CSc 220: Algorithms
Homework 1 Solutions

Problem 1: Assume that \( k, \epsilon \) are constants with \( k \geq 1 \) and \( 0 < \epsilon < 1 \). Also \( \log \) denotes the logarithm in base 2, and \( \ln \) the natural logarithm in base \( e \). State which among \( A = \Omega(B) \), \( A = \Theta(B) \) and \( A = \Omega(B) \) is correct for each pair of function below. Justify your answer. 2 points per question.

• \( A = \log^k n \) and \( B = n^\epsilon \)
• \( A = n^k \) and \( B = n \log n \)
• \( A = 3^n \) and \( B = 5^n \)
• \( A = e^{\ln n} \) and \( B = e^{3n} \)
• \( A = 4^n \) and \( B = 3^{n^2} \)

Solution:

• \( \log^k n = O(n^\epsilon) \). Actually we have that \( \log^k n = o(n^\epsilon) \) since \( \lim_{n\to\infty} \frac{\log^k n}{n^\epsilon} = 0 \) for any \( k \geq 1 \) and \( 0 < \epsilon < 1 \). Note that this means that any power of the log function (no matter how big) still grows slower than any \( \delta \)-root of \( n \) (no matter how big \( \delta = \epsilon^{-1} \)).

• This depends on the value of \( k \). By the laws of logarithms we know that \( \log n = \log e \cdot \ln n \) and therefore \( n^{\frac{\log n}{\ln n}} = n^{\log \epsilon} \)

If \( k < \log e \) then \( A = o(B) \). If \( k = \log e \) then \( A = \Theta(B) \) and if \( k > \log e \) then \( A = \omega(B) \).

• \( 3^n = O(5^n) \). Actually \( 3^n = o(5^n) \) since \( \lim_{n\to\infty} \frac{3^n}{5^n} = \lim_{n\to\infty} \left(\frac{3}{5}\right)^n = 0 \).

• Remember that \( e^{\ln n} = n \), therefore \( e^{3n} = n^{3n} \) and therefore \( B = A^{\ln n} \). So we have that \( A = O(B) \) (and therefore also \( A = \Theta(B) \)) since \( \lim_{n\to\infty} \frac{A}{B} = \lim_{n\to\infty} A^{-\ln n} = 0 \)

• Note that \( 4^n = 3^n \) where \( c = \log_3 4 \) which is a constant. So we are comparing \( A = 3^n \) to \( B = 3^{n^2} \). Since \( cn = o(n^2) \) we have that \( 4^n = o(3^{n^2}) \).

Problem 2: For each of the following recurrences: (i) describe what kind of “divide and conquer” algorithm would give rise to such a recurrence; (ii) give asymptotic upper and lower bounds on \( T(n) \). Make your bounds as tight as possible and justify your answer. Assume \( T(n) \) is a constant for \( n \leq 2 \).

• \( T(n) = 4T(n/4) + n \log n \) [2 points]
• \( T(n) = \sqrt{n}T(\sqrt{n}) + n \) [4 points]
• \( T(n) = 3T(n/3) + \frac{n}{\log_3 n} \) [4 points]

Solution:

• This is a divide and conquer algorithm that splits the input into 4 parts of equal size \( n/4 \) and recurs on all of them. The splitting and the recombining of the solution requires \( n \log n \) steps. If we draw a recursion tree we have that
  – at level 0, the root, we pay \( n \log n \).
– at level 1, we pay \(4 \cdot (n/4) \cdot \log(n/4) = n \log n - 2n\)
– in general at level \(i\) we pay \(4^i \cdot (n/4^i) \cdot \log(n/4^i) = n \log n - 2in\)

Note that we have \(\log_4 n\) levels. So the total cost is

\[
\sum_{i=0}^{\log_4 n} (n \log n - 2in) = n \log n \log_4 n - 2n \sum_{i=0}^{\log_4 n} i = n \log n \log_4 n - 2n \frac{\log_4 n (\log_4 n + 1)}{2}
\]

Recall that \(\log_4 n = \frac{\log n}{2}\). Which implies

\[
T(n) = \frac{n \log^2 n}{2} - \frac{n \log^2 n}{8} - \frac{n \log n}{4}
\]

or

\[
T(n) = \frac{3}{8} n \log^2 n - \frac{1}{4} n \log n
\]

which is \(\Theta(n \log^2 n)\)

• This is a divide and conquer algorithm that splits the input into \(\sqrt{n}\) parts of equal size \(\sqrt{n}\) and recurs on all of them. The splitting and the recombining of the solution requires \(n\) steps. If we draw a recursion tree we have that

– at level 0, the root, we pay \(n\).
– at level 1, we pay \(\sqrt{n} \cdot \sqrt{n} = n\); that’s because we have \(\sqrt{n}\) subproblems and each ”costs” \(\sqrt{n}\).
– at level 2 we have \(\sqrt{n} \cdot \sqrt{\sqrt{n}} = n^{1/2} \cdot n^{1/4} = n^{3/4}\) subproblems. Each will be of size \(\sqrt{\sqrt{n}} = n^{1/4}\) so the total cost will be \(n\) again.
– in general at level \(i\) we have

\[
n^{\frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^i}} = n^{1 - \frac{1}{2^i}}
\]

The last step follows from the geometric series formula since

\[
\sum j = 1^k a = \sum j = 0^k a - 1 = \frac{a^{k+1} - 1}{a - 1} - 1 = \frac{a^{k+1} - a}{a - 1}
\]

and when we plug \(k = i\) and \(a = 1/2\) we get

\[
\sum j = 1^{i-1} \frac{1}{2} = \frac{1/2^{i+1} - 1/2}{-1/2} = 2(\frac{1/2}{2^i + 1}) = 1 - \frac{1}{2^i}
\]

