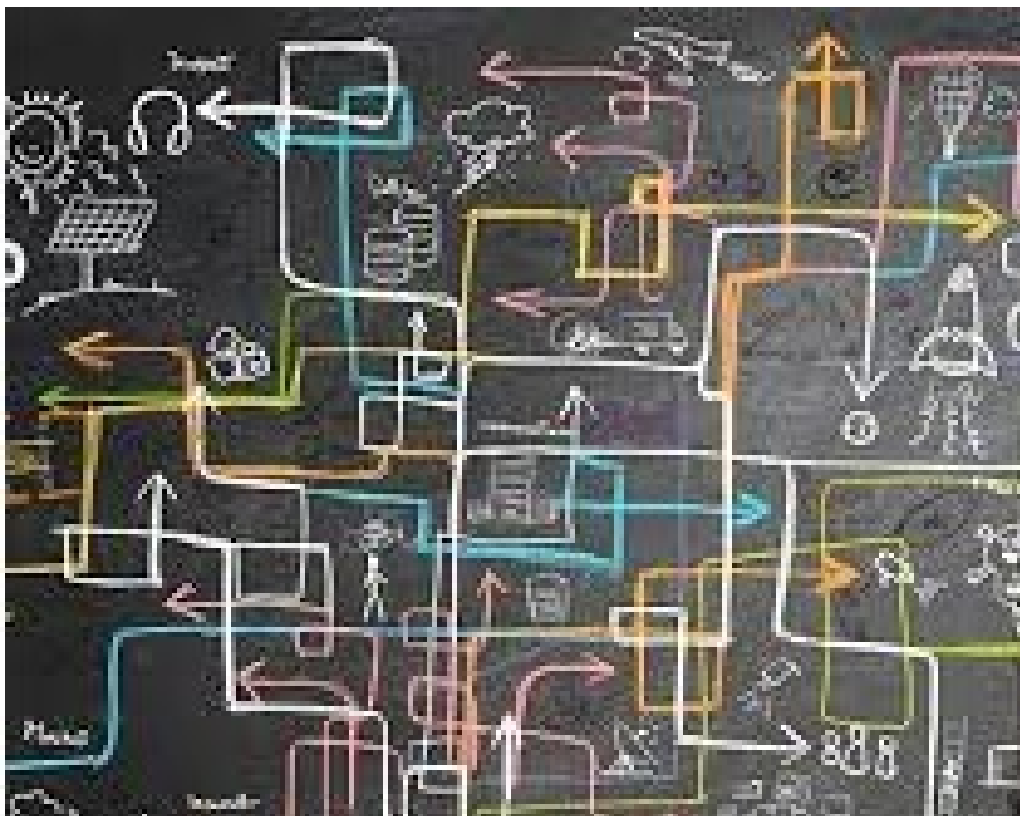


ANÁLISE DE ALGORITMOS



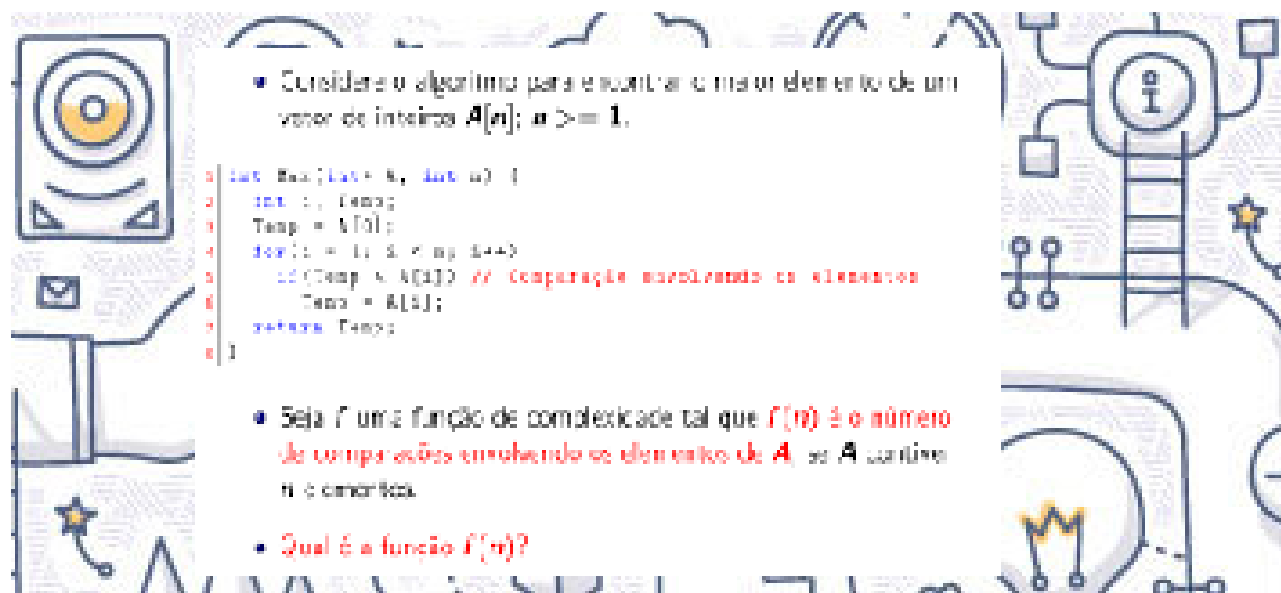


Considerações
Iniciais.

