Dynamic Programming

When a function is defined, or can be defined, recursively, it is natural to compute it with a recursive algorithm. Sometimes, however, this natural solution is much slower than other methods. **Dynamic programming** is a general method for speeding up such algorithms by eliminating redundant computations.

A conceptually simple way to do this is **memoization**. Whenever you compute the value of a function on particular inputs, *store* the result in case you are asked for the same value again. This can be done automatically, but it usually is more practical to create a **table** for the values of the function on any of the inputs that might occur. Once this table is designed, the values can be filled in *bottom-up* rather than *top-down*.

Let's illustrate this with a simple example.

The Fibonacci Function

Define the function F from $\mathbf{N}$ to $\mathbf{N}$ by the following rules:

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(n) &= F(n-1) + F(n-2) \ \ (\text{if } n \geq 2)
\end{aligned}
$$

The natural recursive algorithm, in Java, looks like:

```
int fib (int n) {
   if (n <= 1) return n;
   return (fib(n-1) + fib(n-2));}
```

But this algorithm is hideously slow (why?). Far better is to store the intermediate values in a table:

```
int fib (int n) {
   int [] table = new int [n+1];
   table[0] = 0;
   table[1] = 1;
   for (int i=2; i <= n; i++)
      table[i] = table[i-1] + table[i-2];
   return table[n];}
```

The Knapsack Problem

Here is another example, a typical **optimization problem** called the **knapsack problem**. We have a set of n objects, where object i has **weight** w_i and **value** v_i. Our input is the list of objects plus a **weight limit**, and our goal is to choose the set of objects of maximum value among those sets that satisfy the weight limit.

For simplicity of presentation, we'll confine our attention here to the special case where $v_i = w_i$ for each i. (This is often called the **subset sum** problem, particularly in the version where we are asked whether the weight limit can be met *exactly* by some subset.)

The knapsack problem does not appear to lend itself easily to divide-and-conquer, and it is easy to construct examples where the obvious greedy algorithm fails. (It's easy to see that the family of sets that fit under the weight limit need not be a matroid.) But there is a reasonable recursive attack on the problem.

Consider a set of n items with weight target t. If a set does *not* include item n, it is a subset of the first $n-1$ items that meets the target t. If the set *does* include item n, then the rest of the set is a subset of the first $n-$ items that meets the target $t - w_n$. We can solve the n-item problem with two calls to the $n-1$-item problem:

```
int knap (int i, int t)
{// returns size of largest subset
 // of first i items that fits in t

if (i == 0) return 0;
if (w[i] > t) return knap(i-1, t);
return max (knap(i-1, t),
    w[i] + knap(i-1, t - w[i]));}
```

As written, this algorithm has a running time given by the recurrence

$$T(n) = 2T(n-1) + O(1),$$

which has solution $T(n) = O(2^n)$, unaceptably slow. But note that the recursive calls to `knap` all have inputs i and j with $i \leq n$ and $j \leq t$.

We can speed up the algorithm considerably by creating a two-dimensional table, $n+1$ by $t+1$, for the values of `knap(i,j)`. We then build the table bottom-up:

```
int knap(int n,int t) {
int [][] table = new int[n+1,t+1];
for (int j=0; j <=t; j++) table[0,j] = 0;
for (int i=1; i <= n; i++)
  for (int k=0; k <= t; k++)
    int useI = 0;
    if (k >= w[i])
      useI = w[i] + table[i-1, k - w[i]];
    table[i,k] = max(table[i-1, t], useI);
return table[n, t];}
```

This method clearly uses time $O(nt)$, since the code inside both loops takes $O(1)$ time. Note that this code, like the recursive version, returns the *size* of the heaviest set that meets the weight limit, rather than the set itself. We can get the set by either (1) recording a bit for each i telling whether item i is in the set being built, or (2) in the dynamic programming version, reconstructing the set from the table.

You may have heard that the knapsack and subset sum problems are **NP**-complete. Our running time is polynomial in n and t – should we tell the Clay Mathematics Institute that we have solved the **P** versus **NP** problem?

Unfortunately, no. We have given a polynomial solution to the **unary subset sum problem**, where the input size is considered to be the sum of the weights and the weight target rather than the *number of bits* needed to represent those numbers in binary. If t were a k-bit number, for example, our $O(nt)$ running time would be $(n2^k)$, exponential in the input size. When we prove later that **binary subset sum** is **NP**-complete, we will see that having big numbers as our weights and weight target is crucial to our proof.

Dynamic Programming in General

When can we use dynamic programming to speed up an exponential-time recursive algorithm?

- The problem must break down into subproblems that have the **optimal substructure property** – solving each subproblem optimally must be sufficient to solve the global problem optimally.
- The subproblems must **overlap** in that different subproblems must require us to solve some of the same sub-subproblems. This allows us to save time by avoiding the redundant work.

