

Análise de Complexidade

Algoritmos e Estruturas de Dados 2

2017-1

Flavio Figueiredo (<http://flaviovdf.github.io>)

Lembrando da aula passada

- Criamos um Banco Simples
- TAD para Conta Corrente

conta_bancaria.h e conta_bancaria.c (respectivamente)

```
#ifndef CONTA_BANCARIA_H
#define CONTA_BANCARIA_H

// definição do tipo
typedef struct {
    int numero;
    double saldo;
} ContaBancaria;

// cabeçalho das funções
ContaBancaria* NovaConta(int, double);
void Deposito(ContaBancaria*, double);
void Saque(ContaBancaria*, double);
void Imprime(ContaBancaria*);

#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include "conta_bancaria.h"

ContaBancaria *NovaConta(int num, double saldo) {
    ContaBancaria *conta = malloc(sizeof(ContaBancaria));
    conta->numero = num;
    conta->saldo = saldo;
    return conta;
}

void Deposito(ContaBancaria *conta, double valor) {
    conta->saldo += valor;
}

void Saque(ContaBancaria *conta, double valor) {
    conta->saldo -= valor;
}

void Imprime(ContaBancaria *conta) {
    printf("Numero: %d\n", conta->numero);
    printf("Saldo: %f\n", conta->saldo);
}
```

Vamos adicionar um extrato

- Um extrato é?

Extrato

- Um extrato é?
 - Uma série de **operações bancárias**
 - Precisamos então de uma **transação** antes do extrato

transacao

- Um extrato é?
 - Uma série de **operações bancárias**
 - Precisamos então de uma **transação** antes do extrato
- Novo transacao.h ao lado
- TAD simples
- time.h
 - Funções de data e hora em C
 - time_t guarda a quantidade de segundos desde 01/01/1970

```
#ifndef TRANSACAO_H
#define TRANSACAO_H

#include <time.h>

#define CONTA_VAZIA -1

typedef struct {
    int daConta; //conta que originou a transacao
    int paraConta; //conta receptora da transacao
    double valor;
    time_t data;
} Transacao;

/*
 * Função auxiliar para casos de transferências.
 * Inverte do daConta com o paraConta
 */
Transacao* InverteDePara(ContaBancaria*);
void Imprime(Transacao*);

#endif
```

[Pequena Pausa] Como contamos tempo em C?

- Tempo é uma abstração no computador
- Solução:
 - Fixar um dia
 - Primeiro de Janeiro de 1970
 - Contamos a quantidade de segundos, ou milissegundos, ou micro-segundos desde essa data
 - 2 de Janeiro de 1970
 - 86400 (1 Dia em Segundos)

Extrato

- Fizemos um include de “transacao.h”
 - Qual o motivo?
- A ideia é que cada método como Saque atualize o Extrato da conta
- Novo método Transferencia

```
#ifndef CONTA_BANCARIA_H
#define CONTA_BANCARIA_H

#include “transacao.h”

typedef struct {
    int numero;
    double saldo;
} ContaBancaria;

// Agora retornamos a transacao. Adicionamos a
// transferencia
ContaBancaria* NovaConta(int, double);
void Deposito(ContaBancaria*, double);
void Saque(ContaBancaria*, double);
void Transferencia(ContaBancaria*, ContaBancaria*, double);
void Imprime(ContaBancaria*);

#endif
```


Extrato

- Um extrato é?
 - Uma série de **operações bancárias**
 - Precisamos então de uma **transação** antes do extrato **(DONE!)**
- Precisamos agora da série. Opções?
 - Usar um array

Adicionando um Extrato utilizando um Array

```
#ifndef CONTA_BANCARIA_H
#define CONTA_BANCARIA_H

#include "transacao.h"

#define TAMANHO_INICIAL 30
typedef Transacao Extrato[TAMANHO_INICIAL];

typedef struct {
    int numero;
    double saldo;
    Extrato extrato;
} ContaBancaria;

ContaBancaria* NovaConta(int, double);
void Deposito(ContaBancaria*, double);
void Saque(ContaBancaria*, double);
void Transferencia(ContaBancaria*, ContaBancaria*, double);
void Imprime(ContaBancaria*);
#endif
```

- Vantagens e Desvantagens?
- Quais são as operações que vocês querem fazer no seu extrato?

