


CT-234



Estruturas de Dados,
Análise de Algoritmos e
Complexidade Estrutural

Carlos Alberto Alonso Sanches

CT-234



5) Ordenação

Resoluções simples, Lower bound, MergeSort, RadixSort

Alguns algoritmos de ordenação

- A ordenação é o problema mais clássico da computação.
- Inicialmente, veremos algumas das suas resoluções mais simples:
 - Ordenação pelo método da bolha (*BubbleSort*)
 - Ordenação por seleção (*SelectionSort*)
 - Ordenação por inserção (*InsertionSort*)
- Consideraremos sempre a ordenação de um vetor v de índices $[1..n]$.

Método da bolha (*BubbleSort*)

- É um dos algoritmos mais simples e conhecidos.
- Princípio:
 - Os elementos vizinhos são comparados e, caso estejam fora de ordem, são trocados.
 - A propagação dessas comparações permite isolar o maior (ou o menor) elemento do vetor.
 - Repetindo-se esse processo com as demais posições do vetor, é possível ordená-lo completamente.
 - Este método recebe o nome de bolha, pois os elementos 'sobem' até a sua posição final, de modo semelhante a uma bolha em um tubo com água.

Exemplo para n=8

No esquema abaixo, a bolha "desce"
(como se o tubo estivesse de ponta-cabeça)

1	44	44	12	12	12	12	6	6
2	55	12	42	42	18	6	12	12
3	12	42	44	18	6	18	18	18
4	42	55	18	6	42	42	42	42
5	94	18	6	44	44	44	44	44
6	18	6	55	55	55	55	55	55
7	6	67	67	67	67	67	67	67
8	67	94	94	94	94	94	94	94

Algoritmo

```
BubbleSort() {  
    for (i=1; i<n; i++)  
        for (j=1; j<=n-i; j++)  
            if (v[j] > v[j+1]) {  
                x = v[j];  
                v[j] = v[j+1];  
                v[j+1] = x;  
            }  
}
```

- É lento, pois só faz comparações entre posições adjacentes.
- Pode ser melhorado com testes intermediários para verificar se o vetor já está ordenado.
- Mesmo assim, o tempo de pior caso é $\Theta(n^2)$.

Ordenação por seleção (*SelectionSort*)



- Procedimento:
 - Selecione o menor elemento do vetor e troque-o com o que está na posição 1.
 - Desconsiderando a primeira posição do vetor, repita essa operação com as restantes.

Exemplo com n=6

Vetor inicial:

	1	2	3	4	5	6
	5	6	2	3	4	1
	1	6	2	3	4	5
	1	2	6	3	4	5
	1	2	3	6	4	5
	1	2	3	4	6	5
	1	2	3	4	5	6

Algoritmo



```
SelectionSort() {  
    for (i=1; i<n; i++) {  
        min = i;  
        for (j=i+1; j<=n; j++)  
            if (v[j] < v[min])  
                min = j;  
        x = v[min];  
        v[min] = v[i];  
        v[i] = x;  
    }  
}
```

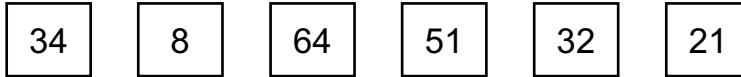
Sempre gasta tempo $\Theta(n^2)$

Ordenação por inserção (*InsertionSort*)

