Técnicas de Desenvolvimento de Algoritmos (parte 2)

Prof. Marcelo Rosa

Algoritmos e Estrutura de Dados 2 (AE43CP)
Engenharia de Computação
Departamento Acadêmico de Informática (Dainf)
Universidade Tecnológica Federal do Paraná (UTFPR)
Campus Pato Branco

Sumário

1. Divisão e Conquista
   - Problema: ordenar um vetor de inteiros
   - Problema: encontrar o maior valor
   - Problema: potenciação

Divisão e Conquista

Ideia básica

- **Divisão**: dividir o problema a ser resolvido em 2 ou mais subproblemas menores e independentes
- **Conquista**: encontrar soluções para as partes recursivamente
  - Se o tamanho do subproblema for pequeno o bastante, então a solução é direta
- **Combinação**: combinar as soluções obtidas em uma solução global

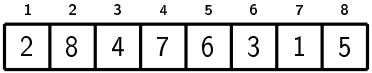
Pseudocódigo: ideia geral

```
1:  function Divisao_e_Conquista(x)
2:      if x é pequeno ou simples then
3:          return resolver(x)
4:      else
5:          decompor x em conjuntos menores x_0, x_1, ... x_n
6:          for (i ← 0 até n) do
7:              y_i ← Divisao_e_Conquista(x_i)
8:          end for
9:          combinar y_i's
10:     end if
11:     return y
12: end function
```

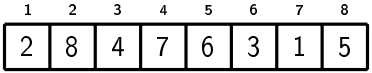
Sumário

1. Divisão e Conquista
   - Problema: ordenar um vetor de inteiros
   - Problema: encontrar o maior valor
   - Problema: potenciação

- O problema consiste em ordenar um array $A[1..n]$ com n inteiros

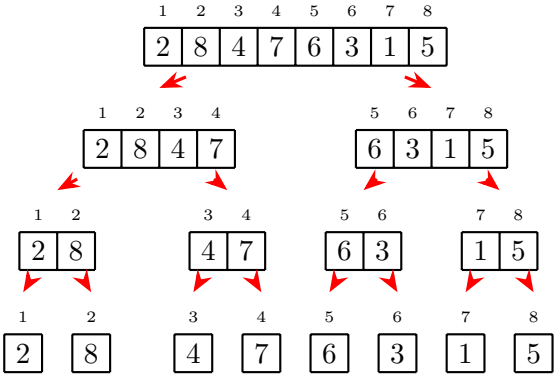


- O problema consiste em ordenar um array $A[1..n]$ com n inteiros



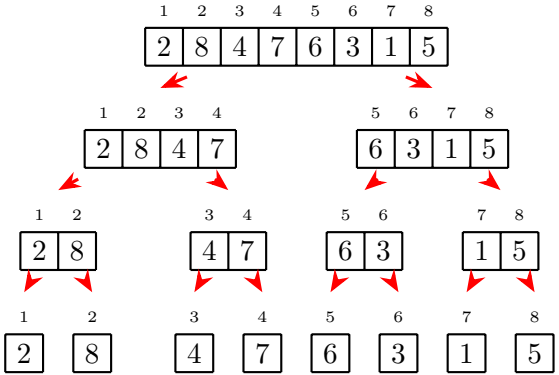
- **Divisão**: o array de n elementos a ser ordenado é divido em dois subarrays de $n/2$ elementos cada.
- **Conquista**: cada subarray é ordenado individualmente, recursivamente, utilizando a ordenação por intercalação.
  - subarray com tamanho 1 já está ordenado.
- **Combinação**: os subarrays ordenados são intercalados para produzir a resposta ordenada.

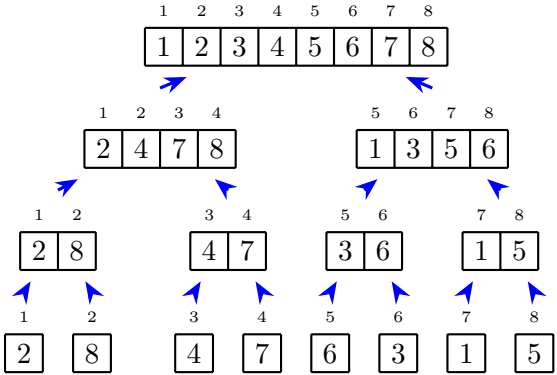
Exemplo 1: ordenar um array de inteiros



Divisão e Conquista

Exemplo 1: ordenar um array de inteiros





Divisão e Conquista

Exemplo 1: ordenar um array de inteiros

Algoritmo Merge Sort

```
1: procedure MERGE-SORT(A, p, r)
2:     if p < r then
3:         q ← ⌊(p + r)/2⌋
4:         MERGE-SORT(A, p, q)
5:         MERGE-SORT(A, q + 1, r)
6:         MERGE(A, p, q, r)
7:     end if
8: end procedure
```

```
1: procedure MERGE(A, p, q, r)
2:     n_1 ← q − p + 1
3:     n_2 ← r − q
4:     sejam L[1 . . . n_1 + 1] e R[1 . . . n_2 + 1] novos arranjos
5:     for i ← 1 to n_1 do
6:         L[i] = A[p + i − 1]
7:     end for
8:     for j ← 1 to n_2 do
9:         R[j] = A[q + j]
10:    end for
11:    L[n_1 + 1] ← ∞
12:    R[n_2 + 1] ← ∞
13:    i ← 1
14:    j ← 1
15:    for k ← p to r do
16:        if L[i] ≤ R[j] then
17:            A[k] = L[i]
18:            i ← i + 1
19:        else
20:            A[k] = R[j]
21:            j ← j + 1
22:        end if
23:    end for
24: end procedure
```

- A função que descreve o número de operações realizada pelo algoritmo é representada pela equação de recorrência

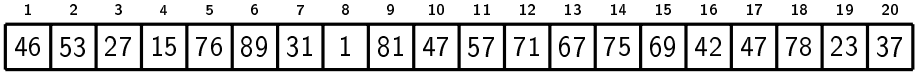
$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & n > 1 \end{cases}$$

- Logo, a complexidade do *Merge Sort* é $\Theta(n \log n)$, pelo caso 2 do método mestre.

