Programación 2

Lección 5. Diseño de algoritmos recursivos por inmersión

1. Qué es un diseño recursivo por inmersión

---

2. Técnicas de inmersión
   - Mediante debilitamiento de la postcondición
   - Mediante fortalecimiento de la precondición

---

3. Aplicación al diseño de la función *raiz(n)*
4. Diseño por inmersión en búsqueda de eficiencia
5. Aplicación a la resolución del problema de las torres de Hanoi

1. Qué es un diseño recursivo por inmersión

Pretendemos diseñar esta función sin programar bucles :

```
/*
 * Pre: cierto
 * Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
 */
double calcular ();
```

¿Podemos aplicar la metodología de diseño recursivo presentada en la lección anterior?

```
/*
 * Pre: cierto
 * Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
 */
double calcular () {
    ... ¿ diseño sin bucles ? ...
}
```

```
/*
 * Pre: desde <= hasta
 * Post: calcular(desde, hasta) =
 *           (SIGMA alfa EN [desde,hasta]. 1.0/alfa)
 */
double calcular (const int desde, const int hasta) {

    // … código de la función calcular(desde,hasta) …

}
```

```
/*
 * Pre: cierto
 * Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
 */
double calcular () {
   return calcular(1, 100);
}
```

```
/*
 * Pre: desde <= hasta
 * Post: calcular(desde, hasta) =
 *           (SIGMA alfa EN [desde,hasta]. 1.0/alfa)
 */
double calcular (const int desde, const int hasta) {

    // … código de la función calcular(desde,hasta) …

}
```

```
// Pre: cierto
// Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
double calcular () {
   return calcular(1, 100);
}
```

```
// Pre: desde <= hasta
// Post: calcular(desde, hasta) =
//           (SIGMA alfa EN [desde,hasta]. 1.0/alfa)
double calcular (const int desde, const int hasta) {
   if (desde == hasta) {
       return 1.0 / desde;
   }
   else {
       return 1.0 / desde + calcular(desde + 1, hasta);
   }
}
```

```
// Pre: cierto
// Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
double calcular () {
   return calcular(1, 100);
}
```

```
// Pre: desde <= hasta
// Post: calcular(desde, hasta) =
//           (SIGMA alfa EN [desde,hasta]. 1.0/alfa)
double calcular (const int desde, const int hasta) {
   if (desde == hasta) {
       return 1.0 / hasta;
   }
   else {
       return 1.0 / hasta + calcular(desde, hasta - 1);
   }
}
```

Código alternativo equivalente

En qué consiste el diseño por inmersión de una función:

1. En el desarrollo de una **función auxiliar más general** (**una generalización o inmersión**)
2. En la escritura del código de la función inicial acotando la **aplicación de la función generalizada al caso que interese**

```
// Pre: cierto
// Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
double calcular ();
```

```
// Pre: desde <= hasta
// Post: calcular(desde, hasta) =
//           (SIGMA alfa EN [desde,hasta]. 1.0/alfa)
double calcular (const int desde, const int hasta);
```

2. Técnicas de inmersión

a) Mediante debilitamiento de la postcondición

```
// Pre: cierto
// Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
double calcular ();
```

```
// Pre: desde <= hasta
// Post: calcular(desde, hasta) =
//            (SIGMA alfa EN [desde,hasta]. 1.0/alfa)
double calcular (const int desde, const int hasta);
```

Un segundo diseño de la misma función por inmersión mediante debilitamiento de su postcondición:

```
// Pre: cierto
// Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
double calcular ();
```

```
// Pre: hasta >= 1
// Post: calcular(hasta) =
//                  (SIGMA alfa EN [1,hasta]. 1.0/alfa)
double calcular (const int hasta);
```

```
/*
 * Pre: cierto
 * Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
 */
double calcular () {
    ... ¿ diseño sin bucles ? ...
}
```

```
/*
 * Pre: hasta >= 1
 * Post: calcular(hasta) = (SIGMA alfa EN [1,hasta]. 1.0/alfa)
 */
double calcular (const int hasta) {

    // … código de la función calcular(hasta) …

}
```

```
/*
 * Pre: cierto
 * Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
 */
double calcular () {
   return calcular(100);
}
```

```
/*
 * Pre: hasta >= 1
 * Post: calcular(hasta) = (SIGMA alfa EN [1,hasta]. 1.0/alfa)
 */
double calcular (const int hasta) {

    // … código de la función calcular(hasta) …

}
```

```
/*
 * Pre: cierto
 * Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
 */
double calcular () {
   return calcular(100);
}
```

```
/*
 * Pre: hasta >= 1
 * Post: calcular(hasta) = (SIGMA alfa EN [1,hasta]. 1.0/alfa)
 */
double calcular (const int hasta) {
   if (hasta == 1) {
       return 1.0;
   }
   else {
       return 1.0 / hasta + calcular(hasta - 1);
   }
}
```

b) Diseño por inmersión mediante fortalecimiento de la precondición de la función auxiliar

```
// Pre: cierto
// Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
double calcular ();
```

```
// Pre: hasta >= 1 AND hasta <= 100 AND
//      sumados = (SIGMA alfa EN [1,hasta]. 1.0/alfa)
// Post: calcular(hasta,sumados) =
//                    (SIGMA alfa EN [1,100]. 1.0/alfa)
double calcular (const int hasta, const double sumados);
```

```
/*
 * Pre: cierto
 * Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
 */
double calcular () {
   ... ¿ diseño sin bucles ? ...
}
```

```
/*
 * Pre: hasta >= 1 AND hasta <= 100 AND
 *      sumados = (SIGMA alfa EN [1,hasta]. 1.0/alfa)
 * Post: calcular(hasta, sumados) =
 *                   (SIGMA alfa EN [1,100]. 1.0/alfa)
 */
double calcular (const int hasta, const double sumados);
```

```
/*
 * Pre: cierto
 * Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
 */
double calcular () {
   return calcular(1, 1.0);
}
```

```
/*
 * Pre: hasta >= 1 AND hasta <= 100 AND
 *      sumados = (SIGMA alfa EN [1,hasta]. 1.0/alfa)
 * Post: calcular(hasta, sumados) =
 *                   (SIGMA alfa EN [1,100]. 1.0/alfa)
 */
double calcular (const int hasta, const double sumados);
```

```
// Pre: cierto
// Post: calcular() = (SIGMA alfa EN [1,100]. 1.0/alfa)
double calcular () {
   return calcular(1, 1.0);
}
```

```
// Pre: hasta >= 1 AND hasta <= 100 AND
//      sumados = (SIGMA alfa EN [1,hasta]. 1.0/alfa)
// Post: calcular(hasta, sumados) =
//                    (SIGMA alfa EN [1,100]. 1.0/alfa)
double calcular (const int hasta, const double sumados) {
   if (hasta < 100) {
      return calcular(hasta + 1, 1.0/(hasta+1) + sumados);
   }
   else {  // hasta = 100 AND
           // sumados = (SIGMA alfa EN [1,hasta]. 1.0/alfa)
      return sumados;
   }
}
```

Otro problema de diseño sin bucles

Diseño sin bucles de esta función de búsqueda en vector:

```
/*
 * Pre: n >= 0  AND  n ≤ #v
 * Post: esta(v,n,dato) = (EX alfa EN [0,n-1]. v[alfa] = dato)
 */
bool esta (const int v[], const int n, const int dato);
```

Se puede aplicar la metodología de diseño recursivo, sin inmersión, presentada en la lección anterior.

```
/*
 * Pre: n >= 0  AND  n ≤ #v
 * Post: esta(v,n,dato) = (EX alfa EN [0,n-1]. v[alfa] = dato)
 */
bool esta (const int v[], const int n, const int dato) {
   if (n == 0) {
       // n = 0
       return false;                          // solución base
    }
    else {
       // n > 0
       if (v[n-1] == dato) {
           return true;                       // solución base
       }
       else {
           // n - 1 >= 0
           return esta(v, n - 1, dato);       // solución recursiva
       }
   }
}
```

¿Se puede aplicar a esta nueva función de búsqueda en vector ordenado la metodología de diseño recursivo sin inmersión?

```
/*
 * Pre: n >= 0  AND  n ≤ #v  AND
 *      (PT alfa EN [0,n-2]. v[alfa] ≤ v[alfa+1])
 * Post: esta(v,n,dato) = (EX alfa EN [0,n-1]. v[alfa] = dato)
 */
bool esta (const int v[], const int n, const int dato);
```

¿Se puede aplicar a esta función de búsqueda la metodología de diseño recursivo sin inmersión?

```
/*
 * Pre: n >= 0  AND  n ≤ #v  AND
 *      (PT alfa EN [0,n-2]. v[alfa] ≤ v[alfa+1])
 * Post: esta(v,n,dato) = (EX alfa EN [0,n-1]. v[alfa] = dato)
 */
bool esta (const int v[], const int n, const int dato);
```

Es posible aplicar la metodología de diseño recursivo, sin inmersión, presentada en la lección anterior, si se plantea una **búsqueda lineal**, comenzando la búsqueda por el elemento v[n-1]

Pero **no se puede** aplicar la metodología de diseño recursivo, sin inmersión, presentada en la lección anterior, si se desea plantear una **búsqueda binaria**, mucho más eficiente que la anterior.

a) Diseño por inmersión mediante debilitamiento de su postcondición:

```
// Pre: n >= 0 AND  n ≤ #v  AND
//      (PT alfa EN [0,n-2].v[alfa] ≤ v[alfa+1])
// Post: esta(v,n,dato) = (EX alfa EN [0,n-1]. v[alfa] = dato)
bool esta (const int v[], const int n, const int dato);
```

```
// Pre: desde >= 0  AND  desde <= hasta  AND  hasta < #v  AND
//      (PT alfa EN [desde,hasta-1].v[alfa] ≤ v[alfa+1])
// Post: esta(v,desde,hasta,dato)
//                = (EX alfa EN [desde,hasta]. v[alfa] = dato)
bool esta (const int v[], const int desde, const int hasta,
           const int dato);
```

```
/*
 * Pre: n >= 0   n ≤ #v  AND
 *      (PT alfa EN [0,n-2]. v[alfa] ≤ v[alfa+1])
 * Post: esta(v,n,dato) = (EX alfa EN [0,n-1]. v[alfa] = dato)
 */
bool esta (const int v[], const int n, const int dato) {
  ... ¿ diseño sin bucles ? ...
}
```

```
/*
 * Pre: desde >= 0  AND  desde <= hasta  AND  hasta < #v  AND
 *      (PT alfa EN [desde,hasta-1]. v[alfa] ≤ v[alfa+1])
 * Post: esta(v,desde,hasta,dato)
 *                = (EX alfa EN [desde,hasta]. v[alfa] = dato)
 */
bool esta (const int v[], const int desde, const int hasta,
           const int dato);
```

```
// Pre: n >= 0  n ≤ #v  AND
//      (PT alfa EN [0,n-2]. v[alfa] ≤ v[alfa+1])
// Post: esta(v,n,dato) = (EX alfa EN [0,n-1]. v[alfa] = dato)
bool esta (const int v[], const int n, const int dato) {
    if (n > 0) {
        // 0 >= 0 AND 0 <= n-1 AND
        // (PT alfa EN [0,n-2].v[alfa] ≤ v[alfa+1])
        return esta(v, 0, n-1, dato);
    }
    else {  // n = 0
        return false;
   }
}
```

```
// Pre: desde >= 0  AND  desde <= hasta  AND  hasta < #v  AND
//      (PT alfa EN [desde,hasta-1]. v[alfa] ≤ v[alfa+1])
// Post: esta(v,desde,hasta,dato)
//                = (EX alfa EN [desde,hasta]. v[alfa] = dato)
bool esta (const int v[], const int desde, const int hasta,
           const int dato);
```

```
// Pre: desde >= 0  AND  desde <= hasta  AND  hasta < #v  AND
//      (PT alfa EN [desde,hasta-1]. v[alfa] ≤ v[alfa+1])
// Post: esta(v,desde,hasta,dato)
//                = (EX alfa EN [desde,hasta]. v[alfa] = dato)
bool esta (const int v[], const int desde, const int hasta,
           const int dato){
   if (desde != hasta) {  // desde < hasta
      int medio = (desde + hasta) / 2;
      if (dato <= v[medio]) {  // desde >= 0 AND desde <= medio AND
                               // (PT alfa EN [desde,medio-1]. v[alfa]≤v[alfa+1])
         return esta(v, desde, medio, dato);
      }
      else {  // medio + 1 >= 0 AND medio + 1 <= hasta AND
              // (PT alfa EN [medio+1,hasta-1]. v[alfa] ≤ v[alfa+1])
         return esta(v, medio + 1, hasta, dato);
      }
   }
   else {  // desde = hasta
      return v[desde] == dato;
   }
}
```

