Recursive Sequence Alignment (Global)

As fun as all this theory is, it's going to be a lot more fun if we actually code up the solution. The pattern will be quite similar to the solution for the Fibonacci sequence - after all, we've got base cases and recursive cases, but the solution will be more detailed, partly because of the data structures R makes available.

To start with, unlike the Fibonacci sequence, an 'answer' actually contains two parts, well, three: the gapped version of x, the gapped version of y, and the score; we'll call these xaln, yaln, and score:

$$\underbrace{\begin{array}{ll} x: & \text{ACTAGC} \\ y: & \text{ATACC} \end{array}}_{\text{input}} \Rightarrow \underbrace{\begin{array}{lll} \text{xaln}: & \text{ACTAGC} & \text{score: } 1 \\ \text{yaln} & \text{A-TACC} & \end{array}}_{\text{answer}}$$

As mentioned, unless otherwise necessary, we'll encode sequences as character vectors rather than longer strings, using the char_vec() function: ~~"ACTAGC"~~ "A", "C", "T", "A", "G", "C"
This allows us to easily concatenate sequences, or get subsequences

```
seq1 <- char_vec("TAC")              # "T" "A" "C"
seq2 <- char_vec("CC")               # "C" "C"
both <- c(seq1, seq2)                # "T" "A" "C" "C" "C"
pre1 <- seq1[1:length(seq1) - 1]     # "T" "A"
end1 <- seq1[length(seq1)]           # "C"
```

We'll encode an 'answer' as a named list, and we'll also store the inputs. We could encode it by hand like so:

```
answer <- list(x = c("A", "C", "T", "A", "G", "C"),
               y = c("A", "T", "A", "C", "C"),
               xaln = c("A", "C", "T", "A", "G", "C"),
               yaln = c("A", "-", "T", "A", "C", "C"),
               score = 1)
```

Though of course we'll be using code to create these. As per the homework, we created a few "helper" functions: one that handled any base case (a tedious little sucker) and one that scored an alignment according to the scoring rules:

```
# Given two char vecs of equal length, returns an
# integer score:
score_aln <- function(xin, yin) {
    :

}

# given two char vecs constituting a base case
# (either of length Ø or both of length 1)
# return an 'answer' object
base_case <- function(xin, yin) {
    :

}
```

Recursivity

Now we get to do the fun part: the recursive case: it also takes two char vectors, x and y, and returns an answer. If x and y are a base case, it just calls the base case function:

```
global_aln <- function(x, y) {
    if(length(x) == 0 | length(y) == 0 | (length(x) == 1 &
                                           length(y) == 1 )) {
        return(base_case(x, y))
    }
```

If it's not a base case, we follow the recursive pattern: we start by computing px, ex, py, and ey as defined in our proof*:

```
px <- x[1:length(x) - 1]
ex <- x[length(x)]
py <- y[1:length(y) - 1]
ey <- y[length(y)]
```

Next, we get to recurse. Let's revisit our little diagram for the three options:

(next page)

* x[a:b] returns x from indices a to b, inclusive.

answer = best of these

A: $\begin{pmatrix} P_x \text{ aligned w/} \\ P_y \end{pmatrix} \begin{matrix} e_x \\ e_y \end{matrix}$, $\quad score_A = S(A) + S(e_x, e_y)$

B: $\begin{pmatrix} P_x e_x \text{ aligned w/} \\ P_y \end{pmatrix} \begin{matrix} - \\ e_y \end{matrix}$, $\quad score_B = S(B) + S(\text{"-"}, e_y)$

C: $\begin{pmatrix} P_x \text{ aligned w/} \\ P_y e_y \end{pmatrix} \begin{matrix} e_x \\ - \end{matrix}$, $\quad score_C = S(C) + S(e_x, \text{"-"})$

So, we need to compute the subalignments, A, B, and C, recursively:

```
A <- global_aln(px, py)
B <- global_aln(c(px, ex), py)
C <- global_aln(px, c(py, ey))
```

Notice the similarity between our conceptual definition and the code! Now we can use these subanswers to compute the three potential overall answer objects, accessing parts of the subanswers as needed w/ [[ ]] notation

```
answera <- list(x = x, y = y,
                xaln = c(A[["xaln"]], ex ),
                yaln = c(A[["yaln"]], ey ),
                score = A[["score"]] + score_aln(ex, ey))
answerb <- list(x = x, y = y,
                xaln = c(B[["xaln"]], "-"),
                yaln = c(B[["yaln"]], ey ),
                score = B[["score"]] + score_aln("-", ey))
```

```
answerc   <- list(x = x, y = y,
                  xaln = c(C[["xaln"]], ex),
                  yaln = c(C[["yaln"]], "-"),
                  score = C[["score"]] + score_aln(ex, "-"))
```