Adicionando um Extrato com Array

```
#ifndef CONTA_BANCARIA_H
#define CONTA_BANCARIA_H

#include "transacao.h"

#define TAMANHO_INICIAL 30
typedef transacao Extrato[TAMANHO_INICIAL];

typedef struct {
    int numero;
    double saldo;
    Extrato extrato;
} ContaBancaria;

ContaBancaria* NovaConta(int, double);
void Deposito(ContaBancaria*, double);
void Saque(ContaBancaria*, double);
void Transferencia(ContaBancaria*, ContaBancaria*, \
    double);
void Imprime(ContaBancaria*);
#endif
```

- Vantagens e Desvantagens?

- Vantagens

- Simples
- Acesso direto pelo índice

- Desvantagens?

- Número fixo de operações
 - Arrays tem tamanho fixo
 - Se aloquei 30 posições
 - Como colocar a 31 transacao?

- Operações

- Como achar uma transacao por data?
- Como achar a transacao de maior valor?

Análise de complexidade

- AEDS2 tem como foco problemas como os do slide anterior
- Vamos sair um pouco do código para discutir este foco agora
- Análise de complexidade vai ajudar nas questões levantadas
- Lembre-se que seu código é:
 - Algoritmos
 - Dados
 - Bons TADs ajudam a gerar bons algoritmos

Análise de complexidade

- O trabalho do programador completo envolve diversos problemas
- Projeto de algoritmos
 - Análise do problema
 - Decisões de projeto
 - Algoritmo a ser utilizado de acordo com seu comportamento
 - Comportamento depende de
 - tempo de execução
 - espaço ocupado

Para um algoritmo particular

- Tempo de execução
 - Quantas passos o algoritmo executa
- Memória
 - [Por exemplo] Tamanho da pilha e alocações
- No nosso estudo de caso:
 - Achar uma transação bancária com maior valor
 - Quantas passos?
 - Quanto de memória alocada?
 - Achar uma transação bancária com menor valor
 - Quantos passos?
 - Quanto de memória alocada?
 - Achar as transações que ocorreram em uma certa data
 - Quantos passos?
 - Quanto de memória alocada?

Classes de algoritmos

- Nos preocupamos com o custo de algoritmos para problemas específicos
 - Pensem em funções. Custo de uma função no seu código
- Para um mesmo problema existem diversos algoritmos
 - Diversas formas de achar a transação bancária de maior valor
- **A análise de complexidade nos permite comparar tais algoritmos**
- Existem algoritmos que são tidos como ótimos:
 - [Exemplo] Podemos provar que eles executam no menor número de passos possível
 - Casos triviais

A representação de dados vai afetar o custo

- Uma boa escolha de um TAD pode melhorar o custo do seu algoritmo
- Uma má escolha vai afetar seu algoritmo
- Capítulo 3 do Programming Pearls!

Medida de custo utilizando um modelo matemático

- Usamos uma **abstração matemática** de do **número de passos** que um computador executa durante um algoritmo
- Vamos abstrair o computador
 - Nosso foco aqui **não** é se um tipo memória RAM é mais rápida do que outra
 - Como também **não** é se um disco rígido é eficiente
 - Como também **não** é no tempo de compilação
 - **Não** é mensurado em segundos, bits etc.
- Vamos focar em número de passos de um algoritmo inicialmente
- Existe também complexidade de memória

Nosso custo é uma **função** de **complexidade**

- Função de complexidade $f(n)$
- $f(n)$ é o número de passos para executar um algoritmo cuja entrada tem tamanho n
- [Geralmente] n é mensurado como o número de passos
 - Em um vetor de 10 posições $n = 10$
 - Não o número de bits
- Falamos ***complexidade para um problema de tamanho n***

Voltando para o exemplo do banco

- Vamos manter o extrato como um array
- Queremos achar a transacao que tem o maior valor

```
double TransacaoDeMaiorValor(int n, Transacao extrato[n]) {  
    double maiorValor = extrato[0].valor;  
    for (int i = 1; i < n; i++) {  
        if (extrato[i].valor > maiorValor) {  
            maiorValor = extrato[i].valor;  
        }  
    }  
    return maiorValor;  
}
```

Achando o maior elemento de um vetor

- O código acima é uma instância de um algoritmo para resolver o problema de achar o maior elemento de um vetor
- Podemos pensar nele independente do tipo que é passado
 - Achar o maior número de um array de inteiros é um algoritmo similar
- Existem outros algoritmos para o problema acima
- Vamos mostrar que o nosso é **ótimo!**
- Vamos computar $f(n)$ para nosso algoritmo

