CS 170 Algorithms Fall 2014 David Wagner Sol 2

1. (10 pts.) Big-Theta running time

In class, we saw big-O notation: e.g., $5n^2 = O(n^3)$. It is also useful to know $\Theta(\cdot)$ notation, so this question will give you practice with that. Roughly speaking, big-O notation is like an asymptotic version of $\leq$, where we don't care about constants. Intuitively, $\Theta(\cdot)$ notation is like an asymptotic version of $=$, where we don't care about constants. More precisely, we write $f(n) = \Theta(g(n))$ if (a) $f(n) = O(g(n))$ and (b) $g(n) = O(f(n))$. (See Chapter 0.3 of the textbook for more details.)

With that in mind, answer the following questions with Yes or No. You do not need to provide any explanation or justification.

(a) Is $5n^2 = \Theta(n^3)$?

Solution: No. While it is true that $5n^2 = O(n^3)$, the reverse does not hold. In other words, it is not the case that $n^3 = O(5n^2)$, because for any constant c, as soon as n becomes larger than $5c$, we have

$$n^3 = n \times n^2 > c \times 5n^2.$$

(b) Is $5n^2 = \Theta(n^2)$?

Solution: Yes. We have $5n^2 \leq 5 \times n^2$ which shows that $5n^2 = O(n^2)$ and also $n^2 \leq 5n^2$ which shows that $n^2 = O(5n^2)$.

(c) Is $5n^2 + 7n + 1 = \Theta(n^2)$?

Solution: Yes. Polynomials have the same order of growth as their leading term. In this case, for $n \geq 1$ we have

$$5n^2 + 7n + 1 \leq 5n^2 + 7n^2 + n^2 = 13n^2$$

which shows that $5n^2 + 7n + 1 = O(n^2)$. For the reverse side note simply that $n^2 \leq 5n^2 \leq 5n^2 + 7n + 1$, which means that $n^2 = O(5n^2 + 7n + 1)$.

(d) Is $n \lg n = \Theta(n^2)$?

Solution: No. It is true that $n \lg n = O(n^2)$, but the reverse does not hold. To see why, note simply that given any arbitrarily large constant c, from some point onwards $n > c \lg n$. This means that $n^2 > cn \lg n$. And this is contrary to n^2 being in $O(n \lg n)$.

There are several ways to prove that $n > c \lg n$ after some point (i.e., that $n > c \lg n$ for all $n \geq n_0$). One method is to appeal to the fact that exponentials grow faster than polynomials. If one rewrites everything in terms of a new variable $m = \lg n$, then $\lg n$ becomes m, a polynomial, and n becomes 2^m, an exponential.

Alternatively, one can prove this directly. For $n > \frac{2c}{\ln 2}$, the derivative of the function $c \lg n$ becomes smaller than $1/2$. This is because the derivative of $c \lg n$ is simply $\frac{c}{n \ln 2}$. Since the derivative of $c \lg n$ is smaller than $1/2$ when n is large, it is growing slower than a line with slope $1/2$, and thus slower than the function n (which has slope 1). Thus eventually it will be strictly smaller than the function n, and stay smaller for all larger n.

2. (16 pts.) Practice with running time analysis

Consider the following two algorithms:

Algorithm F(n):
1. For $i := 0, 1, \ldots, n-1$:
2. (do something)

Algorithm G(n):
1. For $i := 0, 1, \ldots, \lceil \lg n \rceil - 1$:
2. (do something)

(a) Suppose that step 2 takes i^2 steps of computation in the ith iteration. What is the total running time of algorithm F, as a function of n, using $\Theta(\cdot)$ notation? Show your calculation.

Solution:

We need to show a matching upper and lower bound. Since each step takes time i^2 and we have $i \leq n$, each step takes at most time n^2. There are n steps; therefore the easy upper bound is $n \times n^2 = n^3$. In other words, the running time is $O(n^3)$.

For the lower bound note that for $i \geq n/2$, each step takes at least time $i^2 \geq (n/2)^2$. There are $n/2$ such values of i. Therefore we get $(n/2) \times (n/2)^2 = n^3/8$ as a lower bound.

Since the upper and lower bounds are both $\Theta(n^3)$ we are done.