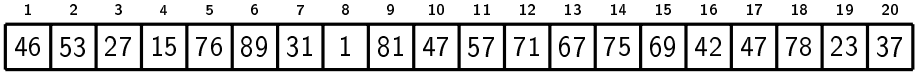
Sumário

1. Divisão e Conquista
   - Problema: ordenar um vetor de inteiros
   - Problema: encontrar o maior valor
   - Problema: potenciação

- O problema consiste em encontrar o maior elemento de um array $A[1..n]$



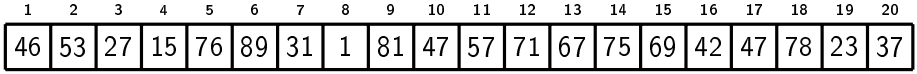
- O problema consiste em encontrar o maior elemento de um array $A[1..n]$



Solução Ingênua

1: $max \leftarrow A[1]$
2: **for** $i \leftarrow 2$ até n **do**
3: **if** $A[i] > max$ **then**
4: $max \leftarrow A[i]$
5: **end if**
6: **end for**
7: **return** max

- O problema consiste em encontrar o maior elemento de um array $A[1..n]$



Solução Ingênua

1: $max \leftarrow A[1]$
2: **for** $i \leftarrow 2$ até n **do**
3: **if** $A[i] > max$ **then**
4: $max \leftarrow A[i]$
5: **end if**
6: **end for**
7: **return** max

- Complexidade de tempo: $\Theta(n)$

Solução por divisão e conquista

```
1:  function Maxim(A, inicio, fim)
2:      if inicio = fim then
3:          return A[inicio]
4:      else
5:          m ← ⌊(inicio + fim)/2⌋
6:          max_esq ← Maxim(A, inicio, m)
7:          max_dir ← Maxim(m + 1, fim)
8:          return max(max_esq, max_dir)
9:      end if
10: end function
```

- A complexidade de tempo $T(n)$ do algoritmo é uma fórmula de recorrência

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \Theta(1) & n > 1 \end{cases}$$

- Logo, por meio do método mestre (caso 1), para a função $T(n)$ foi definida a complexidade de tempo $\Theta(n)$

Sumário

1. Divisão e Conquista
   - Problema: ordenar um vetor de inteiros
   - Problema: encontrar o maior valor
   - Problema: potenciação

- Calcular a^n, para $n \geq 0$

- Calcular a^n, para $n \geq 0$

Solução Ingênua

```
1:  function potencia(a, n)
2:      if n = 0 then
3:          return 1
4:      else
5:          p ← a
6:          for i ← 2 até n do
7:              p ← p × a
8:          end for
9:          return p
10:     end if
11: end function
```

- Calcular a^n, para $n \geq 0$

Solução Ingênua

```
1: function potencia(a, n)
2:     if n = 0 then
3:         return 1
4:     else
5:         p ← a
6:         for i ← 2 até n do
7:             p ← p × a
8:         end for
9:         return p
10:    end if
11: end function
```

- Complexidade de tempo: $O(n)$

- Calcular a^n, para $n \geq 0$

- Calcular a^n, para $n \geq 0$

Ideia básica

$$a^n = \begin{cases} 1 & n = 0 \\ (a^{\lfloor \frac{n}{2} \rfloor})^2 & n \text{ é par} \\ a \times (a^{\lfloor \frac{n}{2} \rfloor})^2 & n \text{ é impar} \end{cases}$$

- Calcular a^n, para $n \geq 0$

Solução divisão e conquista

```
1: function potencia(a, n)
2:     if n = 0 then
3:         return 1
4:     end if
5:     p ← potencia(a, ⌊n/2⌋)
6:     p ← p × p
7:     if n é impar then
8:         p ← a × p
9:     end if
10:    return p
11: end function
```

- A complexidade $T(n)$ do algoritmo é uma fórmula de recorrência, tanto para a análise de tempo quanto de espaço

$$T(n) = \begin{cases} \Theta(1) & n = 0 \\ T(\frac{n}{2}) + \Theta(1) & n \geq 1 \end{cases}$$

- Logo, por meio do método mestre (caso 2), para a função foi definida a complexidade de $\Theta(\log n)$

Divisão e Conquista

- Vantagens:
  - Paralelismo
    - Os subproblemas geralmente são independentes
  - Fácil implementação (recursividade)
  - Simplificação de problemas complexos
    - Por exemplo, a ordenação de arrays pode ser reduzida de $O(n^2)$ para $O(n \log n)$ (Merge Sort)
- Desvantagens:
  - Repetição de subproblemas
    - Problemas como Fibonacci repetem cálculos.
  - Necessidade de memória auxiliar
    - Excesso de chamadas pode causar estouro de pilha (*StackOverflow*)
    - Programação dinâmica

Referências I


Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Clifford, S.
Algoritmos: teoria e prática.
Elsevier, 2012.


Oliva, J. T.
Árvores B. AE23CP - Algoritmos e Estrutura de Dados II.
Slides. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2025.