# Múltiplos

Dados dos enteros *x* y *k* escribid todos los múltiplos de *x* positivos que son menores que *k*.

## Estrategia de la solución

En este problema solo tenemos que implementar un bucle *for*.

## Solución en Python

```python
t = int(input())
for test in range(t):
        strings = input().split()
        x = int(strings[0])
        k = int(strings[1])
        y = x
        while y < k:
        if y+x < k:
                print(y, end=' ')
        else:
                print(y) # último print
        y += x
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

## Solución en C++

```cpp
#include <iostream>

using namespace std;

int main() {
  int t;
  cin >> t;
  while (t--) {
        int x, k;
        cin >> x >> k;
        cout << x;
        int y = 2*x;
        while (y < k) {
        cout << ' ' << y;
        y += x;
        }
        cout << endl;
  }
}
```

Autor de la solución: Cesc Folch (Miembro del comité organizador)

# Ordenando

Tenemos $n$ números positivos y queremos ordenarlos de pequeño a grande, ¡el problema es que los números pueden tener hasta 10000 cifras! La parte buena es que cada número tiene todas sus cifras iguales. Para poder codificar más fácilmente los números se darán dos enteros $d_i$ y $c_i$ que indican el número de dígitos del número y el valor de cada cifra.

Dados los $n$ números escribid el orden de menor a mayor de los índices originales de los números. Es decir, si tenemos los números 111 3 22 la respuesta es 2 3 1 porque el segundo número va primero, después el tercero y por último el primero.

En caso de que dos enteros sean iguales, se conservará el orden relativo original entre ellos. Es decir, si un entero aparecía antes que otro entero igual en el orden original, su índice aparecerá antes que el del otro en el resultado.

## Estrategia de la solución

Para poder ordenar los elementos podemos crear un vector con pares de pares de enteros y un entero para poder guardar la información y ordenarlo con la función sort asociado a este tipo de elementos.
Link pair: https://www.geeksforgeeks.org/pair-in-cpp-stl/
Link sort: https://www.geeksforgeeks.org/sorting-a-vector-in-c/

Otra manera de solucionar el problema es crear una struct o class y una función de orden para poder ordenarlas.
Link struct/class: http://www.cplusplus.com/doc/tutorial/structures/

## Solución en Python

```python
t = int(input())
for test in range(t):
    n = int(input())
    nums = []
    for i in range(n):
        strings = input().split()
        d = int(strings[0])
        c = int(strings[1])
        nums.append((d, c, i))
    nums = sorted(nums)
    for i in range(n):
        idx = nums[i][2]+1
        if i < n-1:
            print(idx, end=' ')
        else:
            print(idx) # último print
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

## Primera solución en C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef pair<int, int> pi;
typedef pair<pi, int> ppi;
typedef vector<ppi> vppi;

int main() {
  int t;
  cin >> t;
  while (t--) {
        int n;
        cin >> n;
        vppi V(n);
        for (int i = 0; i < n; ++i) {
        cin >> V[i].first.first >> V[i].first.second;
        V[i].second = i+1;
        }
        sort(V.begin(), V.end());
        for (int i = 0; i < n; ++i) {
        if (i) cout << ' ';
        cout << V[i].second;
        }
        cout << endl;
  }
}
```

## Segunda solución en C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct numero {
  int d, c;
  int ind;
};

bool comp(numero a, numero b) {
  if (a.d != b.d) return a.d < b.d;
  if (a.c != b.c) return a.c < b.c;
  return a.ind < b.ind;
}

int main() {
```

```
  int t;
  cin >> t;
  while (t--) {
        int n;
        cin >> n;
        vector<numero> V(n);
        for (int i = 0; i < n; ++i) {
        cin >> V[i].d >> V[i].c;
        V[i].ind = i+1;
        }
        sort(V.begin(), V.end(), comp);
        for (int i = 0; i < n; ++i) {
        if (i) cout << ' ';
        cout << V[i].ind;
        }
        cout << endl;
  }
}
```

Autor de la solución: Cesc Folch (Miembro del comité organizador)

# Potencias

Dado un entero $x > 1$ encontrad dos enteros $a$ y $b$ tales que $a^b = x$ y que $b$ sea lo más grande posible.

## Estrategia de la solución

Para resolver el problema solo hacía falta probar todos los valores de a b tales que $a^b \leq x$, para hacer esto para cada b empezamos con $a = 2$ y vamos sumándole 1 hasta que $a^b \geq x$, si $a^b$ = x ya hemos terminado, si no probamos con $b = b\text{-}1$ hasta llegar a $b = 1$.
Este problema se puede resolver de formas mucho más eficientes usando por ejemplo los divisores de $x$ o búsqueda binaria.

