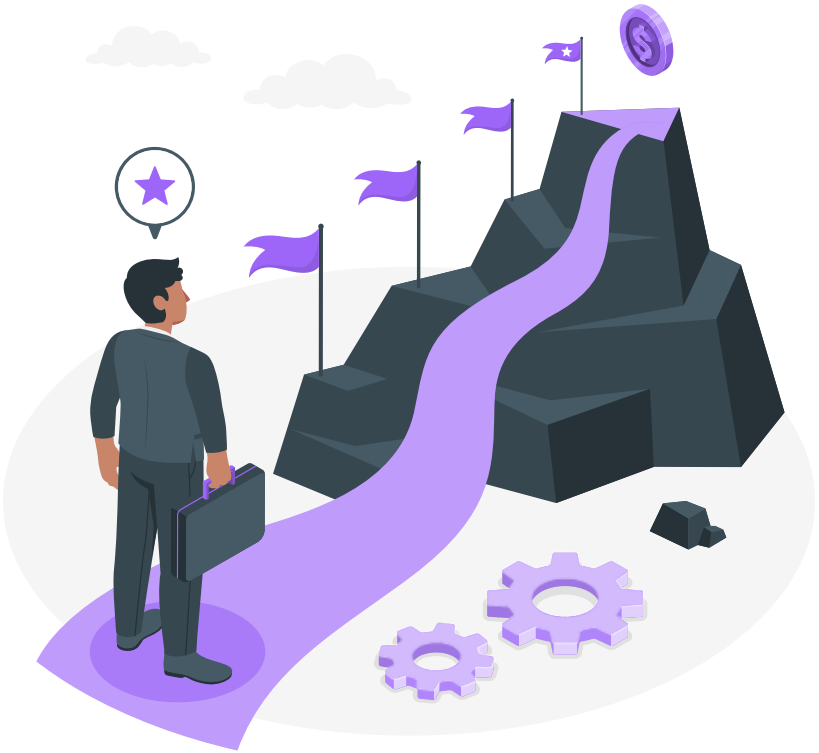
Computer Society
Uniandes

Programación Competitiva

Computer Society

Daniel Diaz



Solución de problemas de Introducción a Greedy

¡Hagámoslo sin *Wrong Answer*!

Minimum Varied Number

El ICPC presenta un reto donde se debe encontrar el número más pequeño posible cuya suma de dígitos sea igual a un valor dado y todos sus dígitos sean únicos. La entrada es un entero que representa la suma deseada de los dígitos y la salida es el número más pequeño con dígitos distintos que sume el valor dado.

20	3	8	9

Para construir el número, seguimos una estrategia voraz que consiste en comenzar siempre con el dígito más grande posible, es decir, el 9. Luego, continuamos con el 8, el 7, y así sucesivamente, asegurándonos de que la suma de los dígitos no exceda el valor de S. Si en algún momento la suma de los dígitos seleccionados supera S, simplemente colocamos el mínimo entre el dígito actual y la cantidad restante necesaria para alcanzar S.

Para concatenar un número a la izquierda de otro, es necesario multiplicarlo por una potencia de 10 (aunque el uso de cadenas de texto es una alternativa, resulta menos eficiente en términos de rendimiento). En el peor de los casos, cuando S = 45 (la suma máxima de los dígitos del 1 al 9), tanto la complejidad temporal como espacial del algoritmo son constantes.

```cpp
/**
    @author: Daniel Diaz
*/
#include <bits/stdc++.h>
using namespace std;

int main(){
    long long t, s, curr, pos, res;
    scanf("%d", &t);
    while (t--) {
        scanf("%lld", &s);
        res = 0;
        curr = 9;
        pos = 1;
        while (s > 0) {
            if (s >= curr) {
                res += curr * pos;
                s -= curr;
            }
            else {
                res += s * pos;
                s = 0;
            }
            curr--;
            pos *= 10;
        }
        printf("%lld\n", res);
    }
}
```

Santa Claus and Candies

Santa Claus tiene n dulces y sueña con regalarlos a los niños. Quiere dar todos los dulces que tiene, asegurándose de que cada niño reciba una cantidad entera positiva distinta. El reto es encontrar el número máximo de niños a los que puede dar dulces cumpliendo estas condiciones.

Dado un número de dulces, se debe indicar cuántos niños pueden recibir dulces y la cantidad exacta para cada niño. Si existen múltiples soluciones, se puede imprimir cualquiera de ellas.

8	1	2	5

En este problema, Santa Claus comienza repartiendo 1 dulce al primer niño, 2 dulces al segundo, 3 dulces al tercero, y así sucesivamente. Llega un momento en el que Santa Claus ha entregado i dulces al i-esimo niño, pero no tiene suficientes dulces (i + 1) para darle al siguiente niño. En este caso, los dulces restantes se le otorgan al niño i.

