Cuestión 1. Veces ejecuta bucle ext. de potenciaPerfecta (bases)

$B^2 \leq n$

$\sqrt{B^2} \leq \sqrt{n}$; $\boxed{-\sqrt{n} \leq B \leq +\sqrt{n}}$ Como mucho se hace $\sqrt{n}$ veces

Cuestión 2. Veces ejecuta bucle inter. de potenciaPerfect (exponentes)

$b^{\frac{\log n}{\log b} - 1 + k} > n$ // Como es $n > b^{exp+k}$ para hacer el bucle, sale si $n \leq b^{exp+k}$

$\log b^{\frac{\log n}{\log b} - 1 + k} > \log n$

$\left(\frac{\log n}{\log b} - 1 + k\right) \log b > \log n$ $\qquad nb^{k-1} > n$

$b^k > b$

$\frac{(\log n - \log b + k \log b) \cancel{\log b}}{\cancel{\log b}} > \log n$; $\cancel{\log} n - \log b + k \log b > \cancel{\log} n$

$-\log b + k \log b > 0$; $(k-1) \log b > 0$; $k - 1 > 0$

Como es do while $k > 1$

$\boxed{k > 1}$ $\quad k = 0$, $k = 1$ $\quad k = 2$ $\quad$ 3 veces maximo

$\boxed{k \geq 1}$ $\quad$ Si se aplica $k \geq 1$ son los 3.

Cuestion 3. Complejidad cuestion2 $\boxed{O(3)}$ o $O(cte)$

Cuestión 4. Veces ejecuta bucle ext. calcR $\boxed{\max(\log^5 n, 3)}$ Bibliografía ~~Por mejora de estimación se usa~~ $\cancel{\boxed{\log^2 n}}$

Cuestion 5. max. iteraciones multiplicativeOrder Se hace como mucho $r-1$ veces por la condición $r > k$, por lo que son: $\boxed{\log^5 n - 1}$ $\cancel{\boxed{\log^2 n - 1}}$

Cuestión 6. Complejidad mcd En cuanto a r, se prueban para cada valor de r $\log^2 n$ exponentes ($n^k \neq 1 \bmod n$ con $k \leq \log^2 n$) y como r puede ser a lo mucho $\log^5 n$, el calculo de r es $O(\log^7 n)$ $\quad 7 = 5 + 2$

Para el mcd, gdc, por sí solo es $O(\log n)$ para cada numero desde 2 hasta el r calculado (r++ cada iteracion) por tanto se hará en el peor caso $\log^5 n$ veces. Resulta en $O(\log^6 n)$ $\quad 6 = 5 + 1$

Finalmente $\boxed{O(\log^6 n)}$, ~~con r = log n es O(log³ n)~~

Cuestion 7.
Veces bucle ext.
totient

for (int i=2; n > i², i++) $\rightsquigarrow$ $\overset{\sim \text{veces}}{k} = (i-2)$; $\overset{\sim \text{parte de 2}}{i = k+2}$

$n > i^2$; $\sqrt{n} > i$; $\sqrt{n} > k+2$; $k < \sqrt{n} - 2$

Se ejecuta por tanto $\boxed{\sqrt{n} - 3}$ veces como mucho
$\simeq \sqrt{n}$ $\quad (\sqrt{n} - 3 < \sqrt{n} - 2)$

Cuestión 8.
Veces bucle while
totient
Complejidad totient

Segun internet es $O(\sqrt{n})$ porque si se hace el interior atenua el exterior

while (n mod i == 0) $\quad n = n/i$

En el peor caso podra quitar k veces el factor i a n

$i^k \leq n$; $k \log i \leq \log n$; $k \leq \frac{\log n}{\log i} \simeq \log n$

La complejidad será $O\left((\sqrt{n}-3)\left(\frac{\log n}{\log i}\right)\right) = \boxed{O(\sqrt{n} + \log n)} = O(\sqrt{n})$

Cuestion 9.
maximo $\phi(r)$

Como r será un numero primo tenemos que el numeros de factores con los que no comparte divisores es r−1
Todos los números entre [1, r) enteros que son $\boxed{r-1}$

