Análisis de la implementación del algoritmo para números primos

27 de Noviembre del 2017

Equipo 2:

María Luisa Borrego.	(1837482)
Rafael Enrique Almaraz.	(1579795)
Artemio Guajardo Aparicio.	(1590417)
Ana Cecilia García.	(1749760)
Haydeé Judith Arriaga.	(1659539)
José Manuel Tapia Avitia.	(1729372)

Sabemos que en matemáticas, un número primo es un número natural mayor que 1 que tiene únicamente dos divisores distintos: él mismo y el 1. Por el contrario, los números compuestos son los números naturales que tienen algún divisor natural aparte de sí mismos y del 1, por lo tanto, pueden factorizarse. El número 1, por convenio, no se considera ni primo ni compuesto. Anteriormente se trabajó con el algoritmo para obtener números primos, en esta ocasión toca analizar los resultados de ésta implementación aplicando ciertas restricciones con el fin de observar la complejidad de las operaciones realizadas según sea la restricción.

1. Implementación del algoritmo

Como se mencionó en la introducción, las condiciones para que un número sea primo son que sea mayor a 1 y que sea únicamente divisible entre 1 y él mismo, pero además se conoce una propiedad que dice que si un número no tiene divisores menores o iguales que su raíz cuadrada, entonces es un número primo. Estas condiciones fueron tomadas en cuenta para implementar el algoritmo, obteniendo una complejidad de $\mathcal{O}(\sqrt{n})$, donde n es el número que se busca saber si es primo o no.

La implementación del algoritmo es la siguiente:

```
def itsPrime(n):
    if n<=1:
        return False
    m=2
    while m*m<=n:
        if n%m==0:
            return False
        m+=1
    return True
```

Para fines del proyecto, de igual forma se implementó una variante de la función anterior, que en vez de retornar si un número es primo o no, regresa la cantidad de operaciones necesarias para determinar si cierto número n es primo o no.

```
def esPrimo(n):
    cnt=0
    if n<=1:
        return cnt
    m=2
    while m*m<=n:
        cnt+=1
        if n%m==0:
            return cnt
        m+=1
    return cnt
```

Para realizar las gráficas de la siguiente sección, se uso la siguiente función que recibe los valores que se tienen para cada uno de los ejes, el nombre y el título de dicha gráfica.

```
def graficar(xs,ys,imagen,titulo):
    x=np.linspace(1,100000)
    plt.plot(x,x**(0.5),label="Raiz de n",color="green")
    plt.plot(xs,ys, label=titulo)
    plt.xlabel("Numero")
    plt.ylabel("Cantidad de operaciones")
    plt.title(titulo)
    plt.legend()
    plt.savefig(imagen)
    plt.show()
```

