Ternary Search Tries (TST)

A simple pattern-matching and substring search algorithm for genome assembly

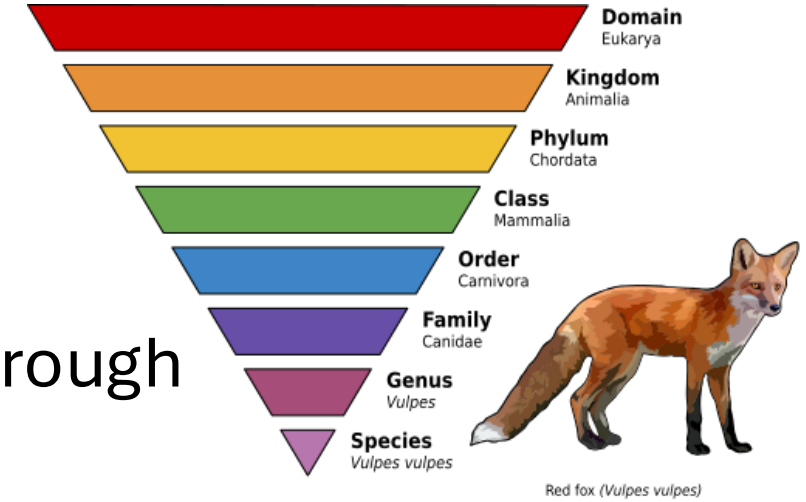
Trinity, Dongwan, Robert, Priscilla

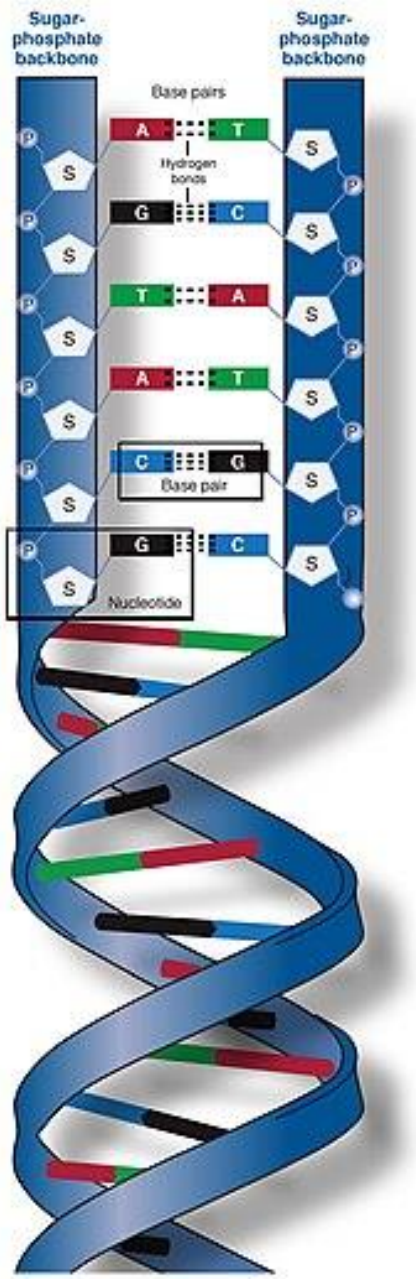
Problem

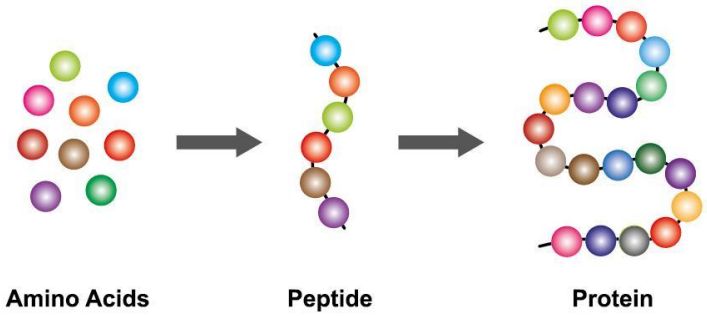
- Long sequences to search through
  - DNA/RNA
  - Proteins
  - Taxonomies