Analisar um algoritmo significa prever os recursos de que o algoritmo necessita.

Ocasionalmente, recursos como memória, largura de banda de comunicação ou hardware de computadores são a principal preocupação.

Porém, frequentemente é o tempo de computação que desejamos medir.



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

```
1 int Max(int n, int a) {
2     int i, Temp;
3     Temp = A[0];
4     for(i = 1; i < n; i++)
5         if(Temp < A[i]) // Comparação envolvendo os elementos
6             Temp = A[i];
7     return Temp;
8 }
```

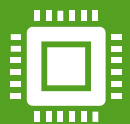
- Seja f uma função de complexidade tal que $f(n)$ é o número de comparações envolvendo os elementos de A , se A contém n elementos.
- Qual é a função $f(n)$?

Antes de analisar um algoritmo, devemos ter um modelo da tecnologia de implementação que será usada, inclusive um modelo para recursos da tecnologia e seus custos.



Consideraremos um modelo de computação genérico de máquina de acesso aleatório (random-access machine, RAM) com um único processador como nossa tecnologia de implementação e entendemos que nossos algoritmos serão implementados como programas de computador.

No modelo RAM, as instruções são executadas uma após outra, sem operações recorrentes.

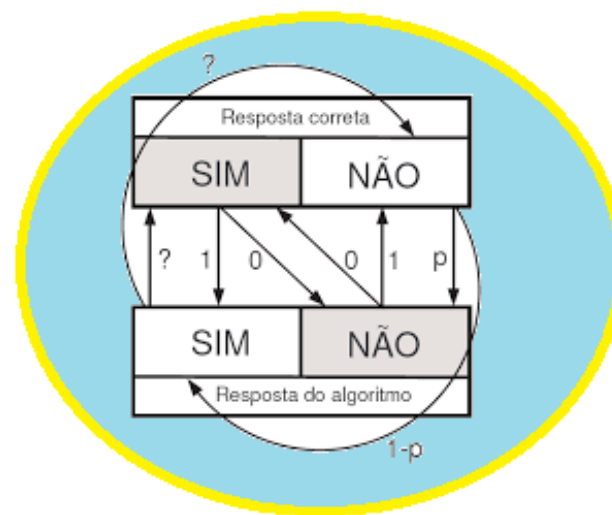


Até mesmo a análise de um algoritmo simples no modelo RAM pode ser um desafio. As ferramentas matemáticas exigidas podem incluir análise combinatória, álgebra, teoria das probabilidades, a capacidade de interpretar os termos mais significativos em fórmula.



Tendo em vista que um algoritmo pode ser diferente para cada entrada possível, precisamos de um meio para resumir esse comportamento em fórmulas mais simples, de fácil compreensão.

Entendemos que um algoritmo resolve um problema quando para qualquer entrada produz uma resposta correta.



Mesmo resolvendo um problema, um algoritmo pode ser não aceitável na prática por requerer muito espaço e tempo.

```
( ) #include <stdio.h>
#include <conio.h>
void main()
{
    int n1, n2;
    float c1,c2, cf;
    printf("n1 = "); scanf ("%d", &n1);
    printf("n2 = "); scanf ("%d", &n2);

    c2 = (n1 + n2)/100;cf= c1 + c2;
    printf("\n cf = %7.2f", cf);
    getch();
}
```


Observação.

Um problema é considerado intratável, se não existe um algoritmo para ele cuja a demanda de recursos computacionais seja razoável.

Algumas perguntas surgem.

- ▶ O problema em questão é tratável?
- ▶ Existe um algoritmo que demande quantidade razoável de recursos computacionais?
- ▶ Quais os custos deste algoritmo?
- ▶ Existe um algoritmo melhor?
- ▶ Como comparar algoritmos?

Observação.

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.



Vamos agora fazer um exercício
referente a um algoritmo simples.



Exercício.

Determine a soma: $1!+2!+3!+4!+\dots+n!$

Considere n aumentando.

Ou seja faça a soma para :

