Apunte Único: Técnicas de diseño de algoritmos - Práctica 2

By naD GarRaz

última actualización 24/01/26 @ 20:09

Choose your destiny:

(click click 🖱 en el ejercicio para saltar)

Esta Guía 2 que tenés se actualizó por última vez:
24/01/26 @ 20:09
Escaneá el QR para bajarte (quizás) una versión más nueva:

Guía 2

El resto de las guías repo en github para descargar las guías con los últimos updates.

Si querés mandar un ejercicio o avisar de algún error, lo más fácil es por Telegram.

Notas teóricas:

- *Fuerza Bruta*:

  Busca todas las combinaciones. Si hay solución la encuentra, porque encuentra TODO.

- *Backtracking*:

  Backtracking es fuerza bruta con podas. Una poda detecta que la rama de recursión no puede ser una solución factible entonces corta los cálculos y continúa por la otra rama recursiva en caso de que satisfaga la guarda de la poda.

- *Programación dinámica*: Es *backtracking* sin calcular 2 veces la misma cosa.

  Viene en 2 sabores *Memoization* = *Top-Down*, *Tabulation* = *Bottom-Up*.

  - *Top-down* sale inmediatamente con una función recursiva bien hecha.
  - *Bottom-up* casi siempre más eficiente en memoria.

  Complejidad temporal de un algoritmo dinámico:

  $$O\big(\textbf{(cantidad de estados)} \times \textbf{(costo de resolver internamente cada estado).}\big)$$

  Complejidad espacial de un algoritmo dinámico:

  $$O\big(\textbf{(cantidad de estados)}\big)$$

  Un *estado* es lo que vale la función para unas entradas en particular. Los valores que se usan para conseguir el estado son las *variables de estado*.

- *Receta para para ejercicio de progamación dinámica:*

  1) Definir función recursiva f de fuerza bruta como siempre.
  2) Explicar lo que hace la función f, lo que hacen sus variables. Tratar de no usar más de dos parámetros para definir un estado.
  3) ¿Para que variables de estado la función devuelve lo que quiero?
  4) Probar que f cumple con la propiedad de superposición de problemas:

     La superposición va a aparecer cuando la función f haga muchas más llamadas que resultados, es decir llamás a $f(i)$ 20 veces con $i = 1$, la función devuelve siempre lo mismo. Entonces si las llamadas de f están en $\Omega(algo)$ y los posibles estados son $\mathrm{O}(algo\ mucho\ menor)$ es que hay superposición de estados o subproblemas.

  5) Memoization, *top-down* sale muy fácil si tenés la función recursiva
  6) Tabulation, *bottom-up* entender como se relaciona un estado con el siguiente, cómo es la dependencia de las variables de estados y estados entre sí.
  7) Decidir cuál variables es el `outer loop` y cual el `inner loop` y llenar la matriz. Esto si se tiene lo anterior es mecánico.
  8) Determinar complejidad del algoritmo. Una vez hecho lo anterior es siempre lo mismo.

- *Greedy:*

  - *Exchange Argument:*
    i) Considero una solución Optima que no sea la *Greedy*.
    ii) Hago un cambio en $O \to O'$ de manera que quede O'*más parecida a la Greedy*
    iii) Me fijo lo que tengo y pruebo que efectivamente O' es mejor que O, y como O' es más parecida a la *Greedy* llego a una contradicción.
    iv) Concluyo entonces que para que la solución sea óptima tiene que ser construída de forma *Greeeeedy*

Ejercicios de la guía:

---

1. *SumaSubconjuntos BT*

En este ejercicio vamos a resolver el problema de suma de subconjuntos con la técnica de *backtracking*. Dado un multiconjunto $C = \{c_1, \ldots, c_n\}$ de números naturales y un natural k, queremos determinar si existe un subconjunto de C cuya sumatoria sea k. Vamos a suponer fuertemente que C está ordenado de alguna forma arbitraria pero conocida (i.e., C está implementado como la secuencia $c_1, \ldots, c_n$ o, análogamente, tenemos un iterador de C). Las *soluciones (candidatas)* son los vectores $a = (a_1, \ldots, a_n)$ de valores binarios; el subconjunto P 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 $p = (a_1, \ldots, a_i)$ de números binarios con $0 \leq i \leq n$. 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 $k = 12$.

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

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

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 conjunto de los incisos anteriores.

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 $\mathbb{N} = \{0, 1, 2, \ldots\}$, V indica verdadero y F falso:

$$ss(\{c_1, \ldots, c_n\}, k) = \begin{cases} k = 0 & \text{si} \quad n = 0 \\ ss(\{c_1, \ldots, c_{n-1}\}, k) \wedge ss(\{c_1, \ldots, c_{n-1}\}, k - c_n) & \text{si} \quad n > 0 \end{cases}$$

Convencerse de que $ss(C, k) = V$ si y solo 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 $a = (a_1, \cdots, a_n)$ para el caso $n > 0$: o bien $a_n = 0$ o bien $a_n = 1$. En el primer caso, existe un subconjunto de $\{c_1, \cdots, c_{n-1}\}$ que suma k; en el segundo, existe un subconjunto de $\{c_1, \cdots, c_n\}$ que suma $k - c_n$.

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, i, j): // implementa $ss(c_1, \cdots, c_i, j)$
2) Si $i = 0$, retornar $(j = 0)$
3) Si no, retornar `subset_sum`$(C, i - 1, j) \vee$ `subset_sum`$(C, i - 1, j - C[i])$

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 solo 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`$(C, i - 1, j - C[i])$

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

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

---

Si bien el enunciado este es demasiado largo para mi gusto, explica el vocabulario de *backtracking* de manera concisa.

a) *Soluciones candidatas:* Como dice en el enunciado, son los vectores con valores 0 o 1:

Son de la forma $a = (a_1, a_2, a_3)$ donde están toooodas las posibles combinaciones (no confundir con *soluciones válidas*):

$$\{(1,1,1),\ (1,1,0),\ (1,0,1),\ (1,0,0),\ (0,1,1),\ (0,1,0),\ (0,0,1),\ (0,0,0)\}$$

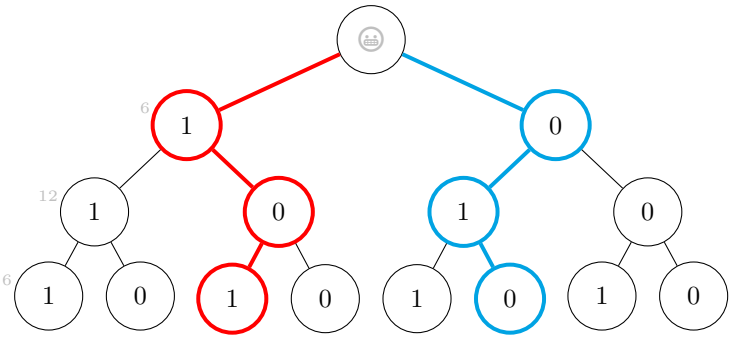
b) *Soluciones válidas:* Como dice el enunciado son de la forma $a = (a_1, a_2, a_3)$ tal que $\sum_{i=1}^{3} a_i c_i = 12$.

$$\{(1,0,1),\ (0,1,0)\}$$

c) *Soluciones parciales:* Son soluciones a ¿medio hacer? Son de la forma $a = (a_1)$, $a = (a_1, a_2)$ o $a = (a_1, a_2, a_3)$

$$\{(1),\ (0),\ (1,0),\ (0,1),\ (1,0,1),\ (0,1,0)\}$$

d) ¿De qué algoritmo habla el enunciado? *Backtracking* para mí es *fuerza bruta*. Fin.



Los caminos marcados son las *soluciones válidas* y cualquier camino dentro serían *soluciones parciales*. Cualquier camino, siempre yendo para abajo es una *solución candidata*.

e) Este ítem tiene como objetivo mostrar que *solo* hay que ver el caso recursivo y el caso base de la función para convencerse de que esto funciona.

Cambio la notación, *because* me parece más fácil para el primer ejercicio. $ss \to sumaSubset$, $C = \{c_1, \ldots, c_n\}$ y escribo la función con las condiciones más explícitas que como está en el enunciado.

$$sumaSubset(C,k) = \begin{cases} V & \text{si} \quad |C| = \varnothing \wedge k = 0 \\ F & \text{si} \quad |C| = \varnothing \wedge k \neq 0 \\ sumaSubset(C - \{c_n\}, k) \vee sumaSubset(C - \{c_n\}, k - c_n) & \text{otro caso} \end{cases}$$

La función esa está tomando en cada llamado recursivo las dos únicas opciones: Usar o no usar c_i, no puede fallar a menos que no exista una combineta de los elementos de C tal que la suma de k.

Más formal escribiendo lo que dice el enunciado:

$$\begin{aligned} \sum_{j=1}^{n-1} a_j c_j &= a_1 c_1 + \cdots + a_{n-1} c_{n-1} = k - 0 \cdot c_n \Leftrightarrow \sum_{j=1}^{n} a_j c_j = k \\ \sum_{j=1}^{n-1} a_j c_j &= a_1 c_1 + \cdots + a_{n-1} c_{n-1} = k - 1 \cdot c_n \Leftrightarrow \sum_{j=1}^{n} a_j c_j = k \end{aligned}$$

Por lo tanto en la disyunción ($\vee$) de la función habrá una proposición verdadera (siempre que exista una solución válida claro).

f) Más feo no se puede escribir un algoritmo, no?

El algoritmo tiene un caso base cuando `i=0`, cuando terminé los elementos del conjunto C. Mientras haya elementos, voy a ir probando con cada elemento para ver si se cumple que la suma de los elementos tomados es igual a `j`.

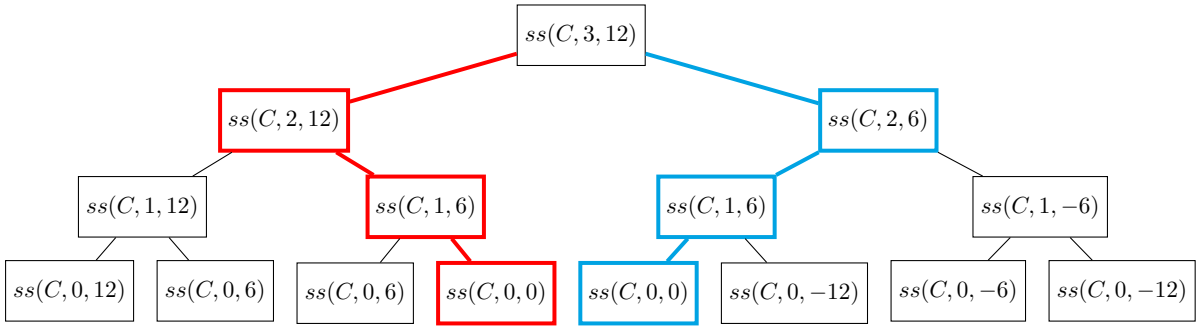
```
1 funcion sumaSubset(C,i,j) → bool
2     si i=0
3         ret j=0
4     sino
5         ret sumaSubset(C,i-1,j) o sumaSubset(C,i-1,j-C[i])
```

El algoritmo tiene una complejidad dada por la función costo:

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 0 \\ 2T(n-1)+1 & \text{si} \quad n > 0 \end{cases} \implies \boxed{T(n) \in O(2^n)}$$

Árbol de 2 ramas con un costo por subproblema de $O(1)$, lo cual equivale a contar cada uno de los nodos del árbol.

g)



Son lo mismo, porque este árbol no tiene ninguna poda. Es un árbol de fuerza bruta que pasa por todas las *soluciones candidatas*.

h) Esa *regla de factibilidad* se puede traducir a lenguaje natural como:

> Tengo algo y si todavía con este algo no me pasé de k tengo chances de que este algo sea una solución válida.

```
1 funcion sumaSubset(C,i,j) → bool
2     si j<0                    // Me pasé del objetivo?
3         ret false
4     si i=0
5         ret j=0
6     sino
7         ret sumaSubset(C,i-1,j) o sumaSubset(C,i-1,j-C[i])
```

Cualquier llamada recursiva a *sumaSubset* que tenga en su tercer argumento algo negativo ¡No continuará haciendo llamadas recursivas la recursión!, eso corta una *subrama* del árbol, es decir que *se podó el árbol* ✂🌲 *get it?*

i) No sé me ocurre nada interesante.

- Debe existir algún elemento $c_i \in C$ tal que $c_i < k$

j) La función recibe un conjunto, la cantidad de elementos, y el objetivo: C, i, j respectivamente. Paso por *referencia* un vector `p` en el cual voy a ir poniendo y sacando elementos, formando así las *soluciones factibles*. Lo imprimo si la solución termina siendo una *solución válida*

La forma en que se arma el vector solución es probando por fuerza bruta cada elemento. Si el llamado recursivo retorna *false* previamente se elimina el `C[i]` que se estaba probando.

```
1  funcion sumaSubset(C,i,j,p)  // p es el vector solución pasado por referencia a la función
2      p.meto(C[i])
3      si j = 0   // Es válida?
4          print p
5          ret true
6
7      si i < 0 o j < 0
8          ret false
9
10     si sumaSubset(C,i-1,j-C[i],p)  // con C[i]
11         p.saco(C[i])               // backtracking. Vuelvo a línea temporal donde nunca agarré el C[i]
12         ret true
13
14     p.saco(C[i])                 // backtracking. Vuelvo a línea temporal donde nunca agarré el C[i]
15     si sumaSubset(C,i-1,j,p)  // sin C[i]
16         ret true
17
18     sino
19         ret false
```

---

2. *MagiCuadrados*

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?

b) Enunciar un algoritmo que use *backtracking* para resolver este problema que se base en las 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 $n = 3$.

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

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?

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.

---

a) puff, un montón! Con un poco más de precisión, puedo poner en cada uno de los n^2 elementos de la matriz

un número del 1 al n^2 sin repetir:

$$n^2! = n^2 \cdot (n^2-1) \cdot (n^2-2) \cdots (n^2-(n^2-1)) = \prod_{j=0}^{n^2-1} n^2 - j$$

Una vez generados esos cuadrados me pongo a mirar cuales cumplen las condiciones para ser *cuadrados mágicos*.

b) El enunciado está bien escrito. Enfoca el ejercicio con la información clave para armar el llamado recursivo. Nos dice entre otras cosas como elegir el siguiente cuadrado para rellenar.

Voy a hacer distintas versiones de resolución por las que pasé a lo largo de esta materia. Sobre gustos no hay nada escrito así que mientras estén bien (cosa que no sé) valen. Hay que tener cuidado que cumplan la complejidad requerida en cada instancia.

Una forma, con $C = \{1, \ldots, n^2\}$, cada llamado recursivo toma un elemento de C y lo quita, de forma que en la próxima llamada agarre un número distinto. El caso base se da cuando $C = \varnothing$:

$$magiCuadrados(C, i, j, n) = \begin{cases} 1 & \text{si} \quad C = \varnothing \\ \sum_{c_i \in C} magiCuadrados(C - \{c_i\}, getNext(i, j, n), n) & \text{si no} \end{cases}$$

Donde $getNext(i, j, n)$:

$$getNext(i, j, n) = \begin{cases} (i+1, 1) & \text{si} \quad j = n \\ (i, j+1) & \text{si no} \end{cases}$$

Un pseudocódigo de esa función: Calcula los cuadrados que se pueden armar sin ninguna poda. Toma y elimina un elemento de un conjunto C y luego hace un llamado recursivo, completando la matriz con un elemento distinto en cada posición.

El caso base lo tengo cuando usé todo el conjunto. Voy llenando la matriz por filas.

```
funcion magiCuadrados(C, i, j)
    si C = ∅
        ret 1
    suma ← 0
    para cada c en C
        suma ← suma+magiCuadrados(C-{c},getNext(i,j,n))
    ret suma
```

El premio a lo más cómodo para lidiar con matrices es usar la función *indices* que te mapea:

$$\underbrace{\begin{pmatrix} 1 & \cdots & n \\ \vdots & \ddots & \vdots \\ n^2-(n-1) & \cdots & n^2 \end{pmatrix}}_{\text{posiciones}} \rightarrow \underbrace{\begin{pmatrix} (1,1) & \cdots & (1,n) \\ \vdots & \ddots & \vdots \\ (n,1) & \cdots & (n,n) \end{pmatrix}}_{\text{índices}}$$

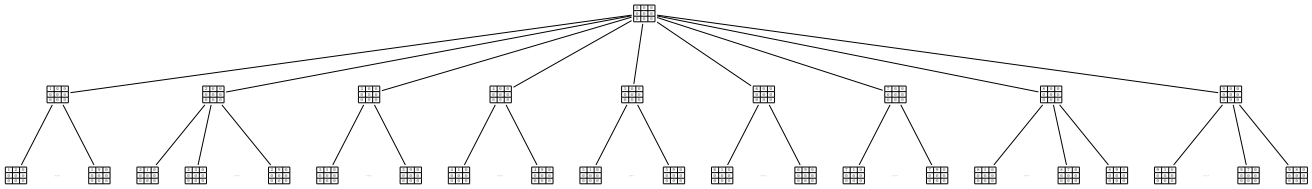
Por lo tanto pasás un solo valor a la función y tenés la funcionalidad de 2, ¡Una ganga!. Cambio el caso base. Cuando la posición sea mayor a n^2 es que completé el cuadrado.

```
funcion magiCuadrados(C, pos, n)
    si pos > n^2
        ret 1
    suma ← 0
    (i,j) ← indices(pos)
    para cada c en C
        suma ← suma+magiCuadrados(C-{c},pos+1, n)
    ret suma
```

```
funcion indices(pos, n)
    pos ← pos - 1
    return (pos/n, pos%n) // 0 indexed
  //return ((pos/n)+1, (pos%n)+1) // 1 indexed
```

Y por último cómo puedo hacer para no pasar ese conjunto `C` por copia: Voy a ir marcando elementos en un bitarray. *Esto es común cuando ves recorridos en grafos, no es lo más intuitivo imho.* Optimizar estas boludeces no debería ser foco de este ejercicio, pero está bueno para lo que viene.

```
funcion magiCuadrados(pos, marcados, n)
    si posicion > n^2
        ret 1
    suma ← 0
    (i,j) ← indices(pos, n)
    para cada c ← 1 .. n^2
        si c not marcados[c]
            marcados[c] ← true
            A[i,j] ← c
            suma ← suma+magiCuadrados(pos+1, marcados, n)
            A[i,j] ← 0                    // Backtrack. Restauro la matriz
            marcados[c] ← false           // Desmarco
    ret suma
```



c) La función de costo $T(n^2)$ debería ser algo así, dado que tengo para elegir n^2 elementos del conjunto $C_{n^2} = \{1, 2, \ldots, n^2 - 1, n^2\}$ y en cada llamado recursivo, el *cardinal* de C_{n^2} disminuye en 1. La función de costo $T(n^2)$ sería algo así:

$$T(n^2) = \begin{cases} 1 & \text{si} \quad i = j = n \\ n^2 T(n^2 - 1) + 1 & \text{si} \quad \textit{sino} \end{cases}$$

Esa ecuación recursiva, sucesión o como se diga tiene como solución a $T(n^2) \in O((n^2)!)$, solución que puede comprobarse como válida reemplazando:

$$\begin{cases} T(n^2) &= (n^2)! \\ T(n^2-1) &= (n^2-1)! \end{cases} \xrightarrow{\text{reemplazando}} \underbrace{(n^2)!}_{\in O((n^2)!)} = \overbrace{\underbrace{n^2 \cdot (n^2-1)!}_{(n^2)!} + 1}^{\in O(n^2)!)}$$

También puedo pensar que estoy en un árbol que tiene costo por *subproblema* de $O(1)$, por lo que la complejidad será contar la cantidad de nodos que hay en árbol:

estos niveles	tendrán	esta cantidad de subproblemas
root = nivel 0	$\rightarrow$	$\binom{n^2}{0} = 1$
nivel 1	$\rightarrow$	$\binom{n^2}{1} = n^2$
nivel 2	$\rightarrow$	$\binom{n^2-1}{1} = n^2 - 1$
$\vdots$	$\vdots$	$\vdots$
nivel $n^2 - 2$	$\rightarrow$	$\binom{n^2-(n^2-2)}{1} = 2$
nivel $n^2 - 1$	$\rightarrow$	$\binom{n^2-(n^2-1)}{1} = 1$

El producto de toda esa 3era columna es $(n^2)!$

d) En este inciso me piden que tenga un registro de los cuadrados A que fui armando (ver último ítem del ejercicio **1.**), porque tengo que sumar sobre valores usados en esa rama en particular. Así que tengo que tener un registro de los valores que fui poniendo en cada lugar de la matriz. *Esto lo destaco, porque antes*

no importaba eso, solo contar las distintas matrices que se podían formar. Así que uso una implementación pasando la matriz por referencia para que no explote la memoria con la complejidad espacial.

