O DNA D/N/A

Lavínia G. Winter*
Escola Politécnica — PUCRS

24 de setembro de 2023

Resumo

Esse relatório discorre sobre o processo de desenvolvimento e aprimoramento do primeiro problema apresentado na disciplina de Algoritmos e Estrutura de Dados II no semestre 2023/2. O objetivo do algoritmo deve simular a mutação de um material genético alienígena de forma eficiente e precisa.

Foram elaboradas, em Java, três tentativas de solução sendo a última aquela que obteve maior sucesso e velocidade nos casos de teste. Para verificação de eficiência serão apresentados o tempo de processamento, o resultado final de cada mutação e a quantidade de bases restantes para cada caso.

Introdução

Após estudos realizados no material genético encontrado no disco voador que caiu no parque da Redenção foi descoberto um DNA com 3 bases coincidentemente nomeadas D/N/A. Quando duas dessas diferentes bases se misturam, elas se deterioram formando a terceira base restante. Em um exemplo simples, temos a sequência DNA, que após mutada deve apresentar o resultado final de AA. Dito isso, o algoritmo deve analisar as sequências genéticas descritas em formato .txt e simular mutações até que restem somente uma única base, não havendo mais possibilidade de mutação. Dentre esses pontos devem ser observados os seguintes:

1. A leitura da cadeia deve ser feita sempre da esquerda para a direita.

2. Os resultados das fusões de bases devem ser agregados ao fim da sentença, enquanto aqueles que não foram fundidos devem se manter no início.

Para essa tarefa foram fornecidos oito casos de testes prontos, porém foi optado a adição de mais dois casos curtos para maior facilidade ao depurar o código, estes identificados como "tes.txt"e "testecurto.txt"e deverão apresentar os respectivos resultados "AA"e "N".

Uma observação encontrada nos arquivos com os casos de testes é a presença de uma quebra de linha no final. Foi preferido a não remoção da quebra de linha, mas sim a implementação de um trecho de código com uma condição lógica responsável por ignorar espaços em branco.

Para testar pessoalmente os resultados dos casos de teste, no código anexado junto a esse PDF, acesse a linha 23 contendo o seguinte trecho de código 'FileReader reader = new FileReader("testes caso2000.txt");'. Edite o "caso2000"para o nome do caso de teste que você deseja testar.

Serão apresentados a seguir os protótipos de algoritmos desenvolvidos no decorrer da resolução do problema, seus resultados bem como seus tamanhos e tempo de processamento.

*lavinia.winter@edu.pucrs.br

Primeira solução

A primeira solução pensada foi a mais simples: um algoritmo recursivo que percorreria uma lista encadeada realizando os passos necessários até que toda a lista estivesse igual. Expandindo a ideia:

1. O algoritmo adiciona em uma lista os caracteres do arquivo recebido na medida que fosse feita sua leitura.
2. Após isso há uma chamada do método "addHash"que recebe como parâmetros o Hash já criado contendo os valores das chaves resultantes das fusões e a lista contendo os caracteres lidos.
3. Seguindo para o método, haverá a criação de duas outras listas "begin"e "end"que receberão, respectivamente, os caracteres que não foram mutados e os resuldos da fusão.
4. Um loop percorrerá a lista armazenando em variáveis as posições de i e de i+1 e as juntando em uma String com dois caracteres.
5. Essa String passará por uma condicional que avaliará se o hash contém uma chave igual a ela. Se sim, o valor referente a chave é armazenado e adicionado em "end". Se ela não entrar na condicional, o valor referenciado por i é adicionado em "begin".
6. No fim as listas são concatenadas e há a verificação de igualdade de seus elementos que ao retornar falso o método é chamado novamente passando o hash original e o begin como novos parâmetros.

Essa solução apresentou diversos problemas, sendo o seu principal, a evidente demora para o processamento das mutações. No primeiro caso de teste (caso0010) o algoritmo assumiu um tempo tanto normal (menos de 1 minuto) para sua execução, o que foi totalmente diferente no segundo caso de teste, onde apresentou demoras que ultrapassavam 20 minutos.

