ENGENHARIA INFORMÁTICA E DE COMPUTADORES

Algoritmos e Estruturas de Dados

(parte 4 – Técnicas de Análise de Algoritmos)

2° Semestre 2022/2023
Instituto Superior de Engenharia de Lisboa
Paula Graça

ANÁLISE DE ALGORITMOS

Not everything that can be counted counts, and not everything that counts can be counted

Albert Einstein

- Análise de Algoritmos
  - A investigação da eficiência dos algoritmos incide em dois recursos
    - Tempo de execução
    - Espaço de memória

ANÁLISE DE ALGORITMOS

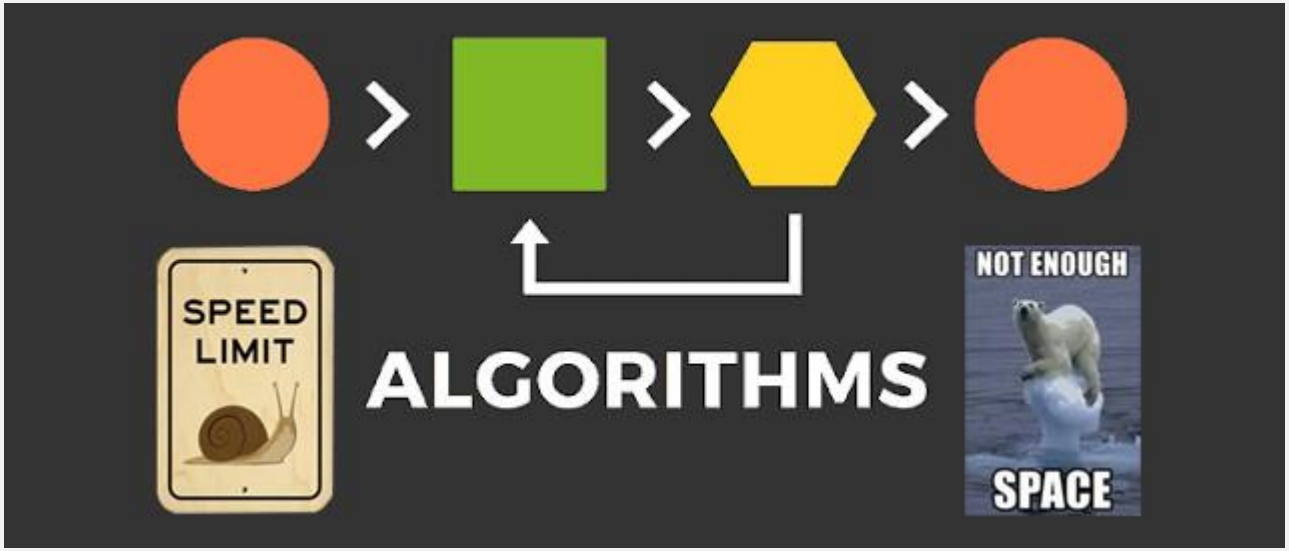
- A avaliação de algoritmos pode ser feita em três dimensões
  - Simplicidade
  - Generalidade
  - Eficiência
- Ao contrário da simplicidade e da generalidade, a eficiência de um algoritmo pode ser avaliada em termos quantitativos precisos

ANÁLISE DE ALGORITMOS

- A eficiência pode ser analisada em termos de
  - Tempo
    - Qual a velocidade de execução do algoritmos
  - Espaço
    - Espaço requerido pelo algoritmo
- Meio século de inovação tecnológica, melhorou enormemente a velocidade e dimensão da memória dos computadores

ANÁLISE DE ALGORITMOS

- Hoje
  - O espaço requerido por um algoritmo, tipicamente já não é tão importante
  - Em relação ao tempo que não diminuiu assim tão significativamente



ANÁLISE DE ALGORITMOS

- 3ª Geração (1965-80)
  - Um desenvolvimento importante durante a 3ª geração, foi o fenomenal crescimento dos Minicomputadores, com início no DEC PDP-1 em 1961
  - Tinha um processador de 18 bits e memória de apenas 4K de palavras de 18 bits

