be applied.

The alternate method is tree/levels method.

Let T(1) =c where c is a constant

* n/2k =1
* cost/instance when there the instant size is 1 is c

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| level | No of instances | Instant size | Cost/instances | Total cost |
| 0 | 1 | n | N 2 log2n | N2 log log n |
| 1 | 41 | n/21 | (n/21 )2 log2 n/2 | N2 \*(loglog n/21) |
| 2 | 42 | n/22 | (n/22 ) 2log2 n/22 | N2 \*( loglogn/22) |
| … | … | … | … | … |
| … | … | … | … | … |
| i | 4i | n/2i | (n/2i )2 log2 n/2i | N2 \*( log log n/2i) |
| k | 4k | n/2k | c | C \* 4k |

c \* 4k +∑ i=0k-1 n2 loglog n/2i

c \* 4k + n2  [ ∑ i=0k-1  log(log n – log 2i ]

c \* 4k + n2  [∑ i=0k-1  log(k– i) ]

c \* 4k + n2  [log(k-0) + log (k-1) + log (k-2)…..log(k-(k-1))]

c \* 4k + n2 [ log( k \* (k-1) \* (k-2) \* …(2) \* (1)]

c \* 4k + n2 log(k!)

Since n=2k

K=log n

* c \* 4log n + n2 log (log n!)
* rearranging the terms in decreasing order, n2 log(log n!) is the highest order term
* therefore T(n) = O(n2 log (log n!))

**(d) T(n) = 5T(n/4) + n / lg2 n**

Sol :

A=5

b=4

f(n) = n/lg2 n

nlog b (a) = nlog 4(5) = n1.15

Now we compare F(n) with Nlog b (a)

Comparing with case 1 of Master’s theorem,

F(n) is O(nlog b (a) -€)

Even if we consider € as very small value, n1.15-€  will be growing at a rate greater than f(n)

Because f(n) has the power of one over n in numerator and because of lg2 n in the denominator, it’s growth rate will be slower than n1.15-€  .

The growth rate of lg2 n is much high, and since it’s in denominator, the overall growth rate of f(n) will be diminishing compared to nlog b (a)

So assuming case 1 of Master’s theorem is applicable, we get T(n) = Θ(n1.15)

**Q4)**

**(a) Prove that the number of key comparisons during insertion sort is ≥ the number of inversions in the original array. Under what condition is the number of key comparisons equal to the number of inversions?**

Sol :

for(int j=1;j<n;j++)

{

key=arr[j];

i=j-1;

while(++compare>0 && i>0 && arr[i]>key) // comparison

{

arr[i+1]=arr[i]; // inversion

inversion++;

i--;

}

arr[i+1]=key;

* The counter for compare in while loop will determine the number of times comparision has been made in the algorithm
* The counter for inversions inside the while loop will determine the number of inversions that have been made by the algorithm.
* In best case , the number of comparisons will be n-1 where n is the number of elements in the array.
* In best case, the array is sorted in correct order, so there are no inversions required. Thus number of inversions required is 0.
* In worst case, where array elements are sorted in reverse order, number of comparisons is

∑j=1n-1 j = n(n-1)/2

* In worst case, where array elements are sorted in reverse order, number of inversions is

∑j=1n (j-1) = [ n(n-3)/2 ] + 1

* Considering both best and worst case :

n-1 >=0 ( LHS is comparisons in best case and RHS is inversions in best case) -- for n>=1

n(n-1)/2 >= [(n(n-3)/2) + 1] (LHS is comparisons in worst case and RHS is inversions in worst case) – for n>=1.

Thus always the number of comparisons is greater than inversions in insertion sort.

* If there is only 1 element in the array, then the number of comparisons is equal to number of inversions.

Substitution of n=1 in the above 2 formulas, we get ‘0’ (zero) in both inversions and comparisons.

**(b) What is the worst-case number of key comparisons as a function of the number of inversions. Your answer should be in the form I(A) + f(n), where I(A) is the number of** inversions in A and f(n) is a function of n. Prove your answer.

Sol :

The number of worst case comparisons is n(n-1)/2.

This expressing as a function of inversions is :

The number of inversions in worst case is [n(n-3)/2 + 1].

So, if we substitute, I(A) = [n(n-3)/2 + 1].

Since we know that number of comparisons is always greater that inversions.

We can write as I(A) + f(n) = C(A)

Where C(A) is number of comparisons.

Now substitute C(A) = n(n-1)/2.

We get f(n) = n-1.

Hence the function f(n) in terms of n is

n-1.

**Q5)**

**(a) Modify Merge-Sort so that it returns the number of inversions in its input (array or list). Prove that your algorithm is correct. This is much easier if you use the linked-list version and recursion invariants (see lecture notes titled recursive list algorithms on the Moodle site).**

Sol :

**(b) Suppose there are no inversions in the original list, i.e., the list is sorted to begin with. Exactly how many key comparisons does Merge-Sort do in this case? What if the list is in reverse order? You may assume that n, the number of elements, is a power of 2 to get an exact answer.**

Sol :