202: Computer Science II

Northern Virginia Community College

Dynamic Programming

Cody Narber, M.S.
December 2, 2017

Dynamic Programming

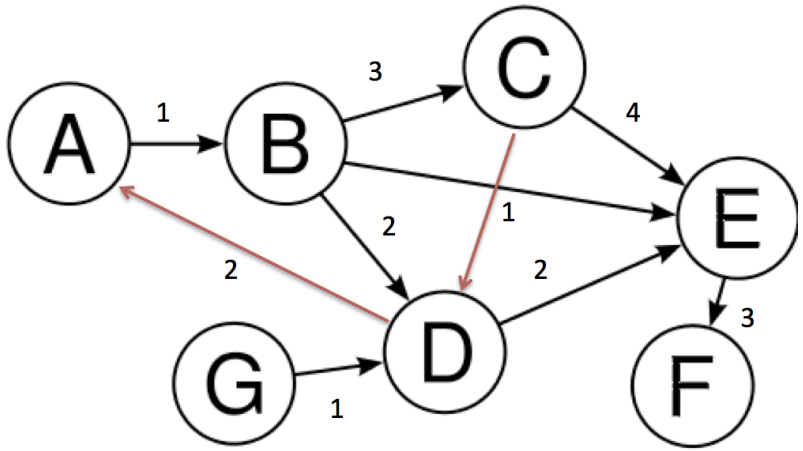
- **Top-Down** - Looking at the big picture and breaking it down into smaller and smaller problems. Recursion and Memoization.
- **Bottom-Up** - Looking at the base/simple parts and build up all possible solutions. Iterative and Dynamic Programming.

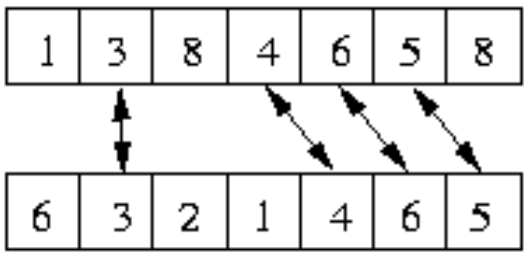
Iterative/Dynamic Programming Factorial Solution:

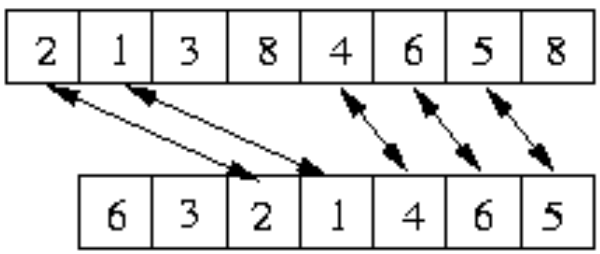
```
1  public class DPFactorial{
2      private int[] savedFacs;
3      public DPFactorial(){
4          savedFacs=new int[999]; //defaults to 0 in all spots
5          savedFacs[0]=1;
6          for(int i=1; i<999; i++)
7              savedFacs[i]=i*savedFacs[i-1];
8      }
9      public int fac(int n){
10       return savedFacs[n];
11     }
12 }
```

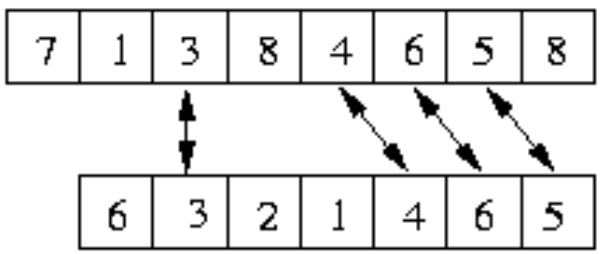
Dynamic Programming

- Change Making Problem
- Viterbi Algorithm
- Floyd – Warshall algorithm
- Longest Common Subsequence









Change Making Problem

Specification: How can a given amount of money be made with the least number of coins of given denominations?

US Currency:

- 1
- 5
- 10
- 25

Made-Up Currency:

- 1
- 3
- 4
- 12

What coins (Greedy)?

- 37 = 1(25), 1(10), 2(1)
- 37 = 3(12), 1(1)
- 30 = 1(25), 1(5)
- 30 = 2(12), 1(4), 2(1)

What coins (Correct)?

