Relatório do Trabalho 1 de Algoritmos e Estrutura de Dados II

Dhruv Babani[*], Bernardo Balzan[†]
Faculdade de Ciência Da Computação — PUCRS

18 de setembro de 2023

Resumo

Este artigo descreve alternativas de solução para o primeiro problema proposto na disciplina de Algoritmos e Estrutura de Dados II, no terceiro semestre, que trata da manipulação de uma cadeia de DNA alienígena. São apresentadas três possibilidades de solução para o problema e sua eficiência é analisada. Em seguida as representações obtidas são mostradas.

Introdução

Dentro do escopo da disciplina de Algoritmos e Estrutura de Dados II, o primeiro problema - DNA D/N/A - pode ser resumido assim: Os cientistas, após examinarem cuidadosamente uma espaçonave que caiu no meio do Parque da Redenção, chegaram a algumas conclusões sobre os seres que a ocupavam e que fugiram em uma cápsula de resgate:

- O DNA dos alienígenas é composto por apenas 3 bases, ao contrário das 4 bases presentes no DNA terrestre, e foram denominadas como D, N e A.
- O DNA alienígena sofre mutações que o deterioram com o tempo. Essas mutações ocorrem de maneira organizada, onde duas bases diferentes que estão adjacentes podem se fundir, criando a terceira base e encurtando a cadeia de DNA.
- A fusão das bases ocorre sempre na dupla de bases diferentes mais à esquerda na cadeia de DNA. A nova base resultante da fusão é adicionada ao final da cadeia de DNA

Por exemplo, seguindo estas regras temos que:

- A pequena cadeia DNA sofre uma deterioração em DN e acaba gerando AA.
- Já uma cadeia maior como DNANDANDANDANADNDDDAN acaba virando simplesmente N.

O problema a ser resolvido é escrever um algoritmo capaz de ler as cadeias que os cientistas colocaram em vários arquivos de teste e depois informar o tamanho da menor cadeia que pode ser obtida em cada caso depois de executar todas as mutações possíveis.

Para resolver o problema proposto, analisaremos duas possíveis alternativas de solução, bem como suas características, optando por uma solução bastante eficiente ainda que pouco intuitiva. Fizemos o algoritmo na linguagem Java e foram utilizados códigos padrões de leitura de arquivo e contagem de tempo em ambas soluções, que não afetam o desempenho do algoritmo. Em seguida os resultados obtidos serão apresentados, bem como as conclusões obtidas no decorrer do trabalho.

[*]d.babani001@edu.pucrs.br
[†]b.balzan@edu.pucrs.br

Primeira solução

Depois de considerar o problema, podemos perceber que a análise da cadeia é feita em pares, como por exemplo, “DN” ou “NA” e etc., portanto, nossa primeira ideia foi transformar a cadeia de String em vários Caracteres, um por um, usando o método charAt(), próprio da linguagem Java, e inserindo em um ArrayList, porém antes mesmo de aplicarmos isso no código, percebemos que a estrutura de dados do ArrayList tem como complexidade a notação O(1), o que nos levou a pensar em usar uma LinkedList. Com esta constatação, e sabendo que a LinkedList tem um tempo de complexidade de O(n/2), podemos propor uma solução bastante simples: criar uma LinkedList de Caracteres e fazer o teste, verificando quais pares (nodos) tem letras (elementos) diferentes um do outro e fazendo as “mutações” para diminuir a cadeia de DNA, caso os elementos, um seguido do outro, forem equivalentes, segue-se para o próximo nodo.

Dessa forma, criamos uma segunda LinkedList auxiliar com o objetivo de aumentar o seu tamanho para que evitasse o erro de “IndexOutOfBoundException”, por conta de alguns cenários de que o número de interações era maior do que o próprio tamanho da lista. Além disso, na criação do laço inserimos uma condição de parada, para que ele não fique infinito. Um algoritmo implementando esta ideia seria parecido com este:

```
1  Algoritmo 1
2      Lista resultante
3      Lista copia
4      Caractere elemento1
5      Caractere elemento2
6      Inteiro j = 0
7
8      Para i de 0 até o comprimento de DNA −1 faça:
9          Adicione DNA[i] a lista resultante
10         Adicione DNA[i] a lista copia
11
12     Para i de 0 até 2 elevado a (tamanho de copia) faça:
13         Se (j + 1) ⩾ tamanho de resultante , então:
14             Pare o loop
15         Fim Se
16
17         Se tamanho de resultante ⩾ 2 , então:
18             elemento1 = resultante[j]
19             elemento2 = resultante[j + 1]
20
21             Se (elemento1 seja 'D' E elemento2 seja 'N') OU (elemento1 seja 'N' E elemento2 seja 'D') , e
22                 Remova resultante[j]
23                 Remova resultante[j]
24                 Adicione 'A' àresultante
25                 j = 0
26             Senão Se (elemento1 seja 'A' E elemento2 seja 'N') OU (elemento1 seja 'N' E elemento2 seja '
27                 Remova resultante[j]
28                 Remova resultante[j]
29                 Adicione 'D' àresultante
30                 j = 0
31             Senão Se (elemento1 seja 'D' E elemento2 seja 'A') OU (elemento1 seja 'A' E elemento2 seja '
32                 Remova resultante[j]
33                 Remova resultante[j]
```

```
                Adicione 'N' àresultante
                j = 0
            Senão Se elemento1 seja igual a elemento2 , então:
                j++
            Fim
        Fim
    Fim
Fim
```

