3.

a)

Algoritmo iterativo para calcular el máximo común divisor de dos números a y b, donde la función `abs` devuelve el valor absoluto de un número, `max` devuelve el más grande entre dos números y `min` el más chico entre dos números.

```
 1: def mcd(a, b):
 2:     a = abs(a)
 3:     b = abs(b)
 4:     x = max(a, b)
 5:     y = min(a, b)
 6:     while y != 0:
 7:         temp = x
 8:         x = y
 9:         y = temp % y
10:     return x
```

b)

Sea $gdc(a, b)$ el máximo común divisor de a y b. Para demostrar la correctitud del algoritmo de este algoritmo se usarán las siguientes propiedades[1]:

1. gcd(a, b) = gcd($|a|$, $|b|$)
2. gcd(a, b) = gcd(b, a)
3. gcd(a, b) = $|a|$ si y sólo si a—b. En particular, si b=0, entonces gcd(a, b) = $|a|$.

Por la propiedad 1 se puede restringir el algoritmo a calcular el máximo común divisor entre dos números no negativos. En las líneas 2 y 3 se obtiene el valor absoluto de a y b antes de empezar a calcular el máximo común divisor.

Teorema 1: Para cualesquiera enteros no negativos a y b, se cumple que gcd(a, b) = gcd(b, a mód b).[2]

Se requiere demostrar que el algoritmo $mcd(a, b)$ devuelve $gcd(a, b)$ y se demostrará usando invariante de ciclo. La invariante de ciclo a demostrar es la siguiente: En cualquier iteración se cumple que $gcd(a, b) = gcd(x, y)$.

Inicio.

Antes de entrar al *while* x tiene el valor más grande entre a y b, mientras que b tiene el más pequeño entre ellos, y por la propiedad 2 se tiene que en este caso $gcd(a, b) = gcd(x, y)$.

Mantenimiento del invariante.

H.I. Suponga que existe una iteración en la cual $gcd(a, b) = gcd(x, y)$ y se supone que el ciclo todavía no ha terminado.

Sean x' y y' los valores de las variables x y y respectivamente en la siguiente iteración. Se tiene que $temp = x$, $x' = y$ y $y' = temp$ mód $y = x$ mód y. Por

[1] Estas propiedades se obtuvieron del libro de *Álgebra superior* de Carmen Gómez Laveaga.

[2] Este teorema se obtuvo de *Introduction to algorithms* de Thomas Cormen et al.

el Teorema 1 se tiene que $gcd(x', y') = gcd(y, x \mod y) = gcd(x, y)$. Por la hipótesis se tiene que $gcd(a, b) = gcd(x, y)$, así que $gcd(a, b) = gcd(x', y')$.

Terminación.

El ciclo termina cuando y vale 0. En este punto se devuelve el valor actual de x, pero por la propiedad 3 se tiene que $x = gcd(x, 0) = gcd(x, y)$. Por la invariante de ciclo se tiene que $gcd(a, b) = gcd(x, y) = x$, al final del ciclo while. Así, el último valor de x (el que se devuelve en la línea 10) será igual al máximo común divisor de a y b, y por lo tanto, $mcd(a, b) = gcd(a, b)$. ■

c)

Ahora se analizará la complejidad de este algoritmo. Primero, observe que siempre se mantiene la propiedad $x \geq y$. Inicialmente, x y y tienen el valor más grande y el valor más chico entre a y b, respectivamente. En una iteración x toma el valor que tenía y, mientras que y toma el valor de $x \mod y$. Entonces, en dos iteraciones x cambia su valor por $x \mod y$. Se demostrará que $x \mod y < x/2$.

- Caso 1: $y \leq x/2$. Se sabe que $x \mod y < y$, y como $y \leq x/2$ entonces $x \mod y < x/2$.
- Caso 2: $y > x/2$. Se tiene que $x \geq y$ y $x \mod y \leq x - y$. Como $y > x/2$, entonces $x - y < x/2$, así que $x \mod y \leq x - y < x/2$.

Con esto se tiene que cada dos iteraciones x cambia su valor a $x \mod y$, lo que significa que x reduce su valor a menos de la mitad cada dos iteraciones. Esto ocurre hasta que y se vuelve 0, mientras que x conserva un valor mayor o igual a 1. Por lo tanto, el número de operaciones realizadas es $O(\log(\text{máx}(a, b)))$; o lo que es lo mismo, es lineal al número de bits de máx(a, b).