Isso não se dá somente ao aumento do número de caracteres, mas ao fato de que as bases presentes no segundo caso de teste (caso0020) eram, em sua maioria, 'A', com pouquíssimas incidências de bases 'N' ou 'D'.

Na primeira ocorrência, o algoritmo processou perfeitamente as bases diferentes de 'A' espalhadas em meio a sequência, mandando-as para o fim da lista até sobrar somente uma diferente de 'A'. Com isso, o algoritmo percorreu toda a lista para fazer somente uma modifiação no final. Uma visualização para melhor compreensão seria a seguinte:

[...AAAAAAAD] -> [...AAAAN] -> [...AAAD] -> [...AAN] -> [...AD] -> [...]

O desenvolvimento de um pseudocódigo para esclarecimento foi reservado para a solução final em prol de fornecê-la um foco maior, uma vez que é considerada mais importante. Esse algoritmo só foi testato até o caso de teste 0020 e seus resultados estão apresentados a seguir.

Caso de teste	Tempo de processamento	Tamanho final lista	Resultado
tes	T < 2s	2	"AA"
testecurto	T < 2s	1	"N"
caso0010	T < 3s	1	"A"
caso0020	T > 1.800s	Inconclusivo	Inconclusivo

Houve uma tentativa de aperfeiçoamento desse código utilizando ArrayList, o que resultou em um Stack Over Flow causado pelo consumo excessivo de memória devido a grande ocupação no heap e na stack. Pois o número de vezes que a recursão é chamada, no pior dos casos, é quase do tamanho do vetor passado. Além de que a criação de novas listas para cara chamada de recursão consome uma memória excessiva.

Segunda solução

Com o primeiro algoritmo realizando operações em tempo inacessível para aplicação no cotidiano, foi desenvolvida uma segunda alternativa, com lógica parecida a primeira, porém sem recursividade e utilizando de ArrayList, assim podendo acessar diretamente uma posição, sem perda de eficiência.

```
1  procedimento AddHash(hash , lista)
2  lista begin
3  lista end
4  int i recebe 0
5  int k recebe 1
6  enquanto (i < tamanhoDaSequenciaDNA) {
7      char b obtem Caractere Posicao(k , sequenciaDNA)
8      char a obtem Caractere Posicao(i , sequenciaDNA)
9          se (b = quebra de linha) {
10             escreva("quebra")
11             pare}
12
13     se (a ≠ b) {
14         texto str = a + b
15         caractere valor = obterValorDoHash(str , hash)
16         adicioneAoFim(valor , end)
17         i = k + 1
18         k = k + 2
19     } senao {
20         adicioneAoInicio(a , begin)
21         i++
22         k++
23     }
24     begin recebe end
25     escreva(begin)
26     se (todosIguais(begin)) {
27         retorne begin
28     }
29     sequenciaDNA recebe begin
30 }
```

Os maiores problemas desse código encontram-se em sua organização, dificuldade de leitura e, obviamente o principal, apesar do breve tempo de processamento, é o fato de os resultados finais apresentados são distintos dos resultados obtidos em testes de mesa. A seguir apresentam-se os resultados.

Caso de teste	Tempo de processamento	Tamanho final lista	Resultado
tes	T < 2s	1	"A"
testecurto	T < 2s	1	"A"
caso0010	T < 3s	1	"D"
caso0020	T <5s	1	"A"
caso0050	T <5s	1	"N"

Infelizmente não foi obtido o sucesso na busca pela causa da divergência de resultados. A melhor

hipótese é o programa estar interrompendo o ciclo de repetição antes do previsto e continuando os demais processos com informações inclompletas.

Terceira solução

Finalmente, prosseguimos para a última e mais aprimorada versão. O algoritmo desenvolvido nessa solução se difere consideravelmente dos outros protótipos. A ideia de sua elaboração surgiu durante a aula de HeapSort, o que será expandido a seguir.

Considerando, de forma resumida, que o método de busca durante uma ordenação de um HeapSort se resume a buscar sempre a raiz da "árvore", sendo esse o maior ou o menor valor presente, foi observado que o índice buscado é sempre aquele referenciado pela raiz, ou seja, é como se a variável que o referenciasse não se movesse. Algo similar foi experimentado nessa solução. Esse novo algoritmo tem por objetivo fazer as referências se moverem o mínimo possível ao invés de percorrerem o vetor N vezes.

