Flujo máximo en redes

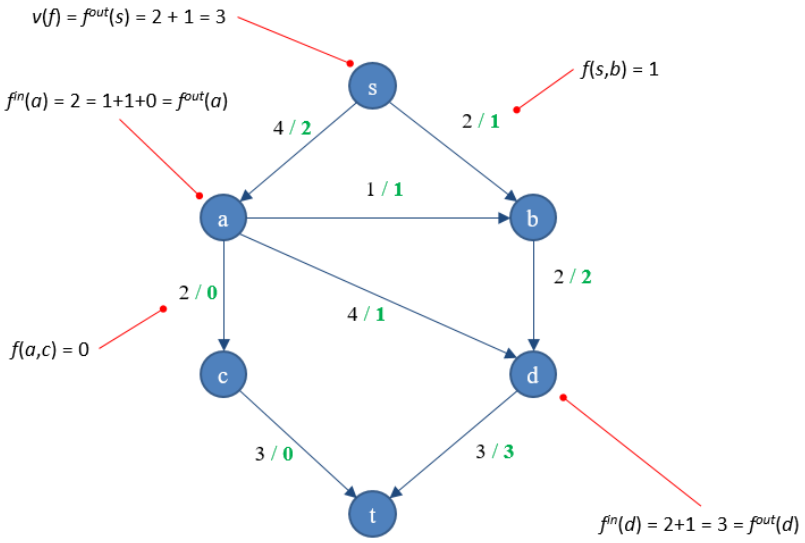
Introducción

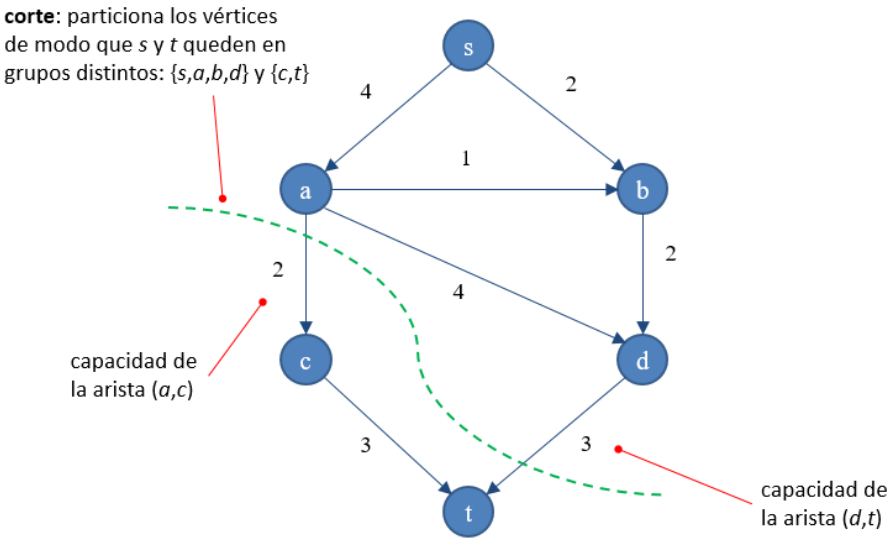
Una red de flujo es un grafo direccional *G = (V, E)* en donde cada arista *e* tiene:

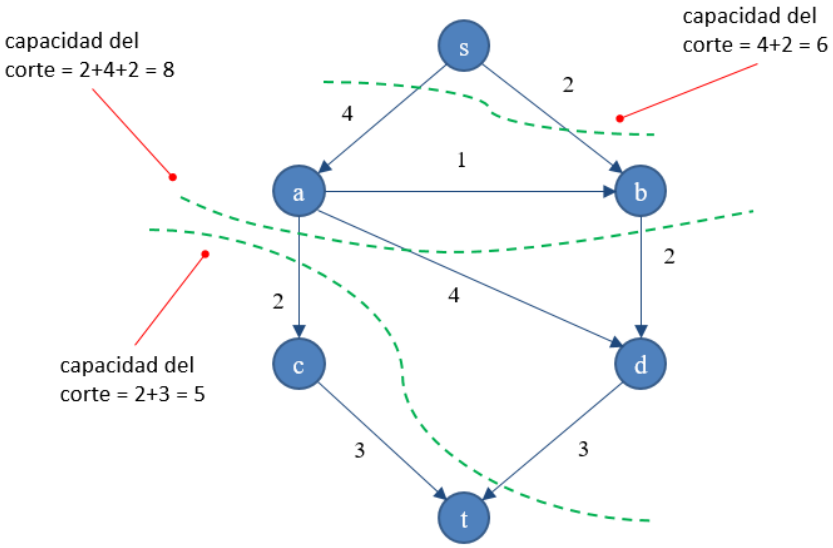
1. **Capacidad** (C_e): un número entero no negativo
2. **Nodo Fuente** (*s*): único, en donde ninguna arista llega a éste. Solo esta puede generar flujo
3. **Nodo Sumidero** (*t*): único, en donde ninguna arista sale de *t*. Solo esta puede consumir flujo

