DNA Sequencing Project Report

Julian Felipe Donoso Hurtado
Code: 20222020203
Systems Engineering
Francisco Jose de Caldas District University
Bogota, Colombia
jfdonosoh@udistrital.edu.co

***Abstract*—The following report provides an analysis of the ADN-Sequences project, which aims to generate theoretical DNA sequences for pattern detection. The system creates DNA sequences using specific values provided by the user to search for randomness and detect patterns in the generated strands. The report includes systemic analysis, complexity analysis, chaos analysis, results, discussion, and conclusions.**

I. System Description

The system aims to generate random DNA sequences and then detect them, using the Java programming language. Below is a detailed analysis of the system's key functionalities, starting with an understanding that it consists of the **`Sequences`** class and the **`main`** method in the main class.

A. Initialization

- The Sequences class is initialized with the following parameters: the number of sequences (**`nSeqs`**), the minimum and maximum length of the sequences (**`minLen`**, **`maxLen`**), probabilities for each base (**`probs`**), and the length of the pattern to detect (**`patLen`**).
- An array of sequences is created and populated with randomly generated sequences.

```
public Sequences(int nSeqs, int minLen, int maxLen, double[] probs, int patLen) {
    this.patCount = new HashMap<>();
    this.probs = probs;
    this.seqs = new String[nSeqs];

    for (int i = 0; i < nSeqs; i++) {
        int seqLen = (int) (Math.random() * (maxLen - minLen + 1)) + minLen;
        this.seqs[i] = genSeq(seqLen);
    }

    detectPat(patLen);
}
```

B. Sequence Generation

- The genSeq method creates DNA sequences based on the given probabilities for bases A, C, G, and T.

```
public Sequences(int nSeqs, int minLen, int maxLen, double[] probs, int patLen) {
    this.patCount = new HashMap<>();
    this.probs = probs;
    this.seqs = new String[nSeqs];

    for (int i = 0; i < nSeqs; i++) {
        int seqLen = (int) (Math.random() * (maxLen - minLen + 1)) + minLen;
        this.seqs[i] = genSeq(seqLen);
    }

    detectPat(patLen);
}
```

C. Pattern Detection

- The **`detectPat`** method searches for patterns of length **`patLen`** in each sequence and counts their occurrences.

```
private void detectPat(int patLen) {
    for (String seq : seqs) {
        for (int i = 0; i <= seq.length() - patLen; i++) {
            String pat = seq.substring(i, i + patLen);
            patCount.put(pat, patCount.getOrDefault(pat, defaultValue:0) + 1);
        }
    }
}
```

II. Complexity Analysis

A. Sequence Generation

The **`genSeq`** method generates a DNA sequence of length len using probabilities for each base (A, C, G, T). The time complexity of this method is O(**`len`**), as the time required is directly proportional to the length of the sequence. Since this process is repeated for **`nSeqs`** sequences, the total complexity for generating all sequences is O(**`nSeqs`** * **`maxLen`**). Here, **`maxLen`** represents the maximum length of the sequences generated. In summary, if you have many sequences or very long sequences, the total time to generate all sequences increases proportionally.

B. Pattern Detection

The **`detectPat`** method searches for patterns of length **`patLen`** in each generated sequence. For each sequence, the method examines each possible **`substring`** of length **`patLen`** to count how many times each pattern appears. This means that the time complexity is O(**`nSeqs`** * **`maxLen`** * **`patLen`**). The reason is that, for each of the **`nSeqs`** sequences, every **`substring`** of length **`patLen`** is checked, and the number of **`substrings`** to review is proportional to the length of the sequence. Therefore, the computational load is high when sequences are long and when the pattern to search is large.

C. Sequence Storage

The space needed to store all generated sequences is O(**`nSeqs`** * **`maxLen`**). This is because each sequence has a maximum length of **`maxLen`**, and there are **`nSeqs`** sequences in total. Therefore, the space required for storing the sequences grows linearly with the number of sequences and the length of each one.

D. Pattern Counting

The space used by the **HashMap** to store pattern counts is $O(n^k)$, where n is the number of distinct patterns that can be formed and k is the pattern length. The reason is that the **HashMap** needs space for each distinct pattern found in the sequences. The total number of possible patterns grows exponentially with the length of the pattern k. Therefore, if the pattern is long or there are many distinct patterns, the required space can be very large.

III. Chaos Analysis

The code demonstrates chaotic behavior through the following key phenomena:

A. Domino Effect

- Probability Adjustment and Sequence Generation (**genSeq** method)
- Explanation: Small changes in nucleotide probabilities (**probs** array) can lead to significant changes in the generated sequences. For instance, if the probability of adenine (**probs[0]**) is slightly increased, the generated sequences from the **genSeq(int len)** method will reflect this change dramatically. This phenomenon illustrates the domino effect, where a minor adjustment in input parameters results in substantial variations in the output sequences.

B. Snowball Effect

- Code Section: Pattern Detection (**detectPat** method)
- Explanation: After sequences are generated by **genSeq**, the **detectPat(int patLen)** method counts occurrences of each pattern of a specified length. Patterns that appear frequently, such as "ATG," will accumulate high counts quickly. This rapid accumulation affects the overall pattern distribution, demonstrating the snowball effect, where initial high occurrences of certain patterns lead to their dominance in the final pattern counts.

