0.1 Insert Sort

El loop principal (`for`) recorre cada elemento. Luego tiene un loope secundario (`while`) que ayuda a insertar cada elemento en el lugar correcto del arreglo, entre sus predecesores (esto es, hay un índice `j` que recorre el arreglo en sentido inverso)

```
procedure insert(T[1..n])
    for i = 2 to n
        // Me paro delante de los numeros que quiero analizar.
        x = T[i]
        j = i - 1
        while j > 0 and x < T[j] do
            // Recorro los números hacia atras,
            // Siempre que mi x (el valor que estoy comparando)
            // sea menor a los valores que me encuentro (T[J]),
            // muevo hacia adelante el valor que me encuentro (T[j])
            T[j+1] = T[j]
            j = j - 1
        // en el final, asígno al último valor de j
        // el que estoy analizando (x)
        T[j+1] = x
```

probemos ejecutar este código con $T = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]$

1. $T = [1, 3, 4, 1, 5, 9, 2, 6, 5, 3]$
2. $T = [1, 3, 4, 1, 5, 9, 2, 6, 5, 3]$
3. $T = [1, 1, 3, 4, 5, 9, 2, 6, 5, 3]$
4. $T = [1, 1, 3, 4, 5, 9, 2, 6, 5, 3]$
5. $T = [1, 1, 3, 4, 5, 9, 2, 6, 5, 3]$
6. $T = [1, 1, 2, 3, 4, 5, 9, 6, 5, 3]$
7. $T = [1, 1, 2, 3, 4, 5, 6, 9, 5, 3]$
8. $T = [1, 1, 2, 3, 4, 5, 5, 6, 9, 3]$
9. $T = [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]$

Si lo ejecutamos con $U = [1, 2, 3, 4, 5, 6]$, queda igual en todas las itereaciones del for.

Si lo ejecutamos con $V = [6, 5, 4, 3, 2, 1]$
1. $V = [5, 6, 4, 3, 2, 1]$
2. $V = [4, 5, 6, 3, 2, 1]$
3. $V = [3, 4, 5, 6, 2, 1]$
4. $V = [2, 3, 4, 5, 6, 1]$
5. $V = [1, 2, 3, 4, 5, 6]$

Observación: En una lista ya ordenada (cómo es el caso de U) vemos que el tiempo de ejecución es lineal: sólo requiere recorrer los elementos de U, pero no comporarlos con el resto.

0.2 Select Sort

Busca el elemento más pequeño del arreglo. Luego, lo mueve al primer lugar. Busca el segundo más pequeño, lo mueve al segundo lugar. Así, sucesivamente.

```
proc select(T[1..n])
  for i=1 to n
    // inicio suponiendo que
    // el "índice mínimo" es i, y el
    // "valor mínimo" es T[i]
    minj = i
```

```
    minx = T[i]

    // corrijo si esto no es así
    for j=i+1 to n
      if T[j] < minx
        minj = j
        minx = T[j]

    // intercambio los valores
    // cómo minj > i (pues j > i)
    // asigno T[i] a minj, y así
    // me queda el mayor con el mayor
    T[minj] = T[i]

    // luego, al índice menor (i),
    // le asigno el valor menor (minx)
    T[i] = minx
```

Ejecución con $V = [1, 3, 4, 1, 5, 9, 2, 6, 5, 3]$

1. $T = [1, 3, 4, 1, 5, 9, 2, 6, 5, 3]$
2. $T = [1, 1, 4, 3, 5, 9, 2, 6, 5, 3]$
3. $T = [1, 1, 2, 3, 5, 9, 4, 6, 5, 3]$
4. $T = [1, 1, 2, 3, 5, 9, 4, 6, 5, 3]$
5. $T = [1, 1, 2, 3, 3, 9, 4, 6, 5, 5]$
6. $T = [1, 1, 2, 3, 3, 4, 9, 6, 5, 5]$
7. $T = [1, 1, 2, 3, 3, 4, 5, 6, 9, 5]$
8. $T = [1, 1, 2, 3, 3, 4, 5, 5, 9, 6]$
9. $T = [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]$
10. $T = [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]$

Observación: La clausula `T[j]<minx` se ejecuta **siempre**. Esto hace que el tiempo de Ejecución sea siempre **cuadrático**

0.3 Tiempos de ejecución

(Agregar el principio de invariancia)

- el peor escenario ← utilizado en aplicaciones críticas
- el mejor escenario ← utilizado nunca jaja
- el escenario promedio ← utilizado cuando hay múltiples ejecuciones en distintas instancias

Para el escenario promedio, en un algoritmo que requieren n elementos, se deben tomar en consideración $n!$ combinaciones posibles.

