Relatório Técnico
Trabalho 1 - Alest II

Bianca S. Alves[1], Pedro S. Medronha[2]
Escola Politécnica — PUCRS

16 de setembro de 2023

Resumo

Este relatório técnico aborda nossa jornada para resolver o desafio de determinar o tamanho da menor cadeia possível após mutações em uma sequência de DNA alienígena. Documentamos seis aplicações distintas em nossa busca por uma solução eficaz, incluindo o uso de diferentes estruturas de dados e aprimoramentos contínuos. Através dessa narrativa, destacamos as lições aprendidas sobre otimização, escolha de estruturas de dados e implementação de algoritmos, resultando em uma solução altamente eficiente para o problema, que pode ser útil para outros no campo de estruturas de dados e da informática em geral.

Introdução

Este relatório descreve o processo de resolução de um desafio científico intrigante apresentado pelos pesquisadores após a descoberta de um disco voador acidentado no Parque da Redenção. Após uma análise detalhada do DNA dos alienígenas a bordo da aeronave, foi revelado que sua estrutura genética difere do DNA terrestre, consistindo em apenas três bases: D, N e A. Além disso, os cientistas descobriram um processo de mutação único que ocorre nesse DNA alienígena, levando à sua deterioração ao longo do tempo, sendo ele: quando duas bases distintas, contíguas uma à outra, se unem, elas podem formar a terceira base, diminuindo assim o tamanho da cadeia de DNA. E isso acontece seguindo alguns passos bem organizados:

- Em uma cadeia de DNA, a fusão de bases ocorre sempre na dupla de bases distintas mais à esquerda.
- A nova base criada com a fusão será anexada ao final da cadeia de DNA.

Por exemplo, considerando a sequência de DNA, é possível concluir que ocorrerá uma deterioração em 'DN', resultando em uma cadeia menor, 'AA'.

O problema principal é determinar o tamanho da menor cadeia que pode ser obtida após todas as mutações possíveis em uma sequência de DNA alienígena. Este relatório documenta a evolução de nossos esforços na resolução desse problema complexo, que envolveu seis tentativas distintas, além da exploração de estruturas de dados diferentes e abordagens algorítmicas, todas implementadas na linguagem Java.

A seguir, apresentaremos como o problema foi modelado em cada tentativa, descreveremos os algoritmos utilizados em cada etapa, demonstraremos os resultados obtidos em casos de teste específicos e, por fim, compartilharemos nossas conclusões sobre a abordagem final adotada para resolver esse desafio único na biologia e na informática.

[1] b.alves03@edu.pucrs.br
[2] santos.pedro09@edu.pucrs.br

Primeira Solução

Após uma minuciosa análise do enunciado do desafio e uma compreensão aprofundada da lógica que deveria ser incorporada ao algoritmo, tomou-se a decisão de adotar a estrutura de dados de lista encadeada (*LinkedList* do Java). Essa decisão foi fundamentada na facilidade que essa estrutura oferece para a realização de operações essenciais do nosso algoritmo. Especificamente, destaca-se as operações de remoção no início e inserção no final da lista, as quais se mostraram indispensáveis para a implementação bem-sucedida da solução.

Partindo para a implementação do algoritmo, foi criada uma classe - inicialmente nomeada "AlgoritmoMutacoes" - responsável por realizar as mutações das cadeias de DNA que eram geradas aleatoriamente por outra classe, "GeradorCasosDeTeste", que também contém a função main responsável por executar toda a aplicação.

A classe AlgoritmoMutacoes era constituída por um método construtor, responsável por instanciar a lista encadeada e adicionar elementos na mesma e por um método principal responsável por realizar as operações de mutações nas cadeias de DNA geradas.

