Tempo de Processamento
Projeto e Análise de Algoritmos

Daniel de Oliveira Capanema
Adaptado: Anna Izabel João Tostes Ribeiro

Universidade de Itaúna

2021

- O projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos.
- 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.
- Muitos desses algoritmos são encontrados em áreas como pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras.
- Tempo e espaço tende a se comportar de forma antagônica, de modo que devemos buscar uma <u>solução de compromisso</u>.
- Soluções de compromisso buscam o equilíbrio entre recursos ou entre recursos e a precisão ou qualidade dos resultados.

A UIT 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).

- Qual é a estrutura de dados mais indicada para esse problema?

A UIT 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 256×10^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.

A UIT 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).

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

- O projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos.
- 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.
- Muitos desses algoritmos são encontrados em áreas como pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras.

1. Análise de um algoritmo particular

2. Análise de uma classe de algoritmos

- 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)

- Análise de um algoritmo particular
  - 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;

- Análise de um algoritmo particular
  - **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.

- Análise de um algoritmo particular

  - 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.

- 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.

- Análise de uma classe de algoritmos
  - 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.

- 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, 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

- Tais medidas 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.
- Apesar disso, há argumentos a favor de se obterem medidas reais de tempo.
  - Ex.: quando há vários algoritmos distintos para resolver um mesmo tipo de problema, todos com um custo de execução dentro de uma mesma ordem de grandeza.
  - Assim, são considerados tanto os custos reais das operações como os custos não aparentes, tais como alocação de memória, indexação, carga, dentre outros.

- Usa um modelo matemático baseado em um computador idealizado.
- Deve ser especificado o conjunto de operações e seus custos de execuções.
- É mais usual ignorar o custo de algumas das operações e considerar apenas as operações mais significativas.
- 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.

- Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou função de complexidade T
- T(n) é a medida do tempo necessário para executar um algoritmo para um problema de tamanho n
- Função de complexidade de tempo: T(n) mede o tempo necessário para executar um algoritmo para um problema de tamanho n
- Função de complexidade de espaço: T(n) mede a memória necessária para executar um algoritmo para um problema de tamanho n
- Utilizaremos T para denotar uma função de complexidade de tempo daqui para a frente
- Na realidade, a complexidade de tempo não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada

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

```
function Max (var A: Vetor): integer;
var i, Temp: integer;
begin
    Temp := A[1];
    for i:=2 to n do if Temp < A[i] then Temp := A[i];
    Max := Temp;
end;
```

- Seja T uma função de complexidade tal que T(n) seja o número de comparações entre os elementos de A, se A contiver n elementos
- Logo $T(n) = n - 1$ para $n \geq 1$
- Vamos provar que o algoritmo apresentado no programa acima é ótimo

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

- **Prova**: Cada um dos $n-1$ elementos tem de ser mostrado, por meio de comparações, que é menor que algum outro elemento

- Logo $n-1$ comparações são necessárias

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

- A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada de 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
- No caso da função `Max` do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho n
- Já para um algoritmo de ordenação isso não ocorre: se os dados de entrada já estiverem quase ordenados, então o algoritmo pode ter que trabalhar menos

- 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 T é uma função de complexidade baseada na análise de pior caso, o custo de aplicar o algoritmo nunca é maior do que $\mathrm{T}(n)$
- **Caso médio** (ou caso esperado):
  - Média dos tempos de execução de todas as entradas de tamanho n

- Na análise do caso esperado, supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho n 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

- Considere o problema de acessar os registros de um arquivo
- Cada registro contém uma chave única que é utilizada para recuperar registros do arquivo
- **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

- 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.