- 37 = 1(25), 1(10), 2(1)
- 37 = 3(12), 1(1)
- 30 = 1(25), 1(5)
- **30 = 2(12), 2(3)**

Greedy Algorithms

Greedy algorithm is a type of algorithm that finds a solution to an optimization problem. The heuristic that a greedy algorithms uses is choosing a local optimal path/choice/solution.

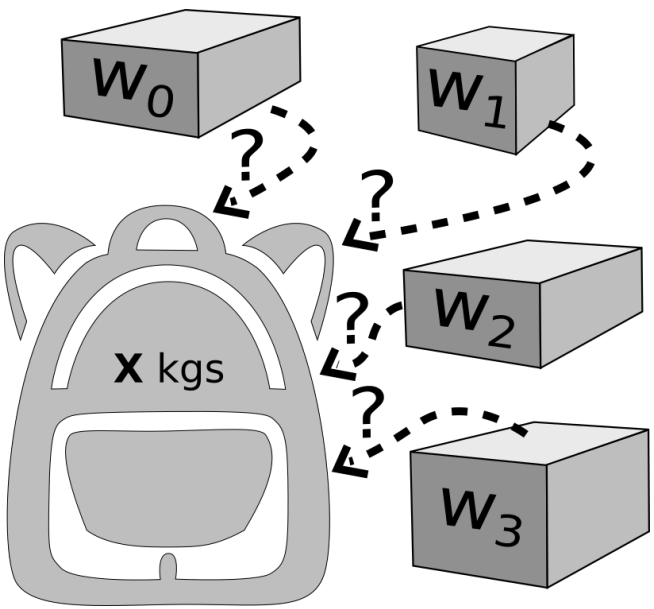
The greedy algorithm for the change problem:

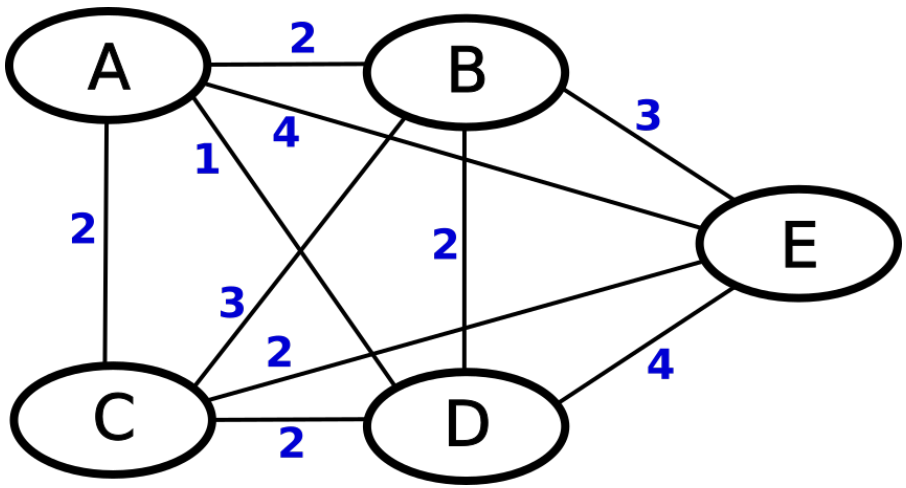
```
1 //Precondition: Change is sorted high to low
2 public int[] getChangeGreedy(int[] change, int total){
3    int[] amounts = new int[change.length];
4    for(int i=0; i<change.length; i++){
5       amounts[i] = total/change;
6       total = total%change;
7    }
8    return amounts;
9 }
```

Another greedy algorithm we have seen is Prim's Algorithm (Finding Minimum Spanning Tree).

Greedy Algorithms - Fail

- Knapsack Problem
- Shortest Distance Between Nodes
- Traveling Salesman Problem





Change Dynamic Programming

Dynamic Programming is a type of algorithm that finds a solution to sub-problems and uses those solutions to generate the solution to the next problem.