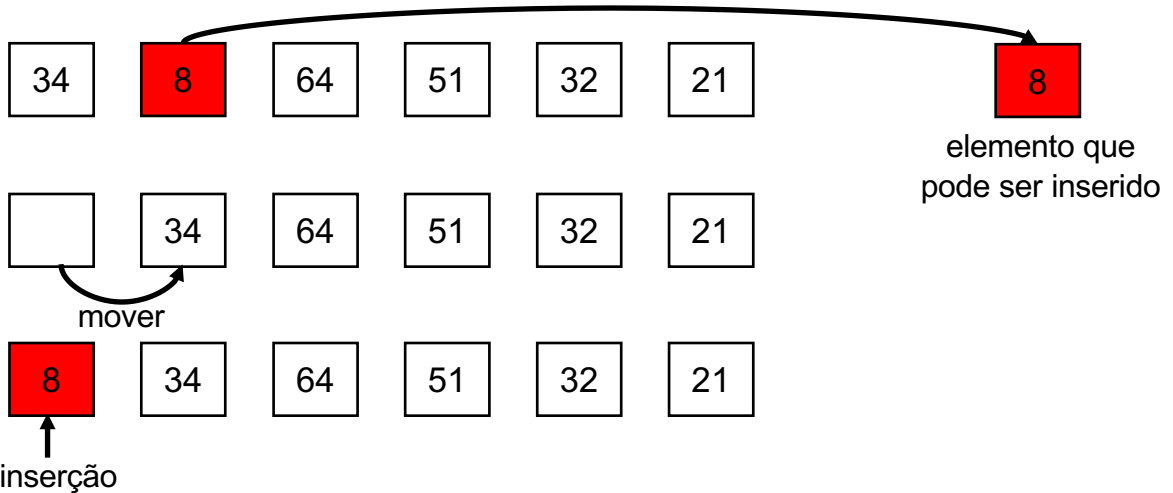


- Semelhante ao método de ordenação das cartas de um baralho.
- Procedimento:
 - Verifica-se se o valor da posição 2 do vetor poderia ser colocado na posição 1.
 - Repete-se este processo para as posições subsequentes, verificando-se o local adequado da inserção.
- A inserção de um elemento na sua nova posição exige a movimentação de vários outros.

