Projeto e Análise de Algoritmos

Análise de Algoritmos
Comportamento Assintótico
Notação Assintótica

Análise de algoritmos

- Eficiência do algoritmo
  - Análise da ordem de crescimento da função de complexidade: tempo ou espaço
  - Notação assintótica
- Corretude do algoritmo
  - Analisar se o algoritmo um resultado correto

Otimizando tour de robô

- Problema: otimizando tour de robô
- Entrada: Um conjunto S de n pontos em um plano
- Saída: Um ciclo que visita cada ponto de S e de menor comprimento possível.

Solução Nearest-neighbor

NearestNeighbor(P)

Pick and visit an initial point p_0 from P

$p = p_0$

$i = 0$

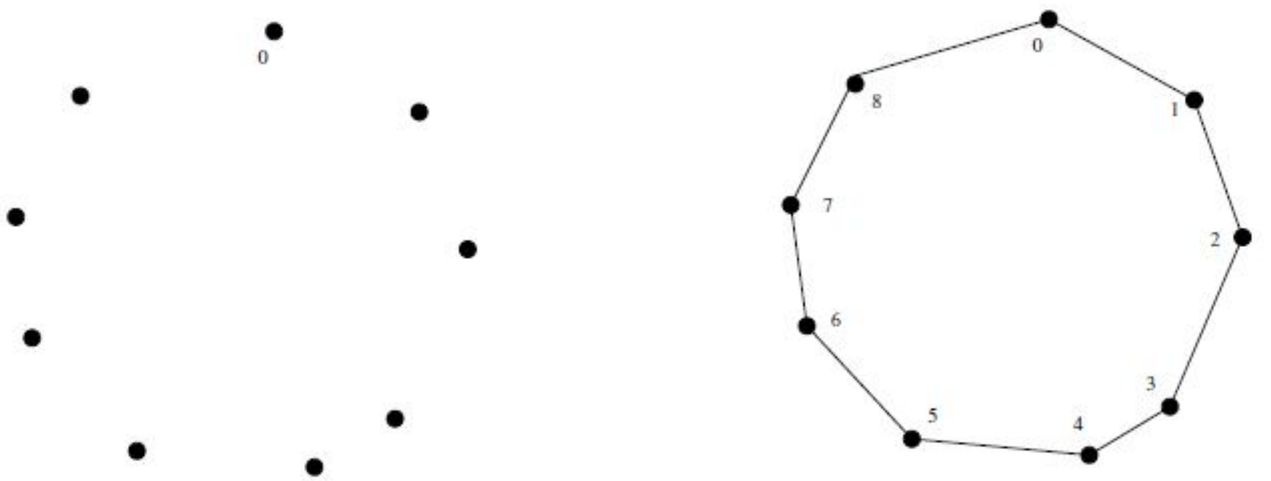
While there are still unvisited points

  $i = i + 1$

  Select p_i to be the closest unvisited point to p_{i-1}

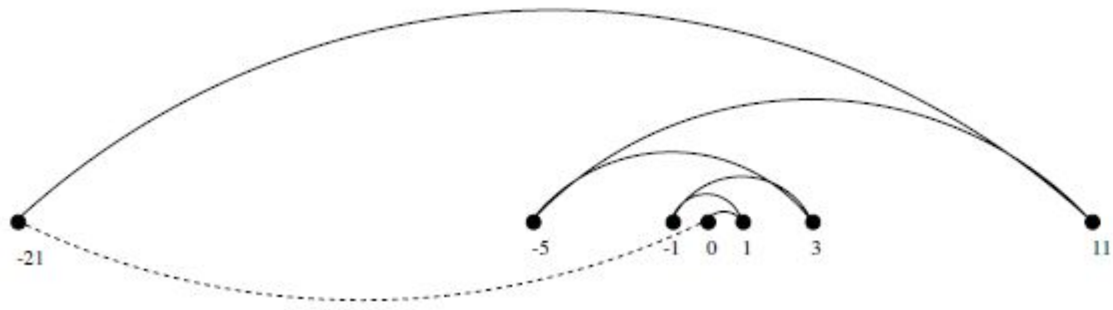
  Visit p_i

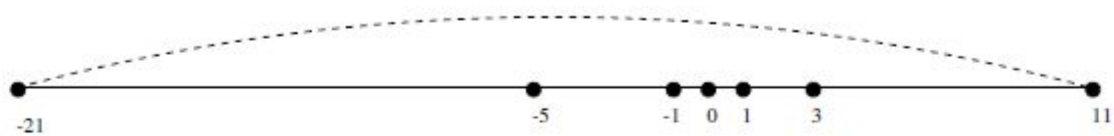
Return to p_0 from p_{n-1}



Solução ótima da solução por Nearest-neighbor

Outra entrada





Solução correta

OptimalTSP(P)

$d = \infty$

For each of the $n!$ permutations P_i of point set P

If $(cost(P_i) \leq d)$ then $d = cost(P_i)$ and $P_{min} = P_i$

Return P_{min}

Considera todas as possíveis permutações, escolhendo aquela de menor custo.
Portanto, o algoritmo é correto.
Porém, para n=20, 20! = 2.432.902.008.176.640.000

Desempenho de um programa

- Tempo de processamento
- Quantidade de memória requerida
- Fator importante a ser considerado em um projeto de algoritmo
  - Adequação dos algoritmos e estruturas de dados escolhidas para resolução de um dado problema

Tempo de execução

- Depende de quais fatores?
  - Hardware (CPU, memória, HD, SDD, etc)
  - Dados de entrada (quantidade, tamanho)
  - Implementação
    - Linguagem de programação
    - Algoritmos e estruturas de dados utilizados
    - Compilador/interpretador
    - Sistema Operacional
    - Tipo de operações realizadas
    - Tipo de memória utilizada
  - ...