The Dynamic Programming Solution (change):

```
1  public int getAmountChangeDP(int[] change, int total){
2     int[][] amounts = new int[total][change.length];
3     for(int i=0; i<total; i++)
4        for(int c=0; c<change.length; c++){
5           if(change[c]==i+1) amounts[i][c] = 1;
6           if(i-change[c] >= 0){
7              amounts[i][c] = 1 + getMin(amounts[i-change[c]]);
8           }
9        }
10    return getMin(amounts[total-1]);
11 }
12 public int getMin(int[] amt){
13    int min = -1;
14    for(int i=0; i<amt.length; i++)
15       if(amt[i]!=0 &&(min==-1 || amt[i]<min))
16          min=amt[i];
17    return min;
18 }
```

Change Dynamic Programming

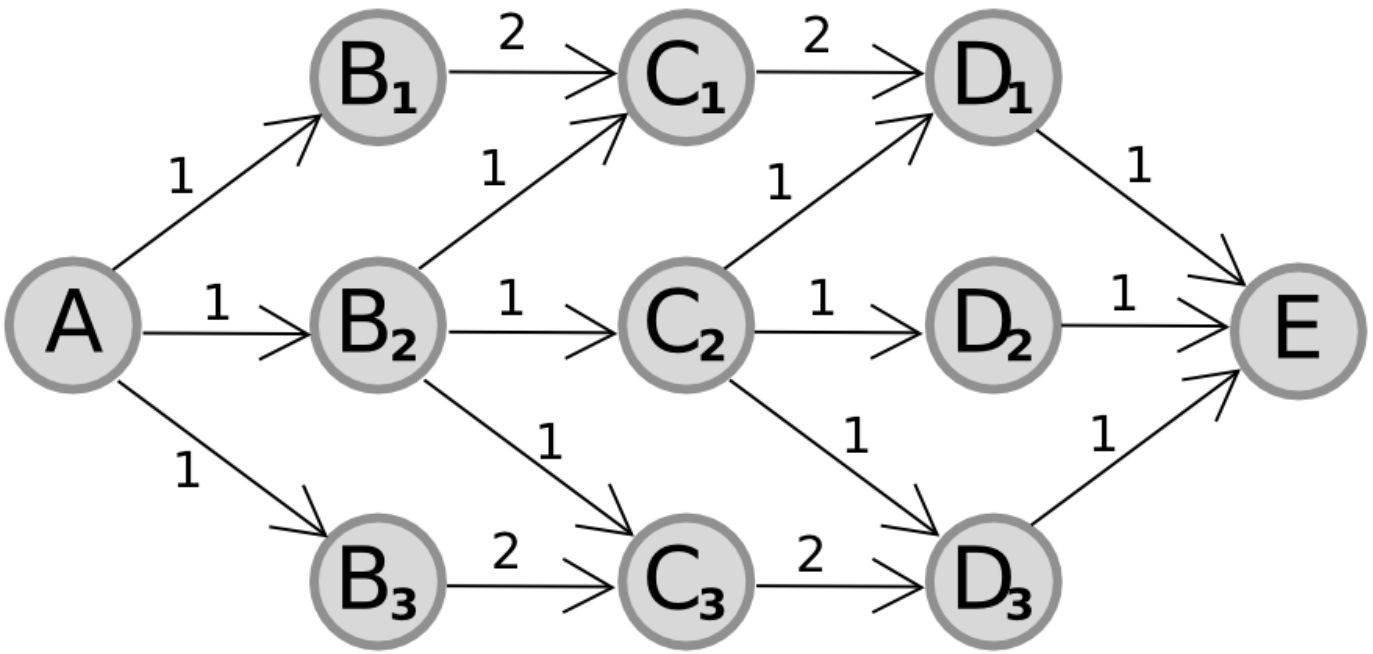
Populating all values up to total (example: $\{12, 4, 3, 1\}$)

	1	2	3	4	5	6	7	8	9	10	11	12
12	0	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	1	2	3	2	2	3	3	3	3
3	0	0	1	2	3	2	3	3	3	3	3	4
1	1	2	3	2	2	3	3	3	3	4	4	4

Matrix transposed from previous slide's code

All Pairs Shortest Path

The all-pairs shortest path problem is when you are given a directed graph $G = (V, E)$, which consists of a set of n verticies $V = \{v_1, v_2, ..., v_{n-1}, v_n\}$ and m edges $E = \{e_1, e_2, ..., e_{m-1}, e_m\}$. The edges are directed meaning that if tracing out a path they can only go in one direction (hence drawn with arrows).



