ALGORITMOS DE ORDENAÇÃO NÃO COMPARATIVOS

UNIVERSIDADE DE SÃO PAULO
INTRODUÇÃO A CIÊNCIA DA COMPUTAÇÃO II

ICMC - USP
17/11/2011

Ricardo Cerri

Algoritmos não Comparativos

- Counting Sort
- Pigeonhole Sort
- Bucket Sort
- Radix Sort

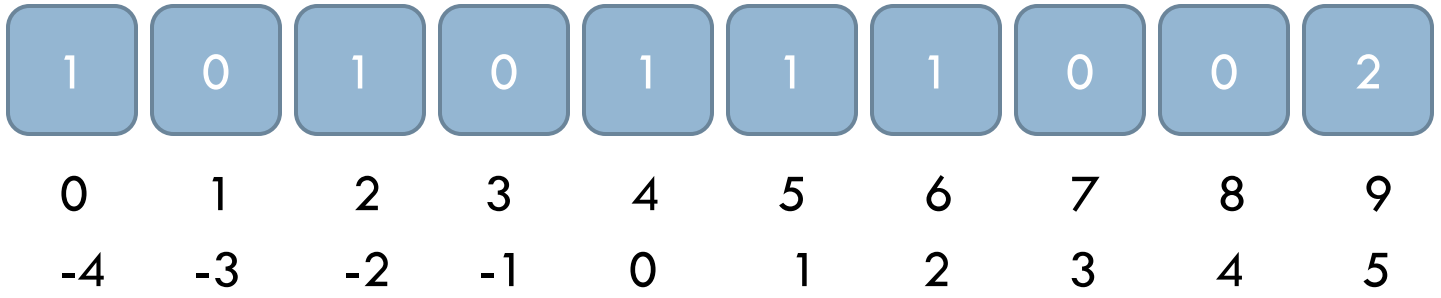
Counting Sort

- Ordena *n* elementos de acordo com *k* chaves
- Para cada chave k_i, conta o número *K* de elementos que têm aquela chave
- Armazena o valor *K* na posição k_i um vetor de *k* posições
- Percorre o vetor colocando os elementos em ordem no vetor original

Counting Sort

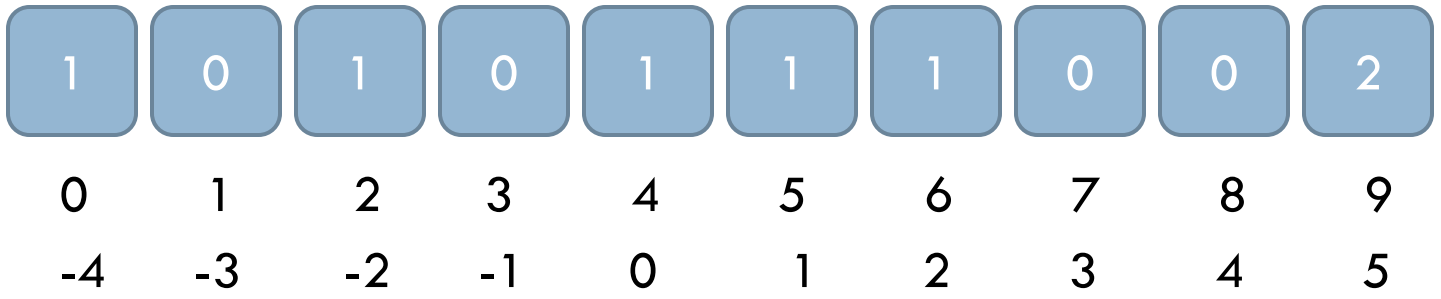
- Exemplo: ordenar vetor v = [-2 -4 2 0 5 5 1]
  - 7 elementos
  - [5 - (-4)] + 1 = 10 possíveis chaves

- Cria vetor de 10 posições e armazena número de vezes que cada elemento aparece
  - `contagem[v[i] - min]++`



Counting Sort

- Percorre vetor com a contagem dos elementos, ordenando os elementos do vetor original de acordo com suas chaves
  - `v = [-2 -4 2 0 5 5 1]`



  - `v[j] = min + i`

Counting Sort

```
//Create a vector to store counts
sizeCount = (max-min)+1;
counting = (int *)malloc(sizeCount * sizeof(int));
for(i=0;i<sizeCount;i++){
        counting[i] = 0;
}

//Here we count the elements
for(i=0;i<size;i++){
        counting[v[i]-min]++;
}

//Iterate over the vector to put elements of the original list in order
j = 0;
for(i=0;i<sizeCount;i++){
        if(counting[i] > 0){
                for(k=0; k<counting[i]; k++){
                        v[j] = min+i;
                        j++;
                }
        }
}
```

