P3 Criptografía&Computación - UGR

Logaritmo discreto, Factorización y Raíces modulares

José Manuel García Rodríguez 75936722-Z

Manuel Alejandro Sánchez Molina 75569927-A

Marcel Kemp Muñoz 44651578-E

Víctor García Carrera 51473495-R

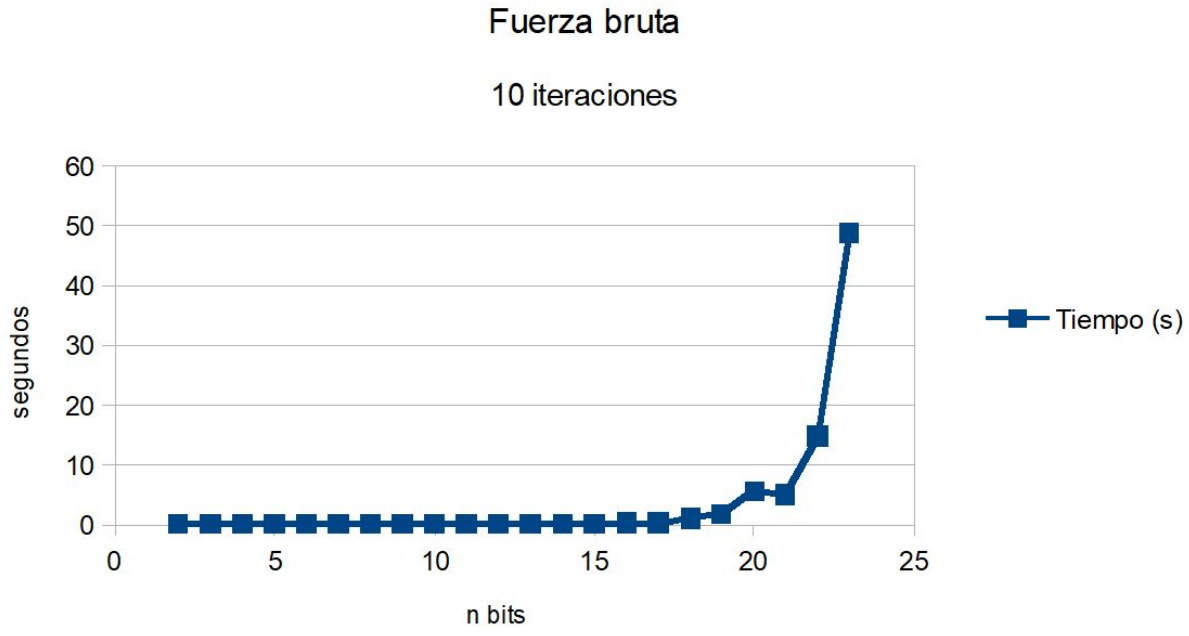
INTRODUCCIÓN

A continuación se detallan el estudio realizado acerca de la complejidad temporal de los diversos algoritmos implementados para la resolución del cálculo del logaritmo discreto (mod Zp : p primo), la factorización de primos y las raíces cuadradas mod Zp primo