Un **flujo s-t** es una función ***f*** que asigna a cada arista *e* un número real no negativo *f(e)* que satisface:

1. Para cada *e*, $0 \leq f(e) \leq c_e$
2. Para cada nodo *v* distinto de *s* y *t*, la suma de los flujos en las aristas que llegan a *v* debe ser igual a la que sale de v.

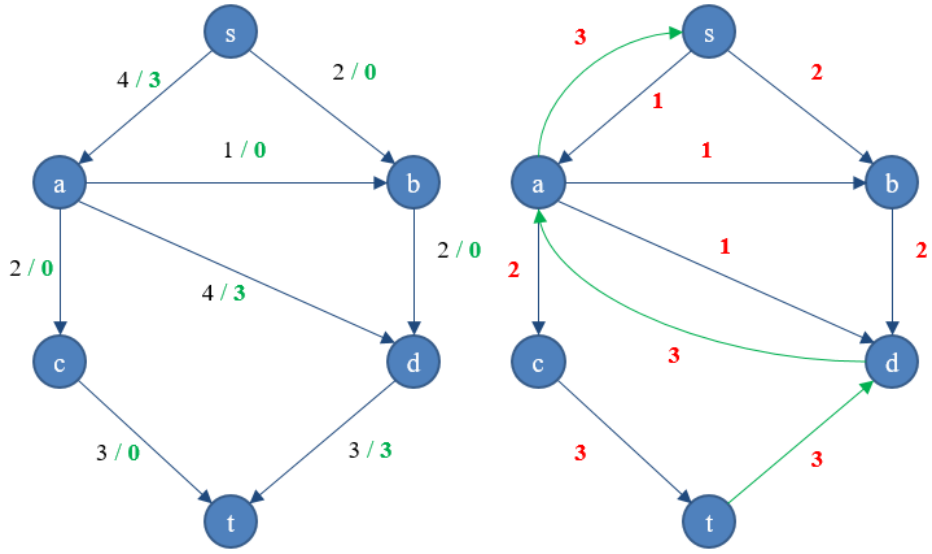






Dada una red *G* y un flujo *f* se define la **red residual G_f** como el conjunto de nodos de *G* en donde:

1. El conjunto de nodos es el mismo que *G*.
2. cada arista $e = (u, v)$ en que $f(e) < c_e$, incluimos la arista $e = (u, v)$ en G_f, con capacidad c_e-$f(e)$: **arista forward**.
3. cada arista $e = (u, v)$ en que $f(e) > 0$, incluimos la arista $e' = (u, v)$ en G_f, con capacidad $f(e)$: **arista backward**.

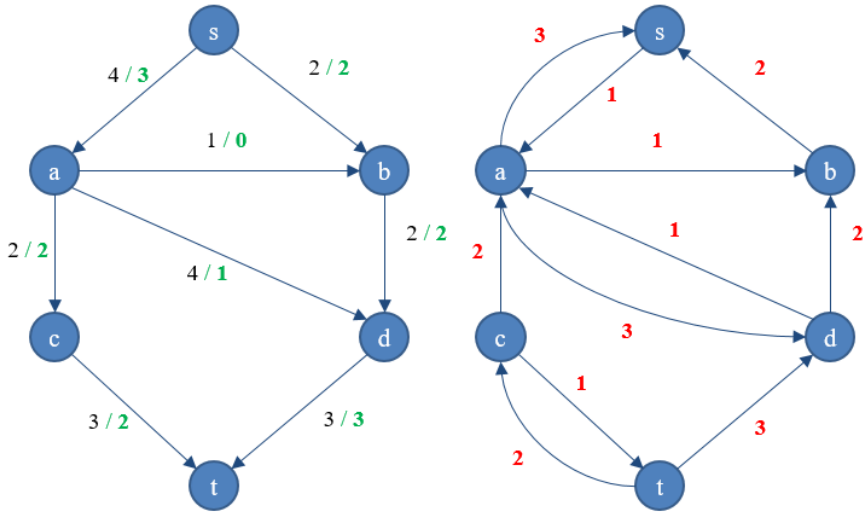


Por lo tanto, cada arista *e* en *G* puede dar origen a una o dos aristas en G_f:

1. Si $0 < f(e) < c_e$, entonces se traduce tanto en una arista *forward* como en una *backwards* en G_f
2. G_f tiene a lo más el doble de aristas que *G*

En G_f podemos encontrar una ruta simple *p* de *s* a *t* llamada **ruta de aumento** de tal manera que se puede definir un nuevo flujo *f'* en *G* tal que para cada arista *e = (u, v)* en *p*:

1. Si es una arista *forward* entonces hay que aumentar *f(e)* en la red *G* en la menor capacidad de cualquier arista en *p* con respecto al flujo *f*
2. Si es una arista *backward* entonces hay que disminuir *f(e)* en la red *G* en la menor capacidad de cualquier arista en *p* con respecto al flujo *f*



La aplicación repetida de la figura anterior constituye el **algoritmo Ford-Fulkerson** para encontrar flujos máximos.

Algoritmo Ford-Fulkerson

El algoritmo se detiene cuando no hay rutas simples de *s* a *t* en G_f, y se puede demostrar que, si *f* es un flujo máximo en *G*, entonces la red residual no tiene rutas de aumento de *s* a *t*, y viceversa.

Algorithm Ford-Fulkerson

Inputs Given a Network $G = (V, E)$ with flow capacity *c*, a source node *s*, and a sink node *t*

Output Compute a flow *f* from *s* to *t* of maximum value

1. $f(u, v) \leftarrow 0$ for all edges (u, v)
2. While there is a path p from *s* to *t* in G_f, such that $c_f(u, v) > 0$ for all edges $(u, v) \in p$:
    1. Find $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$
    2. For each edge $(u, v) \in p$
        1. $f(u, v) \leftarrow f(u, v) + c_f(p)$ *(Send flow along the path)*
        2. $f(v, u) \leftarrow f(v, u) - c_f(p)$ *(The flow might be "returned" later)*

En cualquier etapa intermedia del algoritmo, los valores de flujo y las capacidades residuales son números enteros, por lo que eventualmente se llegará a un tope y el algoritmo terminará.

Además, como el flujo empieza en 0 y aumenta en cada iteración al menos en una unidad y no puede superar la capacidad máxima, entonces el ciclo *while* se ejecutará a lo más *C* veces, por lo que el algoritmo es *O(|E|C)*. Luego, el algoritmo FF es *O(E|f*|)*

Teorema del flujo máximo corte mínimo

Asegura lo siguiente:

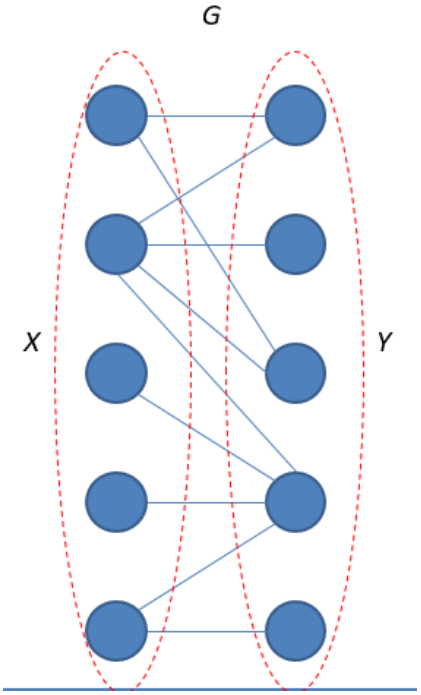
1. *f* es un flujo máximo en *G*
2. La red residual no tiene rutas de aumento
3. *|f|=* capacidad del corte *(s, t)* para algún corte *(s, t)* de *G*

Mejoras

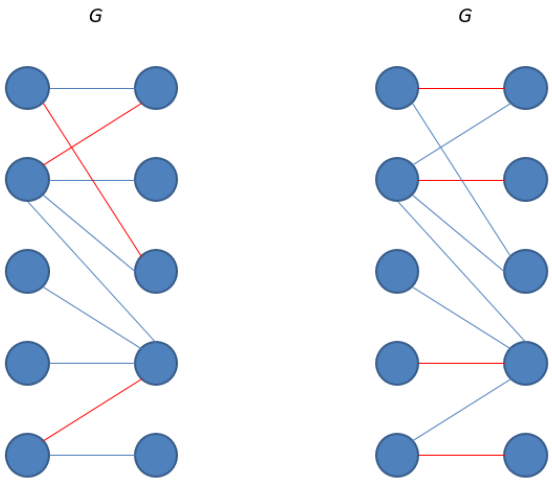
Es posible mejorar la cota *O(E|f*|)* si buscamos rutas *s-t* en la red residual usando BFS, ya que la ruta de aumento es una ruta más corta de *s* a *t*, en que cada arista tiene distancia unitaria (algoritmo de Edmonds-Karp) es $O(VE^2)$, en donde el número de aumentos realizados es *O(VE)*.

