COM SCI 122 Week 1

Aidan Jan

January 17, 2025

Read Matching (Continued)

Kmer Matching

- For each read, split read into kmers
- Find kmer matches in all genomes
- Use match counts to predict presence of organism in sample.'
- Issues:
  - May have mismatches (will be missed)
  - Multiple organisms may match same kmer (homology)
  - True organism may not be in set of genomes

Current Methods:

- Hash Tables
- BWT/FM Index

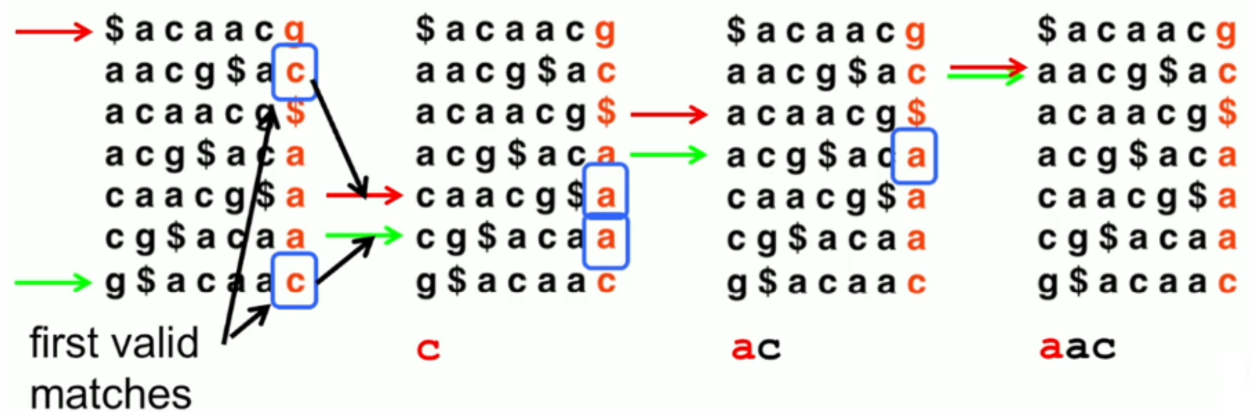
Hash Tables

- Store kmers and positions (use sliding window appraoch)
- We can look up an entry quickly.
- Not space efficient!

Exact Matching with FM Index

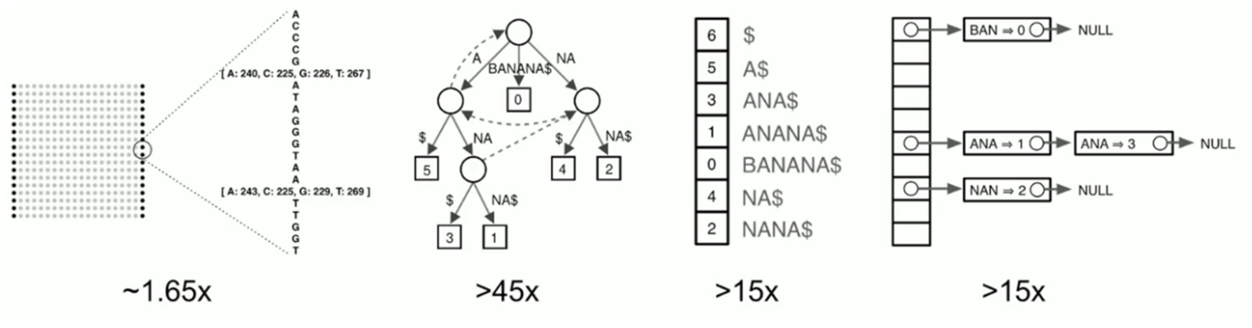
- Look up pattern in reverse
- Use 2 pointers to represent range of matches
- Find first valid match for next symbol in range.

