CS 362, Lecture 5

Jared Saia
University of New Mexico

Today's Outline

- Annihilator Wrap-up
- Intro Dynamic Programming
- String Alignment

Limitations

- Our method does not work on $T(n) = T(n-1) + \frac{1}{n}$ or $T(n) = T(n-1) + \lg n$
- The problem is that $\frac{1}{n}$ and $\lg n$ do not have annihilators.
- Our tool, as it stands, is limited.
- Key idea for strengthening it is *transformations*

Transformations Idea

- Consider the recurrence giving the run time of mergesort $T(n) = 2T(n/2) + kn$ (for some constant k), $T(1) = 1$
- How do we solve this?
- We have no technique for annihilating terms like $T(n/2)$
- However, we can *transform* the recurrence into one with which we can work

Transformation

- Let $n = 2^i$ and rewrite $T(n)$:
- $T(2^0) = 1$ and $T(2^i) = 2T(\frac{2^i}{2}) + k2^i = 2T(2^{i-1}) + k2^i$
- Now define a new sequence t as follows: $t(i) = T(2^i)$
- Then $t(0) = 1$, $t(i) = 2t(i-1) + k2^i$

Now Solve

- We've got a new recurrence: $t(0) = 1$, $t(i) = 2t(i-1) + k2^i$
- We can easily find the annihilator for this recurrence
- $(\mathbf{L}-2)$ annihilates the homogeneous part, $(\mathbf{L}-2)$ annihilates the non-homogeneous part, So $(\mathbf{L}-2)(\mathbf{L}-2)$ annihilates $t(i)$
- Thus $t(i) = (c_1 i + c_2)2^i$

Reverse Transformation

- We've got a solution for $t(i)$ and we want to transform this into a solution for $T(n)$
- Recall that $t(i) = T(2^i)$ and $2^i = n$

$$\begin{aligned}
t(i) &= (c_1 i + c_2)2^i && (1)\\
T(2^i) &= (c_1 i + c_2)2^i && (2)\\
T(n) &= (c_1 \lg n + c_2)n && (3)\\
&= c_1 n \lg n + c_2 n && (4)\\
&= O(n \lg n) && (5)
\end{aligned}$$

Success!

Let's recap what just happened:

- We could not find the annihilator of $T(n)$ so:
- We did a *transformation* to a recurrence we could solve, $t(i)$ (we let $n = 2^i$ and $t(i) = T(2^i)$)
- We found the annihilator for $t(i)$, and solved the recurrence for $t(i)$
- We *reverse transformed* the solution for $t(i)$ back to a solution for $T(n)$

Another Example

- Consider the recurrence $T(n) = 9T(\frac{n}{3}) + kn$, where $T(1) = 1$ and k is some constant
- Let $n = 3^i$ and rewrite $T(n)$:
- $T(3^0) = 1$ and $T(3^i) = 9T(3^{i-1}) + k3^i$
- Now define a sequence t as follows $t(i) = T(3^i)$
- Then $t(0) = 1$, $t(i) = 9t(i-1) + k3^i$

Now Solve

- $t(0) = 1$, $t(i) = 9t(i-1) + k3^i$
- This is annihilated by $(\mathbf{L} - 9)(\mathbf{L} - 3)$
- So $t(i)$ is of the form $t(i) = c_1 9^i + c_2 3^i$

Reverse Transformation

- $t(i) = c_1 9^i + c_2 3^i$
- Recall: $t(i) = T(3^i)$ and $3^i = n$

$$\begin{aligned} t(i) &= c_1 9^i + c_2 3^i \\ T(3^i) &= c_1 9^i + c_2 3^i \\ T(n) &= c_1 (3^i)^2 + c_2 3^i \\ &= c_1 n^2 + c_2 n \\ &= O(n^2) \end{aligned}$$

In Class Exercise

Consider the recurrence $T(n) = 2T(n/4) + kn$, where $T(1) = 1$, and k is some constant

- Q1: What is the transformed recurrence $t(i)$? How do we rewrite n and $T(n)$ to get this sequence?
- Q2: What is the annihilator of $t(i)$? What is the solution for the recurrence $t(i)$?
- Q3: What is the solution for $T(n)$? (i.e. do the reverse transformation)

A Final Example

Not always obvious what sort of transformation to do:

- Consider $T(n) = 2T(\sqrt{n}) + \log n$
- Let $n = 2^i$ and rewrite $T(n)$:
- $T(2^i) = 2T(2^{i/2}) + i$
- Define $t(i) = T(2^i)$:
- $t(i) = 2t(i/2) + i$

A Final Example

- This final recurrence is something we know how to solve!
- $t(i) = O(i \log i)$
- The reverse transform gives:

$$\begin{aligned} t(i) &= O(i \log i) && (6) \\ T(2^i) &= O(i \log i) && (7) \\ T(n) &= O(\log n \log \log n) && (8) \end{aligned}$$

DP Intro

"Those who cannot remember the past are doomed to repeat it." - George Santayana, The Life of Reason, Book I: Introduction and Reason in Common Sense (1905)

What is Dynamic Programming?

