UNIP
EDUCAÇÃO, POR SUA ESCOLHA

CIÊNCIA DA COMPUTAÇÃO

MATERIAL INSTITUCIONAL ESPECÍFICO

TOMO 16

CQA - COMISSÃO DE QUALIFICAÇÃO E AVALIAÇÃO

CIÊNCIA DA COMPUTAÇÃO

MATERIAL INSTRUCIONAL ESPECÍFICO

TOMO 16

Christiane Mazur Doi

Doutora em Engenharia Metalúrgica e de Materiais, Mestra em Ciências - Tecnologia Nuclear, Especialista em Cálculo e Matemática Aplicada, Especialista em Estatística Aplicada e Ciência de Dados, Especialista em Língua Portuguesa e Literatura, Especialista em Recursos Gramaticais para Revisão Textual, Engenheira Química e Licenciada em Matemática, com Aperfeiçoamento em Estatística e MBA em Engenharia Econômica. Professora titular da Universidade Paulista.

Anderson Aparecido Alves da Silva

Doutor em Engenharia Elétrica – Sistemas Eletrônicos, Mestre em Engenharia da Computação e pós-graduado em Administração de Empresas com ênfase em Análise de Sistemas. Professor titular da Universidade Paulista.

Tiago Guglielmeti Correale

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

Questão 1

Questão 1.[1]

Leia o texto a seguir.

Duas técnicas comumente utilizadas para ampliar as informações básicas sobre requisitos são personas e cenários. Frequentemente usadas juntas, essas técnicas se complementam de forma a trazer detalhes realísticos que possibilitam ao desenvolvedor explorar as atividades atuais do usuário, uso futuro de novos produtos e visões futuristas de novas tecnologias. Elas também podem guiar o desenvolvimento ao longo do ciclo de vida do produto.

ROGERS, Y.; PREECE, J.; SHARP, H. *Interaction Design: beyond human-computer interaction*. 5. ed. Indianapolis, IN, USA: John Wiley & Sons, Inc., 2019 (com adaptações).

Com base no texto apresentado e sobre os objetivos do uso de personas e cenários em um processo de elicitação de requisitos, avalie as afirmativas.

I. O uso de personas e cenários, em um processo de elicitação, explicita algumas situações que aparecem implícitas nos requisitos.

II. O uso de personas e cenários, em um processo de elicitação, ajuda o projetista a entender melhor o impacto das decisões de projeto.

III. O uso de personas e cenários, em um processo de elicitação, facilita a especificação formal e não-ambígua dos requisitos de interação.

IV. O uso de personas e cenários, em um processo de elicitação, lembra à equipe de desenvolvimento que pessoas reais usarão o produto.

É correto apenas o que se afirma em

A. I e III.

B. I e IV.

C. II e III.

D. I, II e IV.

E. II, III e IV.

1. Introdução teórica

1.1. Elicitação de requisitos

A elicitação de requisitos é o processo pelo qual analistas e desenvolvedores coletam, identificam, compreendem e documentam os requisitos de um sistema.

[1]Questão 19 – Enade 2021.

O processo de elicitação de requisitos é crucial para o sucesso do desenvolvimento de sistemas, pois ajuda a garantir que a solução proposta atenda às expectativas dos usuários e aos objetivos do projeto. Esse processo geralmente envolve a interação entre os membros da equipe de desenvolvimento e as partes interessadas, como clientes, usuários finais e outros envolvidos no projeto.

Alguns métodos comuns de elicitação de requisitos são entrevistas, workshops, questionários, prototipagem, observação do usuário e análise de documentos existentes. O objetivo é obter uma compreensão clara e abrangente das necessidades e dos requisitos do sistema, para que o desenvolvimento possa ser orientado de maneira eficaz e eficiente.

Existe uma divisão básica e fundamental entre dois tipos de requisitos: os requisitos funcionais e os requisitos não funcionais. Vamos relembrar brevemente a definição desses dois tipos de requisitos.

Os requisitos funcionais devem descrever, de forma clara e detalhada, todas as funcionalidades que o software deve oferecer aos seus usuários. Em outras palavras, os requisitos funcionais especificam as funções e as características que o sistema deve fornecer para atender às necessidades dos usuários e podem ser usados como base para diversas atividades, como projeto, desenvolvimento, testes, validação e verificação do sistema.

Os requisitos não funcionais são critérios que descrevem características do sistema que não se referem diretamente às suas funcionalidades específicas, mas definem as qualidades globais, as restrições ou as propriedades que o sistema deve ter.