Análise de algoritmos

- Analisar um algoritmo é prever os recursos de que o algoritmo necessitará
  - Eficiência
  - Viabilidade
- Analisar de forma que diferentes algoritmos possam ser comparados e que os mais **eficientes** possam ser identificados e escolhidos

Medida do custo pela execução do programa

- Desvantagem
  - Resultados dependem de várias variáveis:
    - Compilador
    - Hardware
- Vantagem
  - Comparar algoritmos com custos de execução da mesma ordem de grandeza
    - Pode-se comparar custos reais das operações

Medida do custo por meio de um modelo matemático

- Considera um computador idealizado
  - Padronização independente de máquina
    - Um processador e memória RAM (acesso aleatório)
  - Realidade: pode variar conforme tipo de instruções, etc
    - Ex: Cálculo de x^y é realizado em tempo constante?
      - Em C e x=2, 1 << y (left-shift)
    - Ex: E multiplicação de dois inteiros, x * y?

- Considera apenas o custo de operações mais significativas
  - Pode ignorar o custo de algumas operações

Funções de complexidade

- Função de complexidade $f(n)$
  - Complexidade de tempo: $f(n)$ mede o tempo necessário para executar um algoritmo em um problema de tamanho n
    - Geralmente calcula-se a quantidade de vezes que algumas operações relevantes são executadas
  - Complexidade de espaço: $f(n)$ mede a memória necessária para executar um algoritmo em problema de tamanho n

Exemplo

- Maior Elemento
  - Encontrar o maior elemento no vetor de inteiros $A[0...n-1]$, $n \geq 1$

```
int Max(TipoVetor A)
{ 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*, sendo *A* um vetor contendo n elementos
- Logo $f(n) = n - 1$, para $n > 0$.

Tamanho da entrada de dados

- Geralmente $f(n)$ é calculada em função do tamanho da entrada n (número de itens)
  - Na função Max(), o custo independe da entrada
  - Para muitos algoritmos, o custo de execução pode depender do tipo de dados de entrada
    - Custos diferentes para entradas de tamanhos iguais
      - Ex: Ordenação de números já ordenados e não-ordenados
    - Multiplicação de dois números inteiros
      - Depende da quantidade de bits necessária para representação
    - Grafos
      - Descrição em termos de números de vértices e arestas (2 variáveis)

Ex: Algoritmo de Seleção