b) Diseño por inmersión mediante fortalecimiento de la precondición de la función auxiliar

```
// Pre: n >= 0  AND  n ≤ #v  AND
//      (PT alfa EN [0,n-2]. v[alfa] ≤ v[alfa+1])
// Post: esta(v,n,dato) = (EX alfa EN [0,n-1]. v[alfa] = dato)
bool esta (int v[], int n, int dato);
```

```
// Pre: desde >= 0  AND  desde <= hasta  AND  hasta < #v  AND
//      (PT alfa EN [desde,hasta-1]. v[alfa] ≤ v[alfa+1]) AND
//      ((EX alfa EN [0,n-1]. v[alfa] = dato)
//              -> (EX alfa EN [desde,hasta]. v[alfa] = dato))
// Post: esta(v,desde,hasta,dato) =
//                  (EX alfa EN [0,n-1]. v[alfa] = dato)
bool esta (const int v[], const int desde, const int hasta,
           const int dato);
```

```
// Pre: n >= 0  AND  n ≤ #v  AND
//      (PT alfa EN [0,n-2]. v[alfa] ≤ v[alfa+1])
// Post: esta(v,n,dato) = (EX alfa EN [0,n-1]. v[alfa] = dato)
bool esta (const int v[], const int n, const int dato) {
   if (n > 0) {  // 0 >= 0 AND 0 <= n - 1 AND
                 // (PT alfa EN [0,n-2]. v[alfa]≤v[alfa+1]) AND
                 // ((EX alfa EN [0,n-1]. v[alfa] = dato)
                        -> (EX alfa EN [0,n-1]. v[alfa] = dato))
       return esta(v, 0, n - 1, dato);
    }
    else {  /* n = 0 */  return false;  }
}
```

```
// Pre: desde >= 0  AND  desde <= hasta  AND  hasta < #v  AND
//      (PT alfa EN [desde,hasta-1]. v[alfa] ≤ v[alfa+1])  AND
//      ((EX alfa EN [0,n-1]. v[alfa] = dato)
//              -> (EX alfa EN [desde,hasta]. v[alfa] = dato))
// Post: esta(v,desde,hasta,dato)
//                = (EX alfa EN [0,n-1]. v[alfa] = dato)
bool esta (const int v[], const int desde, const int hasta,
           const int dato);
```

```
// Pre: desde >= 0  AND  desde <= hasta  AND  hasta < #v  AND
//      (PT alfa EN [desde,hasta-1]. v[alfa] ≤ v[alfa+1])  AND
//      ((EX alfa EN [0,n-1]. v[alfa] = dato)
//           -> (EX alfa EN [desde,hasta]. v[alfa] = dato))
// Post: esta(v,n,desde,hasta,dato)
//                = (EX alfa EN [0,n-1].v[alfa] = dato)
bool esta (const int v[], const int desde, const int hasta,
           const int dato) {
   if (desde != hasta) {  // desde < hasta
      int medio = (desde + hasta) / 2;
      if (dato <= v[medio]) {
          return esta(v, desde, medio, dato);
      }
      else {
          return esta(v, medio + 1, hasta, dato);
      }
   }
   else {  /* desde = hasta */  return v[desde] == dato;   }
}
```

3. Aplicación al diseño de la función *raiz(n)*

Debemos hacer un diseño sin bucles de la función ***raiz(n)***.

```
// Pre: n >= 0
// Post: raiz(n)^2 <= n AND (raiz(n) + 1)^2 > n
int raiz (const int n);
```

n	raiz(n)
…	…
23	4
24	4
25	5
26	5
…	…

n	raiz(n)
…	…
34	5
35	5
36	6
37	6
…	…

Una especificación más legible de la función ***raiz(n)***.

```
// Pre: n >= 0
// Post: raiz(n) = R AND R^2 <= n AND (R + 1)^2 > n
int raiz (const int n);
```

n	raiz(n)
…	…
23	4
24	4
25	5
26	5
…	…

n	raiz(n)
…	…
34	5
35	5
36	6
37	6
…	…

Diseño por inmersión mediante fortalecimiento de la precondición

```
// Pre: n >= 0
// Post: raiz(n) = R AND R^2 <= n AND (R + 1)^2 > n
int raiz (const int n) { //  n >= 0 AND 0^2 <= n

   // … código de la función raiz(n) …

}
```

```
// Pre: n >= 0 AND candidato^2 <= n
// Post: raiz1(n,candidato) = R AND R^2 <= n AND (R+1)^2 > n
int raiz1 (const int n, const int candidato)

   // … código de la función raiz1(n,candidato) …

}
```

Diseño sin bucles de la función `raiz(n)`

```
// Pre: n >= 0
// Post: raiz(n) = R AND R^2 <= n AND (R + 1)^2 > n
int raiz (const int n) {
   //  n >= 0 AND 0^2 <= n
   return raiz1(n, 0);
}
```

```
// Pre: n >= 0 AND candidato^2 <= n
// Post: raiz1(n,candidato) = R AND R^2 <= n AND (R+1)^2 > n
int raiz1 (const int n, const int candidato)

   // … código de la función raiz1(n,candidato) …

}
```

Diseño sin bucles de la función auxiliar `raiz1(n,candidato)`

```
// Pre: n >= 0 AND candidato^2 <= n
// Post: raiz1(n,candidato) = R AND R^2 <= n AND (R+1)^2 > n
int raiz1 (const int n, const int candidato) {
   if ((candidato + 1) * (candidato + 1) > n) {
      // candidato^2 <= n AND (candidato + 1)^2 > n
      return candidato;
   }
   else {
      // n >= 0  AND (candidato + 1)^2 <= n
      return raiz1(n, candidato + 1);
   }
}
```

Otro diseño por inmersión mediante fortalecimiento de la precond.

```
// Pre: n >= 0
// Post: raiz(n) = R AND R^2 <= n AND (R + 1)^2 > n
int raiz (const int n) {  // n >= 0 AND (n + 1)^2 > n
   // … código de la función raiz(n) …
}
```

```
// Pre: n >= 0 AND (candidato + 1)^2 > n
// Post: raiz2(n,candidato) = R AND R^2 <= n AND (R + 1)^2 > n
int raiz2 (const int n, const int candidato) {

   // … código de la función raiz2(n,candidato) …

}
```

Diseño sin bucles de la función `raiz(n)`

```
// Pre: n >= 0
// Post: raiz(n) = R AND R^2 <= n AND (R + 1)^2 > n
int raiz (const int n) {
   // n >= 0 AND (n + 1)^2 > n
   return raiz2(n, n);
}
```

```
// Pre: n >= 0 AND (candidato + 1)^2 > n
// Post: raiz2(n,candidato) = R AND R^2 <= n AND (R + 1)^2 > n
int raiz2 (const int n, const int candidato) {

   // … código de la función raiz2(n,candidato) …

}
```

Diseño sin bucles de la función auxiliar `raiz2(n,candidato)`

```
// Pre: n >= 0 AND (candidato + 1)^2 > n
// Post: raiz2(n,candidato) = R AND R^2 <= n AND (R + 1)^2 > n
int raiz2 (const int n, const int candidato) {
   if (candidato * candidato <= n) {
      // candidato^2 <= n AND (candidato + 1)^2 > n
      return candidato;
   }
   else {
      // n >= 0 AND candidato^2 > n
      return raiz2(n, candidato - 1);
   }
}
```

Diseño por inmersión mediante debilitamiento de la postcondición

```
// Pre: n >= 0
// Post: raiz(n) = R AND R^2 <= n AND (R + 1)^2 > n
int raiz (const int n) {  //  n >= 0 AND 1 >= 1
   // … código de la función raiz(n,) …
}
```

```
// Pre: n >= 0 AND a >= 1
// Post: raiz3(n,a) = R AND R^2 <= n AND (R + a)^2 > n
int raiz3 (const int n, const int a) {

    // … código de la función raiz3(n,a) …

}
```

Diseño sin bucles de la función `raiz(n)`

```
// Pre: n >= 0
// Post: raiz(n) = R AND R^2 <= n AND (R + 1)^2 > n
int raiz (const int n) {
   //  n >= 0 AND 1 >= 1
   return raiz3(n, 1);
}
```

```
// Pre: n >= 0 AND a >= 1
// Post: raiz3(n,a) = R AND R^2 <= n AND (R + a)^2 > n
int raiz3 (const int n, const int a) {

    // … código de la función raiz3(n,a) …

}
```

```
// Pre: n > = 0 AND a >= 1
// Post: raiz3(n,a) = R AND R^2 <= n AND (R + a)^2 > n
int raiz3 (const int n, const int a) {
   if (a * a > n) {
      // 0^2 <= n AND (0 + a)^2 > n
      return 0;
   }
   else  {  // n > = 0 AND 2*a >= 1
      int candidato = raiz3(n, 2*a);
      // candidato^2 <= n AND (candidato + 2*a)^2 > n
      if ((candidato + a) * (candidato + a)) > n) {
         // candidato^2 <= n AND (candidato + a)^2 > n
         return candidato;
      }
      else {
         // (candidato + a)^2 <= n AND (candidato + 2a)^2 > n
         return candidato + a;
      }
   }
}
```

Diseño sin bucles de la función auxiliar `raiz3(n,candidato)`

```
// Pre: n >= 0 AND a >= 1
// Post: raiz3(n,a) = R AND R^2 <= n AND (R + a)^2 > n
int raiz3 (const int n, const int a) {
   if (a * a > n) {
      // 0^2 <= n AND (0+a)^2 > n
      return 0;
   }
   else  {
      // a^2 <= n AND n >= 0 AND 2*a >=1
      int candidato = raiz3(n, 2*a);
      if ((candidato + a) * (candidato + a) > n) {
         return candidato;
      }
      else {
         return candidato + a;
      }
   }
}
```

4. Diseño por inmersión en búsqueda de eficiencia

Este diseño es altamente ineficiente:

```
// Pre: n >= 1
// Post: (n = 1 OR n = 2 -> fibonacci(n) = n-1) AND
//       (n > 2 -> fibonacci(n) = fibonacci(n-2) + fibonacci(n-1))
int fibonacci (const int n) {
    if (n <= 2) { // n = 1 OR n = 2
       return n - 1;
    }
    else {  // n-2 >= 1 AND n-1 >= 1
       return fibonacci(n-2) + fibonacci(n-1);
    }
}
```

¿Cómo abordar un diseño recursivo eficiente de esta función?

```
// Pre: n >= 1
// Post: (n = 1 OR n = 2 -> fibonacci(n) = n-1) AND
//       (n > 2 -> fibonacci(n) = fibonacci(n-2) + fibonacci(n-1))
int fibonacci (const int n) {
    if (n <= 2) {
        return n-1;
    }
    else {
        return fibonacci(n-2) + fibonacci(n-1);
    }
}
```

¿Cómo abordar un diseño recursivo eficiente de esta función?