Esse pseudocódigo descreve a lógica do método em detalhes, incluindo como as mutações são aplicadas e como a lista é manipulada, veja abaixo.

```
1.  Função decomposicao():
2.      Para i de 0 até tamanho(lista) - 1 faça:
3.          Escolha (lista[i])  faça:
4.              Caso 'D':
5.                  Se i == tamanho(lista) - 1, então
6.                      Quebre
7.                  Se lista[i+1] == 'A', então
8.                      Remova o primeiro elemento da lista
9.                      Remova o primeiro elemento da lista
10.                     Adicione 'N' ao final da lista
11.                 Senão, se lista[i+1] == 'N', então
12.                     Remova o primeiro elemento da lista
13.                     Remova o primeiro elemento da lista
14.                     Adicione 'A' ao final da lista
15.
16.             Caso 'N':
17.                 Se i == tamanho(lista) - 1, então
18.                     Quebre
19.                 Se lista[i+1] == 'A', então
20.                     Remova o primeiro elemento da lista
21.                     Remova o primeiro elemento da lista
22.                     Adicione 'D' ao final da lista
23.                 Senão, se lista[i+1] == 'D', então
24.                     Remova o primeiro elemento da lista
25.                     Remova o primeiro elemento da lista
26.                     Adicione 'A' ao final da lista
27.
28.             Caso 'A':
29.                 Se i == tamanho(lista) - 1, então
30.                     Quebre
31.                 Se lista[i+1] == 'D', então
32.                     Remova o primeiro elemento da lista
```

33. Remova o primeiro elemento da lista
34. Adicione 'N' ao final da lista
35. **Senão**, se lista[i+1] == 'N', então
36. Remova o primeiro elemento da lista
37. Remova o primeiro elemento da lista
38. Adicione 'D' ao final da lista
39.
40. **Caso Contrário:**
41. Exceção com a mensagem "Valor inesperado: " concatenado com lista[i]
42. **Fim Para**
43. Retorne a lista como string
44. **Fim Função** decomposicao

No entanto, ao testar o desempenho do algoritmo, ficou claro que a lógica não estava corretamente implementada. Após analisar o algoritmo, alguns fatores que estavam causando o mau desempenho e os resultados incorretos se sobressaíram:

- **Manipulação incorreta da lista**: quando várias mutações ocorrem consecutivamente, a lógica de remoção e adição de bases não conseguia acompanhar corretamente, resultando em saídas incorretas. O código não lidava bem com situações em que uma mutação criava a oportunidade para outra mutação imediata;
- **Iteração:** durante a iteração da lista, os índices não eram tratados corretamente. Isso podia levar a índices fora dos limites da lista e causar exceções de índice fora do intervalo (como *IndexOutOfBoundsException*);
- **Lógica mal implementada:** a lógica responsável pelas mutações não estava considerando todas as possibilidades possíveis. Dessa forma, para entradas muito grandes, por exemplo, a saída acabava sendo errada e muito extensa também.

Após essa primeira abordagem, ficou claro a importância de uma revisão e uma reimplementação que tratasse de possíveis exceções e adequasse corretamente a lógica de mutação de bases para o contexto do código. Nos levando assim para uma nova implementação.

Segunda Solução

Focando na correção dos erros da primeira tentativa, em específico, na iteração da lista e lógica do algoritmo, procuramos uma abordagem com a utilização de dois iteradores: um *for* e um *do-while*. Revisando o enunciado do problema e realizando testes de mesa, percebemos a necessidade do algoritmo reiniciar o percorrimento da lista após a realização de mutações, para validar se haveria novas bases distintas lado a lado. Por isso, se adotou a estratégia da utilização de um *do-while*, com o auxílio de uma variável booleana, para realizar tal façanha. Quanto a estrutura *for*, ela era utilizada da mesma maneira que na primeira implementação, para percorrer cada um dos elementos da lista e realizar comparações.

Feitas as modificações mencionadas, o algoritmo ficou estruturado como é apresentado abaixo. Além das modificações na função “decomposicao”, também foi adotada a criação de uma nova função “realizaMutacao”, para comparar duas bases fornecidas e retornar uma terceira base.

```
1.  Função decomposicao():
2.      Declarar uma variável booleana mutacao
3.
4.      Faça
5.          mutacao = falso
6.          Para i de 0 até tamanho(lista) - 1 faça:
7.              baseAtual = lista[i]
8.              proxBase = lista[i + 1]
9.
10.             Se baseAtual ≠ proxBase então
11.                 novaBase = realizaMutacao(baseAtual, proxBase)
12.                 Se novaBase ≠ '0' então
13.                     Adiciona novaBase ao final da lista
14.                     Remove o elemento na posição [i]
15.                     Remove o elemento na posição [i + 1]
16.                 Fim Se
17.                 mutacao = verdadeiro
18.                 Sair do loop Para
19.             Fim Se
20.         Fim Para
21.     Enquanto mutacao for verdadeiro
22.      Retorne a lista como string
23. Fim Função decomposicao
24.
25. Função realizaMutacao(base1, base2):
26.     Se (base1 = 'D' E base2 = 'A') OU (base1 = 'A' E base2 = 'D') então
27.         Retornar 'N'
28.     Senão, se (base1 = 'D' E base2 = 'N') OU (base1 = 'N' E base2 = 'D') então
29.         Retornar 'A'
30.     Senão, se (base1 = 'A' E base2 = 'N') OU (base1 = 'N' E base2 = 'A') então
31.         Retornar 'D'
32.     Senão
33.         Retornar '0'  // Indica que não houve mutação
34.     Fim Se
35. Fim Função realizaMutacao
```

