Estadísticas y soluciones

11 de marzo de 2023

Clasificación de los problemas

Problema	Categoría
A - El bruxeador	Ordenación, voraz
B - IKERobot	Búsqueda, optimización, grafos, heurística
C - Crimen en Villacepé	Grafos, combinatoria
D - El abeXORro	Constructivo, sistemas de ecs lineales
E - Obras de ingeniería	Programación dinámica
F - Fechas de entrenamiento	Programación dinámica (optimización, geometría, envolvente convexa)
G - El jardín del Edén	Programación dinámica / memoización, restricciones
H - El máximo de diversión	Expresiones, bucle simple
I - Razonamiento numérico	Expresiones, strings, memoización, salida
J - Aviones en guerra	Mapas
K - P=NP	Aritmética modular

Estadísticas

Problema	# casos de prueba	Espacio en disco
A - El bruxeador	36	32M
B - IKERobot	14	112K
C - Crimen en Villacepé	38	32M
D - El abeXORro	44	148M
E - Obras de ingeniería	54	1.1M
F - Fechas de entrenamiento	52	648K
G - El jardín del Edén	3	76K
H - El máximo de diversión	4	1.1M
I - Razonamiento numérico	3	5.8M
J - Aviones en guerra	22	256K
K - P=NP	2	132K
- **Total**	272	220M

Estadísticas*

Problema	Primer equipo en resolverlo	Tiempo
J - Aviones en guerra	C mas^2 sus PYTHONisas	0:14
H - El máximo de diversión	Teamto de Verano	0:15
K - P=NP	(3 caritas Moai)	0:15
I - Razonamiento numérico	πk2s	0:19
C - Crimen en Villacepé	Big O No	0:32
B - IKERobot	(3 caritas Moai)	1:37
A - El bruxeador	(3 caritas Moai)	2:32
D - El abeXORro		
E - Obras de ingeniería		
F - Fechas de entrenamiento		
G - El jardín del Edén		

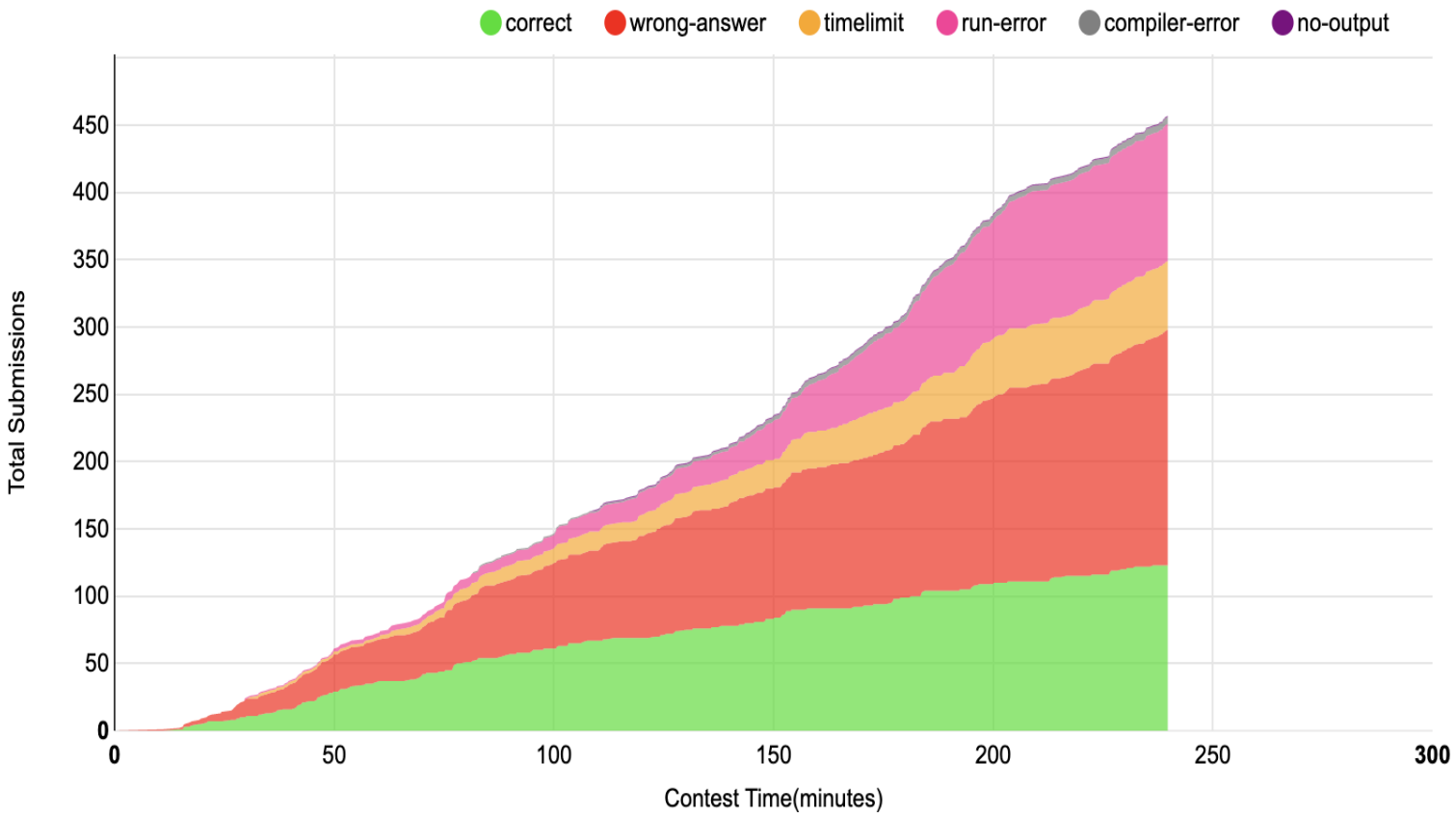
*Antes de congelar el marcador.

Estadísticas*

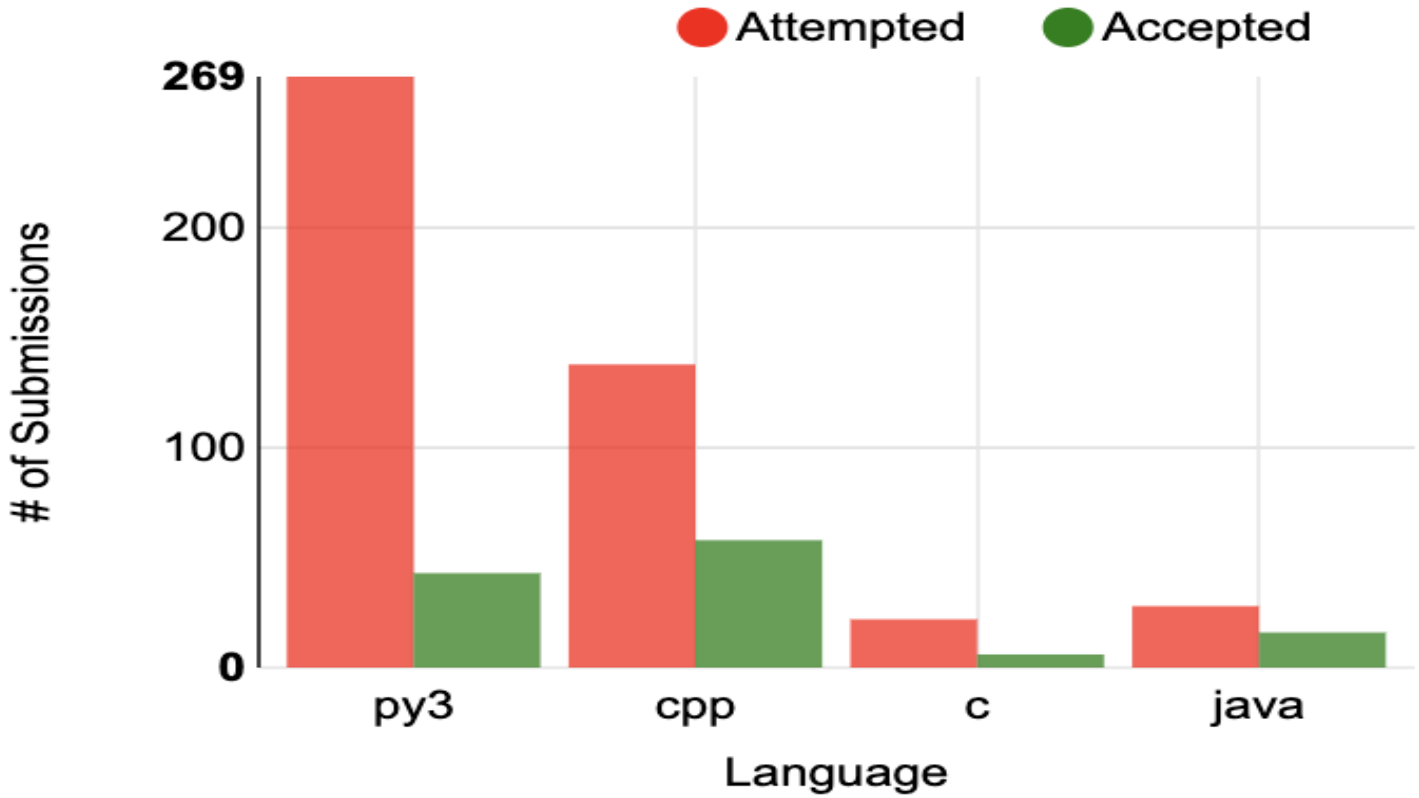
Problema	Envíos	Válidos	% éxito
A - El bruxeador	30	6	20 %
B - IKERobot	16	6	38 %
C - Crimen en Villacepé	8	2	25 %
D - El abeXORro	2	0	0 %
E - Obras de ingeniería	10	0	0 %
F - Fechas de entrenamiento	4	0	0 %
G - El jardín del Edén	0	0	0 %
H - El máximo de diversión	58	42	72 %
I - Razonamiento numérico	74	21	28 %
J - Aviones en guerra	169	32	19 %
K - P=NP	86	14	16 %

*Antes de congelar el marcador.

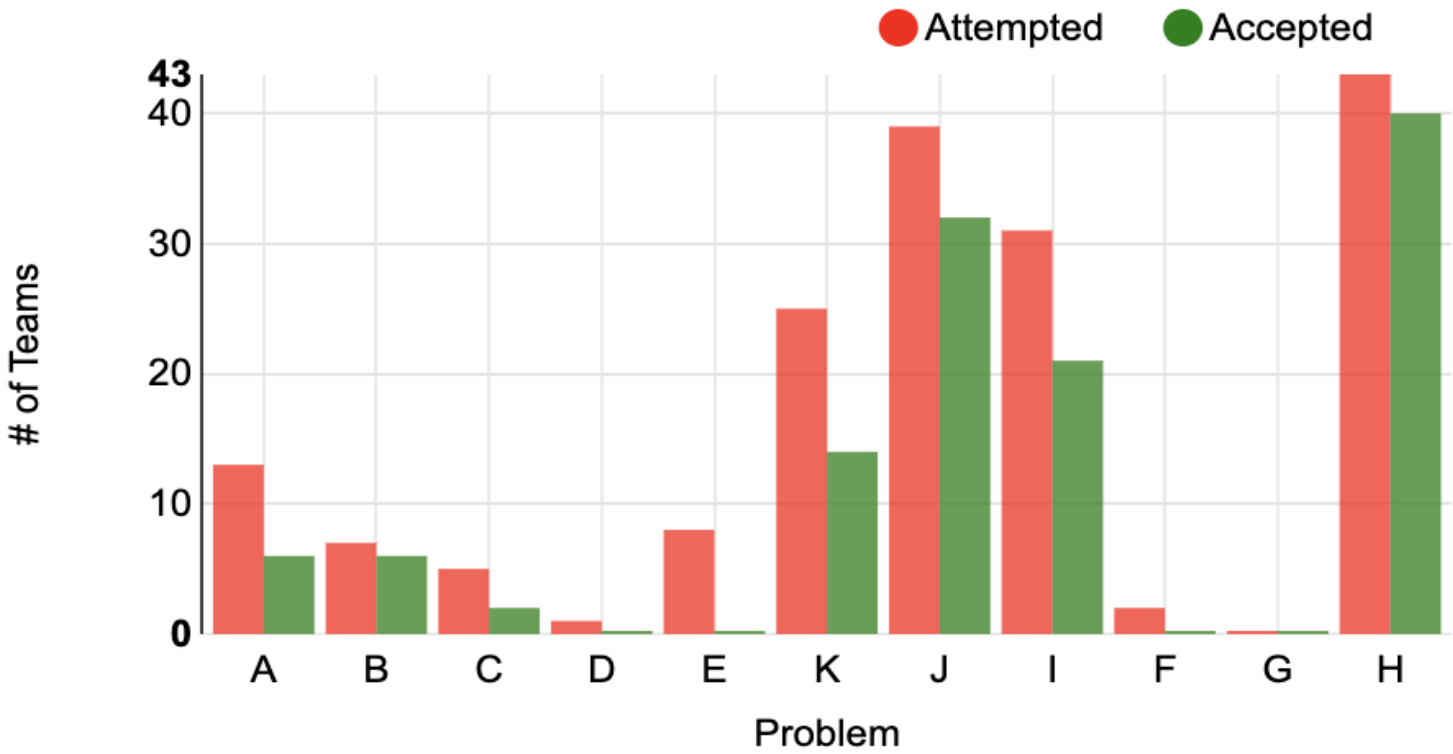
Estadísticas varias



Estadísticas varias



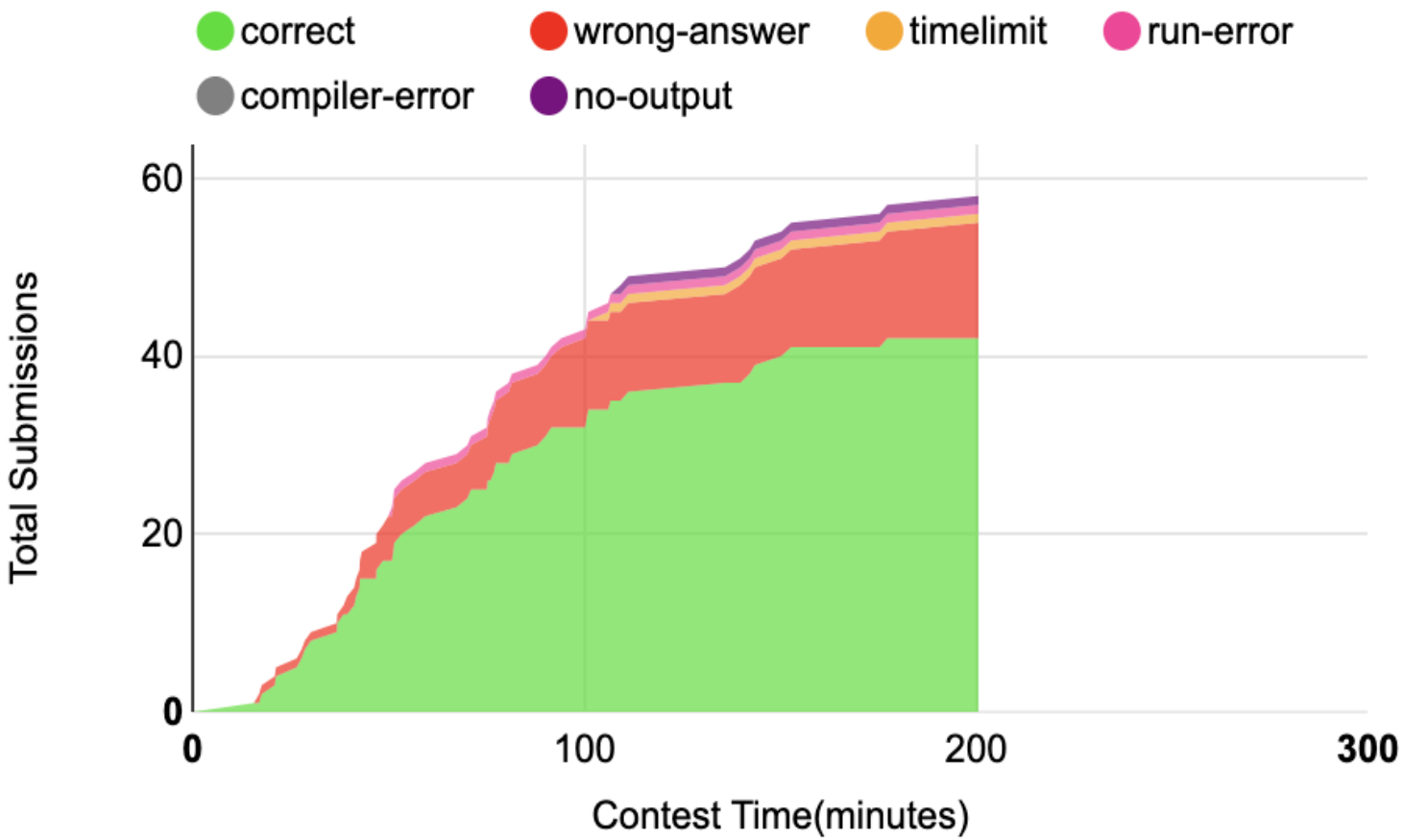
Estadísticas varias



H. El máximo de diversión

Envíos	Válidos	% éxito
58	42	72 %

H. El máximo de diversión



H. El máximo de diversión

La solución a este problema es trivial, solo hay que tener en cuenta unos pocos detalles relativos a la implementación:

1. Utilizar como primer máximo el valor de diversión entre los dos primeros pares o, alternativamente, usar un valor suficientemente pequeño (por ejemplo `Long.MIN_VALUE`)
2. Tener en cuenta que hay $n-1$ pares para n puntos. Y ajustar las veces que se ejecuta el bucle en consecuencia.

