CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Homework #1 Solution

September 22, 2019

Instructor: Sepehr Assadi

Problem 1. Let us revisit the chip testing problem. Recall that in this problem, we are given n chips which may be *working* or *defective*. A working chip behaves as follows: if we connect it to another chip, the original chip will *correctly* output whether the new connected chip is working or is defective. However, if we connect a defective chip to another chip, it may output *any arbitrary answer*.

In the class, we saw that if *strictly* more than half the chips are working, then there is an algorithm that finds a working chip using $O(n)$ tests. In this homework, we examine what will happen if the number of working chips is *equal* to the defective ones.

(a) Prove that even when we only have a single working chip and a single defective chip (i.e., $n = 2$), there is *no* algorithm that can find the working chip in general. **(10 points)**

Solution. Suppose that the defective chip also always outputs 'defective' for the other (working) chip. In this case, whether the input is $[W, D]$ or $[D, W]$, whenever any algorithm tests the only pair in the input, the only answer is (defective, defective) (no matter how many times the algorithm tests these chips). As the answer of every *deterministic* algorithm is uniquely determined by what it receives as input, and the "actual" input to the algorithm is simply a sequence of (defective, defective) answers, the algorithm cannot distinguish between the two cases, namely, whatever chip it outputs can be made wrong on some particular input.

(b) Give an algorithm that for any *even* number of chips n, assuming that the number of working chips is equal to the defective ones, can output a *pair of chips* such that in this pair, one of the chips is working and the other one is defective (the algorithm does not need to identify which chip is working/defective in this pair – this is crucial by part (a)). **(15 points)**

Solution. *Algorithm.* We use the original algorithm for the chip testing problem in the class. We already proved (using the second part of the induction hypothesis) that if number of working chips is equal to defective ones, then the algorithm either returns a working chip, or it returns 'empty'. We use this to solve this problem as well. In particular, we first run that algorithm and then do as follows.

Suppose first that the algorithm returns a single chip. In that case, we go over all other chips and use our working chip to test them and stop the first time our chip declares the other chip defective. We then simply output this pair. Now suppose the algorithm returns 'empty'. Consider the last iteration of the algorithm (in the recursive calls) in which it grouped multiple chips into pairs and tested them against each other (and then discarded *all* of them because it ended up outputting 'empty'). In this case, we output any of these discarded pairs as answer.

Proof of Correctness: For the first case, the correctness follows from the fact that the chip returned by the original algorithm was working and thus the other found chip should be defective and because we will eventually find a defective chip in the process above (as number of defective chips is non-zero). The more interesting case is the second one. We argue that in this case, returning any of the pairs in the last step of recursion right before outputting 'empty' is a correct solution. This is because by our original analysis (in the original algorithm) # of working chips can never become smaller than # of defective chips; this means that in the last iteration before outputting empty, we had equal number of working chips as defective ones (if we had more working chips, by the guarantee of the algorithm, we

would obtain a working chip instead of 'empty'). On the other hand, as we proved before in the class, any pair discarded in the algorithm contains at least one defective chip and since # of working chips is equal to # of defective ones, each pair should also contain a working chip. Hence, returning any pair is correct in this case. This proves the correctness of the algorithm.

Efficiency analysis: The number of tests done in each case is $O(n)$ (for running the original algorithm and by its analysis) plus at most another $O(n)$ in the first case (for testing all other chips against the found working chip). Hence, the total number of tests is $O(n)$.

Problem 2. This question reviews asymptotic notation. You may assume the following inequalities for this question (and throughout the course): For any constant $c \geq 1$,

$$(\log n)^c = o(n) \quad , \quad c^n = o((c+1)^n) \quad , \quad (n/2)^{(n/2)} = o(n!) \quad , \quad n! = o(n^n). \tag{1}$$

(a) Rank the following functions based on their asymptotic value in the increasing order, i.e., list them as functions $f_1, f_2, f_3, \ldots, f_{16}$ such that $f_1 = O(f_2)$, $f_2 = O(f_3), \ldots, f_{15} = O(f_{16})$. Remember to write down your proof for each equation $f_i = O(f_{i+1})$ in the sequence above. **(15 points)**

$\log\log n$	$10^{10^{10}}$	$2^{\sqrt{\log n}}$	n^2
$n^{1/\log\log n}$	$2n$	10^n	$n^{\log\log n}$
2^n	$10n$	$n!$	$\sqrt{n}$
$n/\log n$	$\log(n^{100})$	$(\log n)^{100}$	n^n

