COSC3101 Design and Analysis of Algorithms
Jeff Edmonds & Andy Mirzaian – Fall 00-01
Midterm Test Solutions

1. **True/False:** (20 points)
For each of the statements that follow indicate only whether it is true or false by circling T (true) or F (false). Do not justify your answer. Each correct answer is worth +4 points. Each incorrect answer or no answer is worth 0 points.

   (a) [**T F**]: $2^{6\log n} + 23n^7(\log n)^4 = \mathcal{O}(\frac{n^8}{(\log n)^8})$.

   - Answer: True. The first term on the left hand side is n^6, which is dominated by the second term, which is dominated by the right hand side. Recall that $(\log n)^d = o(n)$ holds for any constant d.

   (b) [**T F**]: The running time of Insertion-Sort is $\Theta(n + I)$, where I is the number of inversions in the input array $A[1..n]$. (An *inversion* is any pair of items $A[i]$ and $A[j]$ such that $A[i] < A[j]$ but $i > j$.)

   - Answer: True. See the InsertionSort algorithm on page 8 of CLR. In the i-th iteration of the main loop, the amount of time spent is in the order of 1 plus the number of items preceding $A[i]$ that form an inversion with $A[i]$. Thus, taking the sum over all iterations, the total time is indeed $\mathcal{O}(n + I)$.

   (c) [**T F**]: In the worst case QuickSort takes $\Theta(n \log n)$ time to sort n elements.

   - Answer: False. It takes $\Theta(n^2)$ time in the worst case, and $\Theta(n \log n)$ time in the expected case.

   (d) [**T F**]: Let $A[1..n]$ be an array of n elements such that we already know $A[i] < A[i + 4]$ for all $i = 1, 2, 3, ..., n - 4$. Even with this extra information as precondition, every decision tree that completes the sorting of $A[1..n]$ must have height at least $\Omega(n \log n)$.

   - Answer: False. The 4 sorted sublists can be merged in $\Theta(n)$ time.

   (e) [**T F**]: Given an arbitrary sorted array $A[1..n]$ of reals, we can determine whether A has a majority element or not in $\Theta(\log n)$ time in the worst case. (A *majority* element in A is one that appears more than $n/2$ times.)

   - Answer: True. Since A is sorted, if A has a majority, it has to be the middle element $x = A[\lceil n/2 \rceil]$. Using two binary searches, we can find the minimum and maximum indices i and j such that $x = A[i] = A[j]$. The number of times x appears in A is $j - i + 1$. If this count is greater than $n/2$, then x is the majority; otherwise there is no majority.

2. $f(n) = n^{\Theta(1)}$

   (a) (1 point) Informally, which functions are included in the classification $f(n) = n^{\Theta(1)}$?

   - Answer: Bounded by a polynomial.

   (b) (3 points) The formal definition of $f(n) = n^{\Theta(1)}$ includes three parameters c_1, c_2, and n_0. Give this formal definition.

   - Answer: The formal definition is $\exists c_1 > 0, c_2 > 0, n_0,\ \forall n \geq n_0,\ n^{c_1} \leq f(n) \leq n^{c_2}$.

   (c) (12 points) Which of the following are $f(n) = n^{\Theta(1)}$? If so, give suitable values of c_1 and c_2 for when $n_0 = 1000000$.

   i. $f(n) = 3n^3 + 17n^2 + 4$ **Answer:** Yes, $c_1 = 3$, $c_2 = 3.1$
   ii. $f(n) = 3n^3 \log n$ **Answer:** Yes, $c_1 = 3$, $c_2 = 3.1$. Note $3n^3 \log n \leq \mathcal{O}(n^{3.1})$.
   iii. $f(n) = n^{3 \log n}$ **Answer:** No, $c = \log n$
   iv. $f(n) = 3 \log n$ **Answer:** No, c_1 would need to be zero.

v. $f(n) = 7^{3\log n}$ **Answer:** Yes. $7^{3\log n} = n^{3\log 7}$. $c_1 = c_2 = 3\log 7$.

vi. $f(n) = \lceil \log n \rceil !$ **Answer:** No. Note that $(\frac{m}{2})^{\frac{m}{2}} \leq m! \leq m^m$. Therefore, $\lceil \log n \rceil ! = (\log n)^{\Theta(\log n)} = n^{\Theta(\log \log n)} \neq n^{\Theta(1)}$.

3. **Sums & Recurrences:** (21 points)
Derive tight asymptotic bound solutions to the following. Mention the method you use for each. (For the recurrences you may assume the usual boundary condition: $T(\mathcal{O}(1)) = \mathcal{O}(1)$.)

(a) $\sum_{i=1}^{n} i^8 \times (\log i)^8 = \Theta($

- Answer: Arithmetic-like sum: $\Theta(n^9 (\log n)^8)$.

(b) $\sum_{i=1}^{n} 3^{2i} \times i^8 = \Theta($

- Answer: Geometric-like sum: $\Theta(3^{2n} \times n^8)$.

