Estrutura de Dados (CC4652)

Aula 4 – Complexidade de Algoritmos

Prof. Luciano Rossi

Ciência da Computação
Centro Universitário FEI

2º Semestre de 2023

- Podemos medir o tempo gasto para executar um programa!
- Mas, essa abordagem pode não ser uma boa opção!
  - Pois depende do hardware!
  - Depende do compilador, que pode otimizar partes do código
- Como podemos fazer então???
  - Podemos estudar o número de vezes que operações são executadas no código
  - Contar as instruções!
  - Contudo, contar todas as instruções pode ser um tanto quanto complicado
  - Então, o foco é sempre maior no termo que mais cresce de acordo com a entrada!

- Dessa forma, a análise de complexidade é feita a partir de uma entrada de n de elementos
- Identificar a complexidade do problema e do algoritmo proposto é importante para verificarmos se temos uma solução eficiente

Complexidade de Algoritmos

Análise de Algoritmos - Exemplo 1

- Determinar quantas vezes o laço executa.

```c
#include <stdio.h>

int main(void) {
        int count = 0;
        int i = 1;
        while(i <= 1000){
                printf("%d\n", i);
                i = i + 1;
                count = count + 1;
        }
        printf("%d\n", count);
  return 0;
}
```

- Como é feita a análise?
- Número de repetições do exemplo 1 = 1000

Complexidade de Algoritmos

Análise de Algoritmos - Exemplo 2

- Determinar quantas vezes o laço executa.

```c
#include <stdio.h>

int main(void) {
        int n;
        scanf("%d", &n);
        int count = 0;
        int i = 1;
        while(i <= n){
                printf("%d\n", i);
                i = i + 1;
                count = count + 1;
        }
        printf("%d\n", count);
  return 0;
}
```

- Qual é a função de complexidade?

- Como é feita a análise?
- Número de repetições do exemplo 2 = n
- Pode-se dizer que o número de repetições segue a entrada n
- Sendo assim, minha função é: $f(n) = n$

- Achar um elemento máximo em um vetor com 10 elementos.

```c
#include <stdio.h>

int main(void) {
	int v[] = {5, 7, 9, 6, 2, 4, 7, 8, 10, 1};
	int max = v[0];
	for(int i = 1; i < 10; i++){
		if(v[i] > max)
			max = v[i];
	}
	printf("%d\n", max);
	return 0;
}
```

- Como é feita a análise?
  - No pior caso, leva 9 iterações
  - Para um vetor com tamanho n, leva $n-1$ iterações
  - Complexidade: $f(n) = n - 1$

- Determinar quantas vezes o laço executa

```c
#include <stdio.h>

int main(void) {
        int count = 0;
        int i = 1;
        while(i <= 1000){
                printf("%d\n", i);
                i = i * 2;
                count = count + 1;
        }
        printf("%d\n", count);
        return 0;
}
```

- Como é feita a análise?
- Se observarmos a operação que altera a variável de controle das repetições: $i = i * 2$, podemos perceber que existe uma relação entre *count* (número de repetições) e i:
  - $i * 2 = 2^{cont}$

i	**i * 2**	**count**	**2^{count}**
1	2	1	2
2	4	2	4
4	8	3	8
8	16	4	16
16	32	5	32
32	64	6	64
,,,	,,,	,,,	,,,

- Como encontrar *count* (número de repetições)?

- Como encontrar *count* (número de repetições)?
- Se $i * 2 = 2^{count}$
- Então $i = \frac{2^{count}}{2} = 2^{count-1}$
- Considerando que $i \leqslant 1000$
- Temos que $2^{count-1} \leqslant 1000 \Rightarrow 2^{count-1} = 1000$
- Portanto:
  - $count - 1 = \log_2 1000$
  - $count - 1 \approx 9,97$
  - $count \approx 9,97 + 1 \approx 10,97$
  - $count \leqslant 10,97$
  - $count = 10$

Complexidade de Algoritmos

Análise de Algoritmos – Exemplo 5

- Determinar quantas vezes o laço executa

```c
#include <stdio.h>

int main(void) {
        int n;
        scanf("%d", &n);
        int count = 0;
        int i = 1;
        while(i <= n){
                i = i * 2;
                count = count + 1;
        }
        printf("%d\n", count);
        return 0;
}
```