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

- 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.*

- Seja T uma função de complexidade tal que $\mathrm{T}(n)$ é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro)
  - Melhor caso: $\mathrm{T}(n) = 1$ (registro procurado é o primeiro consultado)
  - Pior caso: $\mathrm{T}(n) = n$ (registro procurado é o último consultado ou não está presente no arquivo)
  - Caso médio: $\mathrm{T}(n) = \dfrac{(n+1)}{2}$

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

$$\mathrm{T}(n) = (1 \times p_1) + (2 \times p_2) + (3 \times p_3) + \cdots + (n \times p_n)$$

- Para calcular $\mathrm{T}(n)$ basta conhecer a distribuição de probabilidades p_{i}
- Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então $p_i = 1/n, 1 \le i \le n$
- Neste caso, $\mathrm{T}(n) = \frac{1}{n}(1 + 2 + 3 + \cdots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$
- A análise do caso esperado revela que uma pesquisa com sucesso examina aproximadamente metade dos registros

2. No problema de acessar os registros de um arquivo, seja T uma função de complexidade tal que $\mathrm{T}(n)$ é o número de registros consultados no arquivo. Seja q a probabilidade de que uma pesquisa seja realizada com sucesso (chave procurada se encontra no arquivo) e $(1-q)$ a probabilidade da pesquisa sem sucesso (chave procurada não se encontra no arquivo). Considere também que nas pesquisas com sucesso todos os registros são igualmente prováveis. Encontre a função de complexidade para o caso médio.

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

```
procedure MaxMin1 (var A: Vetor; var Max, Min: integer);
var i: integer;
begin
  Max := A[1];  Min := A[1];
  for i := 2 to n do
    begin
      if A[i] > Max then Max := A[i];  {Testa se A[i] contém o maior elemento}
      if A[i] < Min then Min := A[i];  {Testa se A[i] contém o menor elemento}
    end;
end;
```

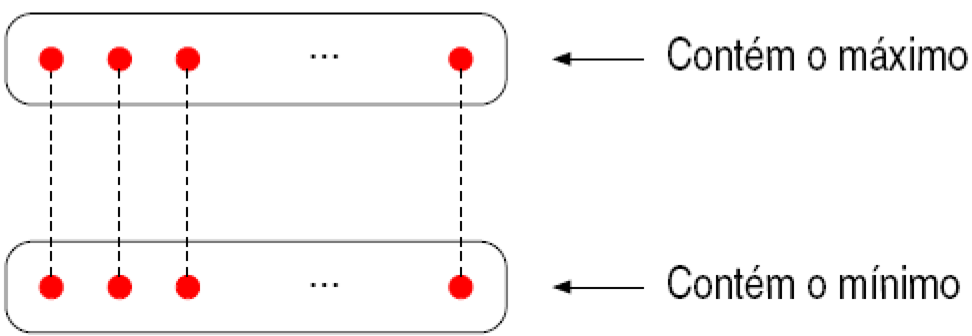
- Seja T(n) o número de comparações entre os elementos de A, se A tiver n elementos
- Logo $T(n) = 2(n - 1)$, para $n > 0$, para o melhor caso, pior caso e caso médio.

- MaxMin1 pode ser facilmente melhorado:
  - A comparação A[i] < Min só é necessária quando o resultado da comparação A[i] > Max for falso

```
procedure MaxMin2 (var A: Vetor; var Max, Min: integer);
var i: integer;
begin
  Max := A[1];  Min := A[1];
  for i := 2 to n do
    if A[i] > Max
    then Max := A[i]
    else if A[i] < Min then Min := A[i];
end;
```

- Para a nova implementação temos:
  - Melhor caso: $T(n) = n - 1$ (quando os elementos estão em ordem crescente)
  - Pior caso: $T(n) = 2(n - 1)$ (quando os elementos estão em ordem decrescente)
  - Caso médio: $T(n) = \frac{3n}{2} - \frac{3}{2}$
- Caso médio:
  - A[i] é maior do que Max a metade das vezes
  - Logo, $T(n) = n - 1 + \frac{n-1}{2} = \frac{3n}{2} - \frac{3}{2}$, para $n > 0$