So the number of subproblems is \(n^{1 - \frac{1}{2^i}}\). Each subproblem has size \(n^{\frac{1}{2^i}}\) since at each recursion step we take a square root of the current size. So the total cost is

\[
n^{\frac{1}{2^i}} \cdot n^{\frac{1}{2^{i+1}}} = n
\]

We are left with the task of computing the height of the tree. It is not hard to see that the this height is \(\Theta(\log \log n)\). To see this write \(n = 2^{\log n}\). This implies that \(\sqrt{n} = n^{1/2} = 2^{\frac{\log n}{4}}\). So at each step with halve the exponent, which will be reduced to 1 after \(\log(\log n) = \log \log n\) steps. So the solution is \(T(n) = \Theta(n \log \log n)\).

Another way to solve this is by renaming the variables. Set \(n = 2^m\). Then we get

\[
T(2^m) = 2^{m/2} T(2^{m/2}) + 2^m
\]

and if we set \(S(m) = T(2^m)\) we have

\[
S(m) = 2^{m/2} S(m/2) + 2^m
\]
If we build a recursion tree for this recurrence we have a tree of depth \( \log m \) where each node at level \( j \) contains \( 2^{m/2^j} \) input values (starting with \( j = 0 \) at the root). Therefore at level \( j \) we must have \( a \) nodes such that \( a2^{m/2^j} = 2^m \), i.e.
\[
a = 2^m - \frac{m}{2^j} = 2^m \left(\frac{2^j - 1}{2^j}\right)
\]
Each node does \( 2^{m/2^j} \) work. So each level does
\[
a2^{m/2^j} = 2^m \left(\frac{2^j - 1}{2^j}\right) + \frac{m}{2^j} = 2^m
\]
work. Since there are \( \log m \) levels, the total work is \( T(n) = 2^m \log m \). Remember now that \( 2^m = n \) and therefore \( m = \log n \), yielding \( T(n) = \Theta(n \log \log n) \).

- **This is a divide and conquer algorithm** that splits the input into 3 parts of equal size \( n/3 \) and recurs on all of them. The splitting and the recombining of the solution requires \( \frac{n}{\log_3 n} \) steps. If we draw a recursion tree we have that
  - at level 0, the root, we pay \( \frac{n}{\log_3 n} \).
  - at level 1, we pay \( 3 \cdot \frac{n/3}{\log_3 (n/3)} = \frac{n}{\log_3 n-1} \).
  - in general at level \( i \) we pay \( 3^i \cdot \frac{n/3^i}{\log_3 (n/3^i)} = \frac{n}{\log_3 n-1} \).

Note that we have \( \log_3 n - 1 \) levels since we must stop at \( n = 3 \) (the recurrence is not defined for \( n = 1 \) since \( \log 1 = 0 \) and we cannot divide for 0). So the total cost is
\[
\sum_{i=0}^{\log_3 n-1} \frac{n}{\log_3 n-i} = n \sum_{i=1}^{\log_3 n} \frac{1}{i}
\]
which is \( \Theta(n \log \log n) \) since \( \sum_{i=1}^{k} \frac{1}{k} = \Theta(\log k) \). Note that we don’t care about the basis of the logarithm since we moved to the asymptotic notation. Also note the change of variables in the summation: \( \sum_{i=1}^{k-1} \frac{1}{k-1} = \sum_{i=1}^{k-1} \frac{1}{i} \).

**Problem 3:** **BubbleSort** is a sorting algorithm that scans the vector from left to right and swaps to adjacent elements that are out of order. It loops the above scan until all elements are sorted. Write some pseudocode for **BubbleSort**. What is the worst-case running time of **BubbleSort**? Justify your answer.

**Solution:** In this pseudocode \( \text{Swap}(A, i, j) \) is a simple procedure that swaps the locations of elements \( i, j \) in a vector \( A \).

**BubbleSort**\( (A) \)
\[
\text{order} \leftarrow \text{FALSE}
\]
\[
\text{WHILE } \text{order} \text{ FALSE}
\]
\[
\text{order} \leftarrow \text{TRUE}
\]
\[
\text{FOR } i = 1 \text{ To } n
\]
\[
\text{IF } A(i) > A(i+1) \text{ THEN}
\]
\[
\text{Swap}(A, i, i+1)
\]
\[
\text{order} \leftarrow \text{FALSE}
\]
\[
\text{END WHILE}
\]
\[
\text{RETURN } A
\]

The worst-case running time of **BubbleSort** is \( \Theta(n^2) \). The FOR loop obviously takes \( O(n) \) to execute. The question is how many times do we execute the WHILE loop.

Note that after we execute the WHILE loop for the \( i \)th time the \( (n - 1 + i) \)th largest element will be in the correct position (i.e. in position \( A[n - 1 + i] \)). In other words after the first execution the maximum is in \( A[n] \), after the second execution the second largest element is in \( A[n - 1] \) and so on. So we are guaranteed
that the loop will not run more than \( n + 1 \) times (we need one additional loop to set the flag variable \( \text{order} \) to True). This proves that the running time is \( O(n^2) \).

If \( A \) is an input array which is in reversed order, then \( \text{BubbleSort} \) will take at least \( \Omega(n^2) \) time to complete since after each \( \text{WHILE} \) loop exactly one element will be in its correct spot.