1. Observe os algoritmos abaixo:

(a)	(b)	(c)
var N, r: Inteiro ler(N); r ← 0 **enquanto** $(r+1)^2 \le N$ **faça** r ← r+1 escrever(r)	**var** N, r: Inteiro ler(N); r ← N **enquanto** $r^2 > N$ **faça** r ← r-1 escrever(r)	**var** N, r, a, m: Inteiro ler(N); r ← 0; a ← N+1 **enquanto** r+1 ≠ a **faça** m ← (r+a) **div** 2 **se** $m^2 \le N$ **então** r ← m **senão** a ← m escrever(r)

a. Descreva sucinta e precisamente o problema que tais algoritmos resolvem.
b. Qual a complexidade de cada algoritmo?

a) O algoritmo (a) lê um valor inteiro N e inicializa uma variável r com 0. Em seguida, ele entra em um loop enquanto o quadrado de (r + 1) for menor ou igual a N. Em cada iteração do loop, o valor de r é incrementado em 1. Ao final de cada iteração, o valor de r é impresso na tela.

O algoritmo (b) lê um valor inteiro N e inicializa uma variável r com o valor de N. Em seguida, ele entra em um loop enquanto o quadrado de r for maior do que N. Em cada iteração do loop, o valor de r é decrementado em 1. Ao final de cada iteração, o valor de r é impresso na tela. O algoritmo (b) faz o que o algoritmo (a) faz, porém ao contrário.

O algoritmo (c) lê um valor inteiro N e inicializa as variáveis r com 0, a com N + 1 e m com o valor da média entre r e a, arredondada para baixo. Em seguida, ele entra em um loop enquanto r + 1 for diferente de a. Dentro do loop, o valor de m é atualizado para a média entre r e a novamente. Se o quadrado de m for menor ou igual a N, o valor de r é atualizado para m, caso contrário, o valor de a é atualizado para m. Ao final de cada iteração, o valor de r é impresso na tela.

b) A complexidade do algoritmo (a) é O(sqrt(N)), pois o loop é executado no máximo sqrt(N) vezes, uma vez que a condição de parada é (r + 1)^2 ≤ N.

A complexidade do algoritmo (b) é O(N), pois o loop pode ser executado até N vezes, caso o valor de N seja pequeno ou até mesmo maior do que N, caso o valor de N seja muito grande.

A complexidade do algoritmo (c) é O(log N), pois a cada iteração do loop, o intervalo de busca (representado pelas variáveis r e a) é dividido pela metade.

Isso resulta em um número máximo de iterações proporcional a log N, em vez de N, como no algoritmo (b).

2. Complete a função abaixo de modo que ela determine se o vetor B é um anagrama do vetor A, isto é, os elementos de B[1..n] consistem de uma permutação daqueles de A[1..n]. Em seguida, determine a complexidade de tempo da função elaborada.

```
função SãoAnagramas(A[]: Caractere, B[]: Caractere, N: Inteiro): Lógico
        //retornar verdadeiro se A[1..n] é anagrama de B[1..n]
       ?
```

```
função SãoAnagramas(A[]: Caractere, B[]: Caractere, N: Inteiro): Lógico

    se tamanho de A ≠ tamanho de B então

        retornar falso // os vetores têm tamanhos diferentes, não são anagramas

    fim se



    contarA ← array de tamanho 26 com todos os elementos inicializados para 0

    contarB ← array de tamanho 26 com todos os elementos inicializados para 0



    // contar a ocorrência de cada caractere em A e B

    para i de 1 até N faça

        indexA ← código ASCII do caractere A[i] - código ASCII de 'a' // obter o índice correspondente no array contarA

        indexB ← código ASCII do caractere B[i] - código ASCII de 'a' // obter o índice correspondente no array contarB

        contarA[indexA] ← contarA[indexA] + 1

        contarB[indexB] ← contarB[indexB] + 1

    fim para
```

```
    // comparar os arrays de contagem

    para i de 0 até 25 faça

        se contarA[i] ≠ contarB[i] então

            retornar falso // os vetores têm quantidades diferentes de cada caractere, não são anagramas

        fim se

    fim para



    retornar verdadeiro // os vetores têm as mesmas quantidades de cada caractere, são anagramas

fim função
```

A complexidade de tempo desta função é O(N), onde N é o tamanho dos vetores A e B. Isso ocorre porque a função percorre os vetores uma vez para contar a ocorrência de cada caractere em ambos os vetores e, em seguida, compara as contagens. As operações dentro dos loops têm uma quantidade constante de tempo, e a comparação dos arrays de contagem também leva tempo constante, já que o tamanho dos arrays é fixo em 26 elementos (representando as 26 letras do alfabeto). Portanto, a complexidade de tempo total é proporcional ao tamanho dos vetores, que é O(N).