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



```
procedure MaxMin3(var A: Vetor;
                  var Max, Min: integer);
var i,
    FimDoAnel: integer;
begin
  {Garante uma qte par de elementos no vetor para evitar caso de exceção}
  if (n mod 2) > 0
  then begin
         A[n+1]    := A[n];
         FimDoAnel := n;
       end
  else FimDoAnel := n-1;

  {Determina maior e menor elementos iniciais}
  if A[1] > A[2]
  then begin
         Max := A[1]; Min := A[2];
       end
  else begin
         Max := A[2]; Min := A[1];
       end;
```

```
i:= 3;
while i <= FimDoAnel do
  begin
  {Compara os elementos aos pares}
  if A[i] > A[i+1]
  then begin
         if A[i] > Max then   Max := A[i];
         if A[i+1] < Min then Min := A[i+1];
       end
  else begin
         if A[i] < Min then   Min := A[i];
         if A[i+1] > Max then Max := A[i+1];
       end;
  i:= i + 2;
  end;
end;
```

- 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`

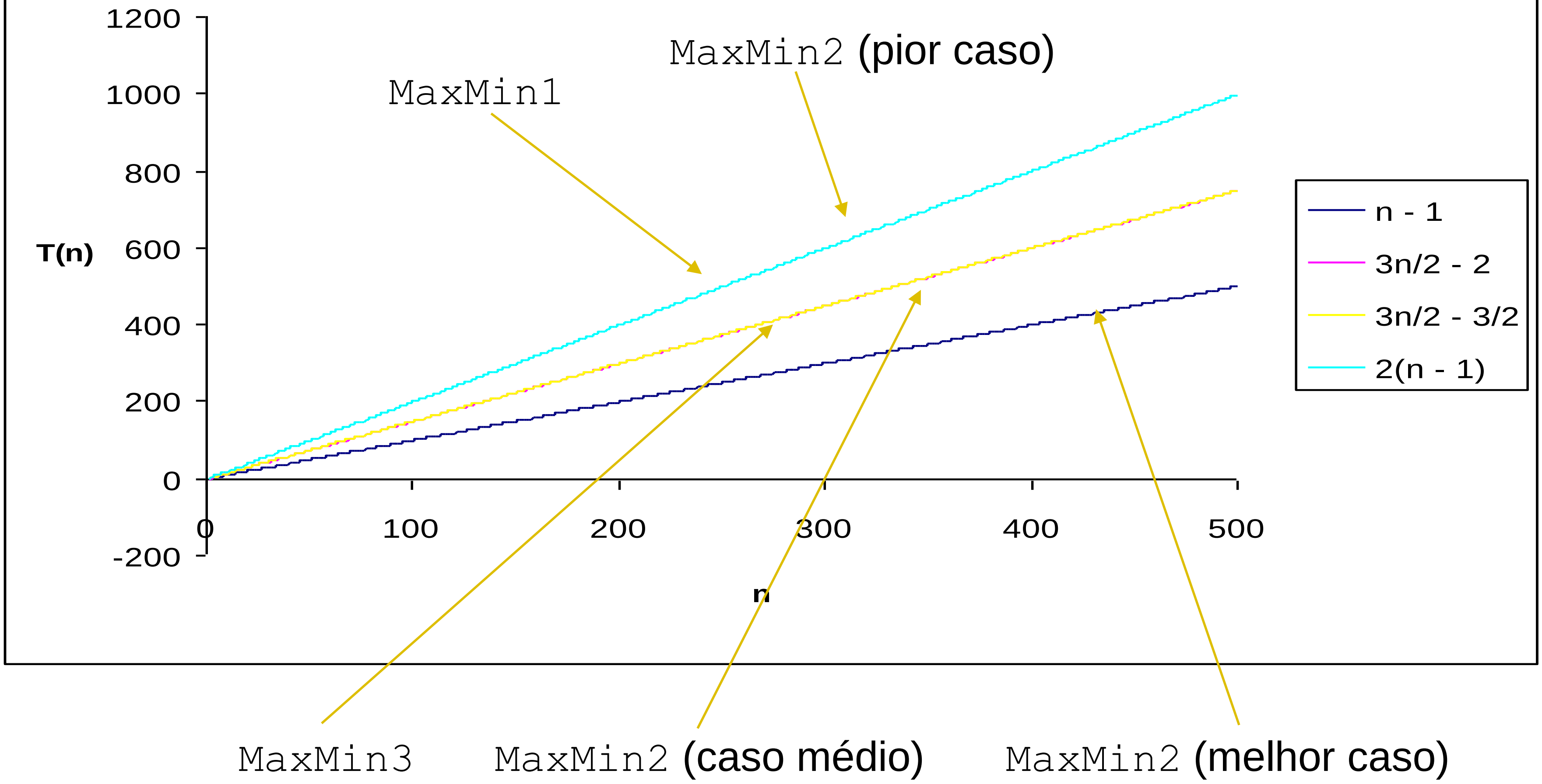
- Quando n é ímpar, o elemento que está na posição `A[n]` é duplicado na posição `A[n+1]` para evitar um tratamento de exceção

