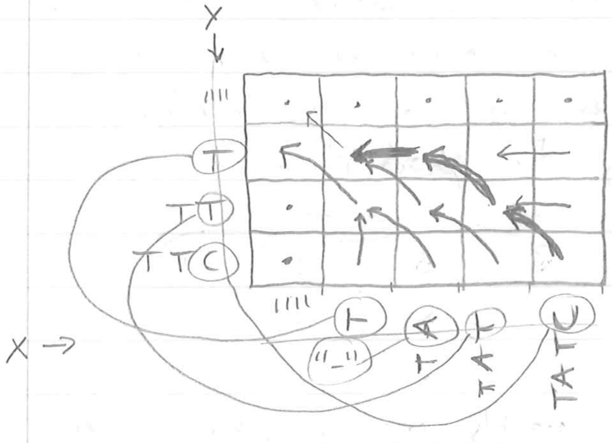
why is 1,1 always in the upper left for computers? Because back when output was printed on tape/paper, that was the first accessible place to print!

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 1,1 & & & & \\ 2 & & & & & \\ 3 & & & & & \\ 4 & & & & & \end{array}$$

The dynamic program

One thing to notice about our 'inspection' plot is that it doesn't display in any way the actual alignments that are stored in the subproblems. Or - does it?

Let's consider the 'From' information for a slightly larger portion of the plot, considering the (sub)problem X=TATC vs Y = TTC, and all smaller subproblems of course



Notice that the lower right square - holding the solution for TATC, TTC, has a diagonal arrow pointing at TAT, TT - corresponding to the best solution being the alignment of $\left(\overset{TAT}{TT}\text{ aligned w/}\right)\overset{C}{C}$, so in the final solution the two C's are aligned w/ each other. Similarly, the solution of that subproblem has a diagonal arrow, meaning it came from $\left(\overset{TA}{T}\text{ aligned w/}\right)$ with the two T's aligned together.

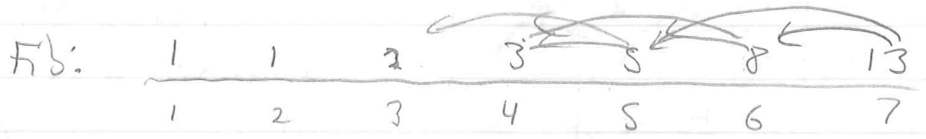
But what of TA, T? This arrow points leftward, to the subproblem $\left(\overset{T}{T}\text{ aligned w/}\right)$, so the A was aligned w/ a gap. Finally, the T,T problem was a base case in our situation, though we could just as easily imagine a diagonal there too indicating these get aligned together. So, even though we haven't displayed the subproblem solutions, the 'From' information is enough to reconstruct the alignment.

As well, I'm sure it hasn't escaped your attention that the last letters of the subproblems, organized this way, make the X and Y sequences themselves! (prefixed, if you will, by a blank '''').

50 (A) The general pattern is: For cell at row i, col j: diag: y[row i] matched w X[col j]
left: "-" matched w/ X[col j]
up: Y[row i] matched w/ "-"

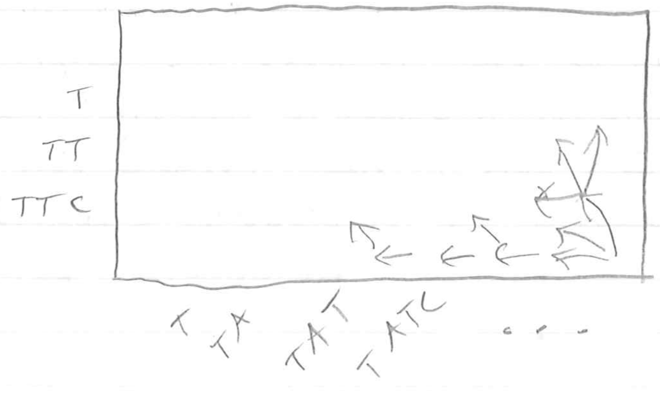
So, if we consider X = c("", "T", "A", "T", "C"), and Y = c("", "T", "T", "C"), then the subproblem at row 3, column 4 (TAT, TT) having a diagonal means the third element of Y matches the fourth of X.