Counting Sort

- Complexidade de tempo
  - $O(n + k)$
- Complexidade de espaço
  - $O(n + k)$
- Adequado quando o número de chaves não é muito maior do que o número de elementos
- É um algoritmo de ordenação estável

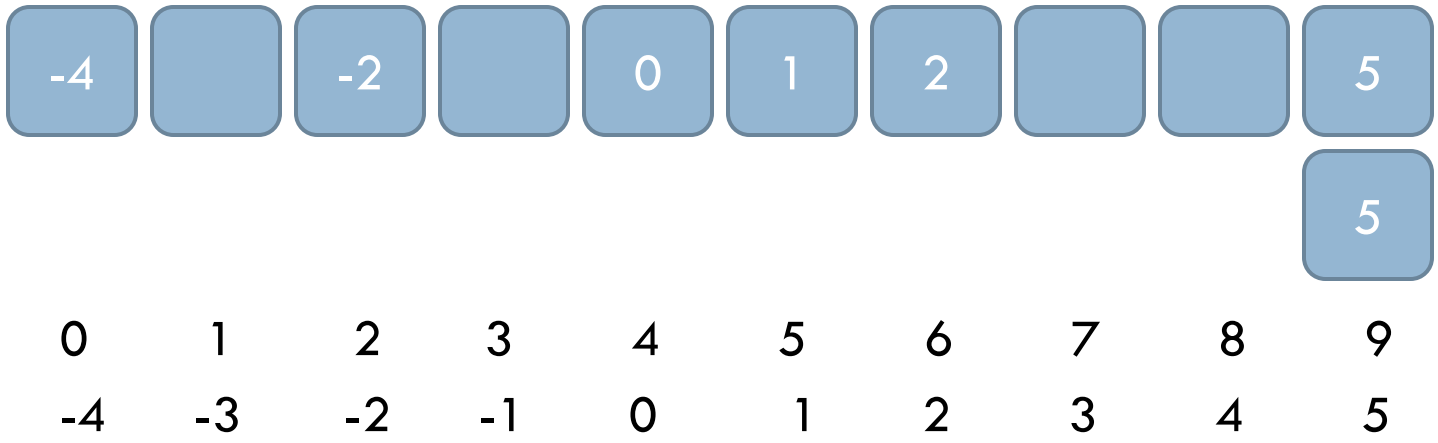
Pigeonhole Sort

- Mesma ideia do Counting Sort, mas ao invés de armazenar a contagem de elementos, armazena os elementos
  - Utiliza o *Pigeonhole Principle*

Se n elementos devem ser colocados em k buracos de pombas, com $n > k$, então pelo menos um buraco conterá mais de um elemento.

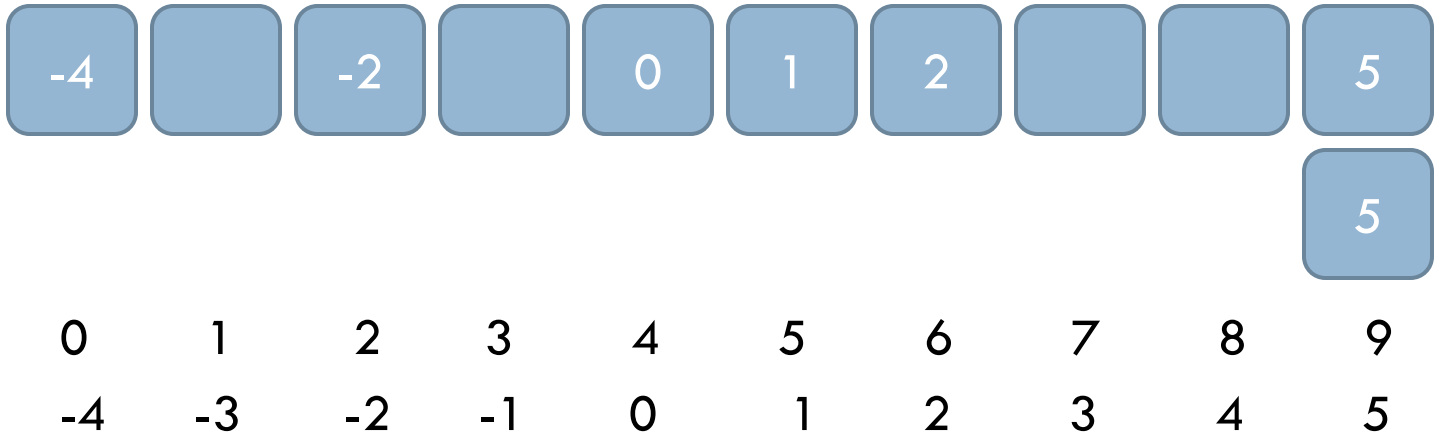
Pigeonhole Sort

- Percorre vetor original colocando os elementos em buracos de pombas
  - `v = [-2 -4 2 0 5 5 1]`
  - `pigeonhole[v[i]-min][j] = v[i]`



