Problemas complejos de programación

Agenda

1. Tipos de problemas
2. Metodología para resolver problemas complejos
3. Como encontrar patrones únicos
4. Otros temas importantes a conocer

1.
Tipos de problemas

Los básicos pero suficientes

Normales (poco comunes)

No tienen características especiales.

Se resuelven con matemática y estructuras de control

Mosaic Decoration I

Time limit: *1000 ms*
Memory limit: *256 MB*

Zapray lives in a big mansion that has N bathrooms. He wants to decorate the bathroom walls using mosaic tiles of two colors: black and pink. The ith bathroom needs B_i black tiles and P_i pink tiles. Mosaic tiles are sold in piles. Zapray can buy one pile of 10 black tiles for C_B dollars, and one pile of 10 pink tiles for C_P dollars. How much money does he need in total to decorate all the N bathrooms?

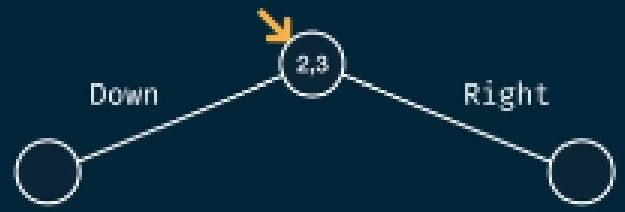
Input	Output
3 5 7 10 10 20 30 30 3	65

```
int cost = cb * ceil(sum_b / 10) + cp * ceil(sum_p / 10);
cout << cost << endl;
```

Matemáticos

Estos ocultan una ecuación matemática que permite llegar a la solución de forma directa.

Grid traveler se clasifica como varios tipos de problema, uno de ellos matemático.



Los caminos posibles son:

Derecha,derecha,abajo
Derecha,abajo,derecha
Abajo,derecha,derecha

Para una matriz 2x3 la respuesta es 3

Si derecha = 1 e
Abajo = 2

Los caminos son:
1,1,2
1,2,1
2,1,1

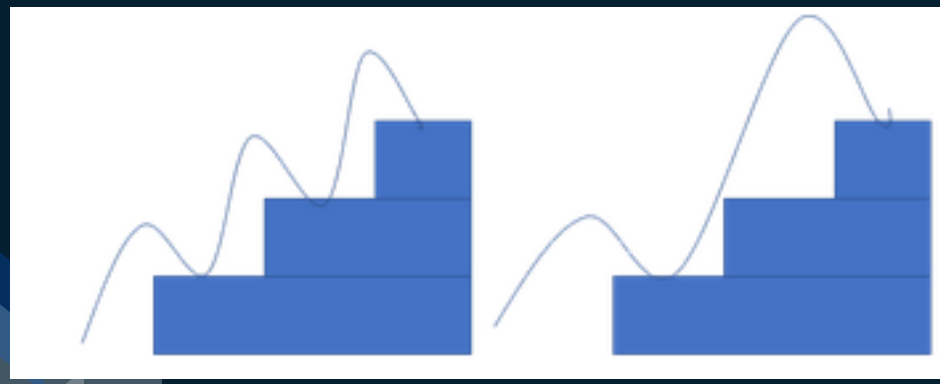
¡ Es una permutación con repetición !
Recursividad es ineficiente vs una fórmula

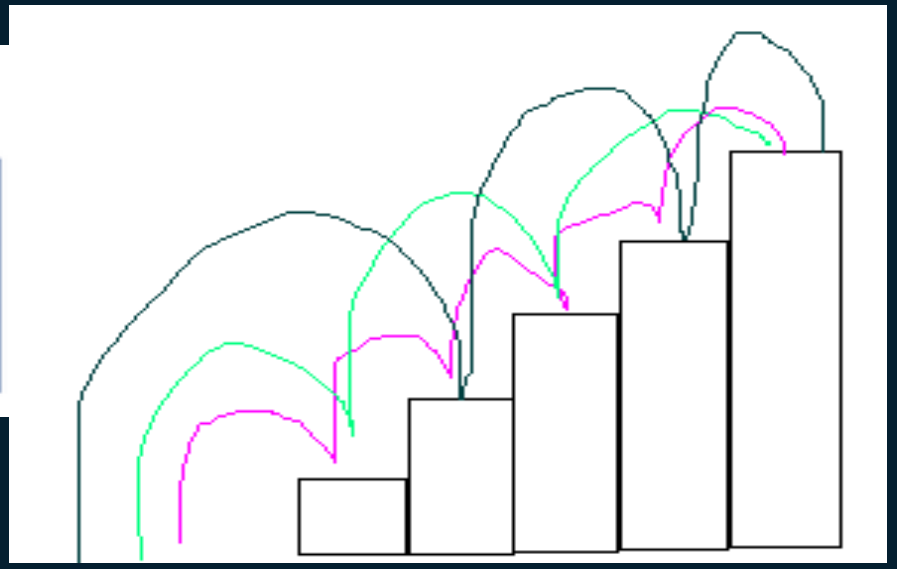
$$P_m^{r_1,r_2\ldots} = \frac{m!}{(r_1)(r_2)\ldots}$$

m = Factores.
r = elemento repetidos

Por patrones únicos (difíciles de ver)

