Solutions to Homework 4

Debasish Das
EECS Department, Northwestern University
ddas@northwestern.edu

1 Problem 2.23

Definition 1 *Majority element of an array A[1 ... n]: An array is said to have a majority element if more than half of its entries are the same.*

A constraint on this problem is that only equality is defined on the objects of the array. You can check if the array elements are equal but there is no $>$ or $<$ relation defined on the elements
(a)We split the array A into 2 subarrays A_1 and A_2 of half the size. We choose the majority element of A_1 and A_2. After that we do a linear time equality operation to decide whether it is possible to find a majority element. The recurrence therefore is given by

$$T(n) = 2T(\frac{n}{2}) + O(n) \tag{1}$$

The complexity of algorithm comes to O($n \log n$)

```
procedure GetMajorityElement(a[1...n])
Input:  Array a of objects
Output:  Majority element of a
if n = 1:  return a[1]
k = $\lfloor \frac{n}{2} \rfloor$
$elem_{lsub}$ = GetMajorityElement(a[1...k])
$elem_{rsub}$ = GetMajorityElement(a[k+1...n]
if $elem_{lsub}$ = $elem_{rsub}$:
  return $elem_{lsub}$
lcount = GetFrequency(a[1...n],$elem_{lsub}$)
rcount = GetFrequency(a[1...n],$elem_{rsub}$)
if lcount $>$ k+1:
  return $elem_{lsub}$
else if rcount $>$ k+1:
  return $elem_{rsub}$
else return NO-MAJORITY-ELEMENT
```

GetFrequency computes the number of times an element ($elem_{lsub}$ or $elem_{rsub}$) appears in the given array a[1...n]. Two calls to GetFrequency is O(n). After that comparisons are done to validate the existence of majority element. GetFrequency is the linear time equality operation.
(b)Using the proposed divide-and-conquer operation, indeed it is possible to give a linear time algorithm. Idea is to pair up the elements arbitrarily to get $\frac{n}{2}$ pairs. In each pair if the two elements are different we discard both of them. If they are same only one of them is kept. Before we give the algorithm, we have to prove the following lemma

Lemma 1 *After the proposed procedure, there are atmost $\frac{n}{2}$ elements left and if A has a majority element, then remaining elements will have the same majority element*

Proof: Let m be a majority element of A. Frequency of m is greater than $\frac{n}{2}$ where n is the number of elements of A. Since we are forming pairs of 2, there has to be at least one pair where both the elements of the pair are m since otherwise frequency of m in A cannot be greater than $\frac{n}{2}$. At most all pairs can have both the elements of the pair as m. So after the procedure at least one element m will be left or at most $\frac{n}{2}$ will be left where each of them is m. Therefore out of at most $\frac{n}{2}$ elements, one of the element must be m.

Consider arbitrary pairs (p,q) formed by the procedure. There are four possible cases

$$\begin{aligned} p &= q & (2)\\ p &\neq q & (3)\\ p = m &\wedge q \neq m & (4)\\ p \neq m &\wedge q = m & (5)\end{aligned}$$

All pairs (p,q) satisfy one of the 4 equations. For Equations 3-5 majority of m is maintained because while removing one occurence of majority element m we also remove another element. For pairs satisfying Equation 2 we keep p. p may or may not be equal to m but keeping one occurence of p guarantees majority of m. □

Note that the lemma holds only if the array A has a majority element m. Therefore once we apply the algorithm and come up with a majority element, it is mandatory to check if m is indeed a majority element of the array. That can be done by a GetFrequency call followed by a check whether the frequency of m in A[1...n] is greater than $\frac{n}{2}$.

```
procedure GetMajorityElementLinear(a[1...n])
Input:  Array a of objects
Output:  Majority element of a
if n = 2:
  if a[1] = a[2] return a[1]
  else return NO-MAJORITY-ELEMENT
Create a temporary array temp
for i = 1 to n:
  if a[i] = a[i+1]:
    Insert a[i] into temp
  i = i+1
return GetMajorityElementLinear(temp)
```

```
procedure CheckSanity(a[1...n])
Input:  Array a of objects
Output:  Majority element of a
$m$ = GetMajorityElementLinear(a[1...n])
freq = GetFrequency(a[1...n],$m$)
if freq $> \lfloor \frac{n}{2} \rfloor + 1$:
  return m
else return NO-MAJORITY-ELEMENT
```

Recurrence relation for the algorithm is given as follows

$$T(n) = T(\frac{n}{2}) + O(n) \tag{6}$$

as we can see the processing of array in the recursive function is done in O(n) time. Complexity of the function GetMajorityElementLinear is O(n). CheckSanity is also O(n). Therefore the whole algorithm is linear in terms of n.

