Ordenación por distribución

Cómo ordenar con coste lineal

Marcos Herrero Agustín

Contenidos

Table of Contents

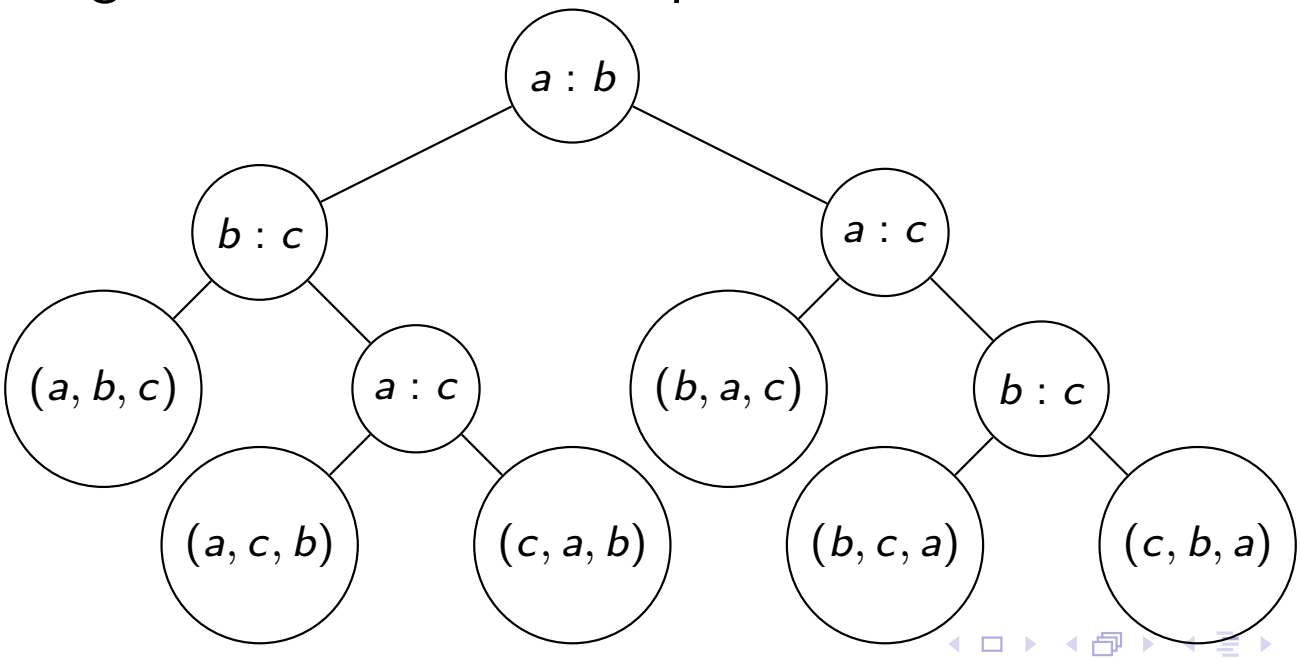
Costes de algoritmos de ordenación vistos hasta ahora

Algoritmo	Caso peor	Caso promedio	Espacio adicional
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Quicksort	$O(n^2)$	$O(n \log n)$	$O(n)$
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(1)$

¿Lo podemos hacer mejor?

Árboles de decisión

Dado un algoritmo de ordenación basado en comparaciones, un árbol de decisión de tamaño n es un árbol binario que representa todas las posibles secuencias de comparaciones que realizaría el algoritmo para llegar a la solución desde una entrada arbitraria de tamaño n. La ejecución del algoritmo para una entrada concreta correspondería a trazar un camino en el árbol desde la raíz hasta una de las hojas.
Por ejemplo, para un tamaño de entrada 3 el siguiente árbol de decisión representa el algoritmo de ordenación por inserción:



Teorema

Todo algoritmo de ordenación basado en comparaciones tiene coste en $\Omega(n \log n)$ en el caso peor

Dem Sea n el tamaño de la entrada. Todo algoritmo de ordenación basado en comparaciones puede representarse como un árbol de decisión, en el que las l hojas representan posibles permutaciones de la entrada. Como todas las permutaciones de la entrada han de estar en el árbol, $n! \leq l$. En el caso peor, el algoritmo realiza tantas comparaciones como la altura h del árbol. Como se trata de un árbol binario , se tiene:

$$n! \leq l \leq 2^h$$

y esto implica

$$h \geq \log(n!) = \sum_{i=2}^{n} \log(i) \geq \int_1^n \log x dx = \frac{1}{ln2}(n \ln n - n + 1)$$

$$\geq n \log n - 1.45n \approx n \log n$$

Por tanto, todo algoritmo de ordenación basado en comparaciones realiza, en el caso peor, un número de comparaciones en el orden de $n \log n$

Se puede probar un resultado similar para el coste en el caso promedio. Como consecuencia, obtenemos que Mergesort y Quicksort son óptimos como algoritmos de ordenación basados en comparaciones entre claves. Vamos ver, sin embargo, que existen otros procedimientos de ordenación no basados en comparaciones y que permiten obtener algoritmos más rápidos, estableciendo ciertas restricciones sobre la entrada.

Table of Contents

Counting Sort es un algoritmo sencillo que permite ordenar de forma rápida un vector de n objetos cuyas claves toman valores en un conjunto ordenado $\mathcal{F}$ de pocos elementos.

- Identificando cada clave con un número natural, supondremos que $\mathcal{F} = \{0, ..., k-1\}$

La idea del algoritmo se basa en recorrer el conjunto de claves posibles (los enteros de 0 a $k-1$) y determinar, para cada una, la posición o posiciones del vector en que deberán situarse los elementos con dicha clave.

