Math 416 Homework 2 Solutions

Updated 9 October, 2005

(1) (CRLS 3-4) **Suppose that f and g are asymptotically positive. Prove or disprove each of the following statements.**

(a) **If $f(n) = O(g(n))$, then $g(n) = O(f(n))$.** This is false. If $f(n) = 1$ and $g(n) = n$ for all natural numbers n, then $f(n) \leq g(n)$ for all natural numbers n, so $f(n) = O(g(n))$. However, suppose $g(n) = O(f(n))$. Then there are a natural number n_0 and a constant $c > 0$ such that $n = g(n) \leq cf(n) = c$ for all $n \geq n_0$, which is impossible.

(b) **$f(n) + g(n) = O(\min\{f(n), g(n)\})$.** This is false. Let f and g be as in (1a). Suppose that $f(n) + g(n) = O(\min\{f(n), g(n)\})$. Then there are a natural number n_0 and a constant $c > 0$ such that $n + 1 = f(n) + g(n) \leq c \min\{f(n), g(n)\} = c$ for all $n \geq n_0$, which is impossible.

(c) **If $f(n) = O(g(n))$ *and* if $f(n), \log g(n) \geq 1$ for sufficiently large n, then $\log(f(n)) = O(\log(g(n)))$.** Notice that this is *not* true if we remove the condition that $\log g(n) \geq 1$ for sufficiently large n. For example, suppose the logarithm is taken base 2 and we put $f(n) = 2$, $g(n) = 1$. Then obviously $f(n) = O(g(n))$, but $\log f(n) = 1 \neq O(0) = O(\log g(n))$. Thus any argument that did not use the condition that $\log g(n) \geq 1$ was incorrect. (The reason for the condition $f(n) \geq 1$ is that we want to be sure that $\log f(n) \geq 0$.)
With the additional conditions imposed, the statement is true. Since $f(n) = O(g(n))$, there are a natural number n_0 and a constant $c > 0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$. Since log is increasing, $\log f(n) \leq \log(cg(n)) = \log(c) + \log(g(n))$ for $n \geq n_0$. Put $c' = \log(c) + 1$. Then

$$\log f(n) \leq \log(c) \log(g(n)) + \log(g(n)) = c' \log(g(n))$$

for $n \geq n_0$ (since $\log g(n) \geq 1$), so $\log f(n) = O(\log g(n))$.

(d) **If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.** This is false. If $f(n) = 2n$ and $g(n) = n$ for all natural numbers n, then $f(n) \leq 2g(n)$ for all natural numbers n, so $f(n) = O(g(n))$. However, suppose $2^{f(n)} = O(2^{g(n)})$. Then there are a natural number n_0 and a constant $c > 0$ such that $4^n = 2^{2n} = 2^{f(n)} \leq c2^{g(n)} = c2^n$, i.e., $2^n = (4/2)^n \leq c$, for all $n \geq n_0$, which is impossible.

(e) **$f(n) = O((f(n))^2)$.** This is false. Let $f(n) = \frac{1}{n}$ for all natural numbers n. Suppose that $f(n) = O((f(n))^2)$. Then there are a natural number n_0 and a constant $c > 0$ such that $\frac{1}{n} = f(n) \leq c(f(n))^2 = c\frac{1}{n^2}$, i.e., $n = \frac{n^2}{n} \leq c$, for all $n \geq n_0$, which is impossible.

(f) **If $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$.** This is true. There are a natural number n_0 and a constant $c > 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$. Put $c' = \frac{1}{c} > 0$. Then $g(n) \geq c'f(n)$ for all $n \geq n_0$, so $g(n) = \Omega(f(n))$.

