Thiago Zilberknop[1], Leonardo Chou da Rosa[2]

Algoritmos e Estrutura de Dados II - Trabalho 1

Faculdade de Engenharia da Computação — PUCRS

9 de Setembro 2023

Resumo

O seguinte artigo descreve o processo para solução do problema proposto pelo Trabalho 1 da disciplina de Algoritmos e Estruturas de Dados II, ministrada pelo professor Alexandre Agustini na Pontifícia Universidade Católica do Rio Grande do Sul. O trabalho trata de formular e analisar um algoritmo capaz de processar entradas de DNA "alienígena". São propostas três soluções — uma na linguagem Python e duas na linguagem C++ — e a eficiência de todas é analisada. Por fim, exemplos de execução dos três algoritmos são mostrados passo a passo.

Introdução

A situação proposta pelo trabalho foi a descoberta de cientistas sobre um código genético alien composto de bases nitrogenadas diferentes de um DNA típico, essas denominadas D, N e A. Uma curiosidade percebida pelos pesquisadores foi a de que as bases do DNA alienígena degeneram rapidamente dependendo da construção do material genético, de acordo com a seguinte lógica:

1) Se duas bases diferentes estão uma do lado da outra, ambas degeneram na terceira base. Por exemplo, AN sofre uma mutação para a base D;
2) A base degenerada é então adicionada ao final da cadeia de DNA. Por exemplo, na sequência AND, o par AN sofre degeneração para D, e essa nova base é adicionada ao final da cadeia, resultando em uma sequência nova DD;
3) Esse processo continua até que não existam pares de bases diferentes.

O problema a ser resolvido, então, era a criação de um algoritmo capaz de deduzir o DNA degenerado de uma sequência qualquer de bases aliens. Uma vez completo, o algoritmo deve ter sua eficiência analisada para determinar se a solução proposta é a mais otimizada possível, e se não, quais mudanças poderiam ser feitas para a melhor adequação do algoritmo.

A importância de desenvolver um algoritmo que resolve o problema proposto está relacionado à administração de uma quantidade grande de dados. No caso proposto, não há limite de caracteres que um DNA alienígena pode conter, portanto é necessário desenvolver um algoritmo que não só consegue executar corretamente, mas também consegue executar em uma eficiência adequada; o algoritmo tem que ser otimizado. Em um problema simples como o proposto, o tempo de execução não vai ser perdurante mesmo em casos de cadeias longas, mas em um contexto mais aberto, onde várias operações

[1] thiago.zilberknop@edu.pucrs.br
[2] l.chou@edu.pucrs.br

são necessárias para um grupo enorme de dados, a otimização se torna algo crucial para qualquer código. O propósito deste trabalho é para os alunos aprenderem como desenvolver e analisar códigos que lidam com muitas informações.

Algoritmo 1 — C++

A abordagem do problema através da língua C++ foi uma de criar uma função iterativa que percorre uma *string* chamada **dna** avaliando pares de caracteres do seu início ao fim. Primeiramente, fora escrito em linguagem natural o que o algoritmo seria:

```
1       string solution(string dna) {
2         para cada par de letras em dna {
3           se o par atual for “DN” ou “ND” {
4             adiciona ‘A’ ao final de dna;
4             remove o par atual de dna;
5           }
6           se o par atual for “DA” ou “AD” {
8             adiciona ‘N’ ao final de dna;
9             remove o par atual de dna;
10          }
11          se o par atual for “AN” ou “NA” {
12            adiciona ‘D’ ao final de dna;
13            remove o par atual de dna;
14          }
15        }
16        retorna dna;
17      }
```

Como pode ser observado, esse trecho de pseudo-código descreve um laço **for** cujo index **i** começa em zero e aumenta até seu valor ser igual ao tamanho de **dna-1.** É importante que o index pare no penúltimo caractere, pois a avaliação é feita par a par, e não caractere a caractere. Isto é, é necessário ter acesso ambos ao caractere **dna[i]** e ao caractere **dna[i+1]**. Dado que a tentativa de acesso a um elemento que estaria além do tamanho pré-definido do vetor causa um erro fatal — e por consequência um fechamento brusco do programa — é primordial que o laço encerre antes do index atingir o tamanho de **dna**. Logo, um refinamento do pseudo-código para algo que mais se assemelhe a um veridico trecho de C++ seria:

```
1       string solution(string dna) {
2         for (int i = 0; i < dna.size() - 1; i++) {
3           ...
            ...
            ...
16        return dna;
17      }
```

Em seguida, observa-se que as declarações condicionais podem ser configuradas como declarações **if** cujas condições são, por exemplo: se **dna[i]** for igual a D, e **dna[i+1]** for igual a N, ou vice-versa, delete ambos caracteres e adicione A no final de **dna**. Traduzindo para C++:

```
1       string solution(string dna) {
2         for (int i = 0; i < dna.size() - 1; i++) {
```

```
3           if ((dna[i] == 'D' && dna[i+1] == 'N') || (dna[i] == 'N' && dna[i+1] == 'D')) {
4             dna += 'A';
5             dna.erase(i, 2);
6           }
7           if ((dna[i] == 'D' && dna[i+1] == 'A') || (dna[i] == 'A' && dna[i+1] == 'D')) {
8             dna += 'N';
9             dna.erase(i, 2);
10          }
11          if ((dna[i] == 'A' && dna[i+1] == 'N') || (dna[i] == 'N' && dna[i+1] == 'A')) {
12            dna += 'D';
13            dna.erase(i, 2);
14          }
15        }
16        return dna;
17      }
```

Aqui, o algoritmo se beneficia da STL (Standard Template Library) de C++, utilizando a sobrecarga pronta do operando +=[3], que adiciona no final de uma *string* o que está à direita do operando, e da função-membro *erase*, que, neste caso, remove caracteres a partir do index até um caractere após — ou seja, o par de letras — e corretamente manipula o resto dos caracteres para que não haja espaços nulos na *string*, como também mudando o tamanho da propria para refletir a perda de duas letras.

No estado atual, o código não está completo, pois ele tem um problema: como **`i`** sempre aumenta uma unidade no final de uma execução do laço, é possível que o algoritmo pule um ou mais pares de letras, uma vez que se um par for removido e uma letra adicionada no final, **`dna`** teve seu tamanho efetivamente reduzido por um. Logo, segue que se a *string* for reduzindo de tamanho, mas o index continua aumentando no mesmo ritmo, invariavelmente o algoritmo "esquece" de letras. Por exemplo, supondo uma *string* DNDNA, o algoritmo teria o seguinte desenvolvimento:

1) Primeira execução do laço, com **`i = 0`**. DN é um par de letras diferentes, logo é deletado e A é adicionado ao final, resultando em DNAA. Por fim, o index é incrementado;
2) Segunda execução do laço, com **`i = 1`**. NA é um par de letras diferentes, logo é deletado e D é adicionado ao final, resultando em DAD. Por fim, o index é incrementado. Isso aconteceu pois **`i`** é 1, e por conseguinte **`dna[i]`** — isto é, **`dna[1]`** — é a segunda letra da *string*, N, e **`dna[i+1]`** é a terceira letra, A, assim formando o par NA, pulando o par correto a ser avaliado, DN, por uma letra;
3) Como o index já atingiu o valor máximo que ele pode ter (**`i = 2`**, uma vez que o tamanho da *string* agora é três), não há uma terceira execução. A função então retorna **`dna`**.

