UNIP
UNIVERSIDADE PAULISTA

CIÊNCIA DA COMPUTAÇÃO

MATERIAL INSTRUCIONAL ESPECÍFICO

TOMO 6

CQA - COMISSÃO DE QUALIFICAÇÃO E AVALIAÇÃO

CIÊNCIA DA COMPUTAÇÃO

MATERIAL INSTRUCIONAL ESPECÍFICO

TOMO 6

Christiane Mazur Doi

Doutora em Engenharia Metalúrgica e de Materiais, Mestra em Ciências - Tecnologia Nuclear, Especialista em Língua Portuguesa e Literatura, Engenheira Química e Licenciada em Matemática, com Aperfeiçoamento em Estatística. Professora titular da Universidade Paulista.

Tiago Guglielmeti Correale

Doutor em Engenharia Elétrica, Mestre em Engenharia Elétrica e Engenheiro Elétrico (ênfase em Telecomunicações). Professor titular da Universidade Paulista.

Questão 1

Questão 1.[1]

Um processo tem um ou mais fluxos de execução, normalmente denominados apenas threads.

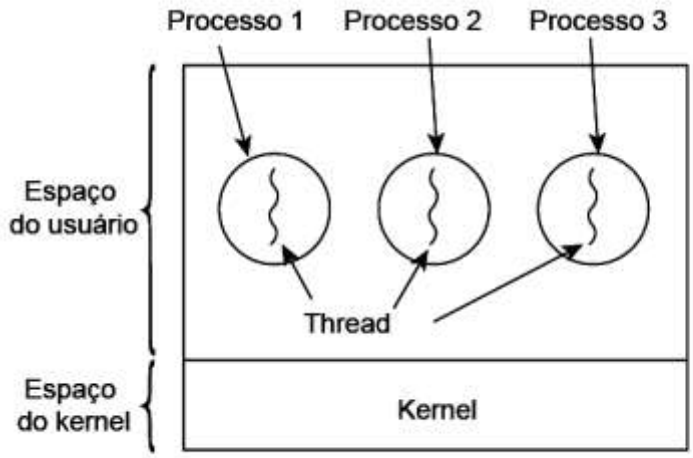


Figura 1

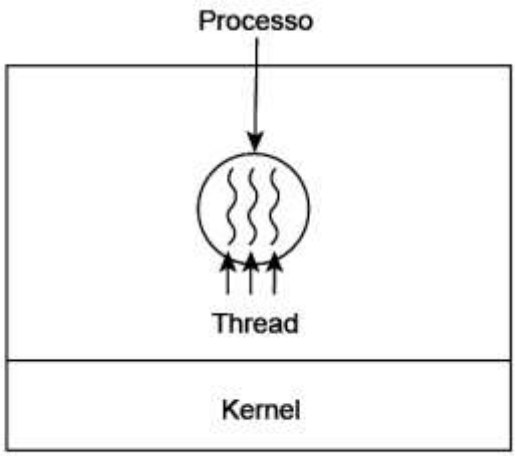


Figura 2

TANENBAUM, A.D. *Sistemas operacionais modernos*. 3. ed. São Paulo: Pearson Prentice Hall. 2010 (com adaptações).

A partir das figuras 1 e 2 apresentadas, avalie as afirmativas.

I. Tanto na figura 1 quanto na figura 2, existem três threads que utilizam o mesmo espaço de endereçamento.

II. Tanto na figura 1 quanto na figura 2, existem três threads que utilizam três espaços de endereçamentos distintos.

III. Na figura 2, existe um processo com um único espaço de endereçamento e três threads de controle.

IV. Na figura 1, existem três processos tradicionais, cada qual tem seu espaço de endereçamento e uma única thread de controle.

V. As threads permitem que várias execuções ocorram no mesmo ambiente de processo de forma paralela umas às outras.

É correto apenas o que se afirma em

A. I, II e III.

B. I, II e IV.

C. I III e V.

D. II, IV e V.

E. III, IV e V.

[1]Questão 20 – Enade 2014 (com adaptações).

1. Introdução teórica

1.1. Paralelismo, interatividade e compartilhamento de recursos

Durante muito tempo, computadores eram capazes de executar um único processo de cada vez, de forma sequencial. O operador dos grandes *mainframes* controlava a execução dos processos, reunidos em lotes de programas executados um após o outro. Nessa situação, os programas não rodavam de forma interativa: o programador não podia interferir nos programas em execução e apenas recebia o resultado do programa após o término do processamento do lote.

Essa forma de utilização dos computadores era lenta, trabalhosa e ineficiente. Além disso, os dispositivos de entrada e de saída, como impressoras e dispositivos de armazenamento em massa, eram (e frequentemente ainda são) muito mais lentos do que a CPU do computador. Assim, a CPU passava grande parte do tempo aguardando a resposta de outros dispositivos externos. Nesse período, a CPU ficava ociosa e o tempo de processamento era desperdiçado.

A pressão pela utilização mais racional dos recursos computacionais levou a três ideias diferentes, mas profundamente interligadas: interatividade, paralelismo e compartilhamento de recursos.

Interatividade é a possibilidade de o usuário interagir diretamente com o programa durante a sua execução. Esse é o cenário contemporâneo: muitas pessoas jovens nem sabem que a forma mais comum de interação entre o usuário e o sistema era o envio de cartões perfurados para processamento e recebimento de um formulário contínuo em um escaninho, que representava a resposta da execução do programa.

Paralelismo significa fazer com que dois ou mais processos sejam executados simultaneamente. O paralelismo é real se existirem várias CPUs capazes de executar várias instruções simultaneamente ou se existir uma CPU capaz de executar mais de uma instrução em um dado instante. Porém, frequentemente, o número de unidades de processamento é inferior ao número de processos e, de fato, apenas uma única CPU executa vários processos.

Quando temos vários processos e apenas uma única CPU, podemos simular a execução paralela por meio da divisão do tempo em pequenos intervalos, cada um associado a um processo. Durante esse pequeno intervalo de tempo, o processador executa as instruções de um dado programa. Ao final desse intervalo, ou quando o

processo em questão para esperando alguma resposta ou algum dado, o sistema operacional é alternado para outro processo. Se o intervalo de tempo for suficientemente pequeno e se o *clock* da máquina for rápido, essa troca ocorre de forma imperceptível para o usuário, que observa vários programas aparentemente sendo executados em paralelo.

As ideias de paralelismo e interatividade permitem que um único usuário interaja com uma máquina de forma muito similar à forma como hoje utilizamos nossos computadores e telefones celulares.

O compartilhamento de recursos de uma máquina entre vários processos e entre vários usuários simultâneos possibilitou grande expansão na comunidade de utilizadores e programadores de computadores em uma época na qual o seu custo era muito alto. Nas décadas de 1960 e 1970, por exemplo, era economicamente interessante fazer com que mais de uma pessoa fosse capaz de utilizar um computador ao mesmo tempo, com o objetivo de amortizar custos.

A interatividade, o paralelismo e o compartilhamento de recursos formaram o tripé que orienta o desenvolvimento da informática até hoje.

1.2. Processos, threads e processamento paralelo

Uma das tarefas dos sistemas operacionais é o gerenciamento da execução dos diversos programas em um computador.

Um programa é um conjunto de instruções que serão posteriormente executadas pelo computador. Os programas ficam, normalmente, armazenados em algum dispositivo de armazenamento em massa (ou, ainda, na memória permanente ROM) e são executados a pedido do usuário. Por exemplo, o usuário de um computador moderno pode ter armazenado em seu computador vários jogos, planilhas e programas para a escrita de documentos de texto. Além disso, vários outros programas auxiliares também podem ser armazenados.

Normalmente, não queremos executar todos os programas ao mesmo tempo, mas apenas um ou alguns simultaneamente. Quando queremos executar um programa específico, normalmente "clicamos" no ícone do programa ou digitamos o seu nome na linha de comando. Nesse momento, o programa é transferido do sistema de armazenamento em massa para a memória principal (normalmente RAM) e começa a ser executado. No momento em que o programa é transferido para a memória principal e é executado, temos um processo. Observe que um mesmo programa pode dar origem a

vários processos, tanto no caso em que se emprega alguma técnica de programação quanto no caso em que o usuário executa um mesmo programa várias vezes (por exemplo, clicando no ícone muitas vezes ou disparando repetidamente o mesmo processo).

A possibilidade de executar vários programas simultaneamente é muito interessante e útil, mas também é um grande desafio: programas rodando em paralelo podem afetar uns aos outros. O sistema operacional deve ser capaz de "isolar" os programas de forma que um não afete de forma negativa o funcionamento dos demais. Uma das maneiras de se fazer isso é por meio da delimitação de espaços de endereçamento. De modo simplificado, podemos pensar que o espaço de endereçamento de um programa corresponde a um conjunto de endereços de memória que pertencem a dado processo e que só podem ser lidos e alterados por esse processo (exceto em algumas situações especiais determinadas pelo programador).

Ao isolarmos os espaços de endereçamento de cada processo, podemos garantir que um programa não vai interferir no funcionamento de outro que seja executado em paralelo. No entanto, o completo isolamento entre processos faz com que esses programas não sejam capazes de trocar nenhuma informação entre si, o que pode ser desejável em algumas circunstâncias.

É possível fazer com que dois ou mais processos consigam enxergar uma área de memória compartilhada. Contudo, a mudança de contexto entre processos tem um custo computacional, que pode prejudicar o desempenho do programa.

Para contornar esse problema, foi criada a ideia de thread. Podemos encarar threads como funções similares às funções da linguagem C. Essas funções executam em paralelo e dentro de um mesmo processo e, portanto, dentro do mesmo espaço de endereçamento. Por executarem dentro do mesmo espaço de endereçamento, threads podem compartilhar variáveis da mesma forma que várias funções em um programa podem compartilhar informações entre si. Assim, os problemas da comunicação e da mudança de contexto ficam resolvidos. Contudo, o programador deve ser cuidadoso ao criar programas que utilizam threads: como elas executam em paralelo e compartilham informações e recursos, vários problemas podem surgir, como, por exemplo, as chamadas condições de disputa. Observe ainda que as threads podem se comunicar dentro de um mesmo espaço de endereçamento. Threads que executam em espaços de endereçamento diferentes (por exemplo, as que executam em diferentes processos) não se comunicam

reciprocamente, a não ser pelos mesmos mecanismos disponíveis para comunicação entre processos e com as mesmas penalidades deles decorrentes sobre o desempenho.

