CS 577: Introduction to Algorithms Fall 2015

Dynamic Programming

Instructor: Dieter van Melkebeek Scribe: Andrew Morgan

Oftentimes, it's easy to construct a recursive algorithm that solves a problem, and usually these algorithms end up being quite slow. However, sometimes these recursive approaches end up repeating a lot of work, re-solving the same subproblem many times over the course of their execution. The key insight with dynamic programming is simple: by simply *remembering* the results of previous work, it is no longer necessary to completely redo an entire subcomputation.

There are a few ways to do this. One very generic way is to have a global table which is accessible by every recursive call. Each entry of the table corresponds to a different setting of parameters to the recursive function. (One can think of the table as a global variable, or else as a parameter to the recursive call which is passed by reference.) In the body of a recursive call, we simply check to see whether the table corresponding to the current call has been filled in. If it has been filled in, we simply returned the filled-in value; otherwise, we compute the answer as normal, except that right before we return our answer, we write our answer into the call's spot on the table. The end result is that the 'actual' computation for each distinct recursive call is done only once—any future repeats are saved by a simple table lookup. This method is referred to as "top-down dynamic programming".

Another method is called "bottom-up dynamic programming". In this method, there is the exact same table as before. However, the way we fill it in is different. Rather than implementing a recursive method to recursively fill in the table, we simply find an ordering on the entries of the table which is compatible with the recursive calls, and then fill the entries in in this order. By 'compatible', we mean that, when filling in a given entry, all of the entries that it depends on should already be filled in.

However, simply avoiding repeated subproblems isn't the whole story. This idea works well for some recursive algorithms, but not for all. If the recursive algorithm still has a lot of distinct subproblems, then this insight isn't enough to make the recursive algorithm efficient. On the other hand, it's quite possible that a different recursive algorithm solves the same problem, but does so with only a few distinct subproblems appearing in its recursion tree. Usually coming up with this better approach is the primary difficulty of applying dynamic programming to a problem, since such approaches are rarely obvious. That said, dynamic programming is very effective technique for solving problems, which we hope to illustrate here with a few examples.

1 Computing the Fibonacci Sequence

Our first example is that of computing the Fibonacci sequence. While more of a toy example, it's intended to illustrate most of the points presented in the introduction without needing to give the intuition behind some creative, nonobvious recursive algorithm.

The Fibonacci sequence is the sequence of natural numbers defined recursively by

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+2} &= F_{n+1} + F_n \qquad (\forall n \geq 0) \end{aligned}$$

Its first few elements are

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657$$

It grows exponentially: the n-th term F_n is $\Theta(\varphi^n)$, where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio. The Fibonacci sequence shows up in many places in mathematics, and we don't even try to cover all of these here. However, one particular occurrence in algorithms is their role in the analysis of the Euclidean algorithm—how many steps does it take to compute $\gcd(F_{n+1}, F_n)$?

For our purposes, we just want to want to compute the n-th Fibonacci number. Formally, we want to write a program meeting the following specification:

Input: A natural number $n \in \mathbb{N}$

Output: The n-th Fibonacci number, F_n

One very natural way to do this is to translate the recursive definition into a recursive procedure. One way to do this is given in Algorithm 1.

Algorithm 1

Input: n, a natural number
Output: F_n, the n-th Fibonacci number

```
1: procedure Fib-Recursive($n$)
2:     if $n \leq 1$ then
3:         return $n$
4:     else
5:         return Fib-Recursive($n - 1$) + Fib-Recursive($n - 2$)
```

Its recursion tree is given in Figure 1. It's not too hard to see that the tree has exponentially many nodes in it—it is complete at least up to the $(n/2)$-th level, and so has at least $2^{n/2}$ nodes in it. A closer analysis reveals that the computation for $n - k$ appears F_{k+1} times, and so the tree has $F_0 + F_1 + \cdots + F_n = \Theta(\varphi^n)$ nodes in it.

On the other hand, there are only $n + 1$ distinct calls being made here, one for each value $0, 1, \ldots, n$. So by simply computing the value of each call once (instead of exponentially many times), we can save a lot of work. More pictorially, we can represent our recursion "tree" as the directed acyclic graph given in Figure 2. The nodes represent distinct recursive calls, corresponding to new work that has to be done, while the arrows indicate the dependencies among these calls.

Note that if we think of storing the values $F_0 = 0$ and $F_1 = 1$ in in the vertices marked 0 and 1, then it's very easy to compute $F_2 = F_0 + F_1$, store it into the vertex marked 2, then compute $F_3 = F_2 + 1$, store it in the vertex marked 3, and so on. The overall time required to do this is linear in n—a huge improvement over the $\Theta(\varphi^n)$ we had before! The code isn't even very long