```
Selecao(A){
  for i=1 to n-1 do
    min = i
    for j=i+1 to n do
      if(A[j] < A[min])
        min = j
    troca(A[min], A[i]);
}
```

Ex: Algoritmo de Seleção

Linha	# execuções
for i=1 to n-1 do	
for j=i+1 to n do	
if(A[j] < A[min])	

Ex: Algoritmo de Seleção

Linha	# execuções
for i=1 to n-1 do	n
for j=i+1 to n do	$\sum_{i=1}^{n-1}(n-i+1)$
if(A[j] < A[min])	$\sum_{i=1}^{n-1}(n-i)$

Seja T(n) uma função de complexidade que fornece o número de comparações (if) realizadas pelo Selection sort

$$T(n) = \sum_{i=1}^{n-1}(n-i)$$

$$\boxed{\sum_{i=1}^{n} i = n(n+1)/2}$$

Ex: Algoritmo de Seleção

Linha	# execuções
for i=1 to n-1 do	n
for j=i+1 to n do	$\sum_{i=1}^{n-1}(n-i+1)$
if(A[j] < A[min])	$\sum_{i=1}^{n-1}(n-i)$

Seja T(n) uma função de complexidade que fornece o número de comparações (if) realizadas pelo Selection sort

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1}(n-i) \\ &= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \\ &= n(n-1) - n(n-1)/2 \\ &= (n^2-n)/2 \end{aligned}$$

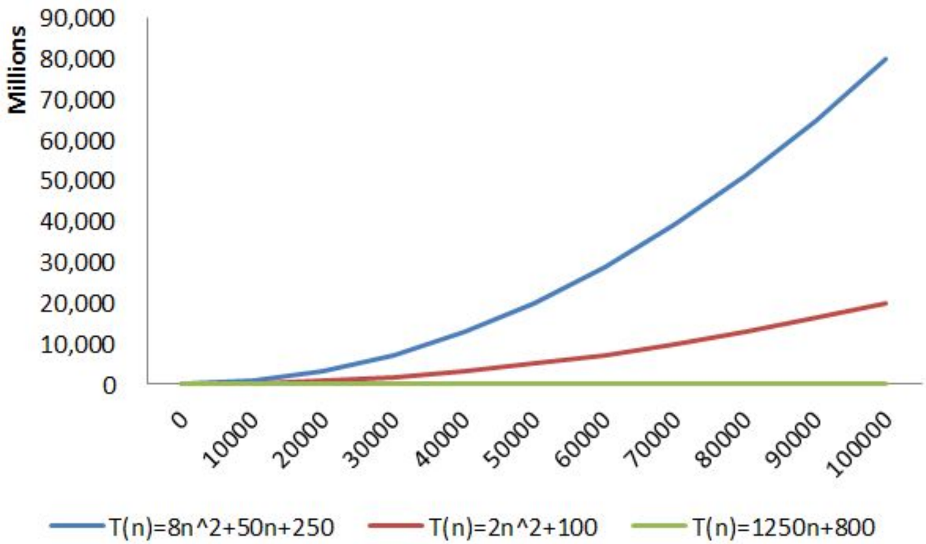
$$\boxed{\sum_{i=1}^{n} i = n(n+1)/2}$$

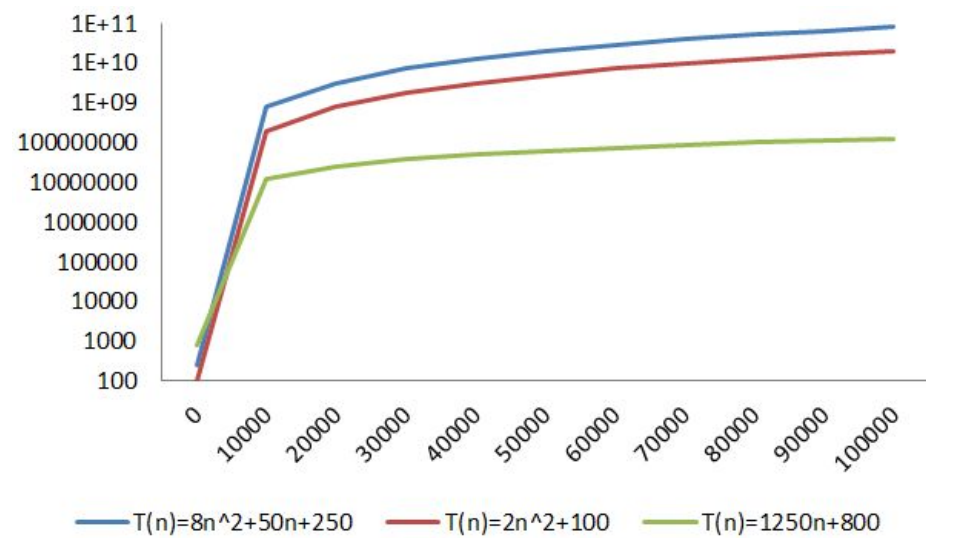
Pior caso x caso médio

- Geralmente estamos interessado em calcular a ordem do **tempo de execução do pior caso**
  - Limite superior para qualquer entrada
  - Pior caso pode ocorrer frequentemente
  - Muitas vezes o caso médio não é melhor que o pior caso
    - Ex: Insertion Sort para números em uma ordenação aleatória
      - $t_i = j/2$

Ordem de crescimento

- O que nos interessa é a **taxa de crescimento**, ou **ordem de crescimento** do tempo de execução
  - Para n suficientemente grande os termos de menor ordem são relativamente insignificantes
