Ordenamiento

Miguel Raggi

Algoritmos
ENES
UNAM

5 de marzo de 2019

Índice:

Índice:

Bubblesort (Ordenamiento Burbuja)

- Es un ordenamiento muy simple: Básicamente es fuerza bruta.

Bubblesort (Ordenamiento Burbuja)

- Es un ordenamiento muy simple: Básicamente es fuerza bruta.
- Empezamos a comparar los primeros dos. Si están mal, los intercambiamos. Si no, continuamos con el primero y tercero, luego etc. de dos en dos.

Bubblesort (Ordenamiento Burbuja)

- Es un ordenamiento muy simple: Básicamente es fuerza bruta.
- Empezamos a comparar los primeros dos. Si están mal, los intercambiamos. Si no, continuamos con el primero y tercero, luego etc. de dos en dos.
- Cada que intercambiamos, volvemos a empezar.

Bubblesort (Ordenamiento Burbuja)

- Es un ordenamiento muy simple: Básicamente es fuerza bruta.
- Empezamos a comparar los primeros dos. Si están mal, los intercambiamos. Si no, continuamos con el primero y tercero, luego etc. de dos en dos.
- Cada que intercambiamos, volvemos a empezar.
- Cuando recorramos la lista completa, ya terminamos.

Bubblesort: Análisis

- ¿Cuántas operaciones tendré que hacer?

Bubblesort: Análisis

- ¿Cuántas operaciones tendré que hacer?
- En el peor de los casos, tendré que comparar todos contra todos?

Bubblesort: Análisis

- ¿Cuántas operaciones tendré que hacer?
- En el peor de los casos, tendré que comparar todos contra todos?
- ¿Cuántas operaciones son, si tengo n elementos?

Bubblesort: Análisis

- ¿Cuántas operaciones tendré que hacer?
- En el peor de los casos, tendré que comparar todos contra todos?
- ¿Cuántas operaciones son, si tengo n elementos?
- $\binom{n}{2} = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \in O(n^2)$

Pseudo-Código

```
1  def Bubblesort(A):
2    cambio_esta_ronda = True
3    while (cambio_esta_ronda):
4      cambio_esta_ronda = False
5      for i in range(1,len(A)):
6        if A[i] < A[i-1]:
7          swap(A[i],A[i-1])
8          cambio_esta_ronda = True
9
```

Ejercicio: ¿Cómo funcionaría este algoritmo en la lista $5, 2, 3, 6, 6, 1$?

Ejercicio: Implementar en C++.

Selectionsort

- Ya vimos un algoritmo para encontrar el menor elemento.

Selectionsort

- Ya vimos un algoritmo para encontrar el menor elemento.
- Podemos entonces encontrar el menor, luego el menor de lo que queda, luego el menor de lo que queda, etc.

Selectionsort

- Ya vimos un algoritmo para encontrar el menor elemento.
- Podemos entonces encontrar el menor, luego el menor de lo que queda, luego el menor de lo que queda, etc.
- Análisis: Hay que comparar todos los elementos contra todos, así que es de orden $O(n^2)$.

Selectionsort

- Ya vimos un algoritmo para encontrar el menor elemento.
- Podemos entonces encontrar el menor, luego el menor de lo que queda, luego el menor de lo que queda, etc.
- Análisis: Hay que comparar todos los elementos contra todos, así que es de orden $O(n^2)$.
- Es lento, pero tiene la ventaja de que es fácil de programar y no utiliza casi memoria adicional.

Pseudo-Código

```
1  def SelectionSort(A):
2    for i in range(len(A)):
3      minHastaAhora = i
4      for j in range(i+1,len(A)):
5        if A[j] < A[minHastaAhora]:
6          minHastaAhora = j
7      swap(A[minHastaAhora],A[i])
```

Ejercicio: ¿Cómo funcionaría este algoritmo en la lista $5, 2, 3, 6, 6, 1$?

Insertionsort

- Es el algoritmo que usarías si te van dando la lista de uno por uno.