3. Elabore os algoritmos abaixo para listas lineares **sequenciais não-ordenadas**. Determine a complexidade dos algoritmos elaborados. Considere a estrutura definida conforme abaixo:

Sequencial: `estrutura ListaLinear:`

```
E[1..1000]: Inteiro //elementos
N: Inteiro //número de elementos
```

a. `procedimento Altera(ref L: ListaLinear, x: Inteiro, y: Inteiro):`
   // substitui o elemento x na lista linear sequencial L por y, mantendo o novo
   // elemento na mesma posição do antigo

b. `função TodosDistintos(ref L: ListaLinear): Lógico`
   // retorna verdadeiro se os elementos de L são todos distintos

a) procedimento "Altera" que substitui o elemento x na lista linear sequencial L por y, mantendo o novo elemento na mesma posição do antigo:

```
procedimento Altera(ref L: ListaLinear, x: Inteiro, y: Inteiro):

  para i de 1 até L.N faça

    se L.E[i] = x então

      L.E[i] ← y // substitui o elemento x por y

      retornar // sai do loop após encontrar o elemento x

    fim se

  fim para

fim procedimento
```

A complexidade de tempo deste procedimento é O(N), onde N é o número de elementos na lista linear. Isso ocorre porque o procedimento percorre a lista linear sequencialmente em busca do elemento x para substituí-lo por y. O pior caso ocorre quando o elemento x está no final da lista ou não está presente na lista, exigindo que se percorra todos os N elementos da lista até encontrar o elemento ou chegar ao final da lista.

b) função "TodosDistintos" que verifica se os elementos da lista linear L são todos distintos:

```
função TodosDistintos(ref L: ListaLinear): Lógico

  para i de 1 até L.N faça

    para j de i + 1 até L.N faça

      se L.E[i] = L.E[j] então

        retornar falso // há elementos repetidos na lista

      fim se

    fim para
```

fim para

retornar verdadeiro // não há elementos repetidos na lista

fim função

A complexidade de tempo desta função é O(N^2), onde N é o número de elementos na lista linear. Isso ocorre porque a função utiliza dois loops aninhados para comparar todos os pares de elementos da lista linear, a fim de identificar se há elementos repetidos. O pior caso ocorre quando não há elementos repetidos na lista e é necessário comparar todos os pares de elementos, resultando em uma complexidade quadrática.

4. Elabore algoritmos para o exercício anterior, considerando agora que as listas são **encadeadas ordenadas**, com estrutura definida conforme abaixo:

Encadeada: `estrutura Nó:`

```
E: Inteiro //elemento
```

```
        Próx: ^Nó //próximo elemento
estrutura ListaLinear:
        Inicio: ^Nó //primeiro elemento
```

a) a. procedimento Altera(ref L: ListaLinear, x: Inteiro, y: Inteiro):

procedimento Altera(ref L: ListaLinear, x: Inteiro, y: Inteiro):

atual ← L.Inicio // aponta para o primeiro nó da lista

enquanto atual ≠ nulo faça // enquanto não chegar ao final da lista

se atual.E = x então // se o elemento do nó atual é igual a x

```
        atual.E ← y // substitui o elemento x por y

        retornar // sai do loop após encontrar o elemento x

      fim se

      atual ← atual.Próx // avança para o próximo nó da lista

  fim enquanto

fim procedimento
```

A complexidade de tempo deste procedimento é O(N), onde N é o número de nós na lista linear. Isso ocorre porque o procedimento percorre a lista linear encadeada em busca do nó com elemento x para substituí-lo por y. O pior caso ocorre quando o nó com elemento x está no final da lista ou não está presente na lista, exigindo que se percorra todos os N nós da lista até encontrar o nó ou chegar ao final da lista.

b) função TodosDistintos(ref L: ListaLinear): Lógico

```
função TodosDistintos(ref L: ListaLinear): Lógico

  atual ← L.Inicio // aponta para o primeiro nó da lista

  enquanto atual.Próx ≠ nulo faça // enquanto não chegar ao último nó da lista

      se atual.E = atual.Próx.E então // se o elemento do nó atual é igual ao elemento do próximo nó

        retornar falso // há elementos repetidos na lista

      fim se

      atual ← atual.Próx // avança para o próximo nó da lista

  fim enquanto



  retornar verdadeiro // não há elementos repetidos na lista

fim função
```

A complexidade de tempo desta função é O(N), onde N é o número de nós na lista linear. Isso ocorre porque a função percorre a lista linear encadeada comparando o elemento de cada nó com o elemento do próximo nó para identificar se há elementos repetidos. O pior caso ocorre quando não há elementos repetidos na lista e é necessário comparar todos os pares de nós, resultando em uma complexidade linear.