Exercício 1:

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

Demonstração da complexidade no pior caso:
Seja T(n) a complexidade de tempo da função, onde n é o tamanho do vetor, existem 3 processos sendo executados na função. Os casos base são ambos de complexidade constante, já que serão executados apenas uma vez a cada iteração. Já a chamada da recursividade será chamada n vezes no pior caso, já que terá percorrido todo o array, então sua complexidade é O(n).
Somando as complexidades das 3 operações, temos: T(n) = O(1) + O(1) + O(n), omitindo as complexidades constantes, ficamos com: T(n) = O(n), que é a complexidade da função.
No pior caso, a complexidade é a mesma da versão iterativa pois será iterada n vezes.

Exercício 2:

```
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 no pior caso:
Mais uma vez, a função é dividida em 3 processos, sendo dois deles (a inicialização das variáveis e a verificação do valor encontrado) constantes. O loop while de busca binária a cada iteração diminui pela metade a faixa de busca, no pior caso o número de iterações necessárias será log de n na base 2, logo a complexidade do pior caso é O(log n).
Somando as complexidades, temos: T(n) = O(1) + O(log n) + O(1), omitindo as complexidades constantes, ficamos com a complexidade T(n) = O(log n).
Demonstração da complexidade no melhor caso:
No melhor caso, já na primeira iteração o valor de x será encontrado, que é quando esse valor estiver na posição do meio do vetor (n/2), então a complexidade é constante Ω(1).
Comparação com a versão recursiva:

A complexidade temporal, tanto no melhor quanto no pior caso, será a mesma, uma vez que o número de iterações necessárias será o mesmo. Contudo, a complexidade espacial da versão recursiva será pior, já que a cada iteração um novo processo será adicionado à pilha, consumindo mais espaço.

Exercício 3:

```
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 no pior caso:
Ignorando as execuções constantes, no pior caso o loop while será iterado n-1 vezes, pois terá que percorrer todo o array (começando em na posição 1, e não 0, por isso n-1) para determinar se está totalmente ordenado e retorna um booleano verdadeiro. Então, já omitindo as execuções constantes (declaração de variáveis e verificações if), a complexidade será $O(n)$.
Demonstração da complexidade no melhor caso:
No melhor caso, já na primeira iteração o algoritmo vai retornar um booleano falso caso a[1] seja menor que a[0], então a complexidade será constante $\Omega(1)$.

Exercício 4:

```
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:
Mais uma vez ignorando as execuções constantes, o loop while sempre será iterado n vezes (uma vez que começa com i = 2 e será iterado enquanto i<=n+1), então a complexidade é $O(n)$.

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 de O(2^n), já que a cada iteração a recursão será chamada 2 vezes, criando uma árvore recursiva que aumenta a complexidade do algoritmo exponencialmente. Sendo mais complexo que a versão iterativa, que tem complexidade de O(n).