PROJETO E ANÁLISE DE ALGORITMOS

PARTE I

COMPLEXIDADE DE ALGORITMOS

1. Critérios de qualidade de um algoritmo

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

- Um algoritmo é **correto** se, para cada instância de entrada, ele pára com a saída correta

- Um algoritmo **incorreto** pode não parar em algumas instâncias de entrada, ou então pode parar com outra resposta que não a desejada

- Algoritmos **eficientes** são os que executam em tempo polinomial

- Algoritmos que necessitam de tempo superpolinomial são chamados de **ineficientes**

- Problemas que podem ser resolvidos por algoritmo de tempo polinomial são chamados de **tratáveis**

- Problemas que exigem tempo superpolinomial são chamados de **intratáveis**

- Um problema é **decidível** se existe algoritmo para resolvê-lo

- Um problema é **indecidível** se não existe algoritmo para resolvê-lo

- Dentro do possível, um algoritmo de qualidade é aquele que é correto e eficiente no consumo de recursos computacionais, notadamente tempo de processamento e memória

a) Análise de algoritmos

- Analisar a complexidade computacional de um algoritmo significa prever os recursos de que o mesmo necessitará:
  - Memória
  - Largura de banda de comunicação
  - Hardware
  - Tempo de execução

- Geralmente existe mais de um algoritmo para resolver um problema

- A análise de complexidade computacional é fundamental no processo de definição de algoritmos mais eficientes para a sua solução

- Em geral, o tempo de execução cresce com o tamanho da entrada

b) Porque estudar análise de algoritmos?

- O tempo de computação e o espaço na memória são recursos limitados
  - Os computadores podem ser rápidos, mas não são infinitamente rápidos
  - A memória pode ser de baixo custo, mas é finita e não é gratuita

- Os recursos devem ser usados de forma sensata, e algoritmos eficientes em termos de tempo e espaço devem ser projetados

- Com o aumento da velocidade dos computadores, torna-se cada vez mais importante desenvolver algoritmos mais eficientes, devido ao aumento constante do tamanho dos problemas a serem resolvidos

Suponha que para resolver um determinado problema você tem disponível um algoritmo exponencial (2^n) e um computador capaz de executar 10^4 operações por segundo

		2^n na máquina 10^4
	tempo (s)	**tamanho**
	0,10	10
	1	13
1 minuto	60	19
1 hora	3.600	25
1 dia	86.400	30
1 ano	31.536.000	38

Compra de um novo computador capaz de executar 10^9 operações por segundo

		2^n na máquina 10^4	2^n na máquina 10^9
	tempo (s)	**tamanho**	**tamanho**
	0,10	10	27
	1	13	30
1 minuto	60	19	36
1 hora	3.600	25	42
1 dia	86.400	30	46
1 ano	31.536.000	38	55

Aumento na velocidade computacional tem pouco efeito no tamanho das instâncias resolvidas por algoritmos ineficientes

Investir em algoritmo:

- Você encontrou um algoritmo quadrático (n^2) para resolver o problema

		2^n na máquina 10^4	2^n na máquina 10^9	n^2 na máquina 10^4	n^2 na máquina 10^9
	tempo (s)	**tamanho**	**tamanho**	**tamanho**	**tamanho**
	0,10	10	27	32	10.000
	1	13	30	100	31.623
1 minuto	60	19	36	775	244.949
1 hora	3.600	25	42	6.000	1.897.367
1 dia	86.400	30	46	29.394	9.295.160
1 ano	31.536.000	38	55	561.569	177.583.783

Novo algoritmo oferece uma melhoria maior que a compra da nova máquina

- Algum dia você poderá encontrar um problema para o qual não seja possível descobrir prontamente um algoritmo publicado

- É necessário estudar técnicas de projeto de algoritmos, de forma que você possa desenvolver algoritmos por conta própria, mostrar que eles fornecem a resposta correta e entender sua eficiência

2. Tempo de execução de um programa

a) Medida do tempo de execução

- O tempo de execução de um programa pode ser calculado da forma:

  Tempo=(No de instruções/programa) x CPI médio x clock

  - O número de instruções executadas por um programa dependerá do algoritmo (programador), do compilador e do tamanho da entrada (volume de dados processados).
  - O CPI (ciclos por instrução) médio depende da arquitetura da máquina (conjunto de instruções), projeto do processador, sistema operacional, hierarquia de memória entre outros.
  - O clock da máquina depende do hardware e determina o tempo de cada ciclo do relógio que sincroniza o processamento. Uma máquina de 1 GHz gasta 1 ns por ciclo.