Esto se ve mucho mejor cuando se hace programación dinámica.

Si el número mágico es M, lo que busco con la poda es cortar las llamadas para no tener resultado que sé que no sirven retornando un cero.

```
funcion magiCuadrados(pos,marcados,n,A,M) // Paso A y marcados por referencia. M: número mágico.
    if pos > n^2
        ret 1
    suma ← 0
    (i,j) ← indices(pos, n)
    para cada c ← 1 .. n^2
        si not marcados[c] y si poda(c,i,j,n,A,M) // puedo usar c?
            marcados[c] ← true
            A[i][j] ← c
            suma ← suma + magiCuadrados(pos+1,marcados,n,A,M)
            A[i][j] ← 0                    // Restauro A
            marcados[c] ← false            // Restauro marcados
    ret suma
```

```
funcion poda(c,i,j,n,A,M)
    si j=n // Se terminó la fila
        si c+∑col A[i,col]=M
          ret true
    si i=n // Se terminó la columna
        si c+∑fil A[fil,j]=M
          ret true
    sino
        si c+∑col A[i,col]<M // fila a medias
          ret true
        si c+∑fil A[fil,j]<M // columna a medias
          ret true
    ret false
```

El hecho de que eso tenga un loop de n^2 iteraciones por llamado recursivo, hace que cada subproblema cueste por lo menos $O(n^2)$. Parecería que hace que explote todo por los aires, pero dentro de todo las operaciones en el arreglo `marcados` son $O(1)$. Usar el conjunto C tendría una complejidad adicional. Con suerte las podas, si agarran casos cerca de la raíz, el árbol de recursiones se achica de forma apreciable.

Podas adicionales habría que controlar las diagonales.

e) ... hay que hacerlo!

3. *MaxiSubconjunto*

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:

$$M = \begin{pmatrix} 0 & 10 & 10 & 1 \\ - & 0 & 5 & 2 \\ - & - & 0 & 1 \\ - & - & - & 0 \end{pmatrix}$$

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.

b) Calcular la complejidad temporal y espacial del mismo.

c) Proponer una poda por optimalidad y mostrar que es correcta.

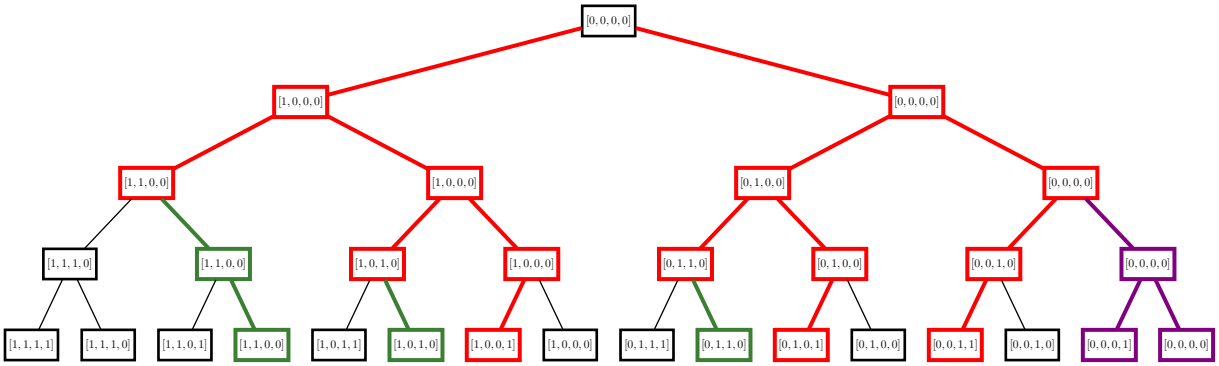
Tener presente que hay distintas formas de implementar estos ejercicios. Dependiendo de como se haga pueden dar distintas complejidades. En este caso no nos piden ninguna complejidad en particular.

a) Lo primero que hay que hacer es entender qué cosa piden. Luego encontrar con *fuerza bruta* una solución. No es buena idea buscar eficiencia, *hackear*, ni un choto. Fuerza bruta y recursión.

Las *soluciones candidatas* van a ser todos los intervalos que me pueda formar tomando k elementos de $\{1, \ldots, n\}$, por ejemplo, $n = 4$ y $k = 2$:

El siguiente árbol de decisión binario tiene todas las 2^4 combinaciones que se pueden formar por fuerza bruta. Los *bitarrays* indican en cada posición si en el conjunto está usando el elemento o no, por ejemplo:

$$[1,0,1,0] \xrightarrow[\text{conjunto}]{\text{equivale al}} \{1,3\} \quad \text{y} \quad [0,1,1,1] \xrightarrow[\text{conjunto}]{\text{equivale al}} \{2,3,4\}$$



Las hojas del subárbol rojo son las soluciones *candidatas* a ser la solución al problema, dado que corresponden a conjuntos de tamaño k. Los nodos en el camino para llegar hasta esas hojas son las *soluciones parciales*.

Las soluciones *válidas* serán un subconjunto de esas hojas mencionadas que maximicen $\sum_{i,j \in I_r} M_{ij}$.

Voy a querer parar de hacer recursión si:

- Cuando se terminó un camino, cuando llegué a una hoja del árbol de recursión, cuándo el itervalo tiene exactamente k elementos distintos, devuelvo una tupla $(I, \sum)$.
- La implementación arranca en un valor $l \in \{1, \ldots, n\}$, así que si $l > n$ ya generé todos los intervalos candidatos. Acá devuelvo una tupla, que tiene la particularidad de nunca ser una respuesta válida.

Luego me quedo con el intervalo I óptimo según la suma $\sum$ que se obtenga en la comparación.

$$maxSubset(k, I, l, n) = \begin{cases} \left(I, \sum\limits_{i,j \in I} M_{ij}\right) & \text{si} \quad |I| = k \\ \left(\varnothing, -\infty\right) & \text{si} \quad l > n \\ \max\limits_{I} \left(\begin{array}{c} maxSubset(k, I, l+1, n), \\ maxSubset(k, I \cup \{l\}, l+1, n) \end{array} \right) & \text{si no} \end{cases}$$

b) Esta función tiene 2 llamados por recursión, lo que daría una complejidad de $O(2^n)$, si bien corta las ramas verdes cuando $|I| = k$ antes de recorrer todo el árbol, no atiende la parte violeta del árbol abajo a la izquierda: Ahí no se detiene. (En el próximo inciso se soluciona eso, paciencia, cdtm)

Cada subproblema consta de sumar $O(k^2)$ elementos de M. Por lo que resulta que el *running time* es:

$$\boxed{T(n) \in O\left(k^2 \cdot 2^n\right)}$$

El árbol de recursión tiene n niveles y teniendo en cuenta que I se pasa por referencia tengo una complejidad espacial:

$$\boxed{S(n) \in O(n)}$$

c) Hay una poda importante que se puede hacer para reducir la complejidad temporal a algo polinomial y es podar las ramas violetas

$$maxSubset(k, I, l, n) = \begin{cases} \left(I, \sum\limits_{i,j \in I} M_{ij}\right) & \text{si} \quad |I| = k \\ \left(\varnothing, -\infty\right) & \text{si} \quad |I| + (n - l + 1) < k \\ \max\limits_{I} \left(\begin{array}{c} maxSubset(k, I, l+1, n), \\ maxSubset(k, I \cup \{l\}, l+1, n) \end{array} \right) & \text{si no} \end{cases}$$

Básicamente dice que si ya no quedan k números para armar un intervalo de tamaño k, que ni lo intente y corte la recursión.

El *running time* con esta poda:

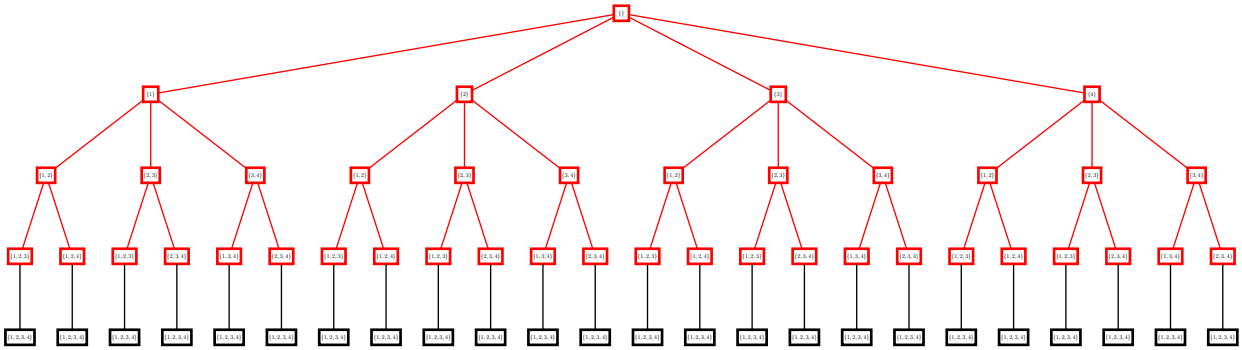
$$\boxed{T(n) \in O\left(k^2 \cdot \binom{n}{k}\right)}$$

Este mismo ejercicio se puede resolver con otra implementación, quizás más intuitiva:

a)

$$maxSubset(I, C) = \begin{cases} \left(I, \sum\limits_{i,j \in I} M_{ij}\right) & \text{si} \quad |I| = k \\ \max\limits_{c \in C} \left(\; maxSubset(I \cup \{c\}, C - \{c\}) \; \right) & \text{si no} \end{cases}$$

¿Cuál es la diferencias? Acá el árbol de esta función tiene otra forma. No es un árbol de decisión binaria, es algo factorial, por ejemplo si $n = 4$ y $k = 3$:



Esos conjuntos que se generan seguro van a resolver el problema, pero hay que *analizar la complejidad* como corresponda.

b) Ese *árbol rojo* tiene $\frac{n!}{(n-k)!}$ hojas. Implementado en un conjunto logarítmico que tiene un costo de agregar de $\log(n)$ cada subproblema tiene un costo de $O(k^2 \cdot \log(k))$:

$$T(n) = O\Big(\log(k) \cdot k^2 \cdot \frac{n!}{(n-k)!}\Big)$$

La complejidad espacial de una recursión de n niveles donde tengo a I y a C, en el caso de que I se pasa por referencia:

$$S(n) \in O(k^2)$$

c) Parecido a lo que se hizo en la implementación anterior, podría ahorrar tiempo guardando la suma parcial de cada intervalo. El hecho de que haya millones de subproblemas iguales me lleva la cabeza a *programación dinámica*, pero hay que ser pacientes.

Y acá un pseudocódigo en imperativo para pasarlo a tu lenguaje favorito un poco más friendly (si tu lenguaje favorito es Haskell, hacete ver) de la primera implementación:

```
1   funcion maxSubset(r,k,I,l,n)
2       si |I|=k
3           ret {I,∑_{i,j∈I} M[i,j]}
4       si |I|+(n-l+1)<k
5           ret {∅, -∞}
6       máximo ← -∞
7       // Arranca rama que usa l
8       I.meto(l)
9       máximo ← max(máximo, maxSubset(r+1,k,I,l+1,n)[2])
10      // Arranca rama que NO usa l
11      I.saco(l)
12      máximo ← max(máximo,maxSubset(r+1,k,I,l+1,n)[2])
13      ret {I,máximo}
```

---

4. *RutaMinima*

Dada una matriz D de $n \times n$ números naturales, queremos encontrar una permutación π 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

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

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

b) Calcular la complejidad temporal y espacial del mismo.

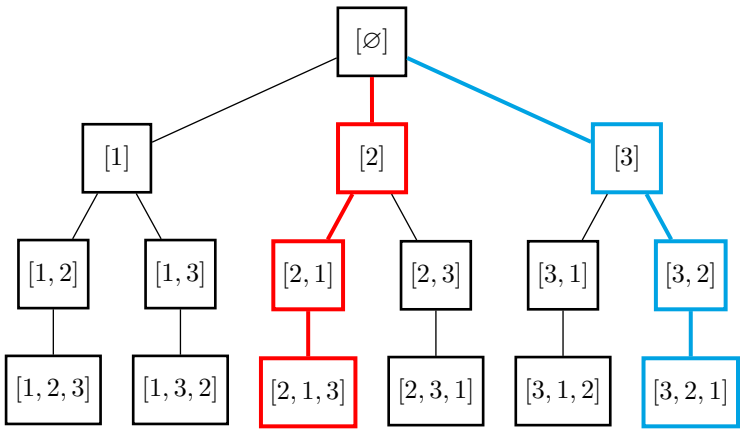
c) Proponer una poda por optimalidad y mostrar que es correcta.

a) Lo primero que hay que hacer es entender qué cosa piden. Luego resolver por *fuerza bruta, no trates de sacar un truco, ni de encontrar hack fantasma, ni fórmula cerrada ni un choto. Fuerza bruta y recursión*:

Las *soluciones candidatas* van a ser todas las funciones biyectivas:

$$\pi : \{1, \dots, n\} \to \{1, \dots, n\}.$$

En otras palabras hay que encontrar todas las permutaciones del conjunto $\{1, \dots, n\}$. Y qué mejor forma de hacer eso que por *fuerza bruta*. Para un conjunto de n elementos puedo encontrar un total de $n!$ funciones. Por ejemplo, $n = 3$:



Las hojas del árbol son los intervalos *candidatos* a ser la solución al problema. Por ejemplo las ramas marcadas en rojo y azul representa a las permutaciones candidatas:

$$\begin{cases} \pi(1) = 2 \\ \pi(2) = 1 \\ \pi(3) = 3 \end{cases} \quad \text{y} \quad \begin{cases} \pi(1) = 3 \\ \pi(2) = 2 \\ \pi(3) = 1 \end{cases}$$

Las soluciones *válidas* serán las soluciones candidatas π que minimicen la suma de $D_{\pi(n)\pi(1)} + \sum_{i=1}^{n-1} D_{\pi(i)\pi(i+1)}$, para las ramas rojas y azules del ejemplo:

$$\begin{aligned} D_{\pi(3)\pi(1)} + \sum_{i=1}^{n-1} D_{\pi(i)\pi(i+1)} &= D_{32} + D_{21} + D_{13} \\ D_{\pi(3)\pi(1)} + \sum_{i=1}^{n-1} D_{\pi(i)\pi(i+1)} &= D_{13} + D_{32} + D_{21} \end{aligned}$$

El algoritmo genera los intervalos π de forma recursiva, eligiendo todos los posibles caminos biyectivos (no repite números), devolviendo una tupla $(\pi, \sum)$. El camínimo mínimo minimizará el valor de $\sum$. Partiendo del conjunto $C = \{1, \dots, n\}$, voy a ir tomando valores de $c \in C$:

- Eliminándolos de C para no volver a usarlos: $C - \{c\}$.
- Concatenándolos a π. Uso el símbolo $\oplus$ para concatenar, dado que π tiene un orden.
- Caso base: Cuando la permutación está completa $C = \varnothing$ se devuelve $(\pi, \sum)$.

$$rutaMinima(\pi, C) = \begin{cases} (\pi, D_{\pi(n)\pi(1)} + \sum_{i=1}^{n-1} D_{\pi(i)\pi(i+1)}) & \text{si} \quad C = \varnothing \\ \min_{c \in C} \left(\; rutaMinima(\pi \oplus [c], C - \{c\}) \; \right) & \text{si no} \end{cases}$$

b) Mirando el árbol de generación de los intervalos veo que tengo un total de $n!$ hojas. El costo de resolver cada subproblema es $O(n)$ dado que tengo que sumar n elementos. Por lo tanto el *running time* es:

$$\boxed{T(n) \in O(n \cdot n!)}$$

El costo es sumar la cantidad de nodos eso es de orden $n!$, las hojas dominan.