Para $n = 0$

Para $n = 1$

Para $n = 2$

·

·

·

Para $n = 10$

·

·

·

Para $n = 20$

·

·

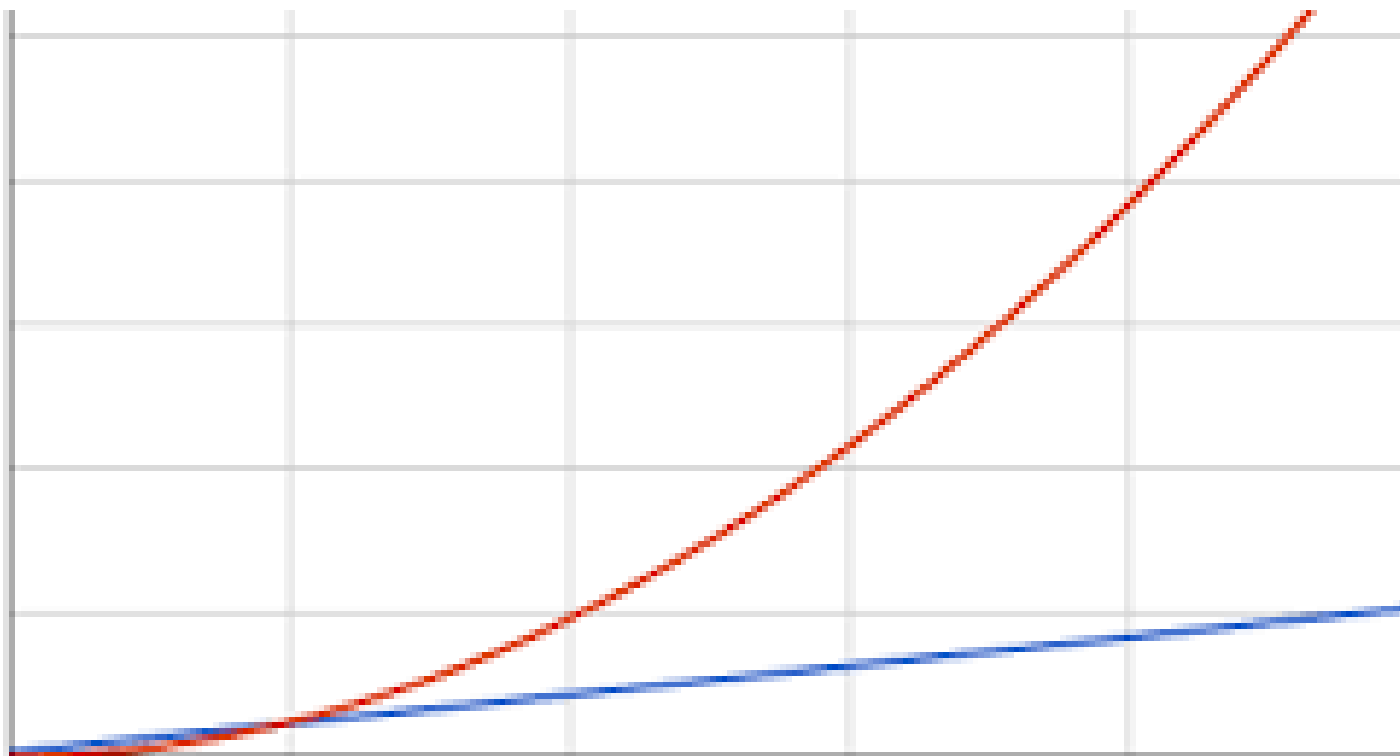
·

Solução

n	Fatorial	Soma
0	1	1
1	1	2
2	2	4
3	6	10
4	24	34
5	120	154
6	720	874
7	5040	5914
8	40320	46234
9	362880	409114
10	3628800	4037914
11	39916800	43954714
12	479001600	522956314
13	6227020800	6749977114
14	87178291200	93928268314
15	1307674368000	1401602636314
16	20922789888000	22324392524314
17	355687428096000	378011820620314
18	6402373705728000	6780385526348310
19	121645100408832000	128425485935180000
20	2432902008176640000	2561327494111820000

O que observamos neste algoritmo simples?

Se N crescer muito o tempo para calcular irá aumentar conforme N aumenta.
O resultado da operação irá tender para números inteiros extremamente grandes.



Custo de um Algoritmo

O custo final de um algoritmo pode estar relacionado a diversos fatores:

- 1 – Tempo de execução;
- 2 – Utilização da memória principal;
- 3 - Consumo de energia, etc.

4 - Custo de implementação;

5 - Portabilidade;

6- Extensibilidade.

Obs 2: extensibilidade pode se referir à capacidade de um sistema de aumentar sua capacidade de computação sob carga aumentada quando recursos (geralmente hardware) são adicionados.

Observação

Medidas importantes em outros contextos.

1 - Legibilidade do código;

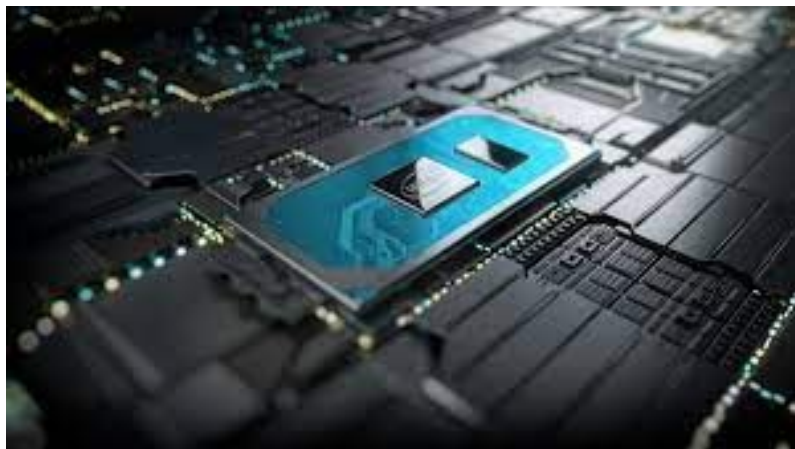
Obs1: Entende-se por **legibilidade**: um atributo manual que os desenvolvedores individuais devem verificar durante todo o seu **código**. A capacidade de gravar **código** que seja fácil de ler deve ser parte da responsabilidade cultural dos desenvolvedores individuais, ampliando seu conjunto de habilidades.