(g) **$f(n) = \Theta(f(n/2))$.** This is false. Let f be as in (1d). Suppose $f(n) = \Theta(f(n/2))$. Then there are a natural number n_0 and a constant $c_2 > 0$ such that $2^n = f(n) \leq c_2 f(n/2) = c_2 2^{n/2} = c_2\sqrt{2}^n$, i.e., $\sqrt{2}^n = (2/\sqrt{2})^n \leq c_2$, for all $n \geq n_0$, which is impossible. (Of course there would also be a constant $c_1 > 0$ such that $f(n) \geq c_1 f(n/2)$, but we don't need that to get a contradiction.)

(h) $f(n)+o(f(n)) = \Theta(f(n))$. This is true. In fact, the stronger statement $f(n)+O(f(n)) = \Theta(f(n))$ is true. (This statement is stronger because there are more functions allowed on the left-hand side; that is, we are asserting that more functions belong to $\Theta(f(n))$.) Suppose that $g(n) = O(f(n))$. Then there are a natural number n_0 and a constant $c > 0$ such that $g(n) \leq cf(n)$ for all $n \geq n_0$. By making n_0 larger, if necessary, we may also assume that $g(n) \geq 0$ for all $n \geq n_0$. Put $c' = c + 1$. Then $f(n) \leq f(n) + g(n) \leq f(n) + cf(n) = c'f(n)$ for all $n \geq n_0$, so $f(n) + g(n) = \Theta(f(n))$.

(2) (CLRS 4.1-5) **Show that, if** $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$, **then** $T(n) = O(n \log n)$. In this problem, we *cannot*, in the proof, ignore the 17 in the recurrence just because it is a lower-order term. This strategy is OK when making a *guess*, but it doesn't prove anything.

We need to choose a base for the logarithm in the statement (although any base will do); for convenience, we will take the logarithm base 2. It turns out that it is very difficult to prove using the substitution method that $T(n) \leq cn \log_2 n$ for $n \geq n_0$, no matter how big we choose c and n_0 to be. Remembering a trick from the text, we think of subtracting lower-order terms. Ordinarily the most natural guess would be that we would want to subtract a multiple of n, but we will actually subtract an even lower-order term. Thus our guess will be that there are a natural number n_0 and constants $c, b > 0$ such that $T(n) \leq c(n - b) \log_2 n$. The recurrence relation doesn't even make sense for $n > 34$ (otherwise, $\lfloor n/2 \rfloor + 17 \geq n$), so we will try $n_0 = 35$. In order to ensure that the right-hand side is nonzero, we will then require that $b \leq 34$. As long as $c \geq \frac{T(35)}{\log_2 35}$, we have that $T(n) \leq c(n - b) \log_2 n$ for $n = 35$. This is our base case.

Now suppose that $n_1 \geq 36$ and we have proven the result for $35 \leq n < n_1$. Notice that $35 \leq \lfloor n_1/2 \rfloor + 17 < n_1$, so

$$T(n_1) = 2T(\lfloor n_1/2 \rfloor + 17) + n_1 \leq 2c(\lfloor n_1/2 \rfloor + 17 - b) \log_2(\lfloor n_1/2 \rfloor + 17) + n_1. \tag{1}$$

What we really want is

$$T(n_1) \leq c(n_1 - b) \log_2 n_1. \tag{2}$$

The easiest way for this to be true is if the constant in front of the logarithm in the right-hand side of (1) is no more than the constant in front of the logarithm in the right-hand side of (2), i.e., if $2c(\lfloor n_1/2 \rfloor + 17 - b) \leq c(n_1 - b)$. By cancelling the common c and replacing $\lfloor n_1/2 \rfloor$ on the left-hand side by $n_1/2$ (which makes it bigger, at worst), we see that it suffices to have $2(n_1/2 + 17 - b) \leq n_1 - b$, i.e., $b \geq 34$. Since we also required $b \leq 34$ above, this gives the unique choice $b = 34$. Now notice that, since $n_1 \geq 36$, we have $n_1 \leq 18(n_1 - 34) \leq 18(n_1 - b)$ and $17 \leq \frac{17}{36} n_1$, so $\lfloor n_1/2 \rfloor + 17 \leq (1/2)n_1 + (17/36)n_1 = (35/36)n_1$. Thus, by (1),

$$T(n_1) \leq c(n_1 - b) \log_2((35/36)n_1) + 18(n_1 - 34) = c(n_1 - b)(\log_2 n_1 + \log_2(35/36) + 18/c).$$

