## 16.2 Dynamic Programming

Consider again the recursive function for computing the $n$th Fibonacci number.

```
int Fibr(int n) {
  if (n <= 1) return 1;            // Base case
  return Fibr(n-1) + Fibr(n-2);     // Recursive call
}
```

The cost of this algorithm (in terms of function calls) is the size of the $n$th Fibonocci number itself, which our analysis showed to be exponential (approximately $n^1$.62) Why is this so expensive? It is expensive primarily because two recursive calls are made by the function, and they are largely redundant. That is, each of the two calls is recomputing most of the series, as is each sub-call, and so on. Thus, the smaller values of the function are being recomputed a huge number of times. If we could eliminate this redundancy, the cost would be greatly reduced.

One way to accomplish this goal is to keep a table of values, and first check the table to see if the computation can be avoided. Here is a straightforward example of doing so.

```
int Fibrt(int n, int* Values) {
  // Assume Values has at least n slots, and all
  // slots are initialized to 0
  if (n <= 1) return 1;              // Base case
  if (Values[n] != 0) return Values[n];
  Values[n] = Fibr(n-1, Values) + Fibr(n-2, Values);
  return Values[n];
}
```

This version of the algorithm will not compute a value more than once, so its cost should be linear. Of course, we didn't actually need to use a table. Instead, we could build the value by working from 0 and 1 up to $n$ rather than backwards from $n$ down to 0 and 1. Going up from the bottom we only need to store the previous two values of the function, as is done by our iterative version.

```
long Fibi(int n) {
  long past, prev, curr;
  past = prev = curr = 1;      // curr holds Fib(i)
  for (int i=2; i<=n; i++) {  // Compute next value
    past = prev; prev = curr; // past holds Fib(i-2)
    curr = past + prev;       // prev holds Fib(i-1)
  }
  return curr;
}
```

This issue of recomputing subproblems comes up frequently. In many cases, arbitrary subproblems (or at least a wide variety of subproblems) might need to be recomputed, so that storing subresults in a fixed number of variables will not work. Thus, there are many times where storing a table of subresults can be useful.

This approach to designing an algorithm that works by storing a table of results for subproblems is called dynamic programming. The name is somewhat arcane, because it doesn't bear much obvious similarity to the process that is taking place of storing subproblems in a table. However, it comes originally from the field of dynamic control systems, which got its start before what we think of as computer programming. The act of storing precomputed values in a table for later reuse is referred to as "programming" in that field.

Dynamic programming is a powerful alternative to the standard principle of divide and conquer. In divide and conquer, a problem is split into subproblems, the subproblems are solved (independently), and the recombined into a solution for the problem being solved. Dynamic programming is appropriate whenever the subproblems to be solved are overlapping in some way. Whenever this happens, dynamic programming can be used if we can find a suitable way of doing the necessary bookkeeping. Dynamic programming algorithms are usually not implemented by simply using a table to store subproblems for recursive calls (i.e., going backwards as is done by `Fibrt`). Instead, such algorithms more typically implemented by building the table of subproblems from the bottom up. Thus, `Fibi` is actually closer in spirit to dynamic programming than is `Fibrt` even though it doesn't need the actual table.

### 16.2.1 Knapsack Problem

Knapsack problem: Given an integer capacity $K$ and $n$ items such that item $i$ has integer size $k_i$, find a subset of the $n$ items whose sizes exactly sum to $K$, if possible. Formally: Find $S \subset \{1, 2, ..., n\}$ such that

$$\sum_{i \in S} k_i = K.$$

Example: $K = 163$ 10 items of sizes 4, 9, 15, 19, 27, 44, 54, 68, 73, 101. What if $K$ is 164?

Instead of parameterizing problem just by $n$, parameterize with $n$ and $K$. $P(n, K)$ is the problem with $n$ items and capacity $K$.

Think about divide and conquer (alternatively, induction). What if we know how to solve $P(n-1, K)$? If $P(n-1, K)$ has a solution, then it is a solution for $P(n, K)$. Otherwise, $P(n, K)$ has a solution $\Leftrightarrow P(n-1, K-k_n)$ has a solution.

What if we know how to solve $P(n-1, k)$ for $0 \leq k \leq K$? Cost: $T(n) = 2T(n-1) + c$. $T(n) = \Theta(2^n)$.

BUT... there are only $n(K+1)$ subproblems to solve! Clearly, there are many subproblems being solved repeatedly. Store a $n \times K + 1$ matrix to contain the solutions for all $P(i, k)$. Fill in the rows from $i = 0$ to $n$, left to right.

```
If $P(n-1, K)$ has a solution,
Then $P(n, K)$ has a solution
Else If $P(n-1, K-k_n)$ has a solution
   Then $P(n, K)$ has a solution
   Else $P(n, K)$ has no solution.
```

Cost: $\Theta(nK)$.

**Example 16.1** Knapsack Example: $K = 10$. Five items: 9, 2, 7, 4, 1.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $k_1 = 9$ | $O$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $I$ | $-$ |
| $k_2 = 2$ | $O$ | $-$ | $I$ | $-$ | $-$ | $-$ | $-$ | $-$ | $-$ | $O$ | $-$ |
| $k_3 = 7$ | $O$ | $-$ | $O$ | $-$ | $-$ | $-$ | $-$ | $I$ | $-$ | $I/O$ | $-$ |
| $k_4 = 4$ | $O$ | $-$ | $O$ | $-$ | $I$ | $-$ | $I$ | $O$ | $-$ | $O$ | $-$ |
| $k_5 = 1$ | $O$ | $I$ | $O$ | $I$ | $O$ | $I$ | $O$ | $I/O$ | $I$ | $O$ | $I$ |

Key:

-: No solution for $P(i, k)$.
O: Solution(s) for $P(i, k)$ with $i$ omitted.
I: Solution(s) for $P(i, k)$ with $i$ included.
I/O: Solutions for $P(i, k)$ with $i$ included AND omitted.

