MATH182 DISCUSSION 3 WORKSHEET SOLUTIONS

1. Divide and conquer

1.1. **The next Fibonacci algorithm is the sum of the last two Fibonacci algorithms.** The MatrixPower algorithm does $\Theta(1)$ work, and calls itself with an input of half the size. Its running time $T(n)$ therefore satisfies the recurrence $T(n) = T(\lfloor n/2 \rfloor) + \Theta(1)$, which has solution $T(n) = \Theta(\lg n)$.

In practice, this does not speed up the calculation of Fibonacci numbers too much because Fibonacci numbers quickly become unmanageably large; you are unlikely to care about the precise value of F_{1000}.

The argument that F_n must take $\Omega(n)$ time to compute is correct (for most models of computation). Our time complexity analysis relied on the assumption that matrix multiplication can be done in constant time, which means producing an arbitrarily large number of digits in constant time. A more accurate analysis would account for the time needed to multiply large numbers.

1.2. **I came, I saw, I divided & conquered.** First, we perform the proposed simplifications, sorting the lines by slope and removing lines with the same slope and lower y-intercept. Then we use a divide and conquer approach on the simplified problem. For the base case, if there are one or two lines, then all lines are visible. If there are more than two, we split the lines into a "left" and "right" half, with the lines in the left half having lower slopes than lines in the right half. We find the lines which are visible in each half.

The visible lines for the overall problem will consist of the first several of those visible in the left half and the last several of those visible in the right half. To find our final answer, we need to find the crossover point. This can be accomplished by looking at the "corners" in each half, the places where one visible line intersects the next. The crossover will lie between two such corners. We can then find the crossover point by iterating through both lists of corners, taking at each step whichever corner is next to the right, and waiting until the line on top in the left half is below the line on top in the right half.

Our algorithm will assume that we are given a list of ordered pairs $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$, representing lines $y = a_i x + b_i$. We first simplify the problem by sorting and eliminating duplicate slopes:

```
VISIBLE(A)
1  sort A, with comparison (a, b) ≤ (a', b') ⟺ a ≤ a'
2  let B be the singleton list [A[1]]
3  for i = 1 to A.length
4       if B[−1][1] == A[i][1]          // If you have two parallel lines
5            B[−1][2] = max(B[−1][2], A[i][2])   // Then only include the higher one
6       else
7            append A[i] to B
8  return VISIBLEMAIN(A)
```

The main algorithm performs the divide and conquer:

```
VISIBLEMAIN(A)
 1  n = A.length
 2  if n ≤ 2
 3       return A
 4  m = ⌊n/2⌋
 5  L = VISIBLEMAIN(A[1 . . m])
 6  R = VISIBLEMAIN(A[m + 1 . . n])
 7  p = L.length
 8  q = R.length
 9  let Lcorners[1 . . p] and Rcorners[1 . . q] be new arrays
    // Lcorners and Rcorners store the x-coordinates of the corners
10  for i = 1 to p − 1
11       Lcorners[i] = (L[i][2] − L[i + 1][2])/(L[i + 1][1] − L[i][1])
12  Lcorners[p] = ∞
    // We append a corner at infinity to simplify conditions later
13  for i = 1 to q − 1
14       Rcorners[i] = (R[i][2] − R[i + 1][2])/(R[i + 1][1] − R[i][1])
15  Rcorners[q] = ∞
16  i = 1
17  j = 1
18  while i + j ≤ p + q − 2
    // Iterate over the corners
19       if Lcorners[i] < Rcorners[j]
         // Check whether the next corner is from the left or the right
20            x = Lcorners[i]
21            if L[i][1] · x + L[i][2] < R[i][1] · x + R[i][2]
              // If we've crossed over, exit the while loop
22                 break
23            i = i + 1
24       else
25            x = Rcorners[j]
26            if L[i][1] · x + L[i][2] < R[i][1] · x + R[i][2]
27                 break
28            j = j + 1
29  return join(L[1 . . i], R[j . . q])
```