Para estimar la complejidad, observamos que el número total de dulces repartidos sigue la serie 1 + 2 + 3 +, ... , + i = N, Usando la fórmula de la suma aritmética, obtenemos: $\sum_{i=1}^{\sqrt{N}} i = \frac{\sqrt{N} * (\sqrt{N}+1)}{2} \approx N$, Por lo tanto, concluimos que tanto la complejidad temporal como espacial del algoritmo es de $O(\sqrt{N})$

```java
import java.util.*;
/**
    @author: ddi4z
*/
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        long n = scanner.nextLong();

        List<Long> nums = new ArrayList<>();
        long i = 1;
        long acum = 0;

        while (acum + i <= n) {
            nums.add(i);
            acum += i;
            i += 1;
        }

        nums.set(nums.size() - 1, nums.get(nums.size() - 1) +
(n - acum));

        System.out.println(nums.size());
        for (long num : nums) {
            System.out.print(num + " ");
        }
        System.out.println();
        scanner.close();
    }
}
```

Hard Problem

El profesor Ball tiene un aula con 2 filas y m asientos en cada fila. Hay a monos que solo quieren sentarse en la fila 1, b monos que solo quieren sentarse en la fila 2, y c monos sin preferencia. Cada asiento puede ser ocupado por un solo mono siguiendo las preferencias. El reto es determinar el número máximo de monos que se pueden sentar.

m	a	b	c
10	5	5	10

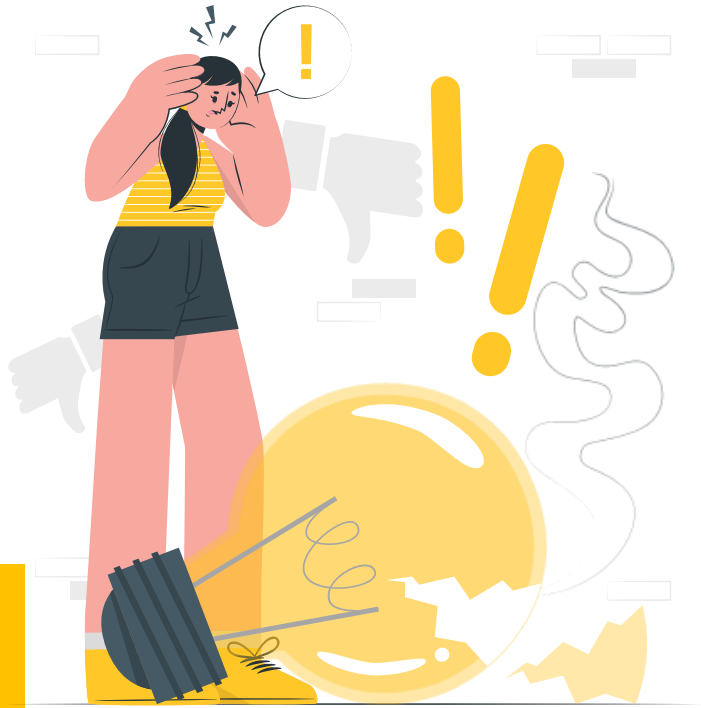
20

La estrategia voraz consiste en asignar la mayor cantidad posible de monos que tienen preferencia por la fila 1 a dicha fila, y la mayor cantidad posible de monos que prefieren la fila 2 a esa fila. Los puestos restantes se asignan a los monos que no tienen preferencia alguna. Dado que el número de filas y monos es fijo, tanto la complejidad temporal como espacial de este enfoque son constantes.

```cpp
/**
    @author: Daniel Diaz
*/
#include <bits/stdc++.h>
using namespace std;

int main() {
    long long t, m,a,b,c, rem;
    scanf("%d", &t);
    while (t--) {
        scanf("%lld %lld %lld %lld", &m, &a, &b, &c);
        rem = m - min(m, a) + m - min(m, b);
        printf("%d\n", min(rem, c) + min(m, a) + min(m, b));
    }
}
```

Phone Desktop



Rosie tiene un teléfono con varias pantallas, cada una de tamaño 5 filas por 3 columnas. Quiere colocar **x** aplicaciones con íconos de 1x1 y **y** aplicaciones con íconos de 2x2, usando el menor número posible de pantallas. Cada celda puede contener solo un ícono.

x	y
1	1

1	1	
1	1	
	1	

La estrategia voraz consiste en priorizar la colocación de los iconos grandes primero, ya que cada icono grande ocupa un espacio de 2x2. Dado que cada pantalla tiene un tamaño de 5x3, es posible colocar hasta dos iconos grandes, dejando 7 espacios libres para los iconos pequeños. Esta estrategia se repite hasta que solo quede un icono grande o no haya más iconos grandes disponibles.

