Algoritmos e Estruturas de Dados I
Aula 02: Análise de Algoritmos

Prof. Márcio Porto Basgalupp

créditos: Prof. Jurandy G. Almeida Jr.

Universidade Federal de São Paulo
Departamento de Ciência e Tecnologia

Como escolher o algoritmo mais adequado para uma situação?

Análise de Algoritmos

- Analisar um algoritmo consiste em “verificar” o que?
  - **Tempo** de Execução
  - **Espaço** Ocupado

- Esta análise é necessária para escolher o algoritmo mais adequado para resolver um dado problema.
- É especialmente importante em áreas como pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras.

Cálculo do Custo

- **Cálculo do Custo pela Execução do Algoritmo**
  - Tal medida é bastante **inadequada** e o resultado não pode ser generalizado
  - Os resultados são dependentes do compilador, que pode favorecer algumas construções em detrimento de outras
  - Os resultados dependem do *hardware*
  - Quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto

Cálculo do Custo

- **Cálculo do Custo pela Execução do Algoritmo**
  - Apesar disso, há argumentos a favor de se obterem medidas reais de tempo
  - Ex.: quando há vários algoritmos distintos para resolver um mesmo tipo de problema, todos com um custo de execução dentro de uma mesma **ordem de grandeza**

Análise de Algoritmos

- Possibilidade de analisar:
  - Um **algoritmo** particular
  - Uma **classe** de algoritmos

Análise de Algoritmos

- Análise de um **algoritmo particular**
  - Qual é o custo de usar um dado algoritmo para um resolver um problema específico?
  - Características que devem ser investigadas:
    - Análise do número de vezes que cada parte do algoritmo deve ser executada (**tempo**)
    - Estudo da quantidade de memória necessária (**espaço**)

Análise de Algoritmos

- Análise de uma **classe de algoritmos**
  - Qual é o algoritmo de menor custo possível para resolver um problema particular?
  - Toda uma família de algoritmos é investigada
  - Procura-se identificar um que seja o melhor possível
  - Coloca-se **limites** para a complexidade computacional dos algoritmos pertencentes à classe

Custo de um Algoritmo

- O menor custo possível para resolver problemas de uma classe nos dá a dificuldade inerente para resolver o problema
- Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é **ótimo** para a medida de custo considerada
  - Podem existir vários algoritmos ótimos para resolver o mesmo problema
- Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado

Custo de um Algoritmo

- Utilizaremos um modelo matemático baseado em um computador idealizado
- Deve ser especificado o conjunto de operações e seus custos de execuções
- É mais usual ignorar o custo de algumas operações e considerar apenas outras mais significativas
- Ex.: algoritmos de ordenação. Considera-se o **número de comparações** entre os elementos do conjunto a ser ordenado e ignoramos as demais operações

Função de Complexidade

- Para medir o custo de execução de um algoritmo vamos definir uma **função de complexidade** ou função de custo ***f***
- Função de **complexidade de tempo**: *f(n)* mede o tempo necessário para executar um algoritmo em um problema de tamanho *n*
- Função de **complexidade de espaço**: *f(n)* mede a memória necessária para executar um algoritmo em um problema de tamanho *n*

Função de Complexidade

- Utilizaremos f para denotar uma função de complexidade de tempo daqui para a frente
- A complexidade de tempo na realidade não representa tempo diretamente
  - Representa o número de vezes que determinadas operações relevantes são executadas

Exemplo: Maior Elemento

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros A[*n*], $n \geq 1$

```
int Max(int* A, int n) {
    int i, Temp;

    Temp = A[0];
    for (i = 1; i < n; i++)
        if (Temp < A[i])
            Temp = A[i];
    return Temp;
}
```

- Seja ***f*** uma função de complexidade tal que ***f(n)*** é o número de comparações envolvendo os elementos de A, se A contiver *n* elementos. **Qual é a função *f(n)*?**

Exemplo: Maior Elemento

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros A[*n*], $n \geq 1$

```
int Max(int* A, int n) {
    int i, Temp;

    Temp = A[0];
    for (i = 1; i < n; i++)
      if (Temp < A[i])
        Temp = A[i];
    return Temp;
}
```

(n-1)

- Seja ***f*** uma função de complexidade tal que ***f(n)*** é o número de comparações envolvendo os elementos de A, se A contiver *n* elementos. **Qual é a função *f(n)*?**

Exemplo: Maior Elemento

UNIFESP
UNIVERSIDADE FEDERAL DE SÃO PAULO
1933

Teorema

Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, $n \geq 1$, faz pelo menos $n - 1$ comparações.

