**CIS129 Final Presentation**

**Introduction**

As the importance of computer processing increases in modern society, the need for efficient programs becomes a critical issue. Ours is an era of “Big Data” and methods of processing the huge amounts of data that are continuously being collected is increasingly important. As a physician, I am aware of great strides that have been made in genetic research. The deciphering of the human genome was initially considered an extremely difficult and expensive task, due to the cost of biochemical analysis and computational processing of results. But biochemical procedures improved, computers gained greater computing power, and computer programs became more efficient. Eventually the cost of processing large amounts of genetic data became so reasonable that millions of people were able to get their genetic profile done for about 100 dollars via the startup company “23 and Me”. In the context of an introductory Python programming course, consideration of the design of an improved search algorithm would illustrate how programming expertise could contribute to better data processing efficiency and lower costs for consumers. Improved program efficiency could and has contributed to scientific advancements in human genetic research.

* + A single human genome contains over **3 billion base pairs**.
  + Sequencing data from even one genome produces **hundreds of gigabytes** of raw data.
  + Analyzing multiple genomes for comparative studies (e.g., across populations) scales to **petabytes** of data.
  + Early manual genome sequencing took **years**.
  + Modern computing tools, including optimized algorithms and parallel processing,

allow researchers to analyze genomes in **hours to days**.

* + Machine learning helps **predict the function** of unknown genes and **classify genetic variants** by potential health impact.
  + Databases like **GenBank** or **Ensembl** store and manage genomic data efficiently.

Software allows rapid search, annotation, and cross-referencing of genetic sequences

Specialized algorithms identify genes, regulatory elements, mutations, and structural variations

**Human Genome Project (1990–2003)**:

Would not have been possible without computers to sequence, assemble, and annotate billions of DNA bases

**Design Approach**

In our class, we learned how to do simple operations like element comparison of strings. We have also studied algorithms to make comparisons between strings and thus we could sort strings by their alphabetic numerically assigned values. Genetic information obtained from biochemical analysis is reported as sequences of letters representing the nucleotides that make up DNA. These sequences or genomes are represented like this, ATCGTG….. combinations of the four nucleotide bases A, T, C, G. These sequences can be sorted by computer programs and the sequences can be compared by computer programs. If we wanted to compare two genomes, or two parts of a genome, the question becomes, what is the most efficient way to accomplish this’. The simplest approach is to simply compare all the elements of two sequences we want to compare, element by element. This is a comprehensive approach but very time consuming. This is called “linear search”.

**Solution Proposal**

A proposed solution is to first break the longer sequence into short snippets called k-mers. A k-mer of 3 character length would be called a tri-mer. The longer sequence could be broken down into shorter k-mer sequences and the k-mer sequences could be sorted alphabetically. The sorted k-mer sequences would contain the position information for where that k-mer existed in the longer genome. Then we can compare the short k-mer sequence, referred to as the test pattern we are searching for, to the sorted k-mers of the larger genome (referred to as the target sequence). Since the sorted k-mer sequence has position information, and because it is alphabetically sorted, we don’t need to compare every element. Instead, we can divide the sorted sequence in half, and only check the half that contains the target k-mer. We can repeat this process for the half that we know contains the k-mer, finding the mid-point of the substring, dividing it in half, and only continuing the search in the half of the target sequence that based on the alphabetical value of the k-mer (or test pattern), must contain the k-mer. This approach could be characterized as “divide and conquer”, and it is made possible by first breaking the long target sequence into k-mers, recording position information, and sorting the resulting k-mer list alphabetically. In this section I run linear search and note the number of comparisons required to find a pattern in a sequence of 15 numbers. I run the program twice with linear search and once with binary search to demonstrate the advantage of binary search. Namely, that

On average, binary search will require fewer comparison steps.

Linear Search Algorithm

* def linearSearchShort(data,key,comparisons):
* for index, value in enumerate(data):
* print('This is index ', index, 'This is value for match ', value)
* comparisons += 1
* if value == key:
* position.append(index)
* return position, comparisons
* return -1
* This strategy begins at the beginning of the list and checks each position for a match with the target pattern.
* When the first match is found, the index position is added to a
* List and the list is returned with the index position of the match.
* If the target pattern occurs early in the sequence, the calculating time is short.

