Estructuras II

Autor: Agustín Fernández Bergé

Estructuras II

Modelo de costo

Modelo de costo
Para analizar el costo de un algoritmo(ya sea de tiempo, memoria o energía) se utilizan dos métricas:

El trabajo(simbolizado como W): Es el costo del algoritmo cuando se ejecuta en una maquina con 1 procesador.

La profundidad o span(simbolizado como S): Es el costo del algoritmo cuando se ejecuta en una maquina con infinitos procesadores.

La idea el trabajo es contabilizar cuantas operaciones se realizan en total.
El span nos da una cota superior a la cantidad de operaciones que puede realizar un procesador si paralelizamos(si con infinitos procesadores no podemos superar cierto costo, no podemos aspirar a mucho con una cantidad finita de los mismos). También nos da una idea de las dependencias entre computaciones.

Ejemplo suma de números:(grafiquito)
La cantidad de operaciones es igual a la cantidad de sumas, podemos ver que es similar a cn
Suponiendo que cada procesador se encarga de una rama del árbol, se tiene que el span es similar a c*lg n.

En el analisis de algoritmos, es util el concepto de analisis asintotico.
Nos da una idea del crecimiento de una funcion.

Cota O:
$f \in O(g)$ si $\exists n_0 \in \mathbb{N}, c > 0 : 0 \leq f(n) \leq cg(n)$
Nos da una cota superior del crecimiento de una funcion

Cota Ω:
$f \in \Omega(g)$ si $\exists n_0 \in \mathbb{N}, c > 0 : 0 \leq cg(n) \leq f(n)$
Nos da una cota inferior del crecimiento de una funcion

Cota Θ:
$f \in \Theta(g)$ si $f \in O(g) \land f \in \Omega(g)$
Nos da una cota ajustada del crecimiento de una función

Otra métrica útil es el paralelismo:
$P = \frac{W}{S}$
Indica cuantos procesadores se pueden usar eficientemente
A la hora de elegir un algoritmo para realizar una tarea, elegimos el algoritmo con mejor trabajo.
Entre estos, elegimos el que tenga mejor paralelismo.

El tiempo que tarda una maquina de p procesadores puede aproximarse como
$T < \frac{W}{p} + S$
Donde p es la cantidad de procesadores

Esta cota asume varias cosas:

- Baja o nula latencia
- Gran ancho de banda
- Scheduler voraz: Los procesadores ejecutan las tareas inmediatamente
- Poco costo de comunicación

Para analizar el costo 2 modelos:
Modelo basado en maquinas:(Consumo de ram, vram, etc)
Modelo basado en lenguajes.

La idea del segundo es definir el costo basado en una operacion del lenguaje, independientemente de la maquina en la que se ejecute.
Dar ejemplos

Dar ejemplo de un algoritmo divide and conquer

RBT

> **RBT**

Recurrencias

> **Recurrencias**
> Cuando se quiere especificar el modelo de costo de un algoritmo recursivo, surgen recurrencias que hay que resolver.
>
> Ejemplo: el calculo del factorial
> Dada la función:
>
> ```
> factorial 0 = 1
> factorial n = n*(factorial n-1)
> ```
>
> Si representamos como $T(n)$ el costo de calcular `factorial n` entonces
> $T(n) = T(n-1) + c$
> Donde c es el costo de realizar la multiplicación.
>
> Como resolver las recurrencias?, hay varias técnicas

Método de sustitución

La idea es dar una cota asintótica usando inducción

Ejemplo:
Sea f una funcion candidato que creemos que cumple $T(n) \in O(n)$
Por definición de O:
$\exists n_0 \in \mathbb{N}, c > 0, \forall n \geq n_0 : 0 \leq T(n) \leq cf(n)$
Luego se prueba por inducción la propiedad:
$\exists c > 0 : 0 \leq T(n) \leq cf(n)$
CB) Probar que vale la propiedad para uno o varios casos bases, esto nos da el n_0
PI) Suponiendo que vale para todo natural menor que n, probar que la propiedad vale para n
En general se utiliza la inducción fuerte para esta prueba, ya que facilita la demostración.

El método de sustitución también puede usarse para demostrar las cotas Ω y Θ

Árbol de recurrencia

Es otro modo de estimar el costo de una recurrencia:
Por ejemplo, dada la siguiente recurrencia
$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$

Se empieza por un árbol cuya raiz tiene el costo inicial, en este caso n
Luego la raiz se divide en ramas indicando las llamadas recursivas, estas ramas se van subdiviendo en mas llamadas recursivas. Cada nodo del arbol contiene el costo de la llamada correspondiente.

Eventualmente se llega al caso base, luego el costo total es equivalente a sumar los costos de cada nivel.

Para la recurrencia anterior:
Cada nivel suma n,
Si el árbol tiene k niveles y la recurrencia finaliza cuando se llega a $n = 1$ entonces se termina cuando $\frac{n}{2^k} = 1$ y por lo tanto $k = lgn$
Luego la suma de todos los niveles es $n \cdot \lg n$.

Si somos prolijos, esto nos da una cota ajustada.
En el peor caso, podemos obtener una aproximación que luego podemos demostrar por el método de sustitución.

Teorema maestro

Muchos algoritmos divide and conquer tienen la misma estructura

- Se divide un problema grande en problemas a mas chicos de tamaño $\frac{n}{b}$(supongase que tiene un costo $f_1(n)$)
- Se solucionan cada uno independientemente.
- Se combinan las soluciones para resolver el problema grande(suponganse que tiene un costo m$f_2(n)$)

Si $T(n)$ es la recurrencia de un problema divide and conquer entonces $T(n)$ tiene la forma

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Donde $f(n) = f_1(n) + f_2(n)$, $a \geq 1$ y $b > 1$
La cota asintotica de estas recurrencias se pueden resolver facilmente usando el "Teorema maestro".