É natural que exista uma preocupação especial com os requisitos funcionais, já que são esses requisitos que vão caracterizar o funcionamento do sistema. Um software que não atende aos seus requisitos funcionais vai ser percebido pelo usuário como inacabado, limitado ou até mesmo inútil, uma vez que o programa pode não ser capaz de atender às expectativas mínimas de funcionalidade. Contudo, não podemos nos esquecer da importância dos requisitos não funcionais. Esses requisitos são igualmente cruciais para o sucesso do produto, uma vez que impactam aspectos como desempenho, segurança, usabilidade e confiabilidade de um sistema. Frequentemente, programas que não atendem aos requisitos não funcionais especificados podem ser percebidos pelo usuário como defeituosos, lentos, complexos ou não confiáveis, o que diminui a qualidade percebida pelo cliente.

1.2. Personas e cenários

O termo **persona,** na computação, surgiu pela primeira vez em 1998, no livro "Os loucos estão controlando o sanatório" (COOPER, 1998).

Alan Cooper começou a desenvolver a técnica de criação de personas após entrevistar colegas que seriam usuários dos softwares que desenvolvia. A técnica passou a ser amplamente adotada nas organizações pelas diversas vantagens que apresentava, sendo utilizada como uma forma de dar à equipe do projeto um "modelo de cliente" com o objetivo de ajudar a compreender seu comportamento e sua forma de pensar.

Basicamente, a técnica de criação de persona descreve um perfil de usuário ou cliente a fim de entender as suas necessidades. Quando se pensa na criação de perfis, o foco deve estar nos gostos, nos interesses, nos hábitos ou até mesmo nos sonhos da persona criada.

Muitas vezes, a criação de uma persona atinge um nível de profundidade maior do que simplesmente o desenvolvimento de um produto para um público-alvo específico, pois o produto criado pode atingir um grupo maior de pessoas que tenham algumas características em comum. Por exemplo, um jogo que originalmente tinha como foco determinada faixa etária pode se tornar popular para um grupo maior de pessoas, de outras faixas etárias, mas com interesses similares.

Os cenários buscam elucidar as situações envolvendo as personas, mapeando as necessidades delas e identificando os requisitos e/ou as características que o produto ou o serviço deve conter, como durabilidade, agilidade e facilidade no uso.

Unindo os conceitos de persona e cenário, é possível criar um contexto sólido do uso e da aplicabilidade dos produtos, tornando-os realmente funcionais e úteis para os clientes finais. A combinação das técnicas também possibilita que se pense nas diferentes formas de uso dos produtos antes mesmo da sua criação, prevendo alguns dos seus aspectos, características e consequências ainda na fase de concepção dos requisitos.

Isso permite que a solução em desenvolvimento atenda às necessidades reais dos usuários e gere economia de tempo em correções em momentos posteriores ou até mesmo que o produto seja um fracasso no momento do lançamento.

2. Exemplo de uso

Como a criação de personas e cenários pode ser fundamental para compreender melhor as necessidades dos usuários e auxiliar no planejamento do seu desenvolvimento, segue um exemplo simples de criação e uso desses conceitos.

Persona

Nome: Ana Silva

Profissão: gerente de projetos

Características: Ana é uma profissional experiente que lida com diversos projetos simultaneamente. Ela valoriza a eficiência e precisa de uma visão clara do progresso de cada projeto sob sua responsabilidade. Ana é ocupada e precisa de um sistema que seja fácil de usar e que forneça informações rapidamente.

Cenário

Contexto: Ana está iniciando um novo projeto com uma equipe distribuída.

Objetivo: Ana precisa atribuir tarefas aos membros da equipe, acompanhar o progresso e garantir que todos estejam alinhados com os objetivos do projeto.

Uso de personas

O desenvolvedor do sistema cria funcionalidades específicas que atendem às necessidades de Ana. Isso ajuda na criação de uma visão consolidada do progresso do projeto e permite uma atribuição de tarefas de forma mais rápida e eficiente, além de fornecer opções de comunicação integradas.

Uso de cenários

O desenvolvedor pode criar cenários de uso específicos para simular como Ana poderia interagir com o sistema em situações do dia a dia. Por exemplo, um cenário pode envolver Ana atribuindo uma tarefa a um membro da equipe, acompanhando o status e recebendo notificações automáticas quando houver atualizações. Ao incorporar

personas e cenários no desenvolvimento do sistema, a equipe pode garantir que o produto final atenda de maneira mais precisa às necessidades dos usuários, tornando-o mais eficaz e agradável de usar.

3. Análise das afirmativas

I – Afirmativa correta.
JUSTIFICATIVA. O uso das técnicas citadas permite que a equipe envolvida em um projeto seja capaz de prever situações no cotidiano dos usuários que podem ser eventualmente ignoradas no levantamento de requisitos.

II – Afirmativa correta.
JUSTIFICATIVA. O uso de personas e cenários oferece uma maneira mais direta e simples de compreensão das necessidades reais dos usuários para todos os envolvidos em um projeto de software. Isso inclui, obviamente, o projetista, que pode tomar decisões de maneira mais adequada.

