COMP 271 Design and Analysis of Algorithms
Spring 2009 Final Exam Solutions

1. 1.1 $\Theta(\log n)$ 1.2 $\Theta(\sqrt{n})$ 1.3 $n \log n$ 1.4 $\Theta(n), \Theta(n), \Theta(1)$

2. **[OMITTED FROM SYLLABUS]**
   2.1 (a) 2.2 (c) 2.3 (a) 2.4 (c)

3. (a) The running time of merge is linear on the input arrays. We will be running this on arrays of size:
   $n + n, 2n + n, \ldots, (k-1)n + n$

   The total cost is the sum of these:

$$\left(n \sum_{i=1}^{k-1} i\right) + (k-1)n$$
$$n\left(\frac{k(k+1)}{2}\right) + (k-1)n$$
$$n\frac{k^2+k}{2} + (k-1)n$$

   Thus, it costs $O(nk^2)$.

   (b)

```
MULTI-MERGE(A[1..k][1..n], i, j)
if i = j then
  return A[i][1..n];
m ← ⌊(i−j)/2⌋;
return MERGE(MULTI-MERGE(A[1..k][1..n], i, m),
MULTI-MERGE(A[1..k][1..n], m+1, j));
```

   In the algorithm above, we first divide the list of k sorted arrays into two lists recursively until there are only two sorted arrays for $MERGE$ operation to merge them into one and we perform such step recursively. We have the recurrence $T(k) = 2T(k/2) + O(nk)$ and $T(1) = n$, which solves to $T(k) = O(nk \log k)$.

4. This problem is the same to solve the series:
   $f(n) = f(n-3) + f(n-2) + f(n-1)$, where $n > 1$
   $f(n) = 1$, where $n = 0, 1$
   $f(n) = 0$, where $n < 0$
   So we can compute this by a bottom-up computation, which can be done in $O(n)time$.

5. Let $T = \langle t_1, \ldots, t_n \rangle$ be the input string of length n. We need to find the longest substring that appears both forward and backward in T, without overlapping.

   Step 1: Space of subproblems
   Let $s[i, j]$ denote the longest substring that appears both forward and backward in T, without overlapping, and with the forward substring starting at $T[i]$ and the backward substring ending at $T[j]$. Define $d[i, j]$ to be the length of $s[i, j]$. Note that the length of the desired substring in T is $\max_{1 \le i < j \le n} d[i, j]$.

   Step 2: Recursive formulation
   Consider the substring from t_i to t_j: $\langle t_i, t_{i+1}, \ldots, t_{j-1}, t_j \rangle$. If $t_i \neq t_j$, then $s[i, j]$ doesn't

exist. If $t_i = t_j$, $s[i,j]$ must be $s[i+1, j-1]$ appended with t_i in the front. Thus,

$$d[i,j] = \begin{cases} 0 & \text{if } t_i \neq t_j \\ 0 & \text{if } i \geq j \\ d[i+1, j-1] + 1 & \text{if } i < j \text{ and } t_i = t_j \end{cases}$$

The base case is easily established. $d[i,j] = 0$ for $i \geq j$ because we do not allow overlapping.

Step3: Bottom-up Computation
We store the $d[i,j](i < j)$ in an table whose entries are computed in order of increasing j using the recurrence, for each j we compute in the decreasing order of i. After computing the d array we run through all the entries to find the maximum value. This is the length of the desired subsequence in T.

Running Time: Since it takes $O(1)$ time to compute each entry of the table, the total time to compute the d array is $O(n^2)$. So the total running time is $O(n^2)$.

6. Sort $\{t_1, t_2, \ldots, t_n\}$ in non-decreasing order, and the resulting sequence $\Sigma = \{s_1, s_2, \ldots, s_n\}$ is the answer.

   To establish correctness of this simple greedy algorithm, observe that in any ordering, the waiting time for the ith person in the sequence is the sum of service times of the $i-1$ people that appear before him/her in the order. We will show that any sequence that is different from the above sorted sequence cannot be optimal.

   Let Σ' be any sequence that is not in sorted order. There must be at least one index j such that $t_j > t_{j+1}$ in this order. If we swap these two consecutive customers, we see that customer j's waiting time has increased by t_{j+1}, and customer $(j+1)$'s waiting time has decreased by t_j. All other customers have the same waiting time in either order. Thus, there has been a net decrease of $t_j - t_{j+1}$ in the total waiting time, which is strictly positive by our assumption that $t_j > t_{j+1}$. Thus, any unsorted sequence is suboptimal, which implies that the sorted sequence is optimal.

7. **[OMITTED FROM SYLLABUS]**
   (a) DS $\in$ NP class. A certificate will be the set of vertices V', and we just count whether the number of edges between them is at least m, which could be done in polynomial time. (b)Given an instance of the DCLIQUE problem: Given a graph $G = (V, E)$ and an integer k, ask if there exist a clique of size k. We reduce it to a DS problem. Let $G' = G, k' = k$, and $m' = k(k-1)/2$, we ask if there exist a subgraph of G' induced by k' vertices such that the number of edges of that subgraph is at least m'. This reduction can obviously be done in polynomial time. (c) We prove that the answer to the DCLIQUE problem is yes if and only if the answer to the corresponding DS problem is yes. "$\Rightarrow$:" if there is a clique of size k in G, the corresponding clique in G' is the subgraph with k' vertices and m' edges. "$\Leftarrow$:" if there is a subgraph H' of G' such that H' has k' vertices and at least m' edges, H' must be a clique of size k'. Therefore the corresponding subgraph H in G must be a clique of size k.