La complejidad espacial la veo con las variables de estado, π y C. Como mucho π y $C \in O(n)$. Tengo ramas de recursión de profundidad n que es lo que toma generar un intervalo π la complejidad espacial.

$$S(n) \in O(\underset{\text{profundidad rama: estados}}{n} \cdot \overset{\text{costo } \pi}{n}) \in O(n^2)$$

Dependiendo la implementación, si se pasa por referencia las variables no aportarían un costo de $O(n)$, obteniéndose una complejidad espacial:

$$S(n) \in O(n)$$

c) No se me ocurre nada con esta implementación consultar.

---

5. *Palabras en cadena*

Dada una cadena de letras sin espacios o puntos queremos analizar si se puede subdividir de forma de obtener todas palabras válidas. Suponiendo que se tiene una función *palabra* que verifica si una cadena c de letras es una palabra en $O(|c|)$.

a) Dar una función recursiva que resuelva el problema.

b) Calcular una cota superior para la complejidad. **Ayuda:** Calcular cantidad de llamadas a *palabra*.

c) Demostrar que el algoritmo es correcto.

Ejemplo:

TDAeslamejormateriadelDC

Se puede subdividir en

TDA|es|la|mejor|materia|del|DC

Mientras que

nosvemoc

No tiene una subdivisión válida.

Como siempre hay que pensar esto de forma *recursiva y fuerza bruta*. Hay que devolver un *booleano*.

a) La función *palabra*(i, j) se encarga de ver si la cadena de caractéres de $[i, j]$ es una palabra o no, así que no hay que pensar en eso. Hay que pensar en recorrer todo lo que se pueda sin dejar títere con cabeza.

La función *divisible*(i), toma un valor i y devuelve un *booleano* si la cadena de caracteres $A[i..n]$ es divisible en palabras. Al operador disyunción me gusta pensarlo como si fuera un *existe:* $\exists$, después de todo es lo mismo, pero bueh, la idea es que:

- Cuando corro *divisible*(i) y encuentro una división entre $[1..i]$. Genial!
- Quiero ahora correr *palabra*$(j > i)$ para ver si encuentro otra división entre $[1..i+1]$ o en $[i+1..j]$
- Encontrar una palabra no quiere decir encontrar una división, por ejemplo, "LARVA" tiene una palabra entre [1..2] pero no la podés dividir ahí, porque "RVA" que yo sepa no es una palabra. Entonces tenés que tener tu *rama recursiva* donde no dividís a pesar de haber encontrado una palabra, así encontrando una palabra entre [1..5].

Bueh, algo así. Si existe división entre i, j, luego necesito que también exista una división (inductivamente hablando) entre $(j+1)$ y n.

$$\mathit{divisible}(i) = \begin{cases} \mathit{true} & \text{si} \quad i > n \\ \bigvee_{j=i}^{n} \left(\mathit{palabra}(i,j) \wedge \mathit{divisible}(j+1)\right) & \text{si no} \end{cases}$$

b) Acá viene el truco para usar en ejercicios donde la llamada recursiva está en un loop o una sumatoria o como en este caso en un operador disyunción que tiene $n - i + 1 \in \underbrace{O(n)}_{\text{ponele}}$ términos en cada llamado recursivo, por lo tanto en cada llamada se llama $n - i + 1$ veces a *palabra* que tiene un costo de $O(n)$.

$$T(n) = \sum_{j=i}^{n} T(i) + O(n)$$

Busco llevar esto a una expresión manejable. Truquito $O(n) \approx \alpha \cdot n$:

$$\begin{array}{rcl} T(n) & \leq & \sum_{i=0}^{n-1} T(i) + \alpha \cdot n \\ - & & \\ T(n-1) & \leq & \sum_{i=0}^{n-2} T(i) + \alpha \cdot (n-1) \\ \hline T(n) - T(n-1) & \leq & T(n-1) + \alpha \end{array} \implies T(n) \leq 2T(n-1) + \alpha \implies \boxed{T(n) \in O(2^n)}$$

Pero si cada subproblema tiene un costo de *palabra*(n)

$$\boxed{T(n) \in O(\mathit{costo}(\mathit{palabra}(n)) \cdot 2^n)}$$

Y acá un pseudocódigo en imperativo para pasarlo a tu lenguaje favorito un poco más friendly (si tu lenguaje favorito es Haskell, hacete ver) de la primera implementación:

```
1 funcion divisible(i)
2     si i > n
3         ret true
4     para j←i..n
5         si palabra(i,j)
6             si divisible(j+1)
7                 ret true
8     ret false
```

c) Demostrar que el algoritmo es correcto en estos ejercicios no requiere ir tan a bajo nivel con *triplas de Hoare* y esas 💩. Hay que probar que la función recursiva funciona. La función recursiva es como en *álgebra I* lo era las sucesiones dadas por recursión, sucesiones que para probar su recurrencia a veces uno "*intuía*" la fórmula cerrada y luego probaba por inducción.

Es lo que hay que hacer acá, solo que en vez de hacer cuentas con numeritos, las *"cuentas"* son con la lógica del algoritmo.

Inducción en la longitud del "*string*" n: Quiero probar que el siguiente predicado:

$p(n)$: Todo *string* S, $|S| = n$ que toma *divisible* retorna *true* si se puede dividir o *false* si no.

es verdadero.

- *Caso base:*

  $p(0)$: Todo *string* S, $|S| = 0$ que toma *divisible* retorna *true* si es divisible o *false* si no.

  No hay mucho que hacer, cae directamente en la guarda $\mathit{divisible}(1)_{n=0} \to \mathit{true}$. Por definición es divisible.

- *Paso inductivo:*
  Asumo que $\forall k \geq 1$ la proposición:

$$p(k): \underbrace{\text{Todo } \textit{string } S, |S| \leq k \text{ que toma } \textit{divisible} \text{ retornará } \textit{true} \text{ si es divisible o } \textit{false} \text{ si no.}}_{\textit{hipótesis inductiva}}$$

  es verdadera. Entonces quiero probar que

$$p(k+1): \text{ Todo } \textit{string } S, |S| \leq k+1 \text{ que toma } \textit{divisible} \text{ retornará } \textit{true} \text{ si es divisible o } \textit{false} \text{ si no.}$$

  también lo sea.

  Tengo 2 posible resultados *true* o *false*.

  true:
  En el caso de obtener *true* debe ocurrir que exista algún prefijo $S[i..j]$ tal que sea $\textit{palabra}(i,j) = \textit{true}$ y un sufijo $S[(j+1)..(k+1)]$ que sea divisible. Esto último debe ocurrir, porque sé que *divisible* funciona *(like a charm)* para $|S| \leq k$ por *hipótesis inductiva* y bueh $|S[(j+1)..(k+1)]| \leq k$.

  false:
  En el caso de obtener *false* debe ocurrir que alguno, el prefijo $S[i..j]$ o el sufijo $S[(j+1)..(k+1)]$ sea *false*. Si $\textit{palabra}(i,j) \to \textit{false}$ listo, la conjunción es *false*. Por otro lado si $\textit{palabra}(i,j) \to \textit{true}$ debe ocurrir que $\textit{divisible}(j+1) \to \textit{false}$ para que la conjunción dé *false*. Y al igual que antes por *hipótesis inductiva*, $\textit{divisible}(j+1)$ funca bien porque $|S[(j+1)..(k+1)]| \leq k$ así que vamos a obtener ese sabroso *false*.

  De esa mostrando que $p(k+1)$ resulta verdadera.

Por principio de inducción $p(n)$ es verdadera para cualquier $\mathbb{N}$, por lo tanto *divisible* te tira la posta sobre la divisibilisitatttt de un *string* de longitud n.

---

6. *Árboles binarios de búsqueda óptimos*

Dado un conjunto de elementos $[n] = \{1, \ldots, n\}$, y una función $f : [n] \to \mathbb{N}$ que nos da la frecuencia de acceso a dichos elementos, decimos que A es un árbol binario de búsqueda óptimo si este minimiza el costo de todos los accesos dados por f.

a) Escribir una función recursiva que devuelva el costo de acceder a todos los elementos usando f.

b) Dar una cota superior para la complejidad. **Ayuda**: Pasar de la función recursiva a una recurrencia que solo dependa del tamaño de la entrada.

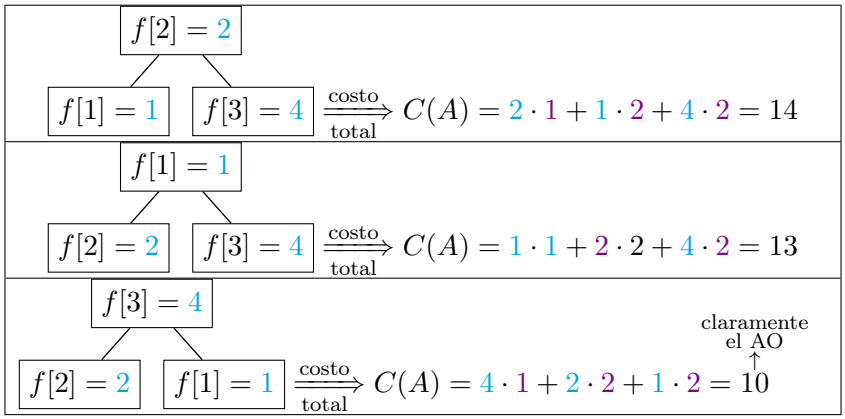
c) Probar que el algoritmo es correcto.

Segunda clase de TDA (ex Algo III) y te dan este ejercicio. *Tienen 30 minutos para pensarlo.*
Ni con 30hrs lo sacaba. En fin.

¿Qué es esto? ¿Qué me están pidiendo que haga? ¡Ayuda por favor!

Para un arreglo $A[1,2,3]$ con frecuencias $f[1,2,4]$, puede formar los siguiente árboles y calculor los costos de

recorrerlos yendo a cada nodo desde la raíz $f[i]$:



Los números en violeta son los ancestros que tiene cada nodo, definiendo que cada nodo es también ancestro de sí mismo, por eso la raíz tiene un (1) ancestro también.

La cuenta para calcular el costo es:

$$C(A) = \sum_{i=1}^{n} f[i] \cdot \#\text{ancestros\_de\_nodo}_i$$

Esto nos da una idea de lo que hay que hacer. El asunto se reduce a ¿Cómo recorro el árbol de forma recursiva para encontrar el árbol mínimo?

Acá es donde el nodo *raíz* toma relevancia, es el único nodo que va a tener 1 solo ancestro, quiere decir que va a ser un *peaje obligado* para cualquier nodo que quiera buscar. Eso no sucede con ningún otro nodo. Eso nos "dice" (entre muchas comillas) que hay una asimetría en la busqueda que sucede o bien a la derecha o a la izquierda. Reescribo la fórmula del costo como:

$$\begin{aligned} C(A) = \sum_{i=1}^{n} f[i] \quad &+ \quad \sum_{i=1}^{r-1} f[i] \cdot \#\text{ancestros\_de\_nodo}_i\text{\_en\_}A_{izq} \\ &+ \quad \sum_{i=r+1}^{n} f[i] \cdot \#\text{ancestros\_de\_nodo}_i\text{\_en\_}A_{der} \end{aligned}$$

La primera sumatoria son los *peajes* del nodo raíz, la segunda los de la rama izquierda y la tercera los de la derecha.

Oka, queda claro como calcular el costo del árbol, pero en el ejercicio no me dan un árbol, me dan un arreglo $A[1..n]$ con frecuencias $f[f_1..f_2]$ y yo tengo que hacer un ABBO con eso. ¿Cómo hago? ¿Cuál es el elemento raíz?

Ahí entra la magia de la recursión usando la definición del costo usada más arriba:

$$\begin{aligned} C(A) = \sum_{i=1}^{n} f[i] \quad &+ \quad C(A_{izq}) \\ &+ \quad C(A_{der}) \end{aligned}$$

a) Si bien no sé cuál es el nodo raíz $A[r]$ hago recursión pasando por el árbol probando cada nodo como raíz:

$$\textit{arbolOptimo}(i,j) = \begin{cases} 0 & \text{si} \quad i > j \\ \sum_{r=i}^{j} f(r) + \min_{i \le r \le j} \left(\begin{array}{c} \textit{arbolOptimo}(i, r-1) \\ + \\ \textit{arbolOptimo}(r+1, j) \end{array} \right) & \text{si no} \end{cases}$$

b) Usando la ayuda del enunciado. Tengo 2 llamados recursivos por llamada a la función *arbolOptimo*:

$$T(n) \le \sum_{i=0}^{n-1} 2T(i) + g(n)$$

Mismo truco que ejercicio **5.**. $O(n) \leq \alpha \cdot n$:

$$\begin{array}{rcl} T(n) & \leq & \sum_{i=0}^{n-1} 2T(i) + \alpha n \\ - & & \\ T(n-1) & \leq & \sum_{i=1}^{n-2} 2T(i) + \alpha(n-1) \\ \hline T(n) - T(n-1) & \leq & 2T(n-1) + \alpha \end{array} \implies \boxed{T(n) \leq 3T(n-1) + \alpha \in O(3^n)}$$

c) Hacer!

---

7. *Dobra*

Dobra se encuentra con muchas palabras en su vida, como es una persona particular la mayoría de estas no le gustan. Para compensar empezó a inventar palabras más agradables. Dobra crea palabras nuevas escribiendo una cadena de caracteres que considera buena, luego borra los caracteres que peor le caen y los reemplaza con _. Luego para mejorar su vida intenta reemplazar estos guiones bajos con letras más aceptables intentando crear palabras más lindas. Dobra considera una palabra como buena si no contiene 3 vocales consecutivas, 3 consonantes consecutivas y al menos contiene una E.

a) Mostrar alguna solución candidata posible y alguna solución parcial.

b) Proponer una función recursiva y estimar su complejidad. Asumir que se tiene una función *verificar* que toma una cadena y devuelve *true* si es como Dobra quiere o *false* en caso contrario.

c) Probar que la función o programa es correcto.

d) Proponer al menos una poda por factibilidad.

e) Si *b)* no tiene una cota superior $O(3^n)$ para la complejidad, analizar el caso donde se separa la recursión en tener o no una letra E y ver si mejora la misma.

Ejercicio de conteo.

a) Una palabra con guiones bajos que *no tiene soluciones válidas*, porque no tendría la E, dado que de ponerla tendría 3 vocales consecutivas.:

$$AA\_ \to AA\underline{c} \xrightarrow{\forall c \in [a-z]} false$$

Una palabra que se puede mejorar sería $A\_\_$, la cual tiene $21 + 21$ soluciones candidatas. Número al que llego poniendo una E en el primer guión y cualquier consonante en el segundo o al revés.

$$A\_\_ \to \underbrace{A\underline{B}\_}_{\substack{\text{solución} \\ \textit{parcial}}} \to ABE \to true$$

b) Tengo una cadena de caracteres, *un string*, que tiene guiones bajos, hay que meter alguna letra ahí, entonces *fuerza bruta*. No sé donde están los guiones bajos, entonces pruebo todo, no sé que letra tengo que poner entonces pruebo todas, *fuerza bruta*! Si la letra funciona o no de eso se encargará la función *verificar*:

La siguiente función es un algoritmo que reemplaza los *guiones bajos* por algún caracter del intervalo $[a-z]$.

Se llama inicialmente con $i = 1$ y el caso base devuelve 0 o 1 una vez recorrido todo el *string*.

$$betterStrings(i, n) = \begin{cases} 1 & \text{si} \quad i > n \wedge verificar(S) \\ betterStrings(S[i], i+1) & \text{si} \quad i \leq n \wedge S[i] \neq \_ \\ \sum_{c \in [a-z]} betterStrings(S[i] \leftarrow c, i+1) & \text{si} \quad i \leq n \wedge S[i] = \_ \\ 0 & \text{si no} \end{cases}$$

¿Te gustó el nombre de la función?

Potencialmente tengo n guiones bajos. Por cada guión bajo tengo potencialmente 26 llamadas recursivas: El costo de cada subproblema es $O(n)$ porque tengo que revisar todo el *string*. Sea como sea:

$$T(n) = 26T(n-1) + n \implies T(n) \in O(n \cdot 26^n)$$

Esto podría afinarse un poco más, por ejemplo argumentando que el número debería ser 25, porque estás cambiando una letra, pero dado que no me pidieron que cumpla ningún número de complejidad lo dejo así.

c) Hacer!

d)
- Al trabajar en $S[i]$ verifico que $S[i-1]$ y $S[i+1]$ no sean ambas vocales o consonantes, ya que ahí la llamada recursiva sería de 21 caracteres o de 5 en lugar de siempre 26.

e) Hacer!

---

8. ***Cadenas de Adición***

Dado un entero n decimos que $C = \{x_1, \ldots, x_k\}$ es una cadena de adición si cumple lo siguiente

- $1 = x_1 < x_2 < \cdots < x_k = n$
- Para cada $2 \leq j \leq n$ existe $k_1, k_2 < j$ tal que $x_{k_1} + x_{k_2} = x_j$

a) Encontrar un algoritmo de backtracking que encuentre, si existe, la cadena de adición de longitud mínima.

b) Proponer, si no agregaron en el inciso anterior, al menos una poda por optimalidad y otra por factibilidad.

Ejemplos:

$$\begin{array}{rcl} n = 7 & \to & 1 \to 2(1+1) \to 3(2+1) \to 6(3+3) \to 7(6+1) \\ n = 15 & \to & 1 \to 2(1+1) \to 3(2+1) \to 6(3+3) \to 12(6+6) \to 15(12+3) \end{array}$$

a) Inicio el algoritmo con $(\{1,2\}, n-2)$

$$\mathit{cadenaAdicion}(C, n) = \begin{cases} (C, |C|) & \text{si} \quad \max\limits_{c \in C}(C) = n \\ (\varnothing, +\infty) & \text{si} \quad \max\limits_{c \in C}(C) > n \vee |C| > n \\ \min\limits_{c_1, c_2 \in C} \left(\mathit{cadenaAdicion}(C \cup (c_1 + c_2), n)\right) & \text{si no} \end{cases}$$

b) Una poda de optimalidad sería que el $c = c_1 + c_2$ que estoy agregando cumpla que $c > \max\limits_{c' \in C}(C)$.

