Tarea 2

Profesor: Pablo Barceló - **Auxiliares:** Bernardo Subercaseaux, Javier Oliva
Ayudantes: Joaquin Cruz, Heinich Porro, Lucas Torrealba, Florencia Yañez
Alumno: Sebastián Sepúlveda A.

Soluciones

P1 Asumiremos distribuciones de probabilidad uniformes.
Sea $I_n = 1, \ldots, n$ y $\pi = \pi_1, \ldots, \pi_n$ una permutación de I_n . Se dice que p es punto fijo de π si $\pi_p = p$. Lo que nos interesa es calcular la probabilidad de obtener k puntos fijos y la esperanza de la cantidad de puntos fijos de una permutación π cualquiera de I_n . Para esto procederemos de la siguiente manera:
Definimos $D_{k,n}$ como la cantidad de permutaciones π en I_n que tienen k puntos fijos.

P1.1 Indique $D_{0,0}$, $D_{0,1}$ y demuestre que $D_{0,n} = (n-1)(D_{0,n-1} + D_{0,n-2}) \forall n \geq 2$.

Dada la definición de $D_{k,n}$, para $D_{0,n}$ tenemos que $0! = 1$ pues es la cantidad de permutaciones en un conjunto sin elementos. Como $\pi_0 \neq 1$ entonces $D_{0,0} = 1$. Para $D_{0,1}$ tenemos $1! = 1$ permutaciones, como $\pi_1 = 1$ entonces $D_{0,1} = 0$.
Vamos a demostrar la fórmula mediante un razonamiento combinatorio. Sea $a = \{a_1, \ldots, a_n\}$ una permutación que no tiene puntos fijos en $\{1, 2, \ldots, n\}$, osea $a_1 \neq 1$, por ejemplo, donde tenemos $(n-1)$ posibilidades para hacer eso, para una mejor ilustración asumimos que $a_1 = 2$. Ahora definimos d_n como la cantidad de permutaciones que no tienen puntos fijos, lo que significa que $D_{0,n} = (n-1)d_n$. Tenemos dos posibilidades:

- $a_2 = 1$ lo cual significa que $a = \{2, 1, a_3, \ldots, a_n\}$ donde $\{a_3, a_4, \ldots, a_n\}$ no deben ser puntos fijos de $\{3, 4, \ldots, n\}$ lo cual es exactamente $D_{0,n-2}$
- $a_2 \neq 1$ con $\{a_2, a_3, \ldots, a_n\}$ una permutación que no tiene puntos fijos en $\{1, 3, 4, \ldots, n\}$ lo cual es $D_{0,n-1}$

Al considerar ambos casos, tenemos que $d_n = D_{0,n-1} + D_{0,n-2}$ y por tanto tenemos que

$$D_{0,n} = (n-1)(D_{0,n-1} + D_{0,n-2})$$

P1.2 Usando inducción y el resultado anterior, demuestre que $D_{0,n} = nD_{0,n-1} + (-1)^n \forall n \geq 1$

Caso Base: $D_{0,n} = 1 \cdot D_{0,0} + (-1)^0 = 1 \cdot 1 - 1 = 0$
Hipotesis inductiva: $\forall k \leq n$ se cumple la propiedad $D_{0,k} = kD_{0,n-1} + (-1)^n$
PDQ: $D_{0,n+1} = (n+1)D_{0,n} + (-1)^{n+1}$
Para ahorrar notación definimos $a_n = D_{0,n}$. Sabemos por el paso anterior que:

$$\begin{aligned} a_{n+1} = n(a_n + a_{n-1}) &= na_n + na_{n-1} \quad (*)\ HI \\ &= na_n + n\left(\frac{a_n - (-1)^n}{n}\right) \\ &= na_n + a_n + (-1)(-1)^n \\ &= (n+1)a_n + (-1)^{n+1} \end{aligned}$$

Donde en $(*)$ se despejó a_{n-1} de la hipotesis inductiva, pues $a_n = n(a_{n-1}) + (-1)^n$

P1.3 Divida la anterior ecuación de recurrencia por $n!$ y encuentre una fórmula para $D_{0,n}$ que no sea de recurrencia. Usando este resultado, obtenga $D_{k,n}$ para $k \in 0, ..., n$.