- Goal
  - When given a sequence, quickly determine

  if it is in our dataset already



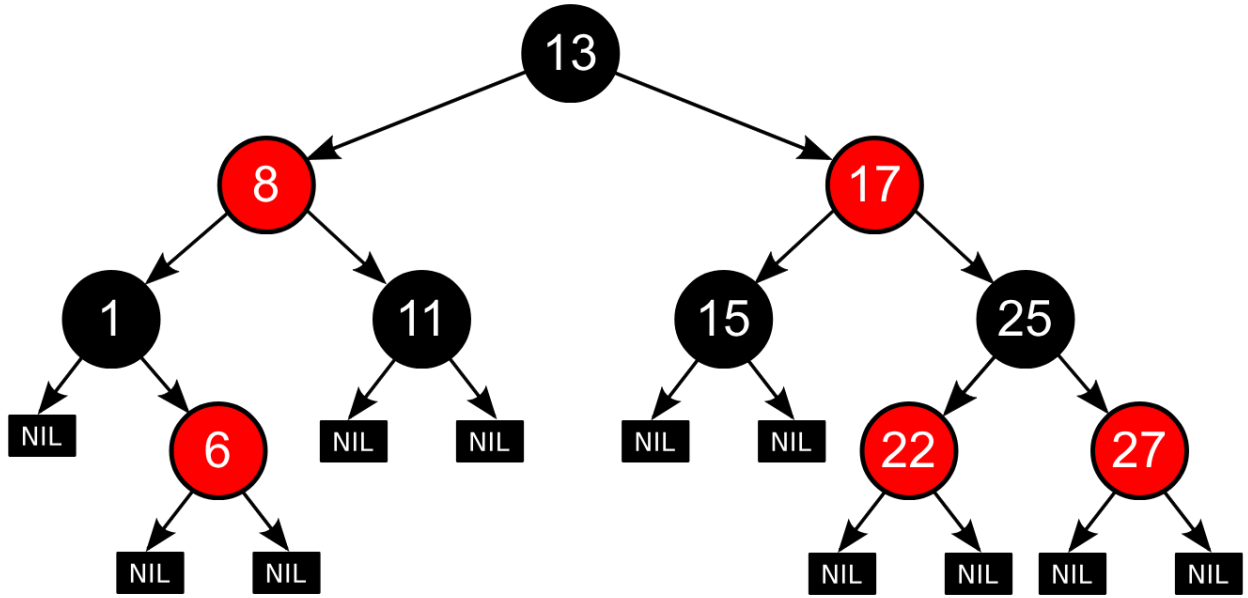




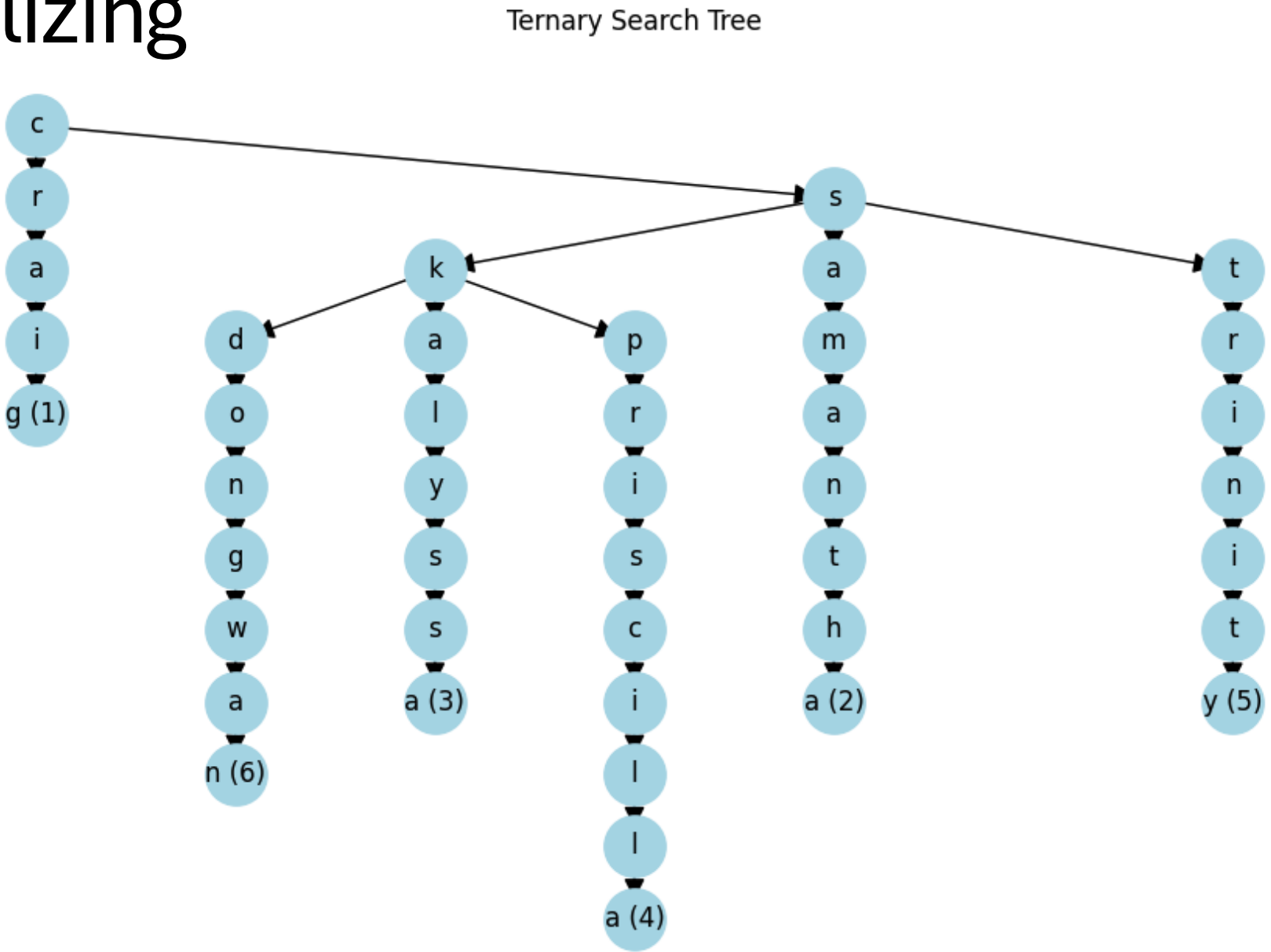
Possible Data Structure Options

- List
- Set
- TST
- R-way Trie
- Red-black Tree

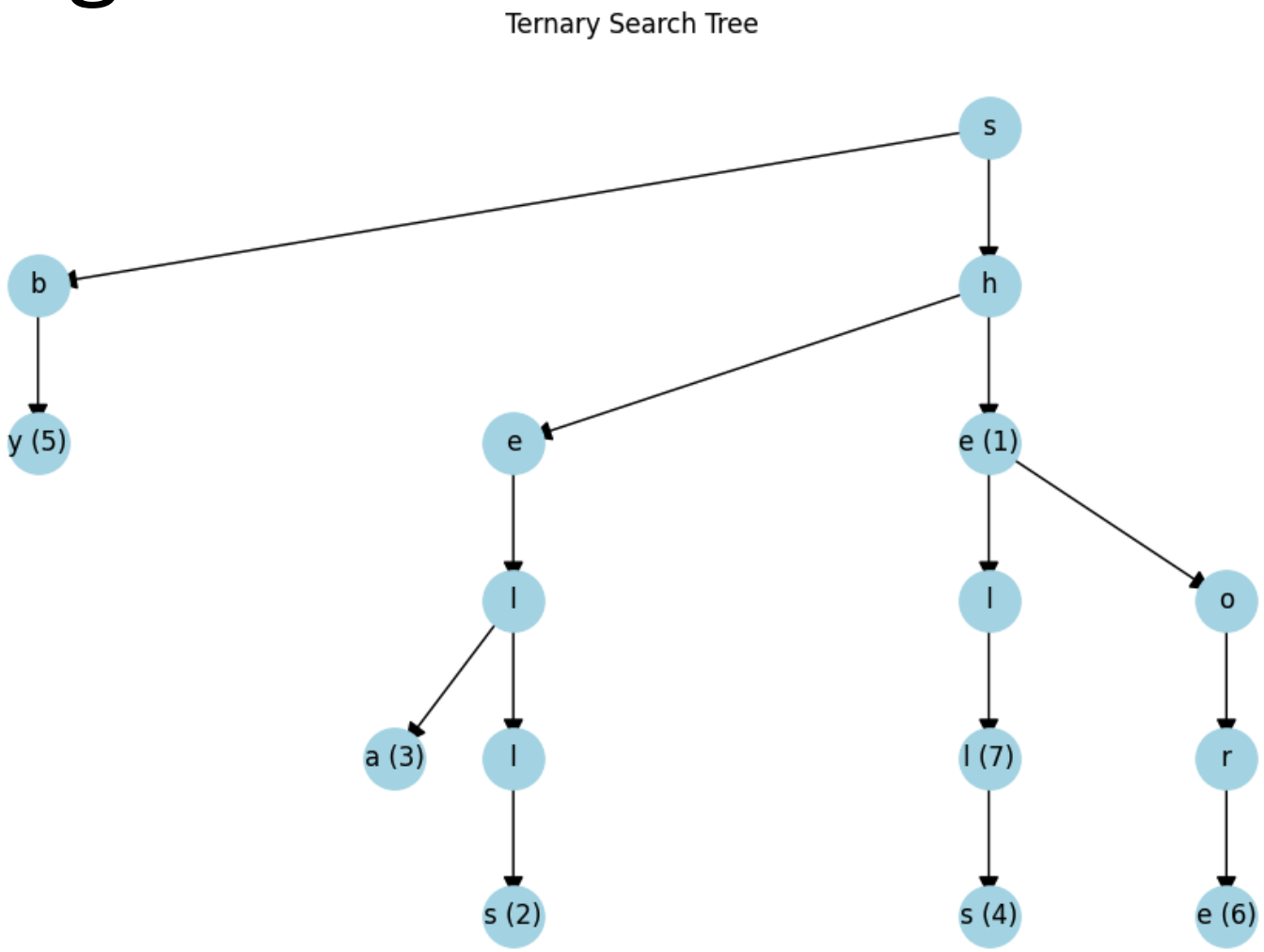
List = [1,2,2,3,3,4,5] → Set = {1,2,3,4,5}