Tendo em vista este erro, é necessário corrigir o valor do index em todo disparo de condicional, senão o algoritmo nunca processará suas entradas de modo correto:

```
1       string solution(string dna) {
2         for (int i = 0; i < dna.size() - 1; i++) {
3           if ((dna[i] == 'D' && dna[i+1] == 'N') || (dna[i] == 'N' && dna[i+1] == 'D')) {
4             dna += 'A';
5             dna.erase(i, 2);
6             i = i - 2;
7             if (i <= 0) i = -1;
```

[3] A sobrecarga do operador += aplica a função-membro **`append()`** de *string*, passando à ela como parâmetro o que está à direita do operador.

```
8           }
9           if ((dna[i] == 'D' && dna[i+1] == 'A') || (dna[i] == 'A' && dna[i+1] == 'D')) {
10            dna += 'N';
11            dna.erase(i, 2);
12            i = i - 2;
13            if (i <= 0) i = -1;
14          }
15          if ((dna[i] == 'A' && dna[i+1] == 'N') || (dna[i] == 'N' && dna[i+1] == 'A')) {
16            dna += 'D';
17            dna.erase(i, 2);
18            i = i - 2;
19            if (i <= 0) i = -1;
20          }
21        }
22        return dna;
23      }
```

Percebe-se que, ao decrementar o index por dois em cada **`if`**, o algoritmo percorre a *string* de maneira plena, sem pular caracteres quaisquer. Porém, esse mesmo decremento tem um efeito indesejável que é corrigido pela condicional no fim do **`for`**, pois ao diminuir **`i`** em dois, se o index estiver valendo zero, ele levará o código a acessar, na próxima execução, uma posição inválida de memória, no caso **`dna[-1]`**.

A condicional em questão, portanto, corrige o index para com que ele mais uma vez conduza o programa ao acesso correto da primeira letra da *string*, assim completando o algoritmo. Foram conduzidos os casos de testes providenciados, e embora inicialmente essa solução tenha sido satisfatória — isto é, o algoritmo retornou as respostas esperadas para cada caso de teste com uma complexidade de tempo adequada — não era evidente que essa solução já seria a mais eficiente para o problema. Foi formulada, então, outra solução em Python.

Algoritmo 2 — Python

Tendo contemplado plenamente o problema com o desenvolvimento da primeira solução, buscamos uma outra maneira de processar o DNA alien em Python. Isso é feito tanto como um método de aprendizagem e prática para a linguagem Python em si, como também para nos beneficiarmos das características da língua, que torna fácil a construção de métodos que por sua vez teriam uma implementação potencialmente complexa numa língua de mais baixo nível como C++.

```
1       def alesti2_t1(string):
2         dictionary = {"AD": "N", "DA": "N", "DN": "A", "ND": "A", "AN": "D", "NA": "D"}
3         for i in range(len(string)):
4           if string[i:i+2] in dictionary:
5             newString = string[:i] + string[i+2:] + dictionary[string[i:i+2]]
6             return alesti2_t1(newString)
7         return operations, string
```

De imediato é evidente a simplicidade de Python em comparação a C++, uma vez que o algoritmo é cerca de três vezes menor em linhas. Entretanto, isso não significa que o algoritmo seja mais eficiente, pois a maior legibilidade do código não implica numa complexidade de tempo menor. De fato, ela na verdade oculta o que realmente está acontecendo no algoritmo, e embora essa solução possua um comportamento semelhante à de C++, uma rápida análise dela torna claro que ela tem um problema severo que se encontra na linha 6:

```
6           return alesti2_t1(newString)
```

Essa linha denota que essa solução é recursiva, e isso foi necessário pois Python — entre outras diferenças com C++ — não permite alterar o valor de variáveis que foram declaradas como *string* ou que, em algum momento de um programa, receberam uma *string*. Isso significa que, em qualquer função em Python onde uma *string* está sendo manipulada, é necessário criar uma nova variável que conterá as alterações feitas na *string* original. Consequentemente, não é possível trabalhar em um mesmo laço com um nome ou variável contendo *strings* diferentes da primeira à qual lhe foi atribuída, assim necessitando uma solução recursiva, onde cada nova chamada da função recebe como parâmetro a nova *string*.

Essa solução, embora funcional, encontra uma barreira definida por padrão em Python: o máximo de "profundidade" que uma função recursiva pode alcançar é mil chamadas recursivas. Logo, o algoritmo se encerra com casos de teste de tamanho pequeno, e mesmo mudando esse limite, tal como:

```
1     import sys[4]
2     sys.setrecursionlimit(100_000_000)
3     ...
```

O programa seguirá se encerrando quando acabar a memória disponível. Claro que, fosse a memória infinita, essa solução teria uma performance semelhante ao primeiro algoritmo, uma vez que o número de execuções dos laços para ambos os algoritmos é o mesmo para cada caso de teste. Porém, no mundo real, o algoritmo se encerra em uma máquina com 32gb de RAM tentando processar o caso de teste com um milhão de letras, logo por mais que ele seja funcional, ele não é aplicável com entradas muito maiores que 50 mil letras, e mesmo assim é apenas funcional condicionado ao tamanho da memória. Enfim, Python se mostrou inadequado para a resolução do problema, mas as reflexões tiradas durante a construção e análise do algoritmo foram fundamentais para a solução final.

Algoritmo 3 — C++ com Mapa

Apesar do segundo algoritmo ter sido falho, o exercício foi produtivo pois a ideia de usar uma estrutura de dados como um dicionário para os pares de letras foi interessante. Foi feita, então, uma terceira possível solução usando esta ideia:

```
1     string solution(string dna) {
2       unordered_map<string, char> dict = {{"AN", 'D'}, {"NA", 'D'}, {"DA", 'N'},
3                                          {"AD", 'N'}, {"DN", 'A'}, {"ND", 'A'}};
4       for (int i = 0; i < dna.size() - 1; i++) {
5         string str = dna.substr(i, 2);
6         auto chara_to_add = dict.find(str);
7         if (chara_to_add != dict.end()) {
8           dna.erase(i, 2);
9           dna += chara_to_add->second;
10          if (i <= 0) i = -1;
11        }
12      }
13      return dna;
```

[4] **sys** é uma biblioteca padrão de Python que lida com funções e parâmetros específicos ao interpretador de Python.