- Medidas de tempo de execução são bastante inadequadas e os resultados jamais devem ser generalizados:

  - 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.
  - Não podemos testar todas as entradas possíveis; algumas delas podem resultar em desempenho particularmente ruim.

- Apesar disso, há argumentos a favor de se obterem medidas reais de tempo:

  - 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;
  - quando o tamanho da entrada é pequeno;
  - assim, são considerados tanto os custos reais das operações como os custos não aparentes, tais como alocação de memória, indexação, carga, dentre outros.

b) Medida do custo por meio de um modelo matemático

- Usa um modelo matemático baseado em um computador idealizado.

- Devem ser especificados o conjunto de operações e seus custos de execução.

- É mais usual ignorar o custo de algumas das operações e considerar apenas as operações mais significativas (críticas).

- Uma operação crítica é aquela que domina o tempo de execução do algoritmo.

- Ex.: algoritmos de ordenação. Consideramos o número de comparações entre os elementos do conjunto a ser ordenado e ignoramos as operações aritméticas, de atribuição e manipulações de índices, caso existam.

3. Soluções de compromisso

- Depois que um problema é analisado e decisões de projeto são finalizadas, é necessário estudar as várias opções de algoritmos a serem utilizados, considerando os aspectos de tempo de execução e espaço ocupado.

- Tempo e espaço tende a se comportar de forma antagônica, de modo que devemos buscar uma solução de compromisso.

- Soluções de compromisso buscam o equilíbrio entre recursos ou entre recursos e a precisão ou qualidade dos resultados.

Exemplo:

A PUC tem *n* alunos, onde cada um possui um número de identificação de *k* dígitos. Dado o número de um aluno, queremos saber o seu nome (256 caracteres).

Modelo A: Arranjo com 10^k posições contendo 256 caracteres em cada entrada. A operação *RecuperaNome(número)* recupera o nome com um acesso, mas o espaço necessário é de 256 x 10^k
caracteres.

Modelo B: Arranjo (lista) de pares do tipo *[número,nome]*. A operação *RecuperaNome(número)* recupera o nome com *n/2* acessos, na média, mas o espaço necessário é de 256*n* caracteres.

- Qual o melhor modelo?
- Qual seria uma solução de compromisso?

4. Análise de algoritmos

a) Análise de um algoritmo particular

- Qual é o custo de usar um dado algoritmo para resolver um problema específico?

- Qual a ordem de grandeza do custo relacionado ao algoritmo,

  - De modo geral (sem conhecimento da entrada)
  - No melhor caso (entrada que faz o algoritmo consumir menor quantidade possível do recurso analisado)
  - No pior caso (entrada que faz o algoritmo consumir maior quantidade possível do recurso analisado)
  - No caso médio (caso “esperado”, considerando probabilidades de cada entrada)

- Características que devem ser investigadas:

  - análise do número de vezes que cada parte do algoritmo deve ser executada: número de comparações, operações aritméticas, interrupções, acesso à memória, etc.;
  - estudo da quantidade de memória necessária;
  - número de acessos a memórias auxiliares;
  - acesso remoto;

De modo geral: analisa-se o recurso crítico que o algoritmo solicita.

O projeto de algoritmos deve considerar soluções de compromisso no uso dos recursos.

- Na procura de um algoritmo que resolva um determinado problema, interessa em geral encontrar um que seja eficiente. Há, no entanto, problemas para os quais não se conhece uma solução eficiente. Esta classe de problemas denomina-se por NP.

- Os problemas NP-difíceis, uma subclasse dos anteriores, são especialmente interessantes porque:

  - aparentemente são simples
  - não se sabe se existe um algoritmo eficiente que os resolva
  - aplicam-se a áreas muito importantes
  - se um deles for resolvível de forma eficiente, todos os outros o serão

- por vezes, ao resolver um problema NP-difícil, contentamo-nos em encontrar uma solução que aproxime a solução ideal, em tempo útil.

b) 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.
  - Colocam-se **limites** para a complexidade computacional dos algoritmos pertencentes à classe.