Since what we want is $T(n_1) \leq c(n_1 - b) \log_2 n_1$, we are really asking that the remaining terms be negative, i.e., that $18/c \leq -\log_2(35/36)$. This

looks bad until we remember that $-\log_2(35/36) = \log_2(36/35) > 0$. (Here the specific numbers aren't important, only that $36/35 > 1$.) Thus, so long as $c \geq 18/\log_2(36/35)$ (a condition which does not involve n_1, hence is legal), the remaining terms are indeed negative and we have $T(n_1) \leq c(n_1 - b)\log_2 n_1$, as desired.

(3) (CLRS 4.2-4) **Suppose that $T(n) = T(n-a) + T(a) + cn$, where $a \geq 1$ and $c > 0$ are constants. By using a recurrence tree, find an explicit (and simple) function f so that $T(n) = \Theta(f(n))$. Justify your answer by using the substitution method.** As usual, when drawing the recurrence tree, we will suppose that n is an exact multiple of a. We will not actually reproduce the recurrence tree in this solution set, but note that every non-leaf node in the tree, corresponding to a subproblem of size m, sprouts two children, corresponding to subproblems of size $m-a$ and a. One of these problems (the one of size a) is a leaf; the other is a leaf only if $m - a = a$, i.e., $m = 2a$. There are $(n/a) - 1$ levels in the tree, labelled $i = 0, \ldots, (n/a) - 2$, where the non-leaf node at level i corresponds to a problem of size $n - ia$. (If we try to go to level $i = (n/a) - 1$, then both subproblems are leaves.) Since the overhead for a problem of size $n - ia$ is $c(n - ia)$, the cost of the non-leaf nodes is

$$\sum_{i=0}^{(n/a)-2} c(n-ia) = cn((n/a)-1) - ca \sum_{i=0}^{(n/a)-2} a$$
$$= cn((n/a)-1) - ca\frac{((n/a)-1)((n/a)-2)}{2} = \Theta(n^2).$$

(Notice that there are $((n/a) - 2) - 0 + 1 = (n/a) - 1$ terms in the sum, not $(n/a) - 2$.) There are (n/a) leaves in the tree, one at each of the levels labelled $i = 1, \ldots, (n/a) - 2$ and two at the level labelled $i = (n/a) - 1$. Thus the total cost of the leaves is $\Theta(n/a) = \Theta(n) = o(n^2)$, so the cost of the entire tree is $T(n) = \Theta(n^2) + o(n^2) = \Theta(n^2)$ (by (1h)).

However, this is just a guess. (We ignored the case where n is not a multiple of a. Although it is possible in this simple case to handle that possibility explicitly, usually it will not be possible.) Let us try to prove that $T(n) = \Theta(n^2)$, that is, that there are a natural number n_0 and constants $c_1, c_2 > 0$ such that $c_1 n^2 \leq T(n) \leq c_2 n^2$ for $n \geq n_0$. (Be careful not to confuse the constants c_1, c_2 here with the constant c occurring in the problem statement!) Since the right-hand side is never 0, we will try $n_0 = 1$. Notice that the recurrence relation makes sense only for $n > a$, so we will have to handle the terms $n \leq a$ separately. If $c_1 \leq \frac{T(1)}{1^2}, \ldots, \frac{T(a)}{a^2}$ and $c_2 \geq \frac{T(1)}{1^2}, \ldots, \frac{T(a)}{a^2}$, then $c_1 n^2 \leq T(n) \leq c_2 n^2$ for $n \leq a$. These a statements will serve as our "base case". (This precision about which statements will serve as the base case is more than is necessary in the solution of a homework or exam problem, unless specifically otherwise stated. We include it here only for completeness.)

