CSE3358 Problem Set 2
Solution

Please review the Homework Honor Policy on the course webpage
http://engr.smu.edu/˜saad/courses/cse3358/homeworkpolicy.html

Problem 1: Practice with asymptotic notations

(a) (3 points) Explain clearly why the statement "The running time of algorithm A is at least $O(n^3)$" does not make sense.

ANSWER: O notation is an upper bound notation. Therefore, $O(n^3)$ by iself means "at most $c.n^3$". So the statement "The running time of algorithm A is **at least at most** $c.n^3$" does not make sense.

(b) (6 points) Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Let $h(n) = \max(f(n), g(n))$. Using the precise mathematical definition of Θ-notation, show that $h(n) = \Theta(f(n) + g(n))$. *Hint*: Remember this means you have to show that $h(n) = O(f(n) + g(n))$ and $h(n) = \Omega(f(n) + g(n))$.

Can we say the same thing about $h'(n) = \min(f(n), g(n)$, i.e. is $h'(n) = \Theta(f(n) + g(n))$? Explain.

ANSWER: To show $h(n) = \Theta(f(n) + g(n))$, we need to show two things:

- $h(n) = O(f(n) + g(n))$
- $h(n) = \Omega(f(n) + g(n))$
  First note that if $f(n) \geq g(n)$, then $h(n) = f(n)$. Similarly, if $g(n) \geq f(n)$, then $h(n) = g(n)$.

For the first part, we need to show that $h(n) \leq c(f(n)+g(n))$ for some positive constant c and large n.

- $f(n) \leq g(n) \Rightarrow h(n) = g(n) \leq f(n) + g(n) = 1.(f(n) + g(n))$
- $f(n) \geq g(n) \Rightarrow h(n) = f(n) \leq f(n) + g(n) = 1.(f(n) + g(n))$

For the second part, we need to show that $h(n) \geq c(f(n) + g(n))$ for some positive constant c and large n.

- $f(n) \leq g(n) \Rightarrow h(n) = g(n)$, but
  $\frac{1}{2}g(n) \geq \frac{1}{2}f(n)$
  $\frac{1}{2}g(n) \geq \frac{1}{2}g(n)$
  Therefore, $g(n) = h(n) \geq \frac{1}{2}(f(n) + g(n))$.
- $f(n) \geq g(n) \Rightarrow h(n) = f(n)$, but
  $\frac{1}{2}f(n) \geq \frac{1}{2}f(n)$
  $\frac{1}{2}f(n) \geq \frac{1}{2}g(n)$
  Therefore, $f(n) = h(n) \geq \frac{1}{2}(f(n) + g(n))$.

Intuitively, $h(n)$ which is the maximum of $h(n)$ and $g(n)$ is $\leq$ their sum but $\geq$ their average, which is their sum divided by 2.

We cannot say the same about $h'(n)$. Although $h'(n) = O(f(n) + g(n))$, it is not true that $h'(n) = \Omega(f(n) + g(n))$. To show this, it is enough to provide a counter example. Let $f(n) = n$ and $g(n) = n^2$. Then $h'(n) = n$. But it is not true that $n = \Omega(n+n^2)$. In deed for this to be true, we need $n \geq c(n+n^2)$ $\forall n \geq n_0$ for some positive constants c and n_0. This means $\frac{n}{n+n^2} \geq c$ for large n. But this cannot be because $\lim_{n\to\infty} \frac{n}{n+n^2} \to 0$.

(c) (3 points) Show that for any real constants a and b, where $b > 0$, $(n + a)^b = \Theta(n^b)$.

ANSWER: First, $n + a = \Theta(n)$ because low order terms and leading constants don't matter (but we can actually prove this using the definition). Therefore, $n + a = \Omega(n)$ and $n + a = O(n)$. So,

$$c_1 n \leq n + a, n \geq n_1$$

$$n + a \leq c_2 n, n \geq n_2$$

Therefore,

$$c_1 n \leq n + a \leq c_2 n, n \geq n_0$$

for some constants c_1, c_2, n_0 ($n_0 = \max(n_1, n_2)$).

Since b is positive, we can take everything to power b, we get:

$$(c_1 n)^b \leq (n + a)^b \leq (c_2 n)^b, n \geq n_0$$

$$(c_1)^b n^b \leq (n + a)^b \leq (c_2)^b n^b, n \geq n_0$$

$$c_1' n^b \leq (n + a)^b \leq c_2' n^b, n \geq n_0$$

Therefore,

$$c_1' n^b \leq (n + a)^b, n \geq n_0 \Rightarrow (n + a)^b = \Omega(n^b)$$

$$(n + a)^b \leq c_2' n^b, n \geq n_0 \Rightarrow (n + a)^b = O(n^b)$$

Therefore, $(n + a)^b = \Theta(n^b)$.

