CSE 101
Sample Midterm: Spring, 2003
Answer Key

Answer all questions. Give informal (at least) proofs for all answers. Grading will be on completeness and logical correctness, as well as correctness. For the data structure question, the efficiency of your solution will also be taken into account.

Analyzing loops-10pts Consider the following iterative algorithm, that uses an $\Theta(I)$ time procedure $proc(I)$.

Algorithm(n: positive integer);

1. begin;
2. $T \leftarrow 1$;
3. While $T^2 \leq n$ do:
4. begin;{while}
5. $\quad proc(T^2)$;
6. $\quad T{+}{+}$;
7. end;{while}
8. end;

Give a time analysis, up to Θ, for this algorithm.

After the i'th iteration, the value of T is i, and the loop ends when $T \geq \sqrt{n}$. Thus, the i'th iteration, the procedure will take time $\Theta(i^2)$, since i^2 is the input; since $i \leq \sqrt{n}$ this is always $O(n)$. Thus the total time is $O(\sqrt{n}n) = O(n^{3/2})$ since the number of iterations is $\sqrt{n}$ and each iteration takes $O(n)$ time. This is also tight, since iterations $i = \sqrt{n}/2....i = \sqrt{n}$ all take time at least $ci^2 \geq cn/4 \in \Omega(n)$, so we have $\Omega(\sqrt{n})$ iterations taking $\Omega(n)$ time.

Alternately, we can characterize the time precisely as $\Theta(\sum_{i=1}^{i=\lfloor\sqrt{n}\rfloor} i^2) = \Theta((\sqrt{n}^3/6 - \sqrt{n}^2/3 + \sqrt{n}/2) = \Theta(\sqrt{n}^3) = \Theta(n^{3/2})$.

Proof of correctness - 10 pts Below is an algorithm, that, given a *sorted* array $A[1..n]$ of integers, and an integer T, decides whether there is a pair $1 \leq K < L \leq n$ with $T = A[K] + A[L]$. After the algorithm, there is a proof of correctness, with some parts missing. Fill in the missing sections in the proof to get a complete proof of correctness.

IsSum[A[1,,n]: sorted array of integers, T: integer];

1. $Found \leftarrow False$;
2. $I \leftarrow 1$;
3. $J \leftarrow n$;
4. While $I < J$ and NOT $Found$ do:
5. begin;{while}
6. $\quad$ IF $A[I] + A[J] = T$ THEN $Found \leftarrow True$;
7. $\quad$ IF $A[I] + A[J] > T$ THEN J– ELSE I++;
8. end; {while}
9. Return Found.

Proof of correctness: (with gaps filled in) Let A[1..n] be a sorted array, i.e., $A[1] \leq A[2]... \leq A[n]$ Let T be any integer. We need to show that if $T = A[K] + A[L]$ for some $1 \leq K < L \leq n$ then $IsSum$ returns $True$; and coversely, that if $IsSum$ returns $True$, then for some $1 \leq K < L \leq n$, $T = A[K] + A[L]$.

To see the first direction, we use the loop invariant method. Assume there are $1 \leq K < L \leq n$ with $T = A[K] + A[L]$. Let I_t and J_t be the values of I, J after t iterations of the while loop. The intuition is that K, L never leave the range I_t, J_t to be searched by the algorithm, i.e, that $I_t \leq K < L \leq J_t$.

We prove this by induction. First we show that it is true at $t = 0$. At initialization, $I_0 = 1$ and $J_0 = n$. Thus, $I_0 \leq K < L \leq J_0$. So the base case is true.

For the induction step, assume the statement is true after some number of loops t, i.e. that $I_t \leq K < L \leq J_t$. We want to show that after the next iteration, $I_{t+1} \leq K < L \leq J_{t+1}$.

There are three cases: if $A[I_t] + A[J_t] = T$, the algorithm halts without changing I or J, and there is nothing to prove.

If $A[I_t] + B[J_t] > T$, then in the next iteration $I_{t+1} = I_t$ and $J_{t+1} = J_t - 1$. Thus, the only way we could not have what we want to show, $I_{t+1} \leq K < L \leq J_{t+1}$, is if $L = J_t$. Now if $L = J_t$ then $T < A[I_t] + A[J_t] = A[I_t] + A[L] \leq A[K] + A[L] = T$. This is a contradiction, so we have shown by contradiction that in this case $I_{t+1} \leq K < L \leq J_{t+1}$ which is what we wanted to prove.

The final case $A[I_t] + B[J_t] < T$ is very similar. In the next iteration $I_{t+1} = I_t + 1$ and $J_{t+1} = J_t$.Thus, the only way we could not have what we want to show, $I_{t+1} \leq K < L \leq J_{t+1}$ is if $K = I_t$. Now if $K = I_t$, then $T > A[I_t] + A[J_t] = A[K] + A[J_t] \geq A[K] + A[L] = T$. This is a *contradiction*so we have shown by contradiction that in this case $I_{t+1} \leq K < L \leq J_{t+1}$, which is what we wanted to prove. Thus, the loop invariant still holds in all three cases.

