CSCI-GA.1170-001/002 Fundamental Algorithms September 24, 2014

Solutions to Problem 1 of Homework 3 (22 points)

Name: Keeyon Ebrahimi *Due: Wednesday, September 24*

The sequence $\{F_n \mid n \geq 0\}$ are defined as follows: $F_0 = 1$, $F_1 = 1$, $F_2 = 2$ and, for $i > 1$, define $F_i := F_{i-1} + F_{i-2} + 2F_{i-3}$.

(a) (6 Points) Notice the following matrix equation:

$$\begin{pmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_i \\ F_{i-1} \\ F_{i-2} \end{pmatrix} = \begin{pmatrix} F_{i+1} \\ F_i \\ F_{i-1} \end{pmatrix}$$

Use this equation and the ideas of fast $O(\log n)$ time exponentiation to build an efficient algorithm for computing F_n. Analyze it's runtime in terms of the number of 3×3 matrix multiplications performed (each of which takes a constant number of integer additions/multiplications).

Solution:

Algorithm for computing F_n

Lets set some variables here for reference. Lets say

$$M = \begin{pmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$v_i = \begin{pmatrix} F_i \\ F_{i-1} \\ F_{i-2} \end{pmatrix}$$

$$v_j = \begin{pmatrix} F_{i+1} \\ F_i \\ F_{i-1} \end{pmatrix}$$

If we think of solving this in an iterative manner, we realize that when we have

$$v_j = M \cdot v_i$$

Now we can just replace the v_i with what we just solved, which is v_j, meaning that to solve v_{j+1} , we can just do the same dot product but with the previous solution. So we do

$$v_{j+1} = M \cdot v_j$$

To better demonstrate,

1. $v_j = M \cdot v_i$
2. $v_{j+1} = M \cdot v_j$
3. $v_{j+2} = M \cdot v_{j+1}$

We can keep doing this dot product and incrementally move up until we get to the appropriate iteration. At that point, the **top value of our solution $3x1$ vector will give us the solution.**

Here is that explained algorithm out in pseudo code. We assume that n is the F_n that we want solved,

$$M = \begin{pmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

, and the initial value for v is

$$v = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}$$

```
for( i=3; i < n+1; i++)
{
        v = dotProduct(M, v);
}
return v[0]
```

Analyze it's runtime in terms of the number of 3×3 matrix multiplications performed (each of which takes a constant number of integer additions/multiplications).

With the algorithm demonstrated above, we are doing $n-2$ matrix multiplications, meaning that the runtime in terms of the matrix multiplications is $O(n)$ $\square$

(b) (6 Points) Prove by induction that, for some constant $a > 1$, $F_n = \Theta(a^n)$. Namely, prove by induction that for some constant c_1 you have $F_n \leq c_1 \cdot a^n$, and for some constant c_2 you have $F_n \geq c_2 \cdot a^n$. Thus, F_n takes $O(n)$ bits to represent. What is the right constant a and the best c_1 and c_2 you can find. (**Hint**: Pay attention to the base case $n = 0, 1, 2$. Also, you need to do *two* very similar inductive proofs.)

Solution: $a = 2,\ c_1 \geq 1,\ c_2 \leq \frac{1}{2}$

By induction, we first isolate a to reach a = 2, and here is how.

We start with this

$$F_n + 1 = F_n + F_{n-1} + 2 * F_{n-2}$$

Now we replace all the F_n's with the $c_1 * a^n$'s and we get.

$$c_1 * a^{n+1} \leq c_1 * a^n + c_1 * a^{n-1} + 2 * c_1 * a^{n-2}$$

We divide by c_1 to isolate the a's and we get

$$a^{n+1} \leq a^n + a^{n-1} + 2 * a^{n-2}$$

We solve for a and we get

$$a = 2$$

Now we have

$$F_n \leq c_1 * 2^n$$

We can also use the same induction as above, but replace the c_1 with c_2 and change the $\leq$ with $\geq$ and we get

