Problemas P y NP

Problemas P y NP

Problema tipo **P**: Aquel que se puede solucionar y verificar con un algoritmo de orden polinómico.

Ejemplos: Todos los problemas vistos en los módulos 1 y 2.

Problema tipo **NP**: Aquel que se puede verificar en tiempo polinomial pero que para solucionarlo requiere de un algoritmo de orden superior.

Ejemplo: SAT (Satisfacibilidad Booleana) ¿Existe alguna asignación booleana para que una expresión lógica sea verdadera?

$$(X \vee \neg Y) \wedge (Y \vee Z)$$

Su complejidad es $O(2^N)$ siendo *N* la cantidad de variables

Problemas NP completos

Un problema *C* es **NP Completo** si se cumplen dos condiciones:

1) *C* es NP.

2) Todo problema NP se puede reducir a *C* en tiempo polinomial. Es decir que una solución para *C* (o parte de ella) se podría usar para resolver los problemas NP.

Los *21 problemas NP-completos de Karp* es un listado de problemas mayormente sobre combinatoria y teoría de grafos, cuya demostración fue elaborada en 1972 por el informático teórico Richard Karp, en su trabajo "*Reducibility Among Combinatorial Problems*" como profundización del trabajo de Stephen Cook, quien en 1971 había demostrado la NP-completitud del problema de Satisfacibilidad Booleana (SAT)

1. **Satisfacibilidad booleana (SAT)**
2. **Satisfacibilidad booleana en forma normal conjuntiva (CNF-SAT)**
3. 3-Satisfacibilidad booleana (3-SAT)
4. 0-1 Programación entera (0-1 Integer Programming)
5. Conjunto independiente máximo en un grafo (Maximum Independent Set)
6. Clique máxima (Clique)
7. **Coloreo de grafos (Graph Coloring)**
8. Conjunto de vértices dominante (Vertex Cover)
9. Conjunto de aristas dominante (Edge Cover)
10. Camino hamiltoniano dirigido (Directed Hamiltonian Circuit)
11. Camino hamiltoniano no dirigido (Undirected Hamiltonian Circuit)
12. Subgrafo Isomorfismo de subgrafos (Subgraph Isomorphism)
13. Camino simple con suma de pesos (Simple Path with Weights ≥ k)
14. Ciclo simple con suma de pesos (Simple Cycle with Weights ≥ k)
15. Circuito de retroalimentación mínimo en grafos dirigidos (Feedback Arc Set)
16. Conjunto de vértices para romper ciclos en grafos no dirigidos (Feedback Vertex Set)
17. Cobertura de conjunto (Set Covering)
18. Empaquetamiento exacto (Exact Cover)
19. División de conjunto (Set Partitioning)
20. **Partición de números (Partition)**
21. **Problema del viajero en su versión de decisión (Traveling Salesman Problem)**

Problemas NP difíciles (hard)

Un problema *C* es **NP Hard** si se cumple que *C* es al menos tan difícil como cualquier problema en NP, pero no necesariamente está en NP. Esto quiere decir que no se requiere poder verificar una solución en tiempo polinomial.

Incluye problemas aún más difíciles que NP, como muchos problemas de optimización, o incluso problemas no computables.

Búsqueda exhaustiva

Búsqueda exhaustiva / por fuerza bruta

Como su nombre lo indica, esta metodología consiste en validar sistemáticamente todos los posibles candidatos para la solución de un problema con el fin de encontrar el que cumple con un criterio dado.

Tiene dos ventajas principales: 1) siempre que exista, encuentra una solución; y 2) generalmente es sencilla de implementar.

Tiene como desventaja que su complejidad suele ser linealmente proporcional al número de soluciones candidatas, el cual suele ser polinomial, exponencial, o factorialmente proporcional al tamaño del problema.

¿Qué algoritmo de los que hemos visto hasta el momento en el curso podría considerarse como búsqueda exhaustiva?