Now suppose that we have proven the statement for all $n < n_1$, and we want to prove it for $n = n_1$. By our choice of base case, we may suppose that $n_1 > a$. Then $T(n_1) = T(n_1 - a) + T(a) + cn_1 \leq c_2(n_1 - a)^2 + T(a) + cn_1 = c_2 n_1^2 + (c - 2ac_2)n_1 + (a^2 c_2 + T(a))$. It is tempting at this point to say

that the terms $(c - 2ac_2)n_1 + (a^2c_2 + T(a))$ are lower-order terms, hence negligible. *This is not OK.* Remember that we saw in class that this kind of reasoning, apparently perfectly reasonable, can be used to "prove" that (say) $T(n) = 2T(n/2) + n$ has solution $T(n) = \Theta(n)$, which we know is untrue.

Thus we must work a little harder. Now remember that $n_1 > a$, so $a^2c_2 < ac_2n_1$. Therefore, $T(n_1) < c_2n_1^2 + (c - ac_2)n_1 + T(a)$. What we really want is $T(n_1) \leq c_2n_1^2$. This is true if $(c - ac_2)n_1 + T(a) \leq 0$ or, rearranging, if $c_2 \geq \frac{c}{a} + \frac{T(a)}{n_1 a}$. However, this condition is not OK, because the right-hand side involves the variable n_1. To get rid of the dependence, just note that $n_1 > a$, so it's enough to require $c_2 \geq \frac{c}{a} + \frac{T(a)}{a^2}$ (a stronger requirement, and one that doesn't involve n_1).

We must also show that $T(n_1) \geq c_1n_1^2$. This proof looks similar, but is actually a little simpler. We have that $T(n_1) = T(n_1 - a) + T(a) + cn_1 \geq c_1(n_1 - a)^2 + T(a) + cn_1 = c_1n_1^2 + (c - 2ac_1)n_1 + (a^2c_1 + T(a))$. Since $a, c_1, T(a) \geq 0$, we may drop all these terms and conclude that $T(n_1) \geq c_1n_1^2 + (c - 2ac_1)n_1$. What we really want is $T(n_1) \geq c_1n_1^2$. This is true if $(c - 2ac_1)n_1 \geq 0$, i.e., if $c - 2ac_1 \geq 0$. Rearranging, we see that it is enough to require that $c_1 \leq \frac{c}{2a}$. Since this condition doesn't depend on n_1, we are finished.

(4) (CLRS 4-5) **The Fibonacci numbers F_i satisfy the recurrence $F_i = F_{i-2} + F_{i-1}$, $i \geq 2$, and $F_0 = 0$, $F_1 = 1$. Define the generating function $\mathcal{F}$ by $\mathcal{F}(z) = \sum_{i=0}^{\infty} F_i z^i$.**

(a) **Show that $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.** In this setting, writing out the two series and showing that the first few terms "match up" is not sufficient. What is needed is an algebraic proof. Actually, this is not much harder.
We have

$$z + z\mathcal{F}(z) + z^2\mathcal{F}(z) = z + \sum_{i=0}^{\infty} F_i z^{i+1} + \sum_{i=0}^{\infty} F_i z^{i+2} = z + \sum_{i=1}^{\infty} F_{i-1} z^i + \sum_{i=2}^{\infty} F_{i-2} z^i,$$

by replacing the variable i in the first sum by $i - 1$, and the variable i in the second sum by $i - 2$. Let's look at the z^1-terms separately. We get

$$z + z\mathcal{F}(z) + z^2\mathcal{F}(z) = (1 + F_{1-1})z^1 + \sum_{i=2}^{\infty} (F_{i-1} + F_{i-2}) z^i.$$

Remembering that $F_0 = 0$, $F_1 = 1$ and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$, this becomes

$$z + z\mathcal{F}(z) + z^2\mathcal{F}(z) = F_0 z^0 + F_1 z^1 + \sum_{i=2}^{\infty} F_i z^i;$$

but this last expression is just $\mathcal{F}(z)$, as desired.