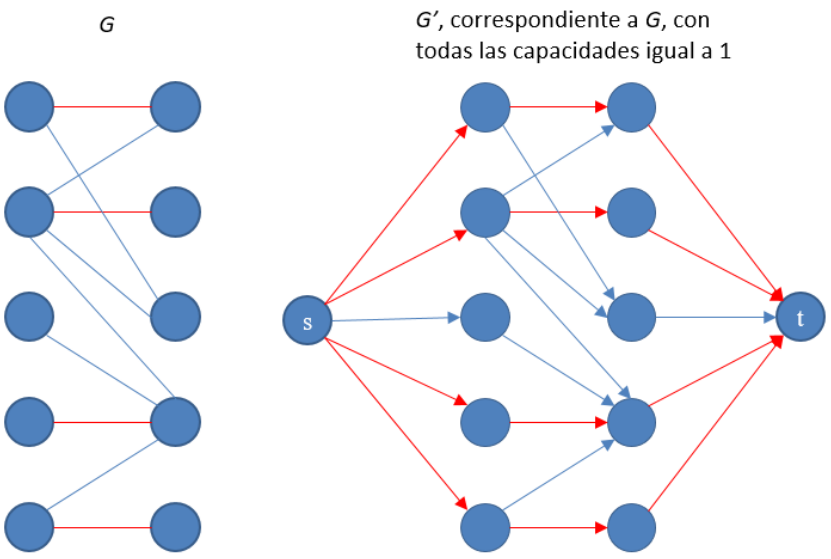
Emparejamiento Bipartito

Un grafo **bipartito** *G = (V, E)* es un grafo no direccionado cuyo conjunto de nodos puede ser particionado como *V = X U Y* tal que toda arista *e* tiene un extremo en *X* y otro en *Y*.



Un emparejamiento *M* en *G* es un subconjunto de las aristas *M* ⊆ *E* tal que cada nodo aparece en a lo más una arista en *M*. El problema del emparejamiento bipartito consiste en **encontrar un emparejamiento en *G* del mayor tamaño posible**.





Los flujos en *G'* representan emparejamientos en *G*. Primero, suponemos que hay un emparejamiento en *G* que contiene *k* aristas, en donde el flujo *f* envía una unidad por cada ruta, luego, las condiciones de capacidad y conservación se cumplen por lo que el flujo *f* es igual a *k*. Segundo, si hay un flujo *f'* en *G'* de valor *k*, si todas las capacidades en la red son enteros, entonces hay un flujo de valor *k* de puros números enteros, y como todas las capacidades son 1, se tiene que *f(e)* es 0 o 1 para cada arista *e*. Por lo tanto, si *M'* es el conjunto de aristas en que el valor del flujo es 1, *M'* contendrá *k* aristas, en donde cada nodo en *X* es la cola de a lo más una arista en *M'*, y cada nodo en *Y* es la cabeza de a lo más una arista en *M'*, luego, el tamaño de un emparejamiento máximo en *G* es igual al valor del flujo máximo en *G'* y las aristas del emparejamiento son las aristas que llevan flujo de *X* a *Y* en *G'*.

Ordenación en O(n)

Introducción

No es posible ordenar más rápido que O(n log(n)) si la única información usada por el algoritmo es el resultado de comparar, repetidamente, dos datos para determinar cuál es mayor, ya que cada comparación puede producir dos resultados, y por eso mismo, es un algoritmo de ordenación por comparación.

Los algoritmos de ordenación por comparación siguen una ruta en el árbol, desde la raíz hasta una hoja, por lo que el desempeño del algoritmo en el peor caso corresponde a seguir la ruta más larga. Ejemplificando, el árbol para ordenar n datos tiene n! hojas, y como es un árbol binario, la longitud h de la ruta más larga cumple con $2h \geq n!$ por lo que $h \geq \log n! = \Omega(n \log n)$.