R/. Las soluciones 1 y 2 al problema *Números de casa*

```
read N
for i=2 to N-1:
   sumaIzq = 0
   for j=1 to i-1:
       sumaIzq += j
   sumaDer = 0
   for k=i+1 to N:
       sumaDer += k
   if sumaIzq = sumaDer:
      print i
      exit
print 'NO'
```

```
read N
for i=2 to N-1:
   sumaIzq = i(i-1)/2
   sumaDer = (N(N+1)-i(i+1))/2
   if sumaIzq = sumaDer:
      print i
      exit
print 'NO'
```

Búsqueda exhaustiva

Dado un problema P, el esquema general de la búsqueda exhaustiva se puede expresar como:

```
c = first(P) //Genera el primer candidato
while c ≠ ∅{
   if valid(P, c) then process(P, c)
   c = next(P, c) //Genera el siguiente candidato
}
```

- ***valid(P, c)*** verifica si el candidato cumple con las condiciones del problema.
- En ocasiones la función ***next*** es tal que dicha verificación no es necesaria.
- ***process*(P, c)** dependerá de lo que exija el problema.

Búsqueda exhaustiva

¿Donde podemos ver este esquema en la solución 2 de *Números de casa*?

```
read N
i = 2
while i ≤ N-1:
   process(i)
   i = i + 1
print 'NO'

process(c):
   sumaIzq = i(i-1)/2
   sumaDer = (N(N+1)-i(i+1))/2
   if sumaIzq = sumaDer:
      print i
      exit
```

Problema del agente viajero

Problema del agente viajero (TSP)

Dada una lista de ciudades y las distancias entre cada par de ellas, se debe determinar lo siguiente.

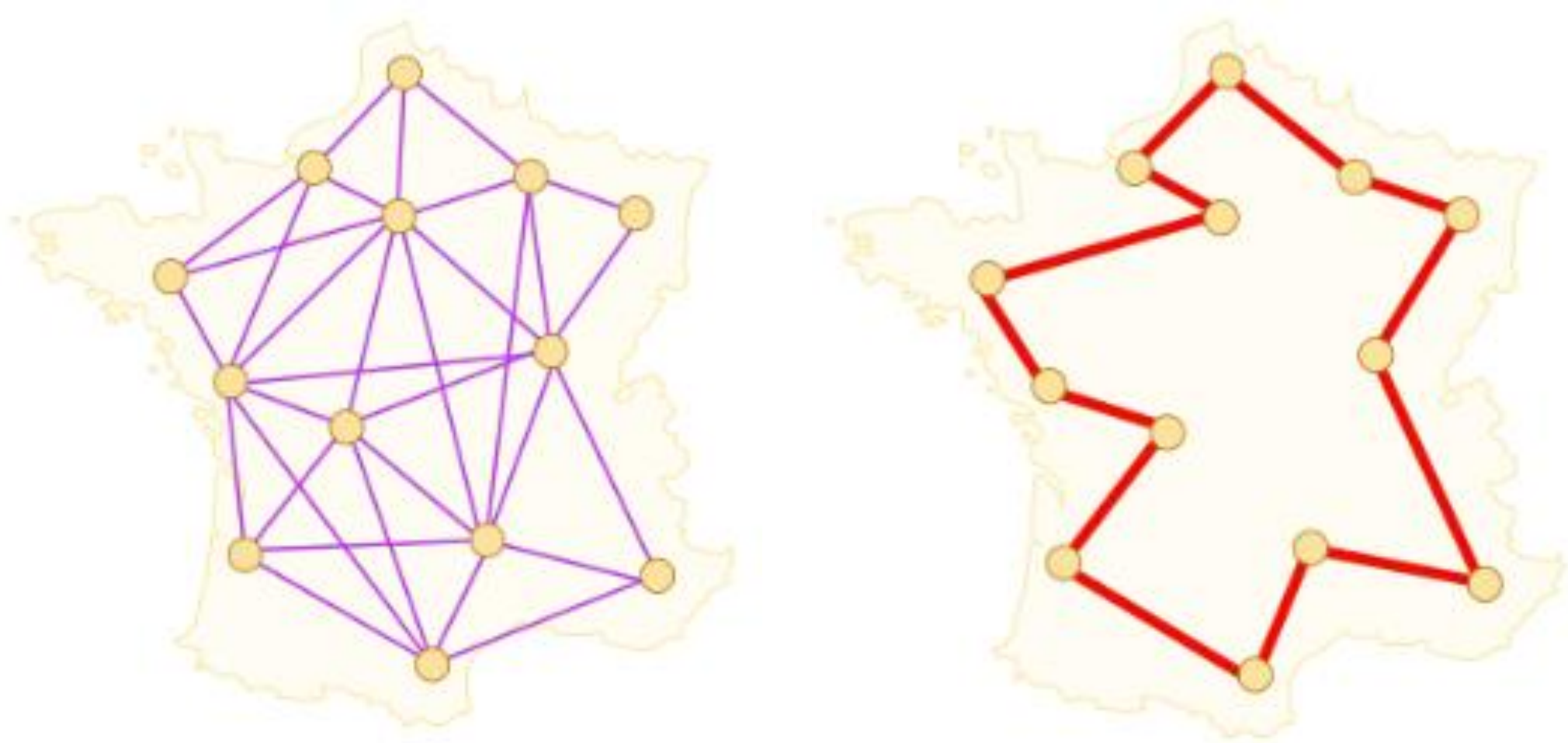
- Versión de decisión: ¿existe por lo menos un recorrido que visite cada ciudad exactamente una vez y cuya longitud total sea menor o igual a k?