La solución mostrada no guarda todo el circuito en memoria (esta característica de la implementación no era estrictamente necesaria)

H. El máximo de diversión

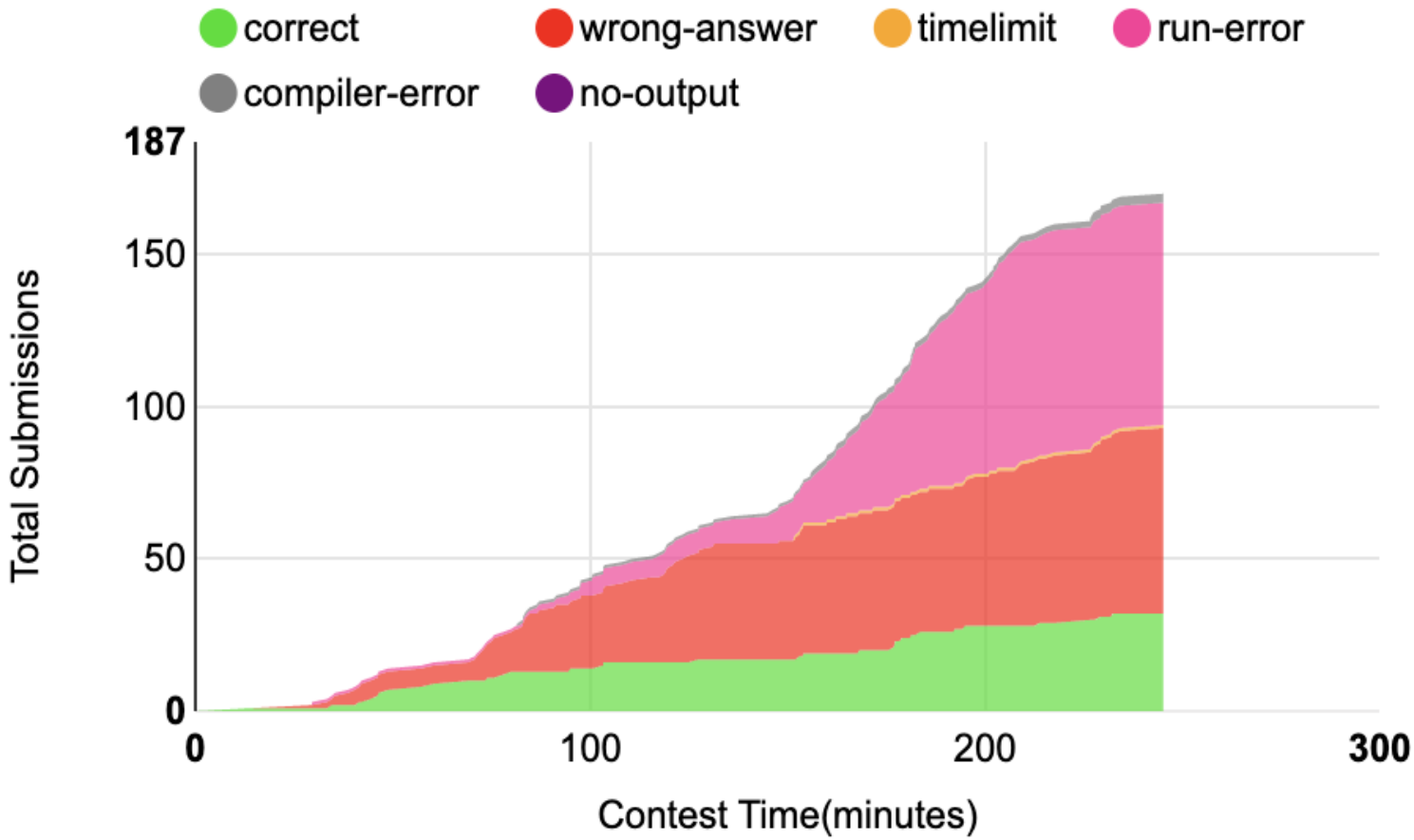
Tomando en cuenta lo anterior, podemos calcular el resultado de un caso de la siguiente manera:

```
leer(n)
leer(x1, y1)
leer(x2, y2)
max := 3 * abs(x2 - x1) + 2 * (y2 - y1)

for i := 2 to n-1
    (x1, y1) <- (x2, y2)
    leer(x2, y2)
    aux := 3 * abs(x2 - x1) + 2 * (y2 - y1)
    if aux > max
        max := aux

escribir(max)
```

J. Aviones en guerra

Envíos	Válidos	% éxito
169	32	19 %

J. Aviones en guerra



J. Aviones en guerra

Dado un avión, determinar qué zona debemos reforzar teniendo en cuenta los disparos.
Para ello, como el avión volvió a casa. Nos fijaremos en las zonas con **menos** disparos, estas zonas son las delicadas para reforzar.
En caso de empates devolveremos la solución según la especificación del enunciado.

J. Aviones en guerra

Algunos errores...

Error Común	Veredicto
Si hay empate no devolver la zona de menor tamaño	WA
Si hay de nuevo empate no devolver el menor identificador	WA
Declarar array de 9 posiciones y no 10	RTE

J. Aviones en guerra

Tomando en cuenta lo anterior, la lectura y almacenamiento

```
int N,M,x,y;
string matriz[M];
mapa{char,int} piezaAvion;
Para todas las M
    Leer fila de matriz[i]
    Para todas las N
        piezaAvion[matrix[i][j]]++;
int balas;
mapa{char,int} golpeado;
for(auto c:piezaAvion)
    if(c.first!='.')
        golpeado[c.first]=0;

Para todas las balas
    Leer x y
    golpeado[matrix[x][y]]++;
...
```

J. Aviones en guerra

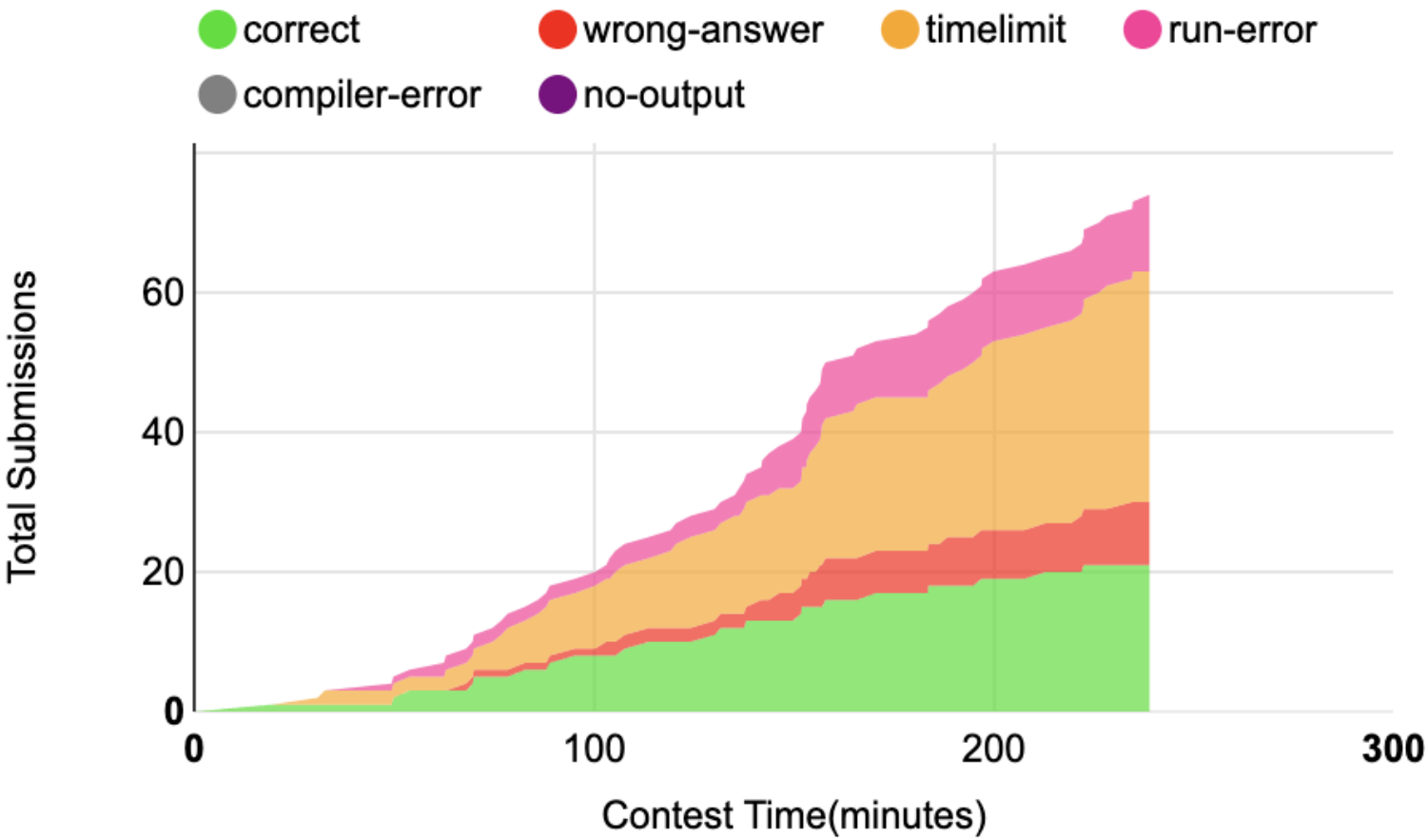
Tomando en cuenta lo anterior, podemos calcular el resultado de la siguiente manera:

```
Para todas las piezas que han sido golpeadas
    si tamañoPiezaActual menor a maxGolpes
        maxGolpes = tamañoPiezaActual;
        maxSize = piezaAvion[piezaActual];
        resultado = piezaActual;
    si en cambio tamañoPiezaActual igual que maxGolpes y
                 maxSize mayor que piezaAvion[piezaActual]
        En caso de empate a golpes, la pieza menor
        maxGolpes = tamañoPiezaActual;
        maxSize = piezaAvion[piezaActual];
        resultado = piezaActual;
    si en cambio tamañoPiezaActual igual que maxGolpes y
                 maxSize igual que piezaAvion[piezaActual]
        En caso de nuevo empate el menor índice
        resultado = mínimo identificador de resultado y piezaActual

Imprimir resultado
```

I. Razonamiento numérico

Envíos	Válidos	% éxito
74	21	28 %

I. Razonamiento numérico



I. Razonamiento numérico

1. Si se observa con un poco de detenimiento las entradas y salidas se puede encontrar que la solución de:

```
12 es 3010
32 es 4935
```

2. y la de 1231 simplemente combina los dos resultado anteriormente conocidos, es decir, la solución de:

```
1232 es 3010 (+) 4935 -> 30104935
```

Asi pues:

1. el cálculo de r para todos los pares de cifras significan muchas operaciones, pero, en realidad
2. sólo hay 81 problemas elementales distintos. Cualquier otro caso se reduce a combinar las soluciones de estos 81. Por otro lado,
3. también (o alternativamente) podemos reducir el cálculo del factorial a solo 9 resultados distintos, puesto que el factorial siempre se aplica a una potencia fija de un número entre 1 y 9.
4. Para calcular el factorial se debe tener en cuenta que: $\text{mód}\ (a * b) = \text{mód}\ (a * \ \text{mód}\ (b))$

I. Razonamiento numérico

Tomando en cuenta lo anterior, antes de calcular los casos, podemos hacer un cálculo previo dado por:

```
for i := 1 to 9
    for j := 1 to
        solucion[i, j] := string(r(i,i+1))
```

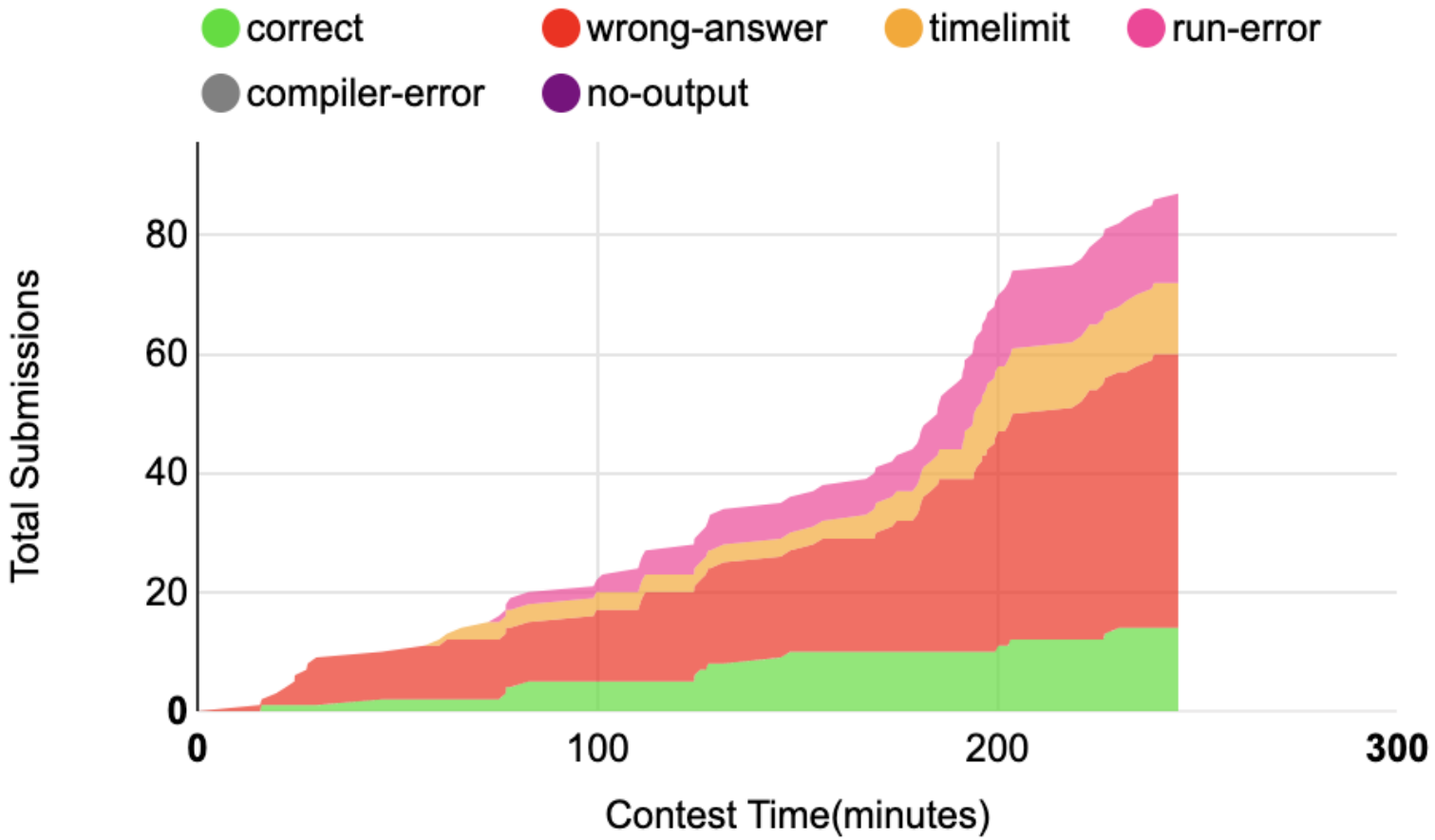
Siendo `r` una función que resuelva las expresiones correspondientes.
Y resolver cada caso con:

```
leer_linea(str)

for i = 1 to length(str) step 2
    resultado := concat(resultado, solucion[i,j])

escribir(resultado)
```

K. P=NP

Envíos	Válidos	% éxito
86	16	16 %

K. P=NP


correct
wrong-answer
timelimit
run-error
compiler-error
no-output
Total Submissions
0
20
40
60
80
0
100
200
300
Contest Time(minutes)

K. P=NP

Problema: Calcular para cuántos valores de P se cumple

$$P=N\cdot P$$

en un *unsigned int* de 32 bits con desbordamiento.