III – Afirmativa incorreta.
JUSTIFICATIVA. O uso das técnicas como personas e cenários não tem como foco a especificação formal, mas sim auxiliar as pessoas envolvidas no projeto e no desenvolvimento de um software no entendimento das necessidades do usuário final.

IV – Afirmativa correta.
JUSTIFICATIVA. O uso dessas técnicas permite aos integrantes da equipe de projeto terem clareza com relação às necessidades e às aspirações dos usuários finais. Isso permite que a equipe de desenvolvimento tenha mais consciência do valor do produto que deve ser entregue. Uma forma de destacar a importância do usuário é ter em mente que os produtos estão sendo feitos para que outras pessoas o utilizem e tenham suas vidas impactadas de modo positivo, com algo que realmente vai corresponder às suas necessidades.

Alternativa correta: D.

4. Indicações bibliográficas

- COOPER, A. *The inmates are running the asylum*. In: Software-Ergonomie'99. Vieweg+ Teubner Verlag, Wiesbaden, 1999.
- CRUZ, P. G. T. *Uso de dados para a construção de personas e geração de estratégias de marketing digital: case Kamoa*. Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro de Comunicação e Expressão, Graduação em Design, Florianópolis, 2021.
- FAUSTINO, P. *Marketing digital na prática: como criar do zero uma estratégia de marketing digital para promover negócios ou produtos*. São Paulo: DVS Editora, 2019.
- NIELSEN, L. *Personas-user focused design*. London: Springer, 2013.
- PRESSMAN, R. S. *Engenharia de Software*. 6. ed. São Paulo: McGraw-Hill, 2006.
- SOMMERVILLE, I. *Engenharia de Software*. 8. ed. São Paulo: Addison-Wesley, 2007.
- VALENTE, M. T. *Engenharia de Software Moderna – Princípios e Práticas para Desenvolvimento de Software com Produtividade*, 2020.
- VELHO, A. L. et al. A importância do marketing e suas estratégias no mercado atual. *Seminário de Iniciação Científica e Seminário Integrado de Ensino, Pesquisa e Extensão (SIEPE)*, p. e25589-e25589, 2020.

Questão 2

Questão 2.[2]

No projeto de redes de computadores, a escolha racional do dispositivo de conexão a ser utilizado é fundamental para o correto funcionamento da rede, bem como para a sua segurança e eficiência. Dispositivos como repetidores, hubs, bridges, switches, roteadores e gateways são muito comuns, mas diferem entre si em detalhes sutis e não muito sutis. Por existir uma grande quantidade desses dispositivos, vale a pena conhecer suas características principais, entender o seu funcionamento e saber quando e como são utilizados. A chave para entender esses dispositivos é observar que eles operam em camadas diferentes, como ilustra a figura 1. A camada é importante, porque diferentes dispositivos utilizam fragmentos de informações diferentes para decidir como realizar a comutação. Em um cenário típico, o usuário gera alguns dados a serem enviados para uma máquina remota. Esses dados são repassados à camada de transporte, que então acrescenta um cabeçalho (por exemplo, um cabeçalho TCP) e repassa o pacote resultante à camada de rede situada abaixo dela. Essa camada adiciona seu próprio cabeçalho para formar um pacote da camada de rede (por exemplo, um pacote IP). Na figura 2, vemos o pacote IP sombreado. Em seguida, o pacote vai para a camada de enlace de dados, que adiciona seu próprio cabeçalho e seu checksum (CRC) e entrega o quadro resultante à camada física para transmissão, digamos, por uma LAN.

Figura 1. Dispositivos presentes em cada camada.

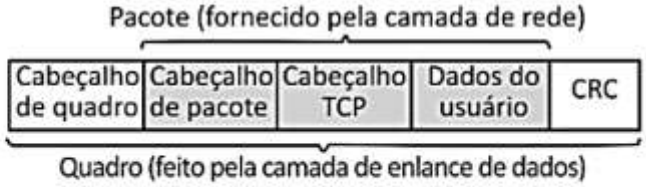

Figura 2. Quadros, pacotes e cabeçalhos.

TANNEBAUM, A. S. WETHERALL, D. *Redes de computadores.* 5. ed. São Paulo: Pearson Prentice Hall, p. 213 e 214, 2011 (com adaptações).

Considerando o contexto das informações e da figura apresentadas, assinale a alternativa correta.

[2]Questão 21 – Enade 2021.

