Homework 1 (70 points), Fall 2004

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

\[
T(2^{k+1})
\]

\[
= 2T(2^{k+1}/2) + 2^{k+1}
\]

\[
= 2T(2^k) + 2^{k+1}
\]

\[
= 2(2^k \log 2^k) + 2^{k+1}
\]

\[
= 2^{k+1}((\log 2^k) + 1)
\]

\[
= 2^{k+1} \log 2^{k+1}
\]

Q2: Exercise 2.3-4 (10 points)

Insertion sort can be expressed as a recursive procedure as follows. In order to sort \( A[1..n] \), we recursively sort \( A[1..n-1] \) and then insert \( A[n] \) into the sorted array \( A[1..n-1] \). Write a recurrence for the running time of this recursive version of insertion sort.

- **Divide:** Divide the problem into one smaller subproblem with size \( n - 1 \).
  **Conquer:** Conquer the subproblem by solving it recursively. After this step, we have a sorted subarray with size \( n - 1 \).
  **Combine:** Insert the element not in the sorted subarray into the sorted subarray.

- Therefore, the recurrence for this version of Insertion-Sort is as follows.

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
T(n - 1) + O(n) & \text{if } n > 1
\end{cases}
\]

Q3: Exercise 2.3-5 (10 points)
Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence $A$ is sorted, we can check the midpoint of the sequence against $v$ and eliminate half of the sequence from further consideration. Binary search is an algorithm that repeats this procedure, having the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\log n)$.

- binary-search($A$, $p$, $r$, $x$)  // $A$: input array;
  // $p$: starting index; $r$: ending index
  // $x$: target element
  1. if $p \leq r$
     2. then $q = (p+r)/2$  // integer division
        3. if $A[q] = x$
           4. then return $q$;
        5. else if $A[q] > x$
           6. then return binary-search($A$, $p$, $q-1$);
        7. else return binary-search($A$, $q+1$, $r$);
  8. else return -1;  // not found

- The worst case running time $T(n)$ can be expressed by the recurrence $T(n) = T(n/2) + 1$. Thus, $T(n) = \Theta(\log n)$.

**Q4: Problem 2-1 parts a, b and c (10 points)**

**Insertion sort on small arrays in 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 $n$. Thus, it makes 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 $\Theta(n \log(n/k))$ 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 $\log(n/k)$ such merges, so merging $n/k$ sublists into one list takes $\Theta(n \log(n/k))$ worst-case time.

c. Given that the modified algorithm runs in $\Theta(nk + n \log(n/k))$ worst-case time, what is the largest asymptotic ($\Theta$-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 $\Theta(nk + n \log(n/k)) = \Theta(n \log n)$, the largest asymptotic value for $k$ is $\Theta(\log n)$. 


Q5: (10 points)

Please use the basic definition of $\Theta$-notation to prove $\frac{1}{4}n^2 - 20 = \Theta(n^2)$.

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

$$0 \leq c_1 n^2 \leq \frac{1}{4}n^2 - 20 \leq c_2 n^2, \forall n \geq n_0$$

Such constants do exist, for example, $c_1 = 1/8$, $c_2 = 1$ and $n_0 = 13$

Therefore, $\frac{1}{4}n^2 - 20 = \Theta(n^2)$

Q6: 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 rephrase 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.

Q7: 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, $2^{2n} \neq O(2^n)$. 
