Derivación de algoritmos iterativos típicos

Evitar bucles anidados.

```
predicate espico(s:seq<int>,i:int)
requires 0<=i<|s|
ensures espico(s,i) == forall k::0<=k<i ==> s[i]>=s[k]

function CountPicos(s:seq<int>,i : int):nat
requires 0<=i<|s| && |s| > 0
ensures i == 0 ==> CountPicos(s,i) == 1
ensures i > 0 && espico(s,i) ==>
    CountPicos(s,i) == 1 + CountPicos(s,i-1)
ensures i > 0 && !espico(s,i) ==>
    CountPicos(s,i) == CountPicos(s,i-1)

method numPicos(v:array<int>) returns (n:int)
requires v!=null && v.Length>0
ensures n==CountPicos(v[..],v.Length-1)
```

Derivación de algoritmos iterativos típicos

```
method numPicos(v:array<int>) returns (n:int)
requires v!=null && v.Length>0
ensures n==CountPicos(v[..],v.Length-1)
{
  var i := 1; var imax := 0;
  n:=1;
  while (i<v.Length)
  invariant 1<=i<=v.Length && 0<=imax<i
  invariant forall l:: 0<=l<i ==> v[imax]>=v[l]
  invariant n==CountPicos(v[..],i-1)
  {
     if (v[i]>=v[imax])
     {
        max:=i;
        n:=n+1;
     }
     i:=i+1;
  }
 }
```

Ejemplos de derivación

Algoritmos con ventana.

Derivar el siguiente algoritmo

```
method Feb2013 (v : array<int>, m : int)
    returns (s : int, p1 : int)
requires v != null
requires 1 <= m <= v.Length
ensures 0 <= p1 < v.Length - m + 1
ensures s == Sum(v[p1..p1+m])
ensures forall u :: 0 <= u < v.Length - m ==>
    Sum(v[u..u+m]) <= s
```

Ejemplos de derivación

```
var i := 0; var sumParcial := 0;
while (i < m)
{ sumParcial := sumParcial + v[i]; i := i + 1;}
s := sumParcial;
p1 := 0;  var j := 0;
while (j < v.Length-m)
{
  sumParcial := sumParcial + v[j+m];
  sumParcial := sumParcial - v[j];
  if (sumParcial > s) {s := sumParcial; p1 := j+1;}
  j := j + 1;
}
```

Ejemplos de derivación

La moda es el valor que mas veces aparece repetido en una colección.

Derivar el siguiente algoritmo:

```
function CountIguales(s:seq<int>, x : int):nat
  ensures s == [] ==> CountIguales(s,x) == 0
  ensures s != [] && s[0] == x ==>
       CountIguales(s,x) == 1 + CountIguales(s[1..], x)
  ensures s != [] && s[0] != x ==>
       CountIguales(s,x) == CountIguales(s[1..],x)

method moda (a : array<int>) returns (m : int)
requires a != null && a.Length > 0
ensures forall k::0<=k<a.Length==>
  CountIguales(a[..],a[k]) <= CountIguales(a[..],m)
```

Derivación de algoritmos iterativos típicos

1. Primera solución ordenar el vector

```
ordenar(a);
var i := 1; m := a[0]; var f := 1; var fm := 1;
while (i < a.Length) {
  if (a[i] == a[i-1]){
    f := f+1;
    if (fm < f) {
      fm:=f; m := a[i];
    }
  }
  else f := 1;
  i := i+1;
}
```

Derivación de algoritmos iterativos típicos

1. Segunda solución, utilizar vector para acumular los valores. Se crea un vector `a` con las apariciones de cada elemento y después se busca el máximo del vector `a`.

```
var k := 0; var i := 0;
while ( i < a.Length) {a[i] := 0; i := i+1;}
while (k < v.Length) {
  a[v[k]] = a[v[k]] + 1;
  k = k + 1;
}
```

Segmento de longitud máxima.

Indicar la subsecuencia mas larga de ceros de una cadena numérica.

```
predicate igualesCero (s : seq<int>)
ensures igualesCero(s)==forall x::0<=x<|s|==> s[x]==0

method intervaloMax (a : array<int>)
    returns (ini : int, fin:int)
requires a != null && a.Length > 0
ensures 0 <= ini <= fin < a.Length
ensures igualesCero(a[ini..fin])
ensures forall i,j::0<=i<=j<a.Length &&
    igualesCero(a[i..j]) ==> fin-ini>=j-i
```

Derivación de algoritmos iterativos típicos

```
{
  var i := 0; var iniUltSeg := 0; var maxLong := 0;
  ini := 0;
  while(i < a.Length)
  {
    if (a[i]==0) {
      if (maxLong < i-iniUltSeg+1) {
        ini := iniUltSeg;
        maxLong := i-iniUltSeg+1;
      }
    }
    else  {iniUltSeg:=i+1;}
    i:=i+1;
  }
  fin:= ini + maxLong -1;
}
```

Segmento de longitud máxima.

