**CSI 3105 ASSIGNMENT 3 Fall 2011**
**SOLUTIONS**

1. [20 marks]
Suppose you are given an undirected graph G=(V,E) with nodes labelled 1, 2, 3,…,n, and a specific node k in V. You wish to answer the following question: Is node k adjacent to all other nodes in the graph? Discuss CLEARLY what the complexity would be to answer this question for each of the following data structures for G (which were discussed in class). Note that while discussing the complexity, you will need to give some description about HOW you would solve this problem in each case (this description does not have to be detailed pseudocode, but instead should be a clear description in English regarding what part of the data structure you need to scan and why--enough that you can talk about the complexity).

a) Edge List
b) Adjacency Matrix
c) Adjacency List.

Solution:

For all three algorithms, we will use the fact that checking that node k is adjacent to all other nodes in the graph is the same as checking that it has degree n-1.

For the analysis of all three algorithms, we will use the following:

Input size: n the number of nodes, m and number of edges.
Basic operation being counted: "Looks" at the data structure.

**Using the Edge List Data Structure**

Algorithm:

Initialize a variable kNeighbour to 0.

Scan through the edge list, looking for node k. Every time it is found, increment kNeighbour by 1. Stop if kNeighbour reaches value n-1, or we finish scanning the edge list.

Check if kNeighbour has a final value of n-1.

Worst Case Complexity of Algorithm

Input which gives worst case: When node k is the second entry in the last row of the data structure.

Analysis:

This scan goes all the way through the list, and thus looks at all 2m entries in it (m edges listed, each with 2 ends). So the complexity here is $\Theta(m)$.

**Using the Adjacency Matrix Data Structure**

Algorithm:

Scan through row k in the adjacency matrix A, counting the number of 1's that occur. Stop if that count reaches n-1, or if we reach the end of the row. The node k is adjacent to all other nodes if the final count is n-1.

Worst Case Complexity of Algorithm:

Input that gives worst case: k is adjacent to all nodes, and in particular to the node that is in the last column

Analysis: We will need to scan all the way through row k in A. Each row has n entries, so this will be n looks at the data structure. So W(m,n) is $\Theta(n)$ for this algorithm.

**Using the Adjacency List Data Structure**

Algorithm:

Scan through the adjacency list for node k, counting the number of entries found. We have that k is adjacent to all nodes if the final count is n-1.

.

Worst Case Complexity Analysis

Input which gives worst case: When k is adjacent to all other nodes.

Analysis:

If k is adjacent to all other nodes, there will be n-1 looks at the data structure, which is $\Theta(n)$.

2. [10 marks]
Using the algorithm as taught in class, solve the following 0-1 knapsack problem where the set S consists of 4 items. Be sure to show your table, as done in class, and state your final solution. You do not need to show your other work.

W = 9

| Item number | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $w_i$ | 2 | 3 | 4 | 5 |
| $p_i$ | 1 | 4 | 3 | 4 |

Solution:

P array

| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 /N | 1 /I | 1 /I | 1 /I | 1 /I | 1 /I | 1 /I | 1 /I | 1 /I |
| **2** | 0 | 0 /N | 1 /N | 4 /I | 4 /I | 5 /I | 5 /I | 5 /I | 5 /I | 5 /I |
| **3** | 0 | 0 /N | 1 /N | 4 /N | 4 /N | 5 /N | 5 /N | 7 /I | 7 /I | 8 /I |
| **4** | 0 | 0 /N | 1 /N | 4 /N | 4 /N | 5 /N | 5 /N | 7 /N | 8 /I | 8 /I |

The maximum profit is P[4][9] = 8.

To find the items which give this profit, we backtrack in the table.
P[4][9] = I, so we include item 4.
P[3][4] = N, so we do not include item 3.
P[2][4] = I, so we include item 2.
P[1][1] = N, so we do not include item 1.

So pack items 2 and 4.

3. [10 marks total]
Consider the Matrix Chain Multiplication problem, for which we designed a dynamic programming algorithm. Suppose that we found that this algorithm was too painful for words, and so we decided to try to design a greedy algorithm for this problem. Let the matrix chain consist of matrices A1*A2*A3*...*An, where each matrix Ai has dimensions $d_{i-1}$ x $d_i$ (this is the same notation we used in class).

a) [ 5 marks] Consider the following idea for a greedy algorithm:

> First multiply the two matrices whose common dimension $d_i$ is the smallest, and continue in the same way.