Estos solo son visibles ante mentes creativas. Son problemas que tienen un patrón y dicho patrón hace que la resolución sea más fácil





Input	Output
1 3	3
1 5	8
2 1 2	1 2

Solución: considerar el problema como permutaciones y combinaciones. Para n=5 por ejemplo:
11111 = 1 forma posible
1211 -> obtener permutaciones con repetición (!4/(3!))
221 -> obtener permutaciones (3!/(2!))

Problema: el tiempo de ejecución es ridículamente exponencial. Hemos implementado la solución y funciona bien para los primeros 10 valores luego de varios minutos de procesamiento...

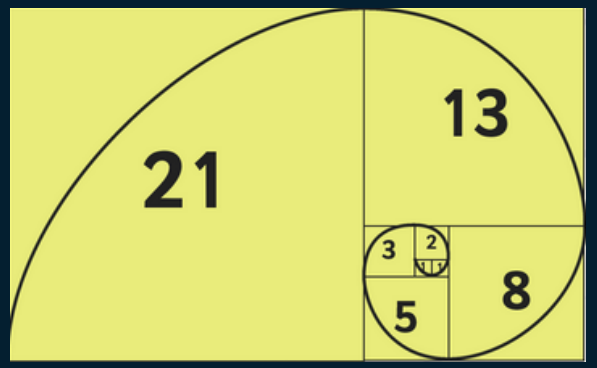
Time limit: *2500 ms*
Memory limit: *128 MB*

Constraints and notes

- $1 \leq t \leq 5$
- $1 \leq n \leq 22\,000$

1	1
2	2
3	3
4	5
5	8
6	13
7	21
8	34
9	55
10	89

Pero, ¿se dieron cuenta? Las respuestas son correctas, pero tienen algo raro...




```python
fibonacci = [1, 1, 2]

def obtenTerminoFibonacci(m):

    if len(fibonacci) > m:
        return fibonacci[m]

    fibonacci.append(obtenTerminoFibonacci(m-1) + obtenTerminoFibonacci(m-2))

    return fibonacci[m]

casos = int(input())

for _ in range(casos):
    print(obtenTerminoFibonacci(int(input())))
```

Score: 100/100 (99 ms - 36.7 MB)

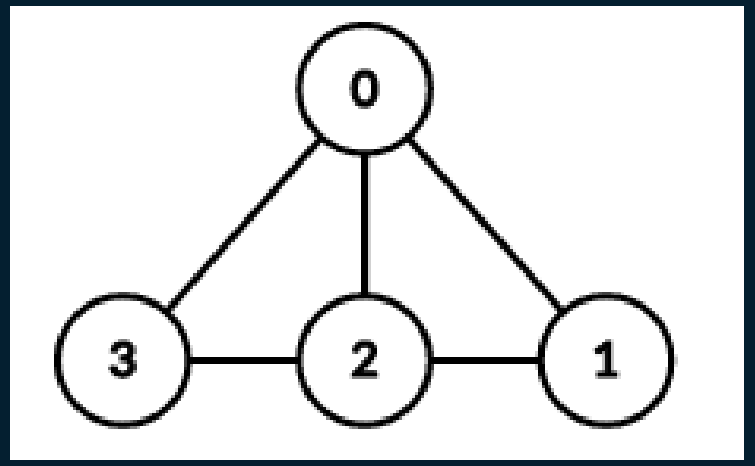
Test Number	CPU Usage	Memory Usage	Result
8	99 ms	36.7 MB	OK
7	95 ms	36.6 MB	OK
6	36 ms	4228 KB	OK

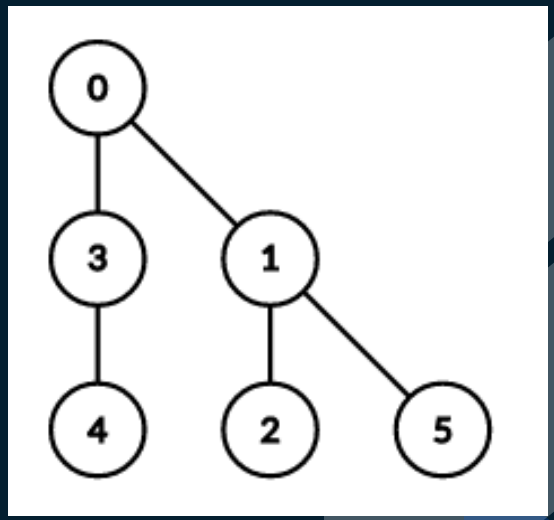
Vangelis the Batbear and the Bubbles challenge

Time limit: *1000 ms*
Memory limit: *256 MB*

Good evening master Wayne.