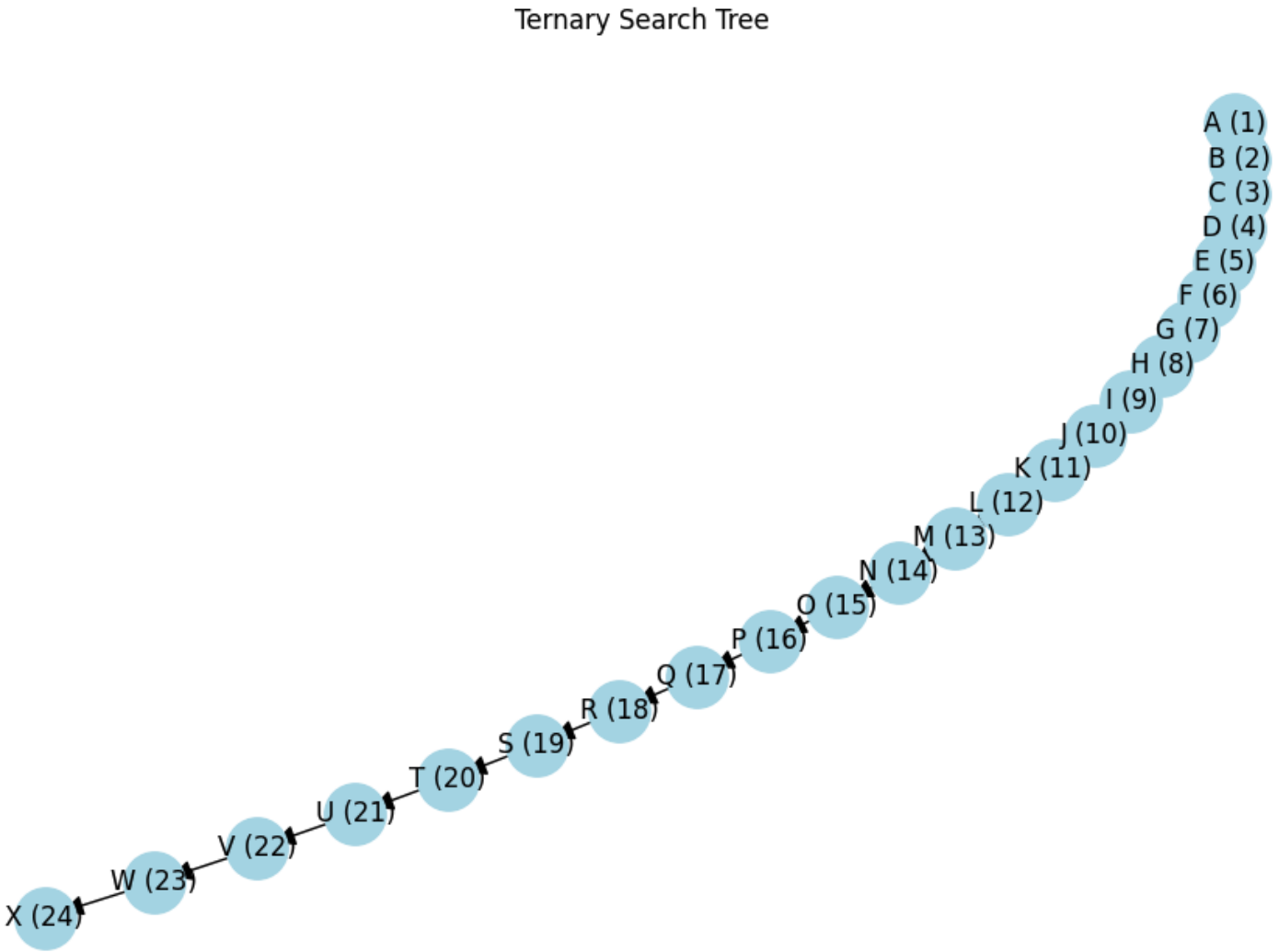
Visualizing



Visualizing



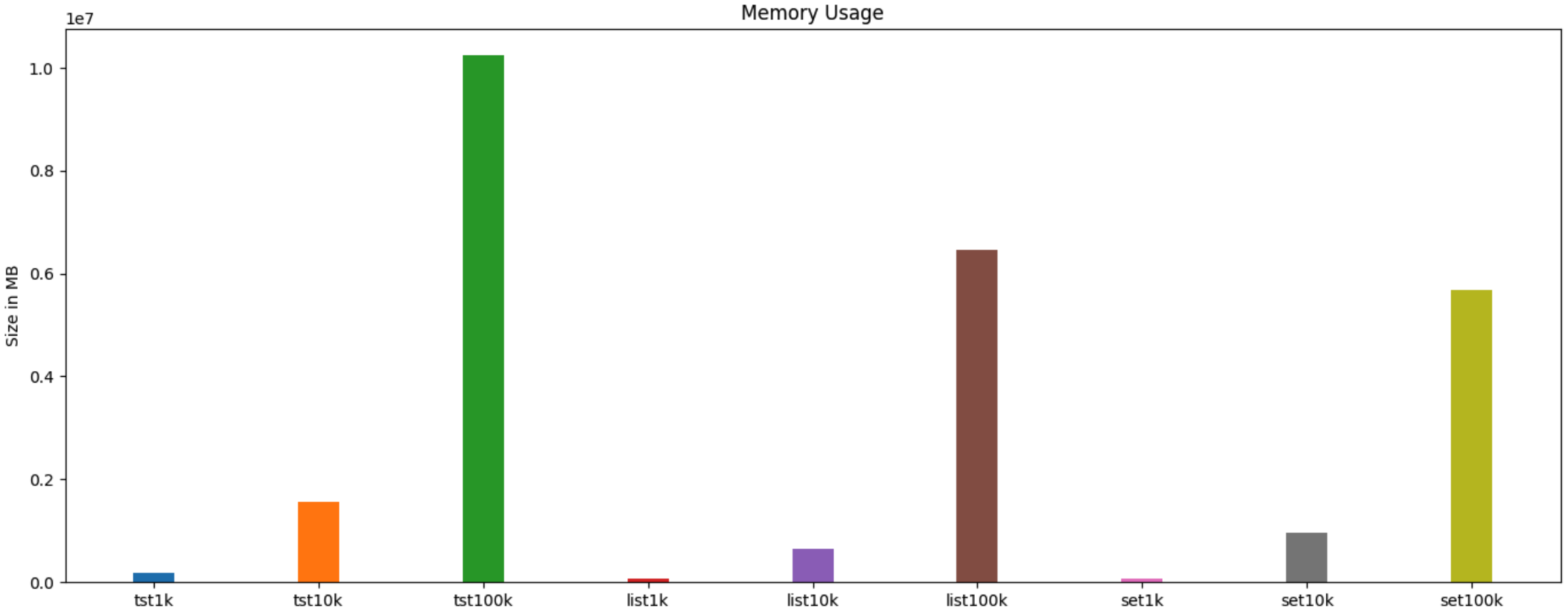
Visualizing



Benchmarking

```python
def generate_random_word(min_length=3, max_length=12):
    # words are just random DNA letters (A, C, T, G), and of random lengths between 3 and 12 letters by default
    # Note that doing random letters from A-Z doesnt have much overlap so the TST took up a lot of space
    # having longer lengths didnt have much overlap either
    s = []
    for _ in range(random.randint(min_length, max_length)):
        letter = chr([A_ascii, C_ascii, G_ascii, T_ascii][random.randint(0, 3)])
        s.append(letter)
    return ''.join(s)
```

