Programación 2

Diseño recursivo de algoritmos por inmersión

Problemas 04

Problema 1. Diseñar sin bucles:

```
/*
 * Pre: (EX alfa EN [0,n-1]. v[alfa] > 0.0)  AND  n <= #v
 * Post: posicionPrimerPositivo(v,n) = PPP  AND
 *       PPP >= 0  AND  PPP <= n-1  AND  v[PPP] > 0.0  AND
 *       (PT alfa EN [0,PPP-1]. v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n);
```

Se propone intentar dos diseños diferentes:

1. Aplicando una inmersión mediante debilitamiento de la postcondición de **`posicionPrimerPositivo`**`(v,n)`
2. Aplicando una inmersión mediante refuerzo de la precondición de **`posicionPrimerPositivo`**`(v,n)`

Se va a diseñar, sin bucles, la función **`posicionPrimerPositivo`**`(v,n)` programando una función auxiliar **`posicionPrimerPositivo`**`(v,n,desde)` cuya postcondición se plantea como un **debilitamiento de la postcondición** de la primera función.

```
/*
 * Pre: (EX alfa EN [0,n-1]. v[alfa] > 0.0)  AND  n <= #v
 * Post: posicionPrimerPositivo(v,n) = PPP  AND
 *       PPP >= 0  AND  PPP <= n-1  AND  v[PPP] > 0.0  AND
 *       (PT alfa EN [0,PPP-1].v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n) {
   ???
}
```

```
/*
 * Pre: (EX alfa EN [desde,n-1]. v[alfa] > 0.0)  AND  n <= #v  AND
 *      desde >= 0
 * Post: posicionPrimerPositivo(v,n,desde) = PPP  AND
 *       PPP >= ???  AND  PPP <= n-1  AND  v[PPP] > 0.0  AND
 *       (PT alfa EN [???,PPP-1]. v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n,
                            const int desde);
```

```
/*
 * Pre: (EX alfa EN [0,n-1]. v[alfa] > 0.0)  AND  n <= #v
 * Post: posicionPrimerPositivo(v,n) = PPP AND
 *       PPP >= 0 AND PPP <= n-1 AND v[PPP] > 0.0  AND
 *       (PT alfa EN [0,PPP-1].v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n) {
   ???
}
```

```
/*
 * Pre: (EX alfa EN [desde,n-1]. v[alfa] > 0.0)  AND  n <= #v  AND
 *      desde >= 0
 * Post: posicionPrimerPositivo(v,n,desde) = PPP  AND
 *       PPP >= desde  AND  PPP <= n-1  AND  v[PPP] > 0.0  AND
 *       (PT alfa EN [desde,PPP-1]. v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n,
                            const int desde);
```

```
/*
 * Pre: (EX alfa EN [0,n-1]. v[alfa] > 0.0)  AND  n <= #v
 * Post: posicionPrimerPositivo(v,n) = PPP  AND
 *       PPP >= 0  AND  PPP <= n-1  AND  v[PPP] > 0.0  AND
 *       (PT alfa EN [0,PPP-1].v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n) {
   return posicionPrimerPositivo(v, n, 0);
}
```

```
/*
 * Pre: (EX alfa EN [desde,n-1]. v[alfa] > 0.0)  AND  n <= #v  AND
 *      desde >= 0
 * Post: posicionPrimerPositivo(v,n,desde) = PPP  AND
 *       PPP >= desde  AND  PPP <= n-1  AND  v[PPP] > 0.0  AND
 *       (PT alfa EN [desde,PPP-1]. v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n,
                            const int desde);
```

```
/*
 * Pre: (EX alfa EN [desde,n-1]. v[alfa] > 0.0)  AND  n <= #v  AND
 *      desde >= 0
 * Post: posicionPrimerPositivo(v,n,desde) = PPP  AND
 *       PPP >= desde  AND  PPP <= n-1  AND  v[PPP] > 0.0  AND
 *       (PT alfa EN [desde,PPP-1]. v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n,
                            const int desde) {
   if (v[desde] > 0.0) {
      return desde;
   }
   else {
      // (EX alfa EN [desde+1,n-1]. v[alfa] > 0.0) AND
      // desde + 1 >= 0
      return posicionPrimerPositivo(v, n, desde+1);
   }
}
```