If the target pattern is near the end of the sequence or is the last element, then all or nearly all the elements must be compared

This is the random data and if we search for element 25 the search is fast, we only need to do two comparisons.

[83 25 65 43 73 74 21 21 66 82 67 65 54 65 66]

This is index 0 This is value for match 83

This is index 1 This is value for match 25

But if we search for value 66, then we need to iterate through the entire sequence and compare every element to find a match.

This is index 0 This is value for match 83

This is index 1 This is value for match 25

This is index 2 This is value for match 65

This is index 3 This is value for match 43

This is index 4 This is value for match 73

This is index 5 This is value for match 74

This is index 6 This is value for match 21

This is index 7 This is value for match 21

This is index 8 This is value for match 66

This is index 9 This is value for match 82

This is index 10 This is value for match 67

This is index 11 This is value for match 65

This is index 12 This is value for match 54

This is index 13 This is value for match 65

This is index 14 This is value for match 66

Now we have to do comparisons with every element in the sequence, we have to do 15 comparisons.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

By Comparison, binary search algorithm first sorts the sequence, then bisects the sorted sequence, and proceeds by divide and conquer.

Simple Binary Search Algorithm Strategy

* Sort the list
* Find the middle of the sorted list
* Compare the value of the target pattern to the middle character
* If there is a match, return the index position
* If there is no match, then consider the half of the list that should contain the test pattern based on sort insertion comparison
* Check the midpoint of the substring. Repeat the above process for smaller substrings, until the entire list has been checked

Example of High, Low, and Middle Values

* This is a copy made with np.sort()
* [14 19 24 26 29 29 29 37 39 52 60 68 70 75 86]
* Enter integer between 10 and 91 or -1 to quit
* 29
* The low postion is 0
* high position is (string length -1) or = 14
* This is the middle 7

A white background with blue text

AI-generated content may be incorrect.

Check the Middle Position, Reassign Middle  
Based on if Key Sorts Above or Below Key

