AED2

Análise de algoritmos

Complexidade computacional

> Custo = memória + tempo

- Memória é mais barato que tempo
- Memória pode ser elástica (malloc)
- Outros custos
    - tráfego de rede
- Sempre o tempo de execução e memória é relacionado ao tamanho da entrada

Tipos de Analises

- Análise empírica
    - Comparação entre programas
    - Avalia o custo de um algoritmo avaliando a execução
    - Cronometrar o tempo de execução e monitorar quanto de memória é utilizado
    - Vantagens
        - Permite avaliar o programa no computador/linguagem especifico
        - Considera custos que a análise matemática não considera, exemplo custo de alocação
        - Permite comparar linguagens
        - Permite comparar computadores
    - Desvantagens
        - Necessita implementar o código
            - Depende da habilidade do programador
        - Resultado pode variar muito
            - Computador utilizado
            - Processos em paralelo no momento da avaliação
        - Depende da natureza dos dados
            - Dados reais,
            - Dados aleatórios
                - Desempenho médio
            - Dados perversos
                - Sempre força o pior caso do algoritmo
- Análise matemática
    - Estudo das propriedades do algoritmo
    - Estudar como o algoritmo se comporta a medida que a entrada cresce em volume
    - Independente de variáveis de máquina
    - Estudo em um computador idealizado (simplificado)
        - Ex.: soma tem um custo maior que a multiplicação, porém para simplificar ignoramos

- Só considera custos dominates do algoritmo
- Para *n* entradas, o custo aumenta em *n*, n^2, *n!*, etc...
- Vejamos o seguinte algoritmo

```
int M = A[0];
for(i = 0; i<n; i++){
  if(A[i] >=M){
    M = A[i];
  }
}
```

- Ignorando dentro do for, o algoritmo executara 3 + 2n instruções
  - *n* = tamanho array
- Logo, considerando um laço vazio, podemos criar uma função matemática
- f(n) = 3+2n
- Considerando agora dentro do if e os seguintes arrays

```
int A1[4] = {1,2,3,4}
int A2[4] = {4,3,2,1}
```

- A1: comando if sempre testa **positivo**
- A2: comando if sempre testa **negativo**
- **Sempre devemos considerar o *MAIOR* número de instruções**
- No pior caso o algoritmo será
  - f(n) = 3+2n+2n

Comportamento assintótico

- Se um algoritmo é mais rápido para um grande conjunto, ele vai ser mais rápido que o outro pra um pequeno conjunto
- A função f(n) = 4n+3
  - 3 é uma *constante de inicialização*, logo desconsideramos

Notação Grande-O

- O pior que o algoritmo pode ser é o Big-O
- Não ultrapassa esse limite
- Normalmente pode ser visto pela quantidade de laços aninhados

- Exemplo com selection sort:

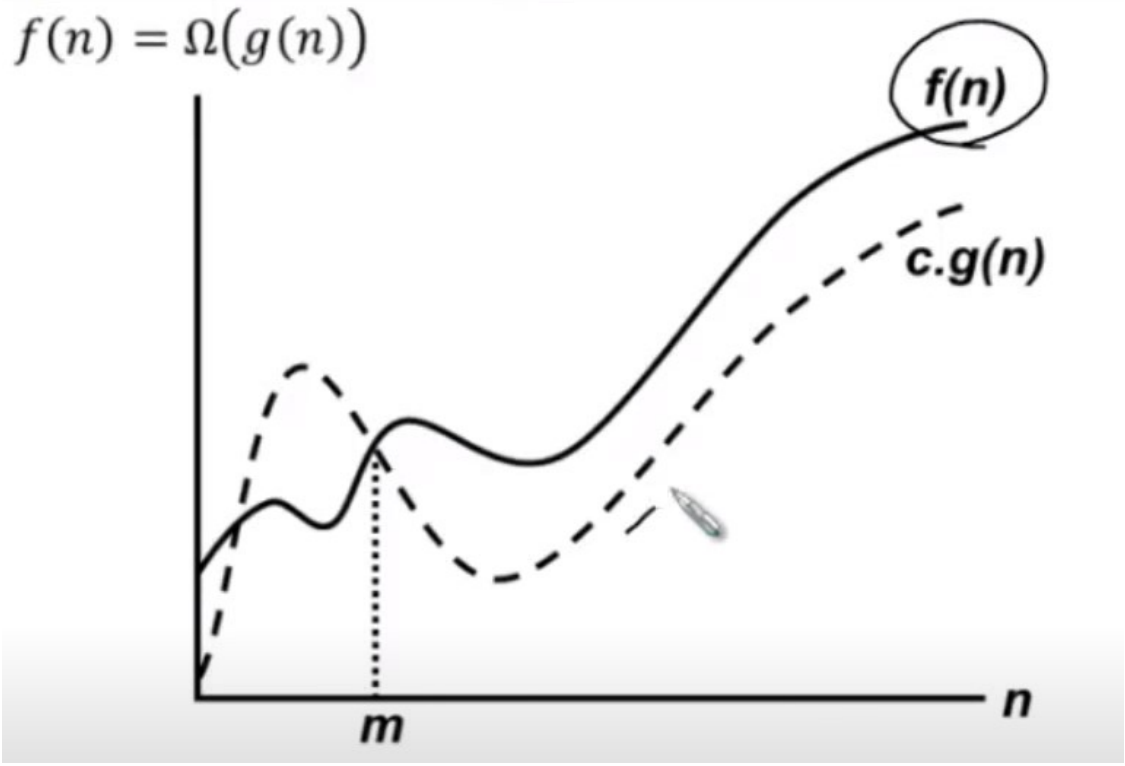
```
int i,j,me,troca;
  for (i = 0; i < (n - 1); i++)
{
  me = i;

  for (j = i+1; j < n; j++)
  {
    if (array[j] < array[me])
      me = j;
  }
  if (i != me)
  {
    troca = array[i];
    array[i] = array[me];
    array[me] = troca;
  }
}
```

- Primeiro deve se calcular a soma `nEx = 1+2+3+...+(n-1)+n`
- nEx é o numero de execuções do laço interno (não é simples de calcular)
- Nesse caso é nEx equivalente a uma *prograssão aritmética* de razão 1
- [n(1+1)]/2
- Alternativa é estimar o *limite superior*
  - Supomos o pior caso para o algoritmo
  - Logo o algoritmo original é tão ruim ou melhor do que esse que calcularemos
  - Como calcular isso?
    - Trocar o laço interno (que varia de tamanho dependendo do laço externo) por um laço constante (n)
    - Simplifica a análise
    - Piora o desempenho
      - Algumas execuções do laço são inúteis
    - Sobramos com 2 laços aninhados cada um executando *n* vezes
    - Logo sobramos com uma função de custo $f(n) = n^2$
    - Utilizando a notação big-O o custo no pior caso é $O(n^2)$
  - Com isso chegamos a função que delimita o limite superior de custo de tempo
  - O custo na prática pode ser melhor que n^2, mas nunca melhor

