Algoritmos y Estructuras de Datos

Ordenamiento

Introducción

Quicksort

Cota inferior para el tiempo de ordenamiento

Bucket y radix sort

Repaso: métodos de ordenamiento que ya vimos

Selection sort

1. Seleccionar el elemento mínimo y ubicarlo al principio.
2. Continuar ordenando el resto de la lista.

```
def selection_sort(a):
    n = len(a)
    for i in range(n):
        # Ubicar el mínimo de a[i : n] en a[i].
        for j in range(i + 1, n):
            if a[i] > a[j]:
                a[i], a[j] = a[j], a[i]
```

Complejidad temporal en peor caso: $O(n^2)$.

Repaso: métodos de ordenamiento que ya vimos

Mergesort

1. Dividir la lista en mitades.
2. Ordenar cada mitad.
3. Mezclar las dos mitades ordenadas.

```
def mergesort(a):
    if len(a) <= 1:
        return a
    m = len(a) // 2
    return merge(mergesort(a[: m]), mergesort(a[m :]))
```

Complejidad temporal en peor caso: $O(n \log(n))$.

Algunas propiedades de los algoritmos

Algoritmos *in-place*

Un algoritmo *in-place* trabaja sobre el espacio de memoria de la entrada, usando **menos** de $O(n)$ memoria extra. Por ejemplo:

1. La variante de mergesort que vimos no es in-place.
2. La variante de selection sort que vimos es in-place.

Algoritmos de ordenamiento estables

Si dos elementos "empatan", un algoritmo estable los mantiene en el orden en el que estaban originalmente. Por ejemplo:

$$[3\heartsuit, 1\clubsuit, 2\clubsuit, 2\diamondsuit, 1\heartsuit, 3\clubsuit, 1\spadesuit] \rightsquigarrow [1\clubsuit, 1\heartsuit, 1\spadesuit, 2\clubsuit, 2\diamondsuit, 3\heartsuit, 3\clubsuit]$$

Generalmente los algoritmos de ordenamiento se pueden hacer estables si se implementan con cuidado.

Quicksort

Para ordenar A usando $\textsc{Quicksort}$:

1. Si $|A| \leq 1$, ya está ordenada. Terminar.
2. Elegir un elemento p de A, al que llamamos “pivote”.
   Por ejemplo, se puede tomar $p := A[0]$.
3. Particionar los elementos de A del siguiente modo:

$$\underbrace{\boxed{\text{elementos} \leq p}}_{B}\boxed{p}\underbrace{\boxed{\text{elementos} > p}}_{C}$$

4. $B := \textsc{Quicksort}(B)$
5. $C := \textsc{Quicksort}(C)$

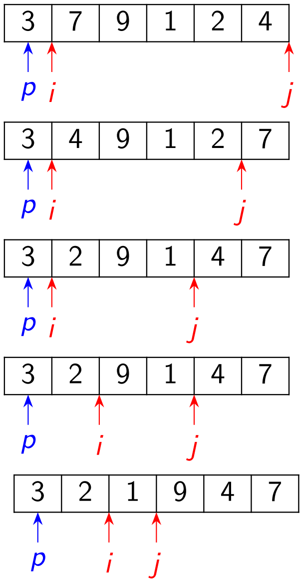
Quicksort en Python

```python
def quicksort(a, desde, hasta):
    if hasta - desde <= 1:
        return a
    p = a[desde]
    i = desde + 1
    j = hasta
    while i < j:
        if a[i] < p:
            i = i + 1
        else:
            a[i], a[j - 1] = a[j - 1], a[i]
            j = j - 1
    a[desde], a[i - 1] = a[i - 1], a[desde]
    quicksort(a, desde, i - 1)
    quicksort(a, i, hasta)
    return a
```

¿Cuál es el invariante del particionamiento?

El costo del particionamiento es $O(n)$, donde $n =$ `desde - hasta`.

Quicksort – Ejemplo de particionamiento



Complejidad de Quicksort

$$T(n) \leq \begin{cases} \mathbf{a} & \text{si } n \leq 1 \\ \mathbf{b}n + T(k_1) + T(k_2) \end{cases}$$

donde $n = 1 + k_1 + k_2$:

- k_1 es el número de elementos $\leq p$ (exceptuando a p),
- k_2 es el número de elementos $> p$.

En el **peor caso**, la complejidad es $O(n^2)$.
El peor caso se da con entradas como $[0, 1, 2, \ldots, n-1]$.

En el **mejor caso**, la complejidad es $O(n \log(n))$.
El mejor caso se da cuando el pivote es exactamente la mediana.

Si asumimos que la entrada está distribuida uniformemente, se puede ver que la complejidad en **caso promedio** es $O(n \log(n))$.

Para evitar que un adversario pueda forzar el peor caso, se puede elegir el pivote de manera aleatoria.

Complejidad de Quicksort en caso promedio

Sea $A = [a_0, \ldots, a_{n-1}]$.