(d) (4 points) Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

Is $2^{n+1} = O(2^n)$? **ANSWER**: yes. $2^{n+1} = 2.2^n$. Is $2^{2n} = O(2^n)$? **ANSWER**: no. $2^{2n} = (2^2)^n = 4^n$ and $4^n \neq O(2^n)$. If $4^n = O(2^n)$ then $4^n \leq c.2^n$ for some constant c and large n. Therefore $\frac{4^n}{2^n} \leq c$ for large n which means that $2^n \leq c$ for large n, impossible.

(e) (4 points) Find two non-negative functions $f(n)$ and $g(n)$ such that neither $f(n) = O(g(n))$ nor $g(n) = O(f(n))$.

ANSWER: Consider $f(n) = 1 + \cos(n)$ and $g(n) = 1 + \sin(n)$. Therefore, both $f(n)$ and $g(n)$ are periodic and take values in $[0..2]$. But when $f(n) = 0$, $g(n) = 2$, and when $g(n) = 0$, $f(n) = 2$. Therefore, we cannot find a positive constant c, such that $f(n) \leq c.g(n)$ for large n, because $g(n)$ always comes back to 0 when $f(n)$ is 2. Therefore, $f(n)$ cannot be $O(g(n))$. The same argument works for the other case.

Problem 2: Recursive Insertion sort

We have seen in class how Merge sort sorts an array by recursively sorting smaller subarrays. A pseudocode similar to the one we saw in class is shown below for convenience.

```
MERGE-SORT(A, p, r)
    if p < r
      then q ← ⌊(p+r)/2⌋
           MERGE-SORT(A, p, q)
           MERGE-SORT(A, q + 1, r)
           MERGE(A, p, q, r)
```

where MERGE(A, p, q, r) merges the two sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Insertion sort can also be expressed as a recursive procedure: In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ and insert $A[n]$ into the sorted array $A[1..n-1]$.

(a) (10 points) Write the pseudocode of Insertion sort as a recursive procedure.

ANSWER:

```
INSERTION-SORT(A, p, r)
    if p < r
      then INSERTION-SORT(A, p, r − 1)
           INSERT(A[p..r − 1], A[r])
```

where INSERT$(A[p..r], key)$ inserts key into the sorted array $A[p..r]$. The pseudocode for INSERT is shown below and it consists of one iteration of the non-recursive INSERTION-SORT.

```
INSERT(A[p..r], key)
   i ← r − 1
   while i > p − 1 and A[i] > key
        do A[i + 1] ← A[i]
           i ← i − 1
   A[i + 1] ← key
```

(b) (10 points) Let $T(n)$ be the running time of Insertion sort on an array of size n. By identifying the running time of each line in the code of part (a) (either as a Θ notation or as a function of T on smaller arguments), obtain a recurrence equation for $T(n)$.

ANSWER:

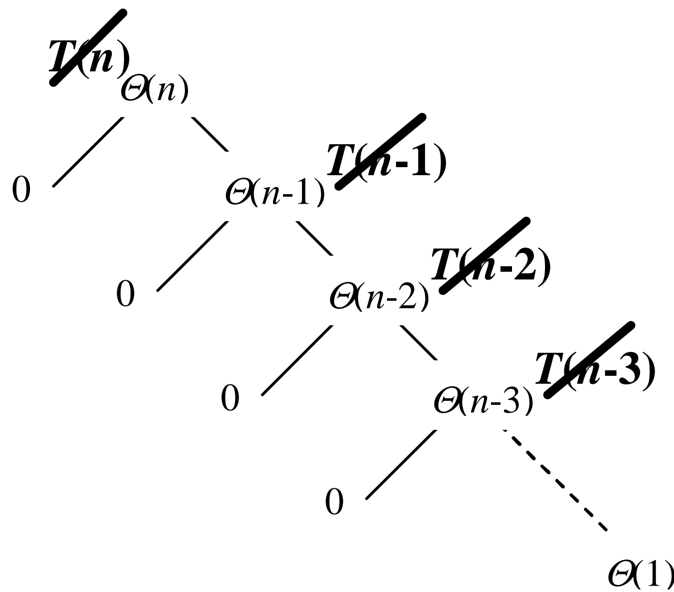
```
INSERTION-SORT(A, p, r)                       T(n)
    if p < r                                  Θ(1)
      then INSERTION-SORT(A, p, r − 1)        T(n − 1)
           INSERT(A[p..r − 1], A[r])          Θ(n)
```

