

Test de primalidad

enumeración, enfoque top-down y optimización

Primalidad - Introducción

Escribamos un programa que toma un número natural y responde si es primo o compuesto.

Tres pasos:

- tomar un número natural por consola
- decidir si el número es primo o compuesto
- imprimir “PRIMO” o “COMPUESTO” según corresponda

Primalidad - Introducción

```
#include<iostream>
using namespace std;
int main() {
```

```
}
```

Primalidad - Introducción

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
```

```
}
```

Primalidad - Introducción

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
```

```
    if (???)
```

```
}
```

Enfoque top-down: “si lo hiciera otro...”

Primalidad - Enfoque top down

Pensar en todos los detalles de un problema al mismo tiempo puede ser un quilombo. Nos podemos marear, trabar, o perder la concentración.

Para evitar esto, podemos hacer programación top down: escribimos una descripción general de nuestro programa, sin ahondar en detalles. Una vez completa, vamos agregando más detalles según haga falta.

Este proceso se repite hasta que terminemos de escribir el programa.

Nos podemos imaginar que usamos partes de nuestro programa antes de que estén escritas como si fuera a venir otra persona a hacerlas por nosotros.

Primalidad - Enfoque top down

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
```

```
    if (???)
```

```
}
```


Primalidad - Enfoque top down

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
```

```
    if (???)
    else
    cout << "PRIMO\n";
    cout << "COMPUESTO\n";
}
```

Una idea: consultar la definición

Primalidad - Consultar la definición

Si no sabemos cómo avanzar con la descomposición de un problema, consultar la definición del problema nos puede ayudar.

Definición de número primo:

Sea $n \in \mathbb{N}$. n es primo si tiene exactamente 2 divisores

Primalidad - Consultar la definición

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
```

```
    if (???)
    else
}
```

```
    cout << "PRIMO\n";
    cout << "COMPUESTO\n";
```

Primalidad - Consultar la definición

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
    int cantidad_divisores = ???;

    if (cantidad_divisores == 2) cout << "PRIMO\n";
    else cout << "COMPUESTO\n";
}
```

Enumeración

Primalidad - Enumeración

No hay una fórmula que nos dé la cantidad de divisores de un número pero, si supiéramos cómo verificar que un número es divisor de otro, podríamos enumerar todas las posibilidades y contar.

Para enumerar casos en C++ usamos la sentencia

```
for (A; B; C) { D }
```

Primero ejecuta A. Luego, mientras B sea verdadero, ejecuta D seguido de C.

Por ejemplo, para enumerar los valores de 1 hasta n (incluido) escribimos:

```
for (int i = 1; i <= n; ++i) { ... }
```

Primalidad - Enumeración

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
    int cantidad_divisores = ???;

    if (cantidad_divisores == 2) cout << "PRIMO\n";
    else cout << "COMPUESTO\n";
}
```


Primalidad - Enumeración

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
    int cantidad_divisores = 0;
    for (int i = 1; i <= n; ++i) {

    }
    if (cantidad_divisores == 2) cout << "PRIMO\n";
    else cout << "COMPUESTO\n";
}
```

Primalidad - Enumeración

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
    int cantidad_divisores = 0;
    for (int i = 1; i <= n; ++i) {
        bool es_divisor = ???;

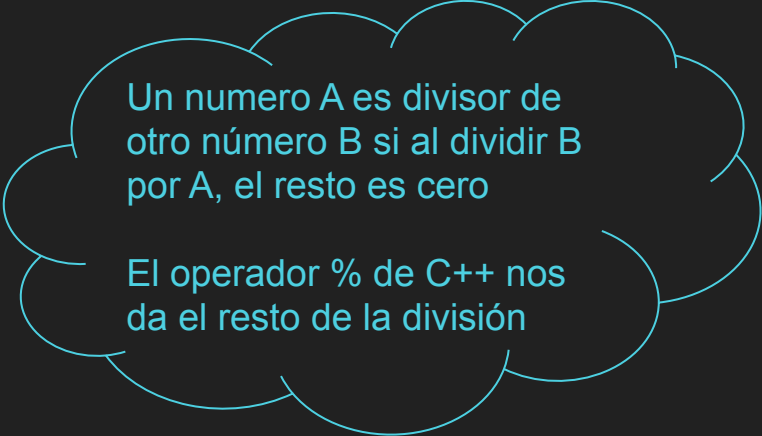
    }
    if (cantidad_divisores == 2) cout << "PRIMO\n";
    else cout << "COMPUESTO\n";
}
```

Primalidad - Enumeración

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
    int cantidad_divisores = 0;
    for (int i = 1; i <= n; ++i) {
        bool es_divisor = ???;
        if (es_divisor) {
            cantidad_divisores += 1;
        }
    }
    if (cantidad_divisores == 2) cout << "PRIMO\n";
    else cout << "COMPUESTO\n";
}
```