Solución trivial:

```
count = 0
for P in range(2**32):
    if P == N*P % (2**32):
        count += 1
print(count)
```

. . . pero es muy lenta, hay que pensar otra cosa.

K. P=NP

Para que se cumpla P=NP en un unsigned de 32 bits con *overflow*, tiene que cumplirse:

$$P = (N \cdot P) \mathrm{mod} (2^{32})$$

Esto es:

$$P = (N \cdot P) - K \cdot 2^{32}$$

$$P \cdot (N - 1) = K \cdot 2^{32}$$

Además de $P = 0$, el mínimo valor de P que cumple esa ecuación es:

$$P_{\min} = \frac{2^{32}}{\mathrm{GCD}(2^{32}, N - 1)}$$

Y si se cumple para un valor de P, también se cumplirá para todos sus múltiplos, o sea, un total de $\frac{2^{32}}{P_{\min}}$ casos.
Por lo tanto, la solución del problema es simplemente:

$$\mathrm{GCD}(2^{32}, N - 1)$$

K. P=NP

Solución completa del problema, en Python:

```
from math import gcd

M = 2**32
t = int(input())

for _ in range(t):
    print(gcd(M, int(input())-1))
```

B. IKERobot

Envíos	Válidos	% éxito
16	6	38 %

B. IKERobot


correct
wrong-answer
timelimit
run-error
compiler-error
no-output
Total Submissions
15
10
5
0
0
100
200
300
Contest Time(minutes)

B. IKERobot

Enunciado y condiciones

- Encontrar camino entre dos puntos en un plano reticulado.
- Se dan coordenadas de principio y final.
- Hay obstáculos (coordenadas por las que no se puede pasar).
- Minimizar **tiempo**.
  - Avanzar entre dos coordenadas: una unidad de tiempo.
  - Girar: cuatro unidades adicionales de tiempo.
  - ¡Camino más corto puede no ser más rápido!
- Máximo de obstáculos, diferencia entre coordenadas.

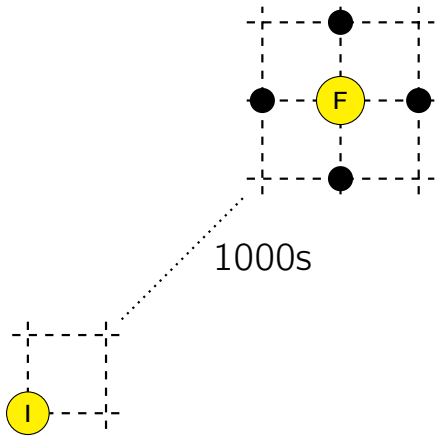
- De manera natural, un problema de grafos:
  - Coordenadas $\longrightarrow$ nodos.
  - Espacio entre coordenadas $\longrightarrow$ aristas.
- Dijkstra.
- Inconveniente: coste del giro.
  - No se adapta bien a las condiciones de Dijkstra.
- Posible solución: *expandir* el grafo con nodos y aristas adicionales para modelizar coste de giros.
  - No es necesario crear un nuevo grafo en memoria — puede ser "virtual".
- Aplicar Dijkstra.
- Grafo más grande, pero debería entrar con los límites del enunciado.

B. IKERobot

Más difícil todavía

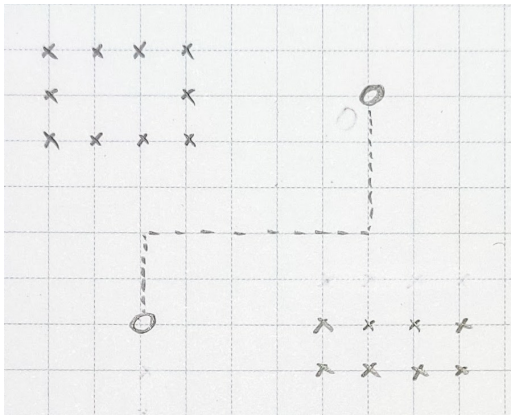
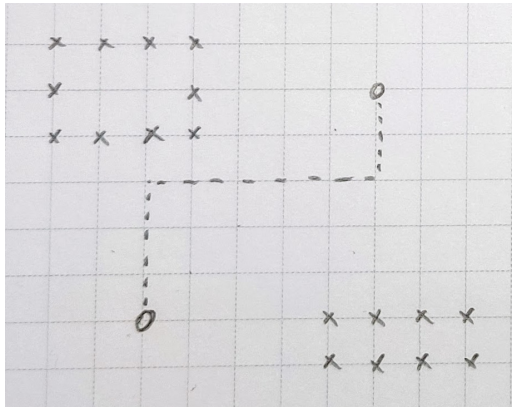
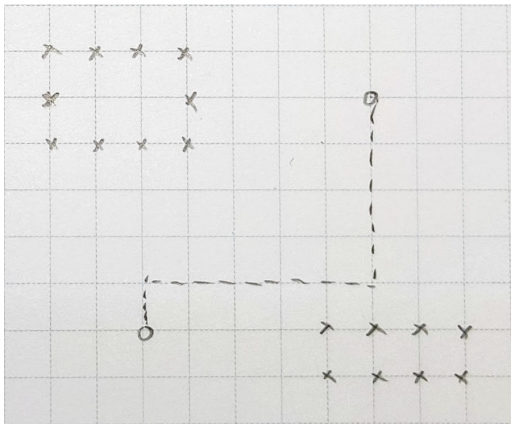
- Dijkstra ignora la posición exacta de origen y destino.
- Intenta hacer trabajo de más.
- Puede ser demasiado lento en grafos muy grandes.
- Posibilidad: **A***.
  - Algoritmo de búsqueda con heurística.
  - Nodos por expandir, ordenados por *coste* asociado a los mismos.
  - Coste: $\overbrace{f(n)}^{\text{Coste total}} = \overbrace{g(n)}^{\text{Coste acumulado}} + \overbrace{h(n)}^{\text{Estimacion del resto}}$
- $h(n)$: aproximación conservadora.
  - P.e., distancia de Manhattan.
  - (Pero Manhattan + 4 si hay giro: más preciso mucho más rápido).
- Garantiza optimalidad si destino alcanzable.
- Pero puede ser caro si no alcanzable (*degenera* en Dijkstra).

- Casos muy grandes con destinos inalcanzables podrían ser infactibles para Djikstra / A*.



(No, no pusimos casos de prueba tan grandes como ese, pero los consideramos).

- Observación: si hay un camino óptimo no trivial que tiene que evitar obstáculos, hay un camino óptimo que **bordea** obstáculos.

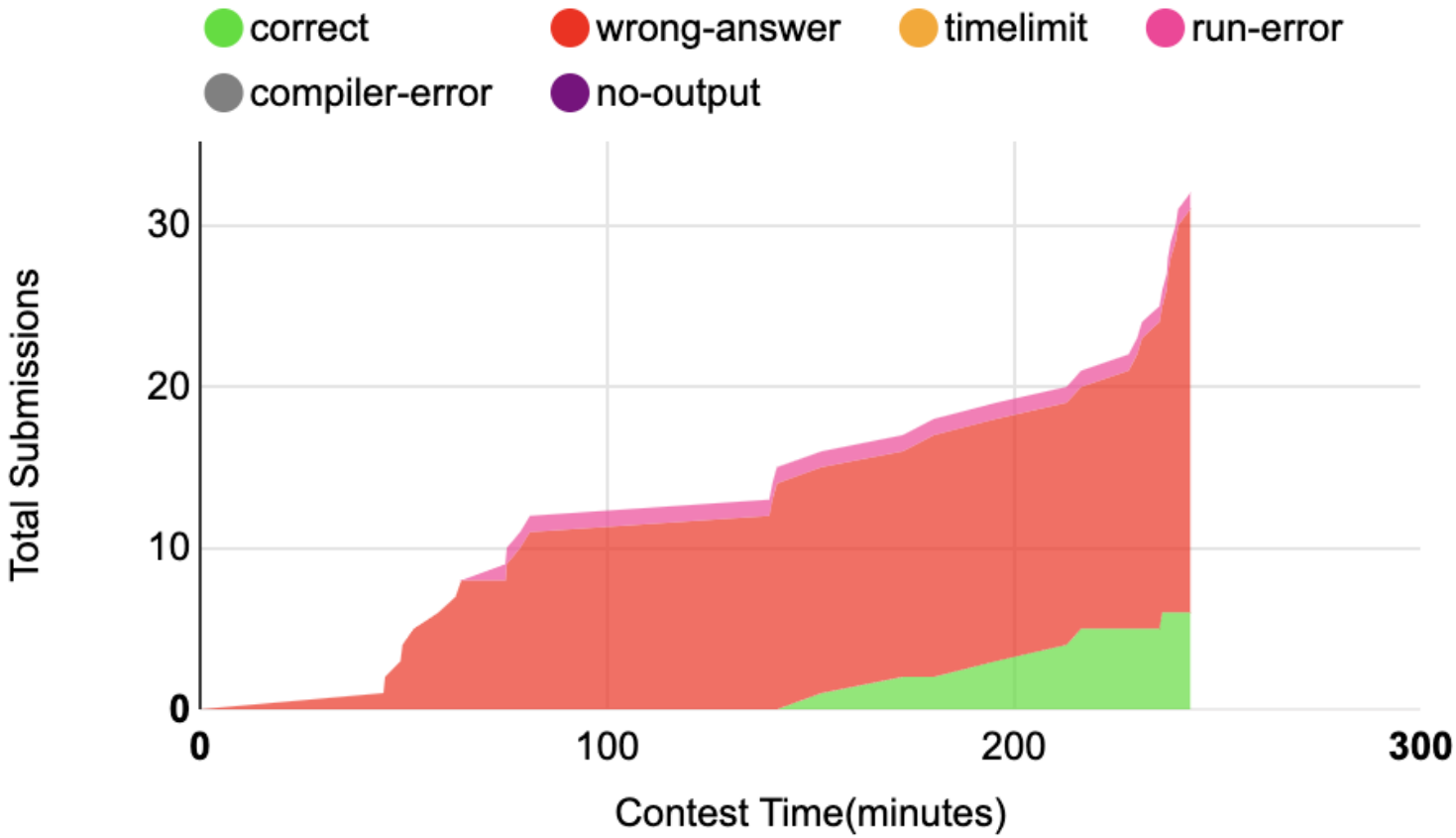




- Conducir el camino bordeando los obstáculos elimina **muchos** caminos superfluos.

A. El bruxeador

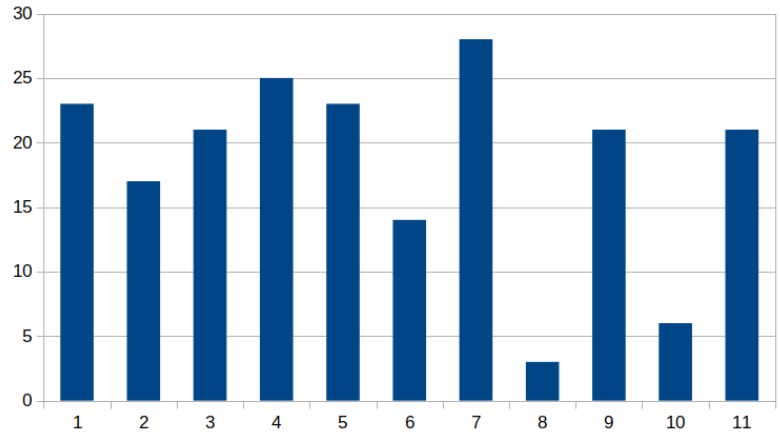
Envíos	Válidos	% éxito
30	6	20 %

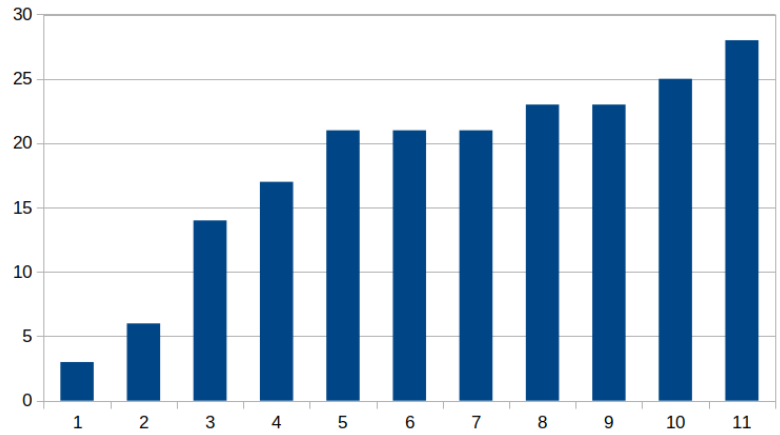
A. El bruxeador



A. El bruxeador

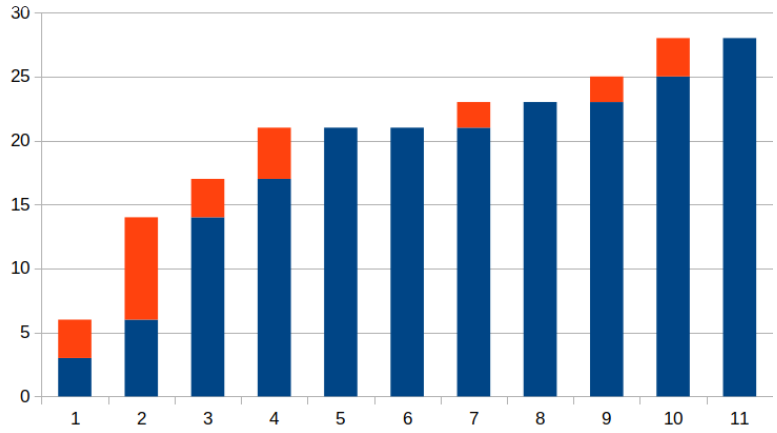
Diferencias consecutivas Se quiere hallar la diferencia entre dos pesas que no tengan ninguna otra con un peso intermedio. La forma más eficiente es ordenar la lista de las pesas.

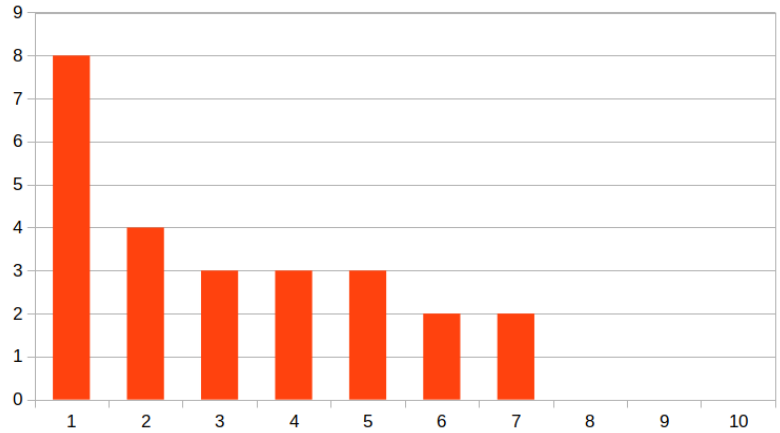




A. El bruxeador

Diferencias consecutivas Una vez que la lista está ordenada, restar cada par de pesas consecutivo para obtener las diferencias.





A. El bruxeador

El resultado es, cada día, quitar la diferencia más grande de las que quedan.
¿Por qué funciona? Separar entre dos seguidos implica que se quita la diferencia entre los dos del coste total.