Teorema

$\textsc{Quicksort}(A)$ hace $O(n \log n)$ comparaciones en caso promedio.

Demostración.

- Notamos X_{ij} a la siguiente variable aleatoria:

$$X_{ij} = \begin{cases} 1 & \text{si } a_i \text{ se compara con } a_j \text{ al hacer } \textsc{Quicksort}(A) \\ 0 & \text{si no} \end{cases}$$

- El total de comparaciones que hace $\textsc{Quicksort}(A)$ es:

$$X = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}$$

- La complejidad temporal en caso promedio está dada por:

$$E[X] = E[\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}] = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} E[X_{ij}] \quad \text{(linealidad de } E\text{)}.$$

Complejidad de QUICKSORT en caso promedio

- $E[X_{ij}]$ es la probabilidad de que a_i se compare con a_j.
- Dicha probabilidad corresponde a la probabilidad de que entre los elementos $a_i, a_{i+1}, a_{i+2}, \ldots, a_{j-1}, a_j$ se elija a_i o a_j como pivote antes que a cualquiera de los restantes elementos $a_{i+1}, a_{i+2}, \ldots, a_{j-1}$.
- Es decir:

$$E[X_{ij}] = \frac{2}{j-i+1}$$

- Por lo tanto:

$$\begin{aligned} E[X] &= \textstyle\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1} \\ &= \textstyle\sum_{i=0}^{n-1} \sum_{k=1}^{n-i-1} \frac{2}{k+1} \qquad \text{(cambio de índices: } k = j - i\text{)} \\ &< \textstyle\sum_{i=0}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\ &= \textstyle\sum_{i=0}^{n-1} O(\log(n)) \qquad = O(n \log(n)) \qquad \square \end{aligned}$$

Introducción

Quicksort

Cota inferior para el tiempo de ordenamiento

Bucket y radix sort

Cota inferior para el tiempo de ordenamiento

Conocemos algoritmos para ordenar listas en:

1. $O(n^2)$ — selection sort, quicksort (peor caso)
2. $O(n \log(n))$ — mergesort, quicksort (caso promedio)

¿Se podrá ordenar en tiempo menor a $O(n \log(n))$?

Depende de qué **operaciones** podamos hacer con los datos.

- Por ejemplo, si sabemos que los datos son dígitos entre 0 y 9: **es posible** ordenarlos en tiempo $O(n)$.
- Si sólo podemos **comparar** (es decir, determinar si $x < y$): **no es posible** ordenarlos en menos de $O(n \log(n))$.

Cota inferior para el tiempo de ordenamiento

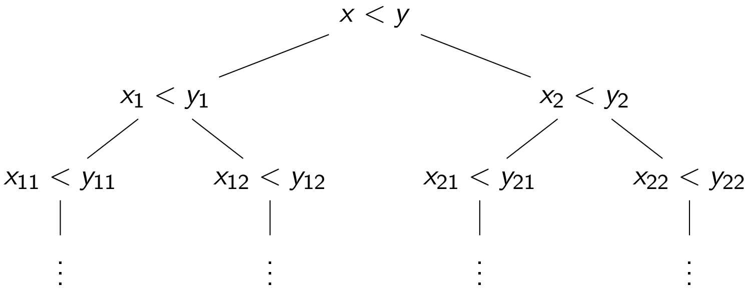
Sea $\mathcal{S}$ un algoritmo de ordenamiento basado en comparaciones.
Sea A un arreglo de entrada de tamaño n.

Teorema

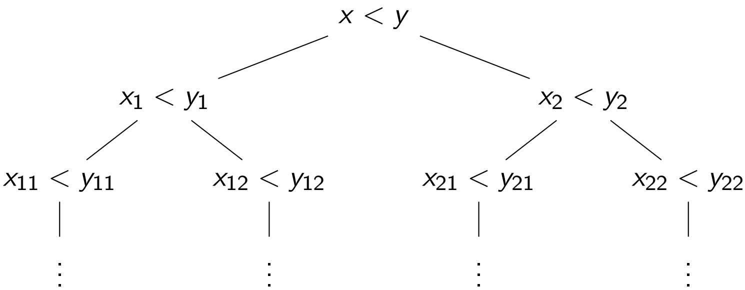
$\mathcal{S}(A)$ hace $\Omega(n \log n)$ comparaciones en peor caso.

Demostración.

- Sea $A = [a_0, \ldots, a_{n-1}]$ el arreglo de entrada.
- La ejecución de $\mathcal{S}(A)$ corresponde a un **árbol de decisión**:



Cota inferior para el tiempo de ordenamiento



- Cada nodo del árbol es una comparación.
- Las hojas son los posibles resultados del ordenamiento.
- El peor caso está dado por la altura del árbol.
- El número total de hojas debe ser **al menos** $n!$.
- Luego la altura del árbol debe ser $h \geq \log_2(n!) + 1$.