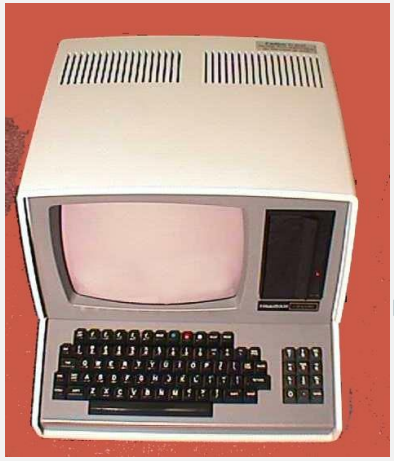
PDP-1

PDP-11

ANÁLISE DE ALGORITMOS

- 4ª Geração (a partir de 1980)
  - Com o aparecimento dos circuitos integrados LSI (*Large Scale Integration*), apenas um centímetro quadrado de silício, continha milhares de transístores, surgem os computadores pessoais, inicialmente designados por microcomputadores
  - A Intel lança o primeiro CPU 8088 de 8 bits

IBM 8080



Zilog Z80

- Gary Kildall contactado pela Intel Corporation criou o sistema operativo CP/M (*Control Program for Microcomputers*) que ocupava menos de 4Kb

ANÁLISE DE ALGORITMOS

- Métricas de um algoritmo
  - Dimensão de Entrada
    - Quantidade de dados de entrada
  - Tempo de Execução
    - Medida do tempo de execução
  - Grau de Crescimento
    - Medida da ordem de crescimento do tempo de execução em função do incremento da dimensão de entrada

MÉTRICAS

- Dimensão de Entrada
  - Quase todos os algoritmos são mais lentos para uma maior dimensão dos dados de entrada
  - Assim, a eficiência de um algoritmo deve ser analisada em função de um parâmetro N que indica a dimensão dos dados de entrada do algoritmo
  - Exemplos
    - Ordenar uma lista de valores (N)
    - Pesquisar numa lista de valores (N)
    - Encontrar o menor numa lista de valores (N)
    - Multiplicação de matrizes (N x M)

MÉTRICAS

- Tempo de Execução
  - Utilização de unidades de tempo tem desvantagens
    - Dependência da velocidade do computador
    - Dependência da qualidade do programa que implementa o algoritmo
    - Dependência do compilador usado para gerar o código máquina
  - É desejável uma métrica que não dependa destes fatores

MÉTRICAS

- T(n) – Tempo de execução de um algoritmo
  - Identificação das várias operações
  - Contabilização do número de vezes em que cada operação é executada
  - Cálculo do tempo de execução de cada operação

TEMPO DE EXECUÇÃO

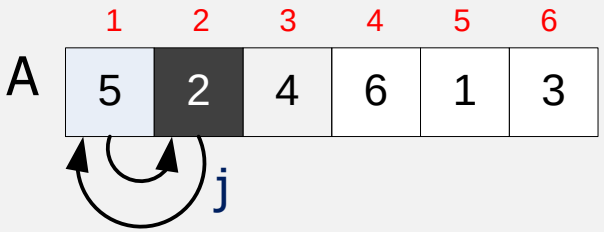
- **T(n)** – Tempo de execução de um algoritmo para uma dada entrada de dimensão **n**