Sea $T(n)$ una recurrencia como la anterior:
Si $f(n) \in O(n^{\log_b a - \epsilon})$ con $\epsilon > 0$, entonces $T(n) \in \Theta(n^{\log_b a})$
Si $f(n) \in \Theta(n^{\log_b a})$, entonces $T(n) \in \Theta(n^{\log_b a} \lg n)$
Si $f(n) \in \Omega(n^{\log_b a + \epsilon})$ con $\epsilon > 0$ y $\exists n_0 \in \mathbb{N}, c < 1, \forall n \geq n_0, af\left(\frac{n}{b}\right) \leq cf(n)$, entonces $T(n) \in \Theta(f)$

A grandes rasgos, el teorema maestro nos hace comparar $n^{\log_b a}$ con $f(n)$

- Caso 1: el primero es mas grande
- Caso 2: Los 2 tienen el mismo orden
- Caso 3: f es mas grande
  Se pide ademas que una tiene que ser polinomialmente mas grande que la otra.

Los tres casos no cubren todas las posibilidades:
Ejemplo:
$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$

Suavidad

El teorema maestro maneja los casos donde los subproblemas tienen el mismo tamaño, sin embargo esto puede no ser así:

Ejemplo, el algoritmo mergesort
$T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n$

Sin embargo, si n es potencia de 2 los pisos y los techos pueden ignorarse y se tiene que
$T(n) = 2T\left(\frac{n}{2}\right) + n$
La cual es resoluble por teorema maestro.

En general, esto pueden ignorarse los pisos y los techos bajo ciertas condiciones.

Una función $f : \mathbb{N} \to \mathbb{R}^+$ es eventualmente no decreciente sii $\exists n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n+1) \geq f(n)$
Una función f es b-suave si

1. Es eventualmente no decreciente
2. $f(bn) \in O(f)$

Se dice que f es suave si $\forall b$: f es b-suave
Ejemplo, n es suave mientras que 2^n no lo es.

Propiedad: Si f es a-suave para $a \geq 2$, entonces es suave para todo b
Primero demuestro que f es a-suave para todo $b = a^k$ con $k \geq 1$
Esto lo pruebo usando inducción.
CB) $k = 1$, $b = a^1 = a$ y f resulta b-suave por hipotesis
HI) f es a^k suave
PI) $f(a^{k+1}n) = f(a \cdot a^k n) \in O(f(a^k n))$ por ser f a-suave, luego por HI $f(a^k n) \in O(f)$
Y por transitividad de O, $f(a^{k+1}n) \in O(f)$

\
Luego para un b genérico, se tiene que para k lo suficientemente grande, $b \leq a^k$(por ser a^k creciente)
Luego $f(bn) \leq f(a^k n)$ para n lo suficientemente grande(ya que f es eventualmente no decreciente). Luego por ser f a^k-suave, $f(bn) \in O(f)$. f es b-suave.

De aqui surge la regla de suavidad:
Sean f una funcion enventualmente no decreciente y g una función suave,
$\forall b \geq 2 : f(b^k) \in \Theta(g(b^k)) \implies \forall n, f(n) \in \Theta(g(n))$
Pruebo el caso del O:
$f(b^k) \in O(g(b^k)) \implies f(b^k) \leq c(g(b^k))$
Sea n tal que $b^k \leq n \leq b^{k+1}$
$f(n) \leq f(b^{k+1}) \leq c(g(b^{k+1})) \leq cc_b g(b^k) \leq cc_b c' g(n)$

Ejemplo, mergesort:
$T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n$
Sea $n = 2^k$, luego
$T(n) = 2T\left(\frac{n}{2}\right) + n$
Por teorema maestro:
$T(n) \in \Theta(n \lg n)$
$n \lg n$ es una función suave, por regla de suavidad, para cualquier n vale la recurrencia

---

Suavidad

Suavidad
Hipotesis) Sea f 2-suave
Sea pruebo primero para b entero de la forma $b = 2^k$
Por inducción sobre k
CB) $k = 1$
Vale por hipotesis
HI) f es 2^k suave
PI)
$f(2^{k+1}n) = f(2 \cdot 2^k n) \in O(f(2^k n))$
Luego por hipotesis inductiva y por transitividad de O
$f(2^{k+1}n) \in O(f)$

Para el caso general, sea b entero, existe k lo suficientemente grande tal que
$b \leq 2^k$ por la propiedad arquimedeana
Como f es eventualmente no decreciente, para n lo suficientemente grande
$f(bn) \leq f(2^k n)$
Luego $f(bn) \in O(f)$

Big oh test

29

> **29**
> Probar que para cualquier $r \in \mathbb{R}$, $n^r \in O(e^n)$
>
> $$\lim_{n\to\infty} \frac{n^r}{e^n} = \lim_{n\to\infty} \frac{e^{r\ln n}}{e^n} = \lim_{n\to\infty} e^{r\ln n - n}$$
>
> $$\lim_{n\to\infty} r\ln n - n = \lim_{n\to\infty} n\left(r\frac{\ln n}{n} - 1\right)$$
>
> $\lim_{n\to\infty} n = +\infty$
> $\lim_{n\to\infty} r\frac{\ln n}{n} - 1 = -1$
> Por lo tanto el limite anterior tiende a $-\infty$
> $\lim_{n\to\infty} \frac{n^r}{e^n} = \lim_{n\to\infty} e^{r\ln n - n} = 0$
>
> Por lo tanto $n^r \in O(e^n)$