(b) **Show that $\mathcal{F}(z) = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi} z}\right)$, where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$.** Since $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$, we have $(1 - z - z^2)\mathcal{F}(z) = z$,

i.e., $\mathcal{F}(z) = \frac{z}{1-z-z^2}$. On the other hand,

$$\frac{1}{\sqrt{5}}\Big(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi}z}\Big) = \frac{((1-\hat{\phi}(z)) - (1-\phi(z)))/\sqrt{5}}{(1-\phi z)(1-\hat{\phi}z)} = \frac{(\phi-\hat{\phi})z/\sqrt{5}}{1-(\phi+\hat{\phi})z+\phi\hat{\phi}z^2}.$$

Since $\phi - \hat{\phi} = \sqrt{5}$, $\phi + \hat{\phi} = 1$ and $\phi\hat{\phi} = -1$, these two expressions are equal, as desired. (In general, one would have to factor $1 - z - z^2 = -(z - \phi^{-1})(z - \hat{\phi}^{-1})$ – that is, $1 - z - z^2 = -(z + \hat{\phi})(z + \phi)$ – and perform a partial fractions decomposition to find constants A and B such that $\frac{z}{1-z-z^2} = \frac{A}{z-\phi^{-1}} + \frac{B}{z-\hat{\phi}^{-1}}$.)

(c) **Show that** $\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)z^i$**.** We have that $\frac{1}{1-\phi z} = \sum_{i=0}^{\infty}(\phi z)^i$ and $\frac{1}{1-\hat{\phi}z} = \sum_{i=0}^{\infty}(\hat{\phi}z)^i$ (as can be verified of multiplying both sides of the first equation by $1 - \phi z$, and similarly for the second equation). Therefore,

$$\mathcal{F}(z) = \frac{1}{\sqrt{5}}\Big(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi}z}\Big) = \sum_{i=0}^{\infty}\frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)z^i,$$

as desired.

(d) **Prove that, for** $i > 0$**,** $\phi^i/\sqrt{5}$**, rounded to the nearest integer, is** F_i**.** By (4c), $F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)$ for $i > 0$ (in fact for $i \geq 0$). Since $5 < 9$, $\sqrt{5} < \sqrt{9} = 3$, so $\hat{\phi} = \frac{1-\sqrt{5}}{2} > -1$. Thus $|\hat{\phi}^i| \leq |\hat{\phi}| = \frac{\sqrt{5}-1}{2} < \frac{\sqrt{5}}{2}$, $i \geq 0$, so $F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)$ is less than $\frac{1}{2}$ away from $\frac{1}{\sqrt{5}}\phi^i$. Since F_i is also an integer, this means that it is the nearest integer to $\frac{1}{\sqrt{5}}\phi^i$. (In fact, we have shown that there is no ambiguity; that is, that there are not two nearest integers.)

(e) **Prove that** $F_{i+2} \geq \phi^i$ **for** $i \geq 0$**.** By (4d), $F_{i+2} = \phi^i\big(\frac{\phi^2}{\sqrt{5}} - \big(\frac{\hat{\phi}}{\phi}\big)^i\frac{\hat{\phi}^2}{\sqrt{5}}\big)$. Since $|\hat{\phi}| = \frac{\sqrt{5}-1}{2} \leq \frac{\sqrt{5}+1}{2} = \phi$, we have that $|(\hat{\phi}/\phi)^i| \leq 1$. Thus, using (4c) again, $F_{i+2} \geq \phi^i\big(\frac{\phi^2-\hat{\phi}^2}{\sqrt{5}}\big) = \phi^i F_2 = \phi^i$, as desired.