NP Completo

- Versión de optimización: ¿cuál es el recorrido más corto posible que visite todas las ciudades una sola vez y regrese al punto de partida?

NP Hard

Versión de optimización



Fuente: https://commons.wikimedia.org/wiki/File:Aco_TSP.svg

¿Cuántas soluciones tiene este problema? $N!$

¿Y diferentes? $\frac{(N-1)!}{2}$

El -1 es porque dada una ruta, da igual el punto de partida. Esto reduce el número de rutas a examinar en un factor *N*. Por ejemplo, si para 3 ciudades la ruta óptima es 1-2-3, también lo es 2-3-1 y 3-1-2

El 1/2 es porque no importa la dirección en que se mueva el agente. Por ejemplo si para 3 ciudades ciudades la ruta óptima es 1-2-3, también lo es 3-2-1

¿Qué significa esto? Pues que por ejemplo para 5 ciudades hay 12 rutas diferentes, para 10 ciudades 181.440, para 20 ciudades ≈ 60 mil billones, etc.

Solución mediante búsqueda exhaustiva

```
min = INF
for i = 0 to N-1:
     c_i = i+1

while nextPermutation(c) ≠ ∅:
     process(c)


process(c):
     sum = distance(c_0, c_N-1)
     for i = 0 to N-2:
          sum += distance(c_i, c_i+1)
     if sum < min:
          min = sum
          best = c
```

¿Cuál es el orden de complejidad de este algoritmo? $O(N!\,N)$

Permutaciones de un arreglo

En el anterior, así como en muchos otros problemas, se requiere encontrar todas las permutaciones de un arreglo de tamaño *N*.

```
permutar(arreglo, inicio, fin):
    if inicio = fin
        procesar(arreglo) #Por ejemplo print(arreglo)
    else:
        for i = inicio to fin + 1
            swap(arreglo_inicio, arreglo_i)
            permutar(arreglo, inicio + 1, fin)
            swap(arreglo_inicio, arreglo_i)

#Llamada inicial: permutar(arreglo, 0, N-1)
```

Permutaciones de un arreglo

Mientras que en Python se puede hacer lo siguiente:

```
from itertools import permutations
array = [1, 2, 3, 4, 5]
p = permutations(array)
for c in p:
   procesar(c) #Por ejemplo print(c)
```

```
(1, 2, 3, 4, 5)
(1, 2, 3, 5, 4)
(1, 2, 4, 3, 5)
...
(5, 4, 3, 2, 1)
```