---

9. *KingArmy*

El rey Cambyses está interesado en armar ejércitos en una serie de días consecutivos. Más aún, le interesa que el número de personas de su ejército en el dia d_i sea equivalente a la suma del número de personas del ejército que formó el dia d_{i-1} e d_{i-2}. La excepción para esto es en 0 (d_0) y 1 (d_1), en cuyo caso la cantidad de personas en esos días va a ser siempre 1. Para él es muy complicado determinar este número, entonces nos pidió que lo ayudemos. Dado un dia n (d_n), tenemos que devolver el número de personas de su ejército. Pensar un algoritmo $O(n)$ para resolver este problema y demostrar su correctitud y complejidad.

En el mundo de la programación dinámica primero hay que hacer *fuerza bruta*, después guardas cositas, porque en la *fuerza bruta* uno repite y repite muchos subproblemas y en vez de recalcularlos y así agregar complejidad temporal ahora los guardamos, haciendo un *trade* entre complejidad espacial y temporal. y los usamos cuando sea necesario.

Este ejercicio es literalmente *Fibonacci*. Un algoritmo de *backtracking, fuerza bruta*:

$$ejercito(i) = \begin{cases} 1 & \text{si} \quad i \leq 1 \\ ejercito(i-1) + ejercito(i-2) & \text{si no} \end{cases}$$

La complejidad temporal $T(n) \in O(2^n)$. Listo ya cumplimos con lo necesario para poder, recién ahora, empezar con la programación dinámica. ¿Por qué recién ahora? Porque observamos que tenemos 2^n llamados recursivos cuando la función de Fibonacci te devuelve como mucho 1 valor cada vez que la evaluás onda, imaginate si querés calcular $ejercito(n = 6)$:

$$\begin{cases} ejercito(0) = 1 \\ ejercito(1) = 1 \\ ejercito(2) = 2 \\ ejercito(3) = 3 \\ ejercito(4) = 5 \\ ejercito(5) = 8 \\ ejercito(6) = 13 \end{cases}$$

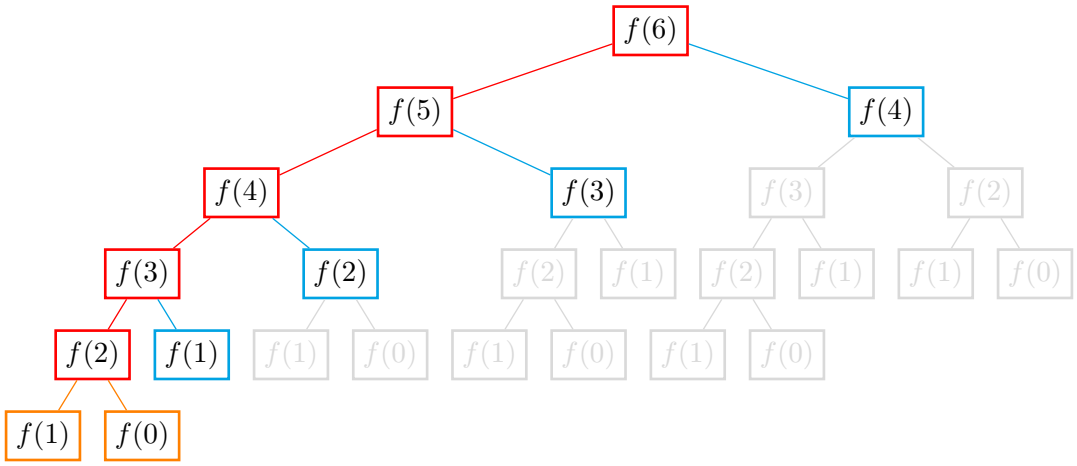
Hice solo $7 = n + 1$ cálculos y no $2^n = 64$: *Algo raro debe haber*. Sí, Superposición de Problemas o dicho de otra manera estoy haciendo llamadas recursivas repetidas al re pedo.

La función *ejercito* se puede escribir usando la técnica de memoization:

$$ejercito(i) = \begin{cases} memo[i] & \text{si} \quad memo[i] \neq 0 \\ 1 & \text{si} \quad i \leq 1 \\ memo[i] \leftarrow ejercito(i-1) + ejercito(i-2) & \text{si no} \end{cases}$$

Este árbol muestra en color:

- En rojo son los llamados a *ejercito* que tienen adentro más llamados recursivos.
- En azul son los llamados a *ejercito* que devuelven un valor de *memo*.
- En naranja son los llamados a *ejercito* que caen en la guarda de los casos base.
- En gris translúcido son los llamados que nunca se hicieron debido a la técnica de memoization



La función se puede escribir en pseudocógido más bonito para pasar a la compu así:

```
1  funcion ejercito(i, memo)
2    si memo[i] not null
3         ret memo[i]
4    si n=1 o n=0
5         ret 1
6    sino
7         memo[i] <- ejercito(n-1) + ejercito(n-2)
```

```
8         ret memo[i]
```

La complejidad de un algoritmo usando *programación dinámica* lo calculás haciendo:

$$T(n) = \texttt{cantidad\_de\_estados} \times \texttt{costo\_de\_estados}$$

El estado lo calculás viendo como son *las variables de estado*, las variables que te construyen el estado. En este caso hay una sola variable: i, que tiene una complejidad $i \in O(n)$.

¿Cuánto cuesta calcular cada subproblema en *ejercito*? Es constante: $O(1)$. Por lo tanto la complejidad temporal del algoritmo usando *programación dinámica*:

$$\boxed{T(n) \in O(1) \cdot O(n) \in O(n)}$$

Si bien el *running time* bajó mucho, la complejidad espacial, la memoria usada en la estructura de memoization aumentó:

$$\boxed{S(n) \in O(n)}$$

Ahí está el trade entre el tiempo y el espacio.

Y todavía tenés la versión aún más eficiente de este algoritmo:

```
ejercito(n)
    ult, penult ← 1
    mientras n>1
        actual ← ult+penult
        penult ← ult
        ult ← actual
        n ← n-1
    ret ult
```

Con una complejidad espacial constante.

10. *Vacations (link a codeforces, recomiendo leer el enunciado ahí click click)*
Tomás tiene n días de vacaciones, donde puede hacer actividades:

- Se pueden hacer 2 actividades: Gimnasio y competencias.
- Cada día puede tener disponible ambas, una o ninguna.

Tomás cada día puede

- Hacer una actividad que esté disponible, siempre que no la haya hecho el día anterior.
- Descansar.

Tomás quiere minimizar los días de descanso.

a) Diseñar un algoritmo $O(n)$ que calcule la mínima cantidad de días de descanso.

b) Probar la correctitud y complejidad del algoritmo.

c) Indicar cómo se puede reconstruir la solución. Es decir, indicarle a Tomás qué hará cada día.

Ayuda: Consideren el siguiente ejemplo:

- $n = 4$
- Días con gimnasio disponibles: 2, 3
- Días con competencias disponibles: 1, 2

Puede lograr tener solo 2 días de descanso.

Acá empiezan los enunciados que tengo que leer en $O\left(\lim_{n\to\infty} n^{n^{n^n}}\right)$ veces para entender. Así que sabé que *no estás solo.*

Si el enunciado está en *Codeforces* leerlo de ahí ayuda. Ver el **input** y el **output** de algún ejemplo te puede dar una idea también.

a) Una vez que tengas una idea de lo que están pidiendo lo que hay que hacer es usar *fuerza bruta para armar algoritmos, funciones* igual que en backtracking. Que la función esté descrita con índices *sin mucha abstracción* facilita luego la parte de programación dinámica.

La siguiente función recibe un arreglo *dias*[1..n] con la información sobre cuáles *actividades* (gimnasio, competencia o descanso) que están disponibles en el día i. El algoritmo decide como armar un *camino de recursiones* con la menor cantidad de días de descanso posibles.

La función tiene 2 variables de estado, i y *prev*, el día actual y la actividad del día anterior respectivamente. El caso base se alcanza una vez recorrido todo el arreglo de n*dias*. Llamo a la función con *vacations*$(1, D)$,

conteo *1-indexed* y al poner D independizo el valor de la actividad del día siguiente a *prev*.

$$vacations(i, prev) = \begin{cases} 0 & \text{si} \quad i > n \\ \text{mín} \begin{pmatrix} 1 + vacations(i+1, D), \\ 0 + vacations(i+1, G) \end{pmatrix} & \text{si} \quad prev \neq G \wedge disponible(i, G) \wedge \neg disponible(i, C) \\ \text{mín} \begin{pmatrix} 1 + vacations(i+1, D), \\ 0 + vacations(i+1, C) \end{pmatrix} & \text{si} \quad prev \neq C \wedge disponible(i, C) \wedge \neg disponible(i, D) \\ \text{mín} \begin{pmatrix} 1 + vacations(i+1, D), \\ 0 + vacations(i+1, C), \\ 0 + vacations(i+1, G) \end{pmatrix} & \text{si} \quad prev \notin \{C, G\} \wedge disponible(i, C) \wedge disponible(i, D) \\ 1 + vacations(i+1, D) & \text{si no} \end{cases}$$

donde la función auxiliar *disponible* me devuelve un *booleano* según la disponibilidad de actividades de el día $i-$ésimo que viene en *días*, input que me dan en el ejercicio que paso por referencia o lo pienso como una variable global.

$$disponible(i, actividad) = \begin{cases} true & \text{si} \quad actividad \in dias[i] \\ false & \text{si no} \end{cases}$$

La función *vacations* tiene un *running time* exponencial $\Omega(2^n)$. La variable i decrece linealmente y hay por lo menos 2 llamadas recursivas por cada llamada a *vacations*, excepto que nunca pueda hacer ninguna *actividad* entonces descanso todos los días, lo cual sería $\Theta(n)$, pero supongo que no es ese caso particular tan extremo.

A medida que se va armando la recursión se pasa por $O(n)$ estados, los n valores que puede tomar el problema con las variables de estado i y *prev*.

Claramente hay superposición de problemas, lo cual me habilita a usar *programación dinámica*. Con la técnica de memoization del enfoque *top-down* puedo usar un arreglo de $n \times 3$ por ejemplo donde cada elemento tiene el resultado correspondiente al estado del día i y actividad previa *prev*.

Es igual a lo del ejercicio **9.** Meto los resultados en la estructura y la uso si vuelvo a llamar a la función con esas mismas *variables de estado*, así evitando repetir cálculos al pedo, podando muchas llamadas recursivas.

```
1  funcion vacations(i,prev)
2      si memo[i,prev] not null
3          ret memo[i,prev]
4      si i>n
5          ret 0
6      si (prev not G) y disponible(i,G) y (not disponible(i,C))
7          memo[i,prev] ← min(1+vacations(i+1,D), 0+vacations(i+1,G))
8      si (prev not C) y disponible(i,C) y (not disponible(i,G))
9          memo[i,prev] ← min(1+vacations(i+1,D), 0+vacations(i+1,C))
10     si (prev not en {C,G}) y disponible(i,C) y disponible(i,G)
11         memo[i,prev] ← min(1+vacations(i+1,D), 0+vacations(i+1,C),0+vacations(i+1,G))
12     sino
13         memo[i,prev] ← 1+vacations(i+1,D)
14
15     ret memo[i,prev]
16
17 funcion disponible(i, actividad)
18     si actividad en dias[i]
19         ret true
20     sino
21         ret false
```

b) Te debo la correctitud! Hacer!

Las variables de estado $i, prev$ varían entre $[1..n]$ y $[D, G, C]$ respectivamente: Puedo formar con eso un total de $3 \cdot n$ estados. La complejidad temporal de un algoritmo usando *programación dinámica* es:

$$T(n) = \texttt{cantidad\_de\_estados} \times \texttt{costo\_de\_estados} \rightarrow 3 \cdot n \times 1 \in O(n)$$

La complejidad espacial es la profundidad de la rama de recursión.

$$S(n) \in O(n \cdot 3)$$

Complejidad temporal o *running time* del algoritmo es:

$$\boxed{T(n) \in O(n)}$$

La complejidad espacial:

$$\boxed{S(n) \in O(n)}$$

c) Puedo modificar el algoritmo para que no solo me dé el mínimo de días de descanso en el día i sino también cuál actividad de realizó ese día. Ese resultado lo puedo guardar en otra estructura `plan[i,prev]` algo así:

```
1  funcion vacations(i,prev)
2      ...
3          ret 0
4
5      si (prev not G) y disponible(i,G) y (not disponible(i,C))
6          GTemp ← 0+vacations(i+1,G)
7          DTemp ← 1+vacations(i+1,D)
8          si GTemp≤DTemp
9              memo[i,prev] ← GTemp
10             plan[i,prev] ← G
11         sino
12             al revés
13     ...
```

Los valores de `plan` me dan el plan de actividades en los n días a seguir para minimizar los descansos.

11. *SumaDinámica*
En este ejercicio vamos a resolver el problema de suma de subconjuntos usando la técnica de programación dinámica.

a) Sea $n = |C|$ la cantidad de elementos de C. Considerar la siguiente función recursiva $ss'_C : \{0,\dots,n\} \times \{0,\dots,k\} \to \{true, false\}$ tal que:

$$ss'_C(i,j) = \begin{cases} j = 0 & \text{si} \quad i = 0 \\ ss'_C(i-1,j) & \text{si} \quad i \neq 0 \wedge C[i] > j \\ ss'_C(i-1,j) \vee ss'_C(i-1,j-C[i]) & \text{si no} \end{cases}$$

Convencerse de que esta es una definición equivalente de la función ss del inciso *e*) del ejercicio **1.**, observando que $ss(C,k) = ss'_C(n,k)$. En otras palabras, convencerse de que el algoritmo del inciso *f*) es una implementación por *backtracking* de la función ss'_C. Concluir, pues, que $O(2^n)$ llamadas recursivas de ss'_C son suficientes para resolver el problema.

b) Observar que, como C no cambia entre llamadas recursivas, existen $O(nk)$ posibles entradas para ss'_C. Concluir que, si $k \ll 2^n/n$, entonces necesariamente algunas instancias de ss'_C son calculadas muchas veces por el algoritmo del inciso *f*). Mostrar un ejemplo donde se calcule varias veces la misma instancia.

c) Considerar la estructura de memoización (i.e., el diccionario) M implementada como una matriz de $(n+1) \times (k+1)$ tal que $M[i,j]$ o bien tiene un valor indefinido $\perp$ o bien tiene el valor $ss'_C(i,j)$, para todo $0 \le i \le n$ y $0 \le j \le k$. Convencerse de que el siguiente algoritmo *top-down* mantiene un estado válido para M y computa $M[i,j] = ss'_C(i,j)$ cuando se invoca $ss'_C(i,j)$.

```
1 M[0..n][0..k] ← ⊥   // inicializar todos los elementos de M como indefinido o NULL.
2 funcion subset_sum(C,i,j)          // implementa ss({c1,...,ci},j) = ss'C(i,j) usando memoización
3     si j<0
4         ret false
5     si i=0
6         ret j=0
7     si M[i,j] = ⊥
8         M[i,j] ← subset_sum(C,i-1,j) o subset_sum(C,i-1,j-C[i])
9     ret M[i,j]
```

d) Concluir que *subset_sum* (C,n,k) resuelve el problema. Calcular la complejidad y compararla con el algoritmo *subset_sum* del inciso *f*) del ejercicio **1.** ¿Cuál algoritmo es mejor cuando $k \ll 2^n$? ¿Y cuándo $k \gg 2^n$?

e) Supongamos que queremos computar todos los valores de M. Una vez computados, por definición, obtenemos que

$$M[i,j] \overset{\text{def}}{=} ss'_C(i,j) \overset{\text{ss'}}{=} ss'_C(i-1,j) \vee ss'_C(i-1,j-C[i]) \overset{\text{def}}{=} M[i-1,j] \vee M[i-1,j-C[i]]$$

cuando $i > 0$, asumiendo que $M[i-1,j-C[i]]$ es *false* cuando $j - C[i] < 0$. Por otra parte, $M[0,0]$ es *true*, mientras que $M[0,j]$ es *false* para $j > 0$. A partir de esta observación, concluir que el siguiente algoritmo *bottom-up* computa M correctamente y, por lo tanto, $M[i,j]$ contiene la respuesta al problema de la suma para todo $\{c_1,\dots,c_i\}$ y j.

```
1 funcion subset_sum(C,k)          // computa M[i][j]  ∀i ∈ [0,n]  y  j ∈ [0,k]
2     para j←0..k                  // inicializa la primera fila de M
3         M[0,j] ← (j=0)   // boolean, wuachín.
4     para i ← 1..n
5         para j ← 0..k
6             M[i,j] ← M[i-1,j] o (j-C[i]≥0 y M[i-1,j-C[i]])
7     ret M[i,j]
```

f) (Opcional) Modificar el algoritmo *bottom-up* anterior para mejorar su complejidad espacial a $O(k)$.

g) (Opcional) Demostrar que la función recursiva del inciso *a*) es correcta. **Ayuda**: Demostrar por inducción en i que existe algún subconjunto de $\{c_1,\dots,c_i\}$ que suma j si y solo si $ss'_C\ (i,j) = V$.

Me tomé la libertad de editar a mi gusto el enunciado. La pedorra edición del enunciado, complica la lectura y le agrega ruido a algo que de por sí ya es *complicado*. Me la baja que se compliquen los ejercicios con cosas evitables.

Hay que ponerle voluntad, gente. Esto no es un invento mío literalmente se gastan recursos cognitivos al recontra pedo con estos enunciados.
Y lo peor es que creo que este ejercicio es, al igual que el ejercicio 1, de lo mejor de la guía.
¿Por qué no es este ejecicio el primero de *programación dinámica* 😳? En fin, mi nivel de *viejoChotismo* está en fase IDDQD, IDKFA.

a) En el ejercicio **1.***e*) escribí una función análoga. Solo que pasándole parámetros distintos. La implementación de este ejercicio ss'_C con los índices i, j es *mucho más mejor* para usar programación dinámica.

La función se llama con los valores $ss'_C\ (n, k)$:

- El caso base se da cuando $i = 0$, es decir se recorrió todo el conjunto en la variable i, y se devuelve el booleano $j = 0$, el cual si resultó *true* es que se pudo encontrar el subconjunto de C de suma k.
- La parte del medio es para iterar en caso de que $k < n$ para llegar al caso base.
- La tercera parte es la decisión *fuerza brutezca* que recorre la posibles opciones en caja elemento de C, así generando el árbol de recursiones.