Counting Sort

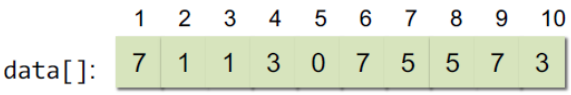
Es un algoritmo de ordenación que no compara los datos que está ordenando. Lo que hace es suponer que cada uno de los n datos es un entero en el rango 0 a k, con k entero, por lo que si k es $O(n)$, entonces *counting sort* corre en tiempo $O(n)$.

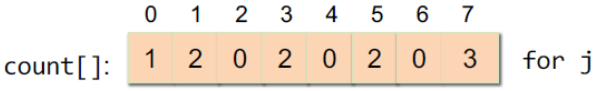
Lo que se hace es determinar, para cada dato x, el número de datos menores que x para ubicar a x directamente en su posición final en el arreglo de salida, pero hay que manejar el caso en que varios datos tengan el mismo valor.

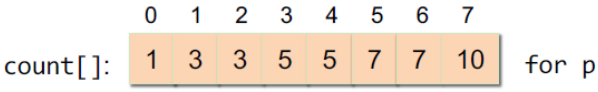
```
countingSort(data, tmp, k, n):
    sea count[0..k] un nuevo arreglo
    for i = 0 … k:
        count[i] = 0
    for j = 1 … n:
        count[data[j]] = count[data[j]]+1
    for p = 1 … k:
        count[p] = count[p]+count[p-1]
    for r = n … 1:
        tmp[count[data[r]]] = data[r]
        count[data[r]] = count[data[r]]-1
```

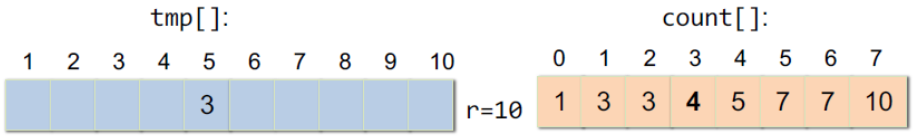
Este algoritmo es (claramente) $\Theta(k+n)$

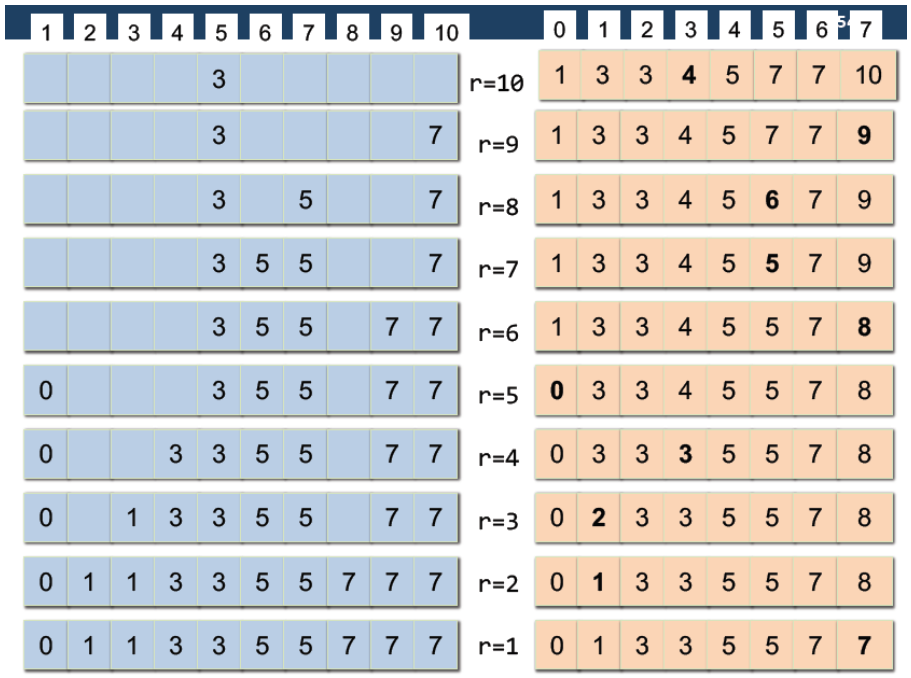
Si k es $O(n)$, entonces countingSort es $\Theta(n)$











Muy útil cuando se quieren ordenar enteros cuyo rango es pequeño

Radix Sort

Algoritmo usado por las máquinas que ordenaban tarjetas perforadas, en donde cada tarjeta tiene 80 columnas, y cada columna puede perforar un hoyo en uno de 12 lugares, por lo que la máquina se programa para examinar una determinada columna de cada tarjeta y distribuir la tarjeta en uno de 12 compartimientos, dependiendo de la perforación.

