German University in Cairo
Department of Computer Science
Dr. Nourhan Ehab

CSEN 703 Analysis and Design of Algorithms, Winter Term 2022
Practice Assignment 6

Exercise 6-1

Give examples of problems and/or algorithms where the Divide and Conquer technique is more efficient than Dynamic Programming and state your reasons.

Solution:

Divide and Conquer is efficient for any problem where the sub-problems are independent and unique e.g. Sorting algorithms and Towers Of Hanoi. If we take merge sort as an example of a sorting algorithm and write it using the dynamic programming technique, the algorithm would start with n arrays of size 1. At every iteration the algorithm will merge two consecutive arrays into one and store the results which is a huge overhead in terms of space complexity. In divide and conquer, the algorithm will merge in a local array and copy the results back to the original one thus saving the space occupied by storing the results of sub-problems.If we consider the Towers of Hanoi, the fact the every disk move is unique make is useless to store any partial results.

Exercise 6-2

Consider the factorial algorithm:

i. Write a recurrence for the algorithm.

Solution:

$$T(n) = \begin{cases} T(n-1) + \Theta(1) & \text{if } n \geq 1; \\ \Theta(1) & \text{if } n = 0. \end{cases}$$

ii. Using the dynamic programming paradigm, provide an algorithm based on the given recurrence. Make sure that you use an efficient data structure to store the partial results.

Solution:

```
function FACTORIAL(int $n$)
    fac ← (int)AllocateArray(n+1)
    fac[0] ← 1
    for $i = 1$ to $n$ do
        fac[$i$] ← $i$*fac[$i-1$]
    end for
    return fac[$n$]
end function
```

Exercise 6-3

Consider the following recursive definition for the problem of calculating a given binomial coefficient $\binom{n}{k}$:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n; \\ 1 & \text{if } k = 0\, or k = n. \end{cases}$$

i. Write a divide-and-conquer algorithm to compute the binomial coefficients.

Solution:

```
function BCRec($int\ n$, $int\ k$)
    if $k = 0$ or $k = n$ then
        return 1
    else
        return BCRec($n-1$, $k$) + BCRec($n-1$, $k-1$)
    end if
end function
```

ii. Provide the recurrence relation for your answer in part (i).

Solution:

$$T(n,k) = \begin{cases} T(n-1,k) + T(n-1,k-1) & \text{if } 0 < k < n; \\ \Theta(1) & \text{if } k = 0 \text{ or } k = n. \end{cases}$$

iii. Write an iterative algorithm that employs the dynamic programming technique using the bottom-up approach.

Solution:

```
function BCDP($int\ n$, $int\ k$)
    $bCDPArray \leftarrow$ integerAllocateArray($n+1, k+1$)
    for $i \leftarrow 0$ to $n$ do
        for $j \leftarrow 0$ to $min(i,k)$ do
            if $j = 0$ or $j = i$ then
                $bCDPArray[i][j] \leftarrow 1$
            else
                $bCDPArray[i][j] \leftarrow bCDPArray[i-1][j] + bCDPArray[i-1][j-1]$
            end if
        end for
    end for
    return $bCDPArray[n][k]$
end function
```

iv. Write an algorithm for the same problem using the memoization technique.

Solution:

```
$bCMemArray \leftarrow$ integerAllocateArray($n+1, k+1$)
                                        ▷ $bCMemArray$ is a global array initialized to -$\infty$
function BCMem($int\ n$, $int\ k$)
    if $k = 0$ or $k = n$ then
        $bCMemArray[n][k] \leftarrow 1$
    else if $bCMemArray[n][k] = -\infty$ then
        $bCMemArray[n][k] \leftarrow$ BCMem($n-1, k$)+ BCMem($n-1, k-1$)
    end if
    return $bCMemArray[n][k]$
end function
```

v. Modify your answer in (iii) to use a single one-dimensional array indexed from 0 to k. What was your tradeoff?

Solution:

```
int[] B = new int[k+1];

for ( int i = 0; i <= k; i++ )
```

```
        B[i] = 1;

    for ( int i = 0; i < n; i++ ) {
        int j = min(i,k);

        while ( j > 0 ) {
            B[j] += B[j-1];
            j--;
        }
    }
```

Exercise 6-4

Suppose that you are given an $n \times n$ checkerboard and a checker. You must move the checker from the bottom edge of the board to the top edge of the board according to the following rule. At each step you may move the checker to one of three squares:

a) the square immediately above,
b) the square that is one up and one to the left (but only if the checker is not already in the leftmost column),
c) the square that is one up and one to the right (but only if the checker is not already in the rightmost column).

Each time you move from square x to square y, you receive $p(x, y)$ dollars. You are given $p(x, y)$ for all pairs (x, y) for which a move from x to y is legal. Do not assume that $p(x, y)$ is positive. Give an algorithm that figures out the set of moves that will move the checker from somewhere along the bottom edge to somewhere along the top edge while gathering as many dollars as possible. Your algorithm is free to pick any square along the bottom edge as a starting point and any square along the top edge as a destination in order to maximize the number of dollars gathered along the way. What is the running time of your algorithm?

Solution:

The recursive definition for calculating the score is as follows:

$$\text{score[i,j]} = \begin{cases} 0 & \text{if } i = 1 \text{ and } 1 \le j \le n; \\ \\ max \begin{cases} \text{score[i-1,j-1]} + \text{p((i-1,j-1),(i,j))} & \text{if } j > 1; \\ \text{score[i-1,j]} + \text{p((i-1,j),(i,j))} & \forall j; \\ \text{score[i-1,j+1]} + \text{p((i-1,j+1),(i,j))} & \text{if } j < n; \end{cases} & \text{and } 2 \le i \le n \end{cases}$$

```
function CHECKERBOARDSCORE(int size, matrix p)
    score ← (float)ALLOCATEARRAY(n,n)

    for j ← 1 to n do
        score[1][j] ← 0
    end for
    for i ← 2 to n do

        for j ← 1 to n do
            point x ← new point(i-1,j)

            point y ← new point(i,j)

            temp ← score[i-1][j] + p(x,y)
            if j > 1 then
```

```
                x ← new point(i-1,j-1)

                if  temp <(score[i-1][j-1] + p(x,y)) then
                    temp ← (score[i-1][j-1] + p(x,y))
                end if
            end if
            if  j < n then
                x ← new point(i-1,j+1)

                if  temp < (score[i-1][j+1] + p(x,y)) then
                    temp ← (score[i-1][j+1] + p(x,y))
                end if
            end if
     score[i][j] ← temp
        end for
    end for
     return score
end function
function CheckerMoves(matrix score)
    index max ← GetMax(score[n][1···n])
    Backtrack(score,n,max)
end function
function GetMax(array score)
    index max ← 1
    for  i ← 2 to n do
        if score[i] > score[max] then
            max ← i
        end if
    end for
     return max
end function
function Backtrack(matrix score, index i, index j)
    if i > 1 then
        if  score[i][j] = score[i-1][j] + p((i-1,j),(i,j)) then
            backtrack (score, i-1,j)
        else if  (j>1) and (score[i][j] = score [i-1][j-1] + p((i-1,j-1),(i,j)) )  then
            backtrack (score,i-1,j-1)
        else
            backtrack(score,i-1,j+1)
        end if
        Print (i j)
    end if
end function
```