Exemplo com $n=6$

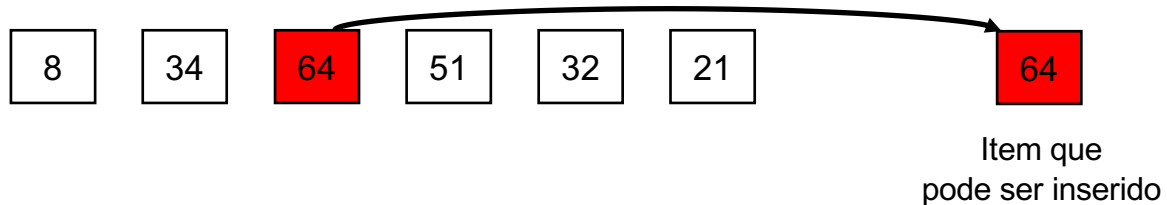


Passo 1



Exemplo com $n=6$

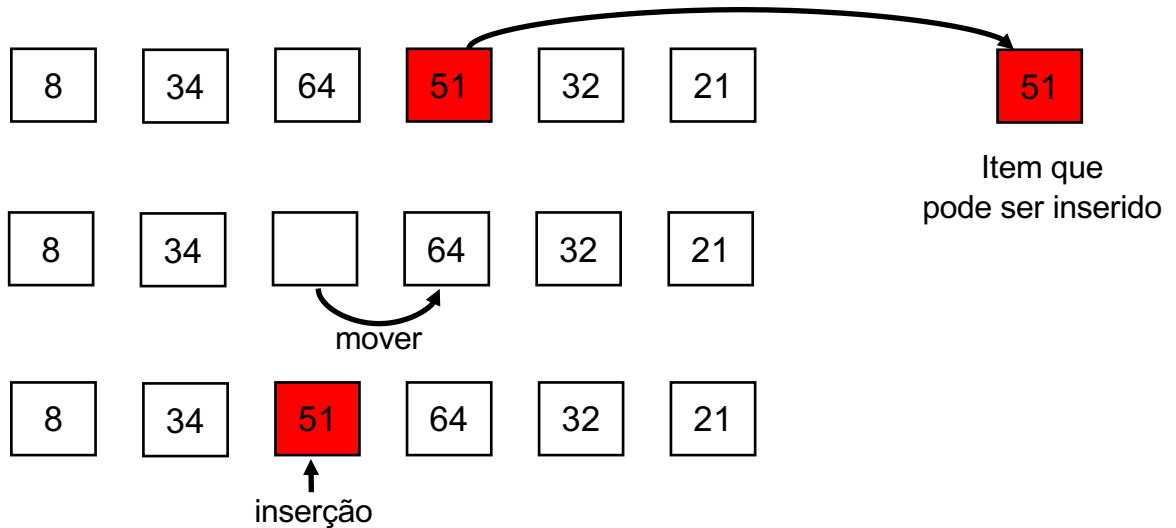
Passo 2



Não ocorre inserção, pois esse elemento já está no seu lugar

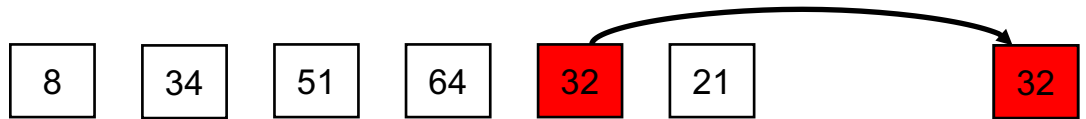
Exemplo com n=6

Passo 3



Exemplo com n=6

Passo 4



Item que
pode ser inserido



mover



mover



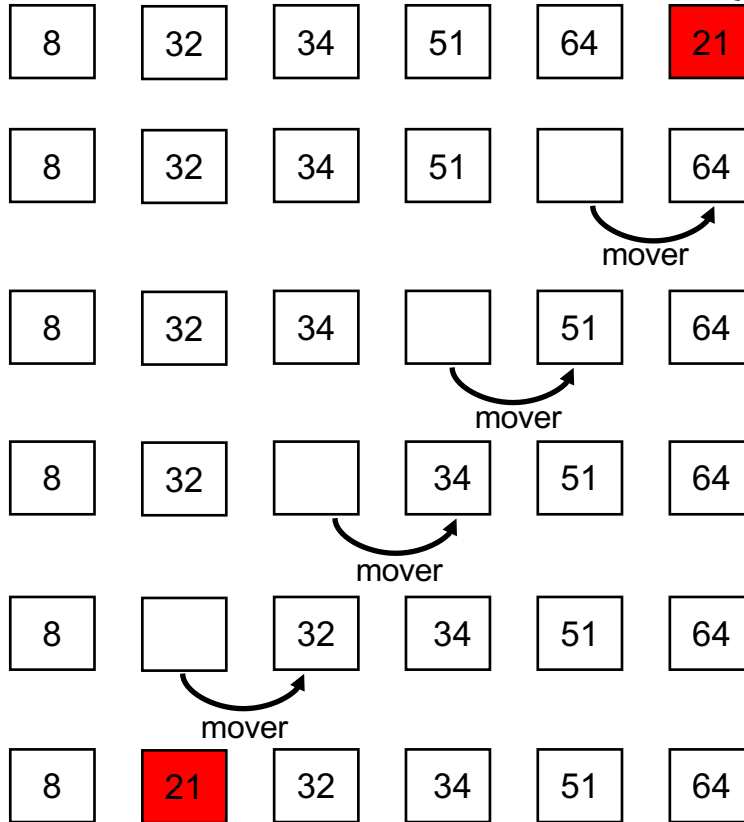
mover



inserção

Exemplo com n=6

Passo 5



Item que
pode ser inserido

← Final

Algoritmo



```
InsertionSort() {  
    for (i=2; i<=n; i++) {  
        x = v[i];  
        for (j=i; j>1 && x<v[j-1]; j--)  
            v[j] = v[j-1];  
        v[j] = x;  
    }  
}
```

- Quando o vetor está ordenado, gasta tempo $\Theta(n)$.
- No entanto, seu tempo de pior caso é $\Theta(n^2)$.

