1 Dynamic Programming

The topic of this section is dynamic programming: a basic and powerful technique for reusing already-computed answers to greatly speed up computations. We start by looking at the Fibonacci sequence through this lens, because the Fibonacci numbers are a familiar example that will make it clear the two sides of the dynamic programming transformation. First, we consider an unreasonably slow algorithm that pedantically applies the definition of the Fibonacci recurrence, and then we do the natural thing and instead compute the Fibonacci numbers using a table, traversed by a single *for* loop. This transformation between 1) an exponential-time wasteful algorithm, and 2) a much faster algorithm that reinterprets the same recurrence via a table, is the essence of dynamic programming.

2 Fibonacci

Problem (Warm-up)
How can we compute Fibonacci numbers?

Starting with the definition of the Fibonacci numbers,

$$F(n) = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

We can write the equivalent recursive function in Matlab (create a new file in Matlab by typing `edit`, and save the following code as `fib.m`).

```
function out=fib(i)
if i==0, out=0;
elseif i==1, out=1;
else out=fib(i-1)+fib(i-2);
end
```

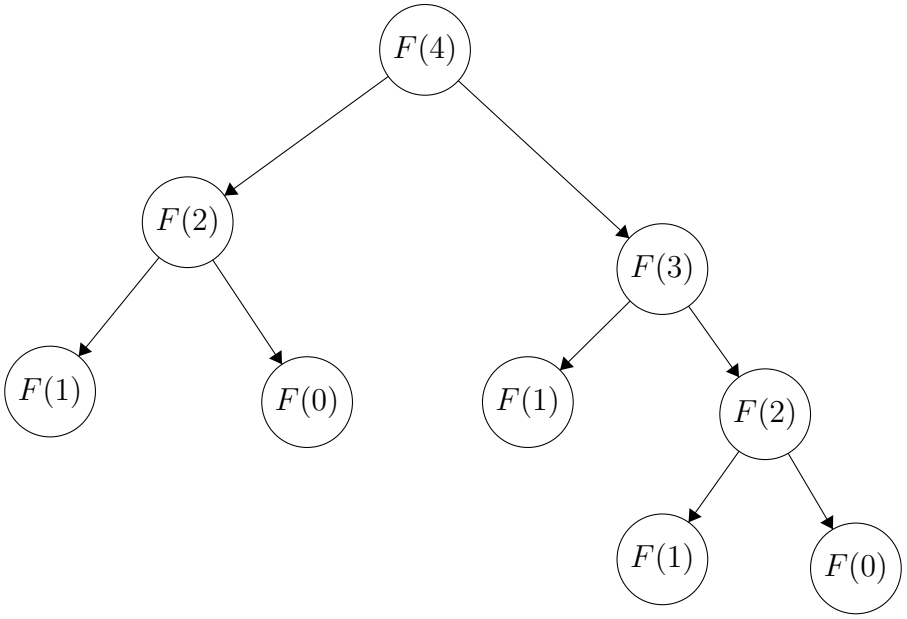
How much time does this take?

Three students suggest:

$$2^n,\ n!,\ \text{Fib}\,(n),$$

with 2^n being a quite popular answer with many people firmly convinced. It's a good habit to draw the recursive tree.

Consider the tree that comes from evaluating $F(4)$—there are 9 nodes, but 9 is not power of 2 nor a factorial nor a Fibonacci number!



Note. *One of the most dangerous things is to fool yourself—keep an eye out for when you convince yourself of something wrong, try to figure out how not to fall into that trap again, and discuss this with other students or a TA! This is one of the most valuable kinds of learning experiences you will have in this course.*

A student pointed out that the time taken by this pseudocode isn't literally the number of nodes in the tree—leaves may take a different amount of time from internal nodes. In general this leads to a recurrence for time of

$$T(n) = T(n-1) + T(n-2) + c,$$

where c is how much time it takes to process an internal node of the tree; the base cases correspond to how much time it takes to process a leaf. See the pdf notes here for a discussion of these recurrences.

3 Edit Distance

The "edit distance" problem, phrased informally, asks what is the best way to edit one word into another word, given that we pay a cost of 1 per insertion, 1 per deletion, and 1 to edit a letter into another letter. This problem shows up in many forms—spell checking is essentially trying to find the dictionary word that has the least edit distance to what you typed. The field of bioinformatics uses edit distance algorithms as one of its main workhorses, because computing the edit distance between two (very long) strings of DNA tells us something about the similarity of the two creatures from which the DNA came.

Problem
What is the minimum edit distance between "sunny" and "snowy"?

The first question we need to ask is: are we transforming the first word into the second, or the second into the first? What exactly is the output we're expecting? When we get confused later, firming up the problem statement and the input/output relation will help a lot.

The most natural approach is to describe how to edit the first word to the second: delete the "u" from the 3rd location in the string, change the "n" in the 4th location to an "o", and insert a "w" before the 5th location in the string ("y").

Alternatively, we could describe how to edit the second word into the first: insert a "u" after the first letter, change "o" to an "n", and delete the "w".

As it turns out, there are some problems with this perspective on edit distance, for a few reasons:

1. Editing the first word into the second is a bit different from editing the second into the first, yet these two directions also seem like they should be the same.