44

> **44**
> Probar que $\log x \in O(x^\epsilon)\ \forall \epsilon > 0$
>
> $$\lim_{n\to\infty} \frac{\log n}{n^\epsilon} \overset{L'hopital}{=} \lim_{n\to\infty} \frac{\frac{1}{n}}{\epsilon \cdot n^{\epsilon-1}} = \frac{1}{\epsilon} \lim_{n\to\infty} \frac{1}{n^\epsilon} = 0$$
>
> Por lo tanto $\log n \in O(n^\epsilon)$
>
> ## Opción sin limites
>
> Hay que probar que para algún natural n_0 y algún real positivo c
>
> $$\forall n \geq n_0 : 0 \leq \log n \leq c \cdot n^\epsilon$$
>
> Suponiendo $c = 1$ hay que probar si para un cierto n_0 se tiene que $\forall n \geq n_0$
> $0 \leq \log n \leq n^\epsilon$
> $0 \leq \log n \iff n \geq 1$
>
> Ahora para probar si $\log n \leq n^\epsilon$
> Sea $f(n) = n^\epsilon - \log n$
> $f'(n) = \epsilon \cdot n^{\epsilon-1} - \frac{1}{n} = \frac{\epsilon \cdot n^\epsilon - 1}{n}$
> $f'(n) > 0 \iff \epsilon \cdot n^\epsilon - 1 > 0 \iff n > \sqrt[\epsilon]{\frac{1}{\epsilon}}$
> Por lo tanto se cumple que $\log n \leq n^\epsilon$ para $n > \sqrt[\epsilon]{\frac{1}{\epsilon}}$
> Tomando $n_0 = \max\left\{\left\lceil\sqrt[\epsilon]{\frac{1}{\epsilon}}\right\rceil, 1\right\}$, se cumple la propiedad. Por lo tanto, $\log n \in O(n^\epsilon)$.

45

> **45**

> $\log^2 x \in O(\log x)$

big_oh_fest

BIG 'OH' NOTATION PRACTICE EXERCISES

Definition 1. For a positive function g, we say that $f(x) = O(g(x))$ as $x \to \infty$ if there exists an x_0 and a constant $C > 0$, independent of x_0, such that

$$|f(x)| \leqslant Cg(x)$$

for all $x \geqslant x_0$.

So, the sign of $f(x)$ is irrelevant as are any constants; we are just interested in the dominant behaviour/general size. Throughout, the big 'Oh' terms are for $x \to \infty$.

1. Polynomials and rational functions

Prove the following

(1) $x = O(x)$
(2) $2x = O(x)$
(3) $-x/5 = O(x)$
(4) $2x + 5 = O(x)$
(5) $-3x + 8 = O(x)$
(6) $x = O(x^2)$
(7) $5x + 2 = O(x^2)$
(8) $x^2 - 2x + 10 = O(x^2)$
(9) $-x^2 + 5x + 1 = O(x^2)$
(10) $x^3 + 3x - 1 = O(x^3)$
(11) $-7x^3 - x^2 + x - 3 = O(x^3)$
(12) $x^4 + x^3 + x^2 + x + 1 = O(x^4)$
(13) $16(x + 7)^4 = O(x^4)$

(14) A general polynomial of degree n is $O(x^n)$.

(15) $\dfrac{1}{x} = O(1)$

(16) $\dfrac{1}{2x+1} = O(1/x)$

(17) $\dfrac{1}{x^2} = O(1)$

(18) $\dfrac{1}{x^2} = O(1/x)$

(19) $\dfrac{1}{2x^2+3x+2} = O(1/x^2)$

(20) $\dfrac{x}{x+1} = O(1)$

(21) $\dfrac{x}{2x^2-1} = O(1/x)$

(22) $\dfrac{x}{1-2x^2} = O(1/x)$

(23) $\dfrac{x}{x^3-1} = O(1/x^2)$

(24) $\dfrac{x^2}{2x^4-4x} = O(1/x^2)$

(25) $\dfrac{x^2}{1-x} = O(x)$

(26) $\dfrac{x^4}{x^2+3x-1} = O(x^2)$

Extra

Cálculo de complejidades a partir de límites

Cálculo de complejidades a partir de límites

> **Cálculo de complejidades a partir de límites**
> Dadas dos funciones $f, g : \mathbb{N} \to \mathbb{R}$ asintóticamente no negativas :
>
> 1. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) \in O(g)$
> 2. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \implies f(n) \in \Omega(g)$
> 3. Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k, k \in \mathbb{R}^+ \implies f(n) \in \Theta(g)$
>
> Demostración de 1)
> Si el limite es 0 entonces
> $\forall \epsilon > 0, \exists H \in \mathbb{N}, \forall n > H : \left|\frac{f(n)}{g(n)}\right| < \epsilon$
> En particular para un ϵ particular y para n lo suficientemente grande se cumple que $-\epsilon \cdot g(n) < f(n) < \epsilon \cdot g(n)$
> Por lo tanto tomando $c = \epsilon$ con ϵ un valor cualquiera y siendo n_0 su H asociado se cumple que $f(n) \in O(g)$.
>
> Demostración de 2)
> Si el limite es $+\infty$ entonces
> $\forall M > 0, \exists H > 0, \forall n > H : \frac{f(n)}{g(n)} > M$
> En particular se tiene que $M \cdot g(n) < f(n)$
> Tomando un M cualquiera como c y n_0 igual a su H asociado, se cumple la propiedad
> Demostración de 3)
> Ejercicio 8 practica 1