Cota inferior para el tiempo de ordenamiento

Es decir, el número de comparaciones en peor caso es:

$$h \geq \log_2(n!) + 1$$

Para concluir, observemos que:

$$\begin{aligned}
\log_2(n!) &= \log_2\left(\prod_{i=1}^{n} i\right) \\
&= \sum_{i=1}^{n} \log_2(i) \\
&\geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \log_2(i) \\
&\geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \log_2\left(\frac{n}{2}\right) \\
&= \left\lceil \frac{n}{2} \right\rceil \log_2\left(\frac{n}{2}\right) \\
&= \left\lceil \frac{n}{2} \right\rceil (\log_2(n) - 1) \quad = \quad \Omega(n \log(n)) \quad \square
\end{aligned}$$

Bucket sort

Entrada: un arreglo A de n enteros en el rango $\{0, \dots, k-1\}$.
Salida: una permutación ordenada de A.

- Construir un arreglo B de tamaño k.
  Inicializar $B[i] = 0$ para todo $0 \leq i < k$.
- Para cada elemento x de A:
  - Incrementar $B[x]$ en uno.

  En este punto, $B[x]$ es el número de apariciones de x en A.
- Para cada x desde 0 hasta $k-1$:
  - Generar en la salida x tantas veces como indique $B[x]$.

La complejidad temporal es $O(n+k)$ en peor caso.

Bucket sort en Python

```python
def bucket_sort(a, k):
    # Contar
    b = [0 for i in range(k)]
    for x in a:
        b[x] = b[x] + 1
    # Generar la salida
    j = 0
    for x in range(k):
        for i in range(b[x]):
            a[j] = x
            j = j + 1
    return a
```

Radix sort

Sean $X_1, \ldots, X_k$ conjuntos.
Sea $Y = X_1 \times \ldots \times X_k$ el conjunto de las k-uplas $(x_1, \ldots, x_k)$.
El conjunto Y se puede ordenar con el **orden lexicográfico**.

Entrada: un arreglo A de tuplas $(x_1, \ldots, x_k)$.
Salida: una permutación ordenada de A.

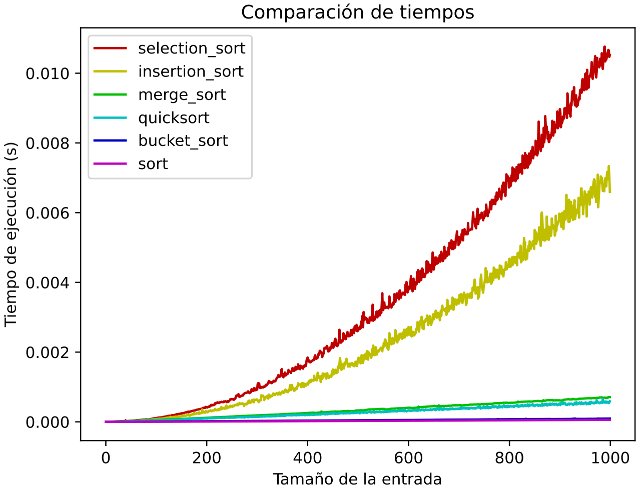
- Para cada i desde k hasta 1:
  - Ordenar el arreglo A de acuerdo con la i-ésima componente, usando un algoritmo de ordenamiento estable.

Ejemplo

Ordenemos las palabras:

`ora, aro, oro, ala, ola, era, ojo`

Evaluación empírica



Evaluación empírica

Una evaluación empírica "seria" debería:

- Hacer varias mediciones y tomar el promedio o mediana.
- Medir la varianza.
- Controlar el entorno de ejecución.
  (Caché, *garbage collection*, *context switching*, ...).
- La distribución de la entrada debería adecuarse al problema.
  (Entradas más grandes, distribución normal, ...).

Problema

Hay n elefantes y m telarañas. Los pesos de los elefantes están dados por un arreglo $[E[0], \ldots, E[n-1]]$. Cada telaraña tiene una capacidad máxima de peso que resiste. Las capacidades de las telarañas están dadas por un arreglo $[T[0], \ldots, T[m-1]]$. Queremos saber si es posible ubicar cada elefante sobre una telaraña, con las siguientes restricciones:

- El peso de cada elefante no debe superar la capacidad de la telaraña sobre la que está ubicado.
- Cada elefante se debe ubicar sobre una telaraña distinta. Es decir, no puede haber un mismo elefante sobre dos telarañas, ni dos elefantes sobre una misma telaraña.
- Todo elefante debe estar ubicado sobre alguna telaraña (no pueden faltar telarañas) pero puede haber telarañas sin elefantes (pueden sobrar telarañas).

Diseñar un algoritmo que dados los arreglos E y T determine si es posible ubicar a los elefantes sobre las telarañas con las restricciones descriptas arriba. La complejidad debe ser $O(m \log m)$.