Cuadrados mágicos

Cuadrados mágicos

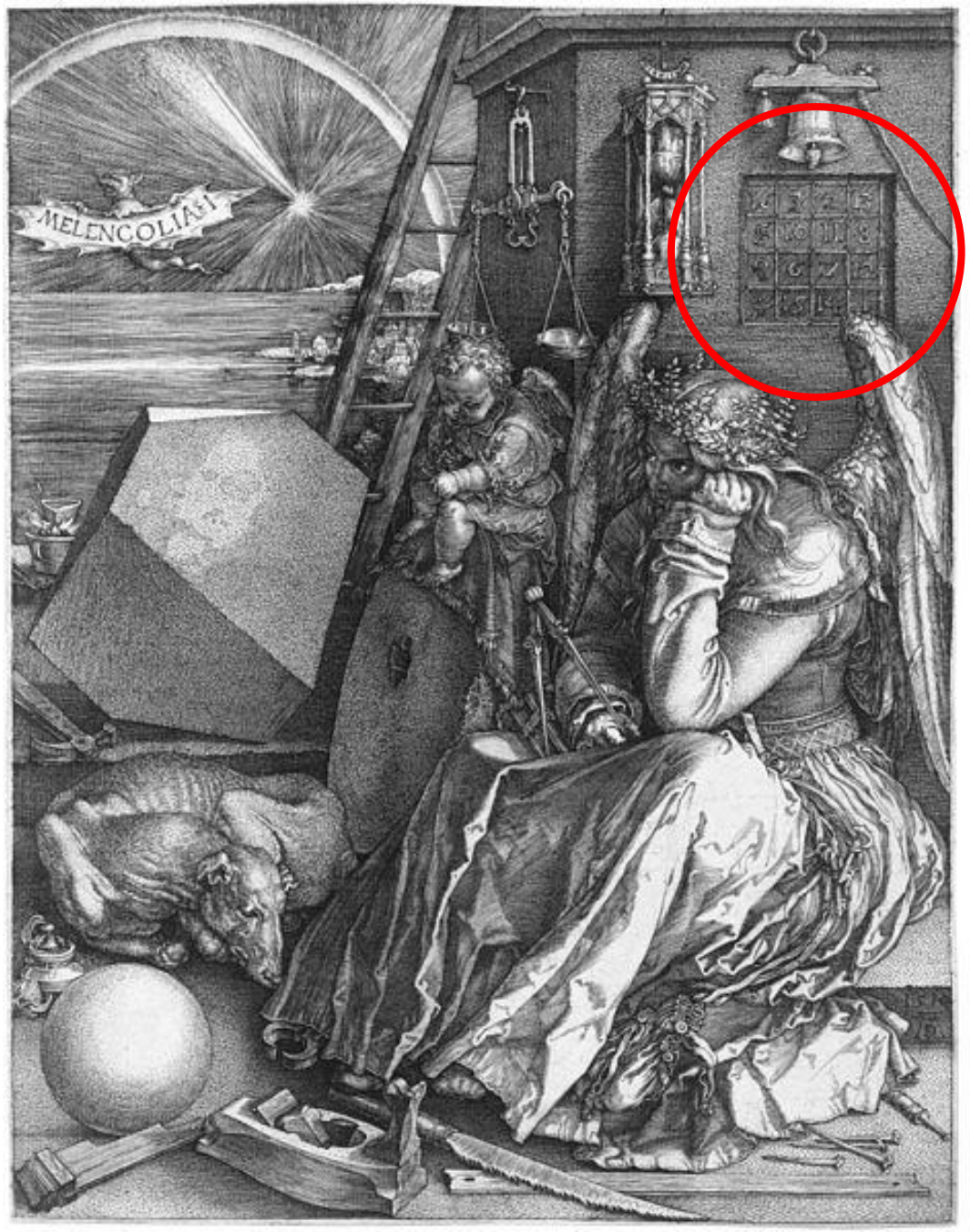
Un cuadrado mágico es una matriz cuadrada de orden $N \geq 3$ en cuyas celdas deben disponerse los números enteros del 1 al N^2 de tal manera que la suma por columnas, filas y diagonales principales de un mismo valor (llamado constante mágica).