Como en el paso anterior definimos $a_n = D_{0,n}$

$$\begin{aligned}
a_n &= na_{n-1} + (-1)^n \quad / \ : n! \\
\Leftrightarrow \frac{a_n}{n!} &= \frac{na_{n-1}}{n!} + \frac{(-1)^n}{n!} \\
\Leftrightarrow \frac{a_n}{n!} &= \frac{a_{n-1}}{n-1!} + \frac{(-1)^n}{n!} \\
\Leftrightarrow \frac{a_n}{n!} - \frac{a_{n-1}}{(n-1)!} &= \frac{(-1)^n}{n!} \\
\Rightarrow \frac{a_{n-1}}{(n-1)!} - \frac{a_{n-2}}{(n-2)!} &= \frac{(-1)^{n-1}}{(n-1)!} \\
&\dots
\end{aligned}$$

Se hace una sumatoria de los elementos desde n hasta 1 quedando:

$$\begin{aligned}
\sum_{1\le i\le n} \frac{a_j}{j!} - \frac{a_{j-1}}{(j-1)!} &= \sum_{1\le i\le n} \frac{(-1)^i}{i!} \\
\Leftrightarrow \frac{a_n}{n!} - \frac{a_0}{0!} &= \sum_{1\le i\le n} \frac{(-1)^i}{i!} \\
\Leftrightarrow \frac{D_{0,n}}{n!} - D_{0,0} &= \sum_{1\le i\le n} \frac{(-1)^i}{i!} \\
\Leftrightarrow D_{0,n} &= n!\left(\sum_{1\le i\le n} \frac{(-1)^i}{i!} + 1\right) \\
\Leftrightarrow D_{0,n} &= n!\left(\sum_{0\le i\le n} \frac{(-1)^i}{i!}\right)
\end{aligned}$$

Para calcular $D_{k,n}$ ocupamos lo que sabemos de $D_{0,n}$ que es la cantidad de permutaciones que no tienen punto fijo en I_n. Para $D_{k,n}$ deducimos que necesitamos k puntos fijos de un conjunto de n y los otros puntos no deben ser fijos, es decir, deben ser de la forma $D_{0,n-k}$. Esto lo escribimos como:

$$\begin{aligned}
D_{k,n} = \binom{n}{k} D_{0,n-k} &= \frac{n!}{(n-k)!k!}(n-k)! \sum_{j=0}^{n-k} \frac{(-1)^j}{j!} \\
&= \frac{n!}{k!} \sum_{j=0}^{n-k} \frac{(-1)^j}{j!}
\end{aligned}$$

P1.4 Calcule la probabilidad de tener k puntos fijos y la esperanza de la cantidad de puntos fijos con n fijo. *Hint:* utilice las variables aleatorias X como la cantidad de puntos fijos y $X_k = [k \text{ es punto fijo}]$

Para calcular $P(X = k)$, ocupamos la definición de probabilidad, donde:
casos favorables: Cantidad de permutaciones que tienen punto fijo igual a k son $(D_{k,n})$
casos totales: Cantidad de permutaciones en un intervalo $(n!)$
Luego :

$$P(X = k) = \frac{1}{n!} D_{k,n} = \frac{1}{n!} \frac{n!}{k!} \sum_{j=0}^{n-k} \frac{(-1)^j}{j!}$$
$$= \frac{1}{k!} \sum_{j=0}^{n-k} \frac{(-1)^j}{j!}$$

Para calcular el valor esperado definimos X como la cantidad de puntos fijos, con n fijo. Definiendo también $X_k = [k \text{ es punto fijo}]$ podemos descomponer X como $X_1 + X_2 + \ldots + X_n$ donde X_k es una *indicatriz*:

$$X_k = \begin{cases} 1 & \text{si k es punto fijo} \\ 0 & \text{si no} \end{cases}$$

Ahora tenemos que:

$$E(X) = E\left(\sum_{0 \le i \le n} X_i \right) = \sum_{0 \le i \le n} E(X_i)$$
$$= \sum_{0 \le i \le n} P(X_k = 1)$$

$P(X_k = 1)$ está definido por:
casos favorables: $(n-1)!$ maneras de fijar un punto en particular.
casos totales: Cantidad de permutaciones en un intervalo $= n!$
Por tanto, $P(X_k = 1) = \frac{1}{n}$. Finalmente:

$$\sum_{0 \le i \le n} P(X_k = 1) = \sum_{0 \le i \le n} \frac{1}{n} = n \frac{1}{n} = 1$$

Concluimos que $E(X) = 1$

P1.5 ¿Qué ocurre con las probabilidades cuando $n \to \infty$? ¿Y con la esperanza? Calcule la esperanza de otra forma y concluya que $\sum_{k=0}^{n-1} \sum_{j=0}^{n-k-1} \frac{(-1)^j}{j!k!} = 1$

Cuando $n \to \infty$ notemos que:

$$\begin{aligned} P(X=k) &= \frac{1}{k!} \sum_{j=0}^{n-k} \frac{(-1)^j}{j!} \xrightarrow{n\to\infty} \frac{1}{k!e} \\ &= 1^k \frac{e^{-1}}{k!} \end{aligned}$$