Show that this greedy algorithm does not always work by providing an example of a matrix chain problem with not more than 4 matrices for which the greedy algorithm described fails. You must also provide the two solutions, i.e. the one from the greedy algorithm, and the optimal solution, and demonstrate that the algorithm fails for your example by giving the number of multiplications for each of your solutions.

b) [5 marks]
Design a different greedy algorithm for this problem, and describe it clearly. Show that it also fails by providing a counter example, in the same manner as a).

Solution:
3. a) Example to show the greedy idea proposed doesn't work:

Consider the matrix chain problem $A_1 * A_2 * A_3$, where $A_1$ is 3x2, $A_2$ is 2x10, and $A_3$ is 10x1.

The greedy method proposed gives the solution $(A_1 * A_2) * A_3$, which results in 3*2*10 + 3*10*1 = 90 multiplications. However the solution $A_1 * (A_2 * A_3)$ results in 2*10*1 + 3*2*1 = 26 multiplications, which is clearly fewer. Thus the example illustrates that this greedy method does not always find the best solution.

4. [20 marks total]
Suppose you have an array S indexed from 1 to n which contains n numbers, not in any particular order, and you wish to count how many times a given number x occurs in S. Consider the recursive algorithm below for this which finds the number of occurrences of x in the index range i...j in S. Of course, solving the problem would involve an initial call to the algorithm for the range 1..n:

```
int CountOccur (int i,j)
{
int mid, count;
if j < i
  return  0;
else {
   mid = ⌊(j + i)/2⌋
   if S[mid] = x
      Count = 1 + CountOccur(i,mid-1) + CountOccur(mid+1,j);
   else
      Count =  CountOccur(i,mid-1) + CountOccur(mid+1,j);
   return Countt;
   }
}
```

a) [15 marks]
Design a dynamic programming version of the above recursive algorithm and write the pseudocode for your algorithm. Your algorithm must be a dynamic programming version of this algorithm. For this algorithm, you should have a table which has an entry for every index pair i and j, $1 \leq i \leq n$, $1 \leq j \leq n$. For index pair i and j your table should contain the number of times x appears in S in the range i...j. And of course, you must allow for the fact that you will need entries for j<i sometimes too (so you may need to enlarge your table for this—think about this).

b) [3 marks]
Illustrate your algorithm on the following example:
S = [2, 3, 6, 2, 7, 8, 2, 9], x =2.

c) [2 marks]
State the complexity of your algorithm, where you are counting the number of table entries calculated.

Solution:

a)

The table here in this solution has entries for i=1 to n+1, and for j=0 to n. This is one way to make sure that the cases for when mid is 1 or n are handled correctly (there are other ways too--perhaps you found them!). The following pseudocode is a dynamic programming version of the recursive algorithm given. It does, however, take advantage of the fact that whenever j<i in the recursive algorithm, we will always have j=i-1 and so it doesn't bother filling in the other entries for j<i (although doing so would still, of course, be correct!). We first set up the base case values in the table--i.e. we fill in the values for j=i-1 as value 0. For the rest of the entries, we build them up along the diagonal iteratively, similar to the matrix chain algorithm. So we do all entries where j-i = 0, then where j-i = 1, j-i = 2, etc. .

```
void FindCountOccur(int S[1..n], int x)
{
  int C[1..n+1][0..n];
  int diff, mid;
  int i, j;
  for (i=1; i<=n+1; i++){
     j=i-1;
     C[i][j] = 0;
  }
  for (diff=0; diff <= n-1, diff++){
     for (i = 1; i<= n-diff; i++){
       j= i + diff;
       mid = ⌊(i + j)/2⌋ ;
       if (S[mid] = x)
          C[i][j] =  1+ S[i][mid-1] + S[mid+1][j];
       else
          C[i][j] =  S[i][mid-1] + S[mid+1][j];
     }
  }
}
```

b) Note: The rows are indexed by i = 1..9, and the columns by j = 0..8.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 |
| | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 |
| | | 0 | 0 | 1 | 1 | 1 | 2 | 2 |
| | | | 0 | 1 | 1 | 1 | 2 | 2 |
| | | | | 0 | 0 | 0 | 1 | 1 |
| | | | | | 0 | 0 | 1 | 1 |
| | | | | | | 0 | 1 | 1 |
| | | | | | | | 0 | 0 |
| | | | | | | | | 0 |

c)
Input size: n, the number of keys in S.
Basic operation counted: Table entries calculated.
Worst case input: All the same.

Analysis: The table is n+1 by n+1, so there are $\Theta(n^2)$ entries. Hence the algorithm is $\Theta(n^2)$.