Exercício 1:

Proposta:

Dado um vetor A de tamanho N, com valores inteiros e um valor inteiro X, implemente um algoritmo de busca sequencial recursivo que retorna o índice do vetor caso X seja encontrado e -1 caso contŕario.

Algoritmo Implementado:

```
int searchValue(int x, int n, int a[]) {
    if(n <= 0) {
        return -1;
    }
    if(a[n-1] == x) {
        return n - 1;
    }
    return searchValue(x, --n, a);
}
```

Demonstração da complexidade no pior caso:

Seja T(n) a complexidade de tempo da função, onde n é o tamanho do vetor, pode-se escrever T(n) da seguinte forma:

$$T(n) = (c_1 + c_2) + T(n - 1) + ... + T(n - n)$$

Onde c_1 e c_2 são as constantes do tempo levado para executar os dois "if" da função. Podemos reescrever essa equação como:

$$T(n) = \sum_{i=0}^{n} (c_1 + c_2) = \sum_{i=0}^{n} (c_3)$$

No algoritmo implementado, n varia decrescentemente, de n até 0. Para facilitar o cálculo escrevi n variando crescentemente, mas isso não modifica a complexidade, já que resulta na mesma expressão final.

Removendo o somatório temos:

$$T(n) = c_3(n + 1) = O(n)$$

Logo, podemos concluir que a complexidade no pior caso é **linear**.

A versão iterativa deste algoritmo deve possuir a mesma complexidade temporal que sua versão recursiva, uma vez que o número de iterações no pior caso será o mesmo, a demonstração segue a mesma lógica.

Exercício 2:

Proposta:

Dado um vetor A de tamanho N, com valores inteiros e ordenado de forma crescente, e um valor inteiro X, implemente um algoritmo de busca binária na forma iterativa, que busca X e retorna seu índice.

Algoritmo implementado:

```
int searchValue(int x, int n, int a[]){
```

```
    int low = 0;
    int high = n;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (a[mid] == x) {
            return mid;
        } else if (a[mid] < x){
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}
```

Demonstração da complexidade:

Uma forma de analisar a complexidade temporal $T(n)$ do algoritmo, é notando que, a cada iteração do laço while, o tamanho do escopo analisado é metade da iteração anterior. Para melhor visualização, podemos escrever a complexidade da versão recursiva:

$$T(n) = T(\frac{n}{2}) + c$$

Onde c é a constante do tempo que leva para a execução de todas operações. Ao expandir mais a função, temos:

$$T(\frac{n}{2}) = T(\frac{n}{4}) + c \Rightarrow T(n) = T(\frac{n}{4}) + 2c$$

$$T(\frac{n}{4}) = T(\frac{n}{8}) + c \Rightarrow T(\frac{n}{2}) = T(\frac{n}{8}) + 2c \Rightarrow T(n) = T(\frac{n}{8}) + 3c$$

Note que podemos reescrever a função com denominador n e numerador de potências de 2, ao passo que muda-se o coeficiente de c. Então, podemos escrever:

$$T(n) = T(\frac{n}{2^k}) + kc$$

Eventualmente, quando $T(\frac{n}{2^k})$, restará apenas um valor no array, então:

$$T(\frac{n}{2^k}) = T(1) \Rightarrow \frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow log_2 n = k$$

Substituindo os valores de k e $T(\frac{n}{2^k})$ na função inicial, temos:

$$T(n) = T(1) + log_2 n \cdot c$$

Como $T(1)$ não depende de variáveis, pode-se substituir por uma constante t:

$$T(n) = t + log_2 n \cdot c = O(log_2 n)$$

Dessa forma, podemos concluir que a complexidade da função (em sua forma recursiva) é **logarítmica**.

Como o valor de n varia igualmente na versão iterativa, podemos concluir que a complexidade temporal no pior caso é a mesma que a de sua versão recursiva. Em ambas as versões (recursiva e iterativa), a complexidade no melhor caso também será igual, visto que será encerrada na primeira iteração, a complexidade será $\Omega(1)$, ou seja, **constante**.

Exercício 3:

Proposta:

Dado um vetor A de tamanho N, com valores inteiros, implemente um algoritmo que verifica se o vetor está ordenado de forma crescente.

Algoritmo implementado:

```
bool isSorted(int n, int a[]) {
    int i = 1;
    while (i < n) {
        if (a[i] < a[i-1]){
            return false;
        }
        i++;
    }
    return true;
}
```

Demonstração da complexidade:

A complexidade dessa função, tanto no pior quanto no melhor caso, é similar a complexidade da função *searchValue* do exercício 1.

No melhor caso, ambos têm complexidade **constante** ($\Omega(1)$), visto que serão encerradas após apenas uma iteração.

Já no pior caso, há apenas uma diferença. Enquanto na função do exercício 1, o valor varia de n à 0, nesta função a variação é de 1 à n, podendo ser escrita desta forma:

$$T(n) = c(n) = O(n)$$

No entanto, isso não afeta a complexidade, uma vez que o valor de constantes é ignorado de qualquer forma, portanto, a complexidade é **linear**.

Exercício 4:

Proposta:

Dado um vetor A de tamanho N, com valores inteiros, e um valor inteiro X, implemente o algoritmo iterativo de fibonacci. De modo a retornar o valor inteiro X na posição N da sequência de fibonacci.

Algoritmo implementado:

```
int fibonacci(int n){
    int current = 0;
    int previous = 1;
    int i = 2;
    while(i <= n + 1) {
        int temp = current + previous;
        previous = current;
        current = temp;
        i++;
    }
```

```
    return current;
}
```

Demonstração da complexidade no pior caso:

Nesta versão de fibonacci iterativo, a complexidade é parecida com as dos exercícios 1 e 3. Como o valor de i (variável que serve como contador) começa em 2 e varia até $n + 1$, podemos escrever a função da seguinte forma:

$$T(n) = \sum_{i=2}^{n+1} (c_1) + c_2$$

Onde c_1 e c_2 são constantes de tempo (referentes às execuções dentro do laço while e fora dele respectivamente). Dessa forma, aplicando as mesmas propriedades do somatório que foram usadas no exercício 1, temos:

$$T(n) = c_1((n + 1) + 1 - 2) + c_2$$

$$T(n) = c_1(n) + c_2 = O(n)$$

Logo, podemos concluir que a complexidade é **linear**.

Comparação com fibonacci recursivo:

```
int fibonacci(int n) {
  int x;

  if (n <= 1) {
    return(1);
  }

  x = fibonacci(n-1) + fibonacci(n-2);
  return(x);
}
```

Por outro lado, o fibonacci recursivo apresentado pelo professor tem complexidade diferente, podemos escrever a função da seguinte forma:

$$T(n) = T(n - 1) + T(n - 2) + c$$

Expandindo as funções T, temos:

$$T(n) = (T(n - 2) + T(n - 3) + c) + (T(n - 3) + T(n - 4) + c) + c$$

$$T(n) = T(n - 2) + 2\,T(n - 3) + T(n - 4) + 3c$$

E isso continuaria até que $T(n - k) \leq 1$. Nota-se que a função é **exponencial**, diferente de sua versão iterativa, que tem complexidade menor.