Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

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 $f(n) = 2n^2$

$G(n) = n^3$

$2n^2 <= c1\ n^3$

Divide the equation by n^3

- ➔ $2/n <= c1$
- ➔ $C1> = 2/n$
- ➔ If we put n=1
- ➔ C1>=2.
- ➔ Now put n=2
- ➔ C1>=1
- ➔ Considering the most restricted value for c1, we can say that c1>=2.
- ➔ Since we have found an lower bound for c, we can say that f(n) is O(g(n)).
- ➔ Where $f(n) = 2n^2$ and $G(n) = n^3$.

Now consider g(n) is O(f(n)) :

$N^3 <= c2 * 2n^2$

- ➔ Divide the whole equation by n^2
- ➔ $N <= c2 * 2$
- ➔ $C2 >= n/2$
- ➔ C2 is a constant and being bounded to the value on n.
- ➔ Since n is a variable here, we cannot get a bounding for c.
- ➔ Thus, g(n) is not O(f(n)).
- ➔ Hence proved.

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

(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 proved.

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

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

Sol : TRUE

→Given : f(n) <=c1 * g(n)

- ➔ 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 proved

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

(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) = $e^{y.}$

- ➔ f(n/2) = $e^{y/2}$
- ➔ Assume it holds for Big O
- ➔ e^{y} <=c $e^{y/2}$
- ➔ Divide both sides by $e^{y/2}$
- ➔ e^{y} / $e^{y/2}$ < = c * $e^{y/2}$ / $e^{y/2}$
- ➔ $e^{y/2}$ <= 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)).

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

(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 (1+ c3)
- ➔ 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 (1 +c3)

Therefore f(n) + g(n) is Ω (f(n))

f(n) + g(n) is Θ(f(n)).

Hence proved.

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

2)

(a) Let $f(n) = 2n^3 + 7n^2$ prove that $f(n) \in O(n^3)$.

Sol :

Method 1 :

Concider : $2n^3 + 7n^2 <= 2n^3 + 7n^3$ for all $n>=1$

- $2n^3 + 7n^2 <= 9n^3$
- $2n^3 + 7n^2 \in O(n^3)$
  For a constant c, where $c=9$ in this above case.
- Hence there exists a constant $c > 1$ and $n_0 >= 1$ for the above functions
- Hence proved.

Method 2:

There exists c,n such that

- $2n^3 + 7n^2 <= cn^3$ for $n>n0$.
- Divide both sides by n^3
- $2 + 7/n <= c$
- If n=1 then $c<=2+7$
- $C>=9$
- If n=2, $2+3.5 <= c$
- $C>=5.5$
- Taking the most restricted value for c, which is $c>=9$, we have found a bound for the value c, the above statement is correct.
- Thus $2n^3 + 7n^2 <= 9n^3$
- Hence there exists a constant $c > 1$ and $n_0 >= 1$ for the above functions
- Hence proved.

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

(b) Let $f(n) = 3n^3 - 5n^2$ and prove that $f(n) \in \Omega(n^3)$.

Sol : Method 1:

There exists c, no such that

$3n^3 - 5n^2 >= c * n^3$ for n>n0.

Divide both sides of equation by n^3

- ➔ $3 - 5/n <= c$
- ➔ If n=5 :
- ➔ $3-1 <= c$
- ➔ $c >= 2$.
- ➔ There fore, we have found a bound for the value of c.
- ➔ Thus,
- ➔ $f(n) \in \Omega(n^3)$ where $f(n) = 3n^3 - 5n^2$.

Method 2 :

Let $3n^3 - 5n^2 >= 3n^3 - n^3$ for n>=5

- ➔ $3n^3 - 5n^2 >= 2n^3$
- ➔ Here we can write it as
- ➔ $3n^3 - 5n^2 >= cn^3$ where c=2
- ➔ Since we have found a bound for c, this statement is true.

Thus $f(n) \in \Omega(n^3)$.

Where $f(n) = 3n^3 - 5n^2$.

