COMP 251: Data Structures and Algorithms

Solution for Assignment #1

1 Slippery induction

The claim doesn't hold for $n = 3$. There are two places where the reasoning is false.

1. "The claim is obviously true for one line…". If you have one line in the plane, there is *no* point in common. Thus, the induction fails for the base case.
2. "But since any lines sharing two points in common are the same line…", "the common point of the first n lines and the last n lines must be the same common point." Notice that the first n lines is not the same set of lines as the last n lines, and hence the argument is invalid.

2 Converting to base b

Note: When writing down an algorithm, it is a good practice to state what the inputs are, and what the desired output is.

```
ConvertBase($m, b$)
    input: integer $m$, integer $b$ as the base
    output: array $A$ of integers such that $b = \sum_{j=1}^{\lfloor \log_b m \rfloor + 1} A[j] \cdot b^{j-1}$
1   $i \leftarrow 1$
2   $n \leftarrow m$
3   while $n \geq 0$
4        do $A[i] \leftarrow n \bmod b$
5           $n \leftarrow n/b$
6           $i \leftarrow i + 1$
7   return $A$
```

Proof. Loop invariant for this algorithm would be as follows:

$$m = n \cdot b^{i-1} + \sum_{j=1}^{i-1} A[j] \cdot b^{j-1} \tag{1}$$

1. *Initialization* Before we start the loop, we have $i = 1$ and $n = m$. Thus, the loop invariant holds.

2. *Maintenance* Suppose the loop invariant held before the previous iteration. During the last iteration, however, we do: $A[i] = n \bmod b$ and $n = \lfloor \frac{n}{b} \rfloor$. So, we have

$$\begin{aligned} m &= n \cdot b^{i-1} + \sum_{j=1}^{i-1} A[j] \cdot b^{j-1} \\ &= (\lfloor \frac{n}{b} \rfloor \cdot b + n \bmod b) \cdot b^{i-1} + \sum_{j=1}^{i-1} A[j] \cdot b^{j-1} \\ &= \lfloor \frac{n}{b} \rfloor \cdot b^{i} + \sum_{j=1}^{i} A[j] \cdot b^{j-1} \end{aligned}$$

, which is what we wanted.

3. *Termination* Upon termination of the loop, we have $n = 0$. Thus, we have that

$$\begin{aligned} m &= n \cdot b^{i-1} + \sum_{j=1}^{i-1} A[j] \cdot b^{j-1} \\ &= \sum_{j=1}^{i-1} A[j] \cdot b^{j-1} \end{aligned}$$

But what is the value of i after the termination? We are dividing n by b until n runs to zero, and we increment the value of i by 1 at each iteration, so $i = \lfloor \log_b m \rfloor + 2$. Substituting this value into the above summation gives exactly the desired output.

$\square$

**Note:* For this problem, many people missed full marks at the Termination phase. Loop invariant is there to show the *correctness* of the algorithm. The first two phases of the proof(Initializiation and Maintenance) resemble a proof by induction to show that the loop invariant holds. In the Termination phase, however, you should show what the loop invariant implies in terms of the correctness of the algorithm. Here, we wish to compute the base b representation of m. Notice that the number of digits in base b is $\lfloor \log_b m \rfloor + 1$. Since the loop invariant tells us that the array A contains exactly the value we wanted, we know our algorithm is correct.

3 Correctness of Horner's rule

a)

Assuming Real RAM model of computation(ie. binary operations between two real numbers takes constant time), the algorithm given runs in $O(n)$ time.

b)

```
NaivePolynomial($A, x$)
    input: array $A$ of integers, integer $x$
    output: integer $y$
1   $y \leftarrow 0$
2   for $i \leftarrow 0$ to $n$
3         do $t \leftarrow 1$
4            for $j \leftarrow 1$ to $i$
5                 do $t \leftarrow t \cdot x$
6            $y \leftarrow y + A[i] \cdot t$
7   return $y$
```

The inner **for** loop runs for i iterations, where i goes from 0 to n. Thus, the total number of iterations is $\sum_{i=0}^{n} i = O(n^2)$, and hence the running time of NaivePolynomial is $O(n^2)$.

c)