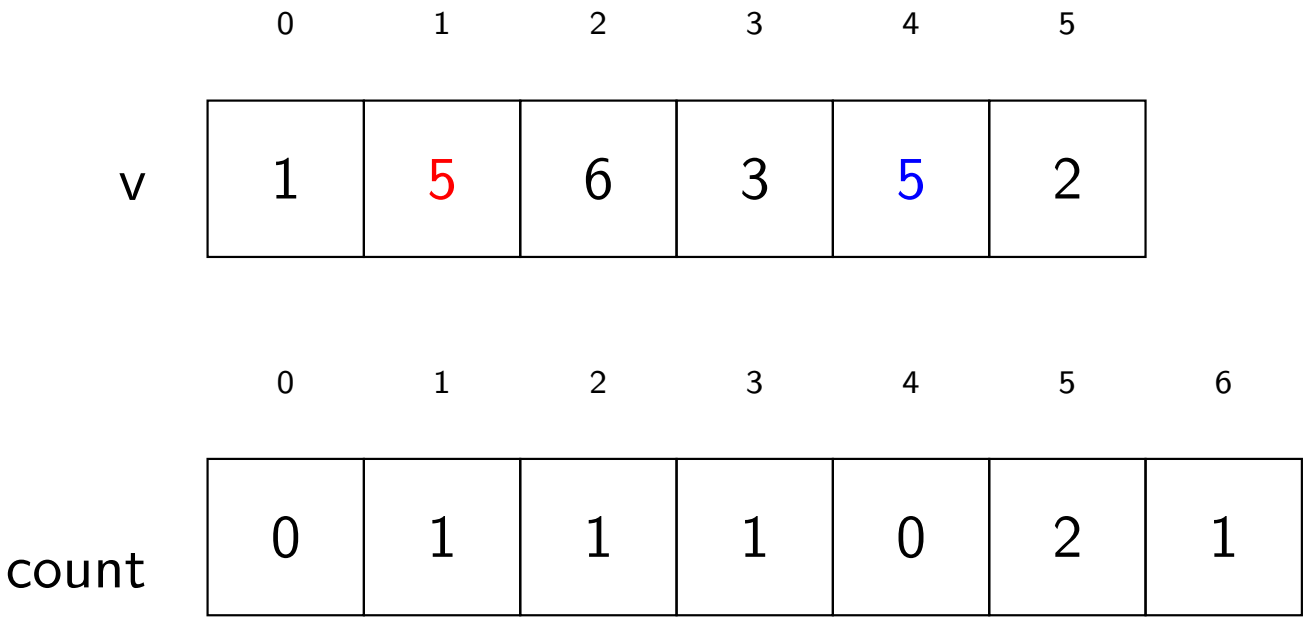
Observación

No se realizan comparaciones entre las claves de los elementos a ordenar, así que el coste en el caso peor de este algoritmo no tiene por qué estar acotado inferiormente por $n \log n$.

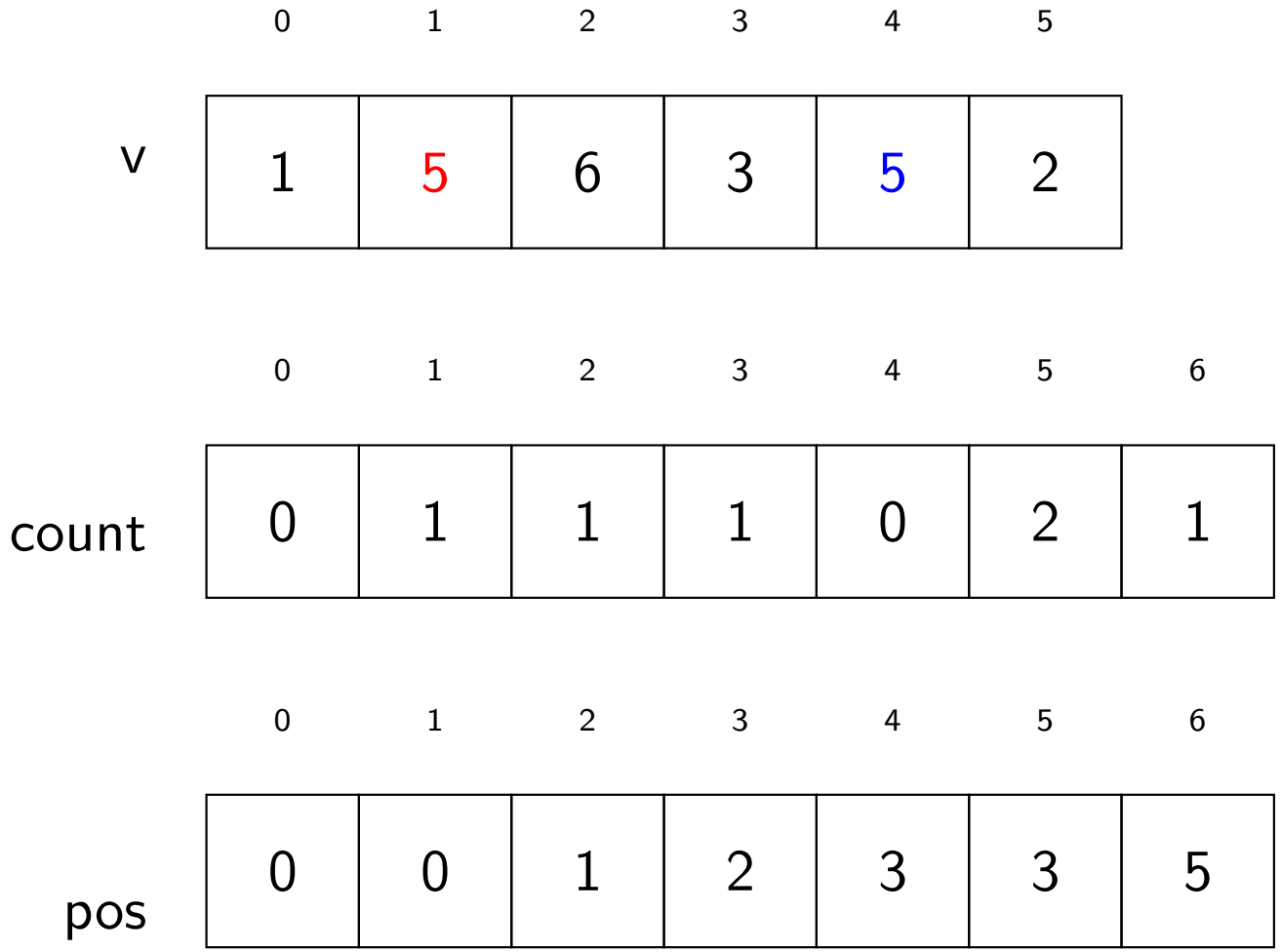
Algoritmo

Vamos a ver el proceso que sigue con un ejemplo:

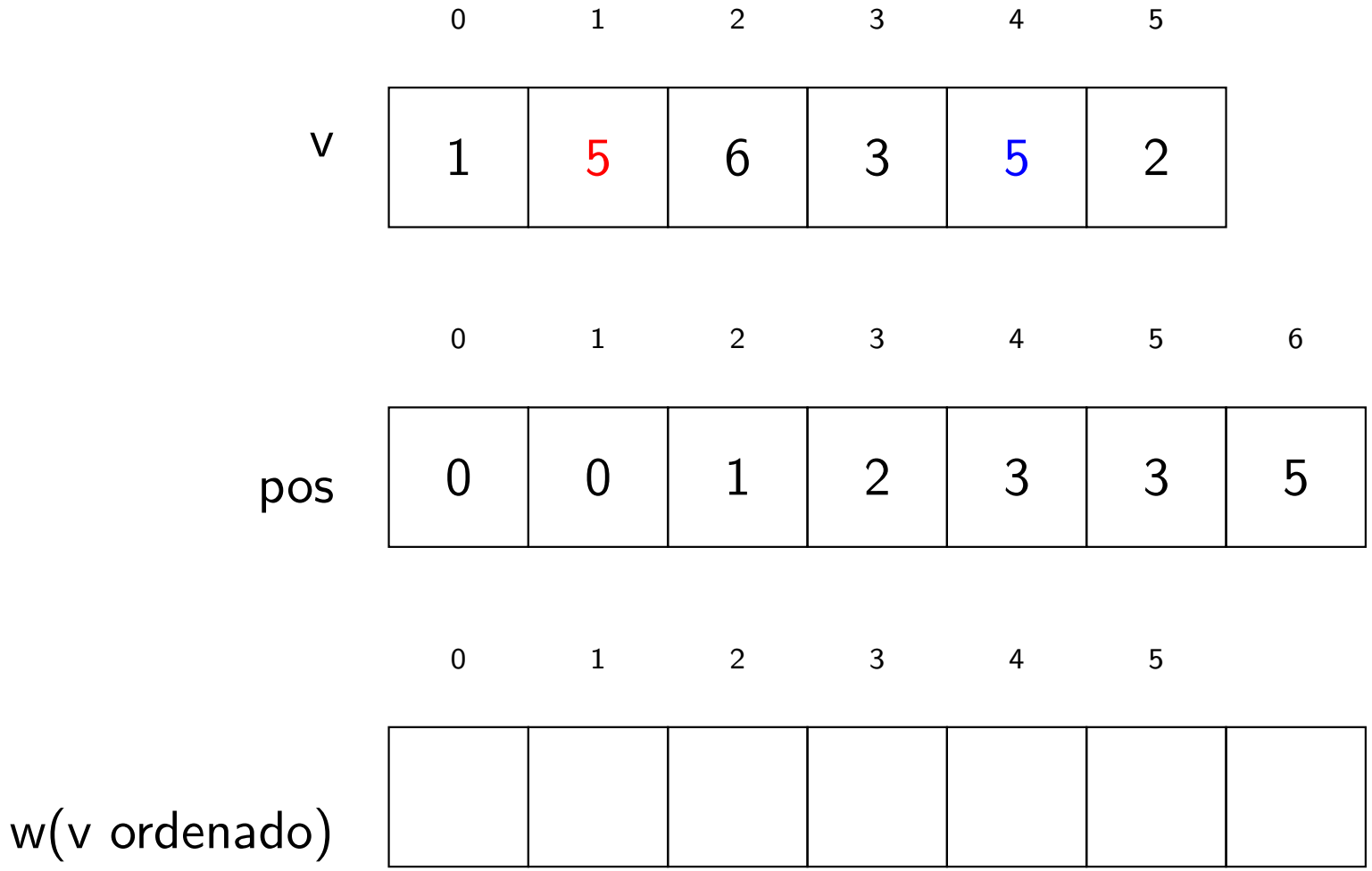
1. Comenzamos contando el número de apariciones de cada clave, llevando la cuenta en un vector de k posiciones.



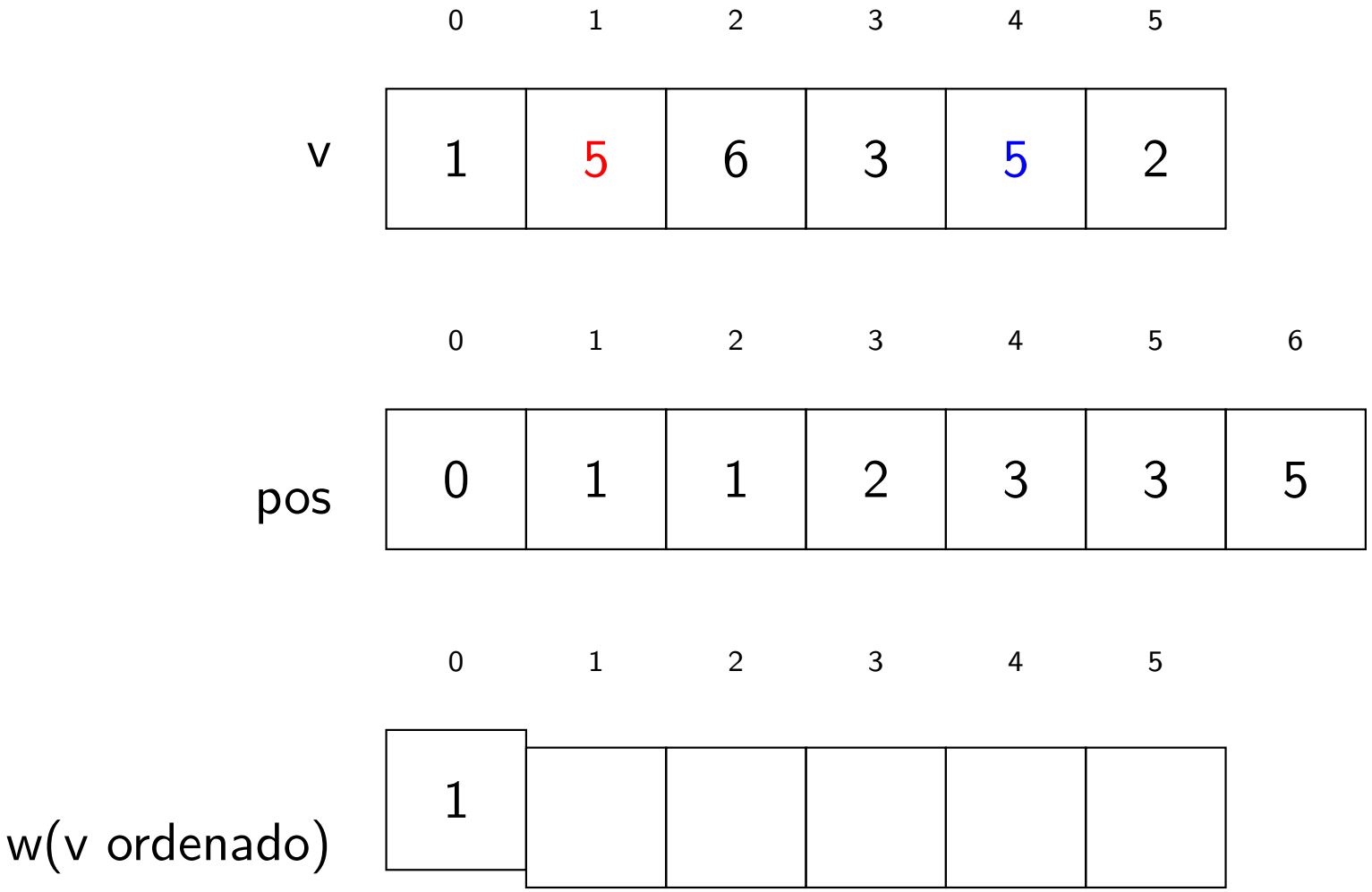
2 A continuación, recorremos este vector, calculando en qué rango de posiciones debería ir cada una de las claves.