Una persona recolecta las tarjetas de cada compartimiento de modo que las tarjetas con la perforación en el primer lugar quedan encima de las tarjetas con la perforación en el segundo lugar, y así consecutivamente. Luego, un número de *d* dígitos ocupa *d* columnas, y como la máquina mira solo una columna a la vez, ordenar *n* tarjetas según el número de *d* dígitos requiere un algoritmo de ordenación.

Podríamos ordenar los números según su dígito más significativo, luego ordenar recursivamente cada compartimiento, y finalmente combinar los contenidos de cada compartimiento. Sin embargo, ordena en base al menos significativo primero, así las tarjetas son combinadas de modo que las que vienen en el compartimiento 0 quedan arriba de las que vienen del 1, y estas arriba del 2, etcétera.

Luego, todas las tarjetas son ordenadas en base al segundo menos significativo y recombinadas simultáneamente. El proceso sigue hasta que las tarjetas han sido ordenadas según los *d* dígitos, luego se requieren solo *d* pasadas por todas las tarjetas.

Arreglo inicial	Ordenado por dígito 1s	Ordenado por dígito 10s	Ordenado por dígito 100s
064	000	000	000
008	001	001	001
216	512	008	008
512	343	512	027
027	064	216	064
729	125	125	125
000	216	027	216
001	027	729	343
343	008	343	512
125	729	064	729

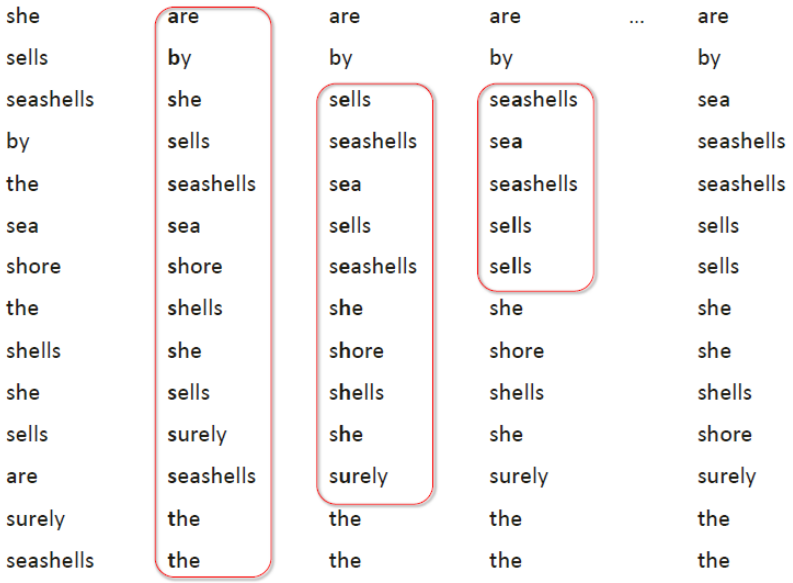
```
void radixSort(a, d)
   for j = 1 … d:
       usando una ordenación estable,
       ordene el arreglo a según el dígito j
```

Si el arreglo *a* contiene *n* números de *d* dígitos, en donde cada dígito puede tomar hasta *k* valores posibles, entonces *radixSort* puede tomar hasta *k* valores posibles, por lo que toma tiempo *O(d(n+k))* en ordenar los *n* números, ergo, si *d* es constante y *k=O(n)* entonces *radixSort* es *O(n)*.

Útil para ordenar strings cuando son del mismo largo.

MSD (Most Significant Digit) StringSort: String de largos diferentes

Se usa *countingSort* para ordenar los strings según el primer carácter, luego, recursivamente, se ordenan los subarreglos correspondientes a cada carácter (excluyendo el primero). Así como *quicksort*, *MSD StringSort* particiona el arreglo en subarreglos que pueden ser ordenados independientemente, pero lo particiona en un **subarreglo para cada posible valor del primer carácter**, en lugar de las dos particiones de *quicksort*.



Cuidados:

1. Fin del string: “she” es menor que “shells”
2. Alfabeto: binario (2), minúsculas (26), minúsculas + mayúsculas + dígitos (64), ASCII (128), Unicode (65536)
3. Subarreglos pequeños: tamaño < 11, cambiar a un *insertionSort* que sepa que los *p* primeros caracte res de los strings que está ordenando son iguales