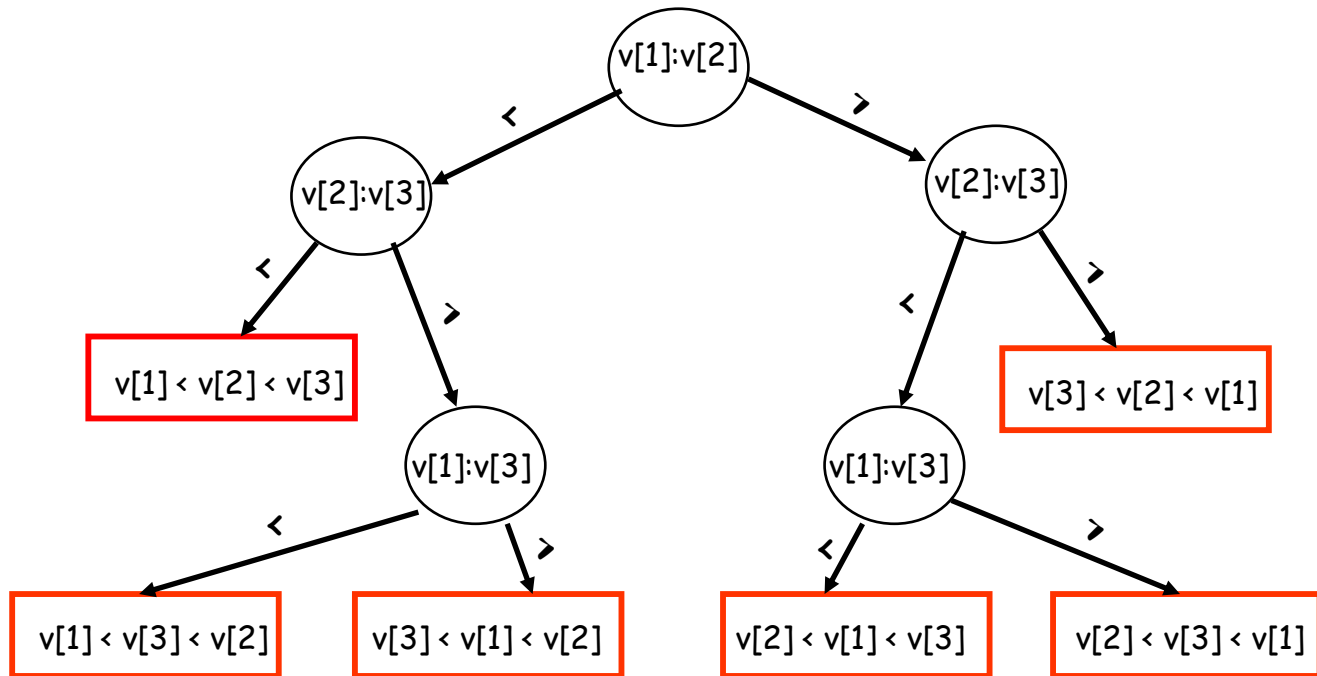
Lower bound para a ordenação

- Até agora, apresentamos algoritmos que ordenam n números em tempo de pior caso $\Theta(n^2)$. Por enquanto, esse é o nosso *upper bound* para o problema da ordenação baseado em comparações.
- Seria possível calcular um *lower bound* para esse problema?
- Em outras palavras, desejamos encontrar um limite inferior teórico para esse problema, isto é, a mínima complexidade de tempo de quaisquer de suas resoluções algorítmicas.

Árvore de comparações

- Qualquer algoritmo de ordenação *baseado em comparações* pode ser representado em uma árvore binária.
- Na raiz fica a primeira comparação realizada entre dois elementos do vetor; nos filhos, as comparações subsequentes. Deste modo, as folhas representam as possíveis soluções do problema.
- A altura dessa árvore é o número máximo de comparações que o algoritmo realiza, ou seja, o seu tempo de pior caso.

Exemplo: com 3 valores distintos



Como estamos ordenando 3 elementos, há $3!$ possíveis resultados

Generalização

- Na ordenação de n elementos distintos, há $n!$ possíveis resultados, que correspondem às permutações desses elementos.
- Portanto, qualquer árvore binária de comparações terá *no mínimo* $n!$ folhas.
- A árvore mínima de comparações tem exatamente $n!$ folhas. Supondo que a altura dessa árvore seja h , então $LB(n) = h$, onde $LB(n)$ é o *lower bound* de tempo para a ordenação de n elementos.
- Sabemos que o número máximo de folhas de uma árvore binária de altura h é 2^h .
- Portanto, $n! \leq 2^h$, ou seja, $h \geq \lg n!$
Logo, $LB(n) \geq \lg n!$

