Capitulo 3
Recursividade e algoritmos de busca e ordenação

Profa. Dra. Laura Rodríguez
E-mail: lmrodrig@uma.pt
Universidade da Madeira

3. Recursividade e algoritmos de busca e ordenação

1. **Recursão**
2. **Ordenação**
3. **Busca**

3. Recursividade e algoritmos de busca e ordenação

1. **Recursão**
2. **Ordenação**
3. **Busca**

3.1 Recursão (ou Recursividade)

- É uma técnica empregada na construção de algoritmos
  - Melhorar a clareza e simplicidade
  - Em alguns casos a solução não-recursiva é bem mais extensa e menos elegante
- Consiste em reduzirmos um determinado problema a **uma instância menor**, do mesmo problema ...

3.1 “Fórmula geral” da recursividade

- Se o problema for pequeno, então resolva-o diretamente
- Se o problema for grande:
  - Reduza o seu tamanho, criando uma instância menor do mesmo problema
  - Resolva a instância menor, aplicando o mesmo método
  - E finalmente, retorne ao problema original

3.1 Algoritmos recursivos

- Algoritmos recursivos precisam ter um **caso trivial**, isto é, aquela instância (pequena) do problema, para a qual se conhece uma solução simples (nem que seja com força-bruta ...)
- Um algoritmo recursivo faz chamadas a si próprio (para resolver a instância menor)

Exemplo

- Provavelmente, o exemplo mais clássico de algoritmo recursivo seja o cálculo do fatorial:

  0! = 1

  1! = 1

  2! = 1*2 = 2

  3! = 1*2*3 = 6

  ...

  N! = 1*...*N

Cálculo do fatorial

- Vejamos um algoritmo (não-recursivo) para o cálculo do fatorial

```
FATORIAL(n)
     fat ← 1;
     cont ← 1;
     enquanto cont ≤ n
          fat ← fat * cont;
          cont ← cont + 1;
     retorne fat;
```

Fatorial usando recursão

- Primeiramente, vamos formular uma solução do problema:

Fat (n) =

1, se n = 0

n*fat(n-1), se n>0

Algoritmo recursivo

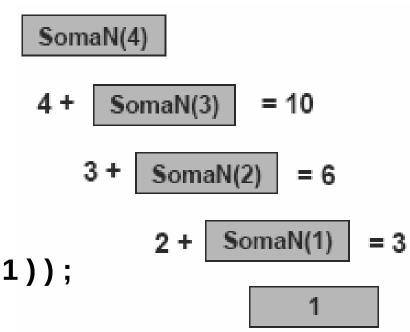
- Vejamos como ficaria o algoritmo recursivo, para o cálculo do fatorial:

```
FAT_REC(n)
   se n = 0
      então retorne 1;
      senão retorne n * FAT_REC(n-1);
```

Exemplo: Somatória

- Exemplo: Escrever uma função recursiva que calcule a soma dos números entre 1 e n.

```
int SomaN (int n )
{
   if ( n == 1) // Caso básico
      return 1 ;
   else // Caso Geral
      return ( n + SomaN ( n - 1 ) ) ;
}
```



Exemplo: Somatória

- **O mesmo problema resolvido da forma convencional:**

```
int SomaN (int n )
{
   int I,Soma=0;
   for(I=1; I<=n; I++)
      Soma+=I;
   return (Soma) ;
}
```

```
int SomaN(int n)
{
    return(n*(float)(1+n)/2);
}
// 4*(1+4)/2
// 4* 2.5 / 2 = 10
```

Recursividade

- **Vantagens:**
  - A utilização de uma função recursiva pode simplificar a solução de alguns problemas;
  - Pode-se obter um código mais conciso e eficaz nessas situações;
  - Uma solução recursiva pode, por outro lado, eliminar a necessidade de manter o controle manual sobre uma série de variáveis normalmente associadas aos métodos alternativos à recursividade.

Recursividade

- **Desvantagens:**
  - As funções recursivas são geralmente mais lentas e ocupam mais memória do que as funções iterativas equivalentes, uma vez que são feitas muitas chamadas consecutivas a funções;
  - Um erro de implementação pode levar ao esgotamento dos recursos associados à pilha que gere a chamada a funções. Isto é, caso não seja indicada nenhuma condição de paragem, ou essa condição foi definida de forma errada e nunca será satisfeita, então o processo recursivo não terá fim.