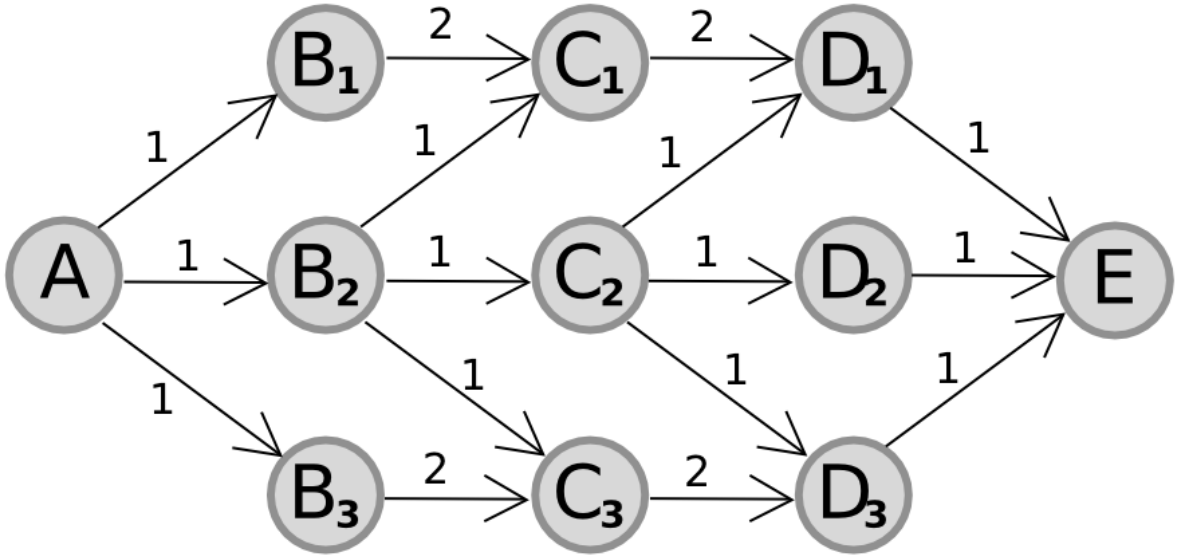
All Pairs Shortest Path - Recursion

Let us first think about the finding the shortest path from one vertex to another. $v_i \rightarrow v_j$. Though we do have to make a assumption that there are no cycles...i.e. their exists a path such that $v_i \rightarrow ... \rightarrow v_i$
RecursivePath(v_i,v_j, G)

- **Base Case:** When $|E|$ is 1 (NOTE: pipes $|$ around E signify cardinality, a.k.a. number of elelements in the set) Then if that edge connects v_i to v_j return that edge's cost otherwise return ∞
- **General Case:** Return the cost of: (e_{ik}+RecursivePath(v_k,v_j, $G - v_i$)) for all v_k in V, such that k minimizes the cost.

When we think about the number of recursive steps it will be: (n-1)*(n-2)*(n-3)*... or (n-1)! and that is just for 1 vertex to 1 vertex! NOWHERE near all-pairs

All Pairs Shortest Path - Bottom-Up



For each Vertex, we want to see if if is connected to every other vertex by a single path...Columns is destination, rows are starting.

	A	B_1	B_2	B_3	C_1	C_2	C_3	D_1	D_2	D_3	E
A	0	1	1	1	∞	∞	∞	∞	∞	∞	∞
B_1	∞	0	∞	∞	2	∞	∞	∞	∞	∞	∞
...											
E	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	0

All Pairs Shortest Path - Single Edge Paths

Populating for all values for a single path...O(n^2)

	A	B_1	B_2	B_3	C_1	C_2	C_3	D_1	D_2	D_3	E
A	0	1	1	1	∞	∞	∞	∞	∞	∞	∞
B_1	∞	0	∞	∞	2	∞	∞	∞	∞	∞	∞
B_2	∞	∞	0	∞	1	1	1	∞	∞	∞	∞
B_3	∞	∞	∞	0	∞	∞	2	∞	∞	∞	∞
C_1	∞	∞	∞	∞	0	∞	∞	2	∞	∞	∞
C_2	∞	∞	∞	∞	∞	0	∞	1	1	1	∞
C_3	∞	∞	∞	∞	∞	∞	0	∞	∞	2	∞
D_1	∞	∞	∞	∞	∞	∞	∞	0	∞	∞	1
D_2	∞	∞	∞	∞	∞	∞	∞	∞	0	∞	1
D_3	∞	∞	∞	∞	∞	∞	∞	∞	∞	0	1
E	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	0

All Pairs Shortest Path - Double Edge Paths

Now that the data for all single paths are filled we can check to see if there exists a cheaper 2 step path. $Start \rightarrow End$...
$Start \rightarrow Mid + Mid \rightarrow End$. This would result in Each Vertex pair having n steps...so O(n^3).