Finalmente após todas as correções, obtivemos a nossa primeira implementação com sucesso do algoritmo.

Apesar do algoritmo estar concluído, ainda acreditávamos no potencial de melhoria dele, pois na realização de testes com entradas maiores em torno de 10.000 caracteres , a performance do algoritmo decaia. Com isso, chegamos a uma terceira implementação de nosso algoritmo.

Terceira Solução

Avaliando nossa solução bem-sucedida, resolvemos testar a aplicação do nosso algoritmo com diferentes estruturas de dados, em busca de uma performance melhor. Inicialmente, procurando diminuir a complexidade de acesso aos elementos da lista, testamos a implementação do nosso algoritmo com a estrutura *ArrayList* do Java.

Antes de apresentar o pseudocódigo do nosso algoritmo, vale destacar que realizamos algumas alterações no mesmo, como a remoção de *if*s e mudanças de nomenclatura de métodos e variáveis, apenas com o intuito de deixar o código mais limpo e compreensível.

```
1.  Função mutation():
2.      Declaração de variável booleana change
3.
4.      Faça
5.          change = falso
6.          Para i de 0 até tamanho(lista) - 1 faça:
7.              current = lista[i]
8.              prox = lista[i + 1]
9.
10.             Se current ≠ prox então
11.                 novaBase = generate(current, prox)
12.                 Adiciona novaBase ao final da lista
13.                 Remove elemento na posição [i]
14.                 Remove elemento na posição [i + 1]
15.                 change = verdadeiro
16.                 Sair do loop Para
17.             Fim Se
18.         Fim Para
19.     Enquanto change for verdadeiro
20.     Retorne a lista como string
21. Fim Função mutation
22.
23. Função generate(baseOne, baseTwo):
24.     Se (baseOne = 'D' E baseTwo = 'N') OU (baseOne = 'N' E baseTwo = 'D') então
25.         Retornar 'A'
26.     Senão, se (baseOne = 'A' E baseTwo = 'D') OU (baseOne = 'D' E baseTwo = 'A') então
27.         Retornar 'N'
28.     Senão, se (baseOne = 'N' E baseTwo = 'A') OU (baseOne = 'A' E baseTwo = 'N') então
29.         Retornar 'D'
30.     Senão
31.         Retornar '#' // Caractere inválido (caso baseOne e baseTwo sejam iguais)
32.     Fim Se
33. Fim Função generate
```

Como é possível observar, o algoritmo em sua estrutura não teve modificações, apenas mudanças de nomenclatura e a alteração de *LinkedList* para *ArrayList*.

Inesperadamente, ao contrário do que acreditávamos, o algoritmo implementado com *ArrayList* teve uma queda em sua performance. Na realização de testes com entradas que consideramos pequenas para a implementação com *LinkedList*, o algoritmo se tornava muito lento implementado com *ArrayList*.

A partir dessa experiência, foi possível observar que apesar da *ArrayList* nos fornecer um ganho no acesso aos elementos, esse ganho não era significativo, levando em consideração que uma das operações imperativas do nosso algoritmo se tratava da remoção de elementos no início da estrutura.

Apesar dessa frustração inicial, ainda estávamos obstinados a melhorar o desempenho da nossa solução. Com isso, fomos para a nossa quarta implementação.

Quarta Solução

Na quarta implementação, iniciamos por uma busca das possíveis estruturas que mais se adequariam ao cenário do problema a ser resolvido, e com isso nos deparamos com a estrutura de dados *Deque* do Java. Tal estrutura trabalha bem com operações de inserção e remoção, no início e no fim da lista.