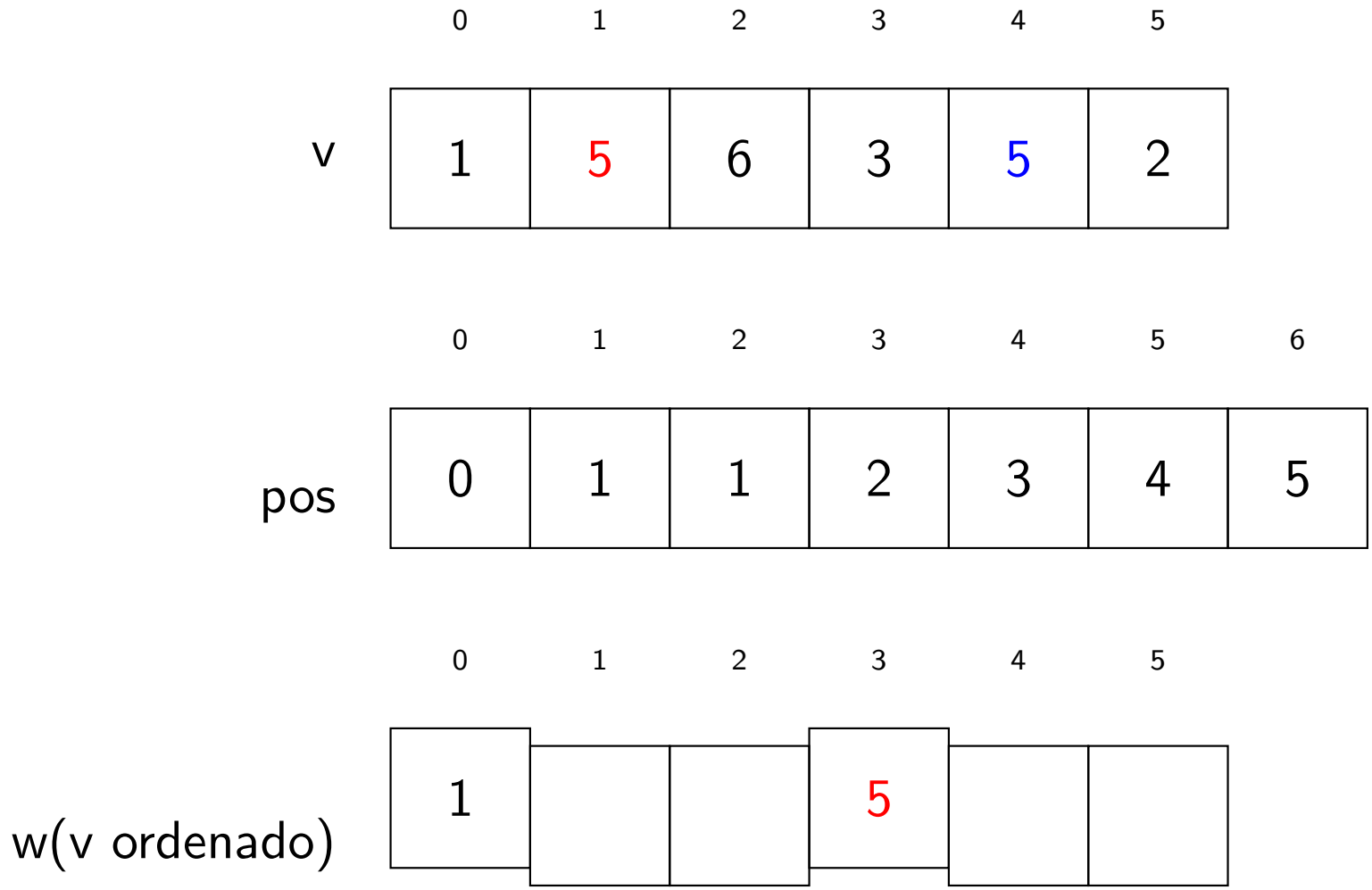
3 Por último, recorriendo nuevamente el vector inicial colocamos cada elemento en la posición que corresponde según su clave.