Joker and his gang attended Black Hat USA 2017 where they learned of a new way on how to damage our city! Specifically, tomorrow night they will try to damage the water pumps of Gotham using bubbles!





Luego de dibujar varios grafos notamos dos cosas: el problema nunca nos pide contar los bucles y existe un patrón en los grafos con bucle.

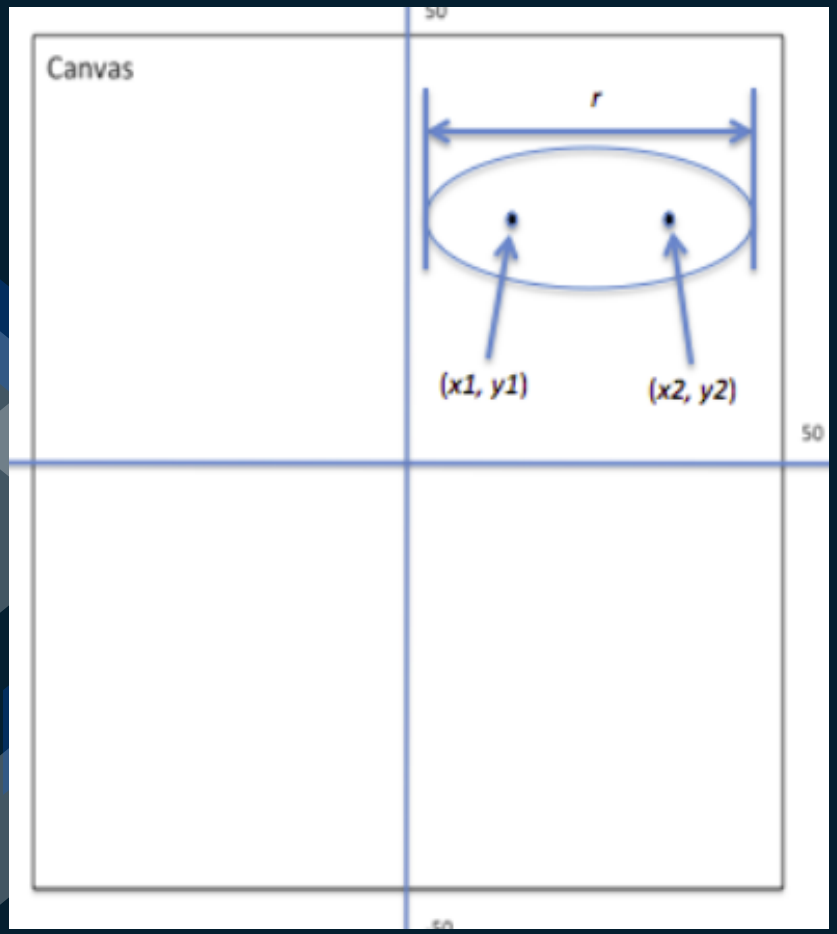
Podemos resolver este problema generando matrices de caminos de forma recursiva, marcando los visitados. Pero esta solución es muy lenta y no escala

Los grafos con al menos un bucle no necesitan nodos hoja. Por lo que si iteramos eliminando los nodos que son hoja:
- {0: [1, 3], 1: [0, 2, 5], 2: [1], 3: [0, 4], 4: [3], 5: [1]}
- {0: [1, 3], 1: [0], 3: [0]}
- {}

Si luego de eliminar hojas el arreglo es "{}" no tiene bucle.
- {0: [1, 2, 3], 1: [0, 2], 2: [0, 1, 3], 3: [0, 2]} -> no hay hojas, tiene bucle

Por algoritmos precisos

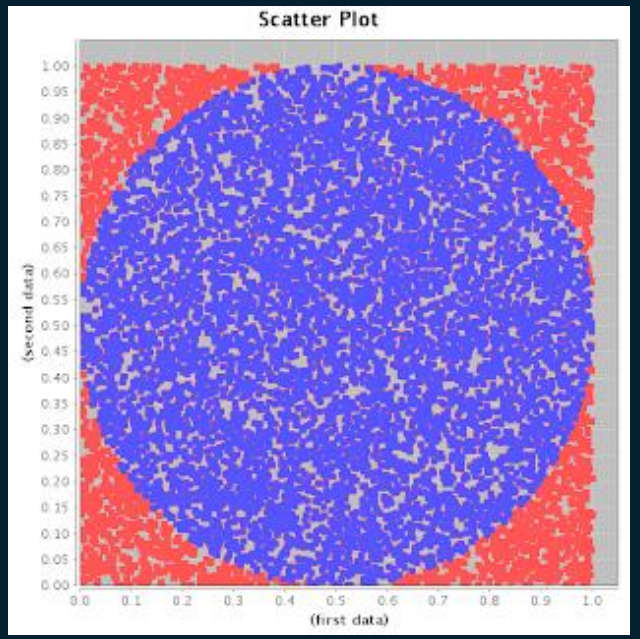
Existen algoritmos bien conocidos como Dijkstra, Beetle bag, Monte carlo, **BACKTRACKING, The two pointers**, sliding window, palindrome, etc.