Insertionsort

- Es el algoritmo que usarías si te van dando la lista de uno por uno.
- Es decir, siempre mantienes ordenada los primeros y vas metiendo en su lugar de uno por uno.

Insertionsort

- Es el algoritmo que usarías si te van dando la lista de uno por uno.
- Es decir, siempre mantienes ordenada los primeros y vas metiendo en su lugar de uno por uno.
- En el peor caso, hay que comparar todos contra todos, así que es de orden $O(n^2)$.

Insertionsort

- Es el algoritmo que usarías si te van dando la lista de uno por uno.
- Es decir, siempre mantienes ordenada los primeros y vas metiendo en su lugar de uno por uno.
- En el peor caso, hay que comparar todos contra todos, así que es de orden $O(n^2)$.
- También el promedio es de ese orden.

Insertionsort

- Es el algoritmo que usarías si te van dando la lista de uno por uno.
- Es decir, siempre mantienes ordenada los primeros y vas metiendo en su lugar de uno por uno.
- En el peor caso, hay que comparar todos contra todos, así que es de orden $O(n^2)$.
- También el promedio es de ese orden.
- Tampoco utiliza memoria adicional (casi).

Insertionsort

- Es el algoritmo que usarías si te van dando la lista de uno por uno.
- Es decir, siempre mantienes ordenada los primeros y vas metiendo en su lugar de uno por uno.
- En el peor caso, hay que comparar todos contra todos, así que es de orden $O(n^2)$.
- También el promedio es de ese orden.
- Tampoco utiliza memoria adicional (casi).
- Interesantemente, cuando la lista es muy pequeña (digamos, 64 elemenots), es de los algoritmos más rápidos por como están hechas las computadoras.

Pseudo-Código

```
1 def Insertionsort(A)
2   for i in range(1,len(A)):
3     j = i
4     while j-1 >= 0 and A[j] < A[j-1]:
5       swap(A[j],A[j-1])
6       j -= 1
7
```

- Ejercicio: ¿Cómo correría el algoritmo en la lista $1, 6, 2, 4, 5, 5, 2$?

Pseudo-Código

```
1 def Insertionsort(A)
2   for i in range(1,len(A)):
3     j = i
4     while j-1 >= 0 and A[j] < A[j-1]:
5       swap(A[j],A[j-1])
6       j -= 1
7
```

- Ejercicio: ¿Cómo correría el algoritmo en la lista $1, 6, 2, 4, 5, 5, 2$?
- Pregunta: ¿Por qué no utilizar búsqueda binaria?
- Respuesta: Si podemos, pero de todos modos tendremos que recorrer todos un lugar a la derecha, así que no vale la pena.

Mergesort (Ordenamiento por Mezcla)

- Es del tipo de algoritmos conocidos como divide y vencerás.

Mergesort (Ordenamiento por Mezcla)

- Es del tipo de algoritmos conocidos como divide y vencerás.
- Divide: Primero dividimos la lista en dos.

Mergesort (Ordenamiento por Mezcla)

- Es del tipo de algoritmos conocidos como divide y vencerás.
- Divide: Primero dividimos la lista en dos.
- Conquista: Después ordenamos cada parte de la lista más pequeña (otra vez, dividiendo en dos, etc.)

Mergesort (Ordenamiento por Mezcla)

- Es del tipo de algoritmos conocidos como divide y vencerás.
- Divide: Primero dividimos la lista en dos.
- Conquista: Después ordenamos cada parte de la lista más pequeña (otra vez, dividiendo en dos, etc.)
- Combina: Después juntamos ambas listas, teniendo en cuenta que el menor siempre será el menor de una o el menor de la otra.