2. How long does it take to find a running time?

Example 1. *Find asymptotic estimates on the solutions to the following recurrences, by whatever method you prefer.*

$$\begin{aligned}
T_1(n) &= T_1(n/2) + \lg n \\
T_2(n) &= 2T_2(2n/3) + n \\
T_3(n) &= 2T_3(n-1) + n^3 \\
T_4(n) &= 3T_4(n/2) + \frac{n^2}{\lg n} \\
T_5(n) &= 4T_5(n/2) + n^2 \lg n \\
T_6(n) &= 2T_6(n/2) + \sqrt{n} \\
T_7(n) &= 3T_7(n/3) + \frac{n}{\lg^2 n} \\
T_8(n) &= T_8(n-1) + T_8(n-2) + n \\
T_9(n) &= 3T_9(3n/4) + n^3
\end{aligned}$$

Solution. We have the following asymptotic estimates:

$$\begin{aligned}
T_1(n) &= \Theta(\lg^2 n) \\
T_2(n) &= \Theta(n^{\log_{3/2} 2}) \\
T_3(n) &= \Theta(2^n) \\
T_4(n) &= \Theta(\frac{n^2}{\lg n}) \\
T_5(n) &= \Theta(n^2 \lg^2 n) \\
T_6(n) &= \Theta(n) \\
T_7(n) &= \Theta(n) \\
T_8(n) &= \Theta(\phi^n), \qquad \text{where } \phi = \frac{1+\sqrt{5}}{2} \\
T_9(n) &= \Theta(n^{\log_{4/3} 3})
\end{aligned}$$

All of these except T_3 and T_8 can be obtained by the master method. For instance, the recurrence for T_1 has $a = 1$, $b = 2$, $\log_b a = 0$, and $f(n) = \lg n = \Theta(n^{\log_b a} \lg^1 n)$, so is in case 2(a), and has solution $T_1(n) = \Theta(\lg^2 n)$. The recurrence for T_5 is also in case 2(a). The recurrences for T_2, T_6, and T_9 are all in case 1, while T_4 is in case 3 and T_7 is in case 2(c).

For T_3, we can draw a recursion tree. The 0^{th} level will have a single $T(n)$, with a cost of n^3. The first level will have two $T(n-1)$s, each costing $(n-1)^3$. The second level will have four $T(n-2)$s, each costing $(n-2)^3$. The k^{th} level will have 2^k $T(n-k)$s, each costing $(n-k)^3$. The tree has n levels, with the last level containing 2^n leaves. The contribution from the leaves is therefore $\Theta(2^n)$, while the total cost from the rest of the levels is $\sum_{k=0}^{n-1} 2^k(n-k)^3 = 2^n \sum_{j=1}^{n} \frac{j^3}{2^j} = \Theta(2^n)$.

We therefore make the guess $T_3(n) = \Theta(2^n)$. For the lower bound, suppose that $T(n) \geq c2^n$ for some c and n. Then

$$T(n+1) \;=\; 2T(n) + (n+1)^3 \;\geq\; c2^{n+1} + (n+1)^3 \;\geq\; c2^{n+1}.$$

For the upper bound, if we simply plugged in $c \cdot 2^n$, we would get $c \cdot 2^{n+1} + (n+1)^3$. So we will instead suppose that $T(n) \le c2^n - 2n^3$ for some c and some large n ($n \ge 10$ is sufficient). Then

$$T(n+1) \;=\; 2T(n) + (n+1)^3 \;\le\; c2^{n+1} + (n+1)^3 - 4n^3 \;\le\; c2^{n+1} - 2(n+1)^3.$$

The last inequality follows from $(n+1)^3 \le \frac{4}{3}n^3$, which is true for large n. Taking constants so that these bounds hold for some n, we find that they hold for all larger n, so $T(n) = \Theta(2^n)$.

