$$
\begin{array}{rcll}
l & \leq & c & \text{para } i \leftarrow 1 \\
 & + & (m+1)c & \text{para las pruebas } i \leq m \\
 & + & mt & \text{para las ejecuciones de } P(i) \\
 & + & mc & \text{para las ejecuciones de } i \leftarrow i+1 \\
 & + & mc & \text{para las operaciones secuencia} \\
 & \leq & (t+3c)m + 2c &
\end{array}
$$

Más aún, este tiempo está claramente acotado inferiormente por mt. Si c es despreciable comparado con t, nuestro aproximado anterior de que l era aproximadamente igual a mt fue por tanto justificado, excepto por un caso crucial: $l \approx mt$ es completamente erroneo cuando $m = 0$ (¡aún es peor cuando m es negativo!). Cuando estudiemos la técnica del barómetro, veremos que despreciar el tiempo requerido para el control del ciclo, puede guiarnos a errores serios en tales circunstancias.

Resista la tentación de decir que el tiempo consumido por el ciclo está en $\Theta(mt)$ con el pretexto de que a la notación Θ se le exige ser efectiva a partir de un umbral tal como $m \geq 1$. El problema con este argumento, es que, si en efecto estamos analizando el algoritmo entero más que simplemente el ciclo **for**, el umbral implicado por la notación Θ involucra a n, el tamaño del ejemplar, en vez de m, el número de veces que pasamos por el ciclo, $m = 0$ pudiera ocurrir para valores de n arbitrariamente grandes. Por otro lado, se puede demostrar (le sugerimos que lo haga) que si t está acotado inferiormente por una constante (lo cual simpre es el caso en la práctica) y si existe un umbral n_0 tal que $m \geq 1$ siempre que $n \geq n_0$, entonces l en verdad está en $\Theta(mt)$ cuando l, m y t son considerados como funciones de n.

El análisis del ciclo **for** es más interesante cuando el tiempo $t(i)$ requerido para $P(i)$ varía como una función de i. (En general, el tiempo requerido para $P(i)$ pudiera depender no sólo de i sino también del tamaño n del ejemplar, incluso del ejemplar mismo.) Si despreciamos el tiempo consumido por el control del ciclo, lo cual usualmente es adecuado cuando $m \geq 1$, el mismo ciclo **for**

$$\textbf{for } i \leftarrow 1 \textbf{ to } m \textbf{ do } P(i)$$

consume un tiempo dado no por una multiplicación sino por una suma: esto es $\sum_{i=1}^{m} t(i)$.

Ilustraremos el análisis de ciclos **for** con un algoritmo iterativo simple para calcular la secuencia de Fibonacci. Recuerde que el algoritmo es

```
function FIBITER(n)
1  i ← 1
2  j ← 0
3  for k ← 1 to n  do
4     j ← i + j
5     i ← j − i
6  return j
```

Si contamos todas las operaciones aritméticas con costo unitario, las instrucciones dentro del ciclo **for** consumen tiempo constante. Suponga que el tiempo consumido por estas instrucciones está acotado superiormente por alguna constante c. Sin tomar en cuenta el control del ciclo, el tiempo consumido por el ciclo **for** está acotado superiormente por n veces esta constante: nc. Ya que las instrucciones antes y después del ciclo consumen un tiempo despreciable, concluimos que el algoritmo consume un tiempo en $O(n)$. Un razonamiento similar nos lleva a que este tiempo también está en $\Omega(n)$, así está en $\Theta(n)$.