Sucesión de Fibonacci: { 0, 1, 1, 2, 3, 5, 8, 13, 21,
34, 55, 89, 144, 233, 377, 610, 987, 1597, … }

Diseño por inmersión de la función `fibonacci(n)`

```
// Pre: n >= 1
// Post: (n = 1 OR n = 2 -> fibonacci(n) = n-1) AND
//       (n > 2 -> fibonacci(n) = fibonacci(n-2) + fibonacci(n-1) )
int fibonacci (const int n) {
   // … código de la función fibonacci(n) …
}
```

```
// Pre: n > 1 AND i > 1 AND i <= n AND
//      fib anterior = fibonacci(i-1) AND fib i = fibonacci(i)
// Post: (n = 2 -> fibonacci(n,i,fib_anterior,fib_i) = n-1) AND
//       (n > 2 -> fibonacci(n,i,fib_anterior,fib_i) =
//                    fibonacci(n-2,i,fib_anterior,fib_i) +
//                    fibonacci(n-1,i,fib_anterior,fib_i) )
int fibonacci (const int n, const int i,
               const int fib_anterior, const int fib_i) {
    // … código de la función fibonacci(n,i,fib_anterior,fib_i) …
}
```

Diseño sin bucles de la función `fibonacci(n)`

```
// Pre: n >= 1
// Post: (n = 1 OR n = 2 -> fibonacci(n) = n-1) AND
//       (n > 2 -> fibonacci(n) = fibonacci(n-2) + fibonacci(n-1) )
int fibonacci (const int n) {
   if (n == 1) { /* n = 1 */  return 0; }
   else {  // n > 1 AND 2 < 1 AND 2 < n AND
           // 0 = fibonacci(0) AND 1 = fibonacci(1)
      return fibonacci(n, 2, 0, 1);
   }
}
```

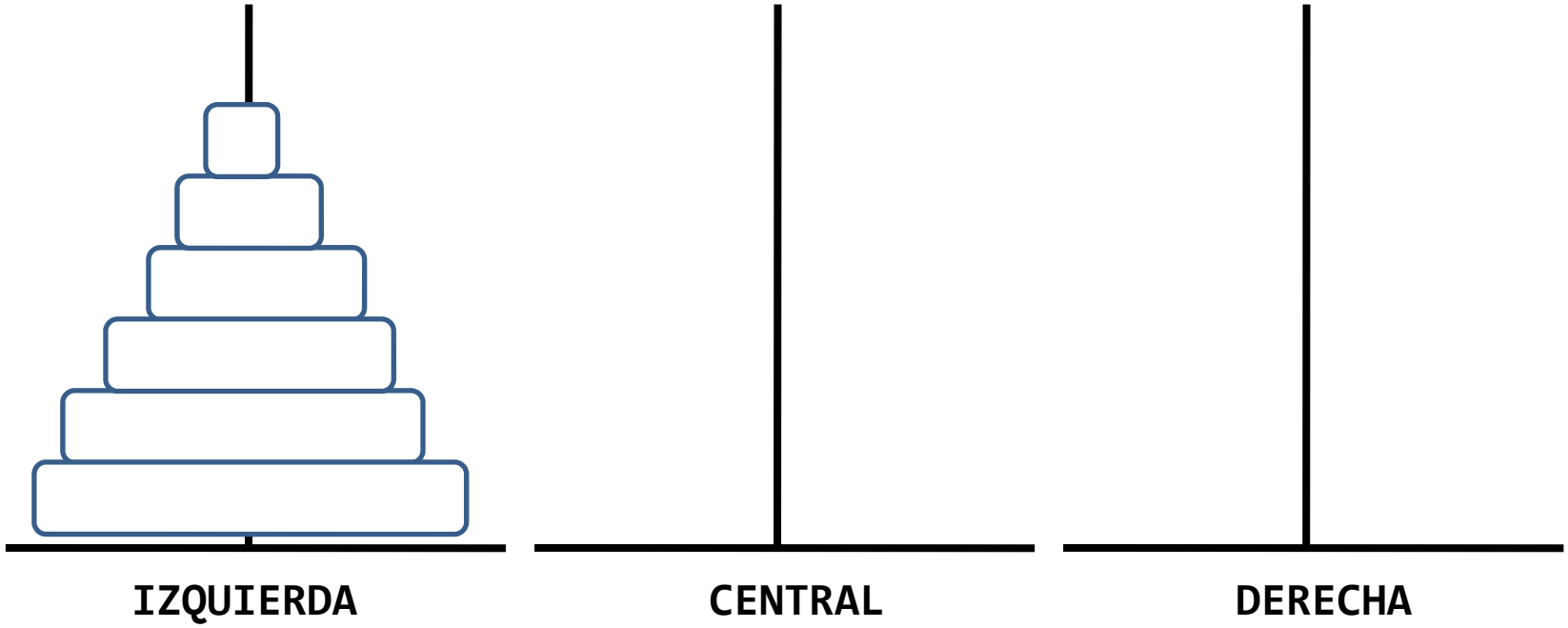
```
// Pre: n > 1 AND i > 1 AND i <= n AND
//      fib_anterior = fibonacci(i-1) AND fib_i = fibonacci(i)
// Post: (n = 2 -> fibonacci(n,i,fib_anterior,fib_i) = n-1) AND
//       (n > 2 -> fibonacci(n,i,fib_anterior,fib_i) =
//                     fibonacci(n-2,i,fib_anterior,fib_i) +
//                     fibonacci(n-1,i,fib_anterior,fib_i) )
int fibonacci (const int n, const int i,
               const int fib_anterior, const int fib_i);
```

Diseño sin bucles de la función auxiliar **fibonacci**(n,i,fib_anterior,fib_i)

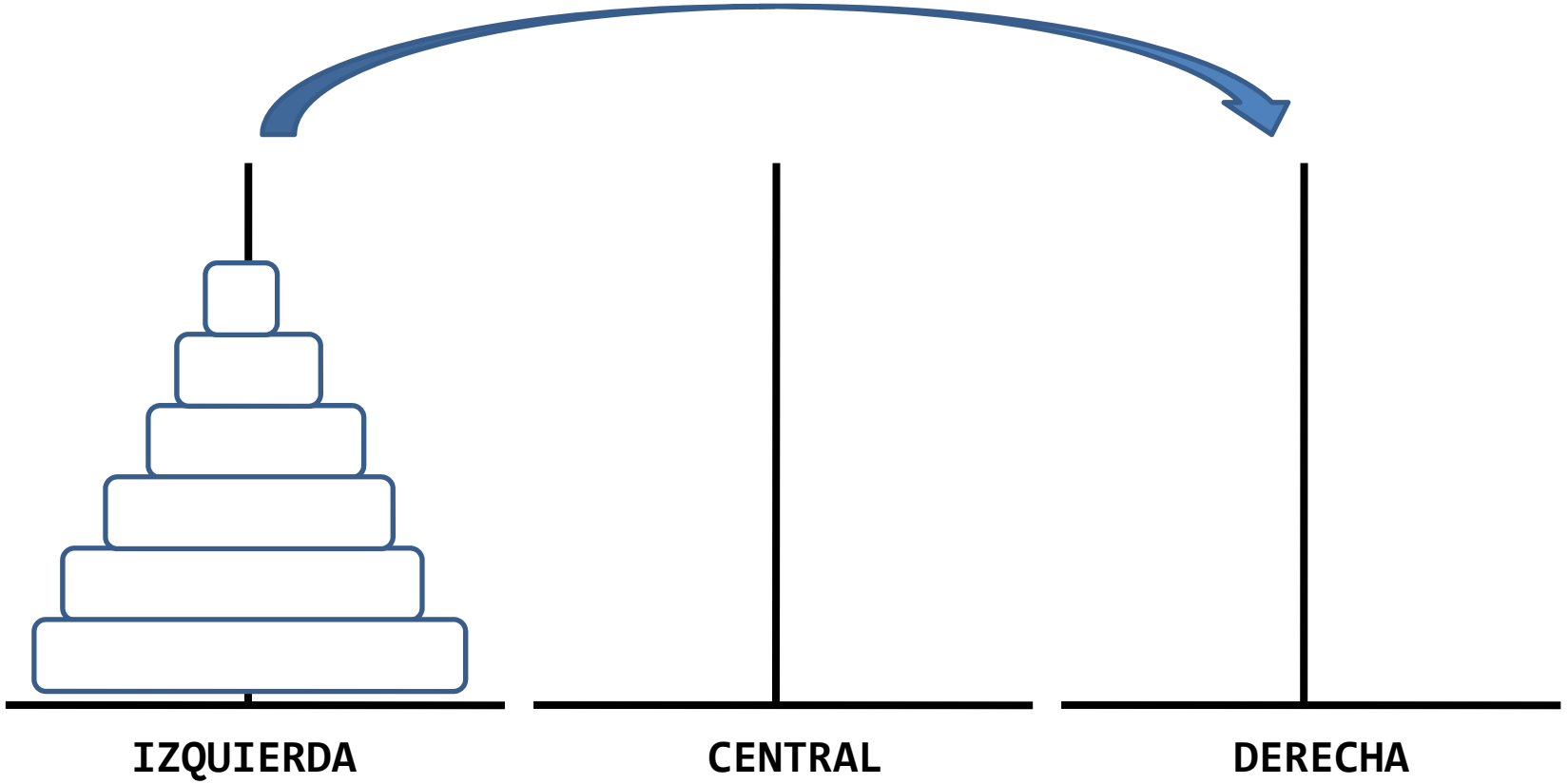
```
// Pre: n > 1 AND i > 1 AND i <= n AND
//      fib anterior = fibonacci(i-1) AND fib i = fibonacci(i)
// Post: (n = 2 -> fibonacci(n,i,fib_anterior,fib_i) = n-1) AND
//       (n > 2 -> fibonacci(n,i,fib_anterior,fib_i) =
//                     fibonacci(n-2,i,fib_anterior,fib_i) +
//                     fibonacci(n-1,i,fib_anterior,fib_i) )
int fibonacci (const int n, const int i,
               const int fib_anterior, const int fib_i) {
   if (i == n) {
      return fib_i;
   }
   else {
      return fibonacci(n, i + 1, fib_i, fib_anterior + fib_i);
   }
}
```

5. Aplicación a la resolución del problema de las torres de Hanoi

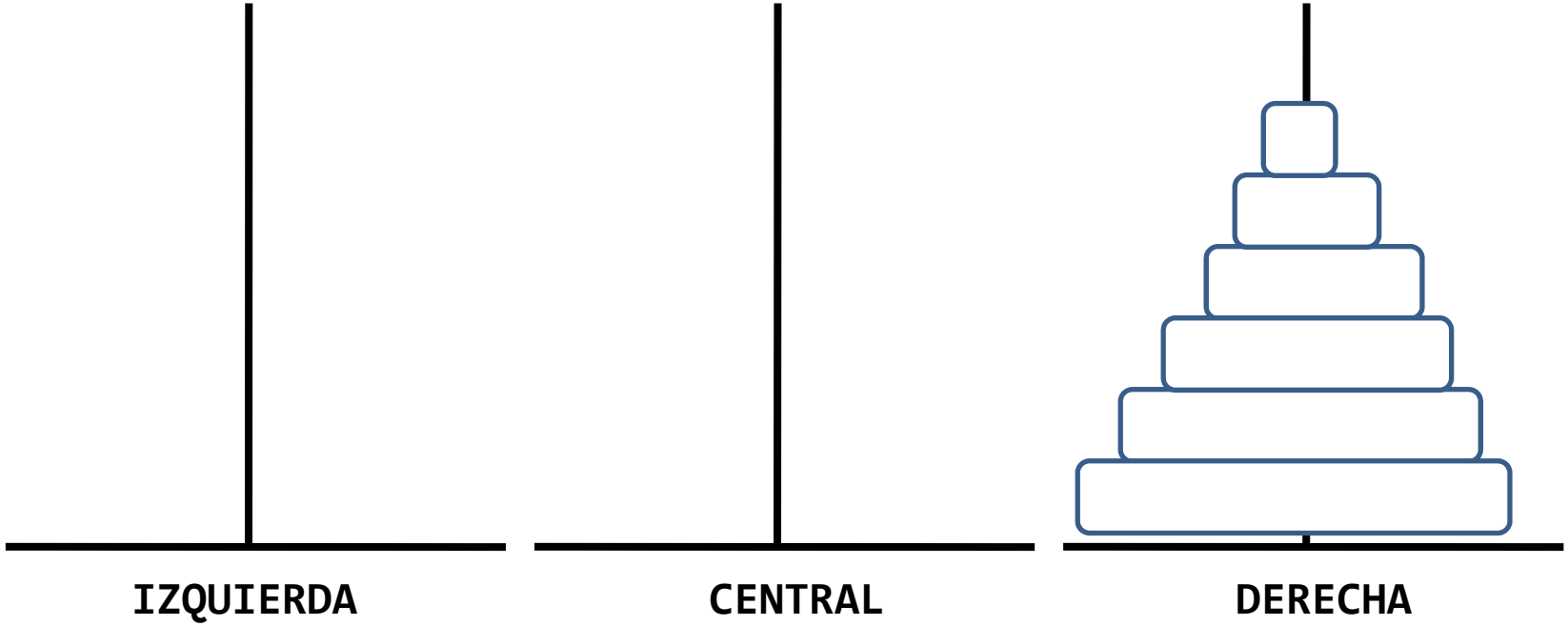
Punto de partida: “n” discos apilados en la torre IZQUIERDA