Benchmarking Memory Usage

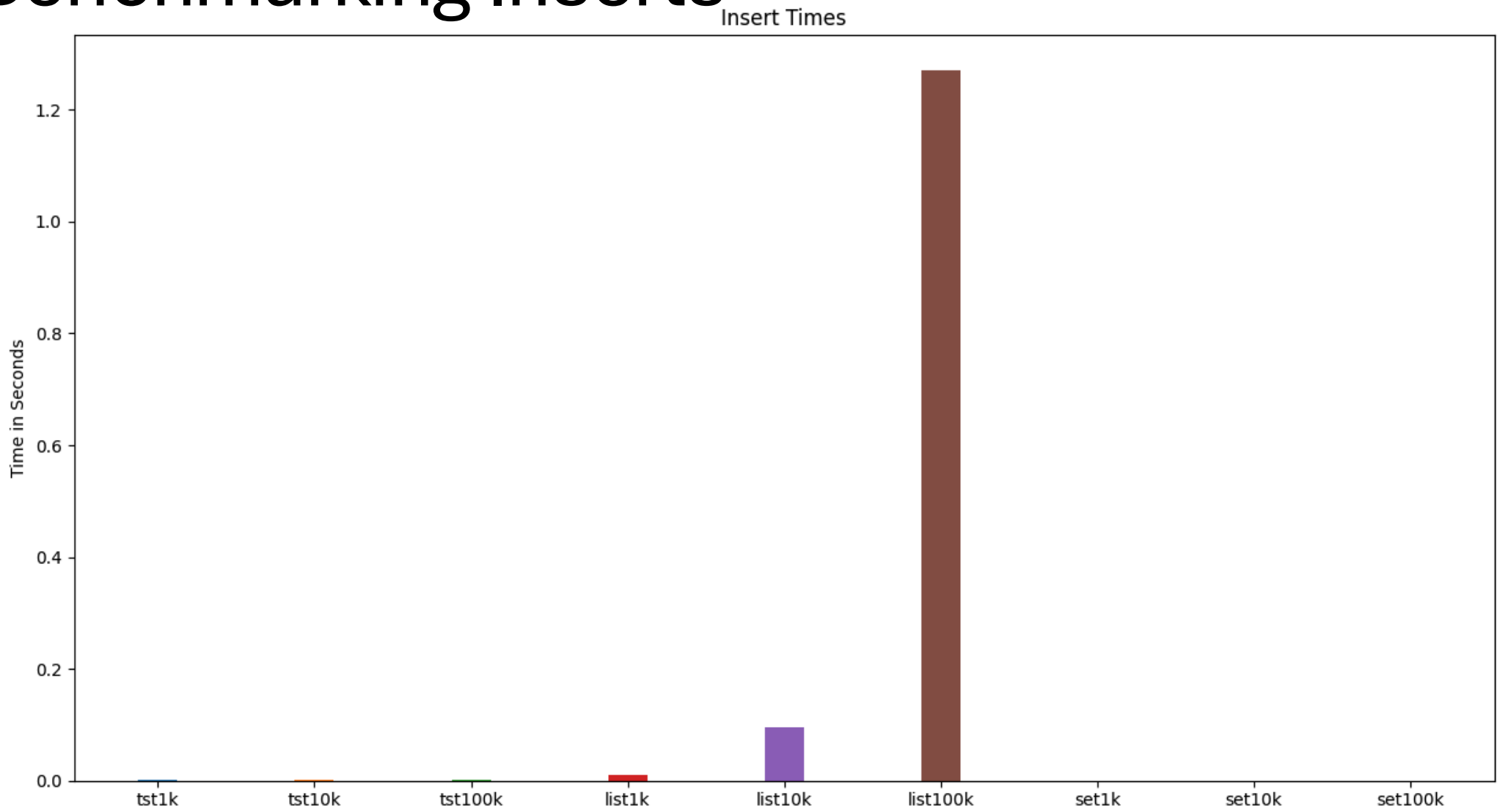


List vs TST

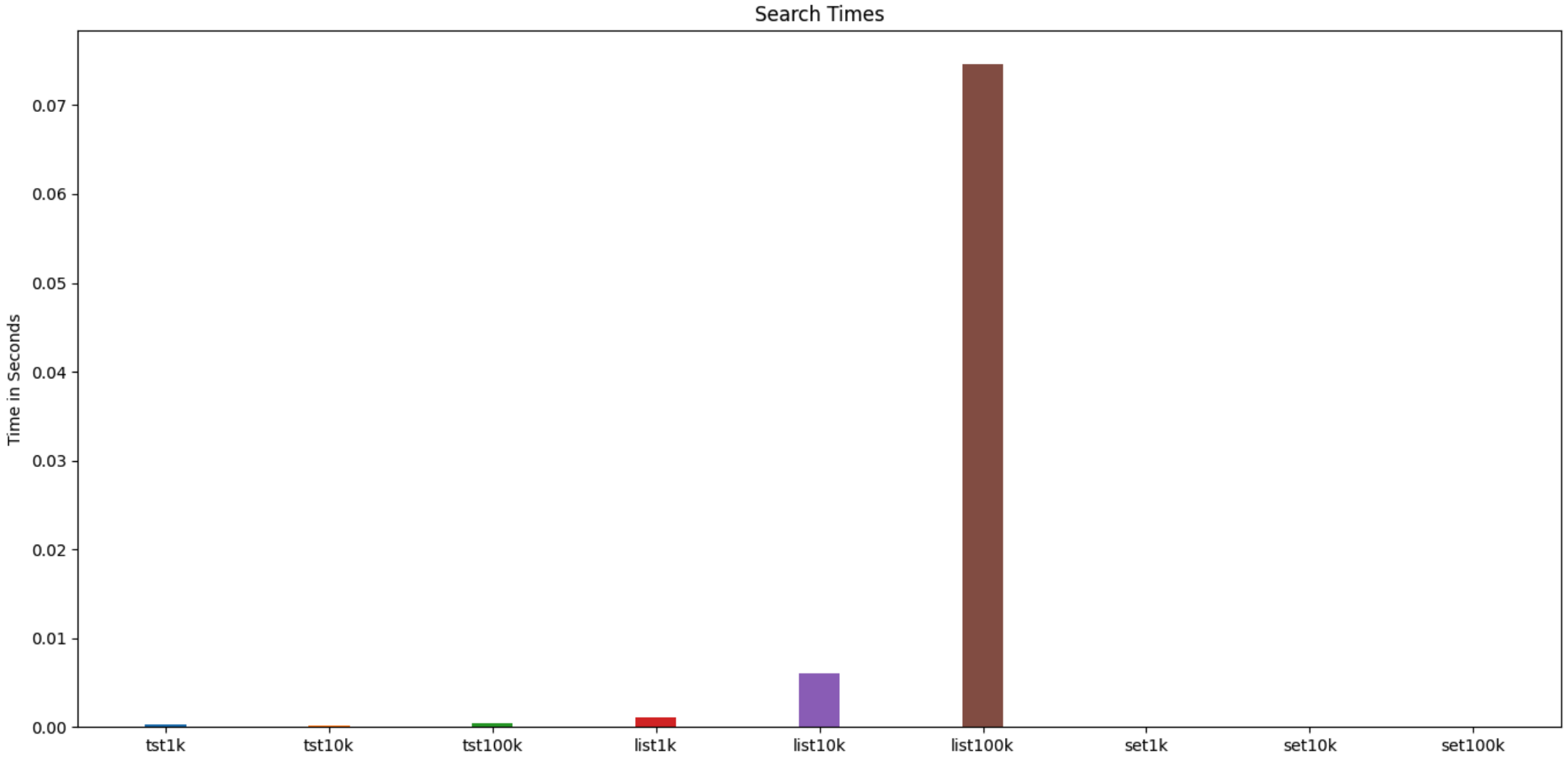
List Characters vs TST Nodes

- given the fake dataset, found about a 5:1 difference
- but this is offset by the fact that the TST nodes contain 5x more info than a character
  - each node will contain its character and val, then pointers to children

