CS/Math 240: Intro to Discrete Math 3/27/2011

Solutions to the Second Midterm Exam

Instructor: Dieter van Melkebeek

Problem 1

This question deals with the following implementation of binary search.

Function BinarySearch(n, A, x)

Input: An integer $n \geq 1$, an array $A[0..n-1]$ of integers that is sorted from smallest to largest, and an integer x

Output: -1 if x does not appear in A, otherwise the first index i such that $A[i] = x$

```
(1) i ← 0; j ← n − 1
(2) while i < j do
(3)     m ← ⌊(i + j)/2⌋
(4)     if A[m] < x then i ← m + 1
(5)                 else j ← m
(6) if A[i] = x then return i
(7)             else return −1
```

Part 1

We prove the following invariant.

Invariant 1. *After every iteration of the loop,* $0 \leq i \leq j \leq n-1$. *Furthermore, if* x *is in the array, its first appearance is in* $A[i..j]$.

Proof. When the program begins, it initializes $i = 0$ and $j = n-1$. Thus, $0 \leq i \leq j \leq n-1$ holds. Because the only valid indices are between 0 and $n-1$ inclusive, it also follows that if x is in A, it is in $A[i..j]$ in this case.

Now suppose that the invariant holds after some iteration of the loop. Consider the next iteration of the loop. Let i' and j' be the values of i and j after that iteration, respectively.

If $i = j$, there isn't going to be another iteration, so assume $i < j$. Since $m = \lfloor (i+j)/2 \rfloor$, we have $i \leq m \leq j$. In fact, since $i \neq j$, we have $i \leq m < j$, and $m + 1 \leq j$. Now there are two cases to consider.

Case 1: $A[m] < x$. In this case, we set $i' = m + 1$ and $j' = j$. We proved in the previous paragraph that $i \leq m + 1 \leq j$, so $i' \leq j'$. We also know that $0 \leq i \leq j \leq n-1$ by the induction hypothesis, so $0 \leq i' \leq j' \leq n-1$ holds too.

Since $A[m] < x$ and x is in A, the first occurrence of x is at index greater than m because A is sorted. Thus, x it appears at index $m + 1$ or higher. By the induction hypothesis, we also know that x first appears at index j or lower, so setting $i' = m + 1$ and $j' = j$ guarantees that if x is in A, its first occurrence is in $A[i'..j']$.

Case 2: $A[m] \geq x$. In this case we set $i' = i$ and $j' = m$. We proved that $i \leq m \leq j$, so $i' \leq j'$. Combining this with the induction hypothesis tells us that $0 \leq i' \leq j' \leq n-1$.

Since $A[m] \geq x$, x first appears at index m or lower because A is sorted. Furthermore, the induction hypothesis tells us that the first occurrence of x is at index i or more, so setting $i' = i$ and $j' = m$ maintains the invariant. □

Part 2

Assume that the algorithm terminates. This means the loop ended, and the loop condition implies $i \geq j$. By Invariant 1, we also have $i \leq j$, so $i = j$. Furthermore, Invariant 1 says that if x is in the array, its first occurrence is in $A[i..j]$. Since $i = j$, this tells us that its first occurrence is at index i. The algorithm returns i in this case, which is correct. Conversely, if x is not in the array, $A[i] \neq x$, so the algorithm correctly returns -1. This proves partial correctness.

Part 3

We describe the recursive version of `BinarySearch` as `RecursiveBinarySearch` below. We need to introduce the bounds of the array range we are searching as input, so we have them as parameters instead of n. We must do this because recursive calls are going to search only parts of the array A. We still need the parameters A and x as before.

Function RecursiveBinarySearch(i, j, A, x)

Input: An integer $n \geq 1$, an array $A[i..j]$ of integers that is sorted from smallest to largest, and an integer x

Output: -1 if x does not appear in A, otherwise the first index i such that $A[i] = x$

```
(1) if i = j then
(2)     if A[i] = x then return i
(3)                 else return −1
(4) m ← ⌊(i + j)/2⌋
(5) if A[m] < x then return RecursiveBinarySearch(m + 1, j, A, x)
(6)             else return RecursiveBinarySearch(i, m, A, x)
```

Now the calls `BinarySearch`(n, A, x) and `RecursiveBinarySearch`$(0, n-1, A, x)$ are equivalent.

