Motivation: Biological sequence alignment

What can we learn from aligning two biological sequences?

- Identify regions of similarity and or difference, then
- Infer information about the new sequence from knowledge about the known sequence

Examples:

Predict function and structure of a novel protein

Identify mutations in genome sequencing data

Determine conserved (and thus important?) sequence

Defining sequence alignment

Problem: Find the optimal alignment with gaps between 2 sequences e.g., ATCGTAC and ATGTTAT.

```
ATCGTAC
ATGTTAT
1222344
```

Score = 4

You need an *objective function*, e.g., match is 1, mismatch or gap is 0

vs.

```
ATCGT-AC
AT-GTTAT
12234455
```

Score = 5

A trick question! You need to define an objective function to determine the optimal alignment. The objective function is one of the key knobs that we can adjust as part of the alignment process, and one of the key mechanisms to apply our biological intuition. This objective function produces alignments with the maximum number of matches (also known as the longest common subsequence). With the objective function, we can now formally define our problem as finding the alignment of 2 strings with the maximum score.

Naïve computational complexity

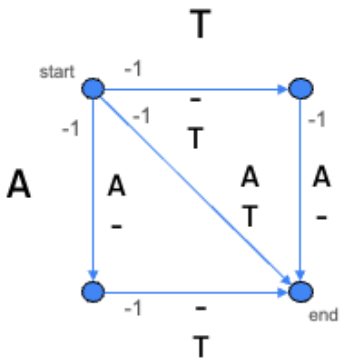
What is the complexity of exhaustive alignment of two strings of length *n* and *m*?

$$O(n,m) = \binom{n+m}{n} = \frac{(n+m)!}{n!m!}$$

Is this just bad, or impossibly bad?

We could represent the alignment as an interleaving of the two sequences. There are then m+n total positions, and we need to set n or m of them to determine the alignment.

Represent alignment as a “walk” through a city (with diagonal streets)



How many ways to align A to T with gaps?

A- -A A

-T T- T

We can determine that from the paths between the “start” and “end” nodes in our graph

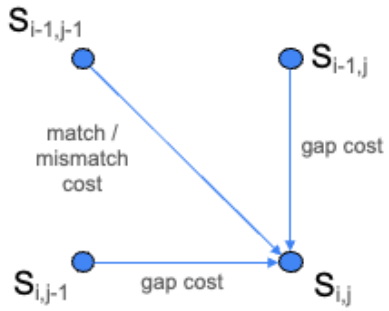
Each edge has an associated alignment (gap or match/mismatch) and a score

Final alignment and score is the combination of all edges (“steps”) in the path

3 ways to align A and T with gaps. We determine that via the number of paths from the start node to the end node. Each of the edge has an associated alignment and score. The overall score and alignment is combination of all edges (“steps”) in the path. The best alignment will be the highest scoring such path, the diagonal path or A aligned to T, in this instance.

This gives up an idea for how to solve our problem. The optimal alignment for A and T is the longest path from start to end, among the 3 possible paths. Here is start is the beginning of the string, but it doesn’t have to be. The same idea holds if ”start” is the optimal alignment (the longest path) of two smaller strings. What does that sound like? Recursion!

Global alignment recurrence relationship



$$S_{i,j} = \max \begin{cases} S_{i-1,j} + \text{“gap cost”} \\ S_{i,j-1} + \text{“gap cost”} \\ S_{i-1,j-1} + \text{“match/mismatch cost”} \end{cases}$$

$$S_{0,0} = 0$$

Which of the following is the correct recursive case?

```
def align(lft_seq, top_seq, lft, top):
 """Score aligning lft_seq[lft-1] & top_seq[top-1]"""
```

$$S_{i,j} = \max \begin{cases} S_{i-1,j} + \text{“gap”} \\ S_{i,j-1} + \text{“gap”} \\ S_{i-1,j-1} + \text{“match/mismatch”} \end{cases}$$

A

```
if lft_seq[lft-1] == top_seq[top-1]:
 dg = MATCH
else:
 dg = MISMATCH
return max(
 align(lft_seq, top_seq, lft, top) + GAP,
 align(lft_seq, top_seq, lft, top) + GAP,
 align(lft_seq, top_seq, lft, top) + dg
)
```

B

```
if lft_seq[lft-1] == top_seq[top-1]:
 dg = MATCH
else:
 dg = MISMATCH
return max(
 align(lft_seq, top_seq, lft, top-1) + GAP,
 align(lft_seq, top_seq, lft-1, top) + GAP,
 align(lft_seq, top_seq, lft-1, top-1) + dg
)
```

C

```
dg = MATCH
return max(
 align(lft_seq, top_seq, lft, top-1) + GAP,
 align(lft_seq, top_seq, lft-1, top) + GAP,
 align(lft_seq, top_seq, lft-1, top-1) + dg
)
```

D

```
if lft_seq[lft-1] == top_seq[top-1]:
 dg = MATCH
else:
 dg = MISMATCH
return (
 align(lft_seq, top_seq, lft-1, top-1) + dg
)
```

Answer: B

We want to implement the recursive relationship shown in top right.