2. Expressing edit distance like this makes it unclear exactly which decisions the algorithm should make, and in what order—inserting the "w" is also tied to the decision to do this before the 5th character, and should this decision happen before or after the decision to delete the "u"?

What we want is a notion of edit distance such that our algorithm makes a series of decisions, and that these decisions collectively describe a clear strategy for making the edits.

3.1 Edit Distance Defines an Alignment

Think of writing the two words ("sunny" and "snowy"), one on top of the other; instead of asking how to move from the top word to the bottom, or from the bottom to the top, we instead take a new perspective and analyze these words from left to right.

Consider the following *alignment* of the two words.

```
s u n n _ y
s _ n o w y
```

See how each character is "aligned" with a corresponding character from the other word: letters that are unchanged between the two words are written one above the other, as are letters that we choose to edit (such as editing the second "n" in "sunny" into an "o" for "snowy"); letters that we delete from the first word are aligned with a gap in the second

word (like the "u" in "sunny" that gets deleted); symmetrically, the same happens to letters that we delete from the second word (like the "w" in "snowy").

Such an alignment has a cost of 1 per character that is aligned with a non-matching character (including gaps).

This alignment is a representation of a strategy to edit the first word into the second (and, even, to edit the second word into the first). Explicitly: starting with "sunny", keep the first character ("s"), delete the second character ("u"), keep the third character ("n"), change the fourth character ("n") into the next character from the second word ("o"), insert the next character from the second word ("w"), and keep the fifth character ("y").

This alignment is just one of many conceivable alignments. Below are two different alignments, with costs 10 and 4, respectively:

```
s u n n y _ _ _ _ _
_ _ _ _ _ s n o w y

s u n n _ _ y
s _ n _ o w y
```

The algorithmic challenge is to find the alignment with the lowest cost. In this manner, we have made the somewhat nebulous challenge of "find the edit distance" precise.

3.2 An Algorithmic Approach to Alignment

At each step of an alignment, there are just 3 possibilities: 1) use a letter from both words; 2) use a letter from the top word (putting a gap on the bottom of the alignment here); and 3) use a letter from the bottom word (putting a gap on the top of the alignment here). In our example alignment of "sunny" and "snowy",

```
s u n n _ y
s _ n o w y
```

the decisions, from left to right, are of types 1 (using letters "s" from both words), 2 (aligning "u" with a gap), 1 (using letters "n" from both words), 1 (using letter "n" from the first word but the *different* letter "o" from the second word), 3 (aligning "w" from the second word with a gap), and 1 (using a "y" from both words). All we need to do to write a correct algorithm, is allow our algorithm to choose among sequences of decisions, so that it might identify "121131" as representing the best edit strategy.

It is important to keep in mind that alignments of types 2 and 3 each have cost 1, whereas the cost of an alignment of type 1 depends on the letters in the words: it has cost 0 if the letters match, and cost 1 otherwise.

At this point, the recursive algorithm almost writes itself, after we make a few bookkeeping decisions. We will process the alignment such that at any moment, the algorithm is being asked: "find a good alignment between the first i characters in word1 and the first j characters in word2". The three possible decisions for what to do at each step are as follows:

1. using a letter from both words corresponds to decreasing both i and j by 1;
2. using a letter from the top word corresponds to decreasing i by 1 but leaving j (the position in word2) unchanged;
3. using a letter from the bottom word corresponds to leaving i unchanged but decreasing j (the position in word2) by 1.

Thus, the recursive structure of our algorithm is that when `editdistance`(i, j) is called it should make three recursive calls: `editdistance`$(i-1, j-1)$, `editdistance`$(i-1, j)$, and `editdistance`$(i, j-1)$, processing each of the possible cases 1, 2, and 3 above, respectively.

To complete the definition of this recursive algorithm, we must specify the base cases, in which we skip the above recursive approach and just output the answer directly. Perhaps the most natural base cases for this problem are to note that when one word has 0 characters, the edit distance is equal to the length of the other word.

We can now write a complete recursive definition of edit distance:

$$\texttt{editdistance}(i,j) = \begin{cases} j & \text{if } i = 0 \\ i & \text{if } j = 0 \\ \min\{ & \\ \quad \texttt{editdistance}(i-1, j-1) + [\text{word1}(i) \stackrel{?}{\neq} \text{word2}(j)], & \\ \quad \texttt{editdistance}(i-1, j) + 1, & \\ \quad \texttt{editdistance}(i, j-1) + 1 & \\ \} & \text{otherwise} \end{cases}$$

where the nonstandard notation "$[a \stackrel{?}{\neq} b]$" represents a 1 when $a \neq b$ and 0 when $a = b$.

It is very important to keep in mind the *meaning* of each thing we compute: `editdistance`(i, j) computes the *cost of the best way to edit the first i characters of word1 into the first j characters of word2*.