Benchmarking Inserts



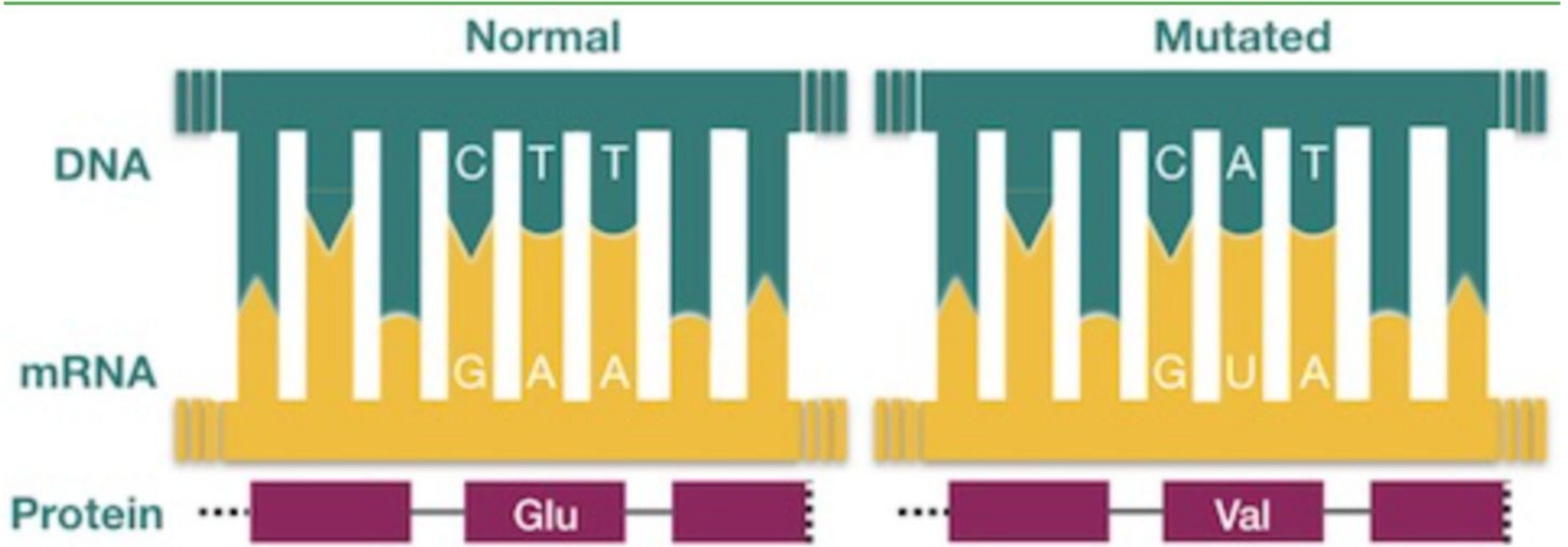
Benchmarking Search



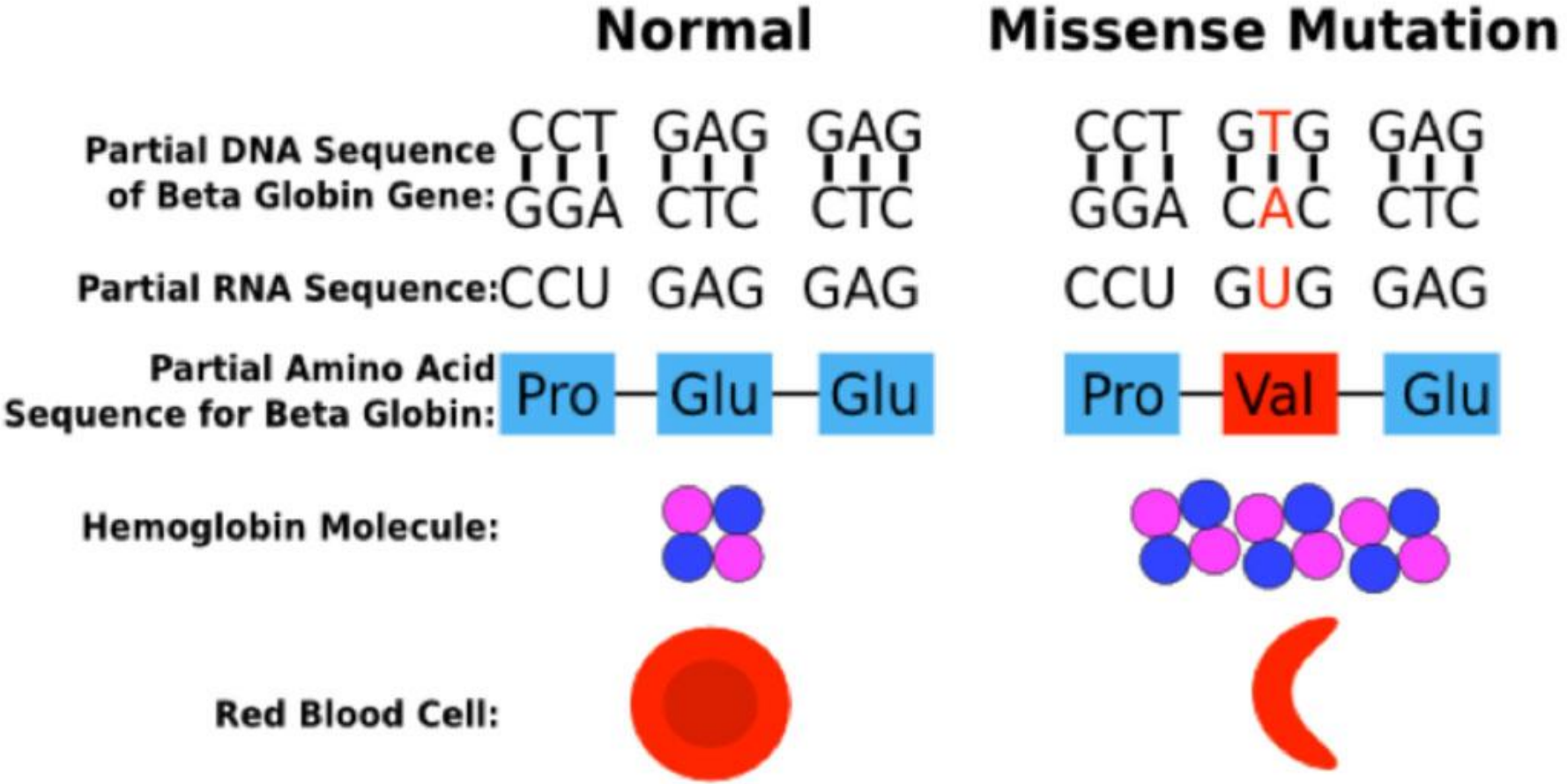
Possible Improvements

- Self balancing
- Wild card
- Nodes with multiple values (SNP)
- Recursive vs Iterative

Source:
https://bioinformaticshome.com/bioinformatics_tutorials/sequence_alignment/pair-wise_sequence_alignment.html



https://medcell.org/tbl/microangiopathy_sickle_cell/reading.php



HBB_region.fasta

brute_force.ipynb

```python
def load_fasta(filepath):
    """
    Load a FASTA file and return the DNA sequence as a single continuous string.
    Header lines starting with '>' are skipped.
    """
    sequence = []
    with open(filepath, "r") as f:
        for line in f:
            line = line.strip()
            if not line.startswith(">"):
                sequence.append(line)
    return "".join(sequence)
```

https://www.ncbi.nlm.nih.gov/nuccore/NG_000007.3

```
                     /db_xref="MIM:141900"
     mRNA            join(70545..70686,70817..71039,71890..72152)
                     /gene="HBB"
                     /gene_synonym="beta-globin; CD113t-C; ECYT6"
                     /product="hemoglobin subunit beta"
                     /transcript_id="NM_000518.5"
                     /db_xref="GeneID:3043"
                     /db_xref="HGNC:HGNC:4827"
                     /db_xref="MIM:141900"
     CDS             join(70595..70686,70817..71039,71890..72018)
                     /gene="HBB"
                     /gene_synonym="beta-globin; CD113t-C; ECYT6"
                     /note="beta globin chain; hemoglobin, beta; hemoglobin
                     beta subunit; Hb Monza protein"
                     /codon_start=1
                     /product="hemoglobin subunit beta"
                     /protein_id="NP_000509.1"
                     /db_xref="CCDS:CCDS7753.1"
                     /db_xref="GeneID:3043"
                     /db_xref="HGNC:HGNC:4827"
                     /db_xref="MIM:141900"
```