Solution. The correct ordering is as follows:

$f_1 = 10^{10^{10}}$	$f_2 = \log\log n$	$f_3 = \log(n^{100})$	$f_4 = (\log n)^{100}$
$f_5 = 2^{\sqrt{\log n}}$	$f_6 = n^{1/\log\log n}$	$f_7 = \sqrt{n}$	$f_8 = n/\log n$
$f_9 = 10n$	$f_{10} = 2n$	$f_{11} = n^2$	$f_{12} = n^{\log\log n}$
$f_{13} = 2^n$	$f_{14} = 10^n$	$f_{15} = n!$	$f_{16} = n^n$.

We now prove each part.

- $10^{10^{10}} = O(\log\log n)$: $10^{10^{10}}$ is a constant (even though a gigantic one) but f_2 grows with n (even though very slowly), hence $\lim_{n\to\infty} \frac{10^{10^{10}}}{\log\log n} = 0$.
- $\log\log n = O(\log(n^{100}))$: Firstly, $\log(n^{100}) = 100\log n = \Theta(\log n)$. So we only need to prove $\log\log n = O(\log n)$. Let us do a change of variable by setting $m = \log n$: we thus need to prove $\log m = O(m)$ – this follows immediately from the first term in Eq (1).
- $\log(n^{100}) = O((\log n)^{100})$: since $\log(n^{100}) = \Theta(\log n)$, we only need to prove $\log n = O((\log n)^{100})$. By a change of variable $m = \log n$, it means we need to prove $m = O(m^{100})$ which we know is true as $\lim_{m\to\infty} \frac{m}{m^{100}} = 0$.
- $(\log n)^{100} = O(2^{\sqrt{\log n}})$: we do a change of variable $m = \sqrt{\log n}$, leaving us with proving $m^{200} = O(2^m)$ – this follows from the first term of Eq (1) (this is easiest to see by doing another change of variable and setting $m = \log k$).
- $2^{\sqrt{\log n}} = O(n^{1/\log\log n})$: this was already proved in the review notes for asymptotic notation.
- $n^{1/\log\log n} = O(\sqrt{n})$: We can write the first term as $2^{\log n/\log\log n}$ and the second term as $2^{\log n/2}$. Now since $\lim_{n\to\infty} \log n/\log\log n - \log n/2 = -\infty$, we have $\lim_{n\to\infty} \frac{2^{\log n/\log\log n}}{2^{\log n/2}} = 0$ (see the review notes for asymptotic notation).

- $\sqrt{n} = O(n/\log n)$: We write the limit: $\lim_{n\to\infty} \frac{\sqrt{n}}{n/\log n} = \lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = 0$; the last equality follows from the first term of Eq (1).
- $n/\log n = O(10n)$: We again write the limit: $\lim_{n\to\infty} \frac{n/\log n}{10n} = \lim_{n\to\infty} \frac{1}{10\log n} = 0$.
- $10n = O(2n)$: by ignoring the constants, both sides are $\Theta(n)$.
- $2n = O(n^2)$: by writing the limit: $\lim_{n\to\infty} \frac{2n}{n^2} = \lim_{n\to\infty} \frac{2}{n} = 0$.
- $n^2 = O(n^{\log\log n})$: We can write the first term as $2^{2\log n}$ and the second term as $2^{\log n \cdot \log\log n}$. Since $\lim_{n\to\infty} 2\log n - \log n \cdot \log\log n = -\infty$, we have $\lim_{n\to\infty} \frac{2^{2\log n}}{2^{\log n\cdot\log\log n}} = 0$ (see the review notes for asymptotic notation).
- $n^{\log\log n} = O(2^n)$: We apply the same argument as the previous case here since $n^{\log\log n} = 2^{\log n\cdot\log\log n}$ and $\lim_{n\to\infty} \log n \cdot \log\log n - n = -\infty$.
- $2^n = O(10^n)$: Follows immediately from the second term of Eq (1).
- $10^n = O(n!)$: We first claim $10^n = O((n/2)^{n/2})$. This is because $10^n = 2^{\log 10\cdot n}$ and $(n/2)^{(n/2)} = 2^{(n/2)\cdot\log n - n/2}$ and we can apply the previous type of arguments based on the following fact that $\lim_{n\to\infty} \log 10 \cdot n - ((n/2)\cdot\log n - n/2) = -\infty$. Now by the third part of Eq (1) and transitivity of O-notation, since $(n/2)^{(n/2)} = O(n!)$, we also have $10^n = O(n!)$.
- $n! = O(n^n)$: Follows immediately from the last term of Eq (1).

