Homework 1 (65 points), Spring 2006

Q1: Exercise 2.3-3 (10 points)

Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \log n$.

- Base Step:
  If $n = 2$, then $T(2) = 2$ and $2 \log 2 = 2$
  Thus, $T(2) = 2 \log 2$
- Hypothesis Step:
  Assuming $T(n) = n \log n$ is true if $n = 2^k$ for some integer $k > 0$
- Induction Step:
  If $n = 2^{k+1}$, then

$$\begin{aligned} & T(2^{k+1}) \\ = & \; 2T(2^{k+1}/2) + 2^{k+1} \\ = & \; 2T(2^k) + 2^{k+1} \\ = & \; 2(2^k \log 2^k) + 2^{k+1} \\ = & \; 2^{k+1}((\log 2^k) + 1) \\ = & \; 2^{k+1} \log 2^{k+1} \end{aligned}$$

Q2: (10 points)

Please analyse the binary search algorithm by the three steps - `divide, conquer` and `combine`, and then write down the running time recurrence equation for the worst case.

- **Divide:** Assume the input array $A[1..n]$, with n elements, is already sorted. We would like to search a target element X (may or may not in A).
  Compare X to $A[m]$, where $m = \lfloor(1+n)/2\rfloor$ and $A[m]$ is the element in the middle of A.

  Conquer:

```
if X = A[m]
   then return m  // X found in A[m]
if X < A[m]
   then recursively search X in the left subarray A[1..m-1]
   else recursively search X in the right subarray A[m+1..n]
```

  Combine: Do nothing

- The worst-case occurs when X is not in A. Thus, the running time recurrence equation is $T(n) = T(n/2) + 1$

Q3: Problem 2-1 parts a, b and c (15 points)

Insertion sort on small arrays in merge sort
Although merge sort runs in $\Theta(n \lg n)$ 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. Thus, it make sense to use insertion sort within merge sort when subproblems become sufficiently small. Consider a modification for merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

a. Show that the n/k sublists, each of length k, can be sorted by insertion sort in $\Theta(nk)$ worst-case time.

- Each sublist with length k takes $\Theta(k^2)$ worst-case time using Insertion-Sort. To sort n/k such sublists, it takes $n/k \times \Theta(k^2) = \Theta(nk)$ worst-case time.

b. Show that the sublists can be merged in $\Theta(n \lg(n/k))$ worst-case time.

- Merging n/k sublists into $n/2k$ sublists takes $\Theta(n)$ worst-case time.
- Merging $n/2k$ sublists into $n/4k$ sublists takes $\Theta(n)$ worst-case time
- ....
- Merging 2 sublists into one list takes $\Theta(n)$ worst-case time
- We have $\lg(n/k)$ such merges, so merging n/k sublists into one list takes $\Theta(n \lg(n/k))$ worst-case time.

c. Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, 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?

- In order for $\Theta(nk + n \lg(n/k)) = \Theta(n \lg n)$, either $nk = \Theta(n \lg n)$ or $n \lg(n/k) = \Theta(n \lg n)$. From the above two possibilities, we know the largest asymptotic value for k is $\Theta(\lg n)$.

Q4: (10 points)

Please use the basic definition of Θ-notation to prove $\frac{1}{5}n^2 - 80 = \Theta(n^2)$.

- We would like to find positive constants c_1, c_2 and n_0, such that

$$0 \le c_1 n^2 \le \frac{1}{5}n^2 - 80 \le c_2 n^2, \forall n \ge n_0$$

  Such constants do exist, for example, $c_1 = 1/10$, $c_2 = 1$ and $n_0 = 40$
  Therefore, $\frac{1}{5}n^2 - 80 = \Theta(n^2)$

Q5: Exercise 3.1-3 (10 points)

Explain why the statement, "The running time of algorithm A is at least $O(n^2)$," is meaningless.

- Let $T(n)$ be the running time for algorithm A and let a function $f(n) = O(n^2)$. The statement says that $T(n)$ is at least $O(n^2)$. That is, $T(n)$ is an upper bound of $f(n)$. Since $f(n)$ could be any function "smaller" than n^2 (including constant function), we can rephase the statement as "The running time of algorithm A is at least constant." This is meaningless because the running time for every algorithm is at least constant.

Q6: Exercise 3.1-4 (10 points)

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

- We can choose $c = 2$ and $n_0 = 0$, such that $0 \leq 2^{n+1} \leq c \times 2^n$ for all $n \geq n_0$. By definition, $2^{n+1} = O(2^n)$.

- We can not find any c and n_0, such that $0 \leq 2^{2n} = 4^n \leq c \times 2^n$ for all $n \geq n_0$. Therefore, $2^{2n} \neq O(2^n)$.