Computer Science Department
College of Computer and Information Sciences
King Saud University

CSC 311: Design and Analysis of Algorithms[1]

Dr. Waleed Alsalih

7.1 Dynamic programming [Chapter 8]

Dynamic programming is a technique for designing algorithms to solve problems by combining solutions to subproblems (just like divide-and-conquer). In some cases, subproblems overlap with each other, and because of this overlapping among subproblems, using simple recursive algorithms to solve the problem may result in solving the same subproblem several times! Dynamic programming avoids this by solving each subproblem only once and recording the result in a table. While divide-and-conquer is applicable to solve problems where subproblems are independent, dynamic programming is applicable to solve problems where subproblems overlap.
This can be illustrated using the problem of computing Fibonacci numbers (F_n), which can be defined as follows.
$F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$.
A simple recursive algorithm to compute F_n could be described as follows:

```
Fibonacci ($n$)
if $n < 2$ then
    return $n$
else
    return Fibonacci ($n-1$)+Fibonacci ($n-2$);
end
```

However, by tracing the above recursive algorithm, one can easily find that many subproblems are solved again and again. For eaxample, let's consider computing F_5 using the

[1]This is a summary of the material we cover from the textbook: *Introduction to the Design & Analysis of Algorithms*, A. Levitin, Second Edition, Pearson Addison-Wesley, 2006.

recursive algorithm. There will be one call for $n = 5$ and $n = 4$; two calls for $n = 3$; three calls for $n = 2$; five calls for $n = 1$; and three calls for $n = 0$. It would be better if we solve the problem by one call for each value of n. Now let's look at a dynamic programming version of the this algorithm.

```
**Fibonacci** $(n)$
$A[0]$:=0;
$A[1]$:=1;
**for** $i = 2$ **to** $n$ **do**
    $A[i]$:=$A[i-1] + A[i-2]$;
**end**
**return** $A[n]$;
```

Note that while the simple recursive algorithm and divide-and-conquer algorithms in general take a top-down direction, dynamic programming algorithms usually take a bottum-up direction.

Example: computing a binomial coefficient $C(n, k)$.
$C(n, k)$, or $\binom{n}{k}$, can be interpretted as the number of different k-element combinations out of n elements.
$C(n, 0) = C(n, n) = 1$ and $C(n, k) = C(n-1, k) + C(n-1, k-1)$, for $n > k > 0$.

```
**Binomial** $(n, k)$
**for** $i = 0$ **to** $n$ **do**
    **for** $j = 0$ **to** $MIN(i, k)$ **do**
        **if** $j = 0$ **or** $j = i$ **then**
            $C[i, j]$:=1;
        **else**
            $C[i, j]$:=$C[i-1, j-1] + C[i-1, j]$;
        **end**
    **end**
**end**
**return** $C[n, k]$;
```

Time complexity

The basic operation in this algorithm is the addition. We can compute the step count of this operation as follows.

$$A[n,k] = \sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1 = \sum_{i=1}^{k}(i-1) + \sum_{i=k+1}^{n} k = \frac{(k-1)k}{2} + k(n-k) \in \Theta(nk).$$

Computer Science Department
College of Computer and Information Sciences
King Saud University

CSC 311: Design and Analysis of Algorithms[1]

Dr. Waleed Alsalih

7.2 Optimal substructure

Some computational problems maintain the **optimal substructure** which can be defined as follows: a solution to a problem contains (or depends on) solutions to subproblems. Such a property may be used as an indicator for the applicability of dynamic programming. This makes sense because dynamic programming is a bottom-up design technique where we start with a very small subproblem and based on solutions to smaller problems we find solutions to bigger problems.
Example:
Consider the following two problems and tell whether they maintain the optimal substructure property or not. Given an unweighted, directed graph, tell if the optimal substructure property applies to the following problems or not:

1. Finding the shortest path between a pair of nodes u and v.
2. Finding the longest simple path between a pair of nodes u and v.