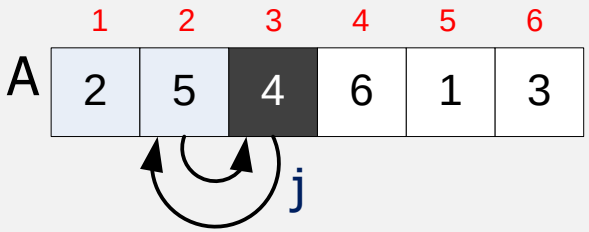
$$T(n) = \sum_i C_i . C(i)$$

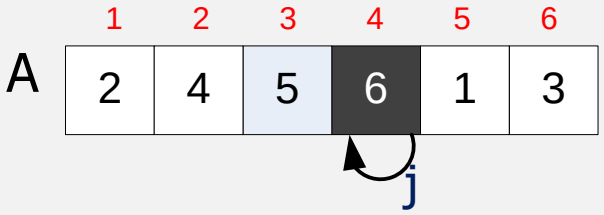
C_i – tempo de execução (**custo**) da operação **i** num determinado computador

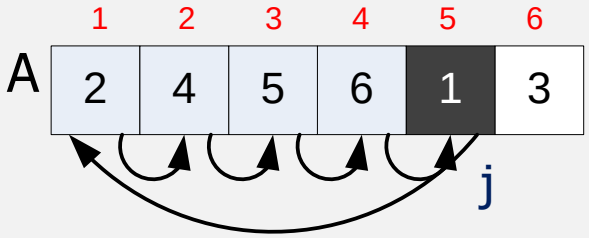
C(i) – número de vezes que o algoritmo executa a operação **i** para uma entrada de dimensão **n**

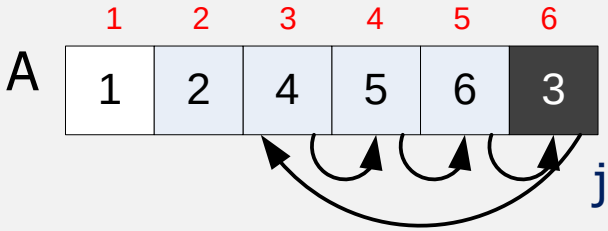
ANÁLISE EMPÍRICA DO INSERTION SORT

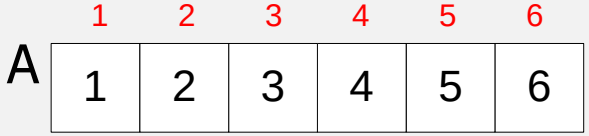


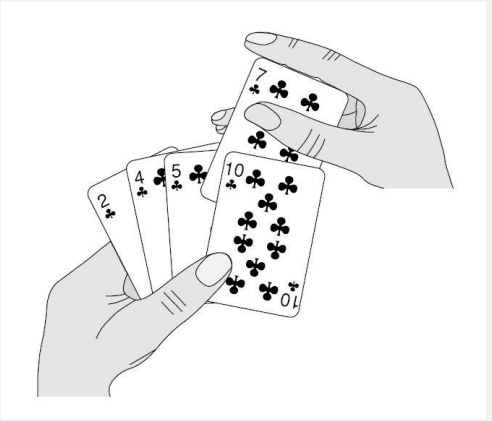












O *array* A é composto por:

A[1 .. j-1] – elementos ordenados
A[j .. N] – elementos desordenados para inserir na posição correta

ANÁLISE EMPÍRICA DO INSERTION SORT

Linha		Custo	Nºvezes
	Insertion-Sort (A)		
1	for j = 2 to A.size	C_1	n
2	curr = A[j]	C_2	n-1
3	// insert A[k] into the sorted sequence	C_3	0
4	k = j - 1	C_4	n-1
5	while k > 0 and curr < A[k]	C_5	$\sum_{j=2}^{n} c_j$
6	A[k+1] = A[k]	C_6	$\sum_{j=2}^{n} c_j - 1$
7	k = k – 1	C_7	$\sum_{j=2}^{n} c_j - 1$
8	A[k+1] = curr	C_8	n-1

n = A.size

C_j = nº de vezes que o *while* é executado para cada valor de j

A condição de teste dos ciclos *for* e *while* significa uma iteração adicional

ANÁLISE EMPÍRICA DO INSERTION SORT

- Assume-se que
  - A execução de cada linha (operação) tem um custo **i**
    - Sendo C_i o seu tempo de execução $C_1, C_2, \ldots, C_n$
  - Sendo n = Array.size
    - C_j é o número de vezes que cada operação é executada para cada valor de **j**
    - Num ciclo, o teste da condição quando falsa, significa uma iteração adicional em relação às operações dentro do ciclo
  - Os comentários têm tempo de execução zero