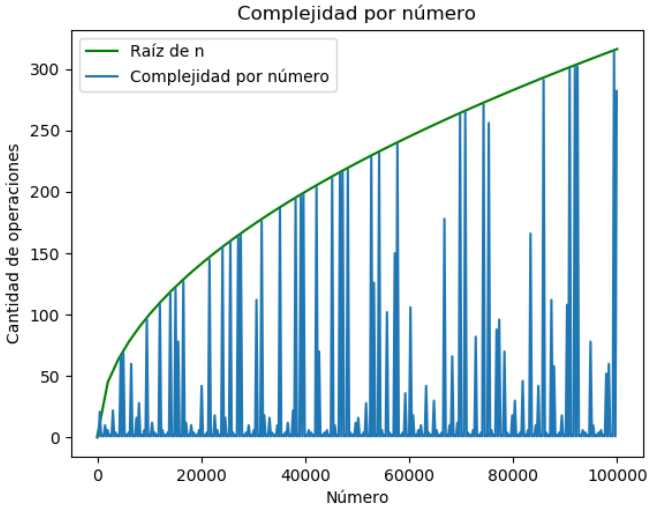
En donde, para llamar a dicha función se realizaron los siguientes precálculos:

```
G1xs=range(1,100001,251)
G1ys=[esPrimo(x) for x in G1xs]
G2xs=range(1,50001,251)
G2ys=[esPrimo(x) for x in G2xs]
G3xs=range(50000,100001,251)
G3ys=[esPrimo(x) for x in G3xs]
G4xs=[x for x in range(1,100001,251) if x%2==0]
G4ys=[esPrimo(x) for x in G4xs]
G5xs=[x for x in range(1,100001,251) if x%2==1]
G5ys=[esPrimo(x) for x in G5xs]
G6xs=[x for x in range(1,100001,251) if x%2==1 if not itsPrime(x)]
G6ys=[esPrimo(x) for x in G6xs]
G7xs=[x for x in range(1,100001,251) if itsPrime(x)]
G7ys=[esPrimo(x) for x in G7xs]
graficar(G1xs,G1ys,"Numeros1al100000.png","Complejidad por numero")
graficar(G2xs,G2ys,"Numeros1al50000.png","Complejidad por numero")
graficar(G3xs,G3ys,"Numeros50000al100000.png","Complejidad por numero")
graficar(G4xs,G4ys,"NumerosPares.png","Complejidad por numero par")
graficar(G5xs,G5ys,"NumerosImpares.png","Complejidad por numero impar")
graficar(G6xs,G6ys,"NumerosImparesnoPrimo.png","Complejidad por numero impa
graficar(G7xs,G7ys,"NumerosPrimos.png","Complejidad por numero primo")
```

2. Análisis de complejidad

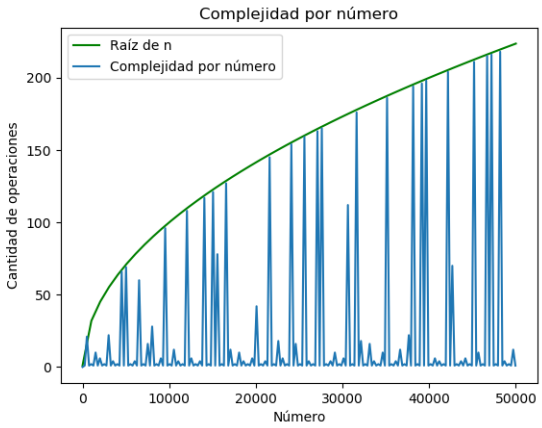
Con el fin de observar mejor el comportamiento de las gráficas, el algoritmo se diseñó para visitar cada 251 números.

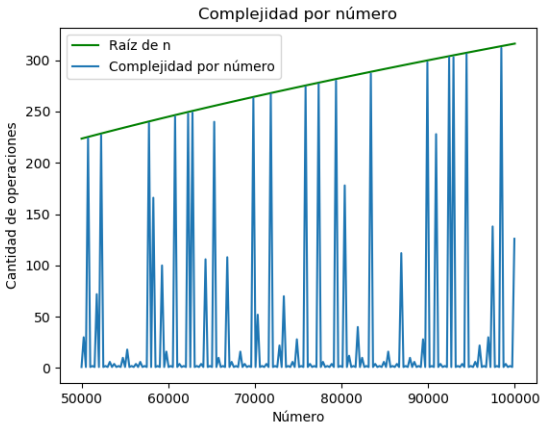
2.1. Primer análisis: Números del 1 al 100,000



En esta gráfica podemos observar que en algunos números el algoritmo realiza una cantidad muy grande de operaciones, esto courre cuando los números son primos pues el algoritmo tiene que verificar si el número es divisible por todos y cada uno de los numeros menores a la raíz. Cuando el algoritmo realiza una menor cantidad de operaciones podemos inferir que se trata de números impares no primos y/o números pares los cuales necesitan la menor cantidad de operaciones debido a que el algoritmo verifica primero si los números son divisibles entre 2.