Exemplo: Maior Elemento

UNIFESP
UNIVERSIDADE FEDERAL DE SÃO PAULO
1933

Teorema

Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, $n \geq 1$, faz pelo menos $n - 1$ comparações.

O teorema acima nos diz que, se o número de comparações for utilizado como medida de custo, então a função Max do programa anterior é **ótima**.

Tamanho da Entrada de Dados

- A medida do custo de execução de um algoritmo **depende principalmente** do tamanho da entrada dos dados
- É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada

Tamanho da Entrada de Dados

- No caso da função Max do programa do exemplo, o **custo é uniforme** sobre todos os problemas de tamanho *n*
- Já para um **algoritmo de ordenação** isso não ocorre: se os dados de entrada já estiverem quase ordenados, então pode ser que o algoritmo trabalhe menos

Casos a Serem Analisados

- **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*

Melhor Caso ≤ Caso Médio ≤ Pior Caso

Análise de Cada Caso

- Na análise do caso médio, supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho *n* sobre a qual o custo esperado é obtido
- A análise do caso médio é geralmente mais difícil de se obter do que as análises do melhor e do pior caso
- É comum supor uma distribuição de probabilidades em que todas as entradas são igualmente prováveis
  - Na prática, isso nem sempre é verdade!

Ex.: Maior e Menor Elemento (1)

- Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros A[n], $n \geq 1$

```
void MaxMin1(int* A, int n, int* pMax, int* pMin)
{
   int i;

   *pMax = A[0];
   *pMin = A[0];
   for (i = 1; i < n; i++) {
      if (A[i] > *pMax) *pMax = A[i];
      if (A[i] < *pMin) *pMin = A[i];
   }
}
```

Qual a Função de Complexidade?

```
void MaxMin1(int* A, int n, int* pMax, int* pMin)
{
    int i;

    *pMax = A[0];
    *pMin = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        if (A[i] < *pMin) *pMin = A[i];
    }
}
```

Qual a Função de Complexidade?

```
void MaxMin1(int* A, int n, int* pMax, int* pMin)
{
    int i;

    *pMax = A[0];
    *pMin = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        if (A[i] < *pMin) *pMin = A[i];
    }
}
```

Qual a Função de Complexidade?

```
void MaxMin1(int* A, int n, int* pMax, int* pMin)
{
    int i;

    *pMax = A[0];
    *pMin = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        if (A[i] < *pMin) *pMin = A[i];
    }
}
```

2*(n-1)

Qual a Função de Complexidade?

- Seja *f(n)* o número de comparações entre os elementos de A. Logo, *f(n) = 2(n - 1)* para o melhor caso, pior caso e caso médio.

```
void MaxMin1(int* A, int n, int* pMax, int* pMin)
{
   int i;

   *pMax = A[0];
   *pMin = A[0];
   for (i = 1; i < n; i++) {
      if (A[i] > *pMax) *pMax = A[i];
      if (A[i] < *pMin) *pMin = A[i];
   }
}
```

Ex.: Maior e Menor Elemento (2)

- MinMax1 pode ser facilmente melhorado: a comparação A[i] < *pMin só é necessária quando a comparação A[i] > *pMax dá falso.

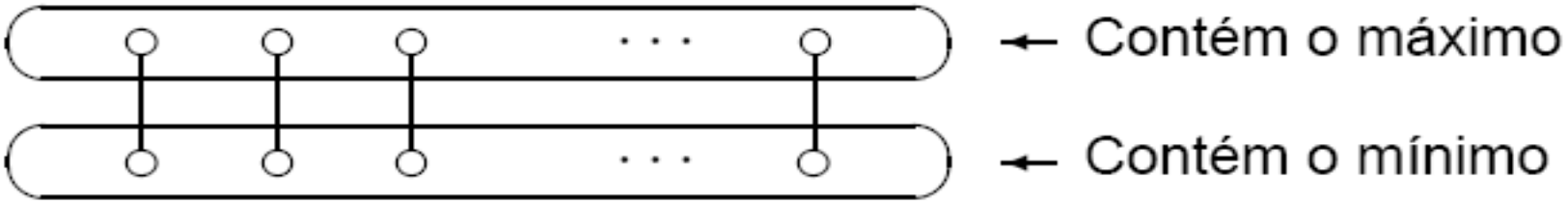
```
void MaxMin2(int* A, int n, int* pMax, int* pMin)
{
   int i;

   *pMax = A[0];
   *pMin = A[0];
   for (i = 1; i < n; i++) {
      if (A[i] > *pMax) *pMax = A[i];
      else if (A[i] < *pMin) *pMin = A[i];
   }
}
```

