Cubeta y Radix

Miguel Raggi

Algoritmos
ENES
UNAM

3 de abril de 2018

Índice:

Índice:

Más de máximos y mínimos

- Hasta ahora hemos visto varios algoritmos para ordenar por comparación.

Más de máximos y mínimos

- Hasta ahora hemos visto varios algoritmos para ordenar por comparación.
- Vimos algoritmos que en su peor caso corren en tiempo $\Theta(n\log(n))$, y vimos que no se puede ordenar en menos.

Más de máximos y mínimos

- Hasta ahora hemos visto varios algoritmos para ordenar por comparación.
- Vimos algoritmos que en su peor caso corren en tiempo $\Theta(n\log(n))$, y vimos que no se puede ordenar en menos.
- Sin embargo, existen algoritmos que no son por comparación que pueden correr en tiempo lineal.

Más de máximos y mínimos

- Hasta ahora hemos visto varios algoritmos para ordenar por comparación.
- Vimos algoritmos que en su peor caso corren en tiempo $\Theta(n \log(n))$, y vimos que no se puede ordenar en menos.
- Sin embargo, existen algoritmos que no son por comparación que pueden correr en tiempo lineal.
- ¿Por qué podría ser más rápido un algoritmo que por comparación?

Más de máximos y mínimos

- Hasta ahora hemos visto varios algoritmos para ordenar por comparación.
- Vimos algoritmos que en su peor caso corren en tiempo $\Theta(n \log(n))$, y vimos que no se puede ordenar en menos.
- Sin embargo, existen algoritmos que no son por comparación que pueden correr en tiempo lineal.
- ¿Por qué podría ser más rápido un algoritmo que por comparación?
- Pues por ejemplo, supongamos que tenemos que ordenar exámenes en orden alfabético y agarramos dos: "Acevedo" y "Zamudio", y los comparamos, pues sabemos que Zamudio es mayor que Acevedo, pero, también sabemos que Acevedo probablemente quedará cerca del principio, y Zamudio quedará cerca del final.

Más de máximos y mínimos

- Hasta ahora hemos visto varios algoritmos para ordenar por comparación.
- Vimos algoritmos que en su peor caso corren en tiempo $\Theta(n\log(n))$, y vimos que no se puede ordenar en menos.
- Sin embargo, existen algoritmos que no son por comparación que pueden correr en tiempo lineal.
- ¿Por qué podría ser más rápido un algoritmo que por comparación?
- Pues por ejemplo, supongamos que tenemos que ordenar exámenes en orden alfabético y agarramos dos: "Acevedo" y "Zamudio", y los comparamos, pues sabemos que Zamudio es mayor que Acevedo, pero, también sabemos que Acevedo probablemente quedará cerca del principio, y Zamudio quedará cerca del final.
- No sabemos exactamente donde van a quedar, pero sabemos más o menos por donde.

Índice:

Counting sort

- Supongamos que queremos ordenar n números, pero que sabemos que todos ellos están entre 1 y k.

Counting sort

- Supongamos que queremos ordenar n números, pero que sabemos que todos ellos están entre 1 y k.
- Entonces podemos crear un arreglo C con k elementos, y hacer que $C[i]$ represente el número de veces que aparece el elemento i.

Counting sort

- Supongamos que queremos ordenar n números, pero que sabemos que todos ellos están entre 1 y k.
- Entonces podemos crear un arreglo C con k elementos, y hacer que $C[i]$ represente el número de veces que aparece el elemento i.
- Eso es muy fácil de hacer, sólo hay que leer cada número del arreglo, e incrementar el C correspondiente.

Counting sort

- Supongamos que queremos ordenar n números, pero que sabemos que todos ellos están entre 1 y k.
- Entonces podemos crear un arreglo C con k elementos, y hacer que $C[i]$ represente el número de veces que aparece el elemento i.
- Eso es muy fácil de hacer, sólo hay que leer cada número del arreglo, e incrementar el C correspondiente.
- ¿Cuánto tiempo toma?

Counting sort

- Supongamos que queremos ordenar n números, pero que sabemos que todos ellos están entre 1 y k.
- Entonces podemos crear un arreglo C con k elementos, y hacer que $C[i]$ represente el número de veces que aparece el elemento i.
- Eso es muy fácil de hacer, sólo hay que leer cada número del arreglo, e incrementar el C correspondiente.
- ¿Cuánto tiempo toma? $\Theta(n)$.