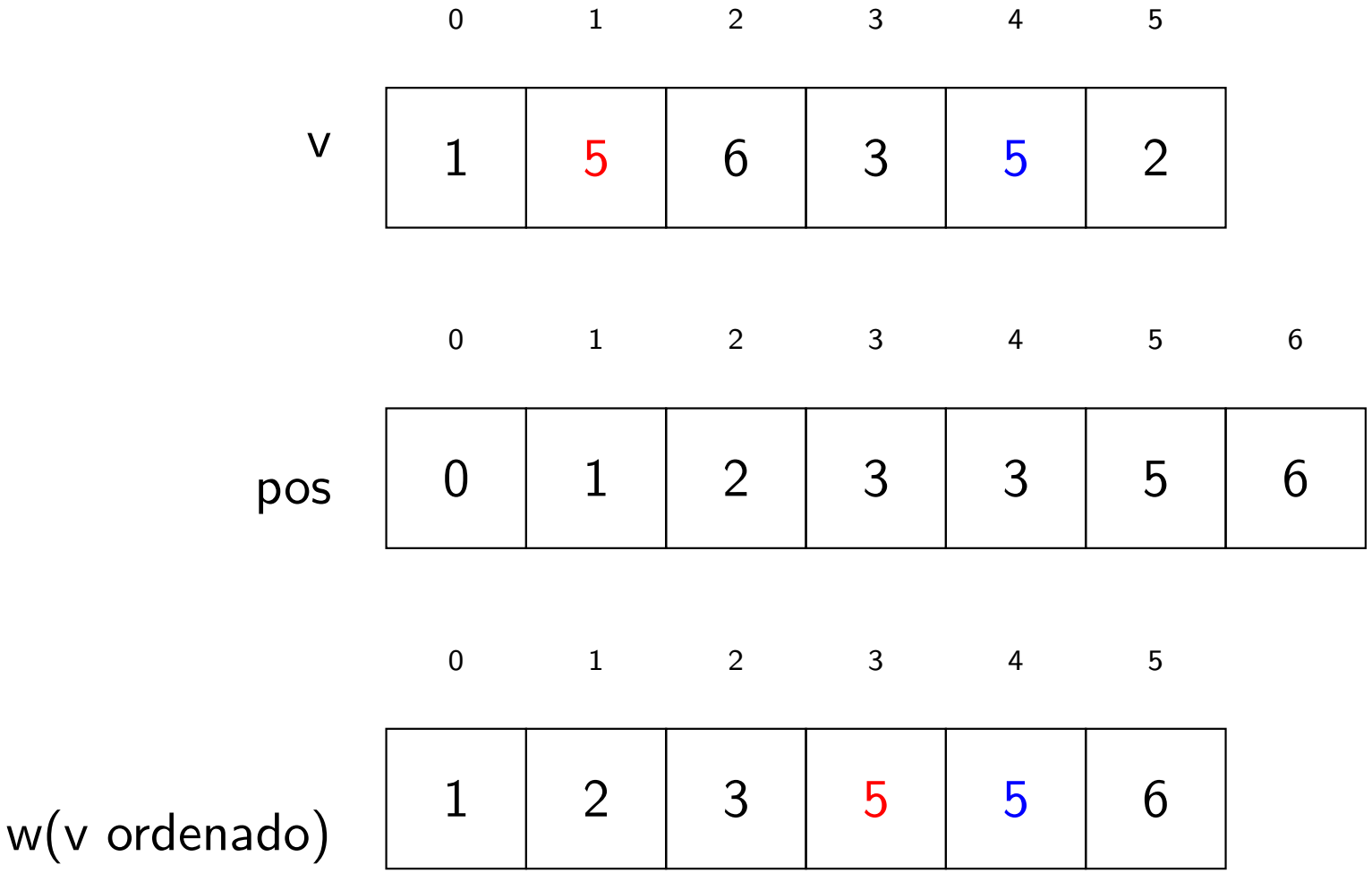
3 Por último, recorriendo nuevamente el vector inicial colocamos cada elemento en la posición que corresponde según su clave.



3 Por último, recorriendo nuevamente el vector inicial colocamos cada elemento en la posición que corresponde según su clave.



3 Por último, recorriendo nuevamente el vector inicial colocamos cada elemento en la posición que corresponde según su clave.



Código

```
void countingSort(std::vector<int> & v, int k) {
  int n = v.size();
  std::vector<int> count(k, 0);

  for (int i = 0; i < n;  ++i) ++count[v[i]];

  int empieza = 0, aux;
  for (int i = 0; i < k; ++i) {
    aux = count[i] + empieza;
    count[i] = empieza;
    empieza = aux;
  }

  std::vector<int> w(n);
  for (int i = 0; i < n; ++i) {
    w[count[v[i]]] = v[i];
    ++count[v[i]];
  }

  v = std::move(w);
}
```

$O(n)$

$O(k)$

$O(n)$

Análisis de la complejidad

Para un vector de n elementos y claves acotadas por k, el algoritmo tiene coste en tiempo $O(n+k)$. Por tanto, si $k \in O(n)$ este es un algoritmo de coste lineal sobre el número de elementos del vector.

El coste en espacio adicional está también en $O(n+k)$ debido a que se requiere almacenar el vector de cuentas y no se ordena sobre el propio vector de entrada, sino en un vector nuevo.

Estabilidad

Es importante notar que, cuando coinciden las claves, Counting Sort respeta el orden inicial de los elementos , es decir, se trata un algoritmo de ordenación estable. Esta propiedad es clave para algunas de las aplicaciones de este algoritmo, como su uso auxiliar en RadixSort.

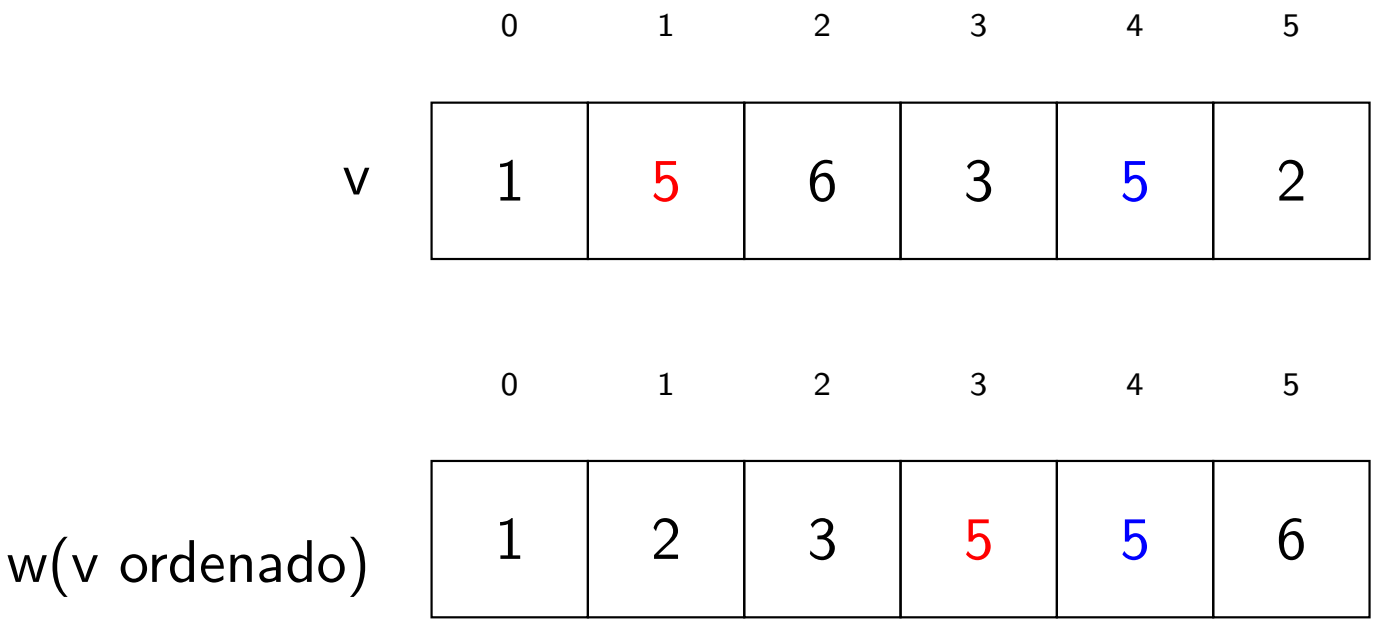


Table of Contents

RadixSort es un algoritmo de ordenación lexicográfica.

