Fuerza bruta

Cristian Nettle

¿Qué es la fuerza bruta?

Fuerza bruta es una técnica que consiste en realizar una búsqueda completa del espacio del problema, ya sea para encontrar una solución óptima o simplemente una solución factible.

Búsqueda completa

La búsqueda completa es un método general que sirve para resolver casi cualquier problema algorítmico. La idea es generar todas las soluciones posibles del problema mediante fuerza bruta y luego, según lo que pida el problema, elegir la mejor solución o contar cuántas hay.

- Útil cuando hay tiempo suficiente para recorrer todas las soluciones.
- Es más sencilla de implementar y siempre da una respuesta válida.

Algunas cosas antes

Conjunto: una colección de elementos distintos, sin un orden particular. Por ejemplo, $S = \{1, 2, 3\}$.

Subconjunto: un conjunto A es subconjunto de S ($A \subseteq S$) si todos los elementos de A pertenecen también a S. El conjunto vacío $\varnothing$ y el propio S siempre son subconjuntos de S.

Un conjunto de n elementos tiene exactamente 2^n subconjuntos.

Generando Conjuntos

Ejemplo: los subconjuntos de $S = \{1, 2, 3\}$ son:

$$\varnothing,\ \{1\},\ \{2\},\ \{3\},\ \{1, 2\},\ \{1, 3\},\ \{2, 3\},\ \{1, 2, 3\}$$

En total $8 = 2^3$ subconjuntos.

Problema: implementar código que genere todos los subconjuntos de n elementos.

Código

Recorremos los elementos uno a uno; para cada elemento decidimos **no incluirlo** o **incluirlo** en el subconjunto actual. Al llegar al final (`k == n`) tenemos un subconjunto listo. Así se generan los 2^n subconjuntos.

```
vector<int> subset;

void search(int k) {
    if (k == n) {
        // procesar subset
    } else {
        search(k + 1);           // no incluir el elemento k
        subset.push_back(k);
        search(k + 1);           // incluir el elemento k
        subset.pop_back();
    }
}
```

Complejidad: $O(2^n)$

Utilizando Bitmask

Un **bitmask** es un número usado como máscara de bits: representamos un subconjunto de n elementos con un número de n bits, donde el bit j vale 1 si el elemento j está en el subconjunto y 0 si no.

Operaciones de bits que necesitamos:

- `1 << j` : desplaza un 1 hacia la izquierda j posiciones, obteniendo 2^j (un 1 en la posición j).
- `mask & (1 << j)` : el operador `&` (AND bit a bit) es distinto de 0 solo cuando el bit j de `mask` está encendido, es decir, cuando el elemento j está en el subconjunto.

Recorriendo los números de 0 a $2^n - 1$ pasamos por todos los subconjuntos posibles.

Código con bitmask

```
for (int mask = 0; mask < (1 << n); mask++) {
    vector<int> subset;
    for (int j = 0; j < n; j++) {
        if (mask & (1 << j))        // ¿está el elemento j en este subconjunto?
            subset.push_back(j);
    }
    // procesar subset
}
```

Generando permutaciones

Definición: una permutación es una forma de ordenar todos los elementos de un conjunto. Un conjunto de n elementos tiene $n!$ permutaciones distintas.

Ejemplo: las permutaciones de $\{1, 2, 3\}$ son:

$$(1, 2, 3),\ (1, 3, 2),\ (2, 1, 3),\ (2, 3, 1),\ (3, 1, 2),\ (3, 2, 1)$$

En total $6 = 3!$ permutaciones.

Problema: imprimir todas las permutaciones de los elementos $\{0, 1, \ldots, n-1\}$.