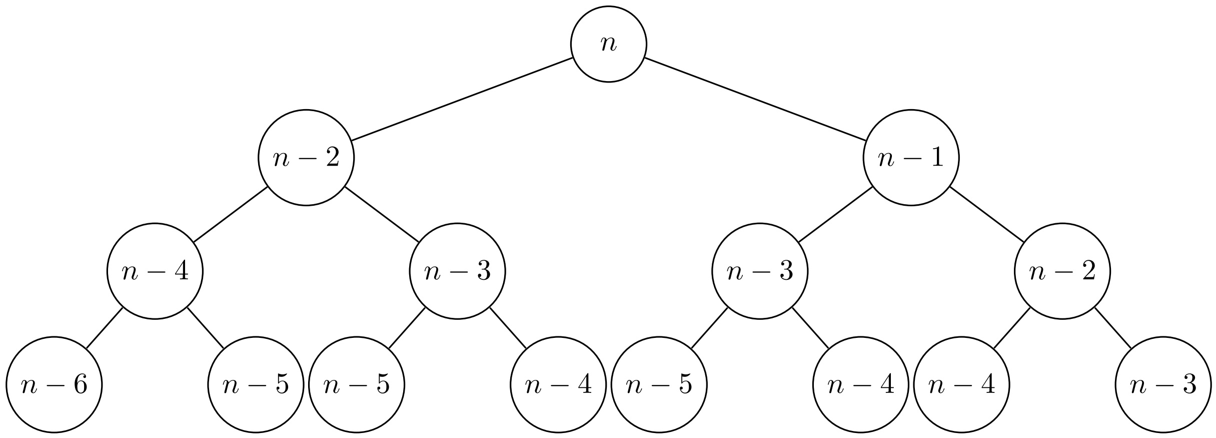


Figure 1: Recursion tree of Algorithm 1

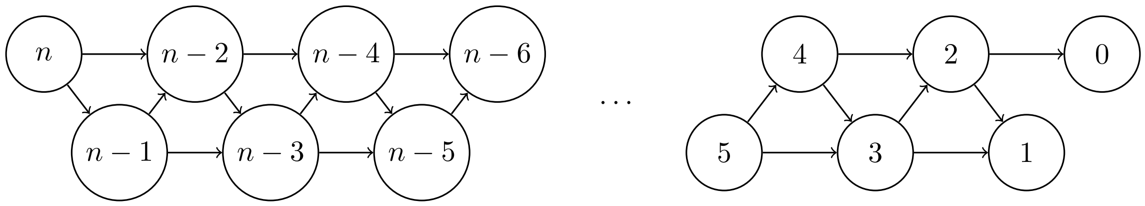


Figure 2: DAG representation of Algorithm 1's recursive calls

either: consider the pseudocode in Algorithms 2 and 3 which contain top-down and bottom-up implementation of our dynamic programming solution to computing the Fibonacci numbers.

One of the primary drawbacks to a dynamic programming algorithm is that the space required to remember the result of every recursive call can often be unreasonably large. However, it is sometimes possible to realize that one only needs to keep track of a small portion of smaller instances in order to compute the values of larger instances. For instance, in our example with the Fibonacci numbers, we use space $\Theta(n)$ to create our table. Instead, we could simply keep track of the last two numbers we've computed, since they are all that's needed to compute the future values.

Algorithm 2

Input: n, a natural number

Output: F_n, the n-th Fibonacci number

```
1: procedure FIB-DP-TOPDOWN($n$)
2:     Table[$0 \cdots n$] $\leftarrow$ a new array of size $n+1$, initialized to none
3:     return FIB-DP-REC(Table,$n$)
4: procedure FIB-DP-REC(Table (by reference),$n$)
5:     if Table[$n$] $\neq$ none then
6:         return Table[$n$]
7:     else
8:         if $n \leq 1$ then
9:             Table[$n$] $\leftarrow n$
10:        else
11:            Table[$n$] $\leftarrow$ FIB-DP-REC(Table, $n$)
12:        return Table[$n$]
```

Algorithm 3

Input: n, a natural number

Output: F_n, the n-th Fibonacci number

```
1: procedure FIB-DP-BOTTOMUP($n$)
2:     Table[$0 \cdots n$] $\leftarrow$ a new array of size $n+1$, initialized to none
3:     for $i = 0 \ldots n$ do
4:         if $n \leq 1$ then
5:             Table[$n$] $\leftarrow n$
6:         else
7:             Table[$n$] $\leftarrow$ Table[$n-1$] + Table[$n-2$]
8:     return Table[$n$]
```

2 Weighted Interval Scheduling

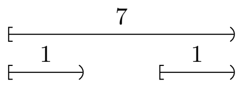
Our next problem exemplifying dynamic programming is a generalization of the interval scheduling problem that we saw in the section on greedy algorithms. As suggested by the name, the generalization is that each interval now has a weight associated with it, and the goal is to find a set of nonconflicting intervals so that the *total weight* of this set is maximized. Formally, we have the following problem description:

Input: A set of intervals described by their endpoints: The i-th is given by $[s_i, e_i)$. Each interval additionally has a weight; the i-th has weight w_i.

Output: A subset $S \subseteq \{1, 2, \ldots, n\}$ of nonconflicting intervals for which the sum $\sum_{i \in S} w_i$ is maximized.

This problem includes the case we saw in the section on greedy algorithms—simply suppose that $w_i = 1$ for every interval i. However, the greedy algorithm won't work for every instance of this problem: consider the input depicted by Figure 3. Instead we will have to do something smarter.

Figure 3: Weighted Interval Scheduling, Greedy's Failure



Greedy chooses the two intervals of weight 1, netting a total weight of 2, while choosing the long interval yields a solution of weight 7.

This is where dynamic programming will come in. However, we have to figure out the details. We have to devise a recursive algorithm which has few distinct recursive calls. A good place to start is with the ideas we had with our greedy algorithm before. We can sort the intervals in increasing order of ending time and consider two choices:

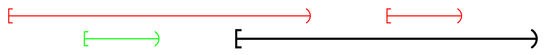
- The first interval is in the solution set
- The first interval is not in the solution set

In the second case, we can just recursively solve the problem on the remaining intervals. In the first case, we can remove all the intervals that conflict with the first interval (since they can't be in a solution that contains the first interval), and what's left is exactly an instance of our original problem.

This is almost the right idea. The problem is that, when we remove the first interval, it has the potential to conflict with an arbitrary subset of the other intervals. The consequence for our recursion is that there may be many distinct calls in the recursion tree—one for each of exponentially many different subsets of intervals. This isn't good enough for dynamic programming purposes; we have to fix this.

But notice what happens if we consider the above logic applied to the *last* interval. When we take the last interval, conflicts arise exactly when some other interval's ending time comes after the last interval's starting time. See Figure 4 for a representative picture. Since the intervals are sorted in order of increasing ending time, this means that the set of non-conflicting intervals corresponds to a *prefix* of the sorted list of intervals.

Figure 4: Weighted Interval Scheduling, Characterization of Conflicts



So now suppose we expressed our recursion in the following way: The input is an integer i indicating that we should only consider the first i intervals in sorted order, and the goal is to produce a subset of these intervals of maximal weight. The recursion recursively computes the solution for the first $i-1$ intervals (checking the case where the i-th interval is not in the optimal solution) and then recursively computes the solution for the first k intervals, where the k-th interval is the last interval in sorted order which does not conflict with the i-th interval. The end result is still a correct algorithm by the reasoning given above. However now the number of distinct recursive calls is quite small—there are only $n+1$ valid choices of i $(0, 1, \ldots, n)$, so there are only $n+1 = O(n)$ many distinct recursive calls. Thus we can apply memoization and have an efficient algorithm. An example bottom-up implementation is given in Algorithm 4.