- Para esta implementação,
  para $n > 0$, para o melhor caso, pior caso e caso médio

$$T(n) = \frac{n}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2$$

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

Os Três Algoritmos	T(n)		
	Melhor Caso	Pior Caso	Caso Médio
MaxMin1	2(n-1)	2(n-1)	2(n-1)
MaxMin2	N-1	2(n-1)	3n/2-3/2
MaxMin3	3n/2-2	3n/2-2	3n/2-2


MaxMin2 (pior caso)
MaxMin1
T(n)
1200
1000
800
600
400
200
0
-200
0
100
200
300
400
500
n
n - 1
3n/2 - 2
3n/2 - 3/2
2(n - 1)
MaxMin3
MaxMin2 (caso médio)
MaxMin2 (melhor caso)

- Existe possibilidade de obter um algoritmo MaxMin mais eficiente?

- Para responder temos de conhecer o **limite inferior** para essa classe de algoritmos.

- Técnica muito utilizada: uso de um oráculo:

> Dado um modelo de computação que expresse o comportamento do algoritmo, o oráculo informa o resultado de cada passo possível (no caso, o resultado de cada comparação).

- Para derivar o limite inferior, o oráculo procura sempre fazer com que o algoritmo trabalhe o máximo, escolhendo como resultado da próxima comparação aquele que cause o maior trabalho possível necessário para determinar a resposta final.

3. Apresente a função de complexidade de tempo para os algoritmos abaixo, indicando em cada caso qual é a operação relevante:

a)

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

b)

```
INSERTION-SORT(A)
for j ← 2 to comprimento[A] do
    chave ← A[j]
    i ← j - 1
    A[0] ← chave    //sentinela
    while A[i] > chave do
        A[i+1] ← A[i]
        i ← i-1
    A[i+1] ← chave
```

c)

```
BUBBLE-SORT(A)
for i ← 1 to comprimento[A] do
    for j ← comprimento[A] downto i+1 do
        if A[j] < A[j-1] then
            A[j] ↔ A[j-1]
```

d)

```
SELECTION-SORT(A)
for i ← 1 to comprimento[A]-1 do
    Min ← i
    for j ← i+1 to comprimento[A] do
        if A[j] < A[Min] then
            Min ← j
    A[Min] ↔ A[i]
```

Prêmio Turing
1974
Medalha Franklin
1988
Medalha John von Neumann IEEE
1995

Para cada erro encontrado em seus livros ele oferece US$2,56, pois "256 centavos são um dólar em hexadecimal".
"Cuidado com os defeitos do código anterior, eu apenas os demonstrei, não os experimentei" (Donald Knuth)

annatostes at gmail.com