Example: Searching for "AAC"



Further Reading: Bowtie sequence alignment

- Entire FM Index of DNA reference consists of:
  - BWT (same size as T)
  - Checkpoints (~15% size of T)
  - SA sample (~50% size of T)
- Total: ~1.65x the size of T



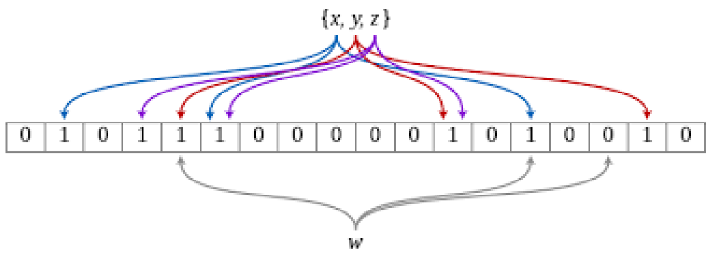
Metagenomics, the study of genomes across multiple organisms has indices so large that even BTW is too inefficient!

Kmer index requirements:

- Fast lookup
- Space requirements need to be sublinear (less space than total length of genomes)

Bloom Filter

- m bit array, initialized to all 0s
- l hash functions, each outputs an index of the m bit array
- For each element you want to add (in this case a kmer), you compute the l hash functions on the kmer and change the corresponding l bits to 1.
- For lookup, you check to see if all l hash functions on the kmer are 1.
- There is a possibility of false positives.



Code for bloom filter:

```
def create_filter(m):
    return [0] * m

def add_to_filter(filter, hash_functions, entry):
    for f in hash_functions:
        index = f(entry)
        filter[index] = 1
    return

def query_filter(filter, hash_functions, query):
    for f in hash_functions:
        index = f(query)
        if filter[index] != 1:
            return False
    return True
```

Optimizing Bloom Filters

- To control false positives (e), you can adjust l and m based on the number of elements you plan to add to the filter (n) which is usually known in advance.
- Optimal values:
  - $l = -\log_2(e)$
  - $m/n = -1.44 \log_2(e)$
- Examples:
  - If e = 0.01, then:
  - l = 6.64
  - m/n = 9.56

How efficient are bloom filters?

- Genome of length 10,000,000 (kmers)
- m/n ∼ 10 bits per kmer
- 10,000,000 bits ∼ 1.25 MB
- 10,000 genomes = 12.5 GB (remember, it's a log scale!)

Hash Functions

- We need multiple hash functions that are independent
- Family of hashes (universal hashes)
  - $h_{a,b} = ((ax + b) \mod p) \mod m$
  - a and b are chosen numbers
  - p is a large prime
  - m is the number of bins (from Bloom filter)
- Other hashes possible.
  - Splitting larger hashes into log m pieces.

Matrix of kmer sets

- Which organism contains a kmer?
- We can create 1 bloom filter for each organism
- Read each sequence 1 time to create the bloom filter. Only store filters in memory.
- We can then check each bloom filter with a kmer to see the set of organisms
- Process 1 read at a time.

Classifying a read

- Issues:
  - Some kmers match more than 1 organism
  - False positives from Bloom filters
- Appraoch:
  - For each kmer, obtain all matches for Bloom filters
  - Use consensus algorithm (or something better)

Metagenomic Trees

- All organisms are related (e.g., all mammals share a substantial amount of DNA)
- Kmers will match many organisms.
- Solution: Index Taxonomy Trees
- For each kmer, identify where it amtches in tree (LCA)
- When performing metagenomics, consider internal nodes as additional organisms.

