PROIECT T.A.S.

Implementare teste automate

Se vor crea minim 15 teste automate in GTest (sau un tool similar) astfel incat:

- Sa se asigure 100% acoperire pe cod (decision coverage)
- Se vor folosi stub-uri si drivere
- Se vor folosi diferite metode de testare
  - Teste positive/negative
  - Clase de echivalenta/boundary test

- Decision coverage măsoară procentajul de ramuri condiționale evaluate atât pentru true, cât și pentru false.

(Numarul de decizii evaluate pentru ambele ramuri (True/False)/Numarul total de decizii)*100

1. Functia f (cautarea binara)

```
if (p1 > p2)        // Decizia 1
   return -1;
else {              // Decizia 2
   if (v[mijloc] == x)     // Decizia 3
      return mijloc;
   else if (x < v[mijloc]) // Decizia 4
      return f(v, p1, mijloc - 1, x);
   else                    // Decizia 5
      return f(v, mijloc + 1, p2, x);
}
```

Deciziile:

1. `p1 > p2` → Testată în:
   a. `EmptySearchSpace` (True)
   b. Alte teste (False)

2. `else` → Această ramură este implicit testată dacă `p1 > p2` este False.
3. `v[mijloc] == x` → Testată în:
    a. `PositiveValueExists` (True)
    b. `ValueDoesNotExist` (False)
4. `x < v[mijloc]` → Testată în:
    a. `BoundaryConditions` (True și False)
5. `else` → Testată în cazurile când `x > v[mijloc]`.

Număr total decizii: 5

Număr decizii testate pentru True/False: 5

Decision Coverage pentru `f`:

(5/5)*100=100%

2. Functia sortAsc (sortare crescatoare)

```
for (i = 0; i < n - 1; i++) {        // Decizia 1
   for (j = i + 1; j < n; j++) {     // Decizia 2
      if (v[i] > v[j]) {             // Decizia 3
         temp = v[i];
         v[i] = v[j];
         v[j] = temp;
      }
   }
}
```

Deciziile:

1. `i < n - 1` → Testată în:
    a. Toate testele de sortare (True și False).
2. `j < n` → Testată în:
    a. Toate testele de sortare (True și False).
3. `v[i] > v[j]` → Testată în:
    a. `ReverseSortedArray` (True)
    b. `AlreadySortedArray` (False)

Număr total decizii: 3

Număr decizii testate pentru True/False: 3

Decision Coverage pentru `sortAsc`:

(3/3)*100=100%

3. Functia sortDesc(sortare descrescatoare)

```
for (i = 0; i < n - 1; i++) {        // Decizia 1
   for (j = i + 1; j < n; j++) {     // Decizia 2
      if (v[i] < v[j]) {             // Decizia 3
         temp = v[i];
         v[i] = v[j];
         v[j] = temp;
      }
   }
}
```

Deciziile:

1. `i < n - 1` → Testată în:
   a. Toate testele de sortare (True și False).
2. `j < n` → Testată în:
   a. Toate testele de sortare (True și False).
3. `v[i] < v[j]` → Testată în:
   a. `SortDescending` (True)
   b. `AlreadySortedArray` (False)

Număr total decizii: 3

Număr decizii testate pentru True/False: 3

Decision Coverage pentru `sortDesc`:

(3/3)*100=100%

4. Functia main

Total decizii:

- Funcția `f`: 5 decizii.
- Funcția `sortAsc`: 3 decizii.
- Funcția `sortDesc`: 3 decizii.

Total: 5+3+3=11

Decizii testate pentru True/False:

Toate cele 11 decizii au fost testate pentru ambele valori.

Decision Coverage total:

(11/11)*100=100%

- Un **stub** este o implementare minimă a unei funcții, utilizată pentru a simula comportamentul acesteia atunci când funcția reală:

- Nu este implementată.
- Este prea complexă pentru a fi utilizată direct în teste.

```
int stub_f(int v[], int p1, int p2, int x) {
   return f(v, p1, p2, x);
}
```

- Un **driver** este o funcție sau un modul special creat pentru a:

- Apela și testa funcțiile unei alte componente (unitate de cod) izolate.
- Furniza date de intrare și verifica dacă rezultatele obținute sunt corecte.

```
void driver_sortAsc(int v[], int n) {
   sortAsc(v, n);}
 void driver_sortDesc(int v[], int n) {
   sortDesc(v, n);}
```

-Test negativ

```
TEST(BinarySearchTests, ValueDoesNotExist)
{ int v[] = { 1, 2, 3, 4, 5 };
int pos = stub_f(v, 0, 4, 6);
EXPECT_EQ(pos, -1); // Test negativ: elementul "6" nu există.
 }
```

```
TEST(SortTests, SortEmptyArray)
{
    int* v = nullptr;
    driver_sortAsc(v, 0);
    EXPECT_EQ(v, nullptr); // Test negativ: vector gol.
}

TEST(SortTests, SortSingleElement)
{
    int v[] = { 1 };
    driver_sortAsc(v, 1);
    EXPECT_EQ(v[0], 1); // Test pozitiv pentru caz limită: vector cu un singur element.
}
```

-Test pozitiv

```
TEST(BinarySearchTests, ValueExists) { int v[] = { 1, 2, 3, 4, 5 };
 int pos = stub_f(v, 0, 4, 3);
 EXPECT_EQ(pos, 2); // Test pozitiv: elementul "3" există în vector.
}
TEST(SortTests, SortAscending)
{
    int v[] = { 5, 4, 3, 2, 1 };
    driver_sortAsc(v, 5);
    for (int i = 0; i < 5; ++i) {
        EXPECT_EQ(v[i], i + 1); // Test pozitiv: sortare crescătoare.
    }
}
```

```
TEST(SortTests, SortDescending)
{
    int v[] = { 1, 2, 3, 4, 5 };
    driver_sortDesc(v, 5);
    for (int i = 0; i < 5; ++i) {
        EXPECT_EQ(v[i], 5 - i); // Test pozitiv: sortare descrescătoare.
    }
}
```

-Test de boundary

```
TEST(BinarySearchTests, BoundaryConditions)
{ int v[] = { 1,2,3,4,5 };
 ASSERT_EQ(f(v, 0, 4, 1), 0);
 ASSERT_EQ(f(v, 0, 4, 5), 4);
}
```

-Test de echivalenta

```
TEST(BinarySearchTests, AllElementsIdentical) {
    int v[] = { 3, 3, 3, 3, 3 };
    int pos = stub_f(v, 0, 4, 3);
    EXPECT_GE(pos, 0); // Orice poziție este corectă.
}
```