1. Demuestre la regla del límite para Θ.
2. Demuestre la prop. 12.
3. Considere un algoritmo que requiera un tiempo en la clase $\Theta(n^{\log 3})$ ¿Es correcto decir que requiere un tiempo en la clase $O(n^{1,59})$?, o ¿en la clase $\Omega(n^{1,59})$? o ¿en la clase $\Theta(n^{1,59})$? (*Nota:* $\log 3 = 1{,}58496\ldots$)
4. Dadas dos funciones de comportamiento f, g demostrar que:
   a) $O(f(n)) = O(g(n))$ sii $f(n) \in \Theta(g(n))$ sii $\Theta(f(n)) = \Theta(g(n))$.
   b) $O(f(n)) \subsetneq O(g(n))$ sii $f(n) \in O(g(n))$ pero $f(n)) \notin \Omega(g(n))$.
5. Ubique en la escala de funciones las siguientes funciones: $n \log n$, n^8, $n^{1+\epsilon}$, $(1+\epsilon)^n$, $n^2/\log n$ y $(n^2 - n + 1)^4$.

3.4. Cotas condicionales

Una clase más y algunas definiciones facilitarán el trabajo con los algoritmos conocidos como "divide y vencerás". Para que una función de comportamiento pertenezca a esta clase es necesario que la variable independiente (tamaño de ejemplar) cumpla una propiedad. El conjunto de funciones acotadas superiormente condicionadas por P es:

$$O(f(n)|P(n)) = \{g(n)|(\exists c > 0, N \in \mathbb{N})(\forall n \geq N)(P(n) \rightarrow g(n) \leq cf(n))\}$$

Similarmente definimos $\Omega(f(n)|P(n))$ y $\Theta(f(n)|P(n))$. Se dice que una función de comportamiento es *asintóticamente no decreciente* (andec) si $\exists N \in \mathbb{N}$ tal que $(\forall n \geq N)(f(n) \leq f(n+1)$. Dada una función andec, f, se dice que es *b-uniforme* si $f(bn) \in O(f(n))$, $b \in \mathbb{R}$. Cuando para toda $b \geq 2$, f es b-uniforme, f se llama *suave*. Las funciones n^k y $n \log n$ son suaves, pero ni $n^{\log n}$, ni 2^n lo son, pues crecen muy rápido. Por ejemplo, si suponemos que $n^{\log n}$ fuera suave entonces $(2n)^{\log 2n} \in O(n^{\log n})$ y por tanto $(2n)^{\log 2n} \leq cn^{\log n}$, pero $(2n)^{\log 2n}/n^{\log n} = 2n^2$ no puede acotarse por una constante. Concluimos la presente sección con el enunciado que generaliza la pertenencia a una clase condicionada para funciones suaves.

Proposición 13 (Regla de la Uniformidad) Si $f(n)$ es suave, $g(n)$ andec y $g(n) \in \Theta(f(n)|n = b^k)$, entonces $g(n) \in \Theta(f(n))$.

Demostración. Supongamos que $g(n) \in \Theta(f(n)|n = b^k)$. Sea $\underline{n} = b^{\lfloor \log_b n \rfloor}$ y $\bar{n} = b^{\lceil \log_b n \rceil + 1}$, así $\underline{n} \leq n \leq \bar{n}$. Demostraremos que $g(n) \in O(f(n))$ y $g(n) \in \Omega(f(n))$, partiendo de que $g(n) \in O(f(n)|n = b^k)$ y $g(n) \in \Omega(f(n)|n = b^k)$. Con base en las hipótesis: $g(n) \leq g(\bar{n}) = g(\underline{n}b) \leq cf(\underline{n}b) \leq acf(\underline{n}) \leq acf(n)$ para $n \geq \max\{1, b^N\}$. Similarmente: $g(n) \geq g(\underline{n}) \geq cf(\underline{n}) \geq \frac{c}{a} f(\underline{n}b) \geq \frac{c}{a} f(\bar{n}) \geq \frac{c}{a} f(n)$. □

Ejercicio 22

1. Diga la razón del uso de cada una de las desigualdades que aparecen en la demostración de la prop. 13.
2. Demuestre que 2^n no es suave.
3. ¿Cómo aplicaría la regla de la uniformidad a un algoritmo que busca un nodo en un árbol binario?