Así por ejemplo, para $N = 3$, existe un único cuadrado mágico (sin contar rotaciones ni reflexiones), cuya constante mágica es 15:

4	9	2
3	5	7
8	1	6

Existen cuadrados mágicos famosos como el de Alberto Durero, tallado en su obra Melancolía I. Tiene como particularidades que la constante mágica (34) también puede encontrarse en la suma de las 4 sub-matrices de 2x2 y que las dos celdas centrales de la última fila se leen como 1514, año de ejecución de la obra.

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1



Fuente: https://commons.wikimedia.org/wiki/File:D%C3%BCrer_Melancholia_I.jpg

Cuadrados mágicos

La constante mágica de cada cuadrado está dada por la fórmula: $\frac{N(N^2+1)}{2}$

Y la cantidad de cuadrados mágicos diferentes es:

N	3	4	5	6	7
#	1	880	~ 2.75E6	~1.77E10	$>10^{20}$ (estimado)

Encontrar un cuadrado mágico de orden *N* no es un problema NP pues existen algoritmos cuadráticos (constructivos) para hacerlo. Lo que si puede ser un problema NP (hard) es, por ejemplo, encontrar la cantidad de cuadrados mágicos de orden *N* que cumplan una determinada propiedad.

Cuadrados mágicos

Para resolver este problema, y en general para cualquiera, hay que preguntarse ¿Cuántas posibles soluciones hay?

$N^2!$, que para el caso $N=3$ corresponde a 9! = 362.880

Una vez determinado este asunto la siguiente pregunta es:
¿un algoritmo de solución mediante búsqueda exhaustiva es factible?

Si la respuesta es afirmativa, ¿cómo diseñar entonces ese algoritmo?

Cuadrados mágicos

```
read N
magic = N*(N*N+1)/2
count = 0
for i = 0 to N*N
     c_i = i+1
while nextPermutation(c)
     process(c)

process(c):
     if sumRows(magic) and sumCols(magic) and sumDiags(magic) and ...
          count += 1
```

Cuya complejidad es $O(N^2!\, N^2)$

Partición y Coloreado de grafos

Particion

Partition (o *partitioning*) es un problema NP-completo y consiste en determinar si, dada una lista *X* de *N* valores numéricos, es posible dividirla en dos sublistas que sumen el mismo valor.

Por ejemplo, si *X* es {10, 20, 30, 40}

Una solución es {10, 40}, {20, 30}

¿Cuántas posibles soluciones tiene este problema? $\mathbf{2^N}$

¿Qué significa esto? Pues que por ejemplo para N = 10 hay 1024 soluciones diferentes, para 20 hay 1048576, para 30 ≈ mil millones, etc.

Solución mediante búsqueda exhaustiva

```
read X
med = sum(X)
for i = 0 to N-1:
      c_i = 0
while nextCombination(c)
      process(c)

process(c):
      sum = 0
      for i = 0 to N-1
            sum += c_i * X_i
      if sum = med:
            return c
```

¿Cuál es la eficiencia de este algoritmo? $O(N2^N)$

Combinaciones de un arreglo

En el anterior, así como en muchos otros problemas, se requiere encontrar todas las combinaciones de longitud *N* de un arreglo de base *b*.

```
combinar(base, N, arreglo):
   if |arreglo| == N
      procesar(arreglo) #Por ejemplo print(arreglo)
   else
      for each e in base:
         nuevo_arreglo = copiar(arreglo)
         nuevo_arreglo.add(e)
         combinar(base, N, nuevo_arreglo)

#Llamada inicial:
base = [0, 1, ...]
combinar(base, N, [])
```