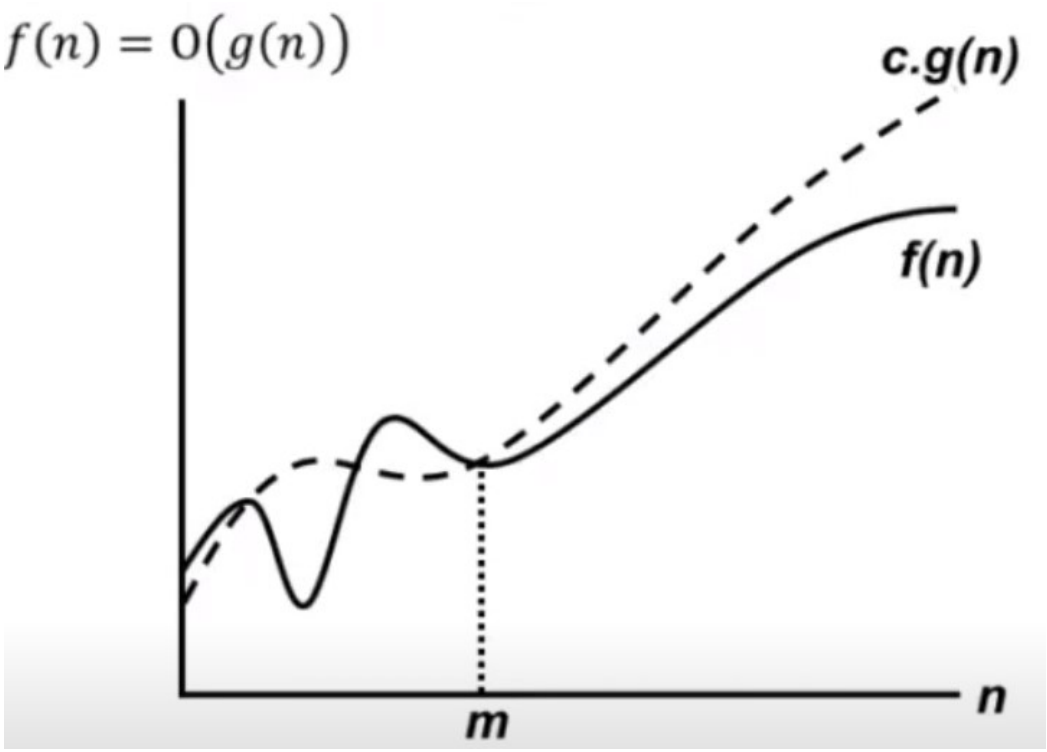
Outros tipos de análise

> Apesar da Big-O ser útil e recorrente, existem outras análises

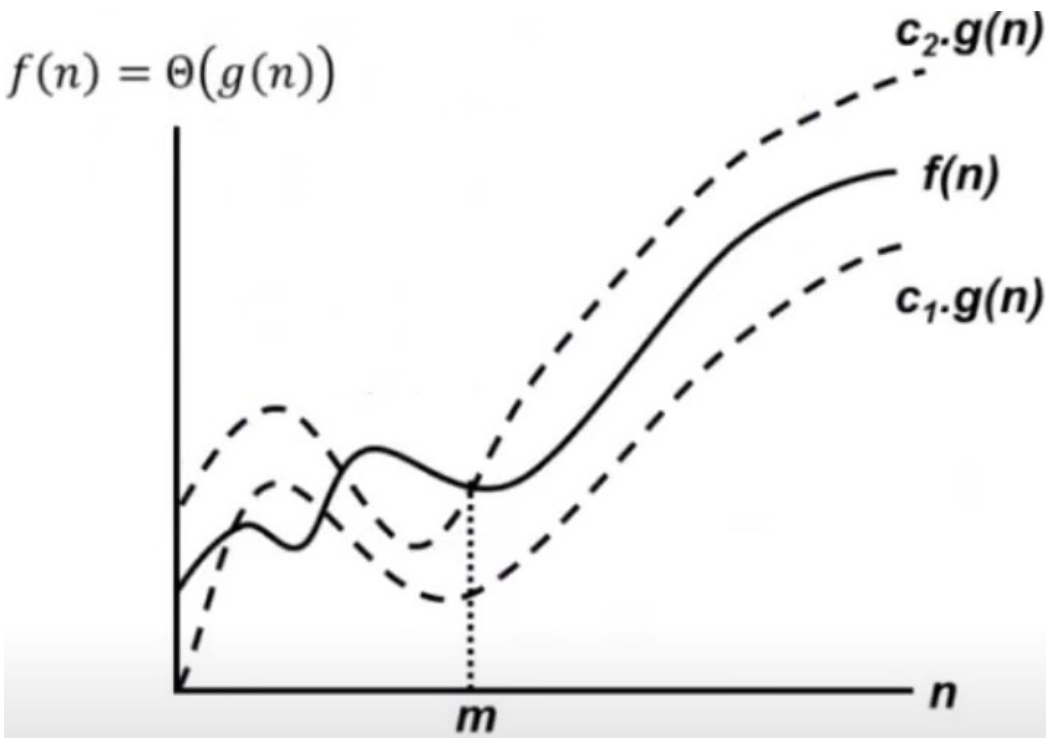
- Grande-Omega
  - Limite assintótico inferior