We can convert this recursive definition into Matlab code, that you can try running yourself. We need to pass "word1" and "word2" into the function, but otherwise, the Matlab code looks *very* similar:

```
function dist=editdist(word1,word2,i,j)
if i==0, dist=j;
elseif j==0, dist=i;
else dist=min([editdist(word1,word2,i-1,j-1)+(word1(i)~=word2(j)),...
        editdist(word1,word2,i-1,j)+1,...
        editdist(word1,word2,i,j-1)+1]);
end
```

(In Matlab, "not equals to" is denoted "`~=`". The Matlab "[" and "]" operators concatenate what is inside into a 3-element vector, which the `min` function can then find the minimum element of.)

3.3 A Dynamic Programming Approach to Edit Distance

Just as we saw with the Fibonacci sequence, our recursive approach to edit distance ends up recomputing the same subproblems many many times. Explicitly, when we are trying to figure out how to edit "sunny" into "snowy", we start the recursion by calling `editdistance`$(5,5)$, (expressing that we want to find a way to edit the first 5 letters of "sunny" into the first 5 letters of "snowy") which then makes three recursive calls to `editdistance`$(4,4)$, `editdistance`$(4,5)$, and `editdistance`$(5,4)$. Each of these 3 recursive calls makes 3 recursive calls themselves, and the issue is that many of the resulting calls deep in the recursive tree are redundant. The total number of calls spawned by this algorithm is exponential; however, crucially, the value of each call `editdistance`(i,j) depends only on the pair (i,j), and there are very few total possibilities for this pair. To compare word1 of length n with word2 of length m, we see that i can vary between 0 and n, while j can vary between 0 and m for a total of $(n+1)\cdot(m+1)$ possibilities for the pair (i,j), which is much better than anything exponential in n or m.

Thus, similarly to the intuitive way to compute Fibonacci numbers efficiently, the natural thing to do here is to make a *table* that we fill in with all possible results for `editdistance`(i,j), starting with small values of i and j, and filling out each entry of the table in terms of entries that we have already computed.

This is the essence of dynamic programming! Make sure you understand the relation between these two ways of looking at the problem: first, as a natural recursion that descends into subproblems, methodically trying all possibilities; second, as filling in a table with exactly the outputs of the previous recursion, though without ever wastefully recomputing the answer to the same subproblem.

The incredibly efficient dynamic programming algorithm to compute edit distance, remarkably, will look almost identical to the natural recursive definition above, except for two changes: 1) we change the function call "`editdistance`(i,j)" into a two-dimensional array access "T[i,j]"; and 2) we traverse through all i and j with a "`for`" loop, filling in each entry of the table in order.

```
for i=0:length(word1)
    for j=0:length(word2)
```

$$\texttt{T}[i,j] = \begin{cases} j & \text{if } i = 0 \\ i & \text{if } j = 0 \\ \min\{ & \\ \quad \texttt{T}[i-1,j-1] + [\text{word1}(i) \stackrel{?}{\neq} \text{word2}(j)], & \\ \quad \texttt{T}[i-1,j] + 1, & \\ \quad \texttt{T}[i,j-1] + 1 & \\ \} & \text{otherwise} \end{cases}$$

Go back and see how similar this code is to the recursive definition of `editdistance` above!

(In practice, you would not write code quite like this—you can fill out each table entry $T[0,j] \leftarrow j$ and $T[i,0] \leftarrow i$ in an initialization loop at the beginning, and then separately use the recursive definition just for $i, j \geq 1$.)

Next, we will discuss how to prove the correctness of this code (and code like it). But for the moment, appreciate how we have a direct quadratic-time algorithm that captures all the capabilities of the exponential-time recursive algorithm implementing the definition of edit distance, and further, is so close to the definition of edit distance, that you can come up with this enormously clever algorithm almost automatically.

4 Induction

Before we return to the analysis of edit distance, it is worth discussing the general proof technique of *induction*. Many people get introduced to induction in a rather formulaic way, where induction seems to consist of a lot of common "boilerplate text" and little explanatory power. Instead, in this course, we will find that induction is the "for" loop of proofs: lots of simple "for" loops look alike, but once one gets used to "for" loops, one uses them as a creative tool to accomplish all sorts of goals, and soon one can't imagine getting along without "for" loops; all this applies to induction as well. One uses a "for" loop when one wants to get many things done without needlessly repeating oneself; induction serves exactly this same end when one is writing a proof.

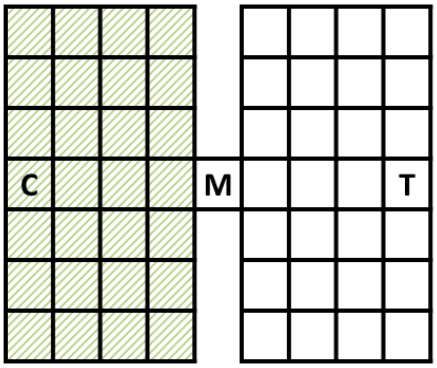
The danger of "boilerplate text" (text copied from a template) is that it loses its meaning, because the reader recognizes that the text was copied instead of produced via an honest effort at communication. Also, when one is writing a proof, inserting boilerplate text may short-circuit the thinking process.

In this section, we thus aim to understand induction in a way where each part of the induction argument makes sense, and each part corresponds to a fundamental component of the intuitive explanation for the statement we are trying to prove.