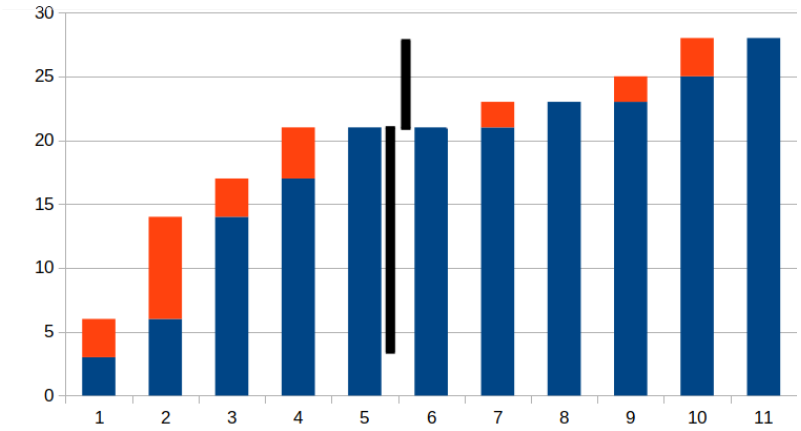


Figura: $c_1 = (28 - 21) + (21 - 3) = 25$

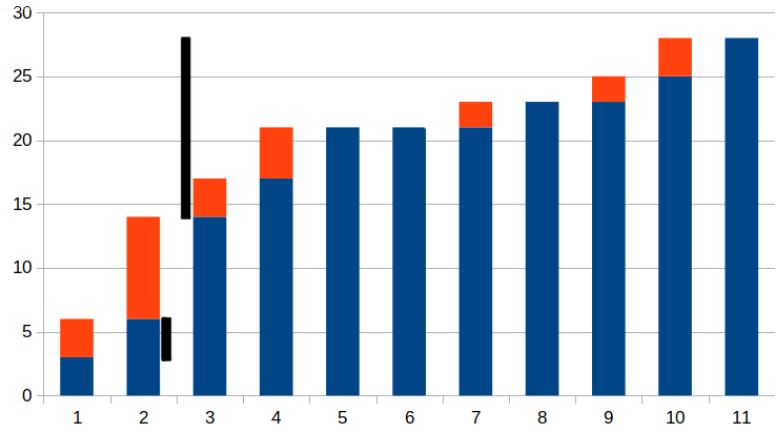


Figura: $c_1 = (28 - 14) + (8 - 3) = 19$

Cada día se quita la diferencia más grande que queda. Al haber quitado las i diferencias más grandes, está garantizado que la solución es óptima.

A. El bruxeador

Pseudocódigo

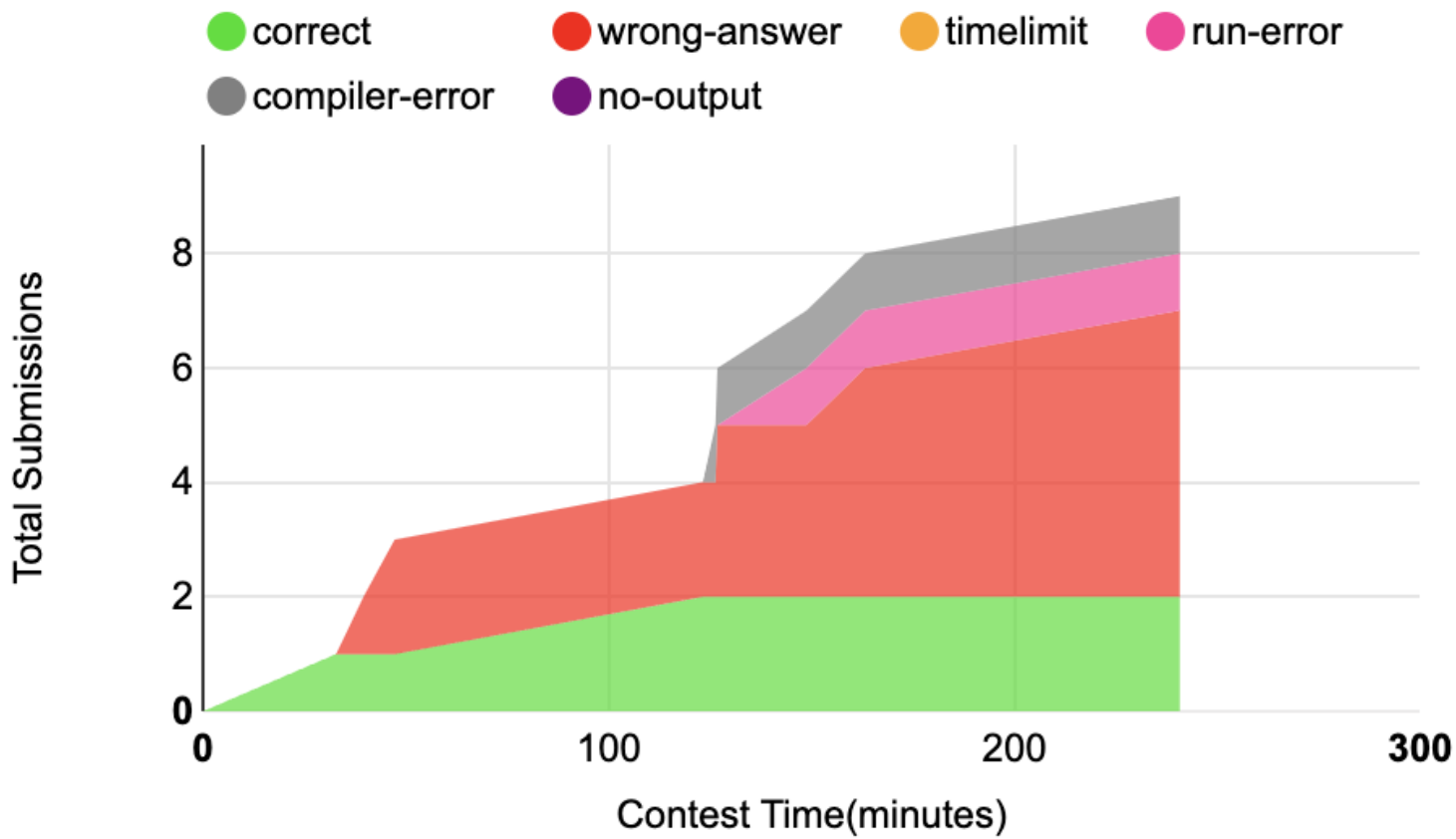
1. Ordenar las pesas
2. Hallar las diferencias consecutivas
3. Ordenar las diferencias (de mayor a menor)
4. Total = Mayor - Menor
5. Para el día $i, 0 \leq i < n$, show(Total), Total -= $diff[i]$.

Complejidad: $\mathcal{O}(n \log n)$

C. Crimen en Villacepé

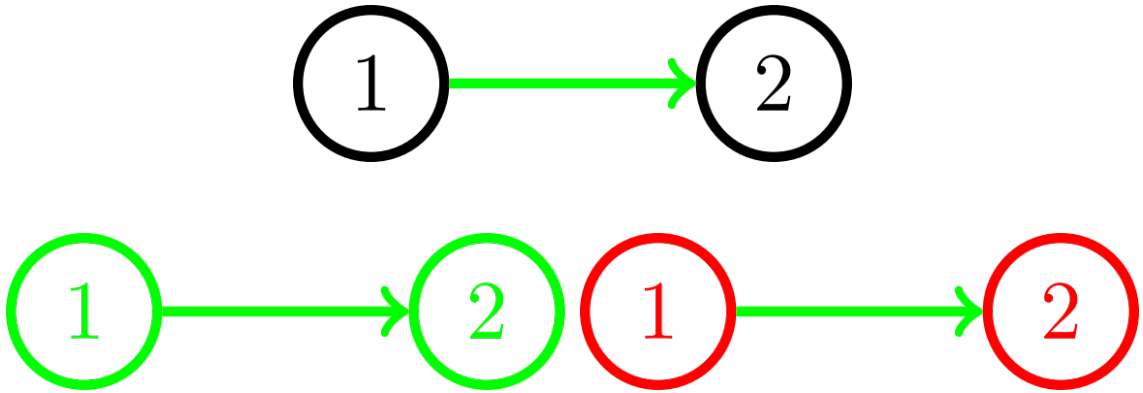
Envíos	Válidos	% éxito
8	2	25 %

C. Crimen en Villacepé

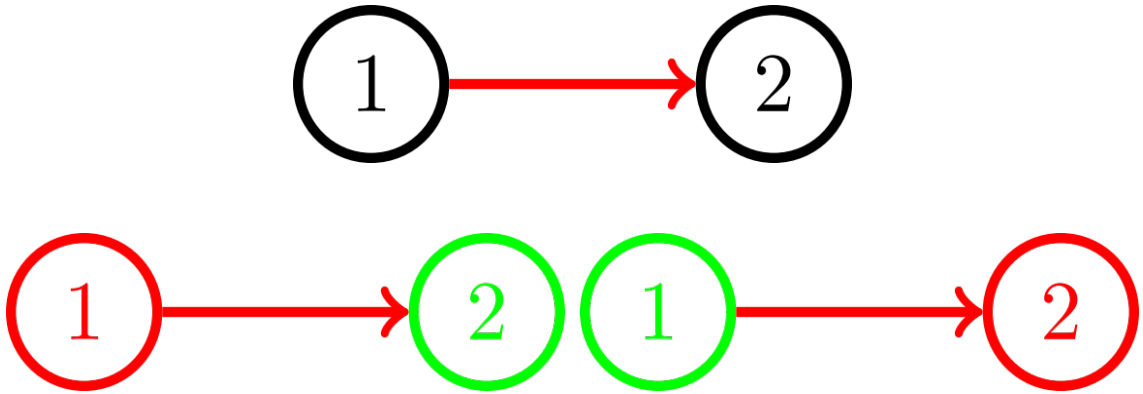


C. Crimen en Villacepé

Simetría de la acusación El grafo, en realidad, no es dirigido. Solo importa que la arista y un vértice determinan completamente al otro vértice.
Cuando la arista es *honesta*:

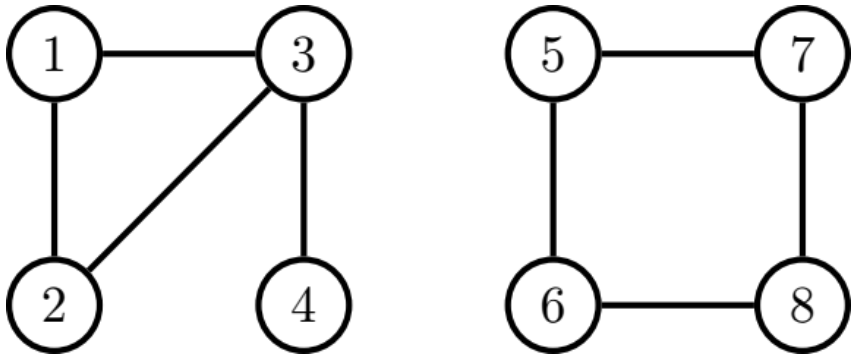


Cuando la arista es *mentira*:



C. Crimen en Villacepé

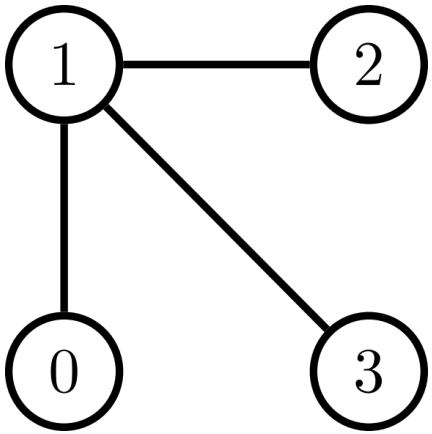
Separar en componentes conexas Cada componente es independiente del resto. El grafo tiene tantas configuraciones como el producto del número de configuraciones sus componentes conexas.



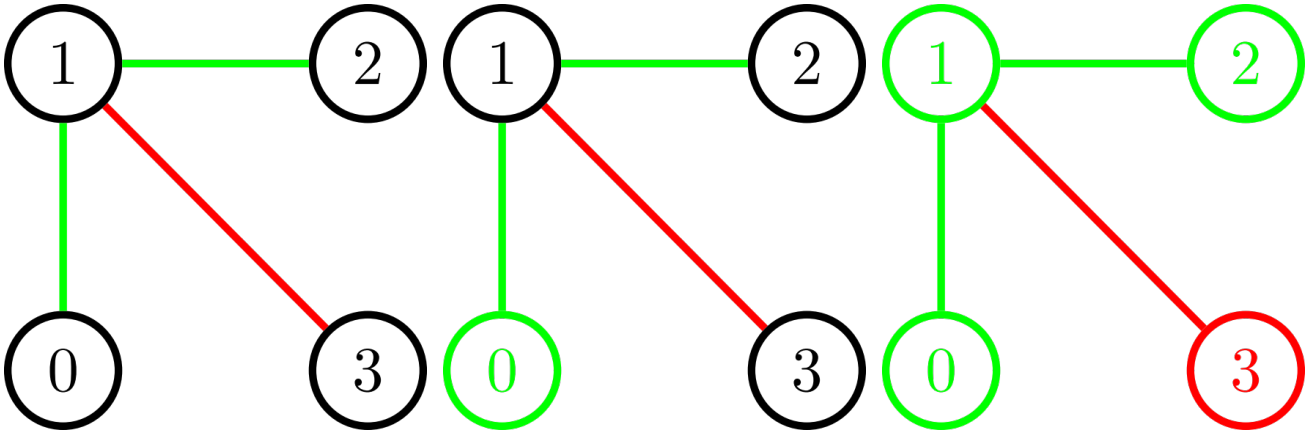
En cada componente hay, como máximo, un ciclo.

C. Crimen en Villacepé

Analizar los árboles Un árbol con n nodos tiene $n-1$ aristas, y permite 2^{n-1} configuraciones distintas.

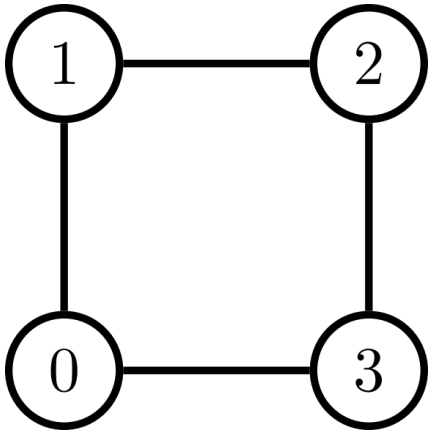


Cualquier configuración de aristas vale, y se pueden asignar los vértices de forma *greedy*.

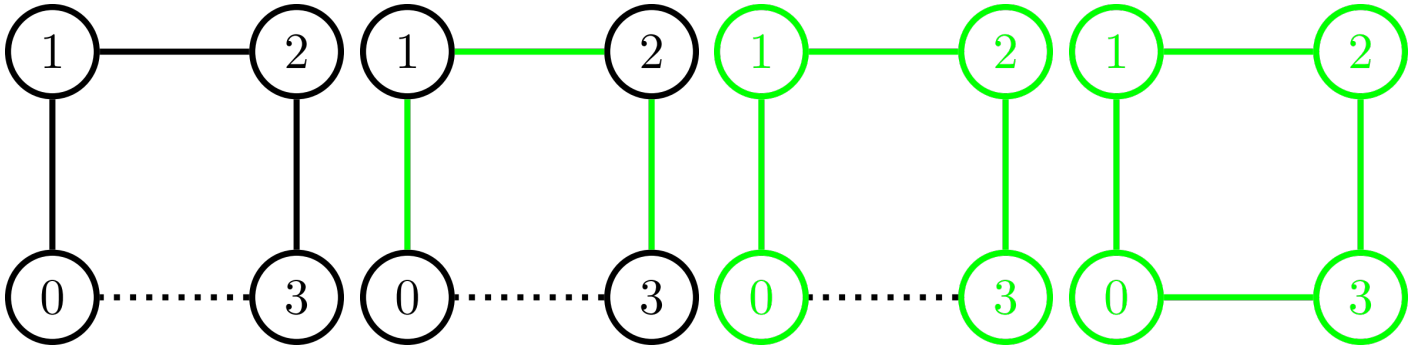


C. Crimen en Villacepé

Analizar los ciclos Un ciclo con n nodos tiene n aristas, y permite 2^{n-1} configuraciones distintas.



Cualquier configuración de $n-1$ aristas vale, pero la última arista queda fijada.