(5) (CLRS 4.3-2) **The running time** $T(n)$ **of an algorithm** A **on an input of size** n **satisfies** $T(n) = 7T(n/2) + n^2$**, whereas the running time** $T'(n)$ **of an algorithm** A' **on an input of size** n **satisfies** $T'(n) = aT(n/4) + n^2$ **for some constant** a**. What is the largest integer** a **such that** A' **is asymptotically faster than** A**?** Since $4 < 7$, we have $2 = \log_2 4 < \log_2 7$, so $n^2 = O(n^{\log_2 7 - \varepsilon})$ for some $\varepsilon > 0$ (specifically, $\varepsilon = \log_2 7 - 2$). Thus, by the Master Theorem, $T(n) = \Theta(n^{\log_2 7})$.

For $T'(n)$, there are three possibilities: $2 < \log_4 a$, $2 = \log_4 a$ or $2 > \log_4 a$. Since we are interested in a as large as possible, if there are any integral values of a such that $2 < \log_4 a$ and A' runs asymptotically faster than A (i.e., $T'(n) = o(T(n))$), then we need not consider the other two cases. In fact, if $2 < \log_4 a$, i.e., $16 = 4^2 < 4^{\log_4 a} = a$, then $n^2 = O(n^{\log_4 a - \varepsilon'})$ for some $\varepsilon' > 0$, so $T'(n) = \Theta(n^{\log_4 a})$. Thus $T'(n) = o(T(n)) = o(n^{\log_2 7})$ if and only if $\log_4 a < \log_2 7$, i.e., if and only if

$$a = 4^{\log_4 a} < 4^{\log_2 7} = (2^2)^{\log_2 7} = (2^{\log_2 7})^2 = 49.$$

Thus there are integers $a > 16$ such that A' runs asymptotically faster than A, so we need not consider the possibility that $2 = \log_4 a$ or $2 > \log_4 a$.

The computation above shows that the largest *integral* value of a such that A' runs asymptotically faster than A is $a = 48$.

(6) (CRLS 4-6) **Suppose one has a collection of n computer chips, some good and some bad, and a device which accepts a pair of chips and uses each to test the other. (A more vivid description is given in the text.) If two good chips test one another, each is reported good; if a good chip tests a bad chip, the bad chip is reported bad; and a bad chip returns a random answer when testing any other chip, good or bad.**

(a) **Show that, if more than $n/2$ chips are bad, this device is not sufficient to identify which chips are bad.** Note that any test results, resulting from any combination of chips, could have been mimicked by a collection of all-bad chips. Thus we cannot know for sure that we do *not* have all bad chips.

We show also that we cannot necessarily know for sure that we have all bad chips. (There are some test results that will definitively indicate bad chips – for example, if the tests $c_1c_2, c_1c_2, \ldots c_1c_n, c_1c_n$, performed successively, yield results $GG, BB, \ldots, GG, BB$ – so the best we can do is to show that the test results *might* be such that we cannot draw a conclusion.)

(b) **Consider the problem of finding one good chip. Show that, if more than $n/2$ of the chips are good, then $\lfloor n/2 \rfloor$ pairwise tests using the device are sufficient to reduce the problem to one of nearly half the size.** Label the chips c_i, $i = 1, \ldots, n$. We suppose we have an algorithm TEST(i, j) which returns a two-entry array T such that $T[1]$ is the result (G or B) of using c_j to test c_i, and $T[2]$ is the result of using c_i to test c_j. If there are no more than two chips, then the fact that more than half of them are good means that they are all good, so we can pick (say) the first one. Otherwise, we consider choosing pairs of chips, testing them against one another, and discarding them unless each reports the other is good. This leaves a smaller pile of chips, which we test recursively. In other words, we have the algorithm

```
FIND-GOOD-CHIP(A)
 1 ◁ The input array A must be non-empty, and its entries must
     index more good chips than bad.
 2 if length[A] ≤ 2
 3     return A[1]
 4 endif
 5 m ← length[A]
 6 for j = 1 to ⌊length[A]/2⌋
 7     T ← TEST(A[2j − 1], A[2j])
 8     if T ≠ ⟨G, G⟩
 9         m ← m − 2
10         A[2j − 1] ← B
11         A[2j] ← B
12     endif
13 endfor
```

```
14 m' ← 2⌊m/2⌋
15 if ⌊m/2⌋ is odd
16       m ← m − 1
17       A[n] ← B
18 endif
19 for j = 1 to ⌊length[A]/2⌋
20       if A[2j − 1] ≠ B
21             m ← m − 1
22             A[2j] ← B
23       endif
24 endfor
25 create array A'[1 . . . m]
26 A' ← READNB(A, m)
27 return FIND-GOOD-CHIP(A')
```

