# Algoritmos de Ordenação

- Ordenação é uma das operações mais usuais em listas lineares
  - Consequentemente, uma implementação ineficiente pode afetar diretamente o desempenho geral da aplicação!
- Por ser um problema bem estudado, existem diversas abordagens disponíveis (veremos as formas clássicas)

# Algoritmos de Ordenação

- Para realizar ordenação, vamos nos apoiar nas seguintes operações básicas:
  - Comparação de valores entre elementos
  - Comparação de valores entre dígitos da representação numérica dos elementos
  - Troca de dois elementos de posição

- Mas outras abordagens existem! (Veja um exemplo no próximo slide)

# Algoritmos de Ordenação

Discrete Mathematics 27 (1979) 47–57.


## BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES
*Microsoft, Albuquerque, New Mexico*

Christos H. PAPADIMITRIOU*†
*Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.*

Received 18 January 1978
Revised 28 August 1978

For a permutation $\sigma$ of the integers from 1 to $n$, let $f(\sigma)$ be the smallest number of prefix reversals that will transform $\sigma$ to the identity permutation, and let $f(n)$ be the largest such $f(\sigma)$ for all $\sigma$ in (the symmetric group) $S_n$. We show that $f(n) \leqslant (5n+5)/3$, and that $f(n) \geqslant 17n/16$ for $n$ a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2-1 \leqslant g(n) \leqslant 2n+3$.

### 1. Introduction

We introduce our problem by the following quotation from [1]

# Algoritmos de Ordenação

Discrete Mathematics 27 (1979) 47–57.


## BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES
*Microsoft, Albuquerque, New Mexico*

Christos H. PAPADIMITRIOU*†
*Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.*

Received 18 January 1978
Revised 28 August 1978

For a permutation $\sigma$ of the integers from 1 to $n$, let $f(\sigma)$ be the smallest number of prefix reversals that will transform $\sigma$ to the identity permutation, and let $f(n)$ be the largest such $f(\sigma)$ for all $\sigma$ in (the symmetric group) $S_n$. We show that $f(n) \leqslant (5n+5)/3$, and that $f(n) \geqslant 17n/16$ for $n$ a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2-1 \leqslant g(n) \leqslant 2n+3$.

### 1. Introduction

We introduce our problem by the following quotation from [1]

# Algoritmos de Ordenação

Discrete Mathematics 27 (1979) 47–57.


## BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES
*Microsoft, Albuquerque, New Mexico*

Christos H. PAPADIMITRIOU*†
*Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.*

Received 18 January 1978
Revised 28 August 1978

For a permutation $\sigma$ of the integers from 1 to $n$, let $f(\sigma)$ be the smallest number of prefix reversals that will transform $\sigma$ to the identity permutation, and let $f(n)$ be the largest such $f(\sigma)$ for all $\sigma$ in (the symmetric group) $S_n$. We show that $f(n) \leqslant (5n+5)/3$, and that $f(n) \geqslant 17n/16$ for $n$ a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2-1 \leqslant g(n) \leqslant 2n+3$.

### 1. Introduction

We introduce our problem by the following quotation from [1]

# Algoritmos de Ordenação

- Algoritmos de Ordenação Clássicos:
  - **InsertionSort** (Ordenação por Inserção)
  - **SelectionSort** (Ordenação por Seleção)
  - **BubbleSort** (Ordenação por Bolha)
  - **MergeSort** (Ordenação por Intercalação)
  - **QuickSort** (Ordenação Rápida)
  - **TreeSort**
  - **HeapSort** (Ordenação em Heap)
  - **BucketSort** ou **BinSort / CountingSort**
  - **RadixSort**

# Algoritmos de Ordenação

- Para todos os algoritmos a seguir, considere o problema de ordenar um vetor B com N elementos

- Analisaremos os algoritmos quanto a:
  - **complexidade de tempo** e **de espaço**
    - **ordenação in loco (in-place)** ⇔ memória auxiliar constante
  - **estabilidade** (se x = y e x vem antes de y na entrada, então x vem antes de y na saída)