Em primeiro momento, pareceu ser a alternativa óbvia para resolução da nossa problemática, entretanto, esquecemos de avaliar o pior cenário da implementação do nosso algoritmo: remoções no meio da lista. Com isso em mente, reavaliamos as opções e decidimos aderir uma abordagem não convencional: utilizar duas estruturas de dados, ao mesmo tempo, na implementação da solução.

Com isso, chegamos a um algoritmo que utiliza uma *Deque* do Java como estrutura principal, ao qual é responsável por percorrer e acessar os elementos, e que utiliza uma *Stack* (pilha) do Java como estrutura auxiliar, para operar nos casos de remoções do meio, ou seja, quando os caracteres iniciais da entrada são repetidos. Veja abaixo o pseudocódigo do método "mutation()" com as alterações realizadas:

```
1.  Função mutation():
2.      Declaração de variáveis:
3.          i = 0
4.          j = 0
5.          stack = Pilha vazia // Estrutura auxiliar
6.          change = falso
7.
8.      Faça
9.          teste = lista.ObterIterador()
10.         change = falso
11.
12.         Enquanto teste.TemProximo() faça:
13.             Tente
14.                 // Acessa os elementos
15.                 current = teste.ObterProximo()
16.                 prox = teste.ObterProximo()
17.
18.                 Se current ≠ prox então
19.                     novaBase = generate(current, prox)
20.                     Adiciona novaBase ao final da lista
21.
22.                     // Remove bases antigas
23.                     Enquanto j <= i faça:
24.                         Se j = i então
25.                             Remove o primeiro elemento da lista
26.                             Enquanto !pilha.EstaVazia() faça:
27.                                 lista.AdicionarNoInicio(pilha.Desempilhar())
28.                             Fim Enquanto
29.                             Sair do loop Enquanto
30.                         Fim Se
31.                         pilha.Empilhar(Remove o primeiro elemento da lista)
32.                         j++
33.                     Fim Enquanto
34.                     change = verdadeiro
35.                     Sair do loop Enquanto
```

```
            Fim Se
            i++
        Capture NoSuchElementException e
            Imprima "ERRO: " + e.Mensagem()
        Fim Tente-Captura
    Fim Enquanto
  Enquanto change for verdadeiro
  Retorne a lista como string
Fim Função mutation
```

O algoritmo deveria operar do seguinte modo:

- Encontrou elementos divergentes lado a lado no início da lista?
  - Remove esses elementos
  - Adiciona a nova base ao fim da lista
- Encontrou elementos divergentes lado a lado no meio da lista?
  - Remove todos os elementos iguais até chegar aos elementos divergentes
  - Adiciona esses elementos iguais na pilha
  - Adiciona a nova base ao fim da lista
  - Devolve para o início da lista os elementos da pilha

Entretanto, como a idealização foge da prática, ao implementar o algoritmo nos deparamos com diversos furos na lógica do mesmo e com erros inesperados (exceções). Procuramos corrigir os erros, mas ao analisarmos mais a fundo o funcionamento do algoritmo, vimos que ao invés de implementar uma solução com desempenho maior, estávamos fazendo o contrário, afinal, nossa proposta incluía transitar os elementos de uma estrutura para outra e isso, por si só, já é uma operação custosa, ainda mais se levarmos em consideração entradas de tamanho significativo.

Por conta disso, resolvemos interromper essa tentativa, tendo em vista que não nos retornaria os resultados esperados, e partimos para nossa quinta implementação.