Cuestion 10.
maximo $\phi(r)$
en terminos n

$\phi(n) = n \prod_{p|n} \left(1 - \frac{1}{p_i}\right)$ Buscamos que se produzcan los menores atenuantes $\left(1 - \frac{1}{p_i}\right)$ que reduciran el n, pero siempre estará sí mismo y nos deja $\cancel{n} \cdot \frac{n-1}{\cancel{n}} = \boxed{n-1}$

$\phi(r) = \cancel{\log^5 n}\left(\frac{\log^5 n - 1}{\cancel{\log^5 n}}\right)$

$= \log^5 n - 1$

Con r igual $\boxed{\log^5 n - 1}$

Operaciones Calculo primalidad $\left\{\begin{array}{l} \log n \text{ multiplicaciones} \\ \text{grado } r \\ \text{coeficientes } \log n \text{ maximo} \end{array}\right\} O(r \log^2 n)$

Paso 1: Potencia perfecta

$n^{1/2}$

$O(cte\sqrt{n}) = O(\sqrt{n})$

$O(\sqrt{n})$

$B^2 \leq n$

$\sqrt{B^2} \leq \sqrt{n}$

$O(cte)$ $k \in \{0,1,2\}$

$k-1 > 0$

```java
public boolean potenciaPerfecta() {
    BigInteger base = BigInteger.valueOf(2);
    BigInteger aSquared;
    do {
        BigInteger result;
        // Se elige power entre el m�ximo entre el logaritmo en base 2 de (n-base)-2 o 1
        int power = Math.max((int) (log() / log(base) - 2), 1);
        int comparison;

        do {
            power++;
            result = base.pow(power);
            comparison = n.compareTo(result); // n ? base^power
        }
        // Se van probando potencias de la base hasta que se tenga un n�mero igual o
        // mayor que n.
        while (comparison > 0 && power < Integer.MAX_VALUE); // n� iteraciones ceil(log_b(n))

        // En el caso de que nuestra potencia sea igual a n, tenemos una potencia
        // perfecta de n, por tanto n NO es primo.
        if (comparison == 0) {
            if (verbose)
                System.out.println(n + " is a perfect power of " + base);
            factor = base;
            n_isprime = false;
            return n_isprime;
        }

        // En caso contrario, no es potencia perfecta, podemos seguir buscando potencias
        // perfectas con otras bases.
        if (verbose)
            System.out.println(n + " is not a perfect power of " + base);

        // Incrementamos la base en 1
        base = base.add(BigInteger.ONE);
        aSquared = base.pow(2);
    }
    // Continuamos si la base al cuadrado no supera al numero que nos planteamos
    while (aSquared.compareTo(this.n) <= 0); // n� iteraciones sqrt(n)
    return false;
}
```

$powes = max\left(1, \frac{\log n}{\log base} - 2\right)$

$base^{power}$

$n > base^{power}$

Salir

base ++

$base^2 \leq n$

Paso 2: Calculo r

$k \to n^k = 1 \bmod n$

$O_r(n) > \log^2 n$

min r

$O(\log^2 n \cdot \log^5 n) = O(\log^7 n)$

Mejor estimación de r.

$O(\log^2 n)$

k en el peor caso es r-1 y como r va de 1 en 1

$k < \log^2 n$

$\equiv r-1 < \log^2 n$

```java
public BigInteger calculoR() {
    double log = this.log();
    double logSquared = log * log;
    BigInteger k = BigInteger.ONE;
    BigInteger r = BigInteger.ONE;
    do {
        r = r.add(BigInteger.ONE);
        if (verbose)
            System.out.println("trying r = " + r);
        k = multiplicativeOrder(r);
    } while (k.doubleValue() < logSquared);
    return r;
}
```