What is the problem with?
A: Recursive problem is not getting smaller
C: “Diagonal” edges can be match or mismatch, but we don’t account for that here.
D: We are missing the vertical and horizontal edges

Putting it all together

```
def global_align(lft_seq, top_seq, lft, top):
    if lft == 0 and top == 0: # Source node
        return 0
    elif lft == 0: # Top row
        return global_align(lft_seq, top_seq, lft, top-1) + GAP
    elif top == 0: # First column
        return global_align(lft_seq, top_seq, lft-1, top) + GAP
    else: # Rest of the lattice
        if lft_seq[lft-1] == top_seq[top-1]:
            diag = MATCH
        else:
            diag = MISMATCH
        return max(
            global_align(lft_seq, top_seq, lft, top-1) + GAP,
            global_align(lft_seq, top_seq, lft-1, top) + GAP,
            global_align(lft_seq, top_seq, lft-1, top-1) + diag
        )
```

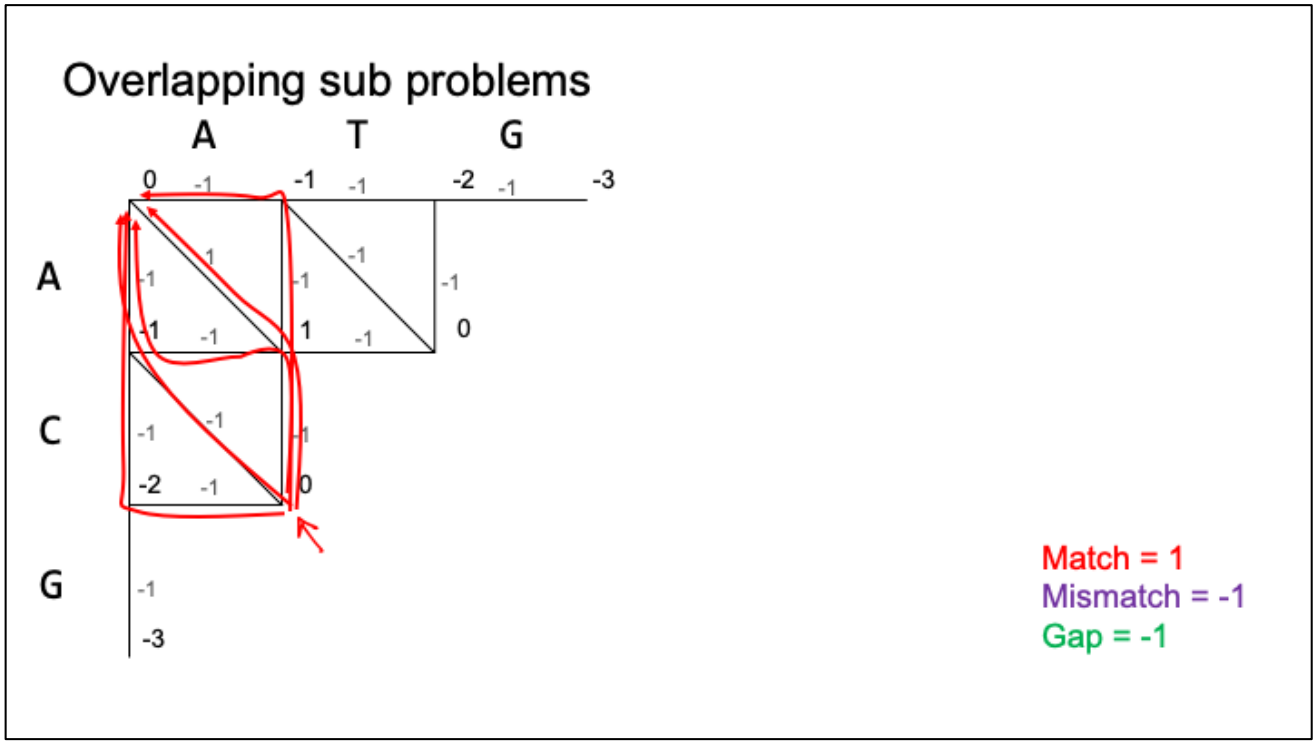
When aligning “ACG” and “ATG”, e.g.,
`align("ACG", "ATG", 3, 3)`,
how many times is `align("ACG","ATG", 1, 1)` called?

A. 1
B. 3
C. 9
D. More than 9

Answer: D

It is actually 13! And grows exponentially with the size of the strings, e.g., the predecessors, align(“ACG”, “ATG”, 1, 0) are called 25 times. But we get the same answer for all those calls.