2.2. Segundo análisis: Números del 1 al 50,000 y del 50,001 al 100,000

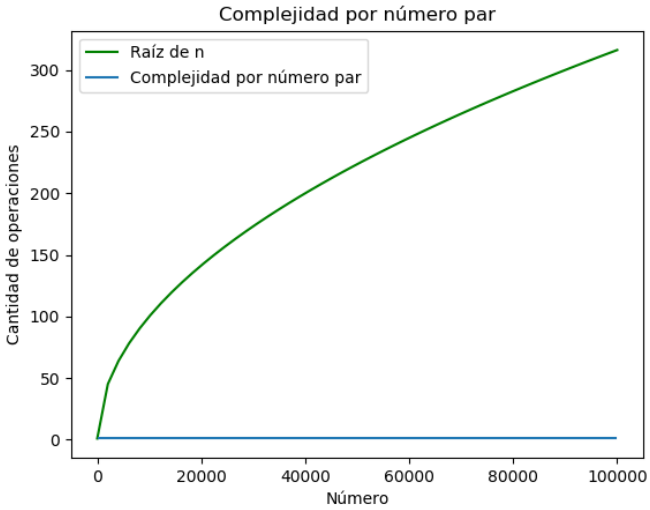




Con estas dos gráficas podemos tomar en cuenta varios factores:

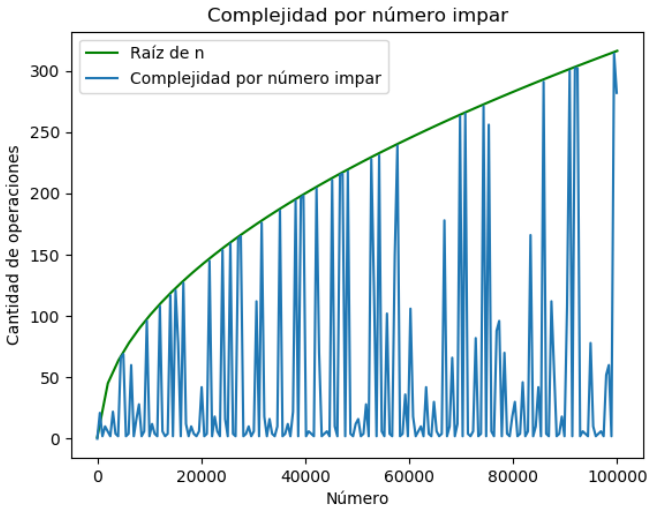
- Se puede ver con mayor precisión la distribución de los números primos
- Hay una mayor cantidad de primos en los primeros 50,000
- La cantidad de operaciones que se realizan en los números del 50,000 al 100,000 es mayor

2.3. Tercer análisis: Números pares



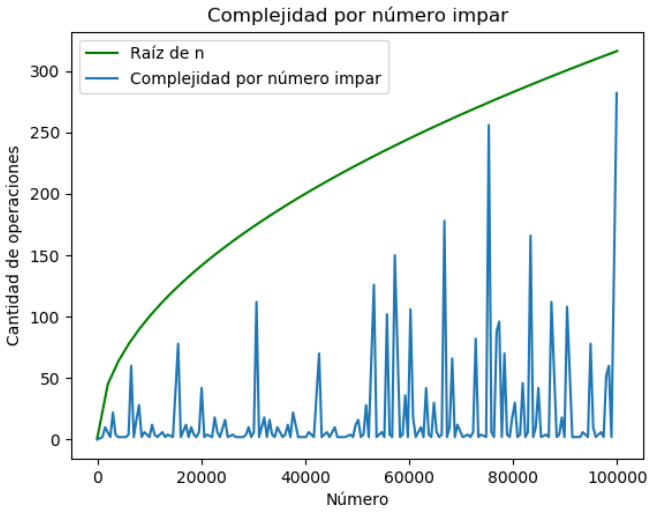
En esta gráfica podemos observar como se dijo en el primer análisis, el algoritmo realiza la cantidad mínima de operaciones debido a que primero se verifica si el número es divisible entre 2, condición que cumple todo número par.