Para encontrar el área de N ellipses normales y rotadas en un cuadrado se requiere hacer integrales muy complicadas, además de primero sacar las funciones.

MONTE CARLO AL RESCATE

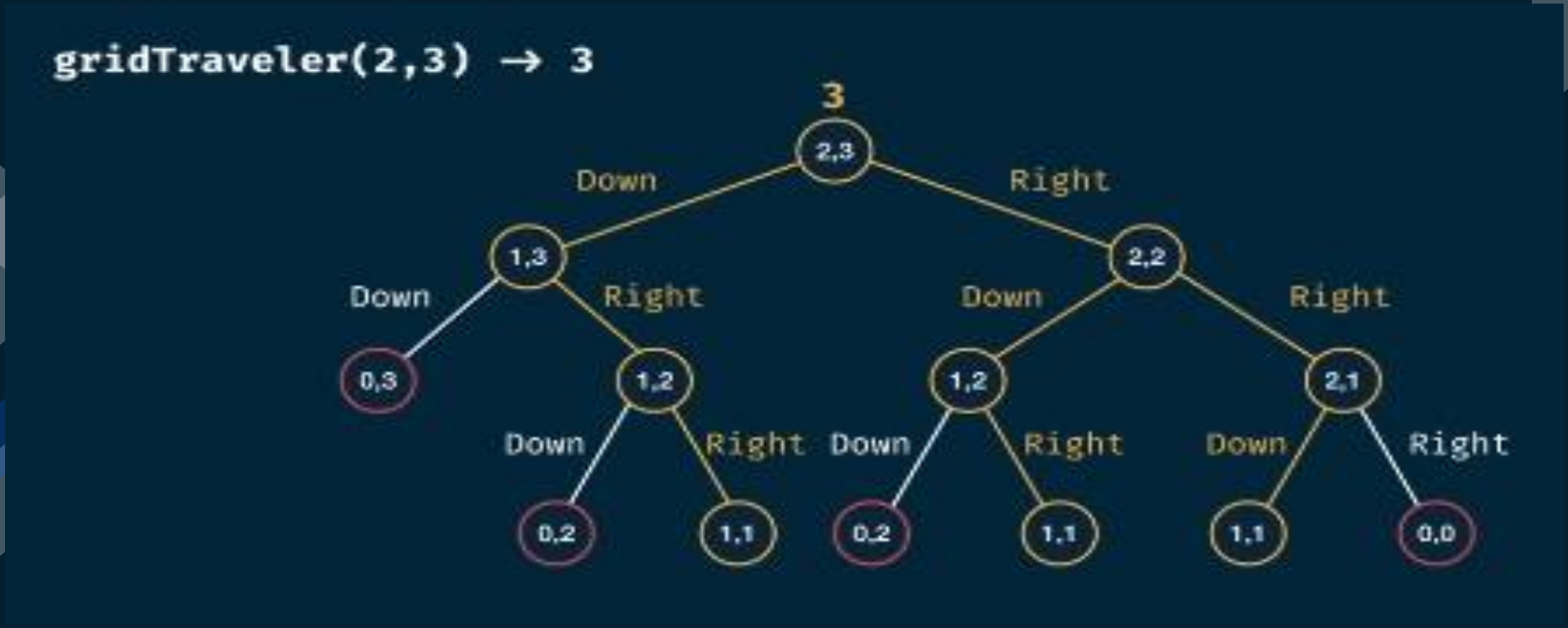
Esta tarea es imposible para una computadora a menos que se la mire con otros ojos. No sacar el area exacta, sino una aproximación al área, utilizando números aleatorios



De programación dinámica (comunes)

Son problemas que deben ser optimizados almacenando valores intermedios, evitando que haga cálculos repetitivos. Métodos: Memoization, mapping, tabulation, etc.

Memoization: Ver el problema como un árbol, generarlo con recursividad (requiere caso base). Usar un objeto "memo" para memorizar cálculos ya hechos y poder reusarlos.