  - Como o pior caso do algoritmo Insertion Sort é uma função quadrática $T(n) = an^2 + bn + c$, logo podemos definir como uma função $\boldsymbol{\Theta(n^2)}$





- Comparação de várias funções de complexidade

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.

N. Ziviani, Projeto de Algoritmos com Implementações em Pascal e C, Pioneira Thomson Learning, 2ª. Edição, (2004).

- Influência do aumento da velocidade dos computadores no tamanho *t* do problema

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	$t_4 + 6,6$	$t_4 + 10$

N. Ziviani, Projeto de Algoritmos com Implementações em Pascal e C, Pioneira Thomson Learning, 2ª. Edição, (2004).

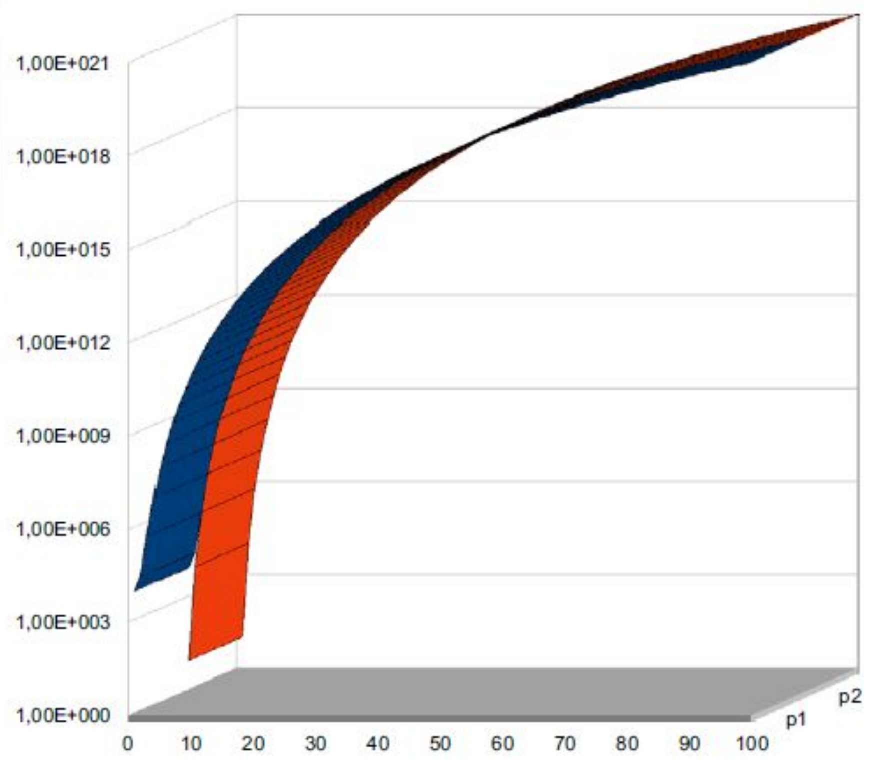
Comportamento Assintótico

- Crescimento de funções
  - Para entradas grandes o bastante
    - Constantes multiplicativas e termos de mais baixa ordem são dominados pelo termo de maior ordem
    - Análise dos termos mais relevantes para um n grande o suficiente: análise da eficiência assintótica

Comportamento assintótico de funções

- O que acontece quando $\boldsymbol{n}$ aumenta?
  - $T_1(n) = 10n^{10} + 100n^2 + 10000n + 1/n$
  - $T_2(n) = 10n^{10}$

- Para n suficientemente grande, somente o termo de mais alta ordem (n^{10}) se torna relevante



Exemplo

- Diferentes formas de se calcular:

$$\text{sum} = \sum_{i=1}^{n} i$$

Exemplo

- Diferentes formas de se calcular:

$$\text{sum} = \sum_{i=1}^{n} i$$

Algorithm A	Algorithm B	Algorithm C

sum = 0
for i = 1 *to* n
  sum = sum + i
	sum = 0
	for i = 1 *to* n
	{ **for** j = 1 *to* i
	    sum = sum + 1
	}
		sum = n * (n + 1) / 2

Exemplo

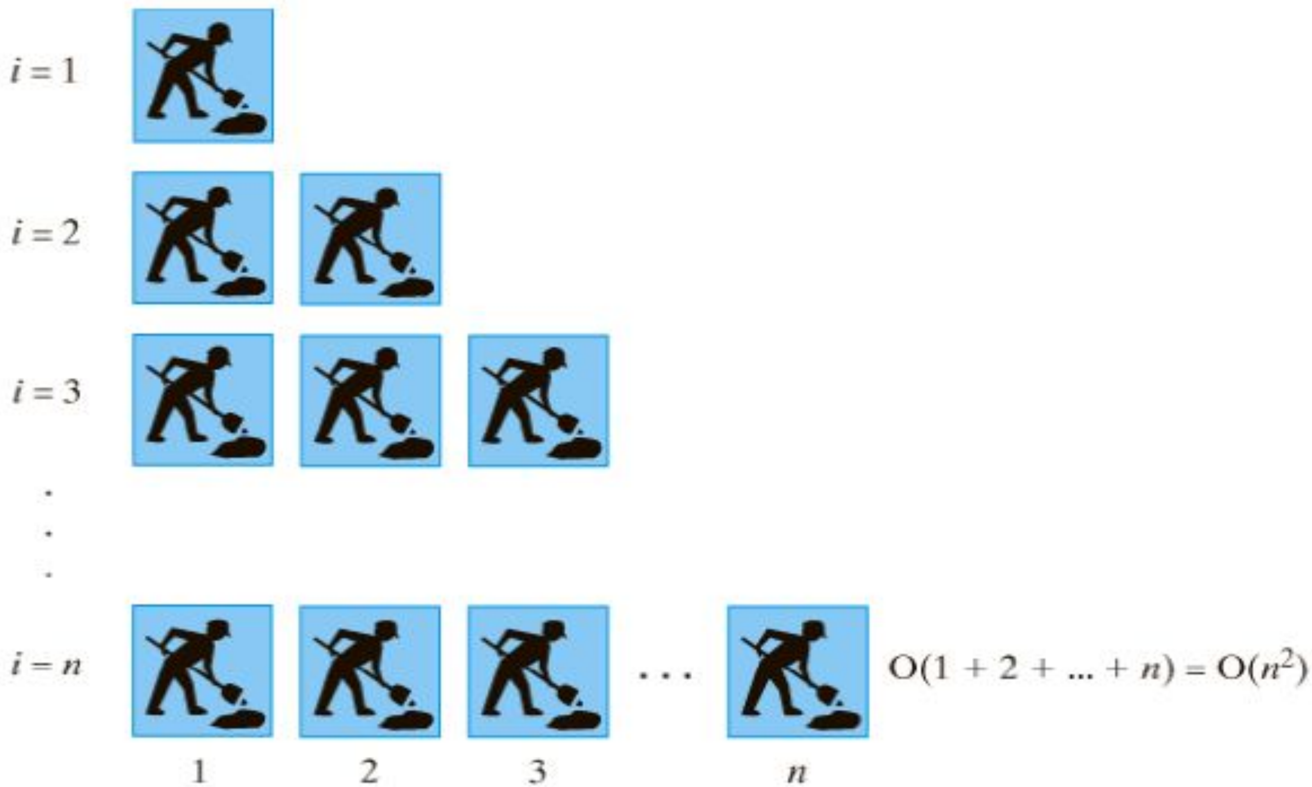
- Algoritmo A

