ALGORITMOS

Disciplina: Estruturas de Dados e Algoritmos

Professor: Rafael Marinho e Silva

CRÉDITOS

O material a seguir contém trechos e adaptações dos originais criados pelos Professores:

- Professor Marcelo Zorzan
- Professora Rachel Reis
- Professor Marcelo Keese Albertini
- Professor André Back

ALGORITMOS

REVISÃO - VARIÁVEIS

Variáveis são **endereços/espaços** de **memória reservados** para guardarem **valores** durante a execução de um programa;

Todas as variáveis em **JAVA** devem ser **declaradas** com um **tipo** seguida do **nome** da variável

Exemplo: **tipo_da_variavel** **nome_das_variaveis** ;

```
int a;
int x, cont, aux;
```

Tipo da Variável

Nome(s) da(s) Variável(eis)

ALGORITMOS

REVISÃO - TIPOS DE VARIÁVEIS

É a quantidade de **memória**, em **bytes**, que a variável ocupará e a maneira como um determinado valor deve ser armazenado**.**

- São cinco tipos básicos de dados em C:

TIPO	BIT	BYTES	FAIXA DE VALORES
char	8	1	de 0 a 65.535
int	16	2	de (-2^31) a (2^31 - 1)
float	32	4	de (-2^-126) a (2^128 - 2^104)
double	64	8	de (-2^-1022) a (2^1024 - 2^970)
booleano	1	1/8	false ou true

REVISÃO - FUNÇÃO printf()

Formato da função printf():

- printf(string_de_controle, lista_de_argumentos);

CÓDIGO	SIGNIFICADO
%d	Inteiro
%f	Float
%c	Caractere
%s	String
%%	Imprime % na tela

```
#include <stdio.h>
/* Exemplo da função printf() */
int main(){
    int  num = 100;
    printf("%d", num);
    return 0;
}
```

ALGORITMOS

REVISÃO - FUNÇÃO scanf()

Formato da função scanf():

- scanf(string_de_controle, lista_de_argumentos);

CÓDIGO	SIGNIFICADO
%d	Inteiro
%f	Float
%c	Caractere
%s	String
%%	Imprime % na tela

```
#include <stdio.h>
/* Exemplo da função scanf() */
int main(){
    int  num;
    printf("Digite um número: ");
    scanf("%d", &num);
    printf("%d", num);
    return 0;
}
```

REVISÃO

OPERADOR DE ATRIBUIÇÃO

Forma geral do operador de atribuição:

- nome_da_variavel = valor

```
#include <stdio.h>
/* Exemplo da função scanf() */
int main(){
    int  num, aux, x;
    num = aux = x = 10;
    return 0;
}
```

OPERADORES ARITMÉTICOS

Lista dos operadores aritméticos da linguagem C:

OPERADOR	AÇÃO
++ --	incremento; decremento
* /	multiplicação; divisão
%	módulo da divisão (resto)
+ -	adição; subtração
== !=	igual a; diferente de

ALGORITMOS

REVISÃO - OPERADORES DE INCREMENTO E DECREMENTO

INCREMENTO: ++

```
x = x + 1;
x++;
```

DECREMENTO: --

```
x = x - 1;
x--;
```

Ambos operadores podem ser utilizados como **prefixo ou sufixo**

```
x = 5;
y = ++x;
```

Executa o incremento **antes** da atribuição do y;

```
x = 5;
y = --x;
```

Executa o decremento **antes** da atribuição do y;

```
x = 5;
y = x++;
```

Executa o incremento **depois** da atribuição do y;

```
x = 5;
y = x--;
```

Executa o decremento **depois** da atribuição do y;

ALGORITMOS

EXERCÍCIOS - OPERADORES DE INCREMENTO E DECREMENTO

Qual é o resultado das variáveis x, y e z, em cada linha das atribuições

a)
```
int x, y, z;
x = y = 5;
z = ++x;
x = y--;
y = x + z - (z--);
```

x	y	z
–	–	–
–	–	–
–	–	–
–	–	–

b)
```
int x, y, z;
x = z = 8;
y = x--;
x = --z;
z = y - x + (++z);
```

x	y	z
–	–	–
–	–	–
–	–	–
–	–	–

EXERCÍCIOS - OPERADORES DE INCREMENTO E DECREMENTO

Qual é o resultado das variáveis x, y e z, em cada linha das atribuições

a)
```
int x, y, z;
x = y = 5;
z = ++x;
x = y--;
y = x + z - (z--);
```

x	y	z
5	5	0
6	5	6
5	4	6
5	5	5

b)
```
int x, y, z;
x = z = 8;
y = x--;
x = --z;
z = y - x + (++z);
```

x	y	z
8	0	8
7	8	8
7	8	7
7	8	9

REVISÃO - ESTRUTURA DE REPETIÇÃO

Comando do while

- Garante que o bloco de instruções seja executado no mínimo uma vez, já que a condição que controla o laço é testada apenas no final do comando.

```
do{
   /* Instrução ou bloco de instruções*/
}while (condição);
```

Comando while

- Executa a repetição de um bloco de instruções enquanto uma condição é verdadeira.

```
while (condição){
   /* Instrução ou bloco de instruções*/
}
```

Comando for

- Executam a repetição de um conjunto de instruções enquanto uma determinada condição é verdadeira.

```
for(valor_inial; condição_final; valor incremento){
   /* Instrução ou bloco de instruções*/
}
```

ALGORITMOS

INTRODUÇÃO

Um **algoritmo** é qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como **entrada** e produz algum valor ou conjunto de valores como **saída**.