We start this procedure with input $\langle 1, \ldots, n \rangle$, meaning that it tests all chips. The **if** test on line 15 is unexpected but important (consider running this algorithm on three chips, of which the first two are good and the last is bad, without it). READNB(A, m) is an algorithm which returns a new array containing the m entries of A which are not marked B. (If desired, it may be improved to read only the entries of A with odd indices, since the entries with even indices are all marked B.) These are the indices of the chips which we have not yet discarded. It is easy to see that this can be done in time $\Theta(n)$, where $n = \text{length}[A]$, but we omit the details.

We claim that "More than half of the chips not marked B are good" is a loop invariant for the **for** loop on line 6. Indeed, we are given that this is true upon initialisation. If the statement was true the last time j incremented, then either TEST returned $\langle G, G \rangle$ and no changes were made, or TEST returned something else. In this case, not both of the chips tested can be good. Thus the number of unmarked chips decreased by 2, while the number of unmarked good chips decreased by at most 1. This is the maintenance step, and, at termination, we simply observe that it is still true that more than half of the chips not marked B are good.

Suppose that, at line 15, exactly g of the chips indexed by the unmarked entries of $A[1 \ldots 2\lfloor n/2 \rfloor]$ are good, and exactly b of the chips indexed by the unmarked entries of $A[1 \ldots 2\lfloor n/2 \rfloor]$ are bad. (Notice that we are ignoring the last entry $A[n]$ if n is odd.) Then $g + b = m'$ (defined on line 14) and

(i) $g > b$ or

(ii) $g = b$, n is odd and the chip indexed by $A[n]$ is good.

For any j between 1 and $\lfloor n/2 \rfloor$, if $A[2j-1]$ is unmarked, then the chips indexed by $A[2j-1]$ and $A[2j]$ are both good or both bad. Thus, when the **for** loop on line 19 has finished, exactly $g/2$ of the chips indexed by the still-unmarked entries of $A[1 \ldots 2\lfloor n/2 \rfloor]$ are good, and exactly $b/2$ of the chips indexed by the still-unmarked entries of $A[1 \ldots 2\lfloor n/2 \rfloor]$ are bad. Moreover, since the **for** loop marks one of the remaining

unmarked entries bad at each iteration, after it has completed there are $m - \lfloor m/2 \rfloor = \lceil m/2 \rceil \leq \lceil n/2 \rceil$ unmarked entries remaining. Then

(i) $g/2 > b/2$ or

(ii) $g/2 = b/2$, n is odd and the chip indexed by $A[n]$ is good.

In the latter case, a total of $(g/2)+1$ chips indexed by unmarked entries of A are good, whereas a total of $b/2$ chips indexed by unmarked entries of A are bad. Since $(g/2) + 1 = (b/2) + 1 > b/2$, we still have more good chips than bad. In the former case, either $g/2 = (b/2) + 1$ or $g/2 > (b/2) + 1$.

Suppose $g/2 = (b/2) + 1$. Then $\lfloor m/2 \rfloor = m'/2 = (g/2) + (b/2) = 2(b/2) + 1$ is odd, so $A[n]$ is marked B. Thus a total of $g/2$ chips indexed by unmarked entries of A are good, whereas a total of $b/2$ chips indexed by unmarked entries of A are bad. Since $g/2 > b/2$, once again we have more good chips than bad. Suppose, on the other hand, that $g/2 > (b/2) + 1$. Then at least $g/2$ chips indexed by unmarked entries of A are good, whereas at most $(b/2) + 1$ chips indexed by unmarked entries of A are bad. Thus in this case too we have more good chips than bad.

