Apunte Único: Técnicas de diseño de algoritmos - Práctica 1

By naD GarRaz

última actualización 27/01/26 @ 23:28

Choose your destiny:

(click click 🖱 en el ejercicio para saltar)

Esta Guía 1 que tenés se actualizó por última vez:
27/01/26 @ 23:28
Escaneá el QR para bajarte (quizás) una versión más nueva:

Guía 1

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:

Repaso complejidad:

- *Big O Notation, la favorita de todos (cota superior)*

$$O(g(n)) = \{f(n) : \exists\, c > 0, n_0 > 0 \text{ tal que } 0 \leq f(n) \leq c \cdot g(n) \ \ \forall n \geq n_0\}$$

⚠ En criollo $f(n) = O(g(n))$ quiere decir que $f(n)$ no crece más rápido $g(n)$ en forma asintótica. ⚠

- *Big Omega Notation (cota inferior)*

$$\Omega(g(n)) = \{f(n) : \exists\, c > 0, n_0 > 0 \text{ tal que } 0 \leq c \cdot g(n) \leq f(n) \ \ \forall n \geq n_0\}$$

⚠ En criollo $f(n) = \Omega(g(n))$ quiere decir que $f(n)$ crece por lo menos igual de rápido que $g(n)$ en forma asintótica. ⚠

- *Big Theta Notation (cota inferior y superior)*

$$\Theta(g(n)) = \{f(n) : \exists\, c_1, c_2 > 0, n_0 > 0 \text{ tal que } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ \ \forall n \geq n_0\}$$

⚠ En criollo $f(n) = \Theta(g(n))$ quiere decir que $f(n)$ y $g(n)$ crecen igual en forma asintótica. ⚠

Una relación para tener en cuenta:

$$f(n) = \Theta(g(n)) \iff \big(f(n) = O(g(n)) \ \wedge \ f(n) = \Omega(g(n))\big)$$

Estos cosos me parece maravillosos

$$\begin{array}{rcl} f(n) = O(g(n)) & \iff & \lim\limits_{n\to\infty} \frac{f(n)}{g(n)} < \infty \\ f(n) = \Omega(g(n)) & \iff & \lim\limits_{n\to\infty} \frac{f(n)}{g(n)} > 0 \\ f(n) = \Theta(g(n)) & \iff & 0 < \lim\limits_{n\to\infty} \frac{f(n)}{g(n)} \leq k < \infty \end{array}$$

¡Divide y reinarás!

- Un problema de `divide and conquer` de tamaño n tiene un "costo":

$$T(n)$$

Calcular el costo suele ser algo complicado, en lo cual hay que considerar:

1. El problema se divide en a subproblemas de tamaño máximo $\frac{n}{b}$, siempre que $\frac{n}{b} > n_0$. Este n_0 será un valor fijado por el usuario o un valor que sea compatible con el subproblema.
2. El costo de hacer el `divide` y `combine`. Estos dos bloques caracterizan las dos papas más grandes del $T(n)$.
3. El costo de los a subproblemas:

$$a \cdot T(\tfrac{n}{b})$$

Como calcular esto es la mismísima 💩 misma, se hacen unas aproximaciones falopa para dar un *uneducated guess*, entre las cuales están:

- Arranco utilizando una función que nadie pidió pero bueh, $g(n)$:

$$g(n) \geq T(n)$$

- Hardcodeo $n_0 = 1$ y si no te gusta, porque te parece berreta:

$$T(\tfrac{n}{n_0}) = T'(n)$$

  y listo, uso $T'(n)$ como si fuera la otra.
- Cota polinómica: $c' \cdot n^d$ un monomio de grado d, se toma como una cota superior al costo de dividir en subproblemas y combinar los resultados para un problema de tamaño n.

Con esto me armo $g(n)$:

$$g(n) = \begin{cases} c = \max\{c', T(1)\} & \text{si} \quad n = 1 \\ a \cdot g(\frac{n}{b}) + c \cdot n^d & \text{si} \quad n > 1 \end{cases}$$

Sin perdida de generalidad elijo un $n \overset{\bigstar^1}{=} b^k$ y ahora voy a calcular la cota:

$$\begin{aligned}
T(n) \quad &\leq \quad g(n) = a \cdot g(\tfrac{n}{b}) + b \cdot n^d \\
&\overset{\bigstar^1}{\iff} \quad g(b^k) = a \cdot g(b^{k-1}) + b \cdot b^{k \cdot d} \\
&\overset{!}{\iff} \quad g(b^k) = a \cdot \left(a \cdot g(b^{k-2}) + b \cdot b^{k \cdot d}\right) + b \cdot b^{k \cdot d} \\
&\iff \quad g(b^k) = a^2 \cdot g(b^{k-2}) + a \cdot b \cdot b^{k \cdot d} + b \cdot b^{k \cdot d} \\
&\overset{!}{\iff} \quad g(b^k) = a^3 \cdot g(b^{k-3}) + a^2 \cdot b \cdot b^{k \cdot d} + a \cdot b \cdot b^{k \cdot d} + b \cdot b^{k \cdot d} \\
&\overset{!}{\iff} \quad \dots \\
&\overset{!}{\iff} \quad g(b^k) = a^j \cdot g(c^{k-j}) + b \cdot \sum_{i=0}^{j-1} a^i c^{(k-i) \cdot d} \quad \bigstar^2
\end{aligned}$$

La expresión en $\bigstar^2$ llega al caso base $g(1) = b$ cuando $k = j \to g(b^0)$ y el término se puede *absorber* en la sumatoria:

$$\begin{aligned}
T(n) \leq g(b^k) = a^j \cdot g(b^{k-j}) + b \cdot \sum_{i=0}^{j-1} a^i b^{(k-i) \cdot d} \quad &\overset{j = k}{\iff} \quad g(b^k) = a^k \cdot \overset{b}{\uparrow}(1) + b \cdot \sum_{i=0}^{k-1} a^i b^{(k-i) \cdot d} \\
&\overset{!}{\iff} \quad g(b^k) = b \cdot b^{kd} \cdot \sum_{i=0}^{k} a^i b^{-di} \\
&\underset{\Rightarrow k = \log_b(n)}{\overset{\text{si } n = b^k}{\iff}} \quad g(n) = b \cdot n^d \cdot \sum_{i=0}^{\log_b(n)} a^i b^{-di}
\end{aligned}$$

El resultado es una expresión para una cota superior de $T(n)$ sin una expresión recursiva:

$$T(n) \leq \underbrace{b \cdot n^d}_{\substack{\text{costo de dividir} \\ \text{y combinar} \\ \text{un problema}}} \cdot \sum_{i=0}^{\log_b(n)} a^i b^{-di}$$

Caso con $a = 1$ *y* $d = 0$: División en 1 subproblema y costo de dividir y combinar un problema es b:

$$b \cdot n^d = b \implies T(n) \leq b \cdot \sum_{i=0}^{\log_b(n)} 1^i b^{0 \cdot i} = b \cdot \log_b(n) \in O(\log_b(n))$$

Caso con $d = 1$: Costo de dividir y combinar un problema es lineal, $b \cdot n$:

$$b \cdot n^1 = b \cdot n \implies T(n) \leq b \cdot n \sum_{i=0}^{\log_b(n)} (\tfrac{a}{b})^i$$

Esa sumatoria es convergente para $\frac{a}{b} < 1$:

$$T(n) = O(n)$$

Si $\frac{a}{b} > 1$ uso la expresión para la serie geométrica:

$$T(n) \leq bn\frac{(\frac{a}{b})^{\log_b(n+1)}-1}{\frac{a}{b}-1} = O\big(n \cdot (\tfrac{a}{b})^{\log_b n}\big) \overset{!!}{=} O\big(a^{\log_b n}\big) \overset{!!}{=} O\Big(a^{\frac{\log_a(n)}{\log_a(b)}}\Big) \overset{!!}{=} O\Big(n^{\frac{1}{\log_a(b)}}\Big) \overset{!!}{=} O\big(n^{\log_b(a)}\big)$$

Todos los pasos !! son propiedades y cambios de base de logaritmos.

$$T(n) = O\big(n^{\log_b(a)}\big)$$

- *Teorema Maestro*

$$T(n) = \begin{cases} a \cdot T(\frac{n}{b}) + f(n) & \text{si} \quad n > 1 \\ 1 & \text{si} \quad n = 1 \end{cases}$$

Estimación según $f(n)$:

1. Si

$$f(n) = O(n^{\log_b(a)-\varepsilon}) \text{ para algún } \varepsilon > 0 \implies \boxed{T(n) = \Theta\big(n^{\log_b(a)}\big)}$$

2. Si

$$f(n) = \Theta(n^{\log_b(a)}(\log(n))^k) \text{ para algún } k \geq 0 \implies \boxed{T(n) = \Theta\big(n^{\log_b(a)} \cdot (\log(n))^{k+1}\big)}$$

3. Si

$$f(n) = \Omega(n^{\log_b(a)+\varepsilon}) \text{ para algún } \varepsilon > 0 \quad \text{y} \quad af(\tfrac{n}{b}) \leq cf(n) \text{ para algún } c \in (0,1) \implies \boxed{T(n) = \Theta(f(n))}$$

- *Máximas. IMHO:*

(1) El Teorema Maestro es medio verga. Retrasó meses mi comprensión sobre el cálculo de la complejidad.

(2) Lo calculable sin Teorema Maestro se tiene que calcular sin Teorema Maestro.

(3) ¡No optimices las cosas! Siempre fuerza bruta primero.

(4) No hagas un programa compilable ¡No codeés!

(5) Codear en el teclado es distinto a *codear* en papel. Si el parcial es en papel no podés probar mil cosas hasta que alguna, de pedo pase tus patéticos tests.

(6) Si estás pensando en *sintaxis*: ¡Salí de ahí, Maravilla!

(7) Leer y escribir pseudocódigo debe ser algo más natural que leer y escribir `Python`

(8) El pseudocódigo es muy flexible, además de ser tu mejor amigo. Sí, por lo menos tenés un amigo.

(9) ¿Te dije algo sobre *no optimizar las putas cosas*? Solo hacelas funcionar.