Let $T(n)$ be the running time of this algorithm. The first line consists of checking a condition which takes $\Theta(1)$ time. The second line is a recursive call to INSERTION-SORT with argument $(A, p, r-1)$, i.e. with an input of size $n-1$, so this takes $T(n-1)$ time. The last line consists of the shifting process of the regular INSERTION-SORT which takes $\Theta(n)$ time. Therefore, $T(n) = \Theta(1) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n)$.

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T(n-1) + \Theta(n) & n > 1 \end{cases}$$

(c) (10 points) Solve for $T(n)$ by expending it in a tree-like structure as we did in class.

$T(n) = \Theta(n) + T(n-1) = \Theta(n) + \Theta(n-1) + T(n-2) = \Theta(n) + \Theta(n-1) + \Theta(n-2) + T(n-3) =$ .... $\sum_{i=2}^{n} \Theta(i) + T(1) = \sum_{i=2}^{n} \Theta(i) + \Theta(1) = \sum_{i=1}^{n} \Theta(i) = \Theta(\sum_{i=1}^{n} i) = \Theta(n^2)$.



Problem 3: Insertion sort and merge sort

Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \log n$ steps.

(a) (5 points) For which values of n does insertion sort beat merge sort?

ANSWER: For $n \leq 43$, $8n^2 \leq 64n \log n$ and insertion sort beats merge sort

Although merge sort runs in $\Theta(n \log n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort make it faster for small n. Therefore, it makes sense to use insertion sort within merge sort when subproblems become sufficiently small. Consider a modification of merge sort in which subarrays of size k or less (for some k) are not divided further, but sorted explicitly with Insertion sort.

```
MERGE-SORT(A, p, r)
    if p < r − k + 1
      then q ← ⌊(p+r)/2⌋
           MERGE-SORT(A, p, q)
           MERGE-SORT(A, q + 1, r)
           MERGE(A, p, q, r)
      else INSERTION-SORT(A[p..r])
```

(b) (5 points) Show that the total time spent on all calls to Insertion sort is in the worst-case $\Theta(nk)$.

ANSWER: As a technical remark, if $n <= k$, then this modified MERGE-SORT will just call INSERTION-SORT on the whole array. Therefore, the running time will be $\Theta(n^2) = \Theta(n.n) = O(nk)$. So let us assume that $n > k$.

First note that the length of subarray $A[p..r]$ is $l = r - p + 1$. So the condition $p < r - k + 1$ is the same as $l > k$. Therefore, any subarray of length l on which we call INSERTION-SORT has to satisfy $l \leq k$ (this is actually in the given of the algorithm). Moreover, any subarray $A[p..r]$ of length l on which we call INSERTION-SORT has to satisfy $\frac{k}{2} \leq l$. The reason for this is the following: if INSERTION-SORT is called on $A[p..r]$, then $A[p..r]$ must be the first or second half of another subarray $A[p'..r']$ that was divided (we assumed that $n > k$ so $A[p..r]$ cannot be $A[1..n]$). Thus, $A[p'..r']$ has length $l' > k$. Therefore, since $l \geq \lfloor \frac{l'}{2} \rfloor$ and $l' > k$, then $l \geq \frac{k}{2}$.