- Determinando o menor custo possível para resolver problemas de uma dada classe, temos a medida da dificuldade inerente para resolver o problema.

- Quando o custo de um algoritmo é igual ao menor custo possível para a classe, o algoritmo é **ótimo** para a medida de custo considerada.

- Podem existir vários algoritmos 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.

5. Funções de custo

- Para medir o custo de execução de um algoritmo é comum definir uma **função de custo** ou **função de complexidade** *f*.

- Função de **complexidade de tempo**: *f(n)* mede o tempo necessário para executar um algoritmo em um problema de tamanho *n*. A função na realidade não representa tempo diretamente, mas o número de vezes que determinadas operações consideradas relevantes são executadas.

- Função de **complexidade de espaço**: *f(n)* mede a memória necessária para executar um algoritmo em um problema de tamanho *n*.

- *f* pode ser uma função de mais de uma variável. Ex: ordenação em memória secundária considera o tamanho do arquivo e o número de unidades de disco/fita.

a) Melhor caso, pior caso e caso médio

- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.

- No caso de uma função que determina o maior valor de um conjunto, 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 o algoritmo pode ter que trabalhar menos.

- **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*. Se *f* é uma função de complexidade baseada na análise de pior caso, o custo de aplicar o algoritmo nunca é maior do que *f(n)*.

- **Caso médio** (ou caso esperado): média dos tempos de execução de todas as entradas de tamanho *n*, **ponderada** pela probabilidade de ocorrerem.

- Na análise do caso esperado, supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas e o custo médio é obtido com base nessa distribuição.

- A análise do caso médio é geralmente muito mais difícil de obter do que as análises do melhor e do pior caso.

- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis. Na prática isso nem sempre é verdade.

Exemplo – Pesquisa sequencial

- Considere o algoritmo para encontrar um elemento *K* em um vetor de *n* inteiros v*[a..b];*

```
                                                    Custo     Vezes
1     int procura(int *v, int a, int b, int k) {
2     int i;
3     i=a;                                          c1        1
4     while ((i<=b) && (v[i]!=k))                   c2        m+1
5           i++;                                    c3        m
6     if (i>b)                                      c4        1
7           return -1;                              c5        1
8     else return i; }                              c5        1
```

- *m* é o número de vezes que a instrução na linha 5 é executada. Este valor dependerá de quantas vezes a condição do ciclo é satisfeita: 0 <= m <= b-a+1.

- Tempo Total de Execução: $T(n)= c_1 + c_2(m+1) + c_3m + c_4 + c_5$

- Para determinado tamanho fixo $n = b-a+1$ da sequência a pesquisar (entrada), o tempo total pode variar com o conteúdo.

- Seja *f* uma função de complexidade tal que *f(n)* é o número de vezes que a consulta é comparada com cada elemento.

melhor caso:

- elemento procurado é o primeiro consultado e o tempo é constante;
- $T(n) = c_1 + c_2 + c_4 + c_5$
- $f(n) = 1$

pior caso:

- elemento procurado é o último consultado ou não está presente no vetor;
- $T(n) = c_1 + c_2(n+1) + c_3 n + c_4 + c_5$
  $= (c_2 + c_3)\, n + (c_1 + c_2 + c_4 + c_5)$
- $f(n) = n$
- logo $T(n)$ e $f(n)$ são funções lineares de n.

caso médio:

- No estudo do caso médio, vamos considerar que toda pesquisa recupera um elemento.
- Se p_i for a probabilidade de que o i-ésimo elemento seja procurado, e considerando que para recuperar o i-ésimo elemento são necessárias i comparações, então

  $f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \ldots + n \times p_n.$

- Para calcular $f(n)$ basta conhecer a distribuição de probabilidades p_i. Se cada elemento tiver a mesma probabilidade de ser acessado que todos os outros, então $p_i = 1/n$. Neste caso:

  $f(n) = (1/n)(1+2+3+\ldots+n) = (1/n)(n(n+1)/2) = (n+1)/2.$

- A análise do caso esperado revela que uma pesquisa com sucesso examina aproximadamente metade dos elementos.

Exemplo - Maior Elemento

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros *A[n]; n >= 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 entre os elementos de A, se A contiver *n* elementos.

- Logo *f(n) = n -1* para *n > 0*.

- O algoritmo apresentado no programa acima é **ótimo**, pois para encontrar o maior elemento de um conjunto com *n* elementos, cada um dos *n -1* elementos tem de ser mostrado, por meio de comparações, que é menor do que algum outro elemento.