# InsertionSort

# Algoritmos de Ordenação

- **InsertionSort:**
  - Método da ordenação de cartas de baralho:
    - Ordene os N-1 primeiros elementos de B
    - Encontre a posição onde B[N] deveria estar entre os N-1 primeiros elementos e o insira nesta posição

- Estável, In loco

# Algoritmos de Ordenação

```
//Recursivo
procedimento InsertionSort(ref B[], N: Inteiro)
//Assume: |B| ≥ N
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
   var j: Inteiro
   se (N > 1) então
      InsertionSort(B, N - 1)
      j ← N-1
      enquanto j > 0 E B[j] > B[j+1] faça
         B[j], B[j+1], j ← B[j+1], B[j], j-1
```

# Algoritmos de Ordenação

```
//Iterativo
procedimento InsertionSort(ref B[], N: Inteiro)
//Assume: |B| ≥ N
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
   var i, j: Inteiro
   para i ← 2 até N faça
      j ← i - 1
      enquanto j > 0 E B[j] > B[j+1] faça
         B[j], B[j+1], j ← B[j+1], B[j], j-1
```

**Tempo:**
Pior Caso: $\theta(N^2)$
Melhor Caso: $\theta(N)$

# SelectionSort

# Algoritmos de Ordenação

- **SelectionSort:**
  - Encontre uma permutação de B na qual B[N] passe a ser o elemento máximo
    - Encontre a posição k do elemento máximo de B
    - Permute B[k] e B[N]
  - Ordene os N-1 primeiros elementos de B

- Estável,

  In loco (se k for a última posição onde há o maior valor)

# Algoritmos de Ordenação

```
//Recursivo
procedimento SelectionSort(ref B[], N: Inteiro)
//Assume: |B| ≥ N
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
   var pmax, j: Inteiro
   se N > 1 então
      pmax ← 1
      para j ← 2 até N faça
         se B[pmax] ≤ B[j] então
            pmax ← j
      B[N], B[pmax] ← B[pmax], B[N]
      SelectionSort(B, N - 1)
```

# Algoritmos de Ordenação

```
//Iterativo
procedimento SelectionSort(ref B[], N: Inteiro)
//Assume: |B| ≥ N
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
   var pmax, i, j: Inteiro
   para i ← N até 2 passo -1 faça
      pmax ← 1
      para j ← 2 até i faça
         se B[pmax] ≤ B[j] então
            pmax ← j
      B[i], B[pmax] ← B[pmax], B[i]
```

**Tempo:**
$\theta(N^2)$

# BubbleSort

# Algoritmos de Ordenação

- **BubbleSort:**
  - Encontre uma permutação de B na qual B[N] passe a ser o elemento máximo
    - Compare B[1] com B[2], e eventualmente permute-os para que B[2] tenha o maior valor. Repita o processo com B[2] e B[3], B[3] e B[4], …, B[N-1] e B[N]. Ao final, B[N] será o maior valor
  - Ordene os N-1 primeiros elementos de B
- Estável, In loco

# Algoritmos de Ordenação

```
//Recursivo
procedimento BubbleSort(ref B[], N: Inteiro)
//Assume: |B| ≥ N
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
   var i, j: Inteiro
   se N > 1 então
      para j ← 1 até N-1 faça
         se B[j] > B[j+1] então
            B[j], B[j+1] ← B[j+1], B[j]
      BubbleSort(B, N - 1)
```

# Algoritmos de Ordenação

```
//Iterativo
procedimento BubbleSort(ref B[], N: Inteiro)
//Assume: |B| ≥ N
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
   var i, j: Inteiro
   para i ← N até 2 passo -1 faça
      para j ← 1 até i-1 faça
         se B[j] > B[j+1] então
            B[j], B[j+1] ← B[j+1], B[j]
```