4.

a)

Para resolver el problema se pueden tener dos apuntadores, digamos i y f. i comienza apuntando al inicio del arreglo mientras que f empieza apuntando al final. Se pregunta si el elemento en la posición i es igual al elemento en la posición f. Si son iguales entonces el apuntador i avanza una posición, mientras que f retrocede una posición; si son diferentes se termina el algoritmo y se devuelve falso. Se realiza esta comprobación iterativamente y el algoritmo termina en alguno de los siguientes casos:

- Tanto i como f llegan a la mitad del arreglo y entonces se devuelve verdadero.
- En algún punto a_i es diferente de a_f y entonces se devuelve falso.

Se dirá que un arreglo B es *reflejo* de un arreglo A si y sólo si la reversa del arreglo B es igual al arreglo A. Esto es, $A[i] = B^r[i]$ para $0 \leq i < n$, donde B^r es la reversa de B.[3].

[3]Esto es sólo notación que se usará para este ejercicio. Desconozco si en algún libro se define así.

Lo que hace este algoritmo es comprobar que la segunda mitad del arreglo sea un reflejo de la primera mitad. El siguiente es un código en Python que implementa este algoritmo.

```
1:  def esPalindromo(A):
2:      i = 0
3:      f = len(A) - 1
4:      mitad = len(A) // 2
5:      while i < mitad:
6:          if A[i] != A[f]:
7:              return False
8:          i += 1
9:          f -= 1
10:     return True
```

b)

Se definirá $A[k...l]$ como un subarreglo de A, donde $A[k...l]$ contiene a todos los elementos de A desde el índice k hasta el índice l, donde $k \leq l$, y si $k > l$ se dirá que el subarreglo es vacío. Para simplificar el lenguaje se va a decir que el inicio del arreglo es la izquierda y el final es la derecha. Se demostrará su correctitud usando invariante de ciclo, donde la invariante es la siguiente:

Al inicio del ciclo *while*, el subarreglo $A[0...i-1]$ es un reflejo del subarreglo $A[f+1...n-1]$.

O dicho de otra forma: todo lo que está a la izquierda del índice i es un reflejo de lo que está a la derecha del índice f en el arreglo A.

Inicio.

Por simplicidad se usará n para denotar la longitud del arreglo ($len(A) = n$). Al inicio del ciclo en la primera iteración se tiene que i vale 0 (el primer índice del arreglo) y f vale $n-1$ (el último índice del arreglo). El subarreglo $A[0...i-1]$ es vacío y el subarreglo $A[f+1...n-1]$ también es vacío, pues no hay nada a la izquierda de i y tampoco hay nada a la derecha de f. Como ambos subarreglos son vacíos y la reversa de un arreglo vacío es un arreglo vacío, entonces se cumple la invariante.

Mantenimiento de la invariante.

Supongamos que para ciertos valores de i y f se cumple la invariante y que el ciclo no ha terminado. Sean i' y f' los valores de las variables en la siguiente iteración; esto es, $i' = i+1$ y $f' = f-1$. Si $A[i]$ es diferente a $A[f]$ entonces se devuelve False, lo que significa que A no es palíndromo. Ahora se analizará el caso en el que $A[i'-1=i]$ es igual a $A[f'+1=f]$. Suponiendo que $SAizq$ es un subarreglo izquierdo y $SAder$ es un subarreglo derecho, se tiene que:

- $SAizq = A[0...i-1] = [a_0, a_1, ..., a_{i-2}, a_{i-1}]$
- $SAder = A[f+1...n-1] = [a_{f+1}, a_{f+2}, ..., a_{n-2}, a_{n-1}]$
- $SAizq^r = [a_{i-1}, a_{i-2}, ..., a_1, a_0]$
- $SAizq' = A[0...i'-1] = [a_0, a_1, ..., a_{i-1}, a_i]$

- $SAder' = A[f'+1...n-1] = [a_f, a_{f+1}, ..., a_{n-2}, a_{n-1}]$

La reversa del arreglo $SAizq'$ es $[a_i, a_{i-1}, ..., a_1, a_0]$. Por hipótesis, se cumple que $SAizq^r = SAder$, esto es, $a_{i-1} = a_{f+1}$, $a_{i-2} = a_{f+2}$, ..., $a_1 = a_{n-2}$, $a_0 = a_{n-1}$. En este caso se sabe que $a_i = a_f$ y por la hipótesis se tiene que $SAizq'^r = SAder'$, es decir, el subarreglo $A[0...i'-1]$ es reflejo del subarreglo $A[f'+1...n-1]$, lo que significa que se cumple la invariante de ciclo.