Exemplo - Maior e Menor Elemento (1)

- Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros *A[1..n], n>=1*.
- Um algoritmo simples pode ser derivado do algoritmo apresentado no programa para achar o maior elemento.

```
void MaxMin1 (int * A, int & max, int & min) {
   int i;
   max = min = A[0];
   for (i = 1; i< n; i++) {
      if ( A[ i ] > max ) max = A[ i ] ;
      if ( A[ i ] < min )  min = A[ i ] ;
  }
}
```

- Seja *f(n)* o número de comparações entre os elementos de A, logo *f(n) = 2(n - 1)* para *n* > 0, para o melhor caso, pior caso e caso médio.

Exemplo - Maior e Menor Elemento (2)

- MaxMin1 pode ser facilmente melhorado: a comparação A[i] < min só é necessária quando a comparação A[i] > max dá falso.

```
void MaxMin2 (int * A, int & max, int & min) {
   int i;
   max = min = A[0];
   for (i = 1; i< n; i++) {
      if ( A[ i ] > max ) max = A[ i ];
     else if ( A[ i ] < min )  min = A[ i ] ;
  }
}
```

- Para a nova implementação temos:
  - melhor caso: *f(n) = n - 1* (quando os elementos estão em ordem crescente);
  - pior caso: *f(n) = 2(n - 1)* (quando os elementos estão em ordem decrescente);
  - caso médio: A[i] é maior do que max a metade das vezes. Logo *f(n) = n - 1 + (n – 1)/2 = 3n/2 – 3/2*.

Exemplo - Maior e Menor Elemento (3)

- Considerando o número de comparações realizadas, existe a possibilidade de obter um algoritmo mais eficiente:

1. 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.

2. O máximo é obtido do subconjunto que contém os maiores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.

3. O mínimo é obtido do subconjunto que contém os menores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.

```
void MaxMin3 (int * A, int & max, int & min) {
    int i, fimAnel;
    if (n % 2 > 0) {
        A [ n ] = A [ n - 1 ];
        fimAnel = n - 1;
   } else fimAnel = n - 2;
   if ( A[0] > A[1] ) {   max = A[0]; min = A[1];   }
   else { max = A[1]; min = A[0]; }
   i = 2;
  while ( i <= fimAnel ) {
        if ( A[ i ] > A[ i +1] ) {
                if ( A[ i ] > max ) max = A[ i ] ;
                if ( A[ i +1] < min ) min = A[ i +1];
        } else {
                if ( A[ i ] < min ) min = A[ i ] ;
                if ( A[ i +1] > max ) max = A[ i +1];
        }
        i += 2;
  }}
```

- Os elementos de A são comparados dois a dois e os elementos maiores são comparados com *max* e os elementos menores são comparados com *min*.

- Quando *n* é ímpar, o elemento que está na posição *A[n-1]* é duplicado na posição *A[n]* para evitar um tratamento de exceção.

- Para esta implementação,

$f(n) = n/2 + (n-2)/2 + (n-2)/2 = 3n/2 – 2,$

para o melhor caso, pior caso e caso médio.

Exercício

Apresente a função de tempo para o algoritmo abaixo:

```
ALG1()
for i ← 1 to 2 do
   for j ← i to n do
      for k ← i to j do
         temp ← temp + i + j + k
```

6. Comportamento assintótico de funções

- O parâmetro *n* fornece uma medida da dificuldade para se resolver o problema.

- Para valores suficientemente pequenos de *n*, qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes.

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*.

- Estuda-se o comportamento **assintótico** das funções de custo (comportamento de suas funções de custo para valores grandes de *n*)