Algorithm 4

```
Input: The $n$ intervals given as $(s_1, e_1), \ldots, (s_n, e_n)$ with weights $w_1, \ldots, w_n$
Output: A conflict-free subset of intervals of maximum weight
 1: procedure WEIGHTED-INT-SCHED-DP-SLOW($s_i, e_i, w_i$)
 2:     Sort the intervals in order of increasing end time
 3:     $S[0, \ldots, n] \leftarrow$ new array of subsets of $\{1, 2, \ldots, n\}$ of size $n+1$     ▷ The actual solutions
 4:     $W[0, \ldots, n] \leftarrow$ new array of integers     ▷ The weight of the solutions
 5:     $S[0] \leftarrow \varnothing$
 6:     $W[0] \leftarrow 0$
 7:     for $i = 1, \ldots, n$ do
 8:         $p \leftarrow 0$     ▷ Find the prefix of intervals that don't conflict with interval $i$
 9:         while $e_{p+1} \leq s_i$ do
10:             $p \leftarrow p + 1$
11:         if $W[i-1] \geq W[p] + w_i$ then     ▷ Compare the two possible options for item $i$
12:             $S[i] \leftarrow S[i-1]$     ▷ Better not to include item $i$
13:             $W[i] \leftarrow W[i-1]$
14:         else
15:             $S[i] \leftarrow S[p] \cup \{i\}$     ▷ Better to include item $i$
16:             $W[i] \leftarrow W[p] + w_i$
17:     return $S[n]$
```

The running time of this approach is on the order of $\Theta(n^2)$. However, we can make this faster. There are two problems we can address: One is to find the value 'p' faster in each iteration of the for loop. There are a couple approaches that work for this:

- If we have the intervals sorted by increasing end time, *and* have another array with the intervals sorted by increasing start time, then we can find the value of 'p' for *every* interval i in linear time. We can compute these values once, store them in an array, and refer to this array during the execution of the for loop.
- We can also do a binary search for p from among the values $0, 1, \ldots, i-1$.

Both approaches lead to an overall running time of $\Theta(n\log(n))$, except for the impact of the second problem.

The second problem is how we are storing the solutions. In particular, note that when we compute the value of $S[i]$, we are copying a set of potentially $\Omega(n)$ elements. Doing this leads to a running time of $\Omega(n^2)$ in the worst case. However, we are only interested in the value $S[n]$ at the output—if we can store the intermediate results more efficiently, then it's possible we can recover the value of $S[n]$ without having to explicitly write down all the values $S[i]$ for $i = 1, 2, \ldots, n$.

One way to do this is as follows: suppose that, instead of storing the whole solution set for each i, we only store a flag indicating whether we constructed $S[i]$ as $S[i-1]$ or as $S[p] \cup \{i\}$. Then we can recursively reconstruct $S[n]$ by simply following these flags: to construct $S[i]$, we look at the flag we stored; if the flag indicates "$S[i] \leftarrow S[i-1]$", then we recursively construct $S[i-1]$ and return that; otherwise, the flag indicates "$S[i] \leftarrow S[p] \cup \{i\}$", and so we recursively construct $S[p]$, add i, and return the result.

A pseudocode implementation of this is given in Algorithm 5. It clearly runs in time $\Theta(n\log(n))$.

Algorithm 5

Input: The n intervals given as $(s_1, e_1), \ldots, (s_n, e_n)$ with weights $w_1, \ldots, w_n$
Output: A conflict-free subset of intervals of maximum weight

```
1:  procedure WEIGHTED-INT-SCHED-DP-SLOW($s_i, e_i, w_i$)
2:      Sort the intervals in order of increasing end time
3:      $W[0, \ldots, n] \leftarrow$ new array of integers
4:      $U[1, \ldots, n] \leftarrow$ new array of booleans
5:      $p[1, \ldots, n] \leftarrow$ new array of integers
6:      $p[i] \leftarrow$ the value '$p$' from Algorithm 4
7:      $W[0] \leftarrow 0$
8:      for $i = 1, \ldots, n$ do
9:          $p \leftarrow p[i]$
10:         if $W[i-1] \geq W[p] + w_i$ then
11:             $U[i] \leftarrow$ false
12:             $W[i] \leftarrow W[i-1]$
13:         else
14:             $U[i] \leftarrow$ true
15:             $W[i] \leftarrow W[p] + w_i$
16:     $S \leftarrow \varnothing$
17:     $i \leftarrow n$
18:     while $i \neq 0$ do
19:         if $U[i]$ then
20:             $S \leftarrow S \cup \{i\}$
21:             $i \leftarrow p[i]$
22:         else
23:             $i \leftarrow i - 1$
24:     return $S$
```

3 Knapsack

Our next example of dynamic programming is the full knapsack problem. We have seen simplified variants of this in the notes on greedy algorithms, but, equipped with dynamic programming, we are now ready to approach the full version.

The full version of the knapsack problem is specified as follows:

Input: A set of n items with nonnegative integer weights $w_1, w_2, \ldots, w_n$, and an nonnegative integral knapsack capacity W.

Output: A subset of the items whose total weight is maximum subject to being at most W.

Let's focus on a slightly different problem, which will help us focus on the underlying dynamic programming ideas. The variant is as follows:

Input: A set of n items with nonnegative integer weights $w_1, w_2, \ldots, w_n$, and an nonnegative integral knapsack capacity W.

Output: A boolean, indicating whether there *exists* a subset of the items whose total weight is *exactly* W.

We now want to give a recursive algorithm which solves this decision version of the knapsack problem and which has few distinct calls in its recursion tree. One simple approach is to try to decide what to do with the last item—should it belong to an optimal solution or not? Since we don't know, we can try both possibilities. This leads to two subproblems:

- Decide whether there is a subset of the items $\{1, 2, \ldots, n-1\}$ of total weight exactly W. This corresponds to the possibility that n is not in a set whose weight is exactly W
- Decide whether there exists a subset of the items $\{1, 2, \ldots, n-1\}$ whose total weight, plus the weight of item n, is exactly W. This corresponds to the possibility that n is in a set whose weight is exactly W.

The first can already be expressed exactly as the exact same problem—we just recursively solve the instance with $n-1$ items of weights $w_1, \ldots, w_{n-1}$ and target weight W. On the other hand, the second does not exactly meet the specifications. However, we can fix this by realizing that we are equivalently asking for a subset of the first $n-1$ items whose weight is exactly $W - w_n$. Once we get a subset of this form, we can add item n to it, and get a subset of weight exactly W.