Cálculo do *lower bound*

- Pela aproximação de Stirling: $n! \approx (2\pi n)^{1/2} n^n e^{-n}$
- Portanto:
 - $\lg n! \approx \lg (2\pi)^{1/2} + \lg n^{1/2} + \lg n^n + \lg e^{-n}$
 - $\lg n! \approx \Theta(1) + \Theta(\log n) + \Theta(n \cdot \log n) - \Theta(n)$
- Como $LB(n) \geq \lg n!$, então $LB(n) = \Omega(n \cdot \log n)$
- Se encontrarmos um algoritmo baseado em comparações que resolva a ordenação em tempo de pior caso $\Theta(n \cdot \log n)$, ele será ótimo em termos de complexidade de tempo, e este problema estará computacionalmente resolvido.

Outra maneira de calcular

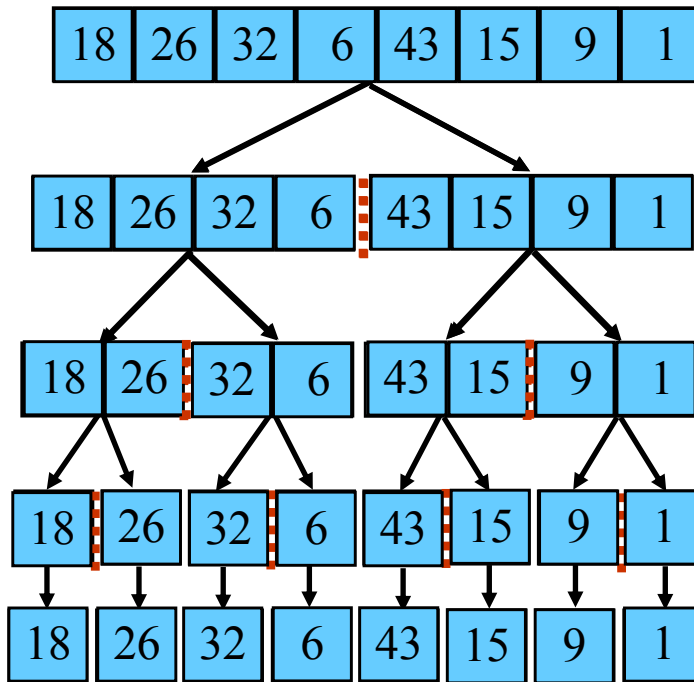
- $\lg n! = \lg (n.(n-1).(n-2). \dots .2.1)$
- $\lg n! = \lg n + \lg (n-1) + \lg (n-2) + \dots + \lg 2 + \lg 1$
- $\lg n! \geq \lg n + \lg (n-1) + \dots + \lg (n/2)$
- Todas essas $n/2$ parcelas são maiores que $\lg (n/2)$
- Portanto:
 - $\lg n! \geq (n/2).\lg (n/2)$
 - $\lg n! = \Omega(n.\log n)$
 - $LB(n) = \Omega(n.\log n)$
- Qual o problema desta demonstração?

MergeSort (Von Neumann, 1945)

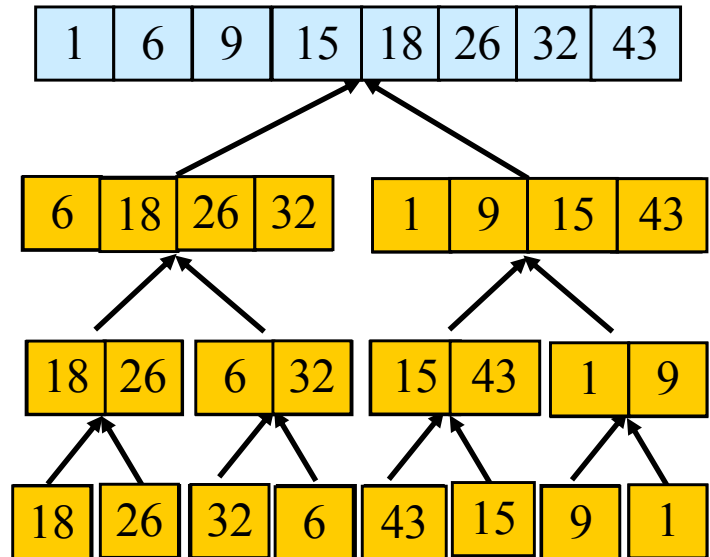
- Este algoritmo é um exemplo do paradigma *Divisão-e-Conquista*, e por isso tem 3 fases:
 - Divisão: o vetor é dividido em duas metades
 - Conquista: cada metade é ordenada recursivamente, dando origem a duas subsoluções
 - Combinação: essas subsoluções são combinadas, formando a solução final
- Condição de parada da recursão: quando for ordenar apenas um elemento. Este caso será a subsolução elementar.