A. Os repetidores não reconhecem quadros ou pacotes, apenas o seu próprio cabeçalho.
B. Um hub tem várias interfaces de entrada/saída conectadas eletricamente; os quadros que chegam a qualquer uma dessas interfaces são enviados a todas as outras e, se dois quadros chegarem ao mesmo tempo, eles serão colocados em buffer de espera e arbitragem de enlace.
C. Uma bridge conecta duas ou mais redes, diferentemente de um hub, cada porta é isolada das demais para criar um domínio próprio de colisão; ela só envia o quadro à porta onde ele é necessário, e pode encaminhar vários quadros ao mesmo tempo, além de examinar o campo de carga útil (pacotes de rede) dos quadros que encaminha, para obter o endereço do destinatário.
D. Os roteadores examinam os endereços em pacotes e efetuam o roteamento com base nesses endereços, de modo que eles só trabalham com os protocolos para os quais foram projetados para lidar; nas redes de broadcast, o problema de roteamento é mais complicado e cabe à camada de rede operar com algoritmos de roteamento apropriados.
E. Os gateways de transporte conectam dois computadores que utilizam diferentes protocolos de transporte orientados a conexões, por exemplo, um computador que utiliza o protocolo TCP/IP orientado a conexões pode se comunicar com um computador que utiliza um protocolo de transporte orientado a conexões diferentes, chamado SCTP.

1. Introdução teórica

1.1. Protocolo TCP/IP

Para a comunicação em rede, sistemas finais trocam informações entre si. O tipo de mensagens ou de informações trocadas depende do que foi projetado e do protocolo definido para a troca de informações. Para prover essa comunicação, há uma série de equipamentos que atuam dentro dos modelos padronizados de camadas e, assim, possibilitam a interoperabilidade. A adoção de camadas é importante para a definição das interfaces e dos serviços entre equipamentos e aplicativos bem como para identificar as informações necessárias em cada nível e para prover o encaminhamento dos dados para o destino correto.

Um padrão internacionalmente adotado é o modelo TCP/IP, que proporciona a comunicação da aplicação (endpoints) enviando informações entre as suas diversas camadas: aplicação, transporte, rede, enlace e física.

Vale destacar que nem todos os autores consideram a camada física na descrição da pilha de protocolos TCP/IP. Contudo, como a sua existência pode ser vista como implícita pelo modelo, alguns autores costumam adicionar essa camada.

Na criação de um pacote para envio, cada camada (iniciando pela camada de aplicação) realiza uma função independente, acrescentando um cabeçalho próprio e repassando o pacote à camada subsequente. Por fim, a camada de enlace envia o pacote completo ao destinatário através da camada física. Esse processo é chamado de encapsulamento. O processo inverso é realizado pelo destinatário, que recebe o pacote na ordem inversa das camadas (iniciando pela camada de enlace) e retira os cabeçalhos à medida que avança entre as camadas, até que os dados enviados sejam recebidos pela aplicação ou pelo processo especificado.

A independência das diversas camadas do modelo TCP/IP contribuiu para a natureza distribuída da Internet. Essa característica permite que seja utilizada uma grande variedade de equipamentos na construção de uma rede de computadores e na interligação dessas redes, dependendo de vários fatores, como performance, custo e simplicidade. Tal diversidade faz com que a informação, ao ser enviada da sua origem até o seu destino, possa passar por uma grande gama de equipamentos e tecnologias diferentes. Nas próximas seções, vamos explorar brevemente alguns dos equipamentos mais utilizados.

1.2. Repetidor

Um repetidor é um dispositivo que opera na camada física do modelo OSI (Open Systems Interconnection) e é projetado para estender o alcance de uma rede local (LAN) ou de outros tipos de redes.

O principal propósito de um repetidor é amplificar e regenerar sinais de dados, permitindo que estes percorram distâncias maiores sem degradação significativa. Um sinal fraco ou atenuado é amplificado e transmitido novamente. Isso é particularmente útil em redes com cabos de cobre, em que a qualidade do sinal pode diminuir ao longo de distâncias maiores devido à atenuação. Apesar de regenerarem bits, eliminando distorções e ruídos da transmissão, vale mencionar que os repetidores não aumentam a taxa de transferência da rede, pois eles apenas amplificam os sinais existentes.

Com o avanço da tecnologia e o surgimento de outras soluções de rede, como switches e roteadores, os repetidores têm visto um declínio em sua utilização, especialmente em ambientes modernos. No entanto, eles ainda podem ser encontrados em situações específicas em que a extensão simples de um sinal é necessária.

1.3. Hub

Trata-se de um dispositivo que opera na camada física do modelo OSI com a função principal de fornecer um ponto centralizado para a conexão de computadores (topologia estrela), impressoras e outros dispositivos em uma rede.