Exemplo: problema de ordenação crescente

- **Entrada**: uma sequência de n números $\mathbf{a_1, a_2, a_3, ..., a_n}$;
- **Saída**: uma reordenação $\mathbf{b_1, b_2, ..., b_n}$, tal que $b_i < b_j$ se $i < j$

Um **algoritmo** é uma sequência de **passos computacionais** que transformam a **entrada** na **saída**.

O **algoritmo** descreve um **procedimento computacional específico** para se alcançar esse relacionamento da **entrada** com a **saída**.

ANÁLISE DE ALGORITMOS

Exemplo: problema de ordenação crescente

- **Entrada**: uma sequência de n números $\mathbf{a_1, a_2, a_3, ..., a_n}$;
- **Saída**: uma reordenação $\mathbf{b_1, b_2, ..., b_n}$, tal que $b_i < b_j$ se $i < j$

A solução: um algoritmo

- **correto:** se para cada instância de entrada, o algoritmo **para** com a saída correta
- se o algoritmo é correto então ele resolve o problema

Instância e tamanho da entrada de um problema

- Uma instância é uma materialização do problema: 3, 1, 2, 4, 1, 5
- Tamanho da entrada N varia de acordo com a instância

ANÁLISE DE ALGORITMOS

CUSTOS

- Quanto espaço de memória o algoritmo vai consumir?
- Quanto tempo o algoritmo leva para executar?
- Quantos acessos a disco o algoritmo fará?
- Qual é o consumo de energia?
- Quantas requisições serão feitas a um banco de dados?
- Etc.

Como responder essas perguntas de acordo com a variável do tamanho do problema *N?*

ANÁLISE DE ALGORITMOS

CONCEITOS

- Vector: **int vetor[] = {6, 2, 8, 4, 1, 9};**
- Variável índice: posição para acesso de elemento
- Variável auxiliar: armazenamento temporário
  - troca de posição de elementos do vetor

Entrada:

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

Saída:

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

ANÁLISE DE ALGORITMOS

Algoritmo de ordenação simples:

Comparar os pares consecutivos de elementos e trocá-los de posição caso o primeiro seja maior que o segundo

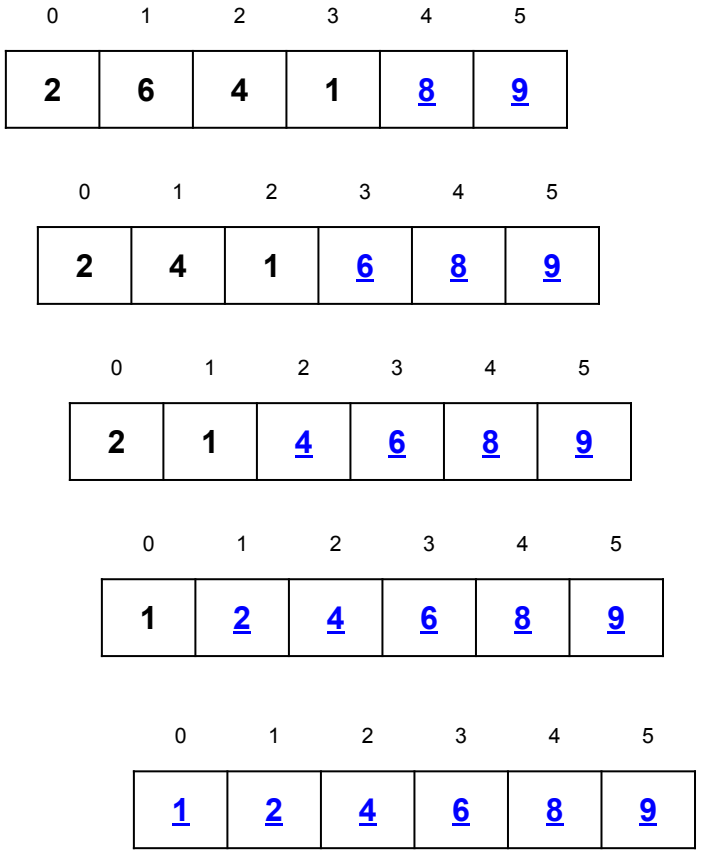
```
1  void troca( int vetor[ ], int i, int j ) {
2      int aux = vetor[ i ];
3      vetor[ i ] = vetor [ j ];
4      vetor[ j ] = aux;
5  }
```