Objetivo: trasladar los “n” discos a la torre DERECHA

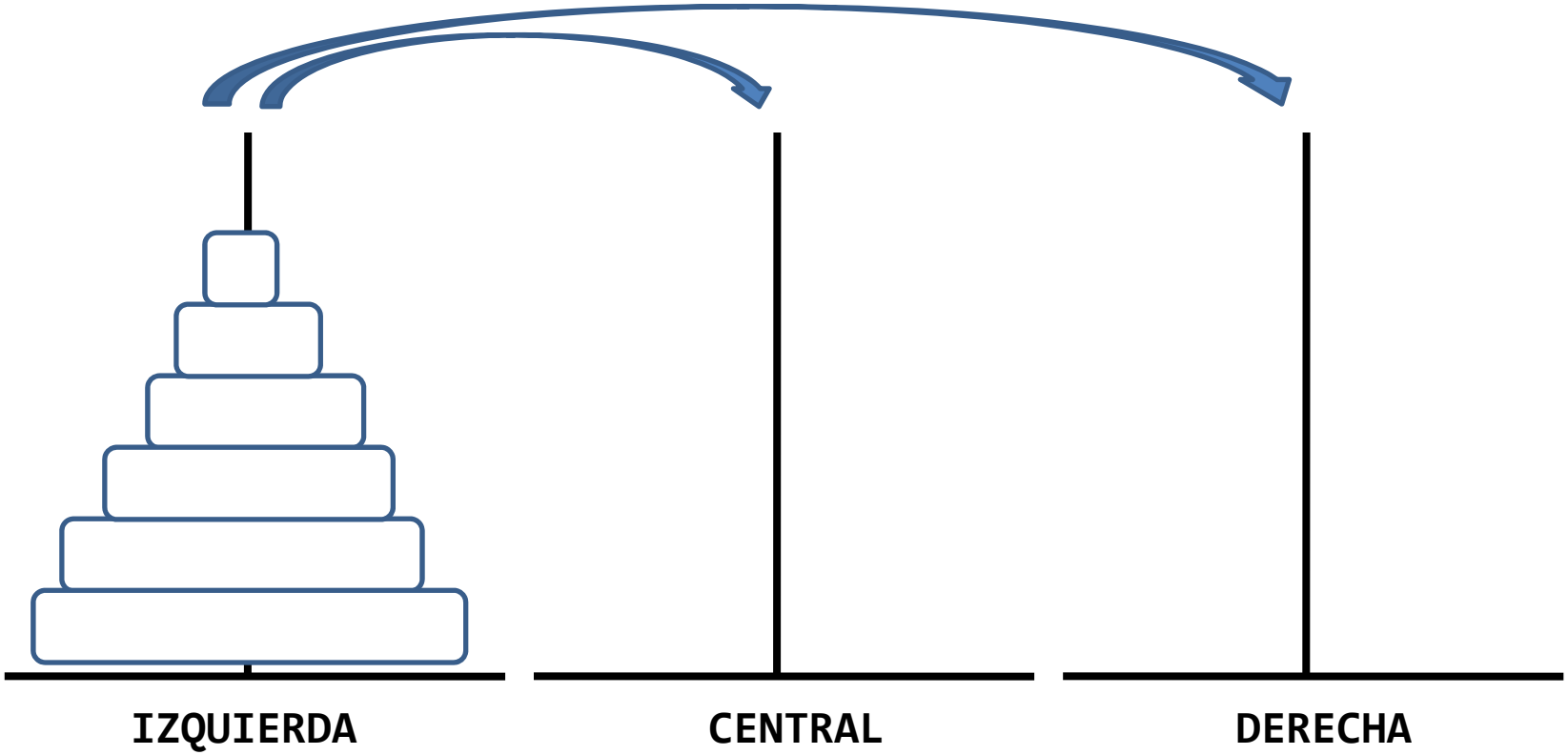


Punto final: “n” discos apilados en la torre DERECHA

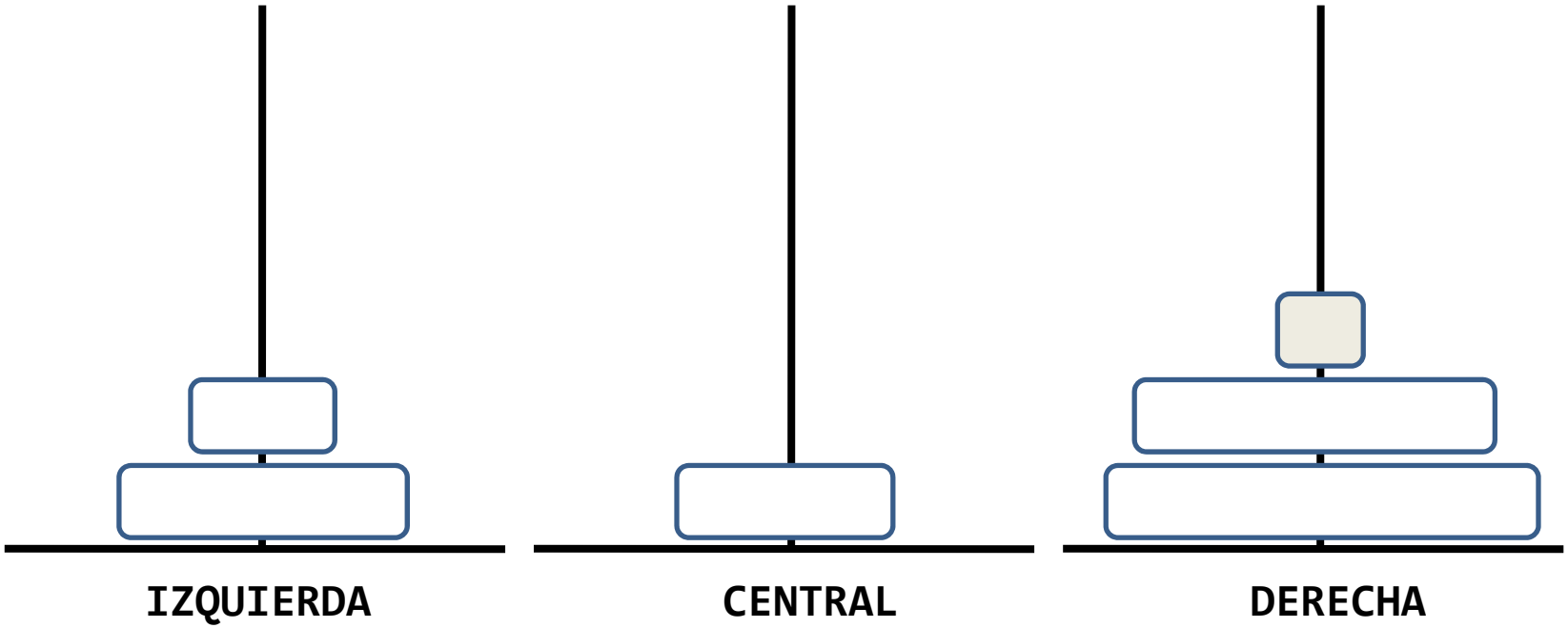


Reglas del juego:

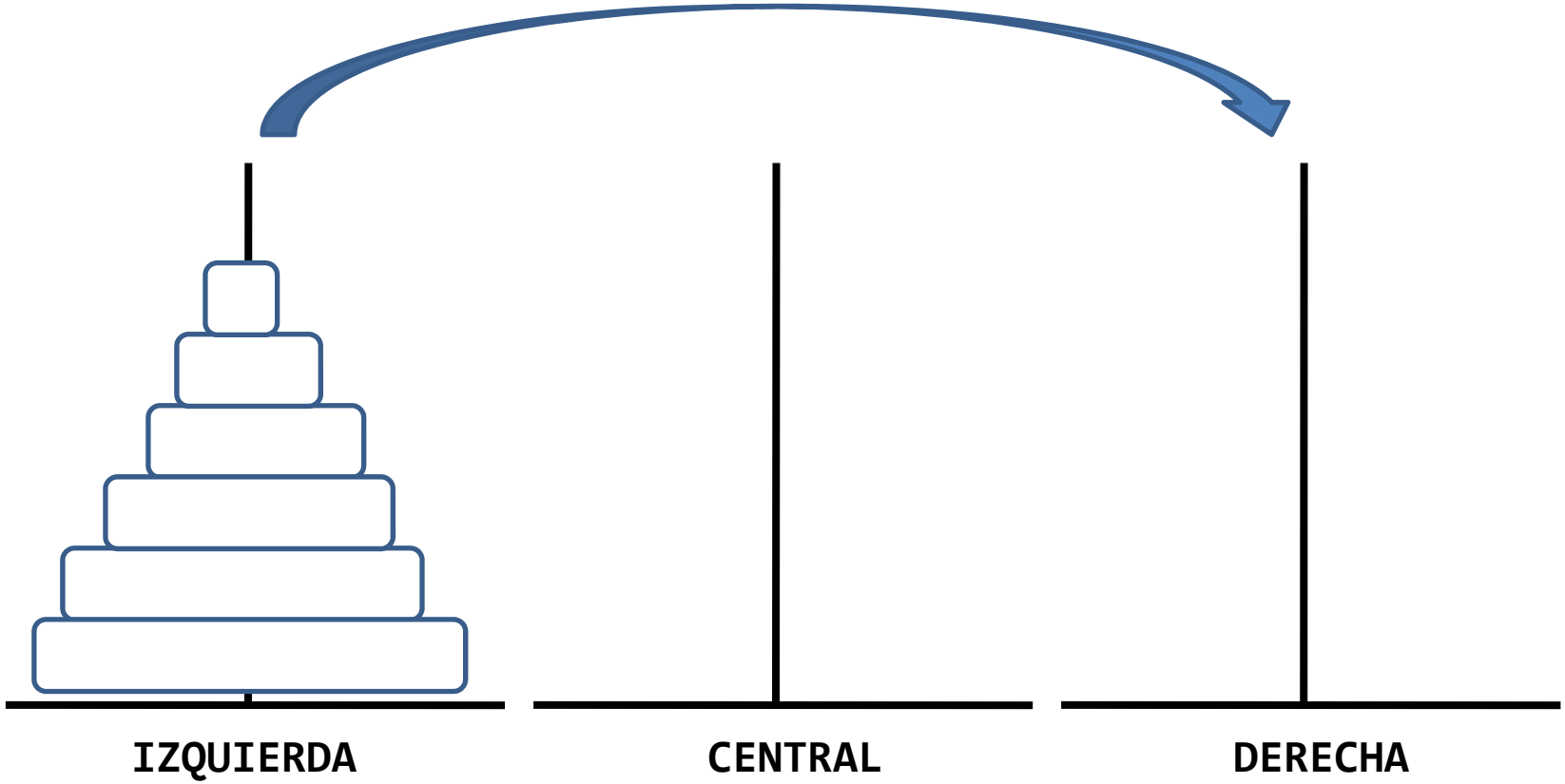
1. Los discos se pueden **trasladar uno a uno** de una torre cualquiera a otra torre cualquiera
2. En cualquiera de las tres torres los discos han de estar apilados formado una pirámide, estando **prohibido que un disco esté por encima de otro de menor diámetro**



