DYNAMIC PROGRAMMING. Dynamic programming applies to problems for which the solutions can be thought of sequences of decisions. In addition, the principle of optimality should hold for the problem.

The general solution technique here typically involves the following steps: 1) define a suitable function such that the outputs of interest are specific values of this function; 2) write a recurrence relation for this function; and 3) solve the recurrence relation to get the values of interest – the base cases for the function are usually the inputs.

For the zero-one knapsack problem, we define \( f_i(y) \) to be the optimal profit obtainable from the objects 1 through \( i \) when the capacity constraint is \( y \). A recurrence relation for \( f_i(y) \) takes the form:

\[
f_i(y) = \max \{ f_{i-1}(y), f_{i-1}(y-w_i) + p_i \}.
\]

When the weights are integers, the above recurrence relation can be used to solve the problem in \( O(mn) \) time.

In the all-pairs shortest paths problem, the input is a directed graph \( G(V,E) \). The goal is to find the shortest path from the node \( i \) to node \( j \) for every pair of nodes \( i \) and \( j \) in \( V \). We define the function \( A^k(i,j) \) to be the shortest path length from \( i \) to \( j \) from among all paths (from \( i \) to \( j \) whose intermediate nodes are \( \leq k \). We are interested in the values \( A^n(i,j) \) (for every \( i \) and \( j \) in \( V \), where \( n = |V| \). A recurrence relation for \( A^k(i,j) \) can be written as follows:

\[
A^k(i,j) = \min \{ A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j) \}.
\]

We start with \( A^0 \) and compute \( A^1, A^2, \ldots, A^n \). The total run time is \( O(n^3) \).

We also showed that the string editing problem on two strings of length \( m \) and \( n \) can be solved in time \( O(mn) \) and that the single source shortest path problem on graphs with arbitrary edge weights can be solved in \( O(|V| + |E|) \).

GRAPH SEARCH. There are many ways to conduct tree traversals (such as In-Order, Pre-Order, and Post-Order). These algorithms take linear time. Depth First Search (DFS) and Breadth First Search (BFS) can be used to conduct generic graph searches. These algorithms take \( O(|V| + |E|) \) time.

PARALLEL ALGORITHMS. The model we used was the PRAM (Parallel Random Access Machine). Processors communicate by writing into and reading from memory cells that are accessible to all. Depending on how read and write conflicts are resolved, there are variants of the PRAM. In an Exclusive Read Exclusive Write (EREW) PRAM, concurrent reads are permitted but concurrent writes are prohibited. In a Concurrent Read Concurrent Write (CREW) PRAM, concurrent reads and concurrent writes are allowed. Concurrent writes can be resolved in many ways. In a Common CRCW PRAM, concurrent writes are allowed only if the conflicting processors have the same message to write (into the same cell at the same time). In an Arbitrary CRCW PRAM, an arbitrary processor gets to write in cases of conflicts. In a Priority CRCW PRAM, conflict writes are resolved on the basis of priorities (assigned to the processors at the beginning).

We presented a Common CRCW PRAM algorithm for finding the Boolean AND of \( n \) given bits in \( O(1) \) time. We used \( n \) processors. As a corollary we gave an algorithm for finding the minimum (or maximum) of \( n \) given arbitrary real numbers in \( O(1) \) time using \( n^2 \) Common CRCW PRAM processors.

We also showed that we can find the maximum of \( n \) given arbitrary real numbers in \( \tilde{O}(1) \) time using \( n \) Common CRCW PRAM processors and that we can find the maximum of \( n \) given integers in the range \( [1,n^{O(1)}] \) in \( O(1) \) time using \( n \) Common CRCW PRAM processors.

We also discussed a CREW PRAM algorithm for the prefix computation problem. This algorithm uses \( n \) processors and runs in \( O(\log n) \) time on any input of \( n \) elements. (For the prefix computation problem the input is a sequence of elements from some domain \( \Sigma: k_1, k_2, \ldots, k_n \) and the output is another sequence: \( k_1, k_1 \oplus k_2, \ldots, k_1 \oplus k_2 \oplus k_3 \oplus \cdots \oplus k_n \), where \( \oplus \) is any binary associative and unit-time computable operation on \( \Sigma \).)

We also proved the following theorem: Prefix computation on \( n \) elements can be done using \( \frac{n}{\log n} \) CREW PRAM processors in \( O(\log n) \) time.

INTRACTABLE PROBLEMS. A problem \( \pi_1 \) is said to be polynomially reducible to another problem \( \pi_2 \) (denoted as \( \pi_1 \preceq \pi_2 \)) if the following statement holds: "If \( \pi_2 \) can be solved in deterministic polynomial time then \( \pi_1 \) can also be solved in deterministic polynomial time".

A problem \( \pi \) is said to be \( \mathcal{NP} \)-hard if \( \pi \preceq \pi' \) for every \( \pi' \in \mathcal{NP} \). Equivalently, a problem \( \pi \) is \( \mathcal{NP} \)-hard if \( \pi' \preceq \pi \) where \( \pi' \) is known to be \( \mathcal{NP} \)-hard. A problem \( \pi \) is \( \mathcal{NP} \)-complete if \( \pi \) is in \( \mathcal{NP} \) and \( \pi \) is \( \mathcal{NP} \)-hard.

The following are examples of \( \mathcal{NP} \)-complete problems: SAT, Clique, NodeCover, 3SAT, Subset Sum, and Partition.