ALGORITMOS

ITERAÇÕES DO ALGORITMO DE ORDENAÇÃO

ATIVIDADES

TERMINAR AS ITERAÇÕES DO ALGORITMO



ALGORITMOS

COMPLEXIDADE DE ALGORITMO

É a quantificação do esforço (quantidade de trabalho) de um algoritmo para resolver um determinado problema

PRINCIPAIS MEDIDAS DE COMPLEXIDADE

- Tempo e espaço relacionados à velocidade e a quantidade de memória

COMPLEXIDADE DE ALGORITMO

Complexidade de tempo

O tempo de execução de um algoritmo é determinado pelas instruções executadas

Complexidade de espaço

Espaço de memória utilizado pelo algoritmo

ANÁLISE MATEMÁTICA

Contando instruções de um algoritmo

Este algoritmo procura o menor valor presente em um array "**vet**" contendo "n" elementos e o armazena na variável "**menor**".

```
1 int menor = vet[0];
2 for (i = 0; i < n; i++){
3     if(vet[ i ] <= menor){
4         menor = vet[ i ];
5     }
6 }
```

Quantidade de "instruções simples" do algoritmo

- atribuição de um valor a uma variável
- acesso ao valor de um determinado elemento do array
- comparação de dois valores
- incremento de um valor
- operações aritméticas básicas (adição, multiplicação, etc.)

ANÁLISE MATEMÁTICA

Contando instruções de um algoritmo

Este algoritmo procura o menor valor presente em um array "**vet**" contendo "n" elementos e o armazena na variável "**menor**".

```
1 int menor = vet[0];
2 for (i = 0; i < n; i++){
3     if(vet[ i ] < menor){
4         menor = vet[ i ];
5     }
6 }
```

linha 1 == "1 instrução" - atribuição
linha 2 == "2 instruções" - atribuição e comparação
linha 2 == "2n instruções" - incremento e comparação
n vezes { linha 3 == "1 instrução" - comparação
n vezes { linha 4 == "1 instrução" - atribuição (dependente do if)

Considerando o custo dominante (pior caso) do algoritmo, sua função matemática em relação ao tamanho do array de entrada é: **"f(n) = 3 + 2n + 2n"**

ANÁLISE MATEMÁTICA

Comportamento assintótico

- Nem todos os termos são necessários
- Deve ser considerado apenas os termos que apresentam o que acontece com a função quando o tamanho dos dados de entrada (“n”) cresce

```
1 int menor = vet[0];
2 for (i = 0; i < n; i++){
3     if(vet[ i ] <= menor){
4         menor = vet[ i ];
5     }
6 }
```

f(n) = 3 + 2n + 2n

f(n) = 3 + 4n

O termo “3” é uma “constante de inicialização”, ele não se altera à medida que “n” aumenta. Portanto, a função pode ser reduzida para “**f(n) = 4n**”. As constantes que multiplicam o termo “**n**” da função também devem ser descartadas. Logo:

f(n) = n

ANÁLISE MATEMÁTICA

Comportamento assintótico

- Nem todos os termos são necessários
- Deve ser considerado apenas os termos que apresentam o que acontece com a função quando o tamanho dos dados de entrada **(“n”)** cresce
- A função tem comportamento assintótico igual a **(1)** quando a função não possui nenhum termo multiplicado por **(“n”)**

Função de custo	Comportamento assintótico
$f(n) = 100$	$f(n) = 1$
$f(n) = 10n + 2$	$f(n) = n$
$f(n) = n^2 + 2n + 4$	$f(n) = n^2$
$f(n) = 5n^3 + 400n^2 + 220$	$f(n) = n^3$