Dynamic programming saves time at the cost of additional space. The recursive versions of our Fibonacci and knapsack algorithms used space mostly in the method stack for the recursive calls: $O(n)$ for `fib(n)` and $O(n)$ for `knap(n,t)`. The dynamic programming table for Fibonacci numbers uses the same $O(n)$ space, but the table for knapsack uses $O(nt)$ space, far more than the recursive version.

In CMPSCI 601, we learn that the problem of evaluating a **boolean circuit** is complete for the class **P**, meaning that any polynomial-time algorithm can be expressed as such a circuit. The obvious recursive algorithm to evaluate a circuit takes exponential time, because it is essentially evaluating the boolean **formula** corresponding to the circuit, which is bigger unless the circuit itself is already a tree.

Applying dynamic programming to this recursive algorithm gives us a table with the value of every gate of the circuit. We fill in these values bottom up as the inputs to each gate become known. This is the natural, polynomial-time way to evaluate the circuit (the time can be made linear in the number of gates in the circuit). But the space used is now proportional to the number of gates. In the recursive algorithm, the space used in the method stack was proportional to the **depth** of the circuit.

The Shortest Path Problem

We turn now to a family of problems where some of the most important algorithms use dynamic programming. Given a directed graph with non-negative weights on the edges, the **shortest path problem** is to find the path from one vertex to another that has the smallest total weight. (So we regard the edge weights as distances.) There are many variants of the problem:

- **Single-Pair:** Input weighted graph G and vertices s and t, and output the shortest path from s to t.
- **Single-Source:** Input G and s and output the shortest path from s to *each other vertex*. Oddly, there is no way known to solve the single-pair problem that is asymptotically better than the best solution to the single-source problem. Note that the paths can easily be reconstructed if we just output the *distance* from s to each other vertex. We'll solve this next lecture with **Dijkstra's Algorithm**.
- **Single-Destination:** Input G and t and output the shortest path from each other vertex to t. This is clearly reducible to the single-source problem, by replacing G with a directed graph G^R that has each arrow of G's graph reversed with the same weight.

- **All-Pairs:** Input G, output the shortest path (or shortest distance) from each s to each t. We'll see two algorithms for this problem today.
- **Unit Weight:** If every edge has weight 1, breadth-first search solves the single-source problem in time $O(|E|)$. We find all vertices at distance 1 from s, then all vertices at distance 2, distance 3, and so on. But this approach fails in general if different edges have different weights.
- **Transitive Closure:** A special case of the shortest-path problem comes when we only ask whether a path exists, ignoring the weights. We can ask this in the single-pair, single-source, or all-pairs versions. We know that DFS or BFS solves the single-source version in $O(|E|)$ time.

Min-Plus Matrix Powering

We can describe an n-vertex weighted graph by an n by n matrix, where entry a_{st} represents the weight of the edge, if any, from s to t. If there is no edge, we assign a_{st} to be ∞. For most applications we set a_{ss} to be 0 for each edge, to account for the fact that we may travel from s to s in distance 0, taking no edges.

The desired output for the all-pairs problem can also be represented by an n by n matrix whose entries are non-negative real numbers or ∞. Now d_{st} records the distance from s to t along the shortest path, or ∞ if there is no path at all. (Again, $d_{ss} = 0$ because of the zero-edge path from s to s.) Thus the all-pairs problem can be thought of as transforming the matrix A to the matrix D. We can express this transformation in terms of a kind of matrix multiplication.

The definition of matrix multiplication requires us to "add" and "multiply" matrix entries. We have a valid definition of matrix multiplication whenever the "addition" and "multiplication" satisfy the axioms for a **semiring** – each operation is associative and has an identity element, addition is commutative, and the distributive law holds. Any ring or field is also a semiring, but so are the natural numbers **N** under $+$ and $\times$ and the boolean semiring $\{0, 1\}$ under OR and AND.

Here we define a new structure that we call the **min-plus semiring**. The elements are the non-negative real numbers together with ∞. The "addition" operation is the **minimum** operation, so that "$a + b$" is the smaller of a and b. The "multiplication" operation is real-number addition, with the additional rule that $\infty + x = \infty$ for any x. The "additive identity" is ∞ and the "multiplicative identity" is 0.

It is easy to prove the following by induction for any semiring S:

Path-Matrix Theorem: Let G be a graph with edges weighted by elements of S and let A be the corresponding matrix. Let s and t be vertices of G and let k be any non-negative integer. Then the (s, t) entry of A^k is the "sum", over all paths of exactly k edges from s to t, of the "product" of the weights of the edges on the path.

In the min-plus semiring, the "product" of the weights of the edges on a path is the sum of the weights, or just the total distance along the path. The "sum" over all paths just gives us the length of the shortest path.