Donde la última expresión es una variable de Poisson de parámetro $\lambda = -1$, por lo que se deduce que $X \sim Poisson(-1)$.
Luego, para la esperanza:

$$\begin{aligned} E(X) = \sum_{k=0}^{n} kP(X=k) &= \sum_{k=0}^{n} k \left(\frac{1}{k!} \sum_{j=0}^{n-k} \frac{(-1)^j}{j!} \right) && \text{Para k=0 la expresion es 0} \\ &= \sum_{k=1}^{n} \frac{1}{(k-1)!} \sum_{j=0}^{n-k} \frac{(-1)^j}{j!} \\ &= \sum_{k=1}^{n} \frac{1}{(k-1)!} \sum_{j=1}^{n-k} \frac{(-1)^j}{j!} \cdot 1 && \text{realizamos cambio de indices} \\ &= \sum_{k=0}^{n-1} \frac{1}{k!} \sum_{j=0}^{n-k-1} \frac{(-1)^j}{j!} \end{aligned}$$

Donde en el último paso se usó un cambio de indices en las sumatorias. Por la pregunta anterior, sabemos que $E(X) = 1$, por lo que concluimos que

$$\sum_{k=0}^{n-1} \sum_{j=0}^{n-k-1} \frac{(-1)^j}{k!j!} = 1$$

P1.6 Estudie la probabilidad de tener k puntos fijos con n fijo mediante una simulación. Entregue un gráfico variando k y otro variando n. ¿Qué puede decir?

Para estudiar el comportamiento de la probabilidad variando k y n ocupamos el resultado de la probabilidad obtenido en el punto 4.

$$P(X=k) = \frac{1}{k!} \sum_{j=0}^{n-k} \frac{(-1)^j}{j!}$$

La función con la cual graficaremos va a ser de la siguiente manera:

```
def p_n(k,n):
  j = np.arange((n-k).all()) #fijamos el rango de j, que será de 0 haste n-k
  N, J = np.meshgrid(n,j) #para cuando variemos k, cambiamos N y n por K y k respectivamente
  val = ( 1 / (misc.factorial(k)) )*((-1)**(J))/misc.factorial(J)
  return np.sum(val, axis=0)
```

Código 1: Funcion que retorna probabilidad, según valores de k o n

Al generar el gráfico que varia los valores de n y mantiene fijo k (ver Figura 1), observamos que la probabilidad se mantiene constante mientras mayor es el intervalo I_n, es decir, si deseamos la misma cantidad de puntos fijos la probabilidad de obtenerlos siempre será la misma. Para este gráfico se observo para $k = 1, 3, 10$ puntos fijos, y nos damos cuenta que a medida que k es mayor, hay menos probabilidad de encontrar esa cantidad de puntos fijos, lo que es acorde a lo que obtenemos con la esperanza.

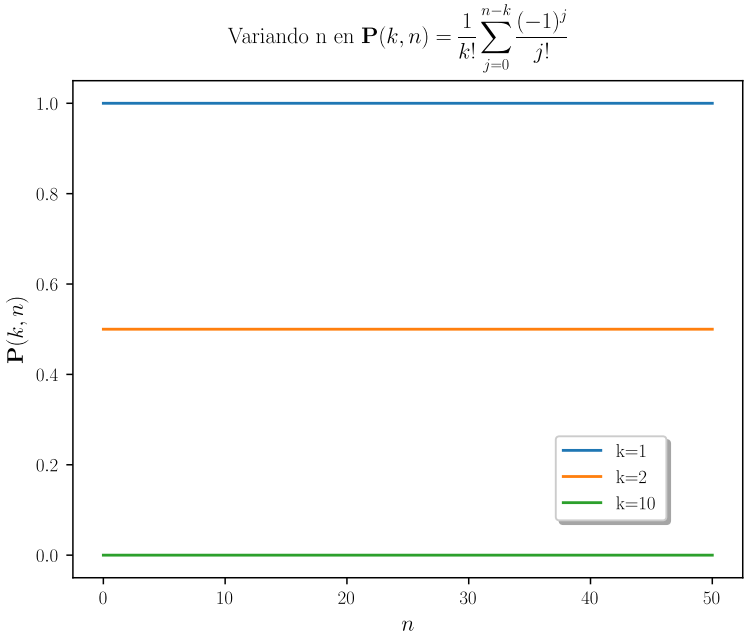


Figura 1: Gráfico variando n y manteniendo k constante

Entre tanto el gráfico que varia los valores de k y mantiene fijo n (ver Figura 2), obtenemos una curva que tiene un punto máximo y luego decae exponencialmente y se mantiene constante en 0. Esto nos señala que la probabilidad de tener k puntos fijos es máxima sólo cuando $k \in [0, 1]$ aproximadamente, o cuando queremos poca cantidad de puntos fijos ($k < 5$) donde, a pesar que no es máxima, la probabilidad es mayor a 0,5. Mientra que, por el contrario, para $k > 5$ la probabilidad tener k puntos fijos se aproxima a 0. El resultado es similar al obtenido en el gráfico anterior, además se obtiene que la probabilidad de que la cantidad de puntos fijos sea igual a n es $P(X = n) = 0$ aproximadamente.

