Hallo!  
  
At the end of the most recent post I mentioned that we could use a technique (or better put, algorithmic framework) called Dynamic Programming to solve the Unidirectional TSP problem in a reasonable amount of time.  
  
You must be wondering: "What is DP???" Long story short, DP is a divide and conquer algorithm where we remember the solution to a subproblem so that we don't do extra work. Each DP problem is characterized by a *state* and a *recurrence relation*.  The *state* is the description of the problem and subproblems. It must be *unique* to the problem/subproblem. The recurrence relation, as we have seen in class, is a formula that gives the solution to the problem based on the solutions of the subproblems, and is defined in terms of the state. Defining the state and recurrence relation is usually the hardest part of solving a DP problem. Everything is cool beans after that.   
  
Enough of theory talk, let's start with the application!  
  
First of all, what is the state of this problem? Before reading on, try to think of one. Literally, stop now and think of one.  
  
Enough thinking? After a while you would probably conclude that a reasonable state for this problem is the coordinates of where you begin: (i,j), where 0 <= i < m, and 0 <= j < n. Half of the hard work is done, yay!  
  
What is the recurrence relation? First, what are the subproblems we need to solve? Try and think it on your own before continuing. After some thought, you notice that the problem pretty much tells you:  
  
- What is the minimum path starting at the northeast neighbour of (i,j)? (Sol. A)  
- What is the minimum path starting at the east neighbour of (i,j)? (Sol. B)  
- What is the minimum path starting at the southeast neighbour of (i,j)? (Sol. C)  
  
Now how do we combine solutions A, B and C and (i,j) to get the solution?  
Since we are searching for the minimum path, we should take the minimum of solutions A,B, and C and add it to the matrix value of M[i,j]! As a formula:  
  
f(i,j) = M[i,j] + min(f(i-1,j+1), f(i,j+1), f(i+1,j+1))  
  
Like all recurrence relations, f needs a base case. This amounts to asking: "When is the solution to the problem no longer a solution of the subproblems?" This happens when j = n-1: we have no more columns to explore from this point on.  
  
So this is our final recurrence relation:  
  
f(i,j) = M[i,j] if j=n-1  
           M[i,j] + min(f(i-1,j+1), f(i,j+1), f(i+1,j+1)) if j < n-1  
  
You may have noticed: what happens if i=0? or i=m-1? Then i-1=-1 and i+1=m make no sense, respectively. So we somehow need a function to say: when i=-1, we mean i=m-1, and when i=m, we mean i=0. This is easy to code (in C++):  
  
int A(i,m)  
{  
     if(i==-1) return m-1;  
     else if (i==m) return 0;  
     else return i;  
}  
  
So our recurrence relation comes out to be:  
  
f(i,j) = M[i,j] if j=n-1  
           M[i,j] + min(f(A(i-1,m),j+1), f(A(i,m),j+1), f(A(i+1,m),j+1)) if j < n-1  
  
  
So far, we have a divide and conquer algorithm. What will make this a DP algorithm? Remembering solutions, and not calculating solutions when they have already been calculated! This implies that we store our solutions somewhere. What data structure do we use? Our state gives us a clue: since each problem is characterized by a pair of indexes i and j, we can use a structure that depends on two indexes. A matrix does exactly that!  
  
Since the problem tells us that m will not be bigger than 10 and n will not be bigger than 100, we can have a matrix defined as follows that remembers solutions to problems already solved (in C++):  
  
 int memo[10][100]; where all entries are initialized to INFINITY.   
  
As we said before, in DP, if a solution has been calculated, there is no need in calculating it yet again. This, in a way, can constitute a base case in our recurrence: it doesn't depend on solving any more subproblems. With this in mind, our recurrence relation reaches its final form:  
  
f(i,j) = M[i,j] if j=n-1  
           memo[i][j]  if  memo[i][j] < INFINITY  
           memo[i][j] = M[i,j] + min(f(A(i-1,m),j+1), f(A(i,m),j+1), f(A(i+1,m),j+1))  
                               if j < n-1  
  
And we're done! With this we can efficiently calculate the cost of the minimum path in a unidirectional TSP! Notice that the problem also asks for the row indexes that constitute the minimum path: I won't delve into that part here, since it strays from the main idea, but realize that it's not too hard to code. Hint: the state contains the row index where we are currently at.  
  
This is normally how DP solutions arise. With a bit of practice, DP should become like an instinct to you. It is an efficient way of coding divide and conquer algorithms in real life, where time and space are real constraints.  
  
A lengthy post, I know, but I hope you have enjoyed it!  
  
Cheers!