```
for i = 1 to n
    sum = sum + i
```

Exemplo

- Algoritmo B

```
for i = 1 to n
{  for j = 1 to i
       sum = sum + 1
}
```



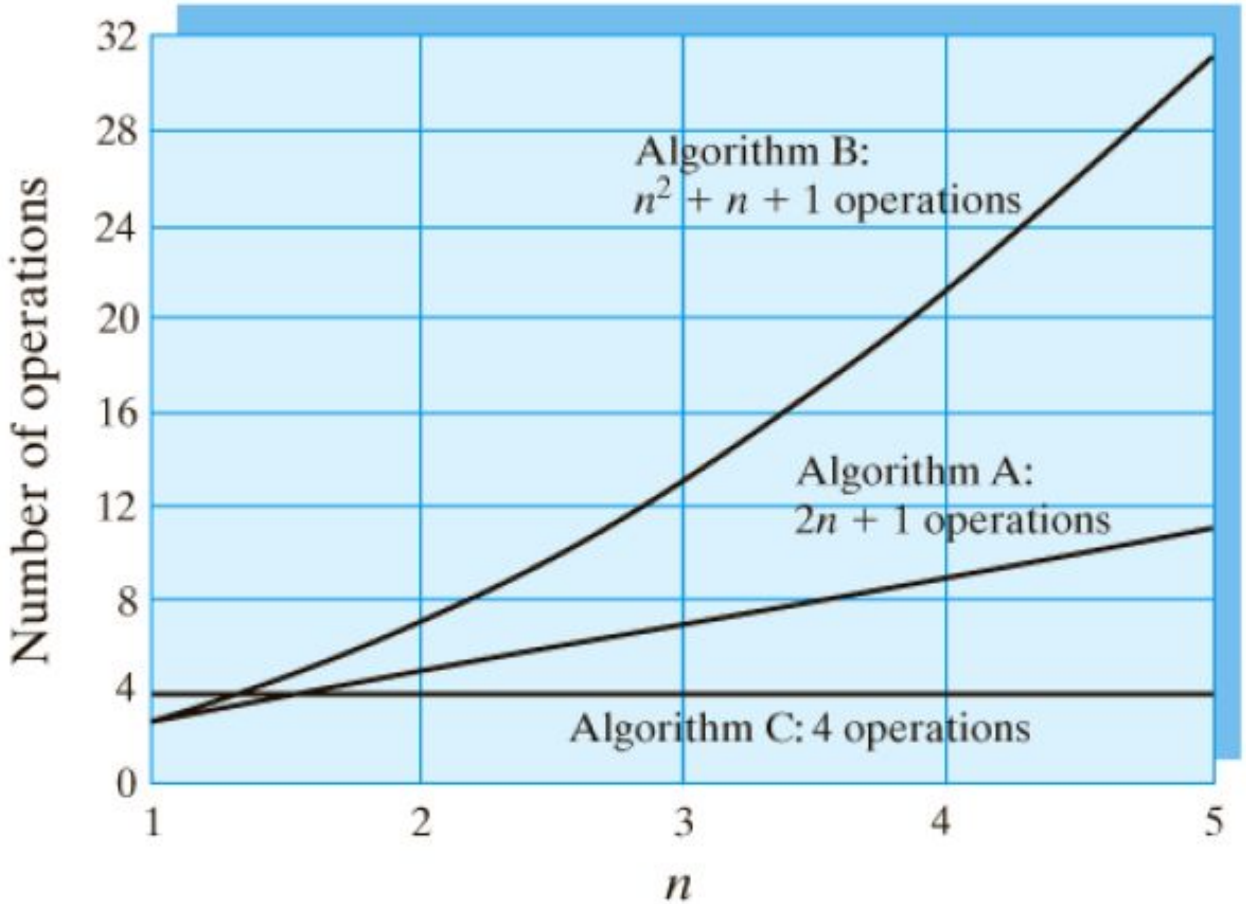
Exemplo

- Número de operações

	Algorithm A	Algorithm B	Algorithm C
Assignments	$n + 1$	$1 + n(n + 1) / 2$	1
Additions	n	$n(n + 1) / 2$	1
Multiplications			1
Divisions			1
Total operations	$\mathbf{2n + 1}$	$\mathbf{n^2 + n + 1}$	**4**

Exemplo

- Gráfico do número de operações



Notação assintótica

– Métodos padrões para descrever o tempo de execução assintótica de um agoritmo de forma simplificada

$\Theta\left(f(n)\right)$: Theta

$O\left(f(n)\right)$: Ó, Ó maiúsculo, Ózão

$o\left(f(n)\right)$: Ó minúsculo, Ózinho

$\Omega\left(f(n)\right)$: Ômega

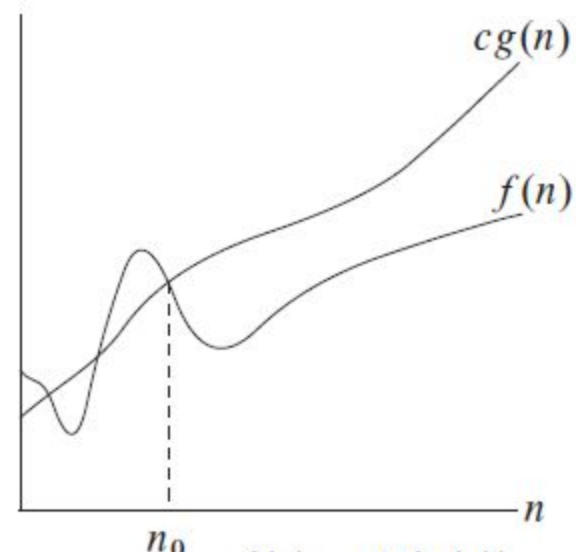
$\omega\left(f(n)\right)$: Ômega minúsculo, Ômegazinho

Notação O

- Dada uma função $g(n)$, denota-se por $O(g(n))$ o conjunto de funções:

$$O\big(g(n)\big) = \{f(n)\text{: existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \le f(n) \le cg(n) \text{ para todos } n \ge n_0\}$$

  - Ou seja, uma função $f(n)$ pertence ao conjunto $O\big(g(n)\big)$ se existir uma constante positiva c tal que faça ela ficar com custo abaixo ou igual a $cg(n)$, para n suficientemente grande



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

(b)

Notação O

- Dominação assintótica
  - Definição: Uma função $g(n)$ domina assintoticamente outra função $f(n)$ se
    $f(n) = O(g(n))$
  - Quando dizemos que o tempo de execução $T(n)$ de um algoritmo é $O(n^2)$, significa que existem constantes c e n_0 tais que, para valores de $n \geq n_0$, $T(n) \leq cn^2$
  - Exemplo:

$$f(n) = (n+1)^2$$

Logo, $f(n)$ é $O(n^2)$, quando $n_0 = 1$ e $c = 4$. Porque $(n+1)^2 \leq 4n^2$ para $n \geq 1$.

Exemplo

- Sejam $f(n) = 3 + 2/n$ e $g(n) = n^0$

$$g(n) = n^0 = 1$$
$$3 + \frac{2}{n} \leq 1 + 3 = 4$$

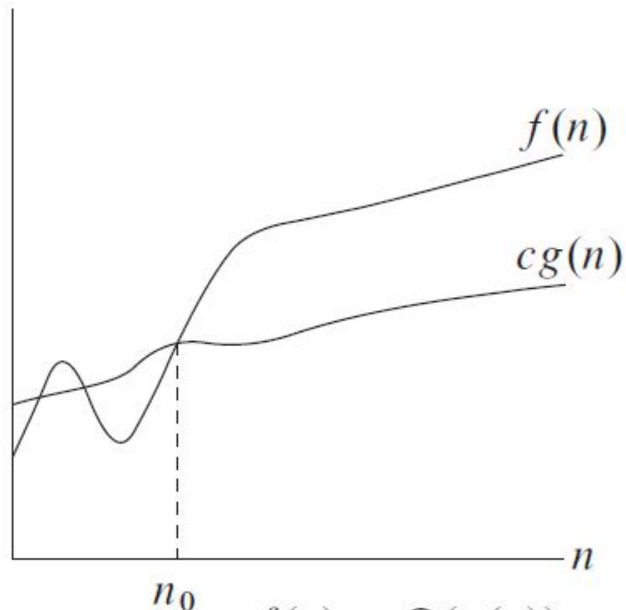
Desde que $n \geq 2$. Logo, $f(n) \leq 4g(n)$, para $n \geq 2$.
Então, para $n_0 = 2$ e $c = 4$, $f(n) = O(g(n))$.

Notação Ω

- Especifica um limite inferior para $g(n)$
  - Dada uma função $g(n)$, denota-se por $\Omega(g(n))$ o conjunto de funções:

$$\Omega\big(g(n)\big) = \{f(n)\text{: existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \le cg(n) \le f(n) \text{ para todos } n \ge n_0\}$$

  - Ou seja, uma função $f(n)$ pertence ao conjunto $\Omega\big(g(n)\big)$ se existir uma constante positiva c tal que faça ela ficar com custo superior ou igual a $cg(n)$, para n suficientemente grande



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

Notação Θ

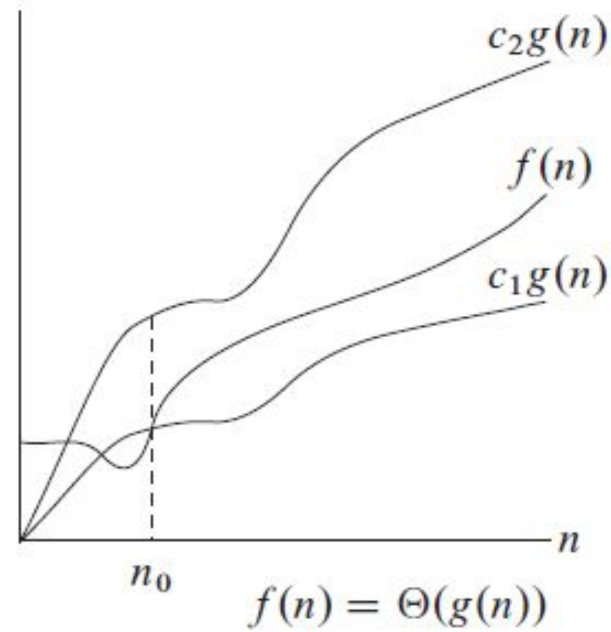
- Dada uma função $g(n)$, denota-se por $\Theta(g(n))$ o conjunto de funções:

$$\Theta(g(n)) = \{f(n)\text{: existem constantes positivas } c_1, c_2 \ e \ n_0 \text{ tais que}$$
$$0 \ \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{para todos } n \geq n_0\}$$