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

(c) Let $f(n) = 8n^3 + 4n^2$ and prove that $f(n) \in O(n^4)$. Note that the exponent on n is 4

Sol : Method 1:

There exists c, n such that

Let $8n^3 + 4n^2 <= cn^4$ for n>n0

Divide the equation by n^4

$8/n + 4/n^2 <= c$

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 $f(n) \in O(n^4)$ where $f(n) = 8n^3 + 4n^2$

Method 2 :

Let $8n^3 + 4n^2 <= 8n^4 + 4n^4$ 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
$8n^3 + 4n^2 <= 12n^4$

- ➔ $8n^3 + 4n^2 \in O(n^4)$
- ➔ Hence prove

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

Q3)

(a) $T(n) = 15T(n/4) + n^2$

Sol :

A=15

B=4

F(n) = n^2

- $N^{\log_b(a)} = n^{\log_4(15)} = n^{1.95}$
- Now compare f(n) with $N^{\log_b(a)}$
- We can take case 3 of Master's theorem,
- If $f(n) = \Omega(n\log^{b(a)-\epsilon})$, then $T(n) = \Theta(f(n))$
- Here we have $f(n) = n^2$ and $N^{\log_b(a)} = n^{1.95}$
- When we compare both, if f(n) is larger, the Master's theorem says that $T(n) = \Theta(f(n))$.
- Thus we can say that case 3 applies and $T(n) = \Theta(n^2)$.

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

(b) $T(n) = 2T(n/2) + n \lg^2 n$

Sol :

A=2

B=2

$F(n)= n \lg^2 n$

$N^{\log b (a)} = n^{\log 2(2)} = n$

Here we cannot compare F(n) and $N^{\log b (a)}$.

So Master's theorem cannot be applied.

The alternate method is tree/levels method.

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

- ➔ $n/2^k = 1$
- ➔ cost/instance when there the instant size is 1 is c

level	No of instances	Instant size	Cost/instances	Total cost
0	1	n	$n\log^2 n$	$N \log^2 n$
1	2^1	$n/2^1$	$n/2 \log^2 n/2$	$N\log^2 n/2^1$
2	2^2	$n/2^2$	$n/2^2 \log^2 n/2^2$	$N \log^2 n/2^2$
…	…	…	…	…
…	…	…	…	…
i	2^i	$n/2^i$	$n/2^i \log^2 n/2^i$	$N \log^2 n/2^i$
k	2^k	$n/2^k$	c	$C * 2^k$

➔ $c * 2^k + \sum_{i=0}^{k-1} n \log^2 n/2^i$

- $c * 2^k + n[\sum_{i=0}^{k-1} \log^2 n - \log^2 2^i]$
- $c * 2^k + n[(K * \log^2 n) - \sum_{i=0}^{k-1} i^2]$
- $c * 2^k + n[(K * \log^2 n) - (0 + 1 + 2^2 + 3^2 + \ldots (k-1)^2)]$
- $c * 2^k + n[(K * \log^2 n) - \{ (k-1) k (2k-1) /6 \}]$
- $c * 2^k + nk \log^2 n - n[(2k^3 - k^2 - 2k^2 - k)/6]$
- $c * 2^k + nk \log^2 n - n[(2k^3 - 3k^2 - k)/6]$
- $c * 2^k + nk \log^2 n - n [(2\log^3 n - 3 \log^2 n - \log n)/6]$

Arranging in descending order of terms, we have

$n/3 \log^3 n + + nk \log^2 n + n/2 \log^2 n + n/6 * (\log n) + c * 2^k$

Therefore the highest order term is $n\log^{3n}$

$T(n) = O(n \log^3 n)$

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

(c) $T(n) = 4T(n/2) + n^2 \lg \lg n$

A= 4

B=2

$F(n) = n^2 \lg \lg n$

$N^{\log b (a)} = n^{\log 2(4)} = n^2$

Now we cannot compare F(n) with $N^{\log b (a)}$

So Master's theorem cannot be applied.

The alternate method is tree/levels method.