Before we go much further, let's recall (from our Fibonacci example) how dynamic programming relates to (memoized) recursion. In recursion, we are solving problems based on subproblems on an as-needed basis (memoizing, or cacheing, for speed if necessary)

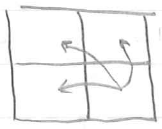


With dynamic programming, we smartly realize what problems will be needed and precompute them from the bottom up:

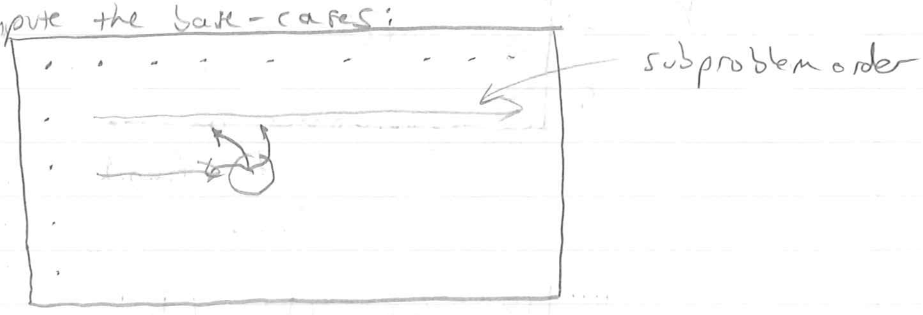
For our sequence-alignment problem, our subproblems can be organized in a table, and the recursive algorithm is solving them as-needed from the lower-right:



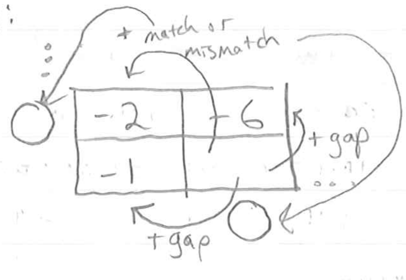
But we know which subproblems will be needed for further computation, as we saw they're arranged as neighbors:



So, if we ensure that we fill from the top left, we'll always have the right subproblems pre-computed; especially if we precompute the base-cases:



As we saw, to compute an alignment, we really only need the table of 'from' information ('diag', 'left', or 'up' for each subproblem). ~~In order to~~ do this for any given cell, what we need is the score of the neighboring cells. Eg:



What is the arrow that goes in the lower-right here? Depends on which the best of the three subproblems, considering the score of their cells plus the score of the induced gap match or match/mismatch.

Options: -6 + gap cost
-1 + gap cost
-2 plus cost of match or mismatch

To determine the 'plus cost', we have to consider the base at X or Y in that position!

Whew! Ok, here's the strategy: we'll keep <u>two</u> matrices, one holding scores, and one holding 'arrows' - ie, just the terms "left", "diag", or "up". The number of colums of the matrix will be the size of the ""-prefixed version of X, and the number of colums will be the ""-prefixed version of Y:

$$Y \quad \begin{array}{c|ccccc} "" & 0 & -4 & -8 & -12 & -16 \\ T & -4 & & & & \\ T & -8 & & \text{etc} & & \\ C & -2 & & & & \\ \hline & "" & T & A & T & C \end{array} \qquad \text{scores}$$

$$X$$

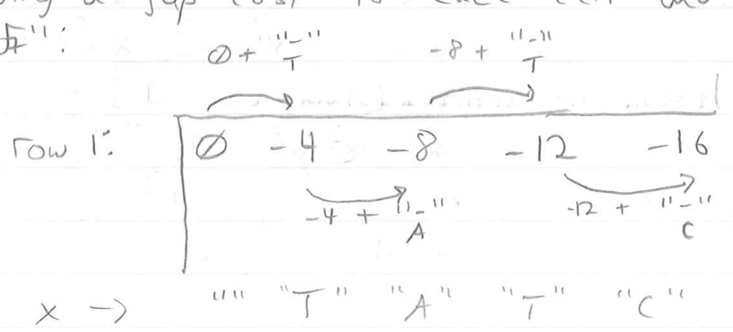
$$\begin{array}{c|ccccc} "" & \text{'?'} & \text{'left'} & \text{'left'} & \text{'left'} & \\ T & \text{'up'} & & & & \\ T & \text{'up'} & & \text{etc} & & \\ C & \text{'up'} & & & & \\ \hline & "" & T & A & T & C \end{array} \qquad \text{'arrows}$$

