Algoritmos de Ordenação

Marcelo K. Albertini

Aula de hoje

Nesta aula veremos:

- selection sort
- insertion sort
- Paradigma: Divisão e Conquista
- quick sort
- merge sort

Problema: Ordenação Interna usando Comparações

ordenar em memória principal

- **Entrada:** vetor em memória principal, inicializado com elementos
- **Saída:** vetor com elementos em ordem crescente (ou decrescente)

3	1	6	2	4	8

Como fica ordenado?

1	2	3	4	6	8

Vetores e Ordenação: exemplos

Exemplos:

- Números inteiros ou ponto flutuante
- Vetor de strings
- Tipos compostos: necessário definir função para comparar
  - **int compare(ITEM item1, ITEM item2);**

Exemplo

```
int compare(Aluno a) {
    if (this.media > a.media) {
        return 1;
    }
    else if (this.media == a.media) {
        return (-1);
    } else {
        return(0);
    }
}
```

Algoritmo de ordenação usando comparações

Definição de ordenação

Sequência de comparações e trocas de posição entre elementos para obter vetor ordenado.

Complexidade

- trocas
- comparações
- memória extra? ou in-place?

Algoritmo Ideal de Ordenação por Comparações

- Pior caso de trocas é $O(n)$
- Pior caso de comparações é $O(n \log n)$
- Opera com dados no lugar, ou seja, usa $O(1)$ de espaço extra
- Adaptabilidade: faz $O(n)$ operações se dados (quase) ordenados
- Estabilidade: elementos iguais não mudam de ordem entre si (útil para dados compostos)

Não há algoritmo que atende a isso tudo. Escolha depende da aplicação.

Ordenação por seleção

Funcionamento

1. seleciona menor elemento de região não ordenada
2. troca o primeiro elemento da região pelo menor elemento
3. diminui tamanho da região não ordenada

Algoritmo: ordenação por seleção

```
// divide array em 2 subarrays
// aumenta subarray ordenado um elemento por vez
// selecionando o menor elemento no subarray à direita
void selectionSort(int[] vet){
  for (int i = 0; i < vet.length-1; i++) {
    int menorI = i;

    for (int j = i+1; j < vetor.length; j++) {
      if (vet[j] < vet[menorI]) {
        menorI = j; // aqui temos um novo menor
      }
    }

    if (menorI != i) {
      int aux = vet[i]; //troca com o menor atual
      vet[i] = vet[menorI];
      vet[menorI] = aux;
    }
  }
}
```

Algoritmo: ordenação por seleção

```
void selectionSort(int[] vet){
 for (int i = 0; i < vet.length-1; i++){
   int menorI = i;

   for (int j = i+1; j < vet.length; j++){
     if (vet[j] < vet[menorI]) //comparação
       menorI = j;
   }

   if (menorI != i) { //troca
     int aux = vet[i];
     vet[i] = vet[menorI];
     vet[menorI] = aux;
   }
 }
}
```

- Quantas comparações?
- Quantas trocas? $O(.)$,$\Omega(.)$?

Algoritmo: ordenação por inserção

- Começa com:
  - um subarray ordenado vazio e
  - um subarray desordenado do tamanho do array
- A cada iteração (uma para cada número no array)
  - Aumenta o subarray ordenado a cada iteração, um elemento por vez
  - Insere novo elemento na sua posição correta

Algoritmo: ordenação por inserção

```
void insertionSort(int[] vetor) {

 int n = vetor.length;
 for (int j = 1; j < n; j++){
  int chave = vetor[j];
  int i = j - 1;

   // procura lugar de insercao e desloca numeros
  while (i >= 0 && vetor[i] > chave) {
    vetor[i+1] = vetor[i];
    i = i - 1;
  }
  vetor[i+1] = chave;
 }
}
```

