Ficha 1 (Frama-C)

Exercício 1

O contrato para a função **change** é:

```
/*@ requires \valid(a) && \valid(b);
    ensures a == \old(b) && b == \old(a);
    assigns *a, *b;
*/
```

```
void change(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Aqui está a função **main** que chama a função anterior e a verifica (assert):

```
int main(void){
    int x = 10;
    int y = 5;
    change(&x,&y);
    //@ assert x == 5 && y == 10;
}
```

Exercício 2

```
/*@ requires N >= 0;
    requires \valid(A+(0..N-1));
    ensures \result != <==> \forall integer i; 0 <= i < N ==> A[i] >= 0;
    assigns \nothing;
*/
```

```
int negs(int A[], int N){
    int k;
    for(k = 0; k < N; k++){
        if (A[k] < 0) return 0;
    }
    return 1;
}
```

Atendendo a seguinte função **main**, acrescente as anotações necessárias para completar a prova com sucesso:

- O que falta são os invariantes de ciclo (loop invariant).

```
void main(void){
    int a[10];
    int b[5] = {3,4,8,3,7};
    int c[5] = {2,4,-28,3,-17};
    int i;

    i = negs(b,5);
    //@ assert i != 0;
    //@ assert i == 1;

    i = negs(c,5);
    //@ assert i == 0;

    /*@ loop invariant 0 <= i <= 10;
        loop invariant \forall integer j; 0 <= j < i ==> a[j] >= 0;
    */
    for(i=0; i<10; i++)
        a[i] = i*3;

    i = negs(a,10);
    //@ assert i != 0;
}
```

Exercício 3

- Segurança:

A pré-condição **requires N >= 0** garante que o tamanho do array seja válido e não negativo.
A pré-condição **requires \valid(A + (0 .. N-1))** garante que o array A seja válido na faixa de índices especificada.
A cláusula **assigns \nothing** garante que a função não modifica nenhum valor fora de seus parâmetros e variáveis locais.

- Correção funcional:

A cláusula **ensures 0 <= \result < N** garante que o resultado retornado esteja dentro dos limites válidos do array.
A cláusula **ensures \forall integer k; 0 <= k < N ==> A[k] >= A[\result]** garante que o valor no índice retornado seja o menor valor do array.

```
/*@ requires N >= 0;
    requires \valid(A+(0..N-1));
    ensures 0 <= \result < N;
```

```
    ensures \forall integer k; 0 <= k < N ==> A[k] >= A[\result];
    assigns \nothing;
*/


  int minarray(int A[], int N){

      if (N == 0) {
          return -1;
      }

      int minIndex = 0;

      /*@ loop invariant 0 < i < N;
          loop invariant \forall integer j; 1 <= j < i ==> a[j] < a[minIndex];
      */
      for (int i = 1; i < N; i++) {
          if (A[i] < A[minIndex]) {
              minIndex = i;
          }
      }

      return minIndex;

  }



  int main() {
      int array[] = {5, 2, 8, 3, 1};
      int size = sizeof(array) / sizeof(array[0]);

      int minIndex = minarray(array, size);

      printf("Índice do menor valor: %d\n", minIndex);
      printf("Menor valor: %d\n", array[minIndex]);

      //@ assert 0 <= minIndex < size;
      //@ assert \forall integer k; 0 <= k < size ==> array[k] >= array[minIndex];

      return 0;
  }
```

Exercício 4

```
/*@ requires N >= 0;
    requires \valid(A+(0..N-1)) && \valid(B+(0..N-1));
    ensures \result != 0 <==> forall integer i; a <= i <= b ==> A[i] == B[i];
    assigns \nothing;
*/
```

```
int equal_seg(int A[], int B[], int a, int b, int N){

    //@ loop invariant a <= i <= b+1;
    //@ loop invariant \forall integer j; a <= j <= i ==> A[j] == B[j];
    //@ loop assigns i;
    //@ loop variant v-i+1;
    for (i = a; i <= b+1; i++) {
        if (A[i] != B[i])
            return false;
    }

    return true;
}



void main(void){
    int a[10] = {4,3,5,3,2,-5,-7,12,6,21};
    int b[8] = {2,1,3,3,2,.5,-7,9};

    x = equal_sign(a,b,3,6,8);
    //@ assert x != 0;

    x = equal_sign(a,b,1,5,8);
    //@ assert x != 0;
}
```

Exercício 5

```
/*@
   predicate belongs(int x, int A[], int n) =
       \exists integer i; 0 <= i < n && A[i] == x;
*/


int where(int A[], int N, int x)
    //@ requires \valid(A+(0..N-1));
    //@ ensures \result == -1 || (0 <= \result < N && A[\result] == x);
    {
    for (int i = 0; i < N; i++)
        //@ loop invariant 0 <= i <= N;
        //@ loop invariant \forall integer j; 0 <= j < i ==> A[j] != x;
        //@ loop assigns i;
    {
        if (A[i] == x)
        return i;
    }
    return -1;
}
```