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

- ➔ $n/2^k = 1$
- ➔ cost/instance when there the instant size is 1 is c

level	No of instances	Instant size	Cost/instances	Total cost
0	1	n	$N^2 \log^2 n$	$N^2 \log \log n$
1	4^1	$n/2^1$	$(n/2^1)^2 \log^2 n/2$	$N^2 *(\log \log n/2^1)$
2	4^2	$n/2^2$	$(n/2^2)^2 \log^2 n/2^2$	$N^2 *(\log\log n/2^2)$
…	…	…	…	…
…	…	…	…	…
i	4^i	$n/2^i$	$(n/2^i)^2 \log^2 n/2^i$	$N^2 *(\log \log n/2^i)$
k	4^k	$n/2^k$	c	$C * 4^k$

$c * 4^k + \sum_{i=0}^{k-1} n^2 \log \log n/2^i$

$c * 4^k + n^2 \ [\sum_{i=0}^{k-1} \log(\log n - \log 2^i]$

$c * 4^k + n^2 \ [\sum_{i=0}^{k-1} \log(k - i) \ ]$

$c * 4^k + n^2 \ [\log(k-0) + \log (k-1) + \log (k-2)\ldots..\log(k-(k-1))]$

$c * 4^k + n^2 \ [\log(k * (k-1) * (k-2) * \ldots(2) * (1)]$

$c * 4^k + n^2 \log(k!)$

Since $n=2^k$

K=log n

- ➔ $c * 4^{\log n} + n^2 \log (\log n!)$
- ➔ rearranging the terms in decreasing order, $n^2 \log(\log n!)$ is the highest order term
- ➔ therefore $T(n) = O(n^2 \log (\log n!))$

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

(d) $T(n) = 5T(n/4) + n / lg^2 n$

Sol :

A=5

b=4

$f(n) = n/lg^2 n$

$n^{\log b (a)} = n^{\log 4(5) =} n^{1.15}$

Now we compare F(n) with $N^{\log b (a)}$
Comparing with case 1 of Master's theorem,

F(n) is $O(n^{\log b (a) -€})$

→Even if we consider € as very small value, $n^{1.15-€}$ will be growing at a rate greater than f(n)

Ex : consider € = 0.15.

$n^{\log b (a) -€} = n^1$, this function still grows faster than $n/lg^2 n$

We can even plot the values and see that $n^{\log b (a) -€}$ grows faster than f(n).

→Even if we change € by small values, the growth rate of f(n) is still smaller than $n^{\log b (a)}$.

Now the case 1 of Master theorem where T(n) = T(n) = $\Theta(n^{1.15})$

→Because f(n) has the power of one over n in numerator and because of $lg^2 n$ in the denominator, it's growth rate will be slower than $n^{1.15-€}$ even for small values for €.

→The growth rate of $lg^2 n$ is much high, and since it's in denominator, the overall growth rate of f(n) will be diminishing compared to $n^{\log b (a)}$

→So assuming case 1 of Master's theorem is applicable, we get T(n) = $\Theta(n^{1.15})$

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

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 :

Algorithm for insertion sort :

```
For j=1 to n
  key=arr[j];
  i=j-1;
  while(i>=0 && ++compare>0 && arr[i]>key)          // comparisons

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

  arr[i+1]=key;
```

- ➔ The counter for compare in while loop will determine the number of times comparison 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 (where elements are sorted in correct order) , the number of comparisons will be n-1 where n is the number of elements in the array.

  This is because $\sum_{j=1}^{n-1} 1 = 1 + 1 + 1 \ldots (n-1 \text{ times}) = n-1$

- ➔ In best case (where elements are sorted in correct order), 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

  $\sum_{j=1}^{n-1} j = 1 + 2 + \ldots + (n-2) + (n-1) = n(n-1)/2$

- ➔ In worst case, where array elements are sorted in reverse order, number of inversions is $\sum_{j=1}^{n-1} j = 1 + 2 + \ldots + (n-2) + (n-1) = n(n-1)/2$

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

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

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

→ In worst case, the number of comparisons is equal to number of inversions.

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

(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 :

Here we want a worst case (maximum) for the function f(n).