$$F_n \geq c_2 * 2^n$$

Now we plug in the base cases into

$$F_n \leq c_1 * 2^n$$

and

$$F_n \geq c_2 * 2^n$$

to get

$F(0)$, $c_1 \geq 1$

$F(1)$, $c_1 \geq \frac{1}{2}$

$F(2)$, $c_1 \geq \frac{1}{2}$

also

$F(0)$, $c_2 \leq 1$

$F(1)$, $c_2 \leq \frac{1}{2}$

$F(2)$, $c_2 \leq \frac{1}{2}$

This means that $a = 2$, $c_1 \geq 1$, $c_2 \leq \frac{1}{2}$ □

(c) (6 points) In your algorithm of part (a) you only counted the number of 3×3 matrix multiplications. However, the integers used to compute (F_i, F_{i-1}, F_{i-2}) are $O(i)$ (in fact, $\Theta(i)$) bits long, by part (b). Thus, the 3×3 matrix multiplication used at that level of recursion will not take $O(1)$ time. In fact, using Karatsuba's multiplication, let us assume that the last matrix multiplication you use to compute (F_i, F_{i-1}, F_{i-2}) takes time $O(i^{\log_2 3})$. Given this more realistic estimate, analyze the actual running time $T(n)$ of your algorithm in part (a).

Solution: Now we know that each step multiplication takes $O(i^{\log_2 3})$. As we solved in part (a), we know that we are doing around n multiplications. So we take our solution from part (a) and multiply it by this given $O(i^{\log_2 3})$ time.

1. $O(i^{\log_2 3}) \approx i^1.5$
2. $n^{1.5} * n = n^{2.5}$

3 Solution $= O(n^{2.5})$

$\square$

(d) (4 points) Finally, let us look at the naive sequential algorithm which computes $F_3, F_4, \ldots, F_n$ one-by-one. Assuming each F_i takes $\Theta(i)$ bits to represent, and that integer addition/subtraction takes time $O(i)$ (multiplication by two can be implemented by addition), analyze the actual running time of the naive algorithm. How does it compare to your answer in part (c)?

Solution: $O(3^n)$

At each level we are creating 3 subbranches that then create 3 more of their own subbranches and so on and so forth. This means that the algorithm has a running time of $O(3^n)$. This running time is significantly more than the $O(n^{2.5})$ running time from part (c). $\square$

CSCI-GA.1170-001/002 Fundamental Algorithms September 24, 2014

Solutions to Problem 2 of Homework 3 (10 **(+6)** points)

Name: Keeyon Ebrahimi *Due: Wednesday, September 24*

The Tower of Hanoi is a well known mathematical puzzle. It consists of three rods, and a number n of disks of different sizes which can slide onto any rod. The puzzle starts with all disks stacked up on the 1st rod in order of increasing size with the smallest on top. The objective of the puzzle is to move all the disks to the 3rd rod, while obeying the following rules.

- Only one disk is moved at a time
- Each move consists of taking one disk from top of a rod, and moving it on top of the stack on another rod
- No disk may be placed on top of a smaller disk.

A recursive algorithm that solves this problem is as follows: We first move the top $n-1$ disks from rod 1 to rod 2. Then we move the largest disk from rod 1 to rod 3 and then move the $n-1$ smaller disks from rod 2 to rod 3. Using the symmetry between the rods, the number of steps that this algorithm takes is given by the recurrence

$$T(n) = 2T(n-1) + 1 ,$$

which can be solved to get $T(n) = 2^n - 1$.

(a) (5 points) Show that the above algorithm is optimal, i.e., there does not exist a strategy that solves the Tower of Hanoi puzzle in less than $2^n - 1$ steps.

Solution: Proof by Induction

We want to prove this by using induction. $T(n) = 2T(n-1) + 1$ will be our equation, and we will use the given $T(n) = 2^n - 1$ as our guess.

First, lets solve the base base.
We know that if there are 0 disks, the run time should be 0, so $T(0) = 0$.
This means that if we place 0 into our guess, we should get 0.

1. $T(0) = 2^0 - 1$
2. $T(0) = 1 - 1$
3. $T(0) = 0$

So this shows that our guess satisfies the base case. So far, so good. Now lets plug our guess into the original equation to see if we can prove the equality.

1. $T(n) = 2T(n-1) + 1$

   ** For the next step we need to place our guess $T(n) = 2^n - 1$ into the original equation of $T(n) = 2T(n-1) + 1$*

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

By substituting our guess into the original equation, we see that we solve and reduce back to our guess, meaning that we know that we have the optimal solution for this problem. □

(b) (5 points) Suppose the moves are restricted further such that you are only allowed to move disks to and from rod 2. Give an algorithm that solves the puzzle in $O(3^n)$ steps.

Solution:
If we describe the new algorithm in the same way the original algorithm was stated in the problem, we can describe it in 6 steps that need to happen n times in order to make this work with the new restriction.

