CSE 321 - Introduction to Algorithm Design
Homework 04

Deadline: 23:55 December 12th, 2016

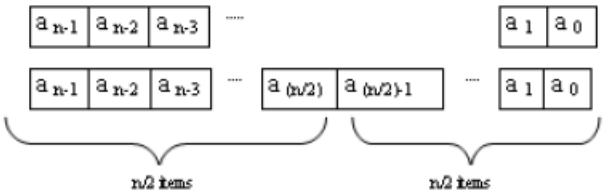
PS: Upload your homework to Moodle website. Do not bring papers to room so that grading homework become faster.

1. Design a divide and conquer algorithm for polynomial evaluation. (For convenience assume that the polynomial is of degree (n-1) and n=2m where m is a positive integer). Calculate the number of additions and multiplications required in your algorithm.

[SOLUTION]

$$P(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x^1 + a_0$$

Polynomial coefficients can be written in an array;



$$P(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_{\frac{n}{2}}x^{\frac{n}{2}} + a_{\frac{n}{2}-1}x^{\frac{n}{2}-1} + ... + a_1x^1 + a_0$$

$$P(x) = (a_{n-1}x^{\frac{n}{2}-1} + a_{n-2}x^{\frac{n}{2}-2} + ... + a_{\frac{n}{2}})x^{\frac{n}{2}} + a_{\frac{n}{2}-1}x^{\frac{n}{2}-1} + ... + a_1x^1 + a_0$$

$$P(x) = P_1(x)x^{\frac{n}{2}} + P_2(x)$$

Let's calculate addition;

$$A(n) = A(\frac{n}{2}) + 1 + A(\frac{n}{2})$$

$$A(n) = 2A(\frac{n}{2}) + 1$$

$$A(n) = \theta(n)$$

For multiplication;

$$M(n) = M(\frac{n}{2}) + \frac{n}{2} + M(\frac{n}{2})$$

$$M(n) = 2M(\frac{n}{2}) + \frac{n}{2}$$

$$M(n) = \theta(n^2)$$

2. Given the array A[1...n] of sorted distinct integers, design a divide and conquer algorithm that finds and index i such that A[i] = i. Your algorithm should run in O(log n) time.

[SOLUTION]

```
Search(A,i,j,key){
    int mid=(i+j)/2;
    if(A[mid]==key && mid==key)  then  return mid;
    else if(A[mid] < key)  then  return Search(A,i,mid-1,key);
    else  then  return -1;
}
```

$$T(n) = 2T(n-1) + 1$$

If we solve the recurrence, $T(n) = O(logn)$

3. Devise a divide and conquer for the Towers of Hanoi problem when moves between peg 1 and peg 3 are not allowed (that is, all moves be either to or from peg 2).

[SOLUTION]

```
ToH(A,C,B,n)
  if(n==1)
    move the disk from A to B
    move the disk from B to C
  else
    ToH(A,C,B,n-1)
    move the largest disk from A to B ToH(C,A,B,n-1)
    move the largest disk from B to C ToH(A,C,B,n-1)
```

4. Write a pseudocode for a divide-and-conquer algorithm for finding values of both the largest and smallest elements in an array of n numbers.

[SOLUTION]

```
Algorithm MinMax(A[l..r], minval, maxval)
//Finds the values of the smallest and largest elements in a given subarray
//Input: A portion of array A[0..n − 1] between indices l and r (l ≤ r)
//Output: The values of the smallest and largest elements in A[l..r]
//assigned to minval and maxval, respectively
if r = l
    minval ← A[l];   maxval ← A[l]
else if r − l = 1
        if A[l] ≤ A[r]
            minval ← A[l];   maxval ← A[r]
        else minval ← A[r];  maxval ← A[l]
else //r − l > 1
    MinMax(A[l..⌊(l + r)/2⌋], minval, maxval)
    MinMax(A[⌊(l + r)/2⌋ + 1..r], minval2, maxval2)
     if minval2 < minval
        minval ← minval2
     if maxval2 > maxval
        maxval ← maxval2
```

5. Chocolate bar puzzle given an n-by-m chocolate bar, you need to break it into nm 1-by-1 pieces. You can break a bar only in a straight line, and only one bar can be broken at a time. Design an algorithm that solves the problem with the minimum number of bar breaks. What is this minimum number? Justify your answer by using properties of a binary tree. Breaking the chocolate bar can be represented by a binary tree.