Problem 2

Part 1

Function GCDa(a, b)

```
(1) if a = b then return a
(2) if a < b then return GCDa(b − a, b)
(3)         else return GCDa(a, a − b)
```

This algorithm doesn't correctly compute the greatest common divisor. In particular, it does not terminate on input $(1, 2)$. When called with this input, the algorithm makes a recursive call on line 2 with input $(2 - 1, 2) = (1, 2)$. This is the same input as the original input, so the algorithm just keeps calling itself with input $(1, 2)$ and never terminates and makes no progress towards a solution.

Part 2

Function GCDb(a, b)

(1) **if** $a > b$ **then** $a \leftarrow a - b$
(2) **else** $b \leftarrow b - a$
(3) **if** $a = b$ **then return** a
(4) **else return** `GCDb`(a, b)

This algorithm also doesn't work. Consider the input $(a, b) = (1, 1)$. Since $a = b$, the first condition is false, so line 2 executes, and we now have $(a, b) = (1, 0)$. Since $a \neq b$, the algorithm makes a recursive call on line 4 with input $(a', b') = (1, 0)$. Now $a' > b'$, but $b' = 0$, so a' doesn't actually change on line 1 in the recursive call. Again, $a' \neq b'$, so the next recursive call is with input $(1, 0)$ again. We see that the algorithm makes an infinite chain of recursive calls with this input, and thus never terminates.

Observe that the input $(1, 0)$ to the recursive call doesn't satisfy the preconditions.

Part 3

Function GCDc(a, b)

(1) **if** $a = b$ **then return** a
(2) **if** $a < b$ **then return** `GCDc`$(a, b - a)$
(3) **else return** `GCDc`$(b, a(b + 1))$

This algorithm works correctly. To show this, we prove partial correctness and termination.

Before we prove partial correctness, let's prove an additional fact about greatest common divisors. Recall that if a, b are positive integers and $a \leq b$, then $\gcd(a, b) = \gcd(a, b - a)$. We use this fact to show by induction that $\gcd(a, b) = \gcd(b, a(b + 1))$. In particular, we prove the following lemma.

Lemma 1. *For all k such that $0 \leq k \leq a$, $\gcd(b, a(b + 1)) = \gcd(b, (a - k)b + a))$.*

Proof. For the base case $k = 0$, note that $a(b + 1) = ab + a = (a - 0)b + a$, so $\gcd(b, a(b + 1)) = \gcd(b, (a - 0)b + a)$ holds.

Now suppose $\gcd(b, a(b + 1)) = \gcd(b, (a - k)b + a)$ and that $k < a$ (we make the latter assumption because we only need to prove the lemma for $k \leq a$, and this assumption allows us to invoke the fact we state in the next sentence). Recall that for any x, y such that $x \leq y$, we have $\gcd(x, y) = \gcd(x, y - x)$. Note that $a > 0$ and $k < a$, so $(a - k)b + a > b$, which means that

$\gcd(b, (a-k)b+a) = \gcd(b, (a-k)b+a-b)$. We can rewrite the second argument as $(a-k)b+a-b = (a-k-1)b+a = (a-(k+1))b+a$, so we have $\gcd(b, (a-(k+1))b+a) = \gcd(b, (a-k)b+a)$, and the latter is equal to $\gcd(b, a(b+1))$ by the induction hypothesis. □

To complete the proof that $\gcd(b, a(b+1)) = \gcd(a, b)$, just use Lemma 1 with $k = a$. Then we get $\gcd(b, a(b+1)) = \gcd(b, (a-a)b+a) = \gcd(b, a) = \gcd(a, b)$.

Now we are ready to prove partial correctness.

First assume $a = b$. Then the algorithm returns a on line 1. This is the correct result because a divides $a = b$, and no integer greater than a divides a.