This is exactly the recurrence we will need for our dynamic program. It's easily seen to be correct, but as-formulated it's a very slow algorithm: the leaves of the recursion tree correspond to the subsets of $\{1, 2, \ldots, n\}$! However, note that every subproblem only considers sets of items that correspond to *prefixes* of $1, 2, \ldots, n$. In other words, the arguments to the recursive calls can be described by two parameters: a number n', indicating that we're considering items $1, 2, \ldots, n'$, and the value W', indicating that we're wanting a subset of weight exactly W'. If we think of the item weights as being fixed, globally accessible constants, this means that we can express our recursion using only the two parameters n' and W'. This is done in pseudocode in Algorithm 6.

It is then just an application of standard dynamic programming techniques to turn Algorithm 6 into an efficient algorithm. As an example, Algorithm 7 gives a bottom-up implementation, including code to meet the specification we defined earlier.

Algorithm 6

Input: The weights of n items $w_1, \ldots, w_n$ as globally accessible constants; a natural number n' and knapsack capacity W'

Output: A boolean, indicating whether there exists a subset of the first n' items whose weight is exactly W'

```
 1: procedure KNAPSACK-DECISION-REC($n', W'$)
 2:     if $n' = 0$ then
 3:         if $W' = 0$ then
 4:             return true
 5:         else
 6:             return false
 7:     PossibleWithLast ← false
 8:     if $w_{n'} \leq W'$ then
 9:         PossibleWithLast ← KNAPSACK-DECISION-REC($n' - 1, W' - w_{n'}$)
10:     PossibleWithoutLast ← KNAPSACK-DECISION-REC($n' - 1, W'$)
11:     return PossibleWithLast or PossibleWithoutLast
```

Algorithm 7 clearly runs in time $O(n \cdot W)$.[1] Its space usage is also on the order of $O(n \cdot W)$, which can be prohibitively large. As we did before though, we can reduce this. Note that entries in the i-th row of the table (entries of the form $T[i][\cdot]$) depend only on the entries of the $(i-1)$-th row of the table. Thus we just have to keep track of the most recent row—we don't need to store the whole table.

Finding a solution Let's now return to the original formulation of the knapsack problem, where the goal was to find a set of maximum weight that satisfies the capacity constraint instead of just deciding whether a set of a specific weight exists. The general idea will be to augment the algorithm we just covered with enough extra information to recover a full solution.

First of all, note that if we run the algorithm in the previous section and look at the resulting table, we can find the maximum possible weight of a set satisfying the capacity constraint. We can just check the table entry $T[n][w]$ for $w = W, W-1, \ldots$, and stop once we find the largest value w so that $T[n][w]$ is set to true. This extra step takes at most $O(W)$ time, so computing the table itself dominates this.

This also gives us a starting point for recovering a full solution. Suppose we started at the value w, *i.e.*, we know that $T[n][w]$ is set to true. Recall that when we computed $T[n][w]$, we looked at $T[n-1][w]$ and (if $w_n \leq w$) $T[n-1][w-w_n]$. Suppose we stored in the table not just a boolean indicating feasibility, but also some indication of which of $T[n-1][w]$ or $T[n-1][w-w_n]$ is set to true. Then we can use this to infer whether item n could belong to some solution set. In particular, if $w_n \leq w$ and the extra information indicated that $T[n-1][w-w_n]$ was set, then we could recursively find a solution set for $T[n-1][w-w_n]$, and then put the n-th item into this set. Otherwise, if the extra information indicated that $T[n-1][w]$ was set to the ture, then we could

[1] This is technically not polynomial time unless W is small. The item weights $w_1, \ldots, w_n$ and capacity W are given in binary in the general knapsack problem, so W could be very large—a truly polynomial-time algorithm would run in time polynomial in n and $\log(W)$. However, for small W (for instance, if the item weights and W are given in *unary*), then this is an efficient algorithm. Running times like this are generally referred to as 'pseudopolynomial'.

Algorithm 7

```
Input: The weights of $n$ items $w_1, \dots, w_n$ and knapsack capacity $W$
Output: A boolean, indicating whether there exists a subset of the items whose total weight is
    exactly $W$
1: procedure KNAPSACK-DECISION-DP($w_1, w_2, \dots, w_n, W$)
2:     $T[0, \dots, n][0, \dots, W] \leftarrow$ new 2-d array of booleans
3:     $T[0][0] \leftarrow$ true                          ▷ These handle the base case of the recursion
4:     for $W' = 1, \dots, W$ do
5:         $T[0][W'] \leftarrow$ false
6:     for $n' = 1, \dots, n$ do                            ▷ These implement the recursive steps
7:         for $W' = 0, \dots, W$ do
8:             $T[n'][W'] \leftarrow T[n'-1][W']$                 ▷ Consider subsets without item $n'$
9:             if $w_{n'} \le W'$ then                          ▷ Consider subsets with item $n'$
10:                $T[n'][W'] \leftarrow T[n'-1][W' - w_{n'}]$
11:    return $T[n][W]$
```

recursively find a solution set for $T[n-1][w]$, and simply output that set as a solution. In either case, the result is a set of total weight w, as desired. A pseudocode implementation of this is given in Algorithm 8.

Space Reduction As we've discussed, reducing the space used during a dynamic program is often possible (and desirable). In the case of knapsack, the space usage we can use depends on the exact application. Our current implementation, we are using $O(n \cdot W)$ space. If we are only interested in the decision version, then we can reduce the space usage to $O(W)$. The observation is that the entries $T[i][\,\cdot\,]$ depend only on the entries in $T[i-1][\,\cdot\,]$; thus we only need to remember a single row of T to compute the next row. On the other hand, if we want to recover the exact solution, then we need to keep most of U anyway, in which case the space usage is still $O(n \cdot W)$.

4 RNA Secondary Structure

Our next problem comes from computational biology. Recall that RNA strands can be thought of as strings of letters, representing the bases found along the strand. The letters are `A`, `C`, `G`, `U`, representing, respectively, adenine, cytosine, guanine, and uracil. The specific string of letters for an RNA strand is referred to as its *primary structure*.

RNA strands can fold in and bond on themselves similar to the bonding that forms the 'double-helix' structure of DNA. In particular, `A`'s can bond with `U`'s, and `C`'s can bond with `G`'s. The way in which a particular RNA strand tends to fold is referred to as its *secondary structure*. Understanding the secondary structure of RNA strands is a question biologists are interested in understanding, as different folding structures can affect the behavior of the strand within the cell.