A opção que mais se encaixou nessa solução foi a utilização de uma lista duplamente encadeada própria com métodos personalizados de mutação e remoção. Primeiramente seriam comparados os dois primeiros nodos e seus respectivos valores. Considere que o primeiro seria o início, há um nodo auxiliar que armazena a referência para ele.

Por que uma lista duplamente encadeada? Simples. Haverão momentos durante a remoção de nodos que o nodo referência voltará a apontar uma posição atrás de si. Para isso, é necessário o conhecimento de um nodo ao seu anterior.

A comparação sempre será sobre o nodo que ele referencia e o próximo de tal. Enquanto o próximo não for nulo, ambos serão juntados em uma string e passados pelo HashMap, se houver um valor para tal chave, esse valor é adicionado ao fim da lista e os dois nodos em questão são removidos, assim o próximo do próximo será o novo início e, portanto, o nodo auxiliar estará apontando para ele.

Se a string de nodos não tiver chave correspondente no HashMap, o que seria o caso de por exemplo, um "AA", então, e somente então, o nodo auxiliar referenciará o próximo, repetindo o método.

```
1  procedimento mutacao(Hashmap < String , Character> )
2      se (vazia) então
3          retorne
4      fim
5      se (tamanho = 1) então
6          retorne
7      Nodo messi recebe inicio ;
8      enquanto (messi não for nulo)
9          se (próximo de messi não for nulo)
10             String chave recebe valor de messi + valor do próximo
11             se (hash contém chave)
12                 adicione chave
13                 messi recebe o retorno de remove
14             fim
15             senão
16                 messi recebe próximo
17             fim
18         fim
19         senão
20             quebra o loop
21         fim
```

```
22      fim
23  fim
```

Explicando o método remove: Para evitar bugs durante a execução foram feitos 3 condicionais.

1. Se o nodo recebido for nulo, o mesmo é retornado.
2. Se o nodo anterior for nulo (significando que o atual é o início) e o nodo após o sequente não for nulo, o início e seu seguinte são removidos e o após o sequente se torna o novo início e novo atual (no pseudocódigo referido como messi).

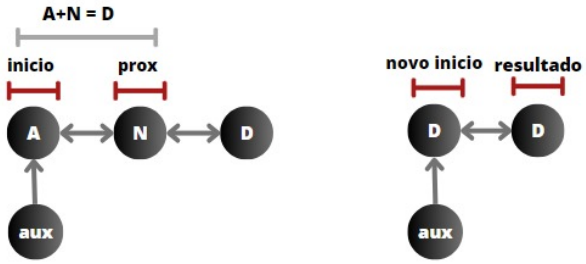

Figura 1: Exemplo de como funciona a primeira condicional do método remove

3. Se o nodo anterior ao atual e o após o sequente não forem nulos, o nodo atual e seu seguinte são removidos e o após o sequente e o anterior do atual, que será o novo atual, irão se referenciar.