To analyze T_8, we could similarly introduce a recursion tree. The analysis of this recursion tree is more complicated, since it is not uniform; it turns out to have a Fibonacci number of nodes and a Fibonacci number of leaves, with $T(n-k)$ appearing F_{k+1} times, with corresponding costs. We can shortcut this analysis, however, by noticing that in the analysis of T_3, the n^3 term ended up being negligible. We therefore take as an initial guess that the n term in the recurrence for T_8 is negligible, which gives an asymptotic estimate of $\Theta(\phi^n)$. We then verify this by substitution in the same way that we did our $\Theta(2^n)$ estimate for T_3, subtracting off $2n^2$ for the upper bound $\square$

Example 2. *For each of the following recurrence, give the best asymptotic estimate you can come up with quickly. This can be a precise big-Θ estimate, different upper and lower bounds, a general class such as "exponential" or "polynomial", etc. Try to come up with an initial estimate in thirty seconds before trying to refine or verify it.*

$$\begin{aligned}
T_1(n) &= T_1(n-1) + n^2 \lg n \\
T_2(n) &= T_2(n/2) + \lg^2 n \\
T_3(n) &= nT_3(n-1) + 1 \\
T_4(n) &= T_4(n-1) + T_4(n-2) + 2T_4(n-3) + n \\
T_5(n) &= 3T_5(n/2) + n \\
T_6(n) &= 4T_6(n/2) + 2^n \\
T_7(n) &= T_7(n-1) + T_7(n-2) + T_7(n/2) + 1 \\
T_8(n) &= T_8(n-1) + T_8(n/2) + 1 \\
T_9(n) &= T_9(n/2) + T_9(n/3) + T_9(n/4) + n \\
T_{10}(n) &= nT_{10}(n/2) + 1
\end{aligned}$$

Solution. We have the following asymptotic estimates:

$$\begin{aligned}
T_1(n) &= \Theta(n^3 \lg n) \\
T_2(n) &= \Theta(\lg^3 n) \\
T_3(n) &= \Theta(n!) \\
T_4(n) &= \Theta(2^n) \\
T_5(n) &= \Theta(n^{\log_2 3}) \\
T_6(n) &= \Theta(2^n) \\
T_7(n) &= \Theta(\phi^n) \\
T_8(n) &= \Theta(n^{\frac{1}{2}\lg n - \lg\lg n + \frac{1}{2} + \frac{1}{\ln 2} + o(1)}) \\
T_9(n) &= \Theta(n^\alpha), \qquad \text{where } 2^{-\alpha} + 3^{-\alpha} + 4^{-\alpha} = 1 \\
T_{10}(n) &= \Theta(n^{\frac{1}{2}\lg n})
\end{aligned}$$

(1) For T_1, we can draw a recursion tree. Each level has a single node, and there are n levels. The total cost is $\sum_{k=1}^{n} n^2 \lg n = \Theta(n^3 \lg n)$. Substituting the sum verifies this guess.

Alternatively, by repeatedly expanding

$$\begin{aligned} T(n) &= T(n-1) + f(n) \\ &= T(n-2) + f(n-1) + f(n) \\ &= T(n-3) + f(n-2) + f(n-1) + f(n) \\ &= \cdots, \end{aligned}$$

we can see that the recurrence for T_1 has solution $T_1(n) = T(1) + \sum_{k=2}^{n} k^2 \lg k = \Theta(n^3 \lg n)$.

(2) We can use the master method to estimate T_2. This is very similar to T_1 in Example 1, and leads to the estimate $T_2(n) = \Theta(\lg^2 n)$.

(3) For T_3, we first notice $T(n) = nT(n-1)$ is precisely the recurrence for the factorial. Since $n!$ is much larger than 1, we expect the 1 to be negligible. Indeed, if $T_3(n) \geq c \cdot n!$, then $T_3(n+1) \geq c \cdot (n+1)!$, and if $T_3(n) \leq c \cdot n! - 1$, then $T_3(n+1) \leq c \cdot (n+1)! - n \leq c \cdot (n+1)! - 1$. Thus $T_3(n) = \Theta(n!)$.

