está también el orden de n^2. Así, el algoritmo de ordenamiento por inserción consume tiempo cuadrático tanto en el peor caso, como en el caso promedio, no obstante que para algunos ejemplares puede ser mucho más rápido. Existe otro algoritmo de ordenamiento (quicksort) que también consume tiempo cuadrático en el peor de los casos, pero que sólo requiere un tiempo en el orden de $n \log n$ en el caso promedio. Aunque este algoritmo tiene un peor caso malo, ya que el desempeño cuadrático es malo para un algoritmo de ordenamiento, en su caso promedio probablemente es el algoritmo conocido más rápido, al menos de entre los algoritmos que usan la técnica de ordenamiento en el lugar, es decir, aquellos que no requieren espacio de almacenamiento adicional.

Usualmente es más difícil analizar el comportamiento promedio de un algoritmo que analizar su comportamiento en el peor caso. Peor aún, el análisis del comportamiento promedio puede ser engañoso, cuando los ejemplares a resolver no son elegidos aleatoriamente, en el momento que el algoritmo es usado en la práctica. Por ejemplo, hemos enunciado que ordenamiento por inserción toma tiempo cuadrático en promedio, cuando todos los $n!$ ordenamientos iniciales posibles son igualmente probables. Pero en muchas aplicaciones esta condición puede no ser real. Por ejemplo, si un programa de ordenamiento es usado para actualizar un archivo, podría la mayoría de las veces enfrentarse con datos que están muy cerca del orden deseado, es decir, con sólo unos pocos elementos fuera de su lugar. En este caso su comportamiento promedio calculado, bajo la hipótesis de que los ejemplares son elegidos aleatoriamente, será una guía pobre con respecto a su desempeño real.

Por lo tanto, para que sea útil el análisis del comportamiento promedio de un algoritmo, se requiere tener algún conocimiento a priori de la distribución de los ejemplares que serán resueltos. Esto normalmente es un requerimiento no realista. Especialmente cuando un algoritmo se usa como un procedimiento interno en algún algoritmo más complejo, podría ser impráctico estimar cuales ejemplares son más probables de encontrar y cuales sólo ocurrirán rara vez. Afortunadamente existen algoritmos que pueden "desordenar" los datos para que queden con cierta distribución.

En lo que sigue estaremos interesados sólo en el análisis del peor caso al menos que enunciemos lo contrario.

2.4. Operación elemental

Una *operación elemental* es aquella cuyo tiempo de ejecución se puede acotar superiormente por una constante, que sólo depende de la implementación que en particular use una máquina, un lenguaje de programación y así sucesivamente. Así la constante no depende ni del tamaño, ni de los otros parámetros del ejemplar que se está considerando. Dado que estamos involucrados en tiempos de ejecución de algoritmos que están por debajo de una constante multiplicativa, lo único que importa en el análisis es el número de operaciones elementales ejecutadas, no el tiempo exacto que requiere cada uno de ellos.

Por ejemplo, suponga que cuando analizamos un algoritmo, encontramos que para resolver un ejemplar de un cierto tamaño se requieren ejecutar a adiciones, m multiplicaciones y s instrucciones de asignamiento. Suponga que también sabemos que una adición nunca consume más de t_a microsegundos, una multiplicación nunca más de t_m microsegundos y un asignamiento nunca más de t_s microsegundos, donde t_a, t_m y t_s son constantes que dependen de la máquina usada. Por tanto, adición, multiplicación y asignamiento pueden ser consideradas operaciones elementales. El tiempo total t que requiere nuestro algoritmo puede acotarse mediante

$$\begin{aligned} t &\leq at_a + mt_m + st_s \\ &\leq \text{máx}(t_a, t_m, t_s) \times (a + m + s), \end{aligned}$$

esto es, t está acotado por un múltiplo constante del número de operaciones elementales que son ejecutadas.

Ya que no es importante el tiempo requerido por cada operación elemental, simplificaremos diciendo que las operaciones elementales se pueden ejecutar con *costo unitario*.

En la descripción de un algoritmo, una sola línea de código puede corresponder a un número variable de operaciones elementales. Por ejemplo, si T es un arreglo de n elementos ($n > 0$), el tiempo requerido para calcular

$$x \leftarrow \text{mín}\{T[i] | 1 \leq i \leq n\}$$

incrementa con n ya que es una abreviación para

```
function MIN(T[1...n])
1  x ← T[1]
2  for i ← 2 to n  do
3     if T[i] < x  then
4        x ← T[i]
5  return x
```

Similarmente, algunas operaciones matemáticas son tan complejas que no se pueden considerar elementales. Si nos permitiéramos contar con costo unitario las operaciones calcular un factorial y prueba por divisibilidad, sin considerar el tamaño de los operandos, entonces por el teorema de Wilson (el cual enuncia que el entero n divide a $(n-1)! + 1$ si y sólo si n es primo para todo $n > 1$) podríamos probar la primalidad de un entero con eficiencia sorprendente.

```
function WILSON(n)
1  if n divide exactamente a  (n − 1)! + 1  then
2     return true
3     else  return false
```

El ejemplo al inicio de esta subsección sugiere que podemos considerar como de costo unitario a las operaciones adición y multiplicación, ya que se asume que el tiempo requerido por estas operaciones se puede acotar mediante una constante. En teoría, no obstante, estas operaciones no son elementales ya que el tiempo necesario para ejecutarlas incrementa con el tamaño de los operandos. En la práctica, por otro lado, pudiera ser adecuado considerarlas como operaciones elementales, siempre y cuando los operandos involucrados sean de un tamaño razonable en los ejemplares que esperamos encontrar. Dos ejemplos ilustrarán lo que queremos decir.

```
function SUM(n)
1  sum ← 0
2  for i ← 1 to n  do
3     sum ← sum + i
4  return sum
```

```
function FIBONACCI(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
```

En el algoritmo llamado *Sum* el valor de *sum* permanece razonable para todos los ejemplares que en la práctica se esperan. Si estamos usando una máquina con palabras de 32 bits, todas las adiciones pueden ser ejecutadas directamente siempre y cuando n no sea mayor a 65535. No obstante, en teoría, el algoritmo debe trabajar para *todos* los valores posibles de n. Ninguna máquina real en efecto puede ejecutar estas adiciones con costo unitario si n es elegido suficientemente grande. El análisis del algoritmo por tanto debe depender del dominio pretendido de la aplicación (no del dominio del problema).

La situación es diferente en el caso de *Fibonacci*. Aquí es suficiente tomar $n = 47$ para que la última adición "$j \leftarrow i + j$" cause desbordamiento aritmético sobre una máquina de 32 bits. Para guardar el resultado que corresponde a $n = 65535$ necesitaríamos 45496 bits, o más de 1420 palabras de computadora. Por la tanto, como un aspecto práctico no es realista considerar que estas operaciones pueden ejecutarse con costo unitario. Más bien, debemos atribuirles un costo proporcional a la longitud de los operandos involucrados. En la sección **??** se mostrará que este algoritmo consume tiempo