CSCI 570 - Fall 2016 - HW 2 Solution

1. Which of the following statements are **true**? **Answer: (c)(e)**

   (a) False. Consider for instance $f(n) = n$ and $g(n) = n^2$. Clearly $n + n^2$ is not in $\Theta(n)$.

   (b) False. The $O(n)$ notation merely gives an upper bound on the running time. To claim one is faster than other, you need an upper bound for the former and a lower bound (Ω) for the latter.

   (c) True. The dominant term is $4n$, which is obviously both $O(n)$ and $\Omega(n)$

   (d) False. For instance, G being a tree is a counterexample.

   (e) True.

2. Reading Assignment: Kleinberg and Tardos, **Chapter 2 and 3**.

3. Solve Kleinberg and Tardos, **Chapter 2, Exercise 3**.

   In ascending order of growth, the list is $f_2(n), f_3(n), f_6(n), f_1(n), f_4(n), f_5(n)$.

4. Solve Kleinberg and Tardos, **Chapter 2, Exercise 4**.

   In ascending order of growth, the list is $g_1(n), g_3(n), g_4(n), g_5(n), g_2(n), g_7(n), g_6(n)$.

5. Solve Kleinberg and Tardos, **Chapter 2, Exercise 5**.

   Assume that functions $f(n)$ and $g(n)$ take nonnegative values.

   (a) False. Consider for example $f(n) = 2, \forall n$ and $g(n) = 1, \forall n$.

   Clearly, $f(n) = \mathcal{O}(g(n))$. Observe that $\log_2(f(n)) = 1, \forall n$ and $\log_2(g(n)) = 0, \forall n$. Hence $\log_2(f(n)) \neq \mathcal{O}(\log_2(g(n)))$.

   Note: If we further add the constraint that $\exists N$ such that $g(n) \geq 2, \forall n > N$, then the statement becomes true.

   (b) False. Consider for example $f(n) = 2n$ and $g(n) = n$. Clearly 4^n is not $\mathcal{O}(2^n)$.

(c) True. Since $f(n) = \mathcal{O}(g(n))$, there exists positive constants c and n_0 such that $f(n) \leq cg(n), \forall n \geq n_0$. This implies $f(n)^2 \leq c^2 g(n)^2, \forall n \geq n_0$, which in turn implies that $f(n)^2 = \mathcal{O}(g(n)^2)$.

6. Solve Kleinberg and Tardos, **Chapter 2, Exercise 6**.

(a) The outer loop of the given algorithm runs for exactly n iterations, and the inner loop of the algorithm runs for at most n iterations. Therefore, the line of code that adds up array entries $A[i]$ through $A[j]$ (for various is and js) is executed at most n^2 times. Adding up any array entries $A[i]$ through $A[j]$ takes $O(j - i + 1)$ operations, which is $O(n)$. Store the results in $B[i, j]$ requires only constant time. Therefore, the running time of the entire algorithm is at most $n^2 \cdot O(n)$, and so the algorithm runs in $O(n^3)$.

(b) Consider the times during the execution of the algorithm when $i \leq \frac{n}{4}$ and $j \geq \frac{3n}{4}$. In this case, $j - i + 1 \geq \frac{3n}{4} - \frac{n}{4} + 1 > \frac{n}{2}$. Therefore, adding up the array entries $A[i]$ through $A[j]$ takes at least $\frac{n}{2}$ operations. How many times during the execution of the algorithm do we encounter such cases ($i < \frac{n}{4}$ and $j > \frac{3n}{4}$)? There are $\left(\frac{n}{4}\right)^2$ pairs of (i, j) with $i < \frac{n}{4}$ and $j > \frac{3n}{4}$. The given algorithm enumerates over all of them, and as shown above, it must perform at least $\frac{n}{2}$ operations for each such pair. Therefore, the algorithm perform at least $\frac{n}{2} \cdot \left(\frac{n}{4}\right)^2 = \frac{n^3}{32}$ operations. This is $\Omega(n^3)$.

(c) Consider the following algorithm:

```
for i = 1, 2, ··· , n do
  Set B[i, i + 1] to A[i] + A[i + 1]
end for
for k = 2, 3, ··· , n − 1 do
  for i = 1, 2, ··· , n − k do
    j = i + k
    B[i, j] to be B[i, j − 1] + A[j]
  end for
end for
```

This algorithm works since the values $B[i, j-1]$ were already computed in the previous iteration of the outer for loop, when k was $j - 1 + i$, since $j - 1 - i < j - i$. It first computes $B[i, i+1]$ for all i by summing $A[i]$ with $A[i + 1]$. This requires $O(n)$ operations. For each k, it then computes all $B[i, j]$ for $j - i = k$ by setting $B[i, j] = B[i, j - 1] + A[j]$. For each k, this algorithm performs $O(n)$ operations since there are at most n $B[i, j]$'s such that $j - i = k$. There are less than n values of k to iterate over, so this algorithm has running time $O(n^2)$.

7. Solve Kleinberg and Tardos, **Chapter 3, Exercise 2**.

   Without loss of generality assume that G is connected. Otherwise, we can compute the connected components in $\mathcal{O}(m+n)$ time and deploy the below algorithm on each component.

   Starting from an arbitrary vertex s, run BFS and obtain a BFS tree (call it T). If $G = T$, then G is a tree and has no cycles. Otherwise, G has a cycle and hence there exists an edge $e = (u, v)$ such that e is in G but not in T. Find the least common ancestor of u and v in the tree. Call the least common ancestor w. There exist a unique path (call P_1) in T from u to w (and likewise a unique path P_2 in T from v to w). These paths can be constructed in $\mathcal{O}(m)$ time by starting from u (respectively from v) and going up the tree until w is reached. Output the cycle e concatenated with P_2 concatenated with $\bar{P}_1$. Here $\bar{P}_1$ denotes P_1 in the reverse order.

8. Solve Kleinberg and Tardos, **Chapter 3, Exercise 6**.

   Assume that G contains an edge $e = (x, y)$ that does not belong to T. Since T is a DFS tree and (x, y) is an edge of G that is not an edge of T, one of x or y is ancestor of the other. On the other hand, since T is a BFS tree if x and y belong to layer L_i and L_j respectively, then i and j differ by at most 1. Notice that since one of x or y is an ancestor of the other, we have that $i \neq j$ and hence i and j differ by exactly 1. However, combining that one of x or y is ancestor of the other and that i and j differ by 1 implies that the edge (x, y) is in the tree T. It contradicts the assumption that $e = (x, y)$ that does not belong to T. Thus G cannot contain any edges that do not belong to T.