MATH182 DISCUSSION 2 NOTES

1. Divide and conquer

1.1. **The next Fibonacci algorithm is the sum of the last two Fibonacci algorithms.** We have already seen how to compute Fibonacci numbers F_n in time $\Theta(n)$. We will now see how to compute them even faster using a divide-and-conquer approach. To do so, we will take advantage of the following fact, which can be proved quickly by induction.

Fact. *For any n,*

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

So in order to quickly compute Fibonacci numbers, it suffices to quickly compute matrix powers. To do this, we rely on an important and well-known fact about exponentiation: for any square matrix A and any $m, n \in \mathbb{N}$,

$$(A^m)^n = A^{mn}.$$

In particular, $A^{2n} = (A^n)^2$. Since also $A^{2n+1} = A \cdot (A^n)^2$, we see that we can divide the computation of A^m into a computation of $A^{\lfloor m/2 \rfloor}$ and a few multiplications. In pseudocode:

MatrixPower(A, n)

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

Find the running time of this algorithm. You may assume that matrix multiplication takes constant time. Deduce that Fibonacci numbers can be computed even faster than previously indicated. However, this is unlikely to significantly speed up the calculation of any Fibonacci number that you actually encounter. Why not? (*Hint:* How large is F_{100}?)

Consider the following argument: In oder to compute a Fibonacci number F_n, we must compute, and in particular output, each of its digits. Since $\log_{10}(F_n) = n \log_{10}(\phi) + O(1)$, where $\phi := \frac{1+\sqrt{5}}{2}$, F_n has $\Theta(n)$ digits. So it must take at least this much time to compute F_n, i.e. computing F_n takes $\Omega(n)$ time. Why does this not contradict our previous analysis?

1.2. **I came, I saw, I divided & conquered.** Consider the following problem: we are give a collection of n lines in the plane, given by equation $y = a_i x + b_i$, $i = 1, \ldots, n$. We wish to determine which of these lines are "visible from above" or "on top", that is, for which i there exists an x_0

such that $a_i x_0 + b_i \geq a_j x_0 + b_j$ for all j. Describe an algorithm to solve this problem in $O(n \log n)$ time. You may find the following facts helpful:

- The slopes of the lines on top at a given point increase from left to right.
- A list of n items can be sorted in $O(n \log n)$ time. If you start by sorting the list of lines by their slopes, you can then solve the problem just for sorted lists of lines.
- If two lines have the same slope, only one of them can be visible from above, and removing duplicates from a sorted list takes time $O(n)$. You may therefore assume that the lines have distinct slopes.

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.*

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

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.*

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

3. Midterm review

The following algorithm is meant to find the longest common (contiguous) sub-array of two given arrays. This is more commonly known as the longest common substring problem, but we haven't discussed strings and the difference is not relevant to us now. This is also a rather inefficient 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
```

Find and prove loop invariants for each of the three loops in this algorithm, and use them to verify its correctness.

Give a careful inductive proof that the recursive algorithm MatrixPower(A, n) in section 1 does in fact return A^n.