CSCI-GA.1170-001/002 Fundamental Algorithms September 20, 2014

Solutions to Problem 1 of Homework 3 (22 points)

Name: Sahil Goel *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:

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

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

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

Lets $i = i + 1$

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

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

Similarly continuing, for $i = n$

$$\begin{pmatrix} F_{n+2} \\ F_{n+1} \\ F_n \end{pmatrix} = M^n \cdot \begin{pmatrix} F_2 \\ F_1 \\ F_0 \end{pmatrix}$$

$$\begin{pmatrix} F_{n+2} \\ F_{n+1} \\ F_n \end{pmatrix} = M^n \cdot \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}$$

Now to calculate M^n, let the function be pow(M,n)

Pow(M,n)
{
$if(n == 0)$
return I; % Where I is identity matrix of 3X3
$temp = Pow(M, n/2)$;
if(n is odd)
return $M * temp * temp$; % (* is matrix multiplication)
else
return $temp * temp$; % (* is matrix multiplication)
}
In terms of matrix multiplications
$T(n) = T(n/2)$+(1 or 2)
Using master's theorem, $f(n) = 1$ and $g(n) = 1$ or 2
Minimum will be $\log n$ when g(n)=1
Maximum will be $2 * \log n$ when g(n)=2

To calculate the exact number of multiplications, lets analyze the number of 3X3 multiplications for the algorithm. Whenever n is even there is a single 3X3 multiplication for that step and whenever it becomes odd, there are two 3X3 multiplications for the same step.
If we represent n in binary ex: 10100 and divide it by 2, it shifts right by 1 bit, so
After 1st divide : 1010
After 2nd divide : 101
After 3rd divide : 10
After 4th divide : 1
After 5th divide : 0
As you can see, it will be odd for the number of times "1" occur in the binary representation of n
Therefore total number of multiplications will be $\log n + k + 1$(for last multiplication with 3X1 Matrix), where k is number of times "1" appears in binary representation of n

□

(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: Let us assume $F_n \leq c1 \cdot a^n$

For $n = 0$, $F_0 \leq c1 \cdot a^0$
$1 \leq c1$
Therefore $c1 \geq 1$

Now for $n = 1, F_1 \leq c1 \cdot a^1$
$1 \leq c1 \cdot a$
$c1 \geq 1/a$

Now for $n = 2, F_2 \leq c1 \cdot a^2$
$2 \leq c1 \cdot a^2$
$c1 \geq 2/a^2$

Now let us assume our assumption is true for n=0,1,2,3,. . . ,n-1
We have to prove that, it is true for n=n
$F_n \leq c1 \cdot a^n$
$F_{n-1} + F_{n-2} + 2 \cdot F_{n-3} \leq c1 \cdot a^n$
$c1 \cdot a^{n-1} + c1 \cdot a^{n-2} + 2 \cdot c1 \cdot a^{n-3} \leq c1 \cdot a^n$
$c1 \cdot a^{n-3} \cdot (a^2 + a + 2) \leq c1 \cdot a^{n-3} \cdot a^3$
$a^2 + a + 2 \leq a^3$
$a^3 - a^2 - a - 2 \geq 0$
$a \geq 2$
and from above equations
$c1 \geq 1$, $c1 \geq 1/a$ and $c1 \geq 2/a^2$
Therefore $c1 \geq 1$
Hence c1=1 and a=2 will be the best solution

Now have to find a and c2 for $F(n) \geq c2 * a^n$
for n=0
$1 \geq c2$
$c2 \leq 1$

for n=1
$1 \geq c2 * a$
$c2 \leq 1/a$

for n=2
$2 \geq c2 * a^2$
$c2 \leq 2/a^2$

Now let us assume our assumption is true for n=0,1,2,3,. . . ,n-1
We have to prove that, it is true for n=n
$F_n \geq c2 \cdot a^n$
$F_{n-1} + F_{n-2} + 2 \cdot F_{n-3} \geq c2 \cdot a^n$
$c2 \cdot a^{n-1} + c2 \cdot a^{n-2} + 2 \cdot c2 \cdot a^{n-3} \geq c2 \cdot a^n$

