

# Secuencias de Langford

Práctica 2 (2 sesiones)

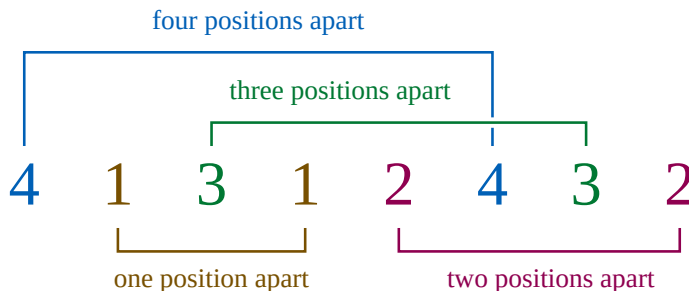
Prácticas de ALT

Algorítmica  
ETSInf

Curso 2019-2020

# Descripción del problema

Una secuencia o **emparejamiento de Langford** de longitud  $2N$  es una ordenación de los números  $1, 1, 2, 2, \dots, N-1, N-1, N, N$  de manera que entre dos números  $i$  hay  $i$  números como en el siguiente ejemplo de [wikipedia](#).



El objetivo de la práctica es realizar un programa que, utilizando la estrategia de **backtracking** (o búsqueda con retroceso), encuentre una o más secuencias de Langford para una talla  $N$ , y que lo haga de dos maneras diferentes:

1. Generando una ordenación de los números  $1, 1, 2, 2, \dots, N-1, N-1, N, N$ .
2. Reduciendo este problema al **cubrimiento exacto** (**exact cover**) y usando un algoritmo (que se os proporciona ya implementado) que resuelve este problema.

# Resolución directa mediante **backtracking**

Consiste en utilizar un vector (en forma de una lista Python) de longitud  $2N$  que estará inicializada a 0s (valor que denota un hueco en el vector).

El algoritmo de backtracking intentará posicionar, una por una, las  $N$  parejas de números empezando por la más alta (la pareja  $N,N$ ) hasta la más baja (la pareja  $1,1$ ).

Ramificar consiste en probar todas las posibles posiciones en que podemos situar cada pareja en el vector. Por ejemplo, si vamos a situar el valor 3 debemos tener en cuenta que si el primero ocupa la posición  $i$  en el vector (con  $i \geq 0$ ), el segundo ocupará la posición  $i + 3 + 1$  ( $i + 3 + 1 \leq 2 * N - 1$ ). Además:

- no podemos poner los valores encima de otros (únicamente podremos situarlos donde previamente tengamos 0s), y
- cuando cambiemos de rama debemos *\*deshacer\** adecuadamente los cambios efectuados poniendo a 0 las posiciones ocupadas previamente.

## ¡Ojo!

Se puede demostrar que solamente existen soluciones para aquellos valores  $N$  tal que el resto de dividir  $N$  entre 4 valga 0 o 3. Por tanto, en esos casos el programa dirá directamente que no hay solución sin intentar backtracking:

```
if N%4 not in (0, 3):  
    yield "no hay solucion"
```

# Resolución directa mediante **backtracking**

Un posible esqueleto del programa:

```
def langford_directo(N, allsolutions):
    N2    = 2*N
    seq   = [0]*N2

    def backtracking(num):
        if num <= 0:
            yield "-".join(map(str, seq))
        else:
            # COMPLETAR

    if N%4 not in (0, 3):
        yield "no hay solucion"
    else:
        count = 0
        for s in backtracking(N):
            count += 1
            yield "solution %04d -> %s" % (count, s)
            if not allsolutions:
                break
```

(continúa)

# Resolución directa mediante **backtracking**

Un posible esqueleto del programa (continuación):

```
if __name__ == "__main__":
    if len(sys.argv) not in (2, 3, 4):
        print('\nUsage: %s N [TODAS] [EXACT_COVER] \n' % (sys.argv[0],))
        sys.exit()
    try:
        N = int(sys.argv[1])
    except ValueError:
        print('First argument must be an integer')
        sys.exit()
    allSolutions = False
    langford_function = langford_directo
    for param in sys.argv[2:]:
        if param == 'TODAS':
            allSolutions = True
        elif param == 'EXACT_COVER':
            langford_function = langford_exact_cover
    for sol in langford_function(N, allSolutions):
        print(sol)
```