- Se aplica sobre conjuntos de elementos cuyas claves pueden verse como secuencias de dígitos (números, palabras,...), que toman valores en un conjunto finito ordenado.
- Dado un problema de este tipo, llamamos alfabeto al conjunto finito de valores que pueden tomar los dígitos de las secuencias. Denotamos por n el número de elementos a ordenar y por k el tamaño del alfabeto.
- Denotamos por d la longitud de la secuencia más larga a ordenar. Rellenando por la izquierda con el dígito de menor valor considerado, puede suponerse que todas las claves de los elementos a ordenar tienen la misma longitud d.

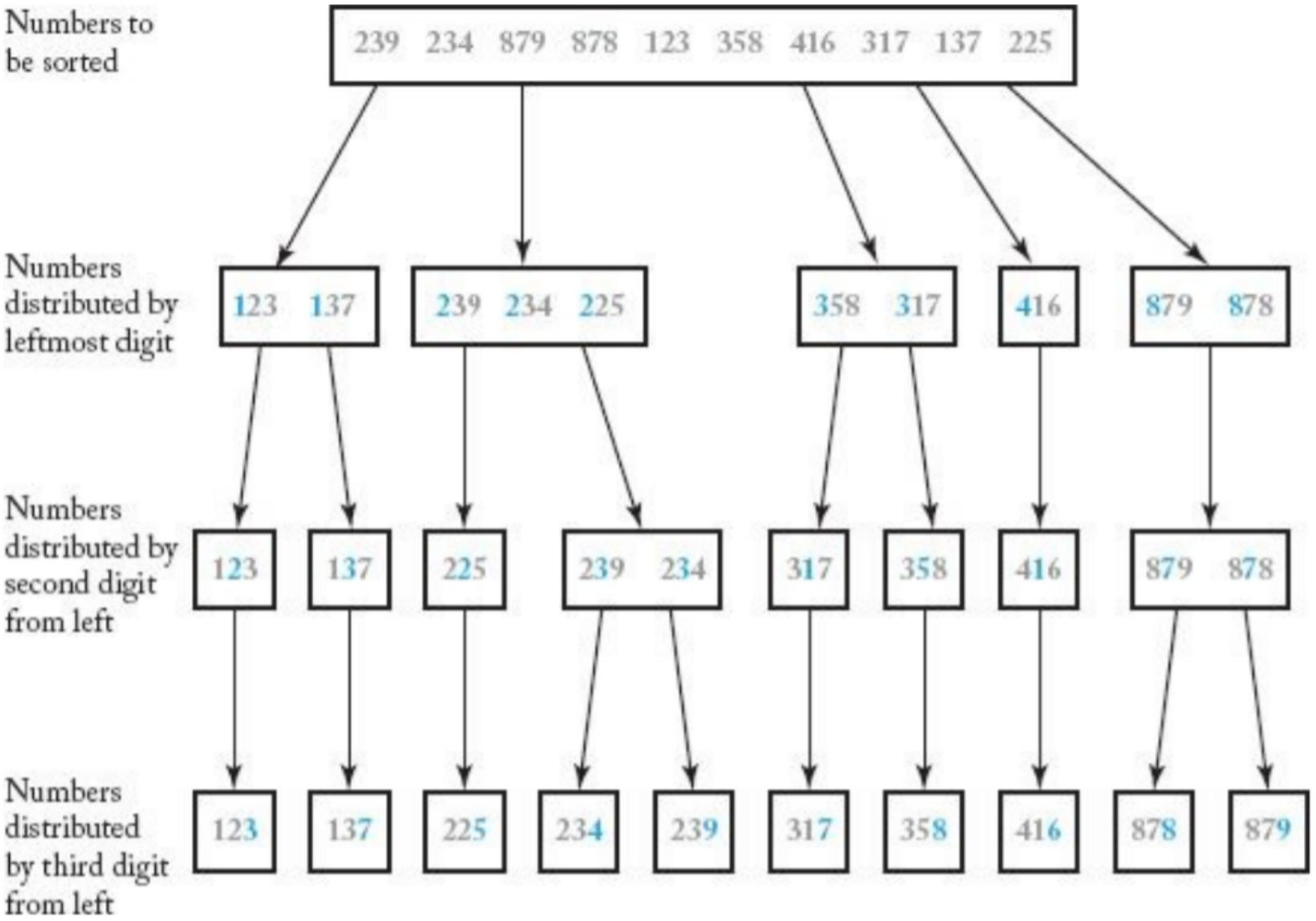
MSDRadixSort

- La idea del algoritmo consiste en ordenar todos los elementos según el dígito más significativo, a continuación resolver todos los empates de la iteración anterior utilizando el siguiente dígito, y proseguir así sucesivamente hasta haber considerado los d dígitos de cada secuencia o hasta haber clasificado todos los elementos sin encontrar empates.
- Al considerar los dígitos de más a menos significativo, la única manera de no perder el trabajo anterior al ordenar por el siguiente dígito es mantener separadas las secuencias que ya fueron diferenciadas por un dígito anterior hasta haber considerado todos los dígitos.