Um hub transmite dados como sinais elétricos ou ópticos sem realizar qualquer tipo de processamento ou inteligência de rede. Assim, quando um dispositivo conectado a um hub envia dados, o hub retransmite esses dados para todas as portas (ou todos os dispositivos) conectadas a ele – esse processo é chamado de broadcast (difusão em português) e pode ser prejudicial, já que pode gerar congestionamento na rede com pacotes desnecessários.

Outra desvantagem do uso de hubs está nas colisões, que diminuem o desempenho da rede. Colisões são transmissões simultâneas de dois dispositivos que resultam na perda e na retransmissão dos pacotes.

Embora os hubs tenham sido amplamente utilizados no passado, eles são considerados obsoletos para muitas aplicações devido às suas limitações em termos de desempenho e de eficiência. Em redes modernas, switches são preferidos, devido à sua capacidade de encaminhamento inteligente e à minimização de colisões.

1.4. Switch

É um dispositivo de rede muito usado em topologias do tipo estrela que opera na camada de enlace (camada 2) do modelo OSI.

A principal função de um switch é encaminhar pacotes de dados dentro de uma LAN, tomando decisões com base nos endereços MAC (Media Access Control) dos dispositivos conectados a ele.

Entre as principais características dos switches, temos as explicadas a seguir.

- **Encaminhamento inteligente**: ao contrário dos hubs, que simplesmente retransmitem dados para todas as portas, um switch é capaz de aprender os endereços MAC dos dispositivos conectados a cada uma de suas portas. Isso permite que o switch tome decisões de encaminhamento inteligentes, direcionando os pacotes apenas para a porta específica onde o dispositivo de destino está localizado.
- **Eliminação de colisões**: devido às capacidades de aprendizado e de encaminhamento inteligente, os switches minimizam o risco de colisões na rede, o que resulta em um desempenho muito melhor em comparação com os hubs.

- **Segmentação de tráfego**: os switches podem segmentar o tráfego da rede, enviando dados apenas para as portas relevantes. Isso melhora a eficiência da rede e reduz a probabilidade de congestionamento.
- **Criação de Virtual LANs (VLANs)**: alguns switches suportam a criação de VLANs, que são redes lógicas dentro de uma única rede física. Isso permite a segmentação lógica de dispositivos, mesmo que fisicamente conectados ao mesmo switch.

1.5. Bridges

Assim como os switches, o principal propósito de uma bridge (a palavra bridge, da língua inglesa, pode ser traduzida para "ponte" em português) é conectar segmentos de rede diferentes, aprendendo e tomando decisões com base nos endereços MAC dos dispositivos conectados a ela.

Embora as bridges tenham sido mais proeminentes em redes mais antigas, os switches modernos acabaram por incorporar muitas das funções das bridges. Na prática, o termo "bridge" ainda é usado, mas, muitas vezes, é associado a dispositivos que conectam redes de tecnologias diferentes ou que têm características específicas relacionadas à segmentação de tráfego.

1.6. Roteadores

O roteador é um dispositivo que opera na camada de rede (camada 3) do modelo OSI.

A função principal de um roteador é encaminhar pacotes de dados entre redes diferentes, tomando decisões com base nos endereços IP dos dispositivos conectados e utilizando tabelas de roteamento.

Podemos destacar as características dos roteadores mostradas a seguir.

- **Encaminhamento de pacotes**: os roteadores são projetados para encaminhar pacotes de dados entre diferentes redes. Eles tomam decisões de roteamento com base nas informações contidas nos cabeçalhos dos pacotes, principalmente nos endereços IP de origem e de destino.
- **Conexão de redes**: roteadores são usados para conectar redes locais (LANs) diferentes ou para conectar uma LAN à Internet. Eles possibilitam a comunicação entre dispositivos em redes distintas.

- **Presença de tabelas de roteamento**: os roteadores mantêm tabelas de roteamento, que contêm informações sobre as rotas disponíveis para alcançar diferentes destinos. Essas tabelas são utilizadas para determinar a melhor rota para encaminhar os pacotes.
- **Endereçamento IP**: os roteadores operam com base em endereços IP. Eles examinam os endereços IP dos pacotes para tomar decisões de roteamento.
- **Realização de Network Address Translation (NAT)**: alguns roteadores têm a capacidade de realizar NAT, que permite que vários dispositivos em uma rede local compartilhem um único endereço IP público para se comunicarem com a Internet.
- **Incorporação de funcionalidades de firewall**: alguns roteadores incorporam funcionalidades de firewall, que podem ser usadas para filtrar tráfego com base em regras de segurança.
- **Controle de banda**: muitos roteadores oferecem recursos avançados, como o controle de banda, que permite priorizar ou limitar a largura de banda para determinados dispositivos ou serviços na rede.

Roteadores desempenham papel crítico na infraestrutura de redes, possibilitando a comunicação eficiente entre diferentes redes. Eles são fundamentais para a conectividade da Internet e para o roteamento eficiente de dados em ambientes complexos de rede.