Exemplo.

Solução de um sistema de equações lineares $AX = 0$.

Existem vários métodos para solucionar o sistema vamos pensar em dois:

- 1 – Método de Cramer (Determinante).
- 2 – Método de Gauss (Escalonamento).



Exemplo da
medição de tempo
resolvendo em um
processador
rodando em 3GHz.

Estimativa empírica de tempo.

n	Cramer	Gauss
2	4 ns	2ns
3	12 ns	8 ns
4	48 ns	20 ns
5	240 ns	40 ns
10	7,3 ms	330 ns
20	152 anos	2,7 ms

Outro exemplo

Busca sequencial.

Dado um vetor de números, verificar se um número chave encontra-se neste vetor.

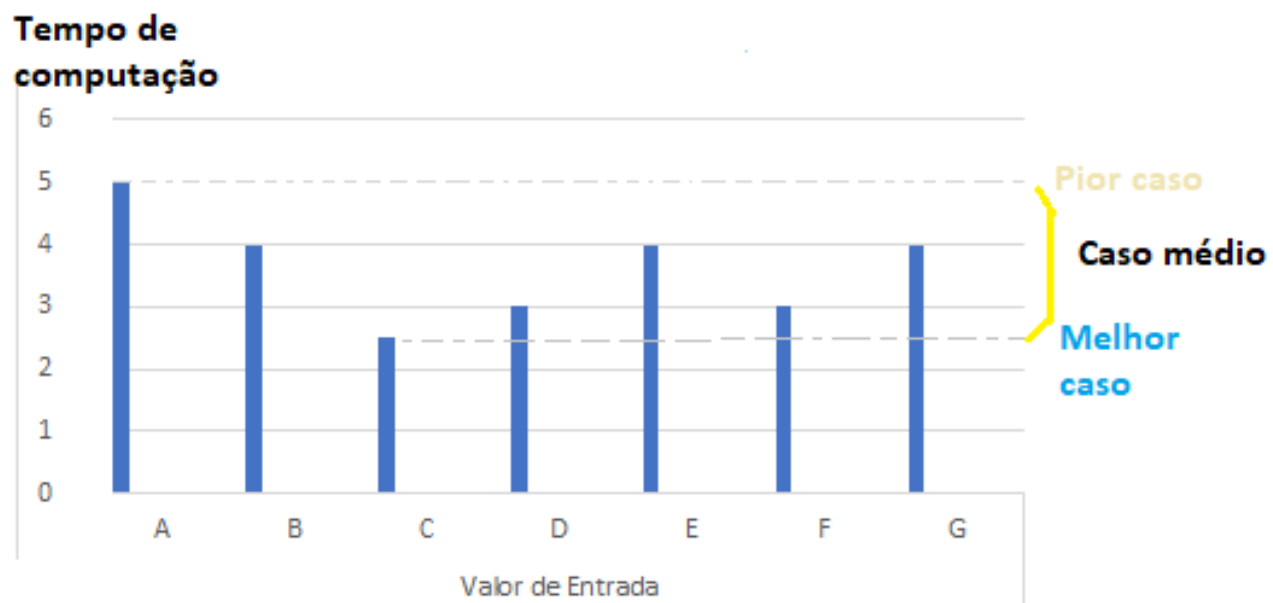
```
char achou = 0;  
i = 0;  
while (!achou && i < n) {  
    achou = (vet[i] == chave);  
    i++;  
}  
if (achou) return (i);  
else return (1)
```

Neste caso de busca sequencial:

- ❑ Não existe um número fixo de operações;
- ❑ Isso vai depender de onde o número chave é encontrado;
- ❑ Então é comum de realizar a contagem para;
 - 1 - Para o melhor caso;
 - 2 - O pior caso;
 - 3 - Para o caso médio.

- Um algoritmo pode rodar mais rápido para certos conjunto de dados do que para outros;
- Determinar um caso médio pode ser muito complicado, assim os algoritmos são geralmente medidos pela complexidade de tempo no pior caso.

Para algumas áreas de aplicação (controle do tráfego aéreo, cirurgias, etc) o conhecimento do pior caso é crucial.



Análise de Complexidade de Tempo.

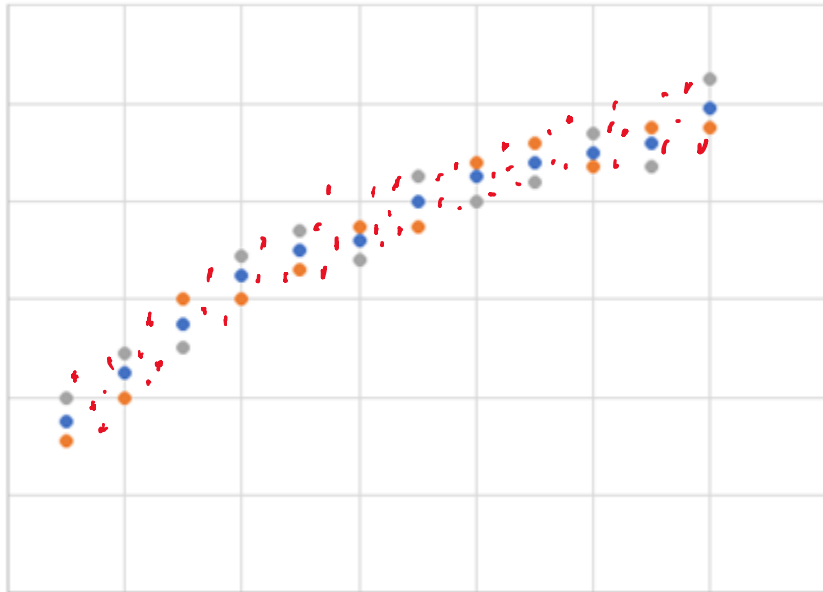
- a. Análise Experimental;
- b. Análise Teórica.