Complexidade do algoritmo

```
double TransacaoDeMaiorValor(int n, Transacao extrato[n]) {  
    double maiorValor = extrato[0].valor;           //1  
    for (int i = 1; i < n; i++) {                  //n-1  
        if (extrato[i].valor > maiorValor) {       //n-1  
            maiorValor = extrato[i].valor;         //n-1 - T  
        }  
    }  
    return maiorValor;                             //1  
}
```

T equivale a quantidade de vezes que entramos no if

Qual o motivo das linhas do **for** serem $n-1$?

Laços geram repetições

- Um for de 0 até n $[0, n)$
 - `for (int i = 0; i < n; i++)`
- Repete tudo dentro dele n vezes
- Nosso for no slide anterior iniciar de 1
 - 1 passo a menos do que iniciar de 0
- Logo:
 - $n - 1$ passos
- Qual o valor de $f(n)$

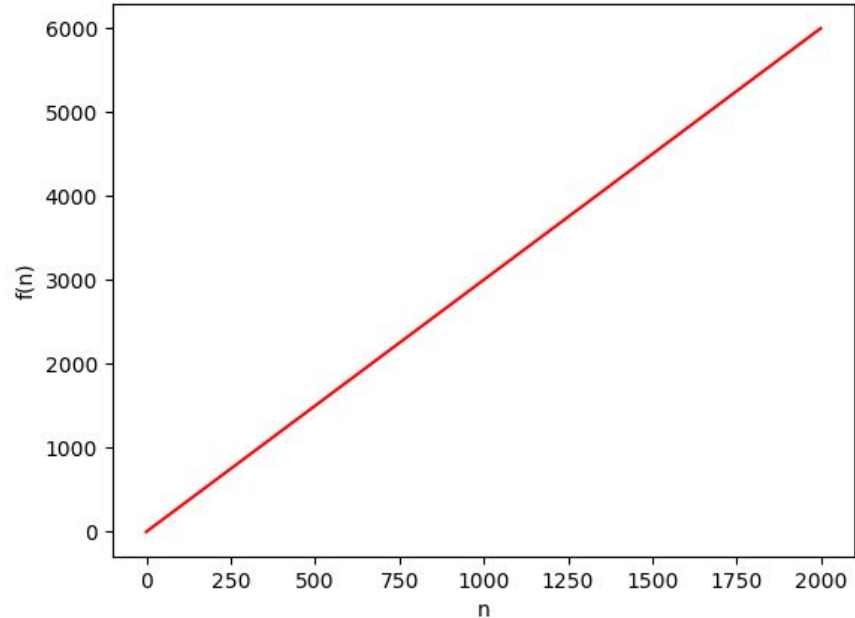
f(n) da função como um todo

- $f(n) = 1 + 1 + 3 * (n - 1)$
- $f(n) = 3n - 3 + 2$
- $f(n) = 3n - 1 - T$
- Qual a implicação disto?

```
double TransacaoDeMaiorValor(int n, Transacao extrato[n]) {
    double maiorValor = extrato[0].valor;           //1
    for (int i = 1; i < n; i++) {                  //n-1
        if (extrato[i].valor > maiorValor) {        //n-1
            maiorValor = extrato[i].valor;          //n-1 - T
        }
    }
    return maiorValor;                               //1
}
```


$f(n)$ da função como um todo

- $f(n) = 3n$
 - Vamos ignorar o -1 e o T
 - Mais na próxima aula
- Para cada elemento de entrada temos 3 operações
- Existe um crescimento linear na complexidade do meu algoritmo com a entrada



Algoritmo Ótimo:

Vamos focar apenas nas comparações

- O algoritmo abaixo faz $n-1$ comparações
- Linha do if
 - Cada if é uma comparação
- É impossível achar o menor elemento fazendo menos comparações
 - Em um vetor fora de ordem como o nosso

```
double TransacaoDeMaiorValor(int n, Transacao extrato[n]) {  
    double maiorValor = extrato[0].valor;  
    for (int i = 1; i < n; i++) {  
        if (extrato[i].valor > maiorValor) { // n-1  
            maiorValor = extrato[i].valor;  
        }  
    }  
    return maiorValor;  
}
```