El *running time* de esa función es $O(2^n)$, dado que hay un laburo:

$$T(n) = 2T(n-1) + 1 \in O(2^n)$$

b) Los parámetros $\begin{cases} i \in [0, n] \\ j \in [0, k] \end{cases}$ así que la función tendrá los estados que esas *variables de estado* puedan formar, un total de $O(n \cdot k)$.

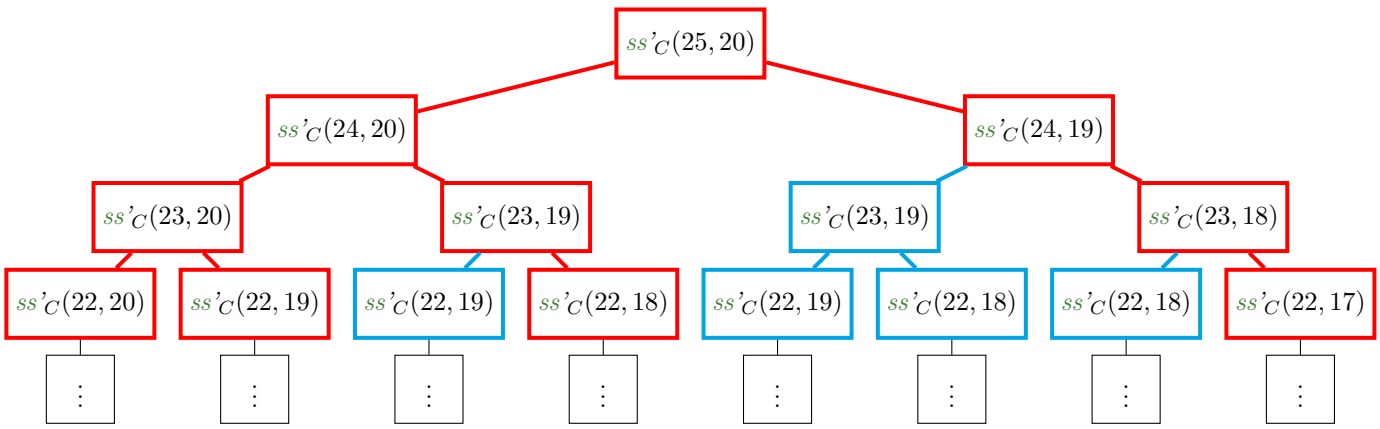
La comparación entre estados y llamadas recursivas es el argumento con el cual se justifica el uso de la programación dinámica:

$$k \ll 2^n/n \Leftrightarrow n \cdot k \ll 2^n$$

Eso quiere decir que voy a estar llamando a ss'_C con *parámetros idénticos* muchas veces (generando respuestas idénticas) es como tener un bolillero con 100 números y sacás una bolita, luego la metés nuevamente en el bolillero y volvés a sacar una bolita y si hacés eso más de 100 veces, es seguro que vas a sacar una bolita repetida ¿Quién lo hubiese pensado, no? Nuevamente:

🖩 Si a una función le metés en la iteración l los parámetros i y j, y después en la iteración $l + 100$ le volvés a meter los mismos parámetros, te va a devolver lo meeeeesmo, *thus dynamic programming, moderfoca!* 🖩

Ponele que tenés el conjunto $C = \{1, 1, 1, 1, 2, 2, 2, 2, 3, 3, \ldots, 1\}$, $|C| = n = 25$ y $k = 20$



- En rojo son nodos que, de una implementación, corresponden a llamar a la función con las variables de estado i, j en cuestión por primera vez.

- En azul son nodos que corresponden a llamados que calcularon previamentes.
- El árbol tendrá cantidad de nodos en el orden de $2^{25} \approx 3 \cdot 10^7$, pero solo habrá $25 \cdot 20 = 500$ valores distintos entre ellos, que son las posibles combinaciones entre i, j.

c) El algoritmo llena el diccionario `M[i,j]` con valores válidos que retorna al función *subset_sum* (i,j)

d) *Algoritmo sin memoization*:

El algoritmo del ejercicio **1.** tenía una complejidad temporal exponencial de $O(2^n)$ y como el árbol tiene una profundidad de n tiene una $S(n) \in O(n)$.

Algoritmo con memoization:

Sí, el pesudocódigo llena los elementos `M[i,j]` con valores booleanos que corresponden a las llamadas con las variables `i,j` y devuelve el valor `M[n,k]`. La complejidad temporal de un algoritmo usando *programación dinámica* es:

$$T(n) = \texttt{cantidad\_de\_estados} \times \texttt{costo\_de\_estados} \rightarrow n \cdot k \times 1 \in O(n \cdot k)$$

La complejidad espacial, las variables de estado varían entre i, j varían entre $[0..n]$ y $[0..k]$ respectivamente:

$$S(n) \in O(n \cdot k)$$

¿Qué pasa ahora cuando tengo valores límites respecto al parámetro k?

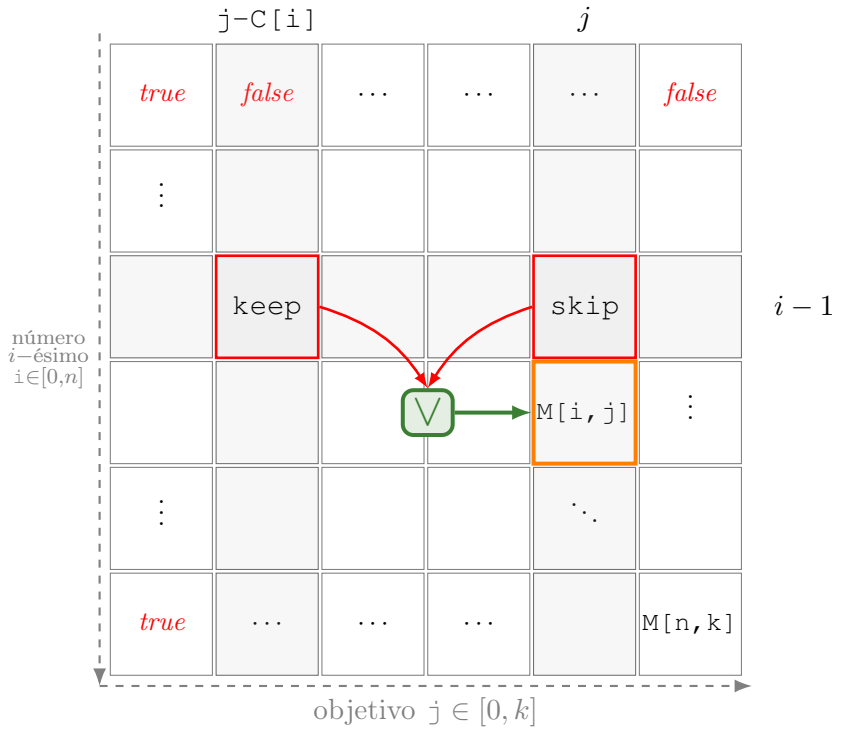
$k \ll 2^n$:

El que haya un k chico en comparación a la cantidad de hojas del árbol, implica que va a haber mucha repetición de estados. Acá es dónde donde la memoization funcionaría mejor.

$k \gg 2^n$:

En este caso la recursión podría llegar a las hojas sin una gran repetición de estados. No obstante la *complejidad espacial* va a ser exponencial $\Omega(n2^n)$ en programación dinámica. Usaría *backtracking* para que no se cuelgue toda la compu.

e) Lindo renglón esa verga. Para nada confuso 😳. La posta en *bottom-up* es entender la relación entre los elementos ★[1]:



Donde `keep` es cuando agarro el número `C[i]` y `skip` donde no.

El `loop externo` me mueve las filas, el `loop interno` las columnas. Lleno fila a fila, porque el estado `i` siempre depende del estado de la fila anterior `i-1`.

El ejercicio implementado con la técnica *bottom-up* no solo resuelve `M[n,k]` sino que cada valor `M[i,j]` es una respuesta para el problema con valores objetivos `i,j`, es decir $M[3,5]$ es el resultado para un conjunto $|C| = 3$ y $k = 5$

f) En este ejercicio (y en muchos va a pasar lo mismo), el elemento que estoy calculando no depende de cualquier fruta sino que es hay un patrón:

- `M[i,j]` siempre depende de la fila `i-1`.
- `M[i,j]` nunca va a ser calculado con estados que tengan `j` entre `[j+1,k]`, porque en cada cálculo o bien uso el mismo `j` o les resto un número natural, i.e. positivo.

¡Esos datos son clave para ir *tirando a la basura* valores que sé que no voy a volver a usar para el cálculo objetivo!

Si mirás la primera columna de la matriz ★[1] eso se llena con *true*, porque cada valor debajo de `M[0,0]`=*true* es el resultado de la disjunción entre `M[0,0]` y ... una chota, porque no tengo `j<0`! Y eso debería ser razonable, porque para sumar un total de `j=k=0` lo que hacés es *no agarrar ningún valor* son todos *skips*.

```
1 funcion subset_sum(C,k)
2     para j←k..0
3         M[j] ← (j=0)   // inicializo iteración 0.
4     para i←1..n
5         para j←k..1 // acá está la papa!
6             M[j] ← M[j] o (j-C[i]≥0 y M[j-C[i]])
7     ret M[k]
```

El algoritmo hace lo mismo que estaba haciendo el anterior, solo que se va reescribiendo en la misma fila.

¿Pero por qué mierda no pierdo valores importantes? Por lo que se observó sobre las dependencias de los estados. Clave el orden de llenado escribo la iteración `i`-ésima empezando el pisado de la iteración anterior por `j = k`, `M[j]` no se usa en el cálculo de `M[j']` con `j'<j` como se mencionó previamente.

Y ahora ese algoritmo tiene una complejidad espacial:

$$S(n) \in O(k)$$

g) ¡Uy! Justo se largó a llover y tengo que sacar la ropa del tender, te la debo. Hacer!

12. *OptiPago*

Tenemos un multiconjunto B de valores de billetes y queremos comprar un producto de costo c de una máquina que no da vuelto. Para poder adquirir el producto debemos cubrir su costo usando un subconjunto de nuestros billetes. El objetivo es pagar con el mínimo exceso posible a fin de minimizar nuestra pérdida. Más aún, queremos gastar el menor tiempo posible poniendo billetes en la máquina. Por lo tanto, entre las opciones de mínimo exceso posible, queremos una con la menor cantidad de billetes. Por ejemplo, si $c = 14$ y $B = \{2, 3, 5, 10, 20, 20\}$, la solución es pagar 15, con exceso 1, insertando sólo dos billetes: uno de 10 y otro de 5.

a) Considerar la siguiente estrategia por *backtracking* para el problema, donde $B = \{b_1, \ldots, b_n\}$. Tenemos dos posibilidades: O agregamos el billete b_n, gastando un billete y quedando por pagar $c - b_n$, o no agregamos el billete b_n, gastando 0 billetes y quedando por pagar c. Escribir una función recursiva $cc(B, c)$ para resolver el problema, donde $cc(B, c) = (c', q)$ cuando el mínimo costo mayor o igual a c que es posible pagar con los billetes de B es c' y la cantidad de billetes mínima es q.

b) Implementar la función de *a*) en un lenguaje de programación imperativo utilizando una función recursiva con parámetros B, i, j que compute $cc(\{b_1, \ldots, b_i\}, j)$. ¿Cuál es la complejidad del algoritmo?

c) Reescribir cc como una función recursiva $cc'_B(i, j) = cc(\{b_1, \ldots, b_i\}, j)$ que implemente la idea anterior **dejando fijo el parámetro** B. A partir de esta función, determinar cuándo cc'_B tiene la propiedad de *superposición de subproblemas*.

d) Definir una estructura de memoización para cc'_B que permita acceder a $cc'_B(i, j)$ en $O(1)$ tiempo para todo $0 \leq i \leq n$ y $0 \leq j \leq k$.

e) Adaptar el algoritmo de *b*) para incluir la estructura de memoización.

f) Indicar cuál es la llamada recursiva que resuelve nuestro problema y cuál es la complejidad del nuevo algoritmo.

g) (Opcional) Escribir un algoritmo *bottom-up* para calcular todos los valores de la estructura de memoización y discutir cómo se puede reducir la memoria extra consumida por el algoritmo.

h) (Opcional) Formalmente, en este problema de vuelto hay que computar el mínimo $(\Sigma V, |V|)$, en orden lexicográfico, de entre los conjuntos $V \subset B$ tales que $\Sigma V \geq c$. Demostrar que la función cc' es correcta. **Ayuda**: Demostrar por inducción que $cc'(i, j) = (v, k)$ para el mínimo (v, k) tal que existe un subconjunto V de $\{b_1, \ldots, b_i\}$ con $\Sigma V \geq j$.

---

a) La estrategia que ofrecen es la misma que se viene usando en todos los algoritmos de *backtracking*, pruebo con/sin cada billete. Cuando el costo restante es negativo o cero, es el caso base donde paro de contar.

Si no llego a pagar todo, es decir me quedo sin billetes, devuelvo un valor para que nunca sea una respuesta posible.

La función devuelve una tupla (exceso, billetes). A la función mín le doy tremendo laburo: Primero busca que sea mínimo el exceso de perdida y en caso de que distintas ramas tengan el mismo exceso luego busca para despempatar la mínima cantidad de billetes entre esas ramas empatadas.

$$cc(B, c) = \begin{cases} (-c, 0) & \text{si} \quad c \leq 0 \\ (+\infty, +\infty) & \text{si} \quad B = \varnothing \\ \min\limits_{1^{er} \to 2^{da}} (cc(B \setminus \{b_n\}, c), (0, 1) + cc(B \setminus \{b_n\}, c - b_n)) & \text{si no} \end{cases}$$

b) Muy parecida a la función anterior. Llamo a la función con los parámetros $cc(B, |B|, j = c)$. Acá se ve la función que tiene mín en detalle. Primero compara la primera coordenada de excesos y luego en caso de haber un empate la segunda de billetes.

La variable i es el índice que se usa para recorrer B.

```
1  funcion cc(B,i,j)
2     si j≤0
3          ret (-j, 0)
4     si i=0
5          ret (+∞,+∞)
6     sino
7          skip ← cc(B, i-1, j)
8          keep ← (0,1) + cc(B, i-1, j-B[i])
9
10         si keep.exceso < skip.exceso              // comparo 1era
11              ret keep
12         sino si keep.exceso > skip.exceso         // comparo 1era
13              ret skip
14         sino                                     //son iguales comparo cantidad de billetes
15              si keep.billetes < skip.billetes
16                   ret keep
17              sino
18                   ret skip
```

El algoritmo es exponencial. En cada llamado recursivo puede tener hasta 2 nuevas llamadas recursivas. El costo de cada subproblema es $O(1)$.

$$\boxed{T(n) \in O(2^n)}$$

c) La orientación del ejercicio va hacia *generar mejores funciones recursivas*, mejores en el sentido de que al usar índices i, j en vez de conjuntos B hace que sea más fácil el pasaje a *programación dinámica*.

Puedo pensar que B es una variable global, tengo una referencia, así que no carga la memoria ni nada *innecesariamente ineficiente*, entonces la uso, aunque no voy a modificarla. i va a ser la variable con la que recorro B y j es el *costo objetivo*.

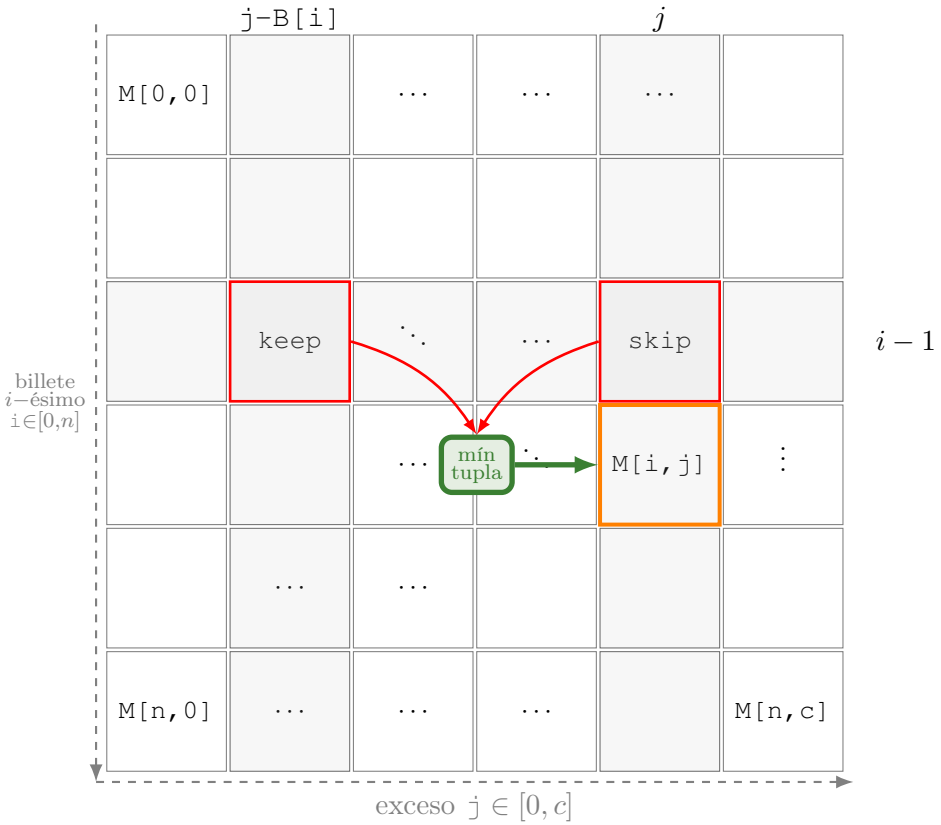
$$cc'_B(i,j) = \begin{cases} (-j, 0) & \text{si} \quad j \leq 0 \\ (+\infty, +\infty) & \text{si} \quad i = 0 \\ \min\limits_{1^{er} \to 2^{da}} (cc(i-1,j), (0,1) + cc(i-1, j-B[i])) & \text{si no} \end{cases}$$

A partir de cuando se van a repetir estados? No sé. Depende de B, de que pueda combinar billetes para que le resten al costo la misma cantidad. Una vez que por dos caminos haya restado la misma cantidad aparecerá en todas las combinaciones posibles un estado repetido y ahí empezará la superposición.