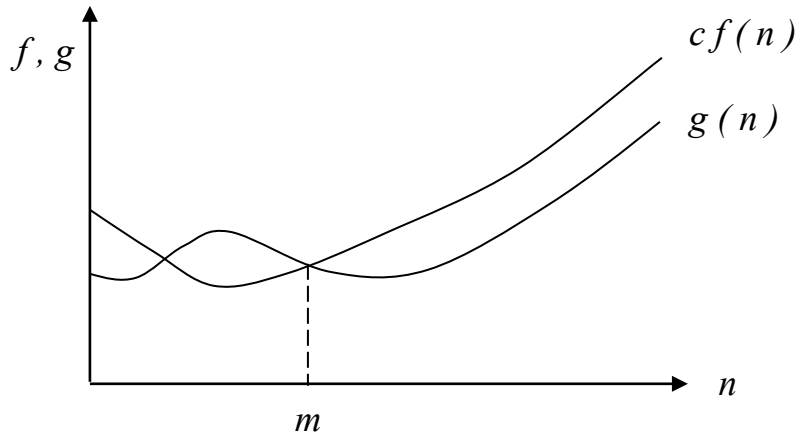
- O comportamento assintótico de *f(n)* representa o limite do comportamento do custo quando *n* cresce.

Dominação assintótica

- A análise de um algoritmo geralmente conta com apenas 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** outra função *g(n)* se existem duas constantes positivas *c* e *m* tais que, para *n* >= *m*, temos |*g(n)*| <= c |*f(n)*|.



Exemplo:

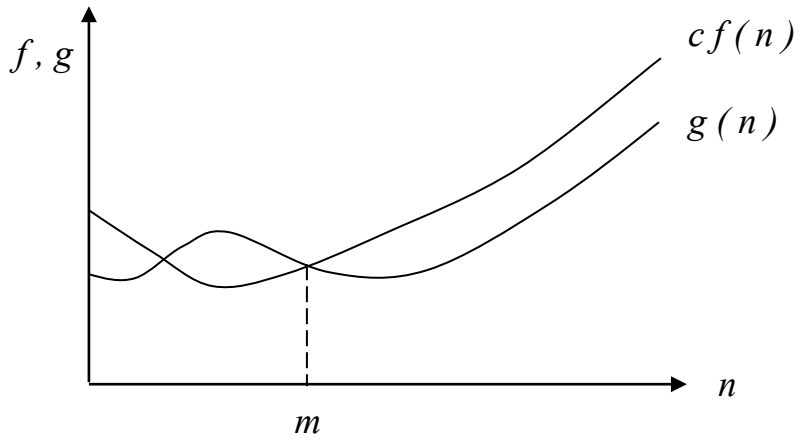
Sejam $g(n) = (n + 1)^2$ e $f(n) = n^2$. As funções *g(n)* e *f(n)* dominam assintoticamente uma a outra, já que $|(n + 1)^2| <= 4 |n^2|$, para $n >= 1$ e $|n^2| <= |(n + 1)^2|$, para $n >= 0$.

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)*" ou simplesmente "*g(n)* é O de *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 $n >= m$, $T(n) <= cn^2$.

- Exemplo gráfico de dominação assintótica que ilustra a 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 $g(n) <= cf(n)$, para todo n>=m.

Exemplos de Notação O

- **Exemplo**: $g(n) = (n + 1)^2$. Logo $g(n)$ é $O(n^2)$, quando m=1 e c=4. Isto porque $(n + 1)^2 <= 4n^2$ para $n>=1$.

- **Exemplo**: $g(n) = n$ e $f(n) = n^2$. Sabemos que $g(n)$ é $O(n^2)$, pois para $n >= 0$, $n <= n^2$.

  Entretanto $f(n)$ não é $O(n)$. Suponha que existam constantes c e m tais que para todo $n >= m$, $n^2 <= cn$. Neste caso, $c >= n$ para qualquer $n >= m$, e não existe uma constante c que possa ser maior ou igual a n para todo n.

- **Exemplo**: $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$. Basta mostrar que $3n^3 + 2n^2 + n <= 6n^3$, para n >= 0.

  A função g(n) = $g(n) = 3n^3 + 2n^2 + n$ é também $O(n^4)$, entretanto esta afirmação é mais fraca do que dizer que $g(n)$ é $O(n^3)$ (limite pouco restrito). Utilizaremos sempre o limite mais restrito.

- **Exemplo**: $g(n) = \log_5 n$ é $O(\log n)$.

  O $\log_b n$ difere do $\log_c n$ por uma constante que no caso é $\log_b c$. Desta forma, $\log_b n = \log_b c \times \log_c n$.