Exemplo para n=8

Vetor original



Vetor ordenado



Algoritmo

```
MergeSort(i, f) {  
  if (i < f) {  
    m = ⌊(i+f)/2⌋;  
    MergeSort(i, m);  
    MergeSort(m+1, f);  
    merge(i, m, f);  
  }  
}
```

Convém que o vetor aux
tenha mesmo tamanho de v e
seja alocado como global

Chamada inicial:

```
MergeSort(1, n)
```

```
merge(i, m, f) {  
  i1 = i;  
  i2 = i;  
  i3 = m+1;  
  while (i2 <= m && i3 <= f)  
    if (v[i2] < v[i3])  
      aux[i1++] = v[i2++];  
    else  
      aux[i1++] = v[i3++];  
  while (i2 <= m)  
    aux[i1++] = v[i2++];  
  while (i3 <= f)  
    aux[i1++] = v[i3++];  
  for (j=i; j<=f; j++)  
    v[j] = aux[j];  
}
```

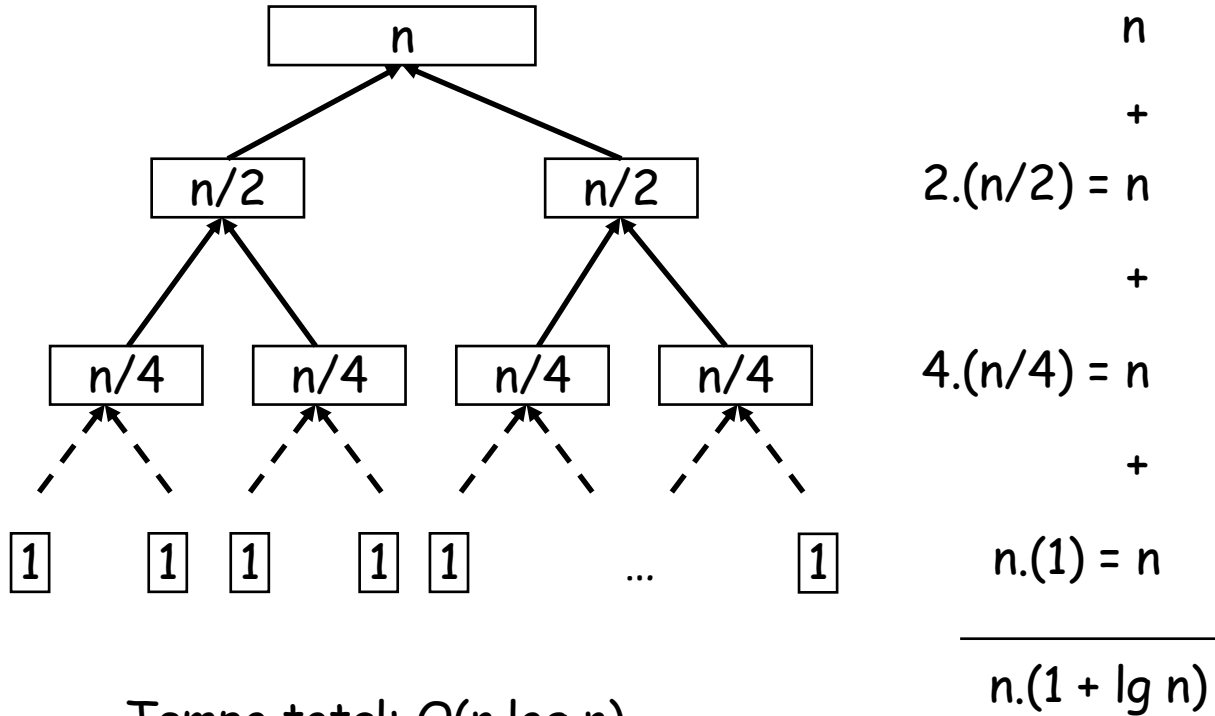
É possível economizar este
laço final. Como?