ANÁLISE EMPÍRICA DO INSERTION SORT

- Assim pela fórmula $T(n) = \sum_i C_i . C(i)$

  - Uma operação (linha) que tenha um custo **C_i** e que execute **n** vezes, contribui com o tempo de execução

    C_i n

  - O tempo total de execução do algoritmo é o somatório de todas as operações

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5 \sum_{j=2}^{n} c_j + C_6 \sum_{j=2}^{n}(c_j - 1) + C_7 \sum_{j=2}^{n}(c_j - 1) + C_8(n-1)$$

ANÁLISE EMPÍRICA DO INSERTION SORT

- Melhor caso 
  - O *array* já está ordenado
  - Para cada j = 2, 3, …, n A[k] <= key
  - Assim, C_j = 1 para j = 2, 3, …, n

$$T_{best}(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5 \sum_{j=2}^{n} 1 + C_8(n-1)$$

ANÁLISE EMPÍRICA DO INSERTION SORT

- Melhor caso (cont.)

Sendo

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

$$T_{best}(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5(n-1) + C_8(n-1)$$

$$T_{best}(n) = (C_1 + C_2 + C_4 + C_5 + C_8)n - (C_2 + C_4 + C_5 + C_8)$$

O tempo de execução pode ser expresso na forma:

$T(n) = an + b$ → função linear de n com a e b constantes

ANÁLISE EMPÍRICA DO INSERTION SORT

- Pior caso 

  - O *array* está por ordem inversa
  - Cada elemento A[j] tem que ser comparado com todo o *subarray* ordenado A[1 .. j -1]

  - Assim, C_j = j para j = 2, 3, …, n

j	k	C_j
2	1 0	2
3	2 1 0	3
4	3 2 1 0	4
…	…	…
n	(n-1) (n-2) … 0	n

$C_j = j$

ANÁLISE EMPÍRICA DO INSERTION SORT

- Pior caso (cont.)
  - Sendo Cj = j, pode ser substituído no somatório
  - O somatório pode ser então calculado

$$\sum_{j=2}^{n} c_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

Dado que $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

$$\sum_{j=2}^{n} (c_j - 1) = \sum_{j=2}^{n} j - 1 = \frac{n(n-1)}{2}$$

Dado que $\sum_{i=1}^{n} i - 1 = \frac{n(n-1)}{2}$

ANÁLISE EMPÍRICA DO INSERTION SORT

- Pior caso (cont.)
  - Substituindo os somatórios, a expressão pode ser simplificada

$$T_{wor\,st}(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5\left(\frac{n(n+1)}{2} - 1\right) + C_6\left(\frac{n(n-1)}{2}\right) + C_7\left(\frac{n(n-1)}{2}\right) + C_8(n-1)$$

$$T_{wor\,st}(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5\left(\frac{n^2}{2} + \frac{n}{2} - 1\right) + C_6\left(\frac{n^2}{2} - \frac{n}{2}\right) + C_7\left(\frac{n^2}{2} - \frac{n}{2}\right) + C_8(n-1)$$

ANÁLISE EMPÍRICA DO INSERTION SORT

- Pior caso (cont.)

$$T_{wor\ st}(n) = \left(\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2}\right) n^2 + \left(C_1 + C_2 + C_4 + \frac{C_5}{2} - \frac{C_6}{2} - \frac{C_7}{2} + C_8\right) n - (C_2 + C_4 + C_5 + C_8)$$

O tempo de execução pode ser expresso na forma:

$$T(n) = an^2 + bn + c$$

→ função quadrática de n com a, b e c constantes

ANÁLISE EMPÍRICA DO INSERTION SORT

- Caso médio 
  - A eficiência do *Insertion-Sort*
    - Se os elementos estiverem quase ordenados, degrada pouco em relação ao melhor caso, aproximando-se de um tempo de execução linear
    - Se os elementos estiverem bastante desordenados, o tempo de execução é quadrático tal como no pior caso
  - A análise exata de um algoritmo, neste caso do *Insertion-Sort*, é frequentemente uma análise difícil e fastidiosa

GRAU DE CRESCIMENTO

- Quanto tempo mais demorará o algoritmo duplicando n?