2 Problem 2.30

(a) $\omega = 3$ or 5 satisfies the equation. $\sum_{i=1}^{6} \omega^i = \sum_{i=1}^{6} i$ (by definition of ω). The summation has a factor of 7 in it which makes the sum modulo 7 0.
(b)

$$M_6(\omega) \cdot X = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 6 & 4 & 5 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 6 & 1 & 6 & 1 & 6 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 5 & 4 & 6 & 2 & 3 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 5 \\ 2 \end{pmatrix} = \begin{pmatrix} 10 \\ 41 \\ 25 \\ 30 \\ 31 \\ 31 \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \\ 4 \\ 2 \\ 3 \\ 3 \end{pmatrix} (mod 7) \tag{7}$$

(c) Inverse of 3 for mod 7 arithmetic is 5. Therefore 3 and 5 are the number and its inverse respectively.

$$M_6^{-1}(\omega) \cdot X = 6 \cdot \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 5 & 4 & 6 & 2 & 3 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 6 & 1 & 6 & 1 & 6 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 3 & 2 & 6 & 4 & 5 \end{pmatrix} \begin{pmatrix} 3 \\ 6 \\ 4 \\ 2 \\ 3 \\ 3 \end{pmatrix} = 6 \cdot \begin{pmatrix} 21 \\ 76 \\ 55 \\ 76 \\ 51 \\ 68 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 5 \\ 2 \end{pmatrix} \tag{8}$$

(d)Let's call $x^2 + x + 1$ as A and $x^3 + 2x + 1$ as B. Computing the polynomials A and B on the given points $\omega^i : i \in (1..6)$, we obtain a and b as follows (note that -1 is 6 in mod 7 arithmetic)

$$a = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}^T$$
$$b = \begin{pmatrix} 6 & 2 & 0 & 1 & 0 & 0 \end{pmatrix}^T$$

To compute the transform we multiply a and b with $M_6(w)$ as shown in part (b)

$$T_a = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 6 & 4 & 5 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 6 & 1 & 6 & 1 & 6 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 5 & 4 & 6 & 2 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \\ 0 \\ 1 \\ 0 \\ 3 \end{pmatrix}$$

$$T_b = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 6 & 4 & 5 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 6 & 1 & 6 & 1 & 6 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 5 & 4 & 6 & 2 & 3 \end{pmatrix} \begin{pmatrix} 6 \\ 2 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 4 \\ 3 \\ 1 \\ 1 \end{pmatrix}$$

The product of T_a and T_b gives the transform for the product of polynomials A and B. Let's call the product polynomial C and transform of C as T_c.

$$T_c = \begin{pmatrix} 3 \cdot 2 \\ 6 \cdot 4 \\ 0 \cdot 4 \\ 1 \cdot 3 \\ 0 \cdot 1 \\ 3 \cdot 1 \end{pmatrix} = \begin{pmatrix} 6 \\ 3 \\ 0 \\ 3 \\ 0 \\ 3 \end{pmatrix} \tag{9}$$

Coefficients of the polynomial C, C_{coeff} will be obtained by performing an inverse transform on T_c as shown in part (c) of this problem

$$C_{coeff} = 6 \cdot \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 5 & 4 & 6 & 2 & 3 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 6 & 1 & 6 & 1 & 6 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 3 & 2 & 6 & 4 & 5 \end{pmatrix} \begin{pmatrix} 6 \\ 3 \\ 0 \\ 3 \\ 0 \\ 3 \end{pmatrix} = \begin{pmatrix} 6 \\ 1 \\ 1 \\ 3 \\ 1 \\ 1 \end{pmatrix} \tag{10}$$

Transforming 6 to -1 in mod 7 arithmetic we get the product polynomial C as

$$1 \cdot x^5 + 1 \cdot x^4 + 3 \cdot x^3 + 1 \cdot x^2 + 1 \cdot x^1 + (-1) \cdot x^0 \tag{11}$$

3 Problem 2.31

This problem presents a fast algorithm for computing the greatest common divisor (gcd) of two positive integers a and b. Before giving the algorithm we need to prove the following properties

$$gcd(a,b) = \begin{cases} 2 \cdot gcd(\frac{a}{2}, \frac{b}{2}) & \text{if a,b are even} \\ gcd(a, \frac{b}{2}) & \text{if a is odd, b is even} \\ gcd(\frac{a-b}{2}, b) & \text{if a, b are odd} \end{cases}$$