Counting sort

- Supongamos que queremos ordenar n números, pero que sabemos que todos ellos están entre 1 y k.
- Entonces podemos crear un arreglo C con k elementos, y hacer que $C[i]$ represente el número de veces que aparece el elemento i.
- Eso es muy fácil de hacer, sólo hay que leer cada número del arreglo, e incrementar el C correspondiente.
- ¿Cuánto tiempo toma? $\Theta(n)$.
- Después, utilizo el arreglo C para crear la lista de números pero ya en orden.

Counting sort

- Supongamos que queremos ordenar n números, pero que sabemos que todos ellos están entre 1 y k.
- Entonces podemos crear un arreglo C con k elementos, y hacer que $C[i]$ represente el número de veces que aparece el elemento i.
- Eso es muy fácil de hacer, sólo hay que leer cada número del arreglo, e incrementar el C correspondiente.
- ¿Cuánto tiempo toma? $\Theta(n)$.
- Después, utilizo el arreglo C para crear la lista de números pero ya en orden.
- En realidad, nunca comparé nada!

Counting sort

- Supongamos que queremos ordenar n números, pero que sabemos que todos ellos están entre 1 y k.
- Entonces podemos crear un arreglo C con k elementos, y hacer que $C[i]$ represente el número de veces que aparece el elemento i.
- Eso es muy fácil de hacer, sólo hay que leer cada número del arreglo, e incrementar el C correspondiente.
- ¿Cuánto tiempo toma? $\Theta(n)$.
- Después, utilizo el arreglo C para crear la lista de números pero ya en orden.
- En realidad, nunca comparé nada!
- Ejercicio: Escribe pseudo-código para counting sort suponiendo que no hay datos satélite

Counting sort: pseudo-código

Podría quedar así.

Counting sort: pseudo-código

Podría quedar así.

```
CountingSort(A):
    Sea C un arreglo de tamaño k, lleno de 0's
    Para a en A
        C[a] += 1
    Sea B un arreglo vacío
    Para i = 0 a k-1
        Agregar C[i] veces i a B
```

Ejercicio: Prográmalo en C++.

Counting sort: pseudo-código

Si sí hay datos satélite, hay que hacer algo un poquito diferente, pues necesitamos transferir los datos satélite de alguna manera.