* if key == data[middle]:
* location = middle # location in the current middle
* elif key < data[middle]: # middle element is too high
* high = middle -1 # eliminate the lower half
* else:
* low = middle +1 # eliminate the higher half
* middle = (low + high + 1)//2 # relocate the middle
* return location, countBisects

`````A white text with blue writing

AI-generated content may be incorrect.

For this example we only had to do 3 comparisons using the binary search approach.

(It really didn’t matter that there were 3 values of 29 here, it would be equally fast with only one element of 29.)

When the target sequences become very long, the advantage of binary search becomes exponentially more efficient.

Real World Applications

This part of the presentation tries to demonstrate the magnitude of the problem of working with genetic “big data”.

The program does linear search to on a small scale sample to demonstrate **how genetic data** could be processed and searched by matching

2 or 3 character nucleotide sequences against a longer snippet, and then it carries out the same process on a “tiny” real virus genome sequence.

First, the short examples with made up genetic sequences.

Here is a string composed of letters A,C,T,G representing

nucleotides in a small piece of DNA

I am going to search for a pattern within this string

first the pattern AT and second the pattern TAG

This is the target string **GCTACGATCTAGAATCTA**

This is the first pattern to check for in the string AT

First run with k-mer equal to 2-mer or dimer

Here is the dimer GC and the position index 0

Here is the dimer CT and the position index 1

Here is the dimer TA and the position index 2

Here is the dimer AC and the position index 3

Here is the dimer CG and the position index 4

Here is the dimer GA and the position index 5

Here is the dimer AT and the position index 6

Here is the dimer TC and the position index 7

Here is the dimer CT and the position index 8

Here is the dimer TA and the position index 9

Here is the dimer AG and the position index 10

Here is the dimer GA and the position index 11

Here is the dimer AA and the position index 12

Here is the dimer AT and the position index 13

Here is the dimer TC and the position index 14

Here is the dimer CT and the position index 15

Here is the dimer TA and the position index 16

This is testTime1 0.265789270401001

The index locations of the test pattern are in this list [6, 13]

Notice that a lot of printing was done for a small string of letters.

Linear search took 0.265789270401000976562500000000 seconds

Second run with k-mer equal to 3-mer or tri-mer

This is test pattern TAG

This is k-mer 3

Remember, this is the target string **GCTACGATCTAGAATCTA**

Here is the dimer GCT and the position index 0

Here is the dimer CTA and the position index 1

Here is the dimer TAC and the position index 2

Here is the dimer ACG and the position index 3

Here is the dimer CGA and the position index 4

Here is the dimer GAT and the position index 5

Here is the dimer ATC and the position index 6

Here is the dimer TCT and the position index 7

Here is the dimer CTA and the position index 8

Here is the dimer TAG and the position index 9

Here is the dimer AGA and the position index 10

Here is the dimer GAA and the position index 11

Here is the dimer AAT and the position index 12

Here is the dimer ATC and the position index 13

Here is the dimer TCT and the position index 14

Here is the dimer CTA and the position index 15

This is testTime2 0.2144308090209961

This [9] is the index location of the pattern TAG in the target string

**REAL WORLD DATA**

Now, lets introduce our “tiny” lambda virus sequence, just to get an idea of the magnitude of the challenge.

Consider finding a small nucleotide sequence in the small lambda virus. The small lambda virus has about 50,000 nucleotides in its sequence.

We might need to know how many times a sequence like “TAG” occurs in the sequence. It would be interesting to compare the length of time it

Takes to do the search by linear versus binary search. Here is a tiny part of the lambda virus.

>gi|9626243|ref|NC\_001416.1| Enterobacteria phage lambda, complete genome

GGGCGGCGACCTCGCGGGTTTTCGCTATTTATGAAAATTTTCCGGTTTAAGGCGTTTCCGTTCTTCTTCG