Os usos de threads e de processos apresentam vantagens e desvantagens. Em situações nas quais temos algoritmos que executam em paralelo e com pouca ou nenhuma comunicação, a utilização de processos é bastante adequada. Quando temos a necessidade de muita comunicação, pode ser interessante o emprego de threads.

2. Análise das afirmativas

I – Afirmativa incorreta.
JUSTIFICATIVA. Na figura I, temos três threads e três processos. Cada thread pertence a um dos respectivos processos. Como processos separados não compartilham memória (pelo menos não normalmente, na maioria dos sistemas operacionais modernos e em condições de execução padrão), não podemos afirmar que as threads utilizam o mesmo espaço de endereçamento.

II – Afirmativa incorreta.
JUSTIFICATIVA. Na figura II, temos apenas um processo com três threads. Como essas três threads pertencem ao mesmo processo, elas compartilham o mesmo espaço de endereçamento.

III – Afirmativa correta.
JUSTIFICATIVA. Quando temos threads de um mesmo processo, elas compartilham o mesmo espaço de endereçamento, como indicado na figura II, na qual as threads estão desenhadas no mesmo círculo (que representa um único processo).

IV – Afirmativa correta.
JUSTIFICATIVA. Na maioria dos sistemas operacionais modernos, cada processo tem pelo menos uma thread associada. Novas threads podem ser disparadas pelo processo, se necessário. De forma similar, um processo também pode criar outros novos processos. Cada processo tem o seu próprio espaço de endereçamento, que é utilizado pela(s) thread(s) do respectivo processo.

V – Afirmativa correta.

JUSTIFICATIVA. Tanto threads quanto processos permitem que o sistema operacional execute operações em paralelo ou que, pelo menos, simule essa execução, no caso de máquinas com apenas um processador e um núcleo. Contudo, threads que executam dentro de um mesmo processo podem facilmente compartilhar dados, pois todas existem no mesmo espaço de endereçamento e podem compartilhar o acesso
às suas variáveis, por exemplo.

Alternativa correta: E.

3. Indicação bibliográfica

- TANENBAUM, A. D. *Sistemas operacionais modernos*. 3. ed. São Paulo: Pearson Prentice Hall, 2010.

Questão 2

Questão 2.[2]

Considere as seguintes tabelas de um banco de dados:

1. Fornecedor (cod_fornec, nome_fornec, telefone, cidade, UF)
2. Estado (UF, nome_estado)

A expressão SQL que obtém os nomes dos estados para os quais não há fornecedores cadastrados é

A.

```
SELECT E.UF
FROM Estado AS E
WHERE E.nome_estado NOT IN (
SELECT F.UF
FROM Fornecedor AS F);
```

B.

```
SELECT E.nome_estado
FROM Estado AS E, FROM
Fornecedor AS F
WHERE E.UF = F.UF;
```

C.

```
SELECT E.nome_estado
FROM Estado AS E
WHERE E.UF NOT IN (
SELECT F.UF
FROM Fornecedor AS F);
```

D.

```
SELECT E.nome_estado
FROM Estado AS E, FROM
Fornecedor AS F
WHERE E.nome_estado = F.UF;
```

[2]Questão 24 – Enade 2014.

```
E.
SELECT E.nome_estado
FROM Estado AS E
WHERE E.UF IN (
SELECT F.UF
FROM Fornecedor AS F);
```

1. Introdução teórica

1.1. Linguagem SQL

Structured Query Language (SQL) é uma linguagem originalmente desenvolvida pela IBM, cujo desenvolvimento está intimamente ligado aos bancos de dados relacionais.

Após o seu desenvolvimento inicial, a linguagem SQL tornou-se um padrão, tendo sido revisada diversas vezes desde a década de 1980 até o início do século XXI. É uma das linguagens mais utilizadas em aplicações que utilizam banco de dados e é amplamente suportada pela maioria dos sistemas de gerenciamento de banco de dados.

Ainda que a maioria dos fornecedores empregue uma versão similar da linguagem SQL, frequentemente existem pequenas diferenças de comandos e funcionalidades que variam de fornecedor para fornecedor, de forma que se deve atentar para a implementação específica que se está utilizando, o que significa sempre consultar a documentação do sistema de gerenciamento de banco de dados em uso.

Segundo Silberschatz, Korth e Sudarshan (2011), a linguagem SQL pode ser dividida nas partes a seguir.

- Data definition language (DDL)
- Data manipulation language (DML)
- Data query language (DQL)
- Comandos de integridade de dados
- Definições de visões (views)
- Controle de transações
- SQL embutido (embedded SQL) e SQL dinâmico (dynamic *SQL*)
- Autorização

Cada uma dessas partes tem um conjunto de expressões e comandos característicos, voltados para uma finalidade específica. Por exemplo, a *data query language* (DQL) permite a consulta ao banco de dados e dispõe do comando SELECT, que seleciona um grupo de colunas em uma ou mais tabelas, baseada em determinado critério de seleção.

1.1.1. Data Query Language (DQL) e comando SELECT

O comando SELECT permite-nos consultar o conteúdo de uma tabela ou de um conjunto de tabelas de um banco de dados. Segundo Silberschatz, Korth e Sudarshan (2011), a estrutura básica de uma consulta a um banco de dados, quando utilizada a linguagem SQL, é composta por três cláusulas: SELECT, FROM e WHERE. De forma simplificada, podemos dizer que o comando seleciona um conjunto de "colunas" especificadas pela cláusula SELECT nas tabelas (ou relações) especificadas pela cláusula FROM, seguindo as "condições" impostas pela cláusula WHERE. Por exemplo, suponha uma tabela (relação) chamada "Compositor", que contenha os atributos nome e gênero, conforme tabela 1.

Tabela 1. Exemplo de uma tabela em um banco de dados.

Compositor

Nome	**Gênero**
Ludwig van Beethoven	Clássico
Johann Sebastian Bach	Barroco
John Coltrane	Jazz
Miles Davis	Jazz
Antônio Carlos Jobim	MPB

Se quisermos selecionar todos os nomes dos compositores dessa tabela, basta fazermos a seguinte consulta:

```
SELECT Nome FROM Compositor;
```

Obtemos, como resultado, a relação da tabela 2.

Tabela 2. Resultado da consulta SELECT Nome FROM Compositor.

Ludwig van Beethoven
Johann Sebastian Bach
John Coltrane
Miles Davis
Antônio Carlos Jobim

Com a cláusula WHERE, podemos adicionar condições à consulta. Por exemplo, se quisermos uma consulta onde o Gênero seja apenas Jazz, fazemos o seguinte:

```
SELECT Nome FROM Compositor WHERE Gênero = 'Jazz';
```

Nesse caso, obtemos a tabela 3 a seguir.

Tabela 3. Resultado da consulta SELECT Nome FROM Compositor WHERE Gênero='Jazz'.

John Coltrane
Miles Davis

2. Resolução da questão

Para resolvermos a questão, devemos selecionar todos os nomes de todos os estados dos registros da tabela de fornecedores. Assim, obtemos uma relação que contém os estados de todos os fornecedores cadastrados. Fazemos isso com a seguinte consulta:

```
SELECT F.UF FROM Fornecedor AS F
```

A parte do comando que diz "AS F" dá um "apelido" para a relação Fornecedor, apenas nessa consulta, que, em vez de ser chamada por todo o nome Fornecedor, passa a ser chamada apenas de F, o que facilita a escrita do comando.

Para saber quais estados faltam, basta selecionarmos todos os estados da tabela Estado, excluindo os estados cadastrados na tabela Fornecedor que obtivemos com a consulta anterior.

A linguagem SQL permite que verifiquemos se tuplas pertencem a uma relação. Segundo Silberschatz, Korth e Sudarshan (2011), o conector IN permite fazer esse tipo de

teste em um conjunto, sendo o conjunto composto pelos dados obtidos pelo comando SELECT. Também podemos utilizar a versão negada do conector IN, NOT IN, que permite selecionar membros que não pertençam a um conjunto. Isso é precisamente o que queremos fazer: selecionar todos os estados que não aparecem na tabela Fornecedor. Para selecionarmos o nome de todos os estados cadastrados na tabela Estado, basta fazermos:

```
SELECT E.nome_estado FROM Estado AS E
```

Para excluirmos todos os nomes dos estados cadastrados na tabela Fornecedor, basta utilizarmos as cláusulas WHERE e NOT IN, obtendo a versão final da consulta:

```
SELECT E.nome_estado
FROM Estado AS E
WHERE E.UF NOT IN (
SELECT F.UF
FROM Fornecedor AS F);
```

Alternativa correta: C.

3. Indicação bibliográfica

- SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. *Database system concepts.* 6. ed. New York: McGraw-Hill, 2011.

Questão 3

Questão 3.[3]

Considere o seguinte argumento:

1. Se existe fogo, então existe oxigênio.
2. Não há oxigênio.
3. Então não há fogo.

A regra de inferência que justifica a validade do argumento acima é

A.

$$\frac{P \rightarrow Q, \neg P}{\neg Q}$$

B.

$$\frac{P \rightarrow Q, \neg Q}{\neg P}$$

C.

$$\frac{P \rightarrow Q, Q}{P}$$

D.

$$\frac{P \rightarrow Q, \neg Q}{\neg \neg P}$$

E.

$$\frac{P \rightarrow Q, P}{Q}$$

[3]Questão 27 – Enade 2014.

1. Introdução teórica

1.1. Proposições

Segundo de Souza (2008), uma proposição "é uma sentença declarativa que pode ser interpretada como verdadeira ou falsa". No mesmo texto, o autor ainda adiciona que uma proposição não deve ser ambígua e não permite mais de uma única interpretação. Segundo Alencar Filho (2002), "chama-se proposição todo o conjunto de palavras ou símbolos que exprimem um pensamento de sentido completo". Essencialmente, dizemos que uma proposição é uma frase para a qual podemos atribuir um valor (único) de verdadeiro ou falso. Temos, a seguir, alguns exemplos de proposições.

a. Júpiter é um planeta do sistema solar (proposição verdadeira).
b. A Terra é um planeta do sistema solar (proposição verdadeira).
c. A Lua é um planeta (proposição falsa).