problema de calcular el mayor número de personas que han nacido en un periodo de p años si tenemos los años de nacimiento de cada una de ellas ordenados de menor a mayor. La especificación de este problema es.

```
method dedos (a : array<int>, p : int) returns (r : int)
requires a != null && p > 0 && a.Length > 0
requires forall k1,k2::0<=k1<k2<a.Length==>a[k1]<=a[k2]
ensures forall i,j::0<=i<=j<a.Length && a[j]-a[i]<p
==> r>=j-i+1
ensures exists i,j::0<=i<=j<a.Length && a[j]-a[i]<p
==> r == j-i+1
```

Derivación de algoritmos iterativos típicos

```
{
var k := 1; r := 1; var ini := 0;
while (k < a.Length)
{
  if (a[k] - a[ini] < p) {
    if (r < k-ini+1) {r := k-ini+1;}
  }
  else {while (a[k] - a[ini] >= p) ini := ini+1;}
  k:= k + 1;
  }
}
```

Algoritmo de partición.

Dado un vector definido entre dos índices a y b, el problema pide separar las componentes del vector, dejando en la parte izquierda aquellos valores que sean menores o iguales que el valor de la componente $v[a]$ y en la parte derecha aquellos valores que sean mayores o iguales que el valor de dicha componente. El valor de $v[a]$ debe quedar separando ambas partes.

Este algoritmo se utiliza en la implementación del algoritmo de ordenación rápida (*quicksort*).

```
method particion (v:array<int>, a:int, b:int)
    returns (p : int)
requires v != null
requires 0 <= a <= b< v.Length
ensures 0 <= a <= p <= b < v.Length
ensures forall x::a <= x < p==> v[x] <= v[p]
ensures forall y::p+1 <= y <= b==> v[y] >= v[p]
modifies v
```

Derivación de algoritmos iterativos típicos

Algoritmo de partición.

```
var i := a+1;
var j := b;
while (i <= j)
{
  if (v[i] > v[a] && v[j] < v[a])
  {
    var aux := v[i]; v[i] := v[j]; v[j]:= aux;
    i:=i+1; j:=j-1;
  }
  else if (v[i] <= v[a]) {i:=i+1;}
  else if (v[j] >= v[a]) {j:=j-1;}
}
p:= j;
var aux := v[a];
v[a]:=v[p];
v[p]:=aux;
}
```

Derivación de algoritmos iterativos típicos

Mezcla de dos vectores ordenados. Dados dos vectores ordenados obtener un tercer vector con la mezcla ordenada.
Un algoritmo semejante, pero sobre el mismo vector se utiliza en la implementación del algoritmo de ordenación por mezclas (*mergesort*).

```
method mezcla (v1 : array<int>, v2 : array<int>)
   returns (v : array<int>)
requires v1 != null && v2 != null
requires v1.Length > 0 && v2.Length > 0
requires forall u,w::0<=u<w<v1.Length==> v1[u] < v1[w]
requires forall u,w::0<=u<w<v2.Length==> v2[u] < v2[w]
ensures v != null
ensures forall u::0<=u<v.Length-1==> v[u] < v[u+1]
ensures forall k::0<=k<v1.Length ==> v1[k] in v[..]
ensures forall k::0<=k<v2.Length ==> v2[k] in v[..]
```

Derivación de algoritmos iterativos típicos

```
  v := new int[v1.Length + v2.Length];
  var i := 0; var j := 0; var k := 0;
  while (i<v1.Length && j < v2.Length)
  {
    if (v1[i] < v2[j]) {v[k] := v1[i]; i := i+1;}
    else if (v2[j] < v1[i]) {v[k] :=v2[j]; j:=j+1;}
    else {v[k] := v1[i]; i:=i+1; j:=j+1;}
    k := k+1;
  }
  while (i < v1.Length)
  {
    v[k]:= v1[i]; i:=i+1; k:=k+1;
  }
  while (j < v2.Length)
  {
    v[k]:= v2[j]; j:=j+1; k:=k+1;
  }
}
```

Algoritmos de matrices.

Comprobar si los elementos de una matriz son positivos:

```
method MatricesPositivas(m:array2<int>) returns (s:bool)
requires m != null;
ensures s==forall i,j::0<=i<m.Length0&&0<=j<m.Length1
    ==>m[i,j]>0
```

- Las matrices son de tipo `array2`
- Las dimensiones son `Legth0` y `Length1`
- Los elementos de la matriz son `m[i,j]`
- Se utilizan dos bucles anidados en el algoritmo. Para cada bucle se define su invariante.

```
method MatricesPositivas(m:array2<int>) returns (s:bool)
requires m != null;
ensures s==forall i,j::0<=i<m.Length0&&0<=j<m.Length1
  ==>m[i,j]>0
{ var k1 := 0; s := true;
  while(s && k1 < m.Length0)
  invariant 0 <= k1 <= m.Length0
  invariant s==forall i,j::0<=i<k1&&0<=j<m.Length1
      ==> m[i,j]>0
  { var k2 := 0;
    while (s && k2<m.Length1)
    invariant 0 <= k2 <= m.Length1
    invariant 0 <= k1 < m.Length0;
    invariant s == forall j::0<=j<k2==>m[k1,j]>0
    {
      s := m[k1,k2]>0;
      k2 := k2 + 1;
    }
    k1 := k1 + 1;
  }
}
```