RESULTADOS

Logaritmo discreto - Fuerza bruta

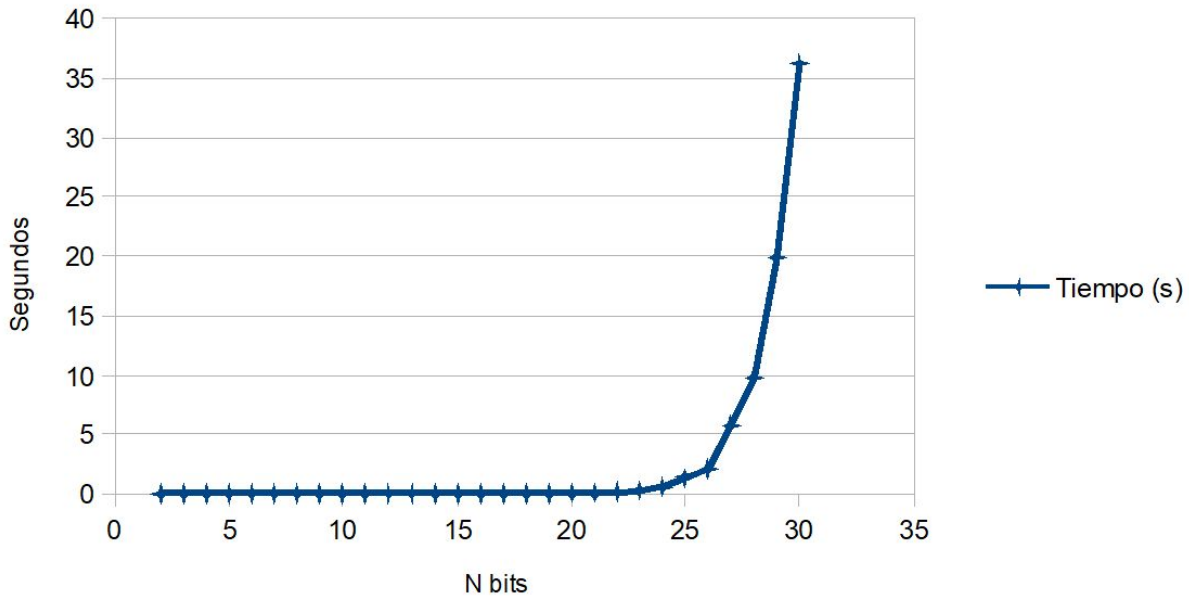
En primer lugar, hemos implementado la función más ineficiente en la cual, pruebas todos los valores de x en el módulo p (número primo), hasta que el resultado de a^x=b. A este método se le llama fuerza bruta, ya que tan solo pruebas valores hasta que salga el correcto. Para comprobar su ineficiencia, hemos realizado una media de tiempos con *a* (entre 2 y p-2) y *b* (entre 1 y p-1) aleatorios, con 10 iteraciones, y cuando ya tardaba demasiado tiempo por cada iteración (más de 30s), le hemos dicho que pare, llegando a ejecutar un primo de 23 bits en un tiempo medio de 49s. Tal y como podemos observar en la siguiente gráfica:



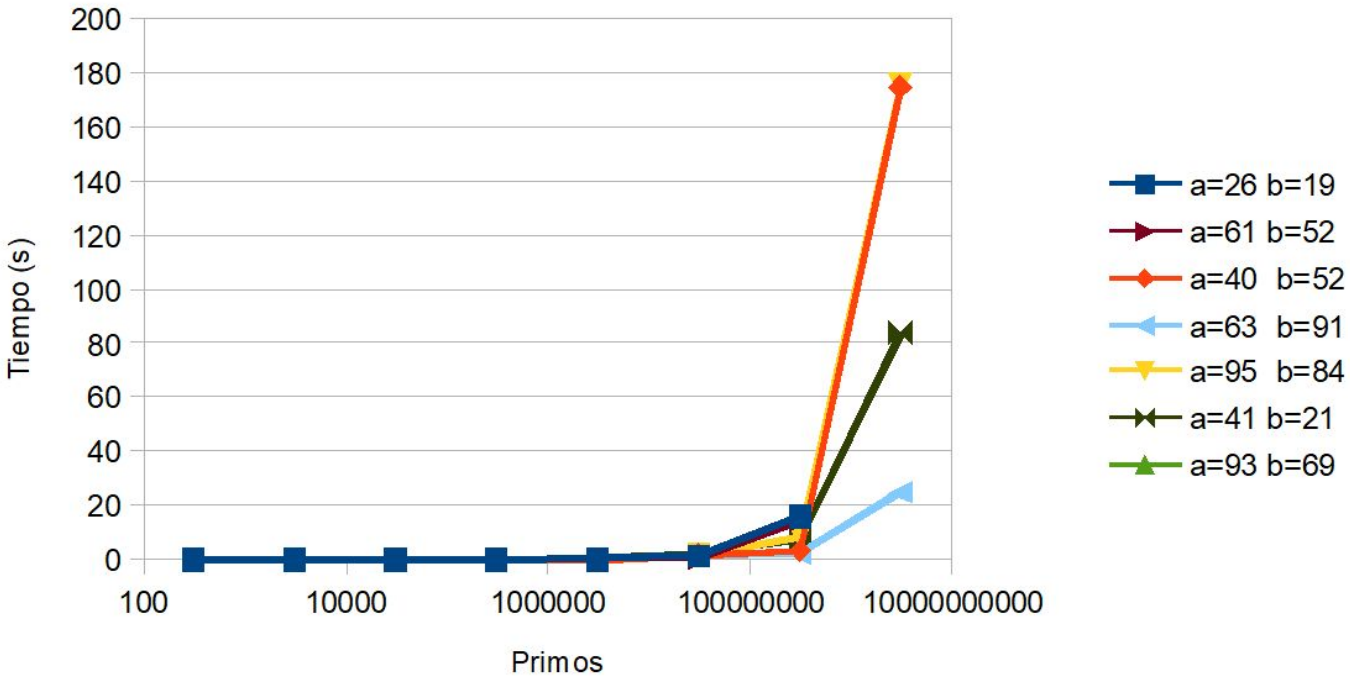
Logaritmo discreto - Paso enano, paso gigante