# Cómo utilizar *yield* en backtracking

```
import sys

def nqueens(n):
    sol = [None]*n
    def show_solution(solution):
        output = [" "+"".join(str((i+1) % 10) for i in range(n))+"\n"]
        for i in range(n):
            output.append("%3d %s\n" % (i+1,"".join("X" if solution[j]==i else "." for j in range(n))))
        return "".join(output)
    def is_promising(longSol, queen):
        return all(queen != sol[i] and longSol-i != abs(queen-sol[i]) for i in range(longSol))
    def backtracking(longSol):
        if longSol == n:
            return show_solution(sol)
        else:
            for queen in range(n):
                if is_promising(longSol, queen):
                    sol[longSol] = queen
                    r = backtracking(longSol+1)
                    if r is not None:
                        return r
            return None # explicit
    return backtracking(0)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print('\nUsage: %s N\n' % (sys.argv[0],))
        sys.exit()
    N = int(sys.argv[1])
    print(nqueens(N))
```

# Cómo utilizar *yield* en backtracking

```
import sys

def nqueens(n, allSolutions):
    sol = [None]*n
    def show_solution(solution):
        output = [" " * i + "X" * (n-i) for i in range(n)]
        for i in range(n):
            output.append("%3d %s\n" % (i+1, " ".join("X" if solution[j]==i else "." for j in range(n))))
        return "\n".join(output)
    def is_promising(longSol, queen):
        return all(queen != sol[i] and longSol-i != abs(queen-sol[i]) for i in range(longSol))
    def backtracking(longSol):
        if longSol == n:
            results.append(show_solution(sol))
        if allSolutions or len(results) == 0:
            for queen in range(n):
                if is_promising(longSol, queen):
                    sol[longSol] = queen
                    backtracking(longSol+1)
        return None # explicit
    results = []
    backtracking(0)
    return results

if __name__ == "__main__":
    if len(sys.argv) not in (2,3):
        print('\nUsage: %s N [TODAS]\n' % (sys.argv[0],))
        sys.exit()
    N = int(sys.argv[1])
    allSolutions = len(sys.argv)==3 and sys.argv[2]=='TODAS'
    for sol in nqueens(N,allSolutions):
        print(sol)
```

# Cómo utilizar *yield* en backtracking

```
import sys

def nqueens(n, allSolutions):
    sol = [None]*n
    def show_solution(solution):
        output = [" " + " ".join(str((i+1) % 10) for i in range(n)) + "\n"]
        for i in range(n):
            output.append("%3d %s\n" % (i+1, " ".join("X" if solution[j]==i else "." for j in range(n))))
        return " ".join(output)
    def is_promising(longSol, queen):
        return all(queen != sol[i] and longSol-i != abs(queen-sol[i]) for i in range(longSol))
    def backtracking(longSol):
        if longSol == n:
            yield show_solution(sol)
        else:
            for queen in range(n):
                if is_promising(longSol, queen):
                    sol[longSol] = queen
                    for s in backtracking(longSol+1):
                        yield s
    count = 0
    for s in backtracking(0):
        count += 1
        yield "Solution %d:\n\n%s" % (count,s)
    if not allSolutions:
        break

if __name__ == "__main__":
    if len(sys.argv) not in (2,3):
        print('\nUsage: %s N [TODAS]\n' % (sys.argv[0],))
        sys.exit()
    N = int(sys.argv[1])
    allSolutions = len(sys.argv)==3 and sys.argv[2]=='TODAS'
    for sol in nqueens(N,allSolutions):
        print(sol)
```



# El problema del cubrimiento exacto

En el problema del cubrimiento exacto (**exact cover**) nos dan un conjunto  $X$  y luego un conjunto  $S$  de subconjuntos de  $X$ .

**Ejemplo:**  $X$  podría ser  $\{A, B, C, D\}$  y  $S = \{\{A\}, \{A, B, C\}, \{A, C\}, \{A, D\}, \{B, D\}\}$

El objetivo es encontrar  $S'$ , subconjunto de  $S$  (i.e.  $S' \subseteq S$ ), tal que sus elementos formen una partición de  $X$ . Es decir, de manera que todos los elementos de  $X$  estén en uno y sólo en uno de los elementos de  $S'$ .

**Ejemplo:** Para el ejemplo anterior una posible solución sería  $S' = \{\{A, C\}, \{B, D\}\}$

Existe una forma sencilla de resolver el problema del cubrimiento exacto mediante **backtracking** y una implementación del mismo, propuesta por Donald Knuth, se denomina el **algoritmo X**.

