CSI 503 – Algorithms and Data Structures – Fall 2009

Solutions to Examination I

Question I

Part (a): Each function in $C$ has a domain with 8 elements and maps each element to a value in \{TRUE, FALSE\}. Therefore, $|C| = 2^8$. Every special function in $C$ maps (TRUE, TRUE, TRUE) and (FALSE, FALSE, FALSE) to FALSE. Each of the other 6 triples in the domain can be mapped to either TRUE or FALSE. Therefore, the number of special functions = $2^6$. Hence, the required probability = $2^6/2^8 = 1/4$.

Part (b): Note that $f(n) \leq \sqrt{n^{3/2} + 1}$. Since $1 \leq n^{3/2}$ for all $n \geq 1$, we have, $f(n) \leq \sqrt{2n^{3/2}}$ for all $n \geq 1$; that is, $f(n) \leq \sqrt{2} n^{3/4}$ for all $n \geq 1$. Hence, $f(n) = O(n^{3/4})$.

Also, $f(n) > \sqrt{n^{3/2} - n^{1/2}}$; that is, $f(n) > \sqrt{0.5n^{3/2} + (0.5n^{3/2} - n^{1/2})}$. Now, $(0.5n^{3/2} - n^{1/2}) \geq 0$ when $0.5n \geq 1$ or $n \geq 2$. Therefore, $f(n) > \sqrt{0.5n^{3/2}}$ for all $n \geq 2$; that is, $f(n) > \sqrt{1/2} n^{3/4}$ for all $n \geq 2$. Hence, $f(n) = \Omega(n^{3/4})$.

In other words, $\sqrt{1/2} n^{3/4} \leq f(n) \leq \sqrt{2} n^{3/4}$ for all $n \geq 2$. That is, $f(n) = \Theta(n^{3/4})$.

Part (c): Since $f(n) = n/\ln n$ and $g(n) = n^{0.9} \ln n$, we have $g(n)/f(n) = (\ln n)^2/n^{0.1}$.

When we take the limit of $g(n)/f(n)$ as $n \to \infty$, we have the indeterminate form $\infty/\infty$. Therefore, we need to use L’Hospital’s rule. The derivative of $(\ln n)^2$ is $2 \ln n/n$ and that of $n^{0.1}$ is $0.1n^{-0.9}$. The ratio of these derivatives is $2 \ln n/(0.1n^{0.1})$.

When we take the limit of the above ratio as $n \to \infty$, we again have the indeterminate form $\infty/\infty$. Therefore, we need to use L’Hospital’s rule again. The derivative of $2 \ln n$ is $2/n$ and that of $0.1n^{0.1}$ is $0.01n^{-0.9}$. The ratio of these derivatives is $2/(0.01n^{0.1})$.

Now, the numerator is a constant while the denominator approaches 0 as $n \to \infty$. Hence, the limit is 0 and it follows that $g(n) = o(f(n))$.

---

Question II

Part (a): The statement is true.

Proof: We note that $2^{\log_2 n} = n$. So, $f(n) = 7n + 8$ and $g(n) = n$. Now, $f(n) \leq 7n + 8n = 15n$ for all $n \geq 1$. That is, $g(n) \geq (1/15)f(n)$ for all $n \geq 1$. Hence $g(n) = \Omega(f(n))$.

Part (b): Given $n = 6^r$, $T(1) = 1$ and $T(n) = 5T(n/6) + n$ for all $n \geq 6$. Now,
\begin{align*}
T(n) &= 5T(n/6) + n \\
&= 5[5T(n/6^2) + n/6] + n \\
&= 5^2T(n/6^2) + n(5/6 + 1) \\
&= 5^2[5T(n/6^3) + n/6^2] + n(5/6 + 1) \\
&= 5^3T(n/6^3) + n[(5/6)^2 + 5/6 + 1] \\
&= \cdots \\
&= 5^rT(n/6^r) + n[(5/6)^{r-1} + (5/6)^{r-2} + \ldots + (5/6) + 1] \\
&= 5^rT(n/6^r) + n\sum_{i=0}^{r-1}(5/6)^i.
\end{align*}

In the last equation above, let \( A_1 = 5^rT(n/6^r) \) and \( A_2 = n\sum_{i=0}^{r-1}(5/6)^i \).

At the last step, \( n/6^r = 1 \) and \( T(1) = 1 \). Thus, \( A_1 = 5^rT(1) = 5^r < 6^r = n \).

We can bound the second term \( A_2 \) as follows.

\[
A_2 = n\sum_{i=0}^{r-1}(5/6)^i \\
= n\left[\frac{1 - (5/6)^r}{1 - (5/6)}\right] \\
= 6n\left[1 - (5/6)^r\right] \\
< 6n \quad \text{(since } 5/6 < 1)\).
\]

Thus, \( T(n) < n + 6n = 7n \) for all \( n \geq 1 \). In other words, \( T(n) = O(n) \). \( \blacksquare \)

**Part (c):**

**Part (i):** The function uses zero multiplications for \( n = 1 \). For \( n \geq 2 \), the function uses two additional multiplications after invoking itself recursively with the parameter \( (n - 1) \). Therefore, the recurrence is given by \( M(1) = 0 \) and

\[
M(n) = M(n - 1) + 2 \quad \text{for } n \geq 2.
\]

**Part (ii):** We are required to prove that \( M(n) \leq cn \) for all \( n \geq 1 \) and for some \( c > 0 \) by the substitution method.

