Práctica 1: Técnicas Algorítmicas

Compilado: 21 de marzo de 2022

Backtracking

1. En este ejercicio vamos a resolver el problema de suma de subconjuntos, visto en la teórica, con la técnica de *backtracking*. A diferencia de la clase teórica, vamos a usar la representación binaria. Dado un multiconjunto $C = \{c_1, \dots, c_n\}$ de números naturales y un natural k, queremos determinar si existe un subconjunto de C cuya sumatoria sea k. Notar que, a diferencia de la teórica, no necesitamos suponer que los elementos de C son todos distintos. Sí vamos a utilizar fuertemente que C está ordenado de alguna forma arbitraria pero conocida (i.e., C está implementado como la secuencia $c_1, \dots, c_n$ o, análogamente, tenemos un iterador de C). Como se discute en la teórica, las *soluciones (candidatas)* son los vectores $a = (a_1, \dots, a_n)$ de valores binarios; el subconjunto de C representado por a contiene a c_i si y sólo si $a_i = 1$. Luego, a es una solución *válida* cuando $\sum_{i=1}^{n} a_i c_i = k$. Asimismo, una *solución parcial* es un vector $p = (a_1, \dots, a_i)$ de números binarios con $0 \le i \le n$. Si $i < n$, las soluciones *sucesoras* de p son $p \oplus 0$ y $p \oplus 1$, donde $\oplus$ indica la concatenación.

a) Escribir el conjunto de soluciones candidatas para $C = \{6, 12, 6\}$ y $k = 12$.

Res = {(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1)}

b) Escribir el conjunto de soluciones válidas para $C = \{6, 12, 6\}$ y $k = 12$.

$a = (1,0,1) \rightarrow k = \sum_{i=1}^{n} a_i \cdot c_i = 1.6 + 0.12 + 1.6 = 12$ ✓

$a = (0,1,0) \rightarrow k = \sum_{i=1}^{n} a_i \cdot c_i = 0.6 + 1.12 + 0.6 = 12$ ✓

Res = {(1,0,1), (0,1,0)}

c) Escribir el conjunto de soluciones parciales para $C = \{6, 12, 6\}$ y $k = 12$.

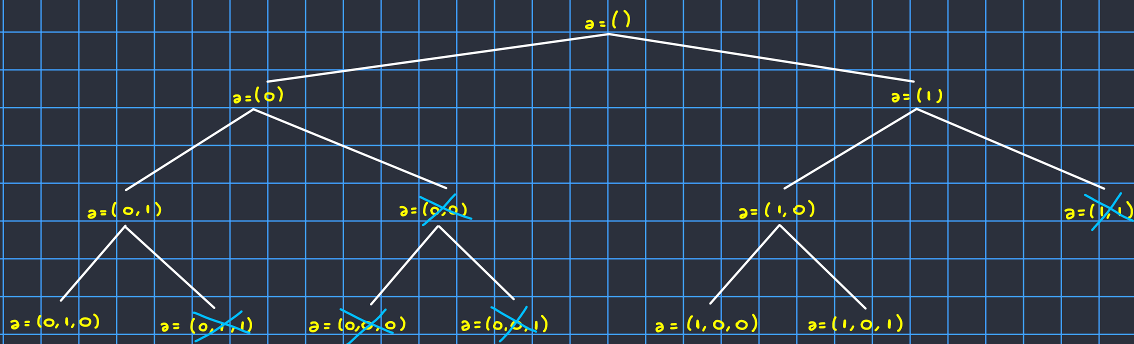
Res = {(1), (0), (0,0), (0,1), (1,0), (1,1), (0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1)}

Duda: ¿se cortan en cualquier momento o cuando notan que está mal?