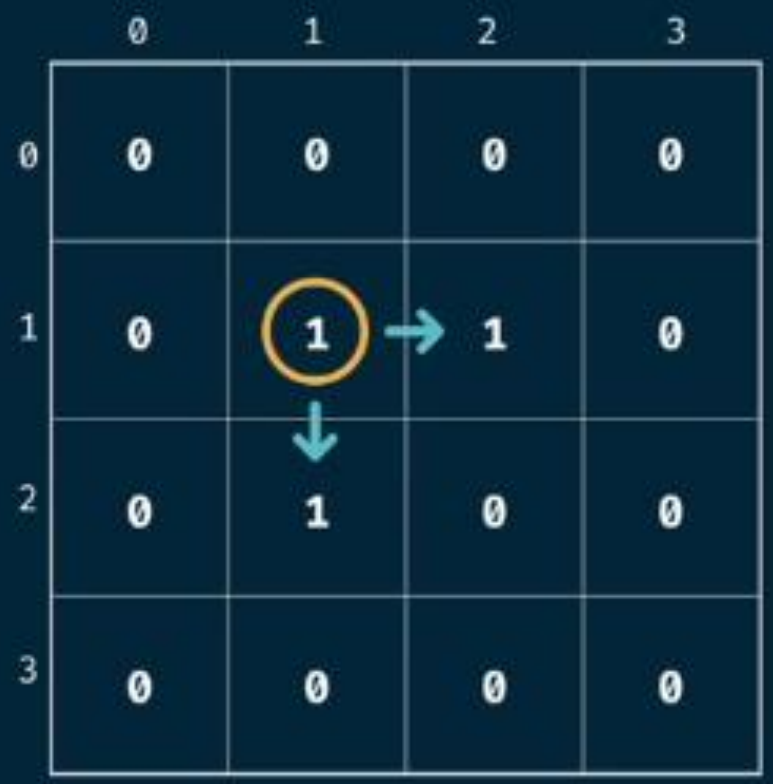
Tabulation, el más poderoso y común de PD

El tamaño de la tabla se relaciona a la entrada. Necesita condiciones iniciales. Los valores siguientes de la tabla se calculan en base a uno anterior. Cada celda representa un subproblema.

Ventaja sobre los otros métodos: cada celda representa una solución, y dicha solución queda guardada para un futuro cálculo. Podemos accederla siempre