(4) The recurrence for T_4 is an inhomogeneous linear recurrence. Homogeneous linear recurrences of this form almost always have exponential solutions; in particular, since the right-hand side contains terms of the form $T_4(n-k)$ with constant coefficients whose sum is greater than 1, T_4 will grow exponentially. The precise rate of growth is determined by the largest solution of the characteristic equation $x^3 = x^2 + x + 2$. (This equation can be derived by setting $T_4(n) = x^n$ and removing the $+n$ term.) The largest solution of this equation is $x = 2$, so we expect $T_4(n) = \Theta(2^n)$. This is verified by substitution as before, with the lower bound being immediate and the upper bound requiring a substitution of the form $T_4(n) \leq c \cdot 2^n - n$. There is a slight difference from T_1 and T_3 in that we need three base cases instead of one, but this does not cause any additional difficulty.

(5) From the form of the recurrence—the fact that it involves only terms of the form $T(cn)$ for $c < 1$ and a constant term which is polynomially bounded—we expect polynomial growth. From the master method, we can find precisely that $T_5(n) = \Theta(n^{\log_2 3})$.

(6) The form of the recurrence for T_6 suggests polynomial growth due to the $4T_6(n/2)$, and size 2^n due to the 2^n. The master method confirms that $T_6(n) = \Theta(2^n)$.

(7) The recurrence for T_7 involves three kinds of terms: terms of the form $T(n-k)$, terms of the form $T(cn)$, and terms of the form $f(n)$. To guess its growth rate, we estimate the growth due to each type of term. The linear recurrence $T(n) = T(n-1) + T(n-2)$ grows exponentially, specifically like ϕ^n. The recurrence $T(n) = T(n/2)$ doesn't grow at all, though it can help growth due to other terms. Lastly, 1 is constant. Since the ϕ^n growth is by far the biggest, we expect $T_7(n) = \Theta(\phi^n)$. The verification by substitution this time has to deal with the $T(n/2)$ term as well as the constant term. A similar approach works: the lower bound is immediate, and for the upper bound we substitute $T(n) \leq c(\phi^n - \phi^{n/2}) - 1$.

(8) Like T_7, the recurrence for T_8 involves three types of terms. This time, however, we can't use the same trick, since none of the terms dominates the others. We need to turn to other methods to find estimates.

For our first bound, we note that T_7 must be eventually increasing (assuming that it is asymptotically positive). Also, for fixed k and large enough n, $n/2 < n - k$. Therefore, if we replace the $T_7(n/2)$ term by $T_7(n-k)$ for a constant k, we should get a recurrence whose solution grows faster than T_7. By the same method as for T_4, the solution to such a recurrence is $\Theta(a_k^n)$, where a_k is the largest (real) solution of $x^k = x^{k-1} + 1$. As k increases, a_k converges to 1, so we find that $T_7(n)$ grows slower than c^n for any $c > 1$, i.e. T_7 is *sub-exponential*.

For our next bound, we use the same trick, but this time replacing $T_7(n-1)$ with $T_7(cn)$ in the recurrence to make the solution grow more slowly. As we see in the analysis of T_9

below, the solution to this modified recurrence is $\Theta(n^{\alpha_c})$, where $c^{\alpha_c} + 2^{-\alpha_c} = 1$. As c approaches 1, α_c becomes arbitrarily large, so $T_7(n)$ grows faster than any power of n: it is *super-polynomial*.

