Utilization of Algorithms, Dynamic Programming, Optimization

- In the last chapter, we saw that greedy algorithms are efficient solutions to certain optimization problems. However, there are optimization problems for which no greedy algorithm exists. In this chapter, we will examine a more general technique, known as dynamic programming, for solving optimization problems

Making Change

- Suppose we wish to produce a specific value n ∈ N from a given set of coin denominations d1 < d2 < · · · < dk, each of which is a positive integer. Our goal is to achieve a value of exactly n using a minimum number of coins. To ensure that it is always possible to achieve a value of exactly n, we assume that d1 = 1 and that we have as many coins in each denomination as we need.

- d1, . . ., dk: C(n, k) = □ □□ □□ n if k = 1 C(n, k − 1) if k > 1, dk > n min(C(n, k − 1), C(n − dk, k) + 1) if k > 1, n ≥ dk. (12.1) This recurrence gives us a recursive algorithm for computing C(n, k). However, the direct recursive implementation of this recurrence is inefficient. In order to see this, let us consider the special case in which di = i for 1 ≤ i ≤ k and n ≥ k 2 . Then for k > 1, the computation of C(n, k) requires the computation of C(n, k − 1) and C(n − k, k). The computation of C(n−k, k) then requires the computation of C(n−k, k−1). Furthermore, n − k ≥ k 2 − k = k(k − 1) ≥ (k − 1)2 for k ≥ 1. Thus, when n ≥ k 2 , the computation of C(n, k) requires the computation of two values, C(n1, k−1) and C(n2, k−1), where n1 ≥ (k−1)2 and n2 ≥ (k − 1)2 . It is then easily shown by induction on k that C(n, k) requires the computation of 2k−1 values C(ni , 1), where ni ≥ 1 for 1 ≤ i ≤ 2 k−1 . In such cases, the running time is exponential in k. A closer look at the above argument reveals that a large amount of redundant computation is taking place. For example, the subproblem C(n− 2k + 2, k − 2) must be computed twice:

Chained Matrix Multiplication

- Recall that the product AB, where A is a k × m matrix and B is an m × n matrix, is the k × n matrix C such that Cij = Xm l=1 AilBlj for 1 ≤ i ≤ k, 1 ≤ j ≤ n. If we were to compute the matrix product by directly computing each of the kn sums, we would perform a total of kmn scalar multiplications. Now suppose we wish to compute the product, M1M2 · · ·Mn, where Mi is a di−1×di matrix for 1 ≤ i ≤ n. Because matrix multiplication is associative, we have some choice over the order in which the multiplications are performed. For example, to compute M1M2M3, we may either • first compute M1M2, then multiply on the right by M3; or • first compute M2M3, then multiply on the left by M1. In other words, we may compute either (M1M2)M3 or M1(M2M3). Now suppose d0 = 2, d1 = 3, d2 = 4, and d3 = 1.

Chapter Notes

- The mathematical foundation for dynamic programming was given by Bellman [10]. The Change algorithm in Figure 12.1 is due to Wright [115]. The ChainedMatrixMult algorithm in Figure 12.2 is due to Godbole [54]. Floyd's algorithm (Figure 12.3) is due to Floyd [38], but is based on a theorem due to Warshall [110] for computing the transitive closure of a boolean matrix. Because a boolean matrix can be viewed as an adjacency matrix for a directed graph, this is the same as finding the transitive closure of a directed graph