Algoritmo: ordenação por inserção

```
insertionSort(int[] vetor){

int n = vetor.length;
for (int j = 1; j<n; j++){
 int chave = vetor[j];
 int i = j - 1;

 // desloca numeros
 while (i >= 0 &&
          vetor[i] > chave){
   vetor[i+1] = vetor[i];
   i = i - 1;
 }
//insere chave
    vetor[i+1] = chave;
 }
}
```

chave $= 5$

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave $= 6$

5	6	2	1	9	6

chave $= 2$

5	6	2	1	9	6
5	6	6	1	9	6
5	5	6	1	9	6
2	5	6	1	9	6

chave $= 1$

2	5	6	1	9	6
2	5	6	6	9	6
2	5	5	6	9	6
2	2	5	6	9	6
1	2	5	6	9	6

chave $= 9$

1	2	5	6	9	6

chave $= 6$

1	2	5	6	9	6
1	2	5	6	9	9
1	2	5	6	6	9

Algoritmo: ordenação por inserção

```
insertionSort(int[] vetor){

int n = vetor.length;
for (int j = 1; j<n; j++){
 int chave = vetor[j];
 int i = j - 1;

 // desloca numeros
 while (i >= 0 &&
        vetor[i] > chave){
  vetor[i+1] = vetor[i];
  i = i - 1;
 }
//insere chave
   vetor[i+1] = chave;
 }
}
```

- Melhor caso?
- Pior caso?

Divisão e conquista

- Divisão: quebrar problemas em sub-problemas
- Conquista: resolver sub-problemas (até caso base)
- Combinar: usar soluções de sub-problemas para obter solução do problema maior

Paradigma de Divisão e Conquista

- Quebrar problemas maiores em menores
  - Ao contrário de programação dinâmica, problemas menores não se repetem
- Custo total depende da relação entre a taxa de criação de subproblemas e o custo de combinar subsoluções em soluções maiores
  - Teoremas de análise de Divisão e Conquista
  - Exemplo:
    - $T(N) = aT(N/b) + N$
      Se $a = b,\ T(N) = N \log_b N$
      Se $a < b,\ T(N) \sim \frac{b}{b-a} N$
      Se $a > b\ T(N) \sim \frac{a}{a-b} (b/a)^{\{\log_b a\}} N^{\log_b a}$

Exemplos de algoritmos de divisão e conquista

- Quicksort
- Mergesort
- Multiplicação de matrizes de Strassen
- Multiplicação de números de Karatsuba
- Transformada Rápida de Fourier

Problema: busca em array ordenado

- **Entrada:** array ordenado e elemento x a ser buscado
- **Saída:** posição onde x se encontra ou valor negativo
- Ideia
  - Comparar x com elemento na metade do array
  - Usar comparação para decidir se busca segue à esquerda ou direita

Busca binária

```
int pos(int chave, int v[]) {
  int inf = 0;
  int sup = v.length - 1;
  while (inf <= sup) {
    // chave está em v[inf...sup] ou não existe
    int meio = inf + (sup-inf) /2;
    if (chave < v[meio]) sup = meio-1;
    else if (chave > v[meio]) inf = meio +1;
    else return meio;
  }
  return -1;
}
```

$B_N = B_{N/2} + 1$ para $N > 1$ com $B_1 = 1$

https://www.cs.usfca.edu/~galles/visualization/Search.html

Problema: achar mediana (*selection*)

- **Entrada:** array de números `S` e inteiro k indicando o ranking do elemento a ser selecionado
  - o mínimo tem ranking $k = 1$
  - a mediana tem ranking $k = (int)N/2$
  - o máximo tem ranking $k = N$
- **Saída:** retornar o elemento com ranking k

Problema: achar mediana (*selection*)

- A cada etapa da recorrência dividir array S em subarrays com números menores que v, S_L, e maiores que v S_R

$$selection(S,k) = \begin{cases} selection(S_L, k) & \text{se } k \leq |S_L| \\ v & \text{se } |S_L| < k \leq |S_L| + |S_v| \\ selection(S_R, k - |S_L| - |S_v|) & \text{se } |S_L| + |S_v| \end{cases}$$

- Se v for bem escolhido, subarrays diminuem na metade a cada recursão

Quicksort

- Inventado por Tony Hoare, Moscou, União Soviética, 1960
- Extremamente difundido: C qsort, java primite types
- Roda rápido, em média $O(n \log n)$, com pouca memória
- Estudado extensivamente (ver livro de Sedgewick e Flajolet)
- Pior caso $O(n^2)$ raríssimo (se bem implementado)