1.2. A proposição condicional

Segundo Alencar Filho (2002), "proposição condicional ou apenas condicional é uma proposição representada por **se p então q**, cujo valor lógico é falso (F) no caso em que p é verdadeira e q é falsa e verdadeiro (V) nos demais casos".

Costumamos expressar a proposição condicional pelo símbolo $\rightarrow$ e podemos observar sua tabela verdade na tabela 1.

Tabela 1. Tabela da verdade para a proposição condicional.

p	**q**	$p \rightarrow q$
V	V	V
V	F	F
F	V	V
F	F	V

2. Resolução da questão

A sentença "se existe fogo, então existe oxigênio" pode ser estudada pelas afirmativas que seguem.

p (antecedente): existe fogo

q (consequente): existe oxigênio

$p \rightarrow q$: se existe fogo, então existe oxigênio

Na segunda afirmativa, não há oxigênio, ou seja, $\neg Q$. Observando a tabela da verdade para a proposição condicional, sabemos que a segunda e quarta linhas da tabela correspondem à situação na qual q é falso. Nessas duas situações, a terceira coluna da tabela apresenta o valor é sempre $\neg P$, levando-nos à seguinte conclusão:

$$\frac{P \rightarrow Q, \neg Q}{\neg P}$$

Alternativa correta: B.

3. Indicações bibliográficas

- ALENCAR FILHO, E. *Iniciação à lógica matemática.* São Paulo: Nobel, 2002.
- DE SOUZA, J. N. *Lógica para a Ciência da Computação.* Rio de Janeiro: Elsevier Brasil, 2008.

Questão 4

Questão 4.[4]

Um cientista afirma ter encontrado uma redução polinomial de um problema NP-Completo para um problema pertencente à classe P.

Considerando que essa afirmação tem implicações importantes no que diz respeito à complexidade computacional, avalie as asserções e a relação proposta entre elas.

I. A descoberta do cientista implica P=NP.

PORQUE

II. A descoberta do cientista implica a existência de algoritmos polinomiais para todos os problemas NP-Completos.

A respeito dessas asserções, assinale a opção correta.

A. As asserções I e II são proposições verdadeiras, e a asserção II justifica a I.

B. As asserções I e II são proposições verdadeiras, e a asserção II não justifica a I.

C. A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.

D. A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.

E. As asserções I e II são proposições falsas.

1. Introdução teórica

1.1. Classes de problemas computacionais

Os problemas computacionais são, normalmente, divididos em classes de complexidade. O objetivo dessa classificação é encontrar uma forma de separar os problemas complexos dos problemas mais simples, já que os primeiros requerem grande capacidade computacional e os últimos requerem recursos computacionais mais modestos.

Aqui, estamos utilizando a palavra complexo de uma forma objetiva: a questão não é se o algoritmo é simples ou se é difícil de ser compreendido, mas sim se a sua execução pelo computador requer muitos recursos em termos de memória e/ou de tempo de execução.

Podemos perguntar: por que investir tempo para caracterizar matematicamente a complexidade de um algoritmo se é possível simplesmente executá-lo? Não seria mais simples apenas executar o programa e cronometrar quanto tempo leva para a sua execução ou monitorar o seu perfil de execução e observar o seu consumo de recursos?

[4]Questão 28 – Enade 2014.

De fato, frequentemente executam-se programas como uma forma de verificar seu desempenho. Ao fazer uma análise matemática detalhada do seu funcionamento, contudo, pode-se compreender muito melhor a natureza da sua execução. Além disso, deve ser considerado que o hardware do computador exerce influência sobre o tempo de execução. Outro item a ser avaliado é a qualidade da implementação de um algoritmo, que pode afetar de forma negativa o tempo de execução. Uma análise puramente matemática permite o estabelecimento de um critério mais consistente de comparação, independentemente do hardware e da implementação específica.

Alguns algoritmos são tão complexos do ponto de vista computacional que não importa o quão cuidadosamente seja realizada sua construção, seu tempo de execução e seu consumo de memória sempre serão extremamente elevados. Nesses casos, em que uma solução exata do problema é impossível do ponto de vista prático (a solução matemática ótima existe, mas o seu cálculo requer uma quantidade de recursos absurdamente elevada), diz-se que esses são problemas intratáveis. Isso não é o mesmo que dizer que o problema é impossível de ser resolvido de forma teórica: a solução teórica existe, mas requer capacidades computacionais extraordinárias.

Essa necessidade leva ao problema da caracterização matemática dos algoritmos. Algoritmos podem ser caracterizados do ponto de vista do seu tempo de execução ou do espaço (memória) necessário para o seu funcionamento.

Se x é um número inteiro que representa o tamanho da entrada de um algoritmo (por exemplo, no caso de um algoritmo de ordenação, x é o número total de elementos a serem ordenados), devemos associar a essa variável uma função y=f(x) que representa o tempo de execução do algoritmo.

Normalmente, não estamos interessados no tempo exato de execução, nem nos valores de f(x) para valores pequenos de x. O objetivo é caracterizar o crescimento da função para valores elevados de x e cotejar funções de diferentes algoritmos, o que fornece um critério de comparação do tempo de execução.

Com base nisso, concluímos que algoritmos com tempos de execução exponencial, indicados por $O(e^x)$, levam tempos muito maiores para serem executados do que algoritmos com tempos de execução polinomial, indicados por $O(x^n)$. Ou seja, para valores elevados de x, algoritmos do tipo $O(x^n)$ são mais rápidos do que algoritmos do tipo $O(e^x)$.

1.2. Problemas polinomiais e exponenciais

Uma das classes de problemas computacionais mais comuns é a classe P, ou seja, a classe dos problemas que podem ser resolvidos em tempo polinomial. Isso não quer dizer, necessariamente, que todos esses problemas possam ser resolvidos de modo rápido. Por exemplo, um algoritmo cujo tempo de execução cresce segundo $O(x^{20})$, em que x é o tamanho do vetor de entrada do problema, certamente não é rápido de ser executado, mas é um algoritmo polinomial. Mesmo para uma entrada de tamanho x=10, temos tempo de execução da ordem de 10^{20}. Normalmente, não estamos utilizando unidades de tempo: isso vai depender do *clock* do computador específico. A finalidade da avaliação matemática dos algoritmos é comparativa, e não visa à obtenção de tempos de execução absolutos.

Além da classe polinomial, temos a classe dos algoritmos exponenciais, ou seja, aqueles cuja complexidade cresce exponencialmente com o tamanho da entrada. Esses algoritmos são considerados intratáveis, pois funções exponenciais crescem de forma tão rápida que, mesmo para entradas pequenas, o tempo de execução é incrivelmente elevado.

1.3. Máquinas deterministas e máquinas não deterministas

A análise formal de algoritmos requer que pensemos em uma máquina computacional abstrata. Não estamos preocupados com uma configuração específica ou uma plataforma. O matemático britânico Alan Turing criou uma máquina abstrata, chamada de máquina de Turing, cujo propósito era precisamente representar o processo computacional de forma genérica, sem se referir a um computador específico.

Outra variação da máquina de Turing foi criada com a finalidade de facilitar a análise de alguns tipos de algoritmos: a máquina de Turing não determinista. A ideia básica da máquina não determinista é de que essa máquina se "divida" e que seja capaz de executar várias sequências de um algoritmo de forma simultânea. Podemos imaginar que isso equivale a um computador com uma quantidade infinita de microprocessadores, todos podendo trabalhar em paralelo e simultaneamente. Obviamente, não existem computadores reais com infinitos microprocessadores. Alguns supercomputadores podem ter milhares de microprocessadores, mas esse número é sempre finito.

A vantagem de uma máquina não determinista é poder executar muitas instruções de forma paralela. Assim, se houver um problema que pode ser sua execução em diversas

execuções paralelas, uma máquina não determinista seria a forma ideal de avaliar a execução desse tipo de algoritmo.

1.4. A classe de problemas NP-completos

Para compreendermos as classes de execução dos algoritmos, podemos pensar em um problema de decisão do tipo sim ou não, como, por exemplo, o reconhecimento de uma linguagem. Define-se a classe NP (*nondeterministic polynomial time*) como as linguagens que podem ser verificadas por um algoritmo em tempo polinomial que execute em uma máquina não determinística (CORMEN et al, 2009). Note que dado problema pode requerer muitas verificações e uma máquina de Turing não determinista pode fazer essas verificações em paralelo.

Outra classe importante são os problemas NP-completos. Os problemas NP-completos constituem uma categoria de problemas NP que são essencialmente equivalentes. Considere um problema de decisão L. Esse problema é NP-completo se ele for do tipo NP e se qualquer outro problema em NP puder ser reduzido ao problema L por um algoritmo em tempo polinomial.

Não sabemos se existe algum algoritmo capaz de resolver um problema NP-completo em tempo polinomial por uma máquina determinista. Se esse algoritmo existir, então todos os problemas da classe NP poderão ser resolvidos em tempo polinomial. Porém, se alguém provar que esse algoritmo não existe, então poderá afirmar que as classes P e NP são diferentes, ou seja, $P \neq NP$.

O fato de ninguém ter encontrado determinado algoritmo não implica, necessariamente, que seja impossível de se construir um algoritmo com determinadas características. É necessário provar, matematicamente, que a construção de um algoritmo seja impossível. Infelizmente, até o momento ninguém encontrou um algoritmo que seja capaz de resolver problemas NP-completos em tempo polinomial, nem provou que esse algoritmo não existe.

2. Análise das asserções

Para a resolução dessa questão, fica mais fácil analisarmos as asserções na ordem decrescente: primeiramente a asserção II e, depois, a asserção I.

II – Asserção verdadeira.

JUSTIFICATIVA. Como um problema NP-completo sempre pode ser redutível a outro problema NP em tempo polinomial (pela definição da classe NP-completo), então essa descoberta implicaria que todos os problemas da classe NP poderiam ser solucionados em tempo polinomial.

I - Asserção verdadeira.