[SOLUTION]

We can represent operations of any algorithm solving the problem by a full binary tree in which parental nodes represent breakable pieces and leaves represent 1-by-1 pieces of the original bar. The number of the latter is nm; and the number of the former, which is equal to the number of the bar breaks, is one less, i.e., $nm - 1$, according to equation (4.5) in Section 4.4. (Note: This elegant solution was suggested to the author by Simon Berkovich, one of the book's reviewers.)

Alternatively, we can reason as follows: Since only one bar can be broken at a time, any break increases the number of pieces by 1. Hence, $nm - 1$

breaks are needed to get from a single n-by-m piece to nm one-by-one pieces, which is obtained by *any* sequence of $nm - 1$ allowed breaks. (The same argument can be made more formally by mathematical induction.)

6. Questions are given below. Please answer them in detail.

a. What does dynamic programming have in common with divide-and conquer?

a. Both techniques are based on dividing a problem's instance into smaller instances of the same problem.

b. What is a principal difference between the two techniques?

b. Typically, divide-and-conquer divides an instance into smaller instances with no intersection whereas dynamic programming deals with problems in which smaller instances overlap. Consequently, divide-and-conquer algorithms do not explicitly store solutions to smaller instances and dynamic programming algorithms do.

7. *World Series odds:* Consider two teams, *A* and *B*, playing a series of games until one of the teams wins *n* games. Assume that the probability of *A* winning a game is the same for each game and equal to *p* and the probability of *A* losing a game is $q = 1 - p$. (Hence, there are no ties.) Let *P(i, j)* be the probability of *A* winning the series if *A* needs *i* more games to win the series and *B* needs *j* more games to win the series.

a. Set up a recurrence relation for *P(i, j)* that can be used by a dynamic programming algorithm.

a. Let $P(i, j)$ be the probability of A winning the series if A needs i more games to win the series and B needs j more games to win the series. If team A wins the game, which happens with probability p, A will need $i - 1$ more wins to win the series while B will still need j wins. If team A looses the game, which happens with probability $q = 1 - p$, A will still need i wins while B will need $j - 1$ wins to win the series. This leads to the recurrence

$$P(i, j) = pP(i - 1, j) + qP(i, j - 1) \quad \text{for } i, j > 0.$$

The initial conditions follow immediately from the definition of $P(i, j)$:

$$P(0, j) = 1 \quad \text{for } j > 0, \quad P(i, 0) = 0 \quad \text{for } i > 0.$$

b. Find the probability of team *A* winning a seven-game series if the probability of it winning a game is 0.4.

b. Here is the dynamic programming table in question, with its entries rounded-off to two decimal places. (It can be filled either row-by-row, or column-by-column, or diagonal-by-diagonal.)

$i \setminus j$	0	1	2	3	4
0		1	1	1	1
1	0	0.40	0.64	0.78	0.87
2	0	0.16	0.35	0.52	0.66
3	0	0.06	0.18	0.32	0.46
4	0	0.03	0.09	0.18	0.29

Thus, $P[4, 4] \approx 0.29$.

c. Write a pseudocode of the dynamic programming algorithm for solving this problem and determine its time and space efficiencies.

c. **Algorithm** *WorldSeries*(n, p)
//Computes the odds of winning a series of n games
//Input: A number of wins n needed to win the series
// and probability p of one particular team winning a game
//Output: The probability of this team winning the series
$q \leftarrow 1 - p$
for $j \leftarrow 1$ **to** n **do**
  $P[0, j] \leftarrow 1.0$
for $i \leftarrow 1$ **to** n **do**
  $P[i, 0] \leftarrow 0.0$
  for $j \leftarrow 1$ **to** n **do**
    $P[i, j] \leftarrow p * P[i - 1, j] + q * P[i, j - 1]$
return $P[n, n]$

Both the time efficiency and the space efficiency are in $\Theta(n^2)$ because each entry of the $n + 1$-by-$n + 1$ table (except $P[0, 0]$, which is not computed) is computed in $\Theta(1)$ time.