Ejercicios de la guía:

---

1. (*MergeSort*)
Dado el algoritmo de *mergesort*, implementado en el siguiente código `Python` :

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    medio = len(arr) // 2
    mitad_izq = merge_sort(arr[:medio])
    mitad_der = merge_sort(arr[medio:])

    return merge(mitad_izq, mitad_der)
```

```python
def merge(izq, der):
    mergeados = []
    i = j = 0

    while i < len(izq) and j < len(der):
        if izq[i] < der[j]:
            mergeados.append(izq[i])
            i += 1
        else:
            mergeados.append(der[j])
            j += 1

    mergeados.extend(izq[i:])
    mergeados.extend(der[j:])
    return mergeados
```

1) Identificar qué líneas son el *divide*, cuáles son el *conquer* y cuáles son el *combine*.
2) ¿En cuántos subproblemas se divide?
3) ¿De qué tamaño son estos subproblemas?
4) ¿Cuál es el costo de combinar los resultados de los subproblemas?
5) Escribir la función $T(n)$ de manera recursiva.
6) Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

Descripción en leguaje natural del *Merge Sort*:
El algoritmo *mergeSort* toma un arreglo de n elementos $A[1..n]$, lo divide en 2 arreglos $B[1..m]$ y $C[m+1..n]$ y luego se llama recursivamente a la función, hasta que ya no pueda dividirse en el *caso base*.
El ordenamiento se realiza en la función auxiliar *merge* que recibe como parametros 2 arreglos y los va ordenando y *mergeando*.

1) En la `línea 5` tengo el *divide*.

   En las `líneas 6, 7` tengo el *conquer*. Es un nombre exótico para el llamado recursivo.

   En la `línea 9` tengo el *combine*, el `merge` me junta los resultados de las recursiones. La función `merge` entera es parte del *combine*.

2) Se divide en 2 *subproblemas*: $\begin{cases} \texttt{arr[:medio]} \\ \texttt{arr[medio:]} \end{cases}$. El parámetro a del Teorema Maestro será: $a = 2$.

3) Los inputs se cortan por la mitad, así que el tamaño será $\frac{n}{2}$. $\begin{cases} \lfloor \texttt{arr[:medio]} \rfloor & \rightarrow & \text{mitad izquierda} \\ \lceil \texttt{arr[medio:]} \rceil & \rightarrow & \text{mitad derecha} \end{cases}$.

   El parámetro c del Teorema Maestro será $c = 2$.

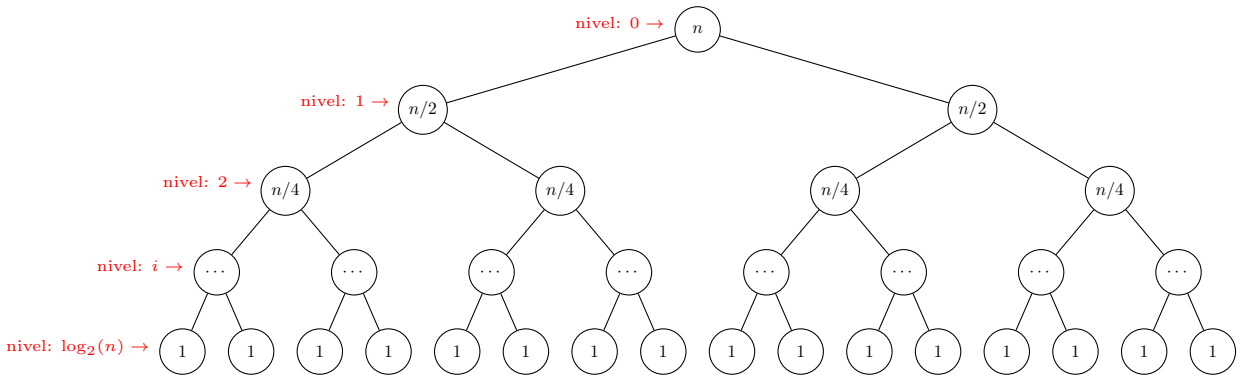
4) El costo de combinar los subproblemas será de $O(n)$. En cada llamado a `recorro` todos, en el peor de los casos, los elementos del `input` a `merge`.

   El total de operaciones: n comparaciones $+$ n inserciones $\rightarrow O(n)$

5) Con cada llamada a la función *mergeSort* se bifurca el árbol de recursión. El costo, *running time*, complejidad de cada subproblema viene en este caso dado por la función *merge*. La función para calcular el *running Time*:

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 1 \ \vee \ n = 0 \\ 2 \cdot T(\frac{n}{2}) + n & \text{si} \quad n > 1 \end{cases}$$

El árbol de recursión, tiene $\log_2(n)$ niveles:



Sumando los nodos de un mismo nivel siempre la suma de la complejidad es: n

$$\begin{cases} \text{nivel } 0 & \rightarrow & n \\ \text{nivel } 1 & \rightarrow & \frac{n}{2} + \frac{n}{2} = n \\ \text{nivel } 2 & \rightarrow & \frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4} = n \\ \quad \vdots & \quad \vdots & \vdots \end{cases}$$

Así que la complejidad si tengo $log_2(n)$ niveles debería ser:

$$T(n) \in \Theta(n \cdot \log(n))$$

La complejidad calculada usando el *teorema maestro* click click :

Estoy dividiendo en 2 subproblemas, $a = 2$ y el tamaño de los subproblemas se achica a la mitad, $c = 2$, respecto del problema anterior:

6) Caso 2 del Teorema Maestro:

$$\boxed{T(n) = \Theta(n \cdot log(n))}$$

Fin.

2. *BusquedaBinaria*
Dado el algoritmo de *búsqueda binaria*, implementado en el siguiente código `Python` :

```python
def busqueda_binaria(arr, objetivo, izq=0, der=len(arr) - 1):
    if izq > der:
        return False
    medio = (izq + der) // 2
    if arr[medio] == objetivo:
        return medio
    elif arr[medio] > objetivo:
        return busqueda_binaria(arr, objetivo, izq, medio - 1)
    else:
        return busqueda_binaria(arr, objetivo, medio + 1, der)
```

1) Identificar qué líneas son el *divide*, cuáles son el *conquer* y cuáles son el *combine*.
2) ¿En cuántos subproblemas se divide?
3) ¿De qué tamaño son estos subproblemas?
4) ¿Cuál es el costo de combinar los resultados de los subproblemas?
5) Escribir la función $T(n)$ de manera recursiva.
6) Determinar la complejidad del algoritmo utilizando el Teorema Maestro.

---

1)
- *Divide:* Cuando divido al `input` a la mitad en la línea 4 y también en la comparación de la línea 7, donde se decide por cual mitad seguir.
- *Conquer:* Llamada de recursión en las líneas 8 y 10.
- *Combine:* En este algoritmo no hay *combine*.

2) El árbol de recursión solo tiene una rama, porque hay solo una llamada recursiva en cada llamada. Cada nodo tendrá la mitad del tamaño del input anterior.

3) El tamaño de cada subproblema será de $\frac{n}{2} > n_0 = 1$.

4) En este algoritmo no hay un *combine* de unir los resultados como en el `merge sort`. La complejidad de devolver `true` o `false` será $O(1)$. Hay solo unas comparaciones y el cálculo de `medio`, todo eso es independiente de n, todo $O(1)$

5)
$$T(n) = \begin{cases} 1 & \text{si} \quad n \leq 1 \\ T(\frac{n}{2}) + 1 & \text{si} \quad n > 1 \end{cases}$$

Esa función recursiva equivale a contar la cantidad de nodos que tendrá el árbol de recursiones: Cada nodo tiene un costo 1 (o O(1), eh lo meesmo.) Por lo tanto sumo todo:

$$T(n) \in \Theta(\log_2(n))$$

6) El *Teorema Maestro* me estima la complejidad total de T(n) según el valor de $f(n)$.

En este caso $f(n) \in O(1)$ y como $a = 1$ y $b = 2$:

$$f(n) \in O(n^{\log_b(a)} \cdot \log^k(n)) \xrightarrow[k=0]{a=1,\, b=1} f(n) \in O(1)$$

Estoy en el caso 2 del teorema:

$$T(n) \in \Theta(n^{\log_b(a)} \cdot \log^{k+1}(n)) \xrightarrow[k=0]{a=1,\ b=1} \boxed{T(n) \in \Theta(\log(n))}$$

Fin.

---

3. *IzquierdaDominante*
Escriba un algoritmo con dividir y conquistar que determine si un arreglo de tamaño potencia de 2 es *más a la izquierda*, donde "más a la izquierda" significa que:

- La suma de los elementos de la mitad izquierda superan los de la mitad derecha.
- Cada una de las mitades es a su vez "más a la izquierda".

Por ejemplo, el arreglo $[8, 6, 7, 4, 5, 1, 3, 2]$ es "más a la izquierda", pero $[8, 4, 7, 6, 5, 1, 3, 2]$ no lo es.
Intente que su solución aproveche la técnica de modo que la complejidad del algoritmo sea estrictamente menor a $O(n^2)$.

⚠ Los casos bordes y yo que sé *son importantes*, pero no hay que obsesionarse con eso. Hay que hacer un algoritmo que tenga sentido, que haga lo que tiene que hacer. Los detalles quedarán para la implementación en la compu, pero la papa de la materia no está ahí, sino que en el pseudocódigo así se aprender a pensar algorítmicamente. ⚠

En presudocódigo:
Recibo un arreglo de $A[1..n]$. Luego parto a la mitad el arreglo y voy a comparar las 2 mitades. Pero si algo aprendí de *mergeSort* es que me conviene comparar las cosas cuando ya se dividieron hasta el último momento.

```
1   funcion izquierdaDominante(A[1..n])
2       si |A| = 1
3           ret true
4
5       m ← n/2
6       izq ← A[1..m]
7       der ← A[m+1..n]
8
9       // Si está todo bien, analizo la "izquierdominancia" de izq y luego de der
10      ret sumoYComparo(izq,der) ∧ izquierdaDominante(izq) ∧ izquierdaDominante(der))
11
12  funcion sumoYComparo(A[1..n], B[1..m])   // --> max(O(n), O(m))
13      sumaA ← ΣA[i] // O(n)
14      sumaB ← ΣB[i] // O(m)
15
16      ret sumaA > sumaB
17
```

Respecto a la complejidad:
Tengo 2 subproblemas por recursión. El tamaño del subproblema se divide entre 2. Puedo escribir al *costo o running time* del algoritmo:

$$T(n) = \begin{cases} 1 & si \quad n = 1 \ \vee \ \varnothing \\ 2 \cdot T(\frac{n}{2}) + n & si \quad n > 1 \end{cases}$$

Esta función tiene el mismo aspecto que *mergeSort* **1.**. La recursión genera un árbol de $\log_2(n)$ niveles y el costo de cada nivel es exactamente n. Por lo tanto:

$$T(n) \in \Theta(n \log(n))$$

Para el cálculo con Teorema maestro click click: Caso 2

$$f(n) = n \implies (n^{\log_b(a)}(\log(n))^k) \xrightarrow[k=0]{a=2,\ b=2} f(n) = n \implies \boxed{T(n) \in \Theta\big(n\log(n)\big)}$$

Fin.

---

4. *ÍndiceEspejo*

Tenemos un arreglo $a = [a_1, \ldots, a_n]$ de n enteros distintos (positivos y negativos) *en orden estrictamente creciente*. Queremos determinar si existe una posición i tal que $a_i = i$. Por ejemplo, dado el arreglo $a = [-4, -1, 2, 4, 7]$, $i = 4$ es esa posición.

Diseñar un algoritmo dividir y conquistar eficiente (cuya complejidad sea de un orden estrictamente menor que lineal) que resuelva el problema. Calcule y justifique la complejidad del algoritmo dado.

Una complejidad de orden menor a n, puede ser algo como $\sqrt{n}$ o muy probablemente algo como $\log(n)$, *because árboles.*

Lo primero que hago es cargar *binarySearch* en el , porque tiene una complejidad $\Theta(\log(n))$ y también funciona para arreglos ordenados, *mooooooy parecido.*

Sobre este algoritmo en particular:

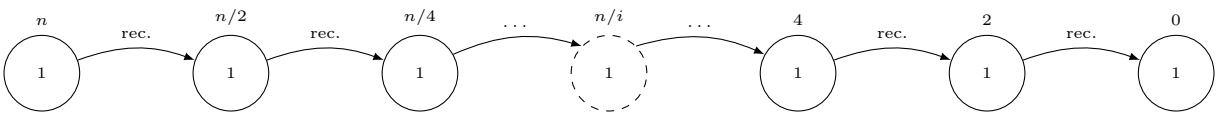
> *Me fijo si el índice medio del arreglo es igual al valor $i = A[i]$. Si lo es, gané. Sino tengo que ir para la derecha o izquierda según la comparación (¡Arreglo ordenado!). Hago así hasta que encuentre un valor verdadero o me quede sin arreglo. Fin.*