Quicksort

Ideia

1. **Desordenar** o vetor
2. **Particionar** tal que para algum elemento na posição `j` (pivot)
   - valor em `v[j]` está na posição correta
   - todos os valores à esquerda de *j* são menores que `v[j]`
   - todos os valores à direita de *j* são maiores que `v[j]`
3. **ordenar** cada pedaço **recursivamente**, mas sem cópia do vetor

entrada	Q	U	I	C	K	S	O	R	T
desordenado	K	R	T	Q	S	O	I	U	C
partição	I	C	K	Q	U	R	T	S	O
ordena esq.	C	I	K	Q	U	R	T	S	O
ordena dir.	C	I	K	O	Q	R	S	T	U
resultado	C	I	K	O	Q	R	S	T	U

Desordenação

Desordenação com complexidade de tempo $\Theta(n)$ e espaço $\Theta(n)$ pode ser feita com o algoritmo de desordenação de Knuth.

```java
public static void shuffle(int[] v) {

  Random r = new Random(System.currentTimeMillis());
  int aux;

  // desordena um elemento por vez
  for (int i = 0; i < v.length; i++) {
    int ir = r.nextInt(i+1); // sorteio aleatorio

    aux = v[i]; // troca com posicao aleatoria
    v[i] = v[ir];
    v[ir] = aux;
   }
}
```

Partição

Objetivo

Dividir vetor em duas regiões separadas pelo `pivot`.

- A região anterior ao `pivot` consiste de elementos menores ou iguais a ele.
- A região posterior ao `pivot` consiste de elementos maiores a ele.

Guardamos duas variáveis de índice: da esquerda `i` e da direita `j`.

$v[p]$ é o elemento pivot.

Antes: | $v[p]$ | $v[\dots]$ |

Durante: | $v[p]$ | $v[\dots] <= v[p]$ | $v[i] \dots v[j]$ | $v[\dots] > v[p]$ |

Depois: | $v[\dots] <= v[p]$ | $v[p]$ | $v[\dots] > v[p]$ |

Partição

Região do vetor a ser particionada é delimitada por `inf` e `sup`

```
particao(int v[], int inf, int sup) {
  int i = inf, j = sup+1, aux;

  while(true) {
    while (v[++i] < v[inf]) // movimento da esq.
        if (i == sup)
          break;

    while (v[inf] < v[--j]) // mov. da direita
        if (j == inf)
          break;

    if (i >= j) break;

    troca(v, i, j);
  }

  troca(v, inf, j); // troca posicao do pivot

  return j; // retorna posicao do pivot
}
```

```c
int particao(int v[], int
    inf, int sup) {
int i=inf, j=sup+1;

while(true) {
 while (v[++i] < v[inf])
  if (i == sup) break;

 while (v[inf] < v[--j])
  if (j == inf) break;

 if (i >= j) break;

 troca(v, i, j);
}

troca(v, inf, j);// pivot

return j; // retorna pivot
}
```

$p = 0,\ v[p] = 5,\ inf = 0,\ sup = 8$

5	6	7	8	1	4	3	2	9
5	6	7	8	1	4	3	2	9
5	2	7	8	1	4	3	6	9
5	2	7	8	1	4	3	6	9
5	2	3	8	1	4	7	6	9
5	2	3	8	1	4	7	6	9
5	2	3	4	1	8	7	6	9
1	2	3	4	5	8	7	6	9

Quicksort

```
public static void quicksort(int[] v, int n) {
  shuffle(v); // desordenar é rápido
  sort(v, 0, v.length-1);
}

public static void sort(int[] v, int inf, int sup) {
  if (inf >= sup) {
    return;
  } else {
     int j = particao(v, inf, sup); // ordena pivot
     sort(v, inf, j-1); // ordena menores
     sort(v, j+1, sup); // ordena maiores
  }
}
```

$p = 0,\ v[p] = 5,\ inf = 0,\ sup = 8$

5	6	7	8	5	4	3	2	9
5	6	7	8	5	4	3	2	9
5	2	7	8	5	4	3	6	9
5	2	7	8	5	4	3	6	9
5	2	3	8	5	4	7	6	9
5	2	3	8	5	4	7	6	9
5	2	3	4	5	8	7	6	9
5	2	3	4	5	8	7	6	9