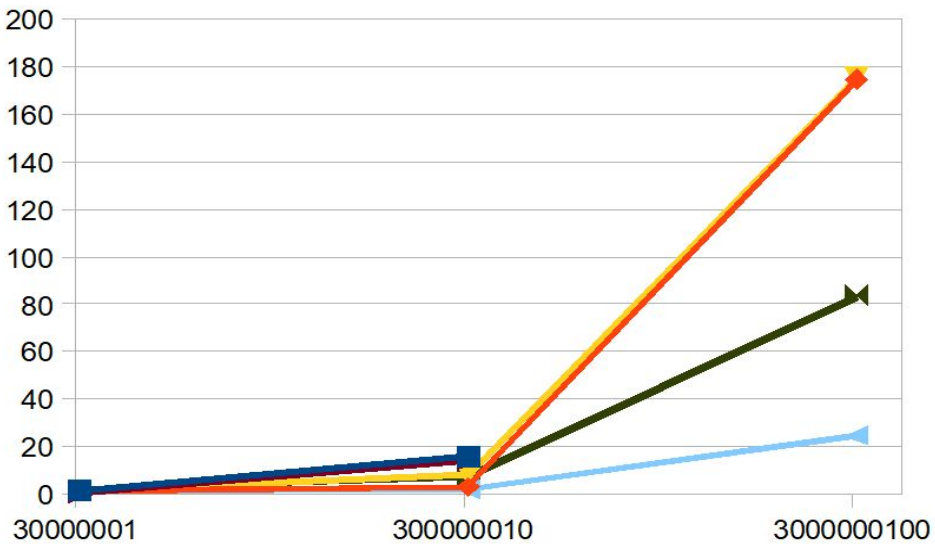
Una vez completado, hemos pasado a realizar el algoritmo de paso enano - paso gigante, en el cual calculamos una tabla S = [c; a*c; a^2*c; ... ; a^r*c; ... ; a^{s-1}*c] y una segunda tabla T= [a^s; a^{2*s}; a^{3*s}; ... ; a^{t*s}; ... ; $a^{s\wedge 2}$], y con ellas vamos a comparar los valores de ambas tablas para ver si existe alguna coincidencia (en la implementación, nos ahorramos crear la tabla T, comprobando los valores de T con los de S conforme vamos obteniéndolos). Tras encontrar alguna coincidencia entre ambas tablas (a^{t*s}=a^r*c), realizamos a^{t*s-r}=c, para así poder sacar el valor *c,* el cual buscamos. En algunos casos no existe coincidencias, por lo cual no existe el logaritmo.

