Solutions for Homework Assignment #4

Answer to Question 1. Let ℓ and r be such that $1 \le \ell \le r \le n$, so $A[\ell..r]$ is a subarray of A. Define the ***maximum value prefix*** of $A[\ell..r]$ to be the maximum possible value of a subarray of $A[\ell..r]$ that starts at ℓ; i.e., $\max_{\ell \le k \le r} \mathbf{val}(A[\ell..k])$; and the ***maximum value suffix*** of $A[\ell..r]$ to be the maximum possible value of a subarray of $A[\ell..r]$ that ends at r; i.e., $\max_{\ell \le k \le r} \mathbf{val}(A[k..r])$.

Now, suppose A is split into two halves, the left and right half. (If n is odd, one "half" has one more element than the other.) Consider a maximum value subarray $A[i..j]$ of A. There are three possibilities:
(a) $A[i..j]$ is entirely within the left half of A; i.e., $1 \le i \le j \le m$, where $m = (\ell + r)$ **div** 2 (the midpoint of ℓ and r);
(b) $A[i..j]$ is entirely within the right half of A; i.e., $m < i \le j \le n$; or
(c) $A[i..j]$ "straddles" the two halves of A, i.e. $1 \le i \le m < j \le n$.

We can deal with possibilities (a) and (b), by simply applying the algorithm recursively to the first and second half of A. For possibility (c) we note that, in this case, the maximum value subarray consists of a maximum value suffix of the first half of A and a maximum value prefix of the second half of A. (This can be shown easily by contradiction, using a "cut-and-paste" argument.) Therefore, to deal with possibility (c), we construct our recursive algorithm so that it returns not only the maximum value of a subarray, but also the maximum value of a prefix and the maximum value of a suffix. In addition, to be able to compute efficiently the maximum value of a prefix or suffix, we make the algorithm return the value of the entire subarray $A[\ell..r]$.

To express the recursive algorithm conveniently, we write it as operating on an arbitrary subarray $A[\ell..r]$ of A. The call $\textsc{RecMaxVal}(A, \ell, r)$, where $1 \le \ell \le r \le n$, returns a tuple (v, p, s, b), where

- v is the maximum value of a subarray of $A[\ell..r]$;
- p is the maximum value of a prefix of $A[\ell..r]$;
- s is the maximum value of a suffix of $A[\ell..r]$; and
- t is the total value of $A[\ell..r]$.

$\textsc{RecMaxVal}(A, \ell, r)$
if $\ell = r$ **then return** $(A[\ell], A[\ell], A[\ell], A[\ell])$
else
  $m := (\ell + r)$ **div** 2
  $(v_L, p_L, s_L, t_L) := \textsc{RecMaxVal}(A, \ell, m)$
  $(v_R, p_R, s_R, t_R) := \textsc{RecMaxVal}(A, m + 1, r)$
  $v := \max(v_L, v_R, s_L + p_R)$
  $p := \max(t_L + p_R, p_L)$
  $s := \max(s_L + t_R, s_R)$
  $t := t_L + t_R$
  return (v, p, s, t)

The correctness of this algorithm (i.e., the fact that it returns a tuple where each component has the value specified above) follows by the preceding discussion. The running time of $\textsc{RecMaxVal}(A, \ell, r)$ is described by the following recurrence:

$$T(k) = 2T(k/2) + c,$$

where $k = r - \ell + 1$, i.e., the length of subarray $A[\ell..r]$. This is a divide-and-conquer recurrence with $a = 2$, $b = 2$, and $d = 0$. Since $a > b^d$, by the Master Theorem the running time of $\textsc{RecMaxVal}$ is $T(k) = O(k^{\log_2 2}) = O(k)$.

To determine the maximum value of a subarray of A we simply run the following algorithm:

MaxVal(A)
$(v, p, s, b) :=$ RecMaxVal($A, 1, n$)
return v

The running time of this algorithm is dominated by the call to RecMaxVal, so it is $O(n)$. This is asympotically faster than the $O(n^2)$ straightforward algorithm mentioned in the assignment.

Answer to Question 2. a. The complete line 3 is: "$x :=$ PowerOfTenToBinary($n/2$)". According to the specification of PowerOfTenToBinary, this gives the binary representation of $10^{n/2}$, so that line 4 computes and returns the binary representation of $10^{n/2} \times 10^{n/2} = 10^n$, which is what we want. The running time of the algorithm is described by the following recurrence:

$$T(n) = T(n/2) + cn^{\log_2 3} \tag{1}$$

This is because the computation involves the following steps:
(a) One recursive call to PowerOfTenToBinary on input $n/2$, resulting in the term $T(n/2)$.
(b) A call to FastMult on inputs x and x, where x is the binary representation of $10^{n/2}$. So, the length of x (in bits) is $\lceil \log_2 10^{n/2} \rceil = \frac{n}{2} \log_2 10 \leq kn$, for some constant k. The call to FastMult therefore takes $O\big((kn)^{\log_2 3}\big) = O(n^{\log_2 3})$ time, resulting in the term $cn^{\log_2 3}$.