Pigeonhole Sort

- Percorre o vetor auxiliar colocando os elementos de volta no vetor original ordenados



- `v[k] = pigeonhole[i][j]`

Pigeonhole Sort

```
//Create the pigeonhole
sizePG = (max-min)+2;
pigeonhole = (int **)malloc((sizePG-1) * sizeof(int *));
for(i=0;i<(sizePG-1);i++){
        pigeonhole[i] = (int *)malloc(sizePG * sizeof(int));
}
for(i=0;i<(sizePG-1);i++){
        for(j=0;j<sizePG;j++){
                pigeonhole[i][j] = 0;
        }
}

//Put the elements in the pigeonholes
for(i=0;i<size;i++){
        //Put elements in the pigeonhole
        for(j=1;j<=pigeonhole[v[i]-min][0];j++){
        }
        pigeonhole[v[i]-min][j] = v[i];
        pigeonhole[v[i]-min][0]++;
}
```

Pigeonhole Sort

```
//Iterate over the pigeonhole to put elements back to the original list
k = 0;
for(i=0;i<(sizePG-1);i++){
        if(pigeonhole[i][0] > 0){
                for(j=1;j<=pigeonhole[i][0];j++){
                        v[k] = pigeonhole[i][j];
                        k++;
                }
        }
}
```

Pigeonhole Sort

- Complexidade de tempo
  - $O(n + k)$
- Complexidade de espaço
  - $O(n \times k)$
- Adequado quando o número de chaves e o número de elementos é aproximadamente o mesmo
- É um algoritmo de ordenação estável

Bucket Sort

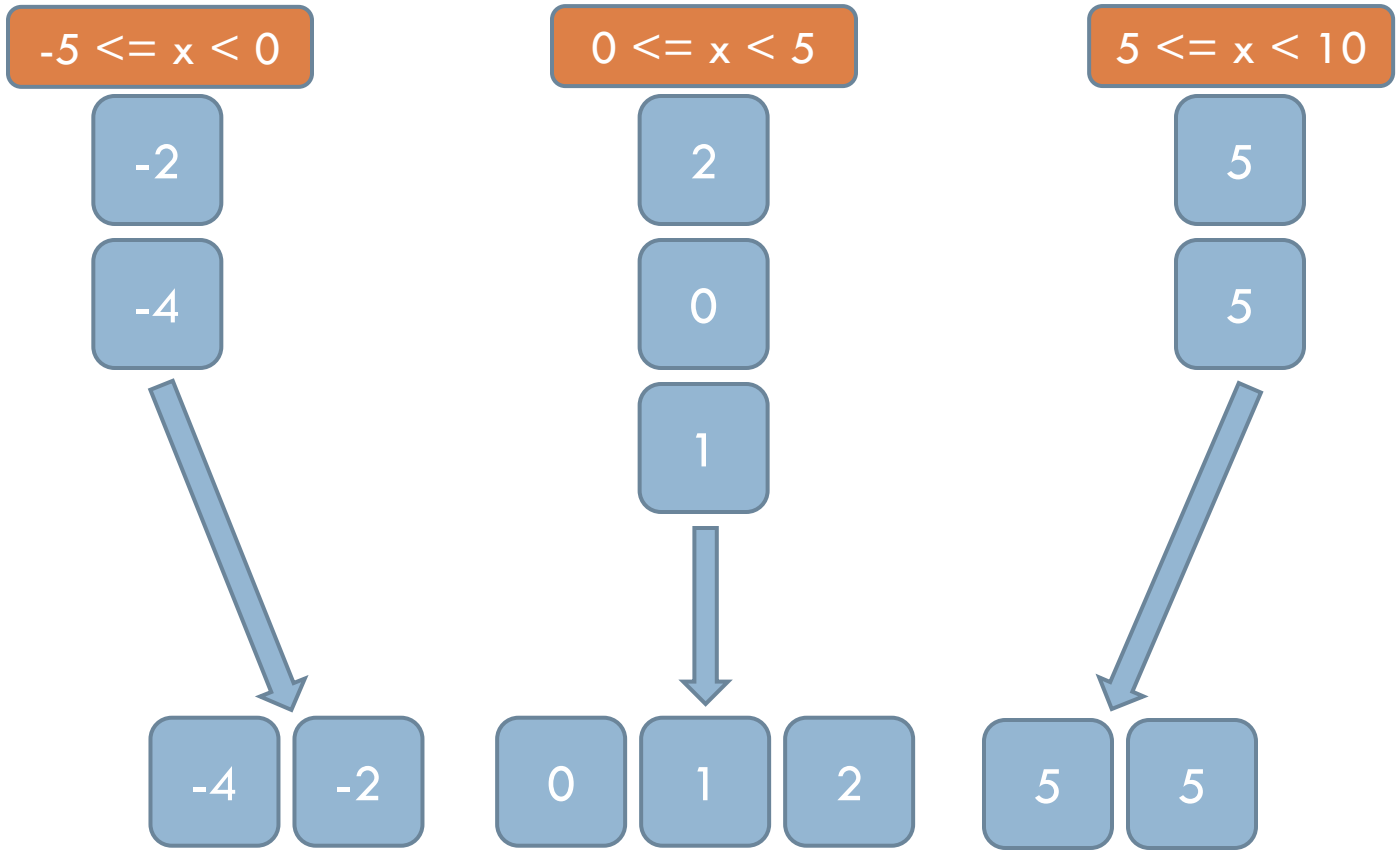
- Divide o vetor de entrada em *buckets* (baldes), sendo que cada balde conterá uma quantidade de elementos