ALGORITMOS

ANÁLISE DE ALGORITMOS

MELHOR CASO

- Menor tempo de execução sobre todas as entradas de tamanho N

PIOR CASO

- Maior tempo de execução sobre todas as entradas de tamanho N

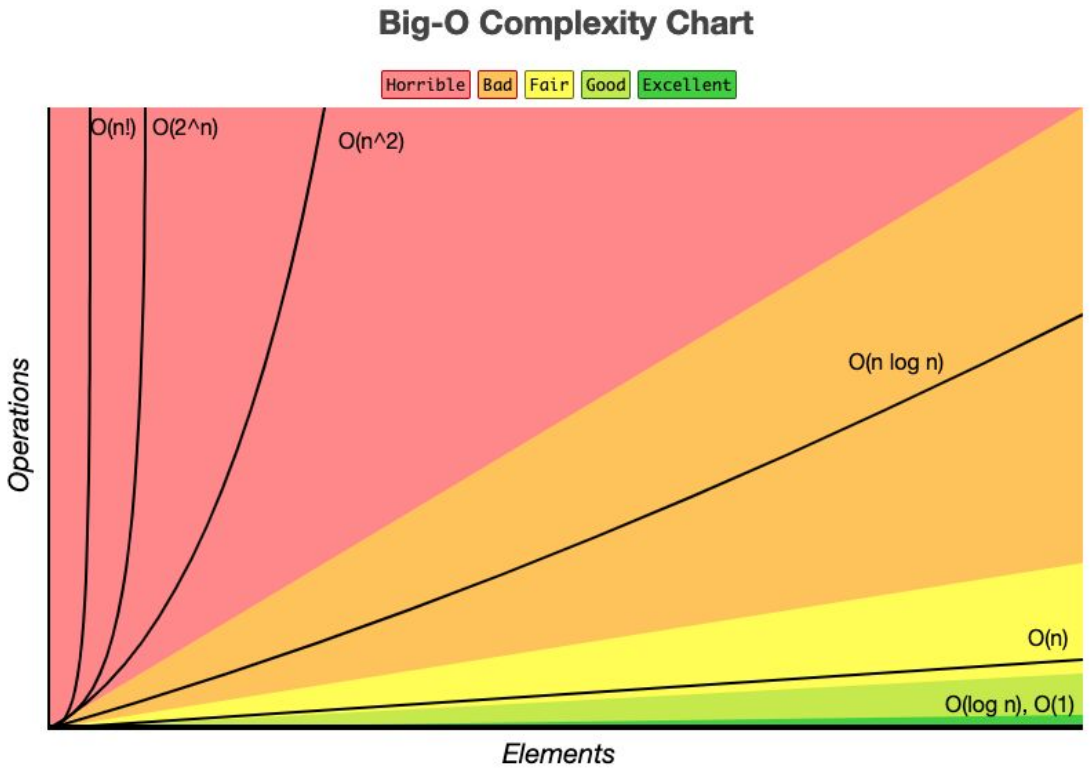
CASO MÉDIO (OU CASO ESPERADO)

- Média dos tempos de execução de todas as entradas de tamanho N
  - Análise mais complexa de obter do que as análises do melhor e pior caso

ANÁLISE DE ALGORITMOS

Notação Grande O

- Representa o custo (tempo ou espaço) do algoritmo no pior caso possível para todas as entradas de tamanho **“n”**
- Ele é o **limite superior** de entrada, ou seja, o algoritmo não pode ultrapassar um certo limite



ANÁLISE DE COMPLEXIDADE

Exemplo 1: Quantas instruções são executadas em função de N?

```
1  int  cont = 0;
2      for( int i = 0; i < N; i++)
3          if ( a [ i ] == 0 )
4              cont++
```

Operações	frequência
Declaração de variável	2
Atribuição de valor	2
Comparação menor que	N + 1
Comparação igual	N
Acesso a array	N
incremento	>= N e <= 2N

Custo do algoritmo no pior caso é O(n)

ANÁLISE DE COMPLEXIDADE

Exemplo 2: Quantas instruções são executadas em função de N?

Obs.: K é uma constante)

```
1  int  cont = 0;
2      for( int i = 0; i < N; i++)
3          for( int j = 0; j < N; j++)
4              if( a[ i ] + a [ j ] == K)
5                  cont++
```

Operações	frequência
Declaração de variável	N + 2
Atribuição de valor	N + 2
Comparação menor que	1 / 2 (N + 1)(N + 2)
Comparação igual	1 / 2 N (N - 1)
Acesso a array	N (N - 1)
incremento	<= N^2

Custo do algoritmo no pior caso é $O(n^2)$

ANÁLISE DE COMPLEXIDADE

Exemplo 2: Quantas instruções são executadas em função de N?

Obs.: K é uma constante)

```
1  int  cont = 0;
2      for( int i = 0; i < N; i++)
3          for( int j = i + 1; j < N; j++)
4              if( a[ i ] + a [ j ] == K)
5                  cont++
```

Operações	frequência
Declaração de variável	N + 2
Atribuição de valor	N + 2
Comparação menor que	1 / 2 (N + 1)(N + 2)
Comparação igual	1 / 2 N (N - 1)
Acesso a array	N (N - 1)
incremento	<= N^2

Custo do algoritmo no pior caso é $O(n^2)$

ANÁLISE DE COMPLEXIDADE

Exemplo 2: Quantas instruções são executadas em função de N?

```
1  void bubblesort (int vetor[ ]) {
2      int N = vetor.length:
3      /*controle do número de interações*/
4      for (int i = 0; i < N-1; i++)
5      /*repetição interna: compara e troca números */
6          for (int j = i+1; j < N; j++)
7              if (vetor[ i ] > vetor[ j ]
8                  troca( vetor, j, j + 1); /*função de troca*?
9  }
```

Operações	frequência
Custo de comparação	$(N - 1)^2$
Elementos a ordenar: número de transações PIX	$N = 4 \times 10^7$
Comparações necessárias	$(4 \times 10^7)^2$
Capacidade do computador:	2^{32} operações por segundo
Tempo necessário em segundos:	$((4 \times 10^7)^2) / 2^{32}$
Tempo de processamento (≅)	