Answer: It applies to the shortest path problem but does not apply to the longest path problem, why?

A general plan for dynamic programming algorithms:

1. Identify and understand the optimal substructure.
2. Find a way to solve subproblems based on solutions of smaller subproblems.

[1]This is a summary of the material we cover from the textbook: *Introduction to the Design & Analysis of Algorithms*, A. Levitin, Second Edition, Pearson Addison-Wesley, 2006.

Examples on dynamic programming algorithms

Maximum Sum Contiguous Subsequence: Design an algorithm that finds the maximum sum contiguous subsequence in an array of integers $A[0..n-1]$ (the array may have negative integers; otherwise, the answer is trivial).
Let $M[i]$ be the maximum sum contiguous subsequence ending at $A[i]$. To solve the maximum sum contiguous subsequence problem, we will need to find i such that $M[i]$ is maximized. Can we build the array $M[0..n-1]$ in a bottom-up fashion? How?
The algorithm below uses dynamic programming to find the maximum sum contiguous subsequence.

```
Algorithm MaximumSumContiguousSubsequence($A[0..n-1]$)
$M[0]$:=$A[0]$;
$MAX$:=0;
for $i = 1$ to $n-1$ do
    if $M[i-1] > 0$ then
        $M[i]$:=$M[i-1] + A[i]$;
    else
        $M[i]$:=$A[i]$;
    end
    if $M[i] > M[MAX]$ then
        $MAX$:=$i$;
    end
end
return $M[MAX]$;
```

It is obvious that this algorithm runs in $O(n)$ time.

Computer Science Department
College of Computer and Information Sciences
King Saud University

CSC 311: Design and Analysis of Algorithms[1]
Dr. Waleed Alsalih

7.3 Transitive closure [Section 8.2]

The transitive closure of a directed graph is a $|V|$-by-$|V|$ boolean matrix in which the element at the ith row and jth column is set to 1 if and only if there is a directed path of a positive length from the ith vertex to the jth vertex; otherwise, it is set to 0.

The adjacency matrix A and the transitive closure T of the the directed graph in Fig. **??** are:

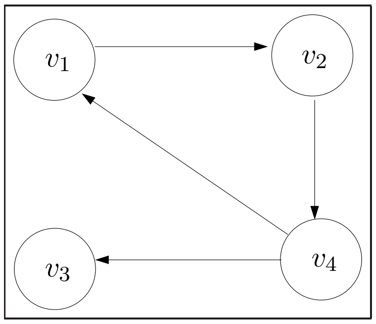

Figure 1: A directed graph.

$$\mathbf{A} = \begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 1 & 0 & 0 \\ v_2 & 0 & 0 & 0 & 1 \\ v_3 & 0 & 0 & 0 & 0 \\ v_4 & 1 & 0 & 1 & 0 \end{array} \qquad \mathbf{T} = \begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ v_1 & 1 & 1 & 1 & 1 \\ v_2 & 1 & 1 & 1 & 1 \\ v_3 & 0 & 0 & 0 & 0 \\ v_4 & 1 & 1 & 1 & 1 \end{array}$$

Warshall's algorithm uses dynamic programming to find the transitive closure through a series of $|V|$-by-$|V|$ metrices: $R^0, R^1, \ldots, R^{|V|}$. Each matrix brings new information based

[1] This is a summary of the material we cover from the textbook: *Introduction to the Design & Analysis of Algorithms*, A. Levitin, Second Edition, Pearson Addison-Wesley, 2006.