```
int main()
{
    int A[] = {1, 3, 5, 7, 9};
    int N = sizeof(A) / sizeof(A[0]);
    int x = 5;

    int result = where(A, N, x);

    //@ assert result == 2;

    return 0;
}
```

Ficha 2 (Frama-C)

Exercício 1

Recorde os predicados **Sorted** e **Swap**, definidos na aula e os contratos das funções maxarray e swap:

- Sorted{List}(Array, start_index, end_index)

```
/*@ predicate Sorted{L}(int *t,integer i,integer j) =
    \forall integer k; i<=k<j ==> \at(t[k],L) <= \at(t[k+1],L);
*/
```

- Swap{List1, List2}(Array, start_index, end_index)

```
/*@ predicate Swap{L1,L2}(int *a, integer i, integer j) =
        \at(a[i],L1) == \at(a[j],L2)
            && \at(a[i],L2) == \at(a[j],L1)
            && \forall integer k;  k!=i && k!=j ==> \at(a[k] ,L1) == \at(a[k] ,L2);
*/
```

- Permut

```
inductive Permut{L1,L2}(int *a, integer l, integer h) {
    case Permut_refl{L}:
       \forall int *a, integer l, h; Permut{L,L}(a, l, h) ;
    case Permut_sym{L1,L2}: \forall int *a, integer l, h;
        Permut{L1,L2}(a, l, h) ==> Permut{L2,L1}(a, l, h) ;
    case Permut_trans{L1,L2,L3}:
        \forall int *a, integer l, h; Permut{L1,L2}(a, l, h) && Permut{L2,L3}(a, l, h)
                                        ==> Permut{L1,L3}(a, l, h) ;
    case Permut_swap{L1,L2}:
        \forall int *a, integer l, h, i, j;
```

```
              l <= i <= h && l <= j <= h && Swap{L1,L2}(a, i, j) ==> Permut{L1,L2}(a, l,
  }



  /*@
      requires 0 < size && \valid(u+ (0..size-1));
      ensures 0 <= \result < size;
      ensures \forall integer a; 0 <= a < size ==> u[a] <= u[\result];
      assigns \nothing;
  */
  int maxarray(int u[], int size);

  /*@
      requires \valid(t+i) && \valid(t+j);
      ensures Swap{Old,Here}(t,i,j);
      assigns t[i], t[j];
  */
  void swap(int t[],int i,int j);


/*@
    requires \valid(a+(0..size-1));
    requires size > 0;
    ensures Sorted{Here}(a,0,size-1);
    assigns a[0..size-1];
*/


  void maxSort (int \*a, int size) {
  int i, j;

      /*@
          loop variant i;
          loop invariant 0 <= i < size;
          loop invariant Sorted{Here}(a,i,size-1);
          loop invariant i < size-1 ==> \forall integer k; 0 <= k <= i ==> a[k] <= a[i+1
          loop assigns i,j,*(a+(0..size-1));
      */
      for (i=size -1; i>0; i--) {
          j = maxarray(a,i+1);
          swap(a,i,j);
      }

  }
```

Exercício 2

```
  /*@
  axiomatic CountAxiomatic {
```

```
  logic integer Count(int* a, integer n, int v) ;

  axiom CountEmpty: \forall int *a, v, integer n; n <= 0 ==> Count(a, n, v) == 0;

  axiom CountOneHit: \forall int *a, v, integer n;
              a[n] == v ==> Count(a, n + 1, v) == Count(a, n, v) + 1;

  axiom CountOneMiss: \forall int *a, v, integer n;
              a[n] != v ==> Count(a, n + 1, v) == Count(a, n, v);
  }
  */


/*@
    requires n >= 0 && \valid(a+(0..n+1));
    ensures \result == Count(a,n,val);
    ensures 0 <= \result <= n;
    assigns \nothing;
*/


  int numOccur(const int* a, int n, int val) {
      int counted = 0;

      /*@ loop invariant 0 <= i <= n;
          loop invariant counted == Count(a,i,val);
          loop variant n-1;
          loop assigns i, counted;
      */
      for (int i = 0; i < n; ++i) {
          if (a[i] == val) {
              counted++;
          }
      }
      return counted;
  }
```

Exercício 3

```
/*@
    requires n >= 0 && \valid(a+(0..n/2));
    ensures \forall integer i; 0 <= i < n ==> \old(A[i]) == A[n-1-i] && \old(A[n-1-i]) == A[i];
    assigns A[0..n-1];
*/


  void reverse (int A[], int n) {
      int i, x;
```

```
    /*@ loop invariant 0 <= i <= n/2;
        loop invariant \forall integer j; 0 <= j < i ==> \old(A[j]) == A[n-1-j] && \ol
        loop variant n-1-i;
        loop assigns i, A[0..n-1];
    */
    for (i=0; i<n/2; i++) {
        x = A[i];
        A[i] = A[n-1-i];
        A[n-1-i] = x;
    }
}
```