Alternative solution: the running time is $0^2 + 1^2 + 2^2 + 3^2 + \cdots + (n-1)^2$. This sums to $(2n^3 - 3n^2 + n)/6$, which is $\Theta(n^3)$.

(b) Suppose that step 2 takes $n - 2i$ steps of computation in the ith iteration. (except that when $2i \geq n$, it takes 1 step of computation). What is the total running time of algorithm F, as a function of n, using $\Theta(\cdot)$ notation? Show your calculation.

Solution:

Again the easy upper bound: each step takes at most n time, and there are n steps. Therefore the total time is upper bounded by $n \times n = n^2$. In other words, the running time is $O(n^2)$.

Now for the lower bound, note that for $i < n/4$, each step takes time $n - 2i \geq n - n/2 = n/2$. So the first $n/4$ steps take time at least $n/2$. Therefore we get $(n/4) \times (n/2) = n^2/8$ as a lower bound.

Again the upper and lower bounds match in terms of order of growth. Therefore we get $\Theta(n^2)$ as the answer.

(c) Suppose that step 2 takes $n/2^i$ steps of computation in the ith iteration. What is the total running time of algorithm G, as a function of n, using $\Theta(\cdot)$ notation? Show your calculation.

Solution:

Here the total running time is

$$T(n) = n + n/2 + n/4 + \cdots + n/2^i + \cdots + n/2^{\lceil \lg n \rceil},$$

which is a geometric sum. Geometric series where each term is a constant times the previous one (and where the constant is not 1) always always have the same order of growth as the largest term. To see this here, note that the whole sum is bounded from below by the first term, i.e., n.

For an upper bound, note that $T(n)$ can be upper-bounded by an infinite sum:

$$T(n) \leq n + n/2 + n/4 + \cdots = n \sum_{i=0}^{\infty} 2^{-i} = 2n$$

which shows that the upper bound is at most twice the lower bound. So the geometric sum must be in $\Theta(n)$. This is clearly $\geq n$.

(d) Suppose that step 2 takes $n/(i+1)$ steps of computation in the ith iteration. Prove that the total running time of algorithm F is $O(n \log n)$.

Hint: Look up the harmonic series online. Or, upper-bound each term n/i by $n/2^k$ for some appropriately chosen k (possibly different for each term).

Solution:

The total running time here is

$$T(n) = n/1 + n/2 + n/3 + \cdots + n/n.$$

When we factor out the term n, we get

$$T(n)/n = 1 + 1/2 + 1/3 + \cdots + 1/n,$$

which is simply called the nth harmonic number and is sometimes written as H_n. Thus $T(n) = nH_n$.

The harmonic numbers are very closely related to the function $\ln n$. The connection stems from the fact that the derivative of $\ln x$ is $1/x$, which means that $\ln n = \int_1^n \frac{1}{x} dx$. Harmonic numbers are similar to the integral, except that instead of an integral, they are a discrete sum. In fact since $1/x$ is a decreasing function, for each i we have $\int_{i-1}^{i} \frac{1}{x} dx \geq \frac{1}{i}$. Summing these for $i = 2, 3, \ldots, n$, we get

$$\sum_{i=2}^{n} \int_{i-1}^{i} \frac{1}{x} dx \geq 1/2 + 1/3 + 1/4 + \cdots + 1/n.$$

But the left hand side is simply $\int_1^n \frac{1}{x} dx$ or simply $\ln n$. Therefore we just showed that

$$1/2 + 1/3 + \cdots + 1/n \leq \ln n.$$

Therefore the total running time is upper-bounded by $n(1 + \ln n) = O(n \lg n)$.

Another way to see this without appealing to integrals is the following: Notice that

$$\begin{aligned} H_n &= (1) + (1/2 + 1/3) + (1/4 + 1/5 + 1/6 + 1/7) + \cdots + 1/n \\ &\leq (1) + (1/2 + 1/2) + (1/4 + 1/4 + 1/4 + 1/4) + \cdots + 1/2^{\lfloor \lg n \rfloor}. \end{aligned}$$

But now the sum of each group of terms within the parenthesis is exactly 1. How many groups are there? Well, there are at most $1 + \lfloor \lg n \rfloor$ groups. Therefore,

$$H_n \leq 1 + \lfloor \lg n \rfloor \leq 1 + \lg n = O(\lg n).$$

This shows that $T(n) = O(n \lg n)$.