on the matrix preceding it. $R^k[i,j] = 1$ if and only if there exists a directed path between the ith vertex and the jth vertex with each intermediate vertex, if any, has a label not higher than k, otherwise, $R^k[i,j] = 0$.
R^k is constructed based on R^{k-1} as follows:
$R^k[i,j] = R^{k-1}[i,j]$ **or** $(R^{k-1}[i,k]$ **and** $R^{k-1}[k,j])$.
The series of matrices generated by Warshall's algorithm are:

$$\mathbf{R^0} = \begin{array}{ccccc} & v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 1 & 0 & 0 \\ v_2 & 0 & 0 & 0 & 1 \\ v_3 & 0 & 0 & 0 & 0 \\ v_4 & 1 & 0 & 1 & 0 \end{array} \qquad \mathbf{R^1} = \begin{array}{ccccc} & v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 1 & 0 & 0 \\ v_2 & 0 & 0 & 0 & 1 \\ v_3 & 0 & 0 & 0 & 0 \\ v_4 & 1 & \mathbf{1} & 1 & 0 \end{array} \qquad \mathbf{R^2} = \begin{array}{ccccc} & v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 1 & 0 & \mathbf{1} \\ v_2 & 0 & 0 & 0 & 1 \\ v_3 & 0 & 0 & 0 & 0 \\ v_4 & 1 & 1 & 1 & \mathbf{1} \end{array}$$

$$\mathbf{R^3} = \begin{array}{ccccc} & v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 1 & 0 & 1 \\ v_2 & 0 & 0 & 0 & 1 \\ v_3 & 0 & 0 & 0 & 0 \\ v_4 & 1 & 1 & 1 & 1 \end{array} \qquad \mathbf{R^4} = \begin{array}{ccccc} & v_1 & v_2 & v_3 & v_4 \\ v_1 & \mathbf{1} & 1 & \mathbf{1} & 1 \\ v_2 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ v_3 & 0 & 0 & 0 & 0 \\ v_4 & 1 & 1 & 1 & 1 \end{array}$$

```
Warshalls ($A[1..|V|, 1..|V|]$)
$R^0$=$A$;
for $k = 1$ to $|V|$ do
    for $i = 1$ to $|V|$ do
        for $j = 1$ to $|V|$ do
            $R^k[i,j] = R^{k-1}[i,j]$ or $(R^{k-1}[i,k]$ and $R^{k-1}[k,j])$;
        end
    end
end
return $R^{|V|}$;
```

Floyd's algorithm

Floyd's algorithm finds all-pairs shortest paths in a weighted directed graph. Floyd's algorithm is very similar to Warshall's algorithm.

```
**Floyds** $(W[1..|V|, 1..|V|])$
$D$=$W$;
**for** $k = 1$ **to** $|V|$ **do**
    **for** $i = 1$ **to** $|V|$ **do**
        **for** $j = 1$ **to** $|V|$ **do**
            $D[i, j]$ =MIN($D[i, j]$,($D[i, k]$ + $D[k, j]$));
        **end**
    **end**
**end**
**return** $D$;
```

Computer Science Department
College of Computer and Information Sciences
King Saud University

CSC 311: Design and Analysis of Algorithms[1]
Dr. Waleed Alsalih

7.4 Exercises on Dynamic programming

Exercise 1:

(a) **2-D random walk**. We have an object that moves in a 2-D plane. At each step, the object can move one unit left, one unit right, one unit up, or one unit down. Describe a dynamic programming algorithm that finds the number of different possible shortest paths to move from the point (0,0) to the point (n,n), where n is a positive integer. Note that if the object is at a point (x,y), it can move to either ($x + 1$,y), ($x - 1$,y), (x,$y + 1$), or (x,$y - 1$).

(b) What is the time complexity of your algorithm?

Exercise 2:
Longest increasing subsequence. Given an array A of distinct integers, find the length of the longest increasing subsequence of elements in A.
Example: The length of the longest increasing subsequence in $[1, 19, 5, 10, 2, 50, 23, 35]$ is 5 which is the length of the subsequence $[1, 5, 10, 23, 35]$.

[1]This is a summary of the material we cover from the textbook: *Introduction to the Design & Analysis of Algorithms*, A. Levitin, Second Edition, Pearson Addison-Wesley, 2006.