JUSTIFICATIVA. Se a asserção II estiver correta, então todos os problemas NP podem ser resolvidos em tempo polinomial, e, portanto, P = NP.

Assim, a asserção II justifica a I.

Alternativa correta: A.

3. Indicações bibliográficas

- CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. *Introduction to algorithms*. 3. ed. Cambridge: MIT Press, 2009.
- TOSCANI, L. V.; VELOSO, P. A. S. *Complexidade de algoritmos.* 2. ed. Porto Alegre: Bookman, 2008.

Questão 5

Questão 5.[5]

Na transmissão de dados em sistemas computacionais, o dispositivo de verificação de erros conhecido como bit de paridade consiste da adição de um bit extra durante a transmissão. O valor associado a esse bit é uma função da quantidade de bits de dados iguais a 1 a serem transmitidos.

Nesse contexto, considere a transmissão de 7 bits de dados, com um bit extra de paridade, em um sistema de comunicação no qual a probabilidade de transmitir um bit de forma incorreta é igual a 10^{-6} e independe de outros erros ocorridos. O bit de paridade também pode sofrer erros.

A probabilidade de ocorrência de transmissão de 2 bits errados, que seria erroneamente detectada como uma transmissão sem erros, é

A. $1{,}0 \times 10^{-12}$.

B. $2{,}0 \times 10^{-12}$.

C. $2{,}8 \times 10^{-11}$.

D. $2{,}0 \times 10^{-6}$.

E. $2{,}8 \times 10^{-5}$.

1. Introdução teórica

1.1. Transmissão digital da informação e detecção de erros

Podemos transmitir e armazenar informações de forma analógica ou digital. No formato analógico, uma onda de natureza elétrica (por exemplo, uma tensão elétrica, uma corrente em um circuito ou uma onda eletromagnética) é construída de forma a variar de forma *análoga a* outra grandeza (temperatura e pressão, entre outras). Essa onda é transmitida para o receptor e novamente convertida em outra forma de onda que pode ser percebida pelos sentidos humanos, como som ou imagem.

Caso exista algum tipo de degradação do sinal, devido a algum tipo de ruído de interferência, é muito difícil diferenciar e separar o sinal original do ruído. Esse problema tende a se tornar ainda mais complexo conforme o sinal vai sendo atenuado.

[5]Questão 31 – Enade 2014.

Para obtermos o formato digital, um sinal analógico é, inicialmente, convertido em uma sequência de números inteiros, que são, posteriormente, representados em binários, tornando-se uma sequência de zeros e uns. Em seguida, essa sequência pode ser transmitida ou gravada de acordo com dada codificação, com uma escolha adequada de símbolos para sua representação.

A vantagem no formato digital de transmissão está na simplicidade de acrescentar-se *redundância* ao sinal original. Esses bits adicionais não aumentam a quantidade de informação transmitida, mas repetem parte da informação originalmente contida nos bits da sequência inicial.

A ideia de redundância pode parecer estranha ou até mesmo ruim, uma vez que usualmente acreditamos que informação "repetida" é um desperdício de tempo e espaço. Contudo, do ponto de vista da transmissão ou armazenamento da informação, a redundância pode ser uma forma eficaz de se acrescentar robustez a um sistema de comunicação ou de armazenamento.

Uma das formas de criarmos um sistema de codificação robusto em relação a ruídos é acrescentarmos bits de checagem ao sinal original (SALOMON, 2005). Os chamados *bits de paridade* fazem isso. Por exemplo, suponha a seguinte cadeia de bits: 0010. Essa cadeia contém quatro bits, sendo três 0 e apenas um 1. Consequentemente, essa cadeia contém um número ímpar de uns. Podemos adicionar um 1 ou um 0 ao final dessa cadeia, antes da sua transmissão ou do seu armazenamento, com o objetivo de gerarmos uma nova cadeia de caracteres com uma quantidade par de uns. Nesse caso, a nova cadeia (com cinco bits) seria: 00101. Observe que essa cadeia tem um bit a mais e tem três zeros e dois uns (um número par de uns). Suponha agora outra cadeia original, por exemplo, 0011. Essa cadeia já contém um número par de uns e, portanto, devemos acrescentar um zero ao seu fim, obtendo 00110. Observe que essa nova cadeia de cinco bits também contém um número par de uns. Esse bit adicionado ao final de uma cadeia com o objetivo de se obter sempre uma sequência com um número par de uns é chamado de *bit de paridade*.

O bit de paridade representa uma redundância: ele pode ser facilmente obtido dos demais bits da cadeia. Contudo, é exatamente por esse motivo que ele é útil: permite que o receptor detecte um erro na transmissão do sinal. Por exemplo, ao recebermos uma cadeia como 00010, que contém um número ímpar de uns, sabemos que ocorreu algum erro na transmissão do sinal, mas não sabemos qual é o bit errado, que pode ser o próprio bit de paridade. Nesse caso, devemos pedir a retransmissão da cadeia de bits original.

Devemos também lembrar que o bit de paridade permite a detecção de um número ímpar de erros, mas não de um número par de erros. Isso ocorre porque, ao termos um número par de erros, a paridade da cadeia de bits não se altera. Por exemplo, suponha a cadeia de bits 0111. Vamos adicionar o bit de paridade 1, obtendo a nova cadeia 01111. Suponha, agora, que essa cadeia seja transmitida e que ocorram dois erros, no primeiro e no último bit, gerando a cadeia 11110. Observe que essa cadeia tem a mesma paridade da cadeia anterior e contém um bit de paridade correto. Entretanto, a probabilidade de mais de um erro em uma cadeia pequena de bits é pequena, bem menor do que a de um único erro.

2. Análise das alternativas

No caso de 2 erros em 8 bits, o número de situações possíveis é dado pela combinação de 8 elementos 2 a 2:

$$C_n^p = C_8^2 = \frac{n!}{p!(n-p)!} = \frac{8!}{2!6!} = 28$$

Sabendo que a probabilidade de erro de um único bit é de 10^{-6}, para termos uma situação com 6 bits corretos e 2 bits errados e considerando probabilidades de erros independentes, temos:

$$p_{6c2e} = p_e^2(1-p_e)^6 = 10^{-12}(1-10^{-6})^6 \approx 10^{-12}$$

Porém, como há 28 possibilidades de termos 6 bits corretos e 2 bits errados, o valor final é $28 \times 10^{-12} = 2{,}8 \times 10^{-11}$.

Alternativa correta: C.

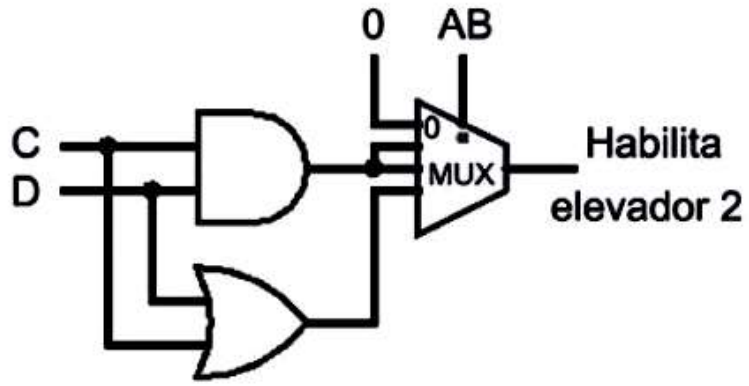
3. Indicações bibliográficas

- FLOYD, T. L. *Sistemas Digitais: Fundamentos e Aplicações.* 9. ed. Porto Alegre: Bookman, 2007.
- SALOMON, D. *Coding for Data and Computer Communications.* Nova Iorque: Springer, 2005.

Questão 6

Questão 6.[6]

Um prédio de 4 andares, sendo o primeiro andar térreo, é servido por 2 elevadores. Por motivo de economia de energia, o elevador 2 só é acionado se for solicitado em mais de 2 andares. Considere um circuito proposto para habilitar o acionamento do elevador 2 conforme é mostrado a seguir. Ele utiliza um multiplexador 4x1, cuja saída é selecionada através da composição dos sinais A e B, que indicam se os andares 1 e 2 solicitaram o serviço do elevador. Assim, o valor $AB=10_{(2)}$ indica que o primeiro andar solicitou o elevador, mas não o segundo. Os sinais C e D indicam se os andares 3 e 4 solicitarem o serviço, respectivamente.



Com base na análise do circuito proposto para o problema, avalie as asserções e a relação proposta entre elas.

I. O circuito não atende às especificações do projeto.

PORQUE

II. A entrada superior do multiplexador com valor constante 0 indica que a saída será 0, independentemente dos valores dos sinais A, B, C e D.

A respeito dessas asserções, assinale a opção correta.

A. As asserções I e II são proposições verdadeiras, e a asserção II justifica a I.

B. As asserções I e II são proposições verdadeiras, e a asserção II não justifica a I.

C. A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.

D. A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.

E. As asserções I e II são proposições falsas.

[6]Questão 35 – Enade 2014.

1. Introdução teórica

Circuitos digitais combinatórios e multiplexadores

Circuitos digitais combinatórios são aqueles cuja saída depende unicamente da(s) sua(s) entrada(s) em dado instante e que não têm nenhuma memória de sua situação anterior. Circuitos lógicos digitais sequenciais dependem não apenas de sua entrada em um dado instante, mas também de seu histórico de ativação.

Um dos dispositivos utilizados em projetos de circuitos digitais é o multiplexador, também chamado de mux. Ele funciona como uma espécie de "chave seletora": recebe várias entradas e seleciona apenas uma, a que será copiada na sua saída. A decisão sobre qual das entradas vai ser selecionada depende de um conjunto de entradas de seleção e de variáveis binárias que especificam quais dos sinais de entrada serão copiados para a saída. Como exemplo, suponha um multiplexador que apresente apenas duas entradas e uma saída, às vezes chamado de multiplexador 2-para-1, mostrado na figura 1.