1. Move $n-1$ disks from rod 1 to rod 2
2. Move those $n-1$ disks from rod 2 to rod 3
3. Move largest disk from rod 1 to rod 2
4. Move $n-1$ disks from rod 3 to rod 2
5. Move $n-1$ disks from rod 2 to rod 1
6. Move largest disk from rod 2 to rod 3

If we do that n times, we will solve the tower of Hanoi problem with the new restriction □

(c) (6 points(**Extra credit**)) Suppose the moves are restricted such that you are only allowed to move from rod 1 to rod 2, rod 2 to rod 3, and from rod 3 to rod 1. Give an algorithm that solves the puzzle in $O((1+\sqrt{3})^n)$ steps.

Solution:
We are once again going to describe this algorithm the same way that the original algorithm stated in the problem was described. This time we only have 5 steps that we need to repeat n times.

1. Move $n-1$ disks from rod 1 to rod 2
2. Move $n-1$ disks from rod 2 to rod 3
3. Move largest disk from rod 1 to rod 2
4. Move $n-1$ disks from rod 3 to rod 1
5. Move largest disk from rod 2 to rod 3

Once again, if we do this process n times, we will solve the tower of Hanoi problem with the newest restriction. □

CSCI-GA.1170-001/002 Fundamental Algorithms September 24, 2014

Solutions to Problem 3 of Homework 3 (5 Points)

Name: Keeyon Ebrahimi *Due: Wednesday, September 24*

Let n be a multiple of m. Design an algorithm that can multiply an n bit integer with an m bit integers in time $O(nm^{\log_2 3-1})$.

Solution:
For this algorithm, we will be using a slightly modified version of Karatsuba's Algorithm, which separates the two numbers to be multiplied into

$$x = x_1 * B^m + x_0$$

and

$$y = y_1 * B^m + y_0$$

In our example, we are dealing with numbers of sizes of varied bit numbers, meaning we should replace B^m with 2^m

We state the n bit number as x and the m bit number as y, and we know that x is a multiple of y. With y being m bits, we know that y_1 always is one. With n being a multiple of m and $B = 2$, we can now equations our solution to be

$$x = x_1 * 2^m + x_0$$

and

$$y = 2^y + y_0$$

Now we do a multiplication of $x * y$ and we get

$$x_1 * 2^{2m} + (x_1 y_0 + x_0) * 2^m + x_0 * y_0$$

.

With large numbers, this algorithm will cut down our running time massively.

This algorithm is a simplified version of Karatsuba's Algorithm. Karatusba's algorithm has a running time of $\theta(n^{log_2(3)})$. Because we know that n is a multiple of m, we can expand the running time of our solution to $O(nm^{\log_2 3-1})$ □

CSCI-GA.1170-001/002 Fundamental Algorithms September 24, 2014

Solutions to Problem 4 of Homework 3 (12 points)

Name: Keeyon Ebrahimi *Due: Wednesday, September 24*

An array $A[0\dots(n-1)]$ is called *rotation-sorted* if there exists some some cyclic shift $0 \le c < n$ such that $A[i] = B[(i + c \bmod n)]$ for all $0 \le i < n$, where $B[0\dots(n-1)]$ is the sorted version of A.[1] For example, $A = (2, 3, 4, 7, 1)$ is rotation-sorted, since the sorted array $B = (1, 2, 3, 4, 7)$ is the cyclic shift of A with $c = 1$ (e.g. $1 = A[4] = B[(4+1) \bmod 5] = B[0] = 1$). For simplicity, below let us assume that n is a power of two (so that can ignore floors and ceilings), and that all elements of A are distinct.

(a) (4 points) Prove that if A is rotation-sorted, then one of $A[0\dots(n/2-1)]$ and $A[n/2\dots(n-1)]$ is fully sorted (and, hence, also rotation-sorted with $c = 0$), while the other is at least rotation-sorted. What determines which one of the two halves is sorted? Under what condition *both halves* of A are sorted?

Solution:
Prove that if A is rotation-sorted, then one of $A[0\dots(n/2-1)]$ and $A[n/2\dots(n-1)]$ is fully sorted (and, hence, also rotation-sorted with $c = 0$), while the other is at least rotation-sorted

When we have something rotation sorted, if $c > 0$ then we have 1 digit that seems out of place because everything is just shifted, not re arranged. Going from a sorted list to a rotation-sorted is done by doing a simple shift. This means that only one number will seem out of place because of the wrap around. When we divide the array by 2, one of the subdivisions will have the out of place element and the other will not. That is why one of the subdivisions is fully sorted while the other one with the shift will just be rotation sorted. Also, when $c = 0$, no shift is being done, so both halves will be sorted.