► A medição empírica.

Pode ser medida com funções de cálculo de tempo.

1. Construímos um programa que implemente o algoritmo;
2. Execute o programa com conjunto de dados vários tamanhos e composições;
3. Use um método para medir o tempo de execução com exatidão;
4. Os dados devem ser parecidos com o gráfico abaixo.

$t(\text{ms})$



Entrada

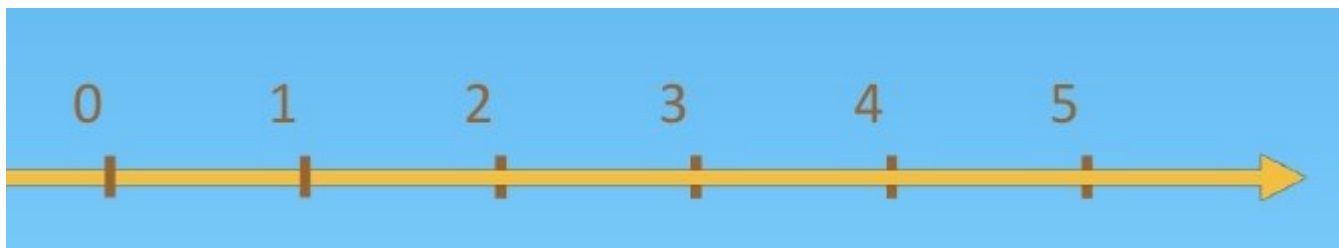
Estudos experimentais possuem várias limitações:

1. É preciso implementar e testar o algoritmo para determinar seu tempo de execução;
2. Os experimentos só podem ser feitos para um conjunto limitado de dados de entrada, assim os tempos de computação resultantes podem não ser indicativos dos tempos de execução para entradas não incluídas no experimento;
3. Para comparar dois algoritmos , devem ser utilizados o mesmo ambiente de hardware e software.

Vamos agora introduzir uma avaliação teórica para medir a eficiência do algoritmo.

O problema de ordenação.

Vamos imaginar que necessitamos ordenar uma sequência de números em ordem não decrescente.



Definindo formalmente o problema de ordenação.



Entrada: Uma sequência de números $(a_1, a_2, a_3, \dots, a_n)$

Saída: Uma permutação (reordenação) $(a'_1, a'_2, a'_3, \dots, a'_n)$

Sequência de saída tal que $(a'_1 \leq a'_2 \leq a'_3 \leq \dots \leq a'_n)$

Exemplo.

Dado uma sequência {31, 41, 59, 26, 41, 58}, um algoritmo de ordenação devolve como saída a sequência {26, 31, 41, 41, 58, 59}.

Análise de Complexidade

- ▶ A análise de complexidade procura estimar a velocidade do algoritmo. Prever o tempo que o algoritmo vai consumir. Verificar o custo que o algoritmo vai consumir para resolver o problema.

Neste aspecto, encontram-se duas dificuldades:

1. O tempo gasto, depende dos dados de entrada;
2. O tempo também depende da capacidade de hardware do computador utilizado

Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou função de complexidade, f . $f(n)$ é a medida de tempo necessário para executar um algoritmo para um problema de tamanho n .

Para entender melhor vamos ver o trecho do código abaixo.

```
int soma = 0;  
for (i=0; i<n; i++)  
    soma = soma + vet[i]
```

```
int soma = 0;  
for (i=0; i<n; i++)  
    soma = soma + vet[i]
```

O custo para execução deste algoritmo ou a função complexidade é $f(n)=n$. Pois o laço executa de acordo com o tamanho de n . Se $n = 10$, logo o laço executará 10 vezes, se for igual a 100 executará 100 vezes.

Podemos tornar o algoritmo mais eficiente fazendo a seguinte modificação:

```
int soma = vet[0]
for [ i=1; i<n; i++]
    soma = soma + vet[i]
```


Assim a função complexidade é:
 $f(n) = n-1$

Pois i inicia em 1, assim se $n=10$ implica que o laço fará o processo 9 vezes.

Os dois trechos de código resolvem o problema, somar os elementos do vetor. Porém, cada um tem um custo de execução diferente do outro;

A partir da análise, podemos definir qual é o algoritmo mais eficiente para resolver este problema específico.

Melhor Caso, Pior Caso, Caso Médio

Na análise da complexidade, definimos para um algoritmo, o melhor caso, o pior caso e o caso médio, como forma de mensurar o custo do algoritmo de resolver determinado problema diante de diferentes entradas.

