due Friday, 26 January 2018, at 11:45 PM

Note: If you choose to work with a partner on this homework, you need to register on GradeScope by 5:00 PM on January 19. Please do the following exactly as written

- Download the h1 registration assignment from Piazza.
- **Type** the names and email addresses of both partners into the appropriate spaces. This is easy to do with the LaTeX source, but you can also edit the pdf file directly.
- *One of the partners* should submit the completed form to GradeScope – the assignment name is Partner registration (Homework 1).
- When you submit, indicate who your partner is. GradeScope has detailed tutorials on just about everything on their support site. Click on the Account icon and look at various topics in the Getting Started guide.

When you submit your assignment solutions please take the time to indicate the starting page of each answer. To make life easy for the TA's put the answer to each problem on a separate page (you may put multiple parts of the same problem on one page)

Group Members	**Student ID**	**Unity ID**
K. Sai Sri Harsha	**200207493**	**skanama**
Akshay Nalwaya	**200159155**	**analway**

Question Number	**Start Page**	**End Page**
1	2	3
2	4	5
3	6	9
4	10	11
5	12	14

[Total points: 30]

1. **[5 points]** ***Understanding asymptotic notation***

   For each of the following statements, prove that it is true or give a counterexample to prove that it is false. If you give a counterexample, you still have to prove that your example is, indeed, a counterexample.

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

   Sol: False. This can be proved by giving a counter example.

   Let $f(n) = n^2$ and $g(n) = n^3$

   We know that for all c =1 and n >= 1, 0 <= f(n) <= c * g(n)

   Therefore, we can say that f(n) ϵ O(g(n)).

   But, consider the following expression 0 <= g(n) <= c * f(n)

   Substituting the functions, we obtain

   $0 <= n^3 <= c * n^2$

   Dividing by 'n^2', we get

   0 <= n <= c

   Clearly, there's no constant 'c' which satisfies the above condition as the value of 'n' varies.

   Therefore, if **f(n) ϵ O(g(n)), then g(n) ∉ O(f(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: Given, f(n) ϵ O(g(n))

   Therefore, we have 0 <= f(n) <= c * g(n)

   Taking $\log_2$ on both the sides we have

   lg f(n) <= lg(c * g(n)) (if a < b, then lg a < lg b)

   lg f(n) <= lg c + lg g(n)

   Therefore, for some constant c_1 we have

   lg f(n) <= lg c + lg g(n) <= c_1 * lg g(n)

   Therefore, **lg f(n) ϵ O(lg g(n))**

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

   Sol: Given f(n) ϵ O(g(n))

   Therefore, 0<= f(n) <= c1 * f(n)

   Upon dividing by c1 we get

   0 <= (1/c1) * f(n) <= g(n)

   Thus, we have g(n) >= c2 * f(n) >= 0 (for c2 = 1/c1)

Therefore, ***g*(*n*) is Ω(*f*(*n*)).**

(d) *f*(*n*) is Θ(*f*(*n/*2))

Sol: False. Let $f(n) = e^n$

$f(n/2) = e^{n/2}$.

For the claim to be true, we need

$c_1 * f(n/2) <= f(n) <= c_2 * f(n/2)$

With $f(n) = e^n$

we can't find a pair of values for c1 and c2, such that

$c_1e^{n/2} <= e^n$ and $c_2e^{n/2}>=e^n$ these conditions hold true for some finite c_1 and c_2.

So, **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: Given g(n) ϵ o(f(n)).

Therefore, we have $0 <= g(n) < c * f(n)$

Adding the expression with f(n), we obtain

$f(n) <= g(n) + f(n) < (c + 1) * f(n)$

Clearly, we have, $c1 * f(n) <= g(n) + f(n) < c2 * f(n)$ (for c1 = 1 and c2 = c + 1)

Therefore, **f(n) + g(n) ϵ Θ(*f*(*n*))**

1 point for each part

2. **[6 points] *Understanding asymptotic notation***

In proving big-oh and big-omega bounds there is a relationship between the *c* that is used and the smallest n_0 that will work (for big-oh, the smaller the *c*, the larger the n_0; for big-omega, the larger the *c*, the larger the n_0). In each of the following situations, describe (the smallest *integer*) n_0 as a function of *c*. You'll have to use the ceiling function to ensure that n_0 is an integer. Your solution should also give you a lower bound (for big-oh) or an upper bound (for big-omega) on the constant *c*.