Example: $M(3, 9)$ contains O because $P(2, 9)$ has a solution. It contains I because $P(2, 2) = P(2, 9-7)$ has a solution. How can we find a solution to $P(5, 10)$? How can we find ALL solutions to $P(5, 10)$?

### 16.2.2 All-Pairs Shortest Paths

We next consider the problem of finding the shortest distance between all pairs of vertices in the graph, called the **all-pairs shortest-paths** problem. To be precise, for every $u, v \in \mathbf{V}$, calculate d($u$, $v$).

One solution is to run Dijkstra's algorithm $|\mathbf{V}|$ times, each time computing the shortest path from a different start vertex. If $\mathbf{G}$ is sparse (that is, $|\mathbf{E}| = \Theta(|\mathbf{V}|)$) then this is a good solution, because the total cost will be $\Theta(|\mathbf{V}|^2 + |\mathbf{V}||\mathbf{E}| \log |\mathbf{V}|) = \Theta(|\mathbf{V}|^2 \log |\mathbf{V}|)$ for the version of Dijkstra's algorithm based on priority queues.

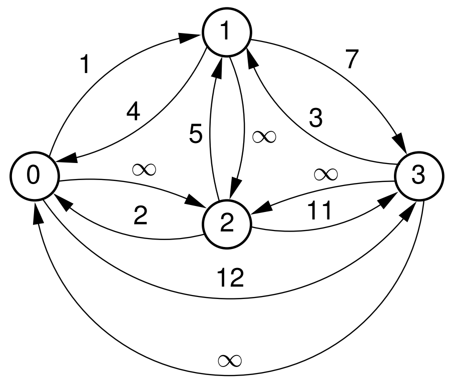

**Figure 16.1** An example of $k$-paths in Floyd's algorithm. Path 1, 3 is a 0-path by definition. Path 3, 0, 2 is not a 0-path, but it is a 1-path (as well as a 2-path, a 3-path, and a 4-path) because the largest intermediate vertex is 0. Path 1, 3, 2 is a 4-path, but not a 3-path because the intermediate vertex is 3. All paths in this graph are 4-paths.

For a dense graph, the priority queue version of Dijkstra's algorithm yields a cost of $\Theta(|\mathbf{V}|^3 \log |\mathbf{V}|)$, but the version using **`MinVertex`** yields a cost of $\Theta(|\mathbf{V}|^3)$.

Another solution that limits processing time to $\Theta(|\mathbf{V}|^3)$ regardless of the number of edges is known as Floyd's algorithm. Define a ***k*-path** from vertex $v$ to vertex $u$ to be any path whose intermediate vertices (aside from $v$ and $u$) all have indices less than $k$. A 0-path is defined to be a direct edge from $v$ to $u$. Figure 16.1 illustrates the concept of $k$-paths.

Define $\mathrm{D}_k(v, u)$ to be the length of the shortest $k$-path from vertex $v$ to vertex $u$. Assume that we already know the shortest $k$-path from $v$ to $u$. The shortest $(k+1)$-path either goes through vertex $k$ or it does not. If it does go through $k$, then the best path is the best $k$-path from $v$ to $k$ followed by the best $k$-path from $k$ to $u$. Otherwise, we should keep the best $k$-path seen before. Floyd's algorithm simply checks all of the possibilities in a triple loop. Here is the implementation for Floyd's algorithm. At the end of the algorithm, array **`D`** stores the all-pairs shortest distances.

```
// Compute all-pairs shortest paths
static void Floyd(Graph G, int[][] D) {
  for (int i=0; i<G.n(); i++) // Initialize D with weights
    for (int j=0; j<G.n(); j++)
      D[i][j] = G.weight(i, j);
  for (int k=0; k<G.n(); k++) // Compute all k paths
    for (int i=0; i<G.n(); i++)
      for (int j=0; j<G.n(); j++)
        if ((D[i][k] != Integer.MAX_VALUE) &&
            (D[k][j] != Integer.MAX_VALUE) &&
            (D[i][j] > (D[i][k] + D[k][j])))
          D[i][j] = D[i][k] + D[k][j];
}
```

Clearly this algorithm requires $\Theta(|\mathbf{V}|^3)$ running time, and it is the best choice for dense graphs because it is (relatively) fast and easy to implement.

## 16.3 Randomized Algorithms

What if we settle for the "approximate best?" Types of guarentees, given that the algorithm produces $X$ and the best is $Y$:

**1.** $X = Y$.

**2.** $X$'s rank is "close to" $Y$'s rank:

$$rank(X) \leq rank(Y) + \text{ "small"}.$$

**3.** $X$ is "usually" $Y$.

$$\mathbf{P}(X = Y) \geq \text{ "large"}.$$

**4.** $X$'s rank is "usually" "close" to $Y$'s rank.

We often give such algorithms names:

**1.** Exact or deterministic algorithm.

**2.** Approximation algorithm.

**3.** Probabilistic algorithm.

**4.** Heuristic.

We can also sacrifice reliability for speed:

**1.** We find the best, "usually" fast.

**2.** We find the best fast, or we don't get an answer at all (but fast).

Choose $m$ elements at random, and pick the best.

- For large $n$, if $m = \log n$, the answer is pretty good.
- Cost is $m - 1$.