El esquema del algoritmo es el siguiente:

1. Clasificar los elementos en k cajas(buckets) según el valor del dígito más significativo
2. Ordenar recursivamente cada caja en base a los dígitos sucesivos
3. Trasladar ordenadamente el contenido de las cajas al vector final

Ejemplo


Numbers to be sorted
239 234 879 878 123 358 416 317 137 225
Numbers distributed by leftmost digit
123 137
239 234 225
358 317
416
879 878
Numbers distributed by second digit from left
123
137
225
239 234
317
358
416
879 878
Numbers distributed by third digit from left
123
137
225
234
239
317
358
416
878
879

Código

```
const int k = 10;
void MSDradixSort(std::vector<std::string> & v, int d){
  int n = v.size();
  if (n > 1 && d > 0) {
    std::vector<std::vector<std::string>> digitos(k);
    for (int i = 0; i < n; ++i) {
      if (v[i].size() < d) digitos[0].push_back(v[i]);
      else digitos[int(v[i][v[i].size() - d]) - int('0')].push_back
    (v[i]);
    }

    int ind = 0;
    for (int i = 0; i < k; ++i) {
      MSDradixSort(digitos[i], d - 1);

      for (int j = 0; j < digitos[i].size(); ++j) {
        v[ind] = digitos[i][j];
        ++ind;
      }

    }
  }
}
```

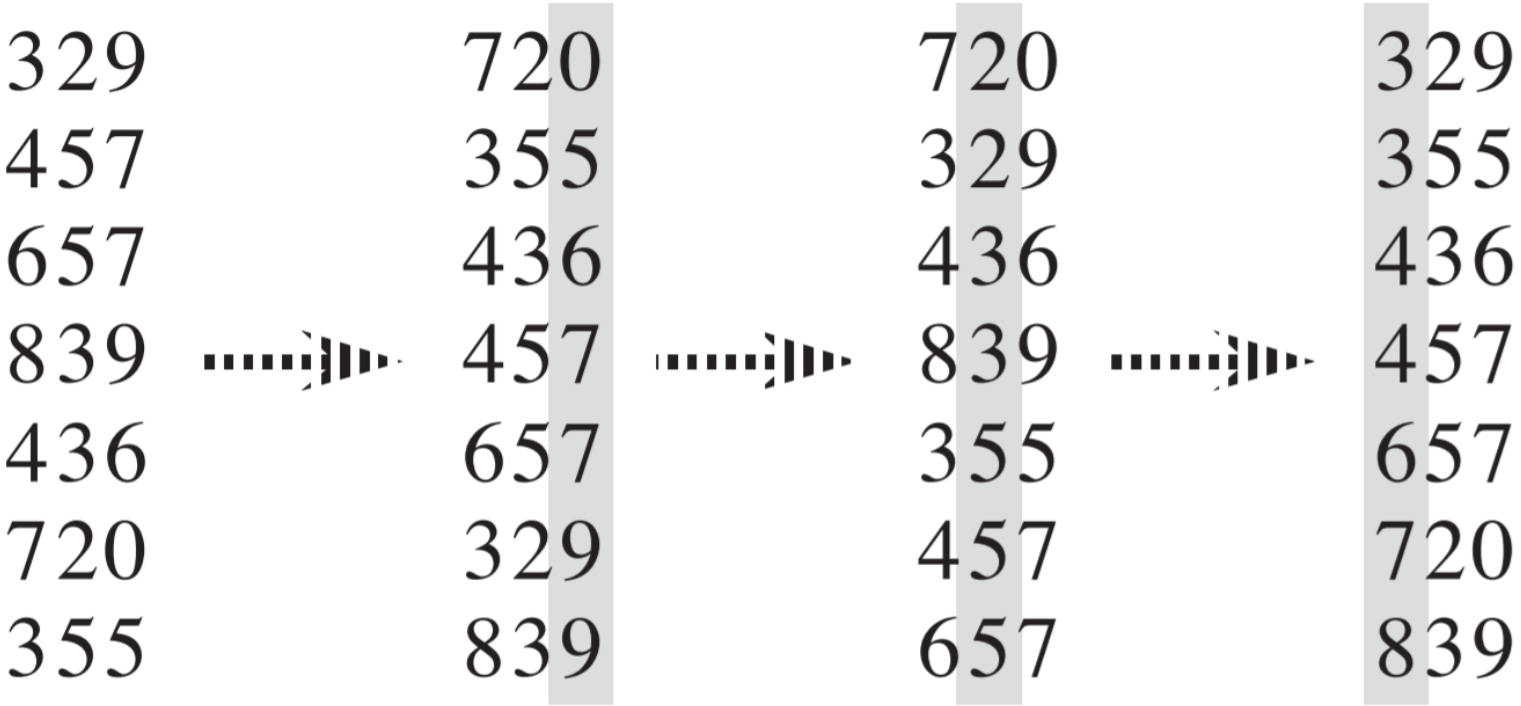
Análisis del coste

- Se tiene un coste en $\Theta(n * d + k^d)$ en el caso peor en que en cada llamada se hacen k llamadas recursivas y la profundidad de todas ellas es de d. Además el coste en espacio adicional es $\Theta(n * d)$, debido a que se han de mantener todas los buckets de las llamadas precedentes hasta el final.
- Estos costes asintóticos resultan inadmisibles, y nos llevan a afrontar este problema desde otro punto de vista.

LSDRadixSort

- El problema del algoritmo anterior radica en que el trabajo a realizar para no perder la ordenación según los dígitos anteriores resulta excesivo. Nos gustaría poder ordenar sucesivamente sobre el vector inicial sin preocuparnos de perder el trabajo anterior.
- Esto puede conseguirse planteando el problema a la inversa: comenzando a ordenar por el dígito menos significativo. De esta manera (dejando el trabajo importante para el final), no hemos de preocuparnos de perder la ordenación según el dígito anterior.
  - Al tratar un dígito r, el orden según los dígitos $r+1,\ldots,d-1$ solo es importante para resolver los posibles empates que haya en el dígito r.
  - Para conservar esta información, basta con que el algoritmo de ordenación utilizado en cada dígito sea estable

LSD Radix Sort: Ejemplo



LSD Radix Sort (apoyado en Counting Sort)

```
const int k = 10;
void LSDradixSortCount(std::vector<std::string> & v, int d){
  int n = v.size();
  for (int i = 0; i < d; ++i) {
    std::vector<int> grupos(k);
    for (int j = 0; j < n; ++j) {
      if (v[j].size() < i+1) ++grupos[0];
      else ++grupos[int(v[j].at(v[j].size() - i-1)) - int('0')];
    }

    int empieza = 0, aux;
    for (int j = 0; j < k; ++j) {
      aux = grupos[j] + empieza;
      grupos[j] = empieza;
      empieza = aux;
    }

    std::vector<std::string> w(n);
    for (int j = 0; j < n; ++j) {
      if (v[j].size() < i+1) {
        w[grupos[0]] = v[j];
        ++grupos[0];
      }
      else {
        w[grupos[int(v[j].at(v[j].size() - i-1)) - int('0')]] = v[j];
        ++grupos[int(v[j].at(v[j].size() - i-1)) - int('0')];
      }
    }

    v = std::move(w);
  }
}
```