3. Recursividade e algoritmos de busca e ordenação

1. Recursão
2. Ordenação
3. Busca

3.2 Ordenação

1. Ordenação por seleção (Selection Sort)
2. Ordenação por bolha (Bubble Sort)
3. Ordenação por Inserção (Insertion Sort)
4. Ordenação “rápida” (Quick Sort)

Problema

- Dado um vetor:

  $a_1, a_2, a_3, \ldots, a_{n-1}, a_n$

- Faça permutações nos elementos para obter um novo vetor:

  $a'_1, a'_2, a'_3, \ldots, a'_{n-1}, a'_n$

  de modo que

  $a'_1 \leq a'_2 \leq \ldots \leq a'_{n-1} \leq a'_n$

Ordenação por trocas

- Ordena o vetor permutanto, ou seja, **trocando posições** de seus elementos
- Existem diversos algoritmos de ordenação por trocas
- O **método da bolha** é um dos mais conhecidos, e fáceis de entender e implementar

3.2 Ordenação

1. **Ordenação por seleção (Selection Sort)**
2. **Ordenação por bolha (Bubble Sort)**
3. **Ordenação por Inserção (Insertion Sort)**
4. **Ordenação “rápida” (Quick Sort)**

Ordenação por seleção

- Repare que o algoritmo anterior pode permutar dois elementos vizinhos v[i] e v[i+1] caso v[i]> v[i+1]
- Até aí tudo bem, mas que dizer se nenhum desses valores for o valor máximo do vetor?
- Não seria melhor escolher o valor máximo, e levar (somente) esse valor para o final do vetor?
- Assim, teríamos apenas uma troca

Método de seleção direta

Vetor inicial:

7	5	9	3	2

Identificamos o maior valor: v[2]

E o colocamos no final do vetor

7	5	2	3	9

Agora procure o 2º maior elemento e leve-o para a penúltima posição ...

Ilustração – Seleção Direta

7	5	9	3	2	disposição inicial
7	5	2	3	9	1ª iteração
3	5	2	7	9	2ª iteração
3	2	5	7	9	3ª iteração
2	3	5	7	9	4ª iteração

Algoritmo

```
void Selecao (int v[], int n) {
   int i, j, aux, imaior;
    for (j=n-1; j>0; j--) {
        imaior = 0;
        for(i=1; i<=j; i++)
           if (v[i] > v[imaior])
              imaior = i;
        aux = v[j];
        v[j] = v[imaior];
        v[imaior] = aux
   }
}
```


Eficiência

- Quantas comparações o algoritmo faria:
  - No pior caso ?
  - No melhor caso ?
- Quantas trocas o algoritmo faria:
  - No pior caso ?
  - No melhor caso ?

Eficiência de algoritmos

- O que significa ser eficiente?
  - **Não desperdiçar o tempo ...**
- Como medir, então, o tempo gasto (ou perdido) por um algoritmo, se:
  - O tamanho do problema pode variar?
  - Isso depende do particular computador no qual iremos executar o algoritmo?
- Qual ao melhor maneira de medir o tempo de execução de um algoritmo?

Medindo o Tempo de Execução

- Não vamos medir tempo em segundos, minutos, horas ...
- Mas sim, em **número de instruções executadas !!!**
- Exemplo: para ordenar um vetor de **N** elementos, um certo algoritmo executa **3*N** instruções:
  - Se um certo computador leva 10^{-3} segundos para executar uma instrução, então vetor de **1000** elementos será ordenado em **3 segundos**, usando tal algoritmo

Medindo o Tempo de Execução

- Vejamos se essa medida é realmente adequada ...
- O que aconteceria se:
  - O mesmo algoritmo executasse em um computador 10 vezes mais rápido?
  - O mesmo algoritmo recebesse um vetor 100 vezes maior?

Profa. Dra. Laura Rodríguez Estruturas de dados e algoritmos 27

Medindo o Tempo de Execução

- Vejamos:
  - Um computador 10 vezes mais rápido levaria 10^{-4} segundos por instrução.
  - E um vetor 100 vezes maior teria **100 mil** elementos.
  - O tempo (cronológico) total seria = numero de instruções executadas * tempo por instrução = 3 * 100 mil * 10^{-4} = 3 * 10^{5} * 10^{-4} = **30 segundos**

Profa. Dra. Laura Rodríguez Estruturas de dados e algoritmos 28

Medindo o Tempo de Execução

- Repare que, tanto no caso I, quanto no caso II, apesar do tempo cronológico variar (3 segundos, e 30 segundos, respectivamente), o tempo de execução **T(n)** foi sempre igual a **3*N**

Obs.: Os computólogos costumam chamar de **T(N)**, o tempo gasto por um certo algoritmo para resolver um problema de tamanho **N**.