Quinta Solução

Na quinta implementação, havíamos retornado para o nosso ponto de partida, nossa solução funcional com *LinkedList* do Java. Desta vez mais cautelosos, avaliamos a possibilidade da implementação da nossa própria *LinkedList*, ou seja, trabalhar diretamente com a manipulação de referências, afinal, onde perdíamos eficiência com a *LinkedList* do Java era, justamente, nos métodos “remove()” e “get()”, que percorriam N elementos da lista até encontrarem o elemento passado como argumento. Abaixo consta o pseudocódigo da implementação da nossa *LinkedList*, a classe “DoubleLinkedListOfCharacter”.

```
Classe DoubleLinkedListOfCharacter:
  Classe Node:
    Variáveis:
      elemento: caractere
      próximo: Node
      anterior: Node
    Função construtora Node(elemento):
      this.elemento = elemento
      próximo = nulo
```

```
    anterior = nulo
  Fim Função construtora Node
Fim Classe Node

Variáveis:
  header: Node
  trailer: Node
  current: Node
  prox: Node
  contagem: inteiro

Função construtora DoubleLinkedListOfCharacter():
  header = novo Node('#')
  trailer = novo Node('#')
  header.próximo = trailer
  trailer.anterior = header
  contagem = 0
Fim Função construtora DoubleLinkedListOfCharacter

Função add(elemento):
  n = novo Node(elemento)

  Se header está conectado a trailer então // Lista está vazia, insere no início
    n.anterior = header
    n.próximo = trailer
    trailer.anterior = n
    header.próximo = n
  Senão // Se há elementos, insere ao fim
    n.próximo = trailer
    n.anterior = trailer.anterior
    trailer.anterior.próximo = n
    trailer.anterior = n
  Fim Se
  contagem++
Fim Função add

Função remove():
  Se o primeiro elemento é igual a current  então
    header.próximo = próximo.próximo
    próximo.próximo.anterior = header
  Senão, se o último elemento é igual a prox então
    trailer.anterior = atual.anterior
    atual.anterior.próximo = trailer
  Senão
    aux = atual
    próximo.próximo.anterior = aux.anterior
    aux.anterior.próximo = próximo.próximo
  Fim Se
  contagem = contagem - 2
Fim Função remove

Função next(): // Método que permite percorrer a lista
  Se prox ≠ trailer então
    prox = próximo elemento
    current = próximo elemento
```

```
      Retornar verdadeiro
    Fim Se
    Retornar falso
  Fim Função next

  Função reset():
    current = primeiro elemento da lista
    prox = segundo elemento da lista
  Fim Função reset

  Função getCurrent():
    Retorna o elemento current
  Fim Função getCurrent

  Função getProx():
    Retorna o elemento prox
  Fim Função getProx

  Função toString():
    construtorDeCadeia = novo StringBuilder()
    aux = primeiro elemento da lista

    Para i de 0 até contagem faça:
      Se i = (contagem - 1) então
        construtorDeCadeia.anexar(aux.elemento)
      Senão
        construtorDeCadeia.anexar(aux.elemento + ", ")
      Fim Se
      aux = aux.próximo
    Fim Para
    Retornar construtorDeCadeia.toString() // Retorna lista como string
  Fim Função toString
Fim Classe DoubleLinkedListOfCharacter
```

Abaixo o pseudocódigo de como ficou a implementação do algoritmo, ou seja, o método "mutation()".

```
1.  Função mutation():
2.      Declaração de variável booleana change
3.
4.      Faça
5.          Posiciona as referências no início da lista
6.          change = falso
7.
8.          Faça
9.              current = lista.getCurrent()
10.             prox = lista.getProx()
11.
12.             // Verifica senão estamos no header ou trailer
13.             Se current ≠ '#' e prox ≠ '#' então
14.                 Se current ≠ prox então
15.                     Adiciona a nova base ao final da lista
16.                     Remove as duas bases antigas
17.                     change = verdadeiro
```

18. **Sair do loop Faça**
19. **Fim Se**
20. **Fim Se**
21. **Enquanto** lista.next() // Iteração que percorre a lista
22. **Enquanto** change for verdadeiro
23. Retorne a lista como string
24. **Fim Função** mutation

Como esperado, a implementação da nossa própria *LinkedList* obteve um desempenho melhor que a *LinkedList* do Java, afinal, como foi mencionado anteriormente, os métodos "remove()" e "get()" tinham um alto custo para o nosso algoritmo, porém, agora não mais.

Entusiasmados com os resultados obtidos, decidimos ir ainda mais além. Um dos principais desafios do algoritmo, é lidar com entradas grandes, as quais possuam uma sequência muito longa de bases repetidas, por exemplo, "AAANDN". Nosso algoritmo, com a lógica atual, funcionaria do seguinte modo:

- Percorreria todas as bases de "A" até encontrar bases distintas lado a lado;
- Após isso realizaria todo o procedimento de inserção da terceira base e removeria as 2 bases antigas;
- Por fim, na próxima iteração do algoritmo, ele voltaria a percorrer a lista pelo seu início, ou seja, passaria novamente pelos elementos aos quais já percorreu anteriormente.