With a well-defined model of how the secondary structure depends on the RNA strand, we can formulate this question as a computational problem. The specific model we will look at for this section is as follows:

- The secondary structure of an RNA molecule depends only on its primary structure. For

Algorithm 8

Input: The weights of n items $w_1, \ldots, w_n$ and knapsack capacity W
Output: A set S of items of maximum weight subject to having total weight at most W

```
1: procedure KNAPSACK-DP($w_1, w_2, \ldots, w_n, W$)
2:     $T[0,\ldots,n][0,\ldots,W] \leftarrow$ new 2-d array of booleans
3:     $U[0,\ldots,n][0,\ldots,W] \leftarrow$ new 2-d array of booleans        ▷ The ‘extra information’
4:     $T[0][0] \leftarrow$ true
5:     for $W' = 1,\ldots,W$ do
6:         $T[0][W'] \leftarrow$ false
7:     for $n' = 1,\ldots,n$ do
8:         for $W' = 0,\ldots,W$ do
9:             $T[n'][W'] \leftarrow T[n'-1][W']$
10:            $U[n'][W'] \leftarrow$ false                                ▷ Do NOT use item $n'$
11:            if $w_{n'} \leq W'$ then
12:                $T[n'][W'] \leftarrow T[n'-1][W' - w_{n'}]$
13:                $U[n'][W'] \leftarrow$ true                             ▷ Actually, DO use item $n'$
14:    $w \leftarrow W$
15:    while $\neg\, T[n][w]$ do                                           ▷ $T[n][0]$ is always set to true
16:        $w \leftarrow w - 1$
17:    $S \leftarrow \varnothing$
18:    for $i = n \ldots 1$ do
19:        if $U[i][w]$ then
20:            $S \leftarrow S \cup \{i\}$
21:            $w \leftarrow w - w_i$
22:    return $S$
```

instance, knowing an RNA molecule has primary structure “AGCGU” is sufficient to determine its secondary structure. Thus we can represent the input to our problem as just a string of letters from the alphabet A, C, G, U.

- Molecules A have the potential to bond with any U, and molecules G have the potential to bond with any C, except as constrained below. There are no further possible bonds.
- No single molecule can bond with more than one molecule. A secondary structure may include molecules which do not bond.
- If two molecules are within five positions of each other, (*e.g.*, the A and U in AGCU, but not the A and U in AGCCGCU), then they *cannot* bond. This is intended to model the flexibility of the RNA strand—it can’t bend too sharply on itself.
- There are no *twists* in the secondary structure. A twist happens when a position i_1 bonds with a position j_1, and a position i_2 bonds with position j_2, and $i_1 < i_2 < j_1 < j_2$. (The name comes from imagining the molecules as being studs on one face of a belt, where bonding molecules corresponds to having the corresponding studs touch physically. The above condition is equivalent to needing to twist the belt to realize the given bonds.)

◦ A ‘typical’ secondary structure is one with the maximum number of bonds present in the secondary structure.

We can formulate the corresponding computational problem as follows:

Input: A string S of length n whose i-th character is denoted $S(i)$, with $i = 1, 2, \ldots, n$, and which has letters from the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}$.

Output: A set of pairs (i, j) indicating that the molecule in position i should be matched to the molecule in position j. These pairs should satisfy the constraints outlined above.

The input corresponds to the primary structure of an RNA molecule, and the output corresponds the ‘typical’ secondary structure of the input molecule.

An algorithm which solves this efficiently, in time $O(n^3)$, uses dynamic programming. To see the recursive approach, let’s start by simply trying to compute the number of bonds in the optimal solution. Usually this is the primary difficulty of finding a dynamic programming approach to a problem, and recovering the actual solution typically follows as a natural extension to a solution to this simpler problem. This will be the case for us here as well.

Next, let’s consider a first decision we can make. We know that the n-th molecule on the input strand has to do one of a few things: either it goes entirely unmatched, or else it matches with some later position. In the former case, the problem reduces to a new instance of itself on a string of shorter length, so recursion will handle this case perfectly.

In the latter case, we actually get a similar sort of behavior. Suppose we match the n-th molecule to the molecule in position k. Because of the ‘no twists’ rule, we can infer that there can be no matches between a molecule in positions $1 \cdots k-1$ and a molecule in positions $k+1 \cdots n-1$. In other words, we can recursively solve the problem on the strings $S(1, \ldots, k-1)$ and $S(k+1, \ldots, n-1)$, and simply combine the resulting solutions with the n, k matching to get a final solution. Since we’re only interested in the value of the optimal solution, this just means adding the value of the optimal solution for $S(1, \ldots, k-1)$ and the value of the optimal solution for $S(k+1, \ldots, n-1)$, and then adding one for the n, k matching.

Thus if we take a maximum over all these possibilities (one for each choice of $k = 1, \ldots, n-1$, and the possibility of not matching the n-th molecule, and ignoring any invalid choices), then we will have computed the optimal value of our original instance. Transforming this into pseudocode, we have Algorithm 9.

Notice that the recursive calls are all contiguous substrings of the original input string. Thus we can parametrize the recursive calls by just the two indices corresponding to the left-most and right-most endpoints of the contiguous substring. Applying a standard memoization technique then yields an algorithm which constructs a table of $O(n^2)$ size, and using $O(n)$ time per table entry, yielding an $O(n^3)$-time algorithm overall. For completeness, Algorithm 10 gives a bottom-up implementation of Algorithm 9.

Algorithm 9

Input: A string $S(1 \dots n)$ of length n.

Output: The maximum number of bonds, subject to the constraints in the RNA Secondary Structure problem description

```
1: procedure RNA-Secondary-Structure-Rec($S$)
2:     if $n \leq 1$ then
3:         return 0
4:     OPT $\leftarrow$ RNA-Secondary-Structure-Rec($S(1, \dots, n-1)$)
5:     for $k = 1 \dots n-1$ do
6:         if $S(k)$ and $S(n)$ can bond then
7:             Left $\leftarrow$ RNA-Secondary-Structure-Rec($S(1, \dots, k-1)$)
8:             Right $\leftarrow$ RNA-Secondary-Structure-Rec($S(k+1, \dots, n-1)$)
9:             OPT $\leftarrow \min(\text{OPT}, 1 + \text{Left} + \text{Right})$
10:    return OPT
```

Algorithm 10

Input: A string $S(1 \dots n)$ of length n.