Comportamento assintótico

- Assim, quando medirmos a eficiência de um algoritmo, achando o seu **T(N)**, não importa:
  - Em qual computador iremos executar o algoritmo
  - Qual o particular valor de N

Exemplo

- Imagine que voce conhece 4 algoritmos de ordenação de vetores, cada um com a sua eficiência:
  - $T1(N) = N$
  - $T2(N) = 3*N$
  - $T3(N) = N * LOG_2^N$
  - $T4(N) = N^2$

Exemplo

N	T1(N)	T2(N)	T3(N)	T4(N)
10	10,0	30	33	100
100	100,0	300	664	10000
1000	1000,0	3000	9966	1000000
1000000	1000000,0	3000000	19931569	1000000000000
1000000000	1000000000,0	3000000000	29897352854	1000000000000000000

3.2 Ordenação

1. **Ordenação por seleção (Selection Sort)**
2. **Ordenação por bolha (Bubble Sort)**
3. **Ordenação por Inserção (Insertion Sort)**
4. **Ordenação “rápida” (Quick Sort)**

Método da bolha

- Percorra o vetor da esquerda para direita:
  - comparando elementos vizinhos estiverem fora de ordem, troque-os de posição
- Considere o vetor inicial:

7	**5**	**9**	**3**	**2**

comparando v[0] com v[1] $\Rightarrow$ **troca**
comparando v[1] com v[2] $\Rightarrow$ **não troca**
comparando v[2] com v[3] $\Rightarrow$ **troca**
comparando v[3] com v[4] $\Rightarrow$ **troca**

- O vetor ficará

5	**7**	**3**	**2**	**9**

Ilustração – Método da Bolha

7	5	9	3	2	disposição inicial
5	7	3	2	9	1ª iteração
5	3	2	7	9	2ª iteração
3	2	5	7	9	3ª iteração
2	3	5	7	9	4ª iteração

Algoritmo - Bolha

- Eis aqui o algoritmo

```
void Bolha (int v[], int n) {
   int i, j, aux;
    for (j=n; j>0; j--)
        for(i=0; i<j; i++)
           if (v[i] > v[i+1]) {
              aux = v[i];
              v[i] = v[i+1];
              v[i+1] = aux;
           }
}
```

Exemplo

- Eis o programa que chama a função Bolha

```
int main()
{
    int v[N]={33,17,-11,45,1,26,54,67,21,10}, i;
    Bolha(v,N);
    for (i=0; i < N; i++)
        cout << "\n v["<<i<<"]="<<v[i];

    cin >> i;
    return 0;
}
```


Eficiência

- Quantas comparações o algoritmo faria:
  - No pior caso ?
  - No melhor caso ?
- Quantas trocas o algoritmo faria:
  - No pior caso ?
  - No melhor caso ?

3.2 Ordenação

1. **Ordenação por seleção (Selection Sort)**
2. **Ordenação por bolha (Bubble Sort)**
3. **Ordenação por Inserção (Insertion Sort)**
4. **Ordenação “rápida” (Quick Sort)**

3.2.3 Ordenação por inserção (Insertion Sort)

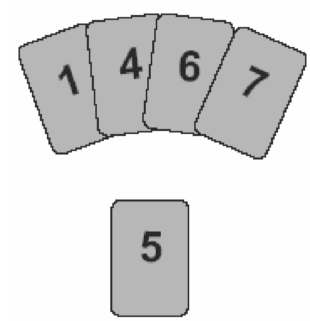
- Efectua apenas uma passagem pela tabela.
  - Cada novo elemento analisado é inserido na sua posição correcta (a procura é feita da direita para a esquerda).
  - Este método implica o deslocamento para a direita de alguns dos elementos já ordenados da tabela.
- O número total de movimentos depende da desordem que reina na tabela.

17	15	55	48	8

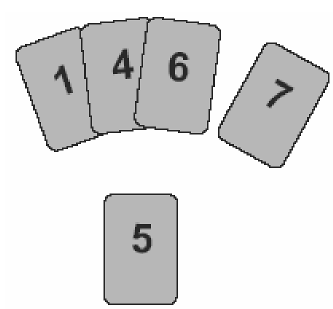
3.2.3 Ordenação por inserção (Insertion Sort)

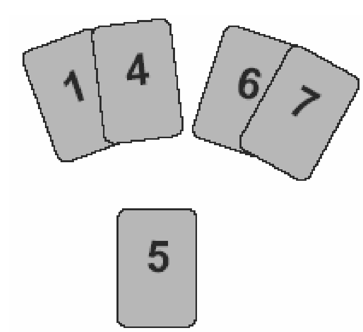
Funciona de forma idêntica à ordenação das cartas na nossa mão num qualquer jogo de cartas.

Para colocar uma nova carta na posição correcta temos que afastar as outras para arranjar espaço para ela.