(c) $\sum_{i=1}^{n} \frac{1}{i^{1.1}} = \Theta($

- Answer: Bounded Tail: $\Theta(1)$.

(d) $T(n) = 2T(n-1) + 1,\ T(n) = \Theta($

- Answer: Unwinds to $T(n) = \sum_{i=1}^{n-1} 2^i = \Theta(2^n)$.

(e) $T(n) = 3T(\frac{n}{3}) + 3(\log n)^3,\ T(n) = \Theta($

- Answer: $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^{\frac{\log 3}{\log 3}}) = \Theta(n)$.

(f) $T(n) = 9T(n/3) + 7n \log n + 2n^2,\ T(n) = \Theta($

- Answer: $\Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^{\frac{\log 9}{\log 3}}) = \Theta(n^2) = \Theta(f(n))$. Therefore, $T(n) = \Theta(f(n) \log n) = \Theta(n^2 \log n)$.

(g) $T(n) = T(n-1) + n,\ T(n) = \Theta($

- Answer: Unwinds to $T(n) = \sum_{i=1}^{n} i = \Theta(n^2)$.

4. **Iteration & Loop Invariants:** (20 points)
We are given an arbitrary **sorted** array A[1..n] of n real numbers. Some items may appear several times in A. The problem is to find an item that occurs most often in A.

Use iteration and loop invariants to design, describe, and prove the correctness of an incremental algorithm for the above problem. Be sure to include ALL required steps.

- Answer:

MostOften(A,n)
⟨Pre-condition⟩: $A[1..n]$ is a sorted array of reals.
$i = 1, p = 1, c = null, q = 0$
loop
% ⟨loop invariant⟩

- The index i has some value in $[1..n+1]$.
- The variable p stores the number of times $A[i]$ appears in $A[1..i]$ if $i \leq n$.
- The variable c stores which element other than $A[i]$ occurs most often in $A[1..i-1]$.
- The variable q stores the number of times c appears in $A[1..i-1]$.

exit when $i = n+1$
if($i < n$ and $A[i+1] = A[i]$) then
$p = p + 1$
else
if($p > q$) then
$c = A[i]$

$q = p$
end if
$p = 1$
end if
$i = i + 1$
end loop
return(c)
⟨Post-condition⟩: Returns an element occurring the most often in $A[1..n]$.
end MostOften

⟨*preCond*⟩ & codeA ⇒ ⟨*loop − invariant*⟩: $A[1]$ appears once $A[1..1]$. No other element appears. Hence, setting $i = 1, p = 1, c = null, q = 0$ satisfies the loop invariant.

⟨*loop − invariant′*⟩ & not ⟨*exitCond′*⟩ & codeB ⇒ ⟨*loop − invariant″*⟩: Because $A[1..n]$ is a sorted array, the equal items appear together in a block. There are two cases. If $i < n$ and $A[i+1] = A[i]$, then the same block is extended one further. Hence, the number of times that $A[i] = A[i+1]$ appears in the subarray increases by one and the largest previous block does not change.
On the other hand, if $A[i+1] \neq A[i]$ or $i = n$, then the current block terminates and a new block begins. This newly completed block becomes a potential largest previous block. If it is the largest, then c and q are changed to reflect this. p is set to one to reflect that the newly created first block is of size one.

⟨*loop − invariant*⟩ & ⟨*exitCond*⟩ & codeC ⇒ ⟨*postCond*⟩: LI and exit condition give "The variable c stores which element (other than $A[n+1]$) occurs most often in $A[1..n]$. The code is such that $A[n+1]$ is never accessed and is assumed not to appear in $A[1..n]$. Hence, the variable c stores which element occurs most often in $A[1..n]$. Returning the value of c satisfies the post condition.

A measure of progress: The value i.

Progress is made (if not exit): i is incremented.

Proof that with sufficient progress, the exit condition will be met: When i reaches $n+1$, we exit.

5. **Recursion on Binary Trees:** (23 points)
We are given an arbitrary **binary search tree** T. Each node of T stores an item which is a real number. The problem is to find a closest pair of items in T, that is, a pair of items (distinct nodes) in T with minimum possible difference in value.
Design a recursive algorithm for this problem, and prove its correctness by induction.

- Answer: We will provide two answers. The first requests extra information from the friend by changing the post conditions and the second provides extra information to the friend by changing the pre conditions.