1. *Initialization* Before the loop begins, we have $i = n$ and $y = 0$:

$$\sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k = 0 = y$$

2. *Maintenance* Assuming that the invariant holds at the start of an iteration, we show that it still holds before the next iteration. So, we assume that $y = \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k$. Then,

$$\begin{aligned} y &= a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k \\ &= a_i + x(a_{i+1}x^0 + a_{i+2}x^1 + \cdots + a_n x^{n-(i+1)}) \\ &= a_i x^0 + a_{i+1}x^1 + \cdots + a_n x^{n-i} \\ &= \sum_{k=0}^{n-i} a_{k+i}x^k \end{aligned}$$

3. *Termination* The loop terminates with the value $i = -1$. Substituting this value into the invariant, we have that $y = \sum_{k=0}^{n} a_k x^k$.

d)

As the input, the algorithm takes in the coefficients of the polynomial in array a, and the value of x. Since the value of the polynomial is $\sum_{k=0}^{n} a_k x^k$, and the loop invariant dictates that the algorithm evaluates such value, we know the algorithm is indeed correct.

4 Computer Politics

a)

In this problem, note that the number of candidates is essentially unbounded. Many people made a mistake by converting the data structure into an array that keeps the count of votes each candidate obtained. The key idea was given in the question: *IF* there is a winner, *THEN* at least one of her voters won't be matched. So then the problem reduces to finding an unmatched voter. Note, however, the converse of the statement isn't true. (eg. Consider the votes [Sam, Matt, Holly]. Holly can't be matched, but she is not the winner!) So the algorithm must first see if there is an unmatched candidate, and check if she obtained more than half the votes.

```
FindWinner($A, n$)
    input: array $A$ of candidates, $n$ the size of array $A$
    output: winner's name if there is one, No otherwise
    for $i \leftarrow 1$ to $n$
        do if $Stack$ is empty or $Stack.Top = A[i]$
            then $Stack.Push(A[i])$
            else $Stack.Pop$
    if $Stack$ is empty
        then return No
    $Winner \leftarrow Stack.Top$
    $Count \leftarrow 0$
    for $i \leftarrow 1$ to $n$
        do if $A[i] = Winner$
            then $Count \leftarrow Count + 1$
    if $Count \geq \lfloor \frac{n}{2} \rfloor + 1$
        then return $Winner$
        else return No
```

Note: Here, I used a stack to keep track of the potential winner. This follows from the hint / explanation I gave you after the class on Jan 17th. ("Put the votes into a bucket and maintain all the unmatched votes as you go through... etc.") The use of a stack isn't crucial to the algorithm, however, since I'm only storing the same candidate in the stack. Same algorithm can be implemented without using the stack, by simply keeping track of the potential winner in a variable, and also the number of unmatched votes.

b)

For the first loop, the loop invariant for the algorithm is as follows: "The stack contains all the unmatched votes so far". Let's see if this invariant holds throughout the algorithm.

1. *Initialization* Before the loop begins, the stack is empty. Since at this point, we have not seen any votes, and thus there is no unmatched vote.

2. *Maintenance* Assume that the invariant held before ith iteration. During the ith iteration, the algorithm checks if $A[i]$ can be matched with the top of the stack. If they are the same, $A[i]$ is again pushed onto the stack, and if they are different(matched), the top of the stack is popped. Thus, the stack correctly maintains all the unmatched votes we have seen thus far.

3. *Termination* Upon termination of the loop, the stack contains all the unmatched votes in the array. Notice that the stack must contain votes for a single candidate, for otherwise a vote for a different candidate could not have been stored in the first place. Therefore, the first **for** loop finds the only potential winner.

After the first loop, the rest of the algorithm simply checks if the potential winner is truly the winner. This is done trivially in the second **for** loop. The first **for** loop compares each element in the array against the stack top, and the second **for** loop compares the potential winner to each element in the array. The overall running time of FINDWINNER is clearly $O(n)$, with $2n$ comparisons in particular.

5 Recursion Trees

Expand the recursion tree, and it is easy to see that the sum of nodes at each level of the tree is cn. The depth of the tree then depends on the value of α, as the depth of the both ends is $\log_{\frac{1}{\alpha}} n$ and $\log_{\frac{1}{1-\alpha}} n$. Thus, we make a guess that the running time $T(n) = \theta(n \log n)$.