Código utilizando bitmask

Usamos un bitmask `usados` para marcar qué elementos ya están en la permutación actual: el bit j en 1 significa que el elemento j ya fue colocado.

```
vector<int> perm;

void generar(int usados) {
    if ((int)perm.size() == n) {
        // procesar perm
        return;
    }
    for (int j = 0; j < n; j++) {
        if (usados & (1 << j)) continue;   // el elemento j ya se usó
        perm.push_back(j);
        generar(usados | (1 << j));
        perm.pop_back();
    }
}
```

Código con next_permutation

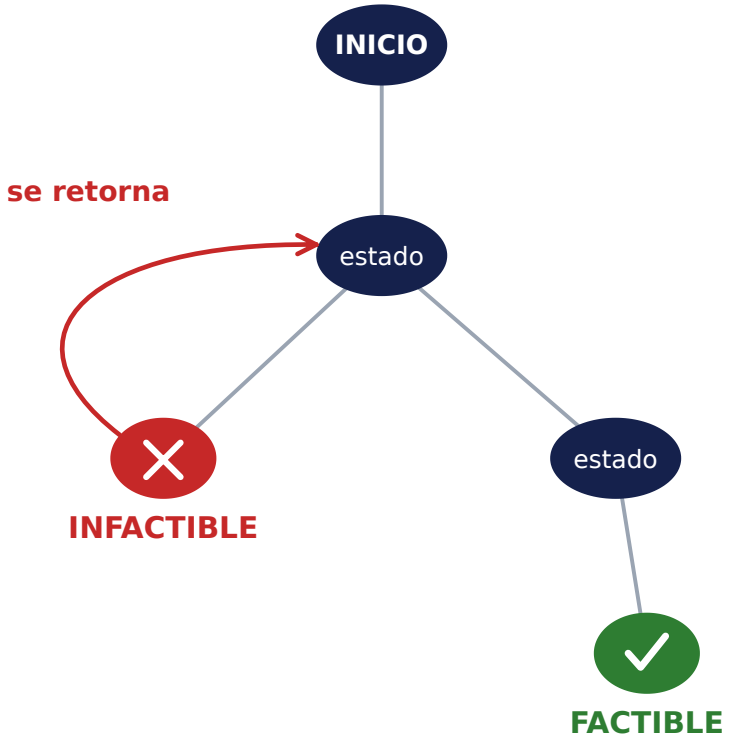
C++ ofrece `next_permutation`, que transforma un arreglo en su **siguiente** permutación en orden lexicográfico. Partiendo del arreglo ordenado, recorremos todas:

```
vector<int> perm = {1, 2, 3, 4};
do {
    for (int x : perm)
        cout << x << " ";
    cout << "\n";
} while (next_permutation(perm.begin(), perm.end()));
```

Complejidad: $O(n \cdot n!)$

Backtracking

Backtracking es una técnica que consiste en construir la solución paso a paso, y si en algún punto la solución se vuelve infactible, se retorna a un estado anterior, hasta encontrar una solución factible.



Problema Prefiprimos

Enunciado: Se busca contar la cantidad de números de tres dígitos, de forma que cumplen que cada prefijo es también un número primo.

Solución con fuerza bruta:

```
int cnt = 0;
for (int a = 1; a <= 9; a++) {
    for (int b = 0; b <= 9; b++) {
        for (int c = 0; c <= 9; c++) {
            if (es_primo(a) && es_primo(10 * a + b) && es_primo(100 * a + 10 * b + c)) {
                cnt++;
            }
        }
    }
}
```

Problema Prefiprimos

Solución con backtracking:

```
int cnt = 0;
for (int a = 1; a <= 9; a++) {
    if (!es_primo(a)) continue;
    for (int b = 0; b <= 9; b++) {
        if (!es_primo(10 * a + b)) continue;
        for (int c = 0; c <= 9; c++) {
            if (!es_primo(100 * a + 10 * b + c)) continue;
            cnt++;
        }
    }
}
```

