CSCE 411-200, Fall 2016
Homework 2 Solutions

Problem 1: Here is an attempt at a decrease-and-conquer algorithm to determine if an undirected graph G with n vertices is connected. Assume G is represented using an n-by-n adjacency matrix A, with columns indexed from 1 to n, and rows indexed from 1 to n.

```
Connected(A):
  let n := dim(A)  // dimension of A, i.e., number of rows
  if n = 1 then return true
  else
     let A be sub-matrix of A consisting of rows 1 to n-1 and columns 1 to n-1
     if (Connected(A) = false) then return false
     for j := 1 to n-1 do
         if (A[n,j] = 1) then return true
     endfor
return false
```

Is this algorithm correct? If so, prove it. If not, give a counter-example.
Solution: The idea of the algorithm is to check recursively if the subgraph excluding vertex n is connected. If not, then report that the graph is not connected. If so, then check whether there is any other vertex that is a neighbor of vertex n. If there is then, then report that the graph is connected, otherwise not.

The algorithm is incorrect. The problem is that vertex n might connect two connected components of the subgraph. For a specific counter-example, consider the chain graph $1 - 3 - 2$. The top-level recursive call will consider vertices 1, 2, and 3. The second recursive call will consider vertices and 1 and 2. The third recursive call will consider vertex 1 and will return true. Then the second recursive call will check whether there is an edge from 1 to 2 and will return false. And then the first recursive call will return false.

Problem 2: Problem 31-1 in the textbook about binary gcd.
(a) Prove that if a and b are both even, then $\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$.
(b) Prove that if a is odd and b is even, then $\gcd(a, b) = \gcd(a, b/2)$.
(c) Prove that if a and b are both odd, then $\gcd(a, b) = \gcd((a - b)/2, b)$.
(d) Design an efficient binary gcd alorithm for input integers a and b, $a \geq b$, that runs in $O(\lg a)$ time. Assume each subtraction, parity test and halving takes unit time.
Solution: Let $g = \gcd(a, b)$. That is, $a = xg$ and $b = yg$ where x and y are relatively prime integers.
(a) Suppose a and b are both even. We'll now show that g must be even. Suppose in contradiction g is odd. Since x and y are relatively prime, they cannot both be even. WLOG, suppose x is odd. But then xg, which equals a, would be odd.

Note that $a/2 = xg/2$ and $b/2 = yg/2$. Since g is even, $g/2$ is an integer. Since x and y are relatively prime, $g/2$ is the gcd of $a/2$ and $b/2$. That is, $\gcd(a, b)$ is twice $\gcd(a/2, b/2)$.

Or appeal to Corollary 31.3 (p. 930): $\gcd((a/2)2, (b/2)2) = 2\gcd(a/2, b/2)$.
(b) Suppose a is odd and b is even. Since a is odd, both x and g must be odd. Since b is even and g is odd, y must be even. So $b/2 = (y/2)g$, where $y/2$ is an integer. Since x and y are relatively prime, the same is true of x and $y/2$. Thus g is the gcd of a and $b/2$.
(c) Suppose a and b are both odd. Let $h = \gcd((a - b)/2, b)$. Show that g divides h and h divides g, which will mean that $g = h$.

Show g divides h: Note that $(a - b)/2 = (xg - yg)/2 = g(x - y)/2$. Since a and b are both odd, it follows that g, x, and y are all odd. Thus $x - y$ is even and $(x - y)/2$ is an integer. Thus g divides $(a - b)/2$. Since g divides both $(a - b)/2$ and b, it follows that g divides h.

Show h divides g: Let u and v be integers such that $(a-b)/2 = uh$ and $b = vh$. Then $(a-b)/2 = uh$, which means $a = 2uh + b = 2uh + vh = (2u+v)h$. that is, h divides a. Since h divides both a and b, it follows that h divides g.
(d) Here is the algorithm:

```
Bin-Euclid(a,b)
1. if b == 0 then return a
2. if a and b are even then return 2*Bin-Euclid(a/2,b/2)
3. if a is odd and b is even then return Bin-Euclid(a,b/2)
4. if a is even and b is odd then return Bin-Euclid(a/2,b)
5. return Bin-Euclid((a-b)/2,b)  // a and b both odd
```