$c2 \cdot a^{n-3} \cdot (a^2 + a + 2) \geq c2 \cdot a^{n-3} \cdot a^3$
$a^2 + a + 2 \geq a^3$
$a^3 - a^2 - a - 2 \leq 0$
$a \leq 2$ Now from above equations
$c2 \leq 1$, $c2 \leq 1/a$ and $c2 \leq 2/a^2$
$c2 \leq 1$, $c2 \leq 1/2$ and $c2 \leq 1/2$
Therefore $c2 \leq 1/2$
Hence the best solution will be a=2 and c2=1/2
$F_n \leq 1 \cdot 2^n$
$F_n \geq 1/2 \cdot 2^n$
Therefore $F_n = \Theta(a^n)$ □

(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:
$T(n) = T(n/2) + T(MatrixMultiplication3X3)$
$T(MatrixMultiplication3X3) = 27(Multiplications) + 18(additions)$
$T(MatrixMultiplication3X3) = (O(n^{\log_2 3})) + (O(n))$
$T(MatrixMultiplication3X3) = (O(n^{\log_2 3}))$
$T(n) = T(n/2) + O(n^{\log_2 3})$
Using masters theorem $T(n) = n^{\log_2 3}$
$T(n) = O(n^{\log_2 3})$
$T(n) = O(n^{1.58})$

□

(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: In this case
$T(n) = T(n-1) + T(Addition)$
$T(Addition) = O(n)$ To add 2 (n-1) bit numbers
$T(n) = T(n-1) + O(n)$
$T(n) = O(n^2)$
The time complexity will be more than the previous case which is $O(n^{1.58})$

□

CSCI-GA.1170-001/002 Fundamental Algorithms September 20, 2014

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

Name: Sahil Goel *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: For n=1, at least one step is required to move from disk 1 to disk 3, therefore it is true for n=1
Suppose for $n = k-1$, it takes $2^{k-1} - 1$ steps, then for the last step the minimum steps that would be needed will be
1. Move k-1 smaller discs from peg 1 to peg 2
2. Move largest disc to peg 3
3. Move k-1 smaller discs from peg 2 to peg 3

Total steps will then be equal to
$S(k) = S(k-1) + 1 + S(k-1)$
These are minimum steps for $S(k), fork >= 2$
$S(k) = 2^{k-1} - 1 + 1 + 2^{k-1} - 1$
$S(k) = 2 \cdot 2^{k-1} - 1$
$S(k) = 2^k - 1$
Hence the minimum number of steps required will be $2^k - 1$

□

(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:

```
toh(char src, char dest, char aux, int n)
{
if(n==1)
{
Put(src,aux);
Put(aux,des);
}
else
{
toh(src,dest,aux,n-1);
Put(src,aux);
toh(dest,src,aux,n-1);
Put(aux,dest);
toh(src,dest,aux,n-1);
}
}
```

Total complexity of solution is $T(n) = 3 * T(n-1) + 2$
Dividing the whole equation by 3^n
$T(n)/3^n = T(n-1)/3^{n-1} + 2/3^n$
Let $S(n) = T(n)/3^n$
$S(n) = S(n-1) + 2/3^n$
$S(n) = [1 - (1/3)^n]$
$T(n) = 3^n \cdot [1 - (1/3)^n]$
$T(n) = 3^n - 1$
Therefore $T(n) = O(3^n)$

□

(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 will need two functions in this case
One function moves from A to C using the correct order
The other function moves from C to A using the correct order

```
tohf(char src, char dest, char aux, int n)
{
if(n==1)
{
Put(src,dest);
}
```

```
else
{
toh(src,aux,dest,n-1);
Put(src,dest);
toh(aux,dest,src,n-1);
}
}

toh(char src, char dest, char aux, int n)
{
if(n==1)
{
put(src,aux);
put(aux,dest);
}
else
{
toh(src,dest,aux,n-1);
Put(src,aux);
tohf(dest,src,aux,n-1);
Put(aux,dest);
toh(src,dest,aux,n-1);
}
}
```

Call from main function toh('1','3','2',n)

Tracing the steps

1. call to toh: It moves n-1 rings to 3rd peg and largest disk to 2nd peg, then passes the control to tohf function
2. Now tohf moves n-2 rings from 3rd peg to 2nd peg, (n-1)th largest ring to 1st peg and then (n-2) rings from 2nd peg to 1st peg.
3. Now toh puts the largest ring from 2nd peg to 3rd peg.
4. Toh now puts n-1 rings from 1st peg to 3rd peg using 2nd peg as auxillary.