**Tempo:**
$\theta(N^2)$

# MergeSort

# Algoritmos de Ordenação

- **MergeSort**
  - Importante método de ordenação, descoberto por John von Neumann, em 1945
  - **Estratégia**: Divida o vetor em duas partes. Ordene cada parte. Faça a operação de mescla das duas partes ordenadas
    - Mescla do vetor A ordenado com B ordenado: o menor elemento dentre todos aqueles de A e de B ou é o 1º elemento de A ou o 1º elemento de B. Remova tal elemento e o coloque na 1ª posição livre de um outro vetor C. Repita a operação até que todos os elementos estejam em C (que então conterá a ordenação dos elementos de A e B).
- Estável, **não é** in loco

# Algoritmos de Ordenação

```
procedimento MergeSort(ref B[], inicio, fim: Inteiro)
//Assume: |B| ≥ fim ≥ inicio
//Garante: B[i] ≤ B[i+1] ∀ inicio ≤ i < fim
    var m: Inteiro
    se inicio < fim então
        m ← (inicio + fim) div 2
        Mergesort(B, inicio, m)
        Mergesort(B, m+1, fim)
        Merge(B, inicio, m, fim)

ler(N, B[1..N])
Mergesort(B, 1, N)
escrever(B[1..N])
```

# Algoritmos de Ordenação

```
procedimento Merge(ref B[], inicio, limite, fim: Inteiro)
//Assume:    B[i] ≤ B[i+1] ∀ inicio ≤ i < limite,
             B[i] ≤ B[i+1] ∀ limite + 1 ≤ i < fim
//Garante:   B[i] ≤ B[i+1] ∀ inicio ≤ i < fim
    var C[1..fim-inicio+1], k, i, j: Inteiro
    i, j ← inicio, limite + 1
    para k ← 1 até fim-inicio+1 faça
        se (j > fim) ou (i ≤ limite E B[i] ≤ B[j]) então
            C[k], i ← B[i], i + 1
        senão
            C[k], j ← B[j], j + 1
    B[inicio..fim] ← C[1..fim-inicio+1]
```

**Tempo:**
θ(fim - inicio)

**Espaço Auxiliar:**
θ(fim - inicio)

# Algoritmos de Ordenação

- Análise de Complexidade:

  T(N): número de passos para ordenar N elementos

  $T(N) = \theta(1)$, se $N \leq 1$. Caso contrário,

  $$T(N) = \theta(1) + 2 \cdot T(N/2) + \theta(N) =$$
  $$= \theta(N) + 2 \cdot T(N/2) =$$
  $$= \theta(N) + 2 \cdot (\theta(N/2) + 2 \cdot T(N/2^2)) =$$
  $$= 2 \cdot \theta(N) + 2^2 \cdot T(N/2^2)) =$$
  $$= \ldots = i \cdot \theta(N) + 2^i \cdot T(N/2^i) =$$
  $$= \lg N \cdot \theta(N) + 2^{\lg N} T(1) =$$
  $$= \theta(N \lg N) + N \cdot \theta(1) =$$
  $$= \theta(N \lg N)$$

# Algoritmos de Ordenação

```
procedimento MergeSort(ref B[], inicio, fim: Inteiro)
//Assume: |B| ≥ fim ≥ inicio
//Garante: B[i] ≤ B[i+1] ∀ inicio ≤ i < fim
    var m: Inteiro
    se inicio < fim então
        m ← (inicio + fim) div 2
        Mergesort(B, inicio, m)
        Mergesort(B, m+1, fim)
        Merge(B, inicio, m, fim)

ler(N, B[1..N])
Mergesort(B, 1, N)
escrever(B[1..N])
```

**Tempo:**
θ(N log N)

**Espaço Auxiliar:**
θ(N)

# QuickSort

# Algoritmos de Ordenação

- **Quicksort**
  - Importante método de ordenação, descoberto por Hoare, em 1962
  - **Estratégia**: se B é um vetor que pode ser dividido em duas partes de tal maneira que cada elemento da primeira parte é menor ou igual a cada elemento da segunda, ordenar B consiste de ordenar individualmente cada uma das duas partes
  - Particionar tal vetor B como acima é fácil?
- Não-estável, in loco