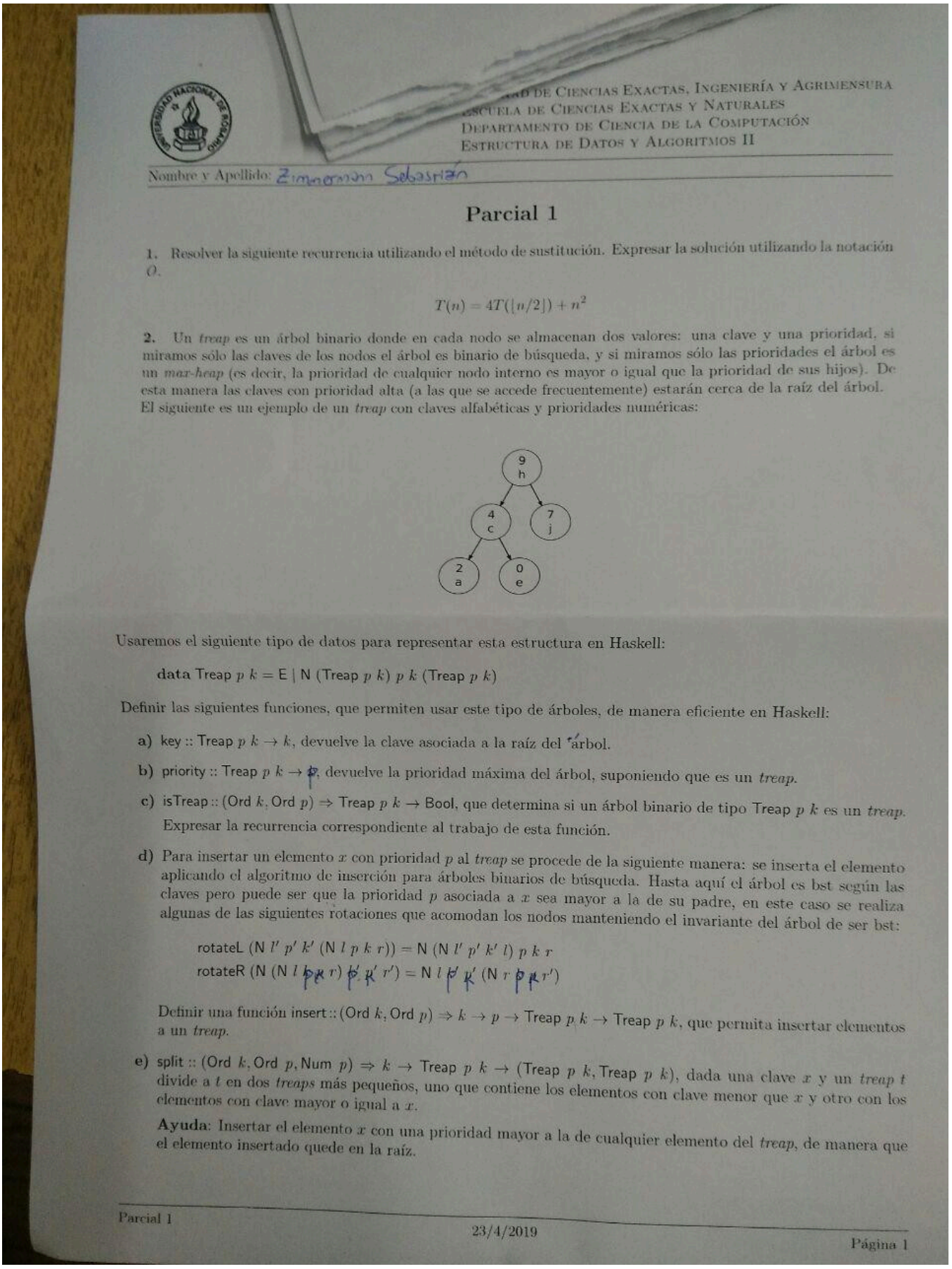
Parciales

📂 Parcial 1 2019
📂 Parcial 1 2023

Parcial 1 2019

📄 Parcial

Parcial



Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencia de la Computación
Estructura de Datos y Algoritmos II

Nombre y Apellido: Zimmermann Sebastián

Parcial 1

1. Resolver la siguiente recurrencia utilizando el método de sustitución. Expresar la solución utilizando la notación O.

$$T(n) = 4T(\lfloor n/2 \rfloor) + n^2$$

2. Un *treap* es un árbol binario donde en cada nodo se almacenan dos valores: una clave y una prioridad. si miramos sólo las claves de los nodos el árbol es binario de búsqueda, y si miramos sólo las prioridades el árbol es un *max-heap* (es decir, la prioridad de cualquier nodo interno es mayor o igual que la prioridad de sus hijos). De esta manera las claves con prioridad alta (a las que se accede frecuentemente) estarán cerca de la raíz del árbol. El siguiente es un ejemplo de un *treap* con claves alfabéticas y prioridades numéricas:

Usaremos el siguiente tipo de datos para representar esta estructura en Haskell:

```
data Treap p k = E | N (Treap p k) p k (Treap p k)
```

Definir las siguientes funciones, que permiten usar este tipo de árboles, de manera eficiente en Haskell:

a) key :: Treap p k → k, devuelve la clave asociada a la raíz del árbol.

b) priority :: Treap p k → p, devuelve la prioridad máxima del árbol, suponiendo que es un *treap*.

c) isTreap :: (Ord k, Ord p) ⇒ Treap p k → Bool, que determina si un árbol binario de tipo Treap p k es un *treap*. Expresar la recurrencia correspondiente al trabajo de esta función.

d) Para insertar un elemento x con prioridad p al *treap* se procede de la siguiente manera: se inserta el elemento aplicando el algoritmo de inserción para árboles binarios de búsqueda. Hasta aquí el árbol es bst según las claves pero puede ser que la prioridad p asociada a x sea mayor a la de su padre, en este caso se realiza algunas de las siguientes rotaciones que acomodan los nodos manteniendo el invariante del árbol de ser bst:

```
rotateL (N l' p' k' (N l p k r)) = N (N l' p' k' l) p k r
rotateR (N (N l p k r) p' k' r') = N l p k (N r p' k' r')
```

Definir una función insert :: (Ord k, Ord p) ⇒ k → p → Treap p k → Treap p k, que permita insertar elementos a un *treap*.

e) split :: (Ord k, Ord p, Num p) ⇒ k → Treap p k → (Treap p k, Treap p k), dada una clave x y un *treap* t divide a t en dos *treaps* más pequeños, uno que contiene los elementos con clave menor que x y otro con los elementos con clave mayor o igual a x.