  - Ou seja, uma função $f(n)$ pertence ao conjunto $\Theta(g(n))$ se existirem constantes positivas c_1 e c_2 tais que ela possa ser imprensada entre $c_1 g(n)$ e $c_2 g(n)$, para n suficientemente grande

Notação Θ

- Apesar de representar um conjunto, comumente escreve-se $f(n) = \Theta(g(n))$ ao invés de $f(n) \in \Theta(g(n))$
- $g(n)$ é um **limite assintoticamente restrito** para $f(n)$
  - Θ define **limites superior e inferior**



$$\Theta(g(n)) = \{f(n)\text{: existem constantes positivas } c_1, c_2\ e\ n_0 \text{ tais que}$$
$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{para todos } n \ge n_0\}$$

Notação Θ

- Exemplo: $f(n) = \frac{1}{2}n^2 - 3n$
- $f(n) = \Theta(n^2)$?
  - $c_1 n^2 \leq (\frac{1}{2}n^2 - 3n) \leq c_2 n^2$ (para todo $n \geq n_0$)
  - $c_1 \leq (\frac{1}{2} - \frac{3}{n}) \leq c_2$
    - Inequação da direita: Para $n \geq 1, c_2 \geq \frac{1}{2}$
    - Inequação da esquerda: Para $n \geq 7, c_1 \leq \frac{1}{14}$
  - Assim, escolhendo $c_1 = \frac{1}{14}, c_2 = \frac{1}{2}$ e $n_0 = 7$, verificamos que:

$$\frac{1}{2}n^2 - 3n = \Theta(n^2)$$

Notação Θ

- Exemplo: $f(n) = 6n^3$
- $f(n) \neq \Theta(n^2)$?
  - Supondo que existam c_2 e n_0 tais que $6n^3 \leq c_2 n^2$ para todo $n \geq n_0$
    - Implica: $n \leq \frac{c_2}{6}$
      - Não é válido para *n* grande e arbitrário, pois c_2 é constante.

Notação o

- Define um limite superior que não é assintoticamente firme (limite estritamente superior)
  - Dada uma função $g(n)$, denota-se por $o(g(n))$ o conjunto de funções:

$$o\big(g(n)\big) = \{f(n)\text{: para qualquer constante positiva } \mathrm{c} > 0 \text{ existe uma constante } n_0 > 0 \text{ tal que } 0 \ \leq f(n) < cg(n) \text{ para todo } n \geq n_0\}$$

  - Ou seja, uma função $f(n)$ pertence ao conjunto $o\big(g(n)\big)$ se $cg(n)$ se mantém como um limite estritamente superior para **todas** constantes $c > 0$ e para algum $n_0 > 0$.
  - A função $f(n)$ se torna insignificante em relação a $g(n)$ quando n tende ao infinito.
    - $\lim\limits_{n \to \infty} \dfrac{f(n)}{g(n)} = 0$

Notação ω

- Por analogia, a notação ω está relacionada a Ω da mesma forma que a notação o está relacionada com O (limite estritamente inferior)
  - Dada uma função $g(n)$, denota-se por $\omega(g(n))$ o conjunto de funções:

$$\omega\left(g(n)\right) = \{f(n)\text{: para qualquer constante positiva } \mathrm{c} > 0\text{, existe uma constante } n_0 > 0 \text{ tal que } 0 \ \leq cg(n) < f(n) \text{ para todo } n \geq n_0\}$$

  - Ou seja, uma função $f(n)$ pertence ao conjunto$\omega\left(g(n)\right)$ se $cg(n)$ se mantém como um limite estritamente inferior para **todas** constantes $c > 0$ e para algum $n_0 > 0$.
  - A função $f(n)$ se torna arbitrariamente grande em relação a $g(n)$ quando n tende ao infinito.
    - $\lim\limits_{n \to \infty} \frac{f(n)}{g(n)} = \infty$

Analogia

- $f(n) = \Theta\left(g(n)\right) \approx a = b$