$k = 1$

$r = 1$

r++

$O(\log^5 n)$

Se hace r prueba de valores k en el peor caso

```java
BigInteger multiplicativeOrder(BigInteger r)
{
    // TODO Consider implementing an alternative algorithm http://rosettacode.org/wiki/Multiplicative_order
    BigInteger k = BigInteger.ZERO;
    BigInteger result;

    do
    {
        k = k.add(BigInteger.ONE);
        result = this.n.modPow(k,r);
    }
    while( result.compareTo(BigInteger.ONE) != 0 && r.compareTo(k) > 0);

    if (r.compareTo(k) <= 0)
        return BigInteger.ONE.negate();
    else
    {
        if (verbose) System.out.println(n + " = " + k + " mod " + r + " = " + result);
        return k;
    }
}
```

$O(\log^2 n)$

k++

$result = n^k \bmod r$

Fallo

$n^k \bmod r \neq 1$

$r > k$

k que cumple $n^k = 1 \bmod r$

Prueba exponentes k mientras $k < r$, como r lo maximo es $\log^5 n$

$O(\log^5 n)$

Paso 3: gcd o mcd

$O(\log^5 n \cdot \log n) = O(\log^6 n)$

$i = 2$

$i \leq r \Rightarrow i \leq \log^5 n$

El peor caso Fib consecutivos de manera que la posición en la serie indica los pasos.

$O(\log^5 n)$ Se hace 2 a r; sumo r veces gcd

```
// If 1 < gcd(a,n) < n for some a <= r, output COMPOSITE
for( BigInteger i = BigInteger.valueOf(2); i.compareTo(r) <= 0; i = i.add(BigInteger.ONE) )
{
    BigInteger gcd = n.gcd(i);
    if (verbose) System.out.println("gcd(" + n + "," + i + ") = " + gcd);
    if ( gcd.compareTo(BigInteger.ONE) > 0 && gcd.compareTo(n) < 0 )
    {
        factor = i;
        n_isprime = false;
        return false;
    }
}
```

BigInteger gcd = n.gcd(i); $\rightarrow O(\log n) \rightarrow$ Es conocido

$mcd(n,i) > 1 \quad mcd(n,i) < n$

factor = i; n_isprime = false; return false; } Es compuesto

gcd(a,b) ~ grande ~ pequeño
while $b \neq 0$
$r = a$
$a = b$
$b = r \% b$
return a

$a > b \geq 0 \quad F_{n+2} > F_{n+1} \geq 0$ Fib. consecutivos $O(\log b) = O(\log n)$

Si necesita n pasos $\Rightarrow$ será el n-ésimo F

$b \geq F_{n+1}$; $b \geq \frac{\phi^{n+1}}{\sqrt{5}}$; $\log_\phi \sqrt{5} b \geq n+1$

$n \leq \log(\sqrt{5} b) - 1$

$F_i = \frac{\phi^i}{\sqrt{5}} \simeq \phi^i \quad \phi = \frac{1+\sqrt{5}}{2}$

$\log^{5 \cdot \frac{1}{2}} n + \log \log^5 n \simeq \log^{5/2} n$

Totient

General: $O(\sqrt{n} + \log n)$ Para llamada r: $O(\sqrt{r} + \log r)$

$n > i^2$; $\sqrt{n} > i$

$O(\sqrt{n})$

En el contexto n es r $O(\sqrt{r})$, $O(\log n)$

Va probando divisores posibles

```
BigInteger totient(BigInteger n)
{
    BigInteger result = n;
    for( BigInteger i = BigInteger.valueOf(2); n.compareTo(i.multiply(i)) > 0; i = i.add(BigInteger.ONE) )
    {
        if (n.mod(i).compareTo(BigInteger.ZERO) == 0)
            result = result.subtract(result.divide(i));
        while (n.mod(i).compareTo(BigInteger.ZERO) == 0)
            n = n.divide(i);
    }
    if (n.compareTo(BigInteger.ONE) > 0)
        result = result.subtract(result.divide(n));
    return result;
}
```

$i = 2 \quad n > i^2 \quad i{+}{+}$

$n \bmod i == 0$

$result = result - result / i$ } Quita los nº no válidos

$\hookrightarrow n \bmod i == 0$ } $O(\log n)$ Elimina un factor por completo.

$n = n/i$

$n > 1$

$result = result - \frac{result}{n}$ ¡Afuera el exterior!