Se va a abordar un segundo diseño, también sin bucles, de la función **`posicionPrimerPositivo`**`(v,n)` programando una función auxiliar **`posicionPrimerPositivo`**`(v,n,desde)` cuya precondición se plantea como un **refuerzo de la precondición** de la primera función.

```
/*
 * Pre: (EX alfa EN [0,n-1]. v[alfa] > 0.0)  AND  n <= #v
 * Post: posicionPrimerPositivo(v,n) = PPP  AND
 *       PPP >= 0  AND  PPP <= n-1 AND v[PPP] > 0.0  AND
 *       (PT alfa EN [0,PPP-1]. v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n) {
   ???
}
```

```
/*
 * Pre: (EX alfa EN [0,n-1]. v[alfa] > 0.0)  AND  n <= #v  AND
 *      desde >= 0  AND  ???
 * Post: posicionPrimerPositivo(v,n,desde) = PPP  AND
 *       PPP >= 0  AND  PPP <= n-1 AND v[PPP] > 0.0  AND
 *       (PT alfa EN [0,PPP-1]. v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n,
                            const int desde);
```

```
/*
 * Pre: (EX alfa EN [0,n-1]. v[alfa] > 0.0)  AND  n <= #v
 * Post: posicionPrimerPositivo(v,n) = PPP  AND
 *       PPP >= 0  AND  PPP <= n-1  AND  v[PPP] > 0.0  AND
 *       (PT alfa EN [0,PPP-1]. v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n) {
   ???
}
```

```
/*
 * Pre: (EX alfa EN [0,n-1]. v[alfa] > 0.0)  AND  n <= #v  AND
 *      desde >= 0  AND  (PT alfa EN [0,desde-1]. v[alfa] <= 0.0)
 * Post: posicionPrimerPositivo(v,n,desde) = PPP  AND
 *       PPP >= 0  AND  PPP <= n-1  AND v[PPP] > 0.0  AND
 *       (PT alfa EN [0,PPP-1]. v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n,
                            const int desde);
```

```
/*
 * Pre: (EX alfa EN [0,n-1]. v[alfa] > 0.0)  AND  n <= #v
 * Post: posicionPrimerPositivo(v,n) = PPP  AND
 *       PPP >= 0  AND  PPP <= n-1  AND  v[PPP] > 0.0  AND
 *       (PT alfa EN [0,PPP-1]. v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n) {
    return posicionPrimerPositivo(v, n, 0);
}
```

```
/*
 * Pre: (EX alfa EN [0,n-1]. v[alfa] > 0.0)  AND  n <= #v  AND
 *      desde >= 0  AND  (PT alfa EN [0,desde-1]. v[alfa] <= 0.0)
 * Post: posicionPrimerPositivo(v,n,desde) = PPP AND
 *       PPP >= 0  AND  PPP <= n-1  AND  v[PPP] > 0.0  AND
 *       (PT alfa EN [0,PPP-1]. v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n,
                            const int desde);
```

```
/*
 * Pre: (EX alfa EN [0,n-1]. v[alfa] > 0.0)  AND  n <= #v  AND
 *      desde >= 0  AND  (PT alfa EN [0,desde-1]. v[alfa] <= 0.0)
 * Post: posicionPrimerPositivo(v,n,desde) = PPP AND
 *       PPP >= 0 AND PPP <= n-1 AND v[PPP] > 0.0 AND
 *       (PT alfa EN [0,PPP-1]. v[alfa] <= 0.0)
 */
int posicionPrimerPositivo (const double v[], const int n,
                            const int desde) {
   if (v[desde] > 0.0) {
      return desde;
   }
   else {
      // (EX alfa EN [n-1]. v[alfa] > 0.0) AND desde + 1 >= 0 AND
      // (PT alfa EN [0,(desde+1)-1]. v[alfa] <= 0.0)
      return posicionPrimerPositivo(v, n, desde + 1);
   }
}
```

Problema 2. Diseñar sin bucles:

```
/*
 * Pre: v = Vo  AND  n >= 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n);
```

Se propone intentar dos diseños diferentes:

1. Aplicando una inmersión mediante refuerzo de la precondición de **`invertir`**`(v,n)`
2. Aplicando una inmersión mediante debilitamiento de la postcondición de **`invertir`**`(v,n)`