```
1  funcion indiceEspejo(A[1..n])
2     si n = 0   // vacío.
3          ret false
4
5     m ← n/2 // agarra el de abajo, no me importa mucho el caso borde. YOLO.
6     si m = A[m]
7          ret true
8     sino
9          si m > A[m]
10              ret indiceEspejo(A[m+1..n]) // busco a la der de m.
11         sino
12              ret indiceEspejo(A[1..m-1]) // busco a la izq de m.
```

La función de costo, pa' calcular el *running Time*:

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 1 \\ T(n/2) + 1 & \text{si} \quad n > 1 \end{cases}$$

Como el costo de resolver cada subproblema es constante, es decir no depende de n, esta función se calcula fácil solo contando los nodos del árbol de recursión. Árbol que en este caso tiene una sola rama, porque por cada recursión se llama como mucha una sola vez a la función. Después de todo lo único que hago es una comparación en cada iteración y nada más.



¿Cuántos niveles, o para este caso particular, nodos tiene? Tiene $\log_2(n)$. Lo cuál se calcula como la cantidad k de veces que tengo que dividir el input hasta dar con el caso base.

$$T(n/2^k) = T(1) \Leftrightarrow n/2^k = 1 \Leftrightarrow n = 2^k \Leftrightarrow k = \log_2(n)$$

Por lo tanto la complejidad es:

$$\boxed{T(n) = \Theta(\log(n))}$$

Con el *Teorema Maestro:*

$$T(n) = \Theta\big(n^{\log_b(a)} \log^{k+1}(n)\big) \xrightarrow[k=0]{b=2 \quad \text{y} \quad a=1} \boxed{T(n) = \Theta\big(\log(n)\big)}$$

Fin.

---

5. *PotenciaLogarítmica*
Encuentre un algoritmo para calcular a^b en tiempo logarítmico en b. Piense cómo reutilizar los resultados ya calculados. Justifique la complejidad del algoritmo dado.

Quiero explotar esto:

$$\begin{cases} \text{este no funca} \rightarrow a^b = \overbrace{a \cdot a \cdots a}^{b \text{ operaciones}} & \rightarrow \Theta(n) \\ \text{esto sí funca} \rightarrow a^b = \begin{cases} (a^2)^{\frac{b}{2}} & \text{si} \quad b \text{ par} \\ (a^2)^{\frac{\lfloor b \rfloor}{2}} \cdot a^1 & \text{si} \quad b \text{ impar} \end{cases} & \rightarrow \Theta(\log_2(n)) \end{cases}$$

Un poco más formalizada:

$$potenciaLogaritmica(a,b) = \begin{cases} a & \text{si} \quad b = 1 \\ potenciaLogaritmica(a^2, b/2) & \text{si} \quad b \text{ es par} \\ a \cdot potenciaLogaritmica(a^2, b/2) & \text{si} \quad b \text{ es impar} \end{cases}$$

Ejemplo ejemplicante:

$$\begin{aligned} potenciaLogaritmica(a, 19) &= a \cdot potenciaLogaritmica(a^2, 9) \\ &= a \cdot a^2 \cdot potenciaLogaritmica((a^2)^2, 4) \\ &= a \cdot a^2 \cdot potenciaLogaritmica(((a^2)^2)^2, 2) \\ &= a \cdot a^2 \cdot potenciaLogaritmica(\underbrace{(((a^2)^2)^2)^2}_{a^{16}}, 1) \leftarrow \text{ caso base} \\ &= a \cdot a^2 \cdot a^{16} \\ &= a^{19} \end{aligned}$$

Este ejemplo genera un árbol de 4 niveles, $4 < \log_2(19) < 5$, o sea que estaría funcionando lo más bien *complejituisticamente hablando*. Con la función de costos se puede encontrar la complejidad o *running time* de forma inmediata:

$$T(b) = \begin{cases} 1 & \text{si} \quad b = 1 \\ T(b/2) + 1 & \text{si} \quad b > 1 \end{cases} \xrightarrow[\textit{running time}]{\text{complejidad}} \boxed{T(b) \in \Theta(\log_2(b))}$$

La función toma un valor a (la base) que siempre es una potencia de a cada vez mayor y un b (el exponente). Devuelve el cuadrado (cuadrado + 1) de la base si el exponente es par (impar), valores para usar en el siguiente llamado. El exponente se reduce a la mitad en cada llamado. El caso base es $b = 1$. Voy a terminar una vez que el exponente sea 1, lo que corresponde a haberlo dividido entre 2 un total de $\log_2(b)$ veces.

```
1  funcion potenciaLogaritmica(a, b)      // a^b
2    si b = 1
3        ret a                            // devuelvo a, ya que a^1 = a.
4
5    si b es par
6        ret potenciaLogaritmica(a · a, b/2)
```

```
7       sino
8           ret a · potenciaLogaritmica(a · a, b/2)
```

Usando el *teorema maestro*:

$$f(b) = \Theta(b^{\log_c(a)} \cdot \log^k(b)) \xrightarrow[k=0]{a=1,\ c=2} \boxed{T = \Theta\big(\log(b)\big)}$$

Fin.

---

6. *MáximoMontaña*

Un arreglo de enteros se denomina montaña si está compuesto por una secuencia estrictamente creciente seguida de una estrictamente decreciente. Dado un arreglo montaña de longitud n, dar un algoritmo que encuentre el máximo del arreglo en complejidad $O(\log(n))$. Por ejemplo, para un arreglo $[-1, 3, 8, 22, 30, 22, 8, 4, 2, 1]$, el máximo está en la posición 4 y vale 30.

El asunto de la complejidad me pone el cerebro en modo *binarySearch*, dado que tengo un arreglo con cierto orden y en cada *iteracion/recursión* voy a estar haciendo comparaciones, para saber para todo seguir, operaciones que tienen una complejidad constante.

Descripción en lenguaje natural del algoritmo:

El input es un arreglo *montaña*. Agarro el elemento del medio del arreglo y me fijo lo que tiene a los costados. Si

```
A[m-1] < A[m] < A[m+1]
```

me quedo con el arreglo $A[m+1..n]$ sino al revés.

Caso base:

Si el arreglo tiene un solo elemento devuelvo ese elemento.

Un poco más formalizado en una función. Pongo 2 sabores. Una en la que paso el arreglo como parámetro, lo que es menos eficiente en la compu que pasar índices, pero es más fácil/intuitivo al principio que laburar solo con los índices:

$$\mathit{maximaMontaña}(A[1..n]) = \begin{cases} A[1] & \text{si} \quad |A| = 1 \\ \max(A[1], A[2]) & \text{si} \quad |A| = 2 \\ \mathit{maximaMontaña}(A[|A|/2 + 1..n]) & \text{si} \quad A[|A|/2 - 1] < A[|A|/2] < A[|A|/2 + 1] \\ \mathit{maximaMontaña}(A[1..|A|/2 - 1]) & \text{si} \quad A[\tfrac{1+n}{2} - 1] > A[|A|/2] > A[|A|/2 + 1] \\ A[|A|/2] & \text{sino} \end{cases}$$

Versión con índices. Más eficiente, pero requieren un poco más de práctica:

$$\mathit{maximaMontaña}(i, d) = \begin{cases} A[i] & \text{si} \quad d = i \\ \max(A[i], A[d]) & \text{si} \quad d - i = 1 \\ \mathit{maximaMontaña}(\frac{i+d}{2} + 1, d) & \text{si} \quad A[\frac{i+d}{2} - 1] < A[\frac{i+d}{2}] < A[\frac{i+d}{2} + 1] \\ \mathit{maximaMontaña}(i, \frac{i+d}{2} - 1) & \text{si} \quad A[\frac{i+d}{2} - 1] > A[\frac{i+d}{2}] > A[\frac{i+d}{2} + 1] \\ A[\frac{i+d}{2}] & \text{sino} \end{cases}$$

```
1 funcion maximaMontana(i, d)              // i: índice izq, d: índice der.
2     si i=d
3         ret A[i]                  // devuelvo A[i], el único elemento
4     si d-i=1                      // tamano de arreglo es 2
5         ret max(A[i],A[d])        // devuelvo máximo
6
7     m ← (i+d)/2
8
9     si (A[m-1] < A[m] < A[m+1])
10        ret maximaMontana(m+1, d)
```