Minimizers

- Example 100 length read breaks into kmers of length 31

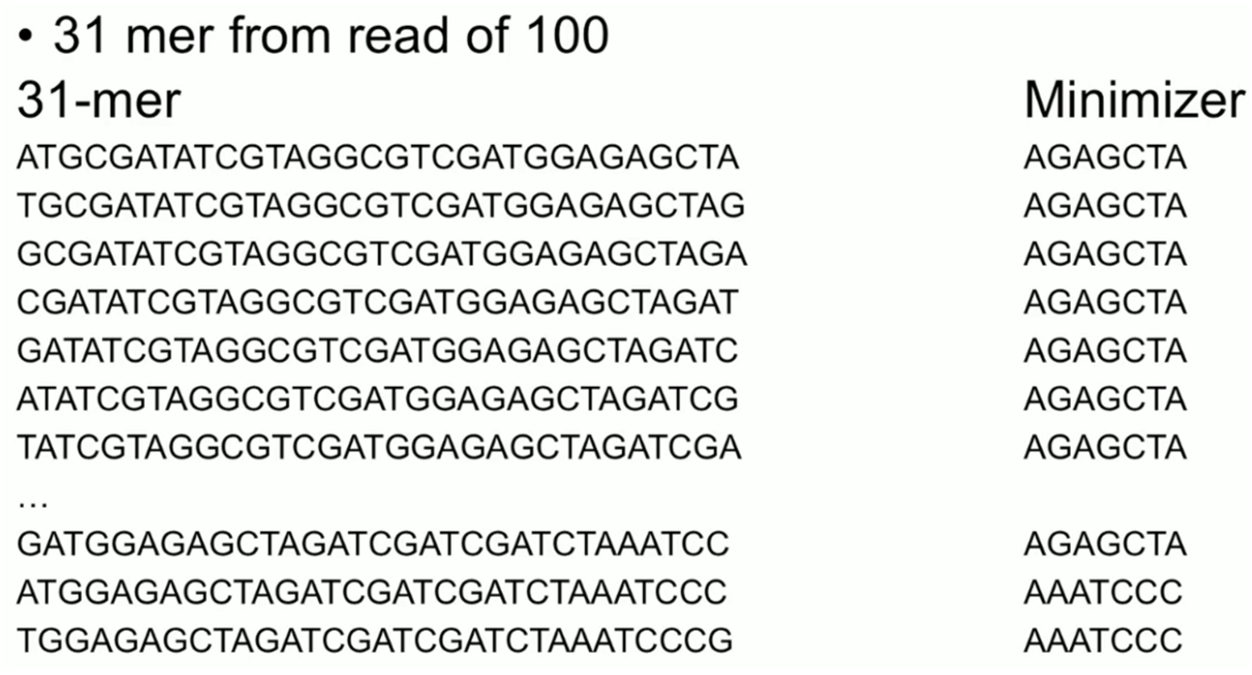
```
ATGCGATATCGTAGGCGTCGATGGAGAGCTAGATCGATCGATCTAAATCC
CGATCGATTCCGAGCGCGATCAAAGCGCGATAGGCTAGCTAAAGCTAGCA
```

- Kmer 1: `ATGCGATATCGTAGGCGTCGATGGAGAGCTA`
- Kmer 2: `TGCGATATCGTAGGCGTCGATGGAGAGCTAG`
- Kmer 3: `GCGATATCGTAGGCGTCGATGGAGAGCTAGA`
- etc.

A minimizer further reduces the size:

- 7 mers of first 31 length kmer:
- `ATGCGAT`, `TGCGATA`, `GCGATAT`, `CGATATC`, `GATATCG`, `ATATCGT`, `TATCGTA`, etc.
- Minimizer is the **first from the set alphabetically**. In our example, it would be `AAAGCGC`