Primalidad - Enumeración

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
    int cantidad_divisores = 0;
    for (int i = 1; i <= n; ++i) {
        bool es_divisor = ???;
        if (es_divisor) {
            cantidad_divisores += 1;
        }
    }
    if (cantidad_divisores == 2) cout << "PRIMO\n";
    else cout << "COMPUESTO\n";
}
```



Un número A es divisor de otro número B si al dividir B por A, el resto es cero

El operador % de C++ nos da el resto de la división

Primalidad - Enumeración

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
    int cantidad_divisores = 0;
    for (int i = 1; i <= n; ++i) {
        bool es_divisor = n % i == 0;
        if (es_divisor) {
            cantidad_divisores += 1;
        }
    }
    if (cantidad_divisores == 2) cout << "PRIMO\n";
    else cout << "COMPUESTO\n";
}
```

Primalidad - Enumeración

Vimos como usar enumeración para contar los divisores de un número, usando un chequeo de divisibilidad. ¡Eso en realidad es bastante general!

Siempre que podamos responder preguntas del estilo ¿X cumple la propiedad Y? (problema de decisión, el resultado es V o F)

Usando enumeración podríamos responder ¿Cuántos X cumplen la propiedad Y? (problema de conteo, el resultado es un número)

Optimizando el algoritmo

Primalidad - Optimización

En las olimpiadas, se determina un límite de tiempo para la ejecución de nuestros programas. En general 1 segundo o 2.

Con este algoritmo, un número muy grande puede tomar casi un segundo en correr.

Veamos un teorema matemático que nos ayuda en este problema.

Primalidad - Optimización

Hipótesis: n es un número natural que no tiene divisores menores o iguales a \sqrt{n}

Tesis: n no tiene divisores mayores a \sqrt{n}

Demostración:

Por absurdo, supongamos que existe un divisor de n mayor a \sqrt{n} , llamado p .

Entonces $n:p$ también es un divisor de n , y es menor a \sqrt{n} . Esto implica que n tiene un divisor menor a \sqrt{n} , lo cual contradice la hipótesis, así que la suposición debe ser inválida.

Por lo tanto, no existe ningún divisor de n que sea mayor a \sqrt{n}

Primalidad - Optimización

De ahí sacamos que: si n no tiene divisores menores o iguales a \sqrt{n} , entonces no tiene ningún divisor.

Esto nos da un mejor algoritmo: Solo hace falta verificar divisores hasta \sqrt{n}

Primalidad - Enumeración

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
    int cantidad_divisores = 0;
    for (int i = 1; i <= n; ++i) {
        bool es_divisor = n % i == 0;
        if (es_divisor) {
            cantidad_divisores += 1;
        }
    }
    if (cantidad_divisores == 2) cout << "PRIMO\n";
    else cout << "COMPUESTO\n";
}
```

Primalidad - Enumeración

```
#include<iostream>
using namespace std;
int main() {
    int n; cin >> n;
    int cantidad_divisores = 0;
    for (int i = 1; i <= sqrt(n); ++i) {
        bool es_divisor = n % i == 0;
        if (es_divisor) {
            cantidad_divisores += 1;
        }
    }
    if (cantidad_divisores == 1) cout << "PRIMO\n";
    else cout << "COMPUESTO\n";
}
```

Esto en realidad rompe un poco porque, para cualquier $n > 1$, el for se pierde todos los divisores mayores a \sqrt{n} (que son exactamente la mitad) así que hay que poner un 1 en el if de abajo. Aparte, queda 1 como caso especial, que no es primo

Primalidad - Conclusiones

- El enfoque top-down nos ayuda a descomponer problemas
- Si no sabemos cómo descomponer, consultar la definición puede ayudar
- Enumerando pasamos de decisión a conteo (“probar caso por caso”)
- Podemos usar propiedades matemáticas para optimizar programas
- podemos usar el operador % para saber si un número es múltiplo de otro