C. Butterfly Effect

- Code Section: Probability Normalization and Pattern Detection (main method)
- Explanation: The sensitivity to initial conditions is evident when minor changes in nucleotide probabilities or sequence lengths, as set in the main method, lead to dramatically different pattern counts. For example, adjusting the probability for guanine **(probs[2])** even slightly can result in a completely different set of frequently occurring patterns. This showcases the butterfly effect, where small variations in initial inputs result in large, unpredictable differences in outcomes.

IV. Results

Sequence Generation: The program allows for the creation of a desired number of DNA sequences (**nSeqs**). The length of each sequence is chosen randomly within a user-defined range, between a minimum (**minLen**) and a maximum (**maxLen**). For example, if **minLen** is 10 and **maxLen** is 20, each sequence will have a length randomly chosen between these two values.

```
int seqLen = (int) (Math.random() * (maxLen - minLen + 1)) + minLen;
```

This code ensures that generated sequences have varied lengths, reflecting the diversity in DNA sequence sizes and simulating randomness in the data.

Pattern Detection: After generating the sequences, the program searches for specific nucleotide patterns of a given length (**patLen**) entered by the user. It scans each sequence and counts how many times each possible pattern of the specified length appears.

```
private void detectPat(int patLen) {
    for (String seq : seqs) {
        for (int i = 0; i <= seq.length() - patLen; i++) {
            String pat = seq.substring(i, i + patLen);
            patCount.put(pat, patCount.getOrDefault(pat, defaultValue:0) + 1);
        }
    }
}
```

At this stage, the code extracts all possible substrings of the pattern length and counts their frequency in the generated sequences. Results are stored in a **HashMap** where each pattern is a key and its occurrence count is the value.

V. Example Result

When running the program with the following parameters:

```
=== Sequence Setup ===
Enter the number of sequences to generate:100
Enter the minimum length of these sequences: 1
0
Now enter the maximum length: 20

Enter the probabilities for each base (should
sum to 1 or less):
A:    0,25
C:    0,50
G:    0,30
T:    0,95

Warning! Probabilities do not sum to 1. Normal
izing.
Enter the pattern length to find: 3
```

Detected patterns and their frequencies might be:

```
=== Pattern Count ===
Pattern: ATT        | Occurrences:   44
Pattern: CTT        | Occurrences:   63
Pattern: TAT        | Occurrences:   55
Pattern: GTT        | Occurrences:   45
Pattern: AAA        | Occurrences:    2
Pattern: CAA        | Occurrences:    4
Pattern: AAC        | Occurrences:    6
Pattern: CAC        | Occurrences:   15
Pattern: GAA        | Occurrences:    3
Pattern: AAG        | Occurrences:    5
Pattern: CAG        | Occurrences:    4
Pattern: GAC        | Occurrences:    6
Pattern: GAG        | Occurrences:    5
Pattern: TGA        | Occurrences:    8
Pattern: TGC        | Occurrences:   21
```

These show us the patterns and frequency of their appreciation. It is worth noting that if the user were to provide the same initial values, the patterns and frequency would still change.

VI. Discussion

The program offers an effective tool for generating and analyzing DNA sequences. Key observations include:

- Probability Normalization: Probabilities are adjusted if their sum exceeds 1. This ensures probabilities are within the valid range, which is crucial for generating accurate sequences.
- Pattern Length Impact: The length of the pattern (**`patLen`**) affects its frequency. Shorter patterns tend to appear more often compared to longer patterns.
- Randomness and Reproducibility: Random sequence generation may vary between runs, but the general pattern distribution should be consistent with the given base probabilities.
- Scalability: The program performs well with a moderate number of sequences. For larger datasets, optimization may be necessary to maintain performance.

VII. Conclusions

Generation and Analysis of Random Sequences: The system effectively generates random DNA sequences and analyzes patterns within them. Using **`genSeq`**, sequences are created based on assigned probabilities for each nucleotide, and **`genSeq`** identifies and counts specific patterns. This provides a good simulation of real biological variability.

Impact of Pattern Length: The length of the pattern (**`patLen`**) affects how often it appears. Shorter patterns are more common than longer ones, reflecting how simpler patterns can be more frequent in DNA sequences.

Computational Complexity: Generating sequences has a complexity of O**`(nSeqs * maxLen)`**, while pattern detection is more costly with a complexity of O**`(nSeqs * maxLen * patLen)`**. Additionally, memory requirements for storing patterns grow exponentially with pattern length and the number of distinct patterns.

Chaotic Effects:

- Domino Effect: Small changes in nucleotide probabilities can lead to large variations in generated sequences.
- Snowball Effect: Frequent patterns can significantly influence the overall pattern distribution.
- Butterfly Effect: Minor changes in probabilities or sequence lengths can result in vastly different patterns.

Scalability and Performance: The program works well with a moderate number of sequences and lengths. For larger datasets, performance optimization may be needed to manage the complexity of pattern detection and storage.

Probability Normalization: Normalizing probabilities ensures they do not exceed 1, which is crucial for accurate sequence generation and avoids errors in simulating biological randomness.

Sequence Diversity: Generating sequences with varied lengths and probabilities reflects the diversity found in real DNA sequences