LSD Radix Sort: Análisis del coste

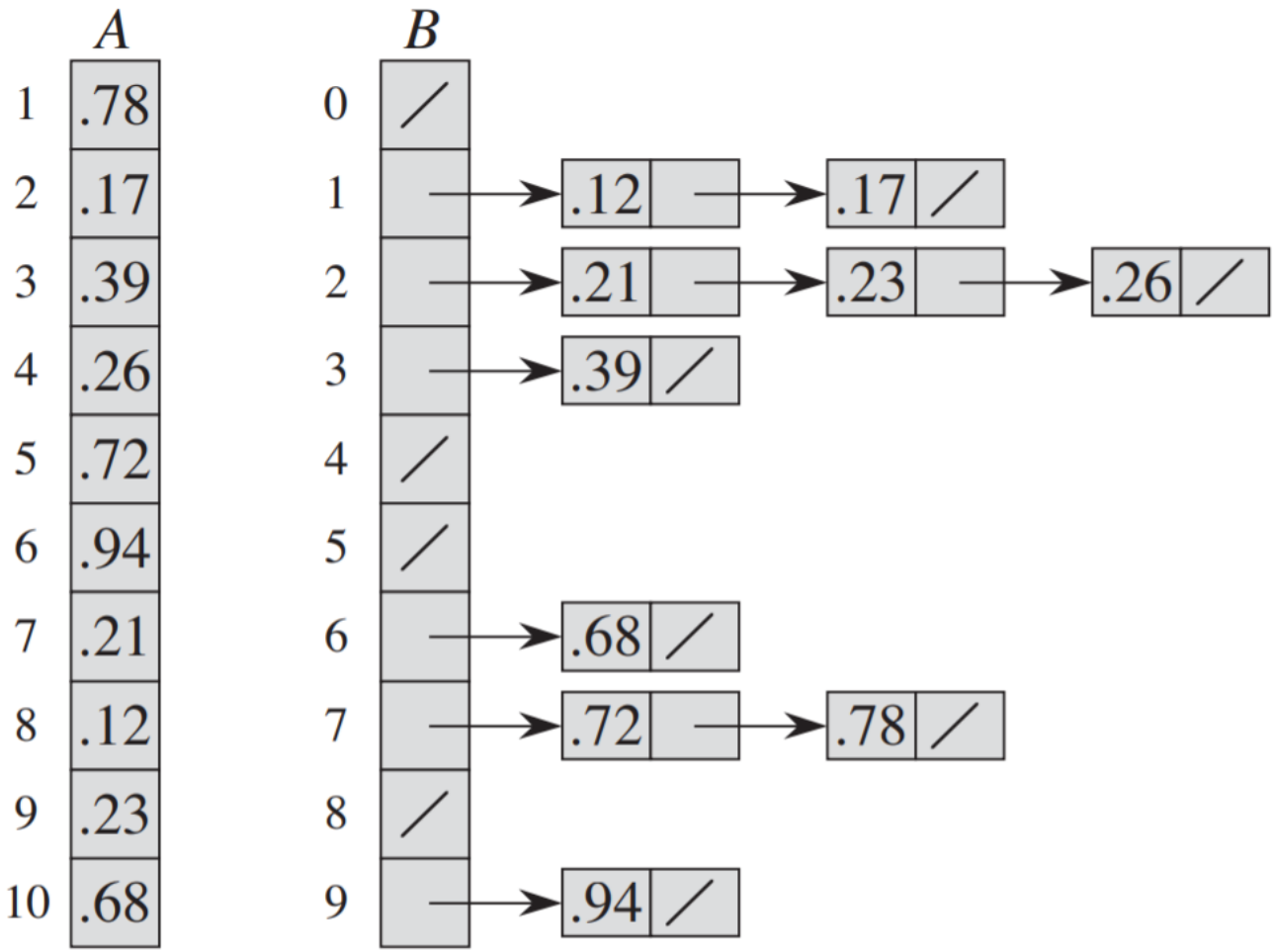
- Se aplica un algoritmo de ordenación auxiliar estable de coste $O(n + k)$ sobre las secuencias a ordenar, tantas veces como dígitos tiene la secuencia más larga. Por tanto, el coste en el caso peor está, en ambos casos, en $O(d * (n + k))$.
- Si $k \in O(n), d \in O(1)$, como es habitual, entonces la implementación de RadixSort que hemos construido tiene coste en $O(n)$, es decir, es lineal sobre el número de elementos a ordenar.

Table of Contents

BucketSort es un algoritmo para ordenar n números reales pertenecientes a un intervalo acotado $[a, b]$. La idea del algoritmo se basa en dividir $[a, b]$ en n subintervalos iguales y guardar los elementos a ordenar en buckets diferentes según a cuál de estos subintervalos pertenezca.

- El caso mejor se da cuando cada uno de los n buckets acaba conteniendo a exactamente un número, en cuyo caso basta con recorrerlos en orden para obtener el conjunto de números ya ordenado
- Si, por el contrario, quedan buckets vacíos y otros con más de un elemento, es posible que estos últimos no estén ordenados. Para ordenarlos podemos utilizar algoritmos de ordenación basados en comparaciones.

BucketSort: Ejemplo



BucketSort: Código

```
void bucketSort(std::vector<int> & v, int n, float a, float b) {
  if (n > 1) {
    int n = v.size();
    std::vector<std::vector<int>> buckets(n);

    for (int i = 0; i < n; ++i) buckets[floor(n*(v[i] - a) / (b - a
    ))].push_back(v[i]);
    for (int i = 0; i < n; ++i) ordenar(buckets[i]);

    int ind = 0, j = 0;
    for (int i = 0; i < n; ++i) {
      while (buckets[ind].size() <= j) {
        ++ind;
        j = 0;
      }

    v[i] = buckets[ind][j];
    ++j;
    }
  }
}
```

BucketSort: Análisis del coste

- El caso peor del algoritmo se da cuando todos los elementos acaban en el mismo bucket. En este caso, el coste en tiempo será el del algoritmo utilizado para ordenar cada bucket, es decir, al menos $n \log n$ en el caso peor.
- No obstante, si suponemos que los elementos a ordenar están uniformemente distribuidos en el intervalo $[a, b]$, cada bucket tendrá una cantidad de elementos independiente de n con probabilidad 1. Entonces, el coste de ordenar cada bucket será constante y, por tanto, el coste total del algoritmo será únicamente el de distribuir los elementos y concatenar los buckets al final: $O(n)$

Bibliografía

- Richard E. Neapolitan *Foundations of Algorithms*
  Capítulo 7
- T. H. Cormen *Introduction to Algorithms. Second edition*
  Capítulo 8