Now we need to figure out which of those three is the best, based on their scores, and return that one. We'll use a dead-simple way of doing that:

```
  bestanswer <- answera
  bestscore  <- answera[["score"]]
  if(answerb[["score"]] > bestscore) {
      bestanswer <- answerb
      bestscore  <- answerb[["score"]]
  }
  if(answerc[["score"]] > bestscore) {
      bestanswer <- answerc
      bestscore  <- answerc[["score"]]
  }

  return(bestanswer)

}
```

And that's it! We've turned a recursive definition (proved correct via induction) into a recursive algorithm*. Let's try it.

```
x <- char_vec("TATCGG")
y <- char_vec("TCTGG")
answer <- global_aln(x, y)        # Sweeeeeet!
print(answer)
```

* Yes, there was some debugging involved – don't for a moment think code like this springs from the forehead of Zeus!

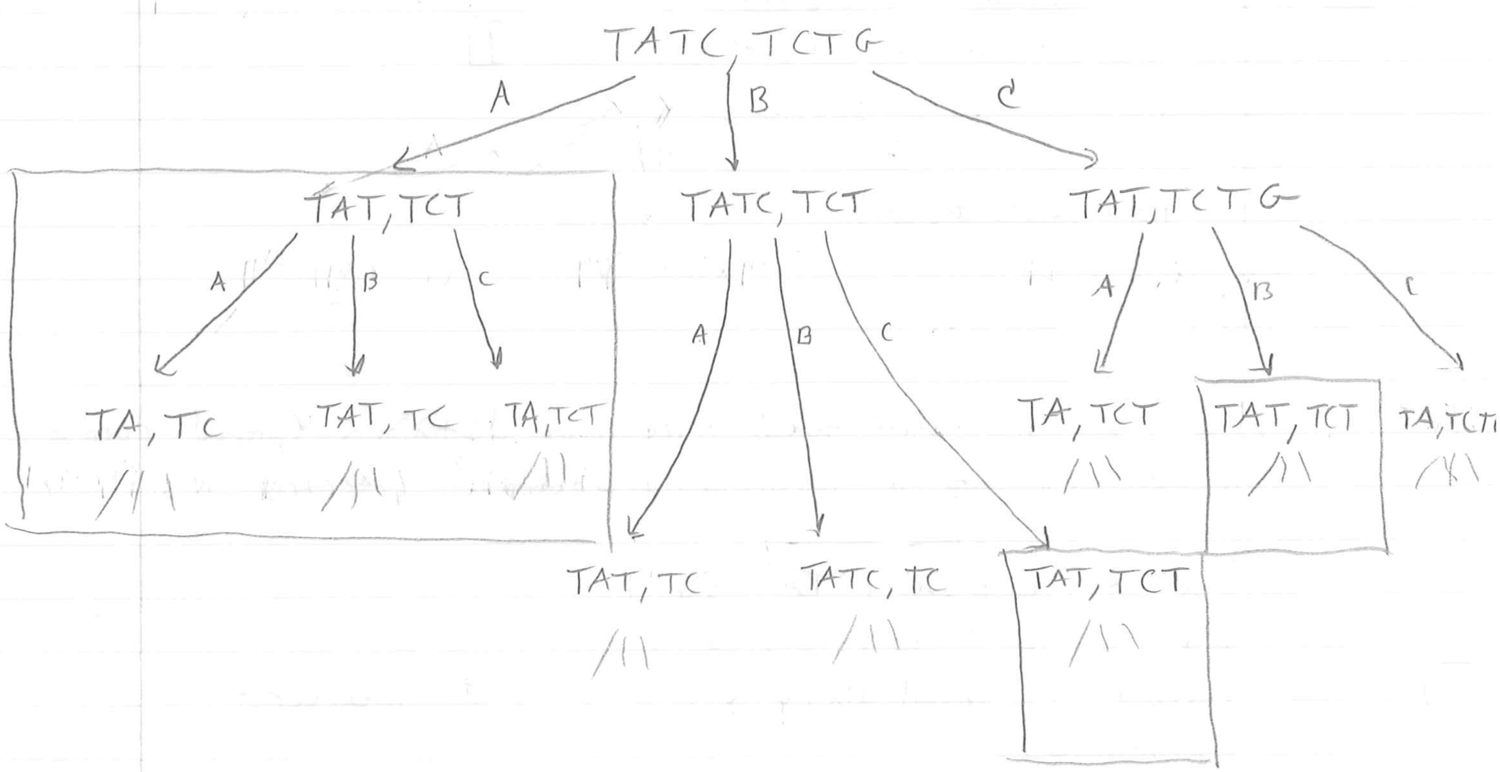
Memoization

We'd find that even slightly bigger instances of the problem take much longer to run.

```
x <- char_vec("TATCGGT")
y <- char_vec("TCTGGTCC")
answer <- global_aln(x, y)          # waiting...
print(answer)
```

This is because, much like the Fibonacci solution we looked at, there is a lot of recursive work being done. While each call to the fib() function resulted in two subcalls, each call to our function is resulting in three. Yikes!

