CS 170 Algorithms — Fall 2014 — David Wagner — Sol 1

1. (30 pts.) Getting started

The goal of this question is to ensure that you have read the course policies.

(a) Please read the course policies on the web page, especially the course policies on collaboration. If you have any questions, ask on Piazza about it. Once you have done this, please write "I understand the course policies." to get credit for this problem.

Solution: I understand the course policies.

(b) You've been working with a homework partner, and together you've managed to solve both problems on the homework. It's been a true joint effort—neither of you could have solved either problem on your own. Your homework partner suggests that you write up the solution for Problem 1, and he'll write up the solution for Problem 2, then you can swap write-ups to see how the other wrote up the solution, as a way to learn from each other. Is this allowed?

Solution: No. As the policies explain, you should never see or have in your possession anyone else's solution, and you should not share your solutions with anyone else (not even if they promise not to copy them). Instead, you must write up your solutions entirely on your own.

2. (35 pts.) Proof of correctness

Define $P(x) = a_n x^n + \cdots + a_2 x^2 + a_1 x + a_0$. Prove the correctness of the following algorithm for evaluating the polynomial P at the value $x = c$, i.e., for computing $P(c)$:

```
Algorithm HORNER($(a_0, a_1, \ldots, a_n), c$):
1. Set $y := a_n$.
2. For $i := 0, 1, 2, \ldots, n-1$:
3.     Set $y := c \times y + a_{n-i-1}$.
4. Return $y$.
```

Hint: Identify an invariant that holds at the start of each iteration of the loop, use proof by induction to show that it is a valid invariant, and then finally show why the invariant guarantees this algorithm will produce the correct output.

Solution:

Let's run the algorithm for a few iterations and print the value of y at the start of iteration i. To differentiate between the values y takes in time, we denote by y_i the value of y at the start of iteration i.

$$\begin{aligned}
i = 0,&\quad y_0 = a_n && = a_n c^0 \\
i = 1,&\quad y_1 = c(a_n c^0) + a_{n-1} && = a_n c^1 + a_{n-} c^0 \\
i = 2,&\quad y_2 = c(a_n c^1 + a_{n-1} c^0) + a_{n-2} && = a_n c^2 + a_{n-1} c^1 + a_{n-2} c^0 \\
&\vdots
\end{aligned}$$

We can notice a few patterns here: (i) there are $i+1$ terms, (ii) the coefficient of first term is a_n, the second a_{n-1}, etc., and (iii) the power of c is i in the first term, $i-1$ in the second term, etc.

We can write this as a summation and derive the following following invariant:

Invariant. At the beginning of the ith iteration,

$$y = \sum_{k=0}^{i} a_{n-k} c^{i-k}$$

Proof: By induction over the number of iterations.

Base case: y is assigned to a_n on line 1. Therefore, at the start of iteration 0,

$$\begin{aligned} y &= a_n \\ &= \sum_{k=0}^{0} a_{n-k} c^0 && \text{which is the invariant for } i = 0. \end{aligned}$$

Inductive step: Assume that the invariant holds at the start of iteration i, that is

$$y_i = \sum_{k=0}^{i} a_{n-k} c^{i-k}. \tag{1}$$

(This is our inductive hypothesis.) We want to show that the invariant holds at the start of iteration $i+1$, that is

$$y_{i+1} = \sum_{k=0}^{(i+1)} a_{n-k} c^{(i+1)-k}.$$

Let us compute the value of y at the start of the next iteration:

$$\begin{aligned} y_{i+1} &= c \times y_i + a_{n-i-1} && \text{line 3 of the algorithm} \\ &= c \sum_{k=0}^{i} a_{n-k} c^{i-k} + a_{n-i-1} && \text{substitute } y_i \text{ according to (1)} \\ &= \sum_{k=0}^{i} a_{n-k} c^{i-k+1} + a_{n-(i-1)} && \text{push } c \text{ inside the summation} \\ &= \sum_{k=0}^{i} a_{n-k} c^{(i+1)-k} + a_{n-(i+1)} && \text{put parentheses around } (i+1) \text{ for emphasis} \\ &= \sum_{k=0}^{i} a_{n-k} c^{(i+1)-k} + \sum_{k=i+1}^{i+1} a_{n-k} c^{(i+1)-k} && \text{rewrite } a_{n-(i+1)} \text{ as a single-term summation} \\ &= \sum_{k=0}^{(i+1)} a_{n-k} c^{(i+1)-k} && \text{combine the two summations} \end{aligned}$$

Which is exactly our invariant for the beginning of the $(i+1)$-th iteration.

Therefore, by induction on i, the invariant holds at the start of every iteration of the loop. □

We can now prove that correctness of the algorithm.