Resultado El número de configuraciones de cada componente es:

$$v(c) = 2^{ciclo(\text{c})-1} \cdot 2^{|c|-\text{ciclo}(c)} = 2^{|c|-1}$$

El resultado para un grafo G es:

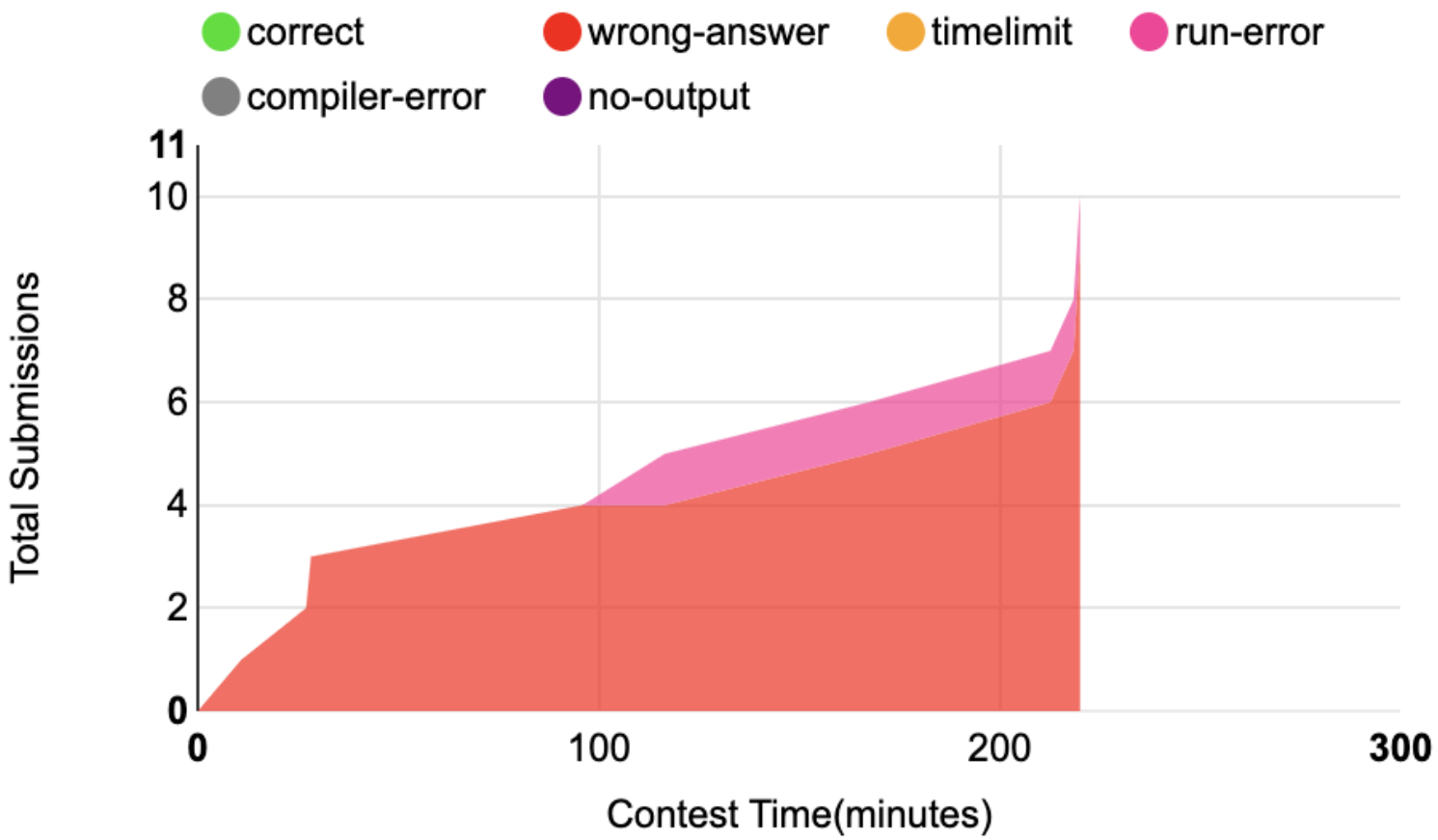
$$R(G) = \prod_c v(c) = 2^{\sum_c |c|-1} = 2^{|G|-\text{num comps}}$$

Complejidad: $\mathcal{O}(n)$
¡¡¡IMPORTANTE!!! siempre calcular el módulo.

E. Obras de ingeniería

Envíos	Válidos	% éxito
10	0	0 %

E. Obras de ingeniería



E. Obras de ingeniería

Problema:

- Array de n números $p_1, p_2, \ldots, p_n$
- Operaciones que cogen un rango de posiciones $[a, b]$ y un número h y cambian todos los números en esas posiciones a h con coste $\sum_{a \leq i \leq b} |h - p_i|$.
- Hacer el array no decreciente usando esas operaciones y minimizando el coste total.

E. Obras de ingeniería

Observaciones:

- El coste de una operación $[a, b]$ es $\sum_{a \leq i \leq b} |h - p_i| = |h - p_a| + \cdots + |h - p_b|$. Es equivalente hacer $b - a + 1$ operaciones, cada una en un rango de una única posición (mismo coste).
- En cada posición se hará como mucho una operación (cambiar por h_1 y luego por h_2 es lo mismo que directamente cambiar por h_2).
- El orden de las operaciones no importa. Vayamos de 1 a n.
- Array no decreciente: si se cambia la posición i por h, en la posición $i - 1$ puede haber cualquier número menor o igual que $h \Rightarrow$ ¿Coste mínimo para array no decreciente en primeras $i - 1$ posiciones y valores menores o iguales que h?

E. Obras de ingeniería

Solución programación dinámica ($dp[i][h]$ = mín coste si se cambia p_i por h):

```
for i in 1..n:
    for h in 1..mx:
        for prev_h in 1..h-1:
            dp[i][h] = min(dp[i][h], dp[i-1][prev_h] + abs(h-a[i]))
```

Demasiado lenta: $O(n \cdot mx^2)$. Optimización: la respuesta para $dp[i][h]$ es el mínimo en un prefijo de $dp[i-1][1], dp[i-1][2], \ldots, dp[i-1][prev_h] \Rightarrow$ Calcular mínimo de prefijos para $dp[i-1]$ antes de pasar a $dp[i]$.

```
for i in 1..n:
    for h in 1..mx:
        dp[i][h] = dp[i-1][h] + abs(h-a[i])

    for h in 1..mx:
        dp[i][h] = min(dp[i][h], dp[i][h-1])
```

Complejidad: $O(n \cdot mx)$

D. El abeXORro

Envíos	Válidos	% éxito
2	0	0 %

D. El abeXORro

D. El abeXORro

Problema:

- Array de n números $a_1, a_2, \ldots, a_n$ y número k.
- Operaciones que cogen dos posiciones distintas i, j y cambian a_i y a_j por x, donde $x = a_i \oplus a_j$ (el XOR *bitwise*).
- Hacer todos los números del array iguales a k usando menos de $4n$ operaciones (o decir que es imposible).

D. El abeXORro

Observaciones:

- El XOR de un número consigo mismo es 0. En general, el XOR de a_i consigo mismo un número par de veces es 0 y un número impar de veces es a_i.
- Los únicos valores posibles son los que se obtienen como el XOR de un subconjunto de los números iniciales del array.
- Si no hay un subconjunto cuyo XOR sea k, la respuesta es NO. Si lo hay, la respuesta es SI ($n \geq 3$).

Si tenemos subconjunto $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$ con XOR k, construcción:

- Obtener k en una posición: obtener sucesivamente $(x_{i_1} \oplus x_{i_2})$, $(x_{i_1} \oplus x_{i_2} \oplus x_{i_3})$, ..., $(x_{i_1} \oplus x_{i_2} \oplus x_{i_3} \oplus \cdots \oplus x_{i_m} = k)$. Ops $\leq n-1$.
- Hacer 0 en las demás posiciones. Por cada dos posiciones que aún no son 0, hacer dos ops en ellas las pone a $a_i \oplus a_j$ y después a $(a_i \oplus a_j) \oplus (a_i \oplus a_j) = 0$. Dos ops por cada dos posiciones. $\Rightarrow$ Ops $\leq n$.
- Hacer operación con posición con valor k y cada una de las otras (que son 0). Ops $\leq n-1$.

¡Ops totales $< 3n$!

D. El abeXORro

¿Cómo buscar un subconjunto $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$ con XOR k?

- Generar todos los posibles subconjuntos y comprobar su XOR: $O(n \cdot 2^n)$, TLE para n grande.
- Programación dinámica $dp[i][x] =$ existe o no un subconjunto con XOR x entre los primeros i elementos: $O(n \cdot 2^{maxbits})$, TLE para n grande o máximo valor del array grande.
- ¿Matrices?

Ejemplo $a = [6,4,1] = [110_2, 100_2, 001_2]$. Obtener $k = 2 = 010_2$. Representamos $(x_1 x_2 x_3)$ donde $x_i = 1$ si se coge a_i en el XOR, 0 si no.

$$\begin{pmatrix}0\\1\\0\end{pmatrix} = x_1\begin{pmatrix}1\\1\\0\end{pmatrix} + x_2\begin{pmatrix}1\\0\\0\end{pmatrix} + x_3\begin{pmatrix}0\\0\\1\end{pmatrix} = \begin{pmatrix}1\cdot x_1 + 1\cdot x_2 + 0\cdot x_3\\1\cdot x_1 + 0\cdot x_2 + 0\cdot x_3\\0\cdot x_1 + 0\cdot x_2 + 1\cdot x_3\end{pmatrix} = \begin{pmatrix}1 & 1 & 0\\1 & 0 & 0\\0 & 0 & 1\end{pmatrix}\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix}$$

¡Resolver un sistema de ecuaciones mód 2! Eliminación de Gauss, $O(m \cdot n \cdot \min(n,m))$, $m = \log(\max a) \Rightarrow O(n \cdot \log^2(\max a))$.

D. El abeXORro

Bonus álgebra lineal: otra forma de encontrar el subconjunto.

- Se pueden ver los números del array como vectores de un espacio vectorial con coordenadas mód 2.
- Si se obtiene una base del espacio vectorial generado por todos los números del array, se puede representar cualquier número (vector) como combinación lineal de la base.
- Ir procesando números de a_1 a a_n. Mantener una base del espacio generado por $a_1, \ldots, a_{i-1}$ y para cada vector de la base, un conjunto de posiciones del array inicial cuyo XOR sea ese vector. Si a_i se puede expresar en esa base no aporta nada, y si no, se añade a la base. Observación: la base no tendrá más de 31 vectores.
- Tenemos la base. Obtener k como combinación de los números iniciales.
- Ops $< \min(3n, 2n + 30)$. Complejidad $O(n \cdot \log^2(\max a))$.

F. Fechas de entrenamiento

Envíos	Válidos	% éxito
4	0	0 %

F. Fechas de entrenamiento

F. Fechas de entrenamiento

Problema:

- Se tienen dos arreglos D y P con números positivos de tamaño N, y un entero K.
- Hay que encontrar K enteros $f_1, \ldots, f_k$ que minimicen la suma $s = \sum_{i<N} P[i] \cdot (f_{m(i)} - D[i])$ donde $m(i) = min\{j : f_j \geq D[i]\}$.

Ejemplo:

- $N = 7$ y $K = 3$
- $D = [1, 3, 5, 6, 8, 9, 10]$
- $P = [8, 13, 14, 9, 12, 5, 11]$

Solución:

- $F = [3, 6, 10]$
- $s = 8 \cdot (3-1) + 13 \cdot (3-3) + 14 \cdot (6-5) + 9 \cdot (6-6) + 12 \cdot (10-8) + 5 \cdot (10-9) + 11 \cdot (10-10) = 59$

F. Fechas de entrenamiento: Solución lenta

Observaciones:

- No es óptimo si f_j no coincide con algún $D[i]$.
- Si se toma $f_j = D[i]$ entonces nos queda resolver el mismo problema en los sufijos $D[i+1\ldots]$ y $P[i+1\ldots]$ con $K-1$.

Solución programación dinámica ($dp[i][k]$: mínimo costo para resolver todos los problemas a partir del día $D[i]$ si se pueden juntar en k fechas):
$dp[i][k] = \text{mín}_{i \le j < N}(dp[j+1][k-1] + \sum_{i \le h \le j}(P[h] \cdot (D[j] - D[h])))$
Complejidad: $\mathcal{O}(N \cdot K \cdot N \cdot N)$. Muy lento.
Vamos a trabajarla...
$dp[i][k] = \text{mín}_{i \le j}(dp[j+1][k-1] + (\sum_{i \le h \le j} P[h]) \cdot D[j] - (\sum_{i \le h \le j}(P[h] \cdot D[h])))$
$dp[i][k] = \text{mín}_{i \le j}(dp[j+1][k-1] + (accP[j] - accP[i]) \cdot D[j] - (accPD[j] - accPD[i]))$
donde: $accP[i+1] = accP[i] + P[i]$ y $accPD[i+1] = accPD[i] + P[i] \cdot D[i]$
Complejidad: $\mathcal{O}(N \cdot K \cdot N)$. Lento.

F. Fechas de entrenamiento: Solución lenta

Observaciones:

- No es óptimo si f_j no coincide con algún $D[i]$.
- Si se toma $f_j = D[i]$ entonces nos queda resolver el mismo problema en los sufijos $D[i+1\ldots]$ y $P[i+1\ldots]$ con $K-1$.

Solución programación dinámica ($dp[i][k]$: mínimo costo para resolver todos los problemas a partir del día $D[i]$ si se pueden juntar en k fechas):
$dp[i][k] = \text{mín}_{i \le j < N}(dp[j+1][k-1] + \sum_{i \le h \le j}(P[h] \cdot (D[j] - D[h])))$
Complejidad: $\mathcal{O}(N \cdot K \cdot N \cdot N)$. Muy lento.
Vamos a trabajarla...
$dp[i][k] = \text{mín}_{i \le j}(dp[j+1][k-1] + (\sum_{i \le h \le j} P[h]) \cdot D[j] - (\sum_{i \le h \le j}(P[h] \cdot D[h])))$
$dp[i][k] = \text{mín}_{i \le j}(dp[j+1][k-1] + (accP[j] - accP[i]) \cdot D[j] - (accPD[j] - accPD[i]))$
donde: $accP[i+1] = accP[i] + P[i]$ y $accPD[i+1] = accPD[i] + P[i] \cdot D[i]$
Complejidad: $\mathcal{O}(N \cdot K \cdot N)$. Lento.

F. Fechas de entrenamiento: Solución lenta

Observaciones:

- No es óptimo si f_j no coincide con algún $D[i]$.
- Si se toma $f_j = D[i]$ entonces nos queda resolver el mismo problema en los sufijos $D[i+1\ldots]$ y $P[i+1\ldots]$ con $K-1$.

Solución programación dinámica ($dp[i][k]$: mínimo costo para resolver todos los problemas a partir del día $D[i]$ si se pueden juntar en k fechas):
$dp[i][k] = \text{mín}_{i \le j < N}(dp[j+1][k-1] + \sum_{i \le h \le j}(P[h] \cdot (D[j] - D[h])))$
Complejidad: $\mathcal{O}(N \cdot K \cdot N \cdot N)$. Muy lento.
Vamos a trabajarla...
$dp[i][k] = \text{mín}_{i \le j}(dp[j+1][k-1] + (\sum_{i \le h \le j} P[h]) \cdot D[j] - (\sum_{i \le h \le j}(P[h] \cdot D[h])))$
$dp[i][k] = \text{mín}_{i \le j}(dp[j+1][k-1] + (accP[j] - accP[i]) \cdot D[j] - (accPD[j] - accPD[i]))$
donde: $accP[i+1] = accP[i] + P[i]$ y $accPD[i+1] = accPD[i] + P[i] \cdot D[i]$
Complejidad: $\mathcal{O}(N \cdot K \cdot N)$. Lento.

F. Fechas de entrenamiento: Solución lenta

Observaciones:

- No es óptimo si f_j no coincide con algún $D[i]$.
- Si se toma $f_j = D[i]$ entonces nos queda resolver el mismo problema en los sufijos $D[i+1\ldots]$ y $P[i+1\ldots]$ con $K-1$.

Solución programación dinámica ($dp[i][k]$: mínimo costo para resolver todos los problemas a partir del día $D[i]$ si se pueden juntar en k fechas):
$dp[i][k] = \text{mín}_{i \le j < N}(dp[j+1][k-1] + \sum_{i \le h \le j}(P[h] \cdot (D[j] - D[h])))$
Complejidad: $\mathcal{O}(N \cdot K \cdot N \cdot N)$. Muy lento.
Vamos a trabajarla...
$dp[i][k] = \text{mín}_{i \le j}(dp[j+1][k-1] + (\sum_{i \le h \le j} P[h]) \cdot D[j] - (\sum_{i \le h \le j}(P[h] \cdot D[h])))$
$dp[i][k] = \text{mín}_{i \le j}(dp[j+1][k-1] + (accP[j] - accP[i]) \cdot D[j] - (accPD[j] - accPD[i]))$
donde: $accP[i+1] = accP[i] + P[i]$ y $accPD[i+1] = accPD[i] + P[i] \cdot D[i]$
Complejidad: $\mathcal{O}(N \cdot K \cdot N)$. Lento.