$n \bmod i == 0$

$i^k \leq n \quad 2^4 \leq 16 \quad n\ 1 \quad 2^4 \leq 48 \quad 3$

$\log i^k \leq \log n$

$k \log i \leq \log n$

$k \leq \frac{\log n}{\log i} \simeq \log n$

Para $n = r$

$k \leq \frac{\log r}{\log i} \simeq \log r$

$O(\log n) \circ O(\log r) = O(\log \log^2 n)$

$i = 2 \quad 12 > 4$ Entra

$n \bmod 2 == 0$

res $12 = 12 - \frac{12}{2} = 12 - 6 = 6$

$n \quad 12 = 12/2 = 6$

$n \quad 6 = 6/2 = 3$

$i = 3 \quad 6 > 9 \sim$ Sale bucle

$6 > 0$ Entra

res $6 = 6 - 6/3 = 6 - 2 = 4$

12/2
6/2
3/3

$2^2 \cdot 3 = 12$

$\phi(12) = 12 \left(\frac{1}{2}\right)\left(\frac{2}{3}\right)$

$\frac{24}{6} = 4$

6/2
3/3

$2 \cdot 3 = 6$

$\phi(6) = \frac{12}{6} = 2$

1 2 3 4 5

1 2 3 4 5 6 nº
7 8 9 10 11

Valores coprimos con 12
no comparten divisores
cotando 1.

1024 n

512 res (los no pares)

n: 256
128
64
32
16
8
4
2
1

$i^k \leq \log^2 n$

$$\frac{7}{2}+\frac{2\cdot 7}{2}=\frac{14+7}{2}=\frac{21}{2}$$

Condición suficiente

$$O(\log^{7/2} n \cdot \log^{7} n) = O(\log^{21/2} n)$$

```java
// sqrt(totient)*log(n)
int limit = (int) (Math.sqrt(totient(r).doubleValue()) * this.log());
// X^r - 1
Poly modPoly = new Poly(BigInteger.ONE, r.intValue()).minus(new Poly(BigInteger.ONE,0));
// X^n (mod X^r - 1, n)
Poly partialOutcome = new Poly(BigInteger.ONE, 1).modPow(n, modPoly, n);
for( int i = 1; i <= limit; i++ )
{
    // i
    Poly polyI = new Poly(BigInteger.valueOf(i),0);
    // X^n + i (mod X^r - 1, n)
    Poly outcome = partialOutcome.plus(polyI);
    // (X+i)^n (mod X^r - 1, n)
    Poly p = new Poly(BigInteger.ONE,1).plus(polyI).modPow(n, modPoly, n);
    // X^n + i != (X+i)^n (mod X^r - 1, n) Arriba, si no siguiente i
    if( !outcome.equals(p) )
    {
        ~~if (verbose) System.out.println( ...~~
        ~~if (verbose) System.out.println( ...~~
        ~~// if (verbose) System.out.println( ...~~
        factor = BigInteger.valueOf(i);
        n_isprime = false;
        return n_isprime;
    }
    ~~else~~
        ~~if (verbose) System.out.println( ...~~
}
```

$\text{limit} = \sqrt{\varphi(r)}\log n$

$O(\log^{7/2} n)$

Se estima en $r\log^2 n$ el $O(\log^7 n)$ coste los cálculos

Sale, es compuesto

$i \le \sqrt{\varphi(n)}\cdot\log n$

$i \le \sqrt{r}\,\log n$

$i \le \sqrt{\log^5 n}\,\log n$; $i \le \log^{5/2} n \log n$; $i \le \log^{5/2+2/2} n$; $i \le \log^{7/2} n$

$$(x+a)^n \equiv x^n + a \ (\underbrace{\text{mod } x^r-1}_{\text{Se halla el módulo de su división con ambos}},\ \underbrace{n}_{\text{Del resto de antes cada coef. es su módulo } n}) ; \ (x+a)^n-(x^n+a)(\text{mod } x^r-1, n) = 0$$

Se halla el resto de $\dfrac{(x+a)^n}{(x^r-1)}$ y sobre es resto a cada coeficiente se le sustituye por su módulo n.