Para comprobar la mejora de eficiencia del algoritmo, conforme al de fuerza bruta, hemos sacado de igual manera una media de tiempos con *a* y *b* aleatorios, con 50 iteraciones (para que nos salga una gráfica sin picos, como veremos luego), y hemos parado la ejecución cuando empezaba a tardar demasiado (más de 30s). De tal forma, el algoritmo ha llegado a ejecutar un primo de 30 bits en un tiempo medio de 32s, mejorando en gran parte los tiempos con el método de fuerza bruta. A continuación podemos observar la gráfica:



Y por último, voy a comentar varios ejemplos donde el algoritmo es muy rápido, o tarda demasiado. En el caso de ser muy rápido, se debe a que encuentra la solución sin tener que recorrer todos lo valores de ambas tablas (oscilando entre que sea en la primera comprobación o en la última, en la cual no habría mejoras), y esto se cumple si a es un número generador (ya que siempre tiene una solución), o en algunos casos si a no es generador (pudiendo no tener solución o tener más de una solución). Y el peor caso, sería cuando no encuentra ninguna solución, ya que debe de recorrer todos los valores de ambas tablas para realizar la comprobación. A continuación os muestro una gráfica donde podemos observar lo explicado previamente, donde si no encontraba la solución antes de 15s, detenemos la ejecución. Podemos observar los valores a y b en la leyenda, y los números primos seleccionados han sido: el siguiente número primo a $(30*10^i)$. Y la segunda gráfica solo es una ampliación de la primera, para observar mejor las diferencias de tiempo.





Como podemos observar, varios valores se han parado en el siguiente primo a $3*10^8$, mientras que otros han llegado a ejecutar el siguiente primo a $3*10^9$, y esto se debe a que los que no han encontrado la solución con el primo de $3*10^8$, han sido detenidos en la ejecución, ya que han tardado un poco más de 15s, mientras que los otros cuatro, han encontrado la solución al primo de $3*10^9$, y dentro de ellos, podemos observar como dos de ellos no han encontrado la solución, llegando a tardar casi 180s, mientras que los otros dos sí la han encontrado, reduciendo así el tiempo en ambos casos.