3.2.3 Ordenação por inserção (Insertion Sort)

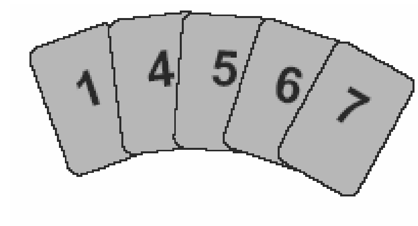




Insertion Sort

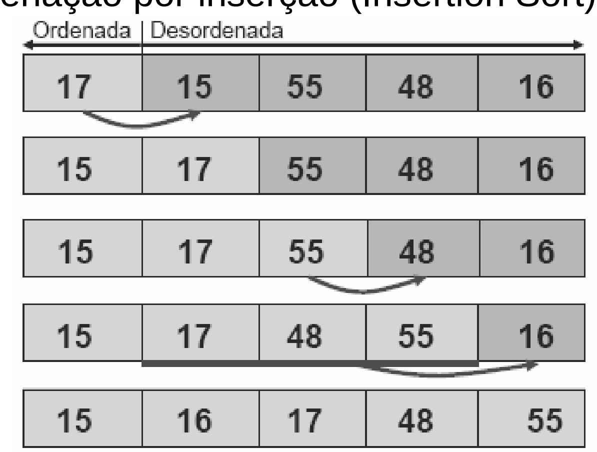
Funciona de forma idêntica à ordenação das cartas na nossa mão num qualquer jogo de cartas.

Para colocar uma nova carta na posição correcta temos que afastar as outras para arranjar espaço para ela.



Insertion Sort

- Ordenação por inserção (Insertion Sort):



Algoritmos de Ordenação

Ordenação por inserção (Insertion Sort):

```
void OrdenacaoInsercao(int Tab[],int Comp)
{
   int I,J;
   int Aux;
   for (I = 1; I < Comp; I++) {
     Aux = Tab[I];
     for (J = I; (J > 0) && (Aux < Tab[J - 1]); J - -)
               Tab[J] = Tab[J-1];
     Tab[J] = Aux;
   }
}
```


Algoritmos de Ordenação

- O número de comparações necessárias para ordenar uma tabela com N elementos é no pior dos casos (tabela ordenada pela ordem inversa):

$$\text{Comp} = 1 + 2 + \ldots + (N-2) + (N-1)$$

$$\boxed{\text{N}^{\text{o}} \text{ de Comp} = \frac{N * (N-1)}{2} = O(N^2)}$$

- O número de deslocamentos (não existem trocas) no pior dos casos é igual ao número de comparações.

3.2 Ordenação

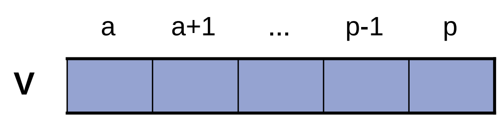
1. **Ordenação por seleção (Selection Sort)**
2. **Ordenação por bolha (Bubble Sort)**
3. **Ordenação por Inserção (Insertion Sort)**
4. **Ordenação “rápida” (Quick Sort)**

3.2.4 Quicksort

- Proposto em 1959
- Tempos de execução:
  - $O(N^2)$ operações, no pior caso
  - no caso médio, é esperado que execute $O(N * \lg N)$ operações
- O Quicksort, como o próprio nome diz, é um algoritmo rápido: apesar do pior caso ser lento, isto na prática não ocorre

3.2.4 Quicksort

- Método baseado no princípio de "divisão e conquista", com as 3 fases:
  - Dividir
  - Conquistar
  - Combinar
- Entrada: o vetor V[a..p], a ser ordenado

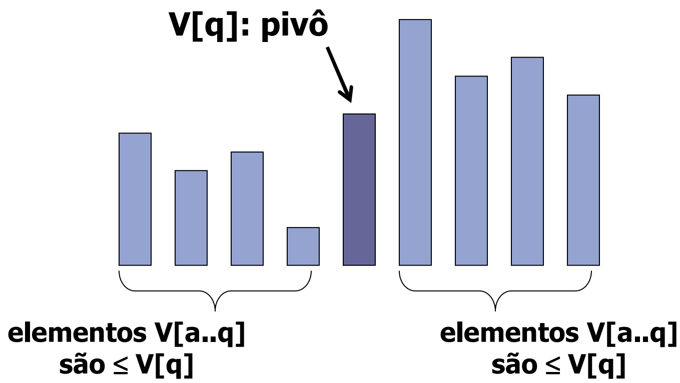


Quicksort – 1a fase

- **Dividir**:
  - Se o tamanho do vetor for 1 ou 0, então nada resta a ser feito (Caso trivial)
  - Se tiver 2 ou mais elementos, escolha um elemento **x**, chamado de **pivô** (geralmente é o primeiro ou o último elemento do vetor). Reorganizar o vetor, de modo que:
    - todos os elementos menores ou iguais que o pivô fiquem à sua esquerda; e
    - todos os elementos maiores ou iguais ao pivô fiquem à sua direita

Quicksort – 1a fase:

- Resultado: vetor V reorganizado



Quicksort – 2a e 3a fases

- **Conquistar:**
  - ordenar o subintervalo esquerdo do vetor V[a..q-1], chamando o quicksort recursivamente
  - ordenar o subintervalo direito do vetor V[q+1..r], chamando o quicksort recursivamente
- **Combinar:** como os subintervalos já foram ordenados no próprio vetor, nada mais resta a ser feito: V[p..r] já está ordenado