What determines which one of the two halves is sorted? Under what condition both halves *of A are sorted?*

$$Sorted\ Half = \begin{cases} First\ Half\ Sorted & \text{if } c > \frac{n}{2} \\ Second\ Half\ Sorted & \text{if } c < \frac{n}{2} \\ Both\ Halves\ sorted & \text{if } c = 0 \text{ or } c = \frac{n}{2} \end{cases}$$

□

(b) (8 points) Assume again that A is rotation-sorted, but you are not given the cyclic shift c. Design a divide-and-conquer algorithm to compute the minimum of A (i.e., $B[0]$). Carefully

[1]Intuitively, A is either completely sorted (if $c = 0$), or (if $c > 0$) A starts in sorted order, but then "falls off the cliff" when going from $A[n-c-1] = B[n-1] = \max$ to $A[n-c] = B[0] = \min$, and then again goes in increasing order while never reaching $A[0]$.

prove the correctness of your algorithm, write the recurrence equation for its running time, and solve it. Is it better than the trivial $O(n)$ algorithm? (**Hint**: Be careful with $c = 0$ an $c = n/2$; you might need to handle them separately.)

Solution:
For this algorithm, we are going to create a pivot at the middle of the array. If the element to the left of the pivot is greater than the element at the pivot, we know that the pivot element is the minimum element. If this is not the case, we just run the same algorithm on the first half of the array and the second half of the array. This will find the minimum of the Array for all C, except for when c = 0. The helper function will take care of this. If we return index -1 after running the entire algorithm. We know that the minimum is at element 0. Here is the code for this algorithm.

```
int min Min(Array A)
{
        indexMin = Alg(A);

        //This if only hits if we don't find the min,
        //meaning the min is at index 0
        if (indexMin < 0)
        {
                indexMin = 0;
        }

        return A[indexMin];
}

int minIndex Alg(Array A)
{
        // This indicates that we have not reached the minimum
        if (A.length == 1)
        {
                return -1;
        }

        pivot = n/2;
        //this means we found the min at index pivot
        if (A[pivot] < A[pivot - 1]
        {
                return pivot;
        }

        else
        {
```

```
                    int A1int = Alg(A[0:n/2]);
                    int A2int = Alg(A[(n/2) + 1 : n]);
                    return max(A1, A2);
          }

}
```

This algorithm will recursively check a split array by checking if the element to the left of the pivot is greater than the pivot itself. This does not work for when $c = 0$, so that is why we have the return -1 if branch, the return max at the end of Alg(A), and also the helper function checking if the returned value is -1.

$\square$

CSCI-GA.1170-001/002 Fundamental Algorithms September 24, 2014

Solutions to Problem 5 of Homework 3 (5 Points)

Name: Keeyon Ebrahimi *Due: Wednesday, September 24*

Find a *divide-and-conquer* algorithm that finds the maximum and the minimum of an array of size n using at most $3n/2$ comparisons.
(**Hint**: First, notice that we are not asking for some iterative algorithm (which is not hard). We are asking for you to *explicitly use recursion.* In fact, your divide/conquer step should take time $O(1)$. Also, you have to be super-precise about constants and the initial case $n = 2$ to get the correct answer.)

Solution:
For this solution, we keep dividing the array into half the size until we get the size to be 2 or less.

If the size is 2, we do a simple comparison and return the one lesser element as the min, and the other as the max

If the size is 1, we just return that singled out element as both the min and the max.

For the conquer step, we just have to compare the min of the first recursive call the the min of the second, and return the min of that. Also need to compare the max of the first recursive call and the max of the second, and return the max of those. Here is some pseudo code demonstrating what I am talking about.

```
// This returns two numbers
// The first return is the min
// The second return is the max

min,max Alg(Array A)
{
        // Here we return the singled out element as the min and max
        if (A.length    == 1)
        {
                return (A[0], A[0]);
        }

        if (A.length == 2)
        {
                if (A[0] < A[1])
                {
                        return (A[0], A[1]);
                }
                else
                {
                        return (A[1], A[0]);
                }
        }

        // Time for recursive calls because size is greater than 2
        min1, max1 = Alg(A[0 : n/2]);
        min2, max2 = Alg(A[(n/2) + 1 : n]);

        // Time to conquer
        if (min1 < min2)
        {
                Finalmin = min1;
        }
        else
        {
                Finalmin = min2;
        }

        if (max1 > max2)
        {
                Finalmax = max1;
        }
        else
        {
                Finalmax = max2;
        }

        return (Finalmin, FinalMax);
}
```

□