**Basis:** \( M(1) = 0 \leq c \times 1 \). Thus, we get the constraint \( c \geq 0 \).

**Inductive hypothesis:** Assume that for some \( k \geq 1 \), \( M(k) \leq ck \).

**To prove:** \( M(k + 1) \leq c(k + 1) \).

**Proof:**

\[
M(k + 1) = M(k) + 2 \\
\leq ck + 2 \quad \text{(applying inductive hypothesis)} \\
= c(k + 1) - (c - 2)
\]

Thus, \( M(k + 1) \leq c(k + 1) \) would hold if \( c - 2 \geq 0 \). We can achieve this by choosing \( c = 2 \).

In other words, \( M(n) \leq 2n \) for all \( n \geq 1 \); that is, \( M(n) = O(n) \). \( \blacksquare \)
Question III

Part (a): Comparing the given recurrence with the template for the Master Theorem, we note that 
\(a = 8\), \(b = 3\) and \(f(n) = 2n^2\).

Here, \(\log_b a = \log_3 8 < 2\). Thus, \(f(n) = O(n^{\log_b a + \epsilon})\), where \(\epsilon = 2 - \log_3 8 > 0\). Therefore, We can
use Part III of Master Theorem, provided we can verify the regularity conditions.

Now, \(a f(n/b) = 8 f(n/3) = 8 \times 2(n/3)^2 = \frac{8}{9}(2n^2)\). Hence, the regularity condition holds with 
\(c = 8/9 < 1\).

Therefore, by Part III of Master Theorem, \(T(n) = \Theta(n^2)\).

Part (b): We let \(f(n) = n^2\) and show that for this choice of \(f(n)\), the solution to the given recurrence
is \(T(n) = \Theta(n^2 \log n)\).

Comparing the given recurrence with the template for the Master Theorem, we note that \(a = 4\) and 
\(b = 2\). Hence, \(\log_b a = \log_2 4 = 2\) and therefore, \(n^{\log_b a} = n^2\); that is, \(f(n) = \Theta(n^{\log_b a})\).

So, we can use Part II of the Master Theorem and conclude that \(T(n) = \Theta(n^2 \log n)\).

Question IV:

Part (a):

Divide step: Let \(m = \lfloor (n+1)/2 \rfloor\) denote the index of the middle element of \(P\). Partition \(P\) into two
subarrays \(P_1[1..m]\) and \(P_2[m+1..n]\), each of size at most \(\lceil n/2 \rceil\).

Conquer step: Recursively compute the lengths \(\ell_1\) and \(\ell_2\) of longest monotone subarrays of \(P_1\) and
\(P_2\) respectively. Recursion stops when the size of the subarray is 1; in that case, the length of a longest
monotone subarray is 1.

Combine step: Let \(\ell_3\) denote the length of a longest monotone subarray that straddles the two halves
of \(P\). We can compute \(\ell_3\) as follows.

(1) If \(P[m] > P[m+1]\), then there is no straddling monotone subarray; that is, \(\ell_3 = 0\).

(2) Otherwise (i.e., \(P[m] \leq P[m+1]\)) we do the following.

(i) Find the smallest index \(j\) in \([1..m]\) such that \(P[j..m]\) is monotone. (This can be done by
a simple backward scan of \(P\) from index \(m - 1\) down to 1.)

(ii) Find the largest index \(k\) in \([m+1..n]\) such that \(P[m+1..k]\) is monotone. (This can be
done by a simple forward scan of \(P\) from index \(m+1\) to \(n\).)

(iii) Now, \(\ell_3 = k - j + 1\).

The combine step returns \(\max\{\ell_1, \ell_2, \ell_3\}\) as the answer.
Part (b):

```c
int longest(P, a, b) { // Finds length of longest monotone subarray of P[a..b]
    size = b-a+1; // Size of subarray.
    if (size == 1)
        return 1; // Answer for a subarray of size 1.
    // Subarray size >= 2.
    mid = (a+b)/2; // Divide step.

    len1 = longest(P, a, mid); // Conquer steps.
    len2 = longest(P, mid+1, b);

    // Pseudocode for Combine step.
    if (P[mid] > P[mid+1]) {
        // No straddling monotone subarray. So,
        len3 = 0;
    }
    else { // Straddling monotone subarray possible.
        // First, find how far we can go on the left.
        j = mid; index = j-1;
        while (index >= a) {
            if (P[index] > P[j]) // Can't extend monotone subarray; so, exit loop.
                break;
            else {
                j = index; index--;
            }
        } /* End of while */
        // Next, find how far we can go on the right.
        k = mid+1; index = k+1;
        while (index <= b) {
            if (P[index] < P[k]) // Can't extend monotone subarray; so, exit loop.
                break;
            else {
                k = index; index++;
            }
        } /* End of while */

        len3 = k-j+1;
    }
}
```
return max(len1, len2, len3);
} // End of longest.

Part (c): Let $T(n)$ denote the time used for an array of size $n$.

When the array size is at most 1, the time used by the method is a constant, say $c_1$. When the array size is 2 or more, the divide step (which involves only the computation of the index of the middle element) uses only constant time.

The divide step ensures that the size of each subarray is at most $\lceil n/2 \rceil$. So, the conquer time is at most $2T(\lceil n/2 \rceil)$.

The combine step uses $O(n)$ time since the two while loops scan the array, spending $O(1)$ time per iteration. So, the time for divide and combine steps is $O(n)$, i.e., at most $c_2 n$ for some constant $c_2$.

Therefore, the recurrence is:

$$T(n) \leq 2T(\lceil n/2 \rceil) + c_2 n$$

with the initial condition $T(1) = c_1$, where $c_1$ and $c_2$ are constants. (One can verify that the asymptotic solution to this recurrence is $T(n) = \Theta(n \log n)$.)