The gap between polynomial growth and exponential growth is pretty large, so just these bounds don't tell us too much. For our next estimate, we use another trick, related to the recursion tree method. In the recursion tree method, we would expand $T_7(n)$ into $T_7(n-1)$ and $T_7(n/2)$, then expand those into $T_7(n-2)$ and $T_7((n-1)/2)$, and $T_7(n/2-1)$ and $T_7(n/4)$, respectively, and continue in this way. What we do instead is we only continue expanding $T(n-1)$. So we write

$$\begin{aligned} T_7(n) &= T_7(n-1) + T_7(n/2) + 1 \\ &= T_7(n-2) + T_7((n-1)/2) + T_7(n/2) + 2 \\ &= T_7(n-3) + T_7((n-2)/2) + T_7((n-1)/2) + T_7(n/2) + 3 \\ &= \cdots \\ &\vdots \\ &= T_7(n/2) + \sum_{k=0}^{n/2-1} T_7((n-k)/2) + n/2. \end{aligned}$$

Then we repeat our previous trick, first replacing all of the $T_7((n-k)/2)$s by $T_7(n/2)$s for an upper bound, then replacing them by $T_7(n/4)$s for a lower bound. (We can get more precise estimates by using different replacements, but won't bother.) The resulting recurrences can be analyzed similarly to T_{10}, with the conclusion that $T_7(n) = 2^{\Theta(\lg^2 n)}$.

Our last method is term-matching: we make a guess as to the form of the answer, then plug it in and try to adjust constants so that it works. Our first guess for the answer comes from our previous estimate: T_7 might grow like $2^{(c+o(1))\lg^2 n}$ for some constant c. If we plug $2^{c\lg^2 n}$ into the recurrence, the equation which should hold is

$$\begin{aligned} 2^{c\lg^2 n} &\approx 2^{c(\lg^2 n + 2(\lg n)\lg(1-1/n) + \lg^2(1-1/n))} + 2^{c(\lg^2 n - 2\lg n + 1)} \\ &= 2^{c\lg^2 n}\Big(1 - \frac{2c\lg n}{n} + O\big(\frac{\lg^2 n}{n^2}\big) + 2n^{-2c}\Big). \end{aligned}$$

From this equation, we can deduce that c should be $\frac{1}{2}$ (if $c < \frac{1}{2}$ or $c > \frac{1}{2}$, the equation doesn't hold). In fact, since we want the terms $-\frac{2c\lg n}{n}$ and $2n^{-2c}$ to cancel, we could do even better by putting $c = \frac{1}{2} - \frac{\lg\lg n}{2\lg n}$. This gives us a guess for the form of our next approximation: $2^{\frac{1}{2}\lg^2 n - c(\lg n)\lg\lg n}$ (we can't immediately jump to $-\frac{1}{2}(\lg n)\lg\lg n$, because our previous expansion assumed that c was constant). This time, we get $c = 1$, and we expect the next correction to add $O(\lg n)$ to the exponent. One more round of this, and simplifying the expression, gives the estimate $T_7(n) = n^{\frac{1}{2}\lg^2 - \lg\lg n + \frac{1}{2} + \frac{1}{\ln 2} + o(1)}$ (the $o(1)$ term is $O\big(\frac{(\lg\lg n)^2}{\lg n}\big)$, but we've already given a more precise estimate than we are likely to ever need).

(9) The recurrence for T_9 contains several terms of the form $T_9(cn)$. Such recurrences can in general be solved with the Akra-Bazzi method, but it is often possible to find solutions by essentially an informed guess.

First, some bounds: we expect T_9 to be increasing, so if we replace $T_9(n/4)$ and $T_9(n/3)$ by $T_9(n/2)$ in the recurrence, the solution should increase, while if we replace the larger two with $T_9(n/4)$, the solution should decrease. Both of these have solutions which are asymptotically equivalent to a power of n.

We therefore might expect the solution to the actual recurrence to be $\Theta(n^\alpha)$ for some constant α. Plugging n^α into the recurrence, and ignoring the constant term, we get $n^\alpha = (n/2)^\alpha + (n/3)^\alpha + (n/4)^\alpha$, i.e. $2^{-\alpha} + 3^{-\alpha} + 4^{-\alpha} = 1$. Let α be the unique solution to this equation. Since $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} > 1$, $\alpha > 1$. Then we can verify that $T_9(n) = \Theta(n^\alpha)$ by substitution as above (we need $\alpha > 1$ to ensure that $cn^\alpha - n > 0$ for large n) .