F. Fechas de entrenamiento: Solución lenta

Observaciones:

- No es óptimo si f_j no coincide con algún $D[i]$.
- Si se toma $f_j = D[i]$ entonces nos queda resolver el mismo problema en los sufijos $D[i+1\ldots]$ y $P[i+1\ldots]$ con $K-1$.

Solución programación dinámica ($dp[i][k]$: mínimo costo para resolver todos los problemas a partir del día $D[i]$ si se pueden juntar en k fechas):
$dp[i][k] = \text{mín}_{i \le j < N}(dp[j+1][k-1] + \sum_{i \le h \le j}(P[h] \cdot (D[j] - D[h])))$
Complejidad: $\mathcal{O}(N \cdot K \cdot N \cdot N)$. Muy lento.
Vamos a trabajarla...
$dp[i][k] = \text{mín}_{i \le j}(dp[j+1][k-1] + (\sum_{i \le h \le j} P[h]) \cdot D[j] - (\sum_{i \le h \le j}(P[h] \cdot D[h])))$
$dp[i][k] = \text{mín}_{i \le j}(dp[j+1][k-1] + (accP[j] - accP[i]) \cdot D[j] - (accPD[j] - accPD[i]))$
donde: $accP[i+1] = accP[i] + P[i]$ y $accPD[i+1] = accPD[i] + P[i] \cdot D[i]$
Complejidad: $\mathcal{O}(N \cdot K \cdot N)$. Lento.

F. Fechas de entrenamiento: Solución lenta

Observaciones:

- No es óptimo si f_j no coincide con algún $D[i]$.
- Si se toma $f_j = D[i]$ entonces nos queda resolver el mismo problema en los sufijos $D[i+1\ldots]$ y $P[i+1\ldots]$ con $K-1$.

Solución programación dinámica ($dp[i][k]$: mínimo costo para resolver todos los problemas a partir del día $D[i]$ si se pueden juntar en k fechas):
$dp[i][k] = \text{mín}_{i \le j < N}(dp[j+1][k-1] + \sum_{i \le h \le j}(P[h] \cdot (D[j] - D[h])))$
Complejidad: $\mathcal{O}(N \cdot K \cdot N \cdot N)$. Muy lento.
Vamos a trabajarla...
$dp[i][k] = \text{mín}_{i \le j}(dp[j+1][k-1] + (\sum_{i \le h \le j} P[h]) \cdot D[j] - (\sum_{i \le h \le j}(P[h] \cdot D[h])))$
$dp[i][k] = \text{mín}_{i \le j}(dp[j+1][k-1] + (accP[j] - accP[i]) \cdot D[j] - (accPD[j] - accPD[i]))$
donde: $accP[i+1] = accP[i] + P[i]$ y $accPD[i+1] = accPD[i] + P[i] \cdot D[i]$
Complejidad: $\mathcal{O}(N \cdot K \cdot N)$. Lento.

F. Fechas de entrenamiento: Solución con optimización D&C

Queremos calcular:
$dp[i][k] = \text{mín}_{i \leq j}(dp[j+1][k-1] + (accP[j] - accP[i]) \cdot D[j] - (accPD[j] - accPD[i]))$

- Sea $opt[i][k]$ el mínimo j tal que $dp[i][k] = dp[j+1][k-1] + C[i][j]$.
  Entonces, $opt[0][k] \leq opt[1][k] \leq \ldots \leq opt[N-1][k]$.
- Podemos aplicar **Divide and Conquer**:
  Si tenemos calculado $dp[i][k-1]$ para todo i, podemos computar $dp[N/2][k]$ junto con $opt[N/2][k]$.
  Luego, cuando computamos $dp[0][k] \ldots dp[N/2-1][k]$ sabemos que el j debe estar entre 1 y $opt[N/2][k]$ mientras que para $dp[N/2+1][k] \ldots dp[N-1][k]$ estará entre $opt[N/2][k]$ y $N-1$.

```
calcuar(k, L, R, optL, optR):
    caso base: L == R
    Sea M = (L+R)/2 calculamos dp[k][M] y
    opt[k][M] recorriendo los valores entre optL y optR
    calcular(k, L, M-1, optL, opt[k][M])
    calcular(k, M+1, R, opt[k][M], optR)
```

- profundidad: $\mathcal{O}(\log(N))$
- operaciones por nivel: $2N$

Complejidad: $\mathcal{O}(N \cdot K \cdot \log(N))$

F. Fechas de entrenamiento: Solución con optimización D&C

Queremos calcular:
$dp[i][k] = \text{mín}_{i \leq j < N}(dp[j+1][k-1] + C[i][j])$

- Sea $opt[i][k]$ el mínimo j tal que $dp[i][k] = dp[j+1][k-1] + C[i][j]$.
  Entonces, $opt[0][k] \leq opt[1][k] \leq \ldots \leq opt[N-1][k]$.
- Podemos aplicar **Divide and Conquer**:
  Si tenemos calculado $dp[i][k-1]$ para todo i, podemos computar $dp[N/2][k]$ junto con $opt[N/2][k]$.
  Luego, cuando computamos $dp[0][k] \ldots dp[N/2-1][k]$ sabemos que el j debe estar entre 1 y $opt[N/2][k]$ mientras que para $dp[N/2+1][k] \ldots dp[N-1][k]$ estará entre $opt[N/2][k]$ y $N-1$.

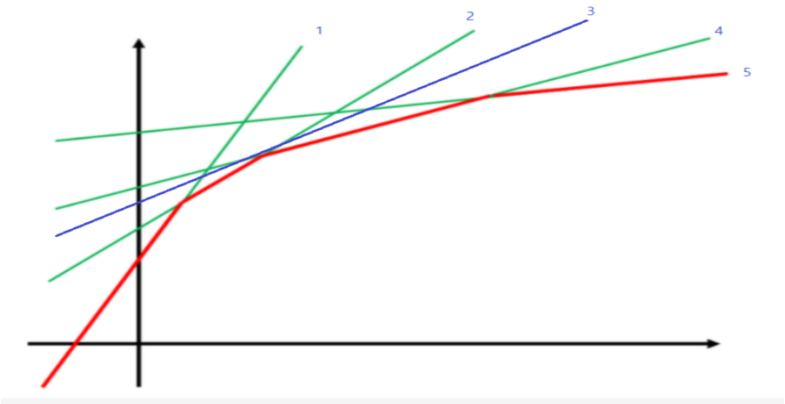
```
calcuar(k,L,R,optL,optR):
    caso base: L == R
    Sea M = (L+R)/2 calculamos dp[k][M] y
    opt[k][M] recorriendo los valores entre optL y optR
    calcular(k,L,M-1,optL,opt[k][M])
    calcular(k,M+1,R,opt[k][M],optR)
```

- profundidad: $\mathcal{O}(\log(N))$
- operaciones por nivel: $2N$

Complejidad: $\mathcal{O}(N \cdot K \cdot \log(N))$

F. Fechas de entrenamiento: Solución con optimización Convex Hull Trick

Queremos calcular
$dp[i][k] = \text{mín}_{i \leq j}(dp[j+1][k-1] + (accP[j] - accP[i]) \cdot D[j] - (accPD[j] - accPD[i]))$
$= accPD[i] + \text{mín}_{i \leq j}(dp[j+1][k-1] + accP[j] \cdot D[j] - accPD[j] - accP[i] \cdot D[j]) =$
$accPD[i] + \text{mín}_{i \leq j}(h_{k-1}[j] + x[i] \cdot m[j])$
Podemos usar la estructura Convex Hull Trick!

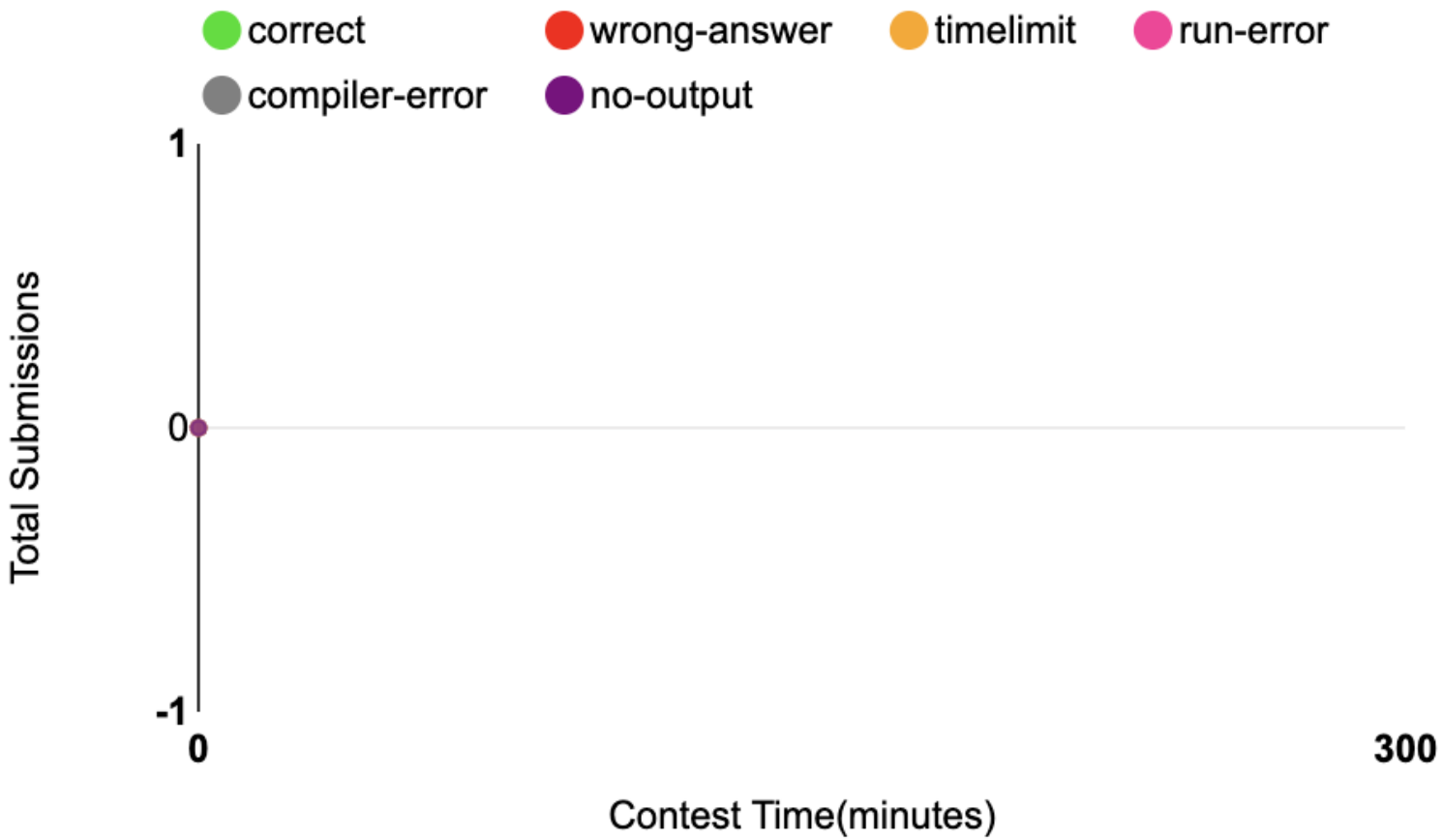


convexHullTrick:
`add(m,h)` agrega la linea `m*x+h` manteniendo la Convex Hull.
`query(x)`: devuelve mínimo `y = m*x+h`.
Se pueden implementar en $\mathcal{O}(1)$ si `m` y `x`. están ordenados, respectivamente.
Complejidad: $\mathcal{O}(N \cdot K)$

G. El jardín del Edén

Envíos	Válidos	% éxito
0	0	0 %

G. El jardín del Edén

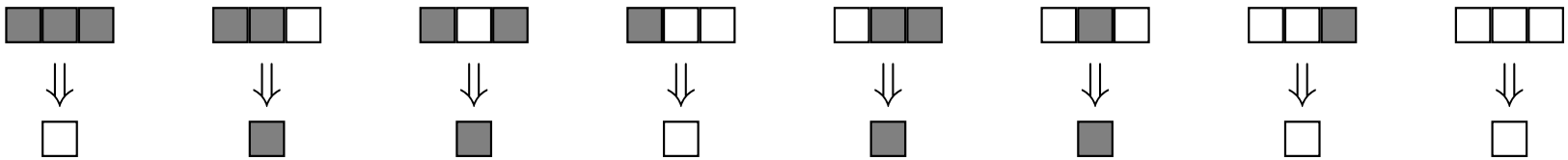


G. El jardín del Edén

Enunciado y condiciones

- Autómata celular lineal: células vivas o muertas.

- Transiciones: células viven o mueren de acuerdo con reglas.



- Para un estado E y una regla dada, ¿hay otro estado **distinto** de E que evolucione a E un paso?
  - Sí: *Jardin del Edén*
- Estado autómata, reglas: representado con bits.
- Límites: estado autómata ≤ 30 células (bits).

G. El jardín del Edén

Primera opción

- Probar estados a ver si son el antecesor de otro dado.

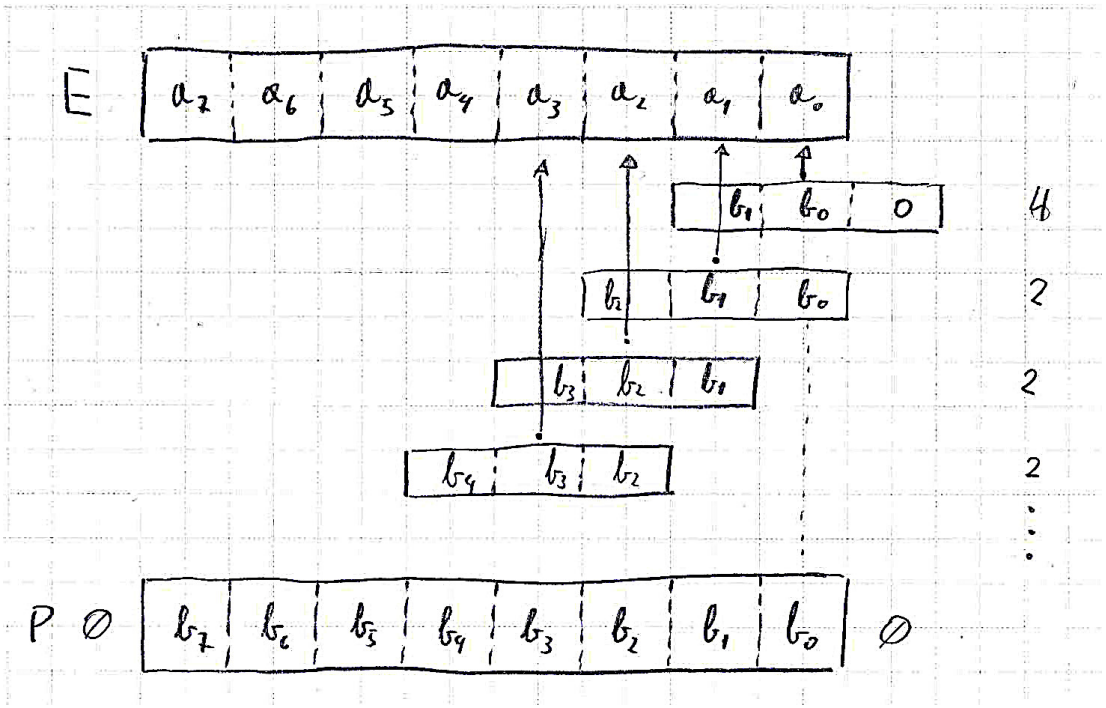
```
for (int posPrev = 0; posPrev < (1 << nCells); posPrev++)
  if (posPrev != posGOE) {
    int next = 0, padded_conf = posPrev << 1;
    for (int cell = 0; cell < nCells; cell++) {
      next |= (1 & (rNumber >>> (padded_conf & 7))) << cell;
      padded_conf >>>= 1;
    }
    if (next == posGOE)
      return false;
  }
return true;
```

- Pequeño problema: enumerar hasta 2^{30} estados.