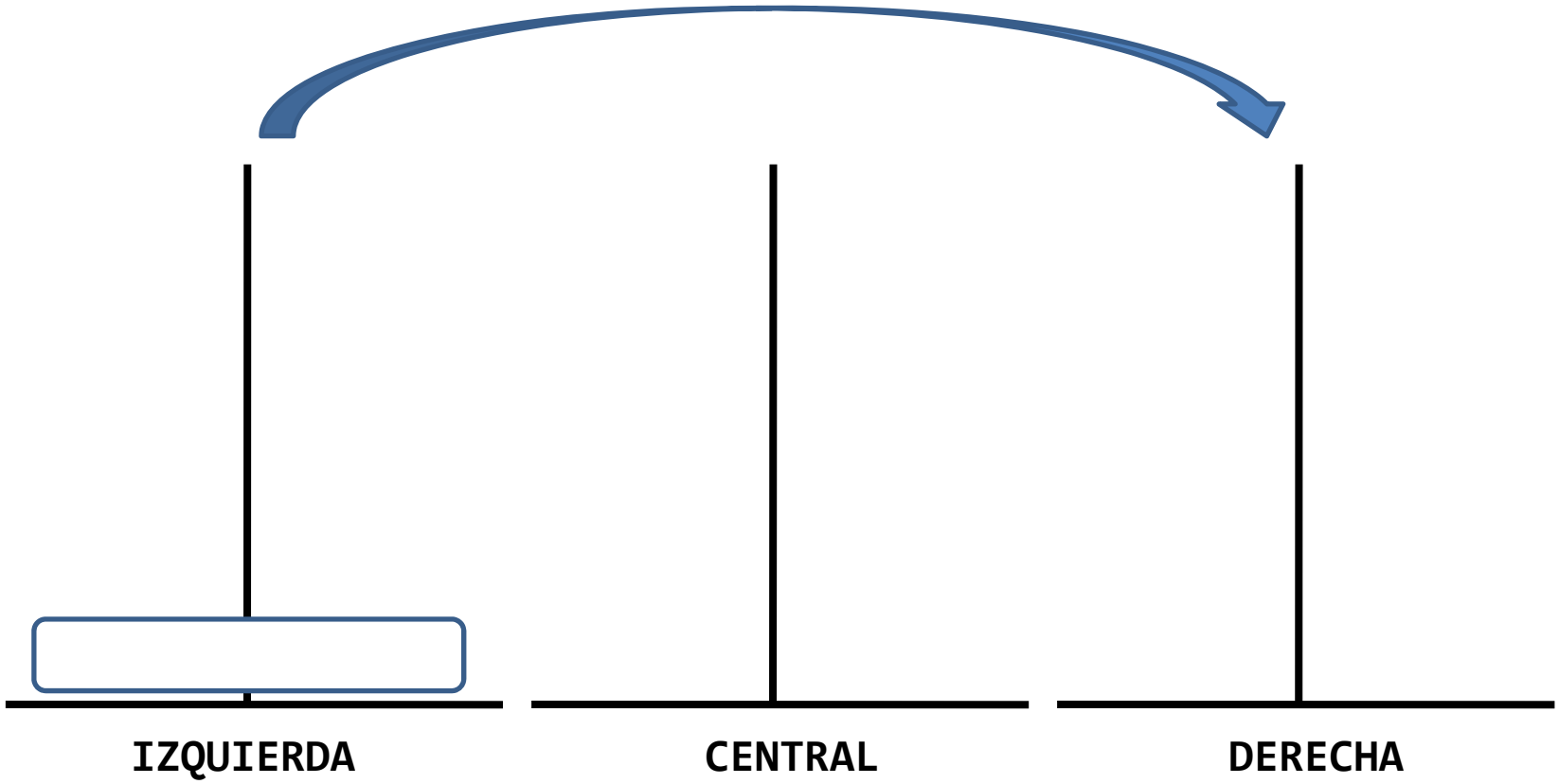
Ejemplo de situación intermedia



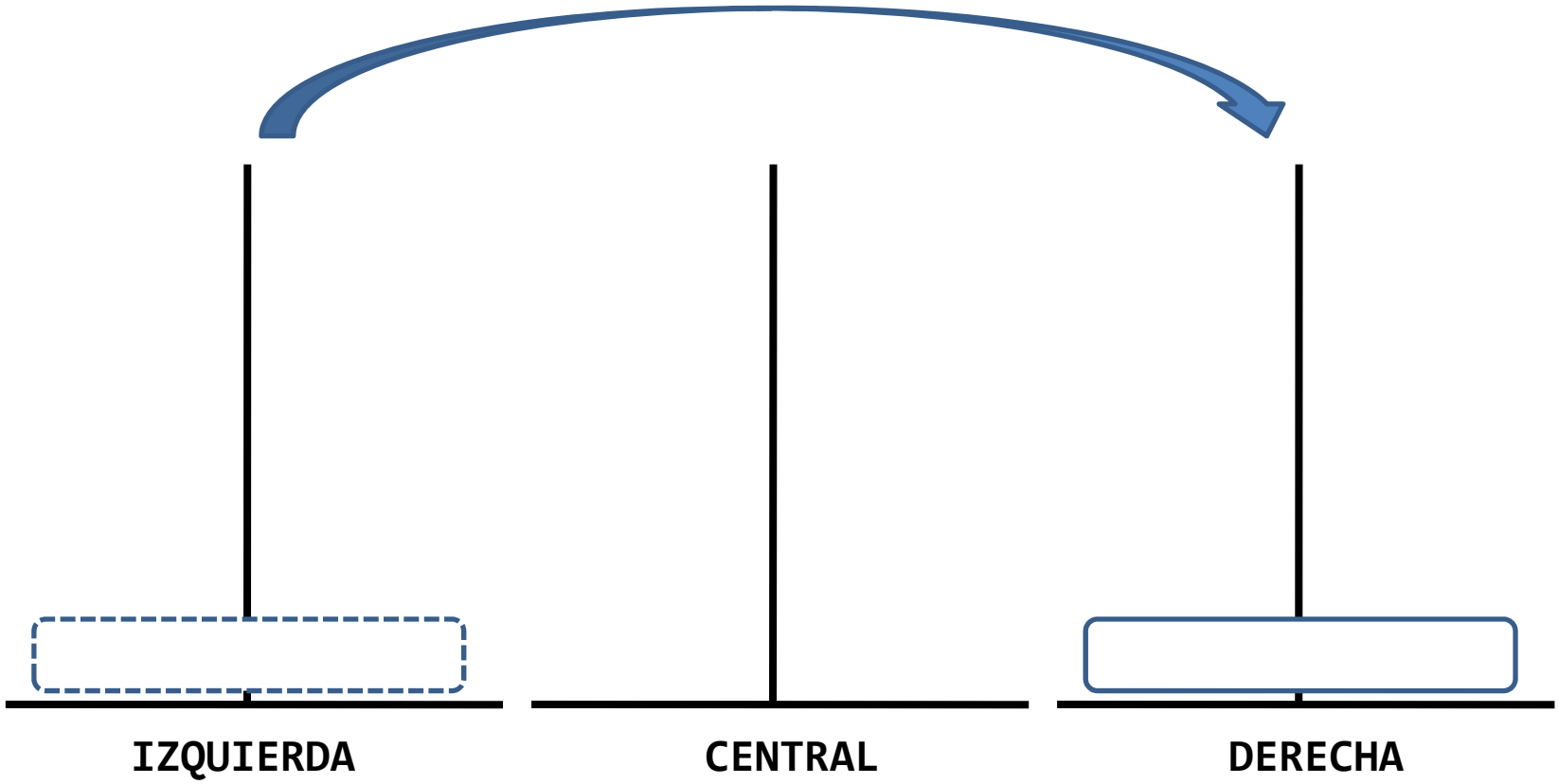
¿Cómo trasladar, uno a uno, todos los discos?



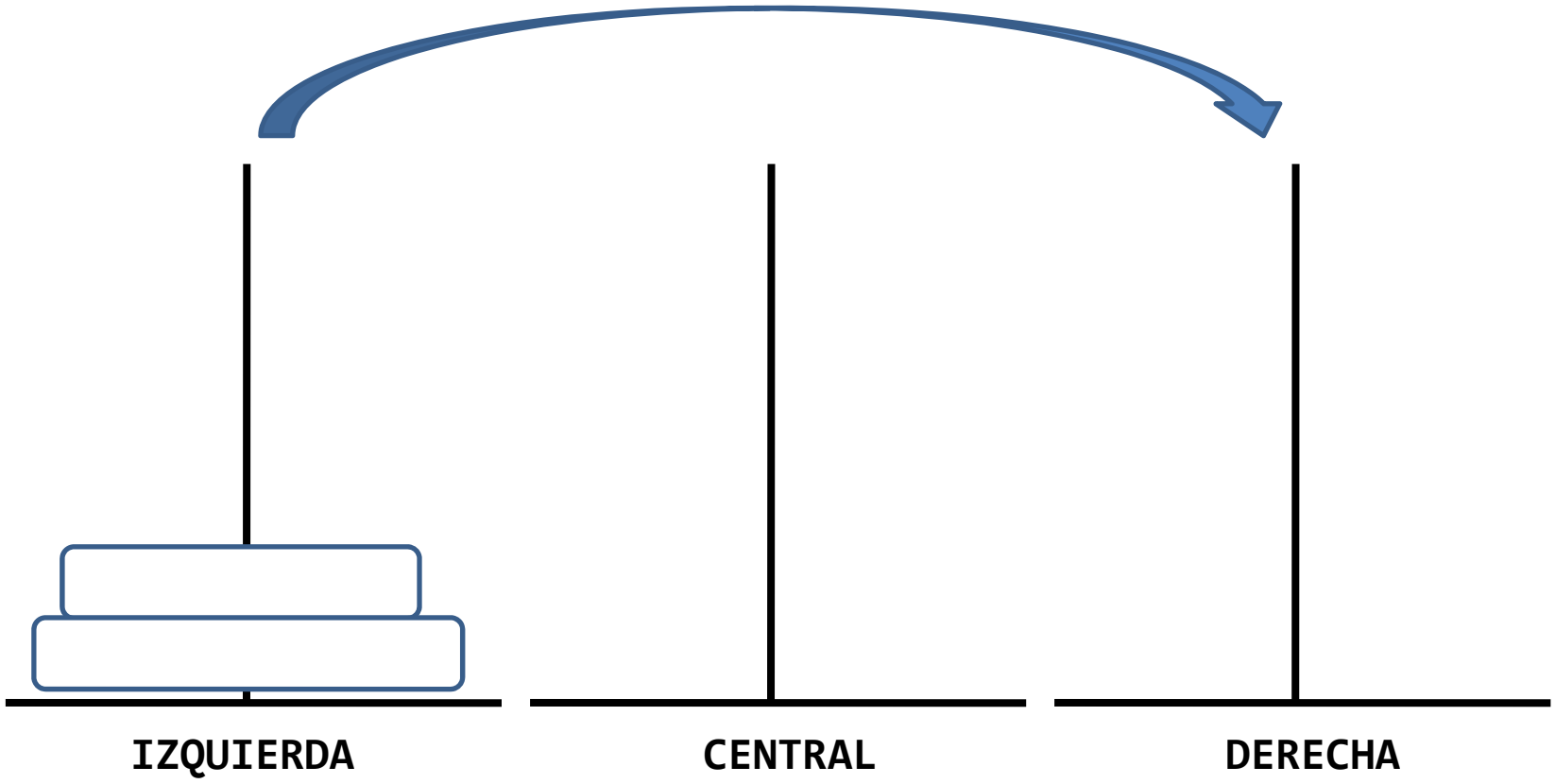
¿Y si el número de discos a trasladar fuera 1?



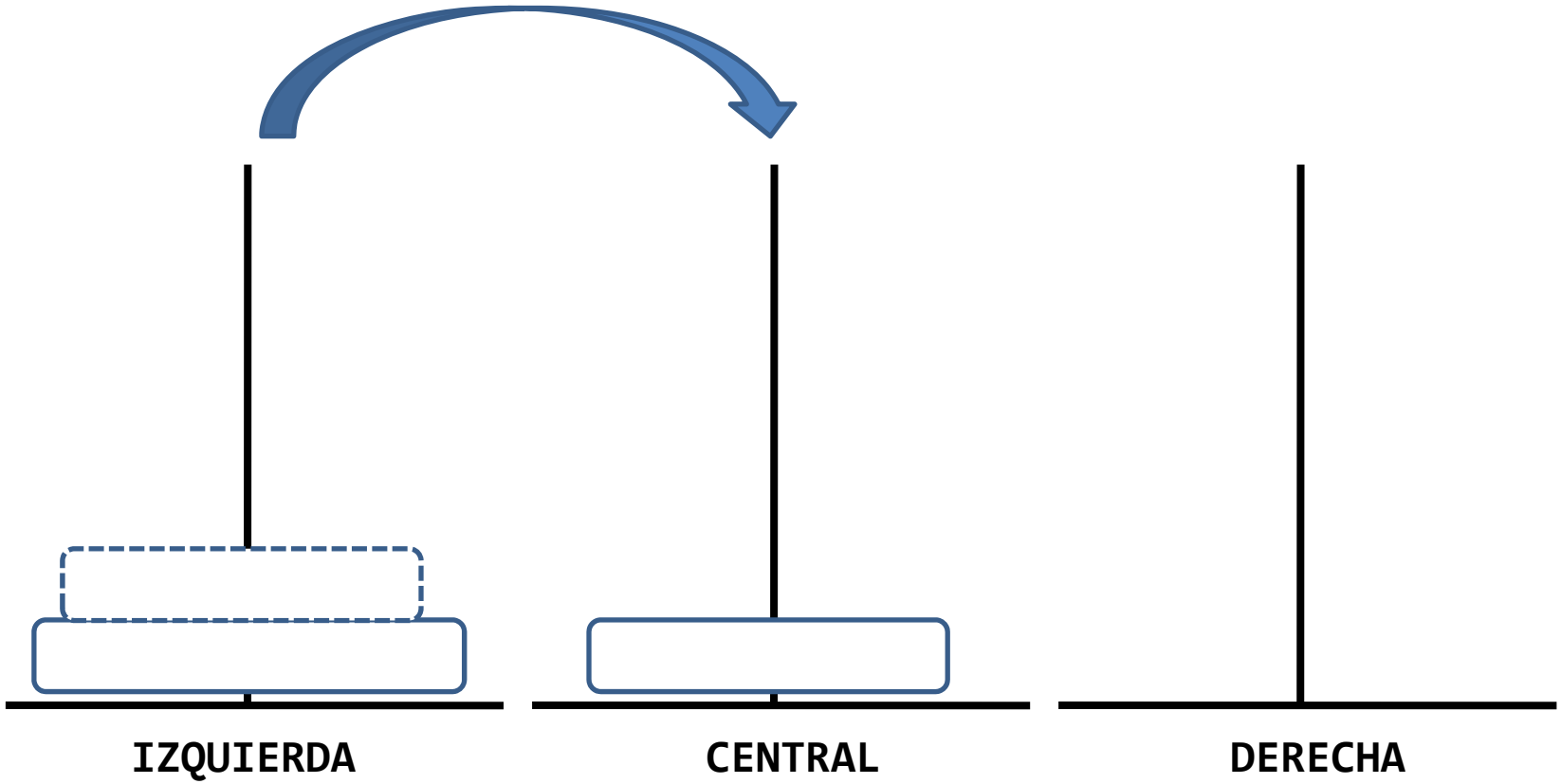
¡ Problema resuelto con un solo movimiento de disco !