- *Melhor caso*
- $\Omega(n^2)$ diz que na melhor das hióteses, o custo é n^2
- Não fica melhor que isso em questão de custo, no máximo igual
- Como definir:
  - Função $f(n)$ é $\Omega(g(n))$
    - Se existem 2 constantes >0 chamamos de *c* e *m*
    - *n>=m*
    - Temos $f(n) >= c* g(n)$
- Para todos os valores de *n* a direita de *m* o resultado de $f(n)$ é sempre >= que o valor da notação Grande-Omega x *c*
- Exemplo:
  - Função com custo $f(n) = 3n^3 + 2n^2$ é $\boldsymbol{\Omega(n^3)}$
  - Consideramos *c = 1* e *n >= 0*
  - Logo $3n^3 + 2n^2 >= 1n^3$
  - 

  - Após o ponto *m* o custo de $f(n)$ é sempre ***maior*** que o custo de $c*g(n)$
- Grande-O
  - Pior caso
  - Função $f(n)$ é $O(g(n))$
  - Se existem 2 constantes >0 chamamos de *c* e *m*
    - *n>=m*
    - Temos $f(n) <= c* g(n)$
      - Para todos os valores de *n* a direita de *m* o resultado de $f(n)$ é sempre <= que o valor da notação Grande-O x *c*
    - Exemplo:
      - Função com custo $f(n) = 3n^3 + 2n^2$ é $\boldsymbol{O(n^3)}$
      - Consideramos *c = 6* e *n >= 0*
      - Logo $3n^3 + 2n^2 <= 6n^3$



    - Após o ponto *m* o custo de `f(n)` é sempre ***menor*** que o custo de `c*g(n)`
  - Regra da Soma
    - Importante quando temos algoritmos em *sequência*
    - Nesse caso a complexidade de execussão é dada pela complexidade do **maior** deles
    - `O(f(n)) + O(g(n)) = O(max(f(n),g(n)))`
    - Exemplo:
      - Primeiro será executado o algoritmo O(n) e depois o algoritmo $O(n^2)$
      - Logo a complexidade de tudo é $O(n^2)$
- Grande-Theta
  - Limite assintótico **firme** ou **estreito**
  - Analisa o limite inferior e superior *ao mesmo tempo*
  - $\theta(n^2)$ diz assintóticamente que o custo do algoritmo original é n^2
  - É como se tivessem 2 constantes, uma restringindo abaixo, e outra restringindo acima
  - Função `f(n)` é `ϑ(g(n))`
  - Se existem 3 constantes >0 chamamos de *c1*, *c2* e *m*
    - Para *n*>=*m*
    - Temos `c1 * g(n) <= f(n) <= c2* g(n)`
      - Para todos os valores de *n* a direita de *m* o resultado de `f(n)` é sempre = ao valor da notação Grande-θ x *c1* e x *c2*
    - Exemplo:
      - Função com custo `f(n) = 1/2n² - 3n` é ***`ϑ(n²)`***
      - Consideramos *c1 = 1/14*, *c2 = 1/2* e *n >= 7*
      - Logo `1/14n² <= 1/2n² - 3n <= 1/2n²`