  $f(n) = O\left(g(n)\right) \approx a \leq b$

  $f(n) = \Omega\left(g(n)\right) \approx a \geq b$

  $f(n) = o\left(g(n)\right) \approx a < b$

  $f(n) = \omega\left(g(n)\right) \approx a > b$

Classes de Comportamento Assintótico

- Se f é uma **função de complexidade** para um algoritmo *F*, então $O(f)$ é considerada a **complexidade assintótica** do algoritmo *F*
  - A relação de dominação assintótica permite comparar funções de complexidade
  - Se as funções f e g dominam assintoticamente uma a outra, os algoritmos associados são equivalentes

Comparações entre algoritmos

- Podemos avaliar algoritmos comparando as suas funções de complexidade, negligenciando as constantes de proporcionalidade.
- Ex 1: Um algoritmo com tempo $O(n)$ é mais rápido que outro com tempo $O(n^2)$
- Ex 2: Um programa leva $100n$ unidades de tempo para ser executado e o outro leva $2n^2$. Qual é melhor?
  - Para $n < 50$, $2n^2$ é melhor.
  - Para $n > 50$, $100n$ é melhor.
- Ex 3: Qual algoritmo é preferível? Um de $O(n \log n)$ ou um $\Omega(n \log n)$?

Logaritmo

- Função exponencial inversa
- $y = b^x$, equivalente a $x = \log_b y$.

- Base 2:
  - Reflete a quantidade de vezes que podemos dobrar algo até obtermos n
    - Altura de árvore binária completa
  - Quantas vezes podemos dividir n ao meio até obtermos 1
    - Busca binária

Busca binária

- Em busca binária, a cada comparação retiramos metade dos itens do espaço de busca.
  - 20 comparações são necessárias para se encontrar um elemento entre 1.000.000 de itens
- Quantas vezes podemos dividir n ao meio até obtermos 1?
  - $\lceil \log n \rceil$

Logaritmo e árvores binárias

- Qual altura de árvore binária necessária para se ter n folhas?
  - Número máximo de folhas dobra a cada nível

Logaritmo e bits

- Quantos bits você precisa para representar números de 0 a $2^i - 1$?
- Cada bit adicional permite o dobro de quantidade de padrões
  - $\log 2^i$

A base do logaritmo não é assintoticamente importante

- Por definição, $c^{\log_c x} = x$
- $\log_b a = \frac{\log_c a}{\log_c b}$

- Ex: Base 2 x Base 100:
  - $\log_2 n = \frac{\log_{100} n}{\log_{100} 2} = 6.643 \log_{100} n$

Principais Classes de Problema

- $f(n) = O(1)$
  - Algoritmos de complexidade $O(1)$ são ditos de **complexidade constante**.
  - O seu tempo de execução independe de n.

- $f(n) = O(\log n)$
  - **Complexidade logarítmica.**
  - Pode-se considerar o tempo de execução como menor do que uma constante grande.
  - Ex: $n = 1000, \log_2 n \approx 10.$

    $n = 1.000.000, \log_2 n \approx 20,\ \log_{10} n = 6$

Principais Classes de Problema

- $f(n) = O(n)$
  - **Complexidade linear**.
  - Em geral, algum trabalho é realizado sobre cada elemento da entrada
    - Melhor solução possível para um algoritmo que precisa processar/produzir n elementos de entrada/saída.

- $f(n) = O(n\log n)$