Minimizers of Kmers



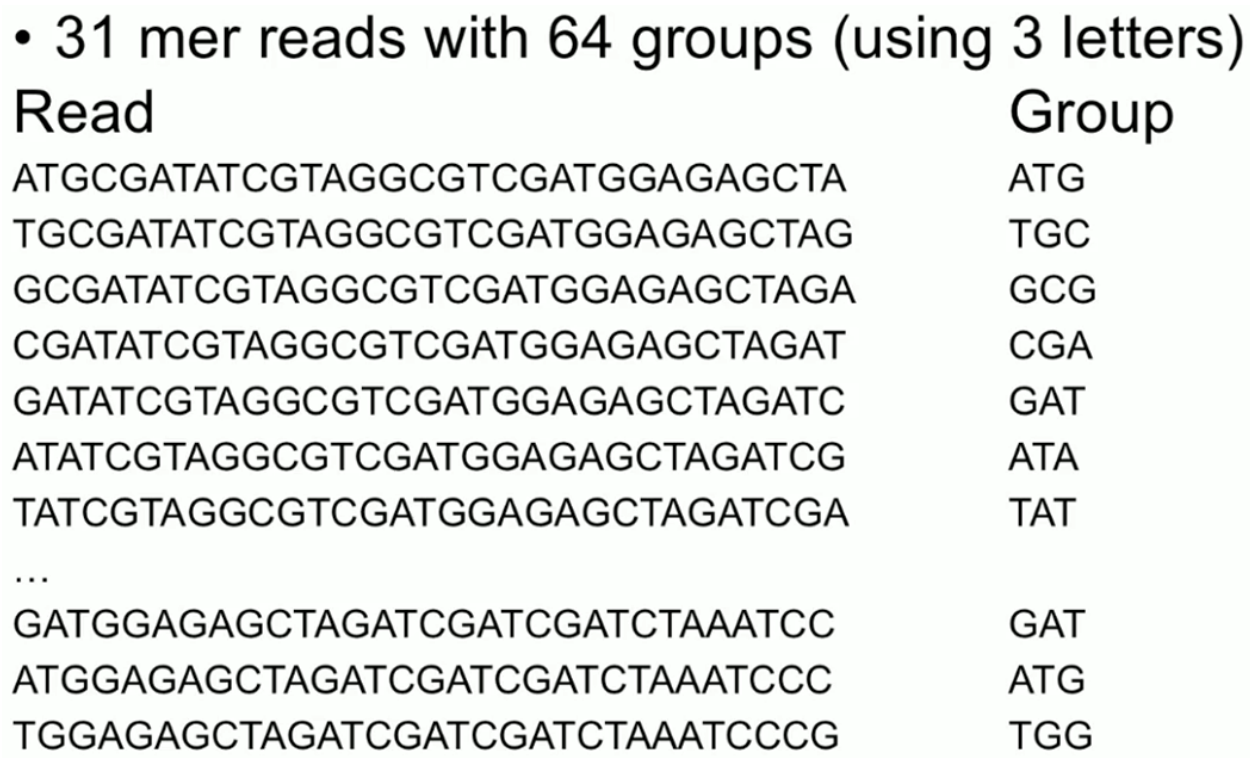
- 31 mer from read of 100

31-mer	Minimizer
ATGCGATATCGTAGGCGTCGATGGAGAGCTA	AGAGCTA
TGCGATATCGTAGGCGTCGATGGAGAGCTAG	AGAGCTA
GCGATATCGTAGGCGTCGATGGAGAGCTAGA	AGAGCTA
CGATATCGTAGGCGTCGATGGAGAGCTAGAT	AGAGCTA
GATATCGTAGGCGTCGATGGAGAGCTAGATC	AGAGCTA
ATATCGTAGGCGTCGATGGAGAGCTAGATCG	AGAGCTA
TATCGTAGGCGTCGATGGAGAGCTAGATCGA	AGAGCTA
…	
GATGGAGAGCTAGATCGATCGATCTAAATCC	AGAGCTA
ATGGAGAGCTAGATCGATCGATCTAAATCCC	AAATCCC
TGGAGAGCTAGATCGATCGATCTAAATCCCG	AAATCCC

- Notice that a minimizer stays for a while, and there are only two situations where it may change:
  - The original minimizer shifts off the genome on the left
  - A sequence scrolls in from the right and is alphabetically in front of the original.

Minimizer Application

- Counting kmers
- If kmers don't fit in memory, we need to store parts of the data structure on disk. How do we do this?
- Traditional approach: use first "n" letters in kmer and partition the kmers into 4^n groups.



• 31 mer reads with 64 groups (using 3 letters)

Read	Group
ATGCGATATCGTAGGCGTCGATGGAGAGCTA	ATG
TGCGATATCGTAGGCGTCGATGGAGAGCTAG	TGC
GCGATATCGTAGGCGTCGATGGAGAGCTAGA	GCG
CGATATCGTAGGCGTCGATGGAGAGCTAGAT	CGA
GATATCGTAGGCGTCGATGGAGAGCTAGATC	GAT
ATATCGTAGGCGTCGATGGAGAGCTAGATCG	ATA
TATCGTAGGCGTCGATGGAGAGCTAGATCGA	TAT
…	
GATGGAGAGCTAGATCGATCGATCTAAATCC	GAT
ATGGAGAGCTAGATCGATCGATCTAAATCCC	ATG
TGGAGAGCTAGATCGATCGATCTAAATCCCG	TGG

This is bad! It will take as much memory to store all the pointers to the next parts to form the genome, as the list of kmers itself!