The latter argument can be written in a more formal way like this: for $i = 2^k, 2^k + 1, \ldots, 2^{k+1} - 1$, we can upper-bound the value $1/i$ by $1/2^k$. There are 2^k such values being upper-bounded, so

$$\sum_{i=2^k}^{2^{k+1}-1} \frac{1}{i} \leq \frac{2^k}{2^k} = 1.$$

Now we can break the sum $1 + 1/2 + 1/3 + \cdots + 1/n$ into roughly $\lg n$ groups. For each k from 0 to $\lceil \lg n \rceil - 1$, the terms $1/i$ for $i = 2^k, \ldots, 2^{k+1} - 1$ are grouped together and as we just showed, these terms sum up to at most 1. Since there are roughly $\lg n$ groups we get $\lg n$ as the final upper bound. Bringing back the factor n, we get the running time upper bound of $O(n \lg n)$.

3. (15 pts.) Out of sorts

Consider the following sorting algorithm:

Algorithm $S(A[0..n-1])$:
1. If $n = 2$ and $A[0] > A[1]$, swap $A[0]$ and $A[1]$.
2. If $n \geq 3$:
3. Let $k := \lceil 2n/3 \rceil$.
4. Call $S(A[0..k-1])$. ("Sort the first two-thirds.")
5. Call $S(A[n-k..n-1])$. ("Sort the last two-thirds.")
6. Call $S(A[0..k-1])$. ("Sort the first two-thirds.")

It turns out that this algorithm will correctly sort the input array. Let's analyze its running time.

(a) Let $T(n) =$ the number of comparisons between array elements when executing S on an array of size n. Write a recurrence relation for $T(n)$.

Solution:

Clearly each call to the function for $n > 2$ peforms no comparisons and does three recursive calls on arrays of size $k = \lceil 2n/3 \rceil$. So if $T(n)$ is the number of comparisons for an array of size n, we get

$$T(n) = 3T(\lceil 2n/3 \rceil)$$

for $n > 2$ and $T(n) = \Theta(1)$ for $n \leq 2$.

(b) Solve the recurrence relation you wrote down in part (a). Express your solution using $\Theta(\cdot)$ notation.
Hint: You should be able to write your answer in the form $T(n) = \Theta(n^c)$, for some constant c.

Solution:

We can either appeal to the master theorem or directly solve this. For demonstration purposes, let's solve it directly using the recursion tree.

Let us look at the recursion tree. For each function call place a node of the tree who has exactly three children (the recursive calls), except the terminal nodes where $n \leq 2$. Then we need to count the number of leaves in this tree. The number of nodes at the root of the tree is 1, and from one level to the next this number gets multiplied by 3. So the total number of leaves is 3^h where h is the height of the tree.

The number h itself is $\log_{3/2} n +$ constant. This is intuitively true because at each level n is getting roughly multiplied by $2/3$ until it reaches 2. Formally we always go from a length of n to a length of at least $2n/3$ (because of the ceiling). So the number h is at least $\log_{3/2} n -$ constant. However if we look at what happens to $n-3$, we see that $n-3$ changes to $\lceil 2n/3 \rceil - 3$ which is at most $2n/3 + 1 - 3 = 2n/3 - 2 = \frac{2}{3}(n-3)$. So that number gets multiplied by at most $2/3$ each time. So it takes at most $\log_{3/2}(n-3) +$ constant many steps before it reaches a small constant number.

So h, within a constant additive term, is just $\log_{3/2} n$. So the answer is $\Theta(3^{\log_{3/2} n})$. Note that constant additive terms in the exponent translate to constant factors which we ignore. We can rewrite this as $\Theta(n^{\log_{3/2} 3})$.

Alternatively, you could appeal to the master theorem. Some caution is needed because the master theorem talks about recurrence relations of the form $T(n) = 3T(2n/3) + \Theta(n^d)$, but the $\Theta(n^d)$ term does not appear in our recurrence from part (a). We can upper-bound $T(n)$ by $U(n)$, where $U(n)$ is defined by the recurrence relation $U(n) = 3U(2n/3) + \Theta(1) = 3U(2n/3) + \Theta(n^0)$. Then the master theorem tells us that $U(n) = \Theta(n^{\log_{3/2} 3})$. It follows that $T(n) = O(n^{\log_{3/2} 3})$.