One bad habit is to copy the statement we are trying to prove into our induction hypothesis without thinking (treating the induction hypothesis as "boilerplate text"). If we want to prove a complicated statement $P(i)$ for all i, then the first instinct is to use $P(i)$ as our induction hypothesis. However copying the objective into the induction hypothesis is just one of many possible options that a creative prover should consider.

4.1 Monster Guarding the Treasure

We illustrate induction proofs with the following scenario: in the diagram below, a character "C" lives in a green shaded area on a grid; each turn, she can move one square up, down, left, or right; she wants to get to the treasure "T", but unfortunately there is a monster "M" on a narrow bridge, blocking the path, and she cannot occupy the same square as the monster. We want to construct a clean proof that she cannot reach the treasure.



A bit more formally, we want to prove the statement "at any time i, the character is not at location T". If we rigidly copy this statement into our induction hypothesis, we get an induction hypothesis that "at time i, the character is not at location T", which looks natural, but does *not* lead to a valid proof: if at time $i = 10$ the character is one square above T, this situation is entirely compatible with our induction hypothesis, and yet the character could choose to move down at this time and collect the treasure at time $i = 11$, which should not happen.

Instead, we need a more sophisticated induction hypothesis, more closely connected to the real reason why the character cannot get the treasure. Pause and think for a moment about how you would explain in a sentence why she cannot reach the treasure.

The board is naturally divided into the green region to the left of the monster, and the unshaded region to the right; C starts on the left, and can never get to the right region where the treasure is, because the monster guards the only path between left and right.

We now reformulate the above explanation into a clean inductive proof. The key property our intuitive argument emphasizes is that C must always remain in the green region; this is the "security property" that the monster is fighting hard to maintain—if the character ever manages to get in the white region, then the monster has lost. When you formulate

induction arguments, think about the "security property that your algorithm is fighting hard to maintain", and make it your induction hypothesis.

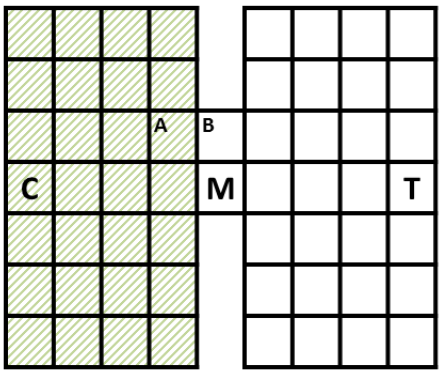
Thus, our new aim is to show by induction that C is always in the shaded region. The proof is straightforward: first we note that C starts in the shaded region (the base case of the induction); and then we note that if C is in the shaded region at some time i, then C must be in the shaded region at the next time, $i+1$, since the only path out of the shaded region goes through the monster square, which C cannot occupy. Thus for all time, C must be in the shaded region and thus cannot get the treasure.

Observe all three components of an inductive proof in the previous paragraph: we first had to carefully define the induction hypothesis, paying attention to shading and not just treasure; second, we had to ensure that our induction hypothesis holds at time 0, because if C gets to start in the white region then the monster security system (and our inductive proof) fails; finally, we check that there is no way for C to transition from being in a shaded square to an unshaded square, which is also a crucial part of the monster's security system.

Compare this inductive proof to the intuitive explanation until both make equal sense to you, and it is clear that they express the same thought. Once you are comfortable with induction, it is a natural and flexible communication tool.

4.1.1 A More Sophisticated Security Scheme

To point out some of the flexibility of induction, we make the situation a bit more complicated by widening the "bridge" in the middle of the diagram from 1 to 2 squares (the squares marked "A" and "B" are used in the proof below):



We now allow the monster to move as well, alternating moves with the character (and, say, not allowed to move onto the same square as the character, only able to block the character). We can think of the monster as being an algorithm we write, where our goal is to design a successful security scheme so that the user, C, never manages to steal the treasure and embarrass our algorithm. Given such an algorithm, we want to prove its correctness.

One potential algorithm is the following lazy strategy: M stays in its original square at the bottom of the two-square bridge, except when C is in square A, M temporarily moves to

square B.

Think for a moment about why this "security system" works. What security property is the monster fighting hard to maintain, and why is the (now more active) monster able to act to maintain this property?

As before, we use as our induction hypothesis that C stays in the green region; more precisely, we assume that when it is the monster's turn to move, C is in the green region. This is clearly true at the start of the game when it is the monster's first move. What happens next? If C is at A, then the monster moves to B, and C is blocked from leaving the green area this turn; otherwise, C is in some other green square (green because of the induction hypothesis), and the monster may freely move to the bottom square of the bridge, blocking the only other path across. Thus in either case, C cannot leave the green area this turn, proving our induction that C always remains in the green area, and letting us conclude that C will never reach T.

This induction proof is slightly more complicated than the previous case where the monster did not move, but the added complexity is essentially nothing more than "telling the story of our code", where we get to assume the induction hypothesis to make our story significantly simpler and cleaner. (For example, if C somehow makes it onto the bridge, then the monster may not be able to move to its desired spot; but we do not need to consider this situation, because of the guarantee of the induction hypothesis.)