Sendo:

$$T(n) = \sum_i C_i . C(i) \approx C_{op} . C(n)$$

C_{op} - tempo de execução da operação que mais contribui para o tempo total
C(n) - o número de vezes que é executada

- Assumindo que num algoritmo com entrada n: $C(n) = \frac{1}{2} n(n-1)$

- Simplificando a expressão: $C(n) = \frac{1}{2} n(n-1) = \frac{1}{2} n^2 - \frac{1}{2} n \approx \frac{1}{2} n^2$

- A questão pode ser respondida sem conhecer o valor de C_{op}

$$\frac{T(2n)}{T(n)} \approx \frac{C_{op}\, C(2n)}{C_{op}\, C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2} n^2} = \frac{4n^2}{n^2} = 4$$

(Demora 4 vezes mais)

GRAU DE CRESCIMENTO

$$T(n) \approx C(n)$$

- Ignorando C_{op}, ou seja, sem conhecer o tempo de execução das operações num determinado computador,
- a métrica C(n), ou seja, o número de vezes em que as operações são executadas em função da dimensão da entrada n,
- dá-nos uma ideia precisa do tempo de execução T(n) do algoritmo relativamente ao seu grau de crescimento

GRAU DE CRESCIMENTO

- A diferença do tempo de execução para entradas pequenas de n, não distingue os algoritmos eficientes dos ineficientes

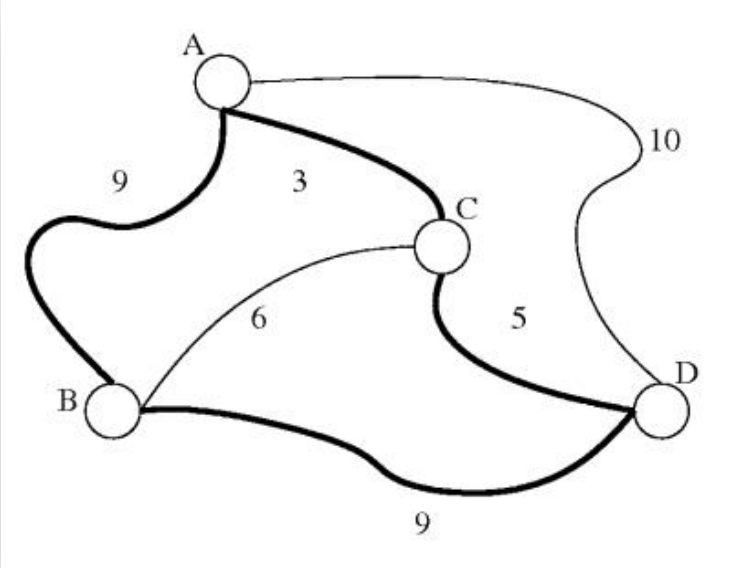
- Problema do Máximo Divisor Comum entre dois números
  - Para valores pequenos de n não é evidente a diferença de eficiência entre os dois algoritmos
  - Para grandes valores de n o algoritmo da direita é bastante mais eficiente