- O melhor caso: é quando o algoritmo representa o 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: é a média dos tempos de execução de todas as entradas de tamanho n .

Desta forma, considere o problema de acessar um determinado registro de um vetor inteiro, cada registro contém um índice único que é utilizado para acessar o elemento do vetor.

O problema: Dada uma chave qualquer, localize o registro que contenha esta chave. O algoritmo de pesquisa mais simples é o que faz a pesquisa sequencial.

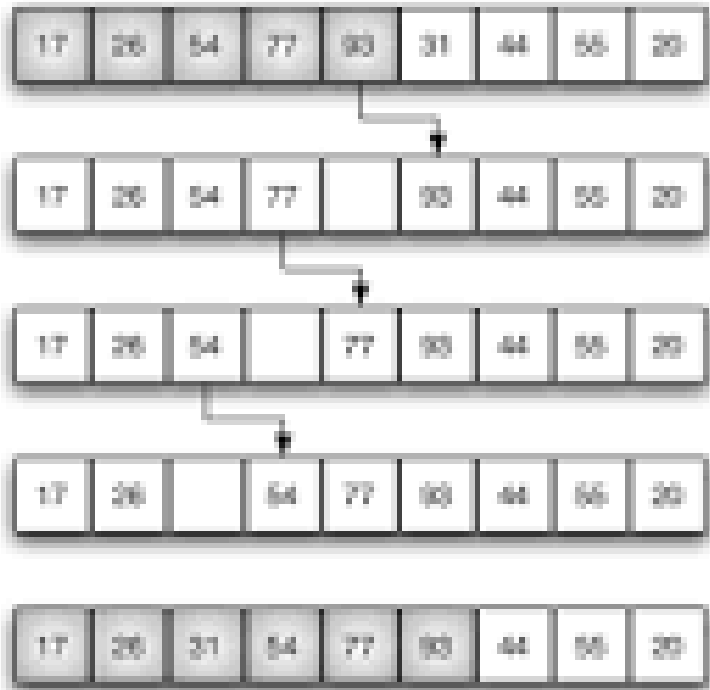
Então,

Seja f uma função complexidade tal que $f(n)$ é o número de registros consultados do arquivo, temos:

1. Melhor caso: $f(n) = 1$ (Registro procurado é o primeiro encontrado).
2. Pior Caso: $f(n) = n$ (Registro procurado é o último consultado ou não está presente no arquivo)
3. Caso médio: $f(n) = \frac{(n+1)}{2}$ (A média aritmética entre o melhor e o pior caso).

Observação:

Além de estudar o tempo, a análise de algoritmos estuda certos paradigmas como: Divisão e conquista; programação dinâmica; gula; Busca local, aproximação, entre outros que se mostraram úteis na criação de algoritmos para vários problemas computacionais.



Análise da ordenação por inserção

Ordenação por Inserção

Entrada: Uma sequência de n números $(a_1, a_2, a_3, \dots, a_n)$

Saída: Uma permutação $(a'_1, a'_2, a'_3, \dots, a'_n)$ da sequência de entrada, tal que
$$a'_1 \leq a'_2 \leq a'_3 \leq \dots \leq a'_n$$

O tempo despendido pelo procedimento Insertion-Sort depende da entrada: ordenar mil números demora mais que ordenar três números. Além disso, Insertion-Sort pode demorar quantidades de tempos diferentes para ordenar duas sequências de entrada do mesmo tamanho, dependendo do quanto elas já estejam ordenadas.

Em geral, o tempo gasto por um algoritmo cresce com o tamanho da entrada; assim, é tradicional descrever o tempo de execução de um programa em função do tamanho da sua entrada. Para isso, precisamos definir os termos “ tempo de execução” e “ tamanho da entrada” com mais cuidado.

A melhor noção para o **tamanho da entrada** depende do problema que está sendo estudado.

No caso de muitos problemas, como ordenação ou cálculo de transformações discretas de Fourier, a medida mais natural é o número de itens na entrada.

Por exemplo, o tamanho n do arranjo de ordenação.

Para muitos outros problemas, como a multiplicação de dois inteiros, a melhor medida do tamanho da entrada é o número total de bits necessários para representar a entrada em notação binária comum.

As vezes, é mais apropriado descrever o tamanho da entrada com dois números em vez de um. Por exemplo, se a entrada de um algoritmo é grafo, o tamanho da entrada pode ser descrito pelos números dos vértices e arestas do grafo.

O **tempo de execução** de um algoritmo em determinada entrada é o número de operações primitivas ou “passos” executados.

É conveniente definir a noção de passo de modo que ela seja tão independente de máquina quanto possível.

Vamos adotar a visão abaixo:

Uma quantidade de tempo constante é exigida para executar cada linha de um pseudo código. Uma linha pode demorar uma quantidade diferente de outra linha, mas consideraremos que cada execução da i -ésima linha leva um tempo c_i , onde c_i é uma constante.

Esse ponto de vista está de acordo com o modelo RAM e também reflete o modo como o pseudo código seria implementado na maioria dos computadores reais.

Vamos dar início a discussão do Insertion-Sort.