Notice that we can easily fill out the 'base cases', and we've set the arrows for the base cases in such a way that the same rules will apply when reconstructing the alignment. How about some flippin' code? Initially we'll set all cells of the score to be 0, and all arrows to be "?".

```
global_aln <- function(x, y) {
    x <- c("", x)
    y <- c("", y)

    scores <- matrix(Ø, ncol=length(x), nrow=length(y))
    arrows <- matrix("?", ncol= length (x), nrows = length(x))
```

Now will fill in the base cases along the top (row = 1), adding a gap cost to each cell and setting the arrow = "left":



```
    row <- 1
    for(col in seq(2, length(x))){
        scores[row, col] <- scores[row, col-1] +
                            score_aln("-", x[col])
        arrows[row, col] <- "left"
    }
```

Similarly, we need to do the same thing down the left side (col = 1):

```
    col <- 1
    for (row in seq(1, length(y))){
        scores[row, col] <- scores[row-1, col] +
                            score_aln (y[row], "-")
        arrows[row, col] <- "up"
    }
```

Now we need to fill in the rest of the table in a left-to-right, top-to-bottom manner - the score and arrow will depend on the best of the 3 options. (And, by doing it this way, we ensure the necessary subproblems have been precomputed).

```
for(col in seq(2, length(x))){                    # left to right
    for(row in seq(2, length(y))){                # top to bottom
        diagscore <- scores[row - 1, col - 1] +
                         score_aln(y[row], x[col])
        leftscore <- scores[row, col - 1] +
                         score_aln(y[row], "-")
        upscore <- scores[row - 1, col] +
                         score_aln("-", x[col])
```

(Yes, this kind of programming will induce dyslexia in pretty much anyone, be careful and try to keep to rules like "always row, col order.) Now we need to decide which of these scores is best, and update the cell in scores and arrows accordingly:

```
        bestscore <- diagscore
        bestarrow <- "diag"
        if(leftscore > bestscore){
            bestscore <- leftscore
            bestarrow <- "left"
        }
        if(upscore > bestscore){
            bestscore <- upscore
            bestarrow <- "up"
        }
        scores[row, col] <- bestscore
        arrows[row, col] <- bestarrow
    }
}
```

And thus ends the nested for-loops that fill out the entirety of both matrices. And, now we can follow the 'from' arrows back, reconstructing the alignment as we discussed. We'll start with empty alnx and alny, and keep a currentrow and currentcol to keep track of which 'cell' we're looking at. We'll trace the results all the way back to the upper left corner (currentrow = 1, currentcol = 1), from the lower right:

```
currentrow <- length(y)      # lower
currentcol <- length(x)      # right
alnx = ""
alny = ""

while(currentrow != 1 & currentcol != 1) {
    arrow <- arrows[currentrow, currentcol]
    if(arrow == "diag") {
        alnx <- c(alnx, x[currentcol])          # aln
        alny <- c(alny, y[currentrow])
        currentrow <- currentrow -1              # move
        currentcol <- currentcol -1
    } else if(arrow == "left") {
        alnx <- c(alnx, x[currentcol])          # aln
        alny <- c(alny, "-")
        currentcol <- currentcol -1              # move
    } else if(arrow == "up") {
        alnx <- c(alnx, "-")                    # aln
        alny <- c(alny, y[currentrow])
        currentrow <- currentrow -1              # move
    }
}
```

Alright! Now we can print alnx and alny, which should (in theory) have the alignment:

```
    print(alnx)
    print(alny)
```

Well, ok, its close - but backwards! That makes sense, we did the reconstruction backwards after all. Let's return a simple 'answer list' with reversed versions:

```
    answer <- list(alnx = unvec_char(rev(alnx)),
                   alny = unvec_char(rev,alny)))
    return(answer)
}
```

And that, my friends, is the Needleman-Wunsch, global alignment dynamic program. We can even use it:

```
x <- char_vec("TATCTGCAACG")
y <- char_vec("TTCTGC")
answer <- global_aln(x,y)
print(answer)
```

The dynamic program - though seemingly impossibly far removed from the inductive proof we started with, does mirror it as well as the recursive (memoized) solution.

There are even some benefits, primarily in the fact that we are <u>not limited by the depth of the call stack</u>. We can align very long sequences this way.

But, how much time is used? At least the length of X (m) times the length of Y (n) - O(mn). If these are both millions of base-pairs, that's a lot of time! Even worse, the size of the matrices (or the memoization cache, if you prefer) is the same. It could take many gigabytes of RAM to align only a couple 'megabase' sequences.

Still, this is where the heart of much of bioinformatics lies, and we'll look at some variations on this theme next.