Um algoritmo de particionamento

- Escolha um pivô (vamos chamá-lo de ***x***). Posicione as variáveis ***esq*** e ***dir*** nos dois extremos de V, movendo-os um em direção ao outro da seguinte forma:
    1. faça ***esq*** avançar para a direita, até encontrar o primeiro valor maior que o pivô
    2. faça ***dir*** avançar para a esquerda, até encontrar o primeiro valor menor ou igual ao pivô
    3. se ***esq < dir***, então troque V[esq]↔V[dir]
- quando ***esq*** e ***dir*** se cruzarem, o algoritmo termina, retornando ***dir*** como resposta

1a Fase: O Particionamento

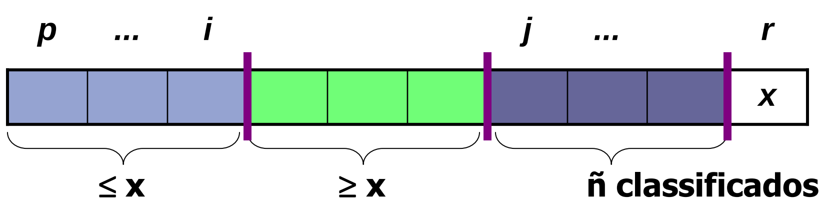
- Primeiramente, escolha um pivô (o último elemento é um bom candidato):

p							*r*
2	**8**	**7**	**1**	**3**	**5**	**6**	**4**
							pivô

- x = v[r] = 4 (este é o pivô)

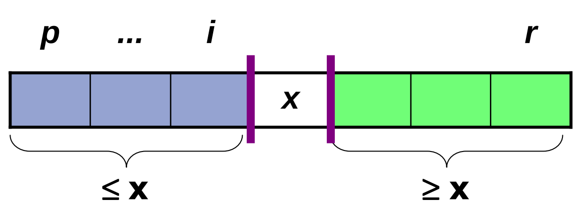
Particionamento

- Em seguida, particione o vetor em 4 intervalos
  - elementos menores que o pivô **V[p..i]**
  - elementos maiores que o pivô **V[i+1..j-1]**
  - elementos ainda não classificados **V[j..r-1]**
  - o pivô **V[r]**

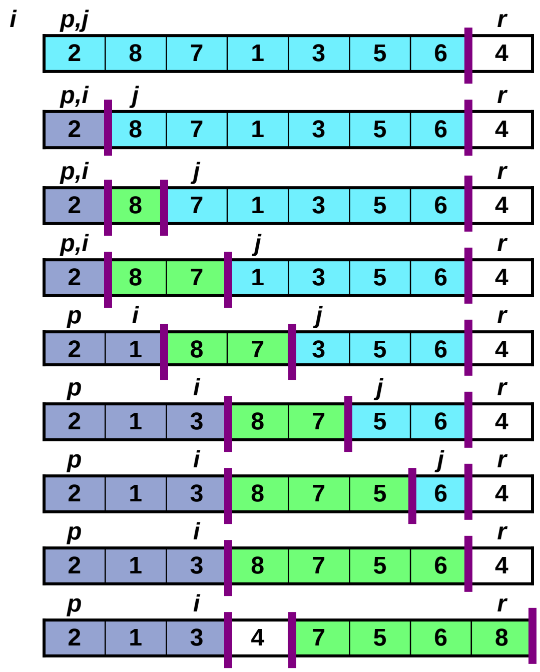


Particionamento

- O algoritmo termina quando o terceiro intervalo for nulo, isto é, todos os elementos tiverem sido classificados
- O vetor ficará assim:



Exemplo



Profa. Dra. Laura Rodríguez

Algoritmo Particionamento

```
int Partition (int V[], int p, int r)
{   int x, i, aux, j;
    x = V[r];
    i = p-1;
    for (j=p; j<r; j++)
        if (V[j] <= x) {
           i++;
           // troque v[i] e v[j]
           aux = V[i];
           V[i]= V[j];
           V[j] = aux;
          }
    // troque v[i+1] e v[r]
    aux = V[i+1];
    V[i+1] = V[r];
    V[r] = aux;
    return i+1;
}
```

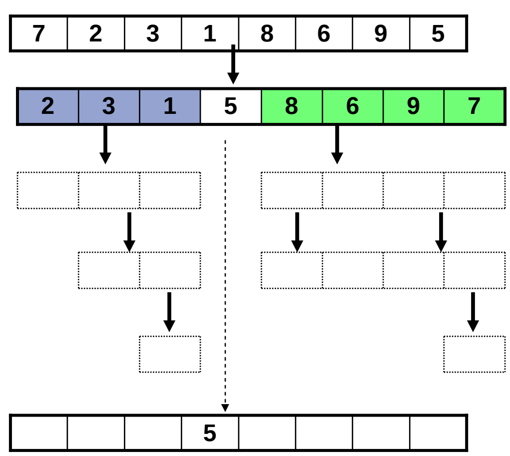
2a Fase: Conquista

- O vetor ainda não foi ordenado, mas se esse mesmo procedimento for aplicado recursivamente a cada um de seus subintervalos ...

```
void Quicksort (int V[], int p, int r)
{
    if (p < r) {
       int q = Partition (V, p, r);
       Quicksort (V, p, q-1);
       Quicksort (V, q+1, r);
     }
}
```