$\ldots(465a^{29}+1)x^2 + 31a + a^{31}$ con $n=31$

$\%31 \quad \%31 \quad \%31 \quad \%31$

$\ldots(0a^{29}+1)x^2 + 0a + 1a^{31} = x^2 + a^{31}$

$\sqrt{r} \qquad r\sqrt{r}\log^3 n$

AKS al ser $O(\underbrace{\sqrt{n}}_{\text{potencia}} + \underbrace{\log^6 n}_{\text{mcd}} + \underbrace{\log r}_{\text{totient}} + \underbrace{\log^{21/2} n}_{\text{cond. suf.}}) = O(\log^{21/2} n)$

$O(\sqrt{n}) \simeq O(\log^3 n)$ en valores grandes

$(a+b)^2 = a^2 + 2ab + b^2$

$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$

$(a+b)^4 = a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4$

$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \end{array}$$

$F_n = \dfrac{\varnothing^n - (1-\varnothing)^n}{\sqrt{5}}$ $\quad \varnothing = \dfrac{1+\sqrt{5}}{2}$ golden ratio

$\log_2\log_2^2 x = 2$

$\log_2^2 x = 2^2$

$2\log_2 x = 2^2$

$\log_2 x = \frac{2^2}{2}$

$x = \left(\frac{2^2}{2}\right)^2 = \left(\frac{4}{2}\right)^2 = (2)^2 = 4$

$\log_2 \log_2^2 4 = \log_2 2^2 = 2$

$a \geq F_{n+1}$

$b \geq F_n$

$a > b \geq 0$

1. Show that if Euclid(a,b) takes more than N steps, then a>=F(n+1) and b>=F(n), where F(i) is the i th Fibonacci number.
   This can easily be done by induction.
2. Show that F(n) ≥ ϕ^{n-1}, again by induction.
3. Using results of Step 1 and 2, we have b ≥ F(n) ≥ ϕ^{n-1}
   Taking logarithm on both sides, $\log_\phi b \geq n-1$

Hence proved, $n \leq 1 + \log_\phi b$

$F_n \geq \varnothing^{n-1}$

$b \geq F_n \geq \varnothing^{n-1}$

$b \geq \varnothing^{n-1}$; $\log_\varnothing b \geq n-1$; $n \leq \log_\varnothing b + 1$

$O(\log b)$

$b^2 < n$ por. bases ~ ~ posibles exponentes $k=0$ $k=1$ $k=2$

potencia Perfecta $O(\sqrt{n}\ \text{cte}) = O(\sqrt{n})$

~ bucle ext. iterativo ~ mutiplicativeOrder se hace r veces

calculo R $O(\log^2 n \cdot \log^5 n) = O(\log^7 n)$

~ r veces ~ coste 1 gcd

gcd $O(\log^5 n \cdot \log n) = O(\log^6 n)$

factores $i^2 < n$ divisiones maximas $i^k \leq n$ while (n mod i == 0)

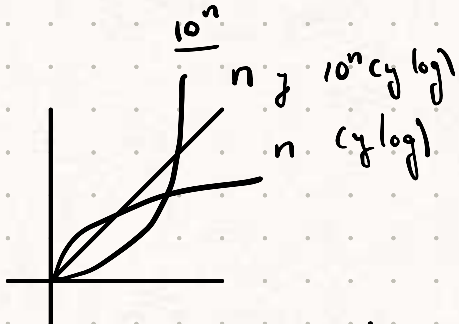
totient $O(\sqrt{n} + \log n) = O(\sqrt{n})$

↳ Atenua los $\sqrt{n}$, entonces por separado

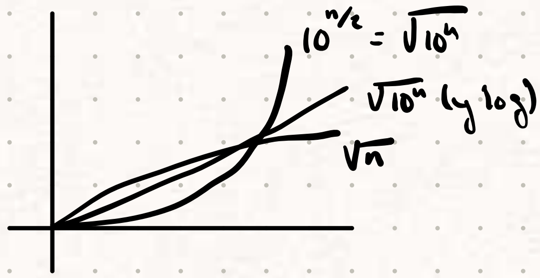
condición Suficiente $O(\sqrt{r}\log n \cdot r\log^2 n) = O(\log^{21/2} n)$