Qual a Função de Complexidade?

- **Melhor caso**:
  - quando os elementos estão em ordem crescente
  - ***f(n) = n - 1***
- **Pior caso**:
  - quando o maior elemento é o primeiro no vetor
  - ***f(n) = 2(n - 1)***
- **Caso médio**:
  - A[i] é maior do que Max a metade das vezes
  - ***f(n) = 1,5(n-1) = 3/2 (n-1) = 3n/2 - 3/2***

Ex.: Maior e Menor Elemento (3)

- Considerando o número de comparações realizadas, é possível obter um algoritmo mais eficiente:
  - Compare os elementos de A aos pares, separando-os em dois subconjuntos (maiores em um e menores em outro), a um custo de $\lceil n/2 \rceil$ comparações.
  - O máximo é obtido do subconjunto que contém os maiores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.
  - O mínimo é obtido do subconjunto que contém os menores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.



Ex.: Maior e Menor Elemento (3)

```
void MaxMin3(int* A, int n, int* pMax, int* pMin)
{
  int i;

  if ((n % 2) > 0)        { *pMax = A[n-1]; *pMin = A[n-1]; }
  else
  {
    if (A[n-2] > A[n-1]) { *pMax = A[n-2]; *pMin = A[n-1]; }   → Comparação 1
    else                 { *pMax = A[n-1]; *pMin = A[n-2]; }
  }

  for (i = 1; i < n-1; i += 2)
  {
    if (A[i - 1] > A[i])                                       → Comparação 2
    {
      if (A[i - 1] > *pMax) *pMax = A[i - 1];                  → Comparação 3
      if (A[i]     < *pMin) *pMin = A[i];                      → Comparação 4
    }
    else
    {
      if (A[i - 1] < *pMin) *pMin = A[i - 1];                  → Comparação 3
      if (A[i]     > *pMax) *pMax = A[i];                      → Comparação 4
    }
  }
}
```

Qual a Função de Complexidade?

- Quantas comparações são feitas em MaxMin3?
  - 1ª comparação feita 1 vez
  - 2ª comparação feita $n/2$ - 1 vezes
  - 3ª e 4ª comparações feitas $n/2$ - 1 vezes

$$f(n) = 1 + n/2 - 1 + 2 * (n/2 - 1)$$

$$f(n) = (3n - 6)/2 + 1$$

$$f(n) = 3n/2 - 3 + 1 = 3n/2 - 2$$

No pior caso, melhor caso e caso médio

Comparação entre os Algoritmos

- A tabela apresenta uma comparação entre os algoritmos dos programas MaxMin1, MaxMin2 e MaxMin3, considerando o número de comparações como medida de complexidade
- Os algoritmos MaxMin2 e MaxMin3 são superiores ao algoritmo MaxMin1 de forma geral.
- O algoritmo MaxMin3 é superior ao algoritmo MaxMin2 com relação ao pior caso e bastante próximo quanto ao caso médio

Os três algoritmos	$f(n)$		
	Melhor caso	Pior caso	Caso médio
MaxMin1	$2(n-1)$	$2(n-1)$	$2(n-1)$
MaxMin2	$n-1$	$2(n-1)$	$3n/2-3/2$
MaxMin3	$3n/2-2$	$3n/2-2$	$3n/2-2$

Comportamento Assintótico

- O parâmetro ***n*** fornece uma medida de dificuldade para se resolver o problema
- Para valores suficientemente pequenos de ***n***, qualquer algoritmo custa pouco para ser executado
- A **escolha do algoritmo** não é um problema crítico para problemas de tamanho pequeno
- Logo, a análise de algoritmos é realizada para valores grandes de ***n***