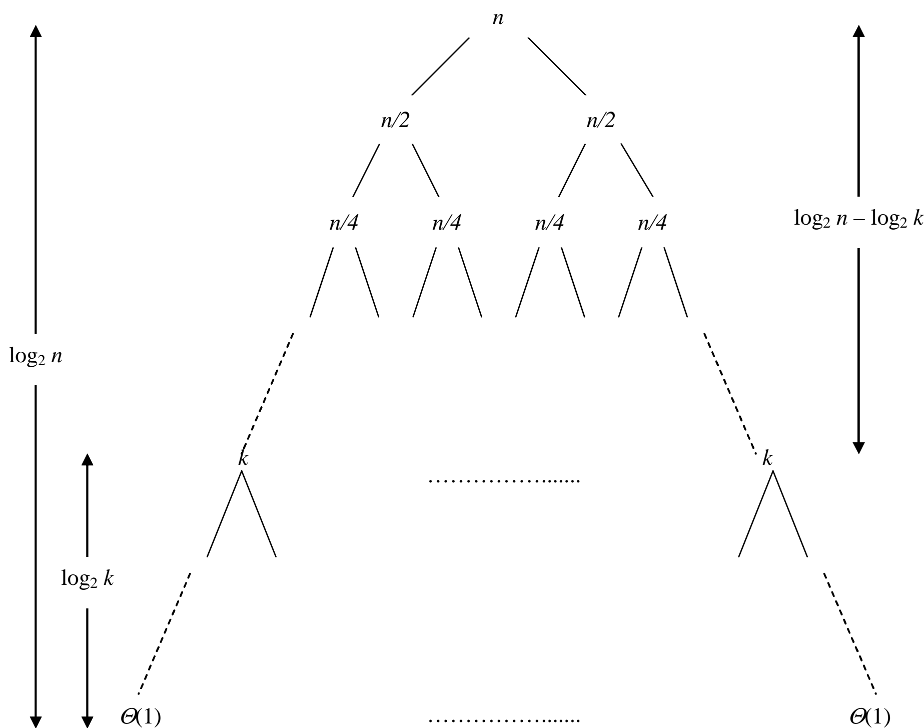
The running time of INSERTION-SORT on $A[p..r]$ is $\Theta(l^2)$, where $l = r - p + 1$ is the length of $A[p..r]$. But since $\frac{k}{2} \leq l \leq k$, then $l = \Theta(k)$ and the running time of INSERTION-SORT on $A[p..r]$ is $\Theta(k^2)$.

Now let us see how many such subarrays (on which we call INSERTION-SORT) we have. Since all the subarrays are disjoint (i.e. their indices do not overlap) and for every one of them $\frac{k}{2} \leq l \leq k$, then the number of these subarrays, call it m, satisfies $\frac{n}{k} \leq m \leq \frac{2n}{k}$. Therefore $m = \Theta(n/k)$.

The total time spent on INSERTION-SORT is therefore $\Theta(k^2).\Theta(n/k) = \Theta(nk)$.

(c) (5 points) Show that the total time spent on merging is in the worst-case $\Theta(n \log(n/k))$.

ANSWER: The merging proceeds in the same way as with the basic version of MERGE-SORT except that it stops whenever the subarray reaches a lenght $l \leq k$ (instead of 1). Therefore, we need to determine the number of levels in the recursive tree. With the basic version of MERGE-SORT, we argued that the number of levels is $\Theta(\log n)$ because we divide n by 2 with every level until we reach 1. With this modification, we divide n by 2 with every level until we reach k (or less). Thefore the number of levels is $\Theta(\log n) - \Theta(\log k)$, which is the number of levels originally minus the number of levels that we would have had if we continued beyond k. But $\Theta(\log n) - \Theta(\log k) = \Theta(\log(n/k)$. So the total time that we spend on merging is $\Theta(n \log(n/k)$ since we spend $\Theta(n)$ time in every level as before.



(d) (5 points) Therefore, this modified merge sort runs in $\Theta(nk) + \Theta(n \log(n/k))$ (sorting + merging). The total running time is therefore $\Theta(nk + n \log(n/k))$. What is the largest asymptotic (Θ-notation) value of k as a function of n for which the modified algorithm has the same asymptotic running time as standard merge sort? How should k be chosen in practice?

ANSWER: Obviously, if k grows faster than $\log n$ asymptotically, then we would get something worse than $\Theta(n \log n)$ because of the $\Theta(nk)$ term. Therefore, we know that $k = O(\log n)$. So lets pick $k = \Theta(\log n)$ and see if this works. If $k = \Theta(\log n)$ then the running time of the modified MERGE-SORT will be $\Theta(n \log n + n \log \frac{n}{\log n}) = \Theta(n \log n + n \log n - n \log(\log n)) = \Theta(n \log n)$. So if $k = \Theta(\log n)$, the running time of the modified algorithms has the same asymptotic running time as the standard one. In practice, k should be the maximum input size on which Insertion sort is faster then Merge sort.

(10 points) [This is intended to (1) practice the implementation of recursive functions and (2) really appreciate the difference in efficiency between Insertion sort and Merge sort on large size inputs]

Implement in a programming language of your choice this modified version of Merge sort. Therefore, you have to have Insertion sort implemented as well (but hopefully you can use your implementation of Problem Set 1). Compare the running times of Merge sort and Insertion sort on large values of n and report your findings.