Proof: The algorithm returns the value of y after the last iteration—which is the value of y at the start of the next iteration, had it been executed. If we substitute $i = n$ in the invariant we get

$$\begin{aligned} y_n &= \sum_{k=0}^{n} a_{n-k} c^{n-k} \\ &= a_n c^n + \ldots + a_2 c^2 + a_1 c + a_0. \end{aligned}$$

This is exactly $P(c)$. Therefore the algorithm returns the value of $P(c)$. □

3. (35 pts.) Prove this algorithm correct

We say that v is a *majority* value for the list L if $> 50\%$ of the elements of L have the value v. Let's prove that, if L has a majority value, then the following algorithm will output it. (If L does not have a majority value, then the algorithm is allowed to output anything it wants, without restriction.)

```
Algorithm BM($A[0..n-1]$):
1. Set $c := 0$. Set $v :=$ null.
2. For $i := 0, 1, 2, \ldots, n-1$:
3.     If $c = 0$, set $v := A[i]$.
4.     If $v = A[i]$, set $c := c+1$, else set $c := c-1$.
5. Output $v$.
```

We'll guide you through a proof that this algorithm is correct, with the following steps:

(a) Prove that c never goes negative.

Solution:

Proof: By contradiction. Consider the first time c becomes negative. c is initialized to 0 and can only change in line 4, where it is either increased or decreased by 1. This means that if c goes negative in this iteration of the loop, c has to be zero at the start of the iteration. But if $c = 0$ at the start of the iteration, then v becomes $A[i]$ in line 3, and the condition in line 4 is true, so c is incremented to 1. This contradicts our assumption that c becomes negative in this iteration. $\square$

Alternative solution: You could also prove this by induction on i, by proving that $c \geq 0$ is an invariant of the loop. The inductive step will do a case-split on whether $c = 0$ or $c > 0$: if $c = 0$, then the code will increment c, preserving the invariant, while if $c > 0$, the code will either increment or decrement it by 1, so $c > -1$, i.e., $c \geq 0$.

(b) The following is an invariant of the algorithm:

> At the start of any iteration of the loop, the elements of $A[0..i-1]$ can be partitioned into two groups: a group U_i of at least c instances of the value v, and a group P_i of elements that can be paired off so that the two elements in each pair differ.

For example, consider the input $[2,3,3]$. After one iteration, $[2]$ can be partitioned into the group $U_1 = [2]$ and $P_1 = []$. After two iterations, $[2,3]$ can be partitioned into the group $U_2 = []$ and $P_2 = [2,3]$. After two iterations, $[2,3,3]$ can be partitioned into the group $U_3 = [3]$ and $P_3 = [2,3]$.

Prove by induction that this invariant holds at the start of each iteration of the loop.

Hint: Show the base case, then the inductive step. Break the inductive step down into three cases: (i) $c = 0$, (ii) $c > 0$ and $v = A[i]$, (iii) $c > 0$ and $v \neq A[i]$. For each case, use the inductive hypothesis to explain how U_{i+1} and P_{i+1} can be formed.

Solution:

We shall use a slightly stronger version of the invariant (difference in bold):

> At the start of any iteration of the loop, the elements of $A[0..i-1]$ can be partitioned into two groups: a group U_i of ~~at least~~ **exactly** c instances of the value v, and a group P_i of elements that can be paired off so that the two elements in each pair differ.

Intuitively, the algorithm maintains a count c of instances of some element v that have yet to be paired with an element that differs from v.

(i) When $c = 0$, all the elements have been paired off, so the algorithm now has an new element $A[i]$ that is yet to be paired off. The count c is set to 1 to reflect that and v is set to this new element.

(i) When $c > 0$ and $v = A[i]$, the algorithm cannot pair the new element $A[i]$ with v because they are the same. The count c is incremented to reflect that there is an additional instance of v to pair.

(i) When $c > 0$ and $v \neq A[i]$, the algorithm can pair v with $A[i]$, so it decrements c to reflect that there is now one less instance of v that needs to be paired.

Proof: By induction on the iteration number, i.

Base case: At the start of iteration 0, $A[0..i-1]$ is empty. It can be partitioned to two empty groups U_0 and P_0, with $c = 0$ (and v does not matter).

Inductive step: We assume the invariant holds at the start of iteration i and prove it holds at the end of the iteration (which is the start of iteration $i+1$). The induction hypothesis ensures that $A[0..i-1]$ can be partitioned into groups P_i and U_i that satisfy the invariant. We will use this fact to show that $A[0..i]$ can be partitioned into P_{i+1} and U_{i+1} that satisfy the conditions of the invariant at the start of iteration $i+1$. As the hint suggests, we split the proof into three cases:

(i) $c = 0$. In this case, the inductive hypothesis implies that U_i is empty, so P_i has to contain all the elements of $A[0..i-1]$. At the end of the iteration, $c = 1$ and $v = A[i]$. We define $P_{i+1} := P_i = A[0..i-1]$, and $U_{i+1} := [A[i]]$. This is a partition of $A[0..i]$. Also, all the elements of P_{i+1} can be paired off, because all the elements of P_i could be paired off. Finally, U_{i+1} contains exactly $c = 1$ instances of $v = A[i]$. Therefore, this choice of P_{i+1}, U_{i+1} satisfies the conditions of the invariant at the start of iteration $i+1$.