In contrast, if we count with minimizers, we can map one minimizer to all of the reads containing that minimizer, as well as the counts for each kmer.

- This takes a lot of pointers too to piece together the genome. However, since all the reads with that minimizer are next to each other, we can exploit spatial locality and load all of them into memory at the same time.
- The number of memory operations this takes is considerably lower than the traditional method.

Why minimizers work?

- Different reasons for different application
  - Indexing/Lookup
  - Counting
  - Read Mapping
- Key properties:
  - Consecutive kmers are redundant
  - Similar sequences will have the same minimizers
- Behavior is highly parameter specific

Minimizers for Read Mapping

- We want to find exact matches of longer kmers (e.g., k=31)
- We can use minimizers to speed things up
  - k = 31, m = 20 (6x reduction in space)
- What are the possible errors/problems?
  - Minimizers match but kmer doesn't
  - Minimizer matches too many sequences.

Minimizer Index

- Only store minimizers
- Many fewer entries in the table (compared to kmer index)
- Many fewer kmers to look up
- Faster and more memory efficient!

The read processing goes:

- For each read, compute all kmers
- For each kmer, compute the minimizer
- Look up lcoation for each minimizer in index
- Match full read to each location in reference sequence.

When does kmer search work?

- Sequences have mutations and reads have errors
- At least one minimizer must not have a change

Examples: 1000000 Length Genome

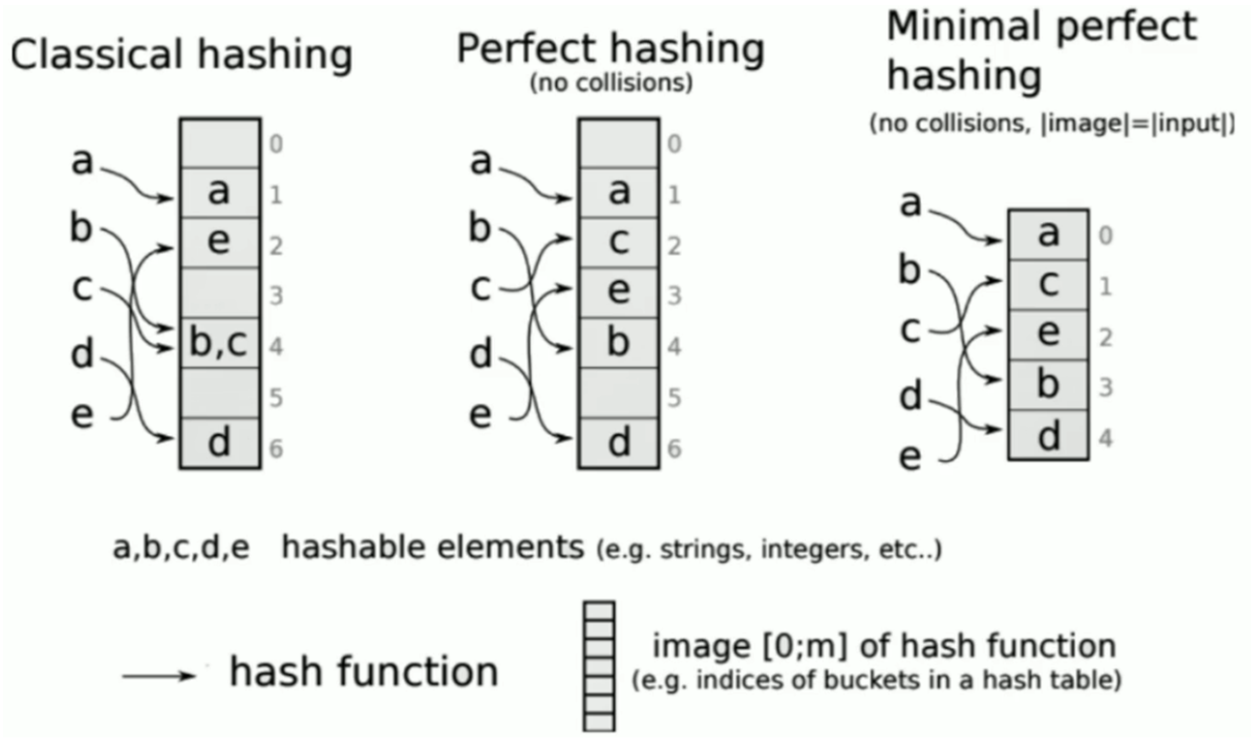
- Kmer length (k) = 15, minimizer length (m) = 10
  - 310k total minimizers in genome (30%)
  - Count: 1, Number: 153831
  - Count: 2, Number: 49904
  - Count: 3, Number: 13760
  - Count: 4, Number: 3035
  - Count: 5, Number: 611
  - Count: 6, Number: 88
  - Count: 7, Number: 10
  - Count: 8, Number: 1
- if k = 25, m = 15
  - 181k total minimizers in genome (18%)
  - Count: 1, Number: 181581