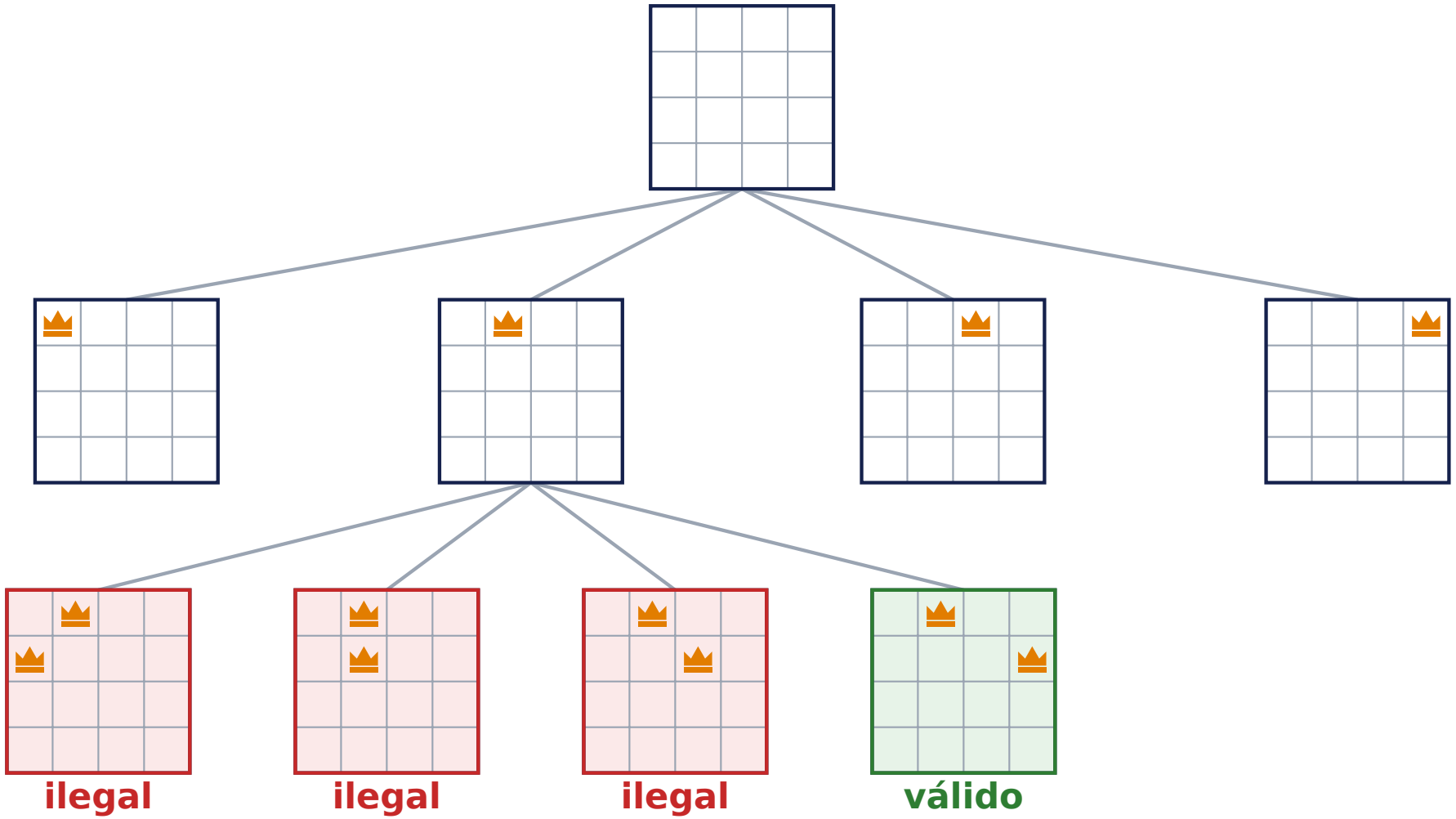
En lugar de probar todas las posibilidades, si en un paso se encuentra con un estado inválido regresa al dígito anterior, lo que reduce significativamente el tiempo de búsqueda.

CSES - Chessboard and Queens

La tarea es colocar 8 reinas en un tablero de ajedrez de forma que las reinas no se ataquen entre sí. Además, existen cuadrados ocupados, y no se puede colocar una reina en esos espacios.

La idea es usar **backtracking** ubicando una reina por fila, y marcando las columnas y diagonales ocupadas para descartar posiciones inválidas.

CSES - Chessboard and Queens



CSES - Chessboard and Queens

Función de backtracking que cuenta las soluciones válidas:

```
void search(int y) {
    if (y == n) {
        cnt++;
        return;
    }

    for (int x = 0; x < n; x++) {
        if (column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;

        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
        search(y+1);
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
    }
}
```

CSES - Chessboard and Queens

Matemática para programación competitiva

CSES - Exponentiation

Enunciado: calcular de forma eficiente el valor de $a^b \bmod (10^9 + 7)$.

