Considere agora a situação em que nós sabemos que todos os números da lista são, digamos, menores do que 1000.

321	786	110	050	488	346	555	487	128	551	329	323	486

A ideia do algoritmo radix consiste em ordenar a lista examinando apenas um dígito dos números de cada vez.

Por exemplo, nós sabemos que `7xy` é menor do que `3wz` sem precisar saber quem são `x,y,w,z`.

Ordenando a lista acima de acordo com o primeiro dígito, nós obtemos

050	110	128	321	346	329	323	488	487	486	555	551	786

A seguir, a lista é reordenada de acordo com o segundo dígito.

Mas, é preciso cuidado para não desfazer o trabalho anterior!

Isto é, os números serão ordenados apenas nas faixas correspondentes ao mesmo primeiro dígito.

050	110	128	321	346	329	323	488	487	486	555	551	786

↓

050	110	128	321	329	323	346	488	487	486	555	551	786

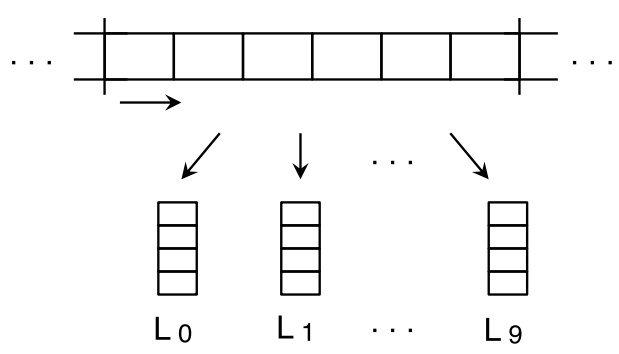
Finalmente, ordenando mais uma vez a lista de acordo com o terceiro dígito (tomando cuidado para não desfazer o trabalho anterior), nós obtemos

050	110	128	321	323	329	346	486	487	488	551	555	786

Qual o tempo de execução desse algoritmo?

A ordenação baseada em dígitos é um procedimento muito rápido.

Basta percorrer a lista uma vez da esquerda para a direita (ou a faixa que se quer ordenar), movendo os números para listas auxiliares $\mathtt{L}_0, \ldots, \mathtt{L}_9$, de acordo com o dígito correspondente



A seguir, basta percorrer as listas $\mathtt{L}_0, \ldots, \mathtt{L}_9$ (nessa ordem), movendo os números de volta para a lista (ou a faixa).

Utilizando esse procedimento, a ordenação da lista de acordo com um dígito (ou de todas as faixas separadamente, de acordo com esse dígito) leva tempo $O(n)$.

Logo, assumindo que o maior número da lista possui k dígitos, o algoritmo radix executa em tempo $O(kn)$.

Isto é, em qualquer situação em que $k < \log n$, o algoritmo radix pode ser uma boa opção.

Mas, ainda há um pequeno inconveniente no algoritmo que nós acabamos de apresentar:

- manter as faixas de valores já ordenadas pelos dígitos anteriores pode ser confuso e difícil de implementar na prática

Isso nos dá a oportunidade de apresentar uma ideia legal!

Suponha que, ao invés de ordenar a lista a partir do dígito mais significativo (como fizemos acima), nós começamos com o dígito menos significativo.

Aplicando essa ideia ao exemplo acima, nós obtemos o seguinte

321	786	110	050	488	346	555	487	128	551	329	323	486

110	050	321	551	323	555	786	346	486	487	488	128	329

110	321	323	128	329	346	050	551	555	786	486	487	488

050	110	128	321	323	329	346	486	487	488	551	555	786

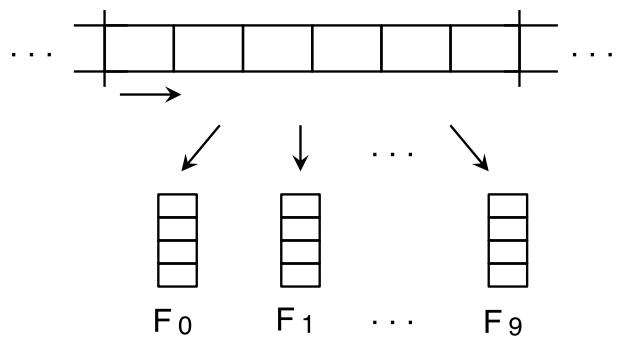
A coisa funciona!

É claro, aqui também é preciso cuidado para que cada ordenação não desfaça o trabalho realizado pelas anteriores.

Mas, nesse caso, o procedimento é bem mais simples

- sempre que houver empate entre dois elementos durante a ordenação por um certo dígito, a ordem entre os dois elementos produzida pelas ordenações anteriores deve ser preservada

Uma maneira simples de garantir isso consiste em armazenar os elementos em filas auxiliares $\mathtt{F}_0, \ldots, \mathtt{F}_9$ durante o processo de varredura.



Finalmente, ao invés de trabalhar com os dígitos $0, \ldots, 9$ da base decimal, nós podemos trabalhar com os bits `0` e `1` da base binária.

Fazendo isso, nós só precisamos de duas filas auxiliares: $\mathtt{F}_0$ e $\mathtt{F}_1$.

Abaixo nós temos o pseudo-código do algoritmo de ordenação radix que utiliza essa ideia.

```
Procedimento ord-Radix ( V[1..n] )
 {
     F0,F1 ←  filas auxiliares vazias
     L ←  número de bits no maior elemento de V
     Para k ← 1 Até L
      {
        Para j ← 1 Até n
         {
           b ← getBit ( V[j], k )
           Se ( b = 0 )   insere V[j] em F0
           Senão          insere V[j] em F1
         }
        j ← 1
        Enquanto ( F0 ≠ vazio ) {   V[j] ← getFirst(F0);   j++   }
        Enquanto ( F1 ≠ vazio ) {   V[j] ← getFirst(F1);   j++   }
      }
 }
```