La cantidad de estados que puede haber está *acotada* por $O(|B| \cdot j)$ y la cantidad de recursiones está en $\Omega(2^n)$.

d) *Algo raro debe haber*

⚠ En el enunciado aparece el valor k ¿No debería ser c? Justamente el segundo argumento de la función $cc'_B(i,j)$ es el exceso de la operación. No sé, yo voy a suponer que $k = c$ ⚠



- Los caso base: $i = 0$, tengo 0 billete no pago un choto así que pongo $(+\infty, +\infty)$.
- En el elemento `M[n,c]` estará la tupla con el valor buscado.
- Cada elemento `M[i,j]` almacena el mínimo excedente con la cantidad mínima de billetes usada en esa rama.

e) Misma idea que los algoritmos *top-down* que hice antes

```
1  M[i..n,j..c] ← null          // inicializo
2  funcion cc(B,i,j)
3      si j≤0
4          ret (-j, 0)
5      si i=0
6          ret (+∞,+∞)
7      si M[i,j] not null
8          ret M[i,j]
9      sino
10         skip ← cc(B, i-1, j)
11         keep ← (0,1) + cc(B, i-1, j-B[i])
12
13         si keep.exceso < skip.exceso              // comparo 1era
14             M[i,j] ← keep
15         sino si keep.exceso > skip.exceso         // comparo 1era
16             M[i,j] ← skip
17         sino                                    //son iguales comparo cantidad de billetes
18             si keep.billetes < skip.billetes
19                 M[i,j] ← keep
20             sino
21                 M[i,j] ← skip
```

f) Supongo que la llamada recursiva sería $cc(n, c)$. Te tiran cada pregunta de cada manera que no sé que mierda quieren que les diga. Preguntar

El nuevo algoritmo usa memoization. La complejidad temporal de estos algoritmos se calcula como:

$$T(n) = \texttt{cantidad\_de\_estados} \times \texttt{costo\_de\_estados} \rightarrow n \cdot c \times 1 \in O(n \cdot c)$$

Donde n será el tamaño del conjunto B y c el costo objetivo. El *running time* es pseudopolinomial y antes de usar la memoization era exponencial. Hubo una mejora.

Como siempre hay un compromiso con la complejidad espacial, la profundidad del árbol es n y cada estado tiene los posibles variables de estado i y j:

$$S(n) \in O(n \cdot c)$$

La complejidad espacial en el algoritmo de *backtracking* es de $O(n)$, así que acá hubo un compromiso de recursos al bajar T y aumentar S.

g) Debe haber una forma más simple de hacer esto. Me salió así.

- El índice difícil fue el j.
- La matriz de más arriba es lo que

```
1  M[0..n,0..c] ← (+∞,+∞) // Inicializo con el valor neutro del mínimo.
2  funcion cc(n, c)
3      para i ← 1..n                        // tengo n billetes
4          para j ← 0..c                    // tengo c posibles excesos
5              // Si decido no usar el billete. Igual que en la función recursiva.
6              skip ← M[i-1,j]
7              // Si decido usar el billete. Igual que en la función recursiva.
8              si j-B[i] ≤ 0                // Que no exploten los índices
9                  keep ← (-(j - B[i]), 1) // Caso base. Esto es una posible solución.
10             sino
11                 keep ← (0,1) + M[i-1, j-B[i]]
12             // Pongo minTupla que me lo ordena las tuplas skip y keep, por exceso (1era) y luego billetes (2da).
13             // Paja escribir las comparaciones. Está hecho en la función anterior.
14             M[i,j] ← minTupla(skip, keep)
15     ret M[n, c]
```

h) Hacer!

13. *AstroTrade*

Astro Void se dedica a la compra de asteroides. Sea $p \in N^n$ tal que p_i es el precio de un asteroide el $i-$ésimo día en una secuencia de n días. Astro Void quiere comprar y vender asteroides durante esos n días de manera tal de obtener la mayor ganancia neta posible. Debido a las dificultades que existen en el transporte y almacenamiento de asteroides, Astro Void puede comprar a lo sumo un asteroide cada día, puede vender a lo sumo un asteroide cada día y comienza sin asteroides. Además, el Ente Regulador Asteroidal impide que Astro Void venda un asteroide que no haya comprado. Queremos encontrar la máxima ganancia neta que puede obtener Astro Void respetando las restricciones indicadas. Por ejemplo, si $p = (3, 2, 5, 6)$ el resultado es 6 y si $p = (3, 6, 10)$ el resultado es 7. Notar que en una solución óptima, Astro Void debe terminar sin asteroides.

a) Convencerse de que la máxima ganancia neta (m.g.n), si Astro Void tiene c asteroides al fin del día j, es:

- indefinido (i.e., $-\infty$) si $c < 0$ o $c > j$, o
- el máximo entre:
  - la m.g.n. de finalizar el día $j-1$ con $c-1$ asteroides y comprar uno en el día j,
  - la m.g.n. de finalizar el día $j-1$ con $c-1$ asteroides y vender uno en el día j,
  - la m.g.n. de finalizar el día $j-1$ con c asteroides y no operar el día j.

b) Escribir matemáticamente la formulación recursiva enunciada en *a*). Dar los valores de los casos base en función de la restricción de que comienza sin asteroides.

c) Indicar qué dat es la respuesta al problema con esa formulación recursiva.

d) Diseñar un algoritmo de PD *top-down* que resuelva el problema y explicar su complejidad temporal y espacial auxiliar.

e) (Opcional) Diseñar un algoritmo de PD *bottom-up*, reduciendo la complejidad espacial.

f) (Opcional) Formalmente, el problema consiste en determinar el máximo $g = \sum_{i=1}^{n} x_i p_i$ para un vector $x = (x_1, \ldots, x_n)$ tal que: $x_i \in \{-1, 0, 1\}$ para todo $1 \leq i \leq n$ y $\sum_{i=1}^{j} x_i \leq 0$ para todo $1 \leq j \leq n$. Demostrar que la formulación recursiva es correcta. **Ayuda**: Primero demostrar qu existe una solución óptima en la que Astro Void se queda sin asteroides en el día n. Luego, demostrar por inducción que la función recursiva respeta la semántica, i.e., que computa la m.g.n. al final del día j cuando Astro Void posee c asteroides.

---

a) ¿Qué significa $c < 0$?:

Sería que vendí más *asteroides* de los que tenía. Lo cual no es posible, *thus* $-\infty, \perp$.

¿Qué significa $c > j$?:

Sería que compré más *asteroides* de lo permitido, dado que solo puedo comprar uno por día, *thus* $-\infty, \perp$.

Las condiciones que siguen son acciones *coherentes* con las restricciones de la compra y venta del enunciado. Y son las posibles variantes como siempre de hacer *fuerza bruta* contemplando todas los *multiversos*:

- El *penúltimo* día tengo $c-1$ asteroides entonces compro uno el último día, así garantizando que $c_j \not< 0$
- El *penúltimo* día tengo $c+1$ asteroides entonces vendo uno el último día, así garantizando que entre platita.
- El *penúltimo* día tengo c asteroides entonces no hago nada

b) En la siguiente función j representa los días y c la cantidad de asteroides que tengo en el día j. La función la llamaría inicialmente con *astroTrade*$(0, 1)$, notando que la rama al vender, sin tener ningún asteroide, moriría instantáneamente porque $c < 0$ y tampoco podría comprar más de un asteroide porque caería en $c > j$.

Llegado al día $n+1$ tengo caso base terminando la ejecución de llamados recursivos.

j: Días, va de $[1..n]$ c: asteroides, va de $[0..n]$

$$astroTrade(c,j) = \begin{cases} 0 & \text{si} \quad j > n \\ -\infty & \text{si} \quad c > j \ \vee \ c < 0 \\ \text{máx} \begin{pmatrix} -p_j + astroTrade(c+1, j+1), \\ p_j + astroTrade(c-1, j+1), \\ astroTrade(c, j+1) \end{pmatrix} & \text{si no} \end{cases}$$

c) No entiendo la consigna. ¿La m.g.n? Pero eso no es un dato. No sé. Consultar

d) Acá aparece una nueva terminología: *Complejidad auxiliar*. ¿Por qué?... *just why?*.

En el siguiente algoritmo se usa la misma estructura que en todos los ejercicio que se vienen usando de *top-down*. Hay que tener cuidado con no *intentar acceder* con índices negativos, la ubicación de la guarda de estructura de memoization es importante, fijate que *atajo* el caso de $c < 0$ antes de intentar evaluar la matriz con un $c < 0$, así salvando al mundo y quizás a la galaxia de una explosión de proporciones inimaginables, gracias, de nada.

```
1 M[0..n,1..n] ← −∞ // Inicializo en −∞ la matriz de (n+1) × n
2 funcion astroTrade(c,j)
3     si j>n
4         ret 0
5     si c>j o c<0
6         ret −∞
7     si M[c,j] ≠ −∞
8         ret M[c,j]
9     sino
10        M[c,j] ← max(-p[j]+astroTrade(c+1,j+1),p[j]+astroTrade(c-1,j+1),astroTrade(c,j+1))
11    ret M[c,j]
```

Este algoritmo tiene una complejidad temporal o running time de:

$$T(n) = c \cdot n \cdot O(1) \in O(c \cdot n).$$

Notable mejora del algoritmo de *backtracking* que es en el *worst case* $O(3^n)$.

La complejidad espacial está dada por los posibles valores que podrían tomar los estados:

$$S(n) = c \cdot n \in O(c \cdot n) \in O(n^2)$$

e) *Bottom-up* y encima me piden que baje la complejidad. Bancá la parada cdth, lo hago como puedo y después optimizo si me quedan ganas.

</>$_1$) Reconstruir la solución de atrás para adelante. Arrancando por los casos base.

</>$_2$) Ver relación entre un estado y el siguiente. ¿Cómo depende entre sí?

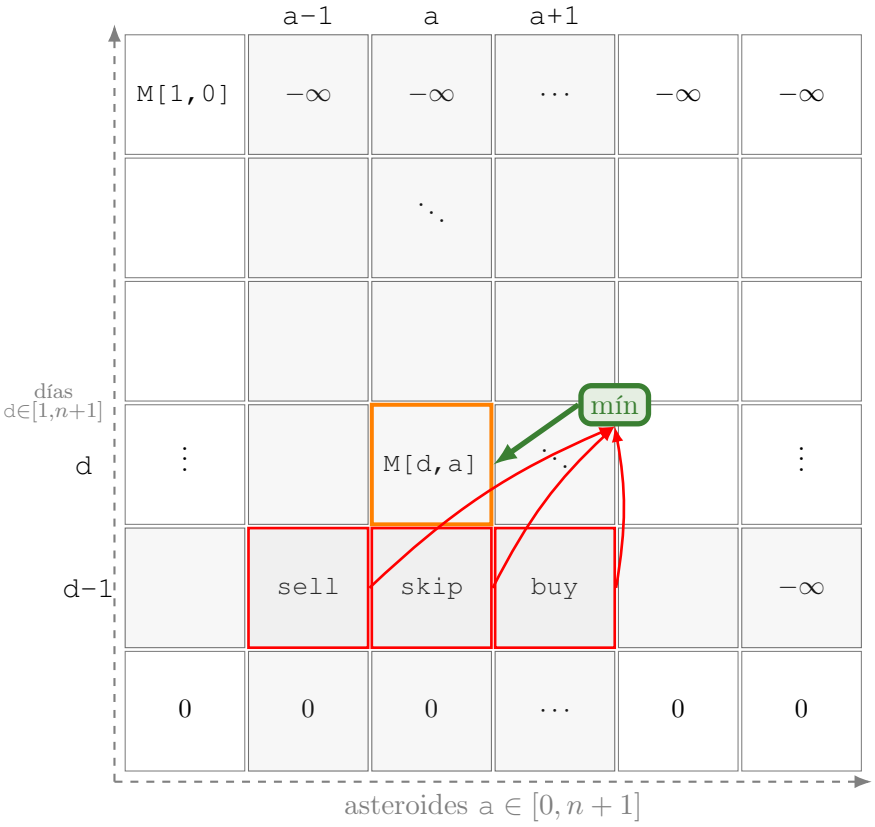
</>$_3$) Si tengo cosas como $i-1$ e $i+1$ voy a agrandar la estructura para evitar errores de bordes.

</>$_4$) Según como llene ubico qué elemento de la matriz representa mi solución.

Por la forma en que armé la función recursiva voy a "*ir para atrás en los días*" para llegar al valor buscado, voy a devolver el valor de $M[1,0]$, empezar el día 1 con 0 asteroides.

Cada nuevo elemento depende de 2 o 3 elementos anteriores ya calculados según casos bordes (obviamente también de los precios de compra y venta, pero eso no importa para entender el armado). El $-\infty$ es el valor neutro a usar para descartar comparaciones que sé que no aportan nada. En la matriz hay un bosquejo de como debería ser el llenado.

El elemento `M[1,0]` es el valor buscado.




```
1 M[1..n+1,0..n+1] ← −∞        // Inicializo en −∞ la matriz de (n+1) × (n+2)
2 M[n+1, 0..n+1] ← 0           // Inicializo la última fila con ceros
3 funcion astroTrade(c,j)
4     para d ← n..1
5         para a ← 0..n
6             sell,buy ← −∞ // valores neutros al max por si los índices explotan
7             si a-1≥0   // Si en "d" tengo "a" y vendo entonces en "d+1" tengo "a−1" y más platita.
8                 sell ← M[d+1,a-1]+p[d]
9             si a+1≤n   // Si en "d" tengo "a" y compro entonces en "d+1" tengo "a+1" y menos platita.
10                buy ← M[d+1,a+1]-p[d]
11            skip ← M[d+1,a] // no hago nada
12
13            M[d,a] ← max(skip,sell,buy)
14    ret M[1,0]
```

Este algoritmo es el que va, el más *común, mecánico, intuitivo y que funciona*, peeeeeero no cumple la complejidad espacial $S(n)$ que pide el enunciado. Para eso hay que notar que siempre que lleno una nueva fila (d) de la matriz `M` nunca vuelvo a usar la fila usada (d+1), y ahí me pregunto:

¿Vale la pena tener guardada en memoria está fila usada la cual de nada me sirve ahora? → No.

Lo que me viene a la cabeza es esa variante de fibonacci en el ejercicio **9.** que va pisando cada cálculo que va haciendo y termina usando solo 3 variables, click click. Quiero hacer algo así modificando el algoritmo *bottom-up* anterior para bajar la complejidad a $S(n) \in O(n)$:

```
1 funcion astroTrade(n,p)
2     new[0..n+1] ← −∞
```

```
3     old[0..n+1] ← 0 // inicializo con casos base
4     d ← n
5     mientras d ≥ 1
6         para a ← 0..n
7             sell,buy ← −∞
8             si a-1≥0
9                 sell ← old[a-1]+p[d]
10            si a+1≤n
11                buy ← old[a+1]-p[d]
12            skip ← old[a]
13            new[a] ← max(sell, buy, skip)
14        old ← new // old apunta a new. Con alguna implementación que no haga copias, ponele.
15        new[0..n+1] ← −∞ // reseteo new
16        d ← d-1
17    ret new[0]
```

La complejidad temporal queda cuadrática porque tengo 2 loops anidados:

$$T(n) \in O(n^2)$$

Y la complejidad espacial ya no es cuádrática sino que lineal como se pedía:

$$S(n) \in O(n)$$

f) Hacer!

---

14. *Fire*
hecho en clase, pasarlo

---

15. *CortesEconómicos*
Ver el Cormen feo.

---

16. *Travesía Vital*

Hay un terreno, que podemos pensarlo como una grilla de m filas y n columnas, con trampas y pociones. Queremos llegar de la esquina superior izquierda hasta la inferior derecha, y desde cada casilla sólo podemos movernos a la casilla de la derecha o a la de abajo. Cada casilla i, j tiene un número entero $A_{i,j}$ que nos modificará el nivel de vida sumándonos el número $A_{i,j}$ (si es negativo, nos va a restar $|A_{i,j}|$ de vida). Queremos saber el mínimo nivel de vida con el que debemos comenzar tal que haya un camino posible de modo que en todo momento nuestro nivel de vida sea al menos 1. Por ejemplo, si tenemos la grilla

$$A = \begin{bmatrix} -2 & -3 & 3 \\ -5 & -10 & 1 \\ 10 & 30 & -5 \end{bmatrix}$$

el mínimo nivel de vida con el que podemos comenzar es 7 porque podemos realizar el camino que va todo a la derecha y todo abajo.

a) Pensar la idea de un algoritmo de *backtracking* (no hace falta escribirlo).

b) Convencerse de que, excepto que estemos en los límites del terreno, la mínima vida necesaria al llegar a la posición i, j es el resultado de restar al mínimo entre la mínima vida necesaria en $i+1, j$ y aquella en $i, j+1$, el valor $A_{i,j}$, salvo que eso fuera menor o igual que 0, en cuyo caso sería 1.

c) Escribir una formulación recursiva basada en *b*). Explicar su semántica e indicar cuáles serían los parámetros para resolver el problema.

d) Diseñar un algoritmo de PD y dar su complejidad temporal y espacial auxiliar. Comparar cómo resultaría un enforque *top-down* con uno *bottom-up*.

e) Dar un algoritmo *bottom-up* cuya complejidad temporal sea $O(m \cdot n)$ y la espacial auxiliar sea $O(\text{mín}(m, n))$.

