Dynamic Programming: 1D Optimization

Fibonacci Sequence

To efficiently calculate $F[x]$, the xth element of the Fibonacci sequence, we can construct the array F from left to right (or "bottom up"). We start with $F[0] = 1$ and $F[1] = 1$ and iteratively calculate the next number in the sequence using $F[i] = F[i-1] + F[i-2]$ until we get $F[x]$.

Crazy 8's

In the game Crazy 8's, we want to find the longest subsequence of cards where consecutive cards must have the same value, same suit, or contains at least one eight. If the cards are stored in array C, we want to keep an auxiliary score array S where $S[i]$ represents the length of the longest subsequence ending with card $C[i]$. Again, we will construct the array C from left to right (or "bottom up").

We start with $S[0] = 1$ since the longest subsequence ending with the first card is that card itself and has a length of 1. We iteratively calculate the next score $S[i]$ by scanning all previous scores and set $S[i]$ to be $S[k] + 1$ where $S[k]$ represents the length of the longest subsequence that card $C[i]$ can be appended to.

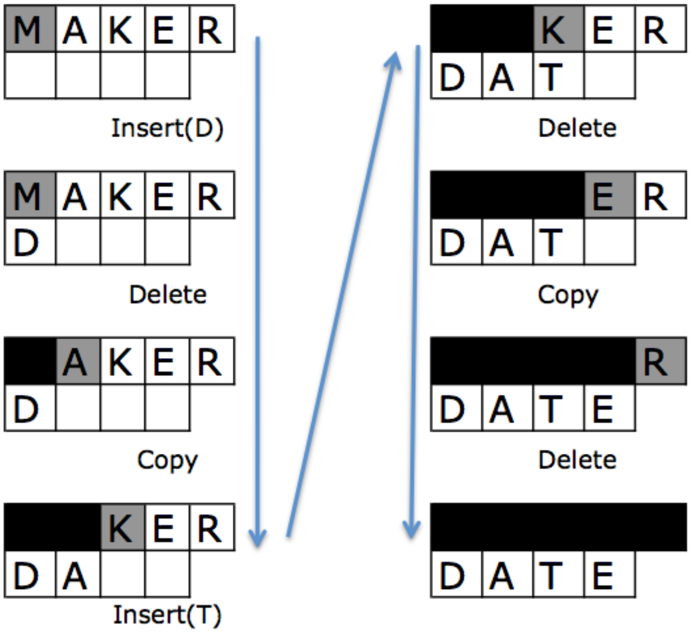
Dynamic Programming: 2D Optimization

Edit Distance

In the edit distance problem, we have two strings X and Y. Our goal is to find out the minimum cost of transforming string X into string Y using a set of editing operations. We will iterate through the characters of X and at each character, we have three editing operations that we can make:

1. **Copy** the current character of the source into the output
2. **Delete** the current character of the source and go onto the next character
3. **Insert** a new character into the output string

For example, let's look at how we can edit MAKER to DATE



Using a sequence of **[Insert(D), Delete, Copy, Insert(T), Delete, Copy, Delete]**, we were able to edit MAKER into DATE. Note that this sequence is not unique. We could have swapped the first Insert(D) and Delete with each other without changing the final result. A naive edit could have been deleting all of MAKER and then inserting all of DATE, making no copies in the edit sequence.

There are many ways we can edit X to Y, but we want to find which edit sequence has minimum cost. Each of the editing operations has some cost associated to it. For instance, we can assign costs as such:

- **Copy** - Cost 5
- **Delete** - Cost 10
- **Insert** - Cost 7

Our editing sequence **[Insert(D), Delete, Copy, Insert(T), Delete, Copy, Delete]** has a total cost of 54 according to our cost model. If we chose to delete all of MAKER and then insert all of DATE, we could have made 5 deletions and 4 insertions, resulting in a total cost of 78. Clearly, our first editing sequence is better than this naive edit, but how can we figure out what's the best overall?

Let X' and Y' be the strings X and Y without their last character (e.g. if X is DATE, X' is DAT). An important observation to make is that we can construct an editing sequence from X to Y in at most three ways:

1. Take the minimum cost editing sequence of X' to Y'. If the next characters are equal, then we can append a Copy to that editing sequence

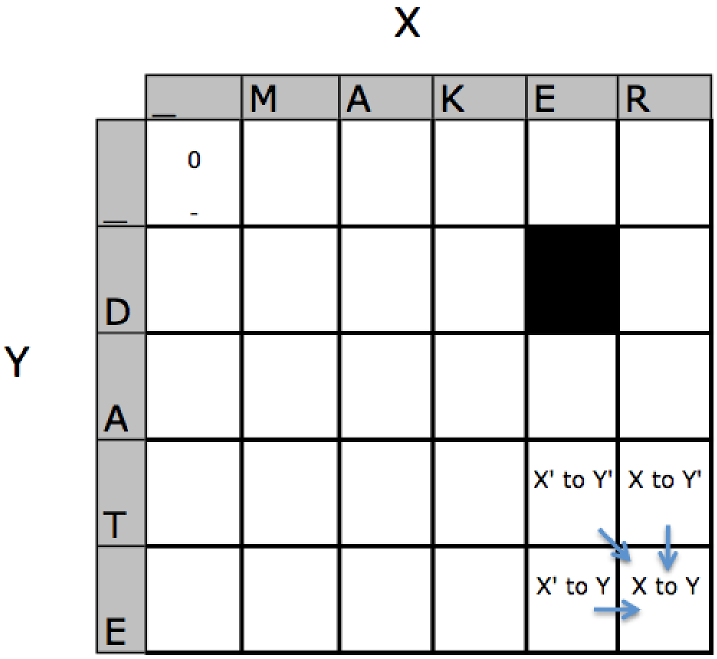
2. Take the minimum cost editing sequence of X to Y' and append an Insert(last character of Y) to that editing sequence