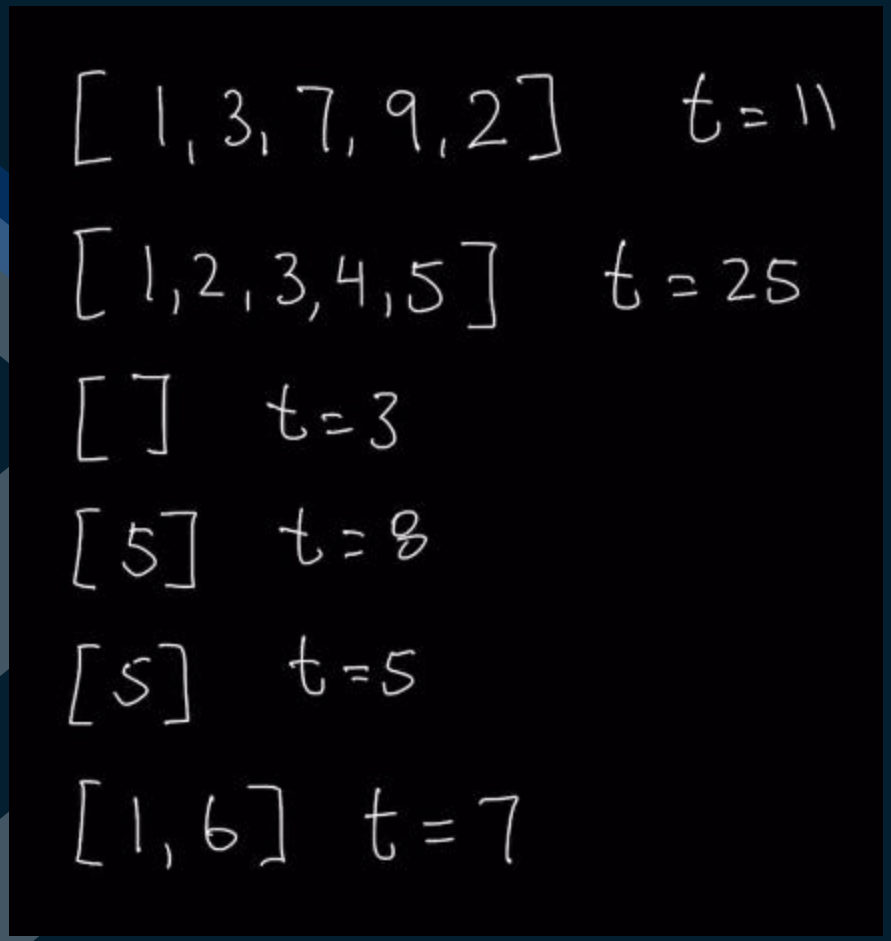
O(mn) time
O(mn) space

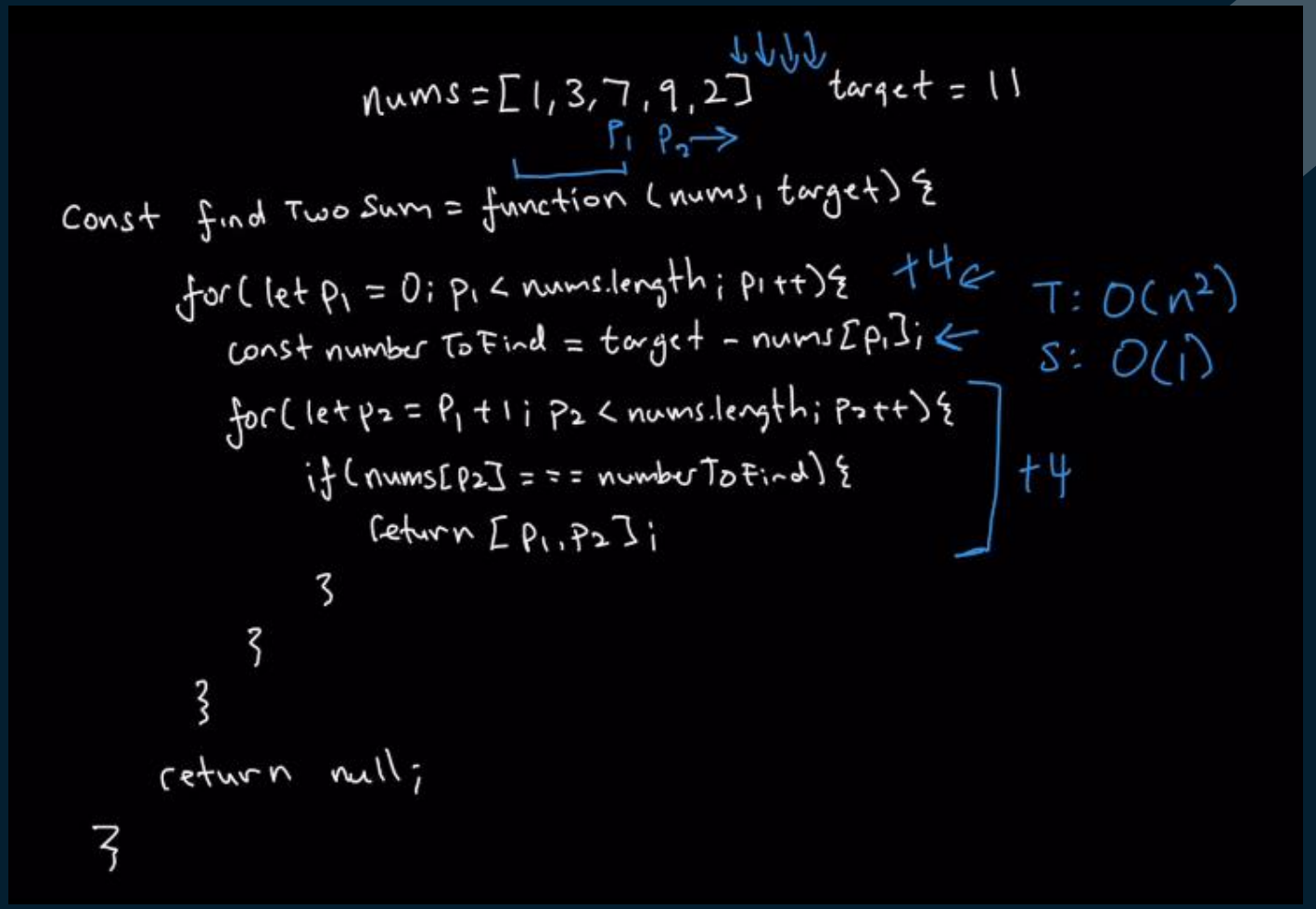
	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	2	3
3	0	1	3	6



Por optimización (comunes)

La solución clara de estos problemas es por fuerza bruta, pero con modos de ser optimizados de forma que se reduce la complejidad Big-O del tiempo, a veces a costa de la complejidad Big-O espacial





2.
Metodología para resolver problemas complejos

Preparación

- Pre generar funciones de utilidad: factorial, fibonacci, imprimir, copiar matriz, etc Solo en caso de que sea legal..
- Generar un template para leer, procesar datos y testear.
- Llevar las librerías adecuadas como boost/multiprecision en el caso de c++, consultar si en la competencia se pueden usar.
- Usar un IDE con debugger.
- Familiarizarse con el inglés y tener un traductor abierto. A veces el no entender bien el lenguaje frustra y hace que se demore mucho la primera fase de la ejecución.