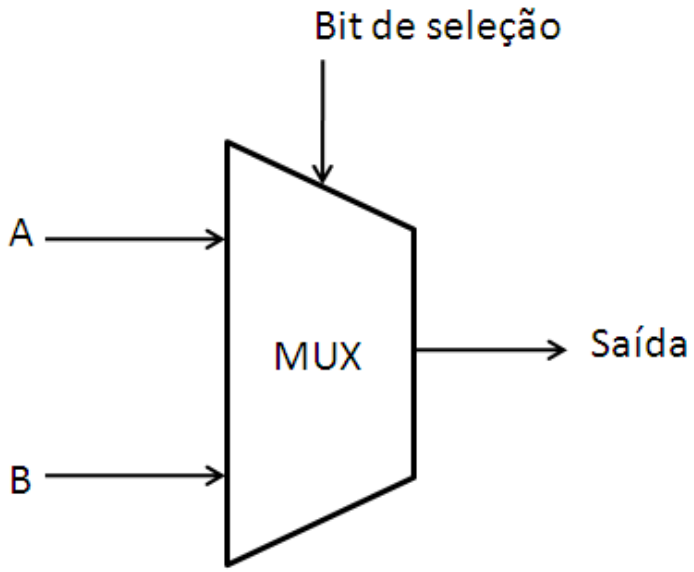


Figura 1. Um multiplexador 2-para-1.

No caso do multiplexador da figura 1, obtemos a tabela verdade a seguir (tabela 1).

Tabela 1. Tabela verdade do multiplexador da figura 1.

A	B	Bit de seleção (S)	Saída (O)
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

Observe que, quando o bit de seleção é igual a 0, a saída do multiplexador é igual ao valor da entrada A. Quando o bit de seleção é igual a 1, a saída do multiplexador é igual ao valor da entrada em B. Um circuito como esse corresponde à equação que segue.

$$O = (A \cdot \overline{S}) + (B \cdot S)$$

Multiplexadores podem ter diversos tamanhos. Na figura 2, vemos um exemplo de um multiplexador 4-para-1. Como temos o dobro de entradas (quatro), precisamos de dois bits para selecionar uma das quatro portas.

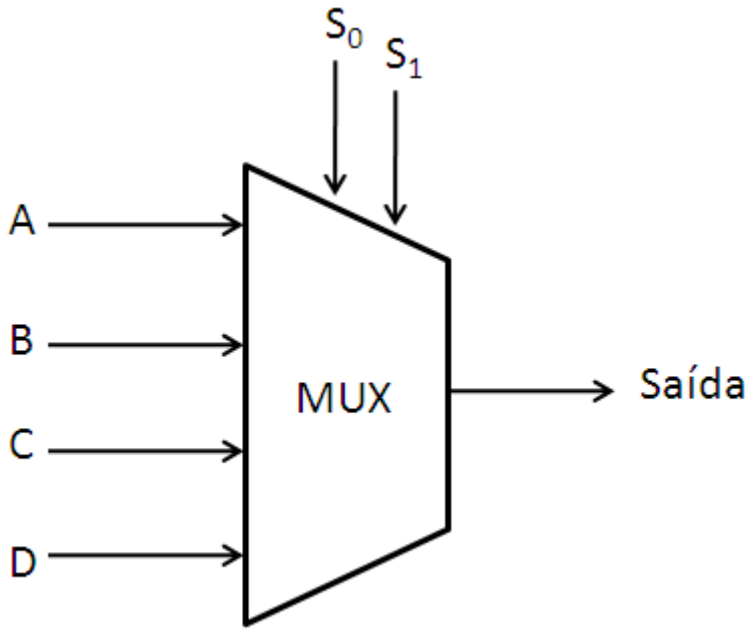


Figura 2. Um multiplexador 4-para-1.

Para esse multiplexador, temos a tabela verdade que segue (tabela 2).

Tabela 2. Tabela verdade do multiplexador da figura 2.

S_0	S_1	Saída (O)
0	0	A
0	1	B
1	0	C
1	1	D

Observe que, dependendo da combinação dos bits S_0 e S_1, temos, na saída, os valores de A, B, C ou D.

2. Resolução da questão e análise das asserções

Para resolver a questão, devemos compreender o circuito da figura do enunciado. Sabemos que as entradas de seleção A e B geram um total de 4 combinações possíveis: $00_{(2)}$, $10_{(2)}$, $01_{(2)}$ e $11_{(2)}$.

Como o elevador só deve ser acionado caso seja chamado em mais de dois andares (ou seja, em três ou quatro andares) e caso A e B sejam iguais a zero, o elevador não deve ser habilitado, independentemente dos valores de C e D. Assim, se $AB=00_{(2)}$ devemos ter como saída do multiplexador zero sempre. Isso corresponde ao zero ligado à primeira porta de entrada do multiplexador. Se tanto A quanto B estiverem acionados (ou seja, se $AB=11_{(2)}$), basta apenas um dos andares C ou D ser ativado (ou ambos) para que o elevador seja acionado. É por isso que a última porta de entrada do multiplexador (a porta selecionada caso $AB=11_{(2)}$) deve ser resultado de um OU entre as entradas C e D. Finalmente, se apenas uma das entradas de controle (A ou B) estiver com nível um e a outra estiver com o nível zero (não importa qual delas), é necessário que ambas as entradas C e D estejam selecionadas para que o elevador seja habilitado, já que apenas assim teremos três andares chamando o elevador.

Dessa forma, podemos afirmar que o circuito atende às especificações do projeto, o que torna a asserção I falsa. Além disso, a entrada superior do multiplexador igual a zero afeta a saída somente se $AB=00_{(2)}$, e não sempre, como diz a asserção II, que, dessa forma, também é falsa.

Alternativa correta: E.

3. Indicação bibliográfica

- RAFIQUZZAMAN, M. *Fundamentals of digital logic microcontrollers.* 6. ed. New Jersey: John Wiley & Sons, 2014.

Questão 7

Questão 7.[7]

Diferentes implementações da linguagem de programação PROLOG permitem predicados com parâmetros, aceitam as operações de conjunção e disjunção lógica, utilizando os símbolos vírgula (conjunção) e ponto e vírgula (disjunção), e a negação lógica com o predicado not.

Considere que um programador propôs as cláusulas mostradas a seguir, definidas em uma linguagem de programação como PROLOG, como parte da verificação de critérios para seleção de candidatos a uma chapa de presidente e vice-presidente de uma empresa. Estas cláusulas apresentam as premissas para chegar às conclusões "selecionados", "desconsiderados" e "descartado", a partir da possibilidade da existência de fatos ou regras com o identificador superior.

```
superior(jorge).
superior(ana).
selecionados(P,Q) :- superior(P), superior(Q).
desconsiderados(P,Q) :- not(superior(P)); not(superior(Q)).
descartado(P) :- not(superior(P)).
```

Considerando apenas as colocações e cláusulas acima e a hipótese de mundo fechado (*closed world assumption*), avalie as afirmativas.

I. Para todos os valores dos parâmetros P e Q, o predicado "selecionados" retornará o valor lógico falso.

II. Para todos os valores de P e Q, os predicados "selecionados" e "desconsiderados" retornarão valores lógicos diferentes.

III. A conjunção dos predicados "selecionados" e "desconsiderados", para quaisquer valores de P e Q, retornará um valor lógico verdadeiro.

IV. Para qualquer valor do parâmetro P, o predicado "descartado" retornará um valor verdadeiro.

V. A disjunção dos predicados "selecionados" e "desconsiderados", para quaisquer valores de P e Q, retornará um valor lógico verdadeiro.

É correto apenas o que se afirma em

A. I e II. B. I e III. C. II e V. D. III e IV. E. IV e V.

[7]Questão 29 – Enade 2014.

1. Introdução teórica

1.1. Paradigmas de linguagens de programação

Muitas das linguagens apresentam características semelhantes e podem ser agrupadas em paradigmas. Segundo Tucker e Noonan (2008), "um paradigma de programação é um padrão de resolução de problemas que se relaciona a um determinado gênero de programas e linguagens".

Podemos identificar quatro paradigmas na atualidade (TUCKER e NOONAN, 2008), conforme segue.

- **Programação imperativa**: nesse paradigma, o programador lista uma série de comandos de modo explícito, essencialmente dizendo ao computador como resolver um problema. É o modelo mais comum e mais antigo. Temos, como exemplos, as linguagens C, Fortran e Perl (entre outras).
- **Programação orientada a objetos**: nesse paradigma, vários objetos comunicam-se trocando mensagens. Linguagens como Java, C# e C++ suportam esse tipo de paradigma.
- **Programação Funcional**: esse paradigma foi inspirado na ideia matemática de função. Nesse tipo de linguagem, a computação é feita pela avaliação das funções que compõem o programa, de forma similar à matemática. Alguns exemplos de linguagens funcionais são as linguagens Lisp, Haskell e Scheme.
- **Programação Lógica**: nessa linguagem, inspirada pela matemática e pela lógica formal, um programa contém um conjunto de declarações (regras) e a computação é uma consequência lógica obtida dessas afirmações. A linguagem PROLOG é um exemplo desse tipo de paradigma.

1.2. Linguagem de programação PROLOG

Todas as linguagens de programação requerem "lógica" para que um programa seja escrito e executado, e o computador em si é uma máquina lógico-digital. Contudo, aqui o termo "lógica" refere-se ao fato de que um programa é uma coleção de declarações sobre determinado domínio de problema, e a computação é a conclusão lógica derivada dessas declarações.

Nas linguagens lógicas, o programador está mais preocupado com o detalhamento do que o programa deve fazer do que "como" o programa deve fazer. O programador escreve uma série de cláusulas, essencialmente afirmações sobre determinado problema. Depois, uma pergunta deve ser feita para ser respondida com base nas cláusulas do programa. O computador vai tentar *deduzir* a resposta com base nas cláusulas apresentadas.

Temos duas abordagens: "mundo fechado" e "mundo aberto". Utilizamos a hipótese "mundo fechado" (*closed world assumption*), na qual apenas dizemos que algo é verdadeiro, se o pudermos provar por meio de uma cláusula ou de uma dedução. A outra possibilidade, chamada de "mundo aberto", admite que algo possa ser verdadeiro, mesmo que isso não seja passível de ser provado apenas com a informação disponível.

2. Análise das afirmativas

Para resolvermos esse problema, criamos uma tabela (tabela 1), contendo as cláusulas descritas no enunciado.

Tabela 1. Tabela verdade para a situação do enunciado.