$p = 0,\ v[p] = 5,\ inf = 0,\ sup = 3$

5	2	3	4	5	8	7	6	9
4	2	3	5	5	8	7	6	9

$p = 0,\ v[p] = 4,\ inf = 0,\ sup = 2$

4	2	3	5	5	8	7	6	9
3	2	4	5	5	8	7	6	9

$p = 0,\ v[p] = 3,\ inf = 0,\ sup = 1$

3	2	4	5	5	8	7	6	9
2	3	4	5	5	8	7	6	9

$p = 5,\ v[p] = 8,\ inf = 5,\ sup = 8$

2	3	4	5	5	8	7	6	9
2	3	4	5	5	6	7	8	9

$p = 5,\ v[p] = 6,\ inf = 5,\ sup = 6$

2	3	4	5	5	6	7	8	9
2	3	4	5	5	6	7	8	9

Análise de complexidade

O custo de usar o quicksort em n é o custo de particionar esses elementos com αn operações mais o custo de aplicar o quicksort nos dois subvetores resultantes, com k e $n - k$ elementos.

Relação de recorrência

$$T(n) = T(k) + T(n-k) + \alpha n$$

Análise de pior caso

Pior caso ocorre quando o `pivot` for sempre o menor elemento do vetor. Ou seja, $k = 1$ em $T(n) = T(k) + T(n-k) + \alpha n$

Relação de recorrência: pior caso k=1

Divide array com $k = 1$	$T(n) = T(n-1) + T(1) + \alpha n$
Obtém eq. para $n - 1$	$T(n) = [T(n-2) + T(1) + \alpha(n-1)] + \alpha n$
Reorganiza	$T(n) = T(n-2) + 2T(1) + \alpha(n-1+n)$

Análise de pior caso k=1: continuando

$$
\begin{array}{rr}
\text{Para } k = 1 & T(n) = T(n-1) + T(1) + \alpha n \\
\text{Eq. de } n-1 & = [T(n-2) + T(1) + \alpha(n-1)] + \alpha n \\
\text{Organiza} & = T(n-2) + 2T(1) + \alpha(n-1+n) \\
\text{Eq. de } n-2 & = [T(n-3) + T(1) + \alpha(n-2)] + 2T(1) + \alpha(n-1) + \alpha n \\
\text{Organiza} & = T(n-3) + 3T(1) + \alpha[(n-2) + (n-1) + n] \\
\text{Eq. de } n-i & = T(n-i) + iT(1) + \alpha[(n-i+1) + \ldots + (n-1) + n] \\
\text{Soma} & = T(n-i) + iT(1) + \alpha \sum_{j=0}^{i-1}(n-j) \\
\text{Vai até } i = n-1 & = T(n-n+1) + (n-1)T(1) + \alpha \sum_{j=0}^{n-1-1}(n-j) \\
\text{Resultado} & = nT(1) + \alpha[(\sum_{j=1}^{n} j) - 1] \\
\text{Só o somatório} & \sum_{j=1}^{n} j = (n+1)n/2
\end{array}
$$

Análise de melhor caso

Melhor caso ocorre quando o `pivot` for sempre o elemento que divide o vetor na metade. Ou seja, $k = n/2$ em $T(n) = 2T(n/2) + \alpha n$

Relação de recorrência: melhor caso k=n/2

$$
\begin{aligned}
& & T(n) &= 2T(n/2) + \alpha n \\
&\text{Para } n/4 & &= 2(2T(n/4) + \alpha n/2) + \alpha n \\
&\text{Organizar} & &= 4T(n/4) + 2\alpha n/2 + \alpha n \\
& & &= 2^2 T(n/2^2) + 2\alpha n \\
&\text{Para } n/8 & &= 2^2[2(T(n/8) + \alpha n/8)] + 2\alpha n \\
&\text{Organizar} & &= 2^3 T(n/2^3) + 3\alpha n \\
&\text{Para } n/2^k & &= 2^k T(n/2^k) + k\alpha n \\
&\text{Até } n = 2^k, \text{ com } k = \log_2 n & T(n) &= nT(1) + \alpha n \log_2 n
\end{aligned}
$$

Resumo quicksort

- Desordenação em $\Theta(n)$
- Quicksort in-place (sem espaço extra)
- Análise de complexidade de pior caso
- Análise de complexidade de melhor caso