Stage	*a*	*b*
1	60	32
2	28	32
3	28	4
4	24	4
5	20	4
6	16	4
7	12	4
8	8	4
9	4	4

Stage	*a*	*b*
1	60	32
2	32	28
3	28	4
4	4	0

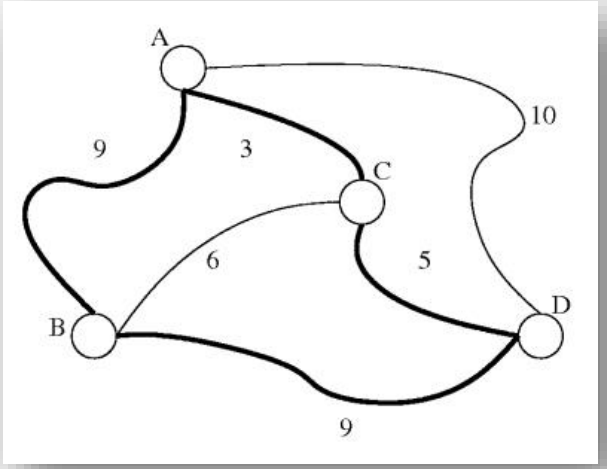
GRAU DE CRESCIMENTO

- Problema: Caixeiro Viajante
  - Um caixeiro viajante tem de visitar um conjunto de cidades interligadas por uma rede de estradas e voltar à cidade de origem
  - A distância a percorrer entre cada par de cidades é variável. Qual o caminho mais curto no qual o caixeiro viajante passe por todas as cidades uma única vez?



GRAU DE CRESCIMENTO

- Exemplo com 4 cidades, as possibilidades são as seguintes:
  - **A -> …**
  - **B -> A -> C -> D -> B = 9 + 3 + 5 + 9 = 26**
  - **B -> D -> C -> A -> B = 9 + 5 + 3 + 9 = 26**
  - **B -> C -> D -> A -> B = 6 + 5 + 10 + 9 = 30**
  - **B -> A -> D -> C -> B = 9 + 10 + 5 + 6 = 30**
  - **B -> D -> A -> C -> B = 9 + 10 + 3 + 6 = 28**
  - **B -> C -> A -> D -> A = 6 + 3 + 10 + 9 = 28**
  - **C -> …**
  - **D -> …**



Retirando os percursos inversos

  - Total = 4! / 2 = 24 / 2 = **12**
- Exemplo com 10 cidades:
  - Total = 10!/2 = **1.814.400** possibilidades diferentes

GRAU DE CRESCIMENTO

- Para grandes valores de entrada n, é o grau de crescimento da função que conta

n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	n!
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Conversão de segundos

segundos	
10^2	1.7 minutos
10^4	2.8 horas
10^5	1.1 dias
10^6	1.6 semanas
10^7	3.8 meses
10^8	3.1 anos
10^9	3.1 décadas
10^{10}	3.1 séculos
10^{11}	*nunca*

GRAU DE CRESCIMENTO

- As funções logarítmicas são as que têm menor grau de crescimento
  - Embora o valor da função algorítmica dependa da base, é sempre possível a conversão de bases através da fórmula

$$log_a n = log_a b * log_b n$$

  - Ou seja

$$log_a n = log_a 2 * log_2 n$$

  - Ficando o grau de crescimento na ordem da função logarítmica de base 2 (na tabela), multiplicada por uma constante

GRAU DE CRESCIMENTO

- As funções exponencial 2^n e factorial n! têm um crescimento tão rápido, que os seus valores são astronomicamente grandes mesmo para pequenos valores de n
  - 2^{100} operações
    - Demorariam $4x10^{10}$ anos a ser executadas por um computador à velocidade de 1 trilião (10^{12}) de operações por segundo
  - 100! operações
    - Demorariam mais de 4.5 biliões ($4.5x10^9$) de anos – a idade estimada do planeta terra

GRAU DE CRESCIMENTO

- Embora exista diferença entre as ordens de crescimento das funções 2^n e $n!$, ambas são designadas por

  Funções de crescimento exponencial

  ou simplesmente Funções exponenciais

Algoritmos que requeiram um número exponencial de operações, servem para resolver apenas problemas de pequena dimensão