P	Q	P e Q (selecionados)	P ou Q	Não P	Não Q	(Não P) ou (Não Q) (desconsiderados)
F	F	F	F	V	V	V
F	V	F	V	V	F	V
V	F	F	V	F	V	V
V	V	V	V	F	F	F

I – Afirmativa incorreta.

JUSTIFICATIVA. Observe que tanto Jorge quanto Ana têm formação superior, o que significa que o predicado "selecionados" retorna verdadeiro. Dessa forma, a afirmativa está incorreta.

II – Afirmativa correta.

JUSTIFICATIVA. Observe que as colunas 3 e 7 sempre apresentam valores opostos: quando o valor de uma coluna é Verdadeiro (V), o valor da outra coluna é Falso (F). Assim, verificamos que os valores retornados pelos predicados serão sempre opostos.

III – Afirmativa incorreta.

JUSTIFICATIVA. Uma conjunção equivale ao operador lógico "E". Como os predicados "selecionados" e "desconsiderados" sempre retornam valores opostos, uma conjunção entre esses predicados equivale a fazer um "E" lógico entre uma variável e sua negação, por exemplo, "X E NÃO(X)", que sempre vai retornar o valor falso (F), e não verdadeiro (V), como é dito na afirmativa III.

IV – Afirmativa incorreta.

JUSTIFICATIVA. Basta observarmos que se P= "jorge", descartado = não(V) = Falso. Logo, a afirmativa IV está incorreta.

V – Afirmativa correta.

JUSTIFICATIVA. Uma disjunção equivale ao operador lógico "OU". Como os predicados "selecionados" e "desconsiderados" sempre retornam valores opostos, uma disjunção entre esses predicados equivale a fazer um "OU" lógico entre uma variável e sua negação, por exemplo, "X OU NÃO(X)", que sempre vai retornar o valor lógico Verdadeiro (V).

Alternativa correta: C.

3. Indicação bibliográfica

- TUCKER, A. B.; NOONAN, R. E. *Linguagens de Programação: Princípios e Paradigmas*. 2. ed. São Paulo: McGrawhill, 2008.

Questão 8

Questão 8.[8]

Qual o valor de retorno da função a seguir, caso n = 27?

```
int recursao (int n) {
        if (n <= 10) {
                return n * 2;
        }
        else {
                return recursao(recursao(n/3));
        }
}
```

A. 8.

B. 9.

C. 12.

D. 16.

E. 18.

1. Introdução teórica

Recursão

Quando, em um programa, temos uma função (ou procedimento) chamando a si mesma, isso é denominado mecanismo de recursão. À primeira vista, o fato de uma função chamar a si mesma pode parecer estranho, e talvez até mesmo errado: se uma função chama a si mesma de forma contínua, quando esse processo termina?

Ao criar uma função recursiva, o programador deve ter o cuidado de evitar situações nas quais o programa nunca termine, com uma função chamando a si mesma sem nunca ocorrer um critério de parada. Por exemplo, na função do enunciado, quando n é menor do que 10 ou igual a 10, a função retorna o dobro do valor de n. Para casos maiores, a função chama a si mesma com o valor de n/3. Dessa forma, há uma condição na qual ocorre recursão e outra condição na qual a função retorna algum valor.

Em um programa bem-comportado, sempre deve haver interrupção do processo de recursão. A função retorna a um valor que vai ser utilizado em cada chamada anterior da função. Programas bem-comportados devem levar um tempo finito para a sua execução.

[8]Questão 23 – Enade 2014.

Um cuidado que se deve ter ao utilizar recursão, mesmo quando não ocorre uma situação com infinitas chamadas, é evitar que o número de chamadas seja muito grande, para que não ocorra estouro de pilha: cada chamada a uma função implica na criação de um item a mais em uma região da memória chamada pilha de execução; se o número de elementos na pilha de execução crescer demasiadamente, a pilha pode estourar, levando ao fim indesejado da execução do programa.

Existem algumas técnicas de otimização que podem evitar situações de estouro de pilha. Uma das mais utilizadas é a chamada de recursão final própria (ou, no inglês, *tail recursion*). Nesse caso, uma chamada pode ser feita sem a necessidade de se adicionar um quadro na pilha de chamada, evitando o seu crescimento desenfreado.

2. Resolução da questão

Para a resolução da função dada na questão, podemos fazer um teste de mesa.

Vamos enumerar as linhas do fragmento de programa:

```
1) int recursao (int n) {
2)      if (n <= 10) {
3)              return n * 2;
4)      }
5)      else {
6)              return recursao(recursao(n/3));
7)      }
8) }
```

Assim, podemos obter o quadro 1.

Quadro 1. Teste de mesa do problema do enunciado.

LINHA	n	Retorno
6	27	recursao(recursao(9))
3	9	18
6	27	recursao(18)
6	18	recursao(recursao(6))
3	6	12
6	18	recursao(12)
6	12	recursao(recursao(4))
3	4	8
6	12	recursao(8)
3	8	16

Dessa forma, o valor retornado para recursão (27) é 16.

Alternativa correta: D.

3. Indicação bibliográfica

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to algorithms.* Cambridge: MIT Press, 2009.

Questão 9

Questão 9.[9]

Em relação à aplicação adequada das técnicas de Inteligência Artificial, avalie as afirmativas.

I. Indução em Árvore de Decisão é utilizada para identificação de fraudes em cartões de crédito.

II. Redes Neurais Artificiais são utilizadas no desenvolvimento de sistemas de análise de risco em aplicações financeiras.

III. Sistemas especialistas baseados em regras são utilizados no desenvolvimento de sistemas de diagnóstico de falhas em hardware.

É correto o que se afirma em

A. I, apenas.

B. III, apenas.

C. I e ii, apenas.

D. II e III, apenas.

E. I, II e III.

1. Introdução teórica

1.1. Árvores de decisão

Segundo Norvig e Russel (2010), uma árvore de decisão pode ser definida como uma função que recebe como entrada um vetor de atributos e retorna uma decisão, ou seja, um único valor de saída. Um exemplo importante de aplicação de árvore de decisão é um problema de classificação booleana, ou seja, quando um conjunto de dados de entrada deve ser classificado em apenas dois grupos: 0/1 ou verdadeiro/falso.

Adaptando o exemplo dado em Norvig e Russel (2010), supomos a seguinte situação: chegamos a uma lanchonete e decidimos se devemos ou não esperar em uma fila para sermos atendidos. Podemos ter como entrada os seguintes dados: tamanho da fila de espera, fome, disponibilidade de outra lanchonete próxima, tempo disponível para a espera etc.

Na figura 1, temos um exemplo de árvore de decisão para a situação em que devemos decidir se esperamos ou não em uma fila em uma lanchonete. Observe que a

[9]Questão 18 – Enade 2014.

resposta final deve ser sim ou não, ou seja, ou esperamos na fila (SIM) ou vamos embora (NÃO).

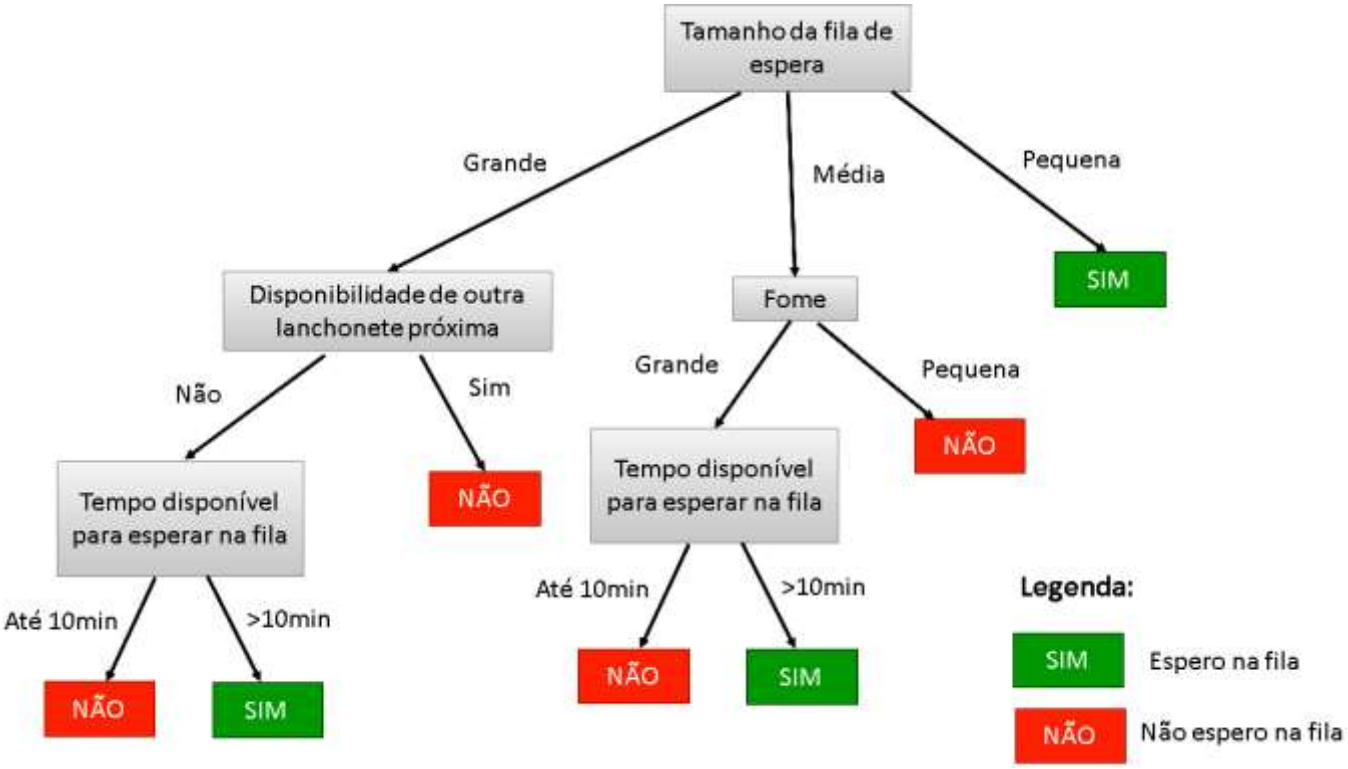


Figura 1. Exemplo de uma árvore de decisão para a espera em uma fila em uma lanchonete.
Fonte. NORVIG e RUSSEL, 2010 (com adaptações).