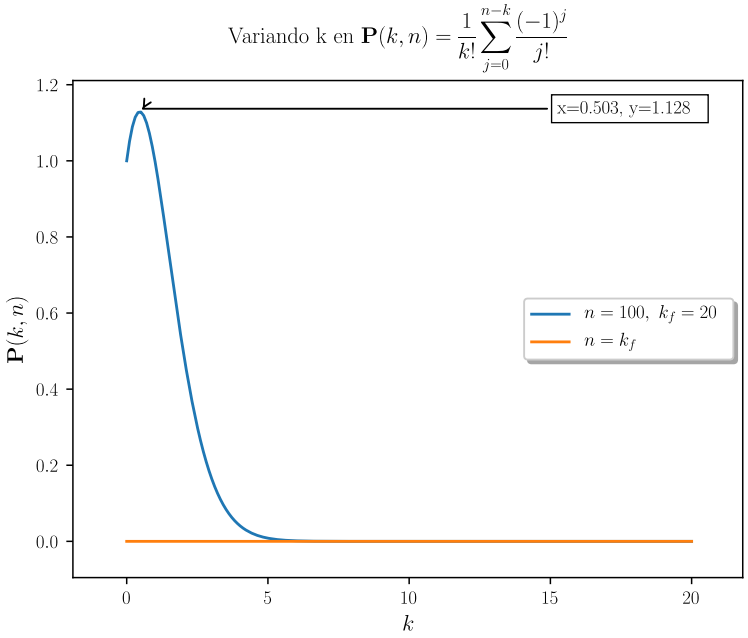


Figura 2: Gráfico variando k y manteniendo n constante

P2 Sea $V = \{a, b\}$ un conjunto de nombres de variables, $C = \{1, 2\}$ subconjunto de los naturales (Se pueden considerar ambos como conjuntos de símbolos). Se define $\mathcal{EXP}$ como un conjunto de expresiones sobre $\Sigma = +, *, =, (,) \cup V \cup C$ como:

- **Regla Base:** Si $a \in V \cup C$ entonces $a \in \mathcal{EXP}$
- **Regla 2:** Si $a, b \in \mathcal{EXP}$ entonces $(a + b) \in \mathcal{EXP}$
- **Regla 3:** Si $a, b \in \mathcal{EXP}$ entonces $(a * b) \in \mathcal{EXP}$
- **Regla 4:** Si $a, b \in \mathcal{EXP}$ entonces $(a == b) \in \mathcal{EXP}$

Nota: Para las siguientes preguntas vamos a considerar en ocasiones $\epsilon \in V \cup C$ y $\sigma \in \{+, *, ==\}$

P2.1 Sea a_n la cantidad de expresiones en $\mathcal{EXP}$ distintas con largo n. Entregue una relación de recurrencia para a_n

Para encontrar a_n primero nos damos cuenta de cómo se construye cada elemento en $\mathcal{EXP}$. Las reglas bases de $\mathcal{EXP}$ definen strings de largo $\{1, 5, 6\}$ donde la regla base permite obtener las combinaciones posibles para $\{(a + b), (a * b), (a == b)\}$ que son $\{4^2, 4^2, 4^2\}$ respectivamente. Luego, cada largo tiene su respectiva combinación, es decir se generan las tuplas de la forma (n, a_n) como $\{(1, 4), (5, 4^2 \cdot 2), (6, 4^2)\}$, donde es importante resaltar que dado a que $n = 5$ tiene 2 maneras de obtenerse, entonces se debe multiplicar por 2 el resultado de a_n. Entonces, para formar la recurrencia, debemos considerar que para cada n el valor de a_n puede estar en 3 casos:

- casos bases: $[n = 1 \Rightarrow a_n = 4], [n = 5 \Rightarrow a_n = 4^2 \cdot 2], [n = 6 \Rightarrow a_n = 4^2]$. Notar que los casos $n < 6$, con $n \notin \{1, 5, 6\}$ implican $a_n = 0$
- caso $(n > 6)$ se divide en tres:
  - caso a_{n-3}, asumiendo que la expresion es de la forma $(e_1\{+, *\}e_2)$, en donde anotamos el largo de e_1 como i, y por tanto el largo de e_2 como $n-3-i$. Luego notamos que e_1 y e_2 poseen largos que pueden ya pudieron ser evaluados, es decir que son obtenidos con a_i y a_{n-3-i} respectivamente. Además, tenemos que si $e_1 \neq e_2$ entonces $e_1+, *e_2 \neq e_2+, *e_1$, por lo que la expresión se puede ordenar de 2 formas distintas. Por tanto nos quedaría la expresión:$= (a_i + a_{n-3-i}) \cdot 2$

$$a_{n-3} = \begin{cases} 0 & si \quad (i \wedge (n-3-i)) \notin \{1, 5, 6\} \\ (a_i + a_{n-3-i}) \cdot 2 \cdot 2 & si \quad e_1 \neq e_2 \\ (a_i + a_{n-3-i}) \cdot 2 & si \quad e_1 = e_2 \end{cases}$$