ANÁLISE DO CUSTO DE ALGORITMOS ITERATIVOS

- Melhor caso, Pior Caso e Caso Médio

  - Muitos algoritmos não dependem apenas da dimensão da entrada n
  - A distribuição dos elementos numa lista com a mesma dimensão, pode influenciar o cálculo do tempo de execução

```
SequentialSearch (A[1..n], k)

i = 1
while i <= n and a[i] ≠ k do
      i = i + 1
if i <= n return i
else return -1
```

ANÁLISE DO CUSTO DE ALGORITMOS ITERATIVOS

- Melhor caso
  - $C_{best}(n)$ – É a eficiência do algoritmo para o melhor caso, ou seja, é quando o algoritmo tem o tempo de execução mais rápido de entre todas as combinações possíveis dos n elementos

- Pior caso
  - $C_{worst}(n)$ – Providencia informação muito importante sobre a eficiência de um algoritmo, pois indica o limite superior do tempo de execução

- Caso Médio
  - $C_{avg}(n)$ – É a eficiência numa entrada típica (aleatória). Não pode ser obtida a partir da média do melhor e pior caso

ANÁLISE DO CUSTO DE ALGORITMOS ITERATIVOS

- Melhor caso

  - O elemento procurado é o primeiro da lista

$$C_{best}(n) = 1$$

  - A análise da eficiência do melhor caso não é tão importante como a do pior caso

  - Pode-se tirar proveito pelo facto de que um bom desempenho no melhor caso nalguns algoritmos, estende-se a algumas combinações de entradas que estejam perto do melhor caso

ANÁLISE DO CUSTO DE ALGORITMOS ITERATIVOS

- Pior caso

  - O elemento procurado é o ultimo da lista

$$C_{worst}(n) = n$$

  - Esta é a eficiência do algoritmo no pior caso, para uma entrada de dimensão n

  - O algoritmo tem o tempo de execução mais lento de entre todas as combinações possíveis dos elementos da lista de dimensão n

ANÁLISE DO CUSTO DE ALGORITMOS ITERATIVOS

- Caso médio
  - A análise da eficiência do caso médio é muito importante e consideravelmente mais difícil que a análise do melhor e pior caso

$$C_{avg}(n) = \frac{(1 + 2 + \cdots + i + \cdots + n)}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

  - Existem muitos algoritmos para os quais a eficiência do caso médio é muito melhor que a excessivamente pessimista do pior caso
  - Sem a análise do caso médio, muitos algoritmos importantes poderiam não ser considerados (exemplo: *Insertion Sort* com os elementos quase ordenados)

PROGRESSÃO ARITMÉTICA

- É uma sequência numérica em que cada termo, a partir do segundo, é igual à soma do termo anterior com uma constante r. O valor r é chamado de razão da progressão aritmética

$$\begin{cases} a_1 = a \\ a_i = a_{i-1} + r, \end{cases} \quad i > 1 \implies a_n = a_1 + r(n-1)$$

- S_n é a soma de todos os termos da progressão aritmética:

$$S_n = \sum_{i=1}^{n} a_i = n(a_1 + a_n)/2$$

PROGRESSÃO ARITMÉTICA

- Exemplos de progressões aritméticas
  - Se a = 0 e r = 1 então $a_1 = 0$ e $a_n = 0 + 1(n - 1) = (n - 1)$

$$S_n = \sum_{i=1}^{n} i - 1 = n(n-1)/2 \qquad \text{k=i-1} \qquad = \sum_{k=0}^{n-1} k = n(n-1)/2$$

  - Se a = 1 e r = 1 então $a_1 = 1$ e $a_n = 1 + 1(n - 1) = n$

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

SOMATÓRIOS

$$\sum_{i=l}^{n} 1 = 1 + 1 + 1 + \cdots + 1 = u - l + 1$$

(u – l + 1 vezes)

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

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2} n^2$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{2n^3 + 3n^2 + n}{6} \approx \frac{1}{3} n^3$$

$$\sum_{i=1}^{n} i^k = 1^k + 2^k + 3^k + \cdots + n^k \approx \frac{1}{k+1} n^{k+1}$$