Complexidade de tempo de merge: $\Theta(f-i)$

Complexidade de tempo do *MergeSort*

- $T(1) = 1$ (tempo constante)
- $T(n) = 2T(n/2) + \Theta(n)$, $n > 1$
- Supondo $n = 2^k$:
 - $T(n) = 2(2T(n/2^2) + \Theta(n/2)) + \Theta(n) = 2^2T(n/2^2) + 2\Theta(n)$
 - $T(n) = 2^2(2T(n/2^3) + \Theta(n/2^2)) + 2\Theta(n) = 2^3T(n/2^3) + 3\Theta(n)$
 - Generalizando: $T(n) = 2^kT(n/2^k) + k\Theta(n)$
- Substituindo $n = 2^k$:
 - $T(n) = n + \Theta(n) \cdot \lg n$
 - $T(n) = \Theta(n \cdot \log n)$
- A generalização para n qualquer não muda a ordem de $T(n)$

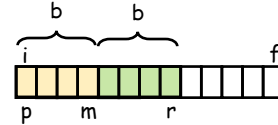
Outro modo de calcular o tempo



Tempo total: $\Theta(n \cdot \log n)$

MergeSort iterativo

- É possível escrever uma variante do *MergeSort* não recursiva e sem pilha, cuja execução é mais rápida.



```
MergeSort(i, f) {  
    b = 1; // b: tamanho de cada bloco  
    while (b < f) {  
        p = i; // p: posição inicial do 1º bloco  
        while (p+b <= f) {  
            r = min{f, p-1+2*b}; // r: posição final do 2º bloco  
            m = p+b-1; // m: posição final do 1º bloco  
            merge(p, m, r);  
            p += 2*b;  
        }  
        b *= 2; // tamanho dos blocos é duplicado  
    }  
}
```

- É uma simulação da versão recursiva, com menor uso da pilha de execução.
- No entanto, as complexidades de tempo e de espaço não são alteradas.

Conclusões

- Qualquer algoritmo que ordenar n números através de comparações em tempo de pior caso $\Theta(n \log n)$ será ótimo em termos de complexidade de tempo.
- *MergeSort* faz isso: portanto, o *upper bound* de tempo para a ordenação através de comparações é $\Theta(n \log n)$.
- Como o *upper* e o *lower bounds* de tempo da ordenação são iguais, podemos dizer que a ordenação através de comparações é um problema computacionalmente resolvido.
- No entanto, o *MergeSort* necessita de espaço extra $\Theta(n)$ para armazenar o vetor temporário (convém que seja alocado uma única vez pelo programa principal). Veremos que ele não é ótimo em termos de complexidade de espaço extra.

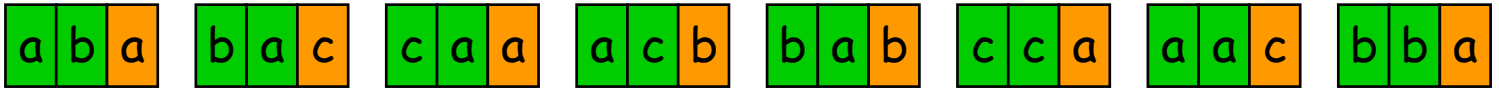
RadixSort



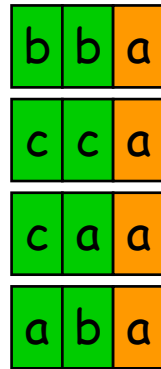
- Em determinadas condições, é possível ordenar em tempo de pior caso $\Theta(n)$.
- Por exemplo, isso ocorre quando:
 - 1) os valores têm um comprimento limitado;
 - 2) a ordenação baseia-se em cálculos com esses valores (e não em comparações).
- Como funciona o *RadixSort*:
 - Os valores de entrada, escritos em alguma base numérica, têm exatamente d dígitos.
 - A ordenação é realizada em d passos: um dígito por vez, começando a partir dos menos significativos.