3. Take the minimum cost editing sequence of X' to Y and append a Delete to that editing sequence

Here, we are using optimal solutions to smaller problems to help us construct an optimal solution to the total problem. Using proof by contradiction, we can prove that the minimum cost editing sequence of X to Y must be one of the three sequences resulting above. To figure out which sequence is the best, simply take the minimum of the three sequences' costs and that will be the minimum editing sequence for X to Y.

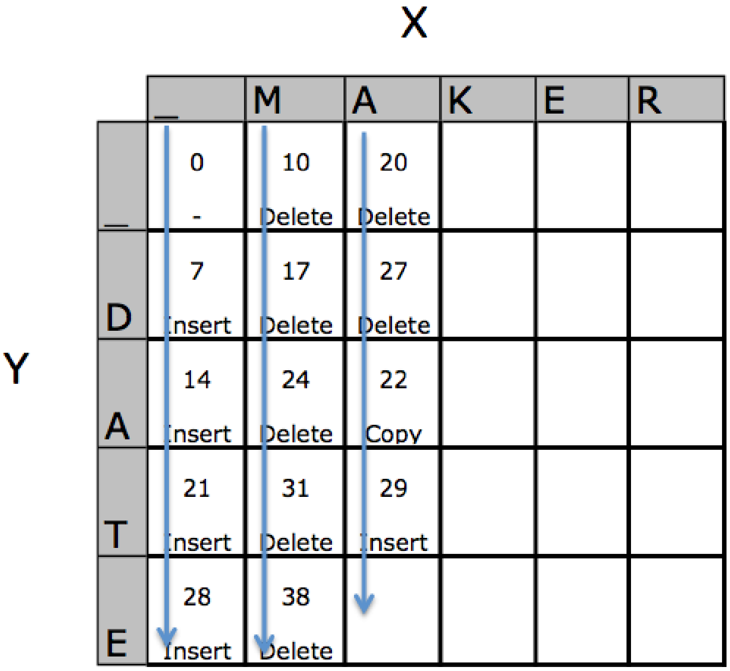
Like the Fibonacci and Crazy 8's examples, we want to construct the minimum editing sequence from bottom up. We do NOT want to start at figuring out the sequence from X to Y and recursively solving for the sequences from X' to Y', X to Y', and X' to Y. Instead, we can start with the smallest sequences and construct larger sequences from optimal solutions that we run into.

To solve this problem, we are going to use a 2D matrix. Each cell in the matrix corresponds to the cost of a minimum cost editing sequence to transform a prefix of X to a prefix of Y. In the example below, the shaded cell will contain the cost of editing MAKE into D.



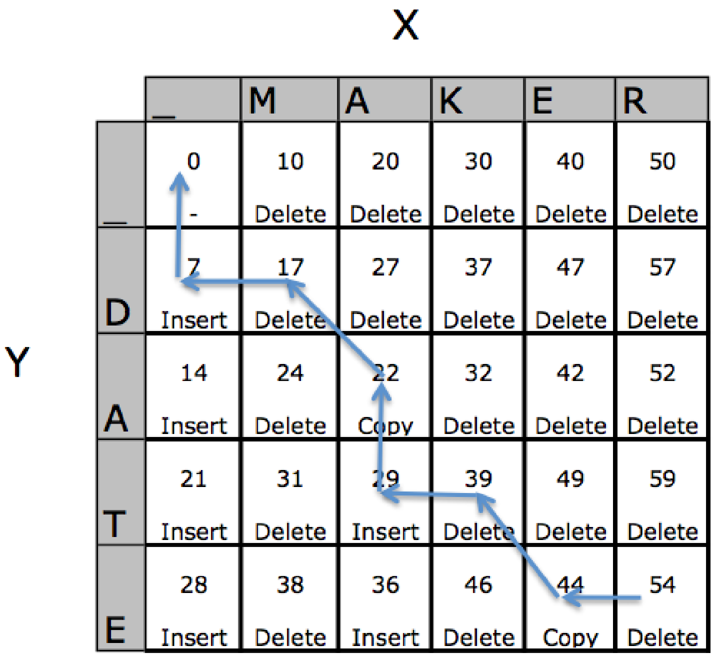
Each cell will also contain the last operation of the editing sequence for that particular transformation. Note that for each cell, based on our important observation above, we can construct an optimal editing sequence only if the above cell, left cell, and above left cell are filled (or nonexistent in the case of border cells).

