Memoization and Dynamic Programming

Memoization is a technique that preserves the results of intermediate computations if they will be needed shortly afterwards, instead of recomputing them. It can be considered a form of caching. The classic example comes from calculating Fibonacci numbers, which is often one of the first examples used to teach recursion:

```
public static ulong FibonacciNumber(uint which) {
  if (which == 1 || which == 2) return 1;
  return FibonacciNumber(which-2) + FibonacciNumber(which-1);
}
```

This method has an appealing look, but its performance is quite appalling. For inputs as small as 45, this method takes a few seconds to complete; finding the 100th Fibonacci number using this approach is simply impractical, as its complexity grows exponentially.

One of the reasons for this inefficiency is that intermediate results are calculated more than once. For example, `FibonacciNumber(10)` is calculated recursively to find `FibonacciNumber(11)` and `FibonacciNumber(12)`, and again for `FibonacciNumber(12)` and `FibonacciNumber(13)`, and so on. Storing the intermediate results in an array can improve this method's performance considerably:

```
public static ulong FibonacciNumberMemoization(uint which) {
  if (which == 1 || which == 2) return 1;
  ulong[] array = new ulong[which];
  array[0] = 1; array[1] = 1;
  return FibonacciNumberMemoization(which, array);
}
private static ulong FibonacciNumberMemoization(uint which, ulong[] array) {
  if (array[which-3] == 0) {
    array[which-3] = FibonacciNumberMemoization(which-2, array);
  }
  if (array[which-2] == 0) {
    array[which-2] = FibonacciNumberMemoization(which-1, array);
  }
  array[which-1] = array[which-3] + array[which-2];
  return array[which-1];
}
```

This version finds the 10,000th Fibonacci number in a small fraction of a second and scales linearly. Incidentally, this calculation can be expressed in even simpler terms by storing only the last two numbers calculated:

```
public static ulong FibonacciNumberIteration(ulong which) {
  if (which == 1 || which == 2) return 1;
  ulong a = 1, b = 1;
  for (ulong i = 2; i < which; ++i) {
    ulong c = a + b;
    a = b;
    b = c;
  }
  return b;
}
```

Note It is worth noting that Fibonacci numbers have a closed formula based on the golden ratio (see http://en.wikipedia.org/wiki/Fibonacci_number#Closed-form_expression for details). However, using this closed formula to find an accurate value might involve non-trivial arithmetic.

The simple idea of storing results required for subsequent calculations is useful in many algorithms that break down a big problem into a set of smaller problems. This technique is often called *dynamic programming*. We now consider two examples.

Edit Distance

The *edit distance* between two strings is the number of character substitutions (deletions, insertions, and replacements) required to transform one

string into the other. For example, the edit distance between “cat” and “hat” is 1 (replace ‘c’ with ‘h’), and the edit distance between “cat” and “groat” is 3 (insert ‘g’, insert ‘r’, replace ‘c’ with ‘o’). Efficiently finding the edit distance between two strings is important in many scenarios, such as error correction and spell checking with replacement suggestions.

The key to an efficient algorithm is breaking down the larger problem into smaller problems. For example, if we know that the edit distance between “cat” and “hat” is 1, then we also know that the edit distance between “cats” and “hat” is 2—we use the sub-problem already solved to devise the solution to the bigger problem. In practice, we wield this technique more carefully. Given two strings in array form, *s[1. . .m]* and *t[1. . .n]*, the following hold:

- The edit distance between the empty string and *t* is *n*, and the edit distance between *s* and the empty string is *m* (by adding or removing all characters).
- If *s[i] = t[j]* and the edit distance between *s[1. . .i-1]* and *t[1. . .j-1]* is *k*, then we can keep the *i*-th character and the edit distance between *s[1. . .i]* and *t[1. . .j]* is *k*.
- If *s[i] ≠ t[j]*, then the edit distance between *s[1. . .i]* and *t[1. . .j]* is the minimum of:
  - The edit distance between *s[1. . .i]* and *t[1. . .j-1]*, +1 to insert *t[j]*;
  - The edit distance between *s[1. . .i-1]* and *t[1. . .j]*, +1 to delete *s[i]*;
  - The edit distance between *s[1. . .i-1]* and *t[1. . .j-1]*, +1 to replace *s[i]* by *t[j]*.