Código

Insertion-Sort (A)

For $j = 2$ to A comprimento

 Chave = $A[j]$

 // inserir $A[j]$ na seq. Ord. $A[1 \dots j-1]$

$i = j - 1$

 while $i > 0$ e $A[i] > \text{chave}$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = \text{chave}$

- ▶ O tempo de execução do algoritmo é a soma dos tempos de execução para cada instrução executada;
- ▶ Uma instrução que demanda C_i passos para ser executada e é executada n vezes contribuirá com $C_i n$ para o tempo de execução total;
- ▶ Então para calcular $T(n)$ o tempo de execução de Insertion-Sort, de uma entrada de n valores, somamos os produtos das colunas dos custos e multiplicamos por n .

obtemos:

$$T(n) = C_1n + C_2(n - 1) + C_4(n - 1) + C_5 \sum_{j=2}^n t_j + C_6 \sum_{j=2}^n (t_j - 1) + \\ C_7 \sum_{j=2}^n (t_j - 1) + C_8(n - 1)$$

Analizando o Insertion-Sort

Onde; tiramos do INSERTION-SORT ABAIXO.

INSERTION SORT (A)

```
1  For j = 2 to A comprimento
2      Chave = A[j]
3      // inserir A[j] na seq. Ord. A [1... j-1]
4      i = j - 1
5      while i > 0 e A[i] > chave
6          A [ i + 1] = A[i]
7          i = i - 1
8      A [ i + 1] = chave
```

INSERTION SORT (A)

CUSTO

```
1  For j = 2 to A comprimento
2      Chave = A[j]
3      // inserir A[j] na seq. Ord. A [1... j-1]
4      i = j - 1
5      while i > 0 e A[i] > chave
6          A [ i + 1] = A[i]
7          i = i - 1
8      A [ i + 1] = chave
```

INSERTION SORT (A)

```
1  For j = 2 to A comprimento
2      Chave = A[j]
3      // inserir A[j] na seq. Ord. A [1... j-1]
4      i = j - 1
5      while i > 0 e A[i] > chave
6          A [ i + 1] = A[i]
7          i = i - 1
8      A [ i + 1] = chave
```

CUSTO

C_1

C_2

C_3

C_4

C_5

C_6

C_7

C_8

→ LINHA DE
COMENTÁRIO
SEM CUSTO.

INSERTION SORT (A)

```
1 For j = 2 to A comprimento
2     Chave = A[j]
3     // inserir A[j] na seq. Ord. A [1... j-1]
4     i = j - 1
5     while i > 0 e A[i] > chave
6         A [ i + 1] = A[i]
7         i = i - 1
8     A [ i + 1] = chave
```

CUSTO

C_1

C_2

C_3

$C_{i,j}$

C_5

C_6

C_7

C_8

VEZES

INSERTION SORT (A)

```
1  For j = 2 to A comprimento
2      Chave = A[j]
3      // inserir A[j] na seq. Ord. A [1... j-1]
4      i = j - 1
5      while i > 0 e A[i] > chave
6          A [ i + 1] = A[i]
7          i = i - 1
8      A [ i + 1] = chave
```

CUSTO

C_1

C_2

C_3

C_4

C_5

C_6

C_7

C_8

VEZES

n

$n-1$

$n-1$

$n-1$

$\sum_{j=2}^n j$

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

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

$n-1$

INSERTION SORT (A)

1 For j = 2 to A comprimento

2 Chave = A[j]

3 // inserir A[j] na seq. Ord. A [1... j⁻¹]

4 i = j - 1

5 while i > 0 e A[i] > chave

6 A[i + 1] = A[i]

7 i = i - 1

8 A[i + 1] = chave

CUSTO

C₁

C₂

C₃

C₄

C₅

C₆

C₇

C₈

VEZES

n

$n-1$

$n-1$

$n-1$

$\sum_{j=2}^m t_j$

$\sum_{j=2}^m (t_j - 1)$

$\sum_{j=2}^m (t_j - 1)$

$n-1$

Obs.

m = QUANT. de vezes

Linha de comentário
NÃO TEM CUSTO

t_j = nº de vezes
que LAÇO WHILE

O tempo de execução de um algoritmo depende de qual entrada é dada. (Além do tamanho de entrada naturalmente).

Por exemplo,

No Insertion-Sort o **melhor caso de entrada** seria o arranjo já está ordenado.

Logo, para cada $j = 2, 3, \dots, n$, $A[i] \leq \text{chave}$, na linha 5 quando i tem seu valor $j-1$,

Portanto $t_j = 1$ para $j = 2, 3, \dots, n$, onde o tempo de execução fica:

$$T_{(n)} = C1n + C2(n-1) + C4(n-1) + C5(n-1) + C8(n-1)$$

O custo C6 e C7 ele não faz, pois o vetor está alinhado

Logo teremos:

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8) \cdot n - (c_2 + c_4 + c_5 + c_8)$$

Considerando que C_1 , C_2 , C_4 , C_5 e C_8 são constantes percebemos que estamos diante de uma função linear, em função de n .

$$T(n) = an + b$$

Agora se estivermos no PIOR CASO.

O arranjo na ordem inversa, ou seja, ordem crescente.

Devemos comparar cada elemento $A[j]$ com cada elemento do subarranjo ordenado inteiro, $A[1, 2, \dots, j-1]$, e então $t_j = j$ para $2, 3, \dots, n$.