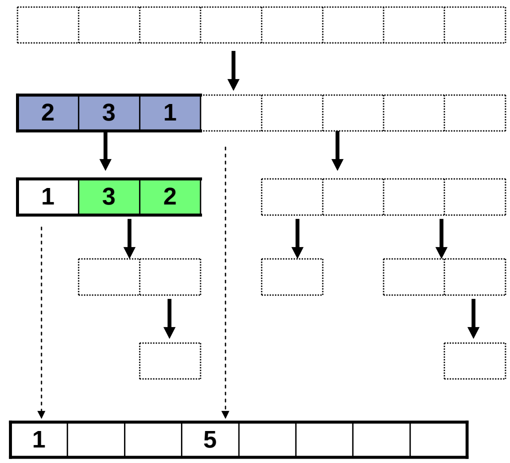

Um exemplo completo

- O valor 5 é escolhido como pivô
- Em seguida, o quicksort será aplicado a cada um dos subintervalor



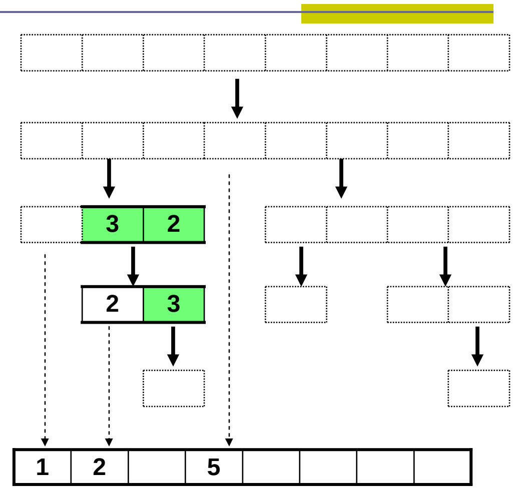
Exemplo

- O primeiro subintervalo é pivoteado ...
- E dividido:
  - intervalo esquerdo vazio
  - intervalo direito com 2



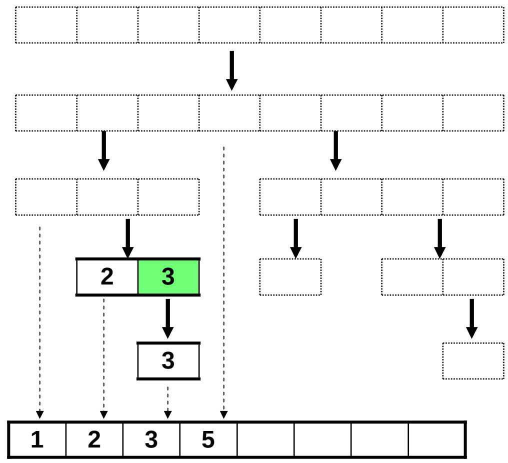
Exemplo

- 2 é o pivô
- intervalo esquerdo é vazio
- e o direito tem um elemento...