    - Após o ponto *m* o custo de `f(n)` é sempre ***menor*** que o custo de `c2 * g(n)` e ***maior*** que o custo de `c1 * g(n)`
- Pequeno-o
    - Parecida com a Grande-O
    - Enquanto no Grande-O a relação é <=
    - Já na pequeno-o a relação é <
    - O custo real sempre será MENOR que o pequeno-o e *nunca igual*
- Pequeno-omega
    - Parecida com a Grande-Omega
    - Enquanto no Grande-Omega a relação é >=
    - Já na pequeno-omega a relação é >
    - O custo real sempre será MAIOR que o pequeno-o e *nunca igual*

Classes de problemas

- `O(1)`
    - Ordem constante
    - Não depende do tamanho da entrada de dados
- `θ(log(n))`
    - Ordem logarítmica
    - Resolve um problema transformando ele em problemas menores
    - Exemplo:
        - Realizar uma busca binária em um vetor de tamanho *n*
- `O(n)`
    - Ordem linear
    - Uma operação é realizada para cada um dos elementos da entrada
    - Exemplo:
        - Printar um vetor de tamanho *n*
- `θ(nlog(n))`
    - Ordem log linear

    - Trabalha com particionamento de dados
    - Transforma o problema em partes menores dele mesmo e resolve de forma idependende, depois une tudo
    - Exemplo:
        - QuickSort
- $O(n^2)$
    - Ordem quadratica
    - Dados são processados em pares
    - Normalmente tem presença de aninhamento de laços
    - Exemplo:
        - Printar uma matriz de tamanho *n* por *n*
- $O(n^3)$
    - Ordem cúbica
    - Normalmente tem presença de aninhamento de 3 laços
    - Exemplo:
        - Printar uma matriz de tamanho *n* por *n* por *n*
- $O(2^n)$
    - Ordem exponencial
    - Solução de força bruta
    - Não é útil do ponto de vista prático
    - Muito lento
    - Exemplo:
        - Quebrar senha de sistema, é necessário testar todas as combinações
- $O(n!)$
    - Ordem fatorial
    - Solução de força bruta
    - Não é útil do ponto de vista prático
    - Muito lento
    - Comportamento **muito** pior que a exponencial

> Consideremos um computador que executa 1000000 operações por segundo vamos criar uma tabela que mostra o tempo de execução do algoritmo com base no tamanho de entrada

$f(n)$	**n = 10**	**n = 20**	**n = 30**	**n = 50**	**n = 100**
n	1x10^-5 segundos	2x10^-5 segundos	4x10^-5 segundos	5x10^-5 segundos	6x10^-5 segundos
$nlog(n)$	3.3x10^-5 segundos	8.6x10^-5 segundos	2.1x10^-4 segundos	2.8x10^-4 segundos	2.5x10^-4 segundos
n^2	1x10^-4 segundos	4x10^-4 segundos	1.6x10^-3 segundos	2.5x10^-3 segundos	3.6x10^-3 segundos
n^3	1x10^-3 segundos	8x10^-3 segundos	6.4x10^-2 segundos	0.13 segundos	0.22 segundos
2^n	0.001 segundos	1 segundos	2.8 dias	35.7 anos	365.6 séculos

f(n)	n = 10	n = 20	n = 30	n = 50	n = 100
3^n	0.059 segundos	58 minutos	3855.2 séculos	2.3x10^8 séculos	1.3x10^13 séculos

> O problema da análise assintótica é que em entradas pequenas as simplificações são problemáticas

- Considerando as funções *f(n) = (10^100)n* e *g(n) = 10n*logn*
  - Big-O da primeira é *O(n)* e da segunda é *O(nlogn)*
  - A primeira seria considerada mais performática de forma assintótica
- Ao analisár a constante multiplicativa de n em f(n) descobrimos o número 10^100
  - Logo *10n*logn* > *f(10^100)n* apenas para ***n>2^(10^99)***
  - para qualquer valor abaixo desse *n* *10n*logn* < *f(10^100)n*
    - Ou seja, na prática *10n*logn* é mais performático