Se va a diseñar la función **`invertir`**`(v,n)`, sin bucles, programando una función auxiliar **`invertir`**`(v,n,inv)` cuya precondición se plantea como un **refuerzo de la precondición** de la primera función.

```
/*
 * Pre: v = Vo  AND  n >= 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n) {    ????    }
```

```
/*
 * Pre: n >= 0  AND  n <= #v  AND  inv >= 0  AND  inv <= n/2  AND
 *      ???
 * Post: (PT alfa EN [0,n-1]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n, const int inv);
```

```
/*
 * Pre: v = Vo  AND  n >= 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n) {    ????    }
```

```
/*
 * Pre: n >= 0  AND  n <= #v  AND  inv >= 0  AND  inv <= n/2  AND
 *      (PT alfa EN [0,inv-1].
 *          v[alfa] = Vo[n-1-alfa] AND v[n-1-alfa] = Vo[alfa]) AND
 *      (PT alfa EN [inv,n-1-inv]. v[alfa] = Vo[alfa])
 * Post: (PT alfa EN [0,n-1]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n, const int inv);
```

```
/*
 * Pre: v = Vo  AND  n >= 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n) {
   invertir(v, n, 0);
}
```

```
/*
 * Pre: n >= 0  AND  n <= #v  AND  inv >= 0  AND  inv <= n/2  AND
 *      (PT alfa EN [0,inv-1].
 *          v[alfa] = Vo[n-1-alfa] AND v[n-1-alfa] = Vo[alfa]) AND
 *      (PT alfa EN [inv,n-1-inv]. v[alfa] = Vo[alfa])
 * Post: (PT alfa EN [0,n-1].v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n, const int inv);
```

```
/*
 * Pre: n >= 0  AND  n <= #v  AND  inv >= 0  AND inv <= n / 2  AND
 *      (PT alfa EN [0,inv-1].
 *          v[alfa] = Vo[n-1-alfa] AND v[n-1-alfa] = Vo[alfa])  AND
 *      (PT alfa EN [inv,n-1-inv].v[alfa] = Vo[alfa])
 * Post: (PT alfa EN [0,n-1].v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n, const int inv) {
   if (inv < n / 2) {
      Dato aux = v[inv];
      v[inv] = v[n-1-inv];
      v[n-1-inv] = aux;
      // n >= 0 AND n <= #v AND inv + 1 >= 0 AND inv + 1 <= n / 2
      // AND (PT alfa EN [0,inv].
      //          v[alfa]=Vo[n-1-alfa] AND v[n-1-alfa]=Vo[alfa])
      // AND (PT alfa EN [inv+1,n-inv].v[alfa] = Vo[alfa])
      invertir(v, n, inv + 1);
   }
}
```

Se presenta un segundo diseño de la función **`invertir`**`(v,n)`, sin bucles, programando una función auxiliar **`invertir`**`(v,n,desde)` cuya postcondición se plantea como un **debilitamiento de la postcondición** de la primera función.

```
/*
 * Pre: v = Vo  AND  n >= 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n) {
   ???
}
```

```
/*
 * Pre: v = Vo  AND  n >= 0  AND  n <= #v  AND  desde >= 0  AND
 *      desde <= n/2
 * Post: (PT alfa EN [???,???]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n, const int desde);
```

```
/*
 * Pre: v = Vo  AND  n >= 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n) {
   ???
}
```

```
/*
 * Pre: v = Vo  AND  n >= 0  AND  n <= #v  AND  desde >= 0  AND
 *      desde <= n/2
 * Post: (PT alfa EN [desde,n-1-desde]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n, const int desde);
```

```
/*
 * Pre: v = Vo  AND  n >= 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n) {
    // n >= 0 AND 0 >= 0 AND 0 <= n/2
    invertir(v, n, 0);
}
```

```
/*
 * Pre: v = Vo  AND  n >= 0 AND n <= #v AND desde >= 0 AND
 *      desde <= n/2
 * Post: (PT alfa EN [desde,n-1-desde]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n, const int desde);
```

```
/*
 * Pre: v = Vo  AND n >= 0  AND n  <= #v AND desde >= 0 AND
 *      desde <= n/2
 * Post: (PT alfa EN [desde,n-1-desde]. v[alfa] = Vo[n-1-alfa])
 */
template <typename Dato>
void invertir (Dato v[], const int n, const int desde) {
    if (desde < n / 2) {
         Dato aux = v[desde];
         v[desde] =  v[n-1-desde];
         v[n-1-desde] = aux;
         // n >= 0 AND n <= #v AND desde + 1 >= 0
         // AND desde + 1 <= n/2
         invertir(v, n, desde + 1);
    }
}
```

Problema 3. Diseñar sin bucles:

```
/*
 * Pre: v = Vo  AND  n > 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1].
 *                 v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int n);
```

Se propone intentar dos diseños diferentes:

1. Aplicando una inmersión mediante refuerzo de la precondición de **`acumular`**`(v,n)`
2. Aplicando una inmersión mediante debilitamiento de la postcondición de **`acumular`**`(v,n)`