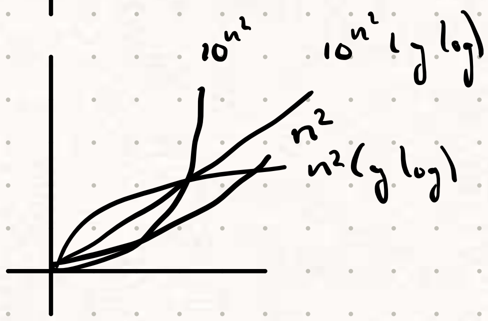
$\sqrt{\text{totient}}\ \log n$ veces bucle externo

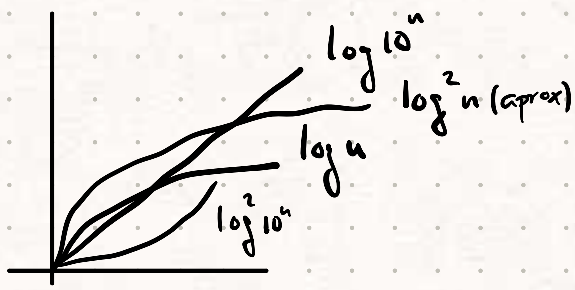
estimacion coste operar ecuacion



$x^n + a \equiv (x+a)^n \pmod{x^r - 1, n}$







Para $n_0 = 10$:

$T(n) = 3n+1$

$g(n) = n$

$$\frac{T(n_0)}{g(n_0)} = \frac{31}{10} = 3'1 \qquad \lim_{n\to\infty} \frac{T(n)}{g(n)} = \lim_{n\to\infty} \frac{3n+1}{n} = 3$$

$\exists c, n_0$ constantes positivas / $\forall n \geq n_0 \quad c \cdot g(n) \geq T(n)$

$$O(g(n)) = \left\{ \begin{array}{l} f(n) / \exists c, n_0 \text{ constantes positivos,} \\ \forall n \geq n_0,\ c \cdot g(n) \geq f(n) \end{array} \right\} \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} = k \geq 0$$

Palindromos

MTD1 $O(n^2)$ De lado a lado comprobando extremos iguales
PC: Palindromo

MTD2 $O(n)$ La copia y va recorriendo desde cada extremo comparando
PC: Palindromo

MTN1 $O(n)$ Transcribe la mitad (ND) en la cinta y otro estado la lee al reves
PC: Palindromo

Suma base 1

MTD1 $O(n^2)$ Mueve 1's de derecha a izquierda
PC: Sumando derecha n-1

MTD2 $O(n)$ Transcribe el sumando 1, borra # y al final va metiendo el copiado
PC: Sumando izquierda n-1

Suma base 2

MTD1 $O(2^n n)$ Va de extremo a extremo restando 1 (1 por 0, mientras 0 por 1) y sumando 1 (0 o # por 1, mientras 1 por 0).
PC: Sumando derecho n-1

MTD2 $O(2^n)$ Transcribe el sumando 1 y va restando y sumando (ahorra en viajes)
PC: Sumando izquierda n-1

Palabras estructura triplicada

abbaab
ab

MTD2 $O(n^2)$ Copia un simbolo, prueba del derecho, del reves y del derecho, si falla copia el siguiente simb.
PC: Falla ultima letra

MTN $O(n)$ El primer estado transcribe w (ND) haciendo la primera lectura, despues compara con w^{-1} borrando y por ultimo si coinciden w borra ambos. Si vacio, un 1.
PC: Cualquiera, siempre tarda n+3

Para sumas en lenguaje natural

MTD1: Mejor base 2 porque el ir de lado a lado en una cadena mas larga con base 1 es peor.

MTD2: Mejor base 1 porque es copiar el sumando y directamente ir poniendo al final, y con base 2 tiene que controlar todo el rato el acarreo y resta binaria