brute_force.ipynb

```python
def make_cDNA(genome):
    """
    Extract the three exons from HBB genomic DNA and concatenate them to create cDNA.
    Exon coordinates are based on NG_000007.3 RefSeqGene reference from NCBI.
    Reference URL: https://www.ncbi.nlm.nih.gov/nuccore/NG_000007.3
    The coordinates are converted from global genomic positions to local coordinates
    relative to our HBB_region.fasta slice (offset: 70,545).
    Args:
        genome (str): Genomic DNA sequence from HBB_region.fasta
    Returns:
        str: cDNA sequence (386 bp) with exons concatenated and introns removed
    """
    offset = 70545 # Genomic start position of the HBB_region.fasta slice (example)

    # The genomic coordinates on NG_000007.3 are converted to local slice coordinates by substracting the defined offset.
    exons = [
        (70573 - offset, 70714 - offset), # exon 1: convert the global genomic coordinates into the local slice coordinates
        (70955 - offset, 71083 - offset), # exon 2: convert the global genomic coordinates into the local slice coordinates
        (71144 - offset, 71261 - offset), # exon 3: convert the global genomic coordinates into the local slice coordinates
    ]

    print("=== Extracting Exons ===")
    cDNA = ""
    for idx, (start, end) in enumerate(exons, 1):
        exon_seq = genome[start:end]
        print(f"Exon {idx}: positions {start} ~ {end}")
        print(f"Sequence ({len(exon_seq)} bp): {exon_seq[:50]}...\n")  # print firtst 50 sequence
        cDNA += exon_seq

    print("=== Done ===")
    print(f"Final cDNA length: {len(cDNA)} bp")

    return cDNA
```

brute_force.ipynb

```
def trim_from_first_ATG(cDNA):
    """
    Return coding sequence starting from the first ATG in ther cDNA.
    If no ATG is found, return the original cDNA.
    """
    start_index = cDNA.find("ATG")
    if start_index != -1:
        return cDNA[start_index:]
    return cDNA
```

brute_force.ipynb

```python
class BruteForce:

    """
    This is a simple brute-force substring search for DNA sequences.
    Checks every starting position in the text.
    Returns all match positions
    """
    def __init__(self, genome, pattern):
        self.genome = genome     # FASTA or TXT file
        self.pattern = pattern # the codon

    def search(self):
        M = len(self.pattern)     # M = codon
        N = len(self.genome)      # N = Genome
        matches = []

        #Check every starting position
        for i in range(N - M + 1):
            j = 0

            # Compare the characters one by one
            while j < M and self.genome[i + j] == self.pattern[j]:
                j += 1

            #If we matched the full pattern, record the index
            if j == M:
                matches.append(i)
        return matches
    #Returns true if pattern is appears at least 1 time
    def contains(self, pattern, text):
        return len(self.search(pattern)) > 0
```

brute_force.ipynb

```python
if __name__ == "__main__":
    genome = load_fasta("HBB_region.fasta")
    # genome = load_fasta("HBB_gene.txt")
    cDNA = make_cDNA(genome)
    coding_seq = trim_from_first_ATG(cDNA)
    pattern = "GTG"   # sickle-cell mutation codon


    bf = BruteForce(coding_seq, pattern)
    result = bf.search()

    print("cDNA length (after ATG trim):", len(coding_seq))
    print("Found matches at positions (in coding sequence):", result)
```

```
=== Extracting Exons ===
Exon 1: positions 28 ~ 169
Sequence (141 bp): TGTAACAGAATAAAAAATCAATTATGTATTCAAGTTGCTAGTGTCTTAAG...

Exon 2: positions 410 ~ 538
Sequence (128 bp): CAAGGGATGGATGAAGGCAGGTGACTCTAACAGAAAGGGAAAGGATGTTG...

Exon 3: positions 599 ~ 716
Sequence (117 bp): ACTTTGAGTTTGTAAGTATATACTTCTCTGTAATGTGTCTGAATATCTCT...

=== Done ===
Final cDNA length: 386 bp
cDNA length (after ATG trim): 363
Found matches at positions (in coding sequence): [17, 138, 280, 328]
```

tst_notebook.ipynb

```python
if __name__ == "__main__":
    genome = load_fasta("HBB_region.fasta")
    cDNA = make_cDNA(genome)
    coding_seq = trim_from_first_ATG(cDNA)
    pattern = "GTG" # sickle-cell mutation codon

    tst= TST()

    # Enter every 3-mer codon into the TST (but in this case it is only "GTG")
    for i in range(len(coding_seq) - 2):
        codon = coding_seq[i:i+3]
        tst.put(codon, i)

    # Find the location of pattern
    if tst.contains(pattern):
        print("GTG found at position:", tst.get(pattern))
    else:
        print("GTG not found")
```

```
=== Extracting Exons ===
Exon 1: positions 28 ~ 169
Sequence (141 bp): TGTAACAGAATAAAAAATCAATTATGTATTCAAGTTGCTAGTGTCTTAAG...

Exon 2: positions 410 ~ 538
Sequence (128 bp): CAAGGGATGGATGAAGGCAGGTGACTCTAACAGAAAGGGAAAGGATGTTG...

Exon 3: positions 599 ~ 716
Sequence (117 bp): ACTTTGAGTTTGTAAGTATATACTTCTCTGTAATGTGTCTGAATATCTCT...

=== Done ===
Final cDNA length: 386 bp
GTG found at position: [17, 138, 280, 328]
```

Why is `17` important

```
Final cDNA length: 386 bp
cDNA length (after ATG trim): 363
Found matches at positions (in coding sequence): [17, 138, 280, 328]
```

pneumococcal vaccine should be administered in all children with SC disease. The routine use of prophylactic penicillin therapy in infants and children with SC disease remained controversial. The mutation in codon 6 of HBB in HbS is GAG (glu) to GTG (val); the mutation in HbC is GAG (glu) to AAG (lys). See also 141900.0039 and 141900.0040.

Source: https://www.omim.org/entry/141900

Brute Force Time and Space Complexity

Time Complexity: **O(P × N × M)**

Let:

- P = number of patterns (codons) we want to test
- N = gene length
- M = pattern length

Then brute-force must do ON x M) work for every pattern, resulting in: **O(P × N × M)**

Space Complexity: O(1)

Brute-force matching uses constant extra memory. It only keeps a couple of index variables and does not store any additional data structures. Thus the space complexity stays: O(1)

TST Time and Space Complexity

Time Complexity:

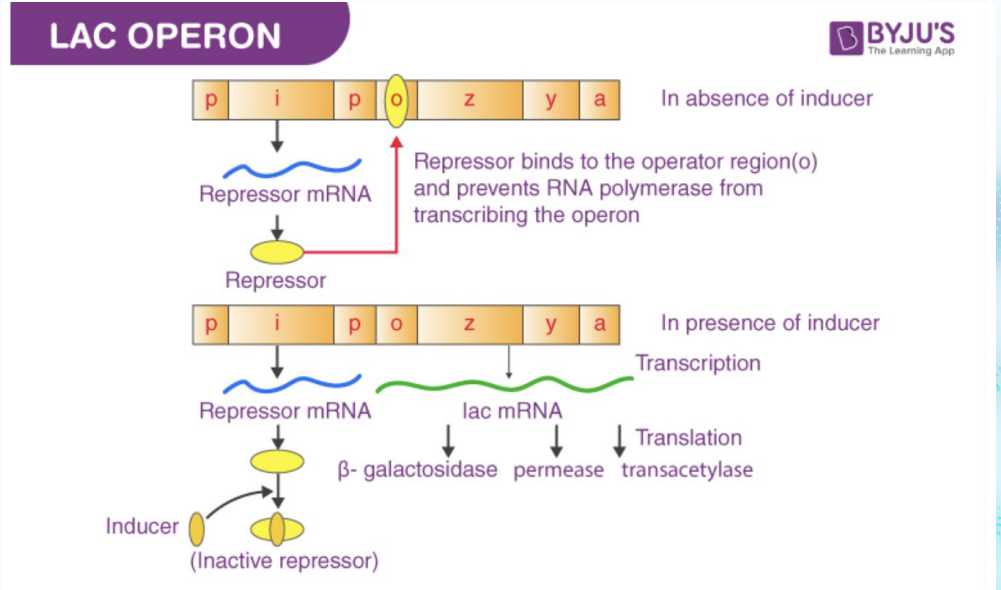
O(P × M)