Se va a diseñar la función **`acumular`**`(v,n)`, sin bucles, programando una función auxiliar **`acumular`**`(v,n,hasta)` cuya precondición se plantea como un **refuerzo de la precondición** de la primera función.

```
/*
 * Pre: v = Vo  AND  n > 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1].
 *                  v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int n) {
   ???
}
```

```
/*
 * Pre: n > 0  AND  n <= #v  AND  hasta >= 0 AND hasta <= n-1  AND
 *      ???
 * Post: (PT alfa EN [0,n-1].
 *              v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int n, const int hasta);
```

```
/*
 * Pre: v = Vo  AND  n > 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1].
 *                  v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int n) {
   ???
}
```

```
/*
 * Pre: n > 0  AND  n <= #v  AND  hasta >= 0 AND hasta <= n-1  AND
 *      (PT alfa EN [0,hasta].
 *            v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )  AND
 *      (PT alfa EN [hasta+1,n-1].v[alfa] = Vo[alfa])
 * Post: (PT alfa EN [0,n-1].
 *              v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int n, const int hasta);
```

```
/*
 * Pre: v = Vo  AND  n > 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1].
 *                  v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int n) {
   acumular(v, n, 0);
}
```

```
/*
 * Pre: n > 0  AND  n <= #v  AND  hasta >= 0 AND hasta <= n-1  AND
 *      (PT alfa EN [0,hasta].
 *            v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )  AND
 *      (PT alfa EN [hasta+1,n-1]. v[alfa] = Vo[alfa])
 * Post: (PT alfa EN [0,n-1].
 *              v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int n, const int hasta);
```

```
/*
 * Pre: n > 0  AND  n <= #v AND  hasta >= 0 AND hasta <= n-1  AND
 *      (PT alfa EN [0,hasta].
 *           v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )  AND
 *      (PT alfa EN [hasta+1,n-1]. v[alfa] = Vo[alfa])
 * Post: (PT alfa EN [0,n-1].
 *             v[alfa] = (SIGMA beta EN [0,alfa]. Vo[alfa]) )
 */
void acumular (int v[], const int n, const int hasta) {
   if (hasta < n - 1) {
       v[hasta+1] = v[hasta] + v[hasta+1];
       // hasta + 1 >= 1 AND hasta + 1 <= n - 1 AND
       // (PT alfa EN [0,hasta+1].
       //     v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) ) AND
       // (PT alfa EN [hasta+2,n-1]. v[alfa] = Vo[alfa])
       //
       acumular(v, n, hasta+1);
   }
}
```

Se presenta un segundo diseño de la función **`acumular`**`(v,n)`, sin bucles, programando una función auxiliar **`acumular`**`(v,n,hasta)` cuya postcondicón se plantea como un **debilitamiento de la postcondición** de la primera función.

```
/*
 * Pre: v = Vo  AND  n > 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1].
 *              v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int n) {
   ???
}
```

```
/*
 * Pre: v = Vo  AND  hasta >= 0  AND  hasta < #v
 * Post: (PT alfa EN [0,???].
 *              v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int hasta);
```

```
/*
 * Pre: v = Vo  AND  n > 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1].
 *              v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int n) {
   ???
}
```

```
/*
 * Pre: v = Vo  AND  hasta >= 0  AND  hasta < #v
 * Post: (PT alfa EN [0,hasta].
 *              v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int hasta);
```

```
/*
 * Pre: v = Vo  AND  n > 0  AND  n <= #v
 * Post: (PT alfa EN [0,n-1].
 *              v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int n) {
   // v = Vo AND n - 1 >= 0
   acumular(v, n - 1);
}
```

```
/*
 * Pre: v = Vo  AND  hasta >= 0  AND  hasta < #v
 * Post: (PT alfa EN [0,hasta].
 *              v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int hasta);
```

```
/*
 * Pre: v = Vo  AND  hasta >= 0  AND  hasta < #v
 * Post: (PT alfa EN [0,hasta].
 *              v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) )
 */
void acumular (int v[], const int hasta) {
   if (hasta > 0) {
      // v = Vo AND hasta - 1 >= 0
      acumular(v, hasta-1);
      // (PT alfa EN [0,hasta-1].
      //      v[alfa] = (SIGMA beta EN [0,alfa]. Vo[beta]) ) AND
      // v[hasta] = Vo[hasta]
      v[hasta] = v[hasta-1] + v[hasta];
   }
}
```