TCATAACTTAATGTTTTTATTTAAAATACCCTCTGAAAAGAAAGGAAACGACAGGTGCTGAAAGCGAGGC

TTTTTGGCCTCTGTCGTTTCCTTTCTCTGTTTTTGTCCGTGGAATGAACAATGGAAGTCAACAAAAAGCA

GCTGGCTGACATTTTCGGTGCGAGTATCCGTACCATTCAGAACTGGCAGGAACAGGGAATGCCCGTTCTG

CGAGGCGGTGGCAAGGGTAATGAGGTGCTTTATGACTCTGCCGCCGTCATAAAATGGTATGCCGAAAGGG

ATGCTGAAATTGAGAACGAAAAGCTGCGCCGGGAGGTTGAAGAACTGCGGCAGGCCAGCGAGGCAGATCT

CCAGCCAGGAACTATTGAGTACGAACGCCATCGACTTACGCGTGCGCAGGCCGACGCACAGGAACTGAAG

AATGCCAGAGACTCCGCTGAAGTGGTGGAAACCGCATTCTGTACTTTCGTGCTGTCGCGGATCGCAGGTG

AAATTGCCAGTATTCTCGACGGGCTCCCCCTGTCGGTGCAGCGGCGTTTTCCGGAACTGGAAAACCGACA

TGTTGATTTCCTGAAACGGGATATCATCAAAGCCATGAACAAAGCAGCCGCGCTGGATGAACTGATACCG

GGGTTGCTGAGTGAATATATCGAACAGTCAGGTTAACAGGCTGCGGCATTTTGTCCGCGCCGGGCTTCGC

TCACTGTTCAGGCCGGAGCCACAGACCGCCGTTGAATGGGCGGATGCTAATTACTATCTCCCGAAAGAAT

CCGCATACCAGGAAGGGCGCTGGGAAACACTGCCCTTTCAGCGGGCCATCATGAATGCGATGGGCAGCGA

CTACATCCGTGAGGTGAATGTGGTGAAGTCTGCCCGTGTCGGTTATTCCAAAATGCTGCTGGGTGTTTAT

GCCTACTTTATAGAGCATAAGCAGCGCAACACCCTTATCTGGTTGCCGACGGATGGTGATGCCGAGAACT

TTATGAAAACCCACGTTGAGCCGACTATTCGTGATATTCCGTCGCTGCTGGCGCTGGCCCCGTGGTATGG

CAAAAAGCACCGGGATAACACGCTCACCATGAAGCGTTTCACTAATGGGCGTGGCTTCTGGTGCCTGGGC

GGTAAAGCGGCAAAAAACTACCGTGAAAAGTCGGTGGATGTGGCGGGTTATGATGAACTTGCTGCTTTTG

ATGATGATATTGAACAGGAAGGCTCTCCGACGTTCCTGGGTGACAAGCGTATTGAAGGCTCGGTCTGGCC

AAAGTCCATCCGTGGCTCCACGCCAAAAGTGAGAGGCACCTGTCAGATTGAGCGTGCAGCCAGTGAATCC

CCGCATTTTATGCGTTTTCATGTTGCCTGCCCGCATTGCGGGGAGGAGCAGTATCTTAAATTTGGCGACA

AAGAGACGCCGTTTGGCCTCAAATGGACGCCGGATGACCCCTCCAGCGTGTTTTATCTCTGCGAGCATAA

TGCCTGCGTCATCCGCCAGCAGGAGCTGGACTTTACTGATGCCCGTTATATCTGCGAAAAGACCGGGATC

TGGACCCGTGATGGCATTCTCTGGTTTTCGTCATCCGGTGAAGAGATTGAGCCACCTGACAGTGTGACCT

TTCACATCTGGACAGCGTACAGCCCGTTCACCACCTGGGTGCAGATTGTCAAAGACTGGATGAAAACGAA

AGGGGATACGGGAAAACGTAAAACCTTCGTAAACACCACGCTCGGTGAGACGTGGGAGGCGAAAATTGGC

GAACGTCCGGATGCTGAAGTGATGGCAGAGCGGAAAGAGCATTATTCAGCGCCCGTTCCTGACCGTGTGG

CTTACCTGACCGCCGGTATCGACTCCCAGCTGGACCGCTACGAAATGCGCGTATGGGGATGGGGGCCGGG

TGAGGAAAGCTGGCTGATTGACCGGCAGATTATTATGGGCCGCCACGACGATGAACAGACGCTGCTGCGT

GTGGATGAGGCCATCAATAAAACCTATACCCGCCGGAATGGTGCAGAAATGTCGATATCCCGTATCTGCT

GGGATACTGGCGGGATTGACCCGACCATTGTGTATGAACGCTCGAAAAAACATGGGCTGTTCCGGGTGAT

CCCCATTAAAGGGGCATCCGTCTACGGAAAGCCGGTGGCCAGCATGCCACGTAAGCGAAACAAAAACGGG…this goes on for about 700 lines…

Third run with very long lambda virus genome, print disabled

The first occurance line 77 position 12 or index 995

The second occurance line 125 position 50 or index 4182

The third occurance line 126 position 40 or index 4242

The length of the lambda virus genome is 48507 almost 50,000 characters!

How long is the human genome ? A single human genome contains over **3 billion base pairs**

The test pattern is still **TAG**. How many **TAG** trimers are there in the lambda virus genome of length 48507?

The length of the index list is 215; there are this many matches

The test pattern is TAG

This is testTime3 0.006540060043334961

Linear search took 0.006540 seconds

**This is the index list result for TAG with 215 matches** [995, 4182, 4242, 7884, 10559, 12998, 13984, 17826, 18441, 18609, 19591, 20693, 20741, 22392, 22612, 22616, 22701, 22922, 22945, 22970, 23147, 23181, 23200, 23414, 23421, 23452, 23460, 23490, 23622, 23685, 23916, 23940, 23952, 23969, 24051, 24059, 24113, 24149, 24161, 24219, 24328, 24402, 24413, 24514, 24546, 24602, 24636, 24746, 24759, 24833, 24904, 24995, 25028, 25124, 25208, 25234, 25392, 25487, 25652, 25855, 25927, 25987, 26036, 26047, 26129, 26212, 26425, 26452, 26650, 26703, 26706, 26886, 26909, 26979, 27082, 27085, 27115, 27129, 27200, 27219, 27236, 27283, 27287, 27302, 27467, 27549, 27614, 27625, 27660, 27934, 27971, 28064, 28639, 28742, 28810, 28844, 29210, 29319, 29497, 29939, 29961, 30105, 30211, 30530, 30687, 30884, 31306, 31509, 31713, 31807, 31863, 32306, 32976, 33061, 33133, 33171, 33192, 33295, 33304, 33313, 33554, 33995, 34279, 34285, 34297, 34375, 34388, 34496, 34685, 34884, 34982, 35042, 35209, 35256, 35383, 35408, 35707, 35734, 35856, 35915, 35929, 36001, 36072, 36178, 36196, 36343, 36420, 36630, 36905, 37260, 37324, 37688, 38914, 38961, 39692, 39857, 40206, 40328, 40745, 40775, 40805, 40937, 40964, 40985, 41099, 42410, 42542, 42903, 42926, 43026, 43108, 43127, 43277, 43906, 44216, 44511, 44618, 44659, 44749, 44753, 44844, 44959, 44984, 44990, 45114, 45160, 45393, 45491, 45501, 45606, 46184, 46429, 46735, 46742, 46801, 46978, 47047, 47066, 47091, 47113, 47168, 47224, 47340, 47344, 47376, 47489, 47565, 47584, 47711, 47807, 47851, 47979, 48161, 48253, 48452]

Now search for a longer pattern using linear search