Si queda un solo icono grande, se coloca en la pantalla, dejando 11 espacios disponibles para los iconos pequeños. Si no hay iconos grandes restantes, se utilizan los 15 espacios de la pantalla exclusivamente para los iconos pequeños.

La complejidad espacial es constante y la temporal es O(X+ Y).

```cpp
/**
    @author: Daniel Diaz
*/
#include <bits/stdc++.h>
using namespace std;

int main() {
    long long t, x, y, screens;
    scanf("%d", &t);
    while (t--) {
        scanf("%lld %lld", &x, &y);
        screens = 0;
        while (y > 0) {
            if (y == 1) {
                y--;
                x -= 11;
                screens++;
            }
            else {
                y -= 2;
                x -= 7;
                screens++;
            }
        }
        while (x > 0) {
            x -= 15;
            screens++;
        }
        printf("%d\n", screens);
    }
}
```

Maximize Mex

Dado un arreglo de **n** enteros positivos y un entero **x**, se pueden realizar operaciones para incrementar los valores de ciertos elementos en **x** unidades. El objetivo es encontrar el valor máximo del MEX (mínimo entero no presente en el arreglo).

n	x
6	2

6

1	3	4	1	0	2

Una propiedad clave del de un arreglo es que su valor siempre está en el rango de 0 y N, donde N es la longitud del arreglo (recomendamos reflexionar sobre por qué esto es cierto).

Para calcular el MEX, podemos recorrer una variable i desde 0 hasta N, si i no está presente en el arreglo, entonces i es el MEX, Si i aparece una sola vez, simplemente pasamos al siguiente número. Sin embargo, si i aparece varias veces (digamos m veces), debemos transformar m – 1 de esas ocurrencias incrementándolas en x unidades. Si al incrementar x el número resultante supera N, lo ignoramos.

Tanto la complejidad temporal como espacial de este enfoque son O(N)

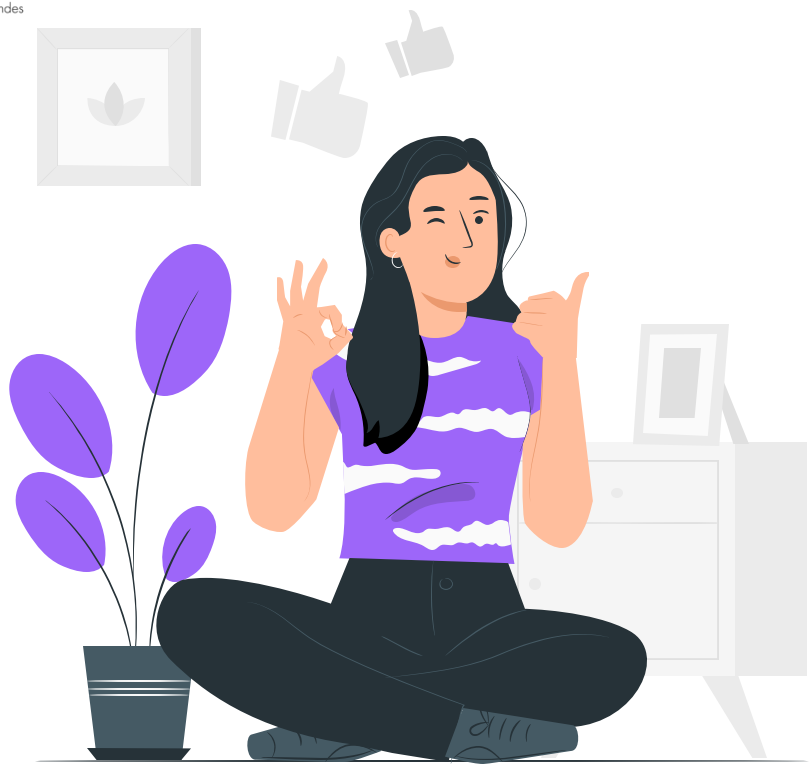
```
"""
    @author: Daniel Diaz
"""

def solve():
    n, x = map(int, input().split())
    a = list(map(int, input().split()))
    freq = [0] * (n + 1)
    for num in a:
        if num < n + 1:
            freq[num] += 1

    for i in range(n + 1):
        if freq[i] == 0:
            print(i)
            return
        if i + x < n + 1:
            freq[i + x] += freq[i] - 1

t = int(input())
for _ in range(t):
    solve()
```

Computer Society
Uniandes



¡Gracias!

Computer Society

Daniel Diaz