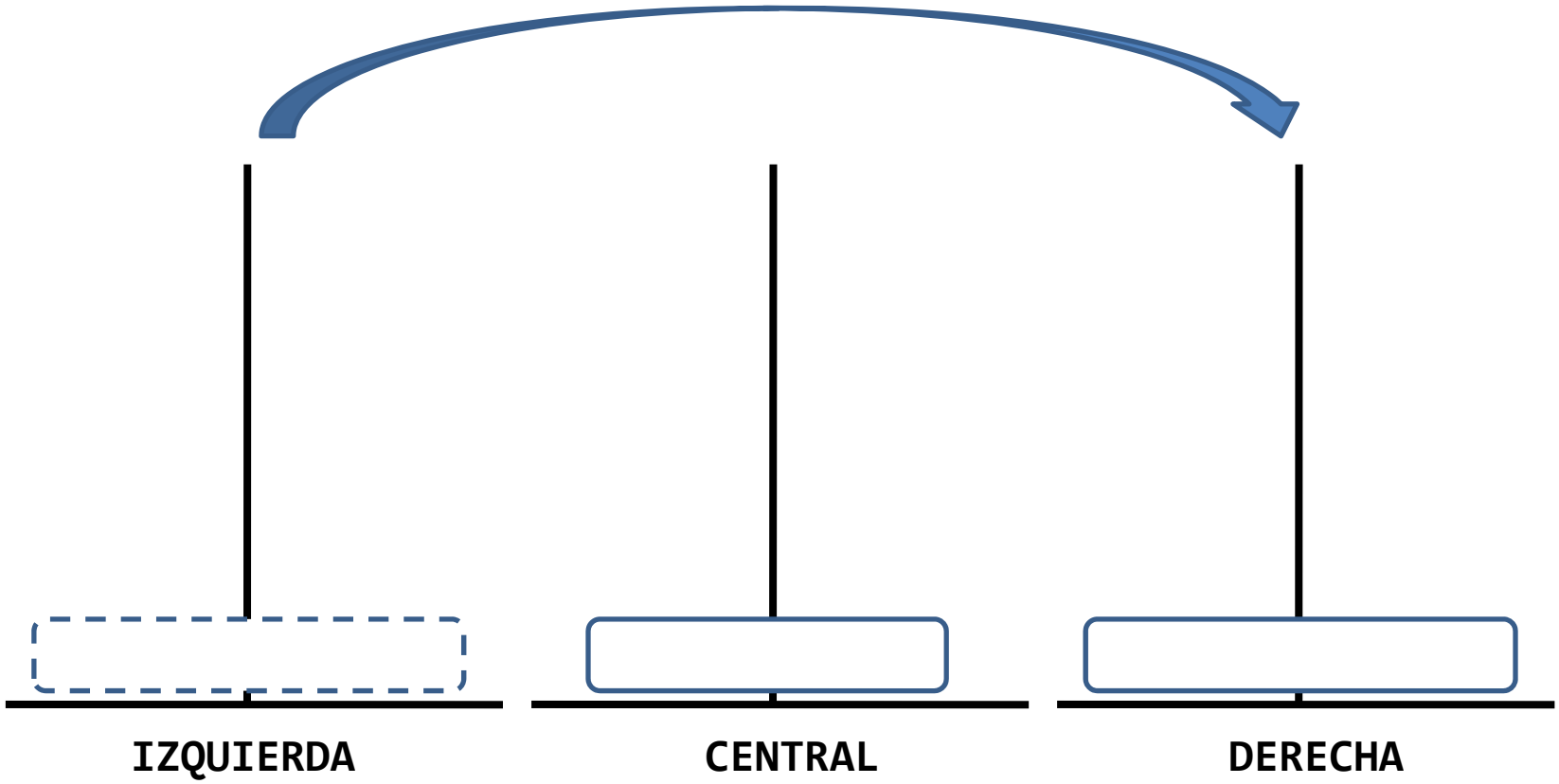
¿Y si el número de discos a trasladar fuera 2?



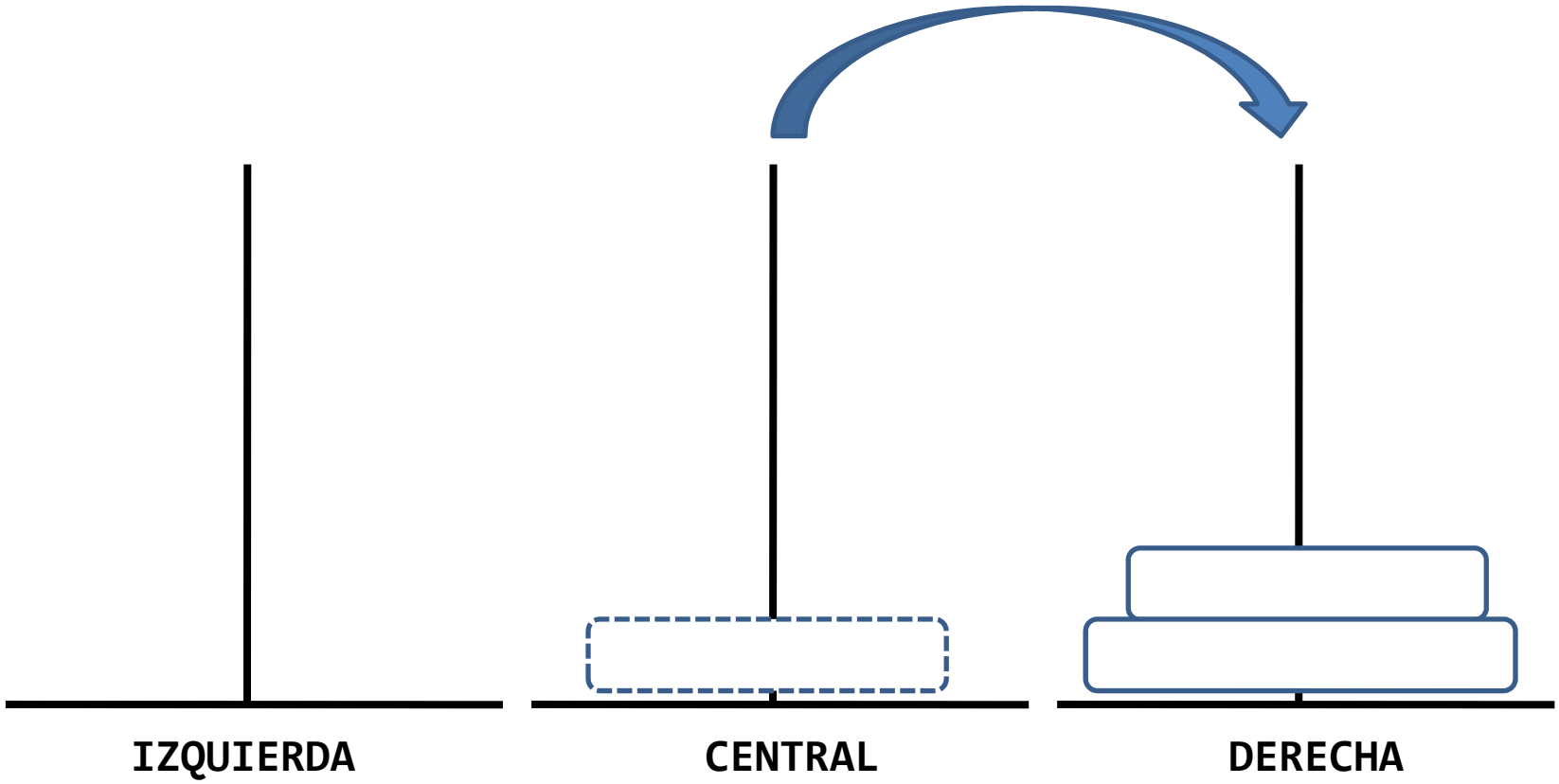
1. Movemos un disco de IZQUIERDA a CENTRAL



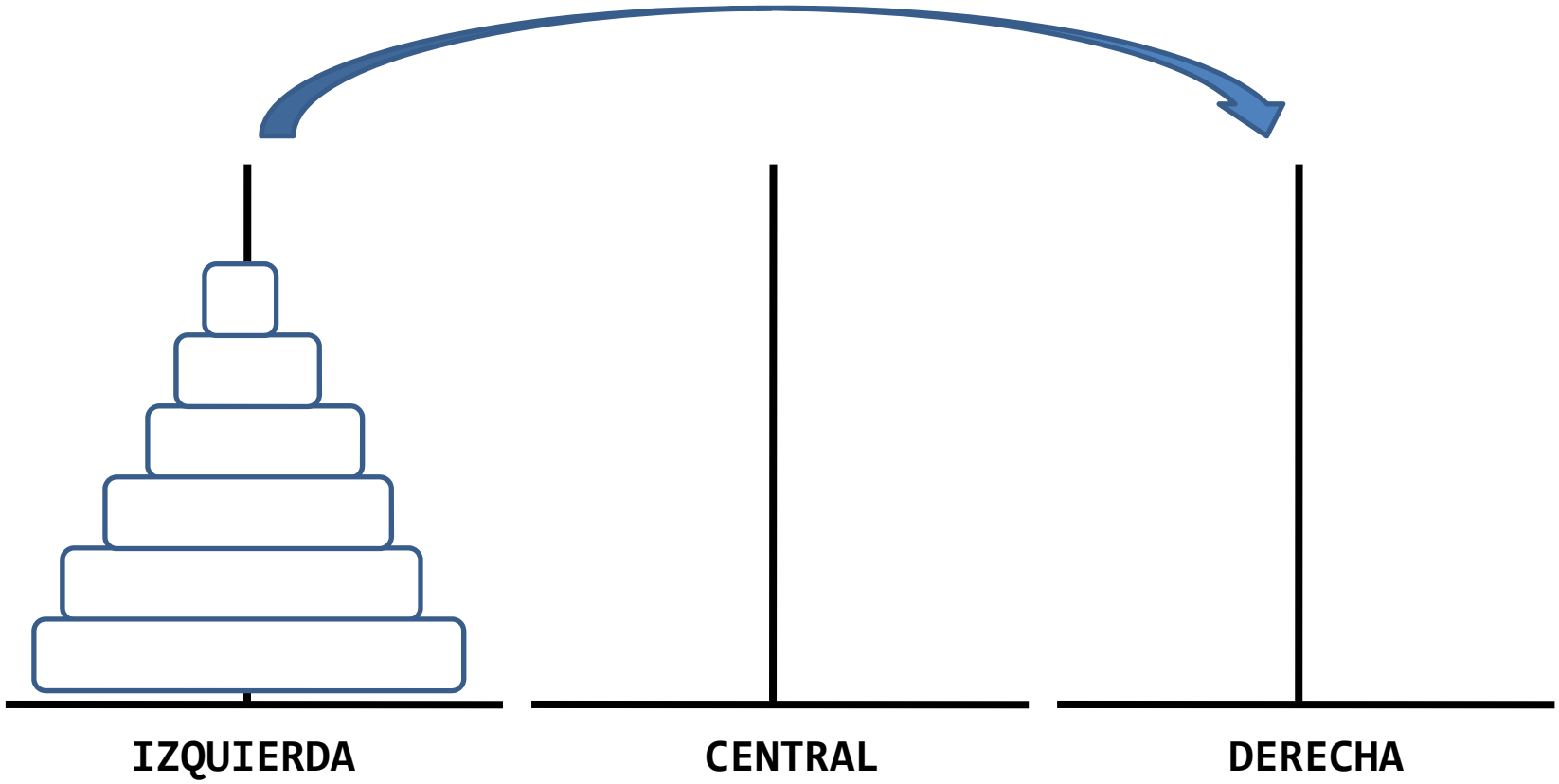
2. Movemos el disco que queda en IZQUIERDA a DERECHA