- Cada balde é ordenado individualmente utilizando outro algoritmo de ordenação (Inserção, Seleção) ou aplicando Bucket Sort recursivamente

- Percorre os baldes em ordem e coloca os elementos de volta no vetor original

Bucket Sort

- `v = [-2 -4 2 0 5 5 1]`



Bucket Sort

```
//Create the buckets
buckets = (int **)malloc(numBuckets * sizeof(int *));
for(i=0;i<numBuckets;i++){
        buckets[i] = (int *)malloc(sizeBuckets * sizeof(int));
}
for(i=0;i<numBuckets;i++){
        for(j=0;j<sizeBuckets;j++){
                buckets[i][j] = 0;
        }
}
//Put the elements in the buckets
for(i=0;i<size;i++){
        k=0;
        for(j=-10;j<=10;j++){
                if(v[i] >= j*10){
                        k++;
                }
                else{
                        break;
                }
        }
        //Put elements in the bucket
        for(l=1;l<=buckets[k][0];l++){
        }
        buckets[k][l] = v[i];
        buckets[k][0]++;
}
```

Bucket Sort

```
//Sort the buckets with a sorting algorithm
for(i=0;i<numBuckets;i++){
        if(buckets[i][0] > 0){
                //Put the elements in a vector to be sorted
                sortedBucked = (int *)malloc(buckets[i][0] * sizeof(int));
                for(j=1;j<=buckets[i][0];j++){
                        sortedBucked[j-1] = buckets[i][j];
                }
                //Sort the elements of the bucket
                selectSort(sortedBucked,buckets[i][0]);
                //Put the elements back to the bucket
                for(j=1;j<=buckets[i][0];j++){
                        buckets[i][j] = sortedBucked[j-1];
                }
                free(sortedBucked);
        }
}
//Iterate over the buckets to put elements back to the original list
k = 0;
for(i=0;i<numBuckets;i++){
        if(buckets[i][0] > 0){
                for(j=1;j<=buckets[i][0];j++){
                        v[k] = buckets[i][j];
                        k++;
                }
        }
}
```

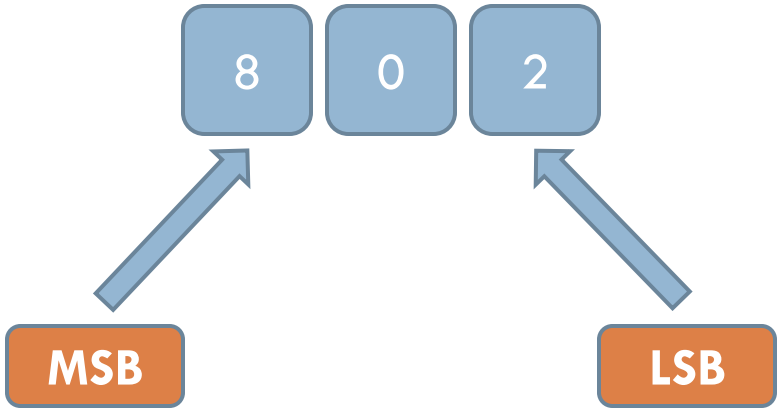
Bucket Sort

- Complexidade de tempo
  - $O(n^2)$
- Complexidade de espaço
  - $O(n)$
- Bom quando o número de chaves é pequeno e há em média poucos elementos por balde
- Sua estabilidade depende do algoritmo de ordenação utilizado para ordenar os baldes

