COMP 252 Assignment 1

Christopher Hundt (110220945)

January 21, 2003

Question 1

Ignoring for the moment the *mod* function, we look at what subsequent applications of the rescursive defintion of x_{n+1} would do without it. We start with the first number, which will be called (for now) x_0. Thus,

$$\begin{aligned}
x_1 &= ax_0 + b \\
x_2 &= ax_1 + b = a\left(ax_0 + b\right) + b = a^2x_0 + ab + b \\
x_3 &= ax_2 + b = a\left(a^2x_0 + ab + b\right) + b = a^3x_0 + a^2b + ab + b \\
x_4 &= ax_3 + b = a\left(a^3x_0 + a^2b + ab + b\right) + b = a^4x_0 + a^3b + a^2b + ab + b
\end{aligned}$$

We see that, in general,

$$\begin{aligned}
x_n &= a^nx_0 + a^{n-1}b + a^{n-2}b + \ldots + ab + b \\
&= a^nx_0 + b\left(a^{n-1} + a^{n-2} + \ldots + a + 1\right) \\
&= a^nx_0 + a^{n-1}b\left(1 + \frac{1}{a} + \left(\frac{1}{a}\right)^2 + \ldots + \left(\frac{1}{a}\right)^{n-2} + \left(\frac{1}{a}\right)^{n-1}\right) \\
&= a^nx_0 + a^{n-1}b\sum_{i=0}^{n-1}\left(\frac{1}{a}\right)^i
\end{aligned}$$

First, we handle the two trivial cases for a: 0 and 1. If $a = 0$, then the equation $x_{n+1} = (ax_n + b)\,mod(m)$ becomes $x_{n+1} = b$, and running time is obviously $\Theta(1)$. If $a = 1$, then $\sum_{i=0}^{n-1}\left(\frac{1}{a}\right)^i = \sum_{i=0}^{n-1} 1 = n$, so $x_n = (x + bn)\,mod(m)$, which again has an obvious running time of $\Theta(1)$ in the RAM model. Otherwise, $\frac{1}{a} < 1$, so

$$\begin{aligned}
x_n &= a^nx_0 + a^{n-1}b\sum_{i=0}^{n-1}\left(\frac{1}{a}\right)^i \\
&= a^nx_0 + a^{n-1}b\left(\frac{\left(\frac{1}{a}\right)^n - 1}{\frac{1}{a} - 1}\right) \\
&= a^nx_0 + b\left(\frac{\frac{1}{a} - a^{n-1}}{\frac{1}{a} - 1}\right) \\
&= a^nx_0 + b\left(\frac{1 - a^n}{1 - a}\right)
\end{aligned}$$

The "true" value for x_n is the same expression as in the preceding line, but with three changes: n must be changed to $n-1$ and x_0 to x_1 (because the actual definition starts with x_1 as a given, not x_0). This does not affect running time. Also, the *mod* operation must be performed, but since we are using the RAM model, this can be performed at the end without loss of time. Using fast exponentiation, the time to compute a^n is $\Theta(\log n)$. The few operations remaining are all $\Theta(1)$, so the total running time is $\Theta(\log n)$, and thus $O(\log n)$.

Question 2

We consider the "levels" of the tree. Level 0, the first node, has simply one red node. Blue nodes have no red child nodes, so we may ignore them when considering the number of red nodes. Each red node has 18 red child nodes. So there are 18 red nodes at level 1, 18^2 red nodes at level 2, and in general 18^k red nodes at level k. So the total number of red nodes is

$$R = 1 + 18 + 18^2 + \ldots + 18^k$$

We note from this that $R \leq 18^{k+1}$ and $R \geq 18^k$.

Now each node, whether red or blue, has 19 children, so the number of nodes at each level is clearly 19^k. So the total number of nodes, both red and blue, is

$$n = 1 + 19 + 19^2 + \ldots + 19^k$$

Now we know that $n \leq 19^{k+1}$ and $n \geq 19^k$. Using both these inequatlities, we see the following:

$$\begin{aligned} n &\leq 19^{k+1} & n &\geq 19^k \\ \log_{19} n &\leq k+1 & \log_{19} n &\geq k \\ \log_{19} n - 1 &\leq k \end{aligned}$$

$$\log_{19} n - 1 \leq k \leq \log_{19} n$$

Now, we find that

$$\begin{aligned} R \leq 18^{k+1} &\leq 18^{\log_{19} n + 1} & R \geq 18^k &\geq 18^{\log_{19} n - 1} \\ R &\leq 18 \cdot 18^{\log_{19} n} & R &\geq \frac{1}{18} \cdot 18^{\log_{19} n} \\ R &\leq 18 \cdot n^{\log_{19} 18} & R &\geq \frac{1}{18} \cdot n^{\log_{19} 18} \end{aligned}$$

for all $n \geq 1$. So, clearly, $R \in O(n^{\log_{19} 18})$, and $R \in \Omega(n^{\log_{19} 18})$ (picking 18 and $\frac{1}{18}$ as the constants, respectively, and 1 as the starting n_0). Thus, $R \in \Theta(n^{\log_{19} 18})$.