¿Qué algoritmo? Lmao

d) Dibujar el árbol de *backtracking* correspondiente al algoritmo descrito arriba para $C = \{6, 12, 6\}$ y $k = 12$, indicando claramente la relación entre las distintas componentes del árbol y los conjuntos de los incisos anteriores.

```
vector<vector<int>> SumaDeConjuntosBT (vector<int> C, int k){
    vector<int> p = {};
    vector<vector<int>> res = {{}};
    SumaDeConjuntosBT_aux(C,k,res,p);
    return res;
}

void SumaDeConjuntosBT_aux (vector<int> C, int k, vector & a, vector p){
    int n = C.size();

    if (p.size() == n){

        if (p es solución válida) {
            a.push_back(p);
        }

    } else {

        vector<int> p0 = p.push_back(0);
        vector<int> p1 = p.push_back(1);

        if (p0 es una solución posible) SumaDeConjuntosBT_aux(C,k,a,p0);
        if (p1 es una solución posible) SumaDeConjuntosBT_aux(C,k,a,p1);
    }
}
```



e) Sea $\mathcal{C}$ la familia de todos los multiconjuntos de números naturales. Considerar la siguiente función recursiva ss: $\mathcal{C} \times \mathbb{N} \to \{V, F\}$ (donde $\mathbb{N} = \{0, 1, 2, \dots\}$, V indica verdadero y F falso):

$$\operatorname{ss}(\{c_1, \dots, c_n\}, k) = \begin{cases} k \overset{==}{=} 0 & \text{si } n = 0 \\ [\operatorname{ss}(\{c_1, \dots, c_{n-1}\}, k)] \vee [\operatorname{ss}(\{c_1, \dots, c_{n-1}\}, k - c_n)] & \text{si } n > 0 \end{cases}$$

Convencerse de que $\operatorname{ss}(C, k) = V$ si y sólo si el problema de subconjuntos tiene una solución válida para la entrada C, k. Para ello, observar que hay dos posibilidades para una solución válida $a = (a_1, \dots, a_n)$ para el caso $n > 0$: o bien $a_n = 0$ o bien $a_n = 1$. En el primer caso, existe un subconjunto de $\{c_1, \dots, c_{n-1}\}$ que suma k; en el segundo, existe un subconjunto de $\{c_1, \dots, c_{n-1}\}$ que suma $k - c_n$.

qvq ss(C, k) = V $\longleftrightarrow$ $\exists$ a $\in$ solucionesSubconjuntos(C, k) tal que $\sum_{i=1}^{n} a_i . C_i = k$

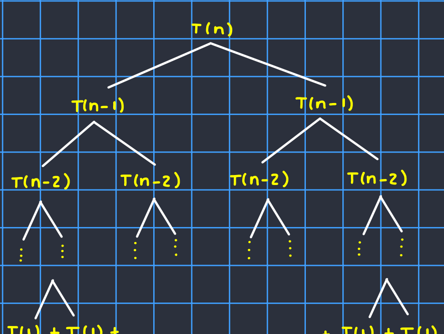
okay tiene sentido

f) Convencerse de que la siguiente es una implementación recursiva de ss en un lenguaje imperativo y de que retorna la solución para C, k cuando se llama con $C, |C|, k$. ¿Cuál es su complejidad?

1) subset_sum($C, \overset{n}{i}, \overset{k}{j}$): // implementa $\operatorname{ss}(\{c_1, \dots, c_i\}, j)$
2) Si $i = 0$, retornar $(j = 0)$
3) Si no, retornar subset_sum$(C, i-1, j) \vee$ subset_sum$(C, i-1, j - C[i])$

Me convence que funciona xq es la implementación literal de lo de arriba

$$\begin{cases} T(1) = O(1) \\ T(n) = 2T(n-1) + O(1) \end{cases}$$



$$\left.\begin{array}{l} T(n) = O(1) \\ + \\ 2T(n-1) = 2.O(1) \\ + \\ 4T(n-2) = 4.O(1) \\ \vdots \\ + \\ 2^n . T(1) = 2^n . O(1) \end{array}\right\} \text{altura} = n$$