```
si else (A[m-1] > A[m] > A[m+1])
    ret maximaMontana(i, m-1)
sino
    ret A[m]
```

Esta función, algoritmo, genera un árbol de una sola rama, dado que en cada llamada solo puede llamar una vez a la otra función con una longitud $\log_2(n)$, donde n es el tamaño del arreglo. Nuevamente el costo de resolver cada subproblema (el $f(n)$) es constante, son solo un par de comparaciones.

$$\boxed{T(n) \in \Theta(\log(n))}$$

A esta altura el *teorema maestro* no puede chuparme tanto un huevo.

Fin.

---

7. *ComplexityQuest*

Calcule la complejidad de un algoritmo que utiliza $T(n)$ pasos para una entrada de tamaño n, donde T cumple:

1) $T(n) = T(n-2) + 5$
2) $T(n) = T(n-1) + n$
3) $T(n) = T(n-1) + \sqrt{n}$
4) $T(n) = T(n-1) + n^2$
5) $T(n) = 2T(n-1)$
6) $T(n) = T(n/2) + n$
7) $T(n) = T(n/2) + \sqrt{n}$
8) $T(n) = T(n/2) + n^2$
9) $T(n) = 2T(n/2) + n^2$
10) $T(n) = 2T(n-4)$
11) $T(n) = 2T(n/2) + \log(n)$
12) $T(n) = 3T(n/4)$
13) $T(n) = 3T(n/4) + n$

En los ejercicios donde $a = 1$ y $b = 1$ el *Teorema Maestro* no está bien definido. En la demo, se usá como hipótesis que $\frac{a}{b} > 1$.

⚠ Si puedo evitar usar de forma explícita el *Teorema Maestro* lo voy a hacer. Te sugiero intentar hacerlos sin esa fórmula fea también, porque sino no entendés un choto de lo que está pasando con la complejidad. ⚠

Voy a estar usando esta identidad: $\boxed{a^{\log_b(c)} = c^{\log_b(a)}}$. Se muestra fácil aplicando $\log_a()$ en ambos miembros

🖩

$$a^{\log_b(c)} = c^{\log_b(a)} \xleftrightarrow[\text{ambos miembros}]{\text{aplico } \log_a()} \log_b(c) \cdot \overbrace{\log_a(a)}^{1} = \log_b(a) \cdot \log_a(c)$$

$$\xleftrightarrow[\text{de base}]{\text{cambio}} \log_b(c) \cdot 1 = \log_b(a) \cdot \frac{\log_b(c)}{\log_b(a)}$$

$$\overset{!}{\iff} \log_b(c) \overset{\checkmark}{=} \log_b(c)$$

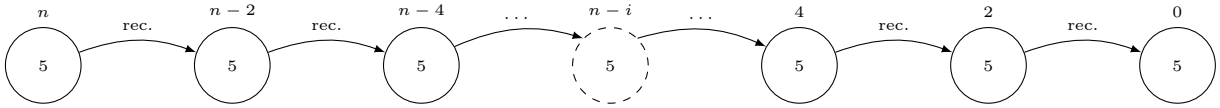
🖩

¿No sabés cambiar la base de un log*? Aprendelo ayer. Hacete un favor y eliminá ruido cognitivo de tu vida, once and for all.*

1) No puedo usar el *Teorema Maestro*. Hay que calcular a mano. El árbol de recursión tiene una sola rama.

$$T(n) = \begin{cases} 5 & \text{si} \quad n = 0 \\ T(n-2) + 5 & \text{si} \quad n > 0 \end{cases}$$

Tengo un costo en cada *subproblema* de 5:



¿Cuál es la altura o longitud de este árbol?
Será la cantidad de veces, k, que tenga que hacer recurrencia hasta llegar al caso base:

$$T(n-2k) \stackrel{?}{=} T(0) \iff n-2k=0 \iff \boxed{k=n/2}$$

El costo usando n_p par o n_i impar, de tal manera que habría $k = n/2$ llamados recursivos (el O se come los casos borde). El *running time* del algoritmo es el costo de resolver cada subproblema, 5 en este caso, por la cantidad de llamados, $n/2$ en este caso:

$$T(n) = \sum_{i=1}^{k} 5 \stackrel{!}{=} 5 \cdot \tfrac{n}{2} \in \Theta(n)$$

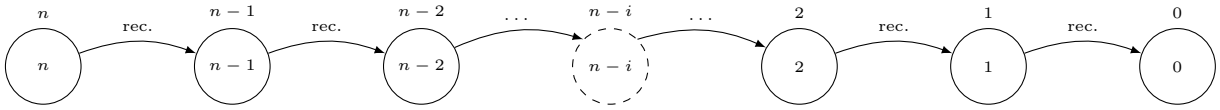
En este caso los llamados recursivos dominan el *running time*:

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

2) Parecido al anterior.

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 0 \\ T(n-1) + n & \text{si} \quad n > 0 \end{cases}$$

Tengo un costo para resolver cada *subproblema* de n:



¿Cuál es la altura o longitud de este árbol?
Será la cantidad de veces, k, que tenga que hacer recurrencia hasta llegar al caso base:

$$T(n-k) \stackrel{?}{=} T(0) \Leftrightarrow n-k=0 \Leftrightarrow \boxed{k=n}$$

Tengo un total de n iteraciones:

$$T(n) = \sum_{i=1}^{n} i = \tfrac{n\cdot(n+1)}{2} \in \Theta(n^2)$$

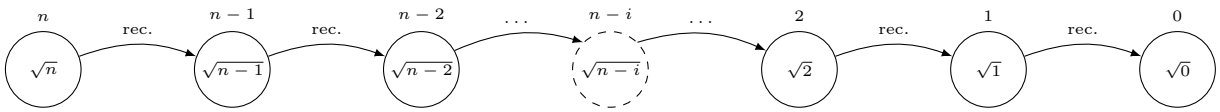
El *running time* T está dominado por la llamada recursiva:

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

3) Parecido al anterior.

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 0 \\ T(n-1) + \sqrt{n} & \text{si} \quad n > 0 \end{cases}$$

Tengo un costo, *running time*, para resolver cada *subproblema* de $\sqrt{n}$:



Tengo un total de n llamados:

$$T(n) = \sum_{i=1}^{n} \sqrt{i} = \sum_{i=1}^{n} i^{\frac{1}{2}} \overset{!!}{\approx} \int_{1}^{n} i^{\frac{1}{2}} di = \frac{2}{3} i^{\frac{3}{2}} \Big|_1^n = \frac{2}{3}(n^{\frac{3}{2}} - 1) \in \Theta(n^{\frac{3}{2}})$$

El llamado recursivo domina el *running time* nuevamente.

$$\boxed{T(n) \in \Theta(n^{\frac{3}{2}})}$$

4) Tengo un total de n iteraciones. Similar a los ejercicios anteriores, sumar todas las contribuciones me da:

$$T(n) = \sum_{i=1}^{n} i^2 \approx \int_{1}^{n} i^2 di = \frac{1}{3} i^3 \Big|_1^n = \frac{1}{3}(n^3 - 1) \in \Theta(n^3)$$

El llamado recursivo domina el *running time* nuevamente.

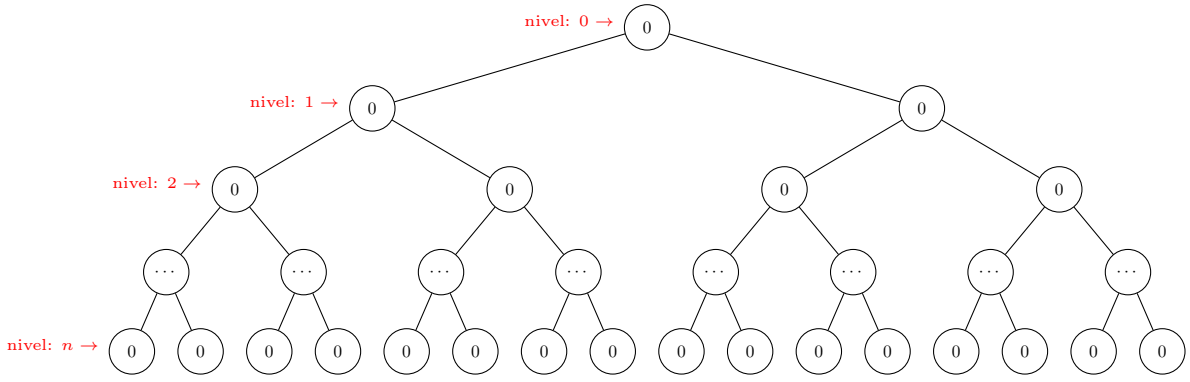
$$\boxed{T(n) \in \Theta(n^3)}$$

5) Por fin empiezan a aparecer *más cantidad de subproblemas* en cada iteración, así le pone un poquito de sal.

$$T(n) = \begin{cases} 0 & \text{si} \quad n = 0 \\ 2T(n-1) & \text{si} \quad n > 0 \end{cases}$$

Si no tengo un costo para resolver subproblemas ($f(n) = 0$) la complejidad depende solo del llamado recursivo. Entonces el *running time, complejidad o como más te guste* es igual a contar la cantidad de nodos del árbol recursivo, la cantidad de llamadas que hace la función.

Estoy partiendo al problema en 2 problemas de (casi) igual tamaño, así que puedo esperar que la complejidad explote con n. Tengo un árbol de n niveles:



Para calcular el costo total, hay que calcular la cantidad de nodos. En el nivel 1, tengo 2^1 nodos, en el nivel i tengo 2^i:

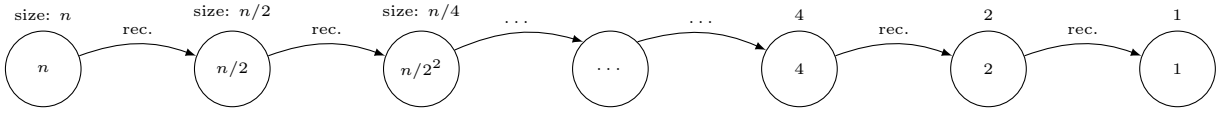
$$T(n) = \sum_{i=0}^{n} 2^i \overset{!}{=} \frac{2^{n+1} - 1}{2 - 1} \in \Theta(2^n)$$

El *running time* está claramente dominado por los llamados recursivos:

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

6) El árbol recursivo tiene una sola rama. El `input` se parte a la mitad en cada iteración, por lo tanto voy a tener un árbol de altura $\log_2(n)$:

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 1 \\ T(n/2) + n & \text{si} \quad n > 1 \end{cases}$$



¿Cuál es la altura o longitud de este árbol?
Será la cantidad de veces, k, que tenga que hacer recurrencia hasta llegar al caso base:

$$T(\tfrac{n}{2^k}) \stackrel{?}{=} T(1) \Leftrightarrow \tfrac{n}{2^k} = 1 \Leftrightarrow \boxed{k = \log_2(n)}$$

Por ejemplo: Si $n = 32$, pasados 5 llamados(en la misma rama) se llega al caso base, dado que $\log_2(32) = 5$

$$\begin{aligned} T(n) = \sum_{i=1}^{\log_2 n} \tfrac{n}{2^i} &\stackrel{!}{=} n \cdot \sum_{i=1}^{\log_2 n} (\tfrac{1}{2})^i \\ &= n \cdot \frac{(\frac{1}{2})^{\log_2(n)+1} - \frac{1}{2}}{-\frac{1}{2}} \\ &= n \cdot (-2) \cdot (\tfrac{1}{2n} - \tfrac{1}{2}) \\ &= n - 1 \\ &\in O(n) \end{aligned}$$

La sumatoria geométrica con razón menor a 1, aporta muy poco, porque *decrece muy rápido asintóticamente*, es por eso que el costo de resolver cada *subproblema* ($f(n) = n$) domina la complejidad.

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

7) Parecido al anterior.

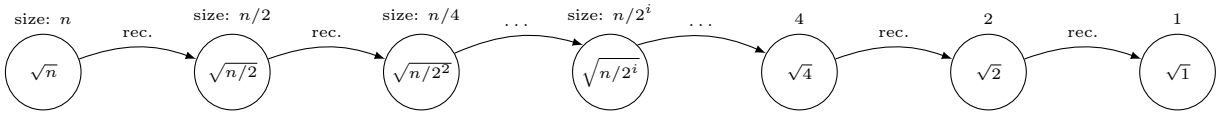
El árbol de recursión va a dar una geométrica de razón menor a 1, por lo tanto la complejidad tiene que estar dominada por el costo de resolver los subproblemas:

$O(\sqrt{n})$ porque el *input* se achica exponencialmente rápido y $\forall k > 0, n^k$ asintóticamente le va a ganar.

...Anyways

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 1 \\ T(n/2) + \sqrt{n} & \text{si} \quad n > 1 \end{cases}$$

El triste árbol de recursión va a tener una sola rama y una altura (¿longitud?) de $\log_2(n)$.



Sumando las contribuciones:

$$\begin{aligned} \sum_{i=1}^{\log_2(n)} \sqrt{\tfrac{n}{2^i}} &\stackrel{!}{=} \sqrt{n} \cdot \sum_{i=1}^{\log_2(n)} (\tfrac{1}{\sqrt{2}})^i \\ &= \sqrt{n} \cdot \frac{(\frac{1}{\sqrt{2}})^{\log_2(n)+1} - \frac{1}{\sqrt{2}}}{\frac{1}{\sqrt{2}} - 1} \\ &= \sqrt{n} \cdot \frac{\frac{1}{\sqrt{n}} \cdot \frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}}{\frac{1-\sqrt{2}}{\sqrt{2}}} \\ &= \frac{1-\sqrt{n}}{1-\sqrt{2}} \\ &\in \Theta(\sqrt{n}) \end{aligned}$$

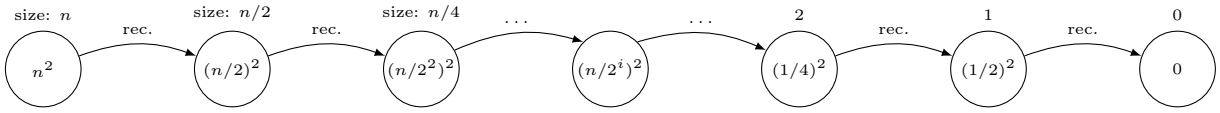
Como era lo esperado, las llamadas recursivas decrecen muy rápido por lo tanto domina el *running time* la complejidad de $f(n)$, el costo de resolver cada subproblema.

$$\boxed{T(n) \in \Theta(\sqrt{n})}$$

8) Idem anterior, esto tiene que dar $\Theta(n^2)$ por la misma razón.

$$T(n) = \begin{cases} 0 & \text{si} \quad n = 0 \\ T(n/2) + n^2 & \text{si} \quad n > 0 \end{cases}$$

El triste árbol de recursión va a tener una sola rama y una altura de $\log_2(n)$.



Sumando las contribuciones:

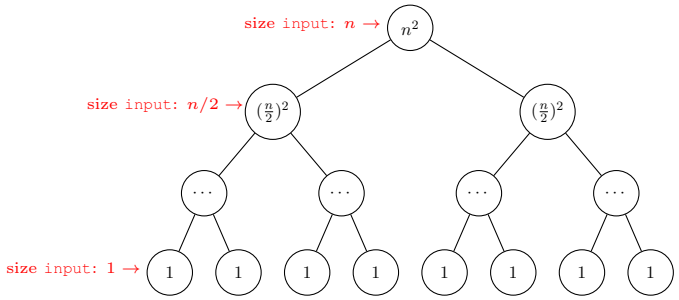