Radix Sort

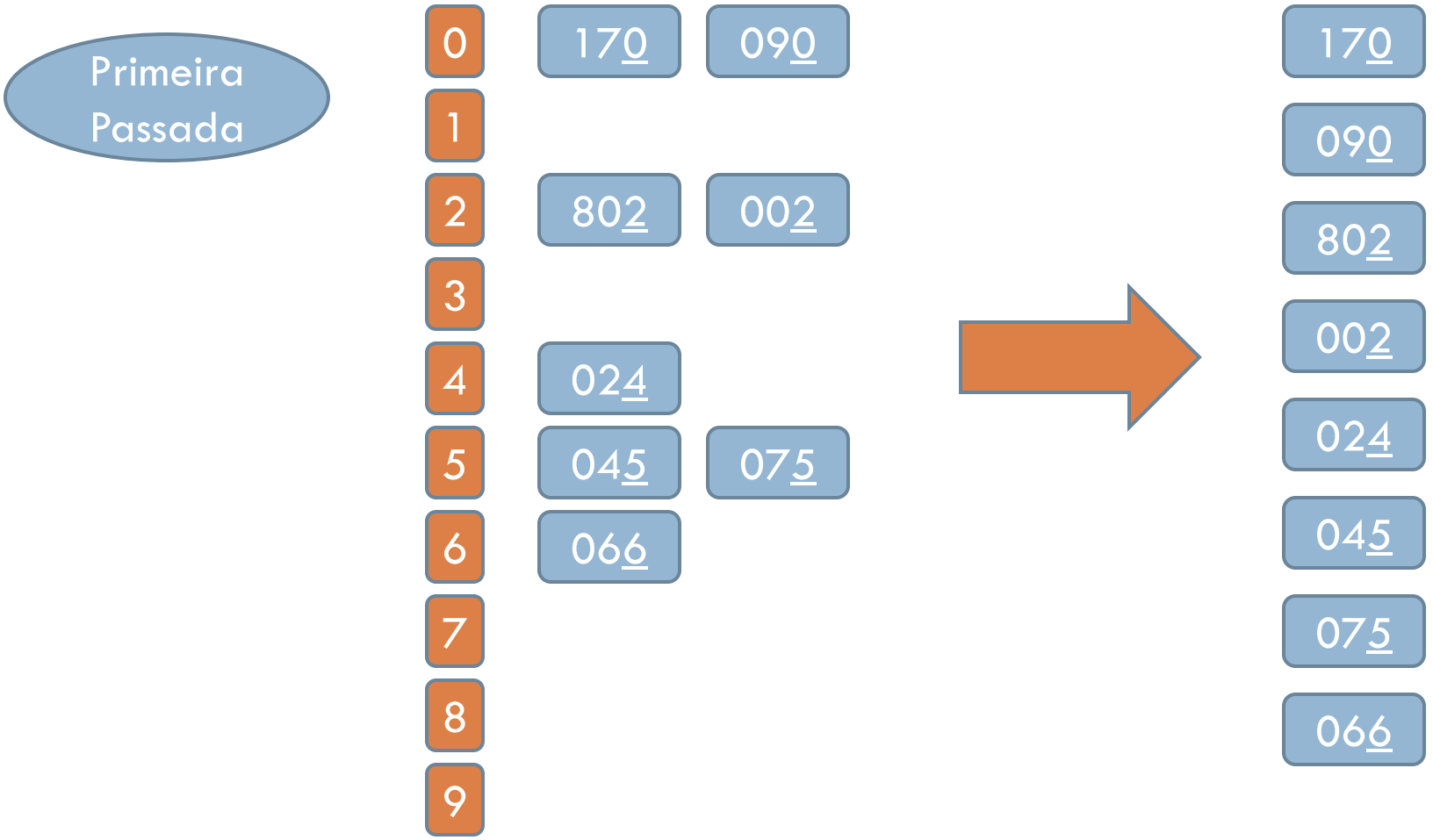
- Ordena o vetor de entrada agrupando as chaves de acordo com as posições e valores dos dígitos que as compõem.