Let's prove the upper bound here. For the base case, we have

$$T(\frac{1}{\alpha}) \leq c_3 \frac{1}{\alpha} \log \frac{1}{\alpha}$$
$$T(\frac{1}{1-\alpha}) \leq c_3 \frac{1}{1-\alpha} \log \frac{1}{1-\alpha}$$

for some constant c_3 such that $c_3 \geq \frac{\alpha T(1/\alpha)}{\log(1/\alpha)}$, and $c_3 \geq \frac{(1-\alpha)T(1/(1-\alpha))}{\log(1/(1-\alpha))}$.

For inductive step, assume, for some constant c_1 and c_2,

$$T(\alpha n) \leq c_1(\alpha n)\log(\alpha n)$$
$$T((1-\alpha)n) \leq c_2((1-\alpha)n)\log((1-\alpha)n)$$

Then we show that $T(n) \leq c_3 n \log n$ for some constant c_3.

$$\begin{aligned}
T(n) &= T(\alpha n) + T((1-\alpha)n) + cn \\
&\leq c_1 \alpha n \log(\alpha n) + c_2 n \log((1-\alpha)n) - c_2 \alpha n \log((1-\alpha)n) + cn \\
&= c_1 \alpha n \log \alpha + c_1 \alpha n \log n + c_2 n \log(1-\alpha) + c_2 n \log n - c_2 \alpha n \log(1-\alpha) - c_2 \alpha n \log n + cn \\
&= (c_1 \alpha \log \alpha + c_2 \log(1-\alpha) - c_2 \alpha \log(1-\alpha) + c)n + (c_1 \alpha + c_2 - c_2 \alpha) n \log n \\
&\leq c_3 n \log n
\end{aligned}$$

for some constant c_3 large enough, so that $c_1\alpha + c_2 - c_2\alpha \leq c_3$, and $c_1 \alpha \log \alpha + c_2 \log(1-\alpha) - c_2 \alpha \log(1-\alpha) + c \ \leq 0$. The lower bound can be shown similarly, by induction.

6 Some practice on recurrence relations

In this section, be careful when you are using the Master Theorem. There are gaps between the three cases, and for the recurrences that do fall into one of the three cases, you need to explicitly state the value of ϵ.(range is OK.)

a)

(By Master Theorem) $a = 4$, $b = 2$, $n^{\log_b a} = n^2$, $f(n) = n$
$n = O(n^{2-\epsilon})$ for $0 < \epsilon \leq 1$.
So, by case 1 of master theorem, $T(n) = \theta(n^2)$.

b)

(By Master Theorem) $a = 2$, $b = 2$, $n^{\log_b a} = n$, $f(n) = \log n$
$\log n = O(n^{1-\epsilon})$ for $0 < \epsilon < 1$.
So, by case 1 of master theorem, $T(n) = \theta(n)$.

c)

(By Master Theorem) $a = 5$, $b = 2$, $n^{\log_b a} = n^{log_2 5} = n^{2.32...}$
Because $n \log n < n^{1+x}$ for some $x > 0$, we have that
$(n \log n)^2 < n^{2+2x} < n^{2.32-\epsilon}$ for $0 < \epsilon < 0.32$.
So, by case 1 of master theorem, $T(n) = \theta(n^{log_2 5})$.

d)

Expand the recursion tree for this recurrence. The depth of the tree is $\log n$, and the sum of nodes for each level is $\frac{n}{\log n}, \frac{n}{\log n-1}, \frac{n}{\log n-2}, \ldots, \frac{n}{\log n-i}$. Thus, we have that

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log n-1} \frac{n}{\log n - i} \\
&= n \sum_{i=1}^{\log n} \frac{1}{i} \\
&= nH(\log n) \\
&\approx n(\log(\log n) + \gamma + O(1/n)) \\
&= \theta(n \log \log n)
\end{aligned}$$

(Recall the approximation of a harmonic series.)

e)

First, we let $n = 2^m$. Then we have $T(2^m) = 2T(2^{m/2}) + m$.
Let $F(m) = T(2^m)$. Then we have

$$F(m) = 2F(m/2) + m$$

Now, apply the Master theorem here.