La idea básica de este algoritmo es elegir un elemento  $x \in X$  y luego ramificar en **backtracking** eligiendo uno de los elementos  $s \in S$  (recordemos que  $s \subseteq X$ ) que contengan a  $x$ . Una vez elegido  $s$ , eliminamos de  $S$  aquellos elementos  $s'$  tales que  $s \cap s' \neq \emptyset$ . Si esta rama no prospera o si nos piden más de una solución, antes de probar con otro hermano de  $s'$  deberíamos deshacer los cambios realizados (ej: volver a incorporar los  $s'$  descartados).

El **algoritmo X** no se limita a recorrer los elementos de  $X$  en un orden predeterminado sino que, en cada momento, elige aquel valor  $x \in X$  que aparece en menos subconjuntos  $s \in S$ , de manera que:

- si un elemento no aparece en ningún subconjunto nos daremos cuenta rápidamente que no hay solución (y haremos **backtracking**),
- cuando un elemento  $x$  aparece únicamente en un subconjunto  $s \in S$ , eligiremos ese conjunto  $s$  puesto que **forzosamente** ha de formar parte de la solución (en este contexto de la búsqueda).

La implementación del **algoritmo X** de Knuth hace uso de una técnica llamada **Dancing Links** donde  $S$  se representa mediante una lista doblemente enlazada. Esta técnica hace muy eficiente eliminar conjuntos de  $S$  y luego deshacer dicha operación cuando sea necesario.

# El problema del cubrimiento exacto

En esta práctica vamos a utilizar una implementación del [algoritmo X](#) que ocupa menos de 30 líneas de Python y que se ha obtenido del [siguiente enlace \(pinchar aquí\)](#).

**Ejemplo:** Esta implementación asume que pasaremos el conjunto  $X$  mediante un **set** python, aunque el conjunto  $S$  (denominado  $Y$  en esta implementación) se proporciona como un diccionario donde la clave se utiliza para dar un nombre único a cada uno de los subconjuntos como en el siguiente ejemplo:

```
X = {1, 2, 3, 4, 5, 6, 7}
```

```
Y = {  
    'A': [1, 4, 7],  
    'B': [1, 4],  
    'C': [4, 5, 7],  
    'D': [3, 5, 6],  
    'E': [2, 3, 6, 7],  
    'F': [2, 7]  
}
```

La única solución para este ejemplo sería ['B', 'D', 'F'].

# El problema del cubrimiento exacto

La implementación propuesta, en lugar de basarse en listas doblemente enlazadas representa  $X$  como un diccionario que asocia a cada elemento del conjunto una lista de los subconjuntos que lo contienen. Para el ejemplo anterior sería así:

```
X = {  
    1: {'A', 'B'},  
    2: {'E', 'F'},  
    3: {'D', 'E'},  
    4: {'A', 'B', 'C'},  
    5: {'C', 'D'},  
    6: {'D', 'E'},  
    7: {'A', 'C', 'E', 'F'}  
}
```

Una forma de obtener esta estructura a partir del conjunto  $X$  original sería como sigue:

```
X = {j: set() for j in X}  
for i in Y:  
    for j in Y[i]:  
        X[j].add(i)
```

# Reduciendo el problema de la secuencia de Langford al cubrimiento exacto

Para reducir el problema de la secuencia de Langford de longitud  $N$  al **exact cover** necesitamos en principio, un conjunto  $X$  de longitud  $2N$  correspondiente a las posiciones que podemos ocupar en la secuencia. Así, por ejemplo, si el vector se indexa entre  $0$  y  $2 * N - 1$ , podríamos utilizar las cadenas 'p0', 'p1', ... como elementos del conjunto  $X$  (para mayor eficiencia Python permite **internalizar** las cadenas con la función **intern** de la biblioteca **sys**).

Para cada posible número entre  $1$  y  $N$  necesitamos crear un subconjunto por cada posible posición en que podemos meter una pareja de estos valores.

**Por ejemplo**, para el valor  $1$  deberíamos considerar los siguientes subconjuntos de  $S$ : {'p0', 'p2'}, {'p1', 'p3'}, {'p2', 'p4'}, ...

## ¿Cuál es el fallo de este razonamiento?

El problema de esta aproximación es que nada impide obtener una solución que utilice más de una vez un mismo número y que queden números por utilizar. Así, por ejemplo, para  $N=4$  en lugar de una secuencia como esta: [4, 1, 3, 1, 2, 4, 3, 2] que se obtendría con el cubrimiento exacto [{'p0', 'p5'}, {'p1', 'p3'}, {'p2', 'p6'}, {'p4', 'p7'}], también se podría obtener otro cubrimiento exacto de 'p0', 'p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7' utilizando [{'p0', 'p2'}, {'p1', 'p3'}, {'p4', 'p6'}, {'p5', 'p7'}] dando la secuencia: [1, 1, 1, 1, 1, 1, 1, 1].