Mergesort

- ordenação estável
- fundamento para outros algoritmos
- autoria Von Neumann em 1945: EDVAC um dos primeiros computadores de propósito geral
- algoritmo que executa em tempo ótimo no pior caso

Mergesort

Ideia

1. dividir vetor em 2 metades
2. recursivamente ordenar cada metade
3. mesclar – merge – as duas metades ordenadas

entrada	5	1	3	6	4	2	9	0
ordena esquerda	1	3	5	5	4	2	9	0
ordena direita	1	3	5	6	0	2	4	9
antes do merge	1	3	5	6	0	2	4	9
ordenado	0	1	2	3	4	5	6	9

Implementação

```
// v: vetor sendo ordenado, aux: vetor auxiliar
// inf: posicao inferior sendo trabalhada
// med: posicao mediana, sup: posicao superior
merge(int[] v, int[] aux, int inf, int med, int sup) {
  for (int k = inf; k <= sup; k++)
    aux[k] = v[k]; // copia dos valores para auxiliar

  int i = inf, j = med+1;
  for (int k = inf; k <= sup; k++) {
    // se subvetor da direita terminou
    if       (i > med)      v[k] = aux[j++];
    // se subvetor da esquerda terminou
    else if (j > sup)      v[k] = aux[i++];
    else // senão, compara e copia o menor valor
    if (aux[j] < aux[i])  v[k] = aux[j++];
    else                   v[k] = aux[i++];
  }
}
```

Mergesort

```
mergesort(int[] v) {
     int aux[] = new int[v.length];
     sort(v, aux, 0, v.length-1);
}

sort(int[] v, int[] aux, int inf, int sup) {
   if (sup <= inf) return;
   int med = inf + (sup - inf)/2;

   sort(v, aux, inf, med);
   sort(v, aux, med+1, sup);
   merge(v, aux, inf, med, sup);
}
```

Mergesort recursivo

Em cinza: posições desconsideradas do `merge` atual.

Em vermelho: subvetor com posições a partir de `inf` até `med`.

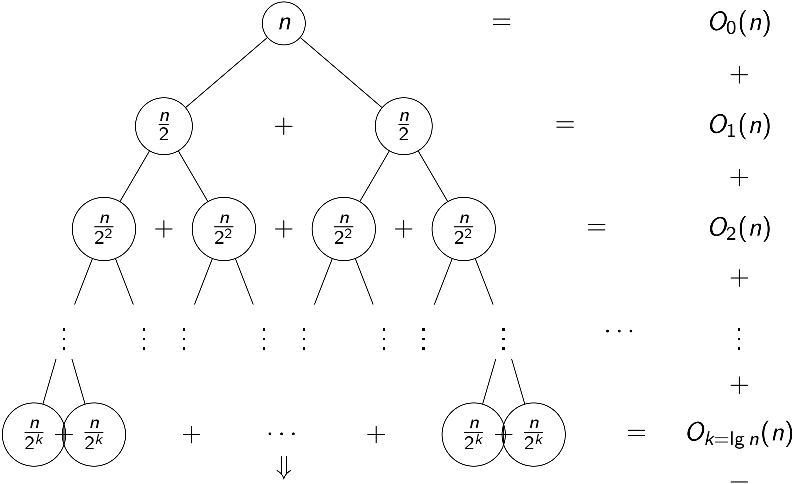
Em azul: subvetor com posições depois de `med` até `sup`.

Em verde: subvetor resultante do `merge`.

6	8	2	9	0	7	4	1	3	5
6	8	2	9	0	7	4	1	3	5
6	8	2	9	0	7	4	1	3	5
2	6	8	9	0	7	4	1	3	5
2	6	8	9	0	7	4	1	3	5
2	6	8	0	9	7	4	1	3	5
2	6	8	0	9	7	4	1	3	5
0	2	6	8	9	7	4	1	3	5
0	2	6	8	9	7	4	1	3	5
0	2	6	8	9	4	7	1	3	5

0	2	6	8	9	4	7	1	3	5
0	2	6	8	9	1	4	7	3	5
0	2	6	8	9	1	4	7	3	5
0	2	6	8	9	1	4	7	3	5
0	2	6	8	9	1	4	7	3	5
0	2	6	8	9	1	3	4	5	7
0	2	6	8	9	1	3	4	5	7
0	1	2	3	4	5	6	7	8	9