The following C# method finds the edit distance between two strings by constructing a table of edit distances for each of the substrings and then returning the edit distance in the ultimate cell in the table:

```
public static int EditDistance(string s, string t) {
  int m = s.Length, n = t.Length;
  int[,] ed = new int[m,n];
  for (int i = 0; i < m; ++i) {
    ed[i,0] = i + 1;
  }
  for (int j = 0; j < n; ++j) {
    ed[0,j] = j + 1;
  }
  for (int j = 1; j < n; ++j) {
    for (int i = 1; i < m; ++i) {
    if (s[i] == t[j]) {
    ed[i,j] = ed[i-1,j-1]; //No operation required
    } else { //Minimum between deletion, insertion, and substitution
    ed[i,j] = Math.Min(ed[i-1,j] + 1, Math.Min(ed[i,j-1] + 1, ed[i-1,j-1] + 1));
    }
    }
  }
  return ed[m-1,n-1];
}
```

The algorithm fills the edit distances table column-by-column, such that it never attempts to use data not yet calculated. Figure 9-2 illustrates the edit distances table constructed by the algorithm when run on the inputs “stutter” and “glutton”.

	g	l	u	t	t	o	n
s	1	2	3	4	5	6	7
t	2	2	3	3	4	5	6
u	3	3	2	3	4	5	6
t	4	4	3	2	3	4	5
t	5	5	4	3	2	3	4
e	6	6	5	4	3	3	4
r	7	7	6	5	4	4	4

Figure 9-2. The edit distances table, completely filled out

This algorithm uses $O(mn)$ space and has running time complexity of $O(mn)$. A comparable recursive solution that did not use memoization would have exponential running time complexity and fail to perform adequately even for medium-size strings.

All-Pairs-Shortest-Paths

The *all-pairs-shortest-paths* problem is to find the shortest distances between every two pairs of vertices in a graph. This can be useful for planning factory floors, estimating trip distances between cities, evaluating the required cost of fuel, and many other real-life scenarios. The authors encountered this problem in one of their consulting engagements. This was the problem description provided by the customer (see Sasha Goldshtein's blog post at http://blog.sashag.net/archive/2010/12/16/all-pairs-shortest-paths-algorithm-in-real-life.aspx for the original story):

- We are implementing a service that manages a set of physical backup devices. There is a set of conveyor belts with intersections and robot arms that manipulate backup tapes across the room. The service gets requests, such as "transfer a fresh tape *X* from storage cabinet 13 to backup cabinet 89, and make sure to pass through formatting station *C* or *D*".
- When the system starts, we calculate all shortest routes from every cabinet to every other cabinet, including special requests, such as going through a particular computer. This information is stored in a large hashtable indexed by route descriptions and with routes as values.
- System startup with 1,000 nodes and 250 intersections takes more than 30 minutes, and memory consumption reaches a peak of approximately 5GB. This is not acceptable.

First, we observe that the constraint "make sure to pass through formatting computer *C* or *D*" does not pose significant additional challenge. The shortest path from *A* to *B* going through *C* is the shortest path from *A* to *C* followed by the shortest path from *C* to *B* (the proof is almost a tautology).

The Floyd-Warshall algorithm finds the shortest paths between every pair of vertices in the graph and uses decomposition into smaller problems quite similar to what we have seen before. The recursive formula, this time, uses the same observation made above: the shortest path from *A* to *B* goes through some vertex *V*. Then to find the shortest path from *A* to *B*, we need to find first the shortest path from *A* to *V* and then the shortest path from *V* to *B*, and concatenate them together. Because we do not know what *V* is, we need to consider all possible intermediate vertices—one way to do so is by numbering them from 1 to *n*.

Now, the length of the shortest path (*SP*) from vertex *i* to vertex *j* using only the vertices 1,. . .,*k* is given by the following recursive formula, assuming there is no edge from *i* to *j*:

$$SP(i, j, k) = min \{ SP(i, j, k-1), SP(i, k, k-1) + SP(k, j, k-1) \}$$

To see why, consider the vertex *k*. Either the shortest path from *i* to *j* uses this vertex or it does not. If the shortest path does not use the vertex *k*, then we do not have to use this vertex and can rely on the shortest path using only the vertices 1,. . .,*k*-1. If the shortest path uses the vertex *k*, then we have our decomposition—the shortest path can be sewn together from the shortest path from *i* to *k* (that uses only the vertices 1,. . .,*k*-1) and the shortest path from *k* to *j* (that uses only the vertices 1,. . .,*k*-1). See Figure 9–3 for an example.