So we would check every single pair is:

- $A \rightarrow A > A \rightarrow A + A \rightarrow A$ $(0 < 0 + 0)$?
- $A \rightarrow A > A \rightarrow B_1 + B_1 \rightarrow A$ $(0 < 1 + \infty)$?
- ...
- $A \rightarrow B_1 > A \rightarrow A + A \rightarrow B_1$ $(1 < 0 + 1)$?
- $A \rightarrow B_1 > A \rightarrow B_1 + B_1 \rightarrow B_1$ $(1 < 1 + 0)$?
- ...
- $A \rightarrow C_1 > A \rightarrow A + A \rightarrow C_1$ $(\infty < 0 + \infty)$?
- $A \rightarrow C_1 > A \rightarrow B_1 + B_1 \rightarrow C_1$ $(\infty < 1 + 2)$? **UPDATE!**
- $A \rightarrow C_1 > A \rightarrow B_2 + B_2 \rightarrow C_1$ $(3 < 1 + 1)$? **UPDATE!**
- ...

All Pairs Shortest Path - Double Edge Paths

We want to repeat the steps for all starting nodes...O(n^3).

	A	B_1	B_2	B_3	C_1	C_2	C_3	D_1	D_2	D_3	E
A	0	1	1	1	2	2	2	∞	∞	∞	∞
B_1	∞	0	∞	∞	2	∞	∞	4	∞	∞	∞
B_2	∞	∞	0	∞	1	1	1	2	2	2	∞
B_3	∞	∞	∞	0	∞	∞	2	∞	∞	4	∞
C_1	∞	∞	∞	∞	0	∞	∞	2	∞	∞	3
C_2	∞	∞	∞	∞	∞	0	∞	1	1	1	2
C_3	∞	∞	∞	∞	∞	∞	0	∞	∞	2	3
D_1	∞	∞	∞	∞	∞	∞	∞	0	∞	∞	1
D_2	∞	∞	∞	∞	∞	∞	∞	∞	0	∞	1
D_3	∞	∞	∞	∞	∞	∞	∞	∞	∞	0	1
E	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	0

All Pairs Shortest Path - n-Edge Paths

Assume we don't know the length of the shortest path...there is a possibility that the path could go through every vertex, therefore we need to perform the previous steps again and again for 3 length paths, 4 length paths...n length paths $O(n^4)$

	A	B_1	B_2	B_3	C_1	C_2	C_3	D_1	D_2	D_3	E
A	0	1	1	1	2	2	2	3	3	3	4
B_1	∞	0	∞	∞	2	∞	∞	4	∞	∞	5
B_2	∞	∞	0	∞	1	1	1	2	2	2	3
B_3	∞	∞	∞	0	∞	∞	2	∞	∞	4	5
C_1	∞	∞	∞	∞	0	∞	∞	2	∞	∞	3
C_2	∞	∞	∞	∞	∞	0	∞	1	1	1	2
C_3	∞	∞	∞	∞	∞	∞	0	∞	∞	2	3
D_1	∞	∞	∞	∞	∞	∞	∞	0	∞	∞	1
D_2	∞	∞	∞	∞	∞	∞	∞	∞	0	∞	1
D_3	∞	∞	∞	∞	∞	∞	∞	∞	∞	0	1
E	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	0

Floyd – Warshall algorithm

Given that we maintain the shortest path from $v_i \rightarrow v_j$ everytime a cell gets updated we no longer have to account for previous calculations (do not recompute double links, triple, ... therefore we can eliminate the redundancies and compute all-pairs shortest path into $O(n^3)$ running time.

```
1  //Assume edges, verticies, and costs arrays exists
2  for(int i=0; i<Vs; i++) //Initialize to max distances
3       for(int j=0; j<Vs; j++) costs[i][j] = Double.MAX_VALUE;
4
5  for(int i=0; i<Vs; i++) costs[i][i] = 0; //distance to self
6  for(int i=0; i<edges.length; i++){
7       int v1 = edges[i].getV1Ind();
8       int v2 = edges[i].getV2Ind();
9       costs[v1][v2] = edges[i].getCost(); //single edge
10 }
11 for(int k =0; k < Vs; k++)
12      for(int i =0; i < Vs; i++)
13           for(int j =0; j < Vs; j++)
14                if(costs[i][j] > costs[i][k] + costs[k][j])
15                     costs[i][j] = costs[i][k] + costs[k][j]
```

Longest Common Subsequence

The longest common subsequence (LCSS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences). It differs from problems of finding common substrings: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences. Used frequently in **Bioinformatics**.