$$\sum_{i=1}^{\log_2(n)} (\tfrac{n}{2^i})^2 \overset{!}{=} n^2 \cdot \sum_{i=1}^{\log_2(n)} (\tfrac{1}{4})^i = n^2 \cdot \frac{(\frac{1}{4})^{\log_2(n)+1} - \frac{1}{4}}{\frac{1}{4}-1} \overset{!}{=} n^2 \cdot \frac{(\frac{1}{n^2}) \cdot \frac{1}{4} - \frac{1}{4}}{\frac{1}{4}-1} \overset{!}{=} \tfrac{1}{2} \cdot (n^2 - 1) \in \Theta(n^2)$$

Era esperado, la complejidad de resolver cada subproblema, domina la complejidad total del algoritmo:

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

9) Acá aparece una función con 2 llamados recursivos, esto cambia un poco el cálculo. El árbol tiene 2 ramas principales ahora que se van bifurcando en cada iteración. El árbol va a tener $\log_2(n)$ niveles.

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 1 \\ 2T(n/2) + n^2 & \text{si} \quad n > 1 \end{cases}$$



Sumando las contribuciones:

Se puede estimar la contribución total. Por ejemplo en el *nivel 0* hay un sólo vértice (la raíz), que tiene una contribución de n^2, en el *nivel* 1 la contribución es:

$$(\tfrac{n}{2})^2 + (\tfrac{n}{2})^2 = 2(\tfrac{n}{2})^2 = \tfrac{1}{2}n^2.$$

La contribución se redujo a la mitad.

En el *nivel* 2 será:

$$(\tfrac{n}{4})^2 + (\tfrac{n}{4})^2 + (\tfrac{n}{4})^2 + (\tfrac{n}{4})^2 = 4(\tfrac{n}{4})^2 = \tfrac{1}{4}n^2.$$

Volvió a disminuir a la mitad

¿De qué tiene pinta esto?
Sí, de una geométrica de razón $\frac{1}{2}$, tuki.

Si esto sigue así ya podemos *vaticinar, profetizar, adivinar, pronosticar* que como el aporte de cada nivel va decrecer exponencialmente la complejidad va a estar dominada por el costo de procesar cada subproblema $f(n) = n^2$.

Arranco la siguiente en sumatoria desde 0, así que la ecuación de la geométrica es $\frac{q^{N+1}-1}{q-1}$.

$$\sum_{i=0}^{\log_2(n)-1} \overset{\text{contribución llamada recursiva}\uparrow}{2^i \cdot (\tfrac{n}{2^i})^2} \overset{!}{=} n^2 \cdot \sum_{i=0}^{\log_2(n)-1} (\tfrac{1}{2})^i = n^2 \cdot \frac{(\frac{1}{2})^{\log_2(n)}-1}{\frac{1}{2}-1} \overset{!}{=} n^2 \cdot \frac{\frac{1}{n^2}-1}{\frac{1}{2}-1} = 2(n^2-1) \in \Theta(n^2)$$

Resultado esperado, la complejidad está dominada por el *running time* de $f(n)$:

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

10) La función se llama dos veces en cada iteración, pero en input no disminuye significativamente. Este va a ser un árbol de 2 ramas pero la longitud de las ramas va a ser lineal y *no logarítmica*. Este ítem es similar al ítem 5) .

¿Qué espero que pase?
La cantidad de nodos del árbol va a crecer en forma exponencial, así como en otros apareció la geométrica, acá también lo hará pero con una razón mayor a 1 *y eso se va ir a la mierda* 💣 .

Es importante notar que no hay costo para resolver cada subproblema, lo que equivale a sumar los nodos que tengo en el árbol de recursión.

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 0 \\ 2T(n-4) & \text{si} \quad n > 0 \end{cases}$$

$$\begin{aligned} T(n) &= 2T(n-4) \\ T(n-4) &= 2T(n-8) \\ T(n-8) &= 2T(n-12) \\ \vdots &\quad \vdots \quad \vdots \\ T(8) &= 2T(4) \\ T(4) &= 2T(0) \end{aligned}$$

El árbol tiene $n/4$ pisos, es lineal.

Sustituyendo los valores de las recursiones. La complejidad queda exponencial:

$$T(n) = 2^2T(n-8) = 2^3T(n-12) = \cdots = 2^{n/4}T(0) \in O((\sqrt[4]{2})^n)$$

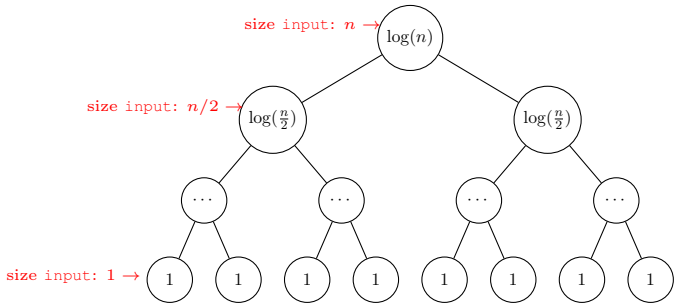
Las llamadas recursivas dominan la complejidad:

$$\boxed{T(n) \in \Theta((\sqrt[4]{2})^n)}$$

11) Este está bueno, parecido al ítem 9) pero al tener un $f(n) = \log(n)$ hace que el costo de cada iteración sea asintóticamente chico. Hay que hacer las cuentas, pero sería esperable que la complejidad esté dominada por la recursión, en este caso: $O(n)$.

El árbol va a tener 2 ramas y $\log_2(n)$ niveles.

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 1 \\ 2T(n/2) + \log(n) & \text{si} \quad n > 1 \end{cases}$$



Cuento a mano la contribución a la complejidad de cada nivel:

$$\begin{cases} \text{nivel } 0 & \rightarrow & \log_2(n) \\ \text{nivel } 1 & \rightarrow & 2 \cdot \log_2(n/2^1) \\ \text{nivel } i & \rightarrow & 2^i \cdot \log_2(n/2^i) \\ & \vdots & \end{cases}$$

En argumento del logaritmo tiene una exponencial decreciente en el denominador, la contribución a la complejidad es nada.

Arranco la siguiente en sumatoria desde 0, así que la ecuación de la geométrica es $\frac{q^{N+1}-1}{q-1}$.

$$\sum_{i=0}^{\log_2(n)-1} 2^i \cdot \log_2(\tfrac{n}{2^i}) = \sum_{i=0}^{\log_2(n)-1} 2^i \cdot (\log_2(n) - i)$$

contribución llamada recursiva ↑

Esa porquería debería dar en $O(n)$, lo calcularía pero, me tengo que ir a la esquina a ver si llueve.

Este es el único ejercicio hasta el momento donde el Teorema maestro te salva de hacer cuentas fuleras que poco aportan:

$$n^{\log_2(2)} = n > \underbrace{\log(n)}_{f(n)}$$

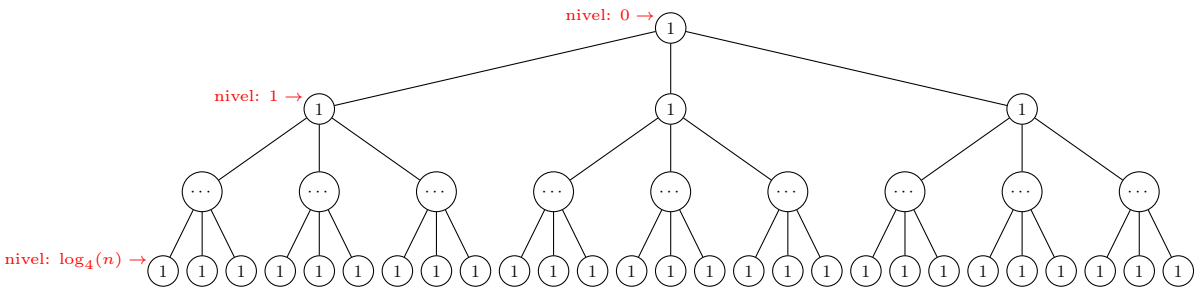
Es el caso 1 del teorema maestro. Como era esperado la complejidad esta dominada por los llamados recursivos queda en:

$$T(n) \in \Theta(n)$$

12) Cada llamada genera 3 ramas nuevas y el tamaño del input se divide en 4 por nivel. Este árbol tiene $\log_4(n)$ niveles. No tengo costo de la resolución de los subproblemas ($f(n) = 0$), por lo cual el problema es contar la cantidad de llamadas recursivas hasta que la función se detenga.

Debería obtener algo como: $\Theta(n^{\log_4(3)})$

$$T = \begin{cases} 1 & \text{si} & n < 4 \\ 3T(n/4) & \text{si} & n \geq 4 \end{cases}$$



Calculo cantidad de nodos dado que no hay costo de resolver un subproblema:

$$T(n) = \sum_{i=0}^{\log_4(n)} 3^i = \frac{3^{\log_4(n)+1}-1}{2} \overset{!}{=} \frac{1}{2} \cdot (3n^{\log_4(3)} - 1) \in \Theta(n^{\log_4(3)})$$

Como era de esperarse el *running time* viene dominado por la recursión.

$$\boxed{T(n) \in \Theta(n^{\log_4(3)})}$$

Este es el caso 1 del *teorema maestro*.

13) Mirando el ítem 12) hay un parecido solo que acá tengo un costo de resolver un subproblema lineal en el input, $f(n) = n$. Dado que antes calculé la complejidad al no tener costo de resolver subproblemas, solo tengo que comparar con el nuevo costo:

$$\overbrace{n^{\log_4(3)}}^{<1} < n$$

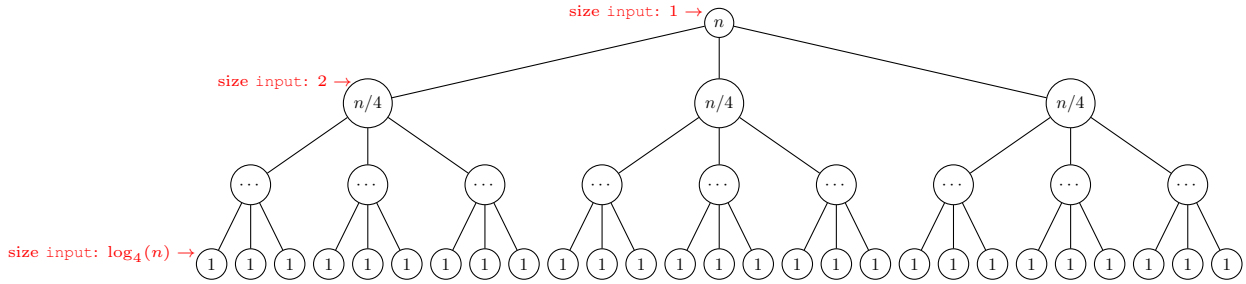
El costo al resolver un subproblema es asintóticamente mayor que el del llamado recursivo de dividir, la complejidad debería ser:

$$T(n) \in \Theta(n)$$

Un poco más desarrollado:

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 1 \\ 3T(n/4) + n & \text{si} \quad n \geq 4 \end{cases}$$

El árbol de recursión tiene $\log_4(n)$ niveles.



Calculo a manopla:

$$T(n) = \sum_{i=0}^{\log_4(n)} 3^i \frac{n}{4^i} \overset{!}{=} n \cdot \sum_{i=0}^{\log_4(n)} (\tfrac{3}{4})^i \overset{!}{=} n \cdot \frac{(\frac{3}{4})^{\log_4(n)+1}-1}{\frac{3}{4}-1} \overset{!!}{=} n \cdot (-4) \cdot \left(\frac{\frac{3}{4}n^{\log_4(3)}}{n} - 1\right) = 4 \cdot (n - 3\overbrace{n^{\log_4(3)}}^{<1}) \in \Theta(n)$$

Entonces:

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

Si bien me cagué en la diferencia entre Θ y O en todo el ejercicio, justifico el usar Θ acá calculando:

$$\lim_{n\to\infty} \frac{4 \cdot \left(n - n^{\log_4(3)}\right)}{n} = 4$$

Y como ese límite no da ni infinito ni 0, el comportamiento de n y la expresión del numerador son asintóticamente iguales. tuki.

Fin.

8. *MaximaSubsecuencia*
Dada una secuencia de n enteros, se desea encontrar el máximo valor que se puede obtener sumando elementos contiguos. Diseñar un algoritmo basado en la técnica de dividir y conquistar que resuelva el problema en $O(n \log n)$. Por ejemplo, para la secuencia $[3, -1, 4, 8, -2, 2, -7, 5]$, este valor es 14, que se obtiene de la subsecuencia $[3, -1, 4, 8]$.

Ejercicio que tiene una parte medio difícil de encontrar. Lo primero que pienso cuando veo la complejidad $O(n \cdot \log(n))$ es *mergeSort*, así que será cuestión de dividir y después hacer con los subproblemas de complejidad $O(n)$. Al igual que en *mergeSort* lo *complicado de describir* no es la recursión sino el *merge*.
Por ejemplo, el arreglo `A=[-1,2,4,-1]` tiene la subsecuencia de máxima suma con `[2,4]`, la cual cruza de una mitad a la otra de `A`.

¿Cómo será la cosa en este ejercicio?

El problema de encontrar la subsecuencia máxima del medio es lo más falopa, pero no es lo más importante del ejercicio. Cuándo pasa esto, voy a suponer que mágicamente tengo la función *findSSMaxMed* y hago todo el resto! Después retomo el asunto, de otra manera pierdo el foco, me frustro y estoy 8 horas sin tocar el tema de recursión, D&C y todo lo que importa.

El algoritmo va a *atomizar el arreglo* devolviendo una secuencia con los índices y suma de la subsecuencia con esos índices.
De todas me voy a quedar con la máxima que se encuentre.
Versión pasando arreglo (más fácil de escribir):