1.7. Comutador de camada 4

Um comutador de camada 4, também conhecido como switch de camada 4, é um dispositivo de rede que opera na camada 4 do modelo OSI.

A camada 4, conhecida como camada de transporte, é responsável por estabelecer, manter e encerrar conexões de comunicação entre dispositivos em uma rede. Ela também é responsável pelo controle de fluxo, pela detecção de erros e pela retransmissão de dados, se isso for necessário. Os protocolos Transmission Control Protocol (TCP) e o User Datagram Protocol (UDP) são comumente associados a essa camada.

Um comutador desse tipo é capaz de tomar decisões com base nas informações contidas nos cabeçalhos dos pacotes da camada de transporte. Isso permite que o comutador faça o roteamento e o encaminhamento de dados com base em informações mais granulares, como portas de origem e destino, em vez de apenas endereços IP.

Esses tipos de dispositivos são mais avançados do que os comutadores de camada 2 (enlace) e 3 (rede) e podem fornecer recursos adicionais, como segmentação de tráfego com base em protocolos de aplicação. Essa capacidade de análise de dados em um nível mais

profundo na pilha de protocolos permite uma gestão mais eficiente do tráfego em redes de computadores.

2. Análise das alternativas

A - Alternativa incorreta.
JUSTIFICATIVA. Os repetidores não reconhecem quadros, pacotes nem cabeçalhos (do próprio equipamento ou de outro), mas apenas os símbolos codificados na interface física. Por exemplo, convertem bits como volts em um meio guiado (cabo).

B - Alternativa incorreta.
JUSTIFICATIVA. Os hubs não têm buffers nem efetuam qualquer tratamento de arbitragem dos dados. Em caso de colisão de quadros que chegarem simultaneamente, cabem aos protocolos das camadas envolvidas as ações necessárias, como a detecção e o reenvio do quadro.

C - Alternativa incorreta.
JUSTIFICATIVA. Ao receber um quadro, a bridge verifica o endereço físico de destino do cabeçalho e acessa uma tabela de mapeamento para saber para qual porta o quadro deve ser enviado. O quadro é enviado somente para a porta onde é necessário, sendo possível também o envio simultâneo de vários quadros. Entretanto, uma bridge não acessa os dados (carga útil) a fim de identificar endereços de outras camadas.

D - Alternativa incorreta.
JUSTIFICATIVA. Devido ao funcionamento em camadas da pilha de protocolos TCP/IP, equipamentos de rede podem trabalhar com diversos protocolos, mesmo que eles não tenham sido projetados especificamente para isso. Quando um roteador trabalha com um pacote, ele só precisa se preocupar com os cabeçalhos das camadas nas quais ele trabalha e das camadas inferiores (ou seja, com a camada de enlace e a camada de rede). As demais informações são vistas apenas como "dados" (para o roteador), mesmo que contenham, na realidade, cabeçalhos e informações de outros protocolos. Essa é uma forma de encapsulamento da informação, que proporciona grande flexibilidade no projeto e na utilização de sistemas complexos, como redes de computadores.

E - Alternativa correta.

JUSTIFICATIVA. O Stream Control Transport Protocol (SCTP) é um protocolo confiável e orientado à conexão que permite diferentes fluxos, bastante utilizado para a transferência confiável de dados sobre redes IP. Gateways de transporte conectam dois computadores que se utilizam de protocolos de transporte diferentes e podem ser usados para a conversão entre o SCTP e outros protocolos como o TCP.

3. Indicações bibliográficas

- COMER, D. E. *Interligação de redes com TCP/IP*. 6. ed. Rio de Janeiro: Elsevier, 2016.
- TANENBAUM, A. S.; FEAMSTER, N.; WETHERALL, D. J. *Redes de computadores*. 6. ed. [São Paulo]: Pearson; Porto Alegre: Bookman, 2021.

Questão 3

Questão 3.[3]

Leia o texto a seguir.

Um compilador é um software que traduz um programa descrito em uma linguagem de alto nível para um programa equivalente em código de máquina para um processador. Em geral, um compilador não produz diretamente o código de máquina, mas, sim, um programa em linguagem simbólica (assembly) semanticamente equivalente ao programa em linguagem de alto nível. O programa em linguagem simbólica é, então, traduzido para o programa em linguagem de máquina por meio de montadores. Para realizar esta tarefa, o compilador executa a análise léxica, sintática e semântica do código-fonte do programa que está sendo executado em linguagem abstrata para depois gerar o código de máquina.

BRANCO, G. A. JR.; TAMAE, R. Y. Uma breve introdução ao estudo e implementação de compiladores. *Revista Científica Eletrônica de Psicologia*. Ano V, n. 08, fev. 2008 (com adaptações).