# Algoritmos de Ordenação

```
procedimento QuickSort(ref B[], inicio, fim: Inteiro)
//Assume: |B| ≥ fim ≥ inicio
//Garante: B[i] ≤ B[i+1] ∀ inicio ≤ i < fim
    var pivo, limmen, limmai: Inteiro
    se inicio < fim então
        pivo ← "escolher um elemento de B[inicio..fim]"
        Particionar(B, inicio, fim, pivo, limmen, limmai)
        //limmen, limmai são variáveis por referência; o retorno delas é tal que:
        //(i) B[limmai+1..fim] ≥ pivo; (ii) B[inicio..limmen-1] ≤ pivo;
        //(iii) limmai < limmen
        QuickSort(B, inicio, limmai)
        QuickSort(B, limmen, fim)

ler(N, B[1..N])
QuickSort(B, 1, N)
escrever(B[1..N])
```

# Algoritmos de Ordenação

```
procedimento Particionar(ref B[], inicio, fim, pivo,
                         ref i, ref j: Inteiro)
//Assume: |B| ≥ fim ≥ pivo ≥ inicio
//Garante:   j < i, B[k] ≤ pivo ∀ inicio ≤ k < i
               B[k] ≥ pivo ∀ j < k ≤ fim
   i, j ← inicio, fim
   enquanto i ≤ j faça
      enquanto B[i] < pivo faça i ← i+1
      enquanto B[j] > pivo faça j ← j-1
      se i ≤ j faça
         B[i], B[j], i, j ← B[j], B[i], i+1, j-1
```

**Tempo:**
θ(fim - inicio)

# Algoritmos de Ordenação

- Análise de Complexidade

  (**Melhor Caso = Máximo Balanceamento**):

  T(N): número de passos para ordenar N elementos

  $T(N) = \theta(1)$, se $N \leq 1$. Caso contrário,

  $T(N) \geq \theta(1) + \theta(N) + 2 \cdot T(N/2)$

  $\geq \theta(N) + 2 \cdot T(N/2)$

  (idem MergeSort)

  $\geq \theta(N \lg N)$

  $\therefore T(N) = \Omega(N \lg N)$

# Algoritmos de Ordenação

- Análise de Complexidade

  (**Pior Caso = Mínimo Balanceamento**):

$$T(N) \leq \theta(N) + T(1) + T(N-1) = \theta(N) + T(N-1)$$
$$\leq 2 \cdot \theta(N) + T(N-2)$$
$$\leq \ldots \leq i \cdot \theta(N) + T(N-i)$$
$$\leq (N-1) \cdot \theta(N) + T(1) = \theta(N^2)$$
$$\therefore T(N) = O(N^2)$$

# Algoritmos de Ordenação

```
procedimento QuickSort(ref B[], inicio, fim: Inteiro)
//Assume: |B| ≥ fim ≥ inicio
//Garante: B[i] ≤ B[i+1] ∀ inicio ≤ i < fim
    var pivo, limmen, limmai: Inteiro
    se inicio < fim então
        pivo ← "escolher um elemento de B[inicio..fim]"
        Particionar(B, inicio, fim, pivo, limmen, limmai)
        //limmen, limmai são variáveis por referência; o retorno delas é tal que:
        //(i) B[limmai+1..fim] ≥ pivo; (ii) B[inicio..limmen-1] ≤ pivo;
        //(iii) limmai < limmen
        QuickSort(B, inicio, limmai)
        QuickSort(B, limmen, fim)

ler(N, B[1..N])
QuickSort(B, 1, N)
escrever(B[1..N])
```

**Tempo:**
Pior Caso: $\theta(N^2)$
Melhor Caso: $\theta(N \lg N)$

# TreeSort

# Algoritmos de Ordenação

- **TreeSort:**
  - Crie uma Árvore Balanceada com os elementos do vetor B como chaves
  - Faça uma percurso in-ordem. Se um nó é o k-ésimo nó visitado, então sua chave deve ser atribuída a B[k]