(b) Consider the following six different functions $f(n)$:

$$n! \qquad \log n \qquad 2^n \qquad n^2 \qquad 10 \qquad 2^{2^n}.$$

For each of these functions, determine which of the following statements is true and which one is false. Remember to write down your proof for each choice. **(10 points)**

- $f(n) = \Theta(f(n-1))$;
- $f(n) = \Theta(f(\frac{n}{2}))$;
- $f(n) = \Theta(f(\sqrt{n}))$;
- $f(n) = \Theta(f(\log n))$.

Solution. (a) For $f(n) = n!$, all the equations are false.

- $n! \neq \Theta((n-1)!)$: this is already proved in the review notes for asymptotic notation.
- $n! \neq \Theta((\frac{n}{2})!)$: we show that $n! \neq O((\frac{n}{2})!)$ or alternatively $(n/2)! = o(n!)$. This is because $\lim_{n\to\infty} \frac{(n/2)!}{n!} = \lim_{n\to\infty} \frac{1}{n\cdot(n-1)\cdots(n/2+1)} = 0$.
- $n! \neq \Theta((\sqrt{n})!)$: we show that $n! \neq O((\sqrt{n})!)$ or alternatively $(\sqrt{n})! = o(n!)$. This is because $\sqrt{n} < n$ and thus $\lim_{n\to\infty} \frac{(\sqrt{n})!}{n!} = \lim_{n\to\infty} \frac{1}{n\cdot(n-1)\cdots(\sqrt{n}+1)} = 0$.
- $n! \neq \Theta((\log n)!)$: we show that $n! \neq O((\log n)!)$ or alternatively $(\log n)! = o(n!)$. This is because $\log n < n$ and thus $\lim_{n\to\infty} \frac{(\log n)!}{n!} = \lim_{n\to\infty} \frac{1}{n\cdot(n-1)\cdots(\log n+1)} = 0$.

(b) For $f(n) = \log n$, the first three equations are correct while the last one is false.

- $\log n = \Theta(\log(n-1))$: for sufficiently large n, $1/2 \cdot \log n \leq \log(n-1) \leq \log n$.
- $\log n = \Theta(\log(n/2))$: $\log(n/2) = \log n - 1 \geq \log n/2$ for $n > 1$ so $\frac{1}{2}\cdot\log n \leq \log(n/2) \leq \log n$.
- $\log n = \Theta(\log\sqrt{n})$: $\log\sqrt{n} = 1/2 \cdot \log n$ and thus $1/2 \cdot \log n \leq \log(\sqrt{n}) \leq \log n$.
- $\log n \neq \Theta(\log(\log n))$: since $\log\log n = o(\log n)$ by the first term of Eq (1) (do a change of variable $m = \log n$ to see this more clearly).

(c) For $f(n) = 2^n$, the first equation is true while the last three are false.

- $2^n = \Theta(2^{n-1})$: $2^{n-1} = 2^n/2$ and thus $1/2 \cdot 2^n = 2^{n-1} \leq 2^n$ for sufficiently large n.

- $2^n \neq \Theta(2^{n/2})$: we show $2^{n/2} = o(2^n)$: $\lim_{n\to\infty} n/2 - n = -\infty$ and thus $\lim_{n\to\infty} \frac{2^{n/2}}{2^n} = 0$.
- $2^n \neq \Theta(2^{\sqrt{n}})$: we show that $2^{\sqrt{n}} = o(2^n)$ – this is by the same exact argument as above.
- $2^n \neq \Theta(2^{\log n})$: $2^{\log n} = n$ and thus we show that $n = o(2^n)$ – this is true by the first term of Eq (1) (do a change of variable $m = 2^n$ ($n = \log m$) to see this more clearly).

(d) For $f(n) = n^2$, the first two equations are true while the last two are false.