Programar Es la parte fácil

La parte dificil es diagramar

A. Analiza en papel

1. Entiende la lógica del enunciado
2. Analiza y crea casos de prueba: uno más sencillo, otro un poco más extenso y otros especiales
3. Genera visualizaciones

5	4	3
3		
0		
1		
2		
2		

Input	Output
2 2 2 1 2 0 2 4 2 0 3	4 5

1	1	1	0
0	0	0	0
1	0	0	0
1	1	0	0
1	1	0	0

3	0	1	2	2

A. Analiza en papel

4. Determina las condiciones necesarias para generar la respuesta. Descartar ramas de razonamiento complejas, generar hipótesis de la solución y para cada solución buscarle un contra ejemplo. Si no hay, es una hipótesis correcta.

R: a las habitaciones con menor cantidad de bien cableados.

¿Cómo sé a dónde hago un "toggle switch" para maximizar?

1	1	1	0
0	0	0	0
1	0	0	0
1	1	0	0
1	1	0	0

3	0	1	2	2

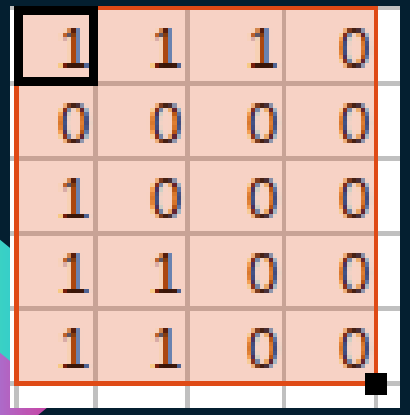
Con las visualizaciones notamos que K se dirige a los elementos menores del arreglo y "les da la vuelta": 0->4, 1->3, 2->2

H1: el valor de encendido es (n – a) donde a es igual a la suma de bien cableados en la fila

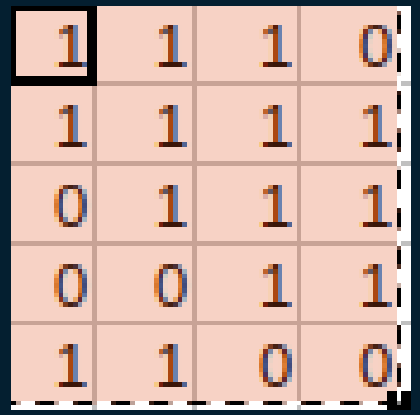
A. Analiza en papel

¿Cuál es el flujo para obtener la suma de habitaciones prendidas?

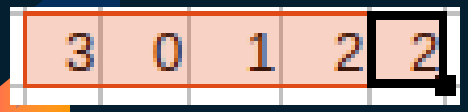
R: Hay que sumar las habitaciones prendidas cuando se da la vuelta. Podemos asumir que las bien conectadas inician prendidas



1	1	1	0
0	0	0	0
1	0	0	0
1	1	0	0
1	1	0	0



1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
1	1	0	0



3	0	1	2	2

3	4	3	2	2

No necesitamos la matriz, pero fue muy útil en la visualización. H2: La condición para llegar al resultado es "Sumar todos los elementos del arreglo, luego de invertir sus valores"

A. Analiza en papel

¿Cuál es la manera más rápida de invertir los valores al arreglo?

Notamos que K se asigna a los números más pequeños del arreglo. Por lo que, h3: ordenar de forma ascendente me permitirá hacer este proceso más rápido

1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
1	1	0	0

3	4	3	2	2

Seleccionar un algoritmo de ordenamiento con máxima complejidad O(n*log(n))

B. Programar

Con la planificación hecha en papel esta es la parte fácil. Si la planificación fue errónea todo se cae.

Si fallas al planificar, estás planificando fallar.

```
qsort(roomWired, m, sizeof(int), compare); // n log(n)

cpp_int sum = 0;
for (size_t indexInvert = 0; indexInvert < k; indexInvert++)
{
    roomWired[indexInvert] = n - roomWired[indexInvert];
    sum += roomWired[indexInvert];
}
for (size_t indexInvert = k; indexInvert < m; indexInvert++)
{
    sum += roomWired[indexInvert];
}
```

C. Test

- Probar el programa primero con todos los casos realizados a mano.
- NO depurar con mensajes en lo posible, siempre usar el debugger de un IDE con inspector de variables como VSCODE.
- Automatizar la ejecución del programa y de las pruebas.
- Si el algoritmo resuelve los casos de ejemplo pero no los reales que no podemos ver, significa que debemos crear más casos de prueba con más combinaciones extrañas y analizar con la herramienta debug que los cálculos se hacen correctamente.

- $1 \leq T \leq 20$
- $1 \leq K \leq M \leq 10^6$
- $1 \leq N \leq 10^6$
- The sum of M across all test cases in a single test file is at most $2 \cdot 10^6$.