Mergesort (Ordenamiento por Mezcla)

- Es del tipo de algoritmos conocidos como divide y vencerás.
- Divide: Primero dividimos la lista en dos.
- Conquista: Después ordenamos cada parte de la lista más pequeña (otra vez, dividiendo en dos, etc.)
- Combina: Después juntamos ambas listas, teniendo en cuenta que el menor siempre será el menor de una o el menor de la otra.
- La ventaja de este algoritmo es que es de orden $O(n \log(n))$

Mergesort (Ordenamiento por Mezcla)

- Es del tipo de algoritmos conocidos como divide y vencerás.
- Divide: Primero dividimos la lista en dos.
- Conquista: Después ordenamos cada parte de la lista más pequeña (otra vez, dividiendo en dos, etc.)
- Combina: Después juntamos ambas listas, teniendo en cuenta que el menor siempre será el menor de una o el menor de la otra.
- La ventaja de este algoritmo es que es de orden $O(n \log(n))$
- Veremos el análisis de estos algoritmos en el siguiente "capítulo".

Pseudo-Código

Primero veremos el código de Mergesort, que es muy sencillo, si suponemos que tenemos la función Merge.

```
1 def Mergesort(A,start=0,end=len(A)):
2   if start < end:
3     mid = [(start+end)/2]
4     Mergesort(A,start,mid)
5     Mergesort(A,mid,end)
6     A[start:end] = Merge(A[start:mid],A[mid:end])
```

La función merge

Ahora, la función merge debe tomar dos intervalos ya ordenados y producir uno nuevo completamente ordenado.

```
1  def Merge(A,B):
2    R = []
3    i,j=0,0
4    a,b=len(A),len(B)
5    A.append(infinito)
6    B.append(infinito)
7    while i < a or j < b:
8      if A[i] < B[i]:
9        R.append(A[i])
10       i += 1
11     else:
12       R.append(B[j])
13       j += 1
14   return R
```

La función merge

Ahora, la función merge debe tomar dos intervalos ya ordenados y producir uno nuevo completamente ordenado.

```
def Merge(A,B):
  R = []
  i,j=0,0
  a,b=len(A),len(B)
  A.append(infinito)
  B.append(infinito)
  while i < a or j < b:
    if A[i] < B[i]:
      R.append(A[i])
      i += 1
    else:
      R.append(B[j])
      j += 1
  return R
```

Ejer: Utiliza mergesort para ordenar la lista $5, 12, 4, 3, 3, 4, 6, 7, 1, 1$

Quicksort (Ordenamiento Rápido)

- Parecido a mergesort, también del tipo Divide y Vencerás.

Quicksort (Ordenamiento Rápido)

- Parecido a mergesort, también del tipo Divide y Vencerás.
- Divide: Tomamos un elemento al azar. Luego tomamos todos los menores que el elemento por un lado, y todos los mayores por el otro.

Quicksort (Ordenamiento Rápido)

- Parecido a mergesort, también del tipo Divide y Vencerás.
- Divide: Tomamos un elemento al azar. Luego tomamos todos los menores que el elemento por un lado, y todos los mayores por el otro.
- Conquista: Ordenamos ambas listas.

Quicksort (Ordenamiento Rápido)

- Parecido a mergesort, también del tipo Divide y Vencerás.
- Divide: Tomamos un elemento al azar. Luego tomamos todos los menores que el elemento por un lado, y todos los mayores por el otro.
- Conquista: Ordenamos ambas listas.
- Combina: ¡Nada!

Quicksort (Ordenamiento Rápido)

- Parecido a mergesort, también del tipo Divide y Vencerás.
- Divide: Tomamos un elemento al azar. Luego tomamos todos los menores que el elemento por un lado, y todos los mayores por el otro.
- Conquista: Ordenamos ambas listas.
- Combina: ¡Nada!
- En la práctica, este algoritmo es muy bueno y muy utilizado, pues las constantes son muy pequeñas.

Quicksort (Ordenamiento Rápido)

- Parecido a mergesort, también del tipo Divide y Vencerás.
- Divide: Tomamos un elemento al azar. Luego tomamos todos los menores que el elemento por un lado, y todos los mayores por el otro.
- Conquista: Ordenamos ambas listas.
- Combina: ¡Nada!
- En la práctica, este algoritmo es muy bueno y muy utilizado, pues las constantes son muy pequeñas.
- Tiene orden de peor caso $O(n^2)$, pero en promedio tarda $O(n \log(n))$.