Once you get comfortable with this, a good induction proof will feel like nothing more than "telling the story of your code." The induction hypothesis sets the scene for how a given iteration of your code starts, and then your task in the induction step is simply to describe what happens next and how it sets the scene for the next iteration of the loop, namely the next iteration of the induction hypothesis.

5 Proofs of Correctness

One of the reasons why we are starting this course with dynamic programming is that the structure of a dynamic programming algorithm is so similar to the structure of a good proof of the dynamic programming algorithm, that learning about either the algorithm or the proof will tell you a lot about the other.

Proofs of correctness of dynamic programming algorithms can almost always be expressed in a very similar way, and the amazing thing is that going through the steps of the proof can often tell you what the algorithm should be.

The next subsection contains a complete proof of correctness of the edit distance algorithm. The magic trick that this proof needs to somehow accomplish is: there are an exponentially large number of strategies S to edit word1 into word2, and somehow, for each and every one of these strategies S, we must show that the small amount of work done by the dynamic programming algorithm will automatically find a strategy for editing word1 into word2 that is at least as good as S. Somehow, each entry in the dynamic programming table is sum-

marizing, and shortcutting, an exponential amount of work. The key concept that lets the algorithm succeed is the fact that each square in the table will store the *best* cost of editing a certain subproblem of the overall editing problem, and that this *best* strategy summarizes everything we need to know about this subproblem. As you read through the proof below, pay special attention to 1) how the *best* solution to a problem is guaranteed to come from the best solution of *one* of its subproblems; and 2) how the proof essentially tells you the reader to considers an arbitrary editing strategy S and shows that the dynamic programming algorithm will always do at least as well, even though of course the dynamic programming algorithm never learns which S you secretly had in mind while reading the next section—a real magic trick!

5.1 Proof of Correctness of the Dynamic Programming Algorithm for Edit Distance

We will show by induction on i, j (traversed in the same order as the algorithm), that the value $T[i, j]$ correctly computes the edit distance between the first i characters of *word1* and the first j characters of *word2*.

Note that the edit distance between a sequence of characters of length k and the empty string will be k, therefore the 0-th row and column of T are correctly filled out, which we use as our base case of the induction.

Now consider, for arbitrary strings, *word1* and *word2*, and positive integers i, j, an *optimal* strategy S to edit the first i characters of *word1* into the first j characters of *word2*. Consider the *last* decision made in S, namely, the last character of the optimal alignment. There are three cases:

1. If a character from *word1* is aligned with a character from *word2*, then strategy S pays a cost of 1 according to whether these two characters differ, and then must use an *optimal* alignment for the remaining $i-1$ letters of *word1* and $j-1$ letters of *word2*; by our induction hypothesis $T[i-1, j-1]$ computes the cost of the optimal alignment in this subproblem, and thus, the first expression in the 3-way minimum of our dynamic program, $T[i-1, j-1] + [\text{word1}(i) \stackrel{?}{\neq} \text{word2}(j)]$ correctly computes the cost of the (optimal) strategy S in this case.

2. If a character from *word1* is aligned with no character from *word2*, then the cost of this alignment is 1 plus the optimal cost the remaining part of the alignment, namely, an optimal alignment of the first $i-1$ characters from *word1* with the first j characters of *word2*; by our induction hypothesis, $T[i-1, j]$ correctly stores this, and thus $T[i-1, j]+1$ correctly computes the optimal alignment cost in this case.

3. Similarly, if a character from *word2* is aligned with no character from *word1*, then the cost of this alignment is 1 plus the cost of an optimal alignment of what remains, the

first i characters from *word1* with the first $j-1$ characters of *word2*; by our induction hypothesis, $T[i, j-1]$ correctly stores this, and thus $T[i, j-1]+1$ correctly computes the optimal alignment cost in this case.

Thus 1 of the 3 options in the 3-way minimum of the dynamic programming algorithm will return the cost of the optimum strategy S, and all 3 of the options return the cost of some valid strategy, implying that the 3-way minimum will return the optimum edit distance. We thus conclude our inductive proof that our algorithm correctly fills out a table of edit distances."

5.2 Proof Outline, Piece by Piece

That is the entire proof. The first sentence is a very rich sentence! It tells the reader exactly what to expect—dynamic programming, the data structure we need (2D table), the meaning of the index values i,j, (positions in *word1* and *word2* respectively), and says what the **meaning** of each entry of the table **should** be at the end of the algorithm. It is very important to tell the reader what you are aiming at before you get started.

Also, note how we use all our ingredients: since the recurrence for edit distance is *defined* using 3 cases, the meat of our proof also has 3 cases, each of which analyzes one of the three portions of the 3-way minimum in our algorithm and shows how it exactly matches up with the definition of edit distance in that case.