(10) We will once again find that the constant term is inconsequential, and our primary task is to analyze the recurrence $T(n) = nT(n/2)$. This can be done a few different ways. One way is to define $\widetilde{T}(n) = \lg T(n)$, so the recurrence becomes $\widetilde{T}(n) = \widetilde{T}(n/2) + \lg n$. We solved this recurrence asymptotically in Example 1, but here we need a more precise solution. We get

$$\widetilde{T}(n) \approx \sum_{k=1}^{\lg n} \lg(n/2^k) = \lg^2 n - \frac{\lg n(\lg n + 1)}{2} = \frac{1}{2}\lg^2 - \frac{1}{2}\lg n.$$

The important part here is the $\frac{1}{2}\lg^2 n$ term, as the other term is too sensitive to how we do the estimate. We therefore make a first guess that $T_{10}(n)$ should grow approximately like $2^{\frac{1}{2}\lg^2 n} = n^{\frac{1}{2}\lg n}$. Substituting, we find that this works exactly, and so obtain $T_{10}(n) = \Theta(2^{\frac{1}{2}\lg^2 n})$.

□

3. Midterm review

We prove correctness of the following algorithm.

LongestCommonSubarray(A, B)

```
1  maxlength = 0
2  for i = 0 to A.length − 1                  // Iterate through A
3      for j = 0 to B.length − 1              // Iterate through B
4          k = 0
5          while A[i + k] == B[j + k]         // Find the longest equal subarrays starting at A[i] and B[j]
6              k = k + 1
7          if k > maxlength
8              maxlength = k
9  return maxlength
```

We will need three loop invariants. The loop invariant for the **while** loop on lines 5-6 is

$A[i\,..\,i+k-1] = B[j\,..\,j+k-1]$ and *maxlength* is unchanged from the start of the loop.

For the inner **for** loop, lines 3-8:

maxlength is the maximum of its value at the start of the loop, and the length of the longest common subarray of A and B which starts at $A[i]$ and some $B[j']$ with $j' < j$.

For the outer **for** loop, lines 2-8:

maxlength is the length of the longest common subarray of A and B which starts at some $A[i_0]$ and $B[j_0]$ such that $i_0 < i$.

We now prove these. First we verify the loop invariant for the **while** loop:

(Initialization) The first time line 5 is run, $k = 0$, and $A[i\,..\,i-1]$ and $B[j\,..\,j-1]$ are both empty arrays. Also, *maxlength* has not been modified.

(Maintenance) Suppose that line 5 has just been run, $k = k_0$, and the loop invariant holds. If $A[i + k_0] \neq B[j + k_0]$, then the loop terminates. Otherwise, line 6 runs, k is incremented to $k = k_0 + 1$. Since $A[i + k_0] = B[j + k_0]$ and $A[i\,..\,i + k_0 - 1] = B[j\,..\,j + k_0 - 1]$, we have

$A[i\,..\,i+k_0] = B[j\,..\,j+k_0]$. Also, *maxlength* was not modified at any point. Therefore, the loop invariant still holds.

(Termination) There are two ways that the loop can terminate. Either $A[i+k] \neq B[j+k]$, or either $A[i+k]$ or $B[j+k]$ does not exist. In either case, $A[i\,..\,i+k-1]$ and $B[j\,..\,j+k-1]$ cannot be extended to longer equal subarrays starting at $A[i]$ and $B[j]$. By the loop invariant, $A[i\,..\,i+k-1] = B[j\,..\,j+k-1]$. Thus k is the length of the longest common subarray of A and B which starts at $A[i]$ and $B[j]$.

Next we verify the loop invariant for the inner **for** loop:

(Initialization) The first time line 3 is run, there are no $j' < j$, and *maxlength* is equal to its initial value.