```
1  funcion maxSS(A[i..d])
2    if |A| = 1
3        ret [A[i]]          // devuelvo, [desde, hasta, suma = A[i]], el único elemento
4
5    m ← |A|/2
6
7    I ← maxSS(A[i,m])
8    D ← maxSS(A[m+1, d])
9    M ← findSSMaxMed(I,D)  // Por lo menos un elemento de I y un elemento de D
10
11   // devuelvo la secuencia de suma (3era coordenada) máxima de entre las 3 secuencias.
12   ret maximaSuma(I, D, M)
```

Versión con índices (más oscura, pero más eficiente y se pasa fácil a la compu):

```
1  funcion maxSS(i, d) -> [desde, hasta, ∑_{j=desde}^{hasta} A[j]]   // Primer llamado con maxSS(1, n)
2    if i = d
3        ret [i, d, A[i]]          // devuelvo, [desde, hasta, suma = A[i]], el único elemento
4
5    m ← (i+d)/2
6
7    I ← maxSS(i,m)
8    D ← maxSS(m+1, d)
9    M ← findSSMaxMed(i, m, d)  // Por lo menos un elemento de I y un elemento de D
10
11   // devuelvo la secuencia de suma (3era coordenada) máxima de entre las 3 secuencias.
12   ret maximaSuma(I, D, M)
```

Este ejercicio al igual que *mergeSort* tiene al principio (y al medio) ese ... *je ne sais quoi*, que te hace pensar:

¿Qué clase de ~~mierda~~ brujería es esta?.

Lo que me sirve (a mí) para salir de ese estado de locura mental es *no hacer el camino de la recurrencia* con un ejemplo y tratar de entenderlo así.

Sí, es anti intuitivo lo que sugiero, pero para problemas *con complejidad como este* eso es muchas veces un *rabbit hole*. Me ayuda más pensar al problema como cuando hago un ejercicio de inducción, no tanto seguir el calculo mental sino creer el paso inductivo y dale no más.

Veamos al ejercicio ~~verga~~ este con aire de inducción:

Acá la *hipótesis* sería algo así como:

Si tengo un arreglo, su suma máxima DEBE estar en la mitad izquierda, en la mitad derecha o en una combinación de ambas (algo de la mitad izquierda y de la mitad derecha).
El caso en que el arreglo tiene longitud 1, fácil hay una única respuesta, ese es nuestro caso base y claramente funciona!
Listo después el llamada recursivo es nuestra hipótesis inductiva, onda vale para $|A| = k$ entonces vale $|A| = k + 1$, y ni lo pruebo, sencillamente tengo fe en la inducción/recurrencia.

A todo esto ni sé que hace: *findSSMaxMed*

La función consigue la secuencia de suma máxima contando desde la mitad para atrás y la secuencia de suma máxima de la mitad para adelante. Luego me devuelve una secuencia [desde, hasta, suma]

La voy a escribir para la versión con índices. La versión en la que se usa el arreglo explícitamente es más sencilla.