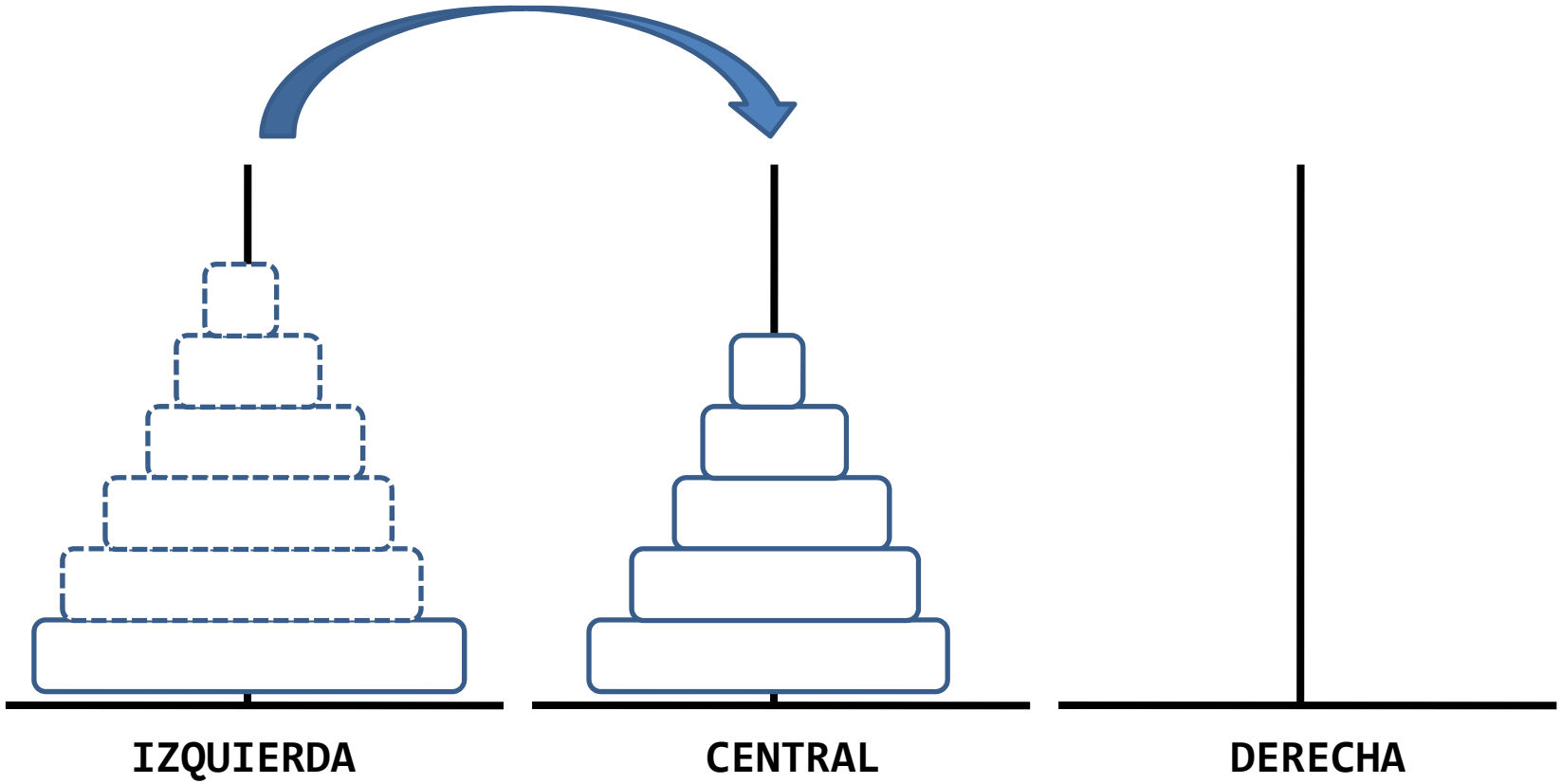
3. ¡ Y movemos un disco de CENTRAL a DERECHA !



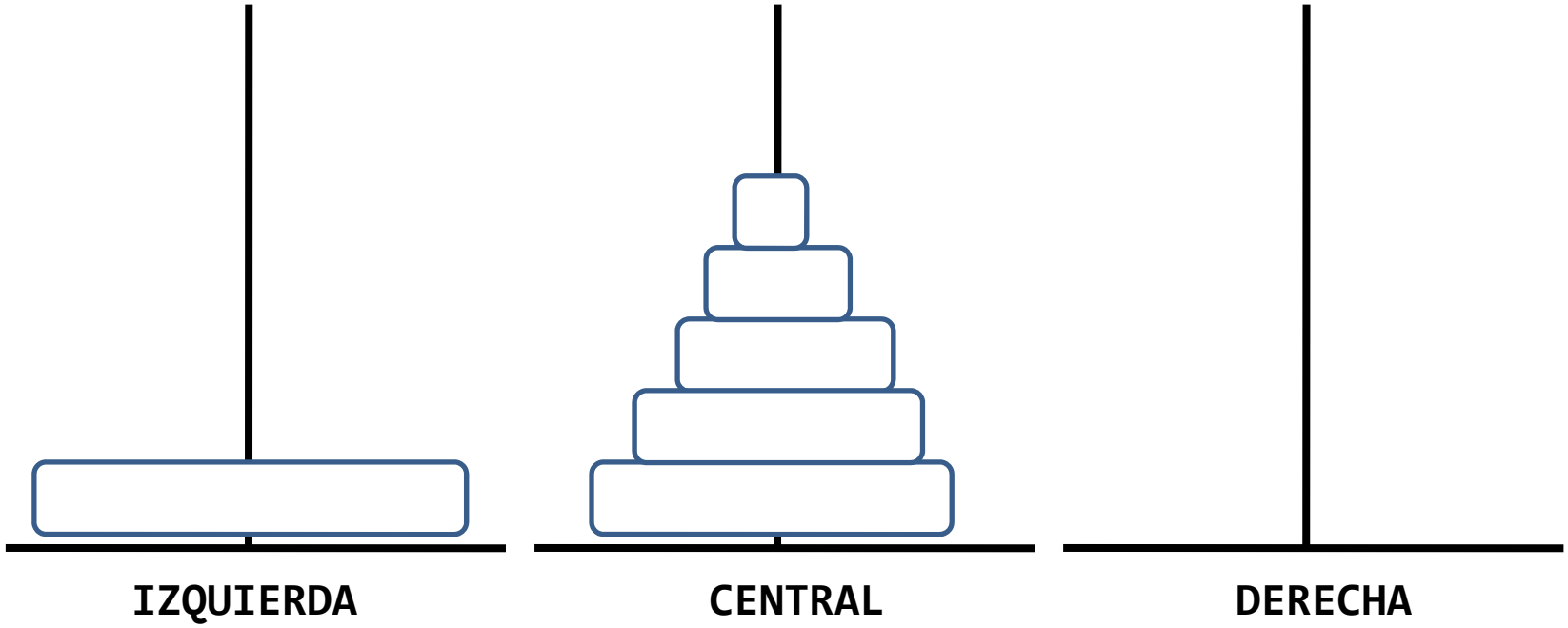
¿Y si el número de discos a trasladar fuera “n”?