- Não-estável, não é in loco

# Algoritmos de Ordenação

```
procedimento TreeSort(ref B[], N: Inteiro)
//Assume: |B| ≥ N
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
   var T: ArvoreBalanceada<Inteiro, Inteiro>
       i, ProxPos: Inteiro
   Constroi(T)
   para i ← 1 até N faça
      Insere(T, B[i], B[i])
   ProxPos ← 1
   Escreve(T.Raiz, B, ProxPos)
   Destroi(T)
```

**Tempo:**
θ(N log N)

**Espaço Auxiliar:**
θ(N)

# Algoritmos de Ordenação

- Percurso em In-Ordem

```
procedimento Escreve(T: ^No, ref B[]: Inteiro,
                     ref ProxPos: Inteiro)
   se T ≠ NULO então
      Escreve(T^.Esq, B, ProxPos)
      B[ProxPos], ProxPos ← T^.Chave, ProxPos+1
      Escreve(T^.Dir, B, ProxPos)
```

**Tempo:**
$\theta(N)$

# HeapSort

# Algoritmos de Ordenação

- **HeapSort:**
  - Crie uma Fila de Prioridade com os elementos do vetor B, usando MaxHeap (quanto menor o elemento, maior sua prioridade)
  - Remova todas as chaves. A k-ésima chave removida deve ser armazenada em B[N-k+1]

- Não-estável, in loco

# Algoritmos de Ordenação

```
procedimento HeapSort(ref B[], N: Inteiro)
//Assume: |B| ≥ N
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
   var MAX_N ← N
   var L: FilaPrioridade<Inteiro>
   Constroi(L, B, B, N)
   para i ← 1 até N faça
      B[N-i+1] ← Remove(L)
   Destroi(L)
```

**Tempo:**
$\theta(N \lg N)$

# BucketSort ou BinSort

# Algoritmos de Ordenação

- **BucketSort ou BinSort:**
  - Assume-se os elementos com chaves no intervalo [0 , 1)
  - Cria-se N "baldes" (listas lineares), onde o i-ésimo balde (1 ≤ i ≤ N) abrigará os elementos com chaves no intervalo [(i-1)/N, i/N)
  - Distribui-se os elementos entre os baldes conforme as respectivas chaves
  - Para i de 1 a N, ordena-se Bucket[i] com algum algoritmo (InsertionSort, por exemplo)
  - Concatena-se as listas Bucket[i] com i de 1 at N (nesta ordem) de volta a B
- Estável (requer cuidado na implementação; não é o caso do algoritmo dado a seguir), não é in loco

# Algoritmos de Ordenação

```
procedimento BucketSort(ref B[]: Real,
                        N: Inteiro)
//Assume: |B| ≥ N, 0 ≤ B[i] < 1 ∀ 1 ≤ i ≤ N
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
    var    i, prox: Inteiro, x: ^No
    Bucket[1..N]: ListaLinear<Real, Real>
    //alocação encadeada não-ordenada
    para i ← 1 até N faça
        Constroi(Bucket[i])
    para i ← 1 até N faça
        Insere(Bucket(⌊B[i]*N⌋+1), B[i], B[i])
    para i ← 1 até N faça
        InsertionSort(Bucket[i])
```

```
estrutura No:
    Elem: Real,
    Prox: ^No
```

```
prox ← 1
para i ← 1 até N faça
    x ← Bucket[i].Inicio
    enquanto x ≠ NULO faça
        B[prox] ← x^.Chave
        prox ← prox + 1
        x ← x^.Prox
    Destroi(Bucket[i])
```

# Algoritmos de Ordenação

```
procedimento BucketSort(ref B[]: Real,
                        N: Inteiro)
//Assume: |B| ≥ N, 0 ≤ B[i] < 1 ∀ 1 ≤ i ≤ N
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
    var    i, prox: Inteiro, x: ^No
    Bucket[1..N]: ListaLinear<Real, Real>
    //alocação encadeada não-ordenada
    para i ← 1 até N faça
        Constroi(Bucket[i])
    para i ← 1 até N faça
        Insere(Bucket(⌊B[i]*N⌋+1), B[i], B[i])
    para i ← 1 até N faça
        InsertionSort(Bucket[i])
```