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

Sol: Given $f(n) = 2n^3 + 7n^2$

To prove $f(n) \in O(n^3)$, we have $f(n) <= c * n^3$

$2n^3 + 7n^2 <= c * n^3$

$7n^2 <= (c - 2) * n^3$

$7 <= (c – 2) * n$

$n >= 7 / (c – 2)$

Therefore, $\mathbf{n_0 = \lceil 7 / (c – 2)\rceil}$

As n_0 should always be positive, we have **$c > 2$**

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

Sol: To prove $f(n) \in O(n^3)$, we have $f(n) <= c * n^3$

$3n^3 – 5n^2 >= c * n^3$

$(3 – c) * n^3 >= 5 * n^2$

$n >= 5 / (3 – c)$

Therefore, $\mathbf{n_0 = \lceil 5 / (3 – c)\rceil}$

As n_0 should always be positive, we have **$c < 3$**

(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: To prove f(n) ∈ O(n^4), we have f(n) <= c * n^4

$8n^3 + 4n^2$ <= c * n^4

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

c * n^2 - 8*n – 4 >= 0

Intuitively, we cannot have c <= 0 as n is always positive. Therefore, **c > 0**

here we have n >= $(8 + \frac{\sqrt{64+16*c}}{2c})$

and n <= $(8 - \frac{\sqrt{64+16*c}}{2c})$

n_0 = $\lceil 8 + \frac{\sqrt{64+16*c}}{2c} \rceil$

2 points for each part

3. **[7 points, not evenly distributed]** ***Purpose: application of the Master Theorem and ability to recognize and deal with situations where it does not apply***
   Give Θ bounds for $T(n)$ in each of the following recurrences. Assume $T(n)$ is constant for small values of n. In situations where the solution is based on the Master Theorem,[1] please state which case of the Master Theorem applies. If the Master Theorem does not apply, solve the recurrence using the tree/levels method. You can choose the value of n at which $T(n)$ is a constant, i.e., you can choose the base case for the recursion to be what ever is most convenient. Also, you can let $n = b^k$ for some k.

(a) $\boldsymbol{T(n) = 15T(n/4) + n^2}$

Sol: T(n) = 15 T(n/4) + n2

Comparing the equation with Master Theorem equation, we have,

a = 15, b = 4, f(n) = n2

$\log_b a = \log_4 15 = 1.95$

We can write, f(n) = Ω(nlog$_4$15 + ϵ), where ϵ = 0.05,

And, (a)*f(n/b) = 15*(n^2/16) = (0.9375)n^2 <= c*n^2 for c = 2.

Hence, it satisfies both the conditions for Case-3 of Master Theorem,

So, T(n) = Θ(f(n)) = Θ(n^2)

[1] There is a more general version of the Master Theorem available on the internet. The term "Master Theorem" here refers to the one in the textbook. If you use the more general version, you have to prove it yourself.

(b) ***T*(*n*) = 2*T*(*n*/2) + *n*lg² *n***

Sol:

Comparing the equation with Master Theorem equation, we have,
a = 2, b = 2, f(n) = n $\lg^2 n$
$\log_b a = \log_2 2 = 1$
f(n) / $n^{\log_b a}$ = $\lg^2 n$ which is not polynomially larger.
So, Master theorem cannot be applied in this case. Hence, using recurrence tree method.

Level	# of Instances	Instance Size	Cost per Instance	Total Cost
0	1	n	n $\lg^2 n$	n $\lg^2 n$
1	2	n/2	(n/2) $\lg^2(n/2)$	(n) $\lg^2(n/2)$
2	4	n/4	(n/4) $\lg^2(n/4)$	(n) $\lg^2(n/4)$
		…		
i	2^i	$n/2^i$	$(n/2^i) \lg^2(n/2^i)$	(n) $\lg^2(n/2^i)$
		…		
k	2^k	$n/2^k$ = 1	$(n/2^k) \lg^2(n/2^k)$ = 0	0

[Assuming $n = 2^k$]

Adding up the costs:

$T(n) = n \sum_{i=0}^{k-1} \lg^2(n/2^i)$
$= n * k * \lg^2(2^k) – n \sum_{i=0}^{k-1} \lg^2(2^i)$
$= n * k^3 – n \sum_{i=0}^{k-1} (i^2)$
$= n * k^3 – n(k-1)(k)/2$
$= n * k^3 – n(k^2 – k)/2$
$= n * \lg^3 n – n(\lg^2 n – \lg n)/2$ *[Using k = lg n]*

Therefore, upper bound of this expression can be concluded as,
$T(n) = \Theta(n \lg^3 n)$

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

Comparing the equation with Master Theorem equation, we have,
$a = 4$, $b = 2$, $f(n) = n^2 \lg\lg n$
$\log_b a = \log_2 4 = 2$
$f(n) / n^{\log_b a} = \lg\lg n$, which is not polynomially larger.
So, Master theorem cannot be applied in this case. Hence, using recurrence tree method.