The first direction follows from the loop invariant, since at the end of the loop, either we return $True$ or $I_t = J_t$. If $I_t = J_t$, by the loop invariant, we must have $I_t \leq K < LleJ_t = I_t$, which is a contradiction. Thus, if $A[K] + A[L] = T$, the algorithm returns $True$.

To see the converse direction, if $IsSum$ returns $True$, then $Found$ must be set to $True$, which can only happen in the THEN clause of line 6. Because of the IF clause of that line, we must have $A[I_t]+A[J_t] = T$ which is what we were proving, since we can pick $K = I_t$ and $L = J_t$.

Using data structures and pre-processing: Consider the following problem. The input is a $m \times n$ matrix $A[1..m][1..n]$ where each row is sorted, $A[I][1] \leq A[I][2] \leq ...A[I][n]$. The problem is to combine all of these sorted rows into a single sorted array $B[1...nm]$, with $B[1] \leq B[2] \leq .... \leq B[nm]$.

A strategy to solve this problem is as follows: Here S represents the set of smallest elements from each list that are not yet output.

AbstractMergeLists(A[1..m][1..n]: matrix of integers)

1. Initialize counters $C[I], 1 \leq I \leq m$ to 1, and a set S of possible next elements in the sorted list to the smallest element from each list, $A[1][1], A[2][1], ...A[m][1]$.
2. FOR $J = 1$ TO mn do:
3. begin; {for}
4. Let X be the smallest element in S.
5. Let I be the row X is from.
6. Delete X from S.
7. $B[J] \leftarrow X$
8. $C[I] + +$
9. IF $C[I] \leq n$ THEN Add $A[I][C[I]]$ to S.
10. end; {for}

Describe what data structures you would use in an algorithm to implement the above strategy. Describe how you initialize the data structures. Then rewrite each line of the strategy with the corresponding operations in your data structures. Give a time analysis of the resulting algorithm, in terms of both n and m.

We need to find the smallest element of S, and add and delete elements to S. That argues for a heap. However, we need to know which row the elements are from, so the heap should also have row markers. The modified algorithm becomes:

MergeLists(A[1..m][1..n]: matrix of integers)

1. Initialize counters $C[I], 1 \leq I \leq m$ to 1, and a min-heap S of pairs $(value, row)$ of size m, keyed by value and initialized to include $(A[1][1], 1), (A[2][1], 2) ... (A[m][1], m)$.
2. FOR $J = 1$ TO mn do:
3. begin; {for}
4. $(X, I) \leftarrow Min$
5. DeleteMin.
6. $B[J] \leftarrow X$
7. $C[I]++$
8. IF $C[I] \leq n$ THEN Insert($(A[I][C[I]], I)$).
9. end; {for}

Since the heap is of size m, heap operations take time $O(\log m)$. The time to initialize the counters is $O(m)$ and to build the heap is $O(m \log m)$. Then each of mn iterations of the FOR loop perform a constant number of heap operations and a constant number of other operations. Thus, the total time is $O(m + m \log m + mn \log m) = O(mn \log m)$, since $n \geq 1, \log m \geq 1$.

Divide-and-Conquer Recurrence: 25 points Consider the following recursive algorithm. Its inputs are two *sorted* n element arrays A and B (so the algorithm assumes $A[1] < A[2] < ... < A[n]$ and $B[1] < B[2] < ... < B[n]$) and a target T. The following algorithm computes whether $T = A[I] + B[J]$ for some $1 \leq I, J \leq n$. Give a recurrence relation for the time the algorithm takes, and give a brief explanation of your answer. Use the recurrence to give a time analysis.

1. Program: TargetSum(A[1..n], B[1..n]: Arrays of Integers, T: integer):
2. IF $n = 0$ return False;
3. IF $n = 1$
   THEN IF $T = A[1] + B[1]$
   THEN return True;
   ELSE return False;
4. $k \leftarrow \lceil n/2 \rceil$
5. $k' \leftarrow \lceil (n+1)/2 \rceil$
6. IF $A[k] + B[k] = T$ THEN return True;
7. IF $A[k] + B[k] > T$ THEN return
   $(TargetSum(A[1..k], B[1..k], T)$
   OR $TargetSum(A[1..k], B[k'..n], T)$
   OR $TargetSum(A[k'..n], B[1..k], T))$;
8. ELSE return
   $(TargetSum(A[k'..n], B[1..k], T)$ OR
   $TargetSum(A[1..k], B[k'..n], T)$ OR
   $TargetSum(A[k'..n], B[k'..n], T))$;

Here, OR is the Boolean OR operation on the values of the returned calls.

The algorithm either makes 3 recursive calls to arrays of roughly size $n/2$ in line 6, or makes a different 3 recursive calls to arrays of roughly size $n/2$ in line 7. Either way, 3 recursive calls are made. The rest of the algorithm is constant time.

Thus, $T(n) = 3T(n/2) + \Theta(1)$, so we can use theorem B.5 with $a = 3, b = 2, k = 0$. Since $3 > 2^0 = 1$, we are in the "bottom-heavy" case , and so $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log 3})$, or approximately $\Theta(n^{1.6..})$. This is better than $\Theta(n^2)$ for exhaustive search, but might still be possible to improve.