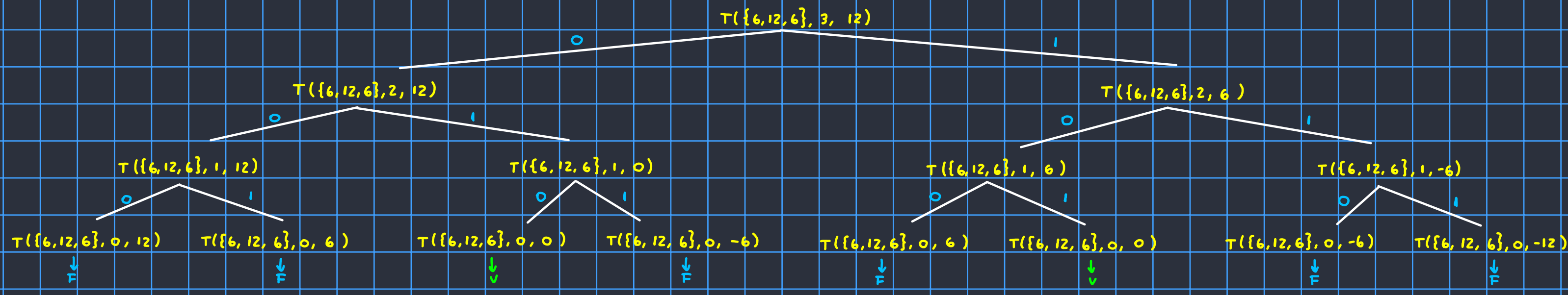
complejidad: $\sum_{i=0}^{n} 2^i . O(1)$

$= \frac{2^{n+1} - 1}{2 - 1}$

$= 2^{n+1} - 1$

$\Rightarrow$ RTA: $\Theta(2^n)$

g) Dibujar el árbol de llamadas recursivas para la entrada $C = \{6, 12, 6\}$ y $k = 12$, y compararlo con el árbol de *backtracking*.



h) Considerar la siguiente *regla de factibilidad*: $p = (a_1, \ldots, a_i)$ se puede extender a una solución válida sólo si $\sum_{q=1}^{i} a_q c_q \leq k$. Convencerse de que la siguiente implementación incluye la regla de factibilidad.

1) subset_sum(C, i, j): // implementa ss$(\{c_1, \ldots, c_i\}, j)$
2) Si $j < 0$, retornar **falso** // regla de factibilidad
3) Si $i = 0$, retornar $(j = 0)$
4) Si no, retornar subset_sum$(C, i-1, j) \vee$ subset_sum$(C, i-1, j - C[i])$

makes sense

i) Definir otra regla de factibilidad, mostrando que la misma es correcta; no es necesario implementarla.

Idea: Si j > 0 entonces me gustaría que la sumatoria de todos los siguientes elementos sea mayor estricto que 0.

(la complejidad queda fea pero bueno)

Idea 2: Si i = 1 y j > 0, entonces no es necesario chequear subset (C, i-1, j) porque se que es falso.

d/ subset (C, i-1, j)
= subset (C, 1-1, j)
= subset (C, 0, j)
= j == 0
= False

1) subset_sum(C, i, j): // implementa ss$(\{c_1, \ldots, c_i\}, j)$
2) Si $j < 0$, retornar **falso** // regla de factibilidad
3) Si $i = 0$, retornar $(j = 0)$
4) Si no, retornar subset_sum$(C, i-1, j) \vee$ subset_sum$(C, i-1, j - C[i])$

evito este chequeo
(la complejidad queda igual pero idk)

Idea 3: cortar el algoritmo apenas encuentro un True.

j) Modificar la implementación para imprimir el subconjunto de C que suma k, si existe. **Ayuda:** mantenga un vector con la solución parcial p al que se le agregan y sacan los elementos en cada llamada recursiva; tenga en cuenta de no suponer que este vector se copia en cada llamada recursiva, porque cambia la complejidad.

```
// Ejercicio 1.j

subset_sum(C,i,j,&p){
    if (j<0) return false;
    if (i==0){
        if (j==0) mostrar p;
        return j==0;
    } else {
        bool recuConCero = subset_sum(C,i-1,j,p)
        p.push_back(C[i]);
        bool recuConUno  = subset_sum(C,i-1,j-C[i],p)
    }
}
```

2. Un *cuadrado mágico de orden* n, es un cuadrado con los números $\{1, \ldots, n^2\}$, tal que todas sus filas, columnas y las dos diagonales suman lo mismo (ver figura). El número que suma cada fila es llamado *número mágico*.