(To be precise, we also need to show a corresponding lower bound before we can conclude that $T(n) = \Theta(n^{\log_{3/2} 3})$. One way to show such a lower bound is by looking at the recursion tree, as above. Another way is to use guess-and-check and prove $T(n) \geq n^{\log_{3/2} 3}$ by strong induction on n. However we won't worry about this, for this homework.)

(c) Based on your answer to part (b), would you expect S to be faster than, slower than, or about the same speed as insertion sort?

Solution:

It is slower than insertion sort. Insertion sort takes time $O(n^2)$, whereas this algorithm takes time $\Theta(n^{\log_{3/2} 3})$. Now note that $\log_{3/2} 3 = 2.7095\ldots$, which is significantly larger than 2.

4. (20 pts.) Merge asymptotics

After grading the exams, the CS 170 staff want to sort all of the exams according to the student ID numbers, so that they can easily retrieve one if the need arises. There are n exams and k GSIs. Each GSI gathers n/k exams and sorts them into a pile. But then the GSIs leave and now David wants to merge all of these piles together. Merging two piles, one with a exams and the other with b exams, takes $\Theta(a+b)$ work. David has two options:

1. Merge pile 1 with pile 2, then merge the result with pile 3, and then merge the result with pile 4, and so on.
2. Split the piles into two roughly equal halves, recursively merge each half, and then use the merge procedure to combine the two halves.

(a) How much work does David do if he uses method 1? Write a recurrence relation, then solve it. Express your final answer using $\Theta(\cdot)$ notation.

Solution:

Let $T(i)$ be the total amount of work done in the first i steps of the process, i.e., when merging i piles together. The resulting large pile before the ith pile gets into the mix has $(i-1)(n/k)$ exams in it. This means that the next merge takes time $\Theta((i-1)(n/k)+(n/k)) = \Theta(i(n/k))$ time. So the recurrence is

$$T(i) = T(i-1) + \Theta(i(n/k)).$$

One can see that

$$T(k) = \Theta((n/k) + 2(n/k) + 3(n/k) + \cdots + k(n/k)).$$

But we know that $1+2+\cdots+k = \Theta(k^2)$ (from previous questions). Therefore the running time is

$$T(k) = \Theta(k^2 \times (n/k)) = \Theta(nk).$$

Caution: Once you have the recurrence

$$T(i) = T(i-1) + \Theta(i(n/k)),$$

you might be tempted to treat n/k as a constant and write this in the form

$$T(i) = T(i-1) + \Theta(i)$$

so you can apply the master theorem. The master theorem will tell you that $T(i) = \Theta(i^2)$ and thus $T(k) = k^2$. However, this is not a valid approach: here we care about how the total running time depends on n and k, so we cannot treat n or k or n/k as a constant. As you can see, this approach leads you to the wrong answer.

Caution: Be careful! It was tempting to try to write a recurrence relation for $T(n)$ or $T(k)$, but this doesn't work.

If you try to write a recurrence for $T(n)$, you get stuck: n is not changing as we proceed through the iterative process, so there is no clear relationship between $T(n)$ and $T(\text{something smaller})$. Similarly, if

you try to write a recurrence relation for $T(k)$, you run into the problem that the size of the unmerged piles (k) is not changing as we proceed through the iterative process. Consequently, $T(k) = T(k-1) + \Theta(n/k)$ is not quite correct, because the size of the individual piles does not increase from n/k to $n/(k-1)$ after one step; the sizes of each individual unmerged pile remains the same. Similarly, trying to write a recurrence relation for $T(n,k)$ runs into similar problems.

You might be tempted to write something like $T(n) = T(n-k) + \Theta(n)$, where $T(n-k)$ represents the work while merging the first $k-2$ piles and $\Theta(n)$ is the amount of work in the final merge—but this isn't quite right either, as merging $k-1$ piles of n/k exams isn't the same as merging k piles of $(n-k)/k$ exams.