Comportamento Assintótico

- Para isso, estuda-se o **comportamento assintótico** das funções de custo
  - Comportamento de suas funções de custo para valores suficientemente grandes de ***n***

- O comportamento assintótico de ***f(n)*** representa o limite do comportamento do custo quando ***n*** cresce

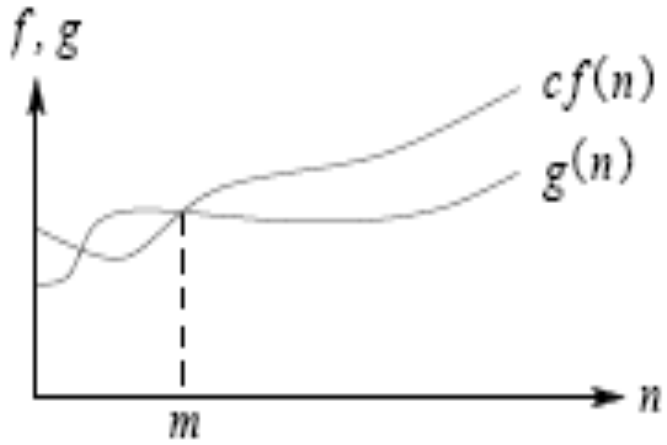
Comportamento Assintótico

- A análise de um algoritmo geralmente conta apenas com algumas operações elementares
- A medida de custo ou medida de complexidade relata o **crescimento assintótico** da operação considerada

- **Definição:** Uma função ***f(n)*** **domina assintoticamente** uma outra função ***g(n)*** se:
  - Existem duas constantes positivas ***c*** e ***m*** tais que, para ***n*** $\geq m$, temos $|g(n)| \leq c\,|f(n)|$

Dominação Assintótica

- ***f(n)* domina assintoticamente *g(n)*** se:
  - Existem duas constantes positivas ***c*** e ***m*** tais que, para ***n*** ***≥ m***, temos ***|g(n)| ≤ c |f(n)|***



Exemplo

- Sejam $g(n) = (n + 1)^2$ e $f(n) = n^2$

- As funções $g(n)$ e $f(n)$ dominam assintoticamente uma a outra desde que

$$|(n + 1)^2| \leq 4\ |n^2| \text{ para } n \geq 1$$

e

$$|n^2| \leq |(n + 1)^2| \text{ para } n \geq 0$$

Exemplo

- Sejam $g(n) = (n + 1)^2$ e $f(n) = n^2$

- As funções $g(n)$ e $f(n)$ dominam assintoticamente uma a outra desde que

$|g(n)| \leq c\ |f(n)|$, para $n \geq m$

$c = 4$ e $m = 1$

$$|(n + 1)^2| \leq 4\ |n^2| \text{ para } n \geq 1$$

e

$$|n^2| \leq |(n + 1)^2| \text{ para } n \geq 0$$

Exemplo

- Sejam $g(n) = (n + 1)^2$ e $f(n) = n^2$

- As funções $g(n)$ e $f(n)$ dominam assintoticamente uma a outra desde que

$$|(n + 1)^2| \leq 4\ |n^2| \text{ para } n \geq 1$$

e

$$|n^2| \leq |(n + 1)^2| \text{ para } n \geq 0$$

$|g(n)| \leq c\ |f(n)|$, para $n \geq m$

$c = 1$ e $m = 0$

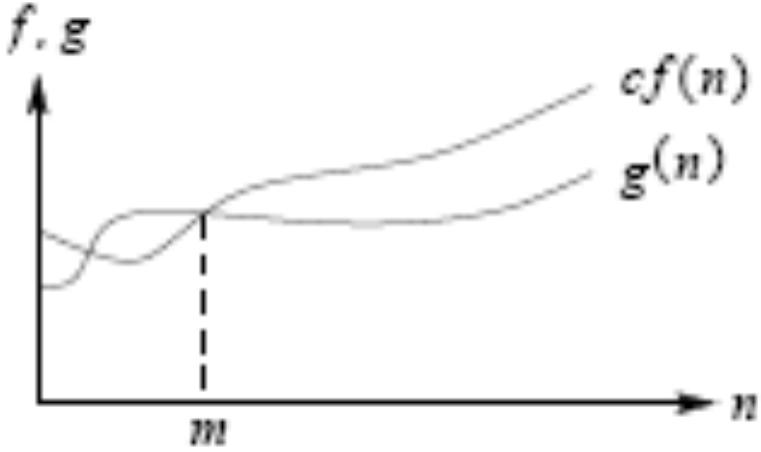
Notação O

- Escrevemos ***g(n) = O(f(n))*** para expressar que ***f(n)*** domina assintoticamente ***g(n)***
  - Lê-se: ***g(n)*** é da ordem no máximo ***f(n)***