ANÁLISE DE COMPLEXIDADE - BUBBLESORT

Exemplo 2: Quantas instruções são executadas em função de N?

```
1  void bubblesort (int vetor[ ]) {
2      int N = vetor.length:
3      /*controle do número de interações*/
4      for (int i = 0; i < N-1; i++)
5      /*repetição interna: compara e troca números */
6          for (int j = 0; j < N-1; j++)
7              if (vetor[ i ] > vetor[ j ]
8                  troca( vetor, j, j + 1); /*função de troca*?
9  }
```

Operações	frequência
Custo de comparação	$(N - 1)^2$
Elementos a ordenar: número de transações PIX	$N = 4 \times 10^7$
Comparações necessárias	$(4 \times 10^7)^2$
Capacidade do computador:	2^{32} operações por segundo
Tempo necessário em segundos:	$((4 \times 10^7)^2) / 2^{32}$
Tempo de processamento (≅)	**4 dias, 7 horas e 28 minutos**

COMPLEXIDADE DE ORDENAÇÃO

PROBLEMA

- Um banco tem 10 milhões clientes que fazem, em média, 4 transações diárias usando o PIX. O Banco Central pede um relatório diário com as transações ordenadas por valor. Suponha que o computador do banco resolva operações simples na taxa de 2^{32} (≈ 4,3 bilhões) operações por segundo.

SE O ALGORITMO DISPONÍVEL FOR O BUBBLESORT, O BANCO:

a. Conseguirá tranquilamente fazer o relatório a tempo;
b. Deverá comprar um computador melhor;
c. Deverá contratar um programador melhor;

ALGORITMOS

ANÁLISE DE COMPLEXIDADE - BUBLESORT SORT

```c
//Exercício-1 - Bubble Sort

#include <stdio.h>

int main(void)
{
    int vetor[10] = {2,4,6,9,11,17,21,28,30,45}; //melhor caso
    int vetor[10] = {45,30,28,21,17,11,9,6,4,2}; //pior caso
    int vetor[10] = {21,11,17,28,30,9,6,45,2,4}; //Valores aleatórios

    int num = 10;       //quantidade de números no "vetor[]"

    int i,j;            //declaração das variáveis "i" e "j" usadas nos laços "for()"
    int aux;
    int cont=1;
    for(i = 0; i < num; i++){
        for(j = 0; j < num-1; j++){

            if(vetor[j] > vetor[j+1]){
                aux = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = aux;
            }
        }

    }

    for(i = 0; i < num; i++){
        printf("[%i]",vetor[i]);
    }
    return 0;
}
```

Link

ANÁLISE DE COMPLEXIDADE - BUBLESORT SORT

```c
//Exercício-1 - Bubble Sort

#include <stdio.h>

int main(void)
{
    int vetor[10] = {2,4,6,9,11,17,21,28,30,45}; //melhor caso
    int vetor[10] = {45,30,28,21,17,11,9,6,4,2}; //pior caso
    int vetor[10] = {21,11,17,28,30,9,6,45,2,4}; //Valores aleatórios

    int num = 10;       //quantidade de números no "vetor[]"

    int i,j;            //declaração das variáveis "i" e "j" usadas nos laços "for()"
    int aux;
    int cont=1;
    for(i = 0; i < num; i++){
        for(j = 0; j < num-1; j++){

            if(vetor[j] > vetor[j+1]){
                aux = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = aux;
            }
        }

    }

    for(i = 0; i < num; i++){
        printf("[%i]",vetor[i]);
    }
    return 0;
}
```

Link

De acordo com a execução do algoritmo Bubble Sort, ilustrado na Figura 1, responda as questões 1, 2 e 3.

1 – Quantas vezes as linhas 20, 21 e 22 são executadas:

2 – Logo, qual é o Big-O do "Melhor e Pior Caso" do algoritmo "Bubble Sort"?