What observe about this problem is that solution is expressed not just in terms of smaller versions of the same problem (termed subproblems), but in terms of overlapping subproblems, i.e., the same smaller version of the problem multiple times. Dynamic programming approaches improve the performance of these algorithms by only solving each sub-problem once!



If we think about computing (2,1), that is naively recursively exploring all of these possible paths. That is (2,1), depends on the score of (1,1), which depends on the (1,0), (0,0), (0,1) …

We can also think about this in terms of recursive calls, e.g.,

```
align(…, 2,1)
        align(…, 2,0)
                align(…., 1, 0)
                        align(…., 0, 0)
        align(…, 1, 0)
                align(…., 0, 0)
        align(…, 1, 1)
                align(…., 1, 0)
                align(…., 0, 0)
                align(…., 0, 1)
```

And so on…

Notice that we traverse the node (1,0) 3 times. But the optimal alignment for that node doesn't change. We should be able to compute that result once and reuse it. That is the key insight underlying dynamic programming approaches. When we

have overlapping subproblems, i.e., (1,0) is a subproblem for (2,0), (2,1), (1,1) etc., we can improve efficiency by only solving those subproblems once.

How could we modify our function to only solve each sub-problem once?

```
def global_align(lft_seq, top_seq, lft, top):
    if lft == 0 and top == 0: # Source
        return 0
    elif lft == 0: # Top row
        return global_align(lft_seq, top_seq, lft, top-1) + GAP
    elif top == 0: # First column
        return global_align(lft_seq, top_seq, lft-1, top) + GAP
    else: # Rest of lattice
        if lft_seq[lft-1] == top_seq[top-1]:
            diag = MATCH
        else:
            diag = MISMATCH
        return max(
            global_align(lft_seq, top_seq, lft, top-1) + GAP,
            global_align(lft_seq, top_seq, lft-1, top) + GAP,
            global_align(lft_seq, top_seq, lft-1, top-1) + diag
        )
```

Return recorded solution for (lft, top) if available

Record solution for (lft, top) for future use

What if we recorded the solution for each subproblem, e.g., for each combination of (lft, top)? Then we look up to see if you we previously solved that subproblem. If so, we return the recorded solution (without doing any more work), if not, we solve the subproblem as before but record the solution before returning.

[click]

This technique is termed “memoization”, that is we store/cache the results of the expensive computations do we don’t have perform those operations more than once. With that in place how many times should we solve each subproblem? Just once!

What is the most appropriate data structure for recording solutions to sub problems in our implementation?

A. List
B. Tuple
C. Set
D. Dictionary

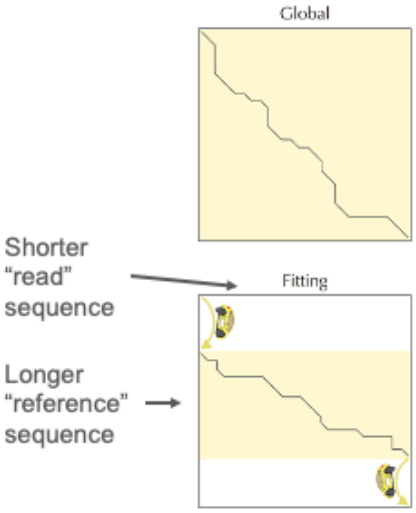
(lft, top) → score[lft, top]

Answer: D

We want to look up the solution by (lft, top) indices, that is want to store (and efficiently) lookup scores associated with specific positions. A dictionary is a natural choice, especially when we may not know which subproblems we will compute.

An alternate approach would be to use a list (or list of lists) that we fill in with values as we compute them (using lft and top as indices). We can do so here because we know we will explore all subproblems, that is all indices, starting from 0. And as a practical matter, this algorithm is often implemented in a bottom-up approach, by filling a 2-D array of scores, from 0 to length, instead of recursively from the “top down”.