Combinaciones de un arreglo

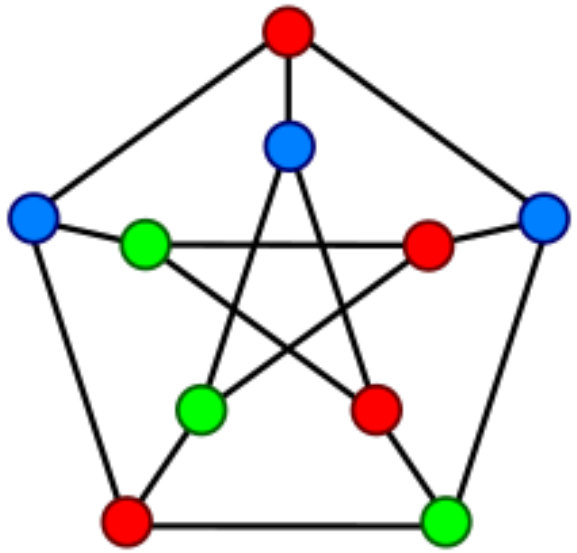
Mientras que en Python se puede hacer lo siguiente:

```
from itertools import product
array = [0, 1] # 0, 1, ... b-1
p = product(array, repeat=4) # repeat = N
for c in p:
   process(c) #Por ejemplo print(c)
```

```
(0, 0, 0, 0)
(0, 0, 0, 1)
(0, 0, 1, 0)
...
(1, 1, 1, 1)
```

Coloreado de grafos

El problema del coloreado de grafos es un problema NP-completo y consiste en determinar, dado un grafo *G* con *N* nodos y *M* aristas, si es posible colorearlo (asignar un color a cada nodo) con *k* colores, o menos, de tal manera que no hayan nodos adyacentes del mismo color.



N = 10
M = 15

Fuente:
ttps://commons.wikimedia.org/wiki/
File:Petersen_graph_3-coloring.svg

Solución mediante búsqueda exhaustiva

Si *G* es totalmente desconectado, la respuesta es 1 (todos los nodos se pueden pintar de un mismo color). En caso contrario:

- Probar todas las combinaciones de 2 colores, si ninguna cumple:
- Probar todas las combinaciones de 3 colores, si ninguna cumple:
- …

- Probar todas las combinaciones de *k* colores

¿Cuál es la eficiencia de este algoritmo? $O\left(N\sum_{i=2}^{k} i^N\right)$

Suma de subconjunto con Ramificación y poda

Suma de subconjunto

El problema de la suma de subconjunto (SSP = *Subset Sum Problem)* es un problema NP-completo y consiste en que, dada una lista *X* de *N* valores enteros positivos y un valor *T*, determinar cuáles subconjuntos suman *T*.

Por ejemplo, si *X* es {80, 30, 50, 40, 10, 20} y *T* es 70

La solución son los subconjuntos {30, 40}, {50, 20}, {40, 10, 20}

¿Cuántas posibles soluciones tiene este problema? $\mathbf{2^N}$

¿Qué significa esto? Pues que por ejemplo para 10 valores hay 1024 soluciones diferentes, para 20 hay 1048576, para 30 ≈ mil millones, etc.

Solución mediante búsqueda exhaustiva

```
slns = 0
for i = 0 to N-1:
     c_i = 0
do:
     if true:
          process(c)
while nextCombination(c)

process(c):
     sum = 0
     for i = 0 to N-1:
          sum += c_i * X_i
     if sum = T:
          slns += 1
```