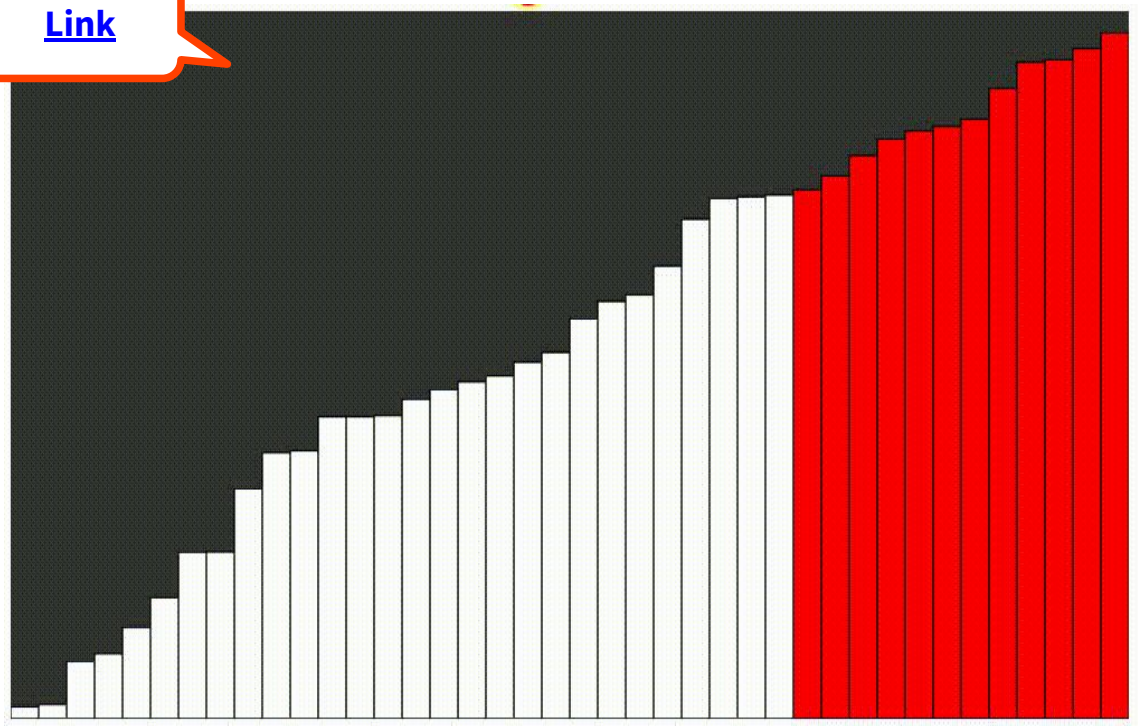
3 – Qual é a sequência dos valores do vetor quando a linha 24 (final do segundo for()) for executada, com o "i==1" e o "j==8"?

ALGORITMOS

ANÁLISE DE COMPLEXIDADE - BUBLESORT SORT

```
1  void bubblesort (int vetor[ ]) {
2      int i, continua, aux, fim = vetor.length;
3      do{
4          continua = 0;
5          for (i = 0; i < fim-1; i++){
6              if( vetor[ i ] > vetor[i + 1]){
7                  aux = vetor[ i ];
8                  vetor[ i ] = vetor[i + 1];
9                  vetor[i + 1] = aux;
10                 continua = i;
11             }
12         }
13         fim--;
14     }while(continua != 0);
15 }
```

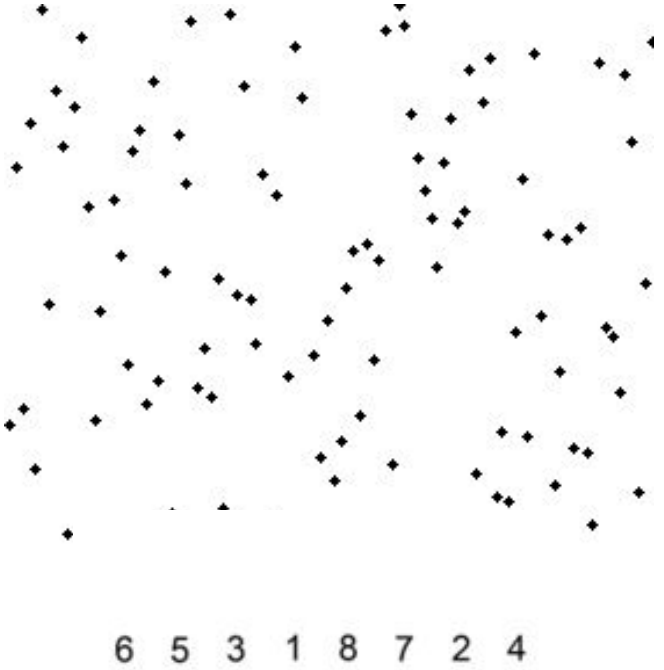
Link



ANÁLISE DE COMPLEXIDADE - INSERTION SORT

Link-1
Link-2

```
1 void insercao (int vet) {
2     int aux, i, j;
3     for(i = 1; i < vet.length; i++){
4         aux = vet[ i ];
5         for(j = i; (j > 0) && (aux < vet[j - 1]); j--){
6             vet[ j ] = vet[j - 1];
7         }
8         vet[ j ] = aux;
9     }
10 }
```

