Materia I1 Estructuras de Datos

Clase 2 / Lunes 11 de marzo:

Un algoritmo se dice correcto si cumple:
- Termina en una cantidad finita de pasos
- Cumple su propósito

Nos interesa estudiar la eficiencia de los algoritmos. Esto le llamaremos la ***complejidad***[1] del algoritmo, es decir, cuántos pasos ejecuta.

- *O(1):* La complejidad del algoritmo es constante, no depende del tamaño del input
- *O($log n$):*
- *O($n log n$):*
- *O(n^2):*
- *O(n^3)*: Es la última complejidad que nos sirve, mas allá de esto es porque probablemente fue mal diseñado y existe algo más eficiente.
- …
- …
- *O(2^n)*: Son problemas intratables, demoran mucho en finalizar.

Cuando la complejidad se reduce con una proporción constante: $n + n/2 + n/4$, entonces generalmente es *O(n)*, mientras que si se reduce en una constante fija:$(n + (n - k) + (n - 2k) \ldots = n(n + 1)/2)$ para cada iteración, va a ser O(n^2).

También nos interesa calcular la cantidad de memoria que usa el algoritmo. Generalmente usamos algunas variables, pero si estas no crecen con el número de datos, entonces no importa.

<u>Algoritmos:</u>

- **Selection Sort:** *O(n^2).* La complejidad del *best-case* es la misma. En cada paso selecciono el número más chico de todos los que me faltan.

Para demostrar que es correcto y que termina podemos hacerlo por inducción. Es un algoritmo *in place*[2]*,* se puede hacer en un sólo arreglo.

1. Defino secuencia ordenada, B, inicialmente vacía.
2. Buscar el menor dato *x* en A.
3. Sacar *x* de A e insertarlo al final de B
4. Si quedan elementos en A, volver a 2.

---

[1] Generalmente se cuenta el peor caso posible

[2] No necesita memoria adicional

```
def selectionSort(arr):
      for i in range(len(A)):
          # Find the minimum element in remaining unsorted array
          min_idx = i
          for j in range(i+1, len(A)):
              if A[min_idx] > A[j]:
                  min_idx = j

          # Swap the found minimum element with the first element
          A[i], A[min_idx] = A[min_idx], A[i]
```

- **Insertion Sort:** *O(n^2).* Insertar un elemento en una lista previamente ordenada. Cuando los elementos a ordenar están ya "casi" ordenados es muy eficiente *O(n),* lo que es útil en las últimas etapas de ordenación. No se necesita una lista auxiliar, ya que se puede hacer en la misma lista agregando un elemento al principio y desplazando los datos.

  1. Defino secuencia ordenada, B, inicialmente vacía.
  2. Tomar primer dato *x de* A y sacarlo de A.
  3. Insertar *x* en B, tal que quede ordenado
  4. Si quedan elementos en A, volver a 2.

```
def insertionSort(arr):
    for i in range(len(arr)):
        key = arr[i]
        # Move elements of arr[0..i-1], that are greater than key,
        # to one position ahead of their current position
        j = i-1
        # the first items are the new sorted list
        while j >=0 and key < arr[j] :
                # desplazo el lado izquierdo
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key
```

Podríamos implementar este algoritmo en una lista ligada, de esta manera nos evitamos desplazar los datos.

	Selection	Insertion	Quick
Peor caso	$O(n^2)$	$O(n^2)$	$O(n^2)$
Mejor caso	$O(n^2)$	$O(n)$	$O(n \log n)$
Promedio	$O(n^2)$	$O(n^2)$	$O(n \log n)$
Memoria	in place	in place	in place*

Clase 3 / Miércoles 13 de marzo:

Conjunto de coordenadas: Coordenadas 2D de puntos en un plano. Queremos hacer una partición de manera que quede la misma cantidad de datos en cada una de las particiones. Para esto tenemos que encontrar la mediana.

Idea 1: Escoger un pivote cualquiera. Separar los datos en menor y mayor con respecto al pivote.

- Misma cantidad de elementos: Encontramos la mediana
- Un grupo queda más grande: Se encuentra la mediana de este grupo y usamos ese como pivote. Repetimos el proceso.

Partition(A, i, f): *O(n),*

En caso promedio toma *O(logn),* se comporta muy bien en la mayoría de los casos.

```
partition(A, i, f):
    p ← un pivote aleatorio en A[i, f]
    m, M ← listas vacias
    for x ∈ A[i, f]:
        if x < p, insertar x en m
        else, insertar x en M
    A[i, f] ← concatenar m con M
    return i + |m|
```

Clase 4 / Lunes 18 de marzo:

Median: El algoritmo es *O(n)* hasta el primer while. Mejor caso *O(n)* y peor caso *O(n^2).* La idea de este algoritmo es que se puede generalizar para encontrar cualquier posición, sólo basta cambiar el *n / 2.*

```
   i(0)                f(n-1)
A [                    ]

0          x           n-1
[          ]  [        ]
   m           M
```

Quickselect: Como se dijo antes, lo que hace es separar un arreglo en un porcentaje deseado. Por ej: si se quiere la mediana, entonces j = n / 2 (no estoy seguro del j…)

```
median(A):
    i ← 0, f ← n − 1
    x ← partition(A, i, f)
    while x ≠ n/2:
        if x < n/2, i ← x + 1
        else, f ← x − 1
        x ← partition(A, i, f)
    return A[x]
```

```
int quickSelect(a, p, r, j):
    if p == r:
        return a[p]
    q = partition(a, p, r)
    k = q-p+1
    if j == k:
        return a[q]
    else:
        if j < k:
            return quickSelect(a, p, q-1, j)
        else:
            return quickSelect(a, q+1, r, j-k)
```

Quicksort: El caso promedio es *O(nlogn),* y el peor caso es *O(n^2).* Las operaciones que realiza son muy eficientes, por lo tanto en términos de tiempo anda muy bien.

```
quicksort(A, i, f):
    if i ≤ f:
        p ← partition(A, i, f)
        quicksort(A, i, p − 1)
        quicksort(A, p + 1, f)
```

El mejor caso del algoritmo es cuando cada partición divide al arreglo en mitades. El peor es cuando el arreglo ya está ordenado, entonces hará particiones de tamaños n-1 y 0.

Posibles mejoras:
- Usar InsertionSort para subarreglos con menos de 20 elementos.
- Usar la mediana de tres elementos como pivote
- Si hay cantidades grandes de claves duplicadas, como fechas, es probable que se particiones subarreglos con claves iguales. Para esto podemos hacer 3 particiones: menores, iguales y mayores al pivote.

Este algoritmo se puede modificar para que funcione con un sólo arreglo, de esta manera podemos ahorrar memoria: El pivote se pone como última posición. Se busca de izquierda a derecha un número que sea mayor que el pivote. Cuando lo encuentro empiezo a buscar de derecha a izquierda (penúltimo dato) uno que sea menor que este, luego se intercambian y se vuelve a realizar. Esta modificación no es estrictamente *in place*, ya que como usa llamados recursivos, necesita memoria para guardar donde tiene que volver de cada llamado.

En la foto tenemos la versión mejorada de *partition* para que funcione *in place.*

```
partition(A, i, f):
    x ← un indice aleatorio en [i, f],    p ← A[x]
    A[x] ⇄ A[f]
    j ← i
    for k ∈ [i, f − 1]:
        if A[k] < p:
            A[j] ⇄ A[k]
            j ← j + 1
    A[j] ⇄ A[f]
    return j
```