Using our program to align sequencing reads



1. Don't penalize gaps in the first column
2. Start backtrack from the highest score in the last column

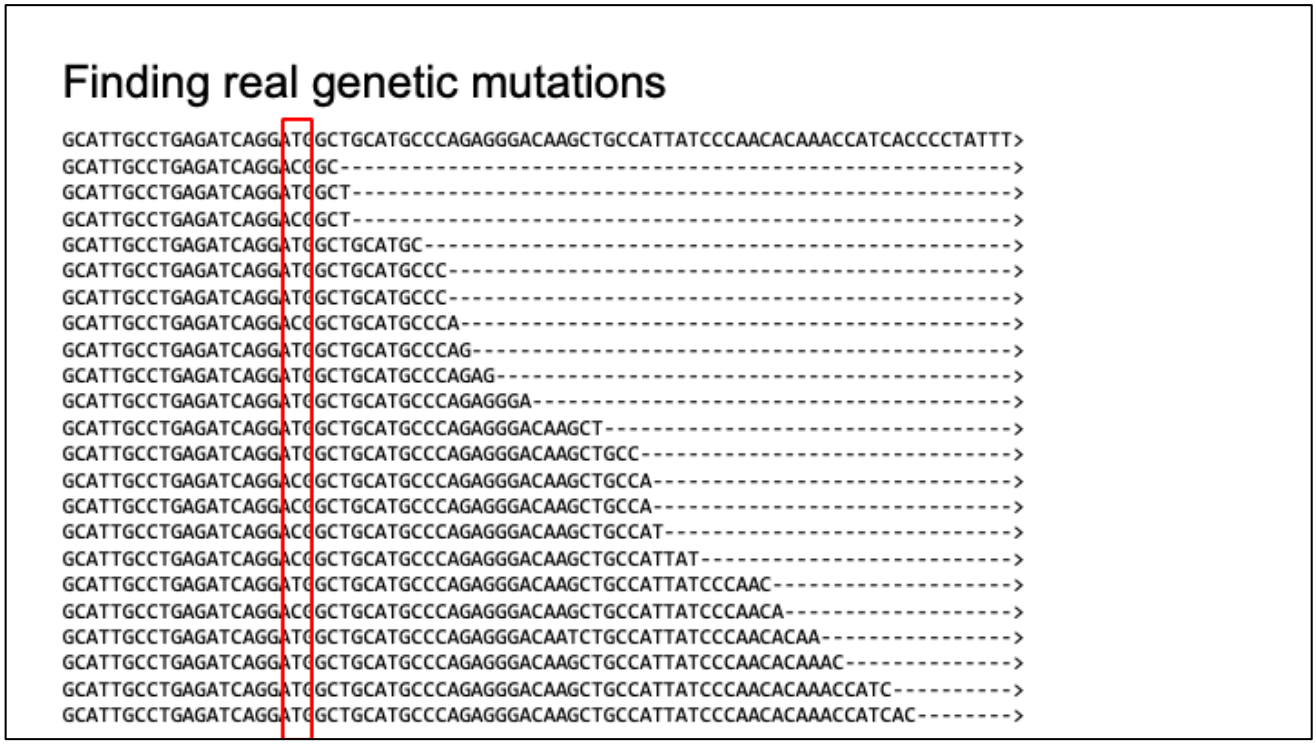
Compeau, Pevzner Bioinformatics Algorithms

Short-read sequencing (SRS) is one of the most commonly used technologies for sequencing the genome, i.e., inferring the genome sequence. It works by fragmenting the DNA into many pieces of approximately 500 nucleotides in length, and then sequencing 100–150 nucleotides of both ends of those fragments. Thus, the reads (each fragment yields two reads) are relatively short (hence the name). The first step is to align those fragments back to the reference genome to infer where in the genome they originated. This is conceptually the same problem we solved above, with one difference. One sequence is very long, the reference genome, and one is very short, the read sequence. The short sequence will necessarily have gaps and the beginning and the end that should not be penalized. To do so we use ”fitting” alignment, a modification, to your previous “global” alignment algorithm.

To eliminate those penalties, we add additional “free” or “taxi” edges into the lattice, that have 0 scores. We can jump from any node in the first column the source and from the sink to any node in the last column with no penalty. The effect of this is to align the shorter sequence, in its entirety, at the best location within the longer sequence. How could we implement this? What changes would be make in

our code?

- Change the recursive case for the first column to compute the max of the existing alignment score, and 0, i.e., it the score will never be less than 0
- Choose the “sink” to be the highest scoring node in the right column instead of the bottom right corner



We can put this together to align a set of real sequencing reads to the human reference genome (from a region on chromosome 7). We print all the reads at their respective alignments to generate a “pileup” visualization. Genome coordinates are across the top. The first line is the human reference genome, the remaining rows are the sequencing reads which provide evidence for this individual true genome sequence in this region. Each column represents the data we have for the individual’s genome at that location. Notice that most columns are the same from top to bottom. But the one highlighted in red has a mix of Ts and Cs. This indicates that this person has a heterozygous variant at this location, that is they inherited a “T” from one parent, and “C” from the other! Recall that humans are diploid organisms, that we is we inherit one copy of our genome from our mother and one from our father. Thus, we can have two different sequences (alleles) at the same location in the genome.

This data is from the NA12878 reference sample, and indeed this is a confirmed genetic variant in this individual. To learn more about it, check out

https://www.cs.middlebury.edu/~mlinderman/myseq/?vcf=https%3A%2F%2Fskylight.middlebury.edu%2F~mlinderman%2Fdata%2FNA12878.gatk-haplotype-annotated.vcf.gz&assumeRefRef=1

And enter chr7:141672604 into the search box.