El mal viaje de esta medición es que, en casos dónde buscamos ordenar una lista, si está parcialmente ordenada no te sirve cómo vara para medir que pasa, pues toma el promedio de las $n!$ combinaciones posibles, y acá no hay ningúna distribución normal, sino que, justamente, puede haber una mayoría de listas parcialmente ordenadas. Bueno, para más data repasar media-mediana-moda y campana de Gauss.

0.3.1 Operaciones elementales

- suma
- resta
- multiplicación
- división

- resto
- comparación
- asignación

En general, se considera con un costo unitario a la suma, resta, la multiplicación, división, módulo, comparaciones y asignaciones. Igual esto depende de qué estemos haciendo, el sistema en el que estamos trabajando, el lenguaje, etc, pero en teoría valen 1, y dsp se usa la razón para determinar si hay alguna complejidad oculta

0.3.2 Orden de operaciones.

Por el principio de invariancia, sabemos que dos implementaciones de un mismo algoritmos difieren en tiempo de ejecución por una constante $t(n) \leq c \cdot t(n)$. Bajo el mismo concepto, podemos determinar el orden de algoritmo, en relación con una funcion $t : \mathbb{N} \to \mathbb{R}^{\geq 0}$

Por ejemplo, tenemos un algoritmo con $t(n) = 24n^2 + 6n + \frac{1}{2}$. Sabemos que, para un n_0 lo suficientemente grande, existe un $n \geq n_0$ tal que

$$\begin{aligned} t(n) &= 24n^2 + 6n + \frac{1}{2} \\ &\leq 24n^2 + 6n^2 + \frac{1}{2}n^2 \\ &\leq \frac{61}{2}n^2 \end{aligned}$$

luego, nuestro algoritmo pertenece al orden de n^2, por el principio de invariancia, pues $24n^2 + 6n + \frac{1}{2} \leq c \cdot n^2$, en particular $c = \frac{61}{2}$, pero este número puede cambiar si elegimos un n_0 distinto.

0.3.3 Big $O(f(n))$

(clarificar la *treshhold rule* ??)

Decimos que Big $O(f(n))$ es el conjunto de todas las funciones que cumplen $t : \mathbb{N} \to \frac{\mathbb{R}^{\geq 0}}{t(n)} \leq c \cdot t(n)$ para cualquier implementación del algoritmo.

$$O(f(n)) = \{t : \mathbb{N} \to \mathbb{R}^{\geq 0} |\ (\exists c \in \mathbb{R}^{+})(\forall n \in \mathbb{R})[t(n) \leq c \cdot t(n)]\}$$

0.3.3.1 Prove that $t(n) \in O(t(n))$

Completar, pag 102. Creo que no es muy necesario 

0.3.3.2 Maximum Rule

Sean $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$, entonces $O(f(n) + g(n)) = \max\{f(n), g(n)\}$.

Además, sean $p, q : \mathbb{N} \to \mathbb{R}^{\geq 0}\ |\ p(n) = f(n) + g(n) \land q(n) = \max\{f(n), g(n)\}$. Entonces, si $t(n) \in O(p(n)) \Rightarrow t(n) \in O(q(n))$

Ejemplos

$$O(n^3 + n^2 + n) = O(n^3) \tag{1}$$

$$O(n^2 - n^2 + n) \neq O(n^2) \tag{2}$$

$$\begin{aligned} O(n^3 \log(n) + n^3 - n^2 + 3) &= O(\max\{n^3 \log(n), (n^3 - n^2), 3\}) \\ &= O(n^3 \log(n)) \end{aligned} \tag{3}$$

Notemos que (2) no funca, porque no cumple la definición de la regla máxima, pues $f(n) = -n^2 \notin f : \mathbb{N} \to \mathbb{R}^{\geq 0}$

En (3) vemos un "workaround" para trabajar esta situación, de forma tal que podamos agrupar todas nuestras funciones en algo de la forma $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$