Pensando na resolução desse empecilho e aproveitando que estamos trabalhando diretamente com as referências da lista, paramos na nossa sexta e última implementação desse desafio.

Sexta Solução

Na sexta implementação, o nosso desafio era lidar com grandes sequências de bases iguais, e para isso, desenvolvemos um método que trabalha com 2 referências que percorrem a nossa lista e que se adaptam conforme a função de remoção faz a sua chamada. Acompanhe a lógica abaixo.

- Resetamos nossas referências para iniciarem ao início da lista e então percorremos a mesma até encontrar bases distintas lado a lado.

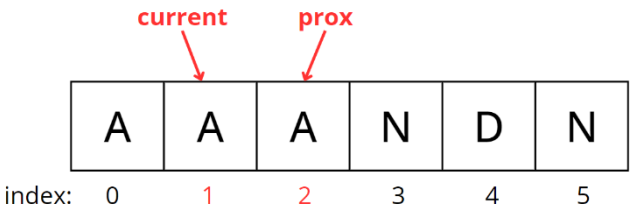

Figura 1: referências alocadas no início da lista

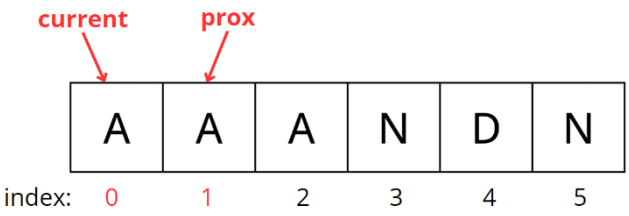

Figura 2: referências percorrendo a lista

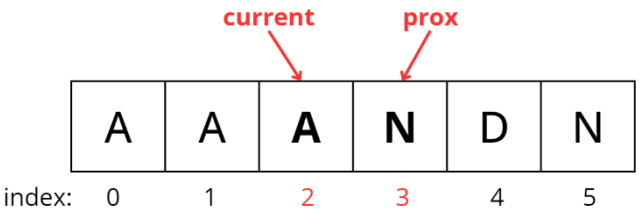

Figura 3: referências encontram bases distintas lado a lado

- Com as bases encontradas, adicionamos ao fim da lista a nova base gerada.

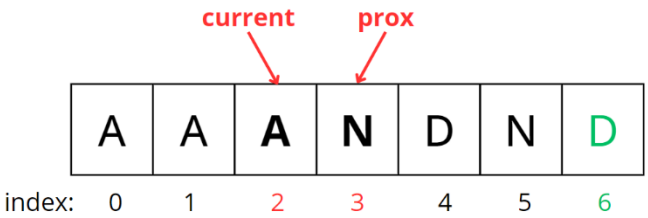

Figura 4: nova base adiciona ao fim da lista

- Antes de realizar a remoção das 2 bases antigas, verificamos se nossas referências se encontram nas extremidades da lista - início ou fim - ou no meio dela, e então as atualizamos de acordo com a sua localização.

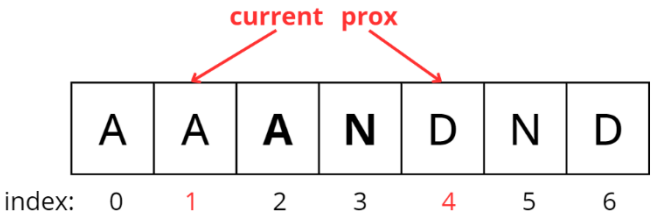

Figura 5: método references() organiza as referências na lista

- Com as referências atualizadas, removemos as 2 bases antigas, e por fim, na próxima iteração é provável que não seja necessário percorrer a lista novamente, já que as referências se encontrarão na exata posição para remover as demais bases.

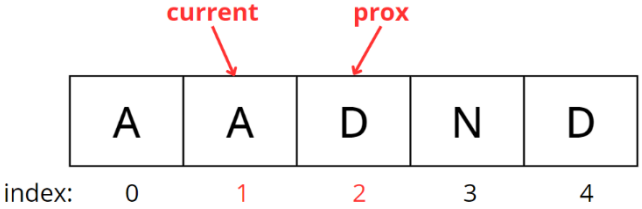

Figura 6: lista pronta para próxima iteração

Para realização desta lógica, foram necessárias poucas modificações na nossa "DoubleLinkedListOfCharacter", como:

- Alteração da Função "remove()"
- Criação da nova Função "references()", responsável por reordenar as referências "current" e "prox" na lista
- Criação da Função "size()", responsável por retornar o tamanho da lista para fazer validações durante a execução do algoritmo

Veja abaixo o pseudocódigo das modificações implementadas.

```
1.  Função remove():
2.     Se contagem = 3 então // Se há apenas 3 elementos
3.        Se current está no início da lista então // Remoção no início
4.           header.próximo = prox.próximo
5.           prox.próximo.anterior = header
6.        Senão // Remoção no fim
7.           current.anterior.próximo = trailer
8.           trailer.anterior = current.anterior
9.        Fim Se
10.    Senão, se current está no início da lista então // Remoção no início
11.       Chamada para a Função references
12.       header.próximo = current
13.       current.anterior = header
14.    Senão // Remoção no meio/fim
15.       Chamada para a Função references
16.       current.próximo = prox
17.       prox.anterior = current
18.    Fim Se
19.    contagem = contagem - 2
20. Fim Função remove
21.
22. Função references():
23.    Se current está no início da lista então
24.       current = prox.próximo
25.       prox = current.próximo
26.    Senão
27.       current = current.anterior
28.       prox = prox.próximo
29.    Fim Se
30. Fim Função references
31.
32. Função size():
33.    Retorna quantidade de elementos na lista
34. Fim Função size
```

Em termos da implementação do algoritmo, não houve grandes mudanças, apenas a realização de algumas validações no método "mutation()".

```
1.  Função mutation():
2.     Posiciona as referências no início da lista
3.     Declaração de variável booleana change
4.
5.     Faça
```

```
6.        change = falso
7.        Faça // Loop que percorre a lista
8.           // Verifica se há elementos para comparar
9.           Se tamanho da lista == 1 então
10.             Interrompa o loop Faça
11.          Fim Se
12.
13.          // Recebe os caracteres que vai comparar
14.          current = lista.getCurrent()
15.          prox = lista.getProx()
16.
17.          // Se os caracteres são diferentes e não são o cabeçalho ou trailer
18.          Se current = '#' ou prox = '#' então
19.             Interrompa o loop Faça
20.          Fim Se
21.
22.          Se current ≠ prox então
23.             Adiciona nova base ao final da lista
24.             Remove as duas bases antiga
25.             change = verdadeiro
26.             Interrompa o loop Faça
27.          Fim Se
28.       Enquanto lista.next()
29.    Enquanto change for verdadeiro
30.    Retorne a lista como string
31. Fim Função mutation
```

Em resumo, após as modificações implementadas na sexta solução, conseguimos alcançar a versão com maior eficiência do nosso algoritmo. Por mais que a utilização de dois iteradores tenha adicionado certa complexidade ao código, os resultados obtidos nos seguintes testes de desempenho demonstraram uma melhora excepcional no tempo de execução. A estrutura final do programa ficou composta por 3 classes:

- **DoubleLinkedListOfCharacter:** estrutura de dados de dupla referência;
- **DNAlgorithm:** classe da lógica do algoritmo (métodos “mutation()” e “generate()”), que implementa a lista, e
- **Main:** classe principal, responsável pela execução do programa, que contém a instância da classe “DNAlgorithm” e dois métodos para realização de testes, sendo um deles por leitura de arquivos.

Por fim, abaixo seguem os resultados obtidos em diferentes versões do programa e, ao final, o teste de desempenho final do algoritmo demonstrado em um gráfico de dispersão.

Resultados

A seguir os resultados com o comparativo do tamanho de entrada e tempo de execução (em nanossegundos) da segunda, quinta e sexta implementação, respectivamente:

Entrada	Tempo(ns)
10	11179000,00
100	8624700,00
1000	16518300,00
10.000	31399100,00
100.00	84535400,00
1.000.000	3583423500,00
10.000.000	33006231200,00
30.000.000	-

Quadro 1: resultados da segunda implementação.

Entrada	Tempo(ns)
10	9802300,00
100	14832600,00
1000	6657800,00
10.000	21219300,00
100.00	37950700,00
1.000.000	287211000,00
10.000.000	4530530700,00
30.000.000	-

Quadro 2: resultados da quinta implementação.