2	7	6
9	5	1
4	3	8

Existen muchos métodos para generar cuadrados mágicos. El objetivo de este ejercicio es contar cuántos cuadrados mágicos de orden n existen.

a) ¿Cuántos cuadrados habría que generar para encontrar todos los cuadrados mágicos si se utiliza una solución de fuerza bruta?

Obs: Mi cuadrado contiene sin repetición los elementos entre el 1 y el n^2.

Hay $n^2!$ formas de generar cuadrados de estas características, ya que es lo mismo que permutar n^2 elementos distintos.

b) Enunciar un algoritmo que use *backtracking* para resolver este problema que se base en la siguientes ideas:

- La solución parcial tiene los valores de las primeras $i-1$ filas establecidos, al igual que los valores de las primeras j columnas de la fila i.
- Para establecer el valor de la posición $(i, j+1)$ (o $(i+1, 1)$ si $j = n$ e $i \neq n$) se consideran todos los valores que aún no se encuentran en el cuadrado. Para cada valor posible, se establece dicho valor en la posición y se cuentan todos los cuadrados mágicos con esta nueva solución parcial.

Mostrar los primeros dos niveles del árbol de *backtracking* para $n = 3$.

$+i \downarrow \; \rightarrow +j$

1	2	3	4
5	6	$\underline{7}$	_
_	_	_	_
_	_	_	_

$i = 2, \; j = 3$

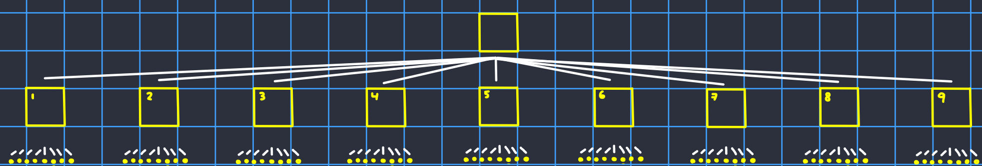
Idea: Voy a ir colocando elementos recursivamente, y cada vez que completo una fila, columna o diagonal me fijo su correctitud.

Obs: Quiero chequear filas cuando $j = n$, columnas cuando $i = n$ y diagonales cuando $(i,j) \in \{(1,n), (n,n)\}$

```
// Fuerza Bruta
int cuantosCuadradosHay(cuadrado, restantes, i, j){
    if (lleneCuadrado && cuadradoEsCorrecto){
        cuadradosPosibles++;
    } else {
        for (int elem : restantes){
            coloco elem en i,j;
            quito elem de restantes;

            if (j==n){
                cuantosCuadradosHay(cuadrado, restantes, i+1, 1);
            } else {
                cuantosCuadradosHay(cuadrado, restantes, i, j+1);
            }

            quito elem de i,j;
            agrego elem en restantes;
        }
    }
}
```

```
// Backtracking
// > PODA 1: Al terminar una fila, columna o diagonal, chequea que está correcta.

int cuantosCuadradosHay(cuadrado, restantes, i, j){
    if (lleneCuadrado){
        cuadradosPosibles++;
    } else {
        for (int elem : restantes){
            coloco elem en i,j;
            quito elem de restantes;

            if (lleneFila && i!=1) me fijo si la fila está correcta
            if (lleneColumna)      me fijo si la columna está correcta
            if (lleneDiagonal)     me fijo si la diagonal está correcta

            if (estaTodoCorrecto){
                if (j==n){
                    cuantosCuadradosHay(cuadrado, restantes, i+1, 1);
                } else {
                    cuantosCuadradosHay(cuadrado, restantes, i, j+1);
                }
            }

            quito elem de i,j;
            agrego elem en restantes;
        }
    }
}
```

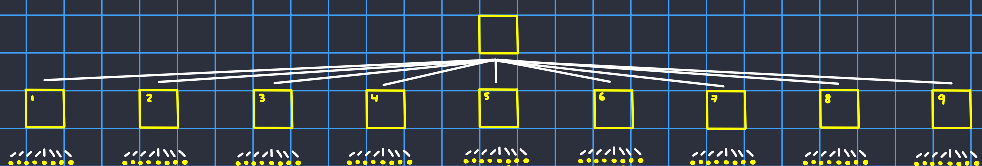
Mostrar los primeros dos niveles del árbol de *backtracking* para $n = 3$.



son cuadraditos vistos desde lejos

8 nodos

c) Demostrar que el árbol de *backtracking* tiene $\mathcal{O}((n^2)!)$ nodos en peor caso.