Considerando as informações do texto, avalie as afirmativas.

I. O analisador sintático tem a função de verificar se a sequência de símbolos gerada pelo analisador léxico compõe um programa válido ou não.

II. Na análise léxica, o analisador irá identificar cada símbolo que tenha significado para linguagem, gerando a mesma classificação para Java, Pascal ou outra linguagem.

III. O analisador semântico utiliza o código fonte para verificar incoerências quanto ao significado das construções implementadas.

IV. A fase de otimização do código procura melhorar o código intermediário, visando a um código de máquina mais rápido em termos de execução.

É correto apenas o que se afirma em

A. I e IV.

B. II e III.

C. II e IV.

D. I, II e III.

E. I, III e IV.

1. Introdução teórica

1.1. Linguagens de programação

Uma linguagem de programação é um conjunto de regras e instruções usado para criar um software ou executar operações específicas em um computador. Ela fornece um meio para

[3]Questão 30 – Enade 2021.

os programadores expressarem o que querem de forma que um computador possa entender e executar.

As linguagens de programação seguem regras específicas de sintaxe que definem como os programas devem ser escritos. A sintaxe diz respeito à forma de construção de um programa, funcionando de forma similar às regras que utilizamos em linguagens humanas (como o português ou o inglês), mas de maneira muito mais precisa. A análise sintática verifica se determinado programa está de acordo com a gramática de uma linguagem de programação. Posteriormente, a análise semântica agrega mais informações ao processo de compilação.

Muitas linguagens contam com bibliotecas ou frameworks que fornecem conjuntos predefinidos de funções e ferramentas para facilitar o desenvolvimento de software. Elas também podem ser categorizadas em paradigmas de programação, como imperativas, orientadas a objetos e funcionais, entre outros. Cada paradigma tem diferentes abordagens para a resolução de problemas e estruturação de código. Diferentes paradigmas podem simplificar a solução de problemas específicos, o que contribui para a existência de inúmeras linguagens de programação. Cada linguagem tem características, vantagens, desvantagens e finalidades específicas.

A seguir, exemplos de linguagens de programação.

- **Linguagem C**: linguagem de programação de propósito geral amplamente utilizada, conhecida por sua eficiência, sua simplicidade e sua portabilidade.
- **Python**: linguagem de alto nível, interpretada, de propósito geral, usada em diversas aplicações, desde o desenvolvimento web até a análise de dados e a inteligência artificial.
- **Java**: linguagem orientada a objetos muito utilizada para o desenvolvimento de aplicativos empresariais, desenvolvimento web e aplicações Android, entre outros.
- **JavaScript**: linguagem interpretada usada principalmente para desenvolvimento web. Ela foi inicialmente criada com o propósito de permitir interatividade em páginas da web, mas, com o tempo, foi sendo modificada e estendida para atender a uma ampla gama de necessidades, especialmente com a utilização de ambientes como o NodeJS.

A escolha de uma linguagem de programação para um projeto envolve vários fatores, muitos deles de natureza técnica. Contudo, outros fatores também devem ser levados em consideração. Por exemplo, fatores econômicos, como o número de programadores que conhecem determinada linguagem (sua popularidade) impactam diretamente no custo de um projeto. Muitas vezes, as escolhas são influenciadas tanto pelas necessidades específicas do projeto quanto pelas preferências dos desenvolvedores.

1.2. Compilação

É importante saber que um computador trabalha, internamente, com uma linguagem conhecida como "linguagem de máquina". Ela é composta de instruções que podem ser diretamente lidas e executadas pelo microprocessador, sendo específicas para cada plataforma. Essas instruções estão tipicamente armazenadas na memória do computador, em um formato binário.

Podemos dizer que seria teoricamente possível para uma pessoa trabalhar diretamente em linguagem de máquina, escrevendo bits na memória do computador, mas isso raramente acontece. Escrever um programa complexo diretamente em linguagem de máquina é algo demasiadamente trabalhoso. Além disso, a manutenção desse tipo de programa seria muito cara e demorada.

Para auxiliar no desenvolvimento de programas, ao longo do tempo foram sendo criadas diferentes linguagens de programação. Essas linguagens são muito mais fáceis de entender do que a linguagem de máquina e foram desenvolvidas para simplificar a solução de alguns tipos de problemas e para ter certa similaridade com as linguagens humanas.

Contudo, isso gerou um novo problema: o computador trabalha internamente com uma linguagem diferente da linguagem que é utilizada para escrevermos os programas. Enquanto o computador entende apenas a linguagem de máquina, um programa pode ser escrito em uma outra linguagem de programação, como C ou Pascal. Dessa forma, surge a questão de de "converter" ou "traduzir" um programa de uma linguagem original para a linguagem de máquina. Existem várias abordagens possíveis para solucionar essa questão, como a abordagem seguida pelas linguagens interpretadas e a abordagem seguida pelas linguagens compiladas.