Entrada: la primera línea contiene un entero n, la cantidad de cálculos. Luego siguen n líneas, cada una con dos enteros a y b.

Salida: por cada cálculo, imprimir el valor de $a^b \bmod (10^9 + 7)$.

Restricciones: $1 \leq n \leq 2 \cdot 10^5$ y $0 \leq a, b \leq 10^9$

Entrada

```
3
3 4
2 8
123 123
```

Salida

```
81
256
921450052
```

CSES - Exponentiation

Aritmética modular

En varios problemas de matemática se pide calcular un número en un módulo específico. Esto es debido a que estos valores crecen rápidamente.

Se dice que un número a es congruente a b módulo m, si es que a deja resto b al dividirse por m. En lenguaje matemático, $a \equiv b \pmod{m}$.

Propiedades básicas:

$$(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$$

$$(a - b) \bmod m = ((a \bmod m) - (b \bmod m)) \bmod m$$

$$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$$

Exponenciación binaria

Podemos calcular a^b multiplicando a el resultado b veces, pero esto tomaría $O(b)$ operaciones. ¿Existe una forma más rápida de calcularlo?

La respuesta es sí, se puede realizar **exponenciación binaria** en $O(\log b)$.

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ es par} \\ x^{n-1} \cdot x & n \text{ es impar} \end{cases}$$

Exponenciación binaria

```
int binpow(int x, int n, int m) {
    if (n == 0)
        return 1;

    int u = binpow(x,n/2,m);
    u = (u*u)%m;

    if (n%2 == 1)
        u = (u*x)%m;
    return u;
}
```

Complejidad: $O(\log n)$

Combinatoria básica

En combinatoria, los dos principios fundamentales para contar el número de formas de elegir o construir objetos son el **Principio de la Suma** y el **Principio de la Multiplicación**.