- Dynamic Programming is basically "Divide and Conquer" with memorization
- Basic Trick is: *Don't solve the same problem more than once!*

Fibonacci Example

Consider the following procedure for computing the n-th Fibonacci number:

```
Fib(n){
  if (n<2)
    return n;
  else
    return Fib(n-1) + Fib(n-2);
}
```

Analysis

- Q: What is the runtime of Fib?
- A: Except for recursive calls, the entire algorithm takes a constant number of steps. If $T(n)$ is the run time of the algorithm on input n, then we can say that:
  $T(0) = T(1) = 1$, $T(n) = T(n-2) + T(n-1) + 1$
- It's easy to show by induction that $T(n) = 2F_{n+1} - 1$. This is very bad!

Aside

- Q: How can we solve $T(n)$ exactly?
- A: We solved this recurrence using annihilaotrs in the last lecture to get $T(n) = c_1\phi^n + c_2\hat{\phi}^n + c_3 1^n$ where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$.

Aside II

- If we solve for constants, we get that:

$$\begin{aligned} T(0) = 1 &= c_1 + c_2 + c_3 \\ T(1) = 1 &= c_1\phi + c_2\hat{\phi} + c_3 \\ T(2) = 3 &= c_1\phi^2 + c_2\hat{\phi}^2 + c_3 \end{aligned}$$

  Solving this system of linear equations (using Gaussian elimination) gives:

$$c_1 = 1 + \frac{1}{\sqrt{5}}, \quad c_2 = 1 - \frac{1}{\sqrt{5}}, \quad c_3 = -1,$$

Aside III

- So our final solution is

$$T(n) = \left(1 + \frac{1}{\sqrt{5}}\right)\phi^n + \left(1 - \frac{1}{\sqrt{5}}\right)\hat{\phi}^n - 1 = \Theta(\phi^n).$$

The Problem

- The reason Fib is so slow is that it computes the same Fibonacci numbers over and over
- In general, there are F_{k-1} recursive calls to Fib(n-k)
- We can greatly speed up the algorithm by writing down the results of the recursive calls and looking them up if needed

DP-Fib

```
DP-Fib(n){
  if (n<2)
    return n;
  else{
    if (F[n] is undefined){
      F[n] = DP-Fib(n-1) + DP-Fib(n-2);
    }
  return F[n];
}}
```


Analysis

- For every value of x between 1 and n, DP-Fib(x) is called exactly one time.
- Each call does constant work
- Thus runtime of DP-Fib(n) is $\Theta(n)$ - a *huge* savings

Take Away

Dynamic Programming is different than Divide and Conquer in the following way:

- "Divide and Conquer" divides problem into independent subproblems, solves the subproblems recursively and then combines solutions to solve original problem
- Dynamic Programming is used when the subproblems are not independent, i.e. the subproblems share subsubproblems
- For these kinds of problems, divide and conquer does more work than necessary
- Dynamic Programming solves each subproblem once only and saves the answer in a table for future reference

The Pattern

- **Formulate the problem recursively.**. Write down a formula for the whole problem as a simple combination of answers to smaller subproblems
- **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution by considering the intermediate subproblems in the correct order.

Note: Dynamic Programs store the results of intermediate subproblems. This is frequently *but not always* done with some type of table.