# Reduciendo el problema de la secuencia de Langford al cubrimiento exacto

Para resolver el fallo anterior se propone incorporar en el conjunto  $X$ , además de elementos representando el hecho de ocupar cada una de las  $2 * N$  posiciones del vector, otros  $N$  valores más que representen el hecho de usar cada uno de los  $N$  números a utilizar.

Debes crear una función que recibe el valor  $N$  y que genere **programáticamente** las estructuras de datos  $X$  e  $Y$  que espera el **exact cover**, tendría un aspecto como sigue:

```
def langford_data_structure(N):
    # n1,n2,... means that the value has been used
    # p1,p2,... means that the position has been used
    def value(i):
        return sys.intern('n%d' % (i,))
    def position(i):
        return sys.intern('p%d' % (i,))

    X = set([value(i) for i in range(1,N+1)] +
            [position(i) for i in range(2*N)])
    Y = {}
    for v in range(1,N+1):
        # COMPLETAR
    X = {j: set() for j in X}
    for i in Y:
        for j in Y[i]:
            X[j].add(i)

    return X,Y
```

# Reduciendo el problema de la secuencia de Langford al cubrimiento exacto

Como puedes observar, el conjunto  $X$  contiene las cadenas ' $n1$ ', ' $n2$ ', ... hasta  $N$ , representando el hecho de usar esos  $N$  valores para llenar el vector, así como las cadenas ' $p0$ ', ' $p1$ ', ... hasta  $2N - 1$ , cada una correspondiente a un índice o posición en el vector.

La parte que debes completar consiste en llenar el diccionario  $Y$  con pares *clave:valor* donde los valores son listas que contienen tripletas. Cada tripleta corresponde al hecho de meter una pareja de números de manera correcta en el vector. Por ejemplo, la tripleta [' $n1$ ', ' $p2$ ', ' $p4$ '] es correcta, ya que si pones un 1 en la posición 2 el otro 1 debe ir en la 4 (a distancia 1 separados por la posición 3). Esta tripleta la crearíamos usando las funciones **value** y **position** así:

```
... = [ value(n), position(p), position(p+(n+1)) ]
```

Por otra parte, las claves o nombres de estas entradas las utilizará la implementación del **exact cover** para devolvernos la solución. Vamos a utilizar estas claves, posteriormente, para extraer la secuencia de valores en el vector. Para ello usaremos el siguiente convenio: Si el valor era [' $n1$ ', ' $p2$ ', ' $p4$ '], la clave correspondiente será la cadena ' $n1p2$ '. El código siguiente muestra el código utilizado para extraer los valores de  $n$  y de  $p$  a partir de esa clave:

```
n, p = map(int, item[1:].split('p'))
```

# Reduciendo el problema de la secuencia de Langford al cubrimiento exacto

Para terminar, necesitamos una manera de convertir la solución devuelta por el **exact cover** en formato de secuencia:

```
def langford_exact_cover(N, allsolutions):
    if N%4 not in (0,3):
        yield "no hay solucion"
    else:
        X, Y = langford_data_structure(N)
        sol = [None]*2*N
        count = 0
        for coversol in solve(X,Y):
            for item in coversol:
                n, p= map(int, item[1:].split('p'))
                sol[p]=n
                sol[p+n+1]=n
            count += 1
        yield "solution %04d -> %s" % (count, "-".join(map(str, sol)))
        if not allsolutions:
            break
```



# ¿Qué entregar?

Un programa que reciban como parámetro (por la línea de comandos):

- el número  $N$ ,
- si el usuario incluye la cadena *EXACT\_COVER*, el programa utilizará la versión basada en cubrimiento exacto,
- si el usuario incluye el parámetro *TODAS* el programa mostrará todas las soluciones (en otro caso, parará tras encontrar una primera solución).

## Ejemplos

```
> python langford.py 6
no hay solucion
> python langford.py 4
solution 0001 -> 4-1-3-1-2-4-3-2
> python langford.py 4 EXACT_COVER TODAS
solution 0001 -> 4-1-3-1-2-4-3-2
solution 0002 -> 2-3-4-2-1-3-1-4
> python langford.py 7 TODAS
solution 0001 -> 7-3-6-2-5-3-2-4-7-6-5-1-4-1
solution 0002 -> 7-2-6-3-2-4-5-3-7-6-4-1-5-1
...
solution 0051 -> 1-5-1-4-6-7-3-5-4-2-3-6-2-7
solution 0052 -> 1-4-1-5-6-7-4-2-3-5-2-6-3-7
```