## Solución en Python

```python
def divisores(n):
	# Esta función calcula los divisores (mayores que 1) de n
	div = []
	i = 2
	while i*i <= x:
	if x%i == 0:
		div.append(i)
		if i*i != x:
		div.append(x//i)
	i += 1
	div.append(n) # Como i empieza por 2, faltaría el divisor n
	return div

t = int(input())
for test in range(t):
	x = int(input())
	div = divisores(x)
	for a in div:
	pot = 1
	b = 0
	while pot < x:
		pot *= a
		b += 1
	if pot == x:
		print(a, b)
		break
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

## Solución en C++

```cpp
#include <iostream>

using namespace std;

typedef long long ll;
```

```
ll potencia(ll x, int e) {
  ll res = 1;
  while (e--) res *= x;
  return res;
}

int main() {
  int t;
  cin >> t;
  while (t--) {
        ll x;
        cin >> x;
        int pote = 32;
        ll val = x;
        while (pote > 1) {
        val = 2;
        while (potencia(val, pote) < x) ++val;
        if (potencia(val, pote) == x) break;
        val = x;
        --pote;
        }
        cout << val << ' ' << pote << endl;
  }
}
```

Autor de la solución: Cesc Folch (Miembro del comité organizador)

# Las Torres de Hanoi

Estás jugando con tu nuevo juguete, las torres de Hanoi. En esta nueva versión hay *n* discos, el disco más pequeño pesa 1 gramo, el segundo 2, el tercero 3... ¿Cuantos gramos vas a tener que mover en total para completar el juego? Cada vez que mueves un disco se suma su peso a los gramos movidos. Como el resultado puede ser muy grande, tenéis que dar el resultado módulo $10^9 + 7$.

## Estrategia de la solución

Para resolver este problema hace falta ver que si *S[n]* es la solución con n piezas, entonces *S[n] = 2*S[n-1] + n*, y que *S[0] = 0*. También se puede ver que *S[n] = 2^(n+1) – n – 2*. Usando programación dinámica es fácil calcular los valores de S[n] a partir de los anteriores.
Link DP: https://aprende.olimpiada-informatica.org/algoritmia-dinamica-1

## Solución en Python

```python
MAX_N = 10**5
MOD = 10**9 + 7

# Precomputamos todos los posibles casos en tiempo O(n)
# utilizando la siguiente recurrencia:
dp = [0 for i in range(MAX_N+1)]
dp[0] = 0
for i in range(1, MAX_N+1):
        dp[i] = (2*dp[i-1] + i) % MOD

t = int(input())
for test in range(t):
        n = int(input())
        print(dp[n])
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

## Solución en C++

```cpp
#include <iostream>
#include <vector>

using namespace std;

typedef long long ll;
typedef vector<ll> vll;

const ll mod = 1e9 + 7;
const int N = 100000;

int main() {
  vll D(N + 1);
```

```
  D[0] = 0;
  for (int i = 1; i <= N; ++i) {
       D[i] = (2*D[i-1] + i)%mod;
  }
  int t;
  cin >> t;
  while (t--) {
       int x;
       cin >> x;
       cout << D[x] << endl;
  }
}
```

Autor de la solución: Cesc Folch (Miembro del comité organizador)

# Permutando palabras

Dada una palabra quieres saber qué palabras puedes formar permutando las letras. Dos palabras son diferentes si en alguna posición tienen letras distintas. Dada una palabra *s* escribid todas las palabras que se pueden formar en orden alfabético.

## Estrategia de la solución

Para resolver este problema hacía falta hacer una búsqueda completa sobre todas las palabras, también podía usarse la función *next_permutation* de C++ o equivalentes pero no era el objetivo del problema.
Link Backtracking:
https://aprende.olimpiada-informatica.org/algoritmia-backtracking-busqueda-completa-1

## Solución en Python

```
def backtrack(s, letras, t):
	if len(t) == len(s):
	for c in t:
		print(c, end='')
	print()
	else:
	for letra in range(26):
		if letras[letra] > 0:
		letras[letra] -= 1
		t.append(chr(ord('a') + letra))
		backtrack(s, letras, t)
		t.pop()
		letras[letra] += 1

t = int(input())
for test in range(t):
	s = input()
	letras = [] # letras[i] indica cuántas letras iguales
		# a `i` quedan (a=0, b=1, ..., z=25)
	for i in range(26):
	letras.append(0)
	for i in range(len(s)):
	letras[ord(s[i])-ord('a')] += 1
	backtrack(s, letras, [])
	print()
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

## Solución en C++

```cpp
#include <iostream>
#include <vector>

using namespace std;

typedef vector<int> vi;

void f(int i, string& s, vi& V) {
  if (i == (int)s.size()) {
        cout << s << endl;
        return;
  }
  for (int j = 0; j < (int)V.size(); ++j) {
        if (V[j]) {
        s[i] = 'a' + j;
        --V[j];
        f(i+1, s, V);
        ++V[j];
        }
  }
}

int main() {
  int t;
  cin >> t;
  while (t--) {
        string a;
        cin >> a;
        vi V(26, 0);
        for (int i = 0; i < (int)a.size(); ++i) {
        ++V[a[i] - 'a'];
        }
        f(0, a, V);
        cout << endl;
  }
}
```

Autor de la solución: Cesc Folch (Miembro del comité organizador)