It is worth writing down a generic form of the above proof, so that you can see how it is a simple template that you can use for future dynamic programming proofs:

> *"We will show by induction on [you should name the variables indexing the table, traversed in the order of the algorithm] that our algorithm correctly computes [you should specify what it is supposed to compute]. As a base case for the induction, we note that [you should justify that the initialization steps of the algorithm correctly computes what it is supposed to for those cases].*
>
> *Now consider [you should describe an arbitrary input to the algorithm], and [describe an arbitrary table entry that is about to be filled out, essentially specifying a moment in time for the algorithm]. Consider an optimal [you should specify the form of a solution to the problem], and consider the last decision made in this optimal solution. There are [you should specify how many cases exist in the definition of your recurrence relation] cases.*
>
> **(The following is the most critical part of a dynamic programming correctness proof. If it is not clear what you should be proving, please come to collaboration hours.)**
>
> *[Go through each case and prove that, assuming that everything previously put into the table is correct, your algorithm will correctly fill out the current entry of the table.*

> *You should do this by showing that, for each possibility for the last choice made by the unknown best editing scheme, there is a case in your recurrence relation that accurately describes this possibility (here is where you relate the possibilities in the real world to the cases covered by your code). Thus your recurrence relation will rely on the previous entries in the table to produce a solution that is at least as good as the optimal solution's choice.]"*

Dynamic programming is an enormously powerful tool. The homework and upcoming team contest will help you understand the many ways dynamic programming can be used.

Soon, we will discuss two powerful ways to augment a dynamic programming algorithm, so that our edit distance algorithm can 1) extract an actual alignment out of the table T via "backtracking"; and 2) count the number of optimal alignments. We then discuss some rather different dynamic programming algorithms for finding shortest paths in graphs.

6 Tips on Induction Proofs

It is worth briefly discussing different ways of thinking about induction. "Strong" induction, is where your induction hypothesis is not just about the $n-1$ case, but rather about *all* previous cases, all $i < n$. We also see inductive proofs on two dimensional objects, for example proving by induction that we can fill in a dynamic programming table correctly.

The thing to keep in mind is that you can *always* think of induction as being on the nonnegative integers; induction *always* has an order: $0, 1, 2, 3, ...$

Suppose we are trying to prove that an algorithm correctly fills in a table, and suppose we are filling in the table in some complicated diagonal order, starting from the top left. Then our induction hypothesis would be something like: "at the time we are trying to fill in entry (i, j), we have already correctly filled in all entries *before* this entry in our diagonal order." Rephrased slightly, we are performing induction on time in the algorithm—our induction hypothesis is essentially: "at time n in the algorithm, anything which should have happened by time $n-1$ happened correctly".

Our base case, $n = 0$, is trivially true, because at time 0 in the algorithm, nothing has happened and thus nothing can have gone wrong already. Then all we need to do is show that *if* nothing has gone wrong by time $n-1$, then nothing will go wrong at time n.

Keep in mind that it is crucially important when you are writing a proof to precisely choose and explain what expectations you have for the algorithm at time n. Part of the art of analyzing an algorithm is choosing exactly which expectations to have that are both 1) strong enough to imply the induction at the next step, but 2) weak enough to be implied by the induction hypothesis from the previous step. (This corresponds to mentally identifying the green region in the examples in Section 4.) "The algorithm is correct at time n" is a phrase that should never appear in your writeup because the reader does not know what you mean by "correct"; instead, you should say something much more precise, like "the algorithm

correctly stores in $T[i, j]$ the edit distance between the first i characters of word1 and the first j characters of word2."

One general hint for induction proofs: the *structure of your induction proof should often follow the structure of your algorithm.* If your algorithm is a `for` loop over the variable i, then you might want to induct over the iterations i of the `for` loop; if your algorithm is a recursive algorithm that calls itself on smaller lists, then you might want to prove a statement about the correctness of your algorithm on all lists, inducting over the size of the lists.

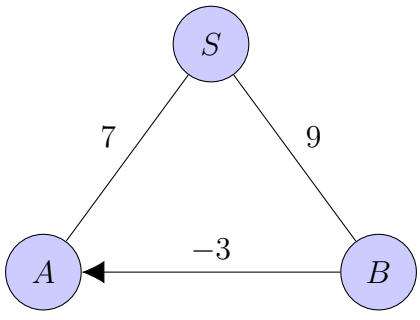
7 Dynamic Programming on Graphs

One setting for dynamic programming rather different from what we have seen is: algorithms for shortest paths on graphs.

Dijkstra's algorithm is the standard algorithm for finding shortest paths on a graph, and it breaks down in interesting ways when the graph has *negative* costs for going down certain edges. Dijkstra's algorithm colors in nodes one-by-one, and at each moment pulls off a heap the uncolored node with the smallest distance to the source, colors it, and then adds all the nodes adjacent to this node to the heap. In this way, Dijkstra's algorithm deals with each edge *once*, which is incredibly efficient. The induction hypothesis that you would want to use when analyzing Dijkstra is that "the moment a node is colored, we have correctly computed its distance to the source.

7.1 Bad Examples

For a slightly troubling example, consider the graph below with three nodes: a source S, and nodes A, B.