Buen enunciado, consignas guiadas haciendo foco en un procedimiento incremental. Contrasta fuertemente con esos ejercicios de verga sacados de *Codeforces* los cuales solo tienen sentido para practicar cuando uno ya "*se siente cómodo*" con la técnica ...que difícil se me hace.

a) ¿Pensar? Eso es la receta para entrar en una vorágine de caca recursiva. No pienso luego no existo, *fuerza bruta*, como le gusta a tu hermana.

b) Me costó implementar lo de "en cuyo caso sería 1", pero tiene sentido. Poner el 1 de prepo de está diciendo que la *energía inicial* será lo que tenga que ser para llegar a esa celda y sobrevivir como un campeón.

c) Ahí te va un primer intento fallido: Matriz $A \in \mathbb{R}^{m \times n}$, global.

$$travesíaVital(i,j) = \begin{cases} 1 & \text{si} \quad i > m \wedge j > n \\ -A_{i,j} + travesíaVital(i+1,j) & \text{si} \quad i \leq m \wedge j > n \\ -A_{i,j} + travesíaVital(i,j+1) & \text{si} \quad i > m \wedge j \leq n \\ -A_{i,j} + \text{mín} \begin{pmatrix} travesíaVital(i+1,j), \\ travesíaVital(i,j+1) \end{pmatrix} & \text{si} \quad i \leq m \wedge j \leq n \end{cases}$$

Si bien está buena, está mal, ha! Esa función permite potencialmente que cualquier celda me dé una "*energía acumulada*" menor a 1. Así que tengo que poner algo para que en ningún momento el camino me dé menor a 1:

$$travesíaVital(i,j) = \begin{cases} 1 & \text{si} \quad i > m \wedge j > n \\ \text{máx}(1, -A_{i,j} + travesíaVital(i+1,j)) & \text{si} \quad i \leq m \wedge j > n \\ \text{máx}(1, -A_{i,j} + travesíaVital(i,j+1)) & \text{si} \quad i > m \wedge j \leq n \\ \text{máx} \left(1, -A_{i,j} + \text{mín} \begin{pmatrix} travesíaVital(i+1,j), \\ travesíaVital(i,j+1) \end{pmatrix} \right) & \text{si} \quad i \leq m \wedge j \leq n \end{cases}$$

La función devuelve el mínimo valor inicial de energía que hay que tener llegar a la posición m, n sin tener nunca una suma acumulada menor a 1 en todo el recorrido. Los parámetros i, j representan las coordenadas en la matriz que se va recorriendo. Llamar a la función con $travesíaVital(1, 1)$ devuelve el valor buscado.

d) Hacer!

e) Hacer!

---

17. *PilaCauta*

Tenemos cajas numeradas de 1 a N, todas de iguales dimensiones. Queremos encontrar la máxima cantidad de cajas que pueden apilarse en una única pila cumpliendo que:

- Solo puede haber una caja apoyada directamente sobre otra;
- Las cajas de la pila deben estar ordenadas crecientemente por número, de abajo para arriba;
- Cada caja i tiene un peso w_i y un soporte s_i, y el peso total de las cajas que están arriba de otra no debe exceder el soporte de esa otra.

Si tenemos los pesos $w = [19, 7, 5, 6, 1]$ y los soportes $s = [15, 13, 7, 8, 2]$ (la caja 1 tiene peso 19 y soporte 15, la caja 2 tiene peso 7 y soporte 13, etc.), entonces la respuesta es 4. Por ejemplo, pueden apilarse de la forma 1-2-3-5 o 1-2-4-5 (donde la izquierda es más abajo), entre otras opciones.

a) Pensar la idea de un algoritmo de *backtracking* (no hace falta escribirlo).

b) Escribir una formulación recursiva que sea la base de un algoritmo de PD. Explicar su semántica e indicar cuáles serían los parámetros para resolver el problema.

c) Diseñar un algoritmo de PD y dar su complejidad temporal y espacial auxiliar. Comparar cómo resultaría un enfoque *top-down* con uno *bottom-up*.

d) (Opcional) Formalizar el problema y demostrar que la función recursiva es correcta.

---

a) No pienso luego no existo.

b) Como todo algoritmo de *backtracking* pruebo todas las instancias armando un árbol de recursión *masivo bro*.

La función devuelve la máxima cantidad de pilas que puedo apilar respetando los soportes. La variable S tiene el valor del mínimo soporte en la instancia i. La idea es que la caja $i+1-$ésima sea soportable por *el cuello de botella* S para poder apilarse. El S se actualiza solo cuando se agrega una caja.

$$pilaCauta(i, S) = \begin{cases} 0 & \text{si} \quad i+1 > N \\ pilaCauta(i+1, S) & \text{si} \quad S < w_{i+1} \\ \max \begin{pmatrix} 1 + pilaCauta(i+1, \min(s_{i+1}, S - w_{i+1})), \\ pilaCauta(i+1, S), \end{pmatrix} & \text{si no} \end{cases}$$

Llamo a la función con $pilaCauta(0, +\infty)$, el piso debería poder soportar cualquier caja, además ese caso cae siempre en el caso de poner una caja nueva.

Esta función recursiva tiene 2 llamadas recursiva por cada llamada a la función (*give or take*) tiene una complejidad temporal o running time:

$$T(n) \in O(2^n)$$

Se le podrían podar las ramas cuando $S < 0$, pero eso no afectaría el valor de la complejidad.

c) ¿Cuántos estados puede devolver esta función? La i es fácil, porque es solo la cantidad de cajás, es decir $i \in [1 \dots N]$.

¿Cuáles valores podría tomar, el soporte cuello de botella S?

Esto no es lo más intuitivo del mundo (*imho* pensar y llegar a esta cota es impedible en un parcial).

Cosas esperables:

- De S me importa lo que uso, no su valor máximo.
- Si el suelo tiene $S_{suelo} = \infty$ quiero encontrar una cota superior menor para S.
- Lo que puedo usar de S en caso óptimo es $\sum_{i=1}^{N} w_i$.
- Una caja i tiene su propiedad $e_i = w_i + s_i$. Es el valor máximo que puede ejercer sobre la caja de abajo $i-1$.
- $e_{max} = \max_{1 \leq i \leq N} (w_i + s_i)$. De poder elegir el orden, la caja con e_{max} sería la candidata para estar en el piso
- Si el suelo fuese una caja con $S = \infty$, el siguiente "suelo" sería la caja con e_{max}.

Viendo la función *pilaCauta*, observamos que S es el resultado de calcular un mínimo. Ese mínimo es lo necesario para bancar al sistema, y para bancar al *sistema* (lo que esté abajo de la caja j) S_{sis} tiene que ser suficientemente poronga como para aguantar a la caja j y todo lo que tiene arriba ($i>j$), por lo que tiene que cumplir que $S_{sis} \geq w_j + \overbrace{s_j}^{\geq W_{i>j}}$. Debería ocurrir algo así:

$$S_{sis} \geq \max_{j \leq i \leq N} (w_i + s_i).$$

y ya que estamos debería ocurrir que $s_j \geq \sum_{i>j}^{N} w_i$, y así teniendo otra posible cota:

$$S_{sis} \geq \sum_{i>j}^{N} w_i$$

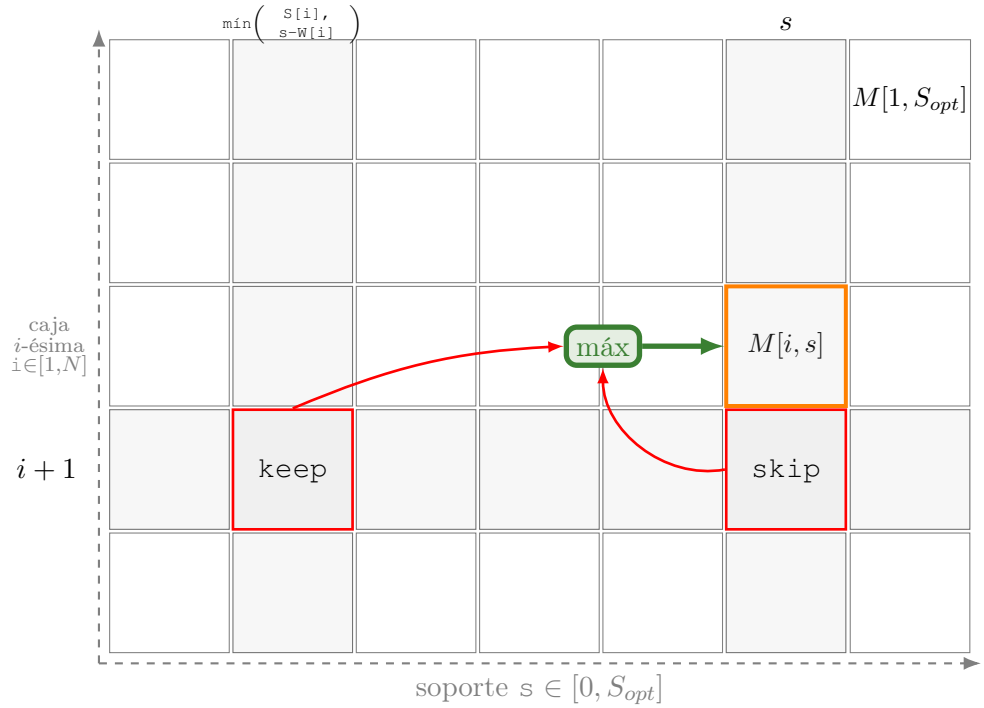
Dado que *que el sobrante de soporte no agrega nuevos estados* me quedo con el mínimo de esas dos expresiones S_{opt}. Sin pérdida de generalidad para $[1, N]$ los valores útiles de S estarán acotados por:

$$S_{opt} = \min \Big(\max_{1 \leq i \leq N} (w_i + s_i), \sum_{i=1}^{N} w_i \Big)$$

¿Cómo se relacionan los estados entre sí?

- El índice complicado es el de S.
- Voy a llenar de abajo para arriba (*outer loop*) la variable `i`. Luego de izquierda a derecha (*inner loop*) o al revés, no importa, la variable `s`.
- Caso base en $i = N + 1$ como en la función recursiva.
- De la forma en que lleno voy a tener el objetivo en $M[1, S_{opt}]$.

El vector de pesos `W`, vector de soportes `S`.




```
1 funcion pilaCauta(N,W[1..N],S[1..N])
2     S_opt ← min(max(W[i]+S[i]),sum(W))    // El detalle de eso, pajilla.
3     M[1..N+1,0..S_opt] ← 0                // Inicializo con 0 cajas. M es de (N+1)×(S_opt+1)
4     para i←N..1
5         para s ←0..S_opt
6             skip ← M[i+1, s]
7             si s-W[i] ≥ 0 // Borde ok?
8                 keep ← 1+M[i+1,min(S[i],s-W[i])]
9             sino
10                keep ← −∞
11            M[i,s] ← max(skip,keep)
12    ret M[1,S_opt]
```

La complejidad temporal o *running time* de este algoritmo:

$$T(n) \in O(N \cdot S_{opt})$$

Es una complejidad pseudopolinomial, una mejora respecto a la función recursiva con complejidad exponencial. La complejidad espacial:

$$S(n) \in O(N \cdot S_{opt})$$

Se puede bajar la complejidad espacial a $O(S_{opt})$. Esta forma no es la más eficientes pero me parece más intuitiva.

```
1 funcion pilaCauta(N,W,S)
2     new[0..S_opt] ← 0
3     old[0..S_opt] ← 0              // inicializo con casos base
4     para i ← N..1
5         para s ← 0..S_opt
6             skip ← old[s]
7             si s-W[i] ≥ 0          // Borde ok?
8                 keep ← 1+old[min(S[i],s-W[i])]
9             sino
10                keep ← −∞
11            new[s] ← max(skip,keep)
```

```
12          old ← new // old apunta a new. Con alguna implementación que no haga copias, ponele.
13      ret new[S_opt]
```

Para mejorar eso un poco más y que baje un poco la complejidad temporal evitando las copias de los arreglos hago mismo encare que ya sé hizo en otros ejercicios. Dada la dependencia de los estados, puedo ir pisando de manera correcta los elementos de un arreglo unidimensional y llegar al resultado buscado con una complejidad espacial óptima:

```
1 funcion pilaCauta(N,W,S)
2     A[0..S_opt] ← 0                          // inicializo con casos base caja (N+1)−esima
3     para i ← N..1
4         para s ← S_opt..0                    // Orden clave para el pisado
5             skip ← A[s]
6             si s-W[i] ≥ 0                    // Borde ok?
7                 keep ← 1+A[min(S[i],s-W[i])]
8             sino
9                 keep ← −∞
10            A[s] ← max(skip,keep)
11    ret A[S_opt]
```

La complejidad espacial de los últimos dos algoritmos es:

$$S(n) \in O(S_{opt})$$

Top-down (↓) vs. *Bottom-up* (↑):

(↓) *Top-Down* está bueno si los estados están dispersos, porque calcula los estados que existen a partir de valores de los vectores `S: soporte` y `W: pesos`.

(↑) Si tuviese una distribución uniforme de estados, preferiría usar *Bottom-up* que va a calcular todos los valores desde el más grande al más chico, y *en promedio va a calcular pocas cosas al pedo.*

(↑) *Bottom Up* Es más fácil de optimizar el uso de memoria como se vio al final.

En conclusión va depender del *input*. En los talleres de *Codeforces, Top-down* no funciona nunca. Quizás por la variedad de los tests, no sé. El sabor que queda es que *Top-Down* es una 💩.

d) Ehm, me acordé de que tengo que hacer algo, después la hago. Hacer!

---

18. *OperacionesSeq*

Sea $v = (v_1, v_2, \ldots, v_n)$ un vector de números naturales, y sea $w \in \mathbb{N}$. Se desea intercalar entre los elementos de v las operaciones $+$ (suma), $\times$ (multiplicación) y $\uparrow$ (potenciación) de tal manera que al evaluar la expresión obtenida el resultado sea w. Para evaluar la expresión se opera de izquierda a derecha ignorando la precedencia de los operadores. Por ejemplo, si $v = (3, 1, 5, 2, 1)$, y las operaciones elegidas son $+, \times, \uparrow$ y $\times$ (en ese orden), la expresión obtenida es $3 + 1 \times 5 \uparrow 2 \times 1$, que se evalúa como $(((3 + 1) \times 5) \uparrow 2) \times 1 = 400$.

a) Escribir una formulación recursiva que sea la base de un algoritmo de PD que, dados v y w, encuentre una secuencia de operaciones como la deseada, en caso de que tal secuencia exista. Explicar su semántica e indicar cuáles serían los parámetros para resolver el problema.

b) Diseñar un algoritmo basado en PD con la formulación de *a*) y dar su complejidad temporal y espacial auxiliar. Comparar cómo resultaría un enfoque *top-down* con uno *bottom-up*.

c) (Opcional) Formalizar el problema y demostrar que la función recursiva es correcta.

---

a) ¿Tengo que dar la secuencia? Voy a elegir que no, solo con encontrar que existe tal secuencia soy feliz y además porque dar la secuencia es más laburo. (En realidad no se me ocurre como escribirlo).

Quiero hacer todas las combinaciones, *fuerzo-bruteo* para encontrar una posible combinación.

- i es la iteración. $i \in [1, n]$
- rP es el *resultado parcial*, un acumulador de las operaciones realizadas en cada iteración.
- El caso base es una proposición que chequea si el resultado de la combinación números-operaciones es w.
- La función combina los valores del arreglo v con los operadores $\{+, \times, \uparrow\}$ y devuelve *true* en caso de existir una combinación que al usar todos los elementos de v equivalga al valor w. Devuelve *false* en caso contrario.

$$operacionesSeq(i, rP) = \begin{cases} rP = w & \text{si} \quad i = n \\ \bigvee_{\dagger}^{+,\times,\uparrow} operacionesSeq(i+1, rP \dagger v[i+1]) & \text{si} \quad i < n \end{cases}$$

Llamo a la función con $operacionesSeq(1, v[1])$, para obtener el resultado buscado. Cada llamada tiene 3 llamados recursivos, lo que daría un árbol que triplica su tamaño en cada iteración, teniendo una complejidad de $O(3^n)$. Lo podo un poco ya que las operaciones que se usan en números naturales siempre dan algo mayor:

$$operacionesSeq(i, rP) = \begin{cases} rP = w & \text{si} \quad i = n \\ false & \text{si} \quad rP > w \\ \bigvee_{\dagger}^{+,\times,\uparrow} operacionesSeq(i+1, rP \dagger v[i+1]) & \text{si no} \end{cases}$$

Podría la sucesión de operadores que resulta en éxito pasando una referencia a un arreglo como parámetro un arreglo de $C[1..n-1]$ que vaya acumulando las operaciones usadas:

$$operacionesSeq(i, rP, C) = \begin{cases} rP = w & \text{si} \quad i = n \\ false & \text{si} \quad rP > w \\ \bigvee_{\dagger}^{+,\times,\uparrow} operacionesSeq(i+1, rP \dagger v[i+1], C[i] = \dagger) & \text{si no} \end{cases}$$

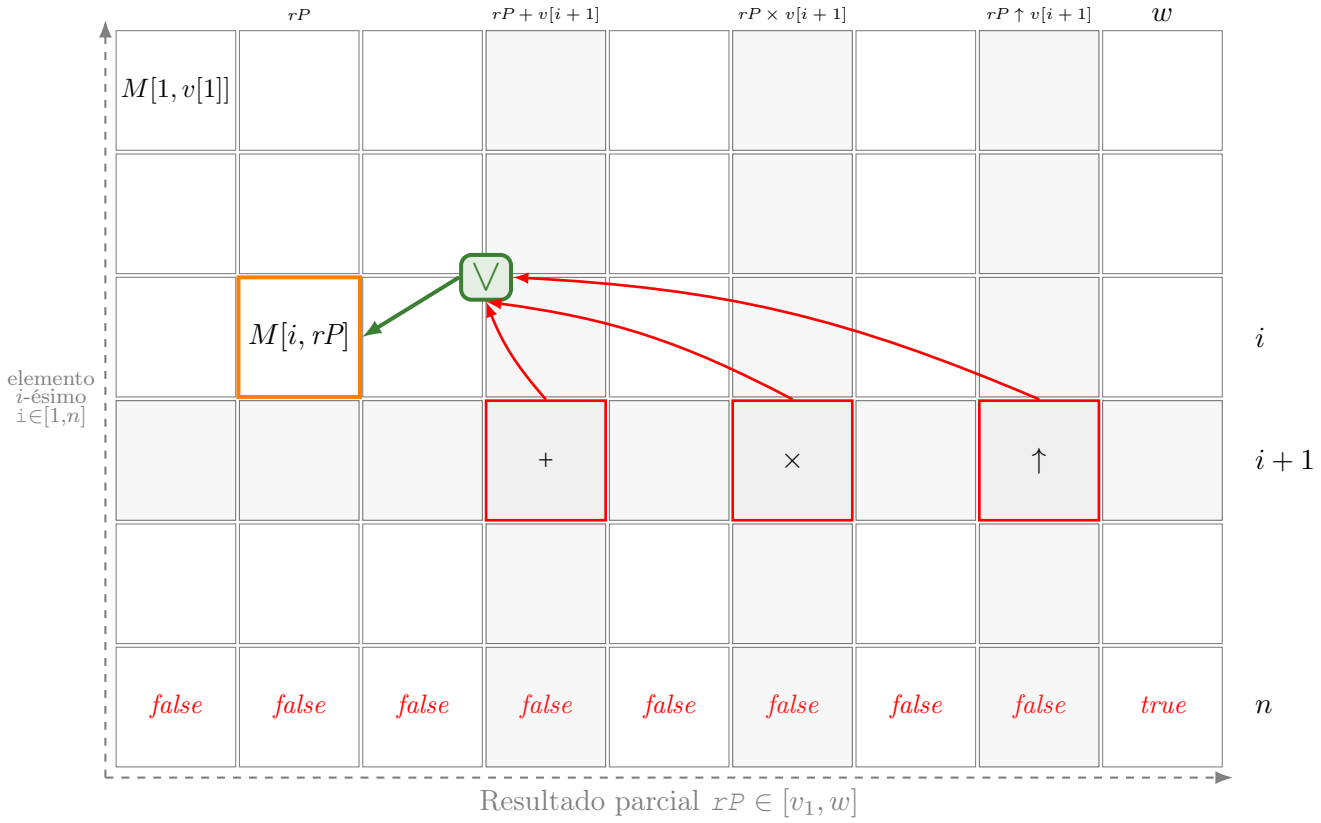
Muy lindo, pero esto no sirve para *programación dinámica*, porque al meter una variable de estado que es un arreglo está complicado para que sea un índice de una matriz. Yo quiero que a partir de las variables de estados la función me retorne algún valor que pueda poner en la matriz y así ser feliz.

b) ¿Tiene sentido hacer un algoritmo de *programación dinámica*? Sí, pues tengo muchas más llamadas recursivas que estados para calcular.

$$\Omega(3^n) \gg O(n \cdot rP)$$

Voy de atrás para adelante. Partiendo del *true* en $i = n$ reconstruyo el camino de *true*.