But, also like the fib() function, many of the subproblems are overlapping. Consider the call tree for x = TATC and y = TCTG.



We've highlighted three of the biggest overlapping subproblems, but there are many more to find. Obviously, memoizing our global alignment function will provide big gains. We'll use as a key the unvec_char()'d version of the inputs separated by a comma (as in "TATC,TCTG"):

```
ALN_CACHE <<- hash()
global_aln <- function(x, y) {
    thiscall <- str_c(unvec_char(x), ",", unvec_char(y))
    if(has.key(thiscall, ALN_CACHE)) {
        return(ALN_CACHE[[thiscall]])
    }
    if(length(x) == 0 | length(y) == 0 | (length(x) == 1 & length(y) == 1)){
        ALN_CACHE[[thiscall]] <<- base_case(x, y)
        return(ALN_CACHE[[thiscall]])
    }
  [rest of function]
    ALN_CACHE[[thiscall]] <<- bestanswer
    return(bestanswer)
}
```

Now our function will run quite fast. Just how fast remains to be seen.

```
x <- char_vec("TATCGGTCTA")
y <- char_vec("TCTGGTCCAC")
answer <- global_aln(x, y)          # much faster
print(answer)
```

Inspecting the cache

In an effort to determine how much time the alignment takes, we can take a look at the cache - since there are no loops in our recursive call, each 'cell' of ~~the~~ cache represents just a few function calls once its been memoized. To do this, we're actually going to visualize the cache, for extra coolness.

But that means we have to get the data in the cache (the values in the hash, which have the answers, inputs, and scores) into a dataframe. Unfortunately, the 'hash' package for R doesn't make this easy, so we'll use rstacks as an intermediary. Our strategy will be:

```
create new rstack
for each key
    extract value (an 'answer' list)
    put value onto rstack
return as.data.frame( the rstack )
```

As we do this, we'll also take any elements of each answer list that are characters, and unvec_char() them, so x = c("A", "C", "T") will become x = "ACT" and so on.

(code on next page)

```r
hash_vals_to_df <- function(thehash) {
   tempstack <- rstack()
   for(key in keys(thehash)) {
      answerlist <- thehash[[key]]
      # unvec_char every element of answerlist if possible
      for(i in seq(1, length(answerlist))) {
         if(is.character(answerlist[[i]])) {
            answerlist[[i]] <- unvec_char(answerlist[[i]])
         }
      }
      tempstack <- insert_top(tempstack, answerlist)
   }
   return(as.data.frame(tempstack, stringsAsFactors = F))
}
```

(The beauty of R is that it's always possible to convert data into another format with a bit of trickery - too bad it's never in the format we want!)

Now we can easily turn such a cache into a nicely organized data.frame:

```r
cache_df <- hash_vals_to_df(ALN_CACHE)
print(head(cache_df))
```

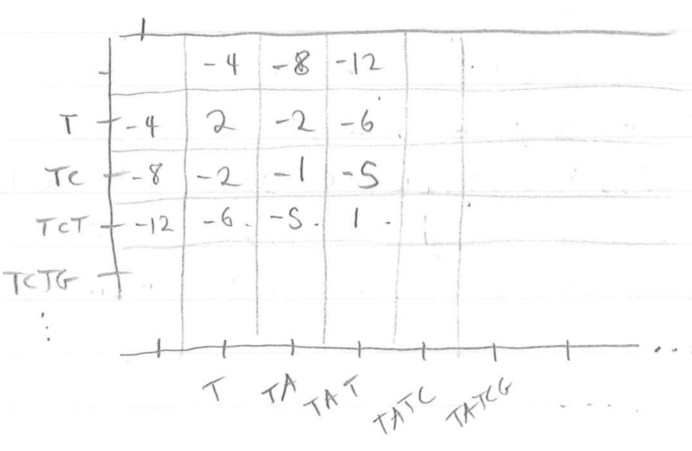
```
        x            y          xaln          yaln       score
   TATCGGT...   TCTGG...   TATCGGT...   TC-GG-...   -15
                  ⋮
```

We can clearly see that each row of the data.frame represents a solved subproblem - Let's plot these subproblems inputs (x and y) and just the score output; we'll do it in a grid, organized with longer x problems along the x axis, and longer y subproblems along the y (w/ longer ones toward the bottom), colored and labeled by score.

```
p <- ggplot(cache_df) +
     geom_tile(aes(x = reorder(x, nchar(x)),
                   y = reorder(y, -1 * nchar(y)),
                   fill = score)) +
     geom_text(aes(x = reorder(x, nchar(x)),
                   y = reorder(y, -1*nchar(y)),
                   label = score)) +
     theme_bw(16) +
     theme(axis.text.x = element_text(angle = 35, hjust = 1)) +
     coord_equal()

plot(p)
```

or scale_y_reverse...