Principio de la Suma

$$n_1 + n_2$$

elegir **una sola** opción del total

esto O lo otro

Principio de la Multiplicación

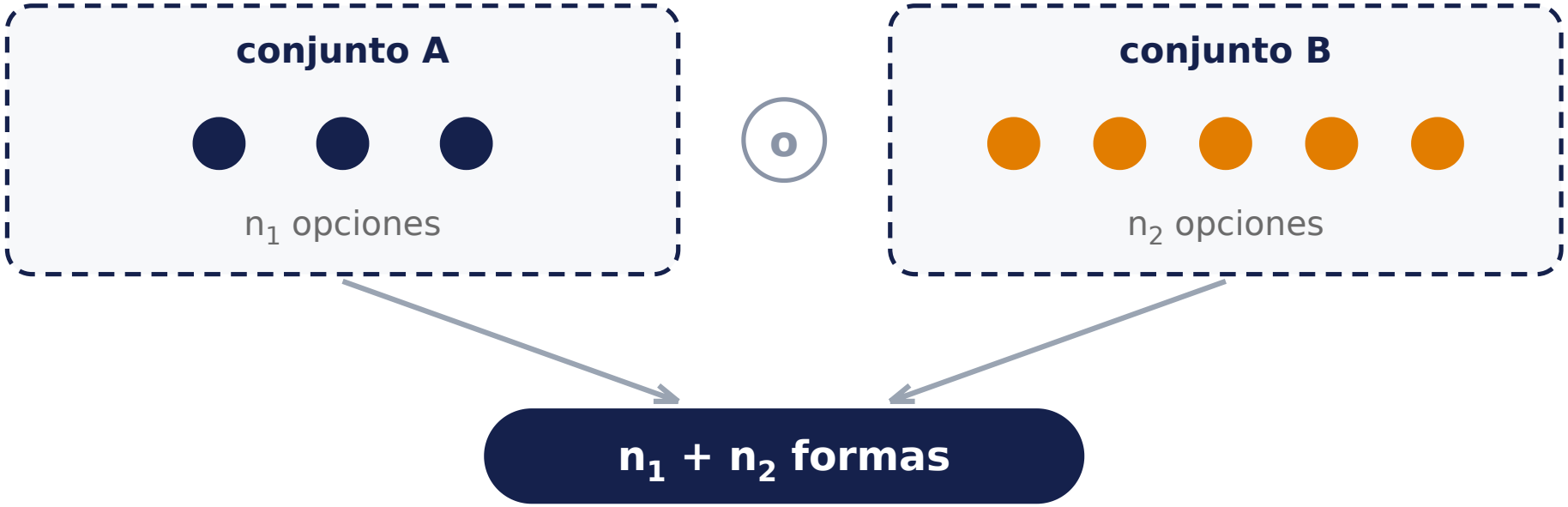
$$n_1 \cdot n_2$$

completar **todas** las etapas

esto Y lo otro

Principio de la Suma

Si tenemos dos (o más) conjuntos **disjuntos** de opciones y queremos elegir **exactamente una** opción de entre todas, el número total de formas es la **suma** de las cantidades de cada conjunto.



Principio de la Suma — Ejemplos

Ejemplo 1 · eliges UNA camisa

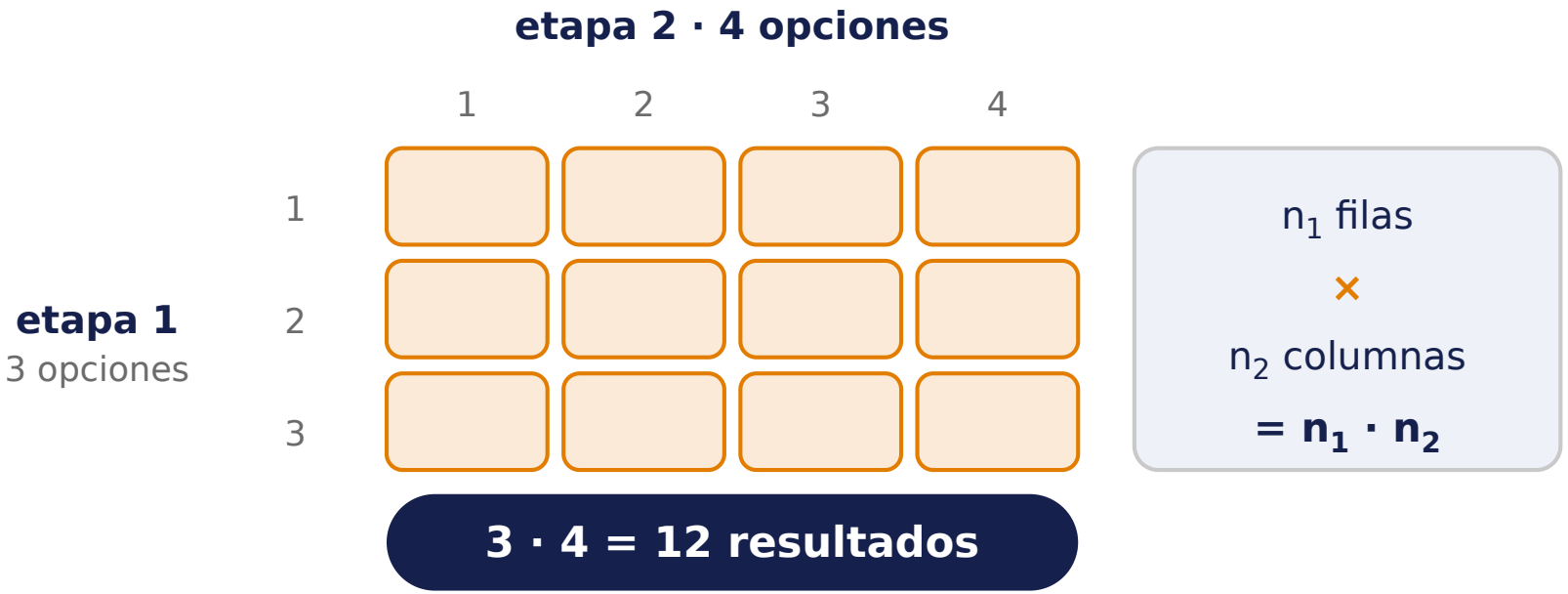
3 rojas + 5 azules = **8 camisas**

Ejemplo 2 · eliges UNA ruta

Ruta A · 4 variaciones + Ruta B · 3 variaciones = **7 maneras**

Principio de la Multiplicación

Si un proceso se descompone en una secuencia de **etapas**, y cada opción de una etapa se combina con **todas** las de la siguiente, el número total de resultados es el **producto** de las cantidades de cada etapa.