Análise de complexidade



$$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) \Leftrightarrow O(n \cdot \lg n)$$

Mergesort bottom-up

Melhorias possíveis

- não fazer merge de dois subvetores já ordenados
  - quando o maior elemento do subvetor nas menores posições for menor que o menor elemento do subvetor nas maiores posições
  - exemplo: | 1 | 3 | 6 | 8 | 10 | 11 |
- eliminar recursão: algoritmo bottom-up
  - começar desde o ínicio com subvetores menores
  - aplicar merge para aumentar subvetores

Mergesort bottom-up

Ideia

- Varrer vetor fazendo merge de subvetores de tamanho 1
- Repetir operação para subvetores de tamanho 2, 4, 8, 16, ...

Mergesort bottom-up

```java
mergesort(int[] v) {
 int n = v.length;
 int[] aux = new int[n];
 // tamanho dobra a cada iteração
 for (int tam = 1; tam < n; tam = tam +tam)
  for (int inf = 0; inf < n -tam; inf += tam+tam)
    // subvetor à esquerda em [inf, inf+tam-1]
    // subvetor à direita em [inf+tam, inf+tam+tam-1]
    // ou, se necessário, em [inf+tam, n-1]
    merge(v, inf, inf+tam-1,
          Math.min(inf+tam+tam-1, n-1));
}
```

Limites

$v[inf]$	$v[\ldots]$	$v[inf + tam - 1]$	$v[inf + tam]$	$v[\ldots]$	$v[inf + tam + tam - 1]$

ou

$v[inf]$	$v[\ldots]$	$v[inf + tam - 1]$	$v[inf + tam]$	$v[\ldots]$	$v[n - 1]$

Mergesort bottom-up

5	4	2	9	6	0	3	8	7	1
4	5	2	9	6	0	3	8	7	1
4	5	2	9	6	0	3	8	7	1
4	5	2	9	6	0	3	8	7	1
4	5	2	9	6	0	3	8	7	1
4	5	2	9	0	6	3	8	7	1
4	5	2	9	0	6	3	8	7	1
4	5	2	9	0	6	3	8	7	1
4	5	2	9	0	6	3	8	7	1
4	5	2	9	0	6	3	8	1	7

4	5	2	9	0	6	3	8	1	7
2	4	5	9	0	6	3	8	1	7
2	4	5	9	0	6	3	8	1	7
2	4	5	9	0	3	6	8	1	7
2	4	5	9	0	3	6	8	1	7
0	2	3	4	5	6	8	9	1	7
0	2	3	4	5	6	8	9	1	7
0	1	2	3	4	5	6	7	8	9

Merge sort é assintoticamente ótimo

Melhor caso

Podemos representar qualquer algoritmo de ordenação baseado em comparações em uma árvore de comparações contém $n!$ elementos. Menor altura, que é proporcional ao número de comparações, dessa árvore é $\log(n!)$.

Usando a fórmula de Stirling

$\log(n!) \approx n \log(n)$

Algoritmo ótimo

Como o pior caso do mergesort é $O(n \log n)$ é igual ao melhor caso do problema, o mergesort é um algoritmo ótimo em relação ao número de comparações.

Counting sort - ordenação sem comparações

Ideia: ordenação de números inteiros

Cada elemento é representado por uma posição em um vetor.
Conta-se as repetições de cada número.

```
void countingSort(int[] vetor, int max) {

    long[] count = new long[max+1];
    for (int i = 0; i < vetor.length; i++)
        count[vetor[i]]++; //conta repetições

    // retirar os numeros das contagens
    for (int j = 0, i = 0; i < vetor.length; j++)
        while (count[j]-- > 0 )
            vetor[i++] = j;
}
```

Outros algoritmos interessantes

- Shell sort
- Radix sort
- Bucket sort
- Heap sort
- Merge sort paralelo

Conclusões

- A escolha de um algoritmo de ordenação depende de:
  - quantidade de dados
  - tipo de dados (inteiros, strings, pontos flutuantes, tipos compostos)
  - modelo computacional (acesso aleatório, ponteiros, paralelismo ...)
  - hierarquia de memória (RAM, disco, cache,)
  - necessidade de estabilidade