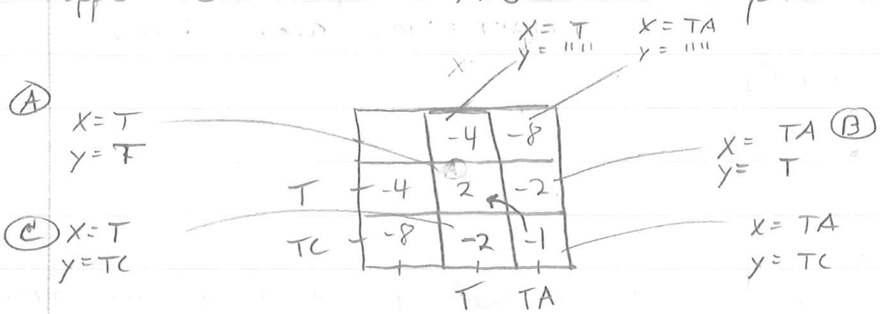
Ahah, very nice - our solved and cached subproblems are layed out in a nice grid:



How big is this grid? (And thus, how many subproblems were considered?) Well, if sequence X was of length n, and Y was of length m, the whole grid is roughly $n \times m$, or $O(nm)$, and this is how much work is required for the memoized solution - not great, but a lot better than the exponential function calls of the non-memoized one!

Subproblems are organized

Let's look at a small portion of this grid, the upper left hand corner to be precise:



First, note that the cells along the top and bottom correspond to base cases - if we wanted we could have considered x = "" y = "" to be a base case with score Ø; what's more interesting are the internal set of cells: in fact, the A, B, and C subproblems for X = TA, y = TC are in the three cells left and up! The solution for that cell did a recursive call for those other three.

Additionally, one of those three contributed to the 'best answer' in this case it would have been subproblem A. In a sense, we can say that the solution for X = TA, y = TC, came 'from' that cell, perhaps we'd say fromx = T, fromy = T.

We can visualize this too, if we encode this information in the answer lists that are cached.

For a base case, we'll simply say that the solution comes 'from' itself - it doesn't much matter, but it does make the eventual plot cleaner:

```
base_case <- function(xin, yin) {
    ...
    # change all answers like so:
    answer <- list(x = xin, y = yin,
                   xaln = xaligned, yaln = yaligned,
                   score = score_aln(xaligned, yaligned),
                   fromx = xin, fromy = yin)
    return(answer)
    ...
```

And in the recursive function, we need to determine the from information as we figure out which one is best:

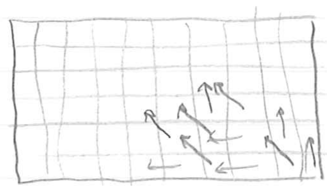
```
global_aln <- function(x, y) {
    ...

    bestanswer <- answera
    bestscore <- answera[["score"]]
    bestanswer[["fromx"]] <- A[["x"]]          # new
    bestanswer[["fromy"]] <- A[["y"]]          # new
    if(answerb[["score"]] > bestscore) {
        bestanswer <- answerb
        bestscore <- answerb[["score"]]
        bestanswer[["fromx"]] <- B[["x"]]      # new
        bestanswer[["fromy"]] <- B[["y"]]      # new
    }
```

```
if(answerc[["score"]] > bestscore ) {
    bestanswer  <- answerc
    bestscore   <- answerc[["score"]]
    bestanswer[["fromx"]]  <-  c[["x"]]
    bestanswer[["fromy"]]  <-  c[["y"]]
}
⋮
```

This information will end up in the plotable dataframe, so we can add a ggplot layer w/ some arrows representing where each 'best' came from. We also add a slight amount of jitter to the arrow end points for readability:

```
⋮
p <- ggplot(cache_df)
    ⋮
    geom_segment(aes(x = x, y = y, xend = fromx, yend = fromy),
                 arrow = arrow(length = unit(0.2, "cm"),
                               type = "closed"),
                 position = position_jitter(width = 0.1, height = 0.1),
                 color = "red") +
    ⋮
```

Now our visualization shows, for each cell, which subproblem produced the 'best' answer.



etc