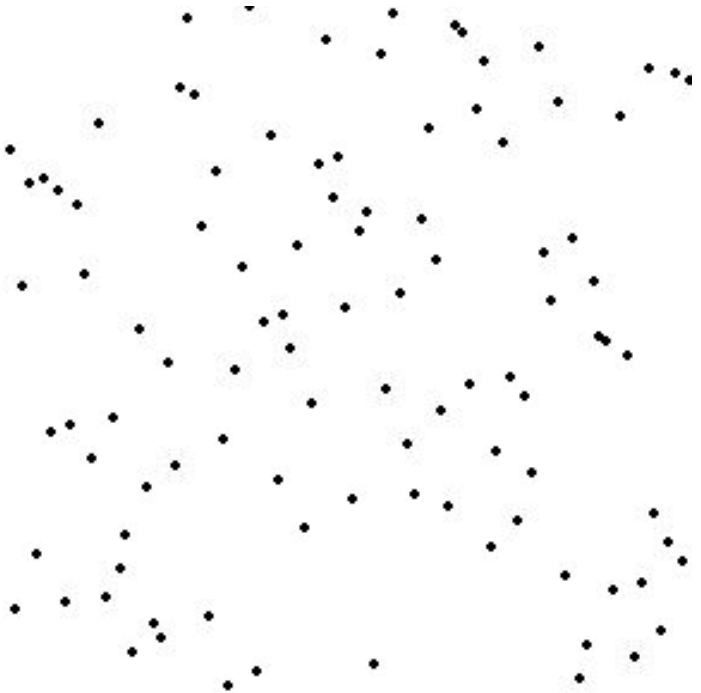


ANÁLISE DE COMPLEXIDADE - SELECTION SORT

```
1 void selecao (int vet, int n) {
2     int i, j, min, x;
3     for ( i = 0; i <= n-1; i++){
4         min = i;
5         for ( j = i+1; j < n; j++){
6             if(vet[ j ] < vet[ min ]);
7                 min = j;
8         }
9         x = vet[ min ];
10        vet[ min ] = vet[ i ];
11        vet[ i ] = x;
12    }
13 }
```

Link-1
Link-2

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

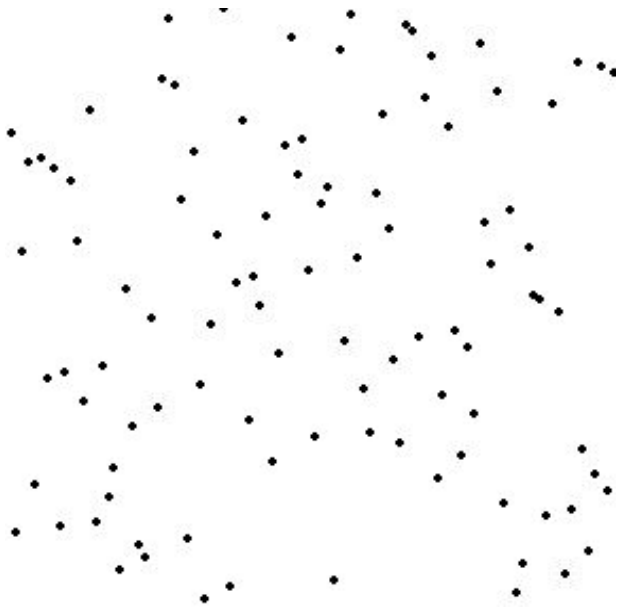


ANÁLISE DE COMPLEXIDADE - SELECTION SORT

```
1 void selectionSort (int vet, int n) {
2     int i, j, menor, troca;
3     for ( i = 0; i < n-1; i++){
4         menor = i;
5         for ( j = i+1; j < n; j++){
6             if(vet[ j ] < vet[ menor ]);
7                 menor = j;
8         }
9         if(i != menor){
10            troca = vet[ i ];
11            vet[ i ] = vet[ menor ];
12            vet[ menor ] = troca;
13        }
14    }
15 }
```

Link-1
Link-2

8
5
2
6
9
3
1
4
0
7



ALGORITMOS

ANÁLISE DE COMPLEXIDADE - PERFORMANCE

BubleSort	• Melhor caso: O(N) • Pior caso: O(N^2) • Não recomendado para grandes conjuntos de dados
InsertionSort	• Melhor caso: O(N) • Pior caso: O(N^2) • Eficiente para conjuntos pequenos de dados • Estável: não altera a ordem de dados iguais • Capaz de ordenar os dados a medida em que os recebe (tempo real)
SelectionSort	• Melhor caso: O(N^2) • Pior caso: O(N^2) • Ineficiente para grandes conjuntos de dados • Estável: não altera a ordem de dados iguais

MERGESORT

MERGESORT OU ORDENAÇÃO POR MISTURA

- Dividir para conquistar;
- Divide, recursivamente, o conjunto de dados até que cada subconjunto possua 1 elemento;
- Combina 2 subconjuntos de forma a obter 1 conjunto maior e ordenado;
- Esse processo se repete até que exista apenas 1 conjunto;

- Melhor Caso: O(N log N);
- Pior Caso: O(N log N);
- Estável: não altera a ordem de dados iguais;
- Desvantagens: Recursivo e usa um vetor auxiliar durante a ordenação.