- As chaves podem ser agrupadas do dígito menos significativo para o mais significativo, ou o contrário



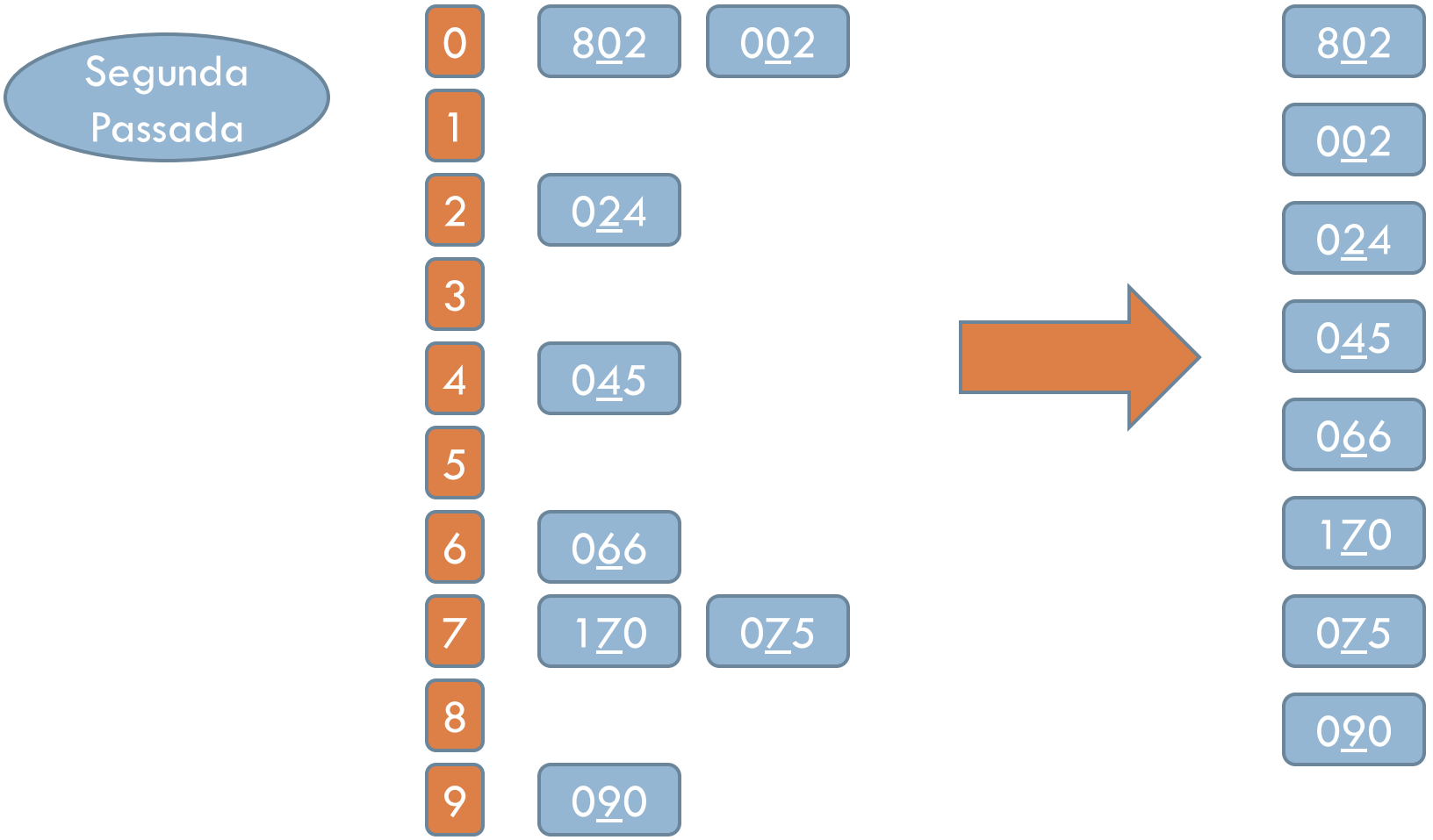
Radix Sort

- v = [170 45 75 90 802 24 2 66]