Space Complexity:

O(M)

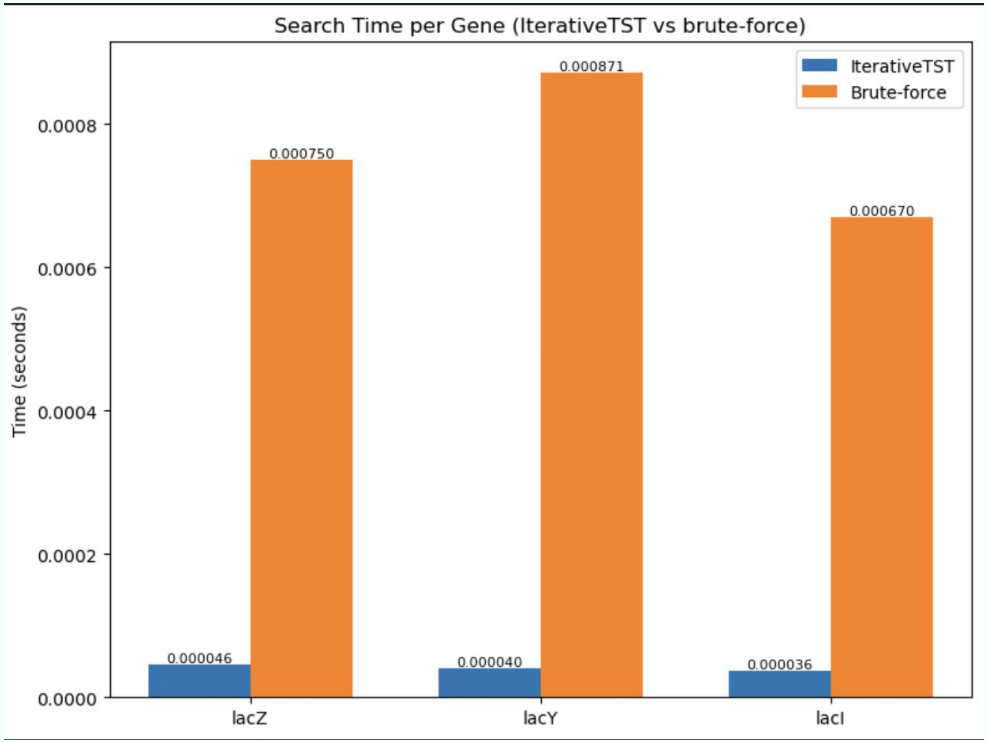
E. coli Testing

- For the *E. coli* tst test we used the test_ecoli_tst.py file
- For the *E. coli* brute force testing we used brute_force.ipynb file
- We tested lacZ, lacY, and lacI gene sequences against the full *E. coli* genome.



BYJU'S. (n.d.). *Lac operon diagram.* BYJU'S Learning App. https://byjus.com/bi

E. coli Results: TST vs Brute Force



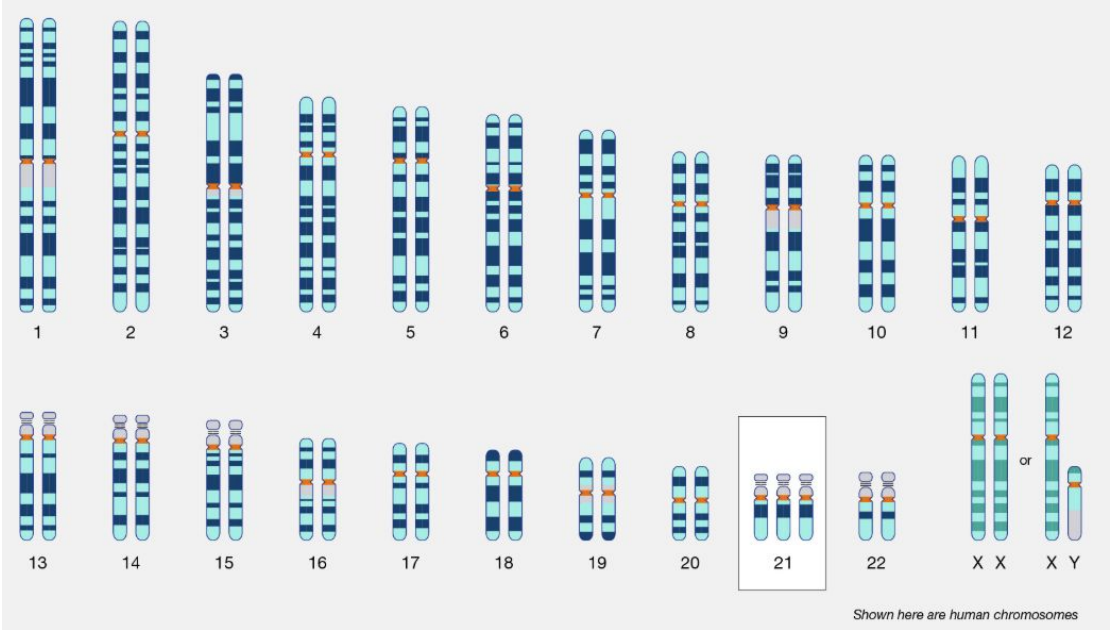
Graph generated from `lacZ_graphs.py` timing results.

Why is TST data structure better than Brute Force

- TST lookup: O(P x M) → only depends on pattern length
- Brute-Force: O(P x N x M) → rescans whole gene each time
- *E. coli*: N is big → brute-force slows down dramatically
- In our results TST was 15-20x faster than brute-force

KCNE1 and KCNE2 in Human Chromosome 21

- Human chromosome 21 has around 48 million base pairs.
- *KCNE* genes code for transmembrane ion transporter proteins
- Ions gradients and flows are essential for many cellular functions (maintaining cell volume, electrical signaling, hormone secretion etc.)
- *KCNE1*
  - Potassium ion channels
- *KCNE2*
  - Iodide ion channels



Source: https://www.genome.gov/genetics-glossary/Down-Syndrome-Trisomy-21

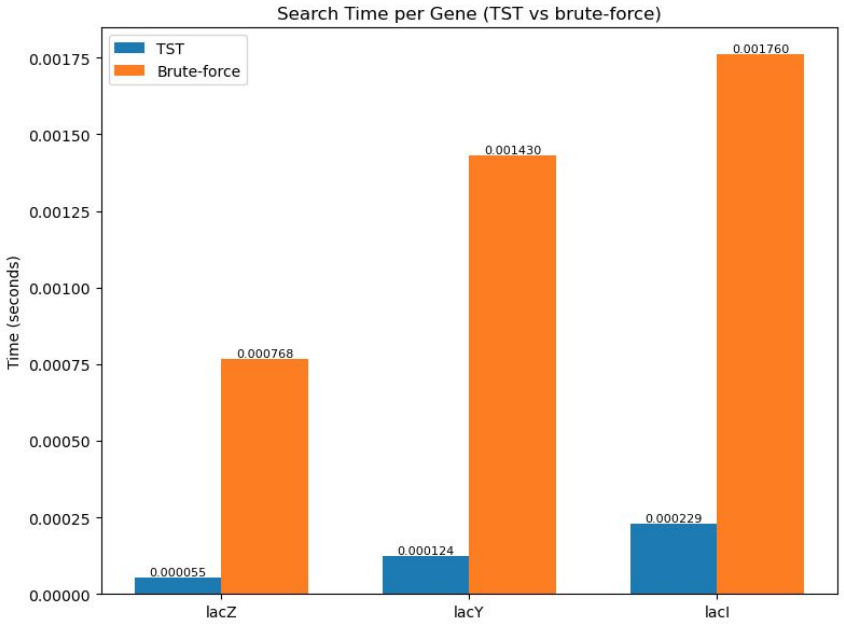
Human Chromosome 21 and TSTs (Recursive and Iterative)