4. Análisis de Algoritmos

Cuando encuentra que varios algoritmos distintos solucionan el mismo problema, tiene que decidir cual es el más apropiado para la aplicación que desea desarrollar. Una herramienta esencial para este propósito es el *análisis de algoritmos*. Sólo después de que ha determinado la eficiencia de los distintos algoritmos será capaz de hacer una decisión bien informada. El problema es que no hay una fórmula mágica para analizar la eficiencia de los algoritmos. Principalmente el análisis es un asunto de juicio, intuición y experiencia. No obstante, existen algunas técnicas básicas que son útiles con frecuencia, tales como saber de que forma tratar con estructuras de control y ecuaciones de recurrencia. Esta sección cubre las técnicas más comúnmente usadas y las ilustra con ejemplos.

4.1. Análisis de estructuras de control

El análisis de algoritmos usualmente se lleva a cabo desde adentro hacia afuera. Primero determinamos el tiempo requerido por las instrucciones individuales (este tiempo frecuentemente está acotado por una constante); luego combinamos estos tiempos de acuerdo a las estructuras de control que componen las instrucciones en el programa. Algunas estructuras de control tales como la secuencia (poner una instrucción después de otra) son fáciles de analizar, mientras que otras, como los ciclos **while**, son más difíciles. En esta subsección daremos principios generales que son útiles en el análisis involucrado con las estructuras de control encontradas más frecuentemente, así como ejemplos de la aplicación de estos principios.

4.1.1. Secuencia

Sean P_1 y P_2 dos fragmentos de un algoritmo. Estos pudieran ser instrucciones simples o subalgoritmos complicados. Sean t_1 y t_2 los tiempos que consumen P_1 y P_2, respectivamente. Los tiempos pudieran depender de varios parámetros, tales como el tamaño del ejemplar. La **regla para secuenciar** dice que el tiempo requerido para calcular "$P_1; P_2$", esto es, primero P_1 y después P_2, simplemente es $t_1 + t_2$. Por la regla del máximo, este tiempo está en $\Theta(\max(t_1, t_2))$.

A pesar de su simplicidad, aplicar esta regla algunas veces es menos obvio de lo que pudiera parecer. Por ejemplo, pudiera suceder que uno de los parámetros que controla a t_2 depende del resultado del cálculo efectuado por P_1. Así, el análisis de "$P_1; P_2$" no simpre puede ser realizado considerando independientemente a P_1 y P_2.

4.1.2. Ciclos "for"

Los ciclos **for** son los más fáciles de analizar. Considere el siguiente ciclo.

for $i \leftarrow 1$ **to** m **do** $P(i)$

Aquí y en el resto de nuestra discusión, adoptaremos la convención de que $m = 0$ no es un error; simplemente significa que la instrucción controlada $P(i)$ nunca es ejecutada. Suponga que este ciclo es parte de un algoritmo más extenso, el cual trabaja sobre un ejemplar de tamaño n. (Tenga cuidado en no confundir m y n) El caso más fácil es cuando el tiempo consumido por $P(i)$ realmente no depende de i, no obstante pudiera depender del tamaño del ejemplar, más generalmente, del ejemplar en sí. Sea t el tiempo requerido para calcular $P(i)$. En este caso, el análisis obvio del ciclo es que $P(i)$ es ejecutado m veces, cada vez con costo t, así el tiempo total requerido por el ciclo simplemente es $l = mt$. A pesar de que este enfoque usualmente es adecuado, existe un peligro potencial: no tomamos en consideración el tiempo necesario para *controlar el ciclo*. Después de todo, nuestro ciclo **for** es una abreviación para algo como el siguiente ciclo **while**.

```
i ← 1
while i ≤ m do
    P(i)
    i ← i + 1
```

En la mayoría de las situaciones es razonable contar como de costo unitario la prueba $i \leq m$, la instrucción $i \leftarrow 1$ y $i \leftarrow i + 1$ y las operaciones secuencia (**go to**) implicitas en el ciclo **while**. Sea c una cota superior sobre el tiempo requerido por cada una de estas operaciones. El tiempo l consumido por el ciclo está acotado superiormente por