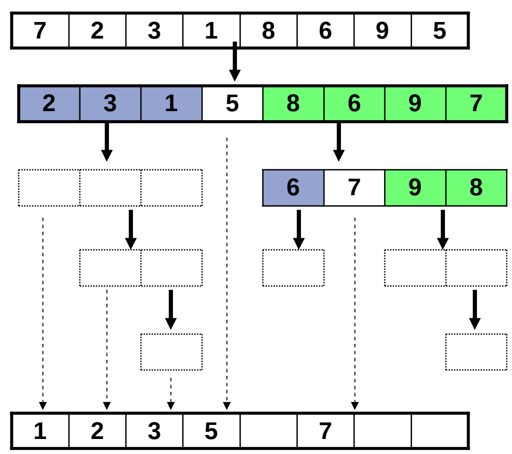
Exemplo

- finalmente, o ultimo intervalo é resolvido



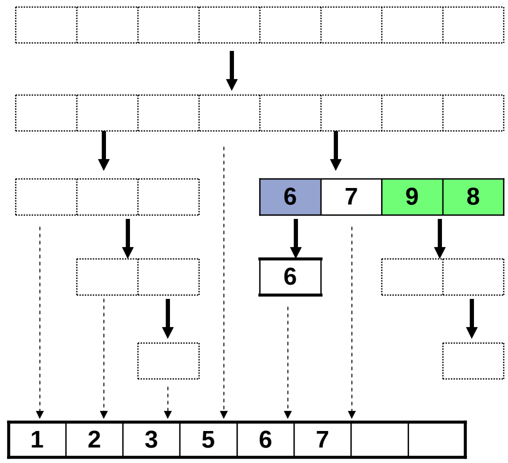
Exemplo

- Agora, resta o lado direito do vetor ...
- O pivô é 7



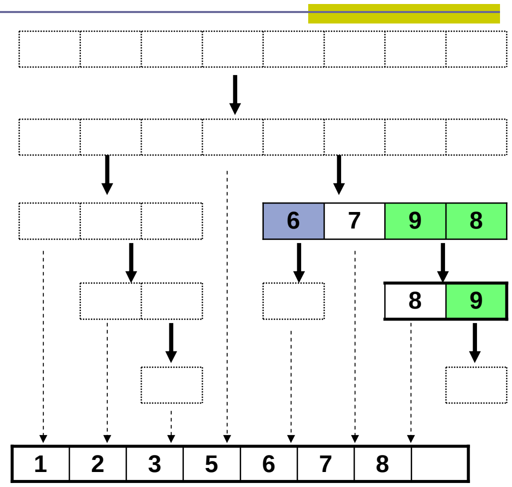
Exemplo

- o subintervalo esquerdo possui um unico elemento



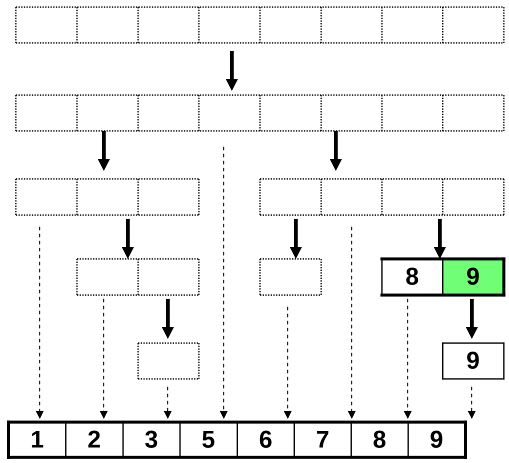
Exemplo

- no subintervalo direito, 8 é o pivô



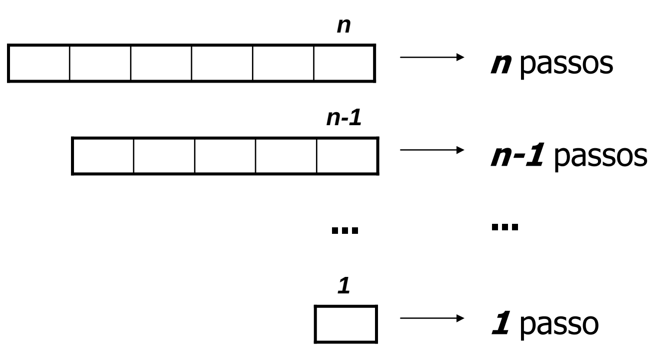
Exemplo

- o subintervalo direito tem um unico elemento
- o algoritmo termina



Eficiência do Quicksort

- **Pior caso**: o pior caso ocorre quando o pivô escolhido ocasiona uma divisão em dois intervalos de tamanhos **ZERO** e **N-1**



Eficiência do Quicksort

- Assim:

$$T(n) = \sum_{1}^{n} i$$

$$T(n) = \frac{1+n}{2} n = \frac{n^2 + n}{2}$$

- Portanto, no pior caso, T(N) = *O* (N^2)

Eficiência do Quicksort

- **No melhor caso**, o algoritmo irá dividir em dois subintervalos de tamanhos iguais (cada um com aproximadamente *n/2* operações)
  - e cada um dos subintervalos será dividido em dois outros de igual tamanho ...
  - como o número de particionamentos é dado por log_2^N e cada nível da árvore do quicksort faz N operações, então T(N) = N.$\log_2^N$

Eficiência do quicksort

- Na prática, o comportamento esperado do quicksort (**caso médio**) se aproxima muito mais do melhor caso, que do pior caso, isto é, $T(N) = N.\log_2^N$
- Em resumo, apesar do Quicksort, no pior caso ser *O* (N^2), isso na prática não ocorre, e ele tem comportamento proximo de $N.\log_2^N$, o que é bem eficiente.

3. Recursividade e algoritmos de busca e ordenação

1. **Recursão**
2. **Ordenação**
3. **Busca**

3.3 Busca

1. **Busca sequencial**
2. **Busca Binária**

3.3.1 Busca Sequencial

- Problema:
  - Determinar se um certo valor ***x*** encontra-se entre os elementos de um dado vetor ***V[0..***n-1***]***.
  - Em outras palavras, verificar se existe um certo ***i***, tal que ***x = V[i]***, para algum 0≤*i* ≤ N-1

x = 498

0	1	2	3	4		n-1	...	...
253	781	215	398	442	...	327		

Primeira tentativa

- O que voce acha desta solução?

```
BUSCA(x, V[1..n])
  i ← 1;
    enquanto i ≤  n
      se   x = V[i]
        então escreva “ACHOU”
        senão escreva “NÃO ACHOU”
```