No obstante, vimos en la subsección 2.4 que no es razonable contar como de costo unitario, las adiciones involucradas en el cálculo de la secuencia de Fibonacci, al menos que n sea muy pequeño. Por lo tanto, debemos tomar en cuenta el hecho de que una instrucción tan simple como "$j \leftarrow i+j$" se encarece incrementalmente cada vez que pasamos por el ciclo. Es fácil programar adiciones y sustracciones de enteros largos, tal que el tiempo necesario para sumar y sustraer dos enteros, esté en el orden exacto del número de dígitos del operando más grande. Para determinar el tiempo consumido por la $k-$ésima pasada por el ciclo, necesitamos conocer la longitud de los enteros involucrados. Se puede demostrar por inducción matemática que los valores de i y j al final de la $k-$ésima iteración son respectivamente f_{k-1} y f_k. Esto es precisamente por lo que el algoritmo trabaja: regresa el valor de j al final de la $n-$ésima iteración, el cual es por lo tanto f_n, como se requiere. Más aún, a partir de la fórmula de de Moivre se puede demostrar que el tamaño de f_k está en $\Theta(k)$. Por lo tanto la $k-$ésima iteración consume un tiempo $\Theta(k-1) + \Theta(k)$, lo cual es lo mismo que $\Theta(k)$. Sea c una constante tal que este tiempo está acotado superiormente por ck para todo $k \geq 1$. Si despreciamos el tiempo requerido por el control del ciclo y por las instrucciones antes y después del ciclo, podemos concluir que el tiempo requerido por el algoritmo está acotado superiormente por

$$\sum_{k=1}^{n} ck = c \sum_{k=1}^{n} k = c\frac{n(n+1)}{2} \in O(n^2).$$

Un razonamiento similar nos conduce a que este tiempo está en $\Omega(n^2)$ y por lo tanto en $\Theta(n^2)$. Así, en el análisis de *Fibiter*, contar como de costo unitario a las operaciones aritméticas, constituye una diferencia crucial con respecto a no contarlas como de costo unitario.

El análisis de ciclos **for** que inician en valores distintos de 1 o que se ejecutan a pasos más grandes, debe ser obvio en este punto. Por ejemplo considere el siguiente ciclo.

$$\textbf{for } i \leftarrow 5 \textbf{ to } m \textbf{ step } 2 \textbf{ do } P(i)$$

Aquí, $P(i)$ se ejecuta $((m-5) \div 2) + 1$ veces siempre que $m \geq 3$. (Para que un ciclo **for** tenga sentido, el punto final debe ser siempre al menos tan grande como el punto inicial *menos* el paso).

4.1.3. Llamados recursivos

El análisis de algoritmos recursivos usualmente es directo, al menos hasta cierto punto. Una simple inspección al algoritmo frecuentemente da lugar a una ecuación de recurrencia que imita el flujo de control en el algoritmo. Una vez que ha sido obtenida la ecuación de recurrencia, se pueden aplicar técnicas generales para resolver dichas ecuaciones y transformarlas a la notación asintótica no recursiva, la cual es más simple.

Como un ejemplo, considere de nuevo el problema de calcular la secuencia de Fibonacci, pero esta vez con el algoritmo recursivo *Fibrec*

```
function FIBREC(n)
1  if n < 2  then
2     return n
3     else  return FIBREC(n − 1) + FIBREC(n − 2)
```

Sea $T(n)$ el tiempo consumido por un llamado a $Fibrec(n)$. Si $n < 2$, el algoritmo simplemente regresa n, lo cual consume tiempo constante a. De otra manera, la mayor parte del trabajo es gastaso en los dos llamados recursivos, los cuales consumen respectivamente tiempo $T(n-1)$ y $T(n-2)$. Más aún, debe ser ejecutada una adición que involucra a f_{n-1} y a f_{n-2} (los cuales son los valores regresados por los llamados recursivos), así como, el control de la recursión y la prueba "**if** $n < 2$". Sea $h(n)$ que denota el trabajo involucrado en esta adición y control, esto es, el tiempo requerido por un llamado a $Fibrec(n)$ ignorando el tiempo gastado dentro de los dos llamados recursivos. Por definición de $T(n)$ y $h(n)$ obtenemos la siguiente recurrencia.

$$T(n) = \begin{cases} a & \text{si } n = 0 \text{ o } n = 1, \\ T(n-1) + T(n-2) + h(n) & \text{en otro caso} \end{cases} \qquad (1)$$

Si contamos las adiciones de costo unitario, $h(n)$ está acotada por una constante. Al aplicarle a (1) técnicas para solución de recurrencias encontramos que $T(n) \in O(f_n)$. Un razonamiento similar muestra que $T(n) \in \Omega(f_n)$ y por tanto $T(n) \in \Theta(f_n)$. Usando la fórmula de de Moivre, concluimos que $Fibrec(n)$ consume tiempo exponencial en n. Éste es *doble* exponencial en el tamaño del ejemplar ya que el *valor* de n es exponencial en el *tamaño* de n.