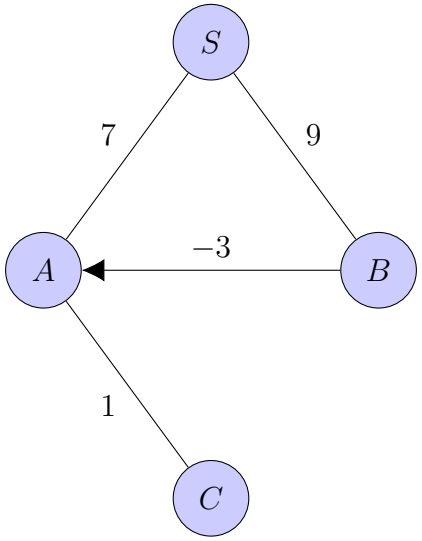


Dijkstra's algorithm starts with node S, then sees that A is the closest node, colors it and labels it with distance 7; then B is the next node, at distance 9; at this point the algorithm might notice that from B it can get to A, leading to a path $S \to B \to A$ of total cost

$$9 - 3 = 6,$$

which is less than 7. Thus our induction hypothesis was wrong! When we colored A, we thought we knew its distance was 7, but we later learn it is in fact 6. In this case we can fix our error later, but this is very dangerous: if your induction hypothesis is *slightly* wrong at one moment, it can get *very* wrong soon after.

For an example of this, consider adding a node C to the graph, with the edge $A \to C$ of cost 1.



Dijkstra's algorithm would consider C *before* it looks at B, and thus when it looks at C, the shortest path to C would be $S \to A \to C$ with distance

$$7 + 1 = 8.$$

However, the shortest path is actually $S \to B \to A \to C$ with distance

$$9 - 3 + 1 = 7.$$

The issue is that by the time we have analyzed edge $B \to A$ and corrected A's distance, we have already analyzed edge $A \to C$. The whole point of Dijkstra's algorithm is to analyze each edge only once, so it is *too late* to propagate our updated knowledge about A along the edge $A \to C$.

Dijkstra's algorithm does not work for negative edges.

Note that Dijkstra's algorithm is not really a dynamic programming algorithm in the sense we have been discussing, because the order in which Dijkstra fills out its table depends on what it has already found so far.

7.1.1 The Meaning of Negative Edges

Often one thinks of the edge weights in a graph as representing distances, and negative edge weights do not make sense from this perspective. More generally, we might think of edge

weights as representing "costs" of routing traffic along an edge, where some costs are positive, and other costs are negative—for example, if an advertiser pays us to route traffic along a certain edge.

In the context of "shortest paths" in a graph, it is worth pointing out that these problems only make sense if there are no "negative cost cycles"—if there is a way to go in a circle for negative cost, then the best way to go from i to j is to first go from i to this cycle, go around the cycle billions of times, and then go to j.

The simplest kind of negative cycle is just a negative edge on which we are allowed to travel in both directions (if an advertiser is paying you every time you send traffic by her in either direction, then you can make billions of dollars by simply directing traffic to go back and forth in front of her billions of times). Thus in the above diagrams, see that we were careful to make the "-3" edge directed from B to A, instead of undirected like the other edges.

7.2 Shortest Path Algorithms allowing Negative Edge Weights

We will introduce the Bellman-Ford and Floyd-Warshall algorithms, two algorithms for finding shortest paths in graphs that are in a sense much cleaner than Dijkstra, because they follow the standard dynamic programming outline. Because they are clean, they are also robust, working for negative edges as well as positive edges.

Think for a moment about how you could design a dynamic programming approach for shortest paths in a graph—what are the simpler subproblems what you would compute first? (How could you add another index to a dynamic programming table that controls the "simplicity" of paths?) The two algorithms below reflect two rather different choices for how to break the shortest path problem into simpler subproblems.

7.2.1 Bellman-Ford Algorithm

Definition 7.1

The Bellman-Ford algorithm finds the shortest path from a fixed node i to all nodes j, where the path has at most k intermediate nodes.

We thus draw a table indexed by j and k (i is fixed), and try to fill it in. The easiest place to start is when $k = 0$: this is "paths" from i to j with no intermediate nodes—thus if there is an edge between i and j, we write its length into the $(j, 0)$ entry of the table, and otherwise write infinity (or, equivalently, a symbol indicating that no path exists). We now need to figure out how to compute the next row of the table: suppose we know everything about paths of up to k nodes, how can we compute the optimal path of $k + 1$ nodes from i to j?

When looking for a dynamic programming algorithm, it is often helpful to think of how one would prove its correctness (even though we don't yet know the algorithm!). The proof of

correctness starts: "Consider a graph, and nodes i, j, and consider an optimal path from i to j of $k+1$ intermediate nodes. Consider the **last** decision on this path...."

In our setting, it is pretty clear what the last decision on a path from i to j is: the second-to-last node on the path. Thus we look at all possible second-to-last-nodes, s. Namely, consider nodes s which are connected to j by an edge—and for each such s, we look up in the table the optimal way to get from i to s with k intermediate nodes, and we add table entry $T(s,k)$ to the length of the edge from s to j, which we denote $len(s,j)$. It is clear that whatever the optimal path from i to j of $k+1$ intermediate nodes is, we have now covered it by one of these cases. Thus we have the recurrence relation:

$$T(j,k+1) = \min_{s \text{ with an edge } s \to j} T(s,k) + len(s \to j)$$

Since for each value of k we need to look at all edges, and k ranges from 0 up to the number of vertices, the total running time of this algorithm is $|V||E|$, the product of the number of vertices and number of edges.