Piso 1: 1 nodo

Piso 2: n^2 nodos

Piso 3: $n^2.(n^2-1)$ nodos

Piso 4: $n^2.(n^2-1).(n^2-2)$ nodos

$\vdots$

Piso n^2-1: $n^2.(n^2-1).(n^2-2) \dots .(n^2-(n^2-1))$ nodos

Complejidad exacta: $\sum_{i=0}^{n^2-1} \frac{n^2!}{(n^2-i-1)!}$

wolfram:

Input interpretation

$$\sum_{x=0}^{-1+m} \frac{m!}{(m-x-1)!}$$

$n!$ is the factorial function

Sum

$$\sum_{x=0}^{m-1} \frac{m!}{(m-x-1)!} = e\, m\, \Gamma(m, 1)$$

jaja wtf

$n!$ is the factorial function

$\Gamma(a, x)$ is the incomplete gamma function

Como está difícil resolver la sumatoria, voy a calcular una cota superior (total me están pidiendo ver si es $O((n^2)!)$)

$$\sum_{i=0}^{n^2-1} \frac{n^2!}{(n^2-i-1)!} \leq \sum_{i=0}^{n^2-1} n^2!$$

$$\leq n^2 . (n^2)!$$ ⟶ y que $O(n^2 . (n^2)!) \subseteq O((n^2)!)$, me fijo si se cumple por límite

$$\lim_{n \to \infty} \frac{n^2 . \cancel{(n^2)!}}{\cancel{(n^2)!}} = n^2 = +\infty \Rightarrow O((n^2)!) \subseteq O(n^2 . (n^2)!)$$

necesito una cota mejor

Quiero demostrar que $O\left(\sum_{i=0}^{m-1} \frac{m!}{(m-i-1)!}\right) \subseteq O(m!)$

$$\lim_{m \to \infty} \frac{\sum_{i=0}^{m-1} \frac{m!}{(m-i-1)!}}{m!} = \frac{\cancel{m!} \sum_{i=0}^{m-1} \frac{1}{(m-i-1)!}}{\cancel{m!}} = \sum_{i=0}^{m-1} \frac{1}{(m-i-1)!} < \infty$$

$$\left[\frac{1}{(m-1)!} + \frac{1}{(m-2)!} + \dots + 1 \right] < m$$

Entonces como la sumatoria es menor estricta que m y m tiende a infinito, mi límite da menos que infinito:

$$\lim_{m \to \infty} \frac{\sum_{i=0}^{m-1} \frac{m!}{(m-i-1)!}}{m!} \in [0, \infty) \iff O\left(\sum_{i=0}^{m-1} \frac{m!}{(m-i-1)!}\right) \subseteq O(m!)$$

d) Considere la siguiente poda al árbol de *backtracking*: al momento de elegir el valor de una nueva posición, verificar que la suma parcial de la fila no supere el número mágico. Verificar también que la suma parcial de los valores de las columnas no supere el número mágico. Introducir estas podas al algoritmo e implementarlo en la computadora. ¿Puede mejorar estas podas?

```
// > PODA 2: Al agregar un elem a una fila, chequeo que la suma parcial de la fila sea menor que el número
// mágico

int cuantosCuadradosHay(cuadrado, restantes, i, j, sumaParcial){
    if (lleneCuadrado){
        cuadradosPosibles++;
    } else {
        for (int elem : restantes){
            coloco elem en i,j;
            quito elem de restantes;

            sumaParcial += elem;

            if (lleneFila && i!=1) me fijo si la fila está correcta
            if (lleneColumna)      me fijo si la columna está correcta
            if (lleneDiagonal)     me fijo si la diagonal está correcta

            if (estaTodoCorrecto && sumaParcialEsCorrecta){
                if (j==n){
                    cuantosCuadradosHay(cuadrado, restantes, i+1, 1, 0);
                } else {
                    cuantosCuadradosHay(cuadrado, restantes, i, j+1, sumaParcial);
                }
            }

            sumaParcial -= elem;

            quito elem de i,j;
            agrego elem en restantes;
        }
    }
}
```