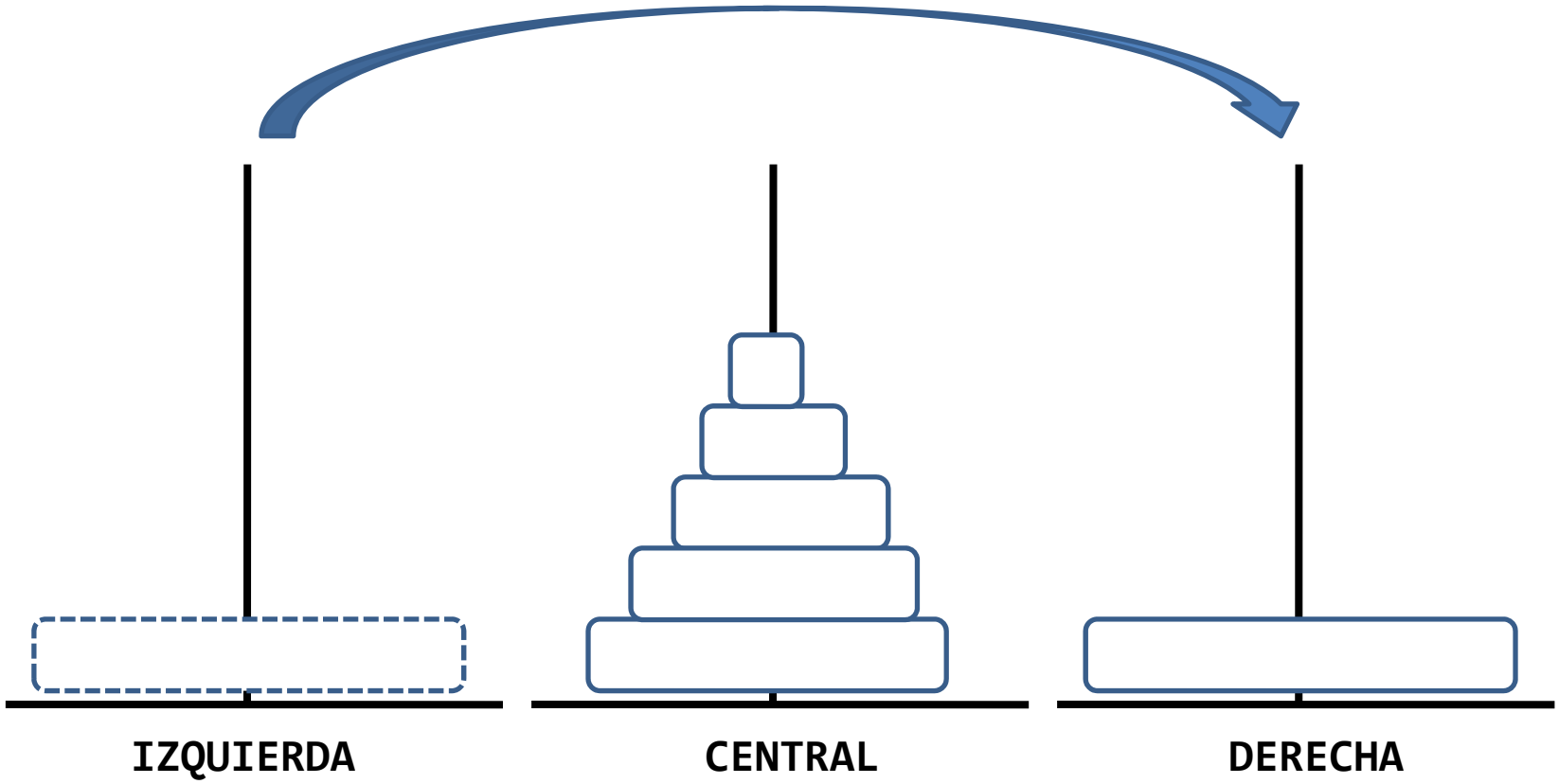


1. Movemos [n-1] discos de IZQUIERDA a CENTRAL

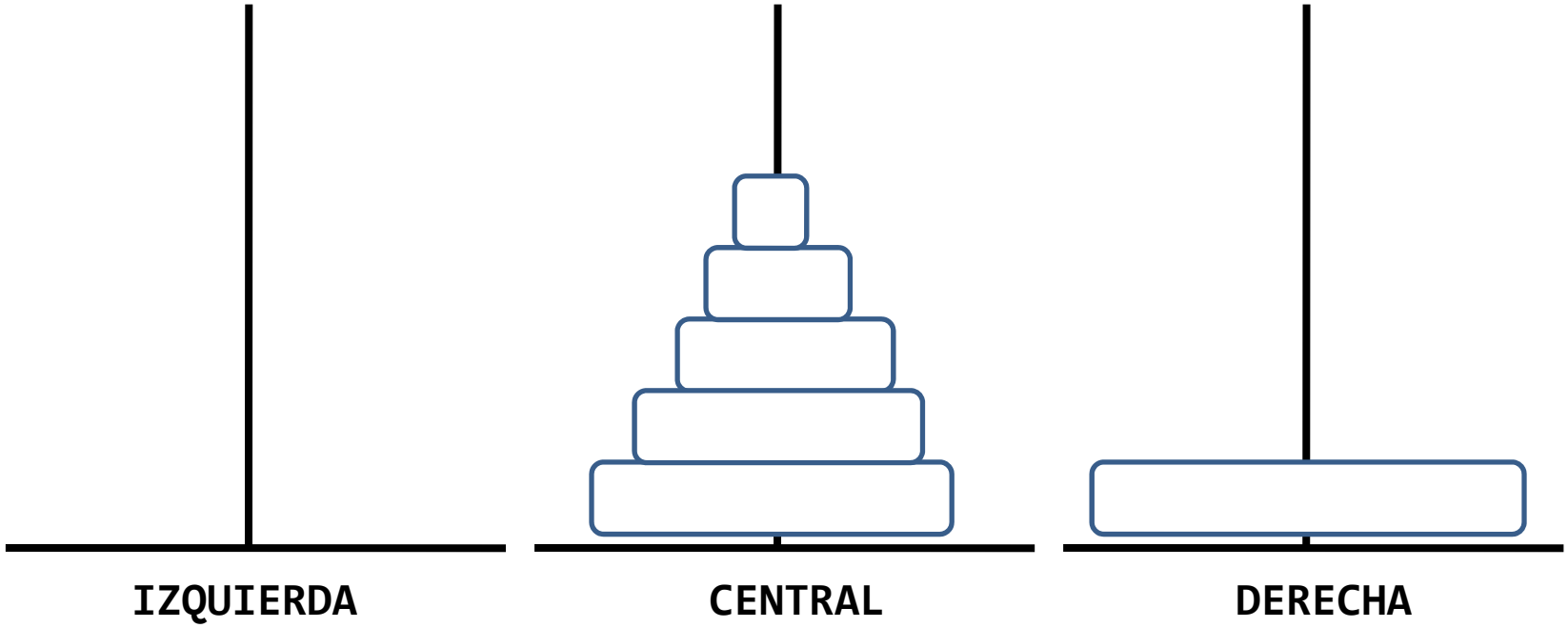


2. Movemos el disco que queda en IZQUIERDA a DERECHA

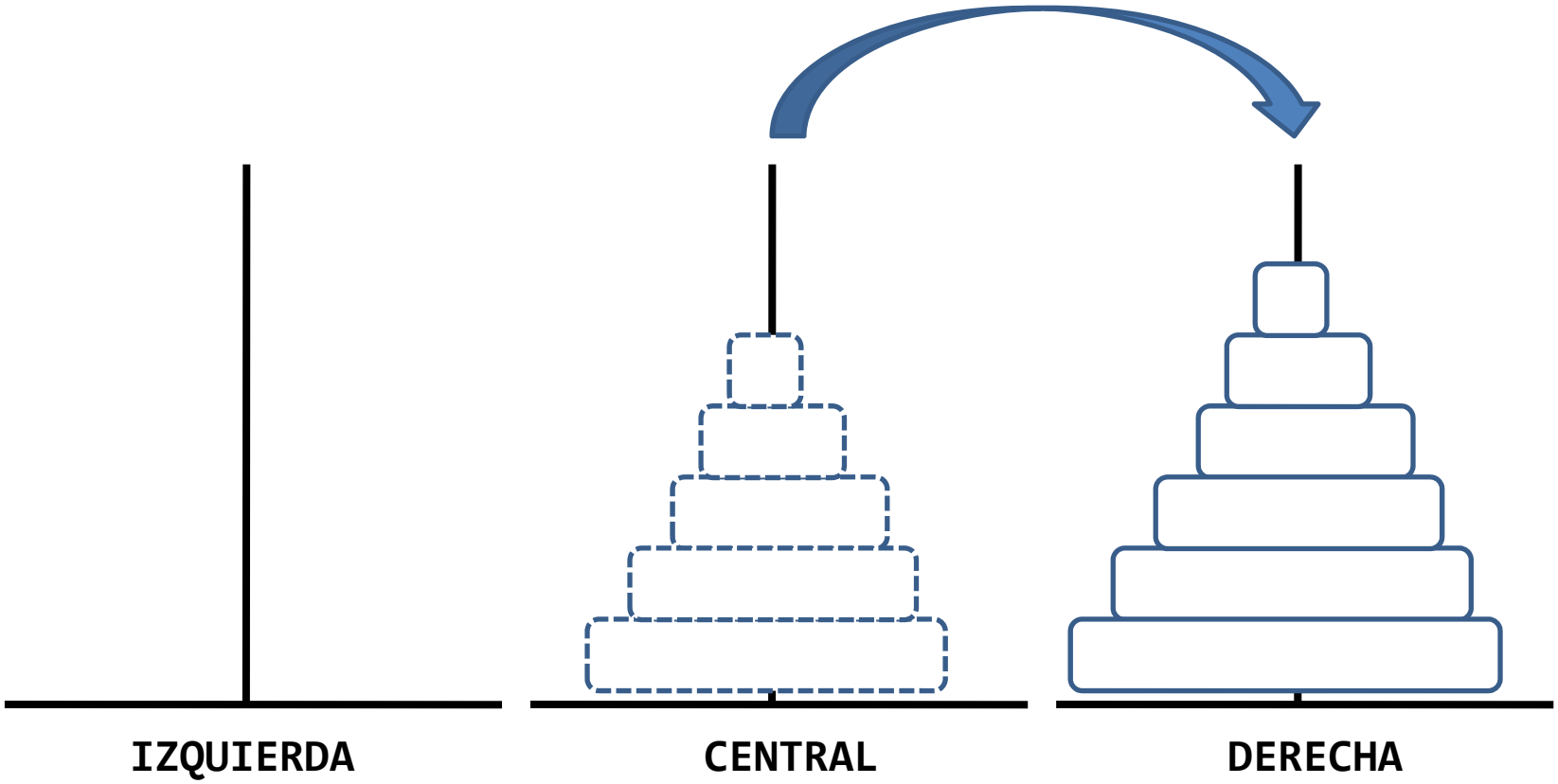



IZQUIERDA
CENTRAL
DERECHA

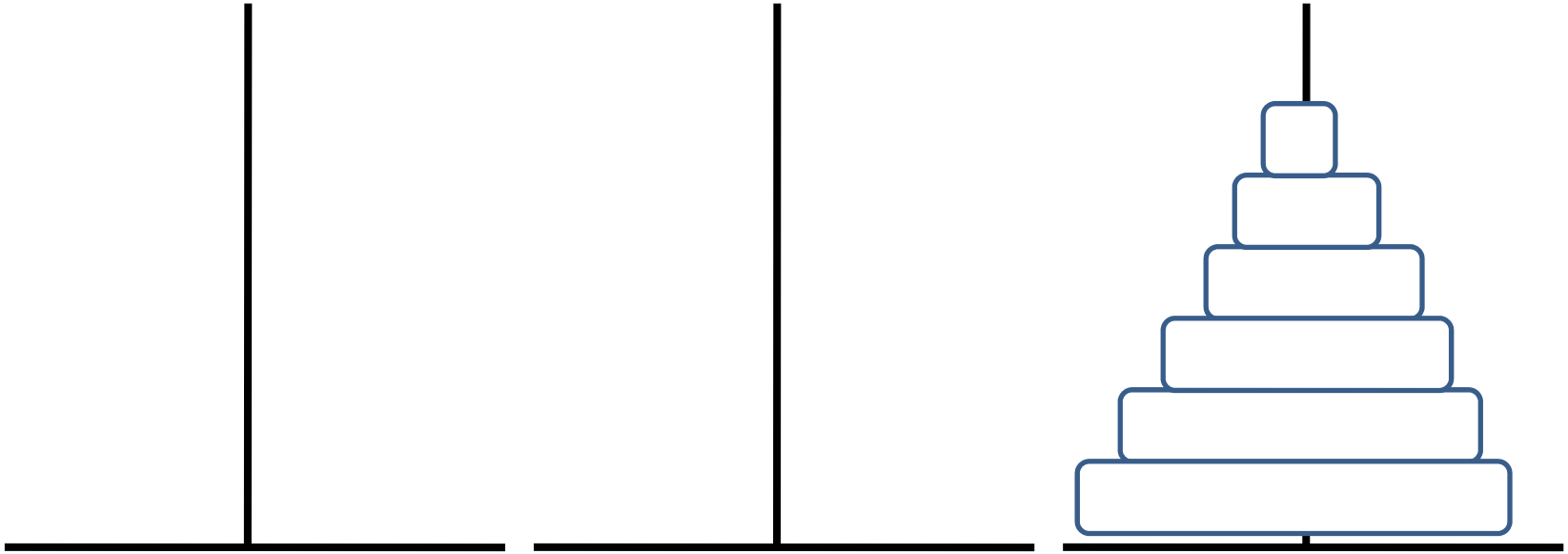
¿Y ahora qué?



3. ¡ Y movemos los [n-1] discos de CENTRAL a DERECHA !



3. ¡ Problema resuelto !



Debemos diseñar una función, que al ser invocada:

```
torresHanoi(5, “IZQUIERDA”, “DERECHA”, “CENTRAL”);
```

Presente un listado de los movimientos de discos necesarios para resolver el problema:

```
Mover disco superior de IZQUIERDA y apilarlo en DERECHA
Mover disco superior de IZQUIERDA y apilarlo en CENTRAL
Mover disco superior de DERECHA y apilarlo en CENTRAL
Mover disco superior de IZQUIERDA y apilarlo en CENTRAL
                    .   .   .
Mover disco superior de CENTRAL y apilarlo en IZQUIERDA
Mover disco superior de CENTRAL y apilarlo en DERECHA
Mover disco superior de IZQUIERDA y apilarlo en DERECHA
```

Diseño por inmersión mediante debilitamiento de la postcondición

```
// Pre: n >= 0
// Post: Presenta por pantalla un listado con la secuencia de movimientos
//       de discos a realizar para resolver el problema de las torres de
//       Hanoi para trasladar una pirámide con [n] discos desde la torre
//       "IZQUIERDA" hasta la torre "DERECHA", utilizando como auxiliar
//       la torre "CENTRAL"
void torresHanoi (const int n);
```

```
// Pre: n >= 0
// Post: Presenta por pantalla un listado con la secuencia de movimientos
//       de discos a realizar para resolver el problema de las torres de
//       Hanoi para trasladar una pirámide con [n] discos desde la torre
//       [origen] hasta la torre [destino], utilizando como auxiliar
//       la torre [auxiliar]
void torresHanoi (const int n, const char origen[], const char destino[],
                  const char auxiliar[]);
```

```
// Pre: n >= 0
// Post: Presenta por pantalla un listado con la secuencia de movimientos
//       de discos a realizar para resolver el problema de las torres de
//       Hanoi para trasladar una pirámide con [n] discos desde la torre
//       "IZQUIERDA" hasta la torre "DERECHA", utilizando como auxiliar
//       la torre "CENTRAL"
void torresHanoi (const int n) {
   torresHanoi(n, "IZQUIERDA", "DERECHA", "CENTRAL");
}
```

```
// Pre: n >= 0
// Post: Presenta por pantalla un listado con la secuencia de movimientos
//       de discos a realizar para resolver el problema de las torres de
//       Hanoi para trasladar una pirámide con [n] discos desde la torre
//       [origen] hasta la torre [destino], utilizando como auxiliar
//       la torre [auxiliar]
void torresHanoi (const int n, const char origen[], const char destino[],
                  const char auxiliar[]);
```

```
// Pre: n >= 0
// Post: Presenta por pantalla un listado con la secuencia de
//       movimientos de discos a realizar para resolver el problema
//       de las torres de Hanoi para trasladar una pirámide con [n]
//       discos desde la torre [origen] hasta la torre [destino],
//       utilizando como auxiliar la torre [auxiliar]
void torresHanoi (const int n, const char origen,
                  const char destino[], const char auxiliar[]) {
   if (n > 0) {
      torresHanoi(n - 1, origen, auxiliar, destino);
      cout << "Mover disco superior de " << origen
           << " y apilarlo en " << destino << endl;
      torresHanoi(n-1, auxiliar, destino, origen);
   }
}
```