Primeira solução

- Uma forma bastante simples (e elegante) de resolver esse problema:

```
BUSCA(x, V[1..n])
     i ← 1;
     enquanto i ≤  n  e  V[i] ≠ x
          faça  i ← i + 1
     se i > n
          então devolva NÃO ACHOU
          senão devolva ACHOU
```

Busca em vetor ordenado

- Escreva um algoritmo para procurar um dado x em um vetor $V[1..n]$, considerando que o vetor já foi previamente ordenado, isto é : $V[1] \leq V[2] \leq . . . \leq V[n]$.
  - **Dica**: O fato de o vetor estar ordenado ajuda bastante, e reduz o tempo de busca. O algoritmo percorre o vetor até que uma das condições abaixo aconteça :
    - Encontrar o fim do vetor (ou seja, a posição n) sem ter encontrado x.
    - Encontrar x em uma posição qualquer do vetor ($V[i]=x$, para algum i)
    - Encontrar um elemento $V[i] > x$, o que significa que x não pode mais ser encontrado no vetor, pois $x < V[i] \leq V[i+1] \leq . . . \leq V[n]$

Eficiência de Busca

- Analise os algoritmos anteriores, e determine quantas tentativas (comparações) o algoritmo deverá fazer para encontrar x:
  - no melhor caso
  - no pior caso
  - no caso médio

Eficiência de Busca

- Mas afinal, qual seria esse melhor caso?
  - No melhor caso, podemos imaginar que o valor ***x*** encontra-se logo na primeira posição do vetor ⇒ e portanto, uma única comparação será suficiente
- E no pior caso?
  - No pior caso, ***x*** não existe em ***V[0..N-1]***, e portanto, o algoritmo precisa procurar ***x*** em todas as posições do vetor ⇒ fazendo, portanto, ***N*** tentativas

Eficiência de Busca

- E no caso médio?
  - Supondo que todos os elementos do vetor tenham a mesma probabilidade de acesso, o número médio de comparações pode ser assim definido:

$$C = \sum_{1}^{n} i.p(i) = \sum_{1}^{n} i.\frac{1}{n} = \frac{1}{n} \cdot \left[\frac{1+n}{2} \cdot n \right] = \frac{1+n}{2}$$

onde:

i é o número de comparações

P(i) é e probabilidade de fazermos *i* comparações

n é o comprimento do vetor

3.3 Busca

1. **Busca sequencial**
2. **Busca Binária**

3.3.2 Busca Binária

- Considere o vetor inicialmente ordenado, isto é, **v[0]≤v[1]≤...≤v[n-1]**
- Verifique qual a posição situada "no meio" do vetor V – chamemos de posição **k**

0	1	...	...	k	...	...	n-2	n-1
71	94	130	321	429	515	600	770	812

x = 320

3.3.2 Busca Binária

- Compare **x** com o valor da posição **k.** Há 3 possíveis situações:
  - x = V[k] ⇒ ACHOU
  - x < V[k] ⇒ descarte a metade direita de V
  - x > V[k] ⇒ descarte a metade esquerda de V

0	1	...	...	k	...	...	n-2	n-1
71	94	130	321	429	~~515~~	~~600~~	~~770~~	~~812~~

x = 320

3.3.2 Busca Binária

- Repita o processo, reduzindo o vetor pela metade, até que:
  - **x** seja encontrado, ou
  - que o intervalo de busca termine

Busca Binária

```
inf = 0;
sup = N-1;
med = (inf+sup)/2;
enquanto (inf <= sup && x!=v[med])
   inicio
      se (x<v[med])
         sup = med - 1;
      senão se (x>v[med])
                inf = med + 1;
      med = (inf+sup)/2;
   fim;
se (inf > sup)
   exibir(" \n  VALOR NAO ENCONTRADO !!!“);
senão exibir(“\n ENCONTRADO NA POSICAO “, med);
```


Eficiência

- Vamos medir a eficiência desse algoritmo
- Para um vetor de **N** elementos, quantas comparações são esperadas?
  - vamos analisar o pior caso: teremos de ir dividindo o vetor ao meio, por diversas vezes, até que o intervalo de busca tenha 1 elemento (e eu encontre o valor **x**), ou que ele se anule.
  - assim, o número de comparações é igual ao número de sucessivas divisões pela metade, até que o intervalo de busca se anule: $\mathbf{log_2^N}$

Exercícios

1. Escreva um programa em C, que implementa a busca binária.
2. Faça uma estimativa sobre o número máximo de tentativas (comparações) desse algoritmo

Exercícios

3. Já vimos que a busca binária é melhor que a busca sequencial. Mas afinal, quanto ela é melhor? Faça uma estimativa, supondo que o comprimento do vetor seja:

N	Sequencial	Binária
10	5	
1.000		
10^6		
10^9		
10^{12}		