Relações de recorrência

> Relembrando conceito de função recursiva

- Função que chama a sí mesmo durante a execução
- Exemplo: fatorial(n)
  - Matemáticamente é definido por *0! = 1; N! = N(N-1)*
  - Implementação em C:

```c
int fat(int n){
  if(n==0)
    return 1;
  else
    return n*fat(n-1);
}
```

> *Recorrência* ou *Relação de recorrência*

- Expressão que descreve a função em termos de entradas menores da função
- Muitos algoritmos são baseados em recorrência
- Importante para solução de problemas combinatórios
- É uma função recursiva

> Relação de recorrência do fatorial

T(n) = T(n-1) + n

> Complexidade da recorrência

- Não utiliza laços de repetição
- Erroneamente imaginamos que a função possuíria complexidade ***O(1)***
- Para saber a complexidade tem que resolver a relação de recorrência
- Encontrar a *fórmula fechada* que da o valor da função em termos de *n*

- Considere o relação `T(n) = T(n-1) + 2n + 3`
  - Para *n* em *{2,3,4,...}*
  - Existem inumeras funções *T* que satisfazem a recorrência
  - Depende do caso base: T(1)
  - Considerando T(1) = 1

n	**1**	**2**	**3**	**4**	**5**
T(n)	1	8	17	28	41

  - Considerando T(1) = 1

n	**1**	**2**	**3**	**4**	**5**
T(n)	5	12	21	32	45

  - **Problema**: *muitas soluções possíveis*
    - Para cada valor *i* e *n* em *{2,3,4,...}* existe ***uma*** função *T* que tem caso base `T(1) = i` e satisfaz a recorrência
  - **Solução**: expandir a relação até encontrar o comportamento no caso geral e assim encontrar a **fórmula fechada**

Expandindo a relação de recorrência

- Considerando a seguinte relação `T(n) = T(n-1) + 3`
  - Representa um algoritmo que possui *3 operações* e *1 chamada recursiva*
- Resolução
  - Substituir o termo `T(n-1)` sobre `T(n)`, teremos:
    - `T(n-1) = T(n-2) + 3`
  - Substituir o termo `T(n-2)` sobre `T(n-1)`, teremos:
    - `T(n-2) = T(n-3) + 3`
  - Juntar as 3 expressões
    - `T(n) = T(n-1) + 3`
    - `T(n) = (T(n-2) + 3) + 3`
    - `T(n) = ((T(n-3) + 3) + 3) + 3`
      - Simplificando tudo (somente para claridade)
        - `T(n) = T(n-1) + 3`
        - `T(n) = T(n-2) + 3*2`
        - `T(n) = T(n-3) + 3*3`
  - A cada passo se soma 3 e o valor de n é diminuído em 1
  - Podemos resumir a expansão com a equação
  - `T(n) = T(n-k) + 3*k`
  - O processo de expansão acaba no caso base T(1), logo quando: `n-k = 1`, ou seja quando `k = n-1`
  - Substituindo *k* na equação `T(n) = T(n-k) + 3*k` por *n-1*
    - `T(n) = T(1) + 3*(n-1)`
    - `T(n) = T(1) + 3n-3`
    -

Analisando a complexidade Big-O dessa recursão

- Complexidade da recorrência `T(n) = T(n-1) + 3`

- `T(n) = T(n-k) + 3k`
- `T(n) = T(1) + 3(n-1)`
- `T(n) = T(1) + 3n-3`
    - `T(1)` é o caso base, que é o retorno e o custo é constante: ***O(1)***
    - Substituindo `T(1)` por `O(1)`
- `T(n) = 3n-3 + O(1)`
- Ou seja, a complexidade é linear `O(n)`