Ciertos ordenes ocurren tan frecuentemente que vale la pena darles un nombre. Por ejemplo, suponga que el tiempo que toma un algoritmo para resolver un ejemplar de tamaño n nunca es mayor a cn segundos donde c es una constante apropiada. Entonces decimos que el algoritmo toma un tiempo en el orden de n, o más simplemente que toma tiempo *lineal*. En este caso también hablamos de un *algoritmo lineal*. Si un algoritmo nunca toma más de cn^2 segundos para resolver un ejemplar de tamaño n, entonces decimos que toma tiempo en el orden de n^2, o tiempo *cuadrático*, y le llamaremos un *algoritmo cuadrático*. Similarmente un algoritmo es *cúbico, polinomial o exponencial* si toma un tiempo en el orden de n^3, n^k o c^n, respectivamente, donde k y c son constantes adecuadas.

No caiga en la trampa de olvidar completamente a las *constantes ocultas*, como son llamadas las constantes multiplicativas usadas en estas definiciones. Comúnmente ignoraremos los valores exactos de estas constantes y asumiremos que todas ellas tienen el mismo orden de magnitud. Esto nos permitirá decir, por ejemplo, que un algoritmo lineal es más rápido que uno cuadrático sin preocuparnos si nuestra afirmación es verdadera en todos los casos. Sin embargo, algunas veces es necesario ser más cuidadoso.

Considere por ejemplo dos algoritmos cuyas implementaciones sobre una máquina dada toman respectivamente n^2 días y n^3 segundos para resolver un ejemplar de tamaño n, solamente para **¡ejemplares que requieren más de 20 millones de años en ser resueltos!** es que el algoritmo cuadrático llega a ser más rápido que el algoritmo cúbico. Desde un punto de vista teórico, el primero es *asintóticamente* mejor que el segundo; es decir, su desempeño es mejor sobre todos los ejemplares suficientemente grandes. Desde un punto de vista práctico ciertamente preferiremos al algoritmo cúbico. No obstante el algoritmo cuadrático puede ser asintóticamente mejor, su constante oculta es tan grande que lo descarta para considerarlo sobre ejemplares de tamaño normal.

2.3. Análisis del peor caso y del caso promedio

El tiempo que consume un algoritmo o el espacio de almacenamiento que usa, puede variar considerablemente entre dos ejemplares distintos del mismo tamaño. Para ilustrar esto, considere dos algoritmos de ordenamiento (ascendente) elementales: ordenamiento por *inserción* y ordenamiento por *selección*.

```
procedure INSERT(T[1...n])
1  for i ← 2 to n  do
2     x ← T[i]
3     j ← i − 1
4     while j > 0  and  x < T[j]  do
5        T[j + 1] ← T[j]
6        j ← j − 1
7     T[j + 1] ← x
```

```
procedure SELECT(T[1...n])
1  for i ← 1 to n − 1  do
2     minj ← i
3     minx ← T[i]
4     for j ← i + 1 to n  do
5        if T[j] < minx  then
6           minj ← j
7           minx ← T[j]
8     T[minj] ← T[i]
9     T[i] ← minx
```

Ejercicio 14

1. Simule la operación de los algoritmos de ordenamiento sobre algunos arreglos pequeños, para asegurarse que entiende cómo trabajan.
2. Simule los algoritmos de ordenamiento por inserción y por selección sobre los siguientes dos arreglos: $U = [1, 2, 3, 4, 5, 6]$ y $V = [6, 5, 4, 3, 2, 1]$ ¿Sobre cuál de los arreglos U o V se ejecuta más rápido ordenamiento por inserción?. La misma pregunta pero ahora considerando ordenamiento por selección. Justifique sus respuestas.
3. Suponga que trata de "ordenar" el arreglo $W = [1, 1, 1, 1, 1, 1]$ cuyos elementos son iguales, usando: (a) ordenamiento por inserción y (b) ordenamiento por selección. ¿Cómo se compara esto a ordenar los arreglos U y V del ejercicio 14.2?