Output: The maximum number of bonds, subject to the constraints in the RNA Secondary Structure problem description

```
1: procedure RNA-Secondary-Structure-DP($S$)
2:     OPT$[1 \dots n][0 \dots n]$ $\leftarrow$ new array of integers
3:     for $i = 1 \dots n$ do                          ▷ The length-0 substrings are the base case
4:         OPT$[i][i-1] \leftarrow 0$
5:     for $\ell = 1 \dots n$ do                                ▷ For each length of an interval
6:         for $i = 1 \dots n - \ell + 1$ do                  ▷ For each starting point of an interval
7:             $j \leftarrow i + \ell - 1$                       ▷ $j$ is the right side of the interval
8:             OPT$[i][j] \leftarrow$ OPT$[i][j-1]$                         ▷ Consider not using $j$
9:             for $k = i \dots j-1$ do                              ▷ Consider pairing $j$ with $k$
10:                if $S(k)$ and $S(j)$ can bond then
11:                    Left $\leftarrow$ OPT$[i][k-1]$
12:                    Right $\leftarrow$ OPT$[k+1][j-1]$
13:                    OPT$[i][j] \leftarrow \max(\text{OPT}[i][j], 1 + \text{Left} + \text{Right})$
14:    return OPT$[1][n]$
```

5 Sequence Alignment

Our next problem is that of sequence alignment. Here the idea is to measure the 'edit distance' between two strings as a way of measuring their similarity. For instance, the string `theat` is, in some sense, close to each of the words `that`, `heat`, `teat`, `threat`, `treat`, and even the pair of words `the at`, but not close to a word like `zipper` or `rutabaga`. Specifically, changing the word `theat` to `that` requires only removing the `e`, while changing `theat` to `rutabaga` is a more involved effort.

We can formalize this as follows: let $a = a_1 \cdots a_n$ and $b = b_1 \cdots b_n$ be two strings of symbols. The *edit distance* between a and b can be defined as the smallest number of insertions, deletions, or symbol changes needed to turn the string a into the string b. For instance, we can turn `theat` into `heat` by removing the first `t`, or turn `theat` into `treat` by changing the `h` to an `r`, so the edit distances for these pairs of strings is one. On the other hand, `theat` can be changed to `zipper` by changing `th` to `zipp` and `at` to `r` at a cost of three symbol changes, two insertions, and one deletion; or by simply turning `theat` into `zippe` and adding an `r` for a total of five symbol changes and one insertion.

The edit distance between words has obvious applications to spell-checking and related tasks, such as suggesting an alternative interpretation of a search query. It's also used in biology as a way of measuring the similarity between two samples of a genetics sequence. For example, if a biologist is sequencing plant DNA, has a guess at the DNA sequence, and wants to eliminate any samples coming from other sources (*e.g.*, insects or lab technicians), she can use the edit distance between her guess and her sample as a way of measuring the likelihood that a given sample comes from the plant DNA rather than any confounding sources.

In all these applications, some variation on the basic notion of edit distance is useful. For instance, following the biology example above, the biologist may have varying degrees of confidence in different parts of her guess. By charging a higher cost to edits in high-confidence portions of the guess sequence, then the edit distance becomes better at predicting useful samples. The dynamic programming approach we give below handles many variations of the cost function without problem. For simplicity, we focus only on the case where all the edits have identical cost. Trying to see how to modify the algorithm for different ways of charging for edits is a good way to improve understanding of this algorithm.

Let's now formally state the problem of computing edit distance:

Input: Two strings, $a = a_1a_2 \cdots a_n$ and $b = b_1b_2 \cdots b_m$ of length n and m respectively.

Output: The fewest number of edits (insertions, deletions, symbol changes) needed to turn the string a into the string b.

The algorithm we will give computes this value in time $O(n \cdot m)$ and can be implemented using space $O(\min(n, m))$. We will later extend this algorithm to one which computes an optimal sequence of edits (rather than just the optimal number of them), also in time $O(n \cdot m)$ and space $O(\min(n, m))$.

Recursive approach The recursive approach which will yield the algorithm alluded to above is straightforward: we can simply begin by considering the operations performed on the symbols a_n and b_m. There are four possibilities: a_n was deleted, b_m was added, a_n was turned into b_m, or none of the above, in which case $a_n = b_m$. These are not necessarily mutually exclusive—different optimal ways of turning a into b may use different operations with a_n and b_m. But we do know that one of these four cases has to happen. Let's see how each of these cases naturally reduces to computing the edit distance of simpler strings.

Consider the first case, in which some optimal solution turns a into b by, at some point, deleting a_n. We can assume that this optimal solution deletes a_n as its final operation. This means that it turns a into a string of the form $b_1 \cdots b_i a_n b_{i+1} \cdots b_m$, and then deletes a_n. Since a_n is the last character of a, all of the operations that left $b_{i+1} \cdots b_m$ in this string were additions; this means that we could equivalently add $b_{i+1} \cdots b_m$ before a_n, yielding a string of the form $b_1 \cdots b_m a_n$. Thus the optimal solution gives an optimal way of turning the string $a_1 \cdots a_{n-1} a_n$ into the string $b_1 \cdots b_m a_n$, or equivalently turns the string $a_1 \cdots a_{n-1}$ into the string $b_1 \cdots b_m$. Therefore the total number of edits it makes is simply the edit distance from $a_1 \cdots a_{n-1}$ to $b_1 \cdots b_m$, plus one to delete a_n.

Consider now the second case, in which some optimal solution turns a into b by, at some point, adding b_m. Note that we can reverse all the operations to get an equivalent way of turning b into a. (Similarly, any way of turning b into a yields a way of turning a into b.) In this case, 'adding b_m' translates to a deletion operation; this means this second case is essentially identical to the first case, where now we want to compute the edit distance between $a_1 \cdots a_n$ and $b_1 \cdots b_{m-1}$ recursively, and then add one to account for the addition of b_m.

Now consider the third case, in which some optimal solution turns a into b by changing the symbol a_n to b_m. Using an argument similar to the one presented in the first case, it's easy to see that these solutions are equivalent to solutions that first transform $a_1 \cdots a_n$ into $b_1 \cdots b_{m-1} a_n$, and then change a_n to b_m. Thus the optimal cost of such solutions is simply the edit distance between $a_1 \cdots a_{n-1}$ and $b_1 \cdots b_{m-1}$, plus one to change a_n to b_m.

Finally, consider the case where both a_n and b_m were not changed, and in which $a_n = b_m$. It's again easy to see that this is equivalent to a sequence of operations in which $a_1 \cdots a_{n-1}$ is transformed into $b_1 \cdots b_{m-1}$.

Since an optimal solution must fall into one of these cases, and each case can be handled recursively, we have a recursive algorithm. Pseudocode is given in Algorithm 11.