Counting sort: pseudo-código

Si sí hay datos satélite, hay que hacer algo un poquito diferente, pues necesitamos transferir los datos satélite de alguna manera.

```
CountingSort(A):
  Sea C un arreglo de tamaño k, lleno de 0's
  Para j = 1 a tamaño de A
    C[A[j]] += 1
  Para i = 1 a k
    C[i] = C[i] + C[i-1]
  Para j = tamaño(A) a 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Preguntas

- ¿Es estable?

Preguntas

- ¿Es estable?
- ¿Es inplace (in-situ)?

Índice:

Radix Sort

- Este es el algoritmo que se utilizaba en las primeras computadoras de tarjetas para ordenar las tarjetas.

Radix Sort

- Este es el algoritmo que se utilizaba en las primeras computadoras de tarjetas para ordenar las tarjetas.
- La idea es ir ordenando los dígitos de los números de uno por uno.

Radix Sort

- Este es el algoritmo que se utilizaba en las primeras computadoras de tarjetas para ordenar las tarjetas.
- La idea es ir ordenando los dígitos de los números de uno por uno.
- Para hacer esto, utilizamos counting sort con $k = 10$.

Radix Sort

- Este es el algoritmo que se utilizaba en las primeras computadoras de tarjetas para ordenar las tarjetas.
- La idea es ir ordenando los dígitos de los números de uno por uno.
- Para hacer esto, utilizamos counting sort con $k = 10$.
- Intuitivamente se siente que debería uno ordenar por el dígito más significativo primero.

Radix Sort

- Este es el algoritmo que se utilizaba en las primeras computadoras de tarjetas para ordenar las tarjetas.
- La idea es ir ordenando los dígitos de los números de uno por uno.
- Para hacer esto, utilizamos counting sort con $k = 10$.
- Intuitivamente se siente que debería uno ordenar por el dígito más significativo primero.
- Sin embargo, Radix funciona justo al revés: Se ordena primero por el dígito menos significativo.

Radix Sort

- Este es el algoritmo que se utilizaba en las primeras computadoras de tarjetas para ordenar las tarjetas.
- La idea es ir ordenando los dígitos de los números de uno por uno.
- Para hacer esto, utilizamos counting sort con $k = 10$.
- Intuitivamente se siente que debería uno ordenar por el dígito más significativo primero.
- Sin embargo, Radix funciona justo al revés: Se ordena primero por el dígito menos significativo.
- La razón es porque cuando ordenas primero por el dígito más significativo, te quedarán varias pilas de números, y habría que tomar en cuenta en donde están las "separaciones" entre un dígito y otro, y eso sería difícil (aunque se puede, claro, pero no es tan eficiente)

Radix Sort

- Este es el algoritmo que se utilizaba en las primeras computadoras de tarjetas para ordenar las tarjetas.
- La idea es ir ordenando los dígitos de los números de uno por uno.
- Para hacer esto, utilizamos counting sort con $k = 10$.
- Intuitivamente se siente que debería uno ordenar por el dígito más significativo primero.
- Sin embargo, Radix funciona justo al revés: Se ordena primero por el dígito menos significativo.
- La razón es porque cuando ordenas primero por el dígito más significativo, te quedarán varias pilas de números, y habría que tomar en cuenta en donde están las "separaciones" entre un dígito y otro, y eso sería difícil (aunque se puede, claro, pero no es tan eficiente)
- Más bien, en radix sort se ordena primero por el dígito menos significativo.

Radix: Ejemplo

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix

- Para ordenar los dígitos, podemos utilizar cualquier ordenamiento estable, pero counting sort funciona muy bien para esto, porque sabemos que los dígitos estarán entre 0 y 9.

Radix

- Para ordenar los dígitos, podemos utilizar cualquier ordenamiento estable, pero counting sort funciona muy bien para esto, porque sabemos que los dígitos estarán entre 0 y 9.
- Claro, no necesariamente debemos hacerlo en base decimal, y de hecho es más eficiente hacerlo en otras bases (potencias de 2, por ejemplo).

Radix

- Para ordenar los dígitos, podemos utilizar cualquier ordenamiento estable, pero counting sort funciona muy bien para esto, porque sabemos que los dígitos estarán entre 0 y 9.
- Claro, no necesariamente debemos hacerlo en base decimal, y de hecho es más eficiente hacerlo en otras bases (potencias de 2, por ejemplo).
- Supongamos que tomamos n números d cifras cada uno en expansión k-aria.

Radix

- Para ordenar los dígitos, podemos utilizar cualquier ordenamiento estable, pero counting sort funciona muy bien para esto, porque sabemos que los dígitos estarán entre 0 y 9.