In this case, the most helpful thing to do is to introduce a new parameter. Since have an iterative process which iterates from 1 to $k-1$, it is natural to introduce a variable i that counts how far we have gotten through the iterative process, and keep track of the total amount of work done after we have gotten through some number of steps of the iterative process. In this way, we expressed our recurrence as $T(i)$, not $T(n)$ or $T(k)$. This is not unusual. When analyzing an algorithm, don't be afraid to introduce a new parameter and use that in your recurrence if that's what is needed.

(b) How much work does David do if he uses method 2? Write a recurrence relation, then solve it. Express your final answer using $\Theta(\cdot)$ notation.

Solution:

If $T(n,k)$ shows the amount of work for n exams and k GSIs, then

$$T(n,k) = 2T(n/2,k/2) + \Theta(n) \qquad \text{for } k > 1,$$

since we recursively solve the problem for two halves, each half having half as many exams and also half as many piles. Then the resulting two stacks are merged in $\Theta(n)$ time.

To solve this recurrence relation, look at the recursion tree. At each level of the recursion the total amount of work is $\Theta(n)$ (because the amount of work can be accounted by the number of exams participating in merges and it is intuitively obvious that each exam participates in exactly one merge at each level of the tree). So the total running time is $\Theta(nh)$ where h is the height of the tree. But the height of the tree is simply $\lg k$, because k gets divided by 2 until it reaches 1. So the total running time is $\Theta(n \lg k)$.

(c) Which method is better?

Solution:

The second method is better because $n \lg k = O(nk)$ and for large enough values of k, nk is much larger than $n \lg k$, because k grows much faster than $\lg k$.

5. (19 pts.) Plurality finding: divide-and-conquer

Definition. An array $A[0..n-1]$ is said to have a $1/3$-plurality element if some value v appears $> n/3$ times in the array; each such value v is called a $1/3$-plurality element.

Design a divide-and-conquer algorithm that, given $A[0..n-1]$, outputs "Yes" and a $1/3$-plurality element if $A[0..n-1]$ has a $1/3$-plurality element, or "No" if $A[0..n-1]$ does not have a $1/3$-plurality element. (If $A[0..n-1]$ has multiple $1/3$-plurality elements, the algorithm can return any one of them.) Your algorithm should have $O(n \lg n)$ running time.

However, there is a special restriction. The only thing you are allowed to do with array elements is compare whether they are identical (test whether $A[i] = A[j]$ for some i, j of your choice). The array elements are not from an ordered domain, so you cannot compare them using $<$ and $>$, and you cannot hash the array elements.

Note: in this class, whenever we ask you to design an algorithm on a homework set, your write-up should include all of the following parts:

- *Explanation:* Explain the main idea behind your algorithm. Conciseness is good: try to explain in at most a few sentences. A good goal is that if another CS 170 student were to read this part, they'd say "oh! now I see how to solve this problem"—you should be explaining the key insight that lets them solve the problem. Don't just repeat what the pseudocode says.
- *Algorithm:* Give the pseudocode of your algorithm.
- *Running time:* State the running time of your algorithm. Then, justify this claim—show the calculation.
- *Proof of correctness:* Prove that your algorithm is correct, i.e., always produces a correct result. State and prove any invariants that help demonstrate its correctness.

Label each of these main parts in your answer, to make it easy for readers to find them.

Solution:

Explanation: Let us design a divide-and-conquer method that finds *all* of the $1/3$-plurality elements. There can be at most 2 such elements since each one has to occupy more than one third of the array. Then the idea is that if we break the array into two halves, each $1/3$-plurality element must be a $1/3$-plurality in at least one of the halves. So we can recursively find all $1/3$-plurality elements in each of the halves, and in the end we have at most 4 candidates. We can check whether they are truly a $1/3$-plurality by just counting how many times they occur in the full array, in $O(n)$ time.

Algorithm: Here is the complete algorithm.

Algorithm FindPlurality($A[0..n-1]$):
1. If $n = 1$ return $\{A[0]\}$.
2. If $n \geq 2$:
3. Let $k := \lfloor n/2 \rfloor$.
4. Let $C := \emptyset$. ("This will be the set of candidates.")
5. Let $M := \emptyset$. ("This will be the set of $1/3$-plurality elements.")
6. Call FindPlurality($A[0..k-1]$) and add the results to C.
7. Call FindPlurality($A[k..n-1]$) and add the results to C.
8. For each x in C:
9. Count the number of occurences of x in $A[0..n-1]$.
10. If this number is greater than $n/3$, add x to M.
11. Return M.