```
funcion findSSMaxMed(i, m, d)
    desde ← 0 // Indice inferior de la subsecuencia
    hasta ← 0 // Indice inferior de la subsecuencia

    // Inicializo para buscar la secuencia en la parte izquierda de atrás para adelante
    sumaTmp ← 0
    maxI    ← 0
    j ← m
    mientras (i <= j)
        tmp  ← tmp + A[j]
        si tmp > maxI
            maxI ← tmp
            desde ← j
        j ← j - 1

    // Inicializo para buscar la secuencia en la parte derecha de adelante para atrás
    sumaTmp ← 0 //reset
    maxD    ← 0
    j       ← m + 1
    mientras (j <= d)
        sumaTmp  ← sumaTmp + A[j]
        si sumaTmp > maxD
            maxD ← sumaTmp
            hasta ← j
        j ← j + 1

    ret [desde,hasta,maxI+maxD] // Devuelvo la lista [desde, hasta, suma]
```

Este código es súper duro. Algo más sencillo y aceptable como algoritmo podría ser:

```
funcion findSSMaxMed(I, D)
  → Consigo la subsecuencia máxima de I, pero contando para atrás.
     Me guardo el índice hasta que llegué → i

```

```
→ Consigo la subsecuencia máxima de D.
   Me guardo el índice hasta que llegué → d

→ devuelvo A[i,d] // Arreglo secuencia máxima con parte de I y de D.
```

Del pseudocódigo sé que la función *findSSMaxMed* es $\Theta(n)$. Por lo tanto la función recursiva tiene una forma:

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 1 \\ 2T(n/2) + n & \text{si} \quad n > 1 \end{cases}$$

Árbol de 2 ramas, dado que hay dos llamados recursivos en cada ciclo. Cómo el input se reduce a la mitad en cada iteración el árbol tendrá $\log_2(n)$ niveles.
Finalmente el *running time*:

$$\boxed{T(n) \in \Theta(n \log(n))}$$

Fin.

---

9. *PotenciaSum*
Suponga que se tiene un método *potencia* que, dada una matriz cuadrada A de orden 4×4 y un número n, computa la matriz A^n. Dada una matriz cuadrada A de orden 4×4 y un número natural n que es potencia de 2 (i.e, $n = 2^k$ para algún $k \geq 1$), desarrollar, utilizando la técnica de dividir y conquistar y el método *potencia*, un algoritmo que permita calcular

$$A^1 + A^2 + A^3 + \cdots + A^n.$$

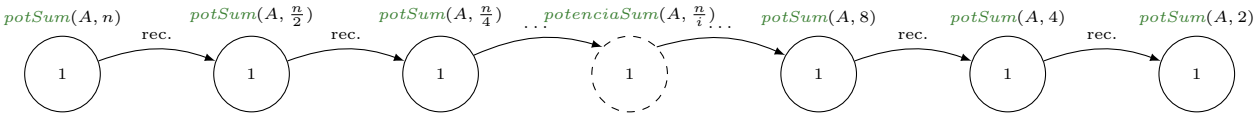
Procure que el algoritmo propuesto aplique el método *potencia*, sume y haga productos de matrices una cantidad estrictamente menor que $O(n)$ veces.

¿Qué ondis? Para tener en mente: Si tengo que implementar *divide & conquer* y además tener un *running time* menor a $O(n)$, seguramente me están pidiendo que la complejidad sea $O(\log(n))$, lo cual implicaría un árbol de una rama $\log_2(n)$ niveles, dividiendo al input a la mitad en cada iteración.
Intento ver si puedo factorizar para calcular esto en forma recursiva:
Dado que $n \in \{2, 4, 8, \ldots, 2^k\}$, el caso base sería con $n = 2$:

$$\begin{array}{lll} n = 2 & \Longrightarrow & A + A^2 = \overbrace{A \cdot (I + A)}^{\text{caso base}} \\ n = 4 & \Longrightarrow & A + A^2 + A^3 + A^4 = A \cdot (I + A + A^2 + A^3) = A \cdot (I + A \cdot (I + A + A^2)) = A \cdot (I + A \cdot (I + \underbrace{A \cdot (I + A)}_{\text{caso base}})) \end{array}$$

Ahí hay un patrón recursivo de esta pinta:

$$potSum(A, n) = \begin{cases} A \cdot (I + A) & \text{si} \quad n = 2 \\ A \cdot (I + potSum(A, \frac{n}{2})) & \text{si} \quad n > 2 \end{cases}$$



Árbol de una sola rama, dado que la función tiene una sola llamada recursiva en cada llamada ¿Cuántos niveles tiene el árbol? ¡Busco la cantidad de iteraciones k hasta llegar al caso base $n = 2$!:

$$T(A, \tfrac{n}{2^k}) \overset{!}{=} T(A, 2) \Leftrightarrow \tfrac{n}{2^k} = 2 \Leftrightarrow n = 2^{k+1} \Leftrightarrow \log_2(n) = k + 1 \Leftrightarrow k = \log_2(n) - 1$$

Los subproblemas tienen un costo constante porque las matrices tienen un tamaño constante de 4×4, por eso le puse un 1 a los nodos del árbol. Por lo tanto estoy sumando 2 matrices y multiplicando 2 matrices por cada subproblema, lo cual es algo constante así que ~~me chupa un huevo~~ no aporta nada. La complejidad o *running time* de este algoritmo *potSum*:

$$\boxed{T(n) \in \Theta(\log(n))}$$

Un pseudocódigo para pasar a la compu:

```
1 funcion potSum(A, n)
2       si n = 2
3           ret A·(I+A)    // O(1)
4       m <- n/2
5       B <- potSum(A,m)
6       ret A·(I+B)        //O(1)
```

Todo parece fácil ahora, pero antes de darme cuenta que la cosa iba por el lado de la recursión y no por el lado de resolver el problema encontrando un súper hack, estuve un día entero falopeando esto:
Haciendo muchas factorizaciones en grupos llegué a esto:

$$A^1 + A^2 + A^3 + \cdots + A^{2^k} = (A + A^2) \cdot \prod_{i=1}^{k-1} (I_n + A^{2^i})$$

Pruebo por inducción que esto es correcto. Proposición:

$$p(n) : \sum_{i=1}^{2^n} A^i = (A + A^2) \cdot \prod_{j=1}^{n-1} (I + A^{2^j}) \quad \forall n \in \mathbb{N}_{\geq 2}$$

Caso base:
Quiero probar que la proposición $p(2)$:

$$p(2) : \sum_{i=1}^{2^2} A^i = (A + A^2) \cdot \prod_{j=1}^{2-1} (I + A^{2^j})$$

es verdadera. Sale inmediato:

$$\sum_{i=1}^{2^2} A^i = A + A^2 + A^3 + A^4 \overset{!!}{=} (A + A^2) + (A + A^2)A^2 = (A + A^2)(I + A^2)$$

y

$$(A + A^2) \cdot \prod_{j=1}^{2-1} (I + A^{2^j}) = (A + A^2)(I + A^2)$$

Por lo tanto la proposición $p(2)$ resultó verdadera

Paso inductivo:
Asumo que para algún valor $h \in \mathbb{N}$ la proposición:

$$p(h) : \underbrace{\sum_{i=1}^{2^h} A^i = (A + A^2) \cdot \prod_{j=1}^{h-1} (I + A^{2^j})}_{\text{hipótesis inductiva}}$$

es verdadera. Entonces quiero probar que la proposición:

$$p(h+1) : \sum_{i=1}^{2^{h+1}} A^i = (A + A^2) \cdot \prod_{j=1}^{h} (I + A^{2^j})$$

también lo sea. Partiendo de $p(h+1)$:

$$\begin{aligned}
\sum_{i=1}^{2^{h+1}} A^i &= (\sum_{i=1}^{2^h} A^i) + \sum_{i=2^h+1}^{2^{h+1}} A^i \\
&\overset{!!}{=} (\sum_{i=1}^{2^h} A^i) + (A + A^2 + \cdots A^{2^h}) \cdot A^{2^h} \\
&\overset{!!}{=} (\sum_{i=1}^{2^h} A^i) + (\sum_{i=1}^{2^h} A^i) \cdot A^{2^h} \\
&= (\sum_{i=1}^{2^h} A^i)(I + A^{2^h}) \\
&\overset{HI}{=} ((A + A^2) \cdot \prod_{j=1}^{h-1} (I + A^{2^j}))(I + A^{2^h}) \\
&= (A + A^2) \cdot \prod_{j=1}^{h} (I + A^{2^j})
\end{aligned}$$

Demostrando así que $p(h+1)$ es verdadera.

Dado que $p(2), p(h)$ y $p(h+1)$ resultaron verdaderas, por principio de inducción también lo es $p(n)$ $\forall n \in \mathbb{N}_{\geq 2}$

Fin.

---

10. *DistanciaMáxima*
Dado un árbol binario cualquiera, diseñar un algoritmo de dividir y conquistar que devuelva la máxima distancia entre dos nodos (es decir, máxima cantidad de ejes a atravesar). El algoritmo no debe hacer recorridos innecesarios sobre el árbol. **Hint**: para saber el camino más largo de un árbol, posiblemente necesite conocer más que sólo los caminos más largos de sus subárboles.

🌲 En un árbol binario, cada *nodo o vértice* tiene dos, uno o ningún hijo.

- Es razonable pensar que *la máxima distancia* entre 2 nodos, sea entre dos *hojas* del árbol, es decir dos nodos sin hijos.

- Como dice el **HINT** del enunciado, tengo que conoce más que las ramas más largas, dado que la distancia máxima entre nodos, podría estar en la izquierda, derecha o cruzando de un lado al otro del árbol.

Me salió esto:

```
funcion distanciaMaxima(nodo)
    si nodo es hoja
        ret 0

    profIzq  ← profundidad(hijoIzq)
    profDer  ← profundidad(hijoDer)
    cruzando ← profIzq + profDer + 2

    potencialMax ← max(profIzq, profDer, cruzando)

    ret max(potencialMax, distanciaMaxima(hijoIzq), distanciaMaxima(hijoDer))

funcion profundidad(nodo)
    si nodo es hoja
        ret 0
    pI ← 1 + profundidad(hijoIzq)
    pD ← 1 + profundidad(hijoDer)
    ret max(pI, pD)
```

Pero no cumple con lo de *no hacer recorridos innecesarios* porque tengo una complejidad de $O(n^2)$, dado que estoy calculando la profundidad de las ramas de cada nodo en cada llamada recursiva.
Así que tengo que solucionar eso para no estar pasando tantas veces por los mismos nodos al pedo.

Fin.

---

11. *DesordenSort*
La cantidad de parejas en desorden de un arreglo $A[1 \dots n]$ es la cantidad de parejas de posiciones $1 \leq i < j \leq n$ tales que $A[i] > A[j]$. Dar un algoritmo que calcule la cantidad de parejas en desorden de un arreglo y cuya complejidad temporal sea estrictamente mejor que $O(n^2)$ en el peor caso. **Hint**: Considerar hacer una modificación de un algoritmo de sorting.

Tiene pinta de *mergeSort*. Solo que debo contar cuando hago los ordenamientos.

</>$_{1)}$ Hago el *divide* como siempre.

</>$_{2)}$ El caso base agarra el arreglo de tamaño 1, donde no hay mucho que hacer.

</>$_{3)}$ Al igual que en *mergeSort* toda la papa está en la función *contar*, que hace lo mismo que la función *merge*, pero me va a ircontando cada vez que haya elementos para ordenar.

</>$_{4)}$ Si el elemento en la parte derecha `A[j]` es menor que el `A[i]`, también lo será $\forall i \in [i, m]$, donde m es el índice que salió del *divide*. Dado que los arreglos de la izquierda y derecha están ordenados al igual que en *mergeSort*.

</>$_{5)}$ Tengo algo así $f(n) = 2f(n/2) + \text{las parejas que ordené}$, lo cual tiene pinta de cumplir la restricción de complejidad pedida.