Quicksort (Ordenamiento Rápido)

- Parecido a mergesort, también del tipo Divide y Vencerás.
- Divide: Tomamos un elemento al azar. Luego tomamos todos los menores que el elemento por un lado, y todos los mayores por el otro.
- Conquista: Ordenamos ambas listas.
- Combina: ¡Nada!
- En la práctica, este algoritmo es muy bueno y muy utilizado, pues las constantes son muy pequeñas.
- Tiene orden de peor caso $O(n^2)$, pero en promedio tarda $O(n \log(n))$.
- Además, al igual que mergesort, es naturalmente paralelizable.

Pseudo-Código

```
1 def quicksort(A,p = 0,r = len(A)):
2   if p < r:
3     q = Parte(A,p,r)
4     quicksort(A,p,q)
5     quicksort(A,q,r)
```

Quicksort: Partir

```
1  Partir(A,p,r)
2    x = A[r-1]
3    i = p-1
4    for j in range(p,r-1)
5      if A[j] <= x:
6        i += 1
7        swap(A[j],A[i])
8    swap(A[i+1], A[r-1])
9    return i+1
```

Índice:

¿Podemos hacerlo mejor?

¿Podemos hacerlo mejor?

- Ahora, tenemos varios algoritmos para ordenar una lista. Los mejores son hasta ahora los que corren en tiempo $O(n \log(n))$.

¿Podemos hacerlo mejor?

- Ahora, tenemos varios algoritmos para ordenar una lista. Los mejores son hasta ahora los que corren en tiempo $O(n \log(n))$.
- La pregunta es: ¿Podemos mejorar?

¿Podemos hacerlo mejor?

- Ahora, tenemos varios algoritmos para ordenar una lista. Los mejores son hasta ahora los que corren en tiempo $O(n \log(n))$.
- La pregunta es: ¿Podemos mejorar?
- La respuesta es: ¡depende!

¿Podemos hacerlo mejor?

- Ahora, tenemos varios algoritmos para ordenar una lista. Los mejores son hasta ahora los que corren en tiempo $O(n \log(n))$.
- La pregunta es: ¿Podemos mejorar?
- La respuesta es: ¡depende!
- Obviamente no podemos hacerlo mejor que $O(n)$, pues al menos tenemos que leer la lista. $O(n \log(n))$ no es mucho peor, pero quizás podríamos mejorar.

¿Podemos hacerlo mejor?

- Ahora, tenemos varios algoritmos para ordenar una lista. Los mejores son hasta ahora los que corren en tiempo $O(n \log(n))$.
- La pregunta es: ¿Podemos mejorar?
- La respuesta es: ¡depende!
- Obviamente no podemos hacerlo mejor que $O(n)$, pues al menos tenemos que leer la lista. $O(n \log(n))$ no es mucho peor, pero quizás podríamos mejorar.
- En todos estos algoritmos estamos utilizando la comparación de dos elementos $(a < b)$ para decidir qué hacer.

¿Podemos hacerlo mejor?

- Ahora, tenemos varios algoritmos para ordenar una lista. Los mejores son hasta ahora los que corren en tiempo $O(n \log(n))$.
- La pregunta es: ¿Podemos mejorar?
- La respuesta es: ¡depende!
- Obviamente no podemos hacerlo mejor que $O(n)$, pues al menos tenemos que leer la lista. $O(n \log(n))$ no es mucho peor, pero quizás podríamos mejorar.
- En todos estos algoritmos estamos utilizando la comparación de dos elementos $(a < b)$ para decidir qué hacer.
- No es difícil probar (¡y lo haremos!) que bajo este "modelo", no podemos hacerlo mejor. Si la única información que tenemos acerca de las llaves es que las podemos comparar, entonces no podemos hacerlo mejor.

¿Podemos hacerlo mejor?