From here, we can now systematically fill out our matrix. For each cell, compare the cost of appending a delete to the above cell, appending an insert to the left cell, and appending a copy to the above-left cell ONLY if the characters are the same. Take the minimum of those costs and append the corresponding operation. We get something like this:



	_	M	A	K	E	R
_	0 -	10 Delete	20 Delete			
D	7 Insert	17 Delete	27 Delete			
A	14 Insert	24 Delete	22 Copy			
T	21 Insert	31 Delete	29 Insert			
E	28 Insert	38 Delete				

Note the order of the cells that we're filling in (top to down from left to right). This order ensures that every time we reach an empty cell, the cells above, left, and above-left will be filled and consequently the empty cell can be filled in as well. There are several orders that work in this case (left to right from top to down, diagonally, etc.) Also, note that we can only make a copy operation when we're in a cell whose row and column are the same (in this case the copy operation was in an A row and an A column). The finished matrix looks like this:



	_	M	A	K	E	R
_	0 -	10 Delete	20 Delete	30 Delete	40 Delete	50 Delete
D	7 Insert	17 Delete	27 Delete	37 Delete	47 Delete	57 Delete
A	14 Insert	24 Delete	22 Copy	32 Delete	42 Delete	52 Delete
T	21 Insert	31 Delete	29 Insert	39 Delete	49 Delete	59 Delete
E	28 Insert	38 Delete	36 Insert	46 Delete	44 Copy	54 Delete

The optimal solution for editing MAKER to DATE is found in the lower right cell and has a cost of 54. We can trace the exact operations in this sequence by starting at a cell and then backtracing all the way to the first cell. We move up on Insert, left on Delete, and up-left on Copy.

The running time of this algorithm is $O(n^2)$ where n is the length of the strings. We need to construct a $n \times n$ size matrix and filling each cell in the matrix takes constant time (assuming the previous cells are already filled). Once we fill out each cell in the matrix, we have a solution to the edit distance problem.

All Pairs Shortest Path: Floyd-Warshall

Bellman-Ford and Dijkstra's algorithms both solve the single source shortest path problem, but in some cases it would be useful to know the shortest path between any pair of vertices in a graph. We could run Bellman-Ford or Dijkstra's V times, using each vertex as a source, to solve the problem. This takes $O(V^4)$ in the worst case for Bellman-Ford in the case of a dense graph. However, we can do better with dynamic programming.

Floyd-Warshall is a dynamic programming algorithm that solves the all pairs shortest path problem. Floyd-Warshall has subproblems where we find the shortest paths between all pairs of vertices with the constraint that only a subset of the total vertices are allowed to be intermediate vertices in a path. We denote $c_{ij}^{(k)}$ to be the shortest path length from v_i to v_j using only the vertices $\{v_1, v_2, ..., v_k\}$ as allowable intermediate vertices. $c_{ij}^{(0)}$ is the shortest path length from v_i to v_j using no intermediate vertices. $c_{ij}^{(n)}$ will be the shortest path length from v_i to v_j using all vertices as allowable intermediate vertices, i.e. $c_{ij}^{(n)}$ will be the shortest path length from v_i to v_j.

The first iteration of Floyd-Warshall is the subproblem where we find the shortest paths between all pairs of vertices with no vertices allowed to be intermediate vertices. This means the only shortest paths we find are the paths with no intermediate vertices, i.e. the paths with single edges. $c_{ij}^{(0)}$ = $w(v_i, v_j)$ if there's an edge from v_i to v_j, ∞ otherwise.

At iteration k of Floyd-Warshall, we add a single vertex v_k into the set of allowable intermediate vertices. For each v_i and v_j, we check to see if allowing this vertex to be an intermediate vertex will improve the shortest path estimate from v_i to v_j by using the relaxation concept. If the original shortest path estimate $c_{ij}^{(k-1)}$ is greater than $c_{ik}^{(k-1)} + c_{kj}^{(k-1)}$, then we update $c_{ij}^{(k)} = c_{ik}^{(k-1)} + c_{kj}^{(k-1)}$. Otherwise, $c_{ij}^{(k)} = c_{ij}^{(k-1)}$ if adding v_k as an intermediate vertex does not improve the shortest path.

The pseudocode in lecture demonstrates the concept succinctly. Assume in this case C is a matrix where $C[i][j]$ refers to the shortest path estimate from v_i to v_j. $C[i][j]$ is initialized to be the weight of the edge from v_i to v_j if it exists, ∞ if it doesn't.

```
for k = 1 to n:
  for i = 1 to n:
    for j = 1 to n:
      if C[i][j] > C[i][k] + C[k][j]:
        C[i][j] = C[i][k] + C[k][j]
```

Once the loops finish, $C[i][j]$ will contain the shortest path length from v_i to v_j. Augmenting the algorithm to maintain path pointers is trickier though. The runtime of Floyd-Warshall is $O(n^3)$ if there are n vertices.