Of course we don't know which value of k will cause A^k to have the length of the longest path, but a simple trick will help us. Remember that A_{ss} is always zero. This means that a k-step path also gives rise to many paths with each number of edges greater than k – paths that use the zero-length "edges" at one of the vertices. So A^k_{st} actually gives us the length of the path that is shortest among all paths of length *less than or equal to* k.

Furthermore, we don't have to worry about very long paths. If a path has more than $n-1$ edges, it must revisit a vertex. We can make a shorter path with the same source and destination by deleting the edges between the two visits to the same node. By repeating this process, we can get to a path with at most $n-1$ edges. Since the weights are all non-negative, this path is no longer than the original one.

To conclude, then, the shortest-path-distance matrix D is related to the single-step path matrix A by the rule:

$$d_{st} = A_{st}^{n-1}$$

where the matrix powering uses matrix multiplication over the min-plus semiring.

How long does it take to compute D? Because the Strassen algorithm requires *subtraction* of matrix elements, and the min-plus semiring does not support subtraction, we must use ordinary matrix multiplication. This uses $O(n^3)$ semiring operations, and since we can take a minimum or sum of two real numbers in $O(1)$ steps this is $O(n^3)$ steps in all.

If we calculate A^{n-1} in the most obvious way we will need $n-2$ matrix multiplications for $O(n^4)$ total time. But it's better to use **repeated squaring**, which needs only $O(\log n)$ matrix multiplications and thus $O(n^3 \log n)$ total time.

Note also that in the same time we can compute the transitive closure of the graph, by powering over the boolean semiring. Now the atomic operations are bit operations.

The Floyd-Warshall Algorithm

The min-plus powering algorithm uses dynamic programming in a sense, because we store the answers to subproblems like the entries of the matrix A^i for certain values of i. But we can solve the all-pairs problem a bit more quickly by using dynamic programming in another way.

We define an **intermediate vertex** of a path to be any vertex other than the source or the destination. (Thus paths with zero or one edge have no intermediate vertices at all.) Our subproblems will be defined in terms of intermediate vertices.

Definition: If $0 \leq k \leq n$, $d_{st}^{(k)}$ is the length of the shortest path from s to t that has only intermediate vertices in the set $\{1, \ldots, k\}$. If there is no such path, $d_{st}^{(k)} = \infty$. The matrix $D^{(k)}$ is defined to have entry $d_{st}^{(k)}$ for every vertex s and t.

The matrix $D^{(0)}$ refers to paths with no intermediate vertices at all, which must be single edges (or trivial paths from a vertex to itself). It is thus exactly the same matrix as A. At the other extreme, $D^{(n)}$ is defined in terms of *all* paths, so it is exactly D, the shortest-path distance matrix. The idea of our algorithm will be to work from $D^{(0)}$ through each $D^{(i)}$ until we have computed $D^{(n)}$.

How can we compute $D^{(i)}$ from $D^{(i-1)}$? We must consider how a path could get from s to t while using only intermediate vertices numbered from 1 to i. One possibility is that it never uses vertex i at all. In this case we have already considered the path, and the length of the smallest such path is $d_{st}^{(i-1)}$. If it uses i as an intermediate vertex, then it must consist of a path from s to i, followed by a path from i to t. Each of these paths only has intermediate vertices numbered from 1 through $i-1$ (why?), so the shortest such paths have lengths $d_{si}^{(i-1)}$ and $d_{it}^{(i-1)}$ respectively. Thus:

$$\begin{aligned} d_{st}^{(i)} &= \min(d_{st}^{(i-1)}, \\ &\qquad d_{si}^{(i-1)} + d_{ti}^{(i-1)}) \end{aligned}$$

The computation of each entry takes only $O(1)$ time, so we can compute $D^{(i)}$ from $D^{(i-1)}$ in $O(n^2)$ time. The total time to compute all n matrices is $O(n^3)$, and this suffices to solve the all-pairs shortest path problem.

In the text this algorithm is described as filling in an n by n by $n+1$ table. This is the natural way to use dynamic programming to speed up a recursive algorithm based on the equation for updating an entry. In an actual implementation, though, we would probably want to save space by *reusing* the space for $D^{(i-2)}$, for example, to store $D^{(i)}$. This would make the space usage $O(n^2)$ rather than $O(n^3)$. (Can a similar trick save space in any of the other algorithms in this lecture?)

The boolean version of this algorithm is usually called **Warshall's algorithm** and finds the transitive closure of a boolean matrix (more technically, the reflexive, transitive closure) in $O(n^3)$ bit operations.

Another use of this same strategy comes in an algorithm to compute a **regular expression** with the same language as a given **deterministic finite automaton**. We may visit this problem on HW#3.