- i varía entre 1 y n.
- El índice raro es el rP. Varía entre v_1 y w.
- Casos base son todos *false*, excepto en la columna w.
- El *outer loop* es i y lo lleno de $n \to 1$
- El *inner loop* no importa, pero lo lleno de $v_1 \to w$. Esta es la dirección que usaría para una baja en la complejidad espacial (mismo truco ya usadísimo).




```
1 funcion operacionesSeq(n,v[1..n],w) → bool
2     M[1..n,1..w] ← false      // inicializo todo false
3     M[n, w] ← true
4     para i ← (n-1)..1
5         para rP ← v[1]..w
6             op+,op×,op↑ ← false   // inicializo los operadores
7             si (rP+v[i+1])≤w
8                 op+ ← M[i+1,rP+v[i+1]]
9             si (rP×v[i+1])≤w
10                op× ← M[i+1,rP×v[i+1]]
11            si (rP↑v[i+1])≤w
12                op↑ ← M[i+1,rP↑v[i+1]]
13            M[i,rP] ← (op+ o op× o op↑) // Queda el true de haberlo
14    ret M[1, v[1]]
```

Ese algoritmo tiene una complejidad temporal o *running time* y espacial:

$$T(n) \in O(n \cdot w) \quad \text{y} \quad S(n) \in O(n \cdot w)$$

Truco de optimización de siempre. Como solo pido info de la fila anterior, puedo ir *pisando*. Atención a la dirección de llenado del *inner loop* que es importante para no pisar data importante.

```
1 funcion operacionesSeq(n,v[1..n],w) → bool
2     A[1..w] ← false                    // inicializo todo false
3     A[w] ← true                        // inicializo caso exitoso
4     para i ← (n-1)..1
5         para rP ← v[1]..w   // Dirección correcta de llenado
6             op+,op×,op↑ ← false   // inicializo los operadores
7             si (rP+v[i+1])≤w
8                 op+ ← A[rP+v[i+1]]
9             si (rP×v[i+1])≤w
10                op× ← A[rP×v[i+1]]
```

```
11              si (rP↑v[i+1])≤w
12                  op↑ ← A[rP↑v[i+1]]
13              A[rP] ← (+ o × o ↑) // queda el true de haberlo
14      ret A[v[1]]
```

Mismo *running time* que el anterior pero mejoró la complejidad espacial:

$$\boxed{S(n) \in O(w)}$$

c) Hago inducción en la cantidad de elementos de v:

Quiero probar la proposición:

$$p(n) : \textit{Dada una secuencia } v = \{v_1, \ldots, v_n\}, \textit{ un número objetivo } w \textit{ y el conjunto de operadores } O = \{+, \times, \uparrow\}, \textit{ la función } operacionesSeq(1, v_1) = true \textit{ en caso de existir una combinación ordenada intercalando los elementos de } v \textit{ y } O \textit{ y } false \textit{ sino} \quad \forall n \in \mathbb{N}.$$

Caso base:

$p(1)$ es trivial ya que $operacionesSeq(1, v[1]) = \begin{cases} true & \text{si} \quad v_1 = w \\ false & \text{si no} \end{cases}$

$$p(2) : \textit{Dada una secuencia } v = \{v_1, v_2\}, \textit{ un número objetivo } w \textit{ y el conjunto de operadores } O = \{+, \times, \uparrow\}, \textit{ la función } operacionesSeq(1, v_1) = true \textit{ en caso de existir una combinación ordenada intercalando los elementos de } v \textit{ y } O \textit{ y } false \textit{ sino.}$$

No hay mucha rosca:

$$operacionesSeq(1, v_1) = (v_1 + v_2 = w) \vee (v_1 \times v_2 = w) \vee (v_1 \uparrow v_2 = w)$$

Eso va a devolver *true* si alguna de las operaciones coincide o *false* si ninguna. Por lo tanto $p(1)$ y $p(2)$ son verdaderas. *Paso inductivo:*

Asumo que para un $k \in \mathbb{N}$ la proposición:

$$p(k) : \textit{Dada una secuencia } v = \{v_1, \ldots, v_k\}, \textit{ un número objetivo } w_k \textit{ y el conjunto de operadores } O = \{+, \times, \uparrow\}, \textit{ la función } operacionesSeq(1, v_1) = true \textit{ en caso de existir una combinación ordenada intercalando los elementos de } v \textit{ y } O \textit{ y } false \textit{ sino.}$$

es verdadera. Entonces quiero probar que la proposición:

$$p(k+1) : \textit{Dada una secuencia } v = \{v_1, \ldots, v_{k+1}\}, \textit{ un número objetivo } w_{k+1} \textit{ y el conjunto de operadores } O = \{+, \times, \uparrow\}, \textit{ la función } operacionesSeq(1, v_1) = true \textit{ en caso de existir una combinación ordenada intercalando los elementos de } v \textit{ y } O \textit{ y } false \textit{ sino.}$$

también lo sea.

Hay 2 cosas que pueden ocurrir, ya sea que exista y devuelva *true* o que no exista y devuelva *false*.

Caso true:

$$operacionesSeq(k+1, rP) = true \overset{!!}{\iff} \begin{cases} operacionesSeq(k, rP - v_{k+1}) = true \\ \vee \\ operacionesSeq(k, rP \div v_{k+1}) = true \\ \vee \\ operacionesSeq(k, \sqrt[v_{k+1}]{rP} = true \end{cases}$$

Alguno de los valores: $\left\{ \begin{array}{c} rP - v_{k+1} \\ rP \div v_{k+1} \\ \sqrt[v_{k+1}]{rP} \end{array} \right\}$ tiene que ser el w_k, el valor objetivo del paso k. Por hipótesis inductiva la función en el paso k-ésimo funciona 10 puntos así que eso muestra que está todo bien.

El caso cuando la función retorna *false* es análogo. Solo que al usar la hipótesis inductiva voy a obtener todos valores *false*

Queda así demostrado que la proposición $p(k+1)$ es verdadera.

Por criterio de inducción la proposición $p(n)$ resulta verdadera para todo $n \in \mathbb{N}$.

---

19. *DadosSuma*

Se arrojan simultáneamente n dados, cada uno con k caras numeradas de 1 a k. Queremos calcular todas las maneras posibles de conseguir la suma total $s \in \mathbb{N}$ con una sola tirada. Tomamos dos variantes de este problema.

(A) Consideramos que los dados son **distinguibles**, es decir que si $n = 3$ y $k = 4$, entonces existen 10 posibilidades que suman $s = 6$:

1) 4 posibilidades en las que el primer dado vale 1
2) 3 posibilidades en las que el primer dado vale 2
3) 2 posibilidades en las que el primer dado vale 3
4) Una posibilidad en la que el primer dado vale 4

(B) Consideramos que los dados son **indistinguibles**, es decir que si $n = 3$ y $k = 4$, entonces existen 3 posibilidades que suman $s = 6$:

1) Un dado vale 4, los otros dos valen 1
2) Un dado vale 3, otro 2 y otro 1
3) Todos los dados valen 2

a) Definir en forma recursiva la función $f : \mathbb{N}^2 \to \mathbb{N}$ tal que $f(n, s)$ devuelve la respuesta para el escenario (A) (fijado k).

b) Definir en forma recursiva la función $g : \mathbb{N}^3 \to \mathbb{N}$ tal que $g(n, s, k)$ devuelve la respuesta para el escenario (B).

c) Demostrar que f y g poseen la propiedad de superposición de subproblemas.

d) Definir algoritmos *top-down* para calcular $f(n, s)$ y $g(n, s, k)$ indicando claramente las estructuras de datos utilizadas y la complejidad resultante.

e) Escribir el (pseudo-)código de los algoritmos *top-down* resultantes.

Nota: Una solución correcta de este ejercicio debería indicar cómo se computa tanto $f(n, s)$ como $g(n, s, k)$ en tiempo $O(nk \min\{s, nk\})$.

Tirar los dados se puede modelar con un arreglo $d[1..n]$, donde cada elemento del arreglo toma un valor en $[1, \ldots, k]$. El tema de que sean distinguibles o no en el ejemplo del enunciado diferencia la forma de contar los resultados:

$$\overbrace{\underbrace{[1,2,3][1,3,2][2,1,3][2,3,1][3,1,2][3,2,1]}_{1\to(B)}\underbrace{[4,1,1][1,4,1][1,1,4]}_{1\to(B)}\underbrace{[2,2,2]}_{1\to(B)}}^{10\to(A)}$$

Si los dados son indistinguibles tengo 3 combinaciones y de no serlo 10.

a) Este caso es el fácil, porque no me preocupan las repeticiones. Fuerza bruta para calcular que cada elemento

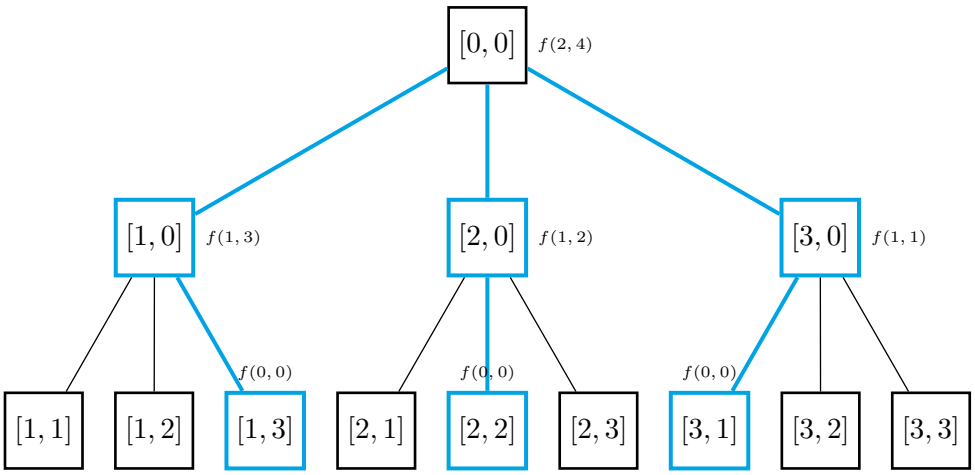
i del arreglo podría tomar cualquier valor de k posible. Hay que notar que para tener *soluciones válidas* s tiene que vivir en $[n, n \cdot k]$.

- La variables n representa un elemento de un arreglo, modelando uno de los n dados.
- La variables s representa el valor que quiero obtener. s decrece en cada llamada recursiva y cuando $s = 0$ estoy en un caso base.
- Los casos bases detectan cuando una combinación da s o cuando no puede seguir esa rama ya que me pasé o quizás *necesitaría que el dado me tire un valor mayor a* k.

$$\textit{dadosSumaF}(n, s) = \begin{cases} 1 & \text{si} \quad n = 0 \land s = 0 \\ 0 & \text{si} \quad n = 0 \lor s < n \lor s > n \cdot k \\ \sum_{i=1}^{k} \textit{dadosSumaF}(n-1, s-i) & \text{si no} \end{cases}$$

Llamo a la función con $\textit{dadosSumaF}(n, s)$ para obtener las posibles combinetas en un lanzamiento de n dados, cada uno con k lados, que sumen s.

Un ejemplo con $n = 2$, $k = 3$ y $s = 4$ de árbol de recursión:



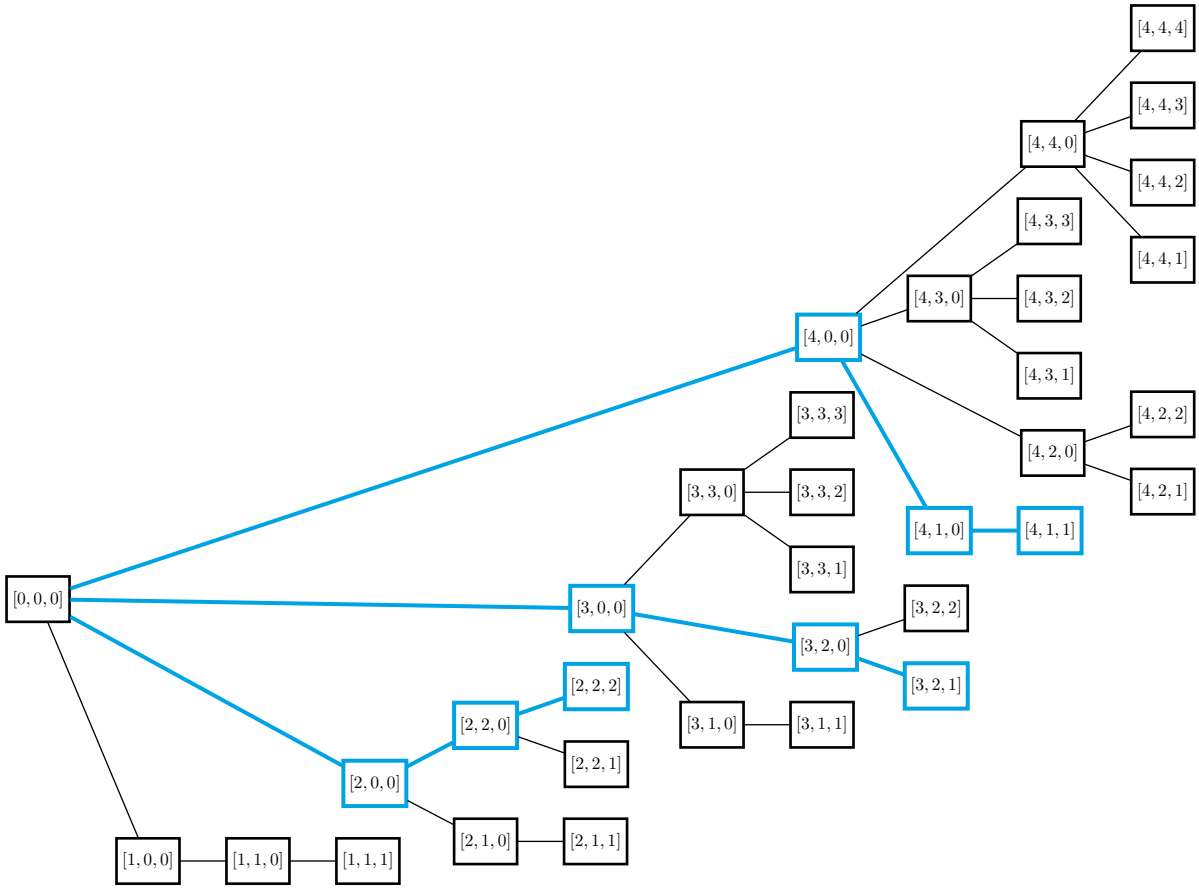
Con k llamadas recursivas y un costo por subproblema de 1 la complejidad temporal temporal será la cantidad de nodos del árbol:

$$T(n) \in O(k^n)$$

b) Necesité olvidarme del ítem anterior para poder pensar este.

La función g tiene 3 parámetros, eso es para que podamos ir *disminuyendo* los valores que puede tomar k. La papa está en ordenar esos valores para *evitar repeticiones*.

Estoy buscando que todas las hojas del árbol sean distintas. La estrategia es *contar hasta el valor que usé*. Onda *armo todo lo que puedo con como máximo el 1*, listo, ahora *armo todo lo que puedo con hasta máximo 2* y así. Hago un árbol con un vector que representa el valor que toma un dado en cadao posición. Por ejemplo si tengo $k = 4$, $n = 3$ y $s = 6$ *¿Cómo ordeno los dados para contar sin repetir?* algo así:



- n representa el dado al que le estamos asignando un valor posible.
- s es la suma objetivo.
- Caso base de éxito se dará cuando $n = s = 0$.
- Caso base poda habrá cuando:
  - Los dados estén todos seleccionados y no sumen s, es decir $n = 0$ pero $s \neq 0$.
  - Cuando lo máximo que queda por acumular no llega a s, es decir $n \cdot k < s$.
  - O bien cuando lo mínimo que queda por acumular se pasa de s, es decir $s < n$.
  - De estos últimos dos se desprende que la condición limitante es mín$(s, n \cdot k)$

$$dadosSumaG(n, s, k) = \begin{cases} 1 & \text{si} \quad n = 0 \wedge s = 0 \\ 0 & \text{si} \quad n = 0 \vee s < n \vee s > n \cdot k \\ \sum_{i=1}^{k} dadosSumaG(n-1, s-i, i) & \text{si no} \end{cases}$$

Llamando a la función con los parámetros $dadosSumaG(n, s, k)$ se obtienen la cantidad de combinaciones buscada.

c)

d)

e)