Provando

- Vamos assumir que:
 - Existe um algoritmo que acha o maior elemento com $n - 1 - 1$ (ou seja $n - 2$) operações
 - Não existe ordem no vetor
- Obviamente:
 - $n - 2 < n - 1$
 - Assumimos que um algoritmo com $n - 2$ comparações existe
 - Vamos mostrar que nunca poderá garantir corretude
- Como o vetor não tem ordem
 - Ao olhar $n - 2$ elementos sempre vai sobrar 1 elemento para comparar
 - Sempre existe uma chance do maior elemento ser este último
 - Assumimos que o vetor não tem ordem
- Por contradição:
 - Um algoritmo com menos $n - 2$ operações não existe
 - Podemos generalizar com o mesmo argumento para $n - 1 - x$, onde x é um inteiro positivo

E se vetor estiver ordenado

789.0	700.0	430.2	150.0	50.0	50.0	22.23	8.50	8.50	2.0
-------	-------	-------	-------	------	------	-------	------	------	-----

- Neste caso achamos o maior elemento facilmente
 - `array[0]`
- Achamos o menor elemento também
 - `array[n - 1]`
- 1 operação cada
- Ou seja:
 - Se você armazenar seu vetor sempre ordenado, a operação **maior** e **menor** tem custo **$f(n) = 1$**
- Existem outras estruturas de dados que resolvem este problema com custo 1
 - MinHeap
 - MaxHeap

Tipos de casos

- Melhor caso
 - Vetor ordenado
 - Resolvo os problemas com 1 operação
- Caso médio
 - Vetor sem ordem
 - Cada elemento tem probabilidade $1/n$ de aparecer na posição n
- Pior caso
 - Vetor na ordem inversa
 - O maior elemento sempre é o último
 - Meu for só encontra ele na última comparação

Maior e Menor Elemento de um Vetor sem Ordem

```
//Vamos sair do mundo bancário.  
//Agora usamos um ponteiro ao invés de um array.  
//Não muda muito nossa vida (lembre-se da revisão)  
void MinMax(int *vec, int n, int *min, int *max) {  
    int i;  
    *min = vec[0];  
    *max = vec[0];  
    for(i = 1; i < n; i++) {  
        if(vec[i] < *min) {  
            *min = vec[i];  
        }  
        if(vec[i] > *max) {  
            *max = vec[i];  
        }  
    }  
}
```

Maior e Menor Elemento de um Vetor sem Ordem

//Vamos sair do mundo bancário.

//Agora usamos um ponteiro ao invés de um array.

//Não muda muito nossa vida (lembre-se da revisão)

```
void MinMax(int *vec, int n, int *min, int *max) {
```

```
    int i;
```

```
    *min = vec[0];
```

```
    *max = vec[0];
```

```
    for(i = 1; i < n; i++) {
```

```
        if(vec[i] < *min) {
```

```
            *min = vec[i];
```

```
        }
```

```
        if(vec[i] > *max) {
```

```
            *max = vec[i];
```

```
        }
```

```
    }
```

```
}
```

n - 1 comparações



$f(n) = 2(n - 1)$ comparações

Melhorando um Pouco

```
void MinMax2(int *vec, int n, int *min, int *max) {
    int i;
    *min = vec[0];
    *max = vec[0];
    for(i = 1; i < n; i++) {
        if(vec[i] < *min) {           //n - 1
            *min = vec[i];         //A (sempre que true)
        } else {
            if(vec[i] > *max) {     //n - 1 - A
                *max = vec[i];
            }
        }
    }
}
```


Existe um algoritmo ótimo para MinMax

- O mesmo executa com $3n/2 - 2$ comparações
- Veremos ele com cuidado em outra aula

Observações

- Estamos falando de $f(n)$ em números de comparações
- Falamos também no caso da função como um todo
 - Faz sentido no problema do mínimo e máximo
- Na próxima aula vamos generalizar
 - Complexidade de qualquer problema

Exercícios

- Copie o esqueleto de código do banco
 - <https://github.com/flaviovdf/AEDS2-2017-1/tree/master/exemplos/banco-01>
- Implemente todas as funções
- [Q1] Implementa uma função para achar a transação de maior e menor valor. Use a MinMax2 como base
- [Q2] Faça seu código funcionar com mais de 30 transações
 - Não vale apenas aumentar a constante para um valor maior
 - Seu código deve funcionar com infinitas transações
 - Você possivelmente vai ter que ficar re-allocando o vetor