Edit Distance

- The *edit distance* between two words is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word into another. For example, the edit distance between `FOOD` and `MONEY` is at most four:

`FOOD` $\longrightarrow$ `MOOD` $\longrightarrow$ `MOND` $\longrightarrow$ `MONED` $\longrightarrow$ `MONEY`

String Alignment

Better way to display this process:

- Place the words one above the other in a table
- Put a gap in the first word for every insertion and a gap in the second word for every deletion
- Columns with two different characters correspond to substitutions
- Then the number of editing steps is just the number of columns that don't contain the same character twice

Example

- String Alignment for “FOOD” and “MONEY”:

F	O	O		D
M	O	N	E	Y

- It's not too hard to see that we can't do better than four for the edit distance between “Food” and “Money”

Example II

- Unfortunately, it can be more difficult to compute the edit distance exactly. Example:

```
A L G O R   I   T H M
A L   T R U I S T I C
```


Key Observation

- If we remove the last column in an optimal alignment, the remaining alignment must also be optimal
- Easy to prove by contradiction: Assume there is some better subalignment of all but the last column. Then we can just paste the last column onto this better subalignment to get a better overall alignment.
- Note: The last column can be either: 1) a blank on top aligned with a character on bottom, 2) a character on top aligned with a blank on bottom or 3) a character on top aligned with a character on bottom

DP Solution

- To develop a DP algorithm for this problem, we first need to find a recursive definition
- Assume we have a m length string A and an n length string B
- Let $E(i, j)$ be the edit distance between the first i characters of A and the first j characters of B
- Then what we want to find is $E(n, m)$

Recursive Definition

- Say we want to compute $E(i, j)$ for some i and j
- Further say that the "Recursion Fairy" can tell us the solution to $E(i', j')$, for all $i' \leq i$, $j' \leq j$, *except* for $i' = i$ and $j' = j$
- Q: Can we compute $E(i, j)$ efficiently with help from the our fairy friend?

Recursive Definition

There are three possible cases:

- **Insertion:** $E(i,j) = 1 + E(i-1,j)$
- **Deletion:** $E(i,j) = 1 + E(i,j-1)$
- **Substitution:** If $a_i = b_j$, $E(i,j) = E(i-1,j-1)$, else $E(i,j) = E(i-1,j-1) + 1$

Summary

Let $I(A[i] \neq B[j]) = 1$ if $A[i]$ and $B[j]$ are different, and 0 if they are the same. Then:

$$E(i,j) = \min \left\{ \begin{array}{l} E(i-1,j)+1, \\ E(i,j-1)+1, \\ E(i-1,j-1)+I(A[i] \neq B[j]) \end{array} \right\}$$

Base Case(s)

It's not too hard to see that:

- $E(0,j) = j$ for all j, since the j characters of B must be aligned with blanks
- Similarly, $E(i,0) = i$ for all i

Recursive Alg

- We now have enough info to directly create a recursive algorithm
- The run time of this recursive algorithm would be given by the following recurrence:

$$T(m,0) = T(0,n) = O(1)$$

$$T(m,n) = T(m,n-1) + T(m-1,n) + T(n-1,m-1) + O(1)$$

- Solution: $T(n,n) = \Theta(1 + \sqrt{2}^n)$, which is terribly, terribly slow.

Better Idea

- We can build up a $m \times n$ table which contains all values of $E(i,j)$
- We start by filling in the base cases for this table: the entries in the 0-th row and 0-th column
- To fill in any other entry, we need to know the values directly above, to the left and above and to the left.
- Thus we can fill in the table in the standard way: left to right and top down to ensure that the entries we need to fill in each cell are always available

Example Table

- Bold numbers indicate places where characters in the strings are equal
- Arrows represent predecessors that define each entry: horizontal arrow is deletion, vertical is insertion and diagonal is substitution.
- Bold diagonal arrows are "free" substitutions of a letter for itself
- Any path of arrows from the top left to the bottom right corner gives an optimal alignment (there are three paths in this example table, so there are three optimal edit sequences).

		A	L	G	O	R	I	T	H	M
	0	→1	→2	→3	→4	→5	→6	→7	→8	→9
A	1	**0**	→1	→2	→3	→4	→5	→6	→7	→8
L	2	1	**0**	→1	→2	→3	→4	→5	→6	→7
T	3	2	1	1	→2	→3	→4	**4**	→5	→6
R	4	3	2	2	2	**2**	→3	→4	→5	→6
U	5	4	3	3	3	3	3	→4	→5	→6
I	6	5	4	4	4	4	**3**	→4	→5	→6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	→5	→6
I	9	8	7	7	7	7	6	5	5	→6
C	10	9	8	8	8	8	7	6	6	6

Analysis

- Let n be the length of the first string and m the length of the second string
- Then there are $\Theta(nm)$ entries in the table, and it takes $\Theta(1)$ time to fill each entry
- This implies that the run time of the algorithm is $\Theta(nm)$
- Q: Can you find a faster algorithm?