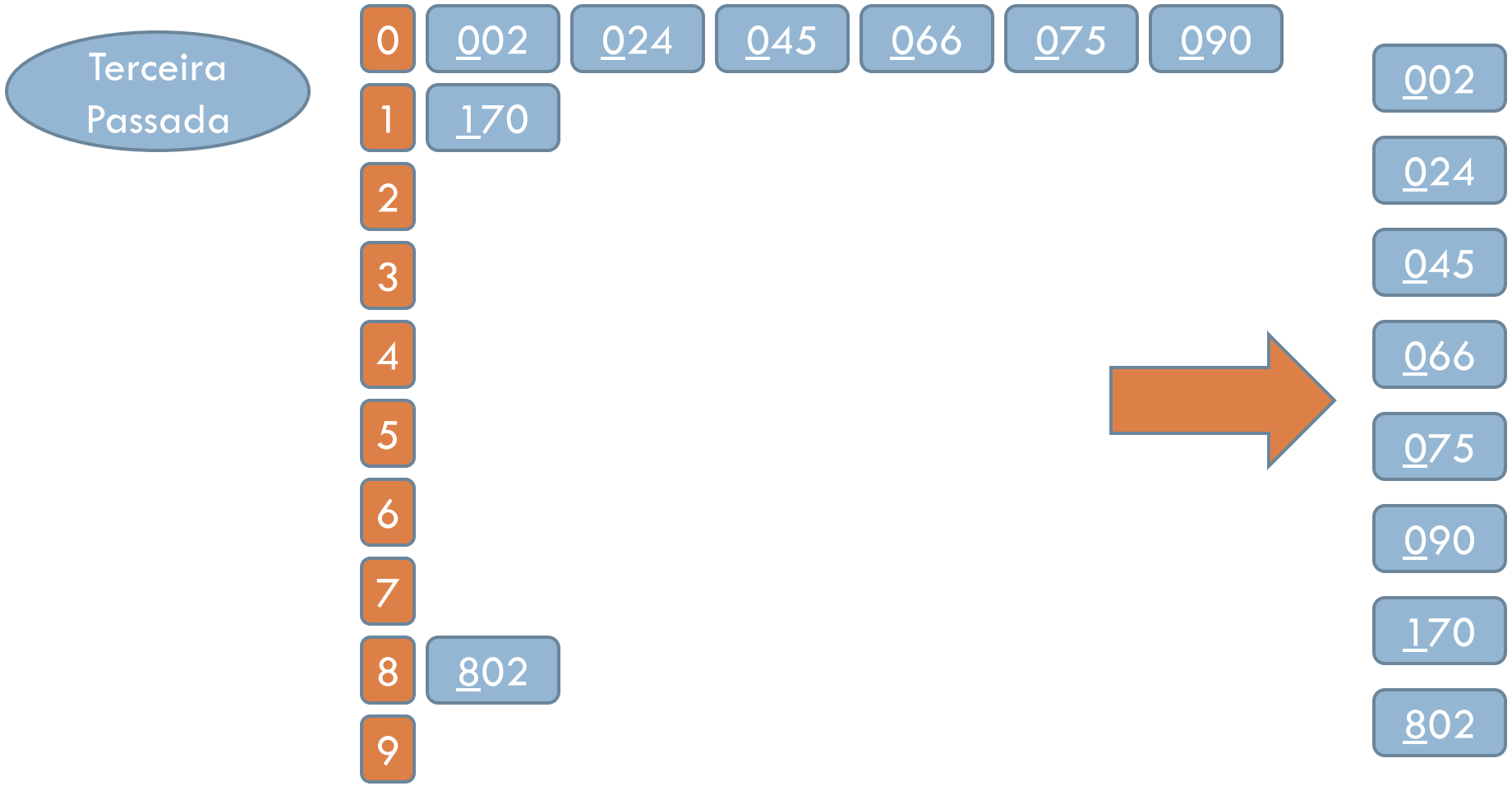
Radix Sort

- v = [170 90 802 2 24 45 75 66]