  - caso a_{n-4}, asumiendo que la expresion es de la forma $(e_3\{==\}e_4)$, en donde asumimos que e_3 tiene largo j, y por tanto e_4 tien largo $n-4-j$, obtenemos un resultado similar al anterior:

$$a_{n-4} = \begin{cases} 0 & si \quad (i \wedge (n-4-i)) \notin \{1, 5, 6\} \\ (a_j + a_{n-4-j}) \cdot 2 & si \quad e_1 \neq e_2 \\ (a_j + a_{n-4-j}) & si \quad e_1 = e_2 \end{cases}$$

De esta forma definimos la recursión como:

$$a_n = a_{n-3} + a_{n-4} \tag{1}$$

P2.2 Programe una función cantidad que reciba como parámetro un natural n y entregue a_n

Para realizar el algoritmo `cantidad` que calcula a_n se opta por 1 idea de 3 que se complementan. Las ideas son:

1. Crear arbol binario con dos llaves, donde la primera llave es el largo del string, y la segunda llave es cuantas maneras distintas puede aparecer una llave con caracteristicas semejantes. Por ejemplo $\{+, *, ==\}$ son nodos $(3, 2)$, $(3, 2)$ y $(4, 1)$ respectivamente (notar que se consideran los parentesis en el conteo), y la misma idea es para los elementos tipo ϵ, pero sin considerar parentésis. El problema con esta idea era la implementación de las funciones para comparar entre arboles, la consideración de parentesis, entre otros.

2. La segunda opción fue elaborar dos matrices, una que almacenara los largos posibles que cumplan las condiciones de pertenecer a $\mathcal{EXP}$, y otra que almacenara las combinaciones posibles de tener un cierto largo. En

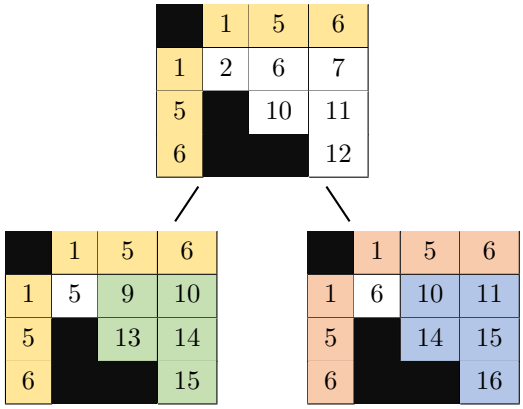


Figura 3: Matriz de largos bases generadora de matrices elementos (+3, izquierda) o (+4, derecha)

la **Figura 3** se muestra una representación de la matriz con los valores posibles sumando los largos bases, es decir $\{1, 5, 6\}$. La matriz generada, se divide en dos posibles resultados futuros: cuando el operador es $\{+, *\}$ *(matriz de la izquieda)* entonces a cada elemento de la matriz que no tiene borde amarillo se le suma 3 (pues se suman 3 nuevos carácteres a la expresión, considerando los paréntesis), cuando el operador es $\{==\}$ *(matriz de la derecha)* se le suma 4.

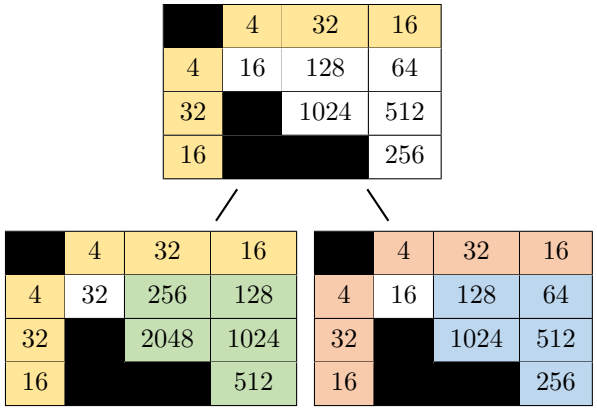


Figura 4: matriz de combinaciones posibles

Mientras que la segunda matriz de combinaciones posibles es como se muestra en **Figura 4**, donde *la matriz de la izquierda* representa cuando el operador principal (entiéndase por principal aquel operador que contiene a dos expresiones y no es hijo de otro operador) es $\{+, *\}$ entonces se multiplica por 2 los elementos de la

matriz, pues hay dos maneras de generarlo. Para *la matriz de la derecha*, el operador principal es {==} por lo que se deja tal cual la matriz, pues hay una forma solamente de formar una expresión así.