Equation (1) is the divide-and-conquer recurrence with $a = 1$, $b = 2$, and $d = \log_2 3$. Since $a < b^d$, by the Master Theorem the running time of PowerOfTen(n) is $T(n) = O(n^{\log_2 3})$.

b. The complete line 6 is:

$$x := \textsc{Sum}\Big(\textsc{FastMult}\big(\textsc{DecimalToBinary}(x_L), \textsc{PowerOfTenToBinary}(n/2)\big), \textsc{DecimalToBinary}(x_R)\Big)$$

By the definition of x_L and x_R we have that $x = x_L \cdot 10^{n/2} + x_R$, and the right-hand-side is precisely what is computed above. (Here we are abusing notation slightly by using the decimal strings x_L and x_R for the numbers they represent.) The running time of this algorithm is described by the following recurrence, where n is the number of digits in the input x:

$$T(n) = 2T(n/2) + cn^{\log_2 3} \tag{2}$$

This is because the computation involves the following steps:
(a) Two recursive calls to DecimalToBinary on inputs x_L and x_R that have $n/2$ digits each, resulting in the term $2T(n/2)$.
(b) A call to PowerOfTenToBinary on input $n/2$. By Part (a), this takes $O\big((n/2)^{\log_2 3}\big) = O(n^{\log_2 3})$ time.
(c) A call to FastMult on inputs the binary representations of x_L and $10^{n/2}$. These are binary representations of $n/2$-digit numbers and therefore have length at most $\lceil \log_2 10 \rceil n = kn$ bits, for some constant k. This call therefore takes $O\big((kn)^{\log_2 3}\big) = O(n^{\log_2 3})$ time.
(d) A call to Sum with inputs the binary representations of $x_L \cdot 10^{n/2}$ and x_R. These are binary representations of an n-digit and an $n/2$-digit number, so they have lengths at most $\lceil \log_2 10 \rceil n$ and $\lceil \log_2 10 \rceil n/2$ bits, respectively. Since both inputs to Sum have length at most kn for some constant k, this call takes $O(n)$ time.

The time required for steps (b) and (c) dominates the time required for step (d), yielding recurrence (2). This is a divide-and-conquer recurrence with $a = 2$, $b = 2$, and $d = \log_2 3$. Since $a < b^d$, by the Master Theorem, the running time of the algorithm DecimalToBinary is $T(n) = O(n^d) = O(n^{\log_2 3})$.

Answer to Question 3.

Algorithm. The idea is to use the order statistics D-Select algorithm to find the $n/4$-th smallest, $n/2$-th smallest, and $3n/4$-th smallest integer in A. If one of them appears more than $n/4$ times in A, we return it; otherwise, it is not hard to argue that no integer appears more than $n/4$ times in A, and so we return nil.

In pseudocode we have the following algorithm. We assume that Count(A, x) returns the number of times that x appears in array A; obviously this can be done in $O(n)$ time.

MoreThanAQuarter(A)
1 $x_1 :=$ D-Select$(A, \lceil n/4 \rceil)$
2 $x_2 :=$ D-Select$(A, \lceil n/2 \rceil)$
3 $x_3 :=$ D-Select$(A, \lceil 3n/4 \rceil)$
4 **if** Count$(A, x_1) > n/4$ **then return** x_1
5 **elsif** Count$(A, x_2) > n/4$ **then return** x_2
6 **elsif** Count$(A, x_3) > n/4$ **then return** x_3
7 **else return** nil

Correctness. By the conditions in lines 4-6 and the fact that Count(A, x) returns the number of occurrences of x in A it is obvious that, if the algorithm returns $x \neq$ nil, then x appears in more than $n/4$ positions of A. It remains to show that

if the algorithm returns nil then no element of A appears in more than $n/4$ positions of A. (*)

So suppose that the algorithm returns nil. By the conditions in lines 4-6,

none of x_1, x_2 or x_3 occurs more than $n/4$ times in A. (**)

Let

- X_1 be the set of elements in A that are strictly less than x_1;
- X_2 be the set of elements in A that are strictly between x_1 and x_2;
- X_3 be the set of elements in A that are strictly between x_2 and x_3; and
- X_4 be the set of elements in A that are strictly greater than x_3.

The illustration below may be helpful in visualizing the situation.

sorted A

All occurrences of elements in X_1	x_1	All occurrences of elements in X_2	x_2	All occurrences of elements in X_3	x_3	All occurrences of elements in X_4

By definition, x_1 is the $\lceil n/4 \rceil$-th element of A in sorted order, so an element in X_1 can appear in at most $\lceil n/4 \rceil - 1 < n/4$ positions in A. Therefore, no element in X_1 occurs more than $n/4$ times in A. Again by definition, x_2 is the $\lceil n/2 \rceil$-th element of A in sorted order, so an element in X_2 can appear in at most $\lceil n/2 \rceil - \lceil n/4 \rceil - 1 \leq (n/2 + 1) - n/4 - 1 = n/4$ positions in A. Therefore, no element in X_2 occurs more than $n/4$ times in A. By similar reasoning no element in X_3 or X_4 occurs more than $n/4$ times in A. By $(**)$ and the fact that no element in X_1, X_2, X_3, or X_4 occurs more than $n/4$ times in A, we conclude that no element of A occurs more than $n/4$ times in A, proving $(*)$.

Running time. By the running time of D-Select, each of lines 1-3 requires $O(n)$ time. Since Count(A, x) takes $O(n)$ time, each of lines 4-6 requires $O(n)$ time. So the entire algorithm runs in $O(n)$ time.