Algorithm 11

Input: Two strings, $a = a_1 \cdots a_n$ and $b = b_1 \cdots b_m$
Output: The edit distance between a and b

```
1:  procedure Sequence-Alignment-Rec($a, b$)
2:      if $n = 0$ then
3:          return $m$
4:      else if $m = 0$ then
5:          return $n$
6:      else
7:          Case1 ← 1+Sequence-Alignment-Rec($a_1 \cdots a_{n-1}, b$)
8:          Case2 ← 1+Sequence-Alignment-Rec($a, b_1 \cdots b_{m-1}$)
9:          Case3 ← 1+Sequence-Alignment-Rec($a_1 \cdots a_{n-1}, b_1 \cdots b_{m-1}$)
10:         if $a_n = b_m$ then
11:             Case4 ← Sequence-Alignment-Rec($a_1 \cdots a_{n-1}, b_1 \cdots b_{m-1}$)
12:         else
13:             Case4 ← $+\infty$
14:         return min( Case1, Case2, Case3, Case4 )
```

Notice that the subproblems that arise always have the 'a' parameter an initial substring of the input a and similarly for the 'b' parameter. This means the total number of distinct recursive calls is bounded by $(n+1)\cdot(m+1) = O(n\cdot m)$. Thus, to get an efficient algorithm, we create a memoization table with $n+1$ rows and $m+1$ columns, where the (i,j)-th entry corresponds to the recursive call whose parameters are the length-i prefix of a and the length-j prefix of b. The (i,j)-th entry of the table depends on the $(i-1,j)$-th, $(i,j-1)$-th, and $(i-1,j-1)$-th entries, so it only takes constant work per cell to compute the entire table. Furthermore, a bottom-up approach can fill in the entries row-by-row, column-by-column, or even diagonal-by-diagonal. A space-conscious approach can compute the edit distance in space $O(\min(n,m))$.

Recovering the optimal edits Using the same ideas as we have already seen, it is straightforward to modify the above dynamic programming algorithm to also compute the optimal sequence of edits. A naïve approach is just to store, in each cell of the table a value indicating which of the four cases encodes the optimal solution.

However doing it this way causes the space complexity to blow back up to $O(n\cdot m)$. It turns out that one can actually recover the optimal solution in space $O(n+m)$ while still keeping the time complexity at $O(n\cdot m)$. (Note that, on some strings, the number of operations needed is $\max(n,m)$, so because $\max(n,m) = \Theta(n+m)$, this space bound is essentially the best we can do.) For simplicity of notation, we'll assume that $n \le m$, since the other case is symmetric. We'll say that the rows of the memoization table correspond to the $n+1$ substrings of a, and that the columns correspond to the $m+1$ substrings of b; our assumption that $n \le m$ simply says this matrix is wider than it is tall.

The general strategy that we'll use is an application of divide and conquer. We can treat the optimal solution found by our recursive procedure as being a path from the $(0,0)$-th entry to the (n,m)-th entry; our goal is to compute this path. Focus on the $m/2$-th column. Suppose we can find some row i^* for which we know that the path corresponding to the optimal solution passes through the cell $(i^*, m/2)$, and suppose that we can do so in time $O(n\cdot m)$ and space $O(\min(n,m))$. Then we can fill in that part of the path. We will also be able to recursively compute the path on the block of the table between $(0,0)$ and $(i^*, m/2)$ and the block of the table between $(i^*, m/2)$ and (n,m). Doing things this way saves us space, because we can *reuse* the space from the first recursive call in the second recursive call. Furthermore, it won't cost us much time, because as the recursion depth increases, we will see that the total work done at a given level of recursion decays geometrically.

What is (arguably) the most interesting aspect of this is how we find the row i^* in the constraints stated above. It turns out that we can do so by appealing directly to the dynamic program we derived straight from Algorithm 11! The procedure we covered in Algorithm 11 only returns the actual edit distance, not the sequence of edits that witnesses this. However, we will see that this is actually sufficient to find the row i^*.

(Forthcoming: more details!)

6 Shortest Paths, revisited

(Forthcoming: the Bellman-Ford algorithm for single-source shortest graphs)

7 All-Pairs Shortest Paths

In the previous section, we gave the Bellman-Ford algorithm, which computes shortest paths from a single source vertex to every other vertex in the input graph. Sometimes, we are interested in knowing this information for *every* source vertex, rather than a single one. A naïve approach to this is simply to run the Bellman-Ford algorithm (or even Dijkstra's algorithm, if it's applicable) once for each choice of start vertex. In the case of Bellman-Ford, this yields an algorithm running in time $O(n^2m)$, and in the case of Dijkstra's algorithm (with a heap-based priority queue), this leads to a running time of $O(n \cdot (n+m)\log(n)) = O(nm\log(n) + n^2\log(n))$.

On the other hand, shortest paths have nice structure to them. In other words, it's possible for information about shortest paths from s to help us compute shortest paths from s'. Can we exploit this to yield an overall faster algorithm? The answer is yes (sort-of). The Floyd-Warshall algorithm is a dynamic programming approach to the all-pairs shortest path problem, and runs in time $O(n^3)$. This is faster than the naïve approach with the Bellman-Ford algorithm when $m = \Omega(n)$ (*e.g.*, when the input graph is connected). It is even faster than the naïve approach with Dijkstra's algorithm, as long as $m = \Omega(n^2/\log(n))$, (*e.g.*, complete, or almost-complete graphs) and doesn't require the edge lengths be nonnegative. Even better, one of the most salient features of the Floyd-Warshall algorithm is how simple it is to implement in code: look ahead at Algorithm 12 for an idea.

Before diving into the details, let's first formally state the all-pairs shortest paths problem. We focus on just computing the lengths of the shortest paths; the approach we give will easily extend to finding all of the shortest paths themselves, and we will sketch this after presenting the algorithm.

Input: A directed graph $G = (V, E)$, for which the vertices are $V = \{1, 2, \dots, n\}$, and edge-length function $\ell : E \to \mathbb{R}$.

Output: For each pair of vertices s and t, the length of a shortest path from s to t (or $+\infty$ if there is no s-t path, or $-\infty$ if there is an s-t path but no shortest one).

We will represent the output as a table whose rows and columns are indexed by V. We will also focus for intuition's sake on the case where G is strongly-connected, and there are no negative cycles. The approach we give will naturally handle these possibilities.