2.4. Cuarto análisis: Números impares

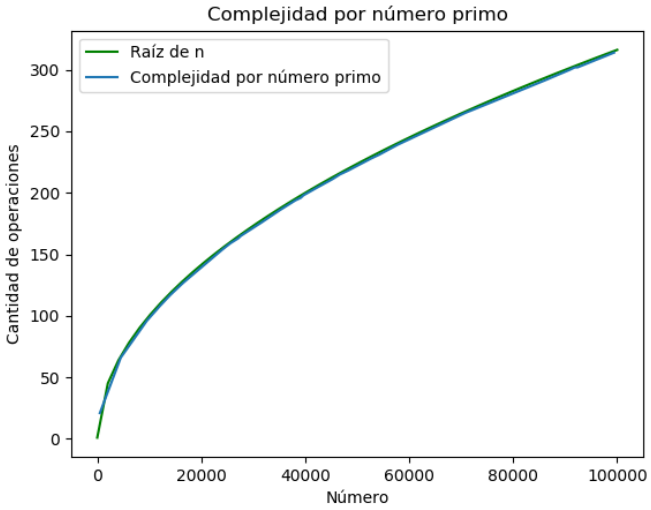


Sabemos que los números primos son impares y debido a eso en esta gráfica podemos observar que se realiza la mayor cantidad de operaciones en algunos números. En los números impares no primos se realiza una gran cantidad de operaciones, mayor a la de los números pares y menor a la de los números primos debido a que el algoritmo tarda más en encontrar un divisor, sin embargo lo encuentra en menos de $\sqrt{n}$ operaciones.

Ésta es la gráfica para los números impares no primos, en la cual verificamos lo mencionado anteriormente.



2.5. Quinto análisis: Números primos



En esta gráfica podemos observar que el algoritmo realiza la máxima cantidad de operaciones para cada número. El algoritmo realiza operaciones para verificar si el número es divisible entre algún número menor a su raíz, lo cual no sucede y ocasiona que el algoritmo visite cada uno de éstos números, por eso es que vemos que la curva que representa las operaciones realizadas está tan cerca de la curva que representa la raíz de cada número.

3. Conclusiones

Interpretando todo lo descrito anteriormente, podemos concluir que la búsqueda de números primos resulta más eficiente si nos apoyamos en herramientas modernas, como lo es el lenguaje de programación Python. En el reporte, esta herramienta nos ayudó a observar con facilidad que, debido a la poca cantidad de divisores, los números primos utilizan una mayor cantidad de operaciones, seguidos de los números impares, y por último los pares. También se observó que, apesar de que los números primos son infinitos, éstos van siendo menos frecuentes conforme van creciendo. Todo lo anterior se observa con facilidad en las gráficas añadidas a lo largo del documento. No obstante, ¿cuál es la funcionalidad de todo esto? Los números primos son muy relevantes en ciertos campos, debido a sus propiedades de factorización. Primeramente, no se puede negar su importancia en Teoría de números, pero además son esenciales en seguridad computacional. Una de las propiedades de los números primos nos indica que, a pesar de ser relativamente fácil encontrar números primos grandes, tratar de factorizar números grandes en números primos es una tarea muy compleja. Es por esto que la criptografía está basada completamente en números primos. Por este motivo las computadoras buscan generar continuamente nuevos números primos, lo cual es de sumo interés ya que el desarrollo moderno termina encontrando una apliación práctica a un área que se había vanagloriado de su inutilidad (Godfrey Harold Hardy). Después de analizar en este reporte a los números primos concluimos que la matemática, por más abstracta e incomprensible que parzca, siempre encontrará un camino hacia el mundo práctico.