Principio de la Multiplicación — Ejemplos

Ejemplo 1 · código de 2 dígitos

dígito 1
0 – 9 · 10 opciones

×

dígito 2
0 – 9 · 10 opciones

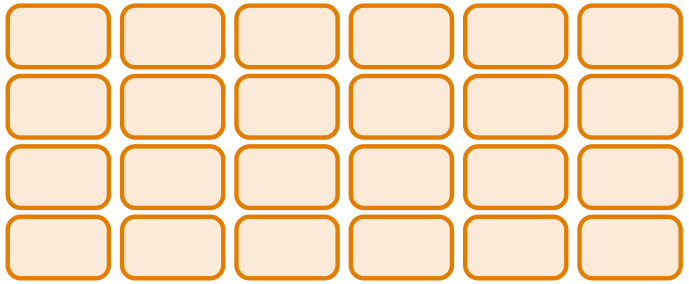
=

100 códigos

Ejemplo 2 · un entrante y un plato principal

6 platos principales

4 entrantes



=

24 menús

Codeforces - Matching

Enunciado: una **plantilla** es un string de dígitos y/o signos `?`. Un entero positivo **calza** con la plantilla si se puede reemplazar cada `?` por un dígito y obtener su representación decimal **sin ceros a la izquierda**. Dada una plantilla de a lo más 5 caracteres, contar cuántos enteros positivos calzan con ella.

Entrada: la primera línea contiene t, la cantidad de casos. Cada caso trae un string s de dígitos y/o `?`.

Salida: por cada caso, cuántos enteros positivos calzan con la plantilla.

Restricciones: $1 \leq t \leq 2 \cdot 10^4$ y $1 \leq |s| \leq 5$

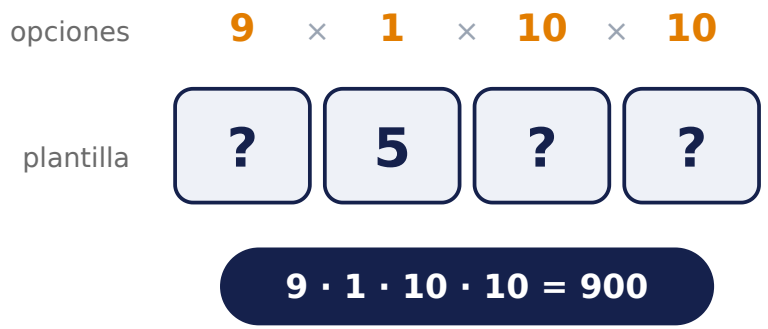
Codeforces 1821A - Matching

Matching - Insight

1. Cada `?` es una posición **independiente**: lo que elijas en una no afecta a las demás. Por el Principio de la Multiplicación se **multiplican** las opciones de cada posición.

2. Un `?` tiene 10 opciones (0–9), pero el **primero** solo 9 (1–9), porque no puede quedar un cero adelante. Un dígito fijo aporta 1 opción.

3. Si la plantilla **empieza con** `0`, ningún número calza: la respuesta es 0.



Descomposición en Factores Primos

La **descomposición prima** (o factorización prima) de un entero $n \geq 2$ consiste en expresar n como el producto de números primos, únicos salvo el orden:

$$n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \dots \cdot p_k^{\alpha_k}$$

donde cada p_i es un número primo y cada α_i es un entero positivo.

Ejemplo: $84 = 2^2 \cdot 3 \cdot 7$

Factorización por División

El método más sencillo es probar divisores crecientes hasta $\sqrt{n}$:

```cpp
using ll = long long;

vector<ll> primeFactors(ll n) {
    vector<ll> factors;
    for (ll i = 2; (i * i) <= n; i++) {
        while (n % i == 0) {
            factors.push_back(i);
            n /= i;
        }
    }
    if (n > 1) factors.push_back(n);
    return factors;
}
```