Level	**# of Instances**	**Instance Size**	**Cost per Instance**	**Total Cost**
0	1	n	$n^2 \lg\lg n$	$n^2 \lg\lg n$
1	4	$n/2$	$(n/2)^2 \lg\lg (n/2)$	$(n)^2 \lg\lg (n/2)$
2	4^2	$n/4$	$(n/4)^2 \lg\lg (n/4)$	$(n)^2 \lg\lg (n/4)$
…				
i	4^i	$n/2^i$	$(n/2^i)^2 \lg\lg (n/2^i)$	$(n)^2 \lg\lg (n/2^i)$
…				
k	2^k	$n/2^k$ $= 1$	$(n/2^k)^2 \lg\lg (n/2^k)$ $= 0$	0

[Assuming $n = 2^k$]

Adding up the costs,
$T(n) = n^2 \sum_{i=0}^{k-1} \lg\lg (n/2^i)$
$\quad = n^2 * k * \lg\lg 2^k - n^2 * \sum_{i=0}^{k-1} \lg\lg (2^i)$
$\quad = n^2 * k * \lg k - n^2 * \sum_{i=0}^{k-1} \lg i$
$\quad = n^2 * k * \lg k - n^2 * \lg (k-1)!$
$\quad = n^2 * \lg n * \lg\lg n - n^2 * \lg (k-1)!$

So, $\mathbf{T(n) = O(n^2 \lg(\lg n!))}$

(d) $\boldsymbol{T(n) = 5T(n/4) + n/\lg^2 n}$

$T(n) = 5T(n/4) + n / \lg^2 n$
$\log_b a = \log_4 5 = 1 + \epsilon$
$n^{\log_b a} = n^{1+\epsilon} = n * n^{\epsilon}$
$f(n) = n / \lg^2 n$

It is clear that f(n) is smaller than $n^{\log_b a}$, hence it can be concluded that it belongs to the case-1 of Master Theorem.
Therefore, **$T(n) = \Theta(n^{\log_4 5})$**

Definition. An *inversion* in a permutation π over $\{1,\ldots,n\}$ is a pair of integers i and j such that $i < j$, but $\pi(i) > \pi(j)$. More concretely: in an array of n elements, it's a pair i,j with $i < j$ and $A[i] > A[j]$.

4. **[4 points]** ***Purpose: understanding the concept of an inversion and applying it to detailed analysis of a sorting algorithm***

   (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: Pseudo Code to count the number of inversions during insertion sort:

```
1)  INVERSIONS = 0, COMPARISONS = 0
2)  FOR I IN 2 TO N:
3)      FOR J IN (I-1) TO 1:
4)          COMPARISONS++
5)          IF(ARR[J] > ARR[J+1]):
6)              SWAP(ARR[J], ARR[J+1])
7)              INVERSIONS++
8)          ELSE
9)                  BREAK
10)         END IF
11)     END FOR
12) END FOR
13)  RETURN (COMPARISONS, INVERSIONS)
```

(i) Clearly, from the above pseudo code, we can observe that the number of inversions is based on the result of comparisons unlike the actual number of comparisons. The highest number of comparisons is $\frac{n*(n-1)}{2}$ when the array is sorted in descending order. And, lowest number of comparisons is (n-1) when the input array is sorted in ascending order. The number of inversions when the array is sorted in descending order is $\frac{n*(n-1)}{2}$.

(ii) The number of inversions = The number of comparisons = $\frac{n*(n-1)}{2}$

This occurs when the input array is sorted in descending order.

(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 worst-case number of key comparisons = I(A) = $\frac{n*(n-1)}{2}$

In this case, f(n) = 0 (lower bound for f(n))

The best-case number of key comparisons = n – 1 (in case of a sorted array)

And I(A) = 0 in this case.