- $n^2 = \Theta((n-1)^2)$: $(n-1)^2 = n^2 - 2n + 1$ and the dominant term is n^2.
- $n^2 = \Theta((n/2)^2)$: $(n/2)^2 = n^2/4$ and we ignore the constant 4 in the Θ-notation.
- $n^2 \neq \Theta((\sqrt{n})^2)$: $(\sqrt{n})^2 = n$ and we have $n = o(n^2)$.
- $n^2 \neq \Theta((\log n)^2)$: $(\log n)^2 = o(n)$ by the first term of Eq (1) and $n = o(n^2)$ by the above.

(e) For $f(n) = 10$, all the equations are true. The proofs of all equations is also by seeing that both side of the equations are constants and we ignore constants in Θ-notation.

(f) For $f(n) = 2^{2^n}$, all the equations are false.

- $2^{2^n} \neq \Theta(2^{2^{n-1}})$: $2^{2^{n-1}} = 2^{2^n/2}$, $\lim_{n\to\infty} 2^n/2 - 2^n = -\infty \implies \lim_{n\to\infty} \frac{2^{2^n/2}}{2^{2^n}} = 0$ which means $2^{2^{n-1}} = o(2^{2^n})$.
- $2^{2^n} \neq \Theta(2^{2^{n/2}})$: We have $\lim_{n\to\infty} 2^{n/2} - 2^n = -\infty$ thus $\lim_{n\to\infty} \frac{2^{2^{n/2}}}{2^{2^n}} = 0$, so $2^{2^{n/2}} = o(2^{2^n})$.
- $2^{2^n} \neq \Theta(2^{2^{\sqrt{n}}})$: We have $\lim_{n\to\infty} 2^{\sqrt{n}} - 2^n = -\infty$ thus $\lim_{n\to\infty} \frac{2^{2^{\sqrt{n}}}}{2^{2^n}} = 0$, so $2^{2^{\sqrt{n}}} = o(2^{2^n})$.
- $2^{2^n} \neq \Theta(2^{2^{\log n}})$: $2^{2^{\log n}} = 2^n$ and $2^n = o(2^{2^n})$ by the first term of Eq (1) (do a change of variable $m = 2^{2^n}$ to see this more clearly).

Problem 3. In this problem, we analyze a *recursive* version of the *insertion sort* for sorting. The input as before is an array A of n integers. The algorithm is as follows (in the following, $A[i:j]$ refers to entries of the array with indices between i and j, for instance $A[2:4]$ is $A[2], A[3], A[4]$):

Recursive Insertion Sort:

1. If $n = 1$ return the same array.
2. Recursively sort $A[1 : n-1]$ using the same algorithm.
3. For $i = n-1$ down to 1 do:
   - If $A[i+1] < A[i]$, swap $A[i+1]$ and $A[i]$; otherwise break the for-loop.
4. Return the array A.

We now analyze this algorithm.

(a) Use *induction* to prove the correctness of the algorithm above. **(10 points)**

Solution. Our induction hypothesis is that for all n, the recursive insertion sort algorithm sorts any array of length n correctly. The base case for $n = 1$ holds by definition.

For the induction step, suppose the hypothesis is true for all integers $n \leq k$ and we prove it for $n = k+1$. By induction hypothesis, we know that $A[1 : n-1]$ is correctly sorted. We now show that the rest of the algorithm places $A[n]$ in its correct position in the sorted array, while *preserving* the order of previously sorted elements. Let m be a pointer that always points to the *original* value of $A[n]$ (meaning that if $A[n]$ gets swapped with some other element, m will point out to the previous index of this new element which now contains $A[n]$). The for-loop continuously swaps m by its immediate left neighbor in the array as long as this number is larger than $A[m]$; this means that when the for-loop breaks, all elements not swapped by m are smaller than $A[m]$ (the original $A[n]$) and all elements swapped are larger (since $A[1 : n-1]$ was sorted). Moreover, the *relative* order of elements in $A[1 : n-1]$ is the same. This implies that $A[1 : n]$ is now correctly sorted.

(b) Analyze the runtime of this algorithm. **(5 points)**

Solution. Define $T(n)$ as the worst-case running time of the recursive insertion sort on an array of size n. By construction, $T(1) = \Theta(1)$ and $T(n) \leq T(n-1) + O(n)$, because the algorithm involves a recursive call to the same problem on an array of size $n-1$ and then uses a for-loop with at most n iterations, each taking $O(1)$ time.