Exemplo com $d=3$

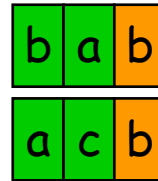
Passo 1a: Separá-los de acordo com o dígito mais à direita



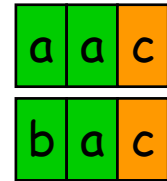
3 filas, pois
a base é 3



a

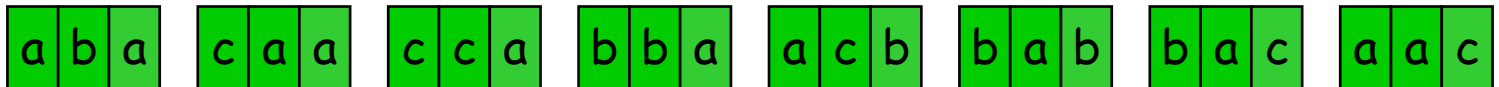


b



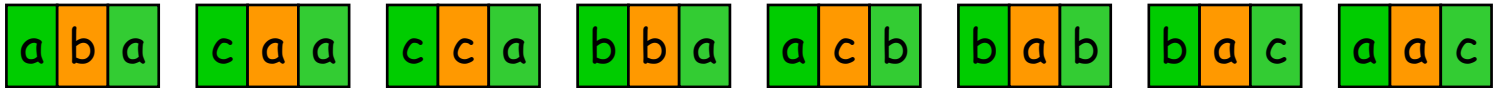
c

Passo 1b: Uni-os seguindo a ordem das filas

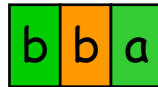


Exemplo com d=3

Passo 2a: Separá-los de acordo com o segundo dígito



a



b



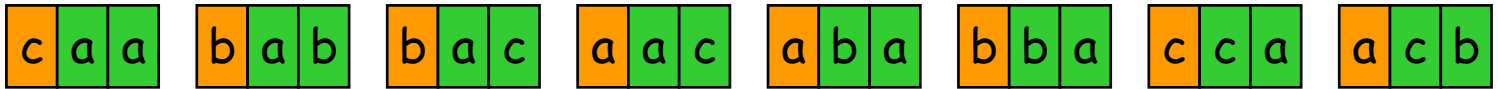
c

Passo 2b: Uni-os seguindo a ordem das filas

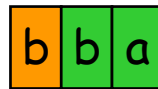


Exemplo com d=3

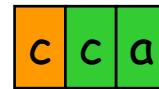
Passo 3a: Separá-los de acordo com o terceiro dígito



a

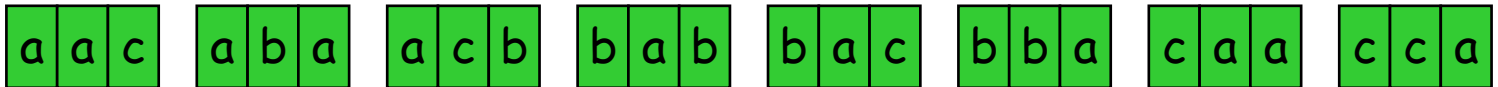


b



c

Passo 3b: Uni-los seguindo a ordem das filas



Algoritmo

```
RadixSort() {
    Queue q[0..base-1];
    d iterações → for (i=0, factor=1; i<d; factor *= base, i++) {
        for (j=1; j<=n; j++)
            q[(v[j]/factor)%base].enqueue(v[j]); }  $\Theta(n)$ 
        for (j=0, k=1; j<base; j++)
            while (!q[j].isEmpty())
                v[k++] = q[j].dequeue(); }  $\Theta(n+base)$ 
    }
}
```

- Tempo: $\Theta(d.(n+base)) = \Theta(n)$, pois d e $base$ são constantes.
- Por que não é usado na prática?
 - 1) A constante d costuma ser grande.
 - 2) *Overhead* da manipulação das estruturas de dados.