Criba de Eratóstenes

Si quisieramos obtener todos los primos hasta n con el método anterior, nos tomaría tiempo $O(n\sqrt{n})$. Para esto existe la **criba de Eratóstenes**.

```
vector<int> primes;
vector<bool> isPrime;

void eratosthenes_sieve(int n) {
    isPrime.resize(n + 1, true);
    isPrime[0] = isPrime[1] = false;
    for (int i = 2; i <= n; i++) {
        if (isPrime[i]) {
            primes.push_back(i);
            for (long long j = (long long)i * i; j <= n; j += i)
                isPrime[j] = false;
        }
    }
}
```

gcd y lcm en C++

El **máximo común divisor** de dos enteros a y b es el número más grande que divide a ambos sin dejar resto.

```
int a = 48, b = 18;
cout << "gcd(" << a << ", " << b << ") = " << gcd(a, b) << "\n";
```

El **mínimo común múltiplo** de dos enteros a y b es el número más pequeño que es múltiplo de ambos.

```
cout << "lcm(" << a << ", " << b << ") = " << lcm(a, b) << "\n";
```

`gcd` y `lcm` están en `<numeric>` y requieren C++17.

Timus - Bald Spot Revisited

Enunciado: los pubs están numerados con enteros positivos y desde el pub n se puede pasar a un pub cuyo número **divide** a n. En cada pub se bebe una jarra. El sueño empieza en el pub a y hay que llegar al pub b bebiendo **lo máximo posible**. Si no se puede llegar, se bebe 0.

Entrada: la primera línea contiene T, la cantidad de casos. Luego siguen T líneas con dos enteros a y b.

Salida: por cada caso, el máximo número de jarras que puede beber en el camino.

Restricciones: $0 \leq T \leq 20$ y $1 \leq a, b \leq 10^9$

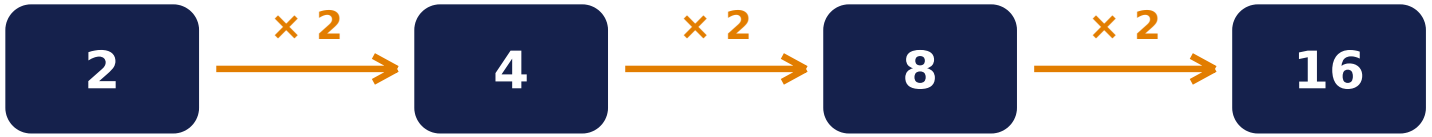
Timus 1355 - Bald Spot Revisited

Bald Spot Revisited - Insight

1. Solo hay camino si a **divide** a b. Si no, la respuesta es 0.

2. Sea $k = b/a$. Cada paso multiplica por un entero > 1, así que para visitar **más** pubs cada paso debe multiplicar por lo más chico posible: un **primo**.

3. Entonces basta **factorizar** k: cada factor primo es un paso. La respuesta es la **cantidad de factores primos de** k, más 1 por el pub inicial.



k = 16 / 2 = 8 = 2 · 2 · 2 → **3 pasos** → **4 pubs**

Codeforces - Spring

Enunciado: Alice visita el manantial cada a días (días $a, 2a, 3a, \ldots$), Bob cada b días y Carol cada c días. Si ese día llega **una** persona se lleva 6 litros; si llegan **dos**, 3 litros cada una; si llegan **tres**, 2 litros cada una. Hay que calcular cuánta agua junta cada uno en los primeros m días.

Entrada: la primera línea contiene t, la cantidad de casos. Cada caso trae cuatro enteros a, b, c y m.

Salida: por cada caso, tres enteros: los litros de Alice, Bob y Carol.

Restricciones: $1 \leq t \leq 10^4$; $1 \leq a, b, c \leq 10^6$; $1 \leq m \leq 10^{17}$

Codeforces 2204C - Spring

Spring - Insight

1. Con $m \leq 10^{17}$ no se puede simular. Solo importa **cuántos** días visita cada uno: $A = m/a, B = m/b, C = m/c$.

2. Dos coinciden en los múltiplos de su **lcm**: $AB = m/\text{lcm}(a, b)$; los tres, $ABC = m/\text{lcm}(a, b, c)$.

3. Alice recibe 6 litros sola, 3 con uno y 2 con dos (análogo para Bob y Carol):

$$\text{Alice} = 6A - 3AB - 3AC + 2ABC$$