- Ahora, tenemos varios algoritmos para ordenar una lista. Los mejores son hasta ahora los que corren en tiempo $O(n \log(n))$.
- La pregunta es: ¿Podemos mejorar?
- La respuesta es: ¡depende!
- Obviamente no podemos hacerlo mejor que $O(n)$, pues al menos tenemos que leer la lista. $O(n \log(n))$ no es mucho peor, pero quizás podríamos mejorar.
- En todos estos algoritmos estamos utilizando la comparación de dos elementos ($a < b$) para decidir qué hacer.
- No es difícil probar (¡y lo haremos!) que bajo este "modelo", no podemos hacerlo mejor. Si la única información que tenemos acerca de las llaves es que las podemos comparar, entonces no podemos hacerlo mejor.
- Sin embargo, si las llaves son algo dado (*i.e.* números enteros), a veces podemos mejorar, y veremos cómo la próxima clase.

¿Por qué no podemos mejorar?

- Vamos a ver que necesitas $\Omega(n \log(n))$ comparaciones.

¿Por qué no podemos mejorar?

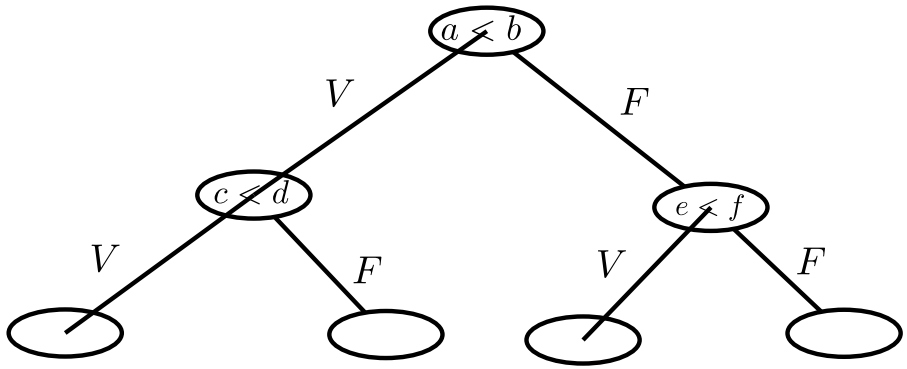
- Vamos a ver que necesitas $\Omega(n \log(n))$ comparaciones.
- Cada comparación, te pueden salir dos resultados: V o F.

¿Por qué no podemos mejorar?

- Vamos a ver que necesitas $\Omega(n \log(n))$ comparaciones.
- Cada comparación, te pueden salir dos resultados: V o F.
- En cada una de esas dos puedes decidir qué hacer.

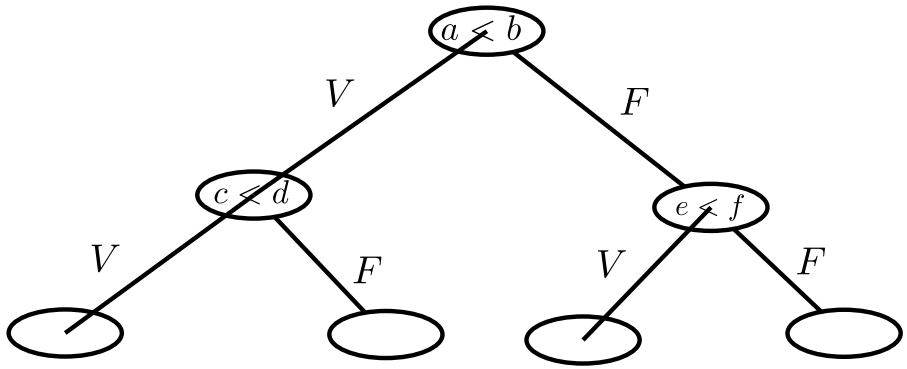
¿Por qué no podemos mejorar?

- Vamos a ver que necesitas $\Omega(n \log(n))$ comparaciones.
- Cada comparación, te pueden salir dos resultados: V o F.
- En cada una de esas dos puedes decidir qué hacer.
- Formamos un "árbol binario" de decisión:



¿Por qué no podemos mejorar?