Correctness follows from parts (a) - (c). This runs in $O(\lg a)$ time since each recursive call reduces the number of bits in the binary representation of the arguments by at least one (division by 2).
Problem 3: Review the discussion on pp. 1042-1043 about the importance of presorting the arrays in the closest pair algorithm in order to get the good running time. Suppose we don't presort, and instead sort the arrays using mergesort as needed in each recursive call. Write the recurrence for this modified algorithm and solve it.
Solution: The recurrence becomes $T(n) = 2T(n/2) + \Theta(n \log n)$. The master theorem doesn't apply since $n^{\log_2 2}$ is not polynomially smaller than $n \log n$ (see discussion at bottom of p. 95). Instead, we'll use the substitution method to solve the recurrence.
$T(n) = 2T(n/2) + cn \log n$
$= 2[2T(n/2^2) + c(n/2)\log(n/2)] + cn \log n = 2^2 T(n/2^2) + cn \log(n/2) + cn \log n$
$= \ldots$
$= 2^{\log_2 n} T(1) + cn \sum_{j=0}^{\log n - 1} \log(n/2^j)$

Note that $T(1)$ is constant and that there are $\log n$ terms in the sum, each of size at most $\log n$. Thus the recurrence is $O(n \log^2 n)$.

We can also show that the recurrence is $\Omega(n \log^2 n)$, which implies it is $\Theta(n \log^2 n)$. Note that half the terms in the sum are at least $\log(n/2^{\log n/2})$, which equals $\frac{1}{2} \log n$.
Problem 4: The general idea of Strassen's algorithm is to do fewer expensive multiplications by carefully choosing which multiplications to do and then combining them with cheaper additions / subtractions in ways so that some of the work can be reused. This problem asks you to apply the same philosophy to the (simpler) problem of multiplying two complex numbers.

Let i be the square root of -1. The definition of $(a+bi) \cdot (c+di)$ as $(ac-bd) + (ad+bc)i$ leads to a brute force algorithm that does four multiplications of real numbers. You are to devise a way to multiply two complex numbers that uses only three multiplications of real numbers to compute $(acbd)$ and $(ad+bc)$. You can use a few extra additions and subtractions (but dont just substitute multiplication with repeated addition).
Solution:
Step 1: Calculate $a \cdot c$, call the result x_1.
Step 2: Calculate $b \cdot d$, call the result x_2.
Step 3: Calculate $(a+b) \cdot (c+d)$, call the result x_3.
Step 4: Calculate $x_1 - x_2$, which equals $ac - bd$.
Step 5: Calculate $x_3 - x_1 - x_2$, which equals $ad + bc$ (check with algebra).

This requires 3 multiplications and 5 additions and subtractions.
Problem 5: (a) Note that the inverse DFT is defined by Equation (30.11) on p. 913. Use this, together with the definition of the DFT in Equation (30.8) on p. 909, to prove Theorem 30.8 on p. 914.

(b) Exercise 30.2-4. Note that a huge hint for the solution is given at the bottom of p. 913 (just after the proof of Theorem 30.7).

Solution:
(a) Show that for any two vectors a and b of length n, where n is a power of 2, $a \otimes b = DFT_{2n}^{-1}(DFT_{2n}(a) \cdot DFT_{2n}(b))$, where the vectors a and b are padded with 0s to length $2n$ and $\cdot$ denotes the componentwise product of two $2n$-element vectors.

By its definition, $DFT_{2n}(a)$ is the vector $[A(\omega_{2n}^0), A(\omega_{2n}^1), \ldots, A(\omega_{2n}^{2n-1})]$, i.e., the values of the polynomial A, whose coefficients are in the a vector, evaluated at the powers of ω_{2n}. And similarly for $DFT_{2n}(b)$ w.r.t. the polynomial B, whose coefficients are in the b vector.

By the discussion on p. 903, the componentwise product of these two vectors gives $[C(\omega_{2n}^0), C(\omega_{2n}^1), \ldots, C(\omega_{2n}^{2n-1})]$, where $C = A \cdot B$. This equals $DFT_{2n}(c)$, where $c = a \otimes b$ (see p. 901).

By Theorem 30.7, $DFT_{2n}^{-1}(DFT_{2n}(c)) = c$.

(b) Here is pseudocode to compute DFT_n^{-1} in $\Theta(n \lg n)$ time: The input is an n-vector **y** and the output is an n-vector **a** such that $\mathbf{a} = DFT_n^{-1}(\mathbf{y})$.

function Recursive-Inverse-FFT(**y**):
    if $n = 1$ then return **y** endif
    p := Recursive-FFT($[y_0, y_2, y_4, \ldots, y_{n-2}]$)
    q := Recursive-FFT($[y_1, y_3, y_5, \ldots, y_{n-1}]$)
    for $k := 0$ to $n/2 - 1$ do
        $a_k := (p_k + (\omega_n^{-1})^k \cdot q_k)/n$ (line 1)
        $a_{k+n/2} := (p_k - (\omega_n^{-1})^k \cdot q_k)/n$ (line 2)
    endfor
    return **a**

Running time is $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$, the same as the FFT algorithm. The correctness follows from the discussion on page 913.