- Claro, no necesariamente debemos hacerlo en base decimal, y de hecho es más eficiente hacerlo en otras bases (potencias de 2, por ejemplo).
- Supongamos que tomamos n números d cifras cada uno en expansión k-aria.
- Entonces ordenar cada dígito toma tiempo $\Theta(n+k)$ (como en counting sort), pero como son d dígitos, toma tiempo $\Theta(d(n+k))$.

Radix

- Para ordenar los dígitos, podemos utilizar cualquier ordenamiento estable, pero counting sort funciona muy bien para esto, porque sabemos que los dígitos estarán entre 0 y 9.
- Claro, no necesariamente debemos hacerlo en base decimal, y de hecho es más eficiente hacerlo en otras bases (potencias de 2, por ejemplo).
- Supongamos que tomamos n números d cifras cada uno en expansión k-aria.
- Entonces ordenar cada dígito toma tiempo $\Theta(n+k)$ (como en counting sort), pero como son d dígitos, toma tiempo $\Theta(d(n+k))$.
- Usualmente d es constante y $k \in O(n)$, así que toma tiempo lineal.

Índice:

Bucket sort: analogía

- Es como si quisiéramos ordenar muchos papeles en orden alfabético (de apellido).

Bucket sort: analogía

- Es como si quisiéramos ordenar muchos papeles en orden alfabético (de apellido).
- Entonces creamos 27 carpetas, una para la “A”, otra para la “B”, etc.

Bucket sort: analogía

- Es como si quisiéramos ordenar muchos papeles en orden alfabético (de apellido).
- Entonces creamos 27 carpetas, una para la "A", otra para la "B", etc.
- Luego acomodamos a cada quien en su carpetas.

Bucket sort: analogía

- Es como si quisiéramos ordenar muchos papeles en orden alfabético (de apellido).
- Entonces creamos 27 carpetas, una para la “A”, otra para la “B”, etc.
- Luego acomodamos a cada quien en su carpetas.
- Finalmente ordenamos lo que hay dentro de cada carpeta con algún otro algoritmo.

Bucket sort: analogía

- Es como si quisiéramos ordenar muchos papeles en orden alfabético (de apellido).
- Entonces creamos 27 carpetas, una para la "A", otra para la "B", etc.
- Luego acomodamos a cada quien en su carpetas.
- Finalmente ordenamos lo que hay dentro de cada carpeta con algún otro algoritmo.
- El problema sería que hubiera mucha gente que se apellidara con la misma letra.

Bucket sort: analogía

- Es como si quisiéramos ordenar muchos papeles en orden alfabético (de apellido).
- Entonces creamos 27 carpetas, una para la "A", otra para la "B", etc.
- Luego acomodamos a cada quien en su carpetas.
- Finalmente ordenamos lo que hay dentro de cada carpeta con algún otro algoritmo.
- El problema sería que hubiera mucha gente que se apellidara con la misma letra.
- Vamos entonces a suponer que estamos ordenando números al azar en un rango específico.

Bucket Sort (ordenamiento cubeta)

- El ordenamiento cubeta sirve cuando todas las llaves están tomadas al azar en un intervalo (a, b) (de manera que no haya algunos que sean mucho más probables que otros).

Bucket Sort (ordenamiento cubeta)

- El ordenamiento cubeta sirve cuando todas las llaves están tomadas al azar en un intervalo (a, b) (de manera que no haya algunos que sean mucho más probables que otros).
- Por ejemplo, si todos los números están entre el 0 y el 1, entonces podemos utilizar ordenamiento cubeta.

Bucket Sort (ordenamiento cubeta)

- El ordenamiento cubeta sirve cuando todas las llaves están tomadas al azar en un intervalo (a, b) (de manera que no haya algunos que sean mucho más probables que otros).
- Por ejemplo, si todos los números están entre el 0 y el 1, entonces podemos utilizar ordenamiento cubeta.
- Es una idea sencilla y en estos casos puede ordenar muy rápido algunas listas.

Bucket Sort (ordenamiento cubeta)

- El ordenamiento cubeta sirve cuando todas las llaves están tomadas al azar en un intervalo (a, b) (de manera que no haya algunos que sean mucho más probables que otros).
- Por ejemplo, si todos los números están entre el 0 y el 1, entonces podemos utilizar ordenamiento cubeta.
- Es una idea sencilla y en estos casos puede ordenar muy rápido algunas listas.
- Es la idea base de las tablas "hash", que veremos después, que son muy muy útiles.

Bucket Sort (ordenamiento cubeta)

- El ordenamiento cubeta sirve cuando todas las llaves están tomadas al azar en un intervalo (a, b) (de manera que no haya algunos que sean mucho más probables que otros).
- Por ejemplo, si todos los números están entre el 0 y el 1, entonces podemos utilizar ordenamiento cubeta.
- Es una idea sencilla y en estos casos puede ordenar muy rápido algunas listas.