Esse terceiro algoritmo faz uso extenso da STL, usando a estrutura de dados *unordered_map*, que é análogo a um mapa de hash. A implementação deste mapa é idêntica ao dicionário que foi feito no algoritmo em Python, fazendo pares de chaves com valores, sendo as chaves os pares de letras e os valores as letras que os pares viram quando são substituídos.

Dentro do **`for`**, uma *substring* é guardada em **`str`** a partir de **`i`** até dois caracteres depois, **`dna[i]`** sendo o primeiro e **`dna[i+1]`** sendo o segundo, como no primeiro algoritmo. Em sequência, uma nova variável **`chara_to_add`**, ou "caractere a ser adicionado" em Português, recebe um ponteiro para o par de chave-valor no **`dict`** que possua a chave **`str`**, ou um ponteiro para o "final" da estrutura se nenhuma chave for encontrada. Por fim, o par de letras é deletado e a letra chaveada àquele par é adicionada no fim de **`dna`**. Em linguagem natural:

```
        ...
5       Guarda o par a ser avaliado em str;
6       Guarda o endereço do par chave-valor que contém str como chave em chara_to_add, ou o
endereço do fim da estrutura dict se o par de letras não existir em dict;
7       Se chara_to_add não apontar para o fim de dict {
8         Remove o par de letras sendo avaliado de dna;
9         Adiciona a letra do par de chave-valor cuja chave é str no final de dna;
        ...
```

Observa-se que, tanto em linguagem natural como no código em si, a linha 6 é a peça de maior complexidade, tal que o tipo *auto*[5] foi utilizado para poupar o esforço de detalhar por extenso o que **`chara_to_add`** realmente é. Destrinchando o *auto*, a linha 6 teria seu comprimento mais que dobrado:

```
6           unordered_map<string, char>::iterator chara_to_add = dict.find(str);
```

Em segundo momento, a função-membro de *unordered_map* `find()` é chamada para "achar" o par chave-valor que tenha a chave **`str`**. `Find()` retorna um *iterator* — tipo especial de ponteiro que aponta para elementos contidos em estruturas de dados — apontando ou para o elemento de **`dict`** que contenha a chave usada como parâmetro ou para o "fim" da estrutura. Fim é usado com aspas pois o *iterator*, diferentemente do que seria intuitivo pensar, não aponta para o último elemento da estrutura, mas sim para uma posição de memória logo após dele, que é especificamente reservada para indicar o término de uma estrutura. A função-membro `end()`, na linha seguinte, retorna este mesmo ponteiro, possibilitando declarações **`if`** tal qual a da linha em questão.

O benefício de usar um mapa como o *unordered_map* é o tempo de complexidade para as funções-membro da classe. No caso, como o mapa funciona como um mapa de hash, a maioria das operações feitas na estrutura tem complexidade de tempo em média constante[6], ou seja, uma única operação para a maioria das funções-membro. Isso é muito mais atrativo como solução do que a série de condicionais do primeiro algoritmo, que, a olho nú, necessitam um número maior de operações para serem realizadas do que uma combinação de `find()`, `end()`, e um **`if`** com apenas uma comparação.

---

[5] "A palavra-chave *auto* direciona o compilador para usar a expressão de inicialização de uma variável declarada ou um parâmetro de expressão lambda para deduzir o tipo. (...) A palavra-chave *auto* é uma maneira simples de declarar uma variável que tenha um tipo complicado. Por exemplo, você pode usar *auto* para declarar uma variável onde a expressão de inicialização envolve modelos, ponteiros para as funções ou ponteiros para os membros." — auto (C++) | Microsoft Learn

[6] De acordo com o website *cppreference.com*, cujo conteúdo é documentação de C/C++ e da STL, um "*unordered_map* é um *container* associativo que contém pares de chave-valor com chaves únicas no qual a busca, inserção, e remoção de elementos tem em média complexidade de tempo constante." — std::unordered_map - cppreference.com

Isto posto, com duas soluções funcionais resta determinar qual é a mais eficiente na resolução do problema. Foram coletados dados experimentais — número de operações, repetições de **`for`**, e até tempo real cronometrado — baseado nos casos de teste providenciados na ementa do trabalho.

Análise

Para determinar se os algoritmos desenvolvidos eram eficientes, nós estabelecemos alguns critérios para análise:

1) A quantidade de loops que o algoritmo faz durante o processo.
2) A velocidade do algoritmo em relação ao tamanho da palavra.
3) A quantidade de operações que o algoritmo exerce na execução.

Loops

A quantidade de **loops** foi determinada pela quantidade de vezes que o **`for`** da função foi chamado. No código, a aplicação ficou descrita assim:

Algoritmo 1

```
1       string solution(string dna) {
2         for (int i = 0; i < dna.size() - 1; i++) {
3           if ((dna[i] == 'D' && dna[i+1] == 'N') || (dna[i] == 'N' && dna[i+1] == 'D')) {
4             dna += 'A';
5             dna.erase(i, 2);
6             i = i - 2;
7             if (i <= 0) i = -1;
8           }
9           if ((dna[i] == 'D' && dna[i+1] == 'A') || (dna[i] == 'A' && dna[i+1] == 'D')) {
10            dna += 'N';
11            dna.erase(i, 2);
12            i = i - 2;
13            if (i <= 0) i = -1;
14          }
15          if ((dna[i] == 'A' && dna[i+1] == 'N') || (dna[i] == 'N' && dna[i+1] == 'A')) {
16            dna += 'D';
17            dna.erase(i, 2);
18            i = i - 2;
19            if (i <= 0) i = -1;
20          }
21         loop++;
22        }
23        return dna;
24      }
```

Algoritmo 2

```
1       string solution(string dna) {
2         unordered_map<string, char> dict = {{"AN", 'D'}, {"NA", 'D'}, {"DA", 'N'},
3                                             {"AD", 'N'}, {"DN", 'A'}, {"ND", 'A'}};
```

```
4         for (int i = 0; i < dna.size() - 1; i++) {
5           string str = dna.substr(i, 2);
6           auto chara_to_add = dict.find(str);
7           if (chara_to_add != dict.end()) {
8             dna.erase(i, 2);
9             dna += chara_to_add->second;
10            i = i - 2;
11            if (i <= 0) i = -1;
12          }
13         loop++;
14        }
15        return dna;
```

Nota-se que, após cada iteração do **for**, a variável **`loop`** foi incrementada. Após esse ajuste, rodamos os dois algoritmos com os casos de testes fornecidos, e inserimos todos os valores em uma tabela:

# Teste	**Tamanho da String**	**Loops (Algoritmo 1)**	**Loops (Algoritmo 2)**	**Palavra Final**
test01	50	80	80	DD
test02	500	863	863	N
test03	5,000	8,781	8,781	N
test04	50,000	87,025	87,025	N
test05	50,001	87,286	87,286	A
test06	50,000	99,995	99,995	N
test07	50,000	49,999	49,999	A*50,000
test08	500,000	872,347	872,347	NN
test09	2,500,000	4,364,877	4,364,877	AA
test10/caso5M	5,000,000	8,725,435	8,725,435	N
test11/caso5Ma	5,000,000	4,999,999	4,999,999	A*5,000,000
test12/caso10M	10,000,000	17,460,908	17,460,908	D

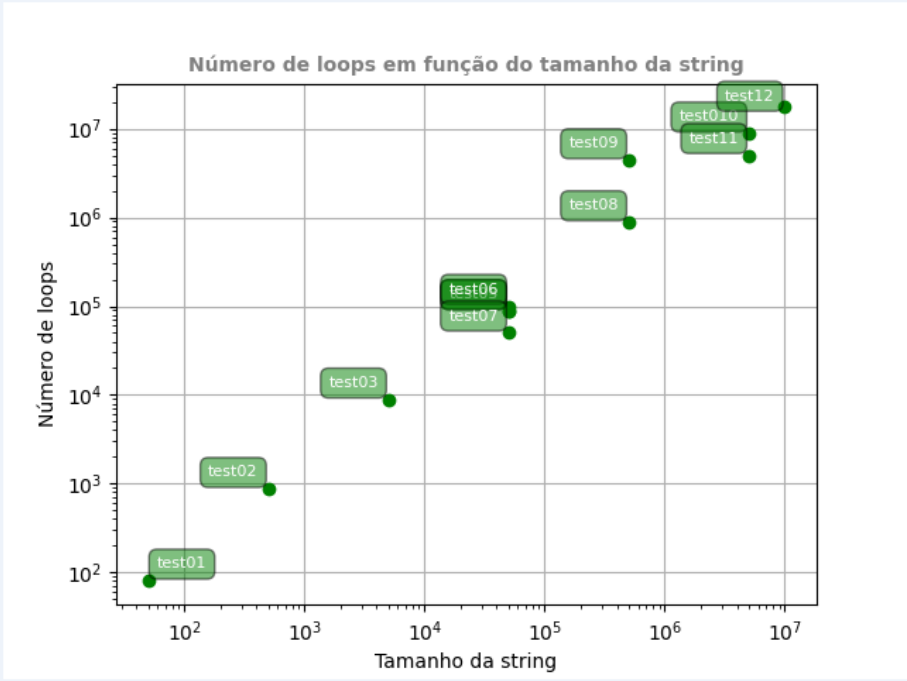

Gráfico que mostra a relação entre a quantidade de loops e o tamanho da string (escalas logarítmicas)

Após os testes de **loops**, determinamos que os algoritmos são extremamente parecidos um com o outro em termos de performance; a quantidade de **loops** mostraram valores exatamente iguais em todos os testes. Somente com essa informação, não podemos determinar se um algoritmo é melhor que o outro, portanto decidimos testar os outros critérios para gerar uma conclusão.

Grau de Complexidade

Considerando que as duas soluções contêm a mesma quantidade de chamadas no **`for`**, podemos inferir que os dois algoritmos contém o mesmo grau de complexidade. A nossa impressão inicial, somente observando a tabela e o gráfico é que os algoritmos são polinomiais, evidente pela reta quando as duas escalas estão em escala logarítmica. No melhor caso, o algoritmo executa em uma complexidade O(n-1), evidente pelos casos de testes "test07" e "caso5Ma", onde todas as letras são iguais. O pior caso deve ser um pouco menos do que O(2n), evidente pelo "test06", onde todas as letras são iguais menos as primeiras duas, forçando uma mutação em cada iteração. Para calcular o grau de complexidade, fizemos o seguinte:

$$b \approx \frac{log(17{,}460{,}908) - log(80)}{log(10{,}000{,}000) - log(50)} \approx 1{,}007$$

A equação nos mostrou que a função polinomial contém uma exponencial $n^{1{,}007}$, porém não é o resultado final que queremos, pois os valores da *string* elevada a esta exponencial não encaixam com o valor de **loops** final da função; há uma constante que multiplica esse valor gerado:

$$10{,}000{,}000^{1{,}007} = 11{,}194{,}378 \neq 17{,}460{,}908$$

Para achar a constante, nós simplesmente utilizamos álgebra:

$$17,460,908 = 11,194,378x$$

$$x = \frac{17,460,908}{11,194,378} = 1,559$$

E concluímos que a função final dos algoritmos propostos é:

$$f(n) = 1,559(n)^{1,007}$$

Onde n é a quantidade de caracteres na *string* e f(n) é a quantidade de **loops** da função.

Tamanho da String	**Loops (calculado pelo algoritmo)**	**Loops (calculado pela função)**
50	80	80
500	863	814
5,000	8,781	8,273
50,000	87,025	84,083
50,001	87,286	84,084
500,000	872,347	854,493
2,500,000	4,364,877	4,320,875
5,000,000	8,725,435	8,683,781
10,000,000	17,460,908	17,452,036

Os valores apresentados na tabela se referem ao caso médio de teste, onde as letras estão em ordens aleatórias. Em casos onde todas as letras são iguais, a quantidade de loops diminuem, e casos onde todas as letras são iguais menos as primeiras duas, a quantidade aumenta.

Tempo

Para achar o tempo de execução do algoritmo, inicialmente tentamos implementar uma forma de relógio dentro do algoritmo, porém não conseguimos achar uma maneira eficiente de executar essa ideia. A segunda solução, que nós acabamos usando, foi de cronometrar o algoritmo com um relógio externo (*stopwatch* no celular), onde ele foi iniciado no momento em que o algoritmo foi iniciado e terminado quando o algoritmo imprimia a palavra final.

O desafio dessa solução é que ela contém uma margem de erro humano; uma pessoa nunca vai conseguir parar exatamente no mesmo ponto do que o programa inicia e termina. Ademais, o tempo de execução real varia de máquina a máquina, sendo possível um mesmo trecho de código ser executado em dois segundos com um computador e em vinte segundos em outro. Em vista disso, determinamos uma margem de erro de meio segundo em cada tempo obtido e usamos a mesma máquina em situações iguais

para a coleta de dados. Similarmente ao teste de loops, utilizamos os arquivos fornecidos para desenvolver uma tabela de tempo:

# Teste	Tamanho da String	Tempo do Algoritmo 1 (em segundos)	Tempo o Algoritmo 2 (em segundos)
test01	50	<1	<1
test02	500	<1	<1
test03	5,000	<1	<1
test04	50,000	<1	<1
test05	50,001	<1	<1
test06	50,000	<1	<1
test07	50,000	<1	<1
test08	500,000	3	3
test09	2,500,000	75 (1 minuto e 15 segundos)	77 (1 minuto e 17 segundos)
test10/caso5M	5,000,000	379 (6 minutos e 19 segundos)	380 (6 minutos e 20 segundos)
test11/caso5Ma	5,000,000	<1	<1
test12/caso10Ms	10,000,000	2344 (39 minutos e 4 segundos)	2348 (39 minutos e 8 segundos)

Gráfico de tempo para cada teste no Algoritmo 1

Gráfico de tempo para cada teste no Algoritmo 2

É curioso destacar que o "caso5Ma" é uma exceção que mostra que o tamanho da *string* não é o fator mais significativo para o tempo de execução. Este teste é diferente pois ele não contém nenhuma instância de mutação entre as letras, consistindo em 5,000,000 de letras "A". Os algoritmos então conseguem percorrer toda a *string* em menos de 1 segundo, sem ter que exercer uma função dentro das condicionais que fazem o algoritmo funcionar.