Now calculating the complexity
Let time for toh be T(n) and time for tohf be T'(n)

$T(n) = 2 * T(n-1) + (T'(n-1) + 2$
Since, $T'(n-1) = 2 * T(n-2) + 1$
$T(n) = 2 * T(n-1) + 2 * T(n-2) + 3$

□

CSCI-GA.1170-001/002 Fundamental Algorithms September 20, 2014

Solutions to Problem 3 of Homework 3 (5 Points)

Name: Sahil Goel *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: In this case, we will proceed as follows
Suppose A is n bit number and B is m bit number where n¿m
1. Initially split A in two parts A_0 : LSB and A_1: MSB of m bits and n-m bits respectively.
2. Call karatsuba multiplication algorithm for A_0 and B
3. Recursively call this procedure with A_1 and B and shift the result by n-m bits and add to result in 2nd step.

nmmultiplication(A,B)
{
$m = bitlengthofB$
$A_0 = A(m-1:0)$ Least significant m bits
$A_1 = A(n-1:m)$ Most significant n-m bits
$temp := karatsuba(A_0, B)$ Both are m bits long
$Prod = Prod + temp + 2^{n-m} \cdot (nmmultiplication(A_1, B))$
}

karatsuba(A,B)
{
m=bit length of A and B
if(m=1)
return A*B;
$A_0 = A((m/2)-1:0)$ and $A_1 = A(m-1:m/2)$
$B_0 = B((m/2)-1:0)$ and $B_1 = B(m-1:m/2)$
$first = karatsuba(A1, B1)$;
$second = karatsuba(A_0, B_0)$;
$third = karatsuba((A1 + A0), (B1 + B0))$;
return $first * 2^{2m} + second + (third - first - second) * 2^m$
}
Now the complexity of the solution can be calculated as follows
Analyzing the nmmultiplication function

$T(n) = T(n-m) + T(Karatsuba)$
$T(n) = T(n-m) + m^{\log_2 3}$
$T(n) = T(n-2m) + 2 * m^{\log_2 3}$
let $n = x * m$
$T(n) = T(n-xm) + x * m^{\log_2 3}$
$T(n) = x * m^{\log_2 3}$

Since x=n/m
Substituting the value for x
$T(n) = \frac{n}{m} * m^{\log_2 3}$
$T(n) = n * m^{log_2 3-1}$
Hence $T(n) = O(nm^{log_2 3-1})$

□

CSCI-GA.1170-001/002 Fundamental Algorithms September 20, 2014

Solutions to Problem 4 of Homework 3 (12 points)

Name: Sahil Goel *Due: Wednesday, September 24*

An array $A[0 \ldots (n-1)]$ is called *rotation-sorted* if there exists some some cyclic shift $0 \leq c < n$ such that $A[i] = B[(i + c \bmod n)]$ for all $0 \leq i < n$, where $B[0 \ldots (n-1)]$ 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 $A[0 \ldots (n/2-1)]$ and $A[n/2 \ldots (n-1)]$ 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: Suppose A is rotationally sorted and at index=k where the cyclic shift starts, then elements are
$A_0, A_1, \cdots, A_k, \cdots, A_n$
Then $A_i > A_{i-1}$ for $i! = k$
and $A(k) > A(k-1)$ and $A(k) > A(k+1)$

Now there are three cases;
1. If $k < n/2$ i.e. if cyclic shift $c > n/2$
then the first half of elements is not sorted since there will be a index k where $A_k > A_{k-1}$ but $A_k > A_{k+1}$. Second half of elements is still sorted because $A_i > A_{i-1}$ for all $i! = k$
Since the first half in this case has only one index k where $A_k > A_{k-1}$, $A_k > A_{k+1}$ and for all other i, $A_i > A_{i-1}$, therefore first half is sorted with cyclic shift $c = (n/2 - k)$

First Half: Rotationally sorted sorted Second half: Sorted

2. if $k >= n/2$ i.e. if cyclic shift $c <= n/2$
Then the second half of elements is not sorted since there will be an index k where $A_k > A_{k-1}$ but $A_k > A_{k+1}$. First half of elements is still sorted because $A_i > A_{i-1}$ for all $i! = k$
Since the second half in this case has only one index k where $A_k > A_{k-1}$, $A_k > A_{k+1}$ and for all other i, $A_i > A_{i-1}$, therefore second half is sorted with cyclic shift $c = (n/2 - k)$

First Half: Sorted Second half: Rotationally Sorted

[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 $A[0]$.