SOMATÓRIOS

$$\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \quad (a \neq 1)$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=1}^{n} i2^i = 1 * 2 + 2 * 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$$

PROPRIEDADES DOS SOMATÓRIOS

$$\sum_{i=1}^{n} c a_i = c \sum_{i=1}^{n} a_i$$

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

NÚMEROS HARMÓNICOS

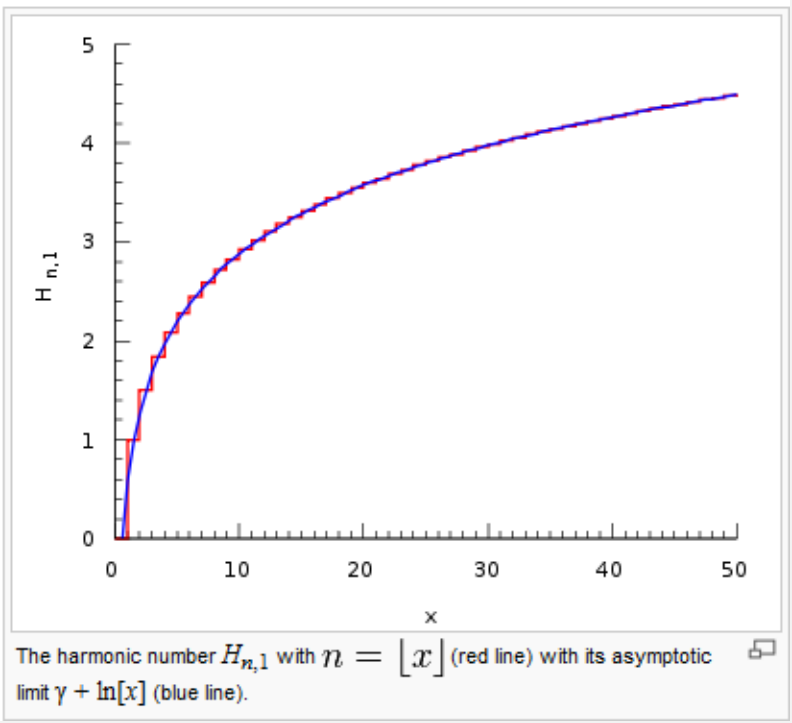
- É a soma dos inversos dos n primeiros números naturais

$$H_N = \sum_{k=1}^{N} \frac{1}{\mathrm{k}} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\mathrm{N}}$$

- Aproxima-se dos logaritmos naturais a menos da constante de Euler

$$H_N \approx \ln \mathrm{N} + \gamma$$

$$\gamma = 0.57721 \text{ (constante de Euler)}$$



The harmonic number $H_{n,1}$ with $n = \lfloor x \rfloor$ (red line) with its asymptotic limit $\gamma + \ln[x]$ (blue line).

- O nome harmónico é devido à semelhança com a proporcionalidade dos comprimentos de onda de uma corda de musica a vibrar

IDENTIDADE DOS LOGARITMOS

$$a = b^{\log_b a}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^n = n\log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b\left(\frac{1}{a}\right) = -\log_b a$$

$$\log_b(a) = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

IDENTIDADE DOS EXPONENCIAIS

$$a^{-1} = \frac{1}{a}$$

$$(a^m)^n = a^{mn}$$

$$a^m a^n = a^{m+n}$$

$$2^{n-1} + 2^{n-1} = 2^n$$

$$3^{n-1} + 3^{n-1} + 3^{n-1} = 3^n$$

NOTAÇÕES DOS LOGARITMOS

$\lg n = log_2 n$ (Algoritmo binário)

$\ln n = log_e n$ (Algoritmo natural)

$lg^k n = (\lg n)^k$ (Exponenciação)

$\lg \lg n = \lg(\lg n)$ (Composição)