Em termos de performance, os algoritmos demonstraram tempos quase idênticos em todos os casos de teste. As únicas diferenças de tempo que os dois algoritmos mostram são de 1 segundo ou menos de 1 segundo, diferença que figura na margem de erro humano. Esses resultados não são surpreendentes, pois os dois algoritmos contém o mesmo grau de complexidade. Percebe-se que, ao observar os gráficos, o tempo de execução do algoritmo está diretamente ligado com a quantidade de **loops**; quanto mais vezes a função é chamada, mais lento o algoritmo ficará. A exceção dessa observação acontece nos casos onde a *string* sendo modificada não contém nenhuma instância de mutação, ou seja, a *string* final é igual a original. Nestes casos mais específicos, o programa executa instantaneamente. Com todas essas informações, geramos as seguintes conclusões:

1) O algoritmo fica mais lento quando a *string* sendo analisada aumenta;
2) Se a *string* sendo analisada não precisar de nenhuma mutação, o programa é executado instantaneamente;
3) Há um aumento exponencial no tempo quando o tamanho da *string* chega em um certo ponto (milhões); o tempo de execução de 5,000,000 caracteres para 10,000,000 aumentou em quase sete vezes, mas o tamanho da *string* aumentou somente em duas vezes.

Operações

Sabendo que a quantidade de loops dos dois algoritmos, a complexidade e o tempo de execução são iguais, nós decidimos contar a quantidade de operações entre os dois para definir qual é mais eficiente. Para calcular a quantidade de operações, uma nova variável global **`numOps`** é introduzida ao código em ambos

algoritmos. Após cada linha, ou antes no caso de condicionais, ela sofre um incremento igual ao número de operações realizadas, sendo essas atribuições, somas, comparações, e etc.

Algoritmo 1

```
1       string solution(string dna) {
2         for (int i = 0; i < dna.size() - 1; i++) {
3           numOps += 7; //comparações do if
4           if ((dna[i] == 'D' && dna[i+1] == 'N') || (dna[i] == 'N' && dna[i+1] == 'D')) {
5             dna += 'A';
6             numOps += dna.size(); //complexidade de append é tamstring no pior caso
7             dna.erase(i, 2);
8             numOps += dna.size(); //complexidade de erase é tamstring no pior caso
9             i = i - 2;
10            numOps += 2; //atribuição e soma
11            if (i <= 0) i = -1;
12            numOps += 2; //comparação do if e atribuição
13          }
14          if ((dna[i] == 'D' && dna[i+1] == 'A') || (dna[i] == 'A' && dna[i+1] == 'D')) {
15            dna += 'N';
16            numOps += dna.size(); //complexidade de append é tamstring no pior caso
17            dna.erase(i, 2);
18            numOps += dna.size(); //complexidade de erase é tamstring no pior caso
19            i = i - 2;
20            numOps += 2; //atribuição e soma
21            if (i <= 0) i = -1;
22            numOps += 2; //comparação do if e atribuição
23          }
24          if ((dna[i] == 'A' && dna[i+1] == 'N') || (dna[i] == 'N' && dna[i+1] == 'A')) {
25            dna += 'D';
26            numOps += dna.size(); //complexidade de append é tamstring no pior caso
27            dna.erase(i, 2);
28            numOps += dna.size(); //complexidade de erase é tamstring no pior caso
29            i = i - 2;
30            numOps += 2; //atribuição e soma
31            if (i <= 0) i = -1;
32            numOps += 2; //comparação do if e atribuição
33            }
34           numOps += 2; //i = i+1
35        }
36        return dna;
37      }
```

Algoritmo 2

```
1       string solution(string dna) {
2         unordered_map<string, char> dict = {{"AN", 'D'}, {"NA", 'D'}, {"DA", 'N'},
3                                             {"AD", 'N'}, {"DN", 'A'}, {"ND", 'A'}};
4         numOps += 6; //atribuição em unordered_map é no caso médio O(n)
5         for (int i = 0; i < dna.size() - 1; i++) {
6           string str = dna.substr(i, 2);
7           numOps += 1 + 2; //atribuição, substring pior caso = tamanho da substring
```

```
8           auto chara_to_add = dict.find(str);
9           numOps += 1 + 1; //atribuição e busca, caso médio constante (1)
10          numOps += 1 + 1; //comparação e acesso ao pointer o fim de um unordered_map, constante
11          if (chara_to_add != dict.end()) {
12            dna.erase(i, 2);
13            numOps += dna.size(); //erase pior caso = tam da string nova
14            dna += chara_to_add->second;
15            numOps += dna.size(); //append/sobrecarga += pior caso = tam da string nova
16            i = i - 2;
17            numOps += 2; //atribuição e soma
18            numOps += 1; //comparação
19            if (i <= 0) i = -1;
20            numOps += 1; //atribuição
21          }
22         numOps += 2; //i = i+1
23         loop++;
24        }
25        return dna;
```

Cada algoritmo tem sua quantidade de operações específica, pois eles contém lógicas diferentes, mesmo sendo muito parecidos em performance. Esses contadores geraram a seguinte tabela*:

# Teste	Operações (Algoritmo 1)	Operações (Algoritmo 2)	# Loops
test01	4,425	3,312	80
test02	270,692	258,617	863
test03	25,210,598	25,087,671	8,781
test07	1,249,978	549,999	49,999
test08	912,815,175	900,602,324	872,347
test09	6,199,856,654	6,138,748,383	4,364,877

*Os testes que estão ausentes na tabela resultaram em números muito grandes ou quebrados, impossibilitando a análise