Considering best case :

I(A) – No of inversions

Let C(A) equal to number of comparisons

I(A) = 0

C(A) = n-1

So I(A) + f(n) = C(A)

- ➔ F(n) = C(A) - I(A)
- ➔ F(n) = (n-1) – (0)
- ➔ F(n) = n-1

Consider the worst case :
I(A) = n(n-1)/2
C(A) = n(n-1)/2

So I(A) + f(n) = C(A)

- ➔ F(n) = C(A) - I(A)
- ➔ F(n) = n(n-1)/2 – n(n-1)/2 =0
- ➔ F(n) = 0

Since we are looking for upper bound on f(n), we consider best case where f(n) = n-1.

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

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 :

The algorithm for Merge sort using linked list representation is :

```
Mergesort(L) is

Inversion1,inversion2,inversion3= 0
List_A, inversion1 +=MergeSort(firstHalf(L))
List _B, inversion 2+= MergeSort(secondHalf(L))          // inversion while calling first half of list
List_C, Inversion3+= Merge( List_A,List_B)               // inversion while calling second half of list
Return inversion1+inversion2+inversion3
end MergeSort


Merge(L1, L2) is
        Int inversion=0
        if L1 is empty then return L2, inversion
        else if L2 is empty then return L1,inversion
        else if first(L2) < first(L1) then

                inversion ++
                Merged_List, Merged_inversion = Merge(L1, rest(L2))
        // We return the list and the inversion count in each instance
                return (first(L2) + MergedList) , (inversion + Merged_inversion)


        Else
                Merged_list, Merged_inversion = Merge(rest(L1), L2)

                 return (first(L1) + Merged_list) , (inversion + Merged_inversion)


end Merge
```

Correctness of the algorithm :

The algorithm works correctly because it will count the number of inversion which are happening while sorting 2 sublists independently.

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

And while combining the 2 sorted sublists, we again compare the elements of 2 submists before merging.
So

(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 :

Number of comparisons in merge sort :

Case 1 :

When an array is sorted in ascending order :

- When there are n elements in the list where $n=2^k$
- The first element in list 1 would be less than first element in list 2
- The same is with all elements in list 1 and list 2
- Therefore in this manner the number of comparisons is n/2 in merge step.
- So considering a recursive call , let C(n) be the number of comparisons.
- C(n) = 2 C(n/2) + n/2
- Since we divide 2 times a list into 2 halves, we get n/2 as the new size in recursive call.
- These 2 recursive calls are from the Merge sort step.
- And since this is done twice we have 2 * C(n/2).

Case 2 :

- ***When an array is sorted in reverse order :***
- When there are n elements in the list where $n=2^k$
- The first element in the list is greater than the first element of list2 .
- The same is with all elements in list 1 and list2.
- Therefore in this manner the number of comparisons is n/2 in merge step.
- So considering a recursive call , let C(n) be the number of comparisons.
- C(n) = 2 C(n/2) + n/2
- Since we divide 2 times a list into 2 halves, we get n/2 as the new size in recursive call.
- These 2 recursive calls are from the Merge sort step.
- And since this is done twice we have 2 * C(n/2).

Team members :
Harika Malapaka (hsmalapa)
Bhanu Sreekar Reddy karumuri (bkarumu)

Now if we add the 2 recursive calls and merge step comparisons, we get :

$C(n) = 2\ C(n/2) + n/2$

Taking forward iteration method :

$C(1) = 0$ base case

Because if we have only 1 element, the number of comparisons is 0.

$C(2^1) = 2\ C(1) +1 = 2\ (0) + 1 = 1$

$C(2^2) = 2\ C(2) + 2 = 2\ (1) + 2 = 4$

$C(2^3) = 2\ C(4) + 4 = 2\ (4) + 4 = 12$

….

$C(2^k) = K * 2^{k-1}$

$C(2^k) = \log n * 2^k /2$

$C(2^k) = \log n * n/2$ since $n= 2^k$

There fore number of comparisons in merge sort is ***n/2 logn*** when the elements are sorted in both ascending or descending order.