(i) $c > 0$ and $v = A[i]$. The inductive hypothesis tells us that U_i contains $c > 0$ instances of $v = A[i]$. In line 4, c is incremented and v is left unchanged. We define $P_{i+1} := P_i$ and $U_{i+1} := U_i + [A[i]]$. This is a partition of $A[0..i]$, all the elements of P_{i+1} can be paired off because it is the same as P_i, and U_{i+} now contains an additional instance of v.

(i) $c > 0$ and $v \neq A[i]$. The inductive hypothesis says that U_i contains $c > 0$ instances of v. Therefore, we can pair one of these instances of v with $A[i]$. We define $P_{i+1} := P_i + (v, A[i])$ and $U_{i+} := U_i - [v]$. This is a partition, all the elements of P_{i+1} can be paired by pairing v and $A[i]$ and pairing the rest according to P_i, and U_i now contains one less instances of v.

We have seen that if the invariant holds at iteration i it will hold at the start of iteration $i+1$. □

(c) Prove why the invariant from part (b) implies that the algorithm is correct.

Hint: Consider what happens if the array has a majority element.

Solution:

Proof:

When the loop terminates, the invariant guarantees that the elements of the array can be partitioned into a group P of elements that can be paired off so that the two elements in each pair differ and a group U of exactly c instances of v.

If the array has a majority element, more than half of the elements have the same value making it impossible to pair every instance of the majority element to an element that differs from it. No matter how we partition the array into P and U, P cannot contain all the instances of the majority element; U has to contain at least one majority element. Moreover, the invariant tells us that at the end v is equal to that majority element.

Therefore, if there is a majority element, the algorithm outputs the majority element. □

(d) (3 pts.) Optional bonus problem: Check my proof

(This is an *optional* bonus challenge problem. Only solve it if you can't get enough of the algorithms goodness. We reserve the right not to grade bonus problems if necessary.)

If $G = (V, E)$ is an undirected graph and $S \subseteq V$ is a set of vertices, call the edge (u, v) *good* if one of u, v is in S and the other is not in S; and call the set S *nice* if it makes at least half of the edges of the graph good. We want an algorithm that, given an undirected graph $G = (V, E)$, outputs a nice set S. Consider the following algorithm and proof of correctness:

```
Algorithm MAKENICE($G$):
1. Set $S := \emptyset$.
2. Loop until $S$ is nice:
3.     Let $w$ be a vertex whose degree is maximal (resolve ties arbitrarily).
4.     Set $S := S \cup \{w\}$.
5.     Delete all edges that have $w$ as an endpoint.
6.     If $S$ is nice (for the original graph), return $S$.
```

Proof: Imagine keeping track of the number of good edges (from the original graph), as the algorithm executes. We will show that the number of good edges increases in each iteration of the loop.

In particular, when we add w to S in step 4, every edge currently in the graph that has w as an endpoint will shift from non-good to good.

(Why? Well, the algorithm immediately deletes all edges whenever one of their endpoints enters S, so whenever execution reaches step 3, all remaining edges connect two vertices in $V \setminus S$. In particular, all edges that haven't been deleted yet are non-good. But the ones that have w as an endpoint will immediately become good when w is added to S.)

Since we strictly increase the number of good edges each time we execute an iteration of the loop, eventually the number of good edges must exceed $|E|/2$. □

Is this proof of correctness valid? Write "Valid" or "Invalid." If you write "Valid", analyze the running time of the algorithm. If you write "Invalid", explain what is wrong (list the first step in the reasoning that doesn't follow).

Solution:

The proof is Invalid. The proof states correctly that:

> ...when we add w to S in step 4, every edge currently in the graph that has w as an endpoint will shift from non-good to good.

It then uses this to claim that:

> Since we **strictly increase** the number of good edges each time we execute an iteration of the loop, eventually the number of good edges must exceed $|E|/2$.

However, this claim does not follow.

The problem is that there is no proof that the number of good edges is strictly increasing, only that in each iteration some edges become good. It ignores the fact that some edges may become non-good again. Indeed, an edge first become good when one of its vertices is added to S and goes back to being non-good when the other vertex is added to S.

Because the proof does not show that in each step of the algorithm the number of edges that become good is larger than the number of edges that stop being good, we cannot conclude that the number of good edges is strictly increasing.

A natural follow-up question is: OK, well, <u>is</u> the algorithm correct? Can we find some other proof of correctness—or is the algorithm simply flawed? It turns out that the algorithm is flawed; it works correctly on many graphs, but there are some graphs where it simply fails to find a nice set. If you enjoyed this bonus problem, you might like trying to find a counterexample where the algorithm fails.