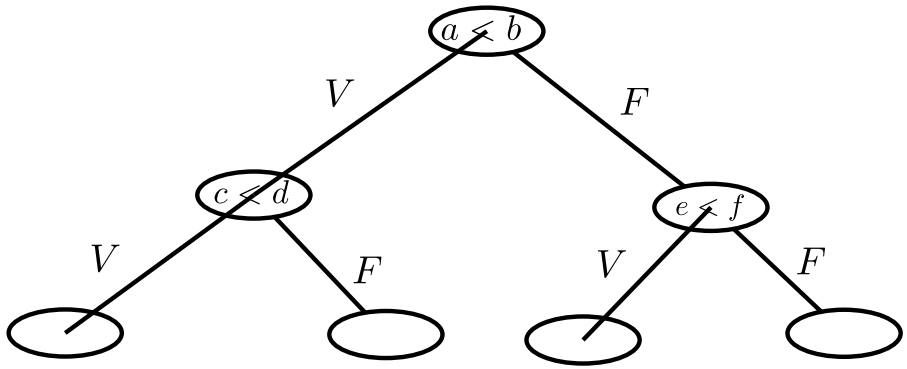
- Vamos a ver que necesitas $\Omega(n \log(n))$ comparaciones.
- Cada comparación, te pueden salir dos resultados: V o F.
- En cada una de esas dos puedes decidir qué hacer.
- Formamos un "árbol binario" de decisión:



- ¿Cuántos posibles órdenes hay de los elementos?

¿Por qué no podemos mejorar?

- Vamos a ver que necesitas $\Omega(n \log(n))$ comparaciones.
- Cada comparación, te pueden salir dos resultados: V o F.
- En cada una de esas dos puedes decidir qué hacer.
- Formamos un "árbol binario" de decisión:



- ¿Cuántos posibles órdenes hay de los elementos? $n!$

¿Por qué no podemos mejorar?

- Cada “nodo final” del árbol (es decir, cada bolita que ya no tiene nada abajo) debe decirte en qué orden está la lista.

¿Por qué no podemos mejorar?

- Cada "nodo final" del árbol (es decir, cada bolita que ya no tiene nada abajo) debe decirte en qué orden está la lista.
- Pero hay $n!$ cosas totales que debe poder decidir.

¿Por qué no podemos mejorar?

- Cada "nodo final" del árbol (es decir, cada bolita que ya no tiene nada abajo) debe decirte en qué orden está la lista.
- Pero hay $n!$ cosas totales que debe poder decidir.
- ¿Cuántos "nodos finales" tiene un árbol donde haces x comparaciones (es decir, con x "niveles")? Pues 2^x!

¿Por qué no podemos mejorar?

- Cada "nodo final" del árbol (es decir, cada bolita que ya no tiene nada abajo) debe decirte en qué orden está la lista.
- Pero hay $n!$ cosas totales que debe poder decidir.
- ¿Cuántos "nodos finales" tiene un árbol donde haces x comparaciones (es decir, con x "niveles")? Pues 2^x!
- Entonces $n! \leq 2^x$. De aquí, $\log(n!) \leq x$.

¿Por qué no podemos mejorar?

- Cada "nodo final" del árbol (es decir, cada bolita que ya no tiene nada abajo) debe decirte en qué orden está la lista.
- Pero hay $n!$ cosas totales que debe poder decidir.
- ¿Cuántos "nodos finales" tiene un árbol donde haces x comparaciones (es decir, con x "niveles")? Pues 2^x!
- Entonces $n! \leq 2^x$. De aquí, $\log(n!) \leq x$.
- Se puede ver (con cálculo integral) que

$$\log(n!) \approx n\log(n) - n + 1 \in \Theta(n\log(n))$$

Índice:

Datos satélite, estabilidad, memoria adicional

Hay 3 cosas de qué preocuparse (bueno, hay más) cuando uno selecciona un algoritmo para ordenar.

Datos satélite, estabilidad, memoria adicional

Hay 3 cosas de qué preocuparse (bueno, hay más) cuando uno selecciona un algoritmo para ordenar.

1. Datos Satélite.
   - Usualmente cuando ordenas una lista (de números, nombres, o lo que sea), también incluyen varios otros datos que no estamos comparando.

Datos satélite, estabilidad, memoria adicional

Hay 3 cosas de qué preocuparse (bueno, hay más) cuando uno selecciona un algoritmo para ordenar.