Luego, es importante recalcar que el resultado final depende de la posición en la que se ubique la combinación, pues para elementos en la posición (i, i) con $i \in \{1, 5, 6\}$ no hay forma de conmutar los indices, pues son igual, en cambio para combinaciones en la posición (i, j) tenemos que $(i, j) \neq (j, i)$ pues nos interesa el orden, por tanto hay que multiplicar por 2 estos resultados en el output.

El problema con esta idea era generar ambas matrices, y para valores de n más grades ocupa mucha memoria. Además, en la búsqueda del elemento de la matriz es necesario consultar primero la matriz de largo, y luego consultar la posición del largo en la matriz de combinaciones. Por último, si se necesita agrandar la matriz de largos, también es necesario agrandar la matriz de combinaciones, por lo que se vuelve más engorroso.

3. La solución a los dos problemas anteriores fue ocupar la idea 2 pero uniendo ambas matrices en una, formando tuplas con el largo del string y con el número de combinaciones. En el siguiente ejemplo **(Ver Figura 5)** la matriz "padre" es la que genera a las matrices que tienen los largos de expresiones posibles con su combinaciones respectivas, semejante a la idea anterior, la matriz de la izquierda suma a los elementos de adentro 3, pues representa a una posible *sub expresión* :

	$\{1,4\}$	$\{5,32\}$	$\{6,16\}$
$\{1,4\}$	$\{2,16\}$	$\{6,128\}$	$\{7,64\}$
$\{5,32\}$		$\{10,1024\}$	$\{11,512\}$
$\{6,16\}$			$\{12,256\}$

	$\{1,4\}$	$\{5,32\}$	$\{6,16\}$
$\{1,4\}$	$\{5,32\}$	$\{9,256\}$	$\{10,128\}$
$\{5,32\}$		$\{13,2048\}$	$\{14,1024\}$
$\{6,16\}$			$\{15,512\}$

	$\{1,4\}$	$\{5,32\}$	$\{6,16\}$
$\{1,4\}$	$\{6,16\}$	$\{10,128\}$	$\{11,64\}$
$\{5,32\}$		$\{14,1024\}$	$\{15,512\}$
$\{6,16\}$			$\{16,256\}$

Figura 5: Matriz generadora, versión tuplas

La ventaja de esta idea es que nos genera matrices que son más fáciles de manejar y de expandir, y que al momento de consultar obtenemos dos datos en el mismo instante. Además, se logra apreciar de mejor manera más formas de encontrar combinaciones de un número. Por ejemplo, en la idea anterior existía el problema que al momento de expandir la matriz de combinaciones **(Ver Figura 6)** se generaban matrices con un largo mayor al doble de la matriz "padre", lo que dificultaba aún más la búsqueda de la combinación correspondiente.

Con la idea de generar las matrices con tuplas se elimina esta dificultad y sólo basta con buscar el largo del elemento en la matriz, obtener la combinación con la cual se obtiene ese largo (tiempo constante, pues es consultar el otro elemento de la tupla) e ir acumulando el resultado junto con todos los largos posibles, con sus respectivas combinaciones.

De esta manera, con la última opción es posible encontrar a_n para cada n ingresado, y el algoritmo logra ser recursivo, pues en los casos distintos a los casos bases, es suficiente buscar en la matriz de tuplas, para los valores de $n-3$ (asumiendo que el nodo puede ser $\{+, *\}$) o $n-4$ (asumiendo que el nodo puede ser {==}), y sumar ambos resultados. En caso de no estar en la matriz por ser mayor a los elementos que están dentro, entonces la matriz se debe agrandar hasta que el elemento que buscamos sea menor al mayor elemento de la matriz (`mat[m-1][m-1]`, con `m: largo de la matriz`).
Las matrices generadas además tienen la ventaja de estar ordenadas de menor a mayor por cada fila, por lo que el algoritmo de búsqueda se puede manejar más fácil. En la implementación realizada en `cantidad` se realiza el

