Relatório – Trabalho 1

Algoritmos e Estrutura de Dados II

Augusto Baldino, Larissa Laier, Vinicius Petersen

Escola Politécnica — PUCRS

18 de agosto de 2023

Resumo:

Cientistas descobriram como funciona o DNA de alienígenas a partir de estudos em um disco voador que cai na Terra. Apesar de eles terem escapado em uma cápsula de resgate, os seus DNAs estavam presentes no disco, possibilitando o estudo. Assim, foi-se descoberto que o DNA alienígena é composto por três bases: D, N e A, e que ao longo do tempo é capaz de sofrer mutações. Essas mutações ocorrem quando há a junção de duas bases diferentes na sequência genética, as quais se transformam em uma terceira base, reduzindo a cadeia do DNA.

A fusão dessas bases se inicia sempre pelo lado esquerdo, e ao juntá-las a base única resultante será aquela não presente na fusão. A nova base se encontrará no final da cadeia genética. A questão que os cientistas buscam resolver é conseguir chegar no menor tamanho possível que uma cadeia de DNA pode chegar após a execução de todas as mutações.

Introdução:

Ao cair um disco alienígena na Terra, cientistas investigam o funcionamento do DNA da espécie extraterrestre. A cadeia, formada pelas bases D, N e A, sofre mutações ao longo do tempo, reduzindo o seu tamanho. Os cientistas estão buscando uma forma de descobrir como ficaria o DNA após passar por todas as mutações possíveis.

Ao longo do estudo, descobriram que as mutações ocorrem de tal modo:

1. Iniciam sempre da esquerda para a direita;

2. Selecionam a primeira combinação de duas bases diferentes, as transformando em uma terceira base, sendo ela a base não existente na combinação determinada;
3. A base resultante vai para o final da sequência;
4. O processo se repete até o DNA estar no seu menor tamanho possível.

Para resolver o problema dos cientistas, foi-se desenvolvido um algoritmo capaz de reduzir o DNA a partir das regras pré-estabelecidas. Foram analisados oito possíveis casos de teste para a pesquisa, visando a eficiência e agilidade do processo de redução da cadeia genética. Após cada caso de teste é apresentada as suas respectivas conclusões.

Primeira solução:

Inicialmente lemos o arquivo de texto utilizando o BufferedReader. Desenvolvemos um método capaz de ler uma String que armazena o caso, usando substrings e condições if para analisar o char de uma determinada posição. A partir desse método, não obtivemos sucesso na resolução dos casos propostos, mas usamos o código como base para desenvolvermos o algoritmo correto.

A partir desse código desenvolvido na tentativa de solucionar o problema proposto, notamos erros de lógica. Ao realizar a mutação de um par de bases diferentes, a lógica faz o processo de mutações seguir em modo sequencial, sem verificar novamente as sequências resultantes de mutações anteriores. Sendo assim, o algoritmo realiza as substituições das bases como deveria, porém não consegue reduzir o tamanho da cadeia até o menor possível.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class DNAMutation {
    public static void main(String[] args) {
        String inputFileName = "Casos//caso1.txt"; // Nome do arquivo de
entrada
        try {
            BufferedReader br = new BufferedReader(new
FileReader(inputFileName));
            String dnaSequence;
```

```
            while ((dnaSequence = br.readLine()) != null) {
                String mutatedDNA = processMutations(dnaSequence);
                System.out.println("Cadeia final: " + mutatedDNA);
            }
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