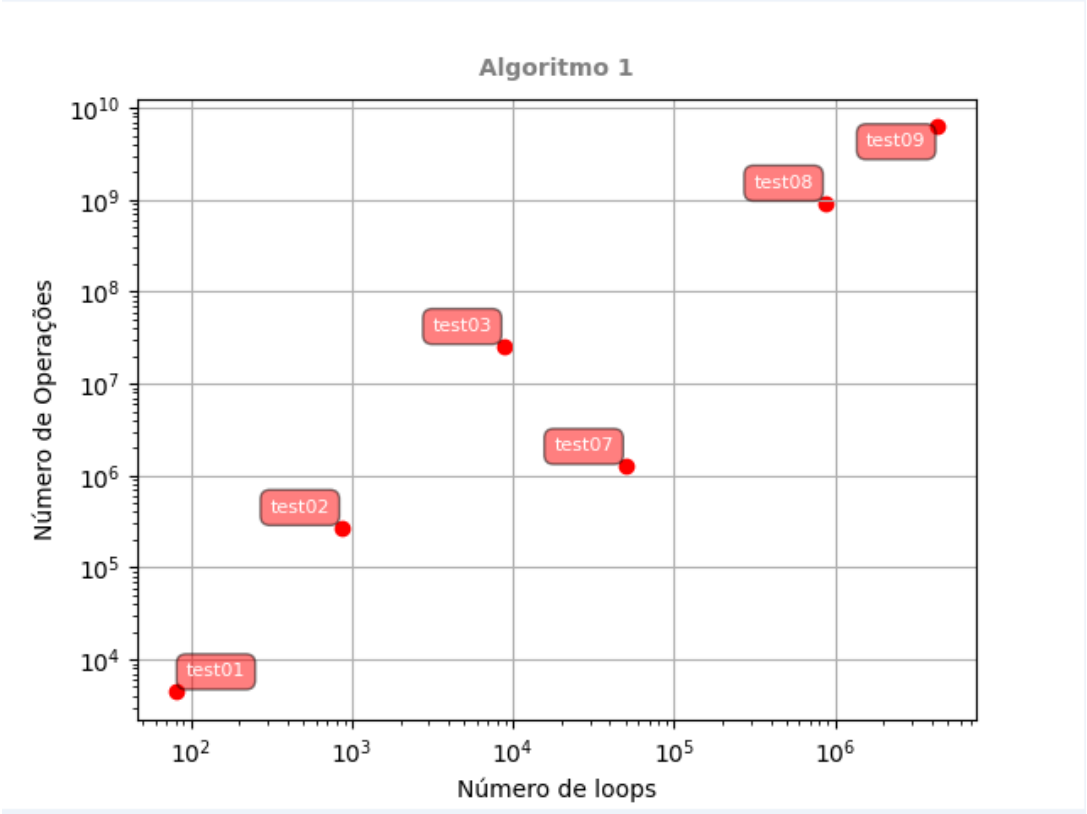

Gráfico que relaciona o número de loops e o número de operações no Algoritmo 1

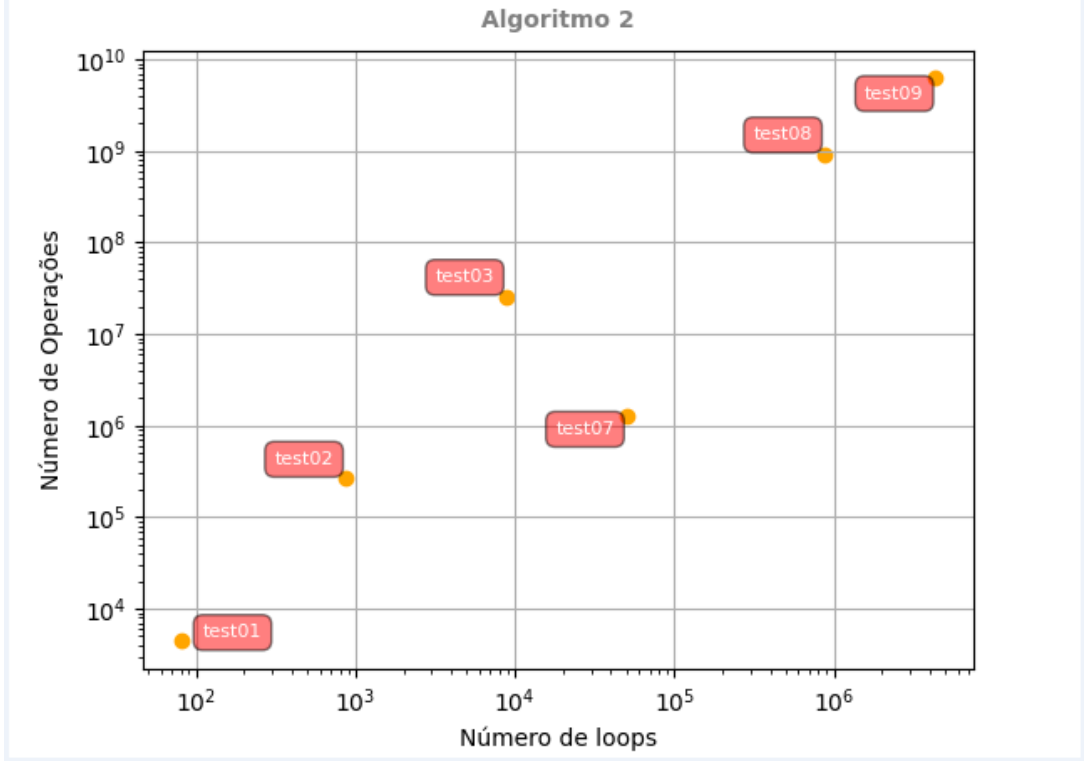

Gráfico que relaciona o número de loops e o número de operações no Algoritmo 2

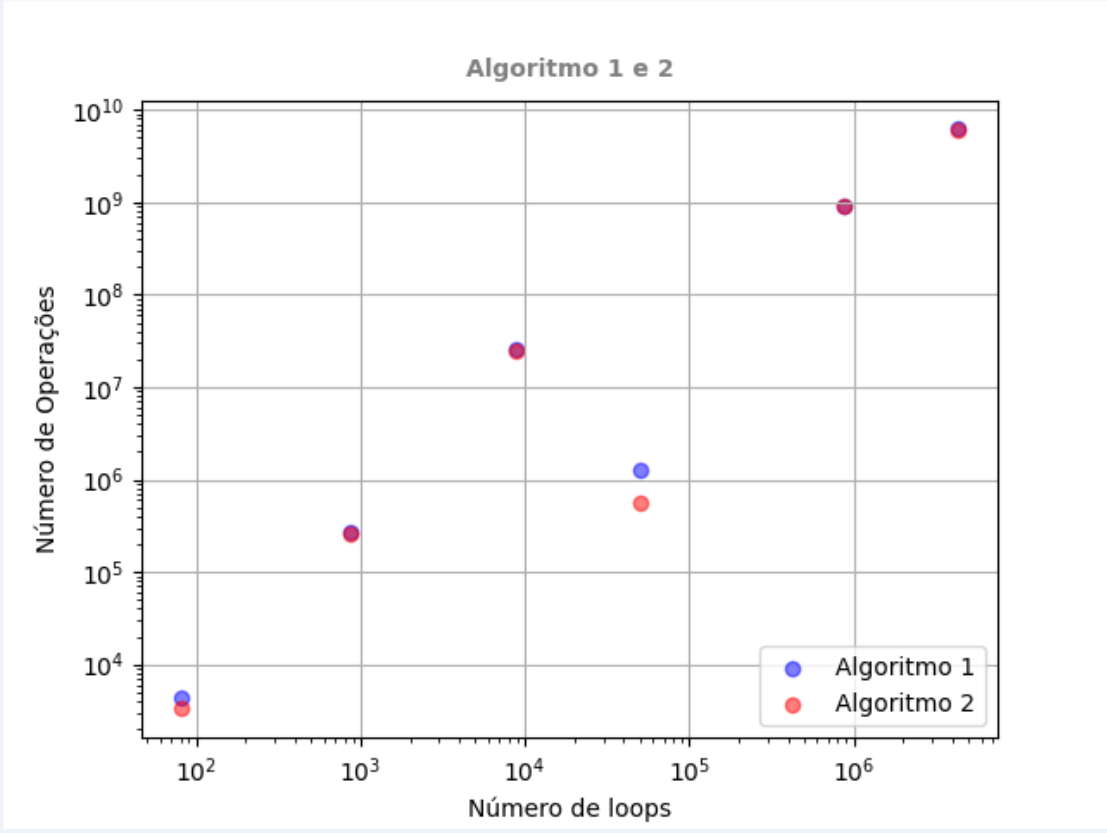

Gráfico que compara o Algoritmo 1 e o Algoritmo 2 em relação ao número de loops e o número de operações

A diferença entre os algoritmos é mínima, mas há uma diferença: o Algoritmo 2 requer menos operações para ser executado em comparação ao Algoritmo 1. O "test07" mostrou a maior diferença entre os dois: quando todas as letras são iguais, o segundo executa uma quantidade substancialmente reduzida de operações.

Conclusão

Para resolver o problema proposto, nós desenvolvemos três algoritmos diferentes, dois em C++ e um em Python. Após testes, determinamos que o programa em Python era insuficiente para resolver esse problema, por causa das *strings* serem imutáveis na linguagem. Para "burlar" essa limitação, uma solução recursiva foi criada, que em torno acabou consumindo uma quantidade enorme de memória do computador. Em situações extremas, como os casos de 2,500,000 e 5,000,000 letras, o programa consome toda a memória disponível, causando um erro de *memory leak*, ou dreno de memória em Português. Por causa desses fatores, determinamos o código em Python como inviável.

Os dois códigos em C++ seguiram uma lógica muito diferente, porém exibiram resultados quase idênticos, onde tivemos que examinar as menores margens para determinar qual era mais eficiente. Após análises, concluímos que o Algoritmo 2 contém a solução mais eficiente desenvolvida por nós, devido a quantidade de operações que ele executa em comparação ao Algoritmo 1. Em prática, essa diferença não afeta os resultados em termos de loops executados e tempo de execução, mas é uma diferença que deve ser considerada.

O Algoritmo 2 é um algoritmo eficiente quando se trata de strings que chegam até um certo limite (strings com caracteres de 1-5,000,000), onde ele consegue devolver a solução do problema em testes em um tempo de execução aceitável (< 10 minutos). Quando ele exerce esse limite, o algoritmo não aparenta

ser otimizado o suficiente para a resolução do problema (no caso de 10,000,000 de caracteres, o algoritmo demorou 40 minutos para resolver). Em situações de strings grandes, é necessário procurar uma solução mais viável. O Algoritmo apresentado serve para casos específicos, mas contém um limite que o torna ineficiente em situações com uma quantidade de caracteres maiores.

Exemplos de Execução

Nesta seção, os três algoritmos propostos serão destrinchados e terão seus funcionamentos demonstrados passo a passo com uma entrada padrão **DNANDANDANDANADNDDDAN**, cuja saída é **N**.