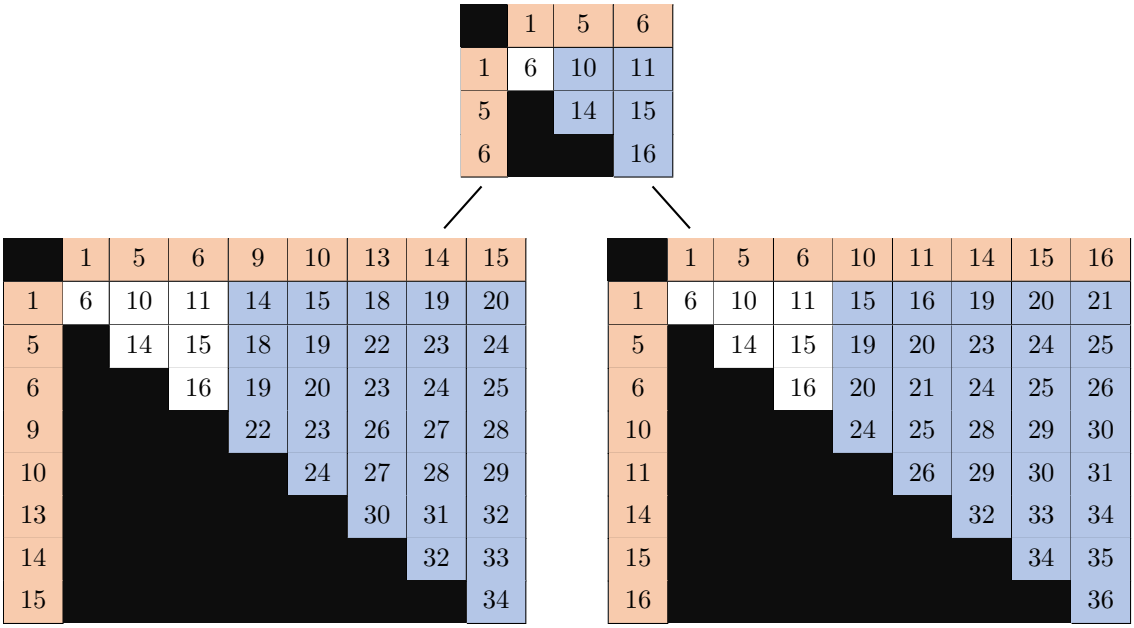


Figura 6: Matrices de combinaciones posible para expresiones con signo "=="

siguiente algoritmo de búsqueda **(Ver Código 2)**, donde `mat` es la matriz generada por las combinaciones de los distintos largos **(revisar Figura 5)**, más el factor `pr` (3 o 4) según la busqueda que se esté realizando (asumiendo que la expresión tiene un operador principal $\{+, *\}$ entonces $pr = 3$, si no entonces tiene oper. ppal. $\{==\}$ y $pr = 4$); `x` es $n - 3$ o $n - 4$, dependiendo de cual buscamos y *col* representa la columna donde debemos empezar a buscar (columnas pintadas en Figura 5):

Código 2: Búsqueda en matriz

```
def search(mat, x, pr,col):
  n= len(mat[0]) - 1
  i = 0
  j = n
  arr=[]
  while ( i <= n and j >= col ): # se comienza en el punto (col,n-1)
    v = mat[i][j]
    if (v[0] == x):
      if pr==3: v[1]*=2
      arr.append(v)
    if (v[0] > x ): j -= 1
    else:  # if v[0] < x
      i += 1
  return arr
```

Finalmente, las otras funciones realizadas para complementar a la función `cantidad` serán explicadas brevemente pues repiten el procedimiento ya explicado anteriormente:

- `triSup(casos_bases)`: Recibe una lista ($\mathcal{L}$) que representa la columna en amarillo en **Figura 5** y genera una matriz triangular superior $\mathcal{M}$ con tuplas, donde cada tupla $(i, j) \in \mathcal{M}$ representa a $(f_1 + c_1, f_2 * c_2)_(i, j)$, con

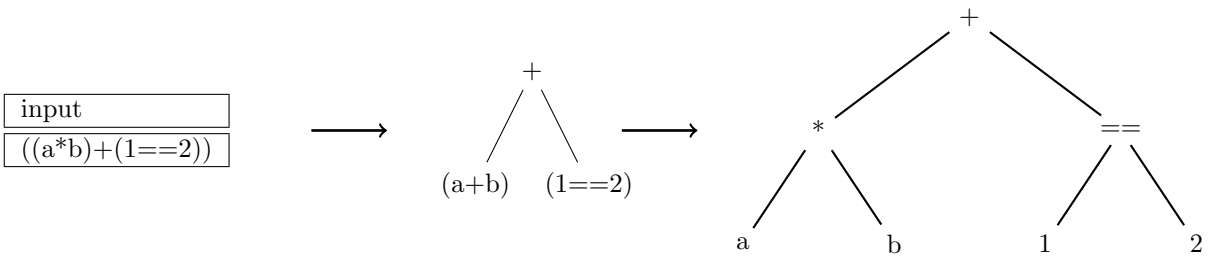
$(f_1, f_2)_i \in \mathcal{L}$, y $(c_1, c_2)_j \in \mathcal{L}$ de la lista transpuesta.

- `agrandar(matriz, paramétro, columna)`: Matriz representa la matriz que se quiere agrandar, parametro es (3 o 4) dependiendo si se quiere sumar 3 o 4 a los nuevos elementos bases de la futura matriz generadora, y columna representa desde cuál columna es necesario agregar elementos.
- `supMat(matriz, casos_bases, columna)`: combina los procesos de las os funciones anteriores, retornando la matriz y la lista de `casos_bases` agrandados.
- `ctd(n,parametro)`: Realiza el trabajo de cantidad, pero dependiendo del parametro (3 o 4). Con 3 asumimos que la expresión se puede formar con el simbolo "+,*", con 4 asumimos que se puede formar con "==".
- `cantidad(n)`: Ocupa `ctd(n,3)` y `ctd(n,4)` y retorna la suma de ambos, resolviendo la recursión.