- Como é feita a análise?
- Mesmo procedimento do exemplo 4, mas com n no lugar do 1.000
- $\log_2 n = count - 1$
- Exemplo: $n = 50$
- $\log_2 50 = count - 1 \Rightarrow count \approx 6.64$
- $count \leqslant 6.64 \Rightarrow 6$ repetições

Complexidade de Algoritmos

Análise de Algoritmos – Exemplo 6

- Determinar quantas vezes o laço executa

```c
#include <stdio.h>

int main(void) {
        int n;
        scanf("%d", &n);
        int count = 0;
        for(int i = 1; i <= n; i++)
                for(int j = 1; j <= n; j = j * 2)
                        count = count + 1;
        printf("%d\n", count);
        return 0;
}
```

centro universitário FEI

- Como é feita a análise?
- O primeiro for tem o número de repetições $= n$
- O segundo for tem número de repetições $= \log_2 n$
- Combinando os dois temos $n \times \log_2 n$
- Complexidade: $f(n) = n \log_2 n$

Complexidade de Algoritmos

Análise de Algoritmos – Exemplo 6

- Determinar quantas vezes o laço executa

```c
#include <stdio.h>

int main(void) {
	int n;
	scanf("%d", &n);
	int count = 0;
	for(int i = 1; i <= n; i++)
		for(int j = 1; j <= n; j++)
			count = count + 1;
	printf("%d\n", count);
	return 0;
}
```

- Como é feita a análise?
- O primeiro for tem o número de repetições $= n$
- O segundo for tem o número de repetições $= n$
- Combinando os dois temos $n \times n$
- Complexidade: $f(n) = n^2$

Complexidade de Algoritmos

- Para realizarmos uma análise de forma mais correta, verificamos:
  - Melhor caso
  - Pior Caso
  - Caso Médio

Complexidade de Algoritmos

Busca sequencial

- Na busca sequencial, todos os elementos do vetor são comparados ao elemento buscado, um a um.

$Busca(x, V)$

```
para i ← 0 até |V|
    se x = V[i]
        retorna VERDADEIRO
retorna FALSO
```

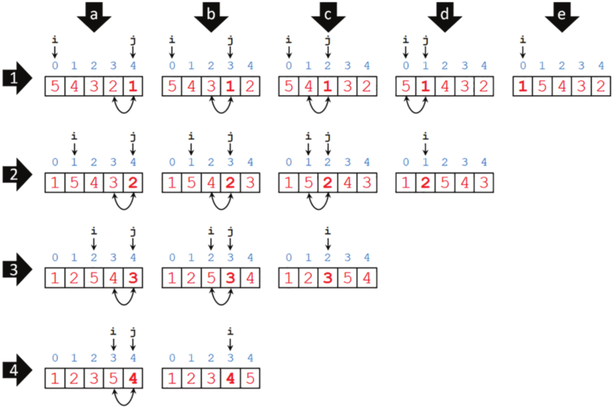
- Pior caso: n
- Melhor caso: 1
- Caso médio: $n/2$

Complexidade de Algoritmos

Bubble Sort

$BubbleSort(V)$

1. $\textbf{\textit{para}}\ i \leftarrow 0\ \textbf{\textit{até}}\ |V|-1$
2. $\quad \textbf{\textit{para}}\ j \leftarrow |V|-1\ \textbf{\textit{até}}\ i+1$
3. $\quad\quad \textbf{\textit{se}}\ V[j] < V[j-1]$
4. $\quad\quad\quad \textbf{\textit{trocar}}\ V[j]\ com\ V[j-1]$



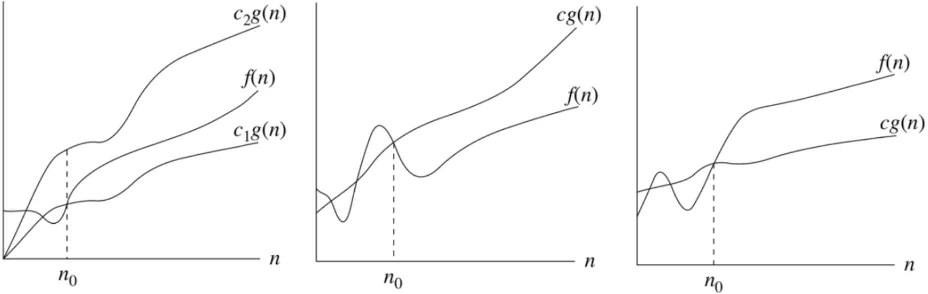
- Pior caso: ?
- Melhor caso: ?
- Caso médio: ?