MERGESORT

MERGESORT OU ORDENAÇÃO POR MISTURA

```
void mergeSort (int *V, int inicio, int fim) {
    int meio;
    if(inicio < fim){
        meio = floor((inicio+fim)/2); //arredonda para baixo
        mergeSort(V, inicio, meio);
        mergeSort(V, meio+1, fim);
        merge(V, inicio, meio, fim);
    }
}
```

Chamada recursiva para as duas metades

Combina as duas metades de forma ordenada

MERGESORT

```
void merge (int *V, int inicio, int meio, int fim) {
   int *temp, p1, p2, tamanho, i, j, k;
   int fim1 = 0; fim2 = 0;
   tamanho = fim - inicio + 1;
   p1 = inicio;
   p2 = meio + 1;
   temp = (int *) malloc(tamanho * sizeof(int));
   if(temp != NULL){
      for(i = 0; i < tamanho; i++){
         if(!fim1 && !fim2){
            if(V[p1] < V[p2])
               temp[i] = V[p1++];
            else
               temp[i] = V[p2++];

            if(p1 > meio) fim1 = 1;
            if(p2 > fim) fim2 = 1;
         }else{
            if( !fim1)
               temp[i] = V[p1++];
            else
               temp[i] = V[p2++];
         }
      }
      for(j = 0, k = inicio; j < tamanho; j++, k++)
        V[k] = temp[j];
   }
   free(temp);
}
```

Combinar ordenado

Vetor acabou?

Copia o que sobrar

Copiar do auxiliar para o original

QUICKSORT

QUICKSORT OU ORDENAÇÃO POR TROCA DE PARTIÇÕES

- Dividir para conquistar;
- Elemento é escolhido como pivô;
- “Particionar”: os dados são rearranjados
  - Os valores menores do que o pivô são colocados antes dele e os maiores, depois;
- Ordena recursivamente as duas partições

- Melhor Caso: O(N log N);
- Pior Caso (raro): $O(N^2)$;
- Estável: não altera a ordem de dados iguais;
- Desvantagens: escolha do pivô

QUICKSORT

QUICKSORT OU ORDENAÇÃO POR TROCA DE PARTIÇÕES

```
void quickSort (int *V, int inicio, int fim) {
    int pivo;
    if(fim > inicio){
        pivo = particiona(V, inicio, fim);
        quickSort(V, inicio, pivo-1);
        quickSort(V, pivo+1, fim);
    }
}
```

Separa os dados em duas partições

Chamada recursiva para as duas metades

```
int particiona (int *V, int inicio, int final) {
    int esq, dir, pivo, aux;
    esq = inicio;
    dir = final;
    pivo = V[inicio];
    while(esq < dir){
        while(esq <= final && V[esq] <= pivo)
            esq++;
        while(dir >= 0 && V[dir] > pivo)
            dir--;
        if(esq < dir){
            aux = V[esq];
            V[esq] = V[dir];
            V[dir] = aux;
        }
    }
    V[inicio] = V[dir];
    V[dir] = pivo;
    return dir;
}
```

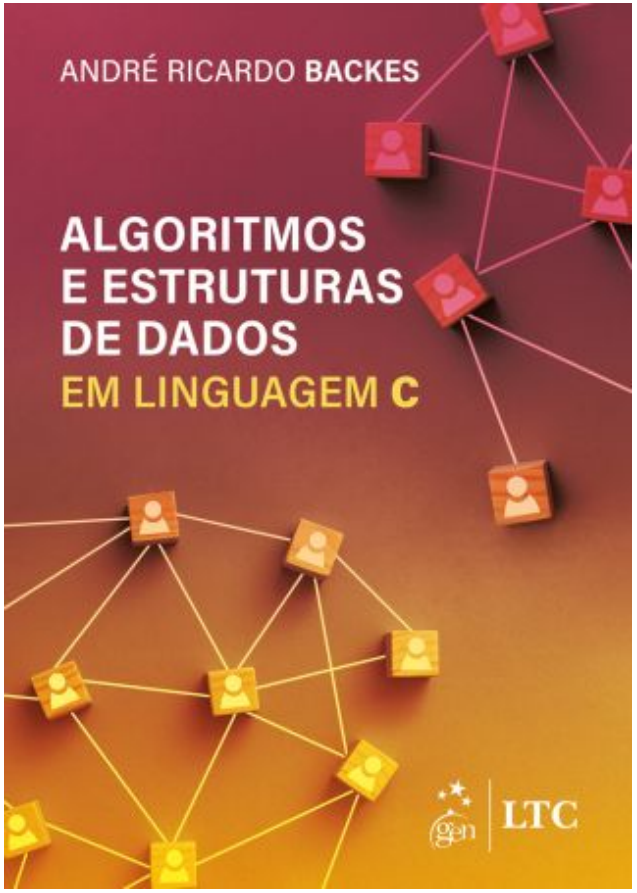
Avança posição da esquerda

Avança posição da direita

Troca esq e dir

ATIVIDADES

LEITURAS RECOMENDADAS



Agradecimentos

OBRIGADO.

Rafael Marinho e Silva
rafaelmarinho@unipam.edu.br