Algoritmo 1 — C++

```
1       string solution(string dna) {
2         for (int i = 0; i < dna.size() - 1; i++) {
3           if ((dna[i] == 'D' && dna[i+1] == 'N') || (dna[i] == 'N' && dna[i+1] == 'D')) {
4             dna += 'A';
5             dna.erase(i, 2);
6             i = i - 2;
7             if (i <= 0) i = -1;
8           }
9           if ((dna[i] == 'D' && dna[i+1] == 'A') || (dna[i] == 'A' && dna[i+1] == 'D')) {
10            dna += 'N';
11            dna.erase(i, 2);
12            i = i - 2;
13            if (i <= 0) i = -1;
14          }
15          if ((dna[i] == 'A' && dna[i+1] == 'N') || (dna[i] == 'N' && dna[i+1] == 'A')) {
16            dna += 'D';
17            dna.erase(i, 2);
18            i = i - 2;
19            if (i <= 0) i = -1;
20          }
21        }
22        return dna;
23      }
```

[0;i=0] A *string* é fornecida como entrada para a função `solution()` em uma chamada da mesma;

[1-2;i=0] A função `solution()` começa a ser executada, sendo **dna** a variável paramétrica cujo valor é o que foi passado na chamada da função; o laço **for** é iniciado com o index **i** valendo zero, o index incrementando em um após cada execução, e o laço continuando enquanto **i** for menor que o tamanho de **dna** menos um. É importante que o laço termine antes de um possível acesso a **dna[dna.**`size()`**]**, algo que ocorreria caso a condição fosse apenas o index ser menor que o tamanho da *string*;

[3;i=0] Um **if** avalia se os caracteres **dna[i]** e **dna[i+1]** constituem ou o par de letras "DN" ou o par "ND", e executa o trecho de código atrelado a ele se a comparação for verdadeira; com **i = 0**, o par sendo avaliado é as primeiras duas letras de **dna**, 'N' e 'D';

[4-8;i=0] O par "ND" retorna verdadeiro na comparação, então o interior do **if** é executado:

[4] ‘A’ é adicionado ao final de **dna**:

`DNANDANDANDANADNDDDAN → DNANDANDANDANADNDDDANA`

[5] O par avaliado é removido da *string*, e a string inteira é “puxada” para trás:

`DNANDANDANDANADNDDDANA → __ANDANDANDANADNDDDANA → ANDANDANDANADNDDDANA`

[6-7] **i** é decrementado por 2, para levar em conta a remoção de 2 caracteres, e caso esse decremento resulte em um número nulo ou negativo, -1 é atribuído a **i**, e não zero, uma vez que o index será incrementado no final de uma execução do laço;

[9-14;i=-1] Em seguida, as mesmas posições na *string* são avaliadas em outro **if** que determina se os caracteres **dna[i]** e **dna[i+1]** formam ou o par “DA” ou o par “AD”. Com o index valendo -1, essa comparação sempre é falsa;

[15-20;i=-1] Em seguida, as mesmas posições na *string* são avaliadas em outro **if** que determina se os caracteres **dna[i]** e **dna[i+1]** formam ou o par “AN” ou o par “NA”. Com o index valendo -1, essa comparação sempre é falsa;