Note: the problem you really want to solve is "shortest path from i to j using **up to** k intermediate nodes" instead of "**exactly** k intermediate nodes." Can you adapt the above algorithm to this?

In practice, the index k is not used in Bellman-Ford, the table is just a 1-dimensional table storing in $T(j)$ the less precise "cost of the best path found so far from i to j". Instead of looping over the table $|V|$ times, one instead loops until the table stops changing, which might be rather earlier. The proof of correctness would point out that the modified table constructed at the k^{th} iteration stores values that correspond to lengths of actual paths, and these paths have length less than or equal to what would be found by the simpler algorithm at time k. So therefore by time $|V|$ both algorithms must be identical; yet if the modified algorithm stops earlier, then it is still correct.

7.2.2 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm solves the shortest path problem for *all* pairs of nodes at once. It thus takes more time than Bellman Ford, but accomplishes more. If you wanted to run a Google Maps-like service, you would want to know the shortest paths between any pair of points, so an "all pairs shortest path" algorithm would be appropriate.

Definition 7.2

Floyd-Warshall fills out a table $T(i,j,k)$ representing the length of the shortest path from node i to node j only using nodes from the set $\{1 \dots k\}$ as intermediate nodes.

The base case of this table construction, that we can fill in immediately, is the same as for Bellman-Ford: when $k=0$, table entry $T(i,j,0)$ asks for the "shortest path from i to j with no intermediate nodes", namely, $len(i \to j)$ if such an edge exists, and infinity otherwise.

To figure out how to update the table, again, it is best to start thinking: Consider a graph, and two nodes i, j, and the shortest path from i to j going through intermediate nodes $1 \dots k+1$.

There are two cases worth considering: if this path does **not** use $k+1$ as an intermediate node, then we have already solved the problem in this case:

$$T(i, j, k+1) = T(i, j, k)$$

Otherwise, if this path does use $k+1$, then it uses node $k+1$ exactly once, and the path must look like the following: it goes from i to $k+1$ using intermediate nodes only $1 \dots k$, and then goes from $k+1$ to j using intermediate notes only $1 \dots k$. Thus in this case

$$T(i, j, k+1) = T(i, k+1, k) + T(k+1, j, k)$$

In sum, we can write our recurrence as the minimum of these two cases:

$$T(i, j, k+1) = \min(T(i, j, k), T(i, k+1, k) + T(k+1, j, k))$$

This algorithm takes time $O(n^3)$ to find all distances on a path with n nodes, since we fill out a $n \times n \times n$ table, doing constant work per entry. This will thus usually be rather faster than repeating Bellman-Ford for every starting vertex i, which would take time $n^2|E|$.

7.3 Negative Cycles

For both the above algorithms, one must be careful of "negative cycles". Any robust shortest path algorithm should report that the problem is unsolvable if it finds a negative cycle. How can we adapt the above algorithms to be this robust?

For Bellman-Ford, we modify the algorithm by asking it to fill out one more row of the table—instead of asking for paths of up to $n-2$ intermediate nodes between nodes i and j, we first do this, and then add one more row to compute the lengths of the (illegitimate) paths with $n-1$ intermediate nodes, and if a new shortest path is found to any node, we conclude that there must be a negative cycle (because any path going through $n-1$ intermediate nodes has $n+1$ total nodes, must reuse some node, and hence must contain a cycle; if this is shorter than the best path with fewer nodes, then the cycle must be negative-cost). We thus output an error in this case. Why can we be sure that if no shortest path of length $n-1$ appears, then further iterations will never discover a negative cycle? Look at each of the expressions involved in computing a new table entry in the Bellman-Ford recurrence: $T(j, k+1)$ comes from the minimum of expressions $T(s, k) + len(s \to j)$, and let s' be the value of s that minimizes this expression; since we assumed that $T(s', k)$ is greater than or equal to some corresponding entry in a previously computed row $T(s', k')$, then the newly computed $T(j, k+1) = T(s', k) + len(s' \to j)$ must be at least as large as the previous entry $T(j, k'+1)$ and hence can't be a newly discovered shortest path. Thus, Bellman-Ford can

safely stop when it reaches the first row that does not discover a new shortest path, and in this case there will be no negative cycles.

Adapting Floyd-Warshall to look for negative cycles is much simpler: the table entry $T(i, i, k)$ asks for the shortest path from node i to *itself*, using only nodes $1, \ldots, k$ as intermediary nodes, and we just check to make sure each of these entries is never negative.

8 In Conclusion

Dynamic programming can often be thought of as a way of doing exponential work in polynomial time, by recognizing that we can reuse many partial solutions. Dynamic programming problems often have natural recursive solutions (that take exponential time), but if you are careful, you can make a small table of all possible recursive calls, and fill out the table in an order so all of your recurrence relations only depend on entries in the table that have already been computed. Homework 1 and the Team Contest give you several interesting examples showing some of the variety of dynamic programming. Can you see why all of them in some sense are getting an exponential amount of work done, despite running much more quickly? Keep this in mind whenever you encounter a dynamic programming problem and you will get a better appreciation of dynamic programming.