- Exemplo:
  - Quando dizemos que o tempo de execução ***T(n)*** de um programa é ***$O(n^2)$***, significa que existem constantes ***c*** e ***m***, tais que, para valores de ***$n \geq m$***, ***$T(n) \leq c\ n^2$***

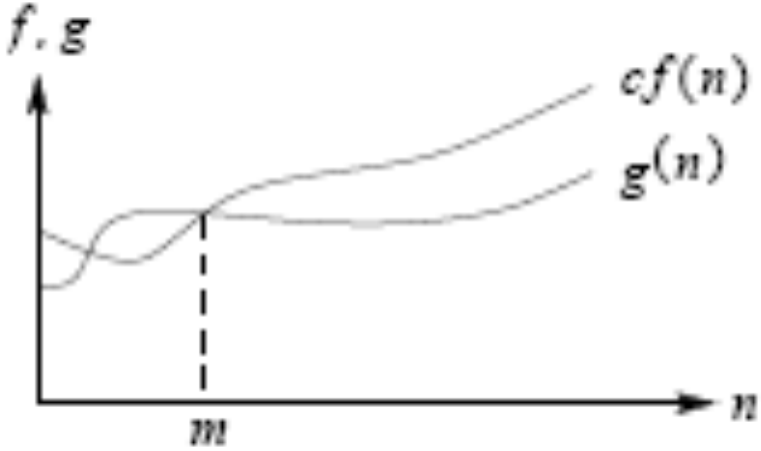
Notação O

- Exemplo de gráfico de dominação assintótica que ilustra a notação O
  - Abaixo, a função ***f(n)*** domina assintoticamente a função ***g(n)***



Notação O

- O valor da constante ***m*** mostrado é o menor valor possível, mas qualquer valor maior também é válido
- **Definição:** uma função ***g(n)*** é ***O(f(n))*** se existem duas constantes positivas ***c*** e ***m*** tais que ***g(n) ≤ c f(n)***, para todo ***n ≥ m***



Exemplo de Notação O

- Exemplo: $g(n) = (n + 1)^2$
  - Logo, $g(n)$ é $O(n^2)$ quando $m = 1$ e $c = 4$
  - Isto porque $(n + 1)^2 \leq 4\ n^2$ para $n \geq 1$

Exemplo de Notação O

- Exemplo: $g(n) = (n + 1)^2$
  - Logo, $g(n)$ é $O(n^2)$ quando $m = 1$ e $c = 4$
  - Isto porque $(n + 1)^2 \leq 4\ n^2$ para $n \geq 1$

existe *c* tal que
g(n) ≤ c f(n)
para *n ≥ m*

Exemplo de Notação O

- Exemplo: $g(n) = n$ e $f(n) = n^2$
  - Sabemos que $g(n)$ é $O(n^2)$, pois para $n \geq 1$, $n \leq n^2$
  - Entretanto, $f(n)$ não é $O(n)$
  - Suponha que existam constantes c e m tais que para todo $n \geq m$, $n^2 \leq c\ n$
  - Se $c \geq n$ para qualquer $n \geq m$, então deveria existir um valor para c que possa ser maior ou igual n para todo n

Exemplo de Notação O

- Exemplo: $g(n) = n$ e $f(n) = n^2$
  - Sabemos que $g(n)$ é $O(n^2)$, pois para $n \geq 1$, $n \leq n^2$
  - Entretanto, $f(n)$ não é $O(n)$
  - Suponha que existam constantes c e m tais que para todo $n \geq m$, $n^2 \leq c\ n$
  - Se $c \geq n$ para qualquer $n \geq m$, então deveria existir um valor para c que possa ser maior ou igual n para todo n

não existe *c* tal que
g(n) ≤ c f(n)
para *n ≥ m*

Exemplo de Notação O

- Exemplo: $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$
  - Basta mostrar que $3n^3 + 2n^2 + n \leq 6n^3$ para $n \geq 0$
  - A função $g(n) = 3n^3 + 2n^2 + n$ é também $O(n^4)$, entretanto, essa afirmação é mais fraca do que dizer que $g(n)$ é $O(n^3)$