```
1 funcion desordenSort(A[1..n])
2     si n = 1
3         ret 0 // nada que hacer
4
5     m ← n/2
6
7     ret desordenSort(A[1..m]) +
8         desordenSort(A[m+1..n]) +
9         contar(A[1..n], m)
```

```
1  funcion contar(A[1..n], m)
2      suma ← 0
3      i ← 1
4      j ← m+1
5      k ← 1 // Para armar la solución B
6      mientras k <= n
7          si j > n // Se terminó la mitad de la derecha?
8              B[k] ← A[i]
9              i ← i + 1
10         sino si i > m // Se terminó la mitad de la izq?
11             B[k] ← A[j]
12             j ← j + 1
13         sino si A[i] < A[j] // Pareja ordenada
14             B[k] ← A[i]
15             i ← i + 1
16         sino // Pareja desordenada
17             B[k] ← A[j]
18             suma ← suma + (m - i) + 1
19             j ← j + 1
20         k ← k + 1
21     k ← 1
22     mientras k <= n
23         A[k] ← B[k] // copio para actualizar A
24
25     ret suma
```

La complejidad de esta función es idéntica a la de *mergeSort*. Hay 2 llamados recursivos de *desordenSort* y en cada llamado hay una llamada a *contar*, esta última recorre el arreglo una vez para ordenar y luego otra vez en el copiado de la respuesta así que *contar(A[1..n], m)* $\in O(n)$. Tengo entonces un árbol de $\log_2(n)$ niveles y cada nivel(sumando todos sus nodos) tiene en total un costo de n. El *running time* del algoritmo queda:

$$T(n) = 2 \cdot T(n/2) + n \implies \boxed{T(n) \in \Theta(n \log(n))}$$

Fin.

12. *CazadorDeFalsos*

Se tiene una matriz booleana A de $n \times n$ y una operación *conjuncionSubmatriz* que toma $O(1)$ tiempo y que dados 4 enteros i_0, i_1, j_0, j_1 devuelve la conjunción de todos los elementos en la submatriz que toma las filas i_0 hasta i_1 y las columnas j_0 hasta j_1. Formalmente:

$$\textit{conjunciónSubmatriz}(i_0, i_1, j_0, j_1) = \bigwedge_{\substack{i_0 \leq i \leq i_1 \\ j_0 \leq j \leq j_1}} A[i, j]$$

1. Dar un algoritmo de complejidad temporal estrictamente menor que $O(n^2)$ que calcule la posición de algún *false*, asumiendo que hay al menos uno. Calcular y justificar la complejidad del algoritmo.

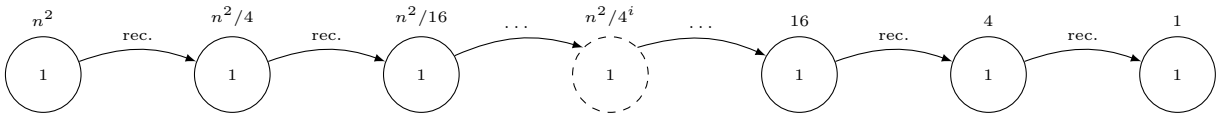
2. Modificar el algoritmo anterior para que cuente cuántos *false* hay en la matriz. Asumiendo que hay a lo sumo 5 elementos *false* en toda la matriz, calcular y justificar la complejidad del algoritmo. Esto se puede lograr con complejidad menor a $O(n^2)$.

1.
- Tengo una función mágica: *conjunciónSubmatriz*. Agarra todos los elementos de la submatriz y si eso da *false*, busco ahí adentro nuevamente para encontrar cuál elemento es el responsable de dar *false*.
- Voy a estar dividiendo el input en 4 submatrices.
- Proceso solo la que me de *false*, por lo tanto resuelvo solo 1 subproblema por llamado recursivo.
- Se parece a un *binarySearch*.

```
funcion cazadorDeFalsos(f1, f2, c1, c2)
    si f1 = f2 ∧ c1 = c2
        si A[f1,c1] = 0
            ret f1, c1
        sino
            ret "Esto no debería retornarse nunca. Hay al menos un 0 en A"

    fm ← (f1+f2)/2
    cm ← (c1+c2)/2

    si (not conjuncionSubmatriz(f1, fm, c1, cm))
        ret cazadorDeFalsos(f1, fm, c1, cm)
    si (not conjuncionSubmatriz(fm+1, f2, c1, cm))
        ret cazadorDeFalsos(fm+1, f2, c1, cm)
    si (not conjuncionSubmatriz(f1, fm, cm+1, c2))
        ret cazadorDeFalsos(f1, fm, cm+1, c2)
    si (not conjuncionSubmatriz(fm+1, f2, cm+1, c2))
        ret cazadorDeFalsos(fm+1, f2, cm+1, c2)

    ret "esto tampoco"
```

Para calcular el *running time* tengo:

$$T(n^2) = T(n^2/4) + 1$$



Como tengo una sola rama, la complejidad será contar la cantidad de iteraciones/nodos hasta llegar al caso base, dado que cada subproblema tiene un costo de 1. La cantidad de niveles k la calculo como:

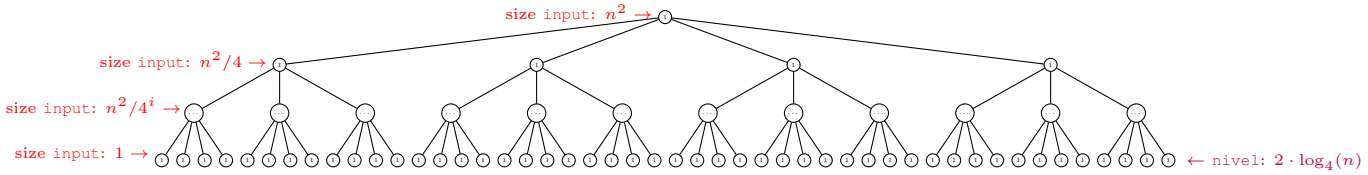
$$T((n^2/4^k)) = T(1) \Leftrightarrow (n^2/4^k) = 1 \Leftrightarrow k = 2 \cdot \log_4(n) \xrightarrow{\text{el } \textit{running time}} \boxed{T(n) \in \Theta(\log(n))}$$

2.

```
funcion cazadorDeFalsos(f1, f2, c1, c2)
    si f1 = f2 ∧ c1 = c2
        si A[f1,c1] = 0
            ret 1
        sino
            ret 0

    fm ← (f1+f2)/2
    cm ← (c1+c2)/2

    suma ← 0

    si (not conjuncionSubmatriz(f1, fm, c1, cm))
        suma = suma + cazadorDeFalsos(f1, fm, c1, cm)
    si (not conjuncionSubmatriz(fm+1, f2, c1, cm))
        suma = suma + cazadorDeFalsos(fm+1, f2, c1, cm)
    si (not conjuncionSubmatriz(f1, fm, cm+1, c2))
        suma = suma + cazadorDeFalsos(f1, fm, cm+1, c2)
    si (not conjuncionSubmatriz(fm+1, f2, cm+1, c2))
        suma = suma + cazadorDeFalsos(fm+1, f2, cm+1, c2)

    ret suma
```

Para calcular el *running time* tengo:

$$T(n^2) = \begin{cases} 1 & \text{si} \quad n = 1 \\ 4T(n^2/4) + 1 & \text{si} \quad n > 1 \end{cases}$$



El árbol de recursión tiene 4 ramas en cada llamado. Dado que la función mágica *conjunciónSubmatriz* es O(1) contar la cantidad de nodos equivale al costo total:

$$\sum_{i=0}^{2\log_4(n)} 4^i = \frac{4^{2\log_4(n)+1} - 1}{3} = \frac{4}{3}n^2 - \frac{1}{3} \in \Theta(n^2).$$

¿Esto me tendría que haber dado menos? Ignoré el dato de que hay *a lo sumo 5 falsos*. El resultado obtenido es equivalente a haber recorrido *absolutamente todos los elementos chequeando* si eran *true* o *false*. Lo cual tendría sentido si tuviese una cantidad arbitraría de 0's.

Pero dado que como mucho tengo 5, el ejercicio es equivalente al anterior, hay una cantidad constante de *matricitas* que se van a detectar en las condiciones con *conjuncionSubmatriz*. En el peor caso tendría:

- Un 0 en cada cuadrante, donde tendría 4 llamadas recursivas, 4/4 (toda la matriz).
- En la segunda llamada ya búsqueda se reduce a solo 5/16 partes de la matriz.
- Y en las sucesivas iteraciones la reducción: $5/4^3 \to 5/4^4 \to \cdots \to 5/4^k$ partes de la matriz, es exponencial.

El denominador de esas fracciones crece exponencialmente hasta llegar a n^2, es decir tiene $\approx 5\log_4(n)$ llamadas recursivas. Un árbol que tiene solo 5 ramas y $2\log(n)$ niveles. Algo así

$$\sum_{i=0}^{5\cdot\log_4(n)} 1 = 5\log_4(n) \implies \boxed{T(n) \in \Theta(\log(n))}$$

Fin.

---

13. *MergeSelectivo*

Dados dos arreglos de naturales, ambos ordenados de manera creciente, se desea buscar, dada una posición i, el $i-$ésimo elemento de la unión de ambos. Dicho de otra forma, el $i-$ésimo del resultado de hacer merge ordenado entre ambos arreglos. Notar que no es necesario hacer el merge completo. Se puede asumir que cada natural aparece a lo sumo en uno de los arreglos, y a lo sumo una vez.

1) Implementar al función *iésimoMerge* que dados los arreglos A y B, y un valor i natural, resuelva el problmea planteado.

2) Calcular y justificar la complejidad del algoritmo propuesto. La complejidad temporal debe ser $O(\log^2 n)$, dónde $n = |A| = |B|$. **Hint:** Observar que, dado el valor de un elemento de alguno de los dos arreglos, se puede averiguar en tiempo $O(\log(n))$ entre qué par de posiciones consecutivas del otro arreglo quedaría, y de allí deducir cuál sería la posición en el merge.

3) **Desafío adicional:** Intente resolver el mismo problema en tiempo $O(\log n)$ (este ítem es bastante más difícil).

... hay que hacerlo!

Si querés mandá la solución → al grupo de Telegram, o mejor aún si querés subirlo en LaTeX→ una *pull request* al.