  - Típico em algoritmos que quebram um problema em outros menores, resolvendo cada um deles independentemente e unindo suas soluções.
  - Ex:

$$n = 1.000.000, n\log_2 n \approx 20.000.000$$
$$n = 2.000.000, n\log_2 n \approx 42.000.000$$

Principais Classes de Problema

- $f(n) = O(n^2)$
  - **Complexidade quadrática**.
  - Em geral, ocorre quando itens são processados aos pares, muitas vezes dentro de dois loops.
  - Ex:

    $n = 1000, n^2 = 1.000.000$

- $f(n) = O(n^3)$
  - **Complexidade cúbica**
  - Úteis apenas para resolver pequenos problemas
  - Ex:

    $n = 100, n^3 = 1.000.000$

Principais Classes de Problema

- $f(n) = O(2^n)$
  - **Complexidade exponencial**.
  - Geralmente não são úteis sob o ponto de vista prático.
  - Ocorrem na solução de problemas complexos por **força bruta.**
  - Ex:
    - $n = 20,\ 2^n = 1.000.000$

- $f(n) = O(n!)$
  - **Complexidade exponencial**
  - Geralmente ocorrem na solução de problemas complexos por **força bruta.**
  - Ex:

    $n = 20, n! = 2.432.902.008.176.640.000$ (19 dígitos)

    $n = 40, n! =$ (número de 48 dígitos)

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$

Algoritmos Polinomiais x Algoritmos Exponenciais

- Algoritmo Polinomial
  - Tempo de execução tem função de complexidade $O(p(n))$, onde $p(n)$ é um polinômio.
- Algoritmo Exponencial
  - Tempo de execução tem função de complexidade $\Omega(c^n)$, para $c > 1$.
- A diferença entre essas duas classes se torna significativa quando o tamanho do problema n cresce.
- Algoritmos exponenciais: variações de pesquisa exaustiva.
  - Em alguns casos podemos obter uma solução polinomial mediante melhor entendimento da estrutura do problema.
- Um problema é considerado:
  - Intratável: se não existe um algoritmo polinomial para resolvê-lo
  - Tratável: quando existe um algoritmo polinomial para resolvê-lo.

Exercícios

1)Verdadeiro ou falso? Justifique.

a) $2^{n+1} = O(2^n)$?

b) $2^{2n} = O(2^n)$?

c)$\sqrt{n} = O(\log n)$?

d) $\sum_{i=1}^{n} 3^i = \Theta(3^n)$?

Exercícios

2) Para cada dos itens seguintes, escolha uma das seguintes relações:

$f(n) = O\big(g(n)\big)$,

$f(n) = \Omega\big(g(n)\big)$ ou

$f(n) = \Theta\big(g(n)\big)$:

a) $f(n) = \log n^2$; $g(n) = \log n + 5$

b) $f(n) = n$; $g(n) = \log^2 n$

c) $f(n) = 2^n$; $g(n) = 3^n$

Exercícios

3) Mostre que:

a) $f(n) = 5n^2 + 10n = \Theta(n^2)$

b) $f(n) = 100n^2 = O(n^2)$

c) $f(n) = 100n^2 = \Omega(n^2)$

Exercícios

4) Qual valor a seguinte função retorna? Expresse sua resposta em função de n. Forneça a complexidade do tempo de execução do pior caso usando a notação *O.*

```
int loops(n){
  r=0;
  for(i=1; i<=n-1; i++)
    for(j=i+1; j<=n; j++)
      for(k=1; k<=j; k++)
        r+=1;
  return r;
}
```

Referências

- CLRS, Introduction to Algorithms, 3rd ed.
  - Cap. 1, 2.2, 3.1, 3.2
- Ziviani, Projeto de Algoritmos, 3ª ed
  - 1.1-1.3
- Skiena, The Algorithm Design Manual, 2nd ed.
  - 1.3