Ainda na figura 1, ao lado das setas, temos os valores associados ao vetor de entrada. Por exemplo, uma fila pode ser grande, média ou pequena. Pode haver a disponibilidade de outra lanchonete próxima ou não, a fome pode ser grande ou pequena e o tempo disponível para a espera (o tempo que podemos esperar na fila) pode ser inferior a 10 minutos (se estivermos com pressa) ou superior (caso contrário). Ao percorrermos a árvore de decisão, podemos decidir se vamos esperar na fila ou não, com base nas condições descritas pelo vetor de entrada.

1.2. Redes neurais artificiais

O cérebro biológico, em especial o cérebro humano, é capaz de armazenar, manipular e processar uma quantidade gigantesca de informações. O cérebro também trabalha de forma paralela, por exemplo: pessoas podem falar e andar ao mesmo tempo. Além disso, o cérebro monitora uma série de outras atividades biológicas do organismo, de forma inconsciente, ao mesmo tempo em que podemos pensar em tarefas de forma consciente.

A capacidade extraordinária de processamento do cérebro surpreende também pelo seu baixo consumo energético, especialmente se comparado aos computadores convencionais. Estima-se que o cérebro humano apresente uma dissipação média de potência de 20W (RABAEY, 2009), enquanto um computador desktop convencional pode dissipar uma potência entre 300W e 800W ou mais, dependendo do caso.

Baseado nessas observações, durante muito tempo, cientistas tentaram (e continuam tentando) desvendar o funcionamento do cérebro. Simultaneamente, a pesquisa na área de inteligência artificial vem buscando encontrar algoritmos que reproduzam ou, ao menos, simulem, a inteligência humana, mesmo que em uma área do conhecimento específica (como por exemplo, jogar xadrez ou reconhecer padrões).

O nome "redes neurais" pode se referir às redes neurais biológicas e às redes neurais artificiais. As redes neurais biológicas (às vezes chamadas de redes neuronais) são aquelas encontradas em seres vivos. As redes neurais artificiais são algoritmos computacionais inspirados nas redes neurais biológicas, porém muito mais simples. É importante que se tenha em mente que as redes neurais artificiais são extremamente simples se comparadas às redes neurais biológicas. Por enquanto, não são capazes de simular de forma completa um cérebro humano. No entanto, as redes neurais artificiais são extremamente úteis para a resolução de diversos tipos de problemas, especialmente de problemas que envolvem a necessidade de aprendizado, por serem algoritmos capazes de aprender a partir de exemplos ou através de seu próprio uso.

Segundo Gurney (2003),

> *uma rede neural é um conjunto interconectado de elementos simples de processamento, chamados de nós ou unidades, cuja funcionalidade é uma simplificação baseado no comportamento de um neurônio biológico real. A capacidade de armazenamento e processamento da rede é dada pelos elementos que conectam esses nós, que possuem um peso (número) associado. Esses pesos são obtidos por um processo de aprendizado, normalmente a partir de padrões de treinamento pré-estabelecidos (ainda que esse não seja sempre o caso).*

Na figura 2, temos um exemplo de um diagrama de um modelo não linear de um único neurônio (no caso, de número k). Observe que, nessa figura, as *p* entradas correspondem às variáveis $x_1,x_2,...,x_p$ multiplicadas pelos pesos sinápticos $w_{k1},w_{k2},...,w_{kp}$ As multiplicações das entradas pelos pesos devem ser somadas, dando origem ao valor u, inserido em uma função de ativação $\varphi(u_k - \theta_k)$. Essa função pode ser do tipo limiar, ou seja, similar à função de Heaviside (igual a zero para argumentos inferiores a zero e maior

do que zero para argumentos maiores ou iguais a zero). O valor de θ_k serve para controlar o limiar da ativação do neurônio.

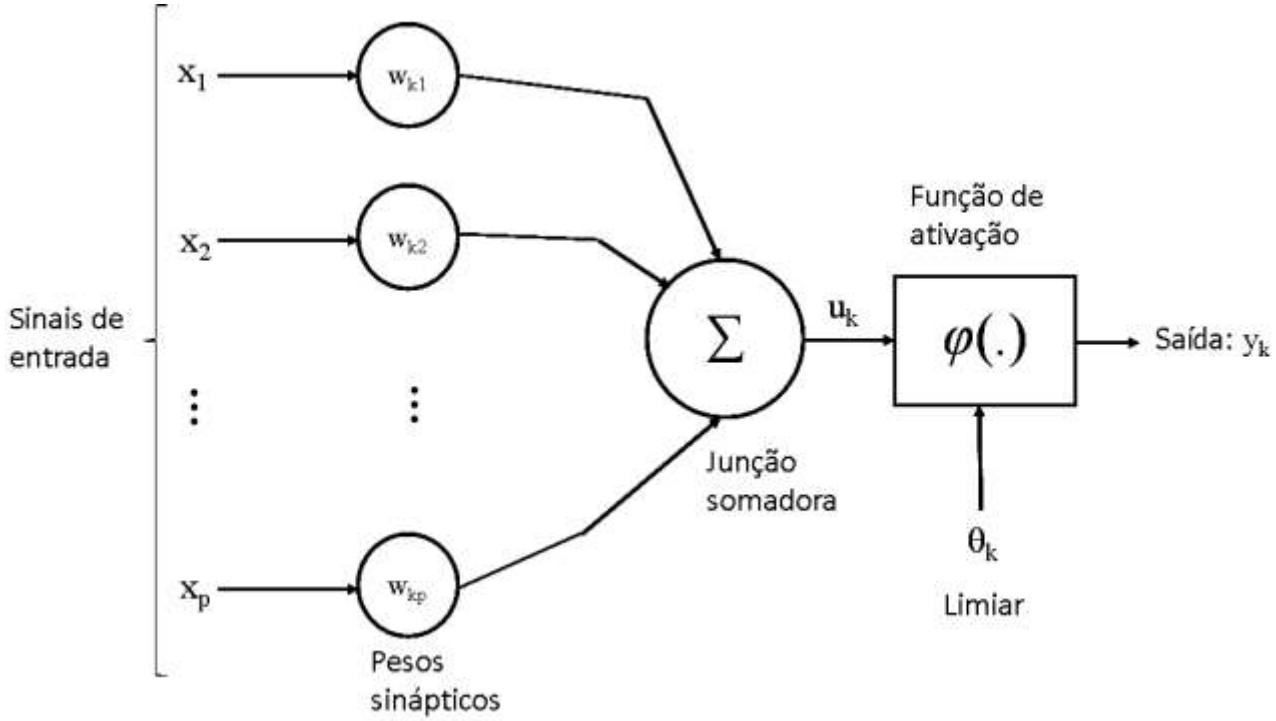


Figura 2. Modelo não linear de um neurônio.
Fonte. Haykin, 1994 (com adaptações).

A junção de diversos neurônios, como os mostrados na figura 3, vai dar origem a uma rede. Essa rede pode estar disposta em camadas, com as saídas dos neurônios de dada camada i servindo como entrada para os neurônios da camada i+1. Os pesos sinápticos devem, então, ser ajustados para que, ao final da rede, as saídas correspondam ao comportamento desejado (por exemplo, detectar, em uma entrada, determinado padrão). Um exemplo de rede neural artificial simples é o que segue (figura 3).

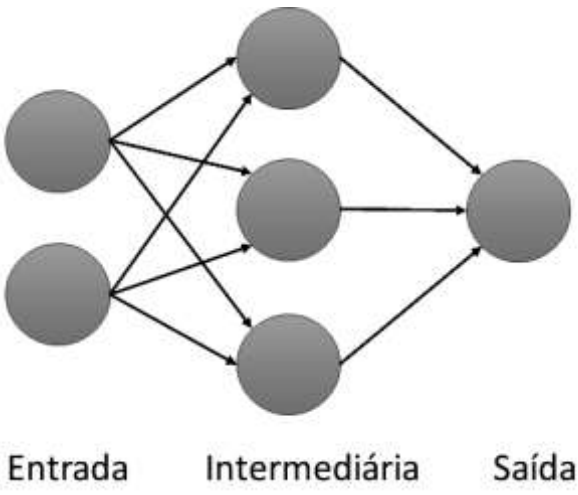


Figura 3. Exemplo de rede neural simples.

1.3. Sistemas especialistas

Heurística é um conjunto de regras gerais que permitem resolver um problema de forma rápida, mas aproximada.

Para Rychener (1988), sistema especialista é um programa de computador que é, em grande parte, uma coleção de regras heurísticas e de dados detalhados de um domínio que se mostraram úteis em resolver os problemas específicos em algum campo técnico. Em outras palavras, sistema especialista é uma coleção de regras (descritas em um formato SE-ENTÃO) que busca auxiliar na resolução de um conjunto de problemas típicos de uma área do conhecimento.

É importante observar que um sistema especialista é tão útil quanto as regras que ele possui. Além disso, alguém deve ser responsável por coletar e introduzir essas regras no sistema especialista (normalmente o desenvolvedor do sistema). O sistema vai ser tão bom quanto as regras que o alimentaram.

2. Análise das alternativas

I – Afirmativa correta.
JUSTIFICATIVA. A identificação de fraudes pode ser uma situação de resposta SIM ou NÃO (existe ou não uma fraude em dada operação). Uma árvore de decisão é uma boa alternativa para a resolução desse tipo de problema.

II – Afirmativa correta.
JUSTIFICATIVA. As redes neurais artificiais apresentam uma série de características úteis para a análise de risco. Além de serem úteis na classificação de um conjunto de dados de entrada, elas podem apresentar não apenas um resultado do tipo SIM ou NÃO, mas também um valor de confiança. Dessa forma, além de podermos utilizar uma rede neural para classificar um grande conjunto de dados, essa rede também pode fornecer o grau de confiança ou de certeza dessa classificação.

III – Afirmativa correta.
JUSTIFICATIVA. Sistemas especialistas são úteis para a detecção de falhas, especialmente em equipamentos. Os primeiros sistemas especialistas foram desenvolvidos, na área médica, para o diagnóstico de doenças, tradicionalmente feito com base em perguntas do

tipo SE-ENTÃO. Dessa forma, o diagnóstico de falhas em hardware é um tipo de aplicação adequada para sistemas especialistas.