8. Given a binary matrix, find out the maximum size square sub-matrix with all 1s. For example, consider the below binary matrix.

```
0 1 1 0 1
1 1 0 1 0
0 1 1 1 0
1 1 1 1 0
1 1 1 1 1
0 0 0 0 0
```

The maximum square sub-matrix with all set bits is 1.

```
1 1 1
1 1 1
1 1 1
```

Result will be 3 as given example above. Write code in Python with dynamic programming approach. Explain your algorithm in detail. (Grade will be given according to your algorithm and output of your program.)

[SOLUTION]

```
def max_size(mat, ZERO=0):
    nrows, ncols = len(mat), (len(mat[0]) if mat else 0)
    if not (nrows and ncols): return 0 # empty matrix or rows
    counts = [[0]*ncols for _ in xrange(nrows)]
    for i in reversed(xrange(nrows)):     # for each row
        assert len(mat[i]) == ncols # matrix must be rectangular
        for j in reversed(xrange(ncols)): # for each element in the row
            if mat[i][j] != ZERO:
                counts[i][j] = (1 + min(
                    counts[i][j+1],  # east
                    counts[i+1][j],  # south
                    counts[i+1][j+1] # south-east
                    )) if i < (nrows - 1) and j < (ncols - 1) else 1 # edges
    return max(c for rows in counts for c in rows)

arr = [
    [0, 0, 1, 1, 1],
    [0, 0, 1, 1, 1],
    [0, 0, 1, 1, 1],
    [0, 0, 0, 0, 1],
    [0, 0, 0, 1, 1]
]

result = max_size( arr, 0)
print "Result : " + str(result)
```

9. Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that can be solved using dynamic programming. Given a sequence of matrices, the goal is to find the most efficient way to multiply these matrices. The problem is not actually to perform the multiplications, but merely to decide the sequence of the matrix multiplications involved. Here are many options because matrix multiplication is associative. In other words, no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices A, B, C, and D, we would have: ((AB)C)D = ((A(BC))D) = (AB)(CD) = A((BC)D) = A(B(CD)).

However, the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, if A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix, then computing (AB)C needs (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations, while computing A(BC) needs (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Write code in python to find out how many operations are needed in order to solve MCOP with dynamic programming. Explain your algorithm in detail. (Grade will be given according to your algorithm and output of your program.)

[SOLUTION]

```
import sys
```

```
# Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
def MatrixChainOrder(p, n):
        # For simplicity of the program, one extra row and one
        # extra column are allocated in m[][]. 0th row and 0th
        # column of m[][] are not used
        m = [[0 for x in range(n)] for x in range(n)]

        # m[i,j] = Minimum number of scalar multiplications needed
        # to compute the matrix A[i]A[i+1]...A[j] = A[i..j] where
        # dimension of A[i] is p[i-1] x p[i]

        # cost is zero when multiplying one matrix.
        for i in range(1, n):
                m[i][i] = 0

        # L is chain length.
        for L in range(2, n):
                for i in range(1, n-L+1):
                        j = i+L-1
                        m[i][j] = sys.maxint
                        for k in range(i, j):

                                # q = cost/scalar multiplications
                                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
                                if q < m[i][j]:
                                        m[i][j] = q

        return m[1][n-1]

# Driver program to test above function
arr = [1, 2, 3 ,4]
size = len(arr)

print("Minimum number of multiplications is " +
        str(MatrixChainOrder(arr, size)))
```

10. Watch Devrim Arabaları movie and write an essay. (It should be minimum half page A-4 paper size or maximum one page A-4 paper size. Also you are expected to use 1,5 text-space and Times New Roman font family in your essay.)