        public static String processMutations(String dna) {
        int i = 0;
        while (i < dna.length()) {
            if (i < dna.length() - 1 &&
                ((dna.charAt(i) == 'D' && dna.charAt(i + 1) == 'N') ||
                (dna.charAt(i) == 'N' && dna.charAt(i + 1) == 'D'))) {
                dna = dna.substring(0, i) + dna.substring(i + 2) + "A";
                i = 0;
            } else if (i < dna.length() - 1 &&
                       ((dna.charAt(i) == 'A' && dna.charAt(i + 1) == 'N') ||
                       (dna.charAt(i) == 'N' && dna.charAt(i + 1) == 'A'))) {
                dna = dna.substring(0, i) + dna.substring(i + 2) + "D";
                i = 0;
            } else if (i < dna.length() - 1 &&
                       ((dna.charAt(i) == 'A' && dna.charAt(i + 1) == 'D') ||
                       (dna.charAt(i) == 'D' && dna.charAt(i + 1) == 'A'))) {
                dna = dna.substring(0, i) + dna.substring(i + 2) + "N";
                i = 0;
            } else if (i < dna.length() - 2 &&
                       (dna.charAt(i) == dna.charAt(i + 1) &&
                        dna.charAt(i) == dna.charAt(i + 2))) {
                i += 3;
            } else {
                i++;
            }
            System.out.println(dna);
        }
        return dna;
    }
}
```

Segunda solução:

Na tentativa se solucionar os problemas resultados da primeira solução, fizemos algumas alterações no código. O funcionamento do algoritmo já resulta nos resultados esperados, usando o método *substring* para retirar caracteres. As alterações em questão foram:

1. Realização do de mutações em um loop *while (true)* para processamento conseguirmos continuar com as mutações até o menor DNA possível ser o resultado;
2. A variável *mutated* introduzida controlar se o loop *for* deve ter continuidade ou não;
3. Uso de um *break* no *for* depois da realização da mutação para garantir que só haja uma a cada iteração do loop.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class DNAMutation3 {
    public static void main(String[] args) {
        String inputFileName = "Casos/caso7.txt"; // Nome do arquivo de
entrada
        try {
            BufferedReader br = new BufferedReader(new
FileReader(inputFileName));
            String dnaSequence;
            int caseNumber = 1;

            while ((dnaSequence = br.readLine()) != null) {
                String finalDNA = processMutations(dnaSequence);
                System.out.println("Cadeia final no caso " + caseNumber + ": "
+ finalDNA);
                caseNumber++;
            }
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static String processMutations(String dna) {
        while (true) {
            boolean mutated = false;
            for (int i = 0; i < dna.length() - 1; i++) {
```

```
                char current = dna.charAt(i);
                char next = dna.charAt(i + 1);
                if ((current == 'D' && next == 'N') || (current == 'N' && next
== 'D')) {
                    dna = dna.substring(0, i) + 'A' + dna.substring(i + 2);
                    mutated = true;
                    break;
                } else if ((current == 'A' && next == 'N') || (current == 'N'
&& next == 'A')) {
                    dna = dna.substring(0, i) + 'D' + dna.substring(i + 2);
                    mutated = true;
                    break;
                } else if ((current == 'A' && next == 'D') || (current == 'D'
&& next == 'A')) {
                    dna = dna.substring(0, i) + 'N' + dna.substring(i + 2);
                    mutated = true;
                    break;
                } else if (i < dna.length() - 2 && current == next && next ==
dna.charAt(i + 2)) {
                    dna = dna.substring(0, i) + dna.substring(i + 2);
                    mutated = true;
                    break;
                }
            }
            if (!mutated) {
                break;
            }
        }

        return dna;
    }
}
```

A solução encontrada era funcional, no entanto não eficiente para os casos 5, 6 e 7. Decidimos explorar estratégias mais eficientes para otimizar o algoritmo, a fim de conquistar um desempenho bom e consistente em todos os casos.

Terceira solução:

Para conquistar a eficiência procurada, como mencionamos anteriormente, realizamos mais algumas modificações no algoritmo, sendo elas:

1. Implementação de uma LinkedList em vez de uma String para representar o DNA, permitindo a inserção e remoção de caracteres mais eficiente nas mutações.
2. Ao invés do método *substring* utilizado anteriormente para a manipulação das sequência do DNA, usamos operações de inserção e remoção de caracteres na *LinkedList*.
3. A sequência final é construída a partir de elementos da *LinkedList* após as mutações aplicadas, usando um *StringBuilder* para criar a sequência de maneira eficiente.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;


public class DnaMutation2 {
    public static void main(String[] args) {
        String inputFileName = "Casos/caso8.txt";  // Nome do arquivo de
entrada
        try {
            BufferedReader br = new BufferedReader(new
FileReader(inputFileName));
            String dnaSequence;
            int caseNumber = 1;


            while ((dnaSequence = br.readLine()) != null) {
                Map<String, String> dnaSequences = new HashMap<>();
                dnaSequences.put("original", dnaSequence);


                // Processa as mutações e armazena a sequência final no
HashMap
                processMutations(dnaSequences);
```

```
                // Imprime a sequência final
                System.out.println("Cadeia final no caso " + caseNumber + ": "
+ dnaSequences.get("final"));
                caseNumber++;