  - Count: 2, Number: 62
- if k = 21, m = 15
  - 271k total minimizers in genome (27%)
  - Count: 1, Number: 271674
  - Count: 2, Number: 94
- if k = 31, m = 20
  - 167k total minimizers in genome (16%)
  - Count: 1, Numnber: 167878

Examples: Minimizers Simulated Read Error

- Reads of length 50 with e errors/mutations
- If at least 1 minimizer doesn't change, we can map it to the genome.
- k = 15, m = 10
  - e = 4, Unmapped Rate = 0.5%
- k = 25, m = 15
  - e = 4, Unmapped Rate = 4.6%
  - e = 3, Unmapped Rate = 17%
- k = 21, m = 15
  - e = 2, Unmapped Rate = 1.3%
  - e = 1, Unmapped Rate = 9.2%
- k = 35, m = 25
  - e = 1, Unmapped Rate = 21.2%

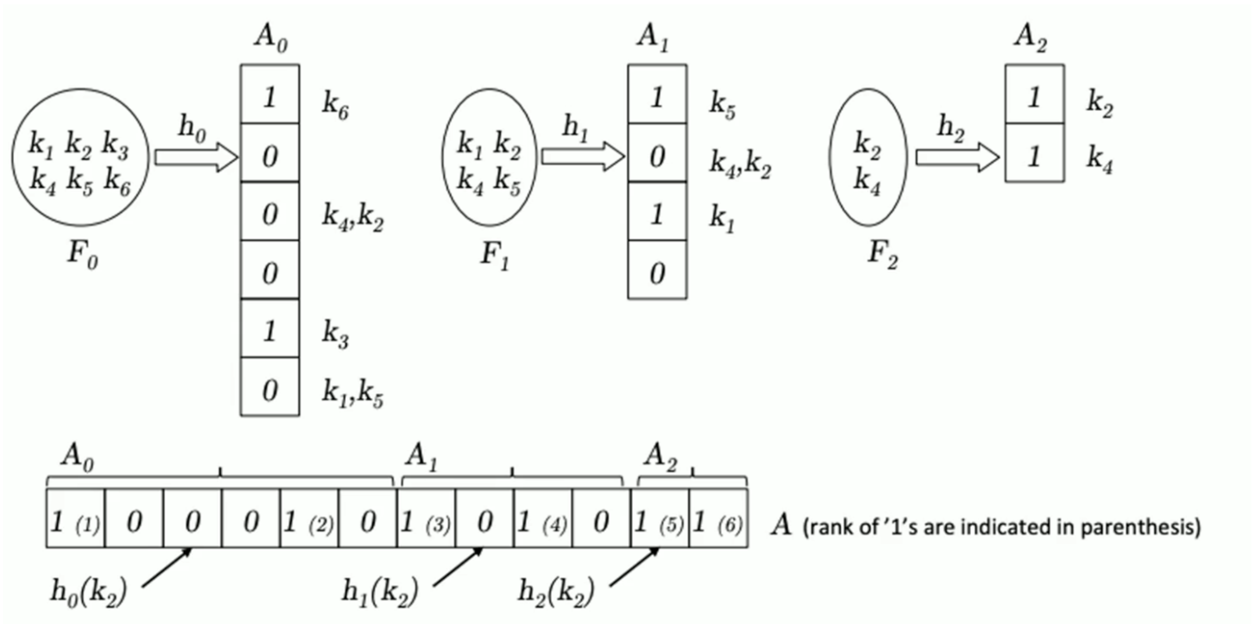
Minimum Perfect Hashing


Classical hashing
Perfect hashing
(no collisions)
Minimal perfect hashing
(no collisions, |image|=|input|)
a,b,c,d,e hashable elements (e.g. strings, integers, etc..)
hash function
image [0;m] of hash function
(e.g. indices of buckets in a hash table)

A minimum perfect hashing function is a function that maps each of N elements to a number from 1 to N.

- Can be constructed in many ways

BBHash (2017)



BBHash essentially does many hashes in a row using a classical hash

- First hash uses all the n elements with n indices of the resulting table.

- Those entries with collisions or no entries mapped, as well as the elements, are repeated with a second hash.
- This pattern continues until there are no collisions left.
- The result is the concatenation of all three structures.
- To query, you apply the first hash. If the result is 1, you retrieve the value. If the result is a 0, you apply the second hash, and go to the next section, etc.

Dynamic Programming

- Go search this up on google, there are many examples of the type of problems, as well as the ways to solve them.

Alignment With Dynamic Programming

- Suppose you have two strings `ATACGTA` and `ATTACGA`.
- The optimal alignment is:

```
AT-ACGTA
ATTACG_A
```

- "edit distance of 2"
- How do we find this alignment?

Steps:

1. To start, let L and R represent the two strings to be aligned. Let l and r denote the lengths of L and R, respectively. Let e represent the maximum number of errors allowed.
2. Create a matrix, M with $l + 1 + e$ rows and $r + 1 + e$ columns, representing the indices in each string.