Idea para otra poda:

$e)$ Demostrar que el número mágico de un cuadrado mágico de orden n es siempre $(n^3 + n)/2$. Adaptar la poda del algoritmo del ítem anterior para que tenga en cuenta esta nueva información. Modificar la implementación y comparar los tiempos obtenidos para calcular la cantidad de cuadrados mágicos.

n=1

1

• número mágico = 1

n=2

1	4
2	3

• número mágico = no existe mepa

Obs: el núm mág como mínimo es $1+n^2$

n=3

2	7	6
9	5	1
4	3	8

• Sigo después

3. Dada una matriz simétrica M de $n \times n$ números naturales y un número k, queremos encontrar un subconjunto I de $\{1, \ldots, n\}$ con $|I| = k$ que maximice $\sum_{i,j \in I} M_{ij}$. Por ejemplo, si $k = 3$ y:

$$M = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{pmatrix} 0 & 10 & 10 & 1 \\ 10 & 0 & 5 & 2 \\ 10 & 5 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{pmatrix} \end{array},$$

???

entonces $I = \{1, 2, 3\}$ es una solución óptima.

$a)$ Diseñar un algoritmo de *backtracking* para resolver el problema, indicando claramente cómo se codifica una solución candidata, cuáles soluciones son válidas y qué valor tienen, qué es una solución parcial y cómo se extiende cada solución parcial.

$I \subseteq \{1, 2, \ldots, n\}$, por lo que probar con todos los posibles I es $O(2^n)$

pero yo se que $|I| = k$, entonces hay $\binom{n}{k}$ posibles I.

1- Idea backtracking

• Caso base: $|I|=k$, si encuentro un máximo lo guardo.

• Recursión:

```
for (int i = 0,  i < n, i++){
    if (i ∉ I){
        I = I ∪ {i}
        Función (Matriz, I, k)
        I = I - {i}
    }
}
```

2. Idea backtracking (con poda)

• Caso base: $|I|=k$, si encuentro un máximo lo guardo.

• Recursión:

```
for (int i = j,  i < n, i++){
    if (i ∉ I){
        I = I ∪ {i}
        Función (Matriz, I, k, j+1)
        I = I - {i}
    }
}
```

Yo quiero encontrar un intervalo I ($I \subseteq \{1, 2, \ldots, n\}$) tal que maximice la operación $\sum_{i,j \in I} M_{ij}$, siendo M una matriz simétrica. Como para armar cada posible I tengo que elegir k elementos del conjunto $\{1, 2, \ldots, n\}$, se que en total hay $\binom{n}{k}$ posibles I.

Voy a armar un algoritmo que en cada paso recursivo pruebe agregarle un nuevo elemento a I. El caso base es $|I| = k$, y ahí es cuando evalúo si la suma lograda con ese I es la mejor hasta ahora. Si encontré un máximo, lo guardo junto con I en una variable global.

Al finalizar el algoritmo, en las variables globales van a estar guardadas las respuestas.

• Una solución candidata se representa con un arreglo de 0 y 1, que indican qué elementos de $\{1, 2, \ldots, n\}$ pertenecen a la solución.

Se cumple que $\sum_{i=0}^{n-1} I[i] = k$. Si la sumatoria es menor estricta que k, entonces se la considera una solución parcial.

Las soluciones parciales se extienden cambiando 0 por 1.

• Posibles podas:

1. Observación: no nos importa el orden del conjunto ($\{1,2,3\} =_{obs} \{3,2,1\}$), y ésto no lo estoy teniendo en cuenta al momento de extender mis soluciones parciales, porque en la instancia j estoy agregando items que ya agregué en instancias $j-1$.

Resuelvo ésto en el algoritmo 2.

b) Calcular la complejidad temporal y espacial del mismo.

La complejidad espacial es $O(n^2)$, ya que estoy guardando una matriz y dos vectores I. Si pasase la matriz por referencia, la complejidad espacial es $O(n)$.