(Situacional) Optimización

- Si la eficiencia es relativamente buena, buscar bajar la complejidad Big-O del programa almacenando cálculos repetitivos. Usando librerías más optimizadas.
- Si la eficiencia es mala, buscar una integración de la programación dinámica y aumentar el uso de memoria RAM.
- Siempre apuntar a reducir Big-O del tiempo, y aumentar el BIg-O de la memoria.
- Si la eficiencia no mejora, entonces implementar la solución en un lenguaje de más bajo nivel como c++ (si el concurso lo permite)
- Si la eficiencia no mejora a partir de este punto ve a otro ejercicio

Post resolución

- La memoria es frágil
- Si se está practicando se debe anotar todos los inconvenientes y errores para no volverlos a cometer en el futuro.
- Y anotar la lógica de programación que usamos para resolver ese problema para no olvidar el nuevo recurso cognitivo que se ganó.
- Si solo resuelven problemas al practicar sin documentar, al paso de unos meses seguirán cometiendo los mismos errores y la lógica de programación puede irse.

3.
Patrones únicos

Todos los problemas de programación tienen patrones

- Para encontrar patrones únicos es importante hacer buenos casos de prueba y buenas visualizaciones. Primero analizar el caso de prueba más simple.
- Todo comienza con una suposición "¿Y si..." Al momento de diagramar en papel pensar que todo se puede programar sin límites.
- A veces el "Eureka" ocurre viendo los casos de prueba, las relaciones de las salidas o de las entradas.
- Si el problema tiene alguna serie de números, no descartar la idea de que tanto la entrada y la salida son una serie.

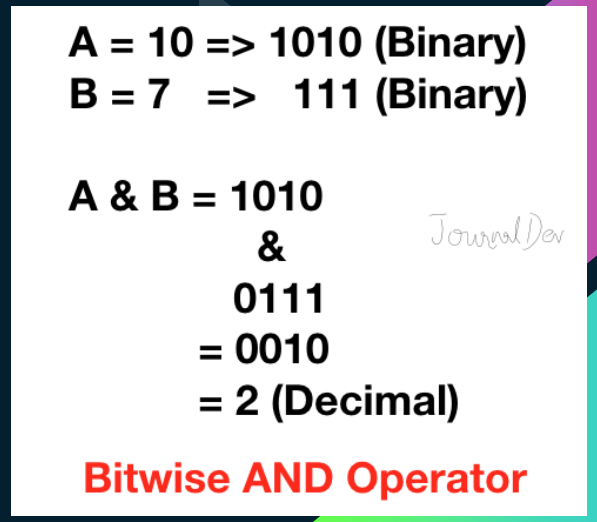
4.
Temas que debes conocer

Temas

- Bitwise operations
- Data structures
- Dynamic programing
- Hash map
- Trees!!!!!!!!!!!!!!!!!!
- Lists
- Sets
- Two pointers
- Sliding window
- String processing
- Montecarlo
- Geometry

https://medium.com/techie-delight/top-algorithms-data-structures-concepts-every-computer-science-student-should-know-e0549c67b4ac

- Breadth First Search (BFS) Algorithm
- Depth First Search (DFS) Algorithm
- Inorder, Preorder, Postorder Tree Traversals
- Insertion Sort, Selection Sort, Merge Sort, Quicksort, Counting Sort, Heap Sort
- Kruskal's Algorithm
- Floyd Warshall Algorithm
- Dijkstra's Algorithm
- Bellman Ford Algorithm
- Kadane's Algorithm
- Lee Algorithm



Siguientes ejercicios

Troll Coder

IEEEXtreme 12.0

66%
1448/2168

Fibonacci

IEEEXtreme 11.0

Matrix Exponentiation
Math
EASY

52%
1303/2466

6. Clock Angles - 20 points

Given an **array of strings** containing string representations of time in `hh:mm` pattern, where `hh` is an integer between 00 and 23 and `mm` is an integer between 00 and 59, calculate the sum of the angles measured between the clock hands. If the input is not on the `hh:mm` pattern, you need to subtract 100 from the result. The angle **MUST** be the one between the hour hand and the minute hand, clockwise, and not the other way around. We ignore the movement caused by seconds. Every angle must be between 0 and 2π (between 0 and 360°).

Example:

Input: `["12:00", "17:30", "blabla", "20:21", "26:88"]`
Output: `50.5`

The function will receive a **string[ ]**, and it must return a **floating point number**.

Ejecución Python vs c++ en Troll Coder

Score: 100/100 (46 ms - 4240 KB)

Test Number	CPU Usage	Memory Usage	Result
102	34 ms	4228 KB	Ok! Used 6 queries. (n+1)
101	34 ms	4236 KB	Ok! Used 6 queries. (n+1)
100	40 ms	4228 KB	Ok! Used 88 queries. (n+0)

Score: 100/100 (10 ms - 1020 KB)

Test Number	CPU Usage	Memory Usage	Result
102	6 ms	976 KB	Ok! Used 1 queries. (n-4)
101	6 ms	840 KB	Ok! Used 6 queries. (n+1)
100	6 ms	824 KB	Ok! Used 88 queries. (n+0)

Gracias

Pueden encontrarme en:

https://magody.github.io/