Entrada	Tempo(ns)
10	9821300,00
100	5563500,00
1000	6424000,00
10.000	14478700,00
100.00	27223700,00
1.000.000	236693000,00
10.000.000	3214095800,00
30.000.000	12649536100,00

Quadro 3: resultados da sexta implementação.

Resultado do teste de desempenho final do algoritmo em gráfico de dispersão:

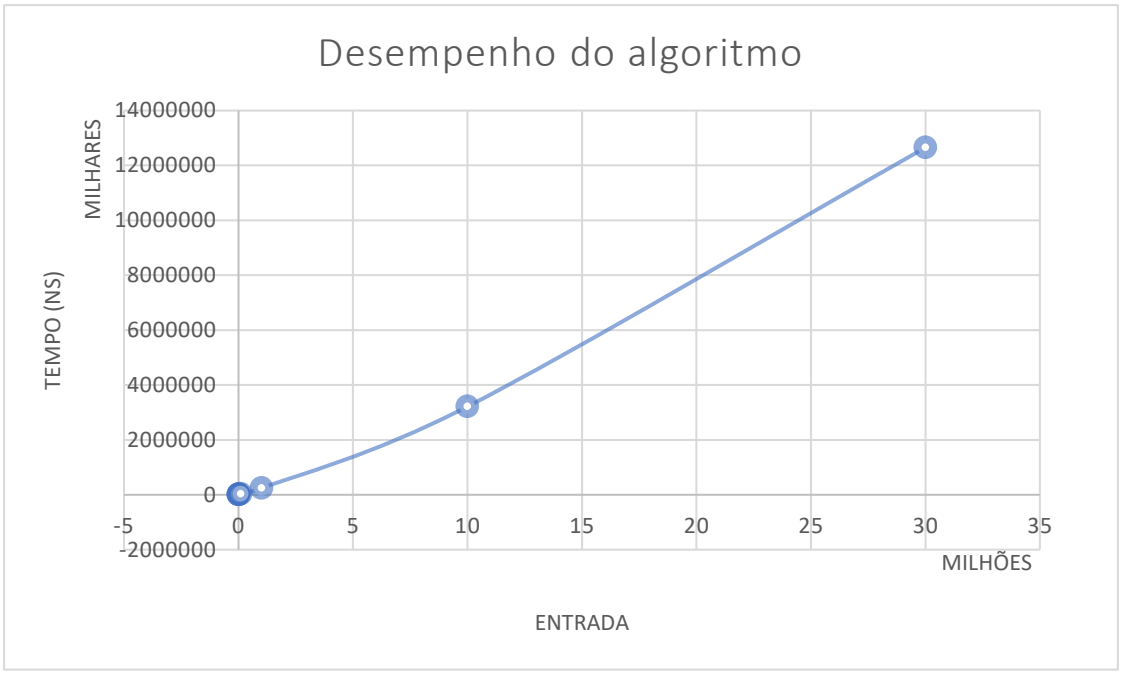

Gráfico 1: demonstração do aumento do tempo (ns), conforme o tamanho da entrada

Conclusão

Na busca por determinar o tamanho da menor cadeia possível após todas as mutações em uma sequência de DNA alienígena, enfrentamos diversos desafios que moldaram nosso conhecimento e aprimoraram nossa abordagem. Este relatório descreve nossa trajetória de seis tentativas distintas para resolver esse complexo problema.

Começamos com uma estrutura de lista encadeada e lógica de mutação de bases, mas logo enfrentamos problemas de desempenho e erros lógicos. Isso nos levou à segunda solução com dois iteradores, que resolveu alguns problemas, mas ainda deixou desafios de desempenho em aberto.

Na terceira tentativa, exploramos a estrutura *ArrayList* do Java, mas seu desempenho não atendeu às nossas expectativas. Retornamos à *LinkedList* do Java e até criamos nossa própria implementação para otimizar a remoção. No entanto, o grande avanço veio com a sexta e última solução, que lidou efetivamente com sequências de bases repetidas, usando referências dinâmicas.

Essa jornada ressalta a importância da otimização, da escolha da estrutura de dados e da atenção aos detalhes na implementação de algoritmos. Finalmente, alcançamos uma versão altamente eficiente do algoritmo que resolveu o desafio com rapidez e precisão, mesmo para entradas extensas. Estamos orgulhosos dessa conquista e esperamos que nossa experiência beneficie outros enfrentando desafios semelhantes no futuro.