Operações com a Notação O

- $f(n) = O(f(n))$
- $c\ O(f(n)) = O(f(n))$, c = constante
- $O(f(n)) + O(f(n)) = O(f(n))$
- $O(O(f(n)) = O(f(n))$
- $O(f(n)) + O(g(n)) = O(max(f(n); g(n)))$

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

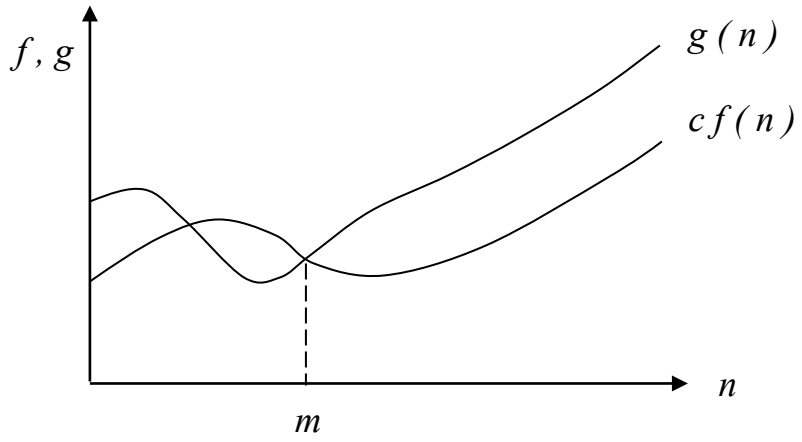
Exemplo: regra da soma $O(f(n)) + O(g(n))$.

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 $O(n^2)$.

Notação Ω

- Especifica um limite inferior para g(n).

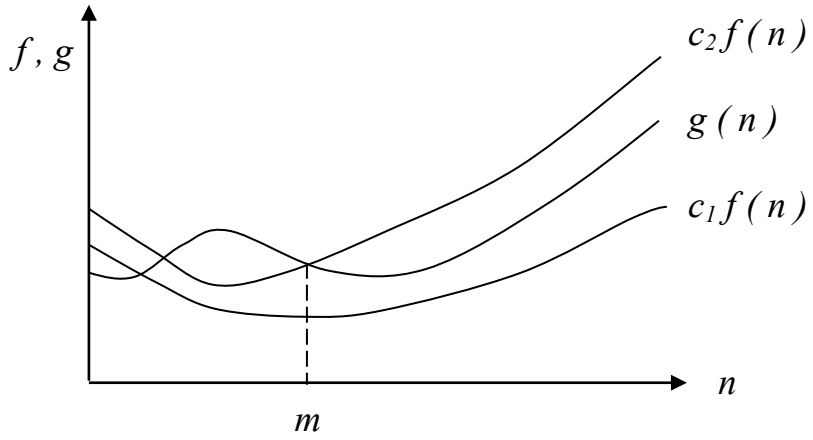
Definição: Uma função $g(n)$ é $\Omega(f(n))$ se existirem duas constantes c e m tais que $g(n) >= cf(n)$, para todo $n>=m$.



- **Exemplo**: Para mostrar que $g(n) = 3n^3 + 2n^2$ é $\Omega(n^3)$ basta fazer c = 1, e então $3n^3 + 2n^2 >= n^3$ para $n >= 0$.

Notação Θ

Definição: Uma função *g(n)* é Θ*(f(n))* se existirem constantes positivas c_1, c_2 e *m* tais que $0 <= c_1 f(n) <= g(n) <= c_2 f(n)$, para todo $n >= m$.



- Dizemos que *g(n)* = Θ*(f(n))* se existirem constantes c_1, c_2 e *m* tais que, para todo $n >= m$, o valor de *g(n)* está sobre ou acima de $c_1 f(n)$ e sobre ou abaixo de $c_2 f(n)$, isto é a função *g(n)* é igual a *f(n)* a menos de uma constante.

- Neste caso, *f(n)* é um **limite assintótico firme**.

- **Exemplo:** Seja $g(n) = n^2/3 - 2n$. Vamos mostrar que $g(n) = \Theta(n^2)$.
  - Temos de obter constantes c_1, c_2 e *m* tais que $c_1 n^2 <= n^2/3 - 2n <= c_2 n^2$ para todo $n >= m$.
  - Dividindo por n^2 leva a $c_1 <= 1/3 – 2/n <= c_2$.
  - O lado direito da desigualdade será sempre válido para qualquer valor de n >=1 quando escolhemos c_2 >1/3.
  - Escolhendo c_1 <= 1/21, o lado esquerdo da desigualdade será válido para qualquer valor de n>=7.