Finalización.

El ciclo termina cuando $i = mitad$. Si n es par entonces $f = mitad - 1$, en otro caso $f = mitad$. Como se cumple la invariante de ciclo, esto significa que el subarreglo $A[0...mitad-1]$ es reflejo del subarreglo $A[mitad...n-1]$ para el caso de n par. O bien, se cumple que $A[0...mitad-1]$ es reflejo de $A[mitad+1...n-1]$ si n es impar.

El algoritmo comprueba correctamente si la primera mitad del arreglo es igual a la reversa de la segunda mitad. Ahora hay que comprobar que si se cumple esa condición entonces $A = A^r$. Para ello se usará la siguiente propiedad de la reversa: Si A y B son arreglos, entonces $(A++B)^r = B^r++A^r$, donde $++$ es el operador para concatenar dos arreglos.

Sea PM el subarreglo formado con la primera mitad de A y SM el subarreglo formado con la segunda mitad. Supongamos que se cumple que $PM^r = SM$ (y, evidentemente, también se cumple $PM = SM^r$). Entonces se tiene que $A^r = (PM++SM)^r = SM^r++PM^r = PM++SM = A$. Con esto se tiene que el algoritmo comprueba correctamente si un arreglo es palíndromo. ■

c)

El conteo de los ciclos se puede ver con el número de incrementos de la variable i, la cual empieza en 0, incrementa en 1 en cada iteración y termina cuando su valor es $n/2$. Así, el ciclo while se ejecuta a lo más $n/2$ veces y dentro del while se realizan un número constante de operaciones: una comprobación, un autoincremento y un autodecremento. Por lo tanto, la complejidad de este algoritmo es $O(n)$.

5.

a)

El algoritmo se describe de la siguiente forma:

Inicialmente se tiene un apuntador al primer índice del arreglo (izq) y un apuntador al último índice (der). Se calcula el índice que está a la mitad de izq y der y se verifica alguno de los siguientes casos:

1. x es igual a $A[mitad]$: en este caso se considera que el elemento fue encontrado y se devuelve $mitad$.

2. x es menor que $A[mitad]$: en este caso se realiza búsqueda pero ahora en el subarreglo $A[izq...mitad-1]$.

3. x es mayor que $A[mitad]$: en este caso se realiza la búsqueda en el subarreglo $A[mitad+1...der]$.

El algoritmo termina en alguno de los siguientes casos:

1. Ya se encontró a x en el arreglo. En este caso se devuelve el índice donde está x.

2. El subarreglo $A[izq...der]$ es vacío, lo que significa que $izq > der$. En este caso se devuelve -1 porque un elemento no puede estar en un arreglo vacío.

Las siguiente función *busquedaBinaria* resuelve el problema de búsqueda. Utiliza como auxiliar la función *bbRec*.

```
1: def busquedaBinaria(A, x):
2:     return bbRec(A, x, 0, len(A)-1)
```

```
1: def bbRec(A, x, izq, der):
2:     if izq > der:
3:         return -1
4:     mitad = (izq+der) // 2
5:     if A[mitad] == x:
6:         return mitad
7:     elif x < A[mitad]:
8:         return bbRec(A, x, izq, mitad-1)
9:     else:
10:        return bbRec(A, x, mitad+1, der)
```

b)

Lo que hace la función *bbRec* es buscar un elemento en el subarreglo $A[izq...der]$, donde izq y der son números que recibe como parámetros. Se demostrará la correctitud de la función *bbRec* por inducción.

Caso base: Se considerarán dos casos base: uno en el que x sí está y otro en el que x no está. Se considerará que el caso base es cuando $A[izq...der]$ tiene tamaño 1, es decir, cuando $izq = der$.

Subcaso 1: x sí está en el subarreglo. Como el subarreglo $A[izq...der]$ tiene tamaño 1 y se está suponiendo que x sí está ahí, entonces necesariamente $x = A[izq]$. Como $izq = der$, entonces $mitad$ es igual a $(izq + der)/2 = (2izq)/2 = izq$. Como $izq = mitad$ entonces x es igual a $A[mitad]$ y la comprobación de la línea 5 es verdadera; lo que significa que devuelve el índice $mitad = izq$, y en este caso se devuelve el resultado correcto.