Now suppose $a \neq b$. If $a < b$, then both a and $b - a$ are positive integers. Also, if a and b are positive integers, then so are b and $a(b+1)$. It follows that all recursive calls to `GCDc` are with inputs that satisfy the preconditions, which means we can assume they return the correct values. Since we know that $\gcd(a, b) = \gcd(a, b-a)$ when $a < b$, and since we showed $\gcd(a, b) = \gcd(b, a(b+1))$, the correct value returned by the recursive calls is actually $\gcd(a, b)$, and the algorithm returns that.

Before we start the proof of termination, we observe that the first argument to any recursive call to `GCDc` is at most as large as the first argument to the original call. If the call is made from line 2, the first argument stays the same, and if the call is from line 3, the first argument is $b < a$, so it decreases. Thus, to show termination, it suffices to show that line 3 is reached after some number of recursive calls. This will allow us to use strong induction on the value of the first argument to show termination.

For the base case, consider an input (a, b) with $a = 1$. The algorithm keeps making recursive calls on line 2, with first argument equal to 1 and second argument decreasing by one in each call. Thus, after $b - 1$ recursive calls, the first and the second arguments are equal, and the algorithm returns from line 1 in that recursive call. All the other recursive calls terminate, one by one, right after that.

Now suppose that the algorithm terminates whenever the value of the first argument is m such that $m \leq a$. Consider a call to `GCDc` with input $(a+1, b)$. We show by cases that `GCDc` terminates on this input too.

Case 1: $b < a+1$. The next call is with b as the first argument, and this call leads to termination by the induction hypothesis. The algorithm `GCDc` returns right after that recursive call returns.

Case 2: $b = a+1$. The algorithm returns right away in this case.

Case 3: $b > a+1$. The second argument, b, decreases by a nonzero integer amount in the next recursive call (from line 2). The second argument keeps decreasing by this amount in subsequent recursive calls until, after $\lfloor b/a \rfloor$ recursive calls, it becomes less than or equal to the first argument. At that point we reach one of the first two cases, which we have shown lead to termination. Thus, the algorithm terminates in this case as well.

This completes the proof of the inductive step, and proves termination.

Part 4

Consider the input $(3, 2)$, and note $\gcd(3, 2) = 1$. The algorithm returns 2, which isn't correct. To see this, observe that the first recursive call is from line 3, and the input is $(2, 6)$. The two further recursive calls made on this input are with inputs $(2, 4)$ and $(2, 2)$. The last recursive call returns 2, which is also what the original call returns.

Function GCDd(a, b)

(1) **if** $a = b$ **then return** a
(2) **if** $a < b$ **then return** `GCDd(`$a, b - a$`)`
(3) **else return** `GCDd(`b, ab`)`

Problem 3

Part 1

C_0 is 1; the empty sequence is the only length 0 sequence. C_1 is 2; either of the sequences 1 or 0 satisfy the requirements.

To reason about C_n, we will look by cases to all possible first numbers in a sequence S in C_n.

Case One: *S starts with 0*

In this case, the final $n - 1$ numbers in S must have no two consecutive 1s, but other than that, there are no restrictions on their form. This is because the initial 0 can be followed by a string of length $n - 1$ that has not consecutive 1s and starts with either a 1 or a 0. So the total number of strings fitting this case is equal to C_{n-1}.

Case Two: *S starts with 1*

In this case, S's second number cannot be 1, because S would then start with 11. So S actually starts with 10. Then, the final $n - 2$ numbers must have no two consecutive 1s, but other than that, there are no restrictions on their form. So the total number of strings fitting this case is equal to C_{n-2}.

These two cases cover all possibilities for a sequence S in C_n, and they are disjoint (no S can start with both 1 and 0). So $C_n = C_{n-1} + C_{n-2}$.

Part 2

The above recurrence is simply a shifted version of the Fibonacci sequence, where $C_n = F_{n+1}$. This can be argued by induction on n. The base cases $n = 0$ and $n = 1$ hold since $C_0 = 1 = F_2$ and $C_1 = 2 = F_3$. The induction step follows because the recurrence equation for C_n is the same as for F_n.

