Problema 18. Especifica y deriva un algoritmo de coste lineal que calcule el número de miradores de un vector.

Especificación:

$P \equiv \{N > 0\}$ → se pide un vector no vacío.

fun num-miradores(int X[], int N) return ~~m~~ int m

$Q \equiv \{m = \#\, i : 0 \leq i < N : \text{es-mirador}(X, i)\}$

donde

es-mirador$(X, i) \equiv \forall j : i < j < N : X[i] > X[j]$

Derivar el algoritmo

1.- Obtener el invariante y la ~~postco~~ condición del bucle a partir de la postcondición. Debilitamos el extremo izquierdo de la postcondición porque para desarrollar el bucle necesitamos información de la parte derecha del vector.

$$Q \equiv \{\underbrace{m = \#\, i : n \leq i < N : \text{es-mirador}(X, i)}_{I} \wedge \underbrace{n = 0}_{\neg B}\}$$

2.- Inicialización de variables.

Se puede inicializar $n = N$ con lo que el rango del operador de conteo es vacío y por lo tanto $m = 0$.

Como la propiedad es-mirador incluye un $\forall$ podemos suponer que necesitaremos un acumulador para obtener un coste lineal en el algoritmo. Puede ser interesante que el rango del operador de conteo no sea vacío en la inicialización. Como el vector es no vacío (precondición) ~~po~~ también es ~~válid~~ correcta la inicialización

$n = N-1$
$m = 1$ ya que la propiedad es-mirador$(X, N-1)$ es cierta por ser el rango del $\forall$ vacío.

3.- Función cota e instrucción de avance.

La inicialización de la variable n es $N-1$ y la condición de parada del bucle $n != 0$. Por lo tanto la instrucción de avance debe hacer decrecer n. La función cota es $\boxed{t(n) = n}$ la instrucción de avance $\boxed{n = n-1}$
No se puede decrementar la variable en una cantidad mayor porque hay que comprobar todas las componentes del vector.

4.- Instrucción para restablecer el invariante.

El algoritmo obtenido hasta este momento es:

$n = N-1$; $m = 1$.
while ~~(n != 0)~~ $(n != 0)$ { // también podríamos utilizar $n > 0$ ya que la precond. nos garantiza $N > 0$
...
$R \equiv \{ m = \#i : n-1 \leq i < N : \text{es-mirador}(X, i) \wedge \dots \}$
$n = n-1$
$I \equiv \{ m = \#i : n \leq i < N : \text{es-mirador}(X, i) \wedge \dots \}$
}

$$R \equiv \{ m = \# i : n \le i < N : es\text{-}mirador(\bar{X}, i) + \begin{cases} 1 & si\ es\text{-}mirador(\bar{X}, n-1) \\ 0 & si\ \neg es\text{-}mirador(\bar{X}, n-1) \end{cases}$$

Para restablecer el invariante necesitamos una instrucción que sume 1 al contador cuando es-mirador $(\bar{X}, n-1)$ y no cambie el contador cuando $\neg$es-mirador $(\bar{X}, n-1)$.

$\boxed{\text{if (es-mirador } (\bar{X}, n-1)) \quad m = m+1 .}$

La condición es-mirador $(\bar{X}, n-1)$ requiere mirar todas las componentes desde n hasta N comprobando si hay alguna mayor que $X[n-1]$.

Si lo comprobamos en cada vuelta del bucle el coste del algoritmo será cuadrático. Pero no hay que comprobar con todas las componentes, basta con compararla con la mayor. Almacenamos en una variable p el valor mayor entre n y N en cada vuelta del bucle.

$\boxed{p = max\ j : n \le j < N : X[j]}$

Inicialización de la variable p.

Si $n = N$ el rango es vacío y el operador máximo no está definido. Esta inicialización de n no es válida

Si $n = N-1$ $\boxed{p = X[n]}$ o $p = X[N-1]$

Instrucción para restablecer el valor de p en el invariante.

$$R \equiv \{ m = \# i : n-1 \le i \ldots\ldots \wedge p = max\ j : n-1 \le j < N : X[j] \}$$

~~p = max j~~:

Si $X[n-1] > p$ entonces $X[n-1] = \max j : n-1 \leq j < N : X[j]$

Si $X[n-1] <= p$ entonces $p = \max j : n-1 \leq j < N : X[j]$.

Debemos cambiar el valor de p por $X[n-1]$ cuando $X[n-1] > p$ y dejarlo igual en caso contrario.

~~Para~~ ~~la~~

La instrucción sería:

```
if (X[n-1]>p) p = X[n-1];
```

La instruccion if (es-mirador(X, n-1)) {m=m+1} se puede implementar como

```
if (X[n-1]>p){m=m+1;}
```

Suponiendo $p = \max j : n \leq j < N : V[j]$, es decir, en p ya hemos recuperado el invariante que habia sido modificado por la función de avance.

El algoritmo sería en este momento:

```
n=N-1; m=1; p = X[n];
while (n!=0){
    if (X[n-1]>p){m=m+1;}
    if (X[n-1]>p){p = X[n-1];}
    n=n-1
}
```

Integramos las 2 instrucciones condicionales, ya que la condición es la misma.

```
n= N-1; m=1; p=X[n];
while (n!=0) {
    if (X[n-1]>p) {m=m+1; p=X[n-1]; }
    n=n-1
}
```