Subcaso 2: x no está en el subarreglo. Por lo descrito en el subcaso 1, se está analizando solamente un suarreglo de tamaño 1, (al elemento $A[mitad]$). Como se está suponiendo que x no está en $A[izq...der]$ y este subarreglo solo tiene un elemento, entonces se tiene que $x \neq A[mitad]$.

Subcaso 2.1: Si $x < A[mitad]$ se hace una llamada recursiva a la función con parámetros izq y $mitad - 1$; o dicho de otra forma, en la siguiente llamada el valor de der será $mitad - 1$, mientras que izq no cambia. Como se tenía (en el caso de tamaño 1) que $izq = der = mitad$, entonces en la siguiente llamada der será igual a $izq - 1$ y se hará verdadera la condición de la línea 2. Por lo

tanto, se devolverá -1, que significa que x no está, y en este caso se devuelve el resultado correcto.

Subcaso 2.2: Si $x > A[mitad]$ se hace una llamada recursiva con parámetros $mitad + 1$ y der. Se tendrá entonces que izq será igual a $der + 1$ y se cumplirá la condición de la línea 2. Por lo tanto se devolverá -1 y entonces el resultado es correcto.

H.I. Sea $n = der - izq + 1$ (el tamaño del subarreglo). Supongamos que la función *bbRec* resuelve el problema correctamente para un subarreglo de tamaño menor o igual a $n/2$. Se demostrará que *bbRec* también devuelve el resultado correcto para un subarreglo de tamaño n.

Primero observe lo siguiente: $mitad - 1 - izq + 1 = (izq + der)/2 - izq = izq/2 + der/2 - izq = der/2 - izq/2 = (der - izq)/2 \leq n/2$. La última desigualdad se sigue de que $der - izq + 1 = n \therefore (der - izq + 1)/2 = n/2 \therefore (der - izq)/2 \leq n/2$. Con esto se tiene que el subarreglo $A[izq...mitad - 1]$, donde se hace la llamada recursiva de la línea 8, tiene tamaño menor o igual a $n/2$. Haciendo un análisis similar se obtiene que el subarrelgo $A[mitad + 1...der]$, donde se hace la búsqueda recursivamente en la línea 10, tiene tamaño menor o igual a $n/2$.

Se van a considerar tres subcasos:

Subcaso 1: x es igual a $A[mitad]$. En este caso, la condición de la línea 5 se cumple y se devuelve $mitad$, por lo tanto para este caso es correcto.

Subcaso 2: x es menor a $A[mitad]$. En este caso se cumple la condición de la línea 7 y se devuelve el resultado de aplicar recursivamente la función bbRec al subarreglo $A[izq...mitad - 1]$ (línea 8). Si x está en $A[izq...der]$, como el arreglo A está ordenado entonces, x está en $A[izq...mitad - 1]$ y por la H.I. $bbRec(A, x, izq, mitad - 1)$ devuelve el índice donde se encuentra x. Si x no está en A, por H.I., $bbRec(A, x, izq, mitad - 1)$ devuelve -1.

Subcaso 3: Si x es mayor a $A[mitad]$ se hace un análisis similar al caso anterior, pero con $bbRec(A, x, mitad + 1, der)$.

La primera llamada a la función se hace en la función *busquedaBinaria* desde el índice 0 hasta el índice $len(A) - 1$, es decir, todo el arreglo. Por lo tanto, la función *busquedaBinaria* devuelve el resultado correcto de buscar x en todo el arreglo A.

c)

Inicialmente, el espacio de búsqueda es $len(A) - 1 - 0 + 1 = len(A)$, es decir, el tamaño del arreglo. Ya se analizó que si $n = der - izq + 1$, entonces $der - (mitad + 1) + 1 \leq n/2$ y $(mitad - 1) - izq \leq n/2$. Esto quiere decir que en cada llamada recursiva se reduce el espacio de búsqueda a la mitad hasta que se encuentra a x o el tamaño del espacio de búsqueda se hace 0. Por lo tanto, la complejidad del algoritmo es $O(\log n)$, donde n es el tamaño de A.