Problem 4

1. The program performs one addition operation lines (4) and (5). These lines are executed every time the loop condition is evaluated to be true. The initial difference between n and i is $n - 3$ as i is initialized to 3. After each iteration of the loop, this difference goes down by one. So, there would be $n - 2$ iterations before i becomes greater than n. Hence, in all there would be $2(n - 2)$ additions.

2. The provided program is recursive. So we would set up a recurrence to solve this problem. Let $T(n)$ be the number of additions required for computing F_n. No additions are performed

for $n \leq 2$. So, we have $T(1) = T(2) = 0$. For computing F_n where $n > 2$, we make two recursive calls, and the results obtained from those recursive calls. Therefore, $T(n) = T(n-1) + T(n-2) + 1$.

Now, we need to solve the recurrence obtained. The recurrence is closely related to the one for the Fibonacci sequence but isn't exactly the same. Let us evaluate $T(n)$ for a few values of n.

$$T(3) = T(2) + T(1) + 1 = 0 + 0 + 1 = 1$$
$$T(4) = T(3) + T(2) + 1 = 1 + 0 + 1 = 2$$
$$T(5) = T(4) + T(3) + 1 = 2 + 1 + 1 = 4$$
$$T(6) = T(5) + T(4) + 1 = 4 + 2 + 1 = 7$$
$$T(7) = T(6) + T(5) + 1 = 7 + 4 + 1 = 12$$

If you look carefully, you realize that for all the n for which we have evaluated $T(n)$, $T(n) = F_n - 1$. Hence, we conjecture that $T(n) = F_n - 1 \ \forall n \geq 1$. We prove this claim using induction on n. As base case, we have $n = 1, 2$. Since $F_1 = 1, F_2 = 1$, the base case holds. Next, assume that $\forall 2 \leq k \leq n$, $T(k) = F_k - 1$. We prove $T(n+1) = F_{n+1} - 1$. We know that $T(n+1) = T(n) + T(n-1) + 1$. Using induction hypothesis, we get that $T(n+1) = F_n - 1 + F_{n-1} - 1 + 1 = F_n + F_{n-1} - 1 = F_{n+1} - 1$. This completes the induction step. Hence, by strong induction, we have shown that the number of additions required for computing F_n by the given program is $F_n - 1$.

3. In this problem we need to develop an algorithm for computing Fibonacci numbers using $O(\log n)$ additions and multiplications. We can write a solution that makes use of the given recurrences to carry out the required computations. The function NaiveFastFib is one such solution.

Function NaiveFastFib(n)

Input: n - integer, $n \geq 1$
Output: F_n

```
(1) if n ≤ 2 then return 1
(2) if n is even then
(3)     a ← NaiveFastFib(n/2 − 1)
(4)     b ← NaiveFastFib(n/2)
(5)     return (2a + b)*b
(6) else
(7)     a ← NaiveFastFib((n + 1)/2)
(8)     b ← NaiveFastFib((n − 1)/2)
(9)     return a ∗ a + b ∗ b
```

The problem with this solution is that it needs $O(n)$ additions and multiplications to carry out the required computations. This can be shown using the recursion tree for the algorithm. The recursion tree will have $O(n)$ nodes with a constant local cost associated with each node. Hence, we would get a total cost of $O(n)$.

In order to achieve a total cost of $O(\log n)$, we need to avoid the second recursive call that we make to the $NaiveFastFib()$ function. To do this, we compute the two values required for computation of F_n in a single recursive call. The algorithm follows –

Function FastFib(n)

Input: n - integer, $n \geq 1$
Output: F_n, F_{n+1}

(1) **if** $n = 1$ **then return** $1, 1$
(2) **else if** $n = 2$ **then return** $1, 2$
(3) **else if** n *is even* **then**
(4) $(a, b) \leftarrow$ `FastFib`$((n/2) - 1)$
(5) $c \leftarrow a + b$
(6) **return** $(2a + b)b, b^2 + c^2$
(7) **else**
(8) $(a, b) \leftarrow$ `FastFib`$((n - 1)/2)$
(9) **return** $a^2 + b^2, (2a + b)b$

Function Fib(n)

Input: n - integer, $n \geq 1$
Output: F_n

(1) $(a, b) \leftarrow$ `FastFib`(n)
(2) **return** a

Let us argue for the correctness of the program FastFib. We start with partial correctness. Case 1: $n = 1$. In this case, the program returns $(1, 1)$ which is correct.