Observando que:

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

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

*Aqui temos a condição **While** quantas vezes
ele é executado para ordenar o arranjo.*

Exemplo.

A=[4,3,2,1]

3,4,2,1

3,2, 4, 1

3, 2 1, 4

2, 3, 1, 4

2, 1, 3, 4

2, 1, 3, 4

1, 2, 3, 4

1,2, 3, 4

$$\frac{n(n+1)}{2} - 1$$

$$\frac{4(5)}{2} - 1$$

$$\boxed{9}$$

$$T_{(n)} = C1n + C2(n - 1) + C4(n - 1) + C5 \left(\frac{n(n + 1)}{2} - 1 \right) + \dots$$

Assim no pior tempo do insertion-sort é:

Ficando:
$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n + (c_2 + c_4 + c_5 + c_8)$$

Logo considerando $C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8$ como constantes temos uma função quadrática.

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

Observação

Em nossa análise da ordenação por inserção, examinamos tanto o melhor caso quanto o pior caso.

Normalmente estamos interessados na procura do pior caso pois estabelece um limite superior para o tempo de execução para qualquer entrada.

Conhecê-lo nos dá a garantia de que um algoritmo nunca demorará mais do que esse tempo.

Temos três razões para trabalharmos com o pior caso.

1 - O tempo de execução do pior caso de um algoritmo estabelece um limite superior para o tempo de execução para qualquer entrada.

Conhecê-lo nos dá uma garantia de que o algoritmo nunca demorará mais que esse tempo.

2 - Para alguns algoritmos, o pior caso ocorre com bastante frequência. Por exemplo, na pesquisa de banco de dados em busca de determinada informação, o pior caso do algoritmos de busca frequentemente ocorre quando a informação não está presente no banco de dados.

3 - Muitas vezes o caso médio é quase tão ruim quanto o pior caso.

Ordem de Crescimento

Usamos em nossa análise algumas abstrações simplificadoras para facilitar o procedimento Insertion-Sort.

Primeiro ignoramos o custo real de cada instrução, usando constantes C_1 , C_2 , C_3 , ..., para representar os custos.

Assim quando determinamos o tempo de execução do pior caso $T(n) = an^2 + bn + c$ acabamos ignorando os custos reais de instrução mas também os custos abstratos.

Agora faremos mais uma abstração simplificadora. É a taxa de crescimento ou ordem de crescimento, do tempo de execução que realmente interessa.

Portanto, consideraremos apenas o termo inicial da fórmula (por exemplo, an^2), já que os termos de ordem mais baixa são relativamente insignificantes para grandes valores de n .

Também ignoramos o coeficiente constante no termo geral inicial, visto que fatores constantes são menos significativos que a taxa de ordenação de crescimento na determinação da eficiência computacional para grandes entradas. No caso do Insertion-Sort acabamos ficando apenas com o fator n^2 .

Assim afirmamos que a ordenação por inserção de
execução do pior caso igual a $O(n^2)$ (Lemos com “
O de n ao quadrado”).

Exercício.

Considere a ordenação de n números armazenados no arranjo A , localizando primeiro o menor elemento de A e permutando esse elemento com o elemento contido em $A[1]$. Em seguida, determine o segundo menor elemento de A e permuta-o com $A[2]$. Continue dessa maneira para os primeiros $n-1$ elementos de A .

- a) Qual algoritmo que apresenta este comportamento?
- b) Qual a função que representa este algoritmo?

Resposta

Ordenação por seleção - Selection Sort.

A ordenação por seleção (do inglês, *selection sort*) é um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os elementos restantes, até os últimos dois.

PORTUGOL

```
programa
{
    inteiro cont, contA, contB, aux
    inteiro vet[3]

    funcao inicio()
    {
        // Preencher o array
        para(cont = 0; cont <= 2; cont++) {
            escreva("Digite um número: ")
            leia(vet[cont])
        }

        para (contA = 0; contA <= 2; contA++) {
            para(contB = contA + 1; contB <= 2; contB++) {
                se(vet[contA] > vet[contB] {
                    aux = vet[contB]
                    vet[contB] = vet[contA]
                    vet[contA] = aux
                }
            }
        }

        para(cont = 0; cont <= 2; cont++) {
            escreva(vet[cont] + " ")
        }
    }
}
```

```

programa
{
    inteiro cont, contA, contB, aux
    inteiro vet[3]

    funcao inicio()
    {
        // Preencher o array
        para(cont = 0; cont <= 2; cont++) {
            escreva("Digite um número: ")
            leia(vet[cont])
        }

        para (contA = 0; contA <= 2; contA++) {
            para(contB = contA + 1; contB <= 2; contB++) {
                se(vet[contA] > vet[contB]) {
                    aux = vet[contB]
                    vet[contB] = vet[contA]
                    vet[contA] = aux
                }
            }
        }

        para(cont = 0; cont <= 2; cont++) {
            escreva(vet[cont] + " ")
        }
    }
}

```

CUSTO

VEZES

0
C1
C2
C3

1
1
1
1

C4
C5
C6

n
n.n
n

C7
C8

1
1

Realizando uma análise similar a que realizamos ao Insertion-Sort

Percebemos que a complexidade deste algoritmo é $O(n^2)$ (O de n ao quadrado)