Running time: The running time is $O(n \lg n)$. The total number of items returned by each call can be at most 2, since there can be at most two $1/3$-plurality elements in any array. Therefore C can have size at most 4, which means that line 9 gets executed at most 4 times, each time taking $O(n)$ time. Now if $T(n)$ is the running time of the algorithm for an array of size n, then we just proved that

$$T(n) = 2T(n/2) + O(n).$$

By appealing to the master theorem, we can see that this results in a running time of $O(n \lg n)$.

Proof of correctness: We will be inductively proving that the algorithm returns all $1/3$-plurality elements in the given array. Let $P(n)$ denote the assertion that, for all arrays A of length n, FindPlurality($A[0..n-1]$) correctly returns all $1/3$-plurality elements of A. We want to prove $\forall n \in \mathbb{N} \,.\, P(n)$. We will prove it by strong induction on n.

Base case: the base case of the induction is when $n = 1$, and in that case the single element in the array is a $1/3$-plurality which is returned by the algorithm (line 1). Thus, we have proven $P(1)$.

Inductive step: We need to prove $P(1) \wedge \cdots \wedge P(n-1) \implies P(n)$. Since we run over all elements of C and check whether they are in fact $1/3$-plurality elements, there is no risk that FindPlurality will return something that isn't a $1/3$-plurality element. We just need to show that each true $1/3$-plurality element x does get inserted into C. Consider one such element x. If x does not get added to C in line 6, it means that it was not a $1/3$-plurality in $A[0..k-1]$, so it appeared at most $k/3$ many times in that part (since $k < n$ and the inductive hypothesis promises that FindPlurality works correctly on an array of size k). Similarly if it does not get added in line 7, it means that it appeared at most $(n-k)/3$ many times in $A[k..n-1]$. So overall, the number of times it can appear in $A[0..n-1]$ is at most $k/3 + (n-k)/3 = n/3$. But this is in contradiction with the assumption that x was a $1/3$-plurality element. So every $1/3$-plurality element will be added to C in either line 6 or line 7, and thus will be returned by FindPlurality.

Alternatively, if you assumed that $n = 3 \times 2^k$, you could prove correctness by doing induction on k (rather than strong induction on n). the reasoning would be the same.

A subtle point: This was an example of a case where you needed to "solve a harder problem," because the natural divide-and-conquer algorithm doesn't work. The homework asked you to find an algorithm that returns at least one $1/3$-plurality element. However, this isn't enough to make the divide-and-conquer algorithm work.

Imagine you run the algorithm and it tells you one $1/3$-plurality element of $A[0..k-1]$, say x, and one $1/3$-plurality element of $A[k..n-1]$, the obvious thing to do is to check if either x or y is a $1/3$-plurality element for $A[0..n-1]$. But what if neither x nor y is a $1/3$-plurality element for $A[0..n-1]$? Can we conclude that $A[0..n-1]$ does not have any $1/3$-plurality element and return "No"? Unfortunately, no: that doesn't work.

For instance, suppose the input is $[1,1,2,2,3,4,4,2,2,5]$. When you recursively look for a $1/3$-plurality element of $[1,1,2,2,3]$, the algorithm might return the number 1. When you recursively look for a $1/3$-plurality element of $[4,4,2,2,5]$, the algorithm might return the number 4. Neither 1 nor 4 is a $1/3$-plurality element for the full array. However, it would be incorrect to conclude that there is no $1/3$-plurality element: 2 is a $1/3$-plurality element for the full array. So if you try the obvious algorithm, you'll discover that it doesn't actually work: you'll get stuck.

How do you get un-stuck? Solve a harder problem. Start by "making a wish." Think about what information you wish the algorithm would return you, when you run it on the subproblems (the left half and the right half of the array). In this case, if only it would return us a list of all $1/3$-plurality elements, we could deal with the tricky case above: in this way we'd quickly discover that 2 is a $1/3$-plurality element. So, change the problem: try to ask for an algorithm that returns all $1/3$-plurality elements... and then everything will work out.

A common confusion: I noticed some students who, when trying to design a divide-and-conquer algorithm, wanted to start with the base case of the recursion and work their way up the tree and figure out what will happen at each level. Unfortunately, this approach isn't so helpful. It's too hard to keep track of all of the levels in your head at once.