Case 2: $n = 2$. In this case, the program returns $(1, 2)$ which is also correct.

Case 3: n is even. From the given recurrences, we observe that computation of F_n requires $F_{(n/2)-1}$ and $F_{n/2}$. Similarly to compute F_{n+1}, we need $F_{n/2}$ and $F_{(n/2)+1}$. The recursive call to `FastFib` in line(4) has the argument $(n/2) - 1$, which is at least 1 since $n > 2$. So the preconditions are satisfied for `FastFib`. Hence, `FastFib` in line (4) returns $a = F_{(n/2)-1}, b = F_{n/2}$. Therefore, $c = F_{(n/2)+1}$. Using the provided recurrences, we can verify that the program correctly returns F_n and F_{n+1}.

Case 3: n is odd. For computing F_n, we need $F_{(n-1)/2}$ and $F_{(n+1)/2}$ which are required for computation of F_{n+1} as well. The argument to `FastFib` in line(8) has the argument $(n-1)/2$, which is at least 1 since $n > 2$. So the preconditions are satisfied for `FastFib`. Hence, `FastFib` in line (4) returns $a = F_{(n-1)/2}, b = F_{(n+1)/2}$. Using the provided recurrences, we can verify that the program correctly returns F_n and F_{n+1}.

This completes the proof for partial correctness.

Termination of the `FastFib` program is evident from the fact that the recursive call to `FastFib` is on an argument $< n$. Therefore, the program will eventually reach the case when $n = 1$ or $n = 2$ for which the program terminates.

Let $T(n)$ = number of additions and multiplications required for computation of F_n using `FastFib`. We argue that $T(n) = O(\log n)$. If we draw out a recursion tree, we would observe that each level of the tree has a constant contribution to the total cost. This is because for computation of F_n, we only perform constant number of addition and multiplication operations, other than those performed in the recursive calls. Since there is only one recursive call at each level, it suffices to prove that the height of recursion tree is $O(\log n)$. At level l, the recursive call is on a value less than $n/2^l$ and the recursion stops when this value is 2 or less. From this inequality, we can conclude that $l = O(\log n)$, and thus that $T(n) = O(\log n)$.

Problem 5

For each function f given in the left column, we need to choose one expression on the right such that $f = O(g)$, the constraint being that each expression on right can only be used once.

(a) 3.2^n is $O(2^n)$.

(b) As n becomes very large, the terms $2n^4$ and n^3 dominate the rest of the terms. Hence, $\dfrac{2n^4+1}{n^3+2n-1} \sim 2n$. This implies that $\dfrac{2n^4+1}{n^3+2n-1} = O(n)$.

(c) $(n^5+7)(n^5-7) = n^{10} - 49$, which is $O(n^{10})$.

(d) As n becomes very large, the terms n^4 and n^2 dominate the numerator and denominator respectively. Hence, $\dfrac{n^4 - n\log n}{n^2+1} \sim \dfrac{n^4}{n^2} = n^2$. This implies that $\dfrac{n^4 - n\log n}{n^2+1} = O(n^2)$.

(e) $\dfrac{n\log n}{n-5} \sim \log n$, so $\dfrac{n\log n}{n-5} = O(\log n)$.

(f) Since $2^3 = 8 < 10$, therefore $2^{3n} < 10^n$. Hence, $2^{3n+1} = O(10^n)$.

(g) $\displaystyle\prod_{i=1}^{n} i \le \prod_{i=1}^{n} n = n^n$. Hence, $\displaystyle\prod_{i=1}^{n} i = O(n^n)$.

(h) For $n \ge 2$, $(n-2)\log(n^3+4) \le n\log(n^4) = 4n\log n$. Therefore, this function is $O(n\log n)$.