To compute $T(n)$, we write $T(n) \leq T(n-1) + c \cdot n$ (for some constant $c > 0$ by definition of $O(n)$). By substitution, we have,

$$T(n) \leq T(n-1) + c \cdot n \leq T(n-2) + c \cdot n + c \cdot (n-1) \leq \ldots \leq \sum_{i=1}^{n} c \cdot (n-i+1) = \sum_{i=1}^{n} c \cdot i = O(n^2).$$

Hence, the algorithm takes $O(n^2)$ time. (Although it is not necessary for the solution to this problem, we can also see that running this algorithm on an array which is ordered in the decreasing order, takes $\Omega(n^2)$ time – this implies that the worst-case runtime of the insertion sort algorithm is $\Theta(n^2)$.)

(c) Suppose we are *promised* that in the input array A, each element $A[i]$ is *at most* k indices away from its index in the sorted array. Prove that the algorithm above will take $O(nk)$ time now. **(10 points)**

Solution. Let $S(n)$ denote the worst-case running time of the recursive insertion sort, but this time on arrays in which each element $A[i]$ is at most k indices away from its correct position. We first prove that $S(n) \leq S(n-1) + O(k)$. This is because if in the array $A[1:n]$ all elements are at most k indices away from their correct position, then in the array $A[1:n-1]$ this is also true: so by recursing on this array, the runtime of this step is at most $S(n-1)$. We now prove that the for-loop also takes $O(k)$ time. This is because $A[n]$ is at most k step away from its position (sorting $A[1:n-1]$ recursively does *not* change this fact) and thus can only be swapped at most k times: this is because we already proved in part (a) that $A[n]$ will be placed in its correct position and thus the for-loop finishes after at most k iterations. As each iteration takes $O(1)$ time, the recurrence for $S(n)$ is correct.

We now solve this recurrence. By the same argument as in the previous part (we again replace $O(k)$ with $c \cdot k$ for some sufficiently large constant c):

$$S(n) \leq S(n-1) + c \cdot k \leq S(n-2) + c \cdot k + c \cdot k \leq \ldots \leq \sum_{i=1}^{n} c \cdot k = c \cdot n \cdot k = O(nk).$$

Hence, the runtime of insertion sort in this case is $O(nk)$.

Problem 4. You are hired to help the rebels fight the evil empire in Star Wars (!). The rebels have n space ships and each space ship i ($1 \leq i \leq n$) has a certain power p_i. Moreover, the empire has m bases where each base j ($1 \leq j \leq m$) has a defensive power d_j and gold g_j. You know that each space ship can attack every base with defensive power *strictly* smaller than the ship's own power and collect its golds.

The rebels need to know that, for each of their space ships, what is the maximum amount of gold this space ship can collect. Design an algorithm with running time $O((n+m) \cdot \log m)$ for this task. **(25 points)**

Example. The correct answer for the following input with $n = 4$ and $m = 5$ is $[8, 5, 7, 11]$:

power of space ships:	$p_1 = 5,$	$p_2 = 3,$	$p_3 = 4,$	$p_4 = 9;$	
defense power of bases:	$d_1 = 3,$	$d_2 = 8,$	$d_3 = 4,$	$d_4 = 0,$	$d_5 = 1;$
amount of gold in bases:	$g_1 = 2,$	$g_2 = 3,$	$g_3 = 1,$	$g_4 = 3,$	$g_5 = 2.$

Hint: Modify the binary search algorithm so that given a sorted array A and an integer t, find the *largest index* i in A such that $A[i] < t$. Then use this algorithm to find the "strongest" base each space ship can attack (you need to preprocess the bases and their golds accordingly).

Solution. Following the hint, we first design a binary search type algorithm for the problem of finding the largest index i in an array A such that $A[i] < t$ for a given t. We then use this to design the whole algorithm. We partition the solution into these two parts separately.

First Part: *Algorithm:* We call the modified binary search algorithm LOWER-BOUND.

- LOWER-BOUND algorithm: Given a sorted array $A[a : b]$ and an integer t;
  1. If $a > b$: return 'no such index'. If $a = b$: $A[a] < t$ return a, otherwise return 'no such index'. If $a + 1 = b$: if $A[b] < t$, return b, else if $A[a] < t$, return a, otherwise return 'no such index'.
  2. Let $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$.
  3. If $A[m] \geq t$, return LOWER-BOUND$(A[a : m-1], t)$.
  4. If $A[m] < t$, return LOWER-BOUND$(A[m : b], t)$.