$$a = 2,\ b = 2,\ m^{log_b a} = m,\ f(m) = m$$

Then, we have $F(m) = \theta(m \log m)$. Substituting back, we get $F(m) = T(2^m) = \theta(m \log m)$, and finally,

$$T(n) = \theta(\log n \log \log n)$$

f)

First, expand the recurrence to see a pattern.

$$\begin{aligned}
T(n) &= T(n-1) + \log n \\
&= T(n-2) + \log(n-1) + \log n \\
&= T(n-3) + \log(n-2) + \log(n-1) + \log n \\
&\vdots \\
&= T(1) + \log(2) + \log(3) + \cdots + \log n \\
&= \log(2 \times 3 \times \cdots \times n) \\
&= \log(n!)
\end{aligned}$$

Now, we make a guess that $T(n) = \theta(n \log n)$. For upper bound,

$$\begin{aligned}
\log n! &= \log(n \times (n-1) \times (n-2) \times \cdots \times 1) \\
&\leq \log(n \times n \times \cdots \times n) \\
&= \log(n^n) = n \log n
\end{aligned}$$

Now, for the lower bound,

$$\begin{aligned}
\log(n!)^2 &= \log(n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 \times \\
&\qquad 1 \times \quad 2 \quad \times \quad 3 \quad \times \cdots \times (n-1) \times n) \\
&= \log(n \times 2(n-1) \times 3(n-2) \times \cdots \times (n-1)2 \times n) \\
&\geq \log(n \times n \times \cdots \times n) = \log(n^n)
\end{aligned}$$

Since $\log(n!)^2 \geq \log(n^n)$, we have $2 \log n! \geq n \log n$, and finally $\log n! = \Omega(n \log n)$.

g) $T(n) = \sqrt{n}T(\sqrt{n}) + n$

Let $n = 2^m$. Then,

$$T(2^m) = 2^{m/2}T(2^{m/2}) + 2^m$$

Let $F(m) = T(2^m)$. Then,

$$F(m) = 2^{m/2}F(m/2) + 2^m$$

Using the substitution method, try expanding the recurrence:

$$\begin{aligned}
F(m) &= 2^{m/2}(2^{m/4}F(m/4) + 2^{m/2}) + 2^m \\
&= 2^{3m/4}F(m/4) + 2 \cdot 2^m \\
&= 2^{7m/8}F(m/8) + 3 \cdot 2^m \\
&\vdots \\
&= 2^{\frac{2^i-1}{2^i}m}F(\frac{m}{2^i}) + i \cdot 2^m \\
&\vdots \\
&= 2^{\frac{2^{\log_2 m}-1}{2^{\log_2 m}}\times m}F(\frac{m}{2^{\log_2 m}}) + \log_2 m \cdot 2^m \\
&= 2^{m-1}F(1) + 2^m \times \log m
\end{aligned}$$

So we claim that $F(m) = \theta(2^m \log m)$. We prove it by induction.
First the upper bound: for the base case, we let $m = 2$, and then $F(m) \leq c \cdot 2^2 \log 2$. For $m = 3$, we have $F(m) \leq c \cdot 2^3 \log 3$.

Assume that $F(m) \leq c \cdot 2^m \log m$ for $2 \leq m \leq 2j - 1$.
Now, for $m = 2j$, we have $F(2j) = 2^j F(j) + 2^{2j}$. Since we know that $F(j) \leq c \cdot 2^j \log j$ from induction hypothesis, we have $F(2j) \leq c \cdot 2^{2j} \log j + 2^{2j}$.
What we wish to find is that $F(2j) \leq c \cdot 2^{2j} \log(2j)$. Let's find the value of c from this inequality. We have that $c \cdot 2^{2j} \log j + 2^{2j} \leq c \cdot 2^{2j} \log 2 + c \cdot 2^{2j} \log j$.(For otherwise, we are done). Cancelling the common terms on both sides, we have $1 \leq c \log 2$. So, by setting $c \geq 1$, we have the upper bound.

The lower bound proof can be done similarly using induction. Hence, we have $F(m) = \theta(2^m \log m)$. Substituting back, we have that $T(n) = \theta(2^{\log_2 n} \log \log_2 n) = \theta(n \log \log n)$.