  First Answer: The routine returns a closest pair $\langle a, b\rangle$ appearing in distinct nodes with $a \leq b$. If there are fewer than two nodes in the tree then no such pair exits. Hence, the routine returns $\langle a, b\rangle$ such that $b - a = \infty$. In addition, the routine returns the minimum Min, and the maximum Max elements in the tree.

  function $\langle a, b, min, max\rangle$ $ClosestPair\,(treeObj\ T)$
  if($T = emptyTree$) then
  return $\langle -\infty, \infty, \infty, -\infty\rangle$
  else
  $\langle lefta, leftb, leftMin, leftMax\rangle = ClosestPair(T.left)$
  $\langle righta, rightb, rightMin, rightMax\rangle = ClosestPair(T.right)$
  $min = \min(leftMin, T.key)$
  $max = \max(rightMax, T.key)$

$\langle a, b\rangle$ = a closest pair from
$\{\langle lefta, leftb\rangle, \langle leftMax, T.key\rangle, \langle T.key, rightMin\rangle, \langle righta, rightb\rangle\}$.
return $\langle a, b, min, max\rangle$
end if
end $ClosestPair$

For each $n \geq 0$, let $S(n)$ be the statement, "The recursive algorithm works for *every* binary search tree T with n nodes." We will prove by strong induction on n that $\forall n \geq 0,\ S(n)$. From this, we will conclude that "The recursive algorithm works for *every* instance."

Proving $S(0)$ involves showing that the algorithm works for the empty tree. In this case, the algorithm returns $\langle a, b\rangle = \langle -\infty, \infty\rangle$, $Min = \infty$ and $Max = -\infty$. As said, these are reasonable answers, because the tree has no pairs and no minimum or maximum nodes.

Let $n > 0$. The strong induction step assumes $S(0), S(1), S(2), \ldots, S(n-1)$ and from it proves $S(n)$. In other words, it first assumes that the algorithm works for every instance of size strictly smaller than n and then proves that it works for every instance of size n. In yet other words, we assume that by "magic" a friend is able to provide the solution to any instance of the problem as long as the instance is strictly smaller than the current instance. To prove that the algorithm works for every instance T of size n, consider an arbitrary instance of size n. Because T's left and right subtrees have fewer than n nodes, we know by the induction hypothesis that the recursive call correctly returns a closest pair of items, the minimum, and the maximum from these subtrees. What remains to be shown is that the $\langle a, b\rangle$, min, and max returned by the algorithm are correct for T.

The closest pair will appear consecutively in the sorted order of the items. Because T is a binary search tree, this order will be $[T.left, leftMax, T.key, rightMin, T.right]$. The left recursion will provide the closest pair $\langle lefta, leftb\rangle$ within $T.left$ and the right recursion will provide the closest pair $\langle righta, rightb\rangle$ within $T.right$. It follows that a closest pair is one of those in $\{\langle lefta, leftb\rangle, \langle leftMax, T.key\rangle, \langle T.key, rightMin\rangle, \langle righta, rightb\rangle\}$. Our routine returns the best of these.

It is best to check to make sure that the routine (and proof) work when one or both of the subtrees are empty. If the left subtree is empty, then pairs $\langle lefta, leftb\rangle$ and $\langle leftMax, T.key\rangle$ involving it will have infinite gap. Hence, these will not be returned (unless this is the closest pair). Similarly, if the right subtree is empty, then pairs $\langle T.key, rightMin\rangle$ and $\langle righta, rightb\rangle$ will have infinite gap.

It follows that the pair $\langle a, b\rangle$ returned by the algorithm is a closest pair in T. It is easy to see that max and min are the maximum and the minimum in the tree.

By way of strong induction, we can conclude that $\forall n \geq 0,\ S(n)$, i.e., The recursive algorithm works for every instance.

Second Answer: This solution provides extra information to the friend by changing the pre conditions. In addition to a binary search tree T, an input instance includes two additional items *pre* and *post* with the requirement that *pre* is less than or equal to and *post* is greater than or equal to all the items within T. The main algorithm returns $ClosestPair(-\infty, T, \infty)$. The routine returns a closest pair $\langle a, b\rangle$ appearing in $pre \cup T \cup post$.

function $\langle a, b\rangle$ $ClosestPair\ (pre, teeObj\ T, post)$
if($T = emptyTree$) then
return $\langle pre, post\rangle$
else
$\langle lefta, leftb\rangle = ClosestPair(pre, T.left, T.key)$
$\langle righta, rightb\rangle = ClosestPair(T.key, T.right, post)$
$\langle a, b\rangle$ = a closest pair from $\{\langle lefta, leftb\rangle, \langle righta, rightb\rangle\}$.
return $\langle a, b\rangle$
end if
end $ClosestPair$

The structure of the proof is similar to that above. Given the empty tree, the routine returns the pair $\langle pre, post \rangle$ because this is the only pair available.

Now consider any tree with at least one node and assume that the friends/recursion provides the correct answer for the (smaller) left and right subtrees. The closest pair will appear consecutively in the sorted order of the items. By the preconditions, this order will be $[pre, T.left, T.key, T.right, post]$. The left recursion will provide the closest pair $\langle lefta, leftb \rangle$ within $[pre, T.left, T.key]$. The right recursion will provide the closest pair $\langle righta, rightb \rangle$ within $[T.key, T.right, post]$. Our routine returns the best of these.