Example of all subsequences of **GTAC** compared to **TGCA**:

G	T	A	C
GT	GA	GC	TA
TC	AC	GTA	GTC
GAC	TAC	GTAC	

T	G	C	A
TG	TC	TA	GC
GA	CA	TGC	TGA
TCA	GCA	TGCA	

Longest Common Subsequence

The longest common subsequence (LCSS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences). It differs from problems of finding common substrings: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences. Used frequently in **Bioinformatics**.

Example of all subsequences of **GTAC** compared to **TGCA**:

G	T	A	C
GT	GA	GC	TA
TC	AC	GTA	GTC
GAC	TAC	GTAC	

T	G	C	A
TG	TC	TA	GC
GA	CA	TGC	TGA
TCA	GCA	TGCA	

Longest Common Subsequence

We can establish a recursive definition to solve the LCSS problem, through several cases. Think about looking at the sequences $X = \{x_0, x_1, ..., x_{n-1}\}$, and $Y = \{y_0, y_1, ..., y_{m-1}\}$. Let us think about some cases:

- Lets assume that the last two characters in both sequences match $X_{n-1} = Y_{m-1}$...then the problem simplifies to finding the LCSS of the two substrings with the last character taken off.
- Otherwise they do not match and one of the last characters will not be used either X_{n-1} or Y_{m-1}
- In the last case, if one of the sequences is empty then there is no sequence.

Longest Common Subsequence

Extending the terminology of our sequences we can define a subsequence as $X_i = \{x_0, x_1, ..., x_i\}$, and $Y_j = \{y_0, y_1, ..., y_j\}$ such that we can define our recursive definition as:

$$LCSS(X_i, Y_j) = \begin{cases} 0 & i = -1 \text{ or } j = -1 \\ LCSS(X_{i-1}, Y_{j-1}) + 1 & x_i = y_j \\ larger(LCSS(X_i, Y_{j-1}), & x_i \neq y_j \\ \qquad LCSS(X_{i-1}, Y_j)) & \end{cases} \tag{1}$$

As can be seen this is the top-down perspective of breaking a problem down into smaller and smaller pieces from the big picture. If we reverse this idea and build up from subsequences, we can define out dynamic programming solution (which is more efficient).

Longest Common Subsequence

Let us look at the concrete example from before **GTAC** compared to **TGCA**:

Let us begin with comparing the first letter to *G* to each other spot, maintaining the best match seen so far:

	T	G	C	A
G	0	1	$\leftarrow$ 1	$\leftarrow$ 1

Now Compare all subsequences of *GT* to what we have

	T	G	C	A
G	0	1	$\leftarrow$1	$\leftarrow$1
T	1	$\leftarrow\uparrow$1	$\leftarrow\uparrow$1	$\leftarrow\uparrow$ 1

Longest Common Subsequence

Continuing to populate the array as we match sequences given the rules that we described with the recursive definition:

	T	G	C	A
G	0	1	←1	←1
T	1	←↑1	←↑1	←↑ 1
A	↑1	←↑1	←↑1	↖ 2
C	↑1	←↑1	↖2	←↑ 2

We can backtrack from the lower corner following the arrows to construct the sequence as well as the letters matched, to arrive at what was seen earlier.

Longest Common Subsequence - Backtrack

Matching the results from before: TC, TA, GC, GA:

	T	G	C	A
G	0	1	←1	←1
T	1	←↑1	←↑1	←↑ 1
A	↑1	←↑1	←↑1	↖ 2
C	↑1	←↑1	↖2	←↑2

	T	G	C	A
G	0	1	←1	←1
T	1	←↑1	←↑1	←↑ 1
A	↑1	←↑1	←↑1	↖ 2
C	↑1	←↑1	↖2	←↑2

	T	G	C	A
G	0	1	←1	←1
T	1	←↑1	←↑1	←↑ 1
A	↑1	←↑1	←↑1	↖ 2
C	↑1	←↑1	↖2	←↑2

	T	G	C	A
G	0	1	←1	←1
T	1	←↑1	←↑1	←↑ 1
A	↑1	←↑1	←↑1	↖ 2
C	↑1	←↑1	↖2	←↑2