Let us search for the 5-mer TTTTT.

The new pattern is TTTTT of length 5

From line 66 at position 16 this pattern TTTTT with 5 characters first occurs

This is testTime4 0.0077075958251953125

Linear search took 0.007708 seconds

The length of the index list is 133 there are this many matches

This is the index list result for TTTTT [84, 142, 171, 2366, 3091, 3092, 3546, 4305, 6119, 6120, 6121, 6132, 6133, 6134, 6217, 7740, 8622, 8623, 8644, 10311, 10632, 13540, 13541, 13917, 17956, 17957, 19637, 19638, 22753, 22798, 22799, 22800, 22801, 22840, 23177, 23543, 23544, 23659, 23681, 23708, 23716, 23765, 23771, 23772, 23773, 23911, 23912, 24074, 24264, 24265, 25048, 25049, 25083, 25135, 25395, 25396, 25482, 25483, 26039, 26040, 26068, 26479, 26756, 26757, 26792, 26834, 26922, 26923, 26924, 27308, 27667, 27668, 27730, 27731, 28169, 28631, 28632, 29627, 30554, 30840, 30866, 30867, 30868, 31405, 31406, 31530, 31531, 32937, 32938, 33006, 33236, 33237, 33819, 33891, 33892, 34269, 34920, 35542, 35665, 35666, 35682, 35683, 35754, 36411, 36520, 36521, 36962, 37081, 37537, 37538, 37740, 37862, 37868, 37869, 37870, 37931, 37932, 38163, 38164, 38165, 40507, 40623, 41716, 44778, 44779, 46747, 46748, 46749, 46774, 47275, 47504, 48212, 48355]

Now we will do a long 27 character test pattern on the lambda virus genome

The new pattern is TTACAACCCCTACAGTTTGATGAGTAT of length 27

From line 539 this pattern TTACAACCCCTACAGTTTGATGAGTAT with 27 characters

This is testTime4 0.005220890045166016

Linear search took 0.005221 seconds

**This is the index list of the 27 character test pattern on the lambda virus genome [34470]**

**The length of the index list is 1, so only on instance of this test pattern in the lambda virus genome.**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

As an extra bonus, we can count the frequency of the occurrence of the different nucleotides in the lambda virus

To see if they all occur in equal proportions. The relative proportion of nucleotides might be useful in identifying

certain organisms.

**Do the four nucleotides appear in the lambda virus genome in equal frequencies?**

The letter count in lambda virus for character A is 12334

The letter count in lambda virus for character T is 11986

The letter count in lambda virus for character C is 11362

The letter count in lambda virus for character G is 12820

**Regarding computer calculating time versus computer printing time:**

Note that when we work on the lambda virus that printing out preliminary steps must be disabled, because the printing process is very slow

And the step of showing all the printing takes forever so I have set flags to prevent printing when the target sequence is the lambda virus genome.

If you change those flags, the program tries to print every step, it takes forever, and it looks like the program is in an infinite loop, but it isn’t, it’s just trying to print all the steps.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

We can run the binary program on a truly short DNA sequence and print out all the steps the program is taking to illustrate the “divide and conquer” algorithm. That is shown here:

Here is the sequence that is the target of **our search t = 'GCTACGATCTAGAATCTA'**, and the pattern that we will look for is ‘**TAG’**. We are using a binary algorithm

And the program will print out the “divide and conquer” steps as it narrows down the search by halving the sorted string and resetting the definitions of high, low, and middle of the target string and subsequent substrings.

the length of t is 18

the length of p is 3

The index.self is the sorted list of trimers from the target sequence with their associated positions collected as a tuple. The index.self is a

List of tuples.

the index.self is [('AAT', 12), ('ACG', 3), ('AGA', 10), ('ATC', 6), ('ATC', 13), ('CGA', 4), ('CTA', 1), ('CTA', 8), ('CTA', 15), ('GAA', 11), ('GAT', 5), ('GCT', 0), ('TAC', 2), ('TAG', 9), ('TCT', 7), ('TCT', 14)]