Problem 6: This problem concerns LUP decompositions of matrices.
(a) Exercise 28.1-3.
(b) Suppose A is any n-by-n boolean matrix that has exactly one 1 in each row and exactly one 1 in each column. Calculate an LUP decomposition of A.
Solution: (a) Solve

$$\begin{bmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 12 \\ 9 \\ 5 \end{bmatrix}$$

by using an LUP decomposition. Let A be the 3-by-3 matrix, x be the vector of unknowns (x's), and b be the vector on the right of the equals sign.

Since we did not cover how to get an LUP decomposition (only how to use one), I used `http://www.gregthatcher.com/Mathematics/LU_Factorization.aspx` to find an LU decomposition of A:

$$\begin{bmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 5 & \frac{17}{10} & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 5 & 4 \\ 0 & -10 & -5 \\ 0 & 0 & -\frac{19}{2} \end{bmatrix}$$

Letting L be the lower-triangular matrix, U be the upper-triangular matrix, and P be the identity matrix (which is a permutation matrix), we have that $P \cdot A = L \cdot U$.

Step 1: Compute $Pb = b'$: Since P is the identity matrix, $b' = b$.

Step 2: Solve $Ly = b'$ for y using forward substitution:
$y_1 = 12$.

$2y_1 + y_2 = 9$ implies $y_2 = -15$.
$5y_1 + (17/10)y_2 + y_3 = 5$ implies $y_3 = -59/2$.

Step 3: Solve $Ux = y$ for x using backward substitution:
$(-19/2)x_3 = -59/2$ implies $x_3 = 59/19$.
$(-10)x_2 + (-5)x_3 = -15$ implies $x_2 = -1/19$.
$x_1 + 5x_2 + 4x_3 = 12$ implies $x_1 = -3/19$.

(b) The constraints on A mean that A is a permutation matrix. The inverse of a permutation matrix is its transpose (entry (i, j) becomes entry (j, i); think about it), which is also a permutation matrix. So set $P = A^T$. Then $P \cdot A$ equals the identity matrix, so we can set both L and U equal to the identity matrix, which is, conveniently for us, both a unit-lower-triangular matrix and an upper triangular matrix.

Problem 7: Exercises 18.2-1 and 18.3-1.

Solution:

18.2-1: Show results of inserting keys F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E in order into an empty B-tree with minimum degree 2. (Abbreviated format; students should draw trees.)
root: FQS
root: Q. Level 1: CFK, S.
root: FQ. Level 1: C, HKL, STV.
root: FQT. Level 1: C, HKL, S, VW.
root: Q. Level 1: FK, T. Level 2: C, H, LMN, RS, VW.
root: Q. Level 1: FKM, T. Level 2: C, H, L, NP, RS, VW.
root: KQ. Level 1: F, M, T. Level 2: ABC, H, L, NP, RS, VWX.
root: KQ. Level 1: F, M, TW. Level 2: ABC, H, L, NP, RS, V, XY.
root: KQ. Level 1: BF, M, TW. Level 2: A, CDE, H, L, NP, RS, V, XYZ.

18.3-1: Show the results of deleting C, P and V in order from the tree of Fig. 18.8(f). Note $t = 3$ here.

Delete C: Start at root, C is not there, determine that AC is the node to descend to, but it has minimum number of keys. Since AC has no sibling that can spare a key, AC merges with its sibling JK, using E in root. Then descend to node ACEJK. Since this is a leaf and C is there, just delete C. Result:
Root: LPTX
Level 1: AEJK, NO, QRS, UV, YZ.

Delete P: Start at root, P is there. Since left child NO has minimum number of keys, we consider right child QRS which has a key to spare. We delete sucessor of P, which is Q, from QRS, and replace P with Q. Rsult:
Root: LQTX
Level 1: AEJK, NO, RS, UV, YZ.

Delete V: Start at root, V is not there, determine that UV is the node to descend to, but it has minimum number of keys. Since UV has no sibling that can spare a key, UV merges with its sibling YZ (could also choose RS), using T in root. Then descend to node UVXYZ. Since this is a leaf and V is there, just delete V. Result:
Root: LQT
Level 1: AEJK, NO, RS, UXYZ

Problem 8: Exercise 29.1-9. Give an example of a linear program for which the feasible region is not bounded, but the optimal objective value is finite.

Solution: The idea is to have a feasible region that is infinite "upward", but the linear program wants to minimize a value. Here's an example:

$$\begin{gathered}\text{Minimize } x_1 + x_2 \\ \text{subject to} \\ x_1 \geq 0\end{gathered}$$

$$x_2 \geq 0.$$

The feasible region is an entire quadrant, which has infinite area, but the optimal solution is the origin.