3. Populate the 0-th row and column with values of the indices.

0	-1	-2	-3	$\dots$
-1				$\dots$
-2				$\dots$
-3				$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

4. Let i and j be counters for rows and columns, respectively. Going row by row through the table, fill in each tile at location (i, j) following the rules:
   - If the character L_i matches R_j, then fill $M_{i,j}$ with the value of $M_{i-1,j-1} + 1$
   - If the character does not match, then fill $M_{i,j}$ with $\text{MAX}(M_{i-1,j-1},\ M_{i-1,j},\ M_{i,j-1}) - 1$.
5. Repeat until the bottom right tile is filled.
6. Backtrack through the graph, starting from the index with the maximum value.
   - If the value of $M_{i,j}$ is equal to $M_{i-1,j-1} + 1$, then move diagonally up.
   - Otherwise, move to the cell (either left, up-left, or up) with the highest value.
   - Assuming you are trying to match R to L, (L is the reference)

   - Each time you move up left with a $+1$ difference, the letters match
   - Each time you move left, there was an insertion in R.
   - Each time you move up, there was a deletion in R.
   - Each time you move up left, with a -1 difference, there was a substitution in R.

7. Repeat until you reach the beginning of the sequence, the top left corner.

8. This algorithm is $O(m \cdot n)$.

Metagenomics: Key Concepts

- A sample contains multiple organisms
- Sequencing of the sample obtains reads from each of the present organisms
- There is a library of reference genomes available.
- Only a small fraction of the genomes are present in the sample.
- A difficulty is that there are repeated regions in the genomes.
- A read may match multiple organisms because of the repeats.

Metagenomic Formats

- Reference Sequences and reads format is the same as Read Mapping Project
- Your goal is to predict the "origin" of each read

Left is the read number, right is the genome number it came from. The goal is to make a guess for every read.

```
>read_0 Genome_Number_85
>read_1 Genome_Number_67
>read_2 Genome_Number_91
>read_3 Genome_Number_85
>read_4 Genome_Number_85
>read_5 Genome_Number_73
>read_6 Genome_Number_63
>read_7 Genome_Number_91
>read_8 Genome_Number_99
>read_9 Genome_Number_63
```