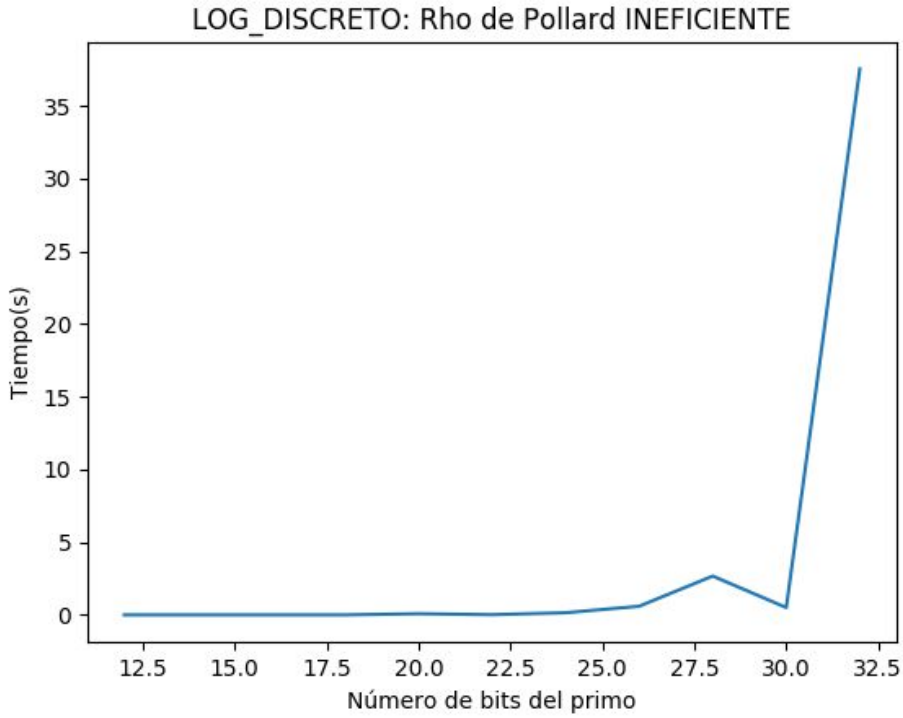
Logaritmo discreto - ρ de Pollard

Este último algoritmo, Ro de Pollard, resulta ser el más eficiente y rápido de los implementados para el cálculo del logaritmo discreto, además de calcular todas las soluciones (nuestra versión implementada, podría ser más rápido al tener que encontrar una única solución). Se basa en construir una sucesión pseudoaleatoria de elementos relacionados con nuestro problema de $a^x=b \bmod p$ y encontrar una coincidencia entre estos, para así resolver una congruencia que nos da las posibles soluciones de nuestro problema original (basta con evaluarlas). Se definen las reglas para construir el siguiente elemento de la sucesión, y trabajamos con 2 maneras diferentes de encontrar una coincidencia entre los elementos de la sucesión, dando lugar a 2 implementaciones distintas, que llamaremos PollardINEFICIENTE y Pollard.

Las gráficas y datos obtenidos a continuación resultan de la ejecución con valores aleatorios del problema del logaritmo discreto, tomando los tiempos de aquellos casos en los que encontraba las soluciones.

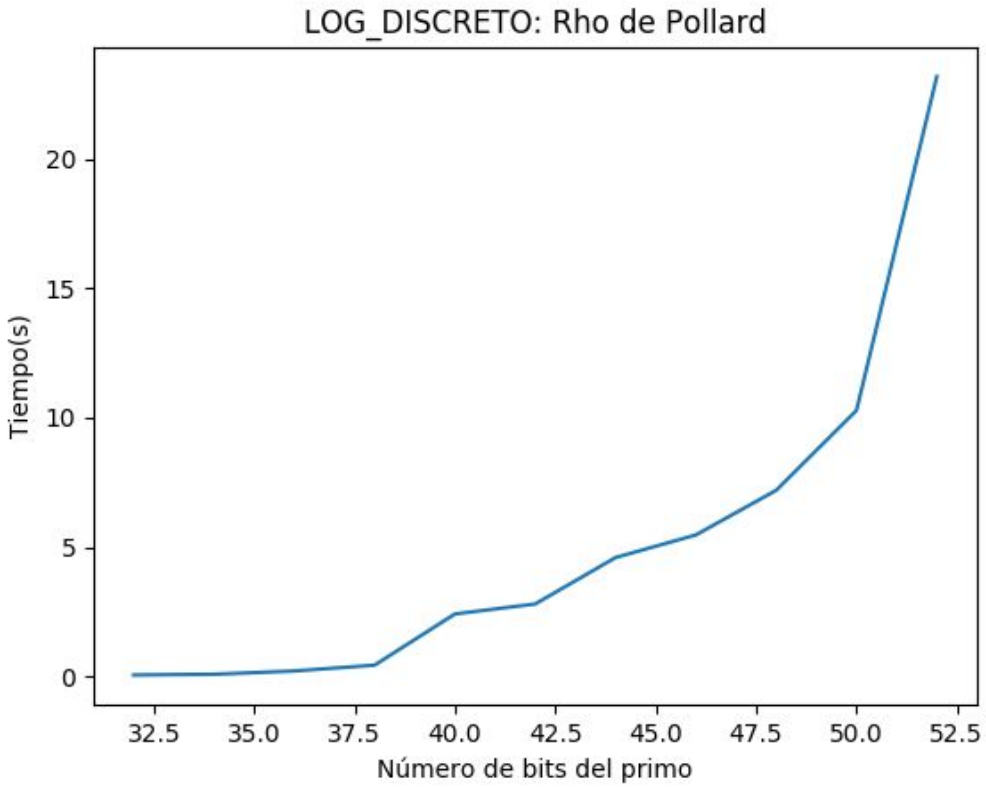
PollardINEFICIENTE: En cada iteración, construye el siguiente término y comprueba si hay coincidencia con alguno de los anteriores calculados. Esto resulta en una complejidad espacial y temporal exponencial.