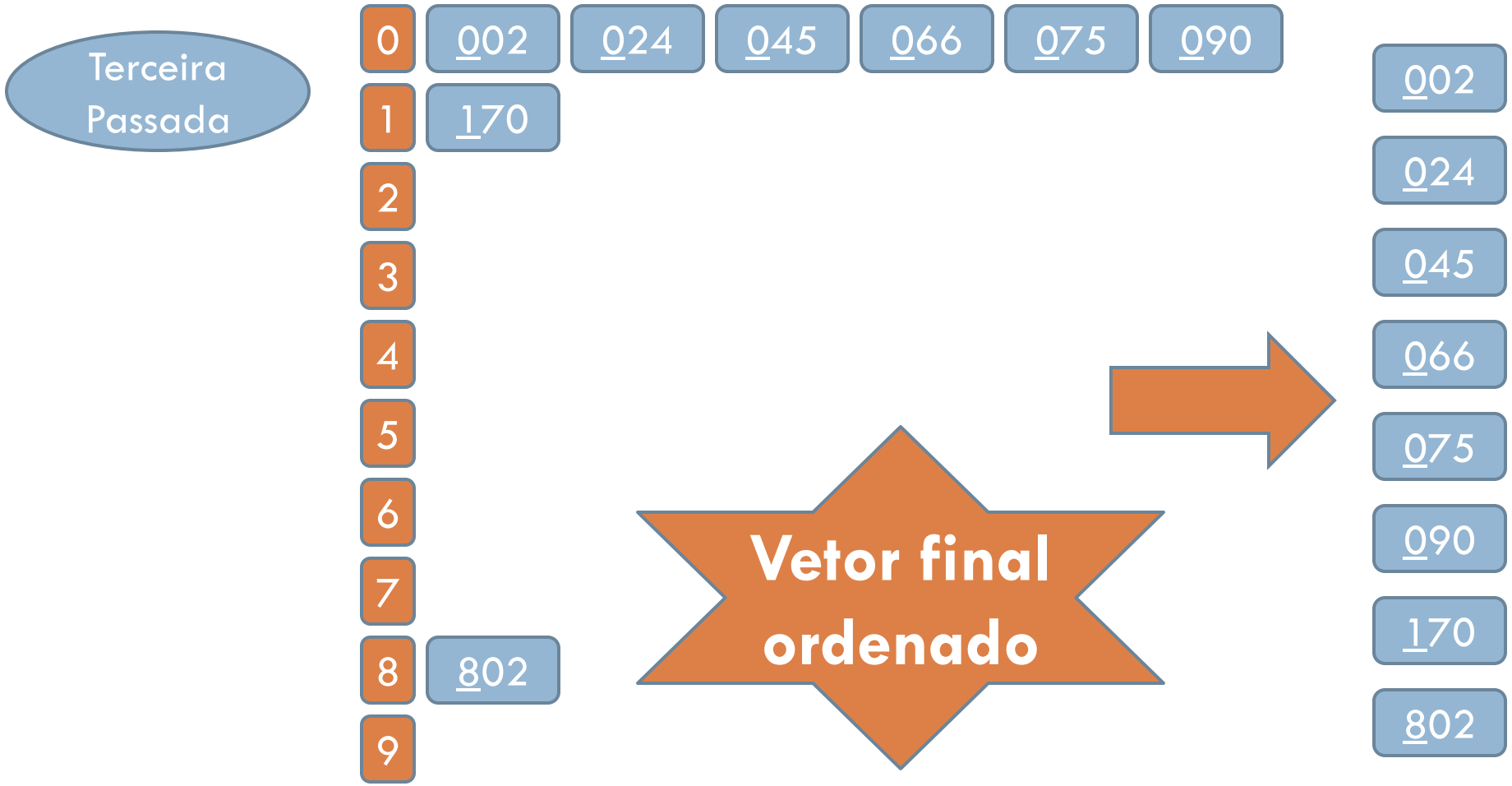
Radix Sort

- v = [802 2 24 45 66 170 75 90]



Radix Sort

- v = [170 45 75 90 802 24 2 66]



Radix Sort

```
//Get the max element to
//Necessary to count the number of digits
int max = getMax(size,v);

//Get the number of digits
int numDigits = getNumberDigits(max);

//Create 10 buckets, one for each possible digit of a number
buckets = (int **)malloc(10 * sizeof(int *));
for(i=0;i<10;i++){
        buckets[i] = (int *)malloc(size * sizeof(int));
}
for(i=0;i<10;i++){
        for(j=0;j<size;j++){
                buckets[i][j] = 0;
        }
}
```

Radix Sort

```
//Put elements in the buckets according to its digits
//from the LSD to the MSD
for(i=0;i<numDigits;i++){
        for(j=0;j<size;j++){
                //Returns the ith digit of the number
                digit = returnDigit(v[j],i);
                //Put elements in the bucket
                for(k=1;k<=buckets[digit][0];k++){
                }
                buckets[digit][k] = v[j];
                buckets[digit][0]++;
        }
        //Put the elements back into the original array
        k = 0;
        for(l=0;l<10;l++){
                if(buckets[l][0] > 0){
                        for(j=1;j<=buckets[l][0];j++){
                                v[k] = buckets[l][j];
                                k++;
                        }
                }
        }
        //Set the buckets empty
        for(k=0;k<10;k++){
                for(l=0;l<size;l++){
                        buckets[k][l] = 0;
                }
        }
}
```

Radix Sort

- Complexidade de tempo
  - $O(n \times m)$: m é a quantidade média de dígitos
- Complexidade de espaço
  - $O(n \times m)$
- É um algoritmo de ordenação estável

Obrigado