the length of index.self is 16 (This is because you can only get 16 groups of 3 out of a list of 18 characters, do it yourself manually)

**--- Binary Search Steps for finding LEFTMOST occurrence o**f TAG ---

Step 1: Examining index 7 with value ('CTA', 8)

Current search range: [0:15]

**Target is greater than mid value, searching right half**

Step 2: Examining index 11 with value ('GCT', 0)

Current search range: [8:15]

**Target is greater than mid value, searching right half**

Step 3: Examining index 13 with value ('TAG', 9)

Current search range: [12:15]

Found k-mer match at index 13!

This is the leftmost occurrence

--- Binary Search Steps for finding RIGHTMOST occurrence of TAG ---

Step 1: Examining index 7 with value ('CTA', 8)

Current search range: [0:15]

Target is greater than mid value, searching right half

Step 2: Examining index 11 with value ('GCT', 0)

Current search range: [8:15]

Target is greater than mid value, searching right half

Step 3: Examining index 13 with value ('TAG', 9)

Current search range: [12:15]

Found k-mer match at index 13!

This is the rightmost occurrence

Standard bisect\_left result: 13

Standard bisect\_right result: 14

Number of occurrences: 1

Collecting all occurrences:

Found occurrence at position 9

print the type index <class '\_\_main\_\_.Index'>

match results list contains the indices for the beginning of the pattern match [9]

matches in text are TAG in the genome t at index 9

**Summary of Findings**

The binary algorithm has already been invented and it is appropriately called “binary search”. When the sequences to be analyzed and compared are very large, the efficiency advantage of binary search is impressive. It is exponentially faster than “linear search”.

**Relative Efficiency Example**

| **Dataset Size (n)** | **Linear Search (O(n))** | **Binary Search (O(log n))** |
| --- | --- | --- |
| 1,000 items | ~1,000 comparisons | ~10 comparisons |
| 1,000,000 items | ~1,000,000 comparisons | ~20 comparisons |

* **Binary search is exponentially faster than linear search — but only works on sorted data**.
* If the data isn't sorted and can't be sorted efficiently, **linear search is necessary**.

Printing is a very slow operation for computers compared to just doing computations. If I print out intermediary steps to show how an algorithm works on a large database, the time needed to do the printing requires a relatively large amount of computing time. If I stop printing intermediate steps the program finishes much faster. I suppose I could write in a condition that prints a fraction of the large data intermediate steps, so that one could see a fraction of the intermediate steps on the lambda virus run.

**Open Questions**

* Future question: *How can algorithms be optimized or parallelized to analyze petabyte-scale datasets in near-real time?* Cloud-native tools, GPU acceleration, and quantum computing may offer breakthroughs?
* Challenge: Deep learning models (like AlphaFold) are powerful but often "black boxes."
* Future question: *Can we design algorithms that explain why a prediction or classification was made in biological terms?* Need: Trustworthy AI in genomics for clinical adoption
* Challenge: A single reference genome does not represent global human diversity.
* Future question: *How do we efficiently search across multiple genomes (pan-genomes) for large structural variants, copy-number variations, and complex rearrangements?*
* Challenge: Genomics is just one layer. Epigenomics, transcriptomics, proteomics, and metabolomics add huge complexity.
* Future question: *How can we develop algorithms to integrate and search across diverse biological data types to understand disease mechanisms holistically?*
* Challenge: Genomic data is sensitive and often subject to regulatory constraints (e.g., HIPAA, GDPR).
* Future question: *Can we perform secure, federated searches across private genomic datasets without revealing individual identities?*

**References**

**John Hopkins School of Public Health – Coursea – Genomic Data Science Specialization**

<https://www.coursera.org/specializations/genomic-data-science>

**Intro to Python for Computer Science and Data Science, Deitel & Deitel**

**and the internet and ChatGpt**