Question 3

Note: In this algorithm, "A from i to j," means all of the elements from A[i] to A[j], inclusive. Assume the indexing starts at 0 and continues to $n-1$. The starting call is `median(A, B, n)`.

```
median(A, B, n):
  if n=3 then
    C <-- merge(A, B)
    return (C[2] + C[3])/2
  midpoint <-- floor(n/2)
  if n is odd then
    if A[midpoint] > B[midpoint] then
      return median(A from 0 to midpoint+1, B from midpoint-1 to n-1, midpoint+2)
    else
      return median(A from midpoint-1 to n-1, B from 0 to midpoint+1, midpoint+2)
  else
    medianA <-- (A[midpoint] + A[midpoint-1])/2
    medianB <-- (B[midpoint] + b[midpoint-1])/2
    if medianA > medianB then
      return median(A from 0 to midpoint, B from midpoint-1 to n-1, n/2 + 1)
    else
      return median(A from midpoint-1 to n-1, B from 0 to midpoint, n/2 + 1)
```

This algorithm uses the fact that the median of the array C which is formed from the union of A and B is between the median of A and the median of B (inclusive) for any A and B. To prove this, assume that the median of A is less than the median of C. This means that more than half of the elements of B are greater than the median of A. Thus, the median of B is greater than the median of A. And thus, the median of B cannot be less than the median of C, because that would imply that the median of B is less than the median of A. Similar logic proves that the medians of A and B cannot both be greater than the median of C.

Next, we consider that, in a group G of an even number of elements (C is one such group), the median is defined as the average of the two middle elements. There are three ways to change the median of G:

1. Add or remove a number of elements less than the median, without adding or removing (respectively) an equal number of elements greater than the median
2. Add or remove a number of elements greater than the median, without adding or removing (respectively) an equal number of elements less than the median
3. Add or remove one of the two middle elements

When we compare the medians of A and B, we know that anything less than the smaller median or greater than the larger median must be less than or greater than the median of C, respectively. So we can *almost* throw out all those numbers. We can't quite, though, because the median, defined as an average, will depend on the numbers that "border" it. So, from the array with the smaller median, we "throw out" all the numbers less than the number immediately before the median. From the aray with the bigger median, we "throw out" all the numbers greater than the number immediately after the median. (Note that this is assuming ascending sort order.)

Using this strategy, we eventually get the arrays down to a size of 3, which can be combined with a simple merge, and then the middle two elements read and used to find the final median.

Through each iteration, this algorithm reduces the size of the portion of A and B being considered from n to $\lceil \frac{n}{2} + 1 \rceil$. Operations within each iteration are just comparisons or quick calculations on the length, which take time $\Theta(1)$. The final merge, because the size of the array is guaranteed to be 3 by then, also runs in time $\Theta(1)$. So the running time is

$$T_n = T_{\lceil \frac{n}{2}+1 \rceil} + \Theta(1)$$

Thus, by the Master Theorem, the total running time is $O(\log n)$.

Question 4

```
POINTERS:
head1: points to the head of list one
head2: points to the head of list two
p1, p2: to be used later

ALGORITHM:
p1 <-- head1
p2 <-- head2
while p1.next is not null and p2.next is not null do
  p1 <-- p1.next
  p2 <-- p2.next
if p1.next is null then
  p1 <-- head2
  while p2.next is not null do
    p1 <-- p1.next
    p2 <-- p2.next
  p2 <-- head1
  while p2.next is not null do
    if p2 and p1 point to same object then return that object
    else
      p1 <-- p1.next
      p2 <-- p2.next
else (p2.next is null)
  p2 <-- head1
  while p1.next is not null do
    p1 <-- p1.next
    p2 <-- p2.next
  p1 <-- head2
  while p1.next is not null do
    if p2 and p1 point to same object then return that object
    else
      p1 <-- p1.next
      p2 <-- p2.next
```

The basic steps of this algorithm (we will call the longer list l_1, for simplicity):

1. Starting at the heads, advance through both lists simulataneously until you hit the end of l_2.
2. Move the pointer from the end of l_2 (we'll call that pointer p_2) to the head of l_1, and advance both pointers until the one farther along l_1 (the pointer p_1) hits the end of l_1. p_2 has now advanced along l_1 a number of steps equal to the difference between r and k.
3. Put p_1 at the head of l_2. Both pointers are now at equal distance from y_1.
4. Check if both pointers point to the same thing (which would be y_1). If they don't, advance them both, and then repeat this step until you have found y_1.

This algorithm works by "offsetting" the pointers in such a way as to ensure that they will both hit y_1 simultaneously if they are advanced at equal rates. In the case that $r = k$, or the lists are the same length, this algorithm will not offset the pointers at all, which is the proper action.

This is a linear algorithm. There are no recursive calls and there are no nested loops. One loop goes until the end of the shorter list is reached, and then the next loop continues until the end of the longer list is reached, and then the third loop goes from the beginning of the lists to where they meet. Each loop executes fewer than n times. So the running time is clearly $O(n)$.