Proof of Correctness: The proof is by induction for the hypothesis that LOWER-BOUND outputs the correct answer for every array of length n (i.e., $n = \min\{b - a + 1, 0\}$).

The induction base when $n = 0$, $n = 1$, $n = 2$ is true by the logic of the algorithm exactly as in the binary search and we do not repeat the argument here.

Suppose the algorithm works for all arrays of length $n \leq k$ and we prove it for $n = k + 1$. Case one is when $A[m] \geq t$. Since the array A is sorted, the correct index, say j, in this case cannot possibly be in $A[m : b]$: this is because $A[j] < t$ and since A is sorted and $A[m] \geq t$, j cannot be part of $A[m : b]$. As such, we only need to search for the correct index in $A[a : m - 1]$. By induction, since size of $A[a : m - 1]$ is smaller than $n = k + 1$, the algorithm finds the correct index in this array. The second case is when $A[m] < t$: in this case the answer can be either $j = m$, or some other index in $[m : b]$. However, since $A[m] < t$ already and we are looking for the *largest* index smaller than t, the correct index j cannot be part of $A[a : m - 1]$; hence, in this case also, since size of $A[m : b]$ is smaller than $n = k + 1$ (here we have to use the fact that $n > 2$ and that is the reason we proved the base case for $n = 2$ also), by induction, we will find the correct index. This proves the correctness of the algorithm.

Runtime Analysis: The runtime is asymptotically the same as that of binary search and is hence $O(\log n)$.

Second Part: *Algorithm:* We design a three step algorithm for the problem using LOWER-BOUND as a subroutine.

(1) We first pair up the defensive power and gold of each base in an array of pairs (d_j, g_j) for $1 \leq j \leq m$ and sort this array (by merge sort) using the defensive power of each chip as the 'key' (in other words, we simply compare the defensive power of each pair to decide which one should be considered 'larger' in the sorted array, breaking the ties arbitrarily). Let D denote the sorted array of defensive powers in this case and G denote the golds – we emphasize that for every $1 \leq j \leq m$, $D[j]$ and $G[j]$ correspond to the *same* base as we sorted both arrays together (although only used defensive power for comparison).

(2) We then create an array G' where for all $1 \leq j \leq m$, $G'[j] = \sum_{k=1}^{j} G[k]$, using the following algorithm: Let $G'[1] = G[1]$ and for $j = 2$ to m, let $G'[j] = G'[j-1] + G[j]$.

(3) Finally, we iterate over space ships one by one, for each one with power p_i, we run the LOWER-BOUND algorithm over the array D and with target $t = p_i$ (i.e., run it on $(D[1 : m], p_i)$). Let j^* be the returned answer. In this case, we output $G'[j^*]$ as the amount of gold that space ship i can collect. This finishes the description of the algorithm.

Proof of Correctness: The correctness of step (1) follows from correctness of merge sort. After this step, we can assume D is sorted correctly based on the defensive power.

For step (2), we prove for every $1 \leq j \leq m$, $G'[j] = \sum_{k=1}^{j} G[k]$ by induction: the base case for $G'[1] = G[1]$ holds by definition of the algorithm, and for the induction step we have $G'[j] = G'[j-1] + G[j] = (\sum_{k=1}^{j-1} G[k]) + G[j] = \sum_{k=1}^{j} G[k]$ as desired. So, step (2) computes the desired array G' correctly.

For step (3), for each space ship i, by the correctness of LOWER-BOUND, we find the *largest* index j^* such that $D[j^*] < p_i$. Since D is sorted, we also know that for all $k < j^*$, $D[k] < p_i$ and for all $k' > j^*$, $D[k'] \geq p_i$. So space ship i can attack the bases corresponding to $D[1 : j^*]$. Hence, the gold collected by this ship is $\sum_{k=1}^{j^*} G[k]$. By the correctness of step (2), this number is equal to $G'[j^*]$, which is returned by the algorithm. This concludes the correctness of the algorithm.

Runtime Analysis: The runtime of the algorithm is $O(m \log m)$ for sorting the array of defensive powers using merge sort in step (1), plus $O(m)$ time for computing the array G' since it is a simple m-iteration for-loop, plus $O(n \log m)$ for step (3) for running LOWER-BOUND (which takes $O(\log m)$ time) for every one of the n space ships. This adds up to $O((n+m) \log m)$ as desired.