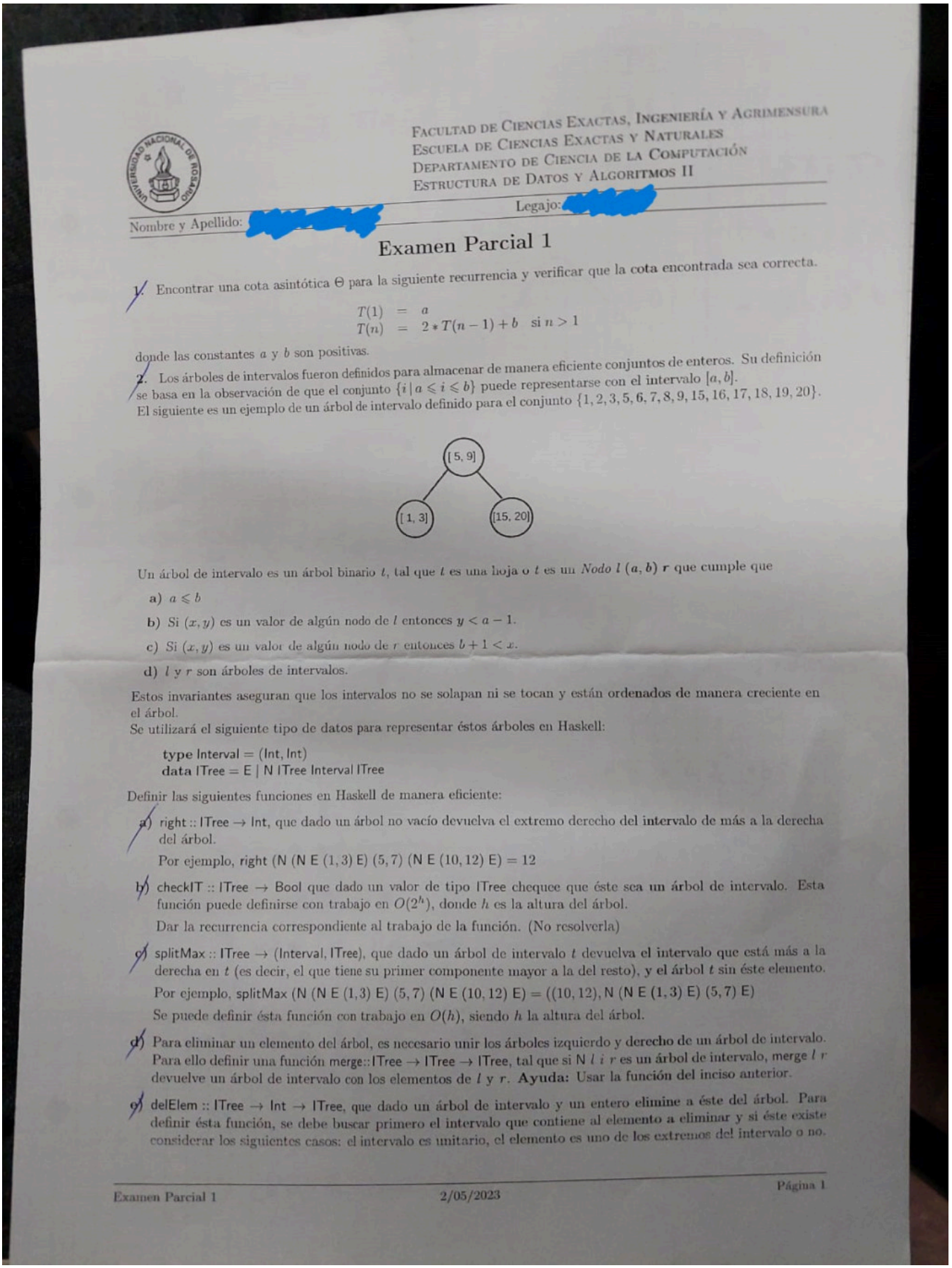
Ayuda: Insertar el elemento x con una prioridad mayor a la de cualquier elemento del *treap*, de manera que el elemento insertado quede en la raíz.

Parcial 1 — 23/4/2019 — Página 1

Parcial 1 2023

- parcial
- Resolución

parcial



FACULTAD DE CIENCIAS EXACTAS, INGENIERÍA Y AGRIMENSURA
ESCUELA DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN
ESTRUCTURA DE DATOS Y ALGORITMOS II

Nombre y Apellido: [redacted] Legajo: [redacted]

Examen Parcial 1

1. Encontrar una cota asintótica Θ para la siguiente recurrencia y verificar que la cota encontrada sea correcta.

$$\begin{aligned} T(1) &= a \\ T(n) &= 2 * T(n-1) + b \quad \text{si } n > 1 \end{aligned}$$

donde las constantes a y b son positivas.