A pesar de llegar a lograr tiempos razonables alrededor de primos de 30 bits, el tiempo se dispara a partir de ese tamaño. Resulta muy similar al rendimiento del algoritmo paso enano, paso gigante.

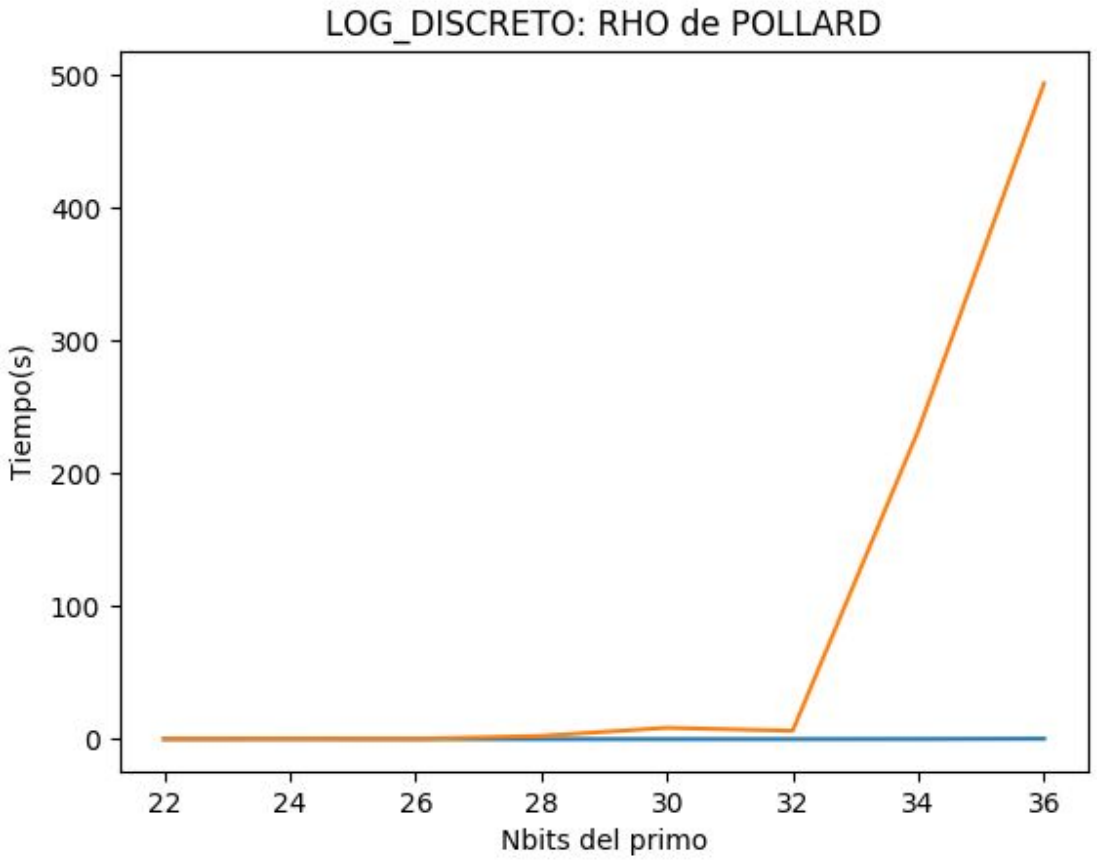


Pollard (Eficiente):