Instead, trust the magic of recursion. Imagine your fairy godmother has given you a magical black box that solves the problem on all arrays of size $\leq n/2$ (isn't she awesome?). Don't worry about how the black box works or what's inside it—just focus on how to use that magical black box to solve the problem on arrays of size n. Keep it simple: your task is just to ensure that your algorithm will be correct, as long as the magical black box works correctly (and you can be sure it will; your fairy godmother would never screw you over).

You might be wondering: but wait, where did your fairy godmother get the magical black box? I'll let you in on the secret: when she was young, *her* fairy godmother gave her a magical black box that works on

all arrays of size $\leq n/4$. And so on. Long ago, some fairy god-god-god-$\cdots$-mother constructed a box that worked on all arrays of size 1, and that got the whole thing started. Or, to put it in a way that avoids fairy tales: if you have an algorithm that works correctly on problems of size n if you give it a subroutine that works correctly on problems of size $n/2$, you can give it a reference to your own algorithm as the subroutine to use, and everything will work out. Thank you, recursion!

Comment about pseudocode: Notice that your pseudocode can be (and should be) at a high level of abstraction. For instance, in step 9, we didn't try to show how to implement the process of counting the number of occurrences of x in $A[0..n-1]$; we assume that any student in this class should be able to do that. So, try to write pseudocode that is readable and concise, and is well-specified enough that any CS 170 student could turn it into a full implementation. Don't worry about low-level coding details—a dozen lines of high-level pseudocode is often easier to understand than a page or two of Python code.

Comment: The divide-and-conquer algorithm is not the only way to solve this problem. In fact this problem can be solved in $O(n)$ time algorithm, using a generalization of the algorithm you saw in the previous homework. Wow!

The high-level idea is to do a single pass over the array, and keep a set S of at most two elements with associated counters. When we see a new element and S contains less than two elements, we add them to S with their counters set to 1. When we see a third element (i.e., if S already has two elements and we see a new element that is different from either of the values in S), we decrease the counters of the two elements in S (here we are tripling the elements off—like pairing them off in the last homework, but here we are grouping them into groups of three distinct elements). Any time a counter gets to zero, we throw the element out of S (though it might be re-introduced later). When we see an element that matches one of the ones in S, we increase its counter. At the end of the algorithm S contains two candidates for $1/3$-plurality. We check which ones are in fact $1/3$-plurality in $O(n)$ time.

6. (20 pts.) More divide-and-conquer

You are given a list of n intervals $[x_i, y_i]$, where x_i, y_i are integers with $x_i \leq y_i$. The interval $[x_i, y_i]$ represents the set of integers between x_i and y_i. For instance, the interval $[3,6]$ represents the set $\{3,4,5,6\}$. Define the *overlap* of two intervals I, I' to be $|I \cap I'|$, i.e., the cardinality of their intersection (the number of integers that are included in both intervals).

Devise a divide-and-conquer algorithm that, when given n intervals, finds and outputs the pair of intervals with the highest overlap. (You can resolve ties arbitrarily.)

It's easy to find an algorithm whose running time is $\Theta(n^2)$. Look for something better.

Hint: try splitting using the left endpoint of the intervals.

As always, include all of the major parts listed above (explanation, algorithm, running time, proof of correctness) in your answer.

Solution:

Explanation: As suggested we break the intervals into two roughly equal sized halves, by splitting them according to whether their left endpoints are below or above a threshold x. We recursively find the largest overlap on the left half, and on the right half. Then we search for the largest overlap between an interval of the left half and an interval of the right half. From the left half we only need to consider the interval whose right endpoint is the highest. So we find that interval (in linear time), and then check its overlap with all of the intervals on the right half, which itself takes linear time.

Algorithm: You can find the pseudocode here. Note that we first sort the intervals and then feed them to the recursive algorithm that does the main part of the algorithm. This is to save some time in finding the threshold that divides the intervals into the left and right halves.

Algorithm FindLargestOverlap($I[0..n-1]$):
1. Sort the intervals $I[0..n-1]$ according to their left endpoints.
2. Call $F(I[0..n-1])$ and return the result.