(Maintenance) Suppose line 3 has just been run, that $j = j_0$ for some $1 \leq j_0 \leq B.\mathit{length} - 1$, and the loop invariant holds. First, we initialize k to zero, then we execute the loop in lines 5-6. Then we move to line 7. By the Termination condition we proved for the **while** loop, k is now the maximum length of equal subarrays starting at $A[i]$ and $B[j_0]$. If this is larger than *maxlength*, then we set *maxlength* equal to k in line 8, otherwise we do nothing; in other words, *maxlength* is set to the maximum of k and its current value. Then we set $j = j_0 + 1$ and return to line 3.

From the loop invariant and the description of k, *maxlength* is now equal to the maximum of the length of the longest common subarray starting at $A[i]$ and $B[j_0]$, the length of the longest common subarray starting at $A[i]$ and $B[j']$ for some $j' < j_0$, and its value at the start of the loop. This is the same as the maximum of its original value and the length of the longest common subarray starting at $A[i]$ and $B[j']$ for some $j' < j_0 + 1 = j$. Thus the loop invariant remains true.

(Termination) The last time line 3 is run, $j = B.\mathit{length}$, and the loop invariant tells us that *maxlength* is the max of its initial value and the length of the longest common subarray which starts at $A[i]$ and $B[j']$ for some $j' < j$. Since this covers all possible starting points in B, *maxlength* is equal to either its original value, or the length of the longest common subarray of A and B which starts at $A[i]$, whichever is greater.

Lastly, the loop invariant for the out **for** loop:

(Initialization) The first time line 2 is run, $i = 0$ and $\mathit{maxlength} = 0$. There are no common subarrays starting at $A[i']$, $i' < i = 0$, except technically the empty subarray, so the maximum length of such is zero.

(Maintenance) Suppose that line 2 has just been run, that $i = i_0$ for some $0 \leq i_0 \leq A.\mathit{length} - 1$, and the loop invariant holds. By the Termination condition for the inner **for** loop, after the loop in lines 3-8 is executed, *maxlength* is the maximum of the length of the longest common subarray starting at $A[i']$ for some $i' < i_0$, and the length of the longest common subarray starting at $A[i_0]$. Then i is set to $i_0 + 1$. As in Maintenance for the inner **for** loop, the loop invariant still holds.

(Termination) The last time line 2 is run, $i = A.\mathit{length}$. The loop invariant then gives that *maxlength* is the maximum length of a common subarray of A and B.

After the outer **for** loop exits, the algorithm returns *maxlength*. By the Termination condition above, this is the desired result.

We now prove correctness of the MatrixPower algorithm:

MatrixPower(A, n)

```
1  if n == 0
2      return identity matrix
3  elseif n is even
4      B = MatrixPower(A, n/2)
5      return B^2
6  else
7      B = MatrixPower(A, (n − 1)/2)
8      return B^2 A
```

As this is a recursive algorithm, we proceed by induction. Fix a square matrix A. The statement which we prove by induction is "for every $n \in \mathbb{N}$, MatrixPower(A, n) returns A^n".

The base case is $n = 0$. If $n = 0$, then MatrixPower(A, n) runs line 1, sees that $n = 0$, and runs line 2, returning the identity matrix. Since A^0 is the identity matrix by definition, this is correct.

Now let $n \geq 1$, and assume that MatrixPower(A, m) return A^m for $0 \leq m \leq n - 1$. The condition in line 1 is false, so the algorithm skips to line 3.

If n is even, then we proceed to line 4, where we set $B =$ MatrixPower$(A, n/2)$. By the induction hypothesis, $B = A^{n/2}$. In line 5, we return $B^2 = (A^{n/2})^2 = A^n$.

If n is odd, we instead skip to line 6, which does nothing, then line 7, where we set $B =$ MatrixPower$(A, (n - 1)/2)$. By the induction hypothesis, $B = A^{(n-1)/2}$. In line 8, we then return $B^2 A = A^n$.

The program thus returns A^n regardless of whether n is even or odd. Hence, by induction, for every $n \in \mathbb{N}$, MatrixPower(A, n) returns A^n.