

CS303 (Spring 2008)— Midterm solutions

- (1) (a) The loop invariant is that right before entering the `for` loop with a given value for i , the variable `count` contains the number of adjacent pairs up to (including) array position i which are in correct order.

We prove this by induction on i . Before entering with $i = 0$, `count` has value 0, which is the number of adjacent pairs up to position 0 which are in correct order. This establishes the base case.

For the inductive step, we assume by hypothesis that `count` contains the correct value up to position i before entering with value i . During this iteration, `count` is incremented if and only if the pair $a[i], a[i + 1]$ is in correct order. Thus, `count` correctly captures the number of pairs in correct order up to position $i + 1$ before entering with value $i + 1$.

- (b) The initialization does constant work, as does each of the $\Theta(n)$ iterations, so the running time is $\Theta(n)$.
- (2) (a) The proof uses induction on the number n of digits. In the base case $n = 1$, the algorithm is correct because it computes the product directly by multiplying two digits.

For the inductive step, we can assume by induction hypothesis (because the number of digits for each of the subproducts is smaller) that the subproducts in lines 5 and 6 are computed correctly. Thus, $z_2 = x_2y_2, z_1 = x_1y_1, z_0 = x_0y_0, w_3 = (x_2 + x_1)(y_2 + y_1), w_2 = (x_2 + x_0)(y_2 + y_0), w_1 = (x_1 + x_0)(y_1 + y_0)$. Substituting these values into the assignment made in line 7 gives us

$$\begin{aligned} z &= x_2y_2 \cdot 2^{4n/3} + ((x_2 + x_1)(y_2 + y_1) - x_2y_2 - x_1y_1) \cdot 2^n \\ &\quad + (x_1y_1 + (x_2 + x_0)(y_2 + y_0) - x_2y_2 - x_0y_0) \cdot 2^{2n/3} \\ &\quad + ((x_1 + x_0)(y_1 + y_0) - x_0y_0 - x_1y_1) \cdot 2^{n/3} + x_0y_0 \\ &= x_2y_2 \cdot 2^{4n/3} + (x_2y_1 + x_1y_2) \cdot 2^n + (x_1y_1 + x_2y_0 + x_0y_2) \cdot 2^{2n/3} \\ &\quad + (x_0y_1 + x_1y_0) \cdot 2^{n/3} + x_0y_0 \\ &= (x_2 \cdot 2^{2n/3} + x_1 \cdot 2^{n/3} + x_0) \cdot (y_2 \cdot 2^{2n/3} + y_1 \cdot 2^{n/3} + y_0) \\ &= x \cdot y. \end{aligned}$$

In the first step, we simply multiplied out and canceled terms, and in the second step, we factored. Thus, the result returned is correct.

- (b) The amount of work for all additions, breaking numbers apart, multiplications with powers of 2 (shifting) etc. is $\Theta(n)$. There are 6 recursive invocations of `Multiply`, each on inputs of $n/3$ bits. Thus, the recurrence relation is $T(n) = 6T(n/3) + \Theta(n)$.
- (c) We can apply case 1 of the Master Theorem, because $a = 6, b = 3$, so $f(n) = \Theta(n) = n^{\log_3 6 - \epsilon}$. This gives us a running time of $T(n) = n^{\log_3 6}$. (This is about $n^{1.631}$, thus slightly worse than the previous algorithm.)
- (3) (a) There are at least three different easy proofs. The first one is to notice that the cost of *each* spanning tree is increased by exactly $n - 1$, because each has $n - 1$ edges, each of which has its cost increase by 1. Because the same number is added to each spanning tree cost, the cheapest tree before is still the cheapest.

A second proof would observe that the cost order of the edges stays the same. Thus, Kruskal's Algorithm will execute in exactly the same way as before, and return the same tree. Because we proved Kruskal's Algorithm to be correct in class, the same tree is an MST.

A third proof would observe that for each cut, the same edge is cheapest across that cut. Thus, all of the same edges must be included in the MST by the Cut Property from class, and thus the MST stays the same.

- (b) A simple counter-example consists of a graph with four nodes s, u, v, t . There is a direct edge from s to t of cost 4, and edges from s to u , from u to v , and from v to t , each of cost 1. The 3-hop path is cheapest before the change, since it costs 3. After adding 1 to all edge costs, the 3-hop path costs 6, while the direct edge costs 5, so it is now cheapest.
- (4) (a) Focus on any one node v , and a number $j \geq 1$. The number of correct j -hop paths into v can be analyzed as follows. Each correct such path must end in an edge $e = (u, v)$ (with the same v) of the correct label $\lambda(e) = \sigma_j$. For each such correctly labeled edge, if there were some number x of different ways of getting to node u with correct $(j - 1)$ -hop paths, then there would be the same number x of paths getting to v using this particular edge e . Each such edge contributes the corresponding number of paths, so we get the recurrence

$$\text{OPT}(v, j) = \sum_{e=(u,v):\lambda(e)=\sigma_j} \text{OPT}(u, j - 1).$$

The base case is $j = 0$. In that case, there is exactly one way to get from s to itself with 0 hops, and no way to get from s to any other node in 0 hops. Thus,

$$\begin{aligned} \text{OPT}(s, 0) &= 1 \\ \text{OPT}(v, 0) &= 0 \quad \text{for } v \neq s. \end{aligned}$$

- (b) The simplest is the following bottom-up implementation:

Algorithm 1 Path Count

```

1: Let  $a[s, 0] := 1$ , and  $a[v, 0] := 0$  for all  $v \neq s$ .
2: for  $j = 1$  to  $k$  do
3:   for all nodes  $v$  do
4:     Let  $a[v, j] := 0$ .
5:     for all edges  $e = (u, v)$  into  $v$  do
6:       Let  $a[v, j] := a[v, j] + a[u, j - 1]$ .
7: return  $\sum_v a[v, k]$ .
```

Notice that we return the sum over all nodes, since we allow the paths to end in any node.