Cabe la posibilidad de recorrer la sucesión de 2 maneras diferentes a la vez, por ejemplo recorriendo sus elementos de 1 en 1 y a su vez recorriéndolos de 2 en 2. En cada iteración se obtiene el elemento *i* y el *2i* de la sucesión, y nos limitamos a encontrar la coincidencia entre estos términos. Esto resulta en una mejora radical en términos de complejidad espacial (ya no es necesario almacenar todos los términos obtenidos en cada iteración anterior) y bastante notable en la complejidad temporal. Sigue sin resultar extremadamente rápido porque es necesario calcular más términos (iterar más veces) para encontrar una coincidencia. Aun así, es el más rápido, y no cabe menospreciarlo en absoluto pues permite obtener todas las soluciones (entre 1 y p-1) del problema del logaritmo discreto módulo p con un primo de 50 bits (unas 16 cifras) en cuestión de segundos, como se muestra a continuación. Sin embargo, presenta un crecimiento exponencial para primos mayores.



Finalmente mostramos una comparativa realizada entre ambas versiones, ineficiente y eficiente. A partir de los 32 bits, el algoritmo ineficiente resulta inviable, mientras que la implementación eficiente se mantiene casi inmutable hasta los 50 bits como mostraba la gráfica anterior.

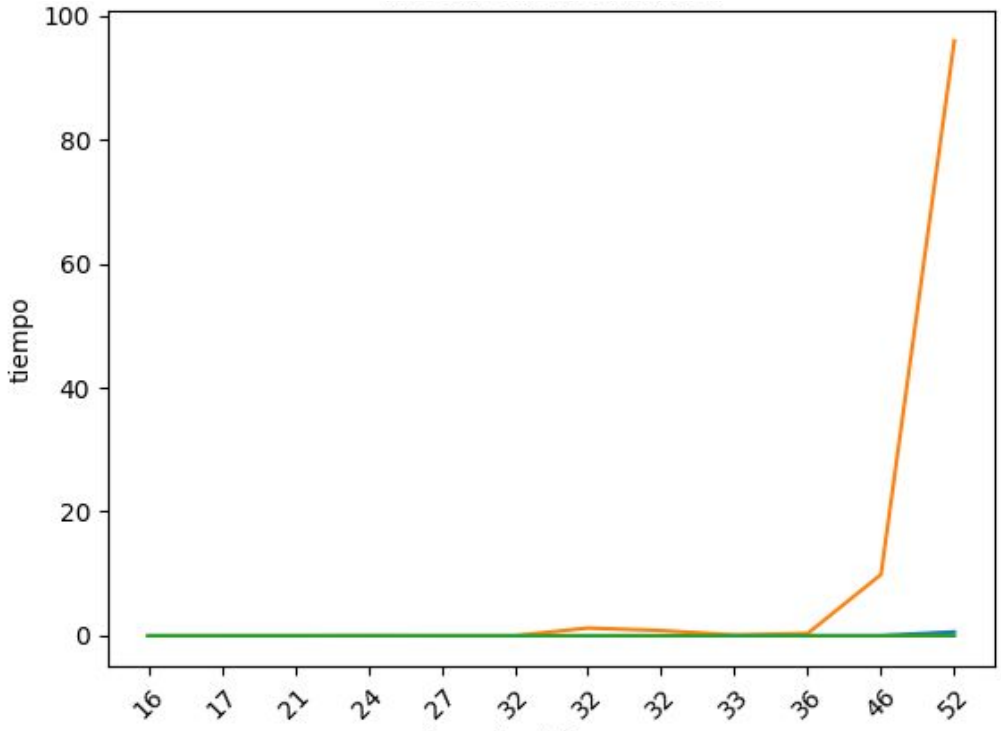


NARANJA - INEFICIENTE

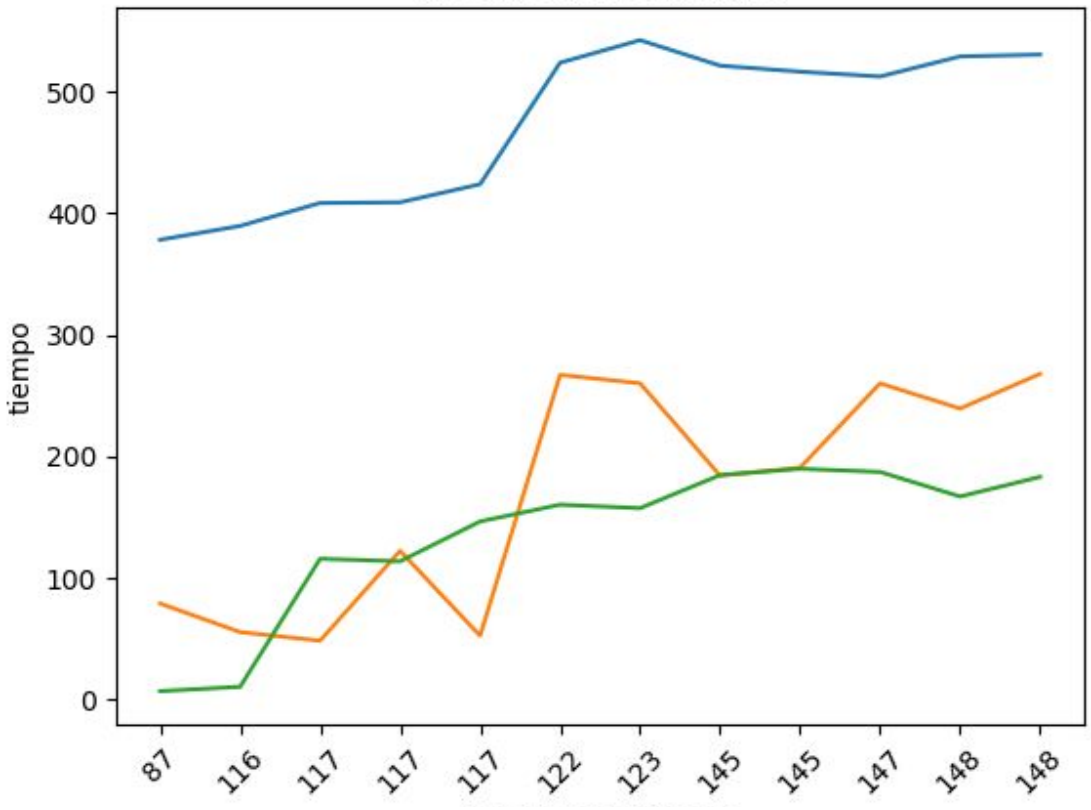
AZUL - EFICIENTE

Factorización