            }


            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }


    public static void processMutations(Map<String, String> dnaSequences) {
        String dna = dnaSequences.get("original");
        boolean mutated=true;



            mutated = false;
            List<Character> newDNA = new LinkedList<>();


            for (int i = 0; i < dna.length(); i++) {
                char nucleotide = dna.charAt(i);
                newDNA.add(nucleotide);
            }



            for (int i = 0; i < newDNA .size();i++) {
                char current = newDNA.get(i);
                char next = (i < newDNA .size() - 1) ? newDNA .get(i + 1) :
'\0';


                if ((current == 'D' && next == 'N') || (current == 'N' && next
== 'D')) {
                    newDNA.set(i, 'A');
                    newDNA.remove(i+1);
                    if(i==0)i= i-1;
                    if(i>=1)i= i-2;
                    mutated = true;
                } else if ((current == 'A' && next == 'N') || (current == 'N'
&& next == 'A')) {
                    newDNA.set(i, 'D');
                    newDNA.remove(i+1);
```

```
                    if(i==0)i= i-1;
                    if(i>=1)i= i-2;
                    mutated = true;
                } else if ((current == 'A' && next == 'D') || (current == 'D'
&& next == 'A')) {
                    newDNA.set(i, 'N');
                    newDNA.remove(i+1);

                    if(i==0)i= i-1;
                    if(i>=1)i= i-2;
                    mutated = true;
                } else if (i < newDNA.size() - 2 && current == next && next ==
newDNA.get(i + 2)) {
                    mutated = true;
                }else{
                    mutated=false;
                }


            }


            // Atualiza a sequência final
            StringBuilder result = new StringBuilder(newDNA.size());
            for (Character c : newDNA) {
                result.append(c);
            }
            dna = result.toString();



        // Atualiza a sequência final no HashMap
        dnaSequences.put("final", dna);
    }
}
```

Ao terminarmos a execução do algoritmo, percebemos uma grande melhora na eficiência e agilidade dele ao compará-lo com os anteriores. A diferença mais notável entre os casos foi no caso 7, que passou de uma execução de cerca de 2 minutos para uma de apenas 5 segundos.

Resultados:

Caso 1 (com passo a passo): A

```
PS C:\Users\User\Desktop\PUCRS\4 Algoritmos e Estrutura de Dados II\T1_Alest_2-main>  c:; cd 'c:\Users\User\Desktop\PUCRS\4 Algoritmos e Estrutura de Dados II\T1_Alest_2-main'; & 'C:\Program Files\Eclipse Adoptium\jre-8.0.352.8-hotspot\bin\java.exe' '-cp' 'C:\Users\User\AppData\Roaming\Code\User\workspaceStorage\74c36d17977bb9260b40e528fed8c115\redhat.java\jdt_ws\T1_Alest_2-main_83ede378\bin' 'DNAMutation'
DDDADAAAND
DDDADAAAND
DDNDAAAND
DADAAAND
NDAAAND
AAAAND
AAAAND
AAAAND
AAAAND
AAADD
AAND
ADD
ND
A
A
Cadeia final no caso 1: A
PS C:\Users\User\Desktop\PUCRS\4 Algoritmos e Estrutura de Dados II\T1_Alest_2-main>
```

Caso 2: NN

Caso 3: AA

Caso 4: N

Caso 5: D

Caso 6: N

Caso 7: NNN

Caso 8:

```
PS C:\Users\sansv\OneDrive\Documents\vscode\projetos\T1_Alest_2-main>  c:; cd 'c:
ws\T1_Alest_2-main_2dd449c9\bin' 'DnaMutation2'
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
        at java.util.LinkedList.linkLast(Unknown Source)
        at java.util.LinkedList.add(Unknown Source)
        at DnaMutation2.processMutations(DnaMutation2.java:46)
        at DnaMutation2.main(DnaMutation2.java:22)
```

Conclusões:

A partir das tentativas de algoritmos desenvolvidos para solucionar o problema dos cientistas, conseguimos chegar a um que resolve o desafio de reduzir a sequência de DNA alienígena a sua forma reduzida, seguindo as regras pré-estabelecidas de mutações. Ainda nas tentativas iniciais, fomos capazes de desenvolver um algoritmo apropriado para a solução no problema, no entanto, ele era considerado ineficiente, por levar muito tempo na execução. Assim, decidimos otimizá-lo, usando uma LinkedList em vez de uma String para representar a sequência genética, melhorando de modo significativo o seu desempenho. Sua implementação permitiu a inserção e remoção de bases da cadeia ser muito mais rápida. A sequência final foi construída com o uso de uma StringBuilder, otimizando ainda mais o seu desempenho.

A partir da otimização desenvolvida, conseguimos obter um algoritmo eficiente e ágil com capacidade de processar as sequências de DNA dos alienígenas em um tempo mais curto, mesmo em casos mais complexos. Como resultado desse estudo, conseguimos retomar conceitos aprendidos anteriormente e desenvolver mais as habilidades de criar algoritmos capazes de resolveres problemas desafiadores.