A particular (if unexpected) consequence is that we have *not* marked all the entries of A. Indeed, if we had marked all the entries of A, then exactly 0 of the chips indexed by unmarked entries of A are good and exactly 0 of the chips indexed by unmarked entries of A are bad, a contradiction of the fact that there should be more of the former than the latter.

Thus the recursive call to FIND-GOOD-CHIP on line 27 accepts as input a non-empty array more of whose entries index good chips than index bad ones, which is precisely the legal input.

(c) **By setting up and solving a recurrence describing the number of tests required, show that, if more than $n/2$ of the chips are good, then the good chips can be identified with $\Theta(n)$ pairwise tests.** Consider the algorithm FIND-GOOD-CHIP of (6b). Denote its worst-case running time on an input of size n by $T(n)$. Then it is easy to see that

$$T(n) = \max\{T(k) : k = 1, \ldots, \lceil n/2 \rceil\} + f(n),$$

where $f(n) = \Theta(n)$. We cannot say that $T(n) = T(\lceil n/2 \rceil) + f(n)$, since all we know is that the recursive call in FIND-GOOD-CHIP takes an input of size *no more than* $\lceil n/2 \rceil$. The master theorem is thus not applicable here. (Even if it were applicable, we would have to verify that $f(n)$ satisfied the condition $af(n/b) \leq \gamma f(n)$ for some $\gamma < 1$. Since this is difficult to do without knowing exactly what form the function f takes, it would be easier to do our usual trick of over- and under-estimating T with asymptotically identical estimates.) In particular, there are a natural number n_0 and a constant $c > 0$ such that $f(n) \leq cn$ for $n \leq n_0$. By making c larger, if necessary, we may ensure that $f(n) \leq cn$ for all n and $c \geq \frac{T(1)}{3}$. We claim that $T(n) \leq 3cn$ for all n.

Since $c \geq 1$, the statement is true for $n = 1$. Now suppose that $n_1 > 1$ and we have proven the statement for all $n < n_1$. Then $\lceil n_1/2 \rceil < n_1$

and $\lceil n_1/2 \rceil \le (2/3)n_1$, so

$$\begin{aligned} T(n_1) &= \max\{T(k) : k = 1, \ldots, \lceil n_1/2 \rceil\} + f(n_1) \\ &\le \max\{3ck : k = 1, \ldots, \lceil n_1/2 \rceil\} + cn_1 \\ &= 3c\lceil n_1/2 \rceil + cn_1 \le 2cn_1 + cn_1 = 3cn_1. \end{aligned}$$

Thus $T(n) = O(n)$. Since also $T(n) = \Omega(f(n)) = \Omega(n)$, we have $T(n) = \Theta(n)$. By the master theorem (with $a = 1$ and $b = 2$), since $n = \Omega(n^{\log_2 1})$ and $1(n/2) \le (1/2)n$, we have that $T(n) = \Theta(n)$. Once we know a single good chip, we can use it to definitively identify every other chip as good or bad. The algorithm

FIND-GOOD-CHIPS(n)

```
 1  g ← 0
 2  i ← FIND-GOOD-CHIP(⟨1, ..., n⟩)
 3  for j = 1 to n
 4      T ← TEST(i, j)
 5      if T[2] = G ◁ T[2] is the result of using the known-good
                          chip c_i to test chip c_j
 6          g ← g + 1
 7          A[i] ← G
 8      endif
 9  endfor
10  return READG(A, g)
```

identifies the (indices of the) good chips. Here, READG(A, g) is an algorithm which returns a new array containing the g entries of A which are marked G. As before, we omit the details. Since the worst-case running time of this algorithm is $T(n) + \Theta(n) = \Theta(n)$, we are finished.