G. El jardín del Edén

Reconstrucción del pasado

- Idea: encontrar qué configuración/es pueden haber dado E.
- Verificar que hay alguna que no es E.
- Búsqueda con algunas restricciones.
- Decidir si se repite configuración inicial: llevar testigo de cambios en cualquier caso, comprobar al final.

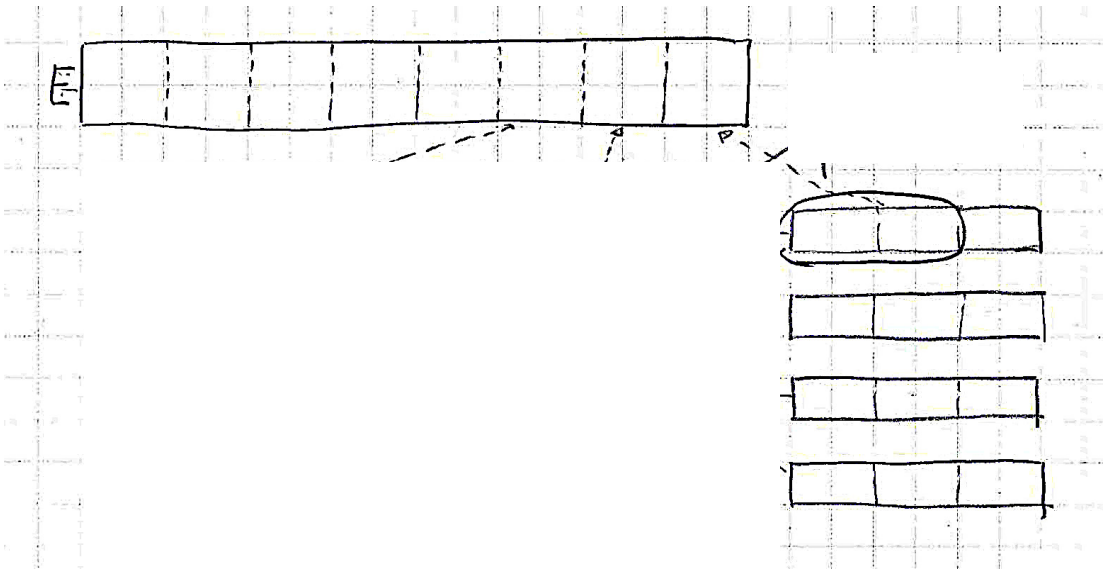


- Sigue siendo problemático: complejidad caso peor 2^n.

G. El jardín del Edén

Reconstrucción del pasado++

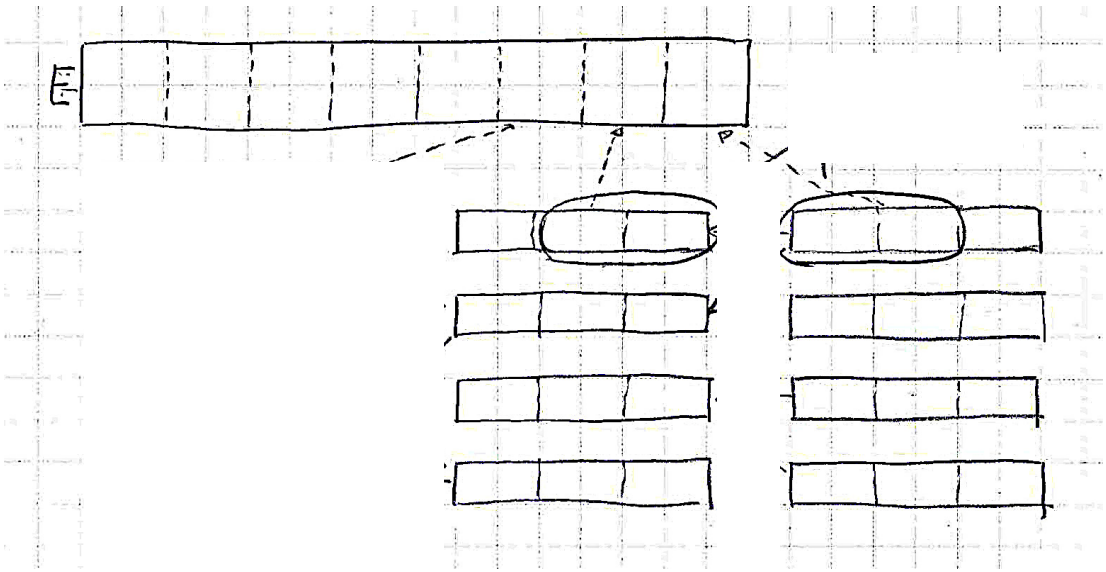
- Mucho trabajo: selección de patrones aplicables en el siguiente paso.



G. El jardín del Edén

Reconstrucción del pasado++

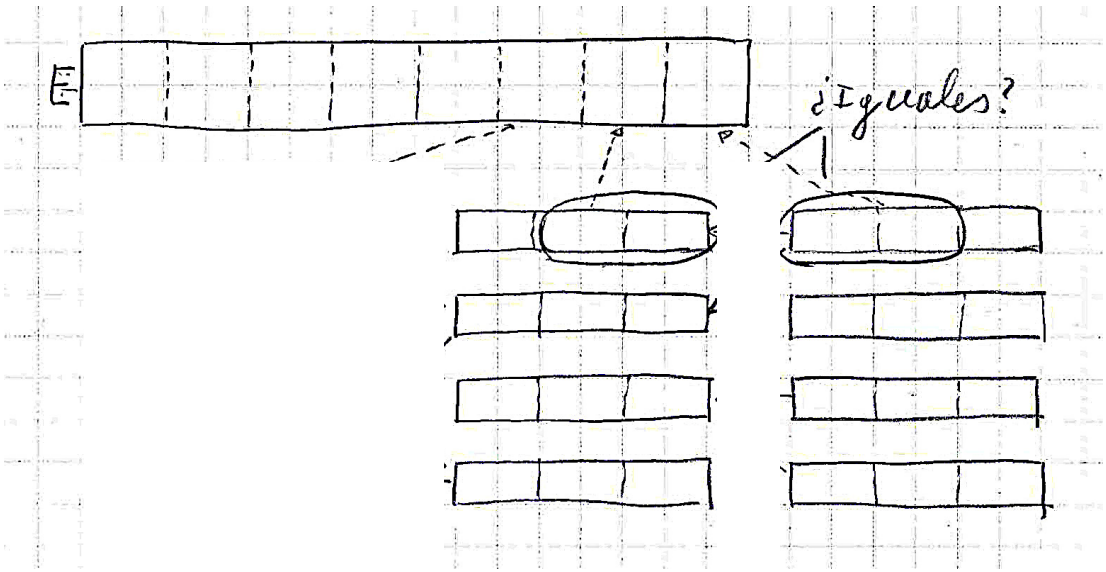
- Mucho trabajo: selección de patrones aplicables en el siguiente paso.



G. El jardín del Edén

Reconstrucción del pasado++

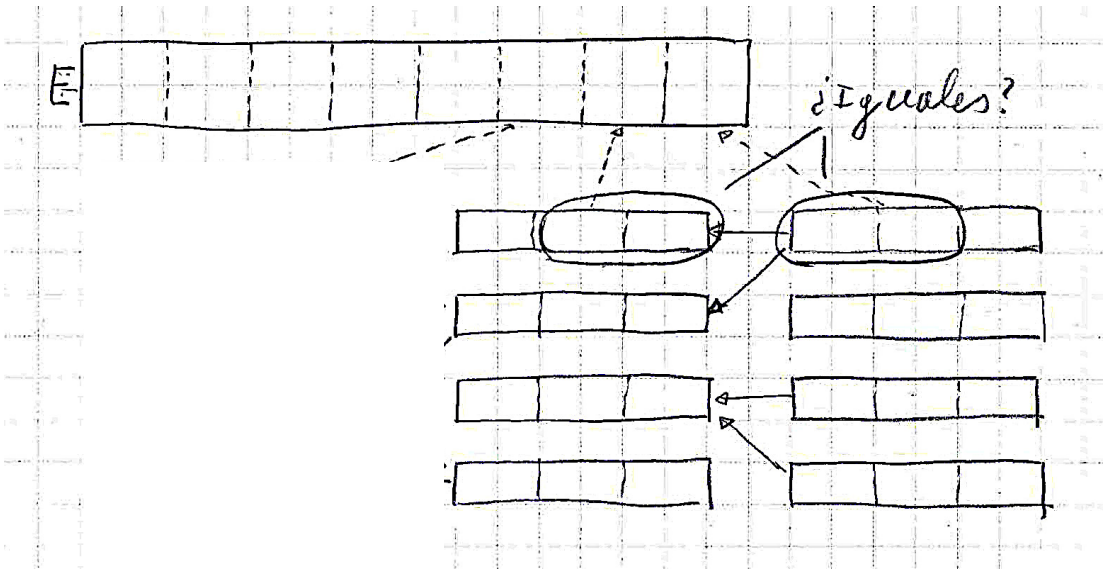
- Mucho trabajo: selección de patrones aplicables en el siguiente paso.



G. El jardín del Edén

Reconstrucción del pasado++

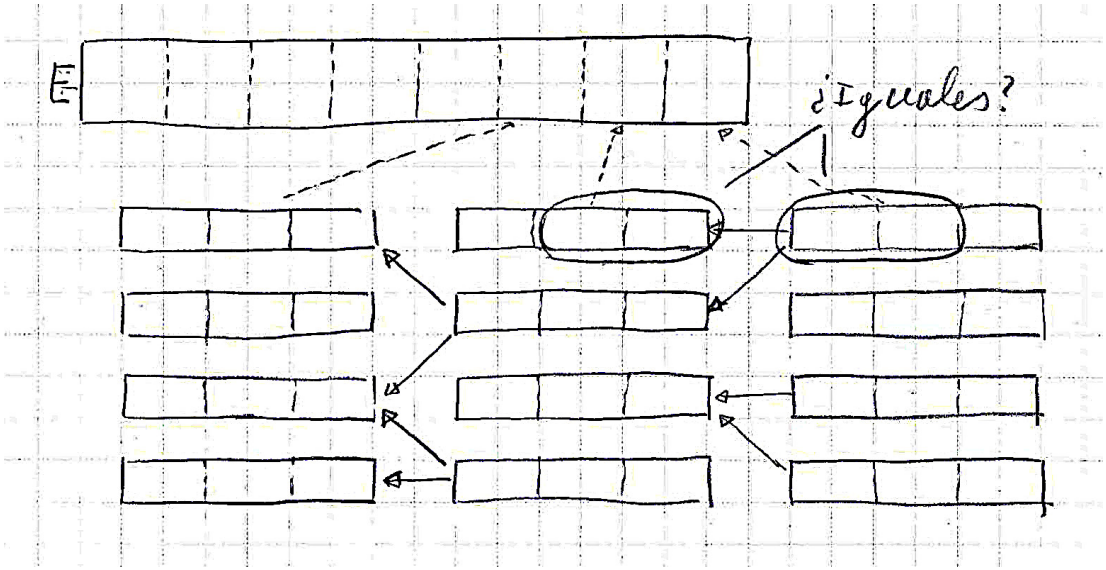
- Mucho trabajo: selección de patrones aplicables en el siguiente paso.



G. El jardín del Edén

Reconstrucción del pasado++

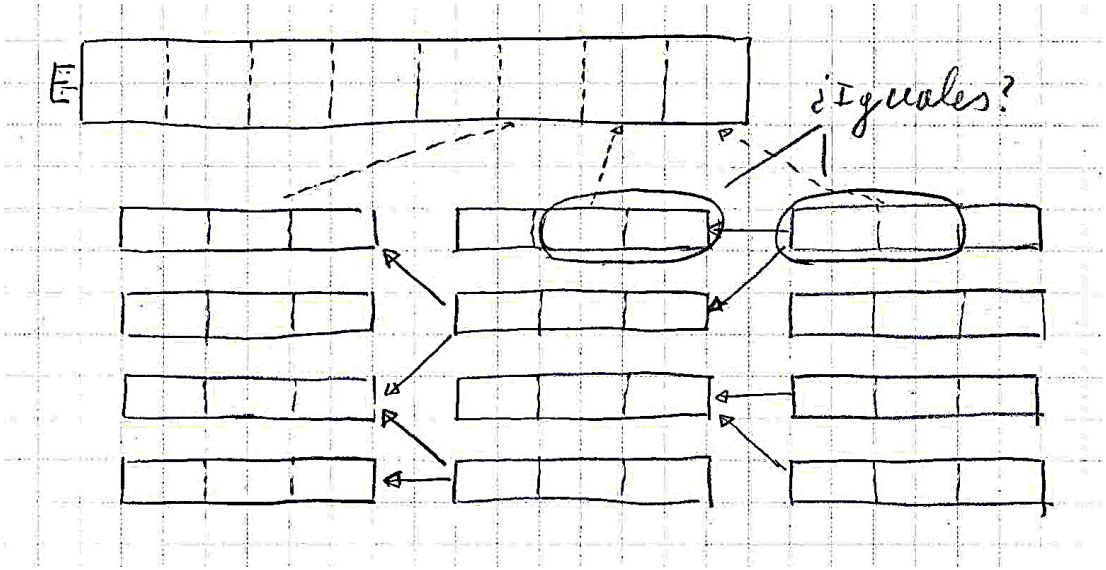
- Mucho trabajo: selección de patrones aplicables en el siguiente paso.



G. El jardín del Edén

Reconstrucción del pasado++

- Mucho trabajo: selección de patrones aplicables en el siguiente paso.

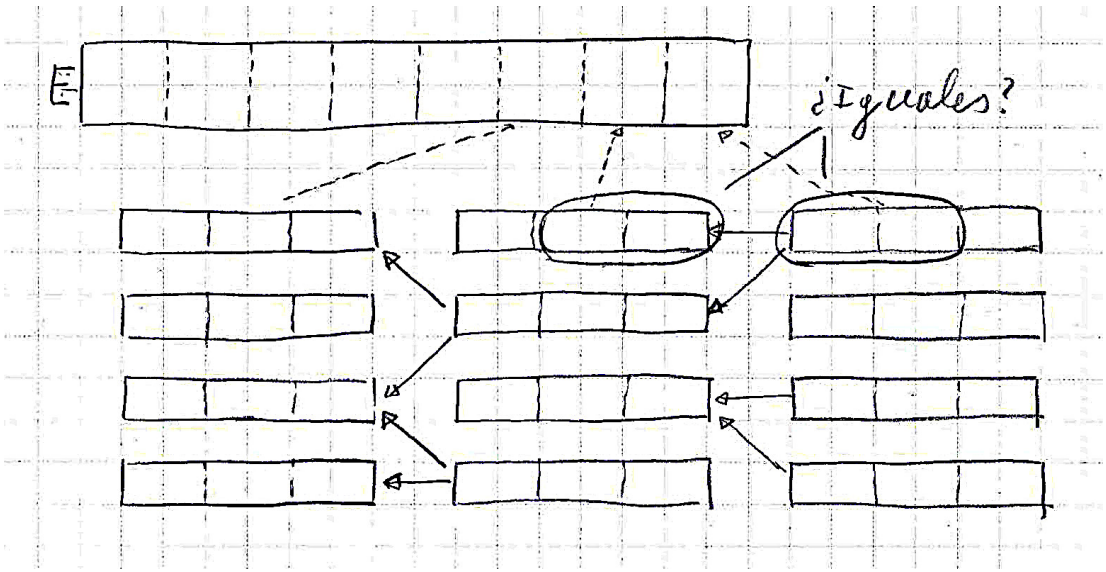


- Búsqueda (dcha. a izqda.): identificar patrón admisible a la izquierda de uno ya fijado.

G. El jardín del Edén

Reconstrucción del pasado++

- Mucho trabajo: selección de patrones aplicables en el siguiente paso.