2. Los árboles de intervalos fueron definidos para almacenar de manera eficiente conjuntos de enteros. Su definición se basa en la observación de que el conjunto $\{i \mid a \leqslant i \leqslant b\}$ puede representarse con el intervalo $[a, b]$.
El siguiente es un ejemplo de un árbol de intervalo definido para el conjunto $\{1, 2, 3, 5, 6, 7, 8, 9, 15, 16, 17, 18, 19, 20\}$.

Un árbol de intervalo es un árbol binario t, tal que t es una hoja o t es un *Nodo* l (a, b) r que cumple que

a) $a \leqslant b$

b) Si (x, y) es un valor de algún nodo de l entonces $y < a - 1$.

c) Si (x, y) es un valor de algún nodo de r entonces $b + 1 < x$.

d) l y r son árboles de intervalos.

Estos invariantes aseguran que los intervalos no se solapan ni se tocan y están ordenados de manera creciente en el árbol.
Se utilizará el siguiente tipo de datos para representar éstos árboles en Haskell:

```
type Interval = (Int, Int)
data ITree = E | N ITree Interval ITree
```

Definir las siguientes funciones en Haskell de manera eficiente:

a) right :: ITree → Int, que dado un árbol no vacío devuelva el extremo derecho del intervalo de más a la derecha del árbol.

Por ejemplo, right (N (N E (1,3) E) (5,7) (N E (10,12) E) = 12

b) checkIT :: ITree → Bool que dado un valor de tipo ITree chequee que éste sea un árbol de intervalo. Esta función puede definirse con trabajo en $O(2^h)$, donde h es la altura del árbol.

Dar la recurrencia correspondiente al trabajo de la función. (No resolverla)

c) splitMax :: ITree → (Interval, ITree), que dado un árbol de intervalo t devuelva el intervalo que está más a la derecha en t (es decir, el que tiene su primer componente mayor a la del resto), y el árbol t sin éste elemento.

Por ejemplo, splitMax (N (N E (1,3) E) (5,7) (N E (10,12) E) = ((10,12), N (N E (1,3) E) (5,7) E)

Se puede definir ésta función con trabajo en $O(h)$, siendo h la altura del árbol.

d) Para eliminar un elemento del árbol, es necesario unir los árboles izquierdo y derecho de un árbol de intervalo. Para ello definir una función merge::ITree → ITree → ITree, tal que si N l i r es un árbol de intervalo, merge l r devuelve un árbol de intervalo con los elementos de l y r. **Ayuda:** Usar la función del inciso anterior.

e) delElem :: ITree → Int → ITree, que dado un árbol de intervalo y un entero elimine a éste del árbol. Para definir ésta función, se debe buscar primero el intervalo que contiene al elemento a eliminar y si éste existe considerar los siguientes casos: el intervalo es unitario, el elemento es uno de los extremos del intervalo o no.

Resolución

Resolución

1. $T(1) = a$
   $2T(n-1) + b$

   Adivino que la recurrencia es $\Theta(2^n)$

   Primero pruebo el caso Ω:

   HI) quiero probar que para $n \geq n_0$ y $c \in \mathbb{R}^+$ con $n_0 \in \mathbb{N}$
   $c \cdot 2^n \leq T(n)$

   PI)
   $T(n+1) = 2 \cdot T(n) + b \geq 2 \cdot c \cdot 2^n + b = c \cdot 2^{n+1} + b \geq c \cdot 2^{n+1}, \forall c \geq 0$

   CB)
   $T(2) = 2a + b \geq 4c \iff c \leq \frac{2a+b}{4}$

   Por inducción, $T(n) \in \Omega(2^n)$ para $n_0 \geq 2$ y $c \leq \frac{2a+b}{4}$

   Ahora pruebo el caso del O:

   HI) quiero probar que para $n \geq n_0$ y $c \in \mathbb{R}^+$ con $n_0 \in \mathbb{N}$
   $T(n) \leq c \cdot 2^n - b$

   PI)
   $T(n+1) = 2T(n) + b \leq 2(c \cdot 2^n - b) + b = c \cdot 2^{n+1} - b, \forall c \geq 0$

   CB)
   $T(2) = 2a + b \leq 4c - b \iff \frac{a+b}{2} \leq c$

   Por inducción, $T(n) \in O(2^n)$

   Por lo tanto, $T(n) \in \Theta(2^n)$
2.

Practica 1

Ejercicio 10

Ejercicio 10

$$\lim_{n\to\infty} \frac{\sum_{i=1}^{n} i^k}{n^{k+1}} = \lim_{n\to\infty} \sum_{i=1}^{n} \frac{i^k}{n^{k+1}} = \lim_{n\to\infty} \sum_{i=1}^{n} \left(\frac{i}{n}\right)^k \cdot \frac{1}{n} = \int_0^1 x^k \, dx$$

La ultima igualdad vale ya que el ultimo limite representa una sumatoria de Riemann que usa intervalos equidistantes.

$$\int_0^1 x^k \, dx = \frac{1}{k+1} > 0$$

Por el Ejercicio 8 de la practica 1, $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$

Ejercicio 7

Ejercicio 7
a) Probar que $(n+a)^b \in \Theta(n^b)$

Primero compruebo que $(n+a)^b \in O(n^b)$
Quiero ver que $\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ \; \forall n \geq n_0 : 0 \leq (n+a)^b \leq c \cdot n^b$

La función $f(x) = \sqrt[b]{x}$ es una función creciente para $x \geq 0$, ya que $f'(x) = \frac{1}{b} \cdot x^{b-1}$ es positiva para $x \geq 0$.