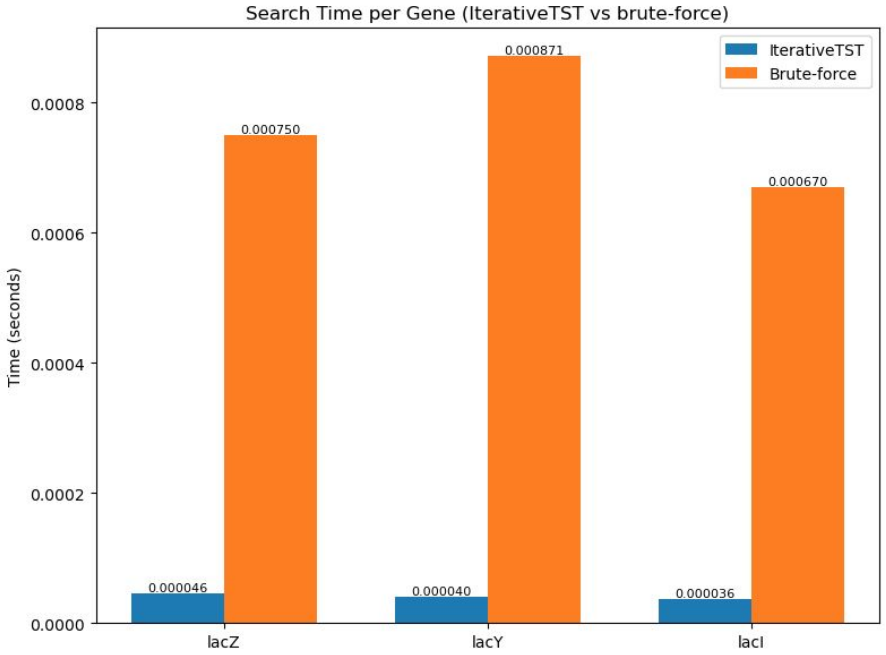
- **Recursive TST** : `RecursionError` appears
  - Calls itself per character when inserting.
  - Python has a recursion depth limited to 1000
  - Error appears when inserting a long sequence
- **Iterative TST:**
  - Uses “while” loops
  - Eliminates recursion

```python
class IterativeTST:
    """
    Iterative implementation of a Ternary Search Trie.
    Eliminates recursion, this has the same behavior but more memory efficient.
    """
    def __init__(self):
        self.root = None

    def put(self, key, value):
        # Insert key but not using recursion
        if not key:
            return
        # Initilize root if empty
        if self.root is None:
            self.root = TSTNode(key[0])
        node = self.root
        index = 0
        # Loop until full key is inserted
        while True:
            c = key[index]
            #Navigate left
            if c < node.character:
                if node.left is None:
                    node.left = TSTNode(c)
                node = node.left
                #Navigate right
            elif c > node.character:
                if node.right is None:
                    node.right = TSTNode(c)
                node = node.right
            else:  # c == node.char
                index += 1
                if index == len(key):
                    node.val = value
                    return
                if node.middle is None:
                    node.middle = TSTNode(key[index])
                node = node.middle
```

```python
def get(self, key):
    """
    Iterative search implementation.
    Returns stored value or None if key not found.
    """
    if not key or self.root is None:
        return None
    node = self.root
    index = 0
    while node:
        c = key[index]
        if c < node.character:
            node = node.left
        elif c > node.character:
            node = node.right
        else:
            index += 1
            if index == len(key):
                return node.val
            node = node.middle
    #If traversal ends with no match
    return None

def contains(self, key):
    return self.get(key) is not None
```

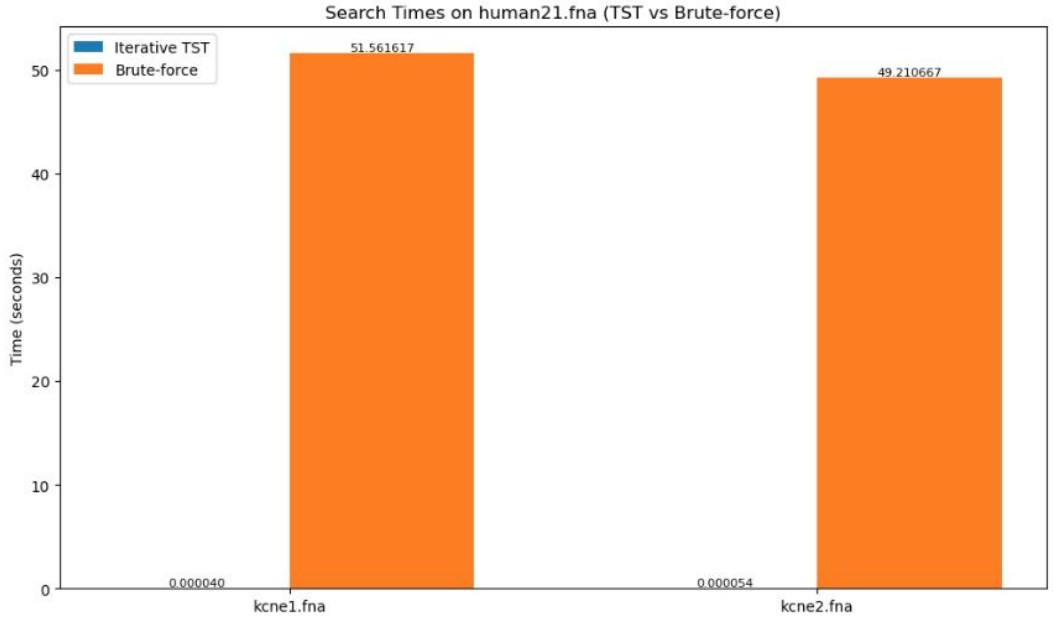
E.coli Results: Recursive vs Iterative TST





Recursive TST and Iterative TST have comparable runtime

KCNE1 & 2 Results: Iterative TST vs Brute Force



Iterative TST bars are too small to be visible in the graph.

Dataset	TST Time	Brute-force Time	Speedup
KCNE1 (Human)	0.000040 s	51.56 s	**>1,000,000×**
KCNE2 (Human)	0.000054 s	49.21 s	**>900,000×**

Advantages and Limitations

Recursive TST

- Advantage: Recursion is readable and intuitive.
- Limitation: Large genomes cannot be handled because Python has a recursion depth limited to 1000.
    - `RecursionError`
- Recursive TST is best for small and medium-sized genome datasets.

Iterative TST

- Advantage: Does not have Python's stack limitation because no recursion is used.
    - Iterative TST can handle large genomic datasets
- Iterative TST has complicated traversal and node comparisons design.

Comparison: Call stack differences of recursive TST and iterative TST

- **Recursive:**
    - Auxiliary space complexity O(M) where M is the length of pattern
    - Uses recursion
- **Iterative:**
    - Auxiliary space complexity O(1)
    - Uses while loop