Notação Assintótica

- A análise da eficiência assintótica dos algoritmos considera uma notação particular que é útil para descrever o tempo de execução do algoritmo, seja qual for a organização da instância de entrada;
- Assim, a notação assintótica fornece um conjunto de "categorias", a partir das quais é possível caracterizar os algoritmos, de acordo com as respectivas eficiências computacionais, seja qual for a entrada considerada.

centro universitário FEI

Notação Assintótica



$f(n) = \Theta(g(n))$ (a)

$f(n) = O(g(n))$ (b)

$f(n) = \Omega(g(n))$ (c)

Notação Assintótica

- Podemos considerar uma analogia para que o entendimento sobre as notações assintóticas apresentadas seja mais consistente;
- A analogia consiste entre a comparação assintótica de duas funções f e g e dois números reais a e b;
- Dessa forma, podemos considerar o seguinte:

$$f(n) = O(g(n)) \Rightarrow a \leq b,$$

$$f(n) = \Omega(g(n)) \Rightarrow a \geq b,$$

$$f(n) = \Theta(g(n)) \Rightarrow a = b,$$

$$f(n) = o(g(n)) \Rightarrow a < b,$$

$$f(n) = \omega(g(n)) \Rightarrow a > b.$$

centro universitário FEI

Notação Assintótica

- Função constante $(f(n) = c)$: nesse tipo de função, independentemente do tamanho da entrada (n), o número de operações executadas será sempre um valor constante;
- Função logarítmica $(f(n) = \log n)$: é uma função comumente observada para algoritmos que consideram a estratégia da divisão e conquista;
- Função linear $(f(n) = n)$: também chamada de função afim ou do primeiro grau, indica que o número de operações realizadas é linearmente proporcional ao tamanho da instância de entrada;

Notação Assintótica

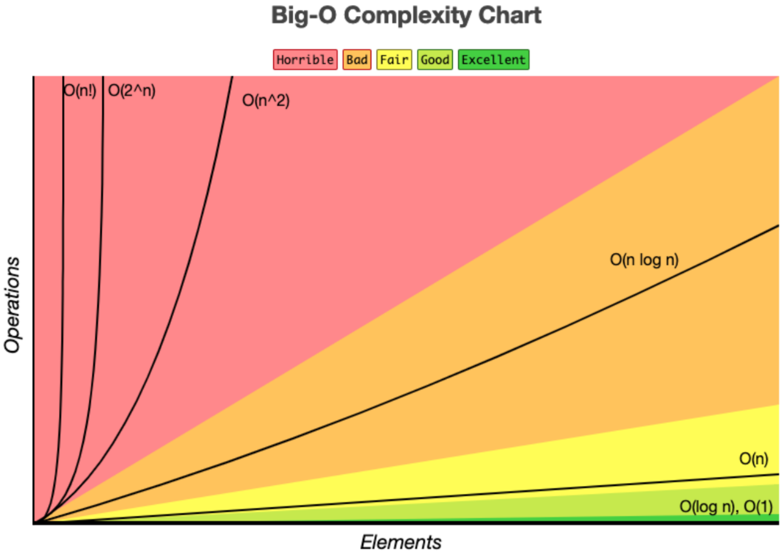
- Função quadrática ($f(n) = n^2$): algoritmos cujo tempo de execução é representado por uma função quadrática normalmente apresentam dois laços de repetição aninhados. Nesses casos, sempre que o tamanho da entrada dobra, o tempo de execução é multiplicado por quatro;
- Função cúbica ($f(n) = n^3$): também conhecida como função do terceiro grau, define o tempo de execução de algoritmos que, comumente, apresentam três laços aninhados, como, por exemplo, na multiplicação de matrizes. Nesse caso, sempre que o tamanho da entrada dobra o tempo de execução é multiplicado por oito;

centro universitário FEI

Notação Assintótica

- Função exponencial ($f(n) = a^n$): algoritmos com o tempo de execução representado por esse tipo de função são pouco úteis, pois sua complexidade cresce muito rapidamente;
- Função fatorial ($f(n) = n!$): tem um comportamento pior que as funções exponenciais; portanto, algoritmos com essa complexidade são, igualmente, pouco úteis do ponto de vista prático.

Notação Assintótica



Estrutura de Dados (CC4652)

Aula 4 – Complexidade de Algoritmos

Prof. Luciano Rossi

Ciência da Computação
Centro Universitário FEI

2º Semestre de 2023