[21;i=-1] Fim da execução do laço; 1 é somado ao index;

[2;i=0] Nova execução do laço; o index é menor que o tamanho da string, então a execução vai Adiante;

[3-8;i=0] Avaliação do par atual de letras **dna[i]** e **dna[i+1]** para determinar se o par formado é “ND” ou “DN”; se sim, executa o interior do **if**; senão, que é o caso uma vez que o par é “AN”, pula a execução do interior;

[9-14;i=0] Avaliação do par atual de letras **dna[i]** e **dna[i+1]** para determinar se o par formado é “DA” ou “AD”; a comparação é falsa, então segue adiante;

[15-20;i=-1] Avaliação do par atual de letras **dna[i]** e **dna[i+1]** para determinar se o par formado é “AN” ou “NA”; a comparação é verdadeira, então o interior do **if** é executado:

[16] ‘D’ é adicionado ao final de **dna**:

`ANDANDANDANADNDDDANA → DNANDANDANDANADNDDDANAD`

[17] O par avaliado é removido da *string*, e a string inteira é “puxada” para trás:

`ANDANDANDANADNDDDANAD → __DANDANDANADNDDDANAD → DANDANDANADNDDDANAD`

[18-19] **i** é decrementado por 2, e caso o decremento resulte em um número nulo ou negativo, -1 é atribuído ao index;

[21;i=-1] Fim da execução do laço; 1 é somado ao index;

Esse processo é repetido até que o index seja do tamanho da *string*, assim encerrando o **`for`** e retornando na linha 22 a *string* resultante.

Algoritmo 2 — Python

```
1       def alesti2_t1(string):
2         dictionary = {"AD": "N", "DA": "N", "DN": "A", "ND": "A", "AN": "D", "NA": "D"}
3         for i in range(len(string)):
4           if string[i:i+2] in dictionary:
5             newString = string[:i] + string[i+2:] + dictionary[string[i:i+2]]
6             return alesti2_t1(newString)
7         return string
```

[0] A *string* é fornecida como entrada para a função `alesti2_t1()` em uma chamada da mesma;

[1-2] A função `alesti2_t1()` começa a ser executada; a variável **`dictionary`** recebe um dicionário que mapeia pares de letras as letras correspondentes que devem ser adicionados ao final da *string*;

[3] Começo do laço **`for`**; o index **`i`** é inicializado em zero e será incrementado por um no fim de cada execução do laço, repetindo esse processo até que o index tenha o mesmo valor do comprimento (`len(string)`) da *string*;

[4] Condicional que compara se a *substring* que é formada por um corte da *string* a partir do index até duas letras depois (contando o index como a primeira) é uma chave para um valor contido no **`dictionary`**; se sim, o que está dentro da condicional é executado. Como o par formado pela *substring* é "DN", e "DN" é uma chave para um valor dentro do dicionário, a comparação é verdadeira;

```
DNANDANDANDANADNDDDAN → substring DN  → DN : A
^^
i i+2
```

[5] Uma nova variável **`newString`** recebe uma nova *string* formada pela *string* original com um 'A' adicionado ao final, menos o par de letras atual (no caso, "DN"):

```
newString ← ANDANDANDANADNDDDANA
```

[6] Essa nova *string* é usada como parâmetro para uma chamada recursiva de `alesti2_t1()`, onde o processo descrito da linha 1 à linha 6 é repetido até que nenhum par de letras dentro de **`dictionary`** seja encontrado dentro da *string* passada como parâmetro:

```
alesti2_t1(DNANDANDANDANADNDDDAN)
↪return alesti2_t1(ANDANDANDANADNDDDANA)
  ↪return alesti2_t1(DANDANDANADNDDDANAD)
    ↪return alesti2_t1(NDANDANADNDDDANADN)
      ↪return alesti2_t1(ANDANADNDDDANADNA)
              ...
          ↪return alesti2_t1(N)
             ↩return N
```

[7] Se nenhum par de letras foi encontrado na *string*, retorna a própria *string*, que é o que ocorre na última chamada recursiva;

Algoritmo 3 — C++ com *unordered_map*

```
1       string solution(string dna) {
2         unordered_map<string, char> dict = {{“AN”, ‘D’}, {“NA”, ‘D’}, {“DA”, ‘N’},
3                                             {“AD”, ‘N’}, {“DN”, ‘A’}, {“ND”, ‘A’}};
4         for (int i = 0; i < dna.size() - 1; i++) {
5           string str = dna.substr(i, 2);
6           auto chara_to_add = dict.find(str);
7           if (chara_to_add != dict.end()) {
8             dna.erase(i, 2);
9             dna += chara_to_add->second;
10            if (i <= 0) i = -1;
11          }
12        }
13        return dna;
14      }
```

[0] A *string* é fornecida como entrada para a função `solution()` em uma chamada da mesma;

[1-3] A função `solution()` começa a ser executada; a variável **`dict`** — um *unordered_map* — recebe um dicionário que mapeia pares de letras as letras correspondentes que devem ser adicionados ao final da *string*;

[4] O laço **`for`** é iniciado com o index **`i`** valendo zero, o index incrementando em um após cada execução, e o laço continuando enquanto **`i`** for menor que o tamanho de **`dna`** menos um. É importante que o laço termine antes de um possível acesso a **`dna[dna.`**`size()`**`]`**, algo que ocorreria caso a condição fosse apenas o index ser menor que o tamanho da *string*;

[5] Uma *substring* feita através da função-membro de *string* `substr()` é feita a partir da posição **`i`** até 2 letras depois (o index conta como a primeira) e guardada em **`str`**;

```
DNANDANDANDANADNDDDAN → substring DN  → str
^^
i i+2
```

[6] Se existir uma chave **`str`**, que no caso é “DN”, dentro de **`dict`**, o ponteiro para o par de chave-valor desta chave é salvo em **`chara_to_add`**; caso contrário, o ponteiro especial que indica o término de uma estrutura de dados é salvo;

[7-10] Se **`chara_to_add`** não apontar para o final de **`dict`**, a chave **`str`** está atrelada a um valor. Logo, como no primeiro algoritmo, é necessário eliminar o par de **`dna`**, adicionar a letra correspondente à *string*, e reduzir o index:

[8] O par atual “DN” é removido de dna, e o resto da string é “puxada” para trás:

```
DNANDANDANDANADNDDDAN → __ANDANDANDANADNDDDAN → ANDANDANDANADNDDDAN
```

[9] A letra chaveada ao par “DN”, ‘A’, é adicionada ao final de dna:

`ANDANDANDANADNDDDAN → ANDANDANDANADNDDDANA`

[10-11] **`i`** é decrementado por 2, e caso o decremento resulte em um número nulo ou negativo, `-1` é atribuído ao index;

Esse processo é repetido até que o index seja do tamanho da *string*, assim encerrando o **`for`** e retornando na linha 13 a *string* resultante.