(a)If a and b are even numbers, 2 is surely a common divisor of a and b. Therefore the greatest common divisor will be 2 times the gcd of numbers $\frac{a}{2}$ and $\frac{b}{2}$. If a is odd and b is even, we know for sure that b is divisible by 2 while a is not. Therefore gcd(a,b) remains same as the gcd of a and $\frac{b}{2}$. The third property follows from the fact that if a and b are odd, then (a-b) will be even. Since gcd(a,b) = gcd(a-b,b) and a-b is even now we can apply the second property to get the desired result.
(b) The recursive algorithm for gcd is given as

```
procedure gcd(a,b)
Input:  Two n-bit integers a,b
Output:  GCD of a and b
if a = b:
  return a
else if (a is even $\wedge$ b is even):
  return 2 $\cdot$ gcd($\frac{a}{2}$, $\frac{b}{2}$)
else if (a is odd $\wedge$ b is even):
  return gcd(a,$\frac{b}{2}$)
else if (a is odd $\wedge$ b is odd $\wedge$ a $>$ b):
  return gcd($\frac{a-b}{2}$,b)
else if (a is odd $\wedge$ b is odd $\wedge$ a $<$ b):
  return gcd(a,$\frac{b-a}{2}$)
```

(c)Complexity analysis of the algorithm: Assume that a and b are n-bit numbers. Size of a and b is $2n$ bits. Out of 4 if conditions, every one except the case when a is odd and b is even, decreases the size of a and b to $2n-2$ bits whereas that case decreases it to $2n-1$ bits. Each of the operations is constant time operation as we are dividing or multiplying the numbers by 2. For two cases subtraction of two n-bit numbers are involved which is c·n where n is the number of bits of the operand. Therefore in the worst case the recurrence is given by

$$\begin{aligned}
T(2n) &= T(2n-1) + cn \\
T(2n-1) &= T(2n-2) + cn \\
T(2n-2) &= T(2n-3) + c(n-1) \text{ both operands are n-1 bits} \\
T(2n-3) &= T(2n-4) + c(n-1) \\
&\ldots \\
T(2) &= T(1) + c
\end{aligned}$$

Using substitution, we can obtain T(2n) as

$$T(2n) = 2c \cdot \sum_{i=1}^{n} i \tag{12}$$

which is O(n^2). Compared to the O(n^3) running time of Euclid's the divide-and-conquer algorithm is faster.

4 Problem 4.4

Counterexample is shown in Figure 1. Using the proposed approach we obtain the shortest cycle as {3, 4, 5, 6, 3} but the shortest cycle in this case is {2,3,6,2}. The depth first search labels for each vertex $i : i \in (1..6)$ is $i - 1$.

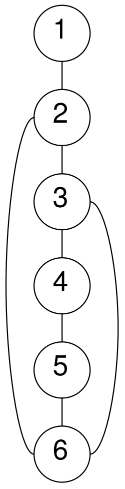

Figure 1: Counterexample

5 Problem 4.7

Given a directed graph G = (V,E) with no constraints on edges along with a specific node s $\in$ V and a tree T = (V,$\hat{E}$) where $\hat{E} \subseteq E$, we have to give a linear time algorithm to check whether T is a shortest path tree for G with starting point S. In this problem the idea is to effectively make use of shortest path distances given on the associated shortest path tree T. Obtain the shortest path distance from each vertex of the tree and annotate the shortest path distance on each vertex of the graph G. Now run subroutine *update* (Bellman-Ford algorithm page 117) on every edge of the graph G. By definition of shortest path distances, *update* should not change any shortest path distance on each node. If *update* changes the shortest path distance (dist) of any vertex, then the given tree is not a shortest path tree. Complexity of the algorithm is O(E).

```
procedure verify-tree(s,T)
Input:  s $\in$ V, T = (V,$\hat{E}$)
Output:  true if T is a shortest path tree
       false otherwise
Q = [s] (queue containing v)
while Q is not empty
  u = eject(Q)
  for each edge (u,v) $\in \hat{E}$:
    dist(v) = dist(u) + l(u,v)
    inject(Q,v)
    E = E-(u,v)
for each edge (u,v) $\in$ E
  temp = dist(v)
  update(e)
  if temp $\neq$ dist(v):
    return false
return true
```

6 Problem 4.10

Given a directed graph with (possibly negative) weighted edges, in which the shortest path between any two vertices is guaranteed to have at most k edgse. We have to give an algorithm to find the shortest path between two vertices u and v in O(k$|E|$) time. The algorithm we present here is a variant of Bellman-Ford algorithm.

```
procedure shortest-paths-k(G,l,s)
Input:  Directed graph G=(V,E):
       edge lengths $\{l_e : e \in E\}$ with no negative cycles:
       vertex u $\in$ V
Output:  For all vertices $v$ reachable from $u$,dist($v$) is
       set to shortest path distance from $u$
for all vertices $v \in V$
  dist(v) = $\infty$
  prev(v) = nil
dist(u) = 0
repeat k times:
  for all e $\in$ E:
    update(e)
```

Complexity Analysis: The algorithm is a modification of Bellman-Ford with shortest path lengths known as k from the problem definition. Complexity of the algorithm is O(k$|E|$).