Alternativa correta: E.

3. Indicações bibliográficas

- GURNEY, K. *An Introduction to neural networks.* London: CRC Press, 2003
- HAYKIN, S. *Neural Networks: A comprehensive foundation*. New York: Macmillan College Publishing Company, 1994.
- RUSSEL, S. J.; NORVIG, P. *Artificial intelligence: A modern approach*. Upper Saddle River: Prentice-Hall, 2010.
- RABAEY, J. *Low Power Design Essentials.* New York: Springer, 2009.
- RYCHENER, M. (Ed.) *Expert Systems for Engineering Design*. San Diego: Academic Press, 1988.
- ROGERS, H. *Theory of recursive functions and effective computability*. Cambridge: MIT Press, 1987.

Questão 10

Questão 10.[10]

Uma pilha é uma estrutura de dados que armazena uma coleção de itens de dados relacionados e que garante o seguinte funcionamento: o último elemento a ser inserido é o primeiro a ser removido. É comum na literatura utilizar os nomes *push* e *pop* para as operações de inserção e remoção de um elemento em uma pilha, respectivamente. O seguinte trecho de código em linguagem C define uma estrutura de dados pilha utilizando um vetor de inteiros, bem como algumas funções para sua manipulação.

```
#include <stdlib.h>
#include <stdio.h>
typedef struct {
      int elementos[100];
      int topo;
}pilha;
pilha * cria_pilha() {
      pilha * p =malloc(sizeof(pilha));
      p->topo = -1;
      return pilha;
}
void push(pilha *p, int elemento) {
      if (p->topo >= 99)
            return;
      p->elementos[++p->topo] = elemento;
}
int pop(pilha *p) {
      int a = p->elementos[p->topo];
      p->topo--;
      return a;
}
```

O programa a seguir utiliza uma pilha.

```
int main() {
      pilha * p = cria_pilha();
      push(p, 2);
      push(p, 3);
      push(p, 4);
      pop(p);
      push(p, 2);
      int a = pop(p) + pop(p);
      push(p, a);
      a += pop(p);
      printf("%d", a);
      return 0;
}
```

[10]Questão 16 – Enade 2014.

A esse respeito, avalie as afirmativas.

I. A complexidade computacional de ambas as funções *push* e *pop* é O(1).

II. O valor exibido pelo programa seria o mesmo caso a instrução a += pop(p); fosse trocada por a += a;

III. Em relação ao vazamento de memória (*memory leak*), é opcional chamar a função free(p), pois o vetor usado pela pilha é alocado estaticamente.

É correto o que se afirma em

A. I, apenas.

B. III, apenas.

C. I e II, apenas.

D. II e III, apenas.

E. I, II e III.

1. Introdução teórica

1.1. Estruturas de dados: pilha

Considere um restaurante do tipo *self service*, no qual se tem uma pilha de pratos na entrada. Idealmente, pode-se apenas retirar os pratos do topo da pilha. Os pratos que estão mais embaixo do prato do topo estão indisponíveis, até que se consiga retirar todos os pratos superiores, um por um. Além disso, suponha também que o garçom possa apenas adicionar pratos no topo da pilha. Assim, um prato novo é adicionado ao topo da pilha e esse mesmo prato é o prato que seria retirado caso algum cliente quisesse retirar um da pilha.

Este cenário, no qual adicionamos e retiramos pratos de uma pilha de pratos em um restaurante, é bastante similar à estrutura de dados pilha, utilizada na computação. Na pilha (da computação), podemos apenas adicionar ou retirar elementos do seu topo. Contudo, diferentemente da pilha de pratos, em uma pilha computacional pode-se adicionar diferentes dados. Pode-se ter uma pilha de números inteiros, em que cada elemento pode ser um número inteiro diferente. De forma similar à pilha de pratos, pode-se retirar o número que está no "topo" e adicionar elementos ao topo da pilha.

Apesar das suas limitações, as pilhas são estruturas de dados extremamente úteis e versáteis. Devido à sua simplicidade, pilhas podem ser implementadas em *hardware* sendo muito rápidas.

Pilhas podem ser implementadas de diversas formas, com estruturas estáticas ou dinâmicas de memória. Quando utilizamos estruturas dinâmicas, podemos fazer com que a memória cresça ou diminua conforme o uso da pilha. Com o uso de estruturas estáticas, o tamanho da memória é fixo. Portanto, devemos atentar para o número máximo de elementos que serão inseridos na pilha, nesse caso.

1.2. Alocação dinâmica de memória

Programas ocupam espaço na memória do computador. Às vezes, sabe-se antecipadamente qual o espaço necessário para a execução de um programa. Outras vezes, não se tem certeza de qual seria o tamanho ideal da memória, mas é preciso definir um valor máximo. Podemos exemplificar isso pensando no caso de uma variável que armazena um nome de uma pessoa. Alguns nomes são bem pequenos, como, por exemplo, "Ana" ou "José". Outros nomes podem ser maiores, como "Evangelina" ou "Valentino". Cada letra deve ser armazenada em um espaço de memória, o que significa que nomes maiores devem ocupar mais espaço na memória. Qual é o tamanho máximo que se deve utilizar?

Se for reservado um espaço fixo grande, tem-se uma vantagem: podem ser utilizados nomes de tamanhos grandes ou pequenos. Por outro lado, se na maior parte do tempo os nomes não forem grandes, na maior parte das vezes haverá desperdício de memória. Como estamos reservando espaços fixos, mesmo um nome pequeno como "Ana" vai ocupar um espaço grande, mas com muitos caracteres vazios depois das letras "A", "n" e "a". Dessa forma, reservar grandes espaços de memória mesmo sem se ter certeza da necessidade leva a um grande desperdício da memória utilizada.

As situações descritas até aqui são chamadas de "alocação estática de memória", pois o programador aloca (separa) uma quantidade fixa de memória para a execução do programa.

Porém há situações em que não se tem ideia do tamanho máximo de memória que pode vir a ser utilizado pelo programa. Isso ocorre em muitos programas nos quais a atividade do usuário pode levar à geração de grandes quantidades de dados que devem ser armazenados em memória durante a execução do programa.

Para tanto, foi desenvolvida outra forma de trabalhar com a memória de um programa: alocação dinâmica de memória. Nesse caso, o programa solicita ao sistema operacional memória durante a execução do programa, sempre que necessário. Se o

computador dispuser da quantidade solicitada de memória no instante da solicitação, o programa recebe um espaço de memória adicional, que pode ser usado para ampliar o armazenamento de dados na memória do computador.

Contudo, um computador sempre apresenta uma quantidade finita de memória disponível. Essa quantidade pode ser extremamente grande, mas é sempre finita. Como muitos programas podem facilmente solicitar grande quantidade de memória simultaneamente (supondo um sistema operacional multitarefa, o que é a realidade na maior parte dos casos), isso pode levar ao consumo de toda a memória disponível. Algumas técnicas podem tentar aumentar esse limite consideravelmente, no entanto, sempre há um limite máximo e o impacto no desempenho pode ser grande. Dessa forma, é importante que um programa seja capaz não apenas de solicitar mais memória para o sistema operacional, mas também devolver a memória que esse programa solicitou no passado e que não está mais utilizando. Algumas linguagens podem ter algoritmos automáticos que permitem a devolução da memória, mas a linguagem C requer que o programador gerencie manualmente a memória sendo utilizada.

Na linguagem C, há, essencialmente, três funções para a alocação dinâmica de memória: malloc, calloc e realloc. Essas três funções permitem que o programador aloque uma quantidade de memória em bytes (no caso, os comandos malloc e calloc) ou redimensione um espaço de memória previamente solicitado (realloc). A função free faz o caminho inverso, devolvendo a memória que tenha sido solicitada anteriormente (por qualquer um dessas funções, seja a malloc, calloc ou realloc).

2. Análise das afirmativas

I – Afirmativa correta.

JUSTIFICATIVA. É fácil observar que tanto na função push quanto na função pop, o número máximo de comandos a serem executados é sempre o mesmo, independentemente dos argumentos dessas funções. Logo, podemos afirmar que essas funções apresentam complexidade computacional O(1).

II – Afirmativa correta.

JUSTIFICATIVA. Imediatamente antes da linha *a+=pop(p)*, temos a linha *push(p, a)*, ou seja, depositamos no topo da pilha o valor de *a*. Dessa forma, no momento em que

executamos *a+=pop(p),* estamos executando *a=a+pop(p)*, mas *pop(p)* é igual a *a*. Dessa forma, poderíamos fazer *a=a+a*, que é equivalente a *a+=a.*

III – Afirmativa incorreta.
JUSTIFICATIVA. Observe que, na função *cria_pilha,* a pilha é alocada na memória com o comando *malloc*, ou seja, a memória é alocada dinamicamente. Isso significa que devemos controlar alocação e a liberação da memória manualmente, ou seja, devemos utilizar o comando *free* para liberar a memória quando ela não for ser mais utilizada. Dessa forma, essa afirmativa está incorreta.

Alternativa correta: C.

3. Indicação bibliográfica

- CELES, W.; CERQUEIRA, R.; RANGEL, J. L. *Introdução à estrutura de dados.* Rio de Janeiro: Elsevier Brasil, 2004.

ÍNDICE REMISSIVO

Questão 1	Programação. Multi-threads. Paralelismo. Compartilhamento de recursos. Sistemas operacionais
Questão 2	Banco de dados. Consultas SQL.
Questão 3	Lógica. Regras de inferência. Proposição condicional.
Questão 4	Classes de complexidade computacional de algoritmos. Problema P versus problema NP.
Questão 5	Comunicações digitais. Análise combinatória. Detecção de erros.
Questão 6	Sistemas digitais. Portas lógicas. Multiplexadores.
Questão 7	Paradigmas de linguagens de programação. Programação lógica. Linguagem PROLOG.
Questão 8	Recursão. Lógica de programação. Linguagem C.
Questão 9	Inteligência artificial. Árvores de decisão. Redes neurais artificiais. Sistemas Especialistas.
Questão 10	Estruturas de dados. Alocação dinâmica de memória. Pilhas. Programação. Linguagem C.