Para la primera prueba he seleccionado un conjunto de 12 números pequeños con factores primos pequeños lo cual favorece al algoritmo de tentativa. Los peores resultados los encontramos con el Fermat(naranja) en este algoritmo dado que si estos números no son cercanos no encuentra la solución y consume todas las iteraciones. Para el caso del rho de Pollard (azul) funciona bastante bien.



En esta segunda prueba he seleccionado un conjunto de números formados solo por productos de dos primos grandes medianamente cercanos. Este conjunto de datos estaba preparado para que el de Fermat(naranja) diera buenos resultados, lo cual ocurre en la gran mayoría de los casos mientras que en algunos agota las iteraciones(picos). El de tentativa(azul) siempre agota las iteraciones, la diferencia de tiempos viene dada por lo que tarda en obtener el siguiente primo cuando los números son más y más grandes. Con el rho de Pollard lo que ocurre es que en los primeros casos consigue encontrar los factores pero a medida que crecen ya solo agota las iteraciones.


tiempo
500
400
300
200
100
0
87
116
117
117
117
122
123
145
145
147
148
148

Raíces cuadradas modulares

Para las pruebas he usado número primos cada vez más grandes, y el número entero a parecidos entre ellos, cuyo símbolo de jacobi sean 1. Se observa que los número usados son muy grandes, y aún así los tiempos son muy pequeños, de hecho, ha habido que realizar el experimento varias iteraciones con el mismo número p y a y luego hacer la media.

En cuanto a la gráfica, se ve que posee un carácter ascendente en el tiempo, pero muy lento, pero en general es capaz de tratar número gigantes en una pequeña cantidad de tiempo.