Depois de implementar o algoritmo da primeira solução em linguagem Java e executá-lo com os casos testes fornecidos, obtivemos os seguintes resultados:

Caso Teste	String Resultante (Menor Tamanho)	Tempo em Segundos	Número de Operações
Caso 10	D (1)	1.232E-4	24
Caso 100	N (1)	3.134E-4	350
Caso 1000	AA (2)	0.0323299	75633
Caso 10k	AAA (3)	23.6762879	11305175

Esta ideia é bastante simples, porém existem alguns testes de maior escala, as quais, mostram que ela demanda uma abundância de tempo e recursos, tais como os casos de testes acima de 100 mil caracteres. Com isto podemos concluir, que este algoritmo não seria válido para este tipo de problema, por conta dos motivos mencionados anteriormente.

Segunda solução

Uma vez que a alternativa mais simples falha em quase metade dos casos, optamos por desenvolver uma abordagem mais cuidadosa, experimentando um truque bastante simples: Ao invés de usar uma LinkedList de Caracteres, optamos por usar uma API do Java, StringBuilder, tornando o algoritmo mais eficiente e simples. Outrossim, analisamos que há certos cenários que uma letra única se repete várias vezes consecutivamente, e logo pensamos que seria melhor fazer a verificação das letras a serem mutadas sem que volte para o início da String toda vez que acontece uma mutação, assim, continuando na mesma posição. Por conseguinte, o algoritmo atualizado seria este:

```

Função calculoDNA(DNA)
    sb = CriarStringBuilder(DNA)
    elemento1
    elemento2

    Para i de 0 até Potência(Comprimento(sb) , 1000) Passo 1

        Se (j + 1) ⩾ Comprimento(sb) Então
            QuebrarLoop
        Fim

        Se Comprimento(sb) ⩾ 2 Então
            elemento1 = ObterCaracter(sb , j)
            elemento2 = ObterCaracter(sb , j + 1)

            Se elemento1 = elemento2 Então
                j = j + 1
```

```
19                Senão Se (elemento1 = 'D' E elemento2 = 'N') OU (elemento1 = 'N' E elemento2 = 'D') Então
20                    RemoverCaracter(sb , j)
21                    RemoverCaracter(sb , j)
22                    AdicionarCaracter(sb , 'A')
23                    Se j diferente 0 Então
24                        j = j −1
25                    Fim Se
26                Senão Se (elemento1 = 'A' E elemento2 = 'N') OU (elemento1 = 'N' E elemento2 = 'A') Então
27                    RemoverCaracter(sb , j)
28                    RemoverCaracter(sb , j)
29                    AdicionarCaracter(sb , 'D')
30                    Se j diferente 0 Então
31                        j = j −1
32                    Fim Se
33                Senão Se (elemento1 = 'D' E elemento2 = 'A') OU (elemento1 = 'A' E elemento2 = 'D') Então
34                    RemoverCaracter(sb , j)
35                    RemoverCaracter(sb , j)
36                    AdicionarCaracter(sb , 'N')
37                    Se j diferente 0 Então
38                        j = j −1
39                    Fim
40                Fim
41            Fim
42        Fim
43    Fim
```

Resultados

Depois de implementar o algoritmo acima em linguagem Java e executá-lo com os casos testes, obtivemos os seguintes resultados para cada caso teste, sendo as letras conjuntas a String resultante após as mutações juntamente com o seu tamanho, o tempo em segundos e o número de operações necessárias até que o algoritmo gerasse a resposta correta.

Caso Teste	String Resultante (Menor Tamanho)	Tempo em Segundos	Número de Operações
Caso 10	D (1)	1.242E-4	13
Caso 100	N (1)	1.956E-4	157
Caso 1000	AA (2)	0.0012952	1971
Caso 10k	AAA (3)	0.0077987	19963
Caso 100k	AAA (3)	0.1588135	199593
Caso 1000k	N (1)	11.4453871	1999315
Caso 10000k	DDDDDD (6)	1165.2157304	19996988

Após a obtenção destes valores, estes foram testados com ferramentas capazes de lidar com grande cadeias de Strings envolvidos e todos os resultados foram confirmados corretos. Todavia, sempre existirá alguns casos de teste que irão levar um tempo desagradável, ou seja, longo.

Conclusão

As primeiras abordagens, mesmo produzindo resultados parciais, desempenharam um papel significativo no aprofundamento da compreensão do problema e pavimentaram o caminho em direção à solução final. Esta última revelou-se surpreendentemente simples e eficaz, embora tenha demandado uma abundância de recursos como tempo e memória, quando a cadeia de String era de grande escala de tamanho. Portanto, não conseguimos otimizar o algoritmo para todos os cenários possíveis, por exemplo quando mais da metade da cadeia apresenta apenas uma letra repetidamente.

Apesar disso, a solução se mostrou muito eficiente quando se trata de uma cadeia com vários caracteres seguidos diferentes um do outro.

Acreditamos ter desenvolvido uma solução interessante a um problema relativamente complexo, e conseguimos determinar resultados corretos que envolvem cadeia de String de tipo DNA com tamanho colossal.