¿Cuál es la eficiencia de este algoritmo? $O(N2^N)$

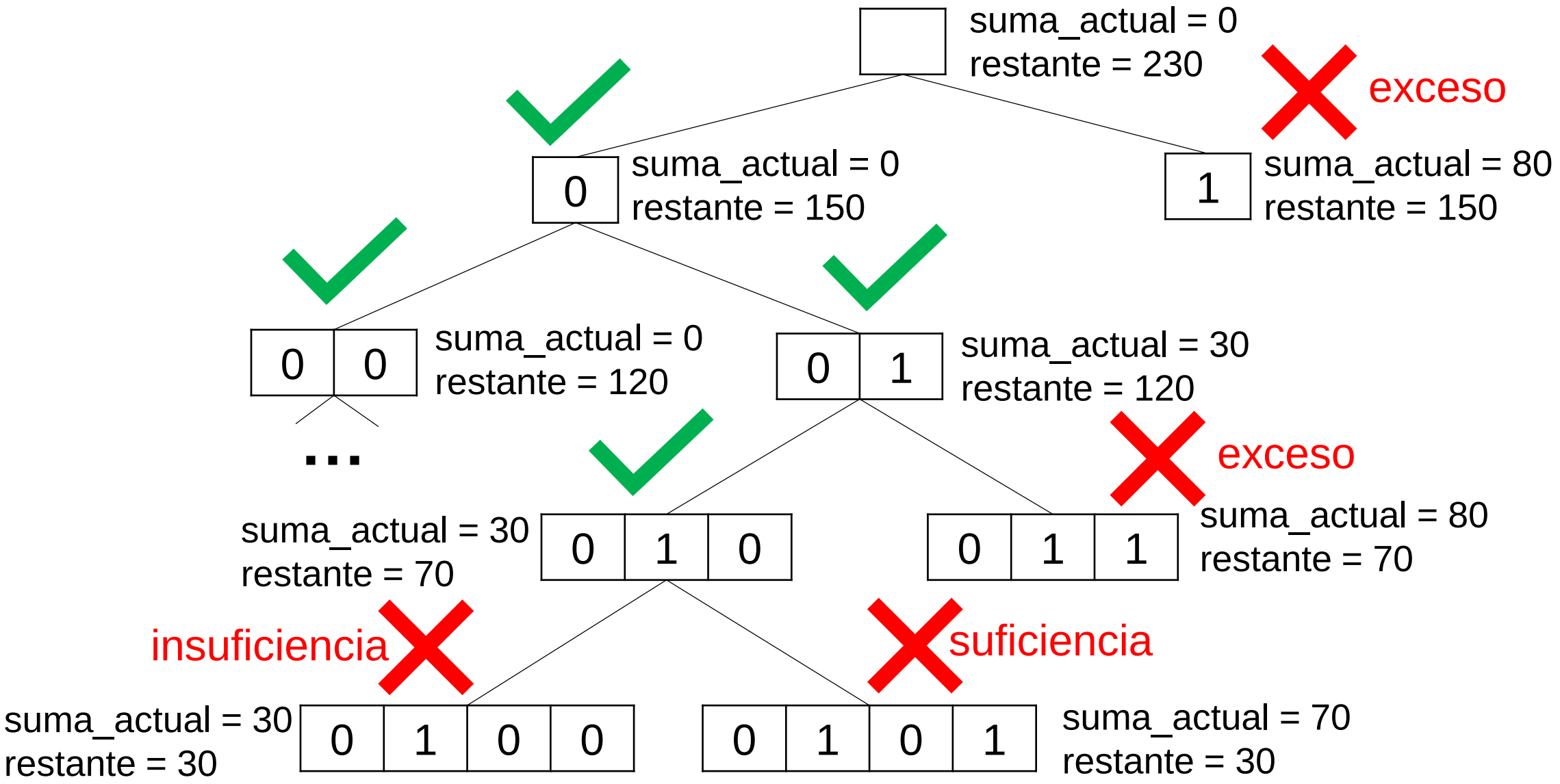
Ramificación y poda

Esta técnica complementaria (del inglés *Branch & Bound*) se emplea principalmente en problemas de búsqueda y optimización. Se basa en analizar el problema a partir de un árbol de posibles soluciones, detectando en qué ramificación ya se alcanzó la solución, o los candidatos pendientes no pueden llegar a ella, y en estos casos podarla de manera que no se malgasten recursos.

```
combinar(base, N, arreglo):
    M = |arreglo|
    if M = N and arreglo*X = T
        print arreglo
    else:
        for each e in base:
            nuevo_arreglo = copiar(arreglo), nuevo_arreglo.add(e)
            suma_actual = nuevo_arreglo*X, restante = sum(X[M:])
            if suma_actual = T: #poda por suficiencia
            else if suma_actual > T: #poda por exceso
            else if suma_actual + restante < T: #poda por insuficiencia
            else:
                combinar(base, N, nuevo_arreglo)

read X, T #[80, 30, 50, 40, 10, 20], 70
base = [0, 1]
N = |X|
combinar(base, N, [])
```

X es {80, 30, 50, 40, 10, 20} y *T* es 70



Reflexión final

¿Cuándo debemos usar búsqueda exhaustiva (con o sin ramificación y poda)?

Cuando sea estrictamente necesario evaluar todos los posibles candidatos en un problema

y/o

Cuando la cantidad de dichos candidatos sea pequeña