**Tempo:**
Caso Médio: θ(N)
(com distribuição uniforme)
Pior Caso: $\theta(N^2)$

**Espaço:**
θ(N)

```
prox ← 1
para i ← 1 até N faça
    x ← Bucket[i].Inicio
    enquanto x ≠ NULO faça
        B[prox] ← x^.Chave
        prox ← prox + 1
        x ← x^.Prox
    Destroi(Bucket[i])
```

# Algoritmos de Ordenação

- Por que usar BucketSort (BinSort), conjuntamente com InsertionSort para cada balde, ao invés de usar InsertionSort diretamente sobre todos os dados?
  - Se a distribuição das chaves no intervalo [0, 1) é perto da uniforme, o número de elementos em cada balde é relativamente baixo (em média, $\theta(1)$!)

# Algoritmos de Ordenação

- **Counting Sort:**
  - Quando a ordenação é apenas de chaves (não há informações adicionais relacionadas às chaves):
    - Cria-se um vetor auxiliar Bucket com $B_{max} - B_{min} + 1$ inteiros, onde $B_{min} = \min \{ B[i] \mid 1 \leq i \leq N \}$ e $B_{max} = \text{máx} \{ B[i] \mid 1 \leq i \leq N \}$. Bucket[i] representará o número de elementos i em B
    - Para i de 1 a N, incrementa-se Bucket(B[i])
    - O vetor ordenado consegue-se com o seguinte processo: para i de $B_{min}$ a $B_{max}$, os próximos Bucket[i] inteiros na ordenação são o inteiro i

# Algoritmos de Ordenação

```
procedimento CountingSort(ref B[], N: Inteiro)
//Assume: |B| ≥ N
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
var    i, j, prox, Bmax, Bmin: Inteiro

Bmin ← min{B[i] : 1 ≤ i ≤ N}
Bmax ← máx{B[i] : 1 ≤ i ≤ N}

var Bucket[Bmin..Bmax]: Inteiro
Bucket[Bmin..Bmax] ← 0
para i ← 1 até N faça
    Bucket[B[i]] ← Bucket[B[i]]+1

prox ← 1
para i ← Bmin até Bmax faça
    para j ← 1 até Bucket[i]
        B[prox] ← i
        prox ← prox + 1
```

**Tempo:**
$\theta(N + (B_{max} - B_{min}))$

**Espaço:**
$\theta(B_{max} - B_{min})$

# Algoritmos de Ordenação

- **Counting Sort (simplificação):**
  - Se B possui elementos distintos, então pode-se economizar espaço, fazendo com que o contador Bucket(i) que indica quantas ocorrências do número i em B seja transformado em um valor lógico indicando se o número i aparece ou não!
    - Se um inteiro ocupa 32 bits, isto significa usar 32 vezes menos memória para armazenar Bucket!

# Algoritmos de Ordenação

```
procedimento CountingSort(ref B[], N: Inteiro)
//Assume: |B| ≥ N,
        B possui elementos distintos
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
    var i, prox, Bmax, Bmin: Inteiro

    Bmin ← min{B[i] : 1 ≤ i ≤ N}
    Bmax ← máx{B[i] : 1 ≤ i ≤ N}

    var Bucket[Bmin..Bmax]: Lógico
    Bucket[Bmin..Bmax] ← F
    para i ← 1 até N faça
        Bucket[B[i]] ← V

    prox ← 1
    para i ← Bmin até Bmax faça
        se Bucket[i] então
            B[prox] ← i
            prox ← prox + 1
```

**Tempo:**
$\theta(N + B_{max} - B_{mini})$

**Espaço:**
$\theta(B_{max} - B_{mini})$
(com menor constante)

# RadixSort

# Algoritmos de Ordenação

- **RadixSort:**
  - Ordena-se todas chaves pelo seu algarismo menos significativo
  - Reordena-se em seguida as chaves pelo seu segundo algarismo menos significativo com algum algoritmo de ordenação estável
    - Uma ordenação é estável se mantém entre duas chaves iguais a mesma ordem relativa na permutação de saída que aquela da entrada
  - Ex: ordenar 912, 811, 251, 290
    - 1a ordenação: 290, 811, 251, 912
    - 2a ordenação: 811, 912, 251, 290