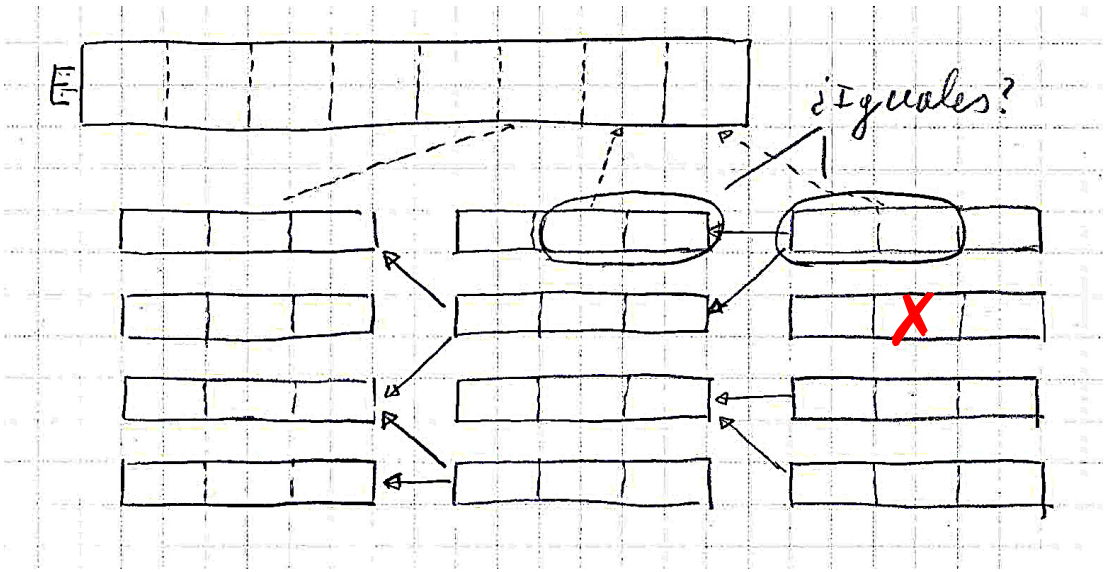


- Búsqueda (dcha. a izqda.): identificar patrón admisible a la izquierda de uno ya fijado.
- Precalcular selección, eliminar patrones "no productivos" (no tienen sucesor o antecesor).

G. El jardín del Edén

Reconstrucción del pasado++

- Mucho trabajo: selección de patrones aplicables en el siguiente paso.

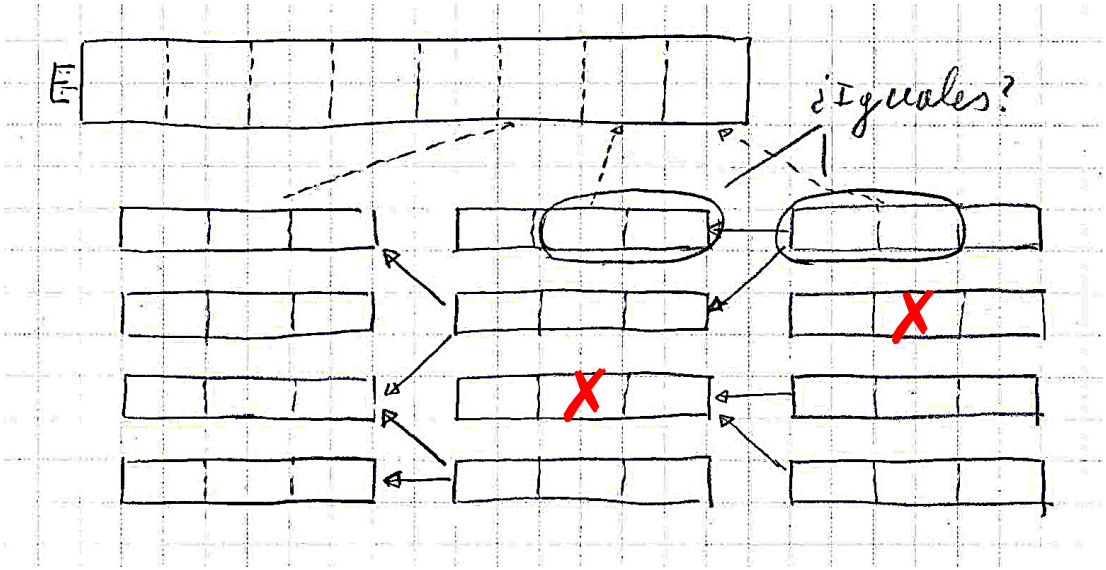


- Búsqueda (dcha. a izqda.): identificar patrón admisible a la izquierda de uno ya fijado.
- Precalcular selección, eliminar patrones "no productivos" (no tienen sucesor o antecesor).

G. El jardín del Edén

Reconstrucción del pasado++

- Mucho trabajo: selección de patrones aplicables en el siguiente paso.

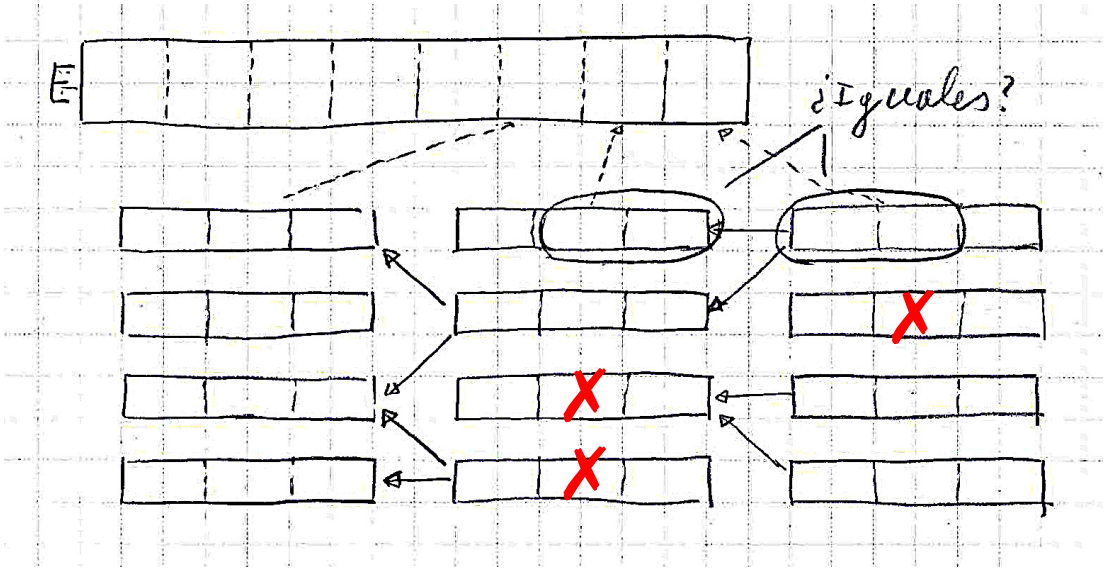


- Búsqueda (dcha. a izqda.): identificar patrón admisible a la izquierda de uno ya fijado.
- Precalcular selección, eliminar patrones "no productivos" (no tienen sucesor o antecesor).

G. El jardín del Edén

Reconstrucción del pasado++

- Mucho trabajo: selección de patrones aplicables en el siguiente paso.

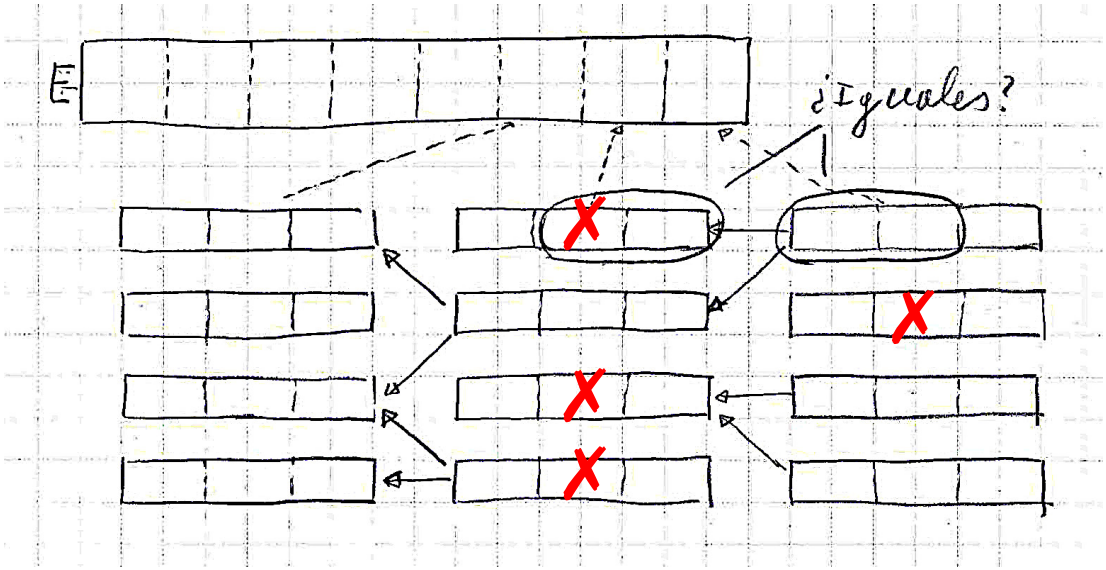


- Búsqueda (dcha. a izqda.): identificar patrón admisible a la izquierda de uno ya fijado.
- Precalcular selección, eliminar patrones "no productivos" (no tienen sucesor o antecesor).

G. El jardín del Edén

Reconstrucción del pasado++

- Mucho trabajo: selección de patrones aplicables en el siguiente paso.

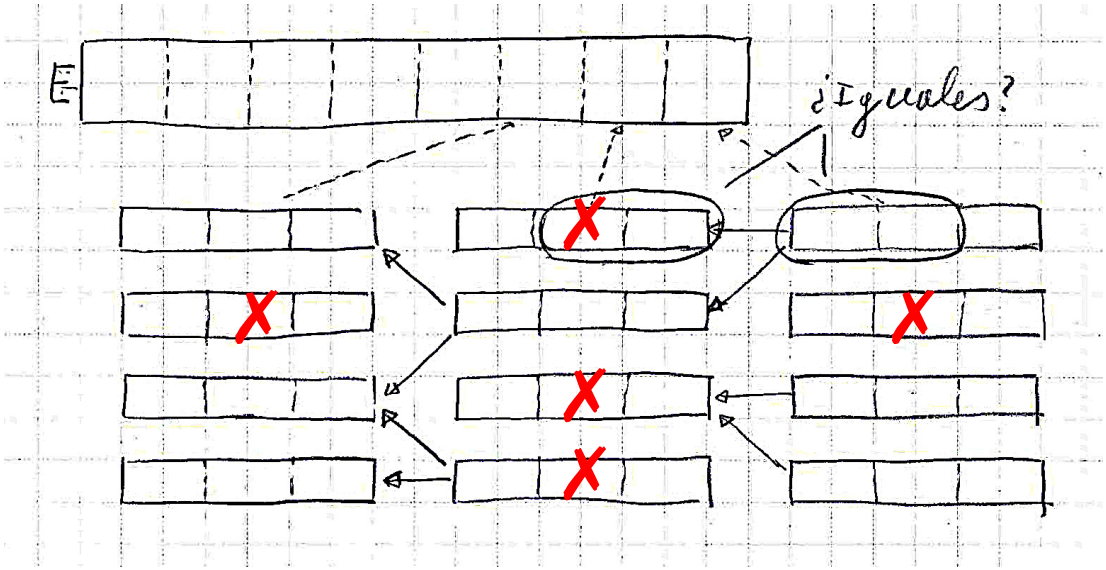


- Búsqueda (dcha. a izqda.): identificar patrón admisible a la izquierda de uno ya fijado.
- Precalcular selección, eliminar patrones "no productivos" (no tienen sucesor o antecesor).

G. El jardín del Edén

Reconstrucción del pasado++

- Mucho trabajo: selección de patrones aplicables en el siguiente paso.

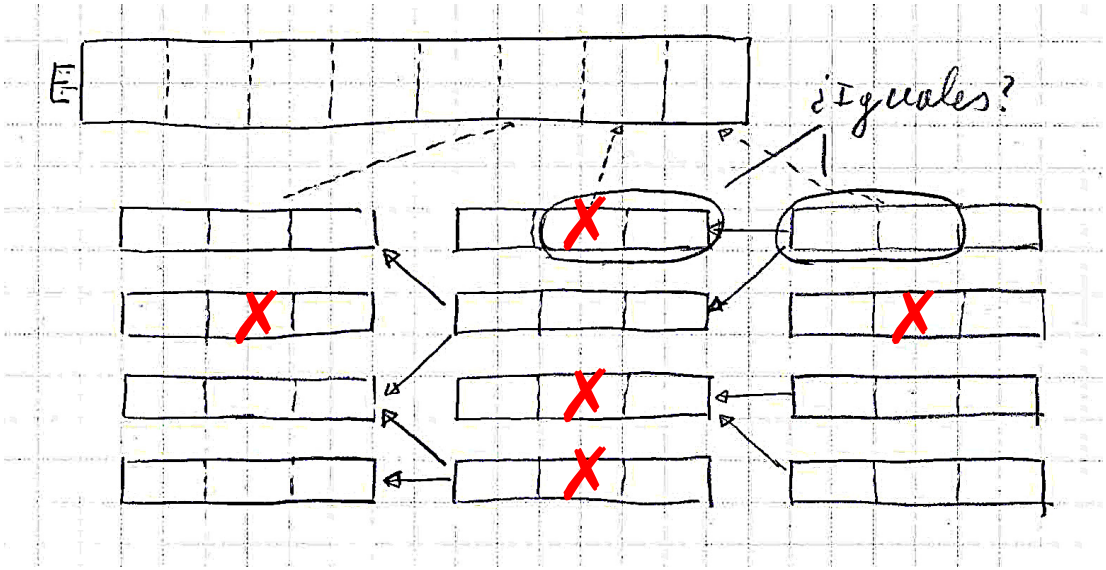


- Búsqueda (dcha. a izqda.): identificar patrón admisible a la izquierda de uno ya fijado.
- Precalcular selección, eliminar patrones "no productivos" (no tienen sucesor o antecesor).

G. El jardín del Edén

Reconstrucción del pasado++

- Mucho trabajo: selección de patrones aplicables en el siguiente paso.

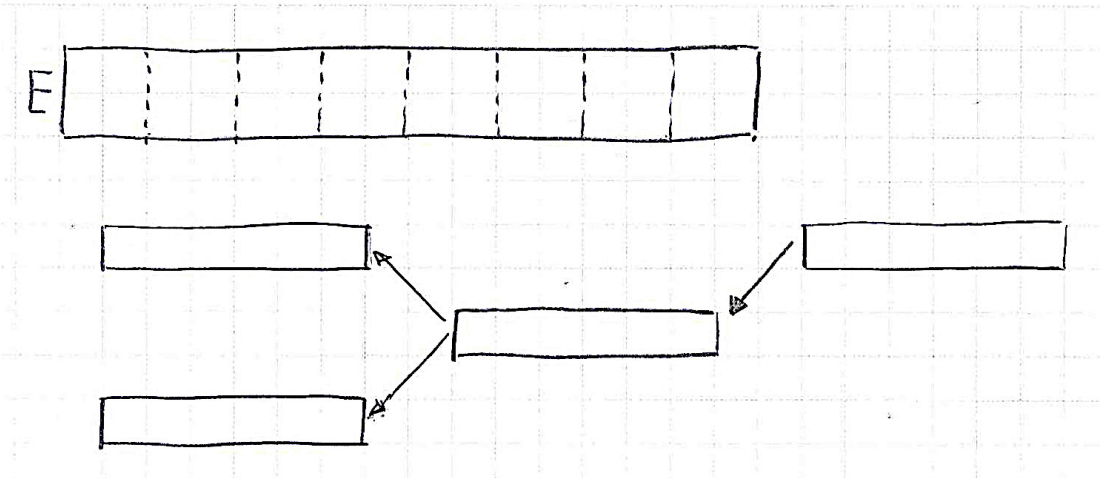


- Búsqueda (dcha. a izqda.): identificar patrón admisible a la izquierda de uno ya fijado.
- Precalcular selección, eliminar patrones "no productivos" (no tienen sucesor o antecesor).
- Grandes potenciales reducciones de candidatos en cada paso.

G. El jardín del Edén

Resultados reducidos

- Reducción de no-determinismo.
- En algunos casos, eliminación completa de alternativas $\longrightarrow$ eliminación de búsqueda.



G. El jardín del Edén

Otro enfoque: programación dinámica / memorización

- Para realizar la búsqueda en cada célula se tiene en cuenta:
  - Estado de la célula (dos posibilidades: viva o muerta).
  - Patrones que pueden dar esa célula (ocho posibles máximo – ocho bits).
  - Filtrados por patrón anterior aplicado (ocho).
- Puede **memorizarse** cada vez que se determina si lleva a un resultado positivo o negativo.
- ¡No es necesario continuar la búsqueda a partir de este momento!