- Es la idea base de las tablas "hash", que veremos después, que son muy muy útiles.
- Es un algoritmo que puede ser muy rápido en una lista muy grande que haya que ordenar, dependiendo de la implementación, pero no es tan eficiente en listas pequeñas.

Bucket Sort

Dado un arreglo A, bucket sort con k cubetas funciona así:

- Creamos k cubetas.

Bucket Sort

Dado un arreglo A, bucket sort con k cubetas funciona así:

- Creamos k cubetas.
- Dividimos el intervalo $[0, 1]$ en k pedazos iguales, que van a corresponder a cada cubeta.

Bucket Sort

Dado un arreglo A, bucket sort con k cubetas funciona así:

- Creamos k cubetas.
- Dividimos el intervalo $[0, 1]$ en k pedazos iguales, que van a corresponder a cada cubeta.
- Para cada elemento $a \in A$, me fijo en qué pedazo del intervalo $[0, 1]$ está, y lo meto a la cubeta correspondiente.

Bucket Sort

Dado un arreglo A, bucket sort con k cubetas funciona así:

- Creamos k cubetas.
- Dividimos el intervalo $[0, 1]$ en k pedazos iguales, que van a corresponder a cada cubeta.
- Para cada elemento $a \in A$, me fijo en qué pedazo del intervalo $[0, 1]$ está, y lo meto a la cubeta correspondiente.
- Después ordeno lo que hay dentro de cada cubeta con quicksort o lo que sea.

Bucket Sort

Dado un arreglo A, bucket sort con k cubetas funciona así:

- Creamos k cubetas.
- Dividimos el intervalo $[0, 1]$ en k pedazos iguales, que van a corresponder a cada cubeta.
- Para cada elemento $a \in A$, me fijo en qué pedazo del intervalo $[0, 1]$ está, y lo meto a la cubeta correspondiente.
- Después ordeno lo que hay dentro de cada cubeta con quicksort o lo que sea.
- Finalmente concateno todas las cubetas.

Ejercicios

1. ¿Qué es lo peor que me puede pasar al hacer este ordenamiento?

Ejercicios

1. ¿Qué es lo peor que me puede pasar al hacer este ordenamiento?
2. Entonces, si utilizo quicksort para ordenar dentro de las cubetas, en el peor caso, ¿cuánto tiempo tarda?

Ejercicios

1. ¿Qué es lo peor que me puede pasar al hacer este ordenamiento?
2. Entonces, si utilizo quicksort para ordenar dentro de las cubetas, en el peor caso, ¿cuánto tiempo tarda?
3. ¿Cómo divido al intervalo $[0, 1]$ en k partes iguales?

Ejercicios

1. ¿Qué es lo peor que me puede pasar al hacer este ordenamiento?
2. Entonces, si utilizo quicksort para ordenar dentro de las cubetas, en el peor caso, ¿cuánto tiempo tarda?
3. ¿Cómo divido al intervalo $[0, 1]$ en k partes iguales?
4. Si tengo un número $a \in [0, 1]$, ¿cómo encuentro en qué cubeta va?

Ejercicios

1. ¿Qué es lo peor que me puede pasar al hacer este ordenamiento?
2. Entonces, si utilizo quicksort para ordenar dentro de las cubetas, en el peor caso, ¿cuánto tiempo tarda?
3. ¿Cómo divido al intervalo $[0, 1]$ en k partes iguales?
4. Si tengo un número $a \in [0, 1]$, ¿cómo encuentro en qué cubeta va?
5. Escribe pseudo-código para bucket sort.

Bucket Sort: Pseudocódigo

- Sea B una lista con k elementos, en donde cada elemento es una sublista vacía.
- Sea $n =$ tamaño(A)
- Para i desde 0 hasta n:
  - Inserta $A[i]$ en la cubeta $B[\lfloor kA[i] \rfloor]$.
- Para cada $b \in B$:
  - Ordena b con quicksort.
- Concatena las listas $B[0]$, $B[1]$, etc.

Generalización

- Bucketsort es una generalización de counting sort, en donde las cubetas sólo guardan “1” valor.

Generalización

- Bucketsort es una generalización de counting sort, en donde las cubetas sólo guardan "1" valor.
- También es una generalización de quicksort, donde hay 2 cubetas, y que siempre se divide justo a la mitad.

Generalización

- Bucketsort es una generalización de counting sort, en donde las cubetas sólo guardan "1" valor.
- También es una generalización de quicksort, donde hay 2 cubetas, y que siempre se divide justo a la mitad.
- Una optimización puede ser así: Antes de hacer quicksort en cada cubeta, mides el tamaño de una cubeta. Si es muy grande (que tanto, depende de la implementación), vuelves a dividir en cubetas. Si no lo es, entonces utilizas quicksort.

Generalización

- Bucketsort es una generalización de counting sort, en donde las cubetas sólo guardan "1" valor.
- También es una generalización de quicksort, donde hay 2 cubetas, y que siempre se divide justo a la mitad.
- Una optimización puede ser así: Antes de hacer quicksort en cada cubeta, mides el tamaño de una cubeta. Si es muy grande (que tanto, depende de la implementación), vuelves a dividir en cubetas. Si no lo es, entonces utilizas quicksort.
- Hay que medir entonces a partir de qué tamaño le gana bucketsort a quicksort.