El ciclo principal en ordenaminto por inserción, busca sucesivamente cada elemento del arreglo desde el segundo hasta el $n-$ésimo y lo inserta apropiadamente entre sus predecesores en el arreglo. Ordenamiento por selección trabaja tomando al elemento más pequeño en el arreglo y llevándolo al inicio; luego toma al siguiente más pequeño y lo pone en la segunda posición en el arreglo; y así sucesivamente.

Sean U y V dos arreglos de n elementos, tales que U está ordenado ascendentemente y V está ordenado descendentemente. Si resolvió bien el ejercicio 14.2 habrá notado que ambos algoritmos consumen más tiempo sobre V que sobre U. En efecto, el arreglo V representa el peor caso posible para estos dos algoritmos: ningún arreglo de n elementos requiere más trabajo. Sin embargo, el tiempo requerido por el algoritmo de ordenamiento por selección no es muy sensible al orden original del arreglo que ha de ordenarse: la prueba "**if** $T[j] < minx$"se ejecuta exactamente el mismo número de veces en cada caso. La variación en tiempo de ejecución, es debida solamente, al número de veces que son ejecutadas las asignaciones en la parte **then** de dicha prueba. Si se programa este algoritmo y se ejecuta sobre una máquina, encontrará que el tiempo requerido para ordenar un cierto número de elementos no variará más del 15 % cualquiera que sea el orden inicial de los elmentos a ordenar. Como se mostrará en la sección **??**, el tiempo requerido por $select(T)$ es cuádratico, sin importar el orden inicial de los elementos.

La situación es diferente si comparamos los tiempos que toma el algoritmo de ordenamiento por inserción sobre los dos arreglos. Ya que la condición que controla el ciclo **while** es falsa siempre desde el principio, $insert(U)$ es muy rápido y consume tiempo lineal. Por otro lado, $insert(V)$ consume tiempo cudrático porque el ciclo **while** se ejecuta $i - 1$ veces para cada valor de i, nuevamente refierase a la sección **??** para más detalle. La variación en tiempo entre estos dos ejemplares es por lo tanto considerable. Más aún, esta variación aumenta conforme aumenta el número de elementos a ordenar. En alguna implementación del algoritmo de ordenamiento por inserción, se encontró que consume menos de un quinto de segundo para ordenar un arreglo de 5000 elementos que originalmente estaban en orden ascendente, pero consumió tres y medio minutos en ordenar un arreglo con la misma cantidad de elementos, pero inicialmente en orden descendente, es decir, mil veces más.

Si pueden ocurrir variaciones tan grandes, ¿Cómo podemos hablar del tiempo que consume un algoritmo solamente en términos del tamaño del ejemplar a resolver? La respuesta es que usualmente consideraremos el *peor caso* del algoritmo, esto es, para cada tamaño de ejemplar sólo consideraremos aquellos ejemplares sobre los que el algoritmo requiere más tiempo. Esto es por lo que dijimos en la sección anterior que un algoritmo debe ser capaz de resolver todo ejemplar de tamaño n en no más de $ct(n)$ segundos, para una constante apropiada c que depende de la implementación, si decimos que un algoritmo se ejecuta en un tiempo en el orden de $t(n)$: implicitamente tenemos en mente el peor caso.

El análisis del peor caso es apropiado para un algoritmo cuyo tiempo de respuesta es crítico. Por ejemplo, si se trata de controlar una planta de energía nuclear, es crucial conocer un límite superior sobre el tiempo de respuesta del sistema, no importando el ejemplar particular que se resolverá. Por otro lado, si un algoritmo será usado varias veces sobre muchos ejemplares distintos, pudiera ser más importante conocer el tiempo de ejecución *promedio* sobre instancias de tamaño n. Vimos que el tiempo que consume el algoritmo de ordenamiento por inserción varía entre n y n^2. Si podemos calcular el tiempo promedio que consume el algoritmo, sobre las $n!$ formas distintas de ordenar inicialmente n elementos distintos, tendremos una idea del tiempo probable que consumirá para ordenar