1. Datos Satélite.
   - Usualmente cuando ordenas una lista (de números, nombres, o lo que sea), también incluyen varios otros datos que no estamos comparando.
   - Por ejemplo, número de cuenta, departamento, teléfono, etc. etc.

Datos satélite, estabilidad, memoria adicional

Hay 3 cosas de qué preocuparse (bueno, hay más) cuando uno selecciona un algoritmo para ordenar.

1. Datos Satélite.
   - Usualmente cuando ordenas una lista (de números, nombres, o lo que sea), también incluyen varios otros datos que no estamos comparando.
   - Por ejemplo, número de cuenta, departamento, teléfono, etc. etc.
   - A esta información adicional Pero no queremos mover tooodo al intercambiar cosas, así que usualmente trabajas con "apuntadores" a los records.

Datos satélite, estabilidad, memoria adicional

Hay 3 cosas de qué preocuparse (bueno, hay más) cuando uno selecciona un algoritmo para ordenar.

1. Datos Satélite.
   - Usualmente cuando ordenas una lista (de números, nombres, o lo que sea), también incluyen varios otros datos que no estamos comparando.
   - Por ejemplo, número de cuenta, departamento, teléfono, etc. etc.
   - A esta información adicional Pero no queremos mover tooodo al intercambiar cosas, así que usualmente trabajas con "apuntadores" a los records.
   - Al dato que utilizamos para comparar le llamamos una key (llave).

Datos satélite, estabilidad, memoria adicional

Hay 3 cosas de qué preocuparse (bueno, hay más) cuando uno selecciona un algoritmo para ordenar.

1. Datos Satélite.
   - Usualmente cuando ordenas una lista (de números, nombres, o lo que sea), también incluyen varios otros datos que no estamos comparando.
   - Por ejemplo, número de cuenta, departamento, teléfono, etc. etc.
   - A esta información adicional Pero no queremos mover tooodo al intercambiar cosas, así que usualmente trabajas con "apuntadores" a los records.
   - Al dato que utilizamos para comparar le llamamos una key (llave).
2. Estabilidad

Datos satélite, estabilidad, memoria adicional

Hay 3 cosas de qué preocuparse (bueno, hay más) cuando uno selecciona un algoritmo para ordenar.

1. Datos Satélite.
   - Usualmente cuando ordenas una lista (de números, nombres, o lo que sea), también incluyen varios otros datos que no estamos comparando.
   - Por ejemplo, número de cuenta, departamento, teléfono, etc. etc.
   - A esta información adicional Pero no queremos mover tooodo al intercambiar cosas, así que usualmente trabajas con "apuntadores" a los records.
   - Al dato que utilizamos para comparar le llamamos una key (llave).
2. Estabilidad
   - Dado que podemos traer datos satélite, muchas veces ocurre que aunque los datos, al compararlos, nos de "igual", no sean realmente iguales.

Datos satélite, estabilidad, memoria adicional

Hay 3 cosas de qué preocuparse (bueno, hay más) cuando uno selecciona un algoritmo para ordenar.

1. Datos Satélite.
   - Usualmente cuando ordenas una lista (de números, nombres, o lo que sea), también incluyen varios otros datos que no estamos comparando.
   - Por ejemplo, número de cuenta, departamento, teléfono, etc. etc.
   - A esta información adicional Pero no queremos mover tooodo al intercambiar cosas, así que usualmente trabajas con "apuntadores" a los records.
   - Al dato que utilizamos para comparar le llamamos una key (llave).
2. Estabilidad
   - Dado que podemos traer datos satélite, muchas veces ocurre que aunque los datos, al compararlos, nos de "igual", no sean realmente iguales.
   - Decimos que un algoritmo es estable si deja fijo el orden que traían dos datos "iguales".

Datos satélite, estabilidad, memoria adicional

3 Memoria Adicional

- Algunos algoritmos se llaman inplace (*en su mismo lugar*), si utilizan a lo más una cantidad constante (es decir, $O(1)$) de memoria (con respecto al tamaño de la lista original).

¡Fin!

¡Gracias!