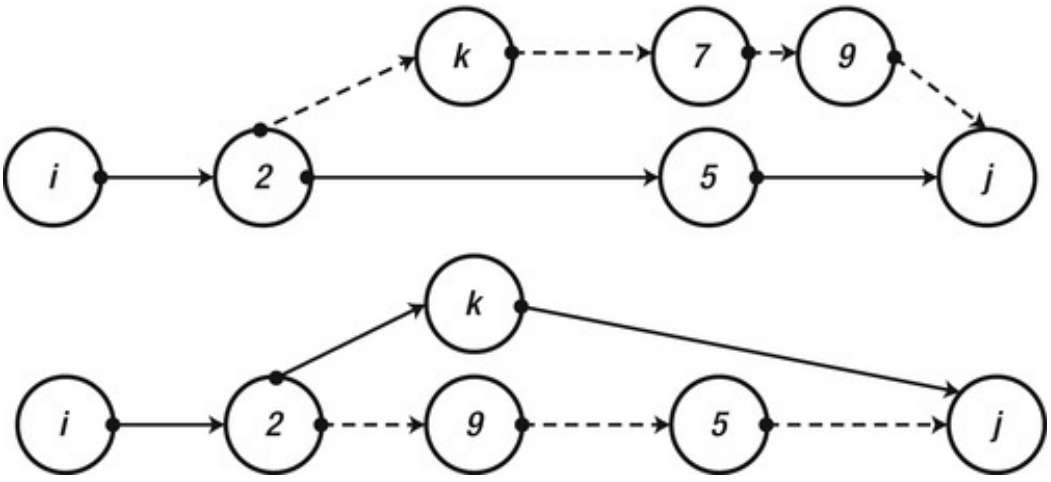

Figure 9-3. In the upper part of the diagram, the shortest path from i to j (through vertices 2 and 5) does not use the vertex k. Hence, we can restrict ourselves to the shortest path using vertices 1,. . .,k-1. In the lower part, the shortest path from i to j uses vertex k. Hence the shortest path from i to j is the shortest path from i to k, sewn together with the shortest path from k to j

To get rid of the recursive calls, we use memoization—this time we have a three-dimensional table to fill, and, after finding the values of *SP* for all pairs of vertices and all values of *k*, we have the solution to the all-pairs-shortest-paths problem.

We can further reduce the amount of storage space by noting that, for every pair of vertices *i*, *j*, we do not actually need all the values of *k* from 1 to *n*—we need only the minimum value obtained so far. This makes the table two-dimensional and the storage only $O(n^2)$. The running time complexity remains $O(n^3)$, which is quite incredible considering that we find the shortest path between every two vertices in the graph.

Finally, when filling the table, we need to record for each pair of vertices *i*, *j* the next vertex *x* to which we should proceed if we want to find the actual shortest path between the vertices. The translation of these ideas to C# code is quite easy, as is reconstructing the path based on this last observation:

```
static short[,] costs;
static short[,] next;

public static void AllPairsShortestPaths(short[] vertices, bool[,] hasEdge) {
```

```
    int N = vertices.Length;
    costs = new short[N, N];
    next = new short[N, N];
    for (short i = 0; i < N; ++i) {
      for (short j = 0; j < N; ++j) {
         costs[i, j] = hasEdge[i, j] ? (short)1 : short.MaxValue;
         if (costs[i, j] == 1)
            next[i, j] = -1; //Marker for direct edge
      }
    }
    for (short k = 0; k < N; ++k) {
      for (short i = 0; i < N; ++i) {
         for (short j = 0; j < N; ++j) {
            if (costs[i, k] + costs[k, j] < costs[i, j]) {
               costs[i, j] = (short)(costs[i, k] + costs[k, j]);
                  next[i, j] = k;
            }
         }
      }
    }
  }
  public string GetPath(short src, short dst) {
     if (costs[src, dst] == short.MaxValue) return " < no path > ";
     short intermediate = next[src, dst];
     if (intermediate == -1)
        return "- > "; //Direct path
     return GetPath(src, intermediate) + intermediate + GetPath(intermediate, dst);
  }
```

This simple algorithm improved the application performance dramatically. In a simulation with 300 nodes and an average fan-out of 3 edges from each node, constructing the full set of paths took 3 seconds and answering 100,000 queries about shortest paths took 120ms, with only 600KB of memory used.