# Algoritmos de Ordenação

- **RadixSort:**
  - Como resultado, os números estão ordenados pelos últimos 2 dígitos significativos!
    - Ou eles foram ordenados porque um número é menor do que o outro, ou eles têm o mesmo dígito e, como o algoritmo de ordenação é estável, as chaves que possuem o menor dígito no próximo dígito significativo vêm antes, pois foram ordenadas no passo anterior
  - O algoritmo prossegue ordenando todos os dígitos
    - 2a ordenação: 811, 912, 251, 290
    - 3a ordenação: 251, 290, 811, 912
- Estável, não é in loco

# Algoritmos de Ordenação

```
procedimento RadixSort(ref B[), N, d: Inteiro)
//Assume: |B| ≥ N, B é vetor de inteiros com d dígitos
//Garante: B[i] ≤ B[i+1] ∀ 1 ≤ i < N
    var F[0..9]: Fila //10 filas para servir de baldes
    var dig: Inteiro
    para i ← 0 até 9 faça
        Constroi(F[i])
    para j ← d até 1 passo -1 faça
        para i ← 1 até N faça
            dig ← d-ésimo dígito de B[i]
            Enfileira (F[dig], B[i] )
        i ← 1
        para dig ← 0 até 9 faça
            enquanto Tamanho(F[dig]) > 0 faça
                B[i], i ← Desenfileira (F[dig]), i + 1
    para i ← 0 até 9 faça
        Destroi(F[i])
```

**Tempo:**
θ(Nd)

**Espaço:**
θ(N)

# Exercícios

# Exercícios

1. Ordene o vetor B = [30 11 18 23 17 5 12 21] por cada um dos algoritmos de ordenação por comparação. Para cada um, reporte o número de comparações entre elementos que foi empregado.

2. Considere o problema de encontrar os p menores elementos de um vetor B[1..$10^7$] de inteiros. Duas soluções foram sugeridas:
   a. faça p buscas em B, sendo que a i-ésima busca tem por objetivo selecionar o i-ésimo menor elemento; cada elemento selecionado numa busca deve ser marcado para ser desconsiderado nas próximas buscas
   b. ordene B e retorne os p primeiros elementos

   Sabendo-se que os tempos de execução no computador onde será executado a solução escolhida são de:

   - 1 segundo para varrer um vetor com $10^6$ elementos, e
   - 1 segundo para ordenar por Mergesort um vetor com $10^5$ elementos,

   pergunta-se: para qual intervalo de valores de p você recomendaria cada uma das soluções?

# Exercícios

3. Considere uma variação da solução (a) do exercício anterior: ao invés de marcar o i-ésimo elemento para não ser considerado, troque-o de posição com o elemento B[i], de modo que as próximas buscas sempre sejam em uma lista com uma unidade a menos que a busca anterior (B[i+1..$10^7$]). Com esta melhoria, revise o intervalo de valores de p para o qual cada uma das duas soluções é mais promissora.

4. Escreva a função Merge() do Mergesort como um algoritmo recursivo.

5. Considere aplicar cada algoritmo de ordenação em uma lista linear com alocação encadeada dos elementos. Quais algoritmos ainda se aplicam com a mesma complexidade de tempo?

# Exercícios

6. Considere variações da linha "`m ← (inicio + fim) div 2`" no algoritmo do Mergesort, conforme cada variação abaixo. Determine a complexidade de tempo do MergeSort para cada uma.
   a. `m ← fim`
   b. `m ← máx {1, fim - 5}`
   c. `m ← inicio + (fim - inicio + 1) div 4`

7. Considere um vetor B[1..N], onde cada elemento é um inteiro entre 1 e 100. Elabore um algoritmo de ordenação para B com complexidade de tempo O(N).