3. if cyclic shift $c = n$ or $c = 0$
Then both the first and second half of arrays are sorted
First Half: Sorted Second half: Sorted

□

(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 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: Minsortedarray(A[ ], low, high)
{
$if(low > high)$
return -1;

else if(low=high-1) % if there are only 2 elements
return minimum(A[low],A[high])

if($A[low] <= A[high]$) % No rotational sorting or c=0
return A[low]

$mid := low + (low + high)/2$

if($A[mid] > A[high]$) %2nd half is rotationally sorted, minimum in 2nd half
Minsortedarray(A, mid+1, high)

else if($A[mid] < A[low]$) %1st half rotationally sorted, minimum in 1st half
Minsortedarray(A, low, mid)

else
return A[mid]

}

Now calculating the complexity of solution,
$T(n) = T(n/2) + 5$
Using masters theorem
$f(n) = n^{\log_b a}$
$f(n) = n^{\log_2 1}$
$f(n) = 1$
$f(n) = O(1)$

$g(n) = 5$
$g(n) = O(1)$

Therefore f(n)=g(n)=O(1)
Hence complexity of algorithm is $T(n) = O(\log_2 n)$
Hence the algorithm is much better than trivial O(n) algorithm

Let us prove the correctness of the algorithm now
Case 1: c=0
A[low] will be less than A[high] and the function will come out of 3rd if equation and return A[low]
Notice that for all other cases A[low] will be greater than A[high] since A[high] will eventually move to before A[low] if $c! = 0$

Case 2: c!=0
Now the function will check which of the 2 halves is already sorted and recursively call the same function with that half of array which is not sorted or rotationally sorted.

Base case when only 2 elements are present
Since the minimum will always lie at $index = n - c$ (for $c! = 0$), and this array is called implies that it has rotationally sorted elements(since only that part of array is recursively called which has rotationally sorted elements) therefore returning the minimum of the two will be the final minimum.

□

CSCI-GA.1170-001/002 Fundamental Algorithms September 20, 2014

Solutions to Problem 5 of Homework 3 (5 Points)

Name: Sahil Goel *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: minmax(A,low,high)
{

if(low=high) 1 element, no comparison needed
{
min=A[low]
max=A[low]
return (min,max)
}

if(low=high-1) 2 elements, only 1 comparison needed
{
if($A[low] > A[high]$)
{
min=A[high];
max=A[low];
}
else
{
min=A[low];
max=A[high];
}
return(min,max);
}

(min1,max1)=minmax($A, low, ((low + high)/2) - 1$)
(min2,max2)=minmax($A, (low + high)/2, high$)

% In this case 2 comparisons would be needed. 1 for global max and 1 for global min
if($min1 < min2$)
min=min1
else
min=min2

if($max1 < max2$)
max=max2
else
max=max1
return(min,max)
}

Now lets calculate the complexity of the solution
T(n)=2*T(n/2) + 2 %Since 2 comparisons are needed when we call recursively
For base case, T(1)=0 and T(2)=1, since 0 and 1 comparisons are needed for 1 and 2 elements respectively
$T(n) = 2 * T(n/2) + 2$
Substituting $n = 2^k$
$T(2^k) = 2 * T(2^{k-1}) + 2$
Substituting $S(k) = T(2^k)$, $S(0) = T(1) = 0$ and $S(1) = T(2) = 1$
$S(k) = 2 * S(k-1) + 2$
Dividing the whole equation by 2^k
$S(k)/2^k = S(k-1)/2^{k-1} + 1/2^{k-1}$
Let $P(k) = S(k)/2^k$, therefore $P(0) = S(0)/2^0 = 0$ and $P(1) = S(1)/2^1 = 1/2$
$P(k) = P(k-1) + 1/2^{k-1}$
$P(k) = P(k-2) + 1/2^{k-2} + 1/2^{k-1}$
$P(k) = P(0) + P(1) + 1/2^1 + \cdots + 1/2^{k-2} + 1/2^{k-1}$
$P(k) = P(0) + P(1) + [1 - (1/2)^{k-1}]$
$P(k) = 0 + 1/2 + [1 - (1/2)^{k-1}]$
Now $S(k) = P(k) * 2^k$
$S(k) = 2^{k-1} + 2^k - 2$
Now $T(2^k) = S(k)$
$T(2^k) = 2^{k-1} + 2^k - 2$
$T(2^k) = 2^{k-1}(3) - 2$
substituting $k = \log n$
$T(n) = 3 * (2^{\log n - 1}) - 2$
$T(n) = \frac{3*(2^{\log n})}{2} - 2$
$T(n) = \frac{3n}{2} - 2$

□