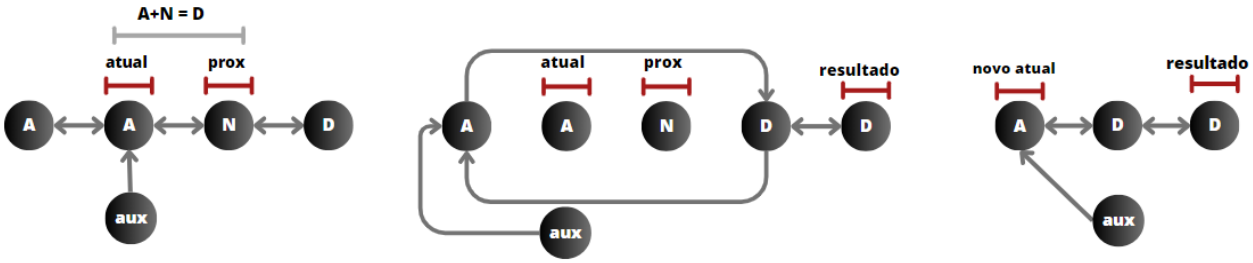

Figura 2: Exemplo de como funciona a segunda condicional do método remove

Para uma representação mais detalhada das funções realizadas nesse método, segue o trecho do algoritmo em pseudocódigo.

```
1  procedimento remover(nodo)
2      se nodo = nulo então
3          retorne nulo
4      fim
5
6      nodoAux1 recebe nulo
7      nodoAux2 recebe nodo.proximo.proximo
8
9      se nodo.anterior = nulo e nodoAux2 ≠ nulo então
```

```
10          nodoAux1 recebe nodo.proximo.proximo
11
12          nodoAux2.anterior recebe nodo.anterior
13          nodo.proximo.anterior recebe nulo
14          nodo.proximo recebe nulo
15          inicio recebe nodoAux1
16          nElementos recebe nElementos −2
17
18      senão se nodo.anterior não for nulo e nodoAux2 ≠ nulo então
19          nodoAux1 recebe nodo.anterior
20          nodo.anterior.proximo recebe nodoAux2
21          nodoAux2.anterior recebe nodo.anterior
22          nodo.anterior recebe nulo
23          nodo.proximo recebe nulo
24          nElementos recebe nElementos −2
25
26      fim
27      retorne nodoAux1
28  fim
```

Esse algoritmo foi bem sucedido em todos os casos de testes apresentados, inclusive demais casos de testes externos e obtiveram o resultado esperado. Segue os mesmos.

Caso de teste	Tempo de processamento	Tamanho final lista	Resultado
tes	T < 50ms	2	"AA"
testecurto	T < 50ms	1	"N"
caso0010	T < 60ms	5	"DDDDD"
caso0020	T <80ms	1	"D"
caso0050	T <150ms	10	"AAAAAAAAAA"
caso0100	T <150ms	1	"D"
caso0200	T <200ms	13	"DDDDDDDDDDDDD"
caso0500	T <250ms	5	"NNNNN"
caso1000	T <350ms	2	"DD"
caso2000	T <700ms	1	"A"

Ao fazer uma análise das operações feitas nesse algoritmo, foi descoberto que ele é, no pior dos casos linear. Esse caso seria todos os caracteres já estarem iguais, da qual ele percorreria o vetor inteiro sem diminuir o tamanho. Já em demais casos, na medida que o vetor é percorrido ele é diminuído, ou seja, o auxiliar andará menos posições do que a quantidade original do vetor.

Houveram diversas tentativas de quebra no código e nenhuma obteve sucesso. Se o algoritmo encontrar um caractere não identificado, o mesmo apenas será ignorado, mantendo a fluidez do código.

Conclusão

A versão final não tem muito em comum com suas versões anteriores, da qual nenhuma entregou resultado satisfatório. O código final é visivelmente menor, mais limpo e legível em comparações com versões anteriores. Precauções sobre quebras de linha foram tomadas, pois as mesmas estão presentes na maioria dos arquivos de teste.

É notável a eficiência tanto em economia de memória quanto de tempo. Isso se deve a lista ser percorrida apenas uma vez, no pior dos casos, sem a necessidade de recursão. No momento em que o loop termina, todos os caracteres estão iguais, não havendo necessidade de uma verificação de igualdade, o que deterioraria a eficiência. A lista será percorrida enquanto houver elementos e se, no meio, forem encontrados elementos diferentes, serão tomadas providências e haverá a regressão em uma posição. Houve também uma preferência em manter a referência se movendo apenas quando necessário, com o bônus de que quando ela se movesse, os caracteres anteriores a ela já estivessem iguais.

As versões anteriores contribuiram para o compreensão do funcionamento de heap e stack durante o processamento de um código. O que antes não era levado com tanta preoucupação, agora era de essencial entendimento. Após aprender sobre como funciona a lógica por trás da locação de memória para a execução de um código, foi facilitado o desenvolvimento de um código que utilizasse o mínimo possível de memória. Somente após entender o problema completamente, foi possível começar a pensar em soluções para ele. Muitas vezes isso pode nos levar para caminhos completamente diferentes do qual estávamos trilhando, como foi o caso da última solução.