Comentario: En las pruebas que se realizó al programa sólo es posible llegar a resultados correctos hasta $n = 20$. Esto es debido a una mala generalización del problema, eliminando, al agrandar la matriz, elementos que igual hay que sumar al resultado final. El resultado correcto se obtiene eliminando los sectores que se revisaron, y considerando el sector que no se ha revisado.

P2.3 Programe una función `esExp` que reciba como parámetro un string y entregue verdadero si pertenece a $\mathcal{EXP}$ o falso si no.

Dado el formato del string, ocupamos el concepto de árbol cartesiano para almacenar los valores de las posiciones. La idea es ir almacenando las operaciones $(+, *, ==)$ en nodos, y las hojas van a ser expresiones basicas, es decir $\{a, b, 1, 2\}$. La construcción del árbol sigue la siguiente secuencia:

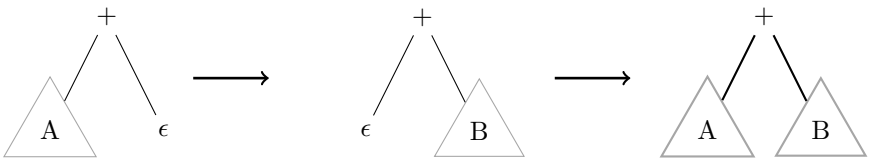


Notar que lo unico necesario para formar el árbol es detectar las operaciones $(+, *, ==)$ en la expresión, pero gracias a la permanencia de los paréntesis esto se traduce a detectar en qué nivel estamos y cuál es la operación principal, es decir, aquella que es nodo padre de las otras dos expresiones. En el ejemplo anterior podemos detectar que primero es $+$, y luego $*$ y $==$ para las otras dos expresiones.

De esta manera, nuestros casos bases son cuando:

- Expresiones de largo igual 1, es decir $\epsilon \in V \cup C$
- Expresiones con $|s| >= 5$ con $|s|$: largo del string, donde las hojas deben ser $\epsilon \in V \cup C$ y los nodos deben ser $\{+, *, ==\}$

Para generar la recurrencia es necesario observar los 3 subcasos del segundo punto:



Las ilustraciones anteriores representan las siguientes formas de expresiones, $(\text{Tree}\{+, *, ==\}\epsilon)$, $(\epsilon\{+, *, ==\}\text{Tree})$, $(\text{TreeA}\{+, *, ==\}\text{TreeB})$, respectivamente.
Para resolver el problema junto con los casos bases y los subcasos seguimos los siguientes pasos:

1. Crear la estructura `Tree`.
2. Crear las funciones `enNum, enOp` que nos devuelven `True` si la expresión está en $V \cup C$ o en $\{+, *, ==\}$ respectivamente.
3. Como se explicó anteriormente, para $|s| >= 5$ se cumple que los nodos son operadores y las hojas símbolos, por lo que para comprobar si el árbol genera una expresion que pertenezca a $\mathcal{EXP}$ creamos la función `isTree` que nos devuelve `True` si cumple pertenecer al conjunto. El caso base es que el nodo sea algún ϵ, si no, es un operador y cada sub árbol debe cumplir las condiciones ya mencionadas. Si no es ninguna de las anteriores entonces la expresión está mal escrita, devuelve `False`.
4. Se crea la función `Constructor` que dado un string nos devuelve un árbol cartesiano que representa al string entregado. Aquí se implementan los casos bases mencionados anteriormente. Los subcasos los resolvemos con la ayuda de los parentesis.

- En el subcaso del nodo con dos subarboles y el nodo con un caracter a la derecha y un subárbol a la izquierda iniciamos un contador que suma 1 según la cantidad de parentesis abiertos y resta 1 según la cantidad de parentesis cerrados desde el indice 1. Si el contador es 0 entonces el siguiente caracter es el "operador padre" y creamos un nodo padre con sus dos hijos, el substring de la izquierda y el substring de la derecha.
- En el subcaso del nodo con un caracter a la izquierda y un subarbol a la derecha no podemos iniciar de inmediato el contador de parentesis, pues un caracter ϵ no lleva. La solución es agregar una nueva condición, y considerar que en este caso el siguiente string será el nodo padre, pues es un operador

5. Finalmente se crea la función `esExp` el cual construye el árbol dado un string, y evaluamos si es árbol cartesiano de $\mathcal{EXP}$ con `isTree`

Nota: Se asume que se entrega un string sin espacios.

P2.4 Entregue un archivo `expresiones.{py,java,cpp}` que reciba por entrada estándar un string s e imprima 2 enteros a, b donde a es booleano que representa si $s \in \mathcal{EXP}$ y b es $a_{|s|}$