Algorithm $F(I[0..n-1])$:
1. If $n = 1$, return 0.
2. If $n \geq 2$:
3. Let $k := \lfloor n/2 \rfloor$.
4. Let $x :=$ left endpoint of $I[k]$.
5. Let $O := 0$. ("This will be the largest overlap.")
6. Let $O := \max(O, F(I[0..k-1]))$.
7. Let $O := \max(O, F(I[k..n-1]))$.
8. Find the interval with the largest right endpoint in $I[0..k-1]$ and call it J.
9. For each interval J' in $I[k..n-1]$:
10. Let $O := \max(O, \text{overlap}(J, J'))$.
11. Return O.

Running time: The running time of this algorithm is $O(n \lg n)$. Lines 8, 9, and 10 take $O(n)$ time, and we recursively call the function on two smaller instances, each of length roughly half. So letting $T(n)$ denote the time to run F on an array of length n, we get the recurrence relation

$$T(n) = 2T(n/2) + O(n).$$

Using the master theorem we know that this solves to $T(n) = O(n \lg n)$. The sorting we do at the beginning of the algorithm also takes time $O(n \lg n)$, so the total running time is $O(n \lg n) + O(n \lg n) = O(n \lg n)$.

Proof of correctness: We will prove that the algorithm returns the largest overlap between the input intervals, using strong induction on n.

Base case: for $n = 1$, there are no two intervals, and the largest overlap is therefore 0. Line 1 of the algorithm correctly handles this case.

Inductive step: The inductive hypothesis tells us that F works correctly on all arrays of length $< n$. Since $k < n$ and $n - k < n$, it follows that lines 6 and 7 find the largest overlap between intervals of the left half and right half respectively. We just need to show the rest of the function finds the largest possible overlap between an interval of the left half and an interval of the right half; we'll do that next.

Suppose $J_1 = [x_1, y_1]$ is any interval of the left half and $J_2 = [x_2, y_2]$ is any interval of the right half. Could this be a larger overlap than anything found by F? We'll prove that it cannot. Since J_2 is an interval of the right half, its left endpoint is at least x, i.e., $x_2 \geq x$. Therefore the intersection $J_1 \cap J_2$ is contained in $[x, \infty)$, so $J_1 \cap J_2 = [x, y_1] \cap J_2$. Let $J = [x_0, y_0]$ be the interval found in step 8. Similarly, we'll have $J \cap J_2 = [x, y_0] \cap J_2$. Now due to the way J was selected in step 8, the right endpoint of J is at least as large as the right endpoint of J_1, i.e., $y_0 \geq y_1$. This means that $[x, y_1] \subseteq [x, y_0]$. It follows that $[x, y_1] \cap J_2 \subseteq [x, y_0] \cap J_2$, i.e., $J_1 \cap J_2 \subseteq J \cap J_2$, i.e., $\text{overlap}(J_1, J_2) \leq \text{overlap}(J, J_2)$. Now given the way the loop in steps 9–10 works, we see that $O \geq \text{overlap}(J, J_2)$, so $O \geq \text{overlap}(J_1, J_2)$. Therefore J_1, J_2 cannot have higher overlap than what was returned by F. This means that F correctly finds the largest possible overlap between any pair of intervals.

Alternate proof of the last paragraph: Suppose J_1 is an interval of the left half and J_2 is an interval of the right half. Then the left endpoint of J_2 is at least x. Therefore their intersection lies in $[x, \infty)$. The left endpoint of J_1 is at most x, therefore it does not affect the size of the overlap. In other words, we can replace the left endpoint of J_1 with x and nothing changes. Now if we hypothetically assume all left endpoints of

the left intervals are x, it is obvious that the best we can do is find the one that has the highest right endpoint (regardless of the choice of J_2 we end up with the highest overlap). Line 9 finds exactly this interval, and then we check its overlap with all intervals of the right half. Therefore we must have considered at least one of the pairs from the left half and the right half that have the highest overlap.

Comment about pseudocode: Notice that we didn't bother spelling out how you find the interval whose right endpoint is maximal (step 8), how to compute the overlap of two intervals (step 10), or other low-level implementation details. Try to follow this practice in your own solutions, too: don't drown the reader in low-level details, write your pseudocode at a high level of abstraction so that it is as easy to read and understand as possible.