Se tiene que
$0 \leq (n+a)^b \leq c \cdot n^b$
$\iff$
$0 \leq \sqrt[b]{(n+a)^b} \leq \sqrt[b]{c \cdot n^b}$
$\iff$
$0 \leq n+a \leq \sqrt[b]{c} \cdot n$
La proposición es verdadera para $n_0 = |\lceil a \rceil|$ y $c = 2^b$

Para el caso de Ω, hay que probar que
$0 \leq c \cdot n^b \leq (n+a)^b$

De la misma forma se tiene que
$0 \leq \sqrt[b]{c} \cdot n \leq n+a$
$n \geq (1 - \sqrt[b]{c}) \cdot n \geq -a$

Lo cual es cierto para $n_0 = 1$ y $c = 1$ si $a \geq 0$
Si $a \leq 0$ entonces es cierto para $n_0 = -a$ y $c = 2^b$

b)
Hay que probar que para todo $n \geq n_0$ existen constantes positivas c_1,c_2 tal que:
$0 \leq c_1 \cdot b^{n+a} \leq b^n \leq c_2 \cdot b^{n+a}$
Lo cual es cierto tomando $n_0 = 1$, $c_1 = c_2 = b^{-a}$

Ejercicio 8

Ejercicio 8

Si $\lim_{n\to\infty} \frac{f(x)}{g(x)} = k$ con $k \in \mathbb{R}^+$, entonces

$$\forall \varepsilon > 0, \exists H \in \mathbb{N}, \forall n > H : \left| \frac{f(n)}{g(n)} - k \right| < \varepsilon$$

En particular:
$(k-\varepsilon)g(n) < f(n) < (k+\epsilon)g(n)$, para ε y H dados

Tomando un $\varepsilon > k$ y su H asociado, se cumple $f(x) \in \Theta(g)$ para $c_1 = k - \varepsilon$ y $c_2 = k + \varepsilon$

Practica 2

Ejercicio 7

Ejercicio 7

Ejercicio 7
c)
Adivino que la solución pertenece a $\Theta(\log\log n)$

Primero pruebo la acotación superior por método de sustitución

Voy a probar por inducción que $T(n) \leq c\log\log n - 1$
PI)
$T(n) = T(\sqrt{n}) + 1 \leq c\log\log\sqrt{n} - 1 + 1 = c\log\log\sqrt{n}$
$= c\log\left(\frac{1}{2}\log n\right) = c\log\frac{1}{2} + c\log\log n = c\log\log n - c$
$\leq c\log\log n - 1 \iff c \geq 1$

CB)
$T(4) = T(2) + 1 \leq c\log\log 4 - 1 = c\log 2 - 1 = c - 1$
$\iff T(2) + 2 \leq c$

En particular se cumple que $T(n) \leq c\log\log n - 1 \leq c\log\log n$

Por lo tanto se cumple $T(n) \in O(\log\log n)$, para $n_0 = 4$ y $c = \max\{T(2) + 2, 1\}$

Ahora pruebo la acotación inferior por el mismo método

Voy a probar por inducción que $T(n) \geq c\log\log n$

PI)
$T(n) = T(\sqrt{n}) + 1 \geq c\log\log\sqrt{n} + 1 = c\log\log n - c + 1$
$\geq c\log\log n \iff c \leq 1$

CB)
$T(4) = T(2) + 1 \geq c\log\log 4 = c, \iff c \leq T(2) + 1$

Por lo tanto se cumple $T(n) \in \Omega(\log\log n)$ para $n_0 = 4$ y $c = \min\{1, T(2) + 1\}$

Practica 5

Ejercicio 12

> **Ejercicio 12**
> a)
> data Bin a = E | Node (Bin a) a (Bin a)
>
> insert:: Ord a => a -> Bin a -> Bin a
> insert a E = (Node E a E)
> insert a (Node l b r) = if a<b
> then Node (insert a l) b r
> else if a>b then
> Node l b (insert a r)
> else Node l b r
> CB) t=E
> E es un BST
> insert a E = Node E a E, por def.1 de insert
> (Nodfe E a E) es un BST. El caso base es valido
>
> HI) Si t es un BST, entonces insert a t es un BST
>
> PI)
> l y r son BSTs
>
> $t =$ Node $l\ b\ r$
>
> Si $a = b$, entonces vale por definición de t
>
> Si $a < b$:
> insert a (Node l b r)
> =\
> Node (insert a l) b r
>
> Por HI, (insert a l) es un BST, ademas todos los elementos en el subarbol izquierdo son menores a b y los elementos de r son mayores a b. Por lo tanto Node (insert a l) b r es un BST.
>
> La prueba con $a > b$ es identica
>
> Por PIP, insert a t es un BST para todo t

Ejercicio 2

> **Ejercicio 2**
> tad Pila (A:Set) where
> import Bool
> empty: Pila A
> push : A -> Pila A -> Pila A
> isEmpty: Pila A -> Bool

top: Pila A -> A
pop: Pila A -> Pila A

```
isEmpty vacia = True
isEmpty (push x q) = False

top (push x q) = x

pop (push x q) = q
```

Especificación usando el tad de secuencias
empty = ⟨⟩
push x $\langle x_1, \ldots, x_n \rangle = \langle x, x_1, \ldots, x_n \rangle$
isEmpty ⟨⟩ = True
isEmpty $\langle x_1, \ldots, x_n \rangle$ = False
top $\langle x_1, \ldots, x_n \rangle = x_1$
pop $\langle x_1, x_2, \ldots, x_n \rangle$ = $\langle x_2, \ldots, x_n \rangle$

Ejercicio 3

Ejercicio 3
tad Conjunto (A : Set) where
import Bool
vacio : Conjunto A
insertar : A → Conjunto A → Conjunto A
borrar : A → Conjunto A → Conjunto A
esVacio : Conjunto A → Bool
union : Conjunto A → Conjunto A → Conjunto A
interseccion : Conjunto A → Conjunto A → Conjunto A
resta : Conjunto A → Conjunto A → Conjunto A