Therefore, number of key comparisons = I(A) + (n – 1)

In this case, f(n) = n – 1 (upper bound for f(n))

3 points for (a), 1 point for (b)

5. **[6 points]** ***Purpose: understanding merge sort; demonstrating abstract problem solving ability needed in CSC 505; also another analysis involving inversions***
   (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: Pseudo code for Merge Sort

```
MERGE_SORT(L):
1)  IF L IS EMPTY OR REST(L) IS EMPTY
2)      RETURN (0, L)
3)  END IF
4)  (INV_COUNT_LEFT_HALF, FIRST_HALF(L)) = MERGE_SORT(FIRST_HALF(L))
5)  (INV_COUNT_RIGHT_HALF, SECOND_HALF(L)) = MERGE_SORT(SECOND_HALF(L))
6)  (INV_COUNT_MERGE, L) = MERGE(FIRST_HALF(L), SECOND_HALF(L))
7)  RETURN (INV_COUNT_LEFT_HALF + INV_COUNT_RIGHT_HALF +
    INV_COUNT_MERGE, L)

MERGE(L1, L2):
1)  IF L1 IS EMPTY:
2)      RETURN (0, L2)
3)  END IF
4)  IF L2 IS EMPTY:
5)      RETURN (0, L1)
6)  END IF
7)  L = EMPTY LIST
8)  INV_COUNT = 0
9)  WHILE REST(L1) IS NOT EMPTY AND REST(L2) IS NOT EMPTY:
10)     IF FIRST(L1) < FIRST(L2):
11)         APPEND(FIRST(L1), L)
12)         L1 = REST(L1)
13)     ELSE
14)         APPEND(FIRST(L2), L)
15)         L2 = REST(L2)
16)         INV_COUNT = INV_COUNT + LENGTH(REST(L1))
17)     END IF
18) END WHILE
19) RETURN (INV_COUNT, L)
```

Proof of Correctness of MERGE(L1, L2):
Case 1: If one of the lists is empty
We check if one of the list is empty, and if found true, we return the other list and the number of inversions as 0. This is correct.

Case 2: None of the lists (L1 and L2) are empty.
In this case, we are comparing the first element of L1 with the first element of L2, the smaller of the two is appended to the resulting list while the pointer is moved forward to point next element in this list. In the next iteration, the same process is repeated, and hence this ensures that in the merged list, new element appended is always less than all the previously appended elements. Thus, we can say that new merged list is sorted in ascending order.
In addition to this, each time an element in L2 (denoting the right half of original input list) is found to be less than element in L1 (denoting the left half of original input list), this is a case of inversion and hence we need to update the value of number of inversions (INV_COUNT).
In the statement INV_COUNT = INV_COUNT + LENGTH(REST(L1)), the number of elements remaining in L1 are added to the inversion count variable because this value denotes the number of inversions that will take place and have this current element as smaller element. Therefore, this case is also handled correctly.

Proof of Correctness of MERGE_SORT(L):

Case 1: The input list is empty or contains a single element
(i) If list is empty, then we just need to return the empty list, which is done in the algorithm.
(ii) If there is only one element in the list, then the list is obviously sorted, so we need to return the same list. This is handled by the algorithm.
Hence, algorithm is correct.

Case 2: The input list contains more than 1 elements
Algorithm recursively partitions the list into two halves until there are only unit sized partitions, it is evident as recursion will continue to run till number of remaining elements is 0 or 1 in the partitioned list which is the base case of MERGE_SORT(L). Hence, this algorithm correctly partitions the list.
Once the partitions are obtained, the MERGE(L1,L2) routine is called which starts merging the two lists. Since we have already proved correctness of MERGE above, we can conclude that it returns the expected output. Now we have merged list and the number of inversions done during this process, and thus we can claim that algorithm works correctly.

(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.

4 points for part (a): 3 for an algorithm that works, 1 for correctness, 2 points for part (b)

Sol***:***

(i) The list is sorted in ascending order:

In case of a sorted array, in the merge function, only the first element of the first subpart is compared to all the elements in the second subpart. Therefore, if the size of the array is 'n', then total cost outside the recursive function would be 'n/2'. Thus, the following recurrence:

$$T(n) = 2T(n/2) + n/2$$

Level	**# of Instances**	**Instance Size**	**# of Comparisons**
0	1	n/2	n/2
1	2	n/4	n/2
2	4	n/8	n/2
		...	
i	2^i	$n/2^{i+1}$	n/2
		...	
k	2^k	$n/2^{k+1}$	n/2

Adding the total number of comparisons, *[Assume $n = 2^k$]*

Total number of key comparisons = $\sum_{i=1}^{k}(n/2)$

= $(k*n)/2$

Substituting k = lg n, we get,

Number of key comparisons = $\frac{n \lg n}{2}$

(ii) The list is sorted in descending order:

In case of an array sorted in descending order, in the merge function, the first element of the second subpart is compared to all the elements of the first subpart. Therefore, if the size of the array is 'n', then the total cost outside the recursion would be 'n/2'. Thus, the following recurrence:

$$T(n) = 2T(n/2) + n/2$$

Using the above solution, we get the **number of key comparisons =** $\frac{n \lg n}{2}$