Exemplo de Notação O

- Exemplo: $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$
  - Basta mostrar que $3n^3 + 2n^2 + n \leq 6n^3$ para $n \geq 0$
  - A função $g(n) = 3n^3 + 2n^2 + n$ é também $O(n^4)$, entretanto, essa afirmação é mais fraca do que dizer que $g(n)$ é $O(n^3)$

existe *c* tal que
g(n) ≤ c f(n)
para *n ≥ m*

Exemplo de Notação O

- Exemplo: $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$
  - Basta mostrar que $3n^3 + 2n^2 + n \leq 6n^3$ para $n \geq 0$
  - A função $g(n) = 3n^3 + 2n^2 + n$ é também $O(n^4)$, entretanto, essa afirmação é mais fraca do que dizer que $g(n)$ é $O(n^3)$

- **Pergunta: $g(n)$ é $O(n^{40})$?**

existe c tal que
$g(n) \leq c\, f(n)$
para $n \geq m$

Exemplo de Notação O

- Exemplo: $g(n) = \log_5 n$ é $O(\log n)$
  - O $\log_a n$ difere do $\log_b n$ por uma constante $c = \log_a b$
  - Como $n = b^{\log_b n}$, tomando o logaritmo base a em ambos os lados da igualdade, temos que

$$\log_a n = \log_a b^{\log_b n}$$

$$\log_a n = \log_a b \times \log_b n$$

Exemplo de Notação O

- Exemplo: $g(n) = \log_5 n$ é $O(\log n)$
  - O $\log_a n$ difere do $\log_b n$ por uma constante $c = \log_a b$
  - Como $n = b^{\log_b n}$, tomando o logaritmo base a em ambos os lados da igualdade, temos que

$$\log_a n = \log_a b^{\log_b n}$$

$$\log_a n = \log_a b \times \log_b n$$

existe c tal que
$g(n) \leq c\, f(n)$
para $n \geq m$

Qual a Complexidade Assintótica?

- Seja *f(n)* o número de comparações entre os elementos de A. Logo, *f(n) = 2(n - 1)* para o melhor caso, pior caso e caso médio.

```
void MaxMin1(int* A, int n, int* pMax, int* pMin)
{
   int i;

   *pMax = A[0];
   *pMin = A[0];
   for (i = 1; i < n; i++) {
      if (A[i] > *pMax) *pMax = A[i];
      if (A[i] < *pMin) *pMin = A[i];
   }
}
```

Qual a Complexidade Assintótica?

- Seja *f(n)* o número de comparações entre os elementos de A. Logo, *f(n) = 2(n - 1)* para o melhor caso, pior caso e caso médio.

```
void MaxMin1(int* A, int n, int* pMax, int* pMin)
{
    int i;

    *pMax = A[0];
    *pMin = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax) *pMax = A[i];
        if (A[i] < *pMin) *pMin = A[i];
    }
}
```

O(n)

Exemplo: Algoritmos MaxMin

Os três algoritmos	$f(n)$		
	Melhor caso	Pior caso	Caso médio
MaxMin1	$2(n-1)$	$2(n-1)$	$2(n-1)$
MaxMin2	$n-1$	$2(n-1)$	$3n/2-3/2$
MaxMin3	$3n/2-2$	$3n/2-2$	$3n/2-2$

- Todos os algoritmos têm a mesma complexidade assintótica, ou seja, são *O(n)*

Operações com a Notação O

$$
\begin{aligned}
f(n) &= O(f(n)) \\
c \times O(f(n)) &= O(f(n)) \quad c = constante \\
O(f(n)) + O(f(n)) &= O(f(n)) \\
O(O(f(n)) &= O(f(n)) \\
O(f(n)) + O(g(n)) &= O(max(f(n), g(n))) \\
O(f(n))O(g(n)) &= O(f(n)g(n)) \\
f(n)O(g(n)) &= O(f(n)g(n))
\end{aligned}
$$

