CSc 220: Algorithms
Homework 1 Solutions

Problem 1: Assume that k, ϵ 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 $A = O(B)$, $A = \Omega(B)$ and $A = \Theta(B)$ is correct for each pair of function below. Justify your answer. 2 points per question.

- $A = \log^2 n$ and $B = n^{1/100}$
- $A = n^k$ and $B = n^{\frac{\ln n}{\log n}}$
- $A = 5^n$ and $B = 4^n$
- $A = n^{\log^2 n}$ and $B = 2^{\log^3 n}$
- $A = 3^n$ and $B = 2^{n^2}$

Solution:

- $\log^2 n = O(n^{1/100})$. Actually we have that $\log^k n = o(n^\epsilon)$ since $\lim_{n\to\inf} \frac{\log^2 n}{n^{1/100}} = 0$ (this can be seen by applying L'Hopital rule twice). In general any power of the log function (no matter how big) still grows slower than any δ-root of n (no matter how big δ).
- Since $\log n = \ln n \cdot \log e$ we have that
$$\frac{\ln n}{\log n} = \frac{1}{\log e} \quad \text{and} \quad B = n^{\frac{1}{\log e}}$$
Since $\log e > 1$ then $\frac{1}{\log e} < 1$ and there $n^k = \omega((n^{\log n})^{\ln n})$ since $k \geq 1$.
- $5^n = \omega(4^n)$ since $\lim_{n\to\inf} \frac{5^n}{4^n} = \lim_{n\to\inf} (\frac{5}{4})^n = \inf$ since $\frac{5}{4} > 1$.
- Remember that $2^{\log n} = n$, therefore $B = 2^{\log^3 n} = n^{\log^2 n} = A$. The two functions being equal, we have $A = \Theta(B)$.
- Note that $3^n = 2^{cn}$ where $c = \log 3$ which is a constant. So we are comparing 2^{cn} to 2^{n^2}, since $cn = o(n^2)$ we have that $3^n = o(2^{n^2})$.

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 $n \leq 2$.

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

Solution:

- 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 $n \log n$ 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 $3 * (n/3) * \log(n/3) = n \log n - n \log 3$

- in general at level i we pay $3^i * (n/3^i) * \log(n/3^i) = n\log n - ni\log 3$

Note that we have $\log_3 n$ levels. So the total cost is

$$\sum_{i=0}^{\log_3 n}(n\log n - ni\log 3) = n\log n\log_3 n - n\log 3\sum_{i=0}^{\log_3 n} i = n\log n\log_3 n - n\log 3\frac{\log_3 n(\log_3 n+1)}{2}$$

Recall that $\log_3 n = a\log n$ where $a = (\log 3)^{-1}$. This implies

$$T(n) = an\log^2 n - \frac{a^2}{2a}n\log^2 n - \frac{a}{2a}n\log n = an\log^2 n - \frac{a}{2}n\log^2 n - \frac{1}{2}n\log n$$

setting $b = a - \frac{a}{2} > 0$ we have

$$T(n) = bn\log^2 n - \frac{1}{2}n\log n$$

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. To solve, substitute $n = 2^m$. 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.

  $$a = 2^{m-\frac{m}{2^j}} = 2^{\frac{m(2^j-1)}{2^j}}$$

  Each node does $2^{m/2^j}$ work. So each level does

  $$a2^{\frac{m}{2^j}} = 2^{\frac{m(2^j-1)}{2^j}+\frac{m}{2^j}} = 2^m$$

  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 2 parts of equal size $n/2$ and recurs on all of them. The splitting and the recombining of the solution requires $\frac{n}{\log n}$ steps. If we draw a recursion tree we have that

  - at level 0, the root, we pay $\frac{n}{\log n}$.
  - at level 1, we pay $2\frac{n/2}{\log(n/2)} = \frac{n}{\log n - 1}$
  - in general at level i we pay $2^i\frac{n/2^i}{\log(n/2^i)} = \frac{n}{\log n - i}$

  Note that we have $\log n - 1$ levels since we must stop at $n = 2$ (the recurrence is not defined for $n = 1$ since $\log 1 = 0$ and we cannot divide for 0). So the total cost is

  $$\sum_{i=0}^{\log n - 1}\frac{n}{\log n - i} = n\sum_{i=1}^{\log n}\frac{1}{i}$$

  which is $\Theta(n\log\log n)$ since $\sum_{i=1}^{k}\frac{1}{i} = \Theta(\log k)$.

Problem 3: Given a set A of n distinct positive integers and another interger t, describe an algorithm that determines whether or not there exists two elements in A such that their product is exactly t. Prove that your algorithm is correct and analyze its running time. Full credit will be given to the fastest algorithm.

Solution: One trivial solution is to try all possible pairs of elements of A and see if their product equals t. This requires $O(n^2)$ operations. A faster algorithm would be to sort the set A and then search for the pair by comparing t with the product of the mimimum and maximum element of A, and discarding either the minimum or the maximum depending on the result. More specifically consider the following algorithm

```
FIND-PRODUCT(A, t)
    i ← 1; j ← n;
    B ⟵ MERGE-SORT(A);
    WHILE j − i > 0 DO
        IF B[i] · B[j] = t THEN RETURN TRUE AND STOP;
        IF B[i] · B[j] < t THEN i ← i + 1;
        IF B[i] · B[j] > t THEN j ← j − 1;
    END WHILE
    RETURN FALSE
```

Let's prove first that this algorithm is correct. Consider the three possible choices inside the WHILE loop. If $B[i] \cdot B[j] = t$ then the algorithm correctly returns TRUE. If $B[i] \cdot B[j] < t$, then since the vector B is sorted then for any k such that $i < k < j$ we have that $B[i] \cdot B[k] \leq B[i] \cdot B[j] < t$ so we can safely discard $B[i]$ since it will never produce t when multiplied with any of the elements left in the array B. Similarly if $B[i] \cdot B[j] > t$, then for any k such that $i < k < j$ we have that $B[k] \cdot B[j] \geq B[i] \cdot B[j] > t$ so we can safely discard $B[j]$ since it will never produce t when multiplied with any of the elements left in the array B.

To analyze the running time, note that the WHILE loop is executed at most n times since at each executions either the algorithm stops or the difference $j - i$ decreases by 1. The work inside the WHILE loop is constant, so the total cost of the WHILE loop is $O(n)$. Therefore the running time of this algorithm is $\Theta(n \log n)$ since the sorting step with MERGE-SORT takes $\Theta(n \log n)$, which dominates the $O(n)$ cost of the WHILE loop.