Extra Credit

Part 1

We have seen in the solution to problem 2 that this version of the algorithm fails on input $(1, 2)$. We show that it actually fails on all inputs (a, b) such that $a \neq b$. That is, the algorithm works only on inputs (a, b) where $a = b$.

If $a = b$, the algorithm returns a right away, and that is the correct answer because a divides both a and b, and no integer greater than a is a divisor of a.

Now we show that if $a \neq b$, the algorithm fails. There are two cases to consider.

Case 1: $a < b$. On the initial input (a, b), the algorithm comes to line 2, and the first recursive call is with input $(a', b') = (b - a, b)$. Note that $b' > a'$, so the recursive call also gets to line 2, and

makes a call with input $(b' - a', b') = (b - (b - a), b) = (a, b)$. Now we see that the algorithm makes an infinite chain of recursive calls and never terminates. In particular, the calls alternate between inputs (a, b) and $(b - a, b)$.

Case 2: $a > b$. We can use the same argument as for case 1, except the recursive calls now alternate between inputs (a, b) and $(a, a - b)$.

Part 2

The algorithm works on all inputs (a, b) such that $a \neq b$. To see this, recall the recursive version of the simple algorithm for computing greatest common divisors from Lecture 10. We rewrite this algorithm so that it looks more similar to the algorithm `GCDb` we are interested in.

Function GCD(a, b)

(1) **if** $a = b$ **then return** a
(2) **if** $a > b$ **then** $a \leftarrow a - b$
(3) **else** $b \leftarrow b - a$
(4) **return** `GCD(`a, b`)`

This looks almost like `GCDb`. The only difference is that `GCD` checks for equality of a and b right away, so it does not miss the opportunity to return a when $a = b$. On the other hand, if $a \neq b$ then the first if in `GCD` has no effect. In `GCDb`, after the first if statement executes, there will be either another recursive call to `GCDb`, or the algorithm will return right away, which is the same behavior as `GCD` (except that `GCD` makes a recursive call where the two arguments are the same before it returns, but that doesn't change the fact that the two algorithms give the same result).

Recall that the algorithm failed on input $(1, 1)$. In fact, it fails on any input (a, b) with $a = b$. To see this, consider the call `GCDb(`a, b`)`. Since $a = b$, the first condition is false, so line 2 executes, and we now have $b = 0$. Since $a \neq 0$, the algorithm makes a recursive call on line 4 with input $(a', b') = (a, 0)$. Now $a' > b'$, but $b' = 0$, so a' doesn't actually change on line 1 in the recursive call. Since $a' \neq b'$, the next recursive call is with input $(a, 0)$ again. We see that the algorithm makes an infinite chain of recursive calls with this input, and thus never terminates.

Part 3

This algorithm is correct, so it works on all inputs (a, b) where a and b are positive integers.

Part 4

The algorithm works correctly if $a = b$.

Observe that on input (a, b) with $a > b$, `GCDd` makes a recursive call with input (b, ab), and $\gcd(b, ab) = b$ for any positive integers a, b. On input (b, ab), `GCDd` makes $a - 1$ recursive calls to itself, with inputs $(b, (a - 1)b)$, $(b, (a - 2)b)$, $\ldots$, (b, b), and the last recursive call with input (b, b) returns b. But $\gcd(a, b) = b$ only if b divides a, so `GCDd` behaves correctly on inputs (a, b) with $a > b$ only if b divides a.

Finally, suppose $a < b$. Then we can write $b = aq + r$ where $q \in \mathbb{N}$ and $0 \leq r < a$. If $r = 0$, the algorithm makes $q - 1$ recursive calls from line 2. The last recursive call is with input (a, a), which

returns a. This is correct because in this case a divides b, and no integer greater than a divides a. If $r > 0$, the algorithm makes q recursive calls from line 2. The last recursive call is with input (a, r) where $a > r$. We showed earlier that the algorithm behaves correctly in this case only if r divides a.

Thus, `GCDd` works on inputs (a, b) such that one input divides the other, or such that the remainder of b after dividing by a is a divisor of a. The algorithm fails on all other inputs.