Operações com a Notação O

- Exemplo: regra da soma $O(f(n)) + O(g(n))$
  - Suponha três trechos cujos tempos de execução são $O(n)$, $O(n^2)$ e $O(n \log n)$
  - O tempo de execução dos dois primeiros trechos é $O(max(n, n^2))$, que é $O(n^2)$
  - O tempo de execução de todos os três trechos é então $O(max(n^2, n \log n))$, que é $O(n^2)$

Comparação de Algoritmos

- Se *f* é uma função de complexidade para um algoritmo *F*, então *O(f)* é considerada a complexidade assintótica ou o comportamento assintótico do algoritmo *F*
- A relação de dominação assintótica permite comparar funções de complexidade
  - Entretanto, se as funções *f* e *g* dominam assintoticamente uma a outra, então os algoritmos associados são equivalentes
  - Nesses casos, o comportamento assintótico não serve para comparar os algoritmos

Comparação de Algoritmos

- Exemplo:
  - Considere dois algoritmos *F* e *G*, em que *F* leva três vezes o tempo de *G*, isto é, $f(n) = 3\,g(n)$, portanto, $O(f(n)) = O(g(n))$
  - Logo, o comportamento assintótico não serve para comparar os algoritmos *F* e *G*, porque eles diferem apenas por uma constante
  - Nesse caso, podemos avaliar algoritmos comparando as funções de complexidade, negligenciando as constantes de proporcionalidade

Comparação de Algoritmos

- Um algoritmo com tempo de execução $O(n)$ é melhor que outro com tempo $O(n \log_{10} n)$
- Porém, as constantes de proporcionalidade podem alterar esta consideração
- Exemplo:
  - Um algoritmo leva $100n$ unidades de tempo para ser executado e outro leva $n \log_{10} n$. Qual dos dois é melhor?
  - **Depende do tamanho do problema**
    - O algoritmo com tempo $100n$ é melhor do que o que possui tempo $n \log_{10} n$ para $n \geq 10^{100}$
    - $10^{100} \approx$ número de átomos do universo observável

Classes de Problemas

- ***f(n) = O(1)***
  - Algoritmos de complexidade *O(1)* são ditos de **complexidade constante**
  - Uso do algoritmo independe de *n*
  - As instruções do algoritmo são executadas um número fixo de vezes

Classes de Problemas

- ***f(n) = O(log n)***
  - Um algoritmo de complexidade *O(log n)* é dito ter **complexidade logarítmica**
  - Típico em algoritmos que transformam um problema em outros menores
  - Pode-se considerar o tempo de execução como menor que uma constante grande
  - Quando *n* é mil, $log_2 n = 10$, quando *n* é 1 milhão, $log_2 n = 20$
    - Para dobrar o valor de *log n* temos de considerar o quadrado de *n*
  - A base do logaritmo muda pouco estes valores
    - quando *n* é 1 milhão, o $log_2 n$ é 20 e o $log_{10} n$ é 6

Classes de Problemas

- ***f(n) = O(n)***
  - Um algoritmo de complexidade *O(n)* é dito ter **complexidade linear**
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada
  - É a melhor situação possível para um algoritmo que tem de processar/produzir *n* elementos de entrada/saída
  - Cada vez que *n* dobra de tamanho, o tempo de execução dobra

Classes de Problemas

- ***f(n) = O(n log n)***
  - Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e combina as soluções depois
  - Quando *n* é 1 milhão, $n \log_2 n$ é cerca de 20 milhões
  - Quando *n* é 2 milhões, $n \log_2 n$ é cerca de 42 milhões, pouco mais do que o dobro

Classes de Problemas

- $f(n) = O(n^2)$
  - Um algoritmo de complexidade $O(n^2)$ é dito ter **complexidade quadrática**
  - Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um laço dentro de outro
  - Quando *n* é mil, o número de operações é da ordem de 1 milhão
  - Sempre que *n* dobra, o tempo de execução é multiplicado por 4
  - Úteis para resolver problemas de tamanhos relativamente pequenos

Classes de Problemas

- $f(n) = O(n^3)$
  - Um algoritmo de complexidade $O(n^3)$ é dito ter **complexidade cúbica**
  - Úteis apenas para resolver pequenos problemas
  - Quando n é 100, o número de operações é da ordem de 1 milhão
  - Sempre que n dobra, o tempo de execução fica multiplicado por 8

Classes de Problemas

- $f(n) = O(2^n)$
  - Um algoritmo de complexidade $O(2^n)$ é dito ter **complexidade exponencial**
  - Geralmente não são úteis sob o ponto de vista prático
  - Ocorrem na solução de problemas quando se usa força bruta para resolvê-los
  - Quando *n* é 20, o tempo de execução é cerca de 1 milhão
  - Quando *n* dobra, o tempo fica elevado ao quadrado

Classes de Problemas

- ***f(n) = O(n!)***
  - Um algoritmo de complexidade *O(n!)* é dito ter complexidade exponencial, apesar de *O(n!)* ter comportamento muito pior do que $O(2^n)$
  - Geralmente ocorrem quando se usa força bruta para na solução do problema
    - $n = 20 \rightarrow 20! = 2432902008176640000$, um número com 19 dígitos
    - n = 40 → um número com 48 dígitos

Comparação de Funções de Custo

Função de custo	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	0,059 s	58 min	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

Comparação de Funções de Custo

Função de custo de tempo	Computador atual	Computador 100 vezes mais rápido	Computador 1.000 vezes mais rápido
n	t_1	$100\ t_1$	$1000\ t_1$
n^2	t_2	$10\ t_2$	$31,6\ t_2$
n^3	t_3	$4,6\ t_3$	$10\ t_3$
2^n	t_4	$t_4 + 6,6$	$t_4 + 10$

Técnicas de Análise de Algoritmos

- Determinar o tempo de execução de um programa pode ser um problema matemático complexo
- Determinar a ordem do tempo de execução, sem preocupação com o valor da constante envolvida, pode ser uma tarefa mais simples
- A análise utiliza técnicas de matemática discreta, envolvendo contagem ou enumeração dos elementos de um conjunto

Análise do Tempo de Execução

- Comando de atribuição, de leitura ou de escrita: *O(1)*
- Sequência de comandos: determinado pelo maior tempo de execução de qualquer comando da sequência
- Comando de decisão: tempo dos comandos dentro do comando condicional, mais tempo para avaliar a condição, que é *O(1)*
- Laço: soma do tempo de execução do corpo do laço mais o tempo de avaliar a condição para terminação, multiplicado pelo número de iterações

Análise do Tempo de Execução

- Procedimentos não recursivos:
  - cada um deve ser computado separadamente um a um
  - inicie com os que não chamam outros procedimentos
  - avalie então os que chamam os já avaliados
  - o processo é repetido até chegar no programa principal

Exercício

```
void exercicio1(int n)
{
  int i, a;
  a=0; i=0;
  while (i<n)
  {
    a+=i;
    i+=2;
  }
}
```

```
void exercicio2(int n)
{
  int i, j, a;
  a=0;
  for (i=0; i<n; i++)
    for (j=0; j<i; j++)
      a+=i+j;
}
```

Exercício

```
void exercicio1(int n)
{
   int i, a;
   a=0; i=0;
   while (i<n)
   {
      a+=i;
      i+=2;
   }
}
```

O(n)

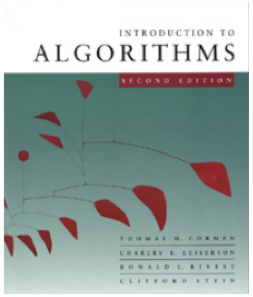
$$\left\lceil \frac{n}{2} \right\rceil$$

```
void exercicio2(int n)
{
   int i, j, a;
   a=0;
   for (i=0; i<n; i++)
      for (j=0; j<i; j++)
         a+=i+j;
}
```

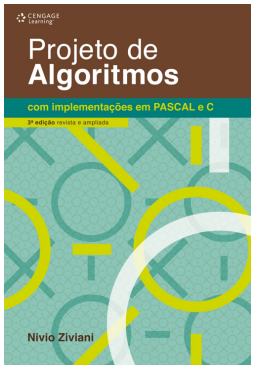
O(n²)

$$\sum_{i=0}^{n-1} i = 0+1+\ldots+n-1 = \frac{n(n-1)}{2}$$

Leitura Recomendada



CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. 3ª Edição. MIT Press, 2009. **Capítulo 3**



ZIVIANI, N. **Projeto de Algoritmos com Implementações em Pascal e C**. 3ª Edição. Cengage Learning, 2010. **Seções 1.3 e 1.4**