The key insight for the Floyd-Warshall algorithm is the following characterization of shortest paths: Let s and t be vertices for which there exists a shortest s-t path. Then every shortest s-t visits the vertex n some (possibly zero, possibly more) number of times. This means we can decompose it into a series of paths $P_1, P_2, \dots, P_k$, where P_1 is a shortest path from s to n, P_k is a shortest path from n to t, and P_i for $i \neq 1, n$ is a shortest path from n to itself. Consider how the cycle P_i with $i \neq 1, k$ must look: if they have nonnegative total length, then we can remove them and obtain a path from s to t which is no longer; if they have negative total length, then we can repeat the cycle again, and obtain a path from s to t which is shorter. Since our path $P_1 P_2 \cdots P_k$ is a shortest path, this latter case can't happen.

Thus we actually know that, if there is a shortest path from s to t, then there is a shortest path which visits the vertex n zero or one time. Furthermore, if such a shortest path visits the vertex n, then we can decompose the path into shortest paths, one from s to n and one from n to t. Thus we can find a shortest path from s to t by considering shortest paths that don't visit vertex n, and by considering shortest paths from s to n and from n to t which also don't visit vertex n except at the end points.

This motivates the following definition: let $S(s,t,k)$ denote the length of a shortest path from s to t using only the vertices $\{1,2,\ldots,k\}$ as intermediate vertices. Our goal is to compute the value $S(s,t,n)$ for all vertices s and t. The above discussion tells us that

$$S(s,t,n) = \min(S(s,t,n-1),\ S(s,n,n-1) + S(n,t,n-1))$$

This generalizes to give the following recurrence, valid for $k \geq 0$ and vertices s and t such that there is either no path from s to t or there is a shortest path from s to t.

$$S(s,t,k+1) = \min(S(s,t,k),\ S(s,k+1,k) + S(k+1,t,k))$$

Additionally, we can compute the base case ($k=0$) very easily:

$$S(s,t,0) = \begin{cases} \ell(s,t) & : \quad (s,t) \in E \\ 0 & : \quad s = t \\ +\infty & : \quad \text{otherwise} \end{cases}$$

These two things can be easily computed.

What remains is computing the values of $S(s,t,k)$ when there is an s-t path, but no shortest s-t path. In this case, we can compute the above recurrence as-stated for all pairs s and t, and $k = 1,\ldots,n$. For vertices v on a negative cycle, we will then see that $S(v,v,n) < 0$. Thus, to check whether $S(s,t,n)$ ought to be $-\infty$, we can simply check for a vertex v so that v is on a negative cycle ($S(v,v,n) < 0$), v is reachable from s ($S(s,v,n) < +\infty$) and t is reachable from v ($S(v,t,n) < +\infty$).

As the recursive implementation of this procedure is obvious, we skip straight to the bottom-up implementation given in Algorithm 12.

Recovering the actual paths We now cover the modification for recovering the actual shortest paths. The key idea here is in figuring out a good output representation. Recall that when we covered Dijkstra's algorithm, we stored the shortest path in an array we called "Parent", which was indexed by the vertices. This array encoded shortest paths from a single source to *every* vertex.

We can use a similar output format here; however, it's slightly more intuitive to call the output array "Next". 'Next' is a two-dimensional array, with both rows and columns indexed by the vertices of the input graph. Next$[s][t]$ contains the vertex which is *next* on a shortest path from s to t.

This can be used to recover a shortest path from s to t as follows: simply take the vertices s, Next$[s][t]$, Next[Next$[s][t]][t]$, etc. We can also compute the Next array while we compute the value S above. A straightforward modification of Algorithm 12 yields Algorithm 13, which computes the Next array.

Algorithm 12

Input: A graph $G = (\{1, 2, \ldots, n\}, E)$ and edge-length function $\ell : E \to \mathbb{R}$

Output: A table $S[s][t]$ containing $\pm\infty$ or a real number, according to the specification of the all-pairs shortest paths problem

```
1: procedure FLOYD-WARSHALL($G$)
2:     $S[\cdot][\cdot] \leftarrow$ new table with rows and columns indexed by $V$, initialized to $+\infty$
3:     for $s$ in $V$ do
4:         $S[s][s] \leftarrow 0$
5:     for $(s, t)$ in $E$ do
6:         $S[s][t] \leftarrow \ell(s, t)$
7:     for $k = 1, \ldots, n$ do                ▷ Compute $S(s, t, k)$ for $k = 1 \ldots n$ in place
8:         for $s = 1, \ldots, n$ do
9:             for $t = 1, \ldots, n$ do
10:                $S[s][t] \leftarrow \min(\ S[s][t],\ S[s][k] + S[k][t]\ )$
11:    for $s = 1, \ldots, n$ do                ▷ Now check for paths with negative cycles
12:        for $t = 1, \ldots, n$ do
13:            for $v = 1, \ldots, n$ do
14:                if $S[s][v] < +\infty$ and $S[v][v] < 0$ and $S[v][t] < +\infty$ then
15:                    $S[s][t] \leftarrow -\infty$
16:    return $S$
```

Algorithm 13

Input: A graph $G = (\{1, 2, \ldots, n\}, E)$ and edge-length function $\ell : E \to \mathbb{R}$
Output: S, as before, and a table Next$[\cdot][\cdot]$ as described above

```
procedure FLOYD-WARSHALL-PATHS($G$)
    $S[\cdot][\cdot] \leftarrow$ new table with rows and columns indexed by $V$, initialized to $+\infty$
    Next$[\cdot][\cdot] \leftarrow$ new 2-d array with rows and columns indexed by $V$
    for $s$ in $V$ do
        $S[s][s] \leftarrow 0$
        Next$[s][s] \leftarrow s$
    for $(s, t)$ in $E$ do
        $S[s][t] \leftarrow \ell(s, t)$
        Next$[s][t] \leftarrow t$
    for $k = 1, \ldots, n$ do
        for $s = 1, \ldots, n$ do
            for $t = 1, \ldots, n$ do
                if $S[s, k] + S[k, t] < S[s][t]$ then
                    $S[s][t] \leftarrow S[s][k] + S[k][t]$
                    Next$[s][t] \leftarrow$ Next$[s][k]$
    for $s = 1, \ldots, n$ do
        for $t = 1, \ldots, n$ do
            for $v = 1, \ldots, n$ do
                if $S[s][v] < +\infty$ and $S[v][v] < 0$ and $S[v][t] < +\infty$ then
                    $S[s][t] \leftarrow -\infty$
    return $S$, Next
```