El algoritmo explora $\binom{n}{k}$ posibles I, y en cada caso base hago una operación $O(n^2)$, por lo que la complejidad temporal es $O(n^2 \cdot \binom{n}{k})$

4. Dada una matriz D de $n \times n$ números naturales, queremos encontrar una permutación π[1] de $\{1, \ldots, n\}$ que minimice $D_{\pi(n)\pi(1)} + \sum_{i=1}^{n-1} D_{\pi(i)\pi(i+1)}$. Por ejemplo, si

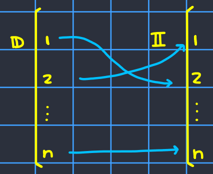
$$M = \begin{pmatrix} 0 & 1 & 10 & 10 \\ 10 & 0 & 3 & 15 \\ 21 & 17 & 0 & 2 \\ 3 & 22 & 30 & 0 \end{pmatrix},$$

entonces $\pi(i) = i$ es una solución optima.

[1] Una permutacion de un conjunto finito X es simplemente una función biyectiva de X en X.

• Quiero encontrar la función $\pi : X \to X$ biyectiva tal que minimice la operación $D_{\pi(n)\pi(1)} + \sum_{i=1}^{n-1} D_{\pi(i)+\pi(i+1)}$.

Obs: El dominio de π es X, y la imagen también.



Quiero buscar la mejor permutación de Π.

Hay $n!$ posibles Π.

a) Diseñar un algoritmo de *backtracking* para resolver el problema, indicando claramente cómo se codifica una solución candidata, cuáles soluciones son válidas y qué valor tienen, qué es una solución parcial y cómo se extiende cada solución parcial.

Idea: Armo un arreglo de n casillas con los números del 1 al n. Éste arreglo va a servir a modo de diccionario para realizar mi operación π:
$\pi(i)$ = Array[i]. Cada solución válida va a ser una instancia permutada de éste mismo arreglo.
En cada paso recursivo i voy a swapear el elem i con un elemento i+1 y shuflear recursivamente el arreglo.
Al llegar al caso base, chequeo cuánto da la operación que queremos minimizar, y si encuentro un nuevo mínimo copio el array y el mínimo encontrado en variables globales.

• El algoritmo:

```
void BuscoImagen(array I, matriz M, int i)
    if (i == n) {
        if (¿la imagen minimiza la operación?) guardo imagen y mínimo
    } else {
        para todo entero j entre i y n {
            swapeo i y j
            shufleo recursivamente el arreglo a partir de i+1
            swapeo i y j
        }
    }
```

• Podas:

1. Teniendo una solución parcial hasta el elemento i, si encuentro que la operación que queremos minimizar (es decir, $D_{\pi(i)\pi(1)} + \sum_{k=1}^{i-1} D_{\pi(k)\pi(k+1)}$) ya supera el mejor mínimo alcanzado, entonces no es necesario completar esa solución.

• El algoritmo:

```
void BuscoImagen(array I, matriz M, int i)
    if (i==n) {
        if (¿la imagen minimiza la operación?) guardo imagen y mínimo
    } else {
        Para todo entero j entre i y n {
            swapeo i y j
            if (¿No superé el mínimo?) shufleo recursivamente el arreglo a partir de i+1
            swapeo i y j
        }
    }
```

b) Calcular la complejidad temporal y espacial del mismo.

• Complejidad espacial: La matriz es $O(n^2)$ y los dos arreglos son $O(n)$. Si la matriz es pasada por referencia, la complejidad espacial es de $O(n)$

• Complejidad temporal: Como cada solución válida es una permutación de $\{1, 2, \dots, n\}$, en total hay $n!$ soluciones posibles.

Piso 1 $\quad a_0$

Piso 2 $\quad a_1 + a_1 + \dots + a_1 + a_1$

Piso 1: 1 nodo

Piso 2: n nodos

Piso 3: $n.(n-1)$ nodos

⋮

Piso n: $n!$ nodos

Van a haber $\sum_{i=1}^{n} \frac{n!}{i!} = e.n!$ nodos, por lo que si no realizo ninguna poda la complejidad sería $O(n!)$

si guardo una variable extra para almacenar la suma actual, chequear que no superé el mínimo es $O(1)$.

Entonces, al aplicar mi poda la complejidad temporal mejora pero no puedo saberla con exactitud (xq no se cuántos nodos estoy descartando)