Programas que estão com a sintaxe e a semântica originais da linguagem nas quais foram criados são chamados de programas-fonte (ou, mais popularmente, código-fonte). Esses programas podem ser alterados por qualquer um que entenda a linguagem original do programa.

Algumas linguagens são interpretadas, o que significa que o código-fonte é lido e executado diretamente por um interpretador em tempo de execução. Outras linguagens são compiladas.

A compilação é um processo no qual o código-fonte original é traduzido para código objeto ou de máquina antes da sua execução. Isso melhora o desempenho e dificulta a leitura

e a alteração do código-fonte por outra pessoa, já que a linguagem de máquina é mais complexa, e vai ser executada diretamente pelo processador.

Todo o processo de compilação é realizado por um compilador, que é, na verdade, um software encarregado de traduzir um programa em linguagem-fonte (mais simples de ser compreendido por um ser-humano) para a linguagem de máquina (mais difícil de ser compreendido por um ser humano). Basicamente, o processo de compilação é dividido em duas fases: análise e síntese.

1.2.1. Fase de análise

Na fase de análise, o programa é dividido em pedaços, seguindo uma estrutura padronizada que é uma representação intermediária do programa.

Inicialmente, o compilador lê o código-fonte e verifica se este código está de acordo com as regras da linguagem. Eventuais erros são repassados para correção pelo usuário.

A fase de análise do processo de compilação é dividida em três partes, explicadas a seguir.

- **Análise léxica**: lê a entrada (código-fonte) e produz uma série de sequências, chamadas de lexemas. A partir dos lexemas, o compilador produz tokens e acrescenta rótulos a eles. Esses rótulos vão ser úteis na fase posterior (a análise sintática). Além disso, é possível que algumas informações sejam adicionadas na tabela de símbolos.
- **Análise sintática**: recebe e processa os tokens com o intuito de reconhecer regras sintáticas. Também notifica o usuário dos eventuais erros sintáticos.
- **Análise semântica**: verifica a consistência semântica da linguagem usada e avisa ao usuário se erros semânticos estão sendo gerados. Leva em conta a análise de contexto e o gerenciamento da tabela de símbolos e realiza uma checagem de tipos, que consiste em uma verificação que identifica se os operadores têm os operandos adequados para a linguagem usada no código-fonte.

1.2.2. Fase de síntese

A fase de síntese é composta por três partes, mostradas a seguir.

- **Geração de código intermediário**: o código intermediário corresponde a uma representação do programa original mais fácil de ser traduzida para a linguagem de

máquina. Esse código também vai servir de entrada para a etapa de otimização, facilitando esse processo.

- **Otimização de código**: o compilador deve tentar aplicar uma série de técnicas e estratégias para melhorar o desempenho do programa original.
- **Geração de código de máquina**: o compilador traduz o código intermediário para o código de máquina específico da arquitetura do computador de destino. Esse código de máquina é o que o processador do computador pode executar diretamente.

Um compilador permite que os programadores escrevam código em uma linguagem de alto nível, mais próxima da linguagem humana, facilitando o desenvolvimento de software complexo.

Além disso, o uso de compiladores também auxilia na portabilidade dos programas: um mesmo código-fonte pode gerar binários executáveis para diferentes plataformas, sem a necessidade de que o código original seja reescrito para uma plataforma específica. Na prática, devido a questões como peculiaridades de uma plataforma específica ou aumento de desempenho, esse processo nunca é perfeitamente automático e simples, requerendo adaptações do código original. Contudo, o uso de compiladores capazes de gerar códigos para diversas plataformas é bastante comum e até mesmo fundamental em vários tipos de projetos.

2. Análise das afirmativas

I - Afirmativa correta.

JUSTIFICATIVA. Essa fase verifica a correção sintática do código a partir da sequência de tokens recebidos da análise léxica e da sua adequação à gramática da linguagem.

II - Afirmativa incorreta.

JUSTIFICATIVA. A geração dos tokens é muito dependente da linguagem específica em uso. De forma geral, podemos dizer que linguagens diferentes vão gerar um fluxo de tokens diferentes. Contudo, é interessante observar que, frequentemente, linguagens de programação apresentam comandos e características em comum (ou similares). Assim, é possível que a análise léxica chegue a resultados parecidos em alguns casos específicos. De qualquer forma, não podemos afirmar que o resultado vai ser sempre igual, especialmente nos casos das linguagens citadas na afirmação.

III - Afirmativa correta.

JUSTIFICATIVA. A análise semântica verifica justamente a consistência do código em