---

insertar x (insertar x c) = insertar x c
insertar x (insertar y c) = insertar y (insertar x c)

borrar x vacio = vacio
borrar x (insertar x c) = c
borrar x (insertar y c) = insertar y (borrar x c)

union vacio b = b
union (insertar x a) b = insertar x (union a b)

resta a vacio = a
resta a (insertar y b) = borrar y (resta a b)

interseccion a b = resta a (resta a b)

La función choose tiene el problema que para conjuntos iguales puede dar valores diferentes
Por ejemplo, sean
$A = insertar\ 1\ insertar\ 2\ vacio$
$B = insertar\ 2\ insertar\ 1\ vacio$
Claramente $A = B$, sin embargo

$choose\ A = 1$
$choose\ B = 2$

Ejercicio 4

Ejercicio 4
tad PriorityQueue (A: Set, B: Ordered Set) where
import Bool
vacia: PriorityQueue A B
poner: A -> B -> PriorityQueue A B -> PriorityQueue A B
primero: PriorityQueue A B -> A
sacar: PriorityQueue A B -> PriorityQueue A B
esVacia: PriorityQueue A B -> Bool
union: PriorityQueue A B -> PriorityQueue A B -> PriorityQueue A B

---

primero (poner a b vacia) = a

primero (poner a x (poner b y c)) = if x< y then primero (poner b y c) else primero (poner a x c)

sacar vacio = vacio
sacar (poner a x vacio) = vacio
sacar (poner a x (poner b y c)) = if x< y then poner a x (sacar b y c) else poner b y (sacar a x c)

esVacia vacia=True
esVacia (poner x y q) = False

union vacia q = q
union (poner a b p) q = insertar a b (union p q)

Ejercicio 5

Ejercicio 5
tad BaIT (A: Ordered Set) where
import Maybe
empty: BaIt A
join: BaIT A -> Maybe A -> BaIT A -> BaIT A
size: BaIT A -> N
expose: BaIT A -> Maybe (BaIT A,BaIT A)

---

size empty = 0

Ejercicio 6

Ejercicio 6
Pruebo por inducción sobre las listas de pares $[(a, b)]$

CB) La lista vacia
((uncurry zip) ∘ unzip) []
<def ∘> =
(uncurry zip) (unzip [])
<def.1 unzip> =

(uncurry zip) ([],[])
<def.1 uncurry zip> =
[]
<.def id>
id []

HI) ((uncurry zip) ∘ zip) ps = id ps

PI)
Sea (xs,ys) = unzip ps
((uncurry zip) ∘ unzip) ((x,y):ps)
<def ∘> =
(uncurry zip) (unzip ((x,y):ps))
<def.3 unzip> =
(uncurry zip) (x:xs,y:ys)
<def.3 uncurry zip>
(x,y) : ((uncurry zip) (xs,ys))
=\
(x,y) : ((uncurry zip) (unzip ps))
<def ∘> =
(x,y) : (((uncurry zip) ∘ unzip) ps)
<.HI> =
(x,y) : (id ps)
<.def id>
(x,y) : ps
<.def id>
id (x,y) : ps

Por PIP y por principio de extensionalidad vale que (uncurry zip) ∘ zip = id

Tp paquito

24.ips.EDyA2.Final.04102024

24.ips.EDyA2.Final.04102024

de 1

Nombre y Apellido: Legajo:

Un lenguaje imperativo simple sólo permites variables de un único tipo, para esto se mantiene un estado con el nombre de las variables y sus valores. Considera el TAD de Estado:

```
tad Estado (N : String, A : Set) where
  import Maybe, Bool
  inicial : Estado N A
  update  : N → A → Estado N A → Estado N A
  lookfor : N → Estado N A → Maybe A
  free    : N → Estado N A → Estado N A
```

donde

- *inicial* representa el estado inicial de un programa donde no sean definido ninguna variable
- *update* permite actualizar el valor de una variable existente y si la variable no existe la agrega al estado con el valor dado.
- *lookfor* dado el nombre de una variable permite obtener el valor de esta si es que existe en el estado.
- *free* dado el nombre de una variable la elimina del estado.

1. Dar la especificación algebraica del **tad** Estado sin definir funciones auxiliares. Considere que pueden compararse los nombres de las variables N : String usando el $\equiv$. Tener en cuenta que para este TAD en particular deberemos también predicar sobre los constructores.

2. Dado los siguientes tipos de datos:

```
type Name = String
data Estado a = Inicial | Define Name a (Estado a)
```

Implementar la función *update*, *lookfor* y *free* especificadas en el TAD Estado

3. Dadas las siguientes definiciones:

$$
\begin{aligned}
&concat\ [\,] = [\,] \\
&concat\ (xs : xss) = xs +\!\!+ concat\ xss \\
&sum\ [\,] = 0 \\
&sum\ (x : xs) = x + sum\ xs
\end{aligned}
\qquad
\begin{aligned}
&[\,] +\!\!+ ys = ys \\
&xs +\!\!+ [\,] = xs \\
&(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)
\end{aligned}
$$

probar por inducción estructural que $(sum \circ concat) = (sum \circ (map\ sum))$
puede asumir válido el lema $sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys$

4. Los profesores Daniel y Alejandro discuten si usar TM o expandir la definición para resolver la siguiente recurrencia:

$$W(1) = 1$$
$$W(n) = 4W(\frac{n}{2}) + n^2$$

muestrales que se puede resolver de las dos formas:

a) Utilizar el teorema maestro y

b) expandiendo la definición en forma algebraica para encontrar las cotas asintóticas.