- Logo, escolhendo $c_1 = 1/21$, $c_2 = 1/3$ e m = 7, verifica-se que g(n) = $\Theta(n^2)$.
- Outras constantes podem existir, mas o importante é que existe alguma escolha para as três constantes.

Obs: as regras de operações da notação O também se aplicam à notação Θ

Propriedades

- Reflexividade:

  $f(n) = O(f(n))$.
  $f(n) = \Omega(f(n))$.
  $f(n) = \Theta(f(n))$.

- Simetria:

  $f(n) = \Theta(g(n))$ se, e somente se, $g(n) = \Theta(f(n))$.

- Simetria Transposta:

  $f(n) = O(g(n))$ se, e somente se, $g(n) = \Omega(f(n))$.

- Transitividade:

  Se $f(n) = O(g(n))$ *e* $g(n) = O(h(n))$, *então* $f(n) = O(h(n))$.
  Se $f(n) = \Omega(g(n))$ *e* $g(n) = \Omega(h(n))$, *então* $f(n) = \Omega(h(n))$.
  Se $f(n) = \Theta(g(n))$ *e* $g(n) = \Theta(h(n))$, *então* $f(n) = \Theta(h(n))$.

Relações úteis

- $\log_b n = O(\log_a n)$ $[a, b > 1]$
- $n^b = O(n^a)$ se $b <= a$
- $b^n = O(a^n)$ se $b <= a$
- $\log^b n = O(n^a)$
- $n^b = O(a^n)$ $[a > 1]$
- $n! = O(n^n)$
- $n! = \Omega(2^n)$
- $\lg(n!) = \Theta(n \lg n)$

Quais as notações mais indicadas para expressar a complexidade de casos específicos de um algoritmo, do algoritmo de modo geral e da classe de algoritmos para o problema?

Casos específicos:

- o ideal é a notação Θ, por ser um limite assintótico firme.
- A notação O também é aceitável e mais comum na literatura.
- Embora possa teoricamente ser usada, a notação Ω é mais fraca neste caso e deve ser evitada para casos específicos (reservá-la para indicar o limite inferior de uma classe de problemas).

Algoritmo de forma geral:

- Se o algoritmo comporta-se de forma idêntica para qualquer entrada, a notação Θ é a mais precisa (lembre-se que $f(n)=\Theta(g(n)) \Rightarrow f(n)=O(g(n))$).
- Se os casos melhor e pior são diferentes, a notação mais indicada é a O, já que estaremos interessados em um limite assintótico superior.

- O pior caso do algoritmo deve ser a base da análise.

Para uma classe de algoritmos:

- Neste caso estamos interessados no limite inferior para o problema e a notação deve ser a Ω.

7. Classes de Comportamento Assintótico

- 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. Nestes casos, o comportamento assintótico não serve para comparar os algoritmos.

- Por exemplo, considere dois algoritmos F e G aplicados à mesma classe de problemas, sendo que F leva três vezes o tempo de G ao serem executados, isto é, $f(n) = 3g(n)$, sendo que $O(f(n)) = O(g(n))$. Logo, o comportamento assintótico não serve para comparar os algoritmos F e G, porque eles diferem apenas por uma constante.

Comparação de Programas

- Podemos avaliar programas comparando as funções de complexidade, negligenciando as constantes de proporcionalidade.

- Um programa com tempo de execução $O(n)$ é melhor que outro com tempo $O(n^2)$.

- Porém, as constantes de proporcionalidade podem alterar esta consideração.

- Exemplo: um programa leva *100n* unidades de tempo para ser executado e outro leva $2n^2$. Qual dos dois programas é melhor?

    - depende do tamanho do problema.
    - Para n < 50, o programa com tempo $2n^2$ é melhor do que o que possui tempo *100n*.
    - Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é $O(n^2)$.
    - Entretanto, quando *n* cresce, o programa com tempo de execução $O(n^2)$ leva muito mais tempo que o programa *O(n)*.

Classes de comportamento

a) *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.

b) *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 1000, lg *n* ≈10, quando *n* é 1000000, lg *n* ≈ 20.
- Para dobrar o valor de lg *n* temos de considerar o quadrado de *n*.
- A base do logaritmo muda pouco estes valores: quando *n* é 1 milhão, o lg *n* é 20 e o log *n* é 6.

c) *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.

d) *f(n) = O(n log n)*

- Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e ajuntam as soluções depois.
- Quando *n* é 1 milhão, *n lg n* é cerca de 20 milhões.
- Quando n é 2 milhões, *n lg n* é cerca de 42 milhões, pouco mais do que o dobro.

e) *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 anel 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.

f) *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.

g) *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.

h) *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 → 20! = 2432902008176640000, um número com 19 dígitos.
- 40! é um número com 48 dígitos.

Comparação de Funções de Complexidade

Função de custo	Tamanho da entrada					
	10	20	30	40	50	60
n	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms	0.06 ms
n^2	0.1 ms	0.4 ms	0.9 ms	1.6 ms	2.5 ms	3.6 ms
n^3	1 ms	8 ms	27 ms	64 ms	125 ms	316 ms
n^5	0.1 s	3.2 s	24.3 s	1.7 min	5.2 min	13 min
2^n	1 ms	1 s	17.9 min	12.7 dias	35.7 anos	366 séculos

Algoritmos Polinomiais e Não-polinomiais

- **Algoritmo exponencial** no tempo de execução tem função de complexidade $O(c^n), c > 1$.

- **Algoritmo polinomial** no tempo de execução tem função de complexidade $O(p(n))$, onde $p(n)$ é um polinômio.

- A distinção entre estes dois tipos de algoritmos torna-se significativa quando o tamanho do problema a ser resolvido cresce.

- Por isso, os algoritmos polinomiais são muito mais úteis na prática do que os exponenciais.

- Algoritmos exponenciais são geralmente simples variações de pesquisa exaustiva.

- Algoritmos polinomiais são geralmente obtidos mediante entendimento mais profundo da estrutura do problema.

- Um problema é considerado:
  - intratável: se não existe um algoritmo polinomial para resolvê-lo.
  - bem resolvido: quando existe um algoritmo polinomial para resolvê-lo.

- A distinção entre algoritmos polinomiais eficientes e algoritmos exponenciais ineficientes possui várias exceções.

- Exemplo: um algoritmo com função de complexidade $f(n) = 2^n$ é mais rápido que um algoritmo $g(n) = n^5$ para valores de n menores ou iguais a 20.

- Também existem algoritmos exponenciais que são muito úteis na prática. Exemplo: o algoritmo Simplex para programação linear possui complexidade de tempo exponencial para o pior caso mas executa muito rápido na prática.

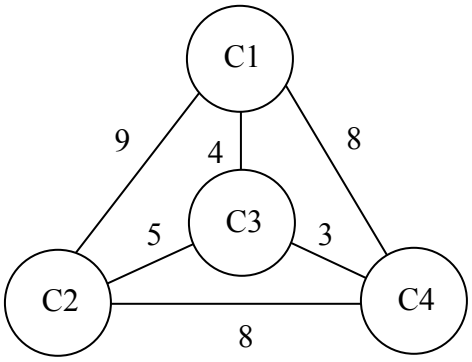
- Tais exemplos não ocorrem com frequência na prática, e muitos algoritmos exponenciais conhecidos não são muito úteis.

Exemplo de Algoritmo Não-polinomial

Um **caixeiro viajante** deseja visitar *n* cidades de tal forma que sua viagem inicie e termine em uma mesma cidade, e cada cidade deve ser visitada uma única vez.

- Supondo que sempre há uma estrada entre duas cidades quaisquer, o problema é encontrar a menor rota para a viagem.

- A figura ilustra o exemplo para quatro cidades c1, c2, c3, c4, em que os números nos arcos indicam a distância entre duas cidades.



- O percurso <c1, c3, c4, c2, c1> é uma solução para o problema, cujo percurso total tem distância 24.

- Um algoritmo simples seria verificar todas as rotas e escolher a menor delas.

- Há (n-1)! rotas possíveis e a distância total percorrida em cada rota envolve *n* adições, logo o número total de adições é n!.

- No exemplo anterior teríamos 24 adições.

- Suponha agora 50 cidades: o número de adições seria 50! ≈ 10^{64}. Em um computador que executa 10^{9} adições por segundo, o tempo total para resolver o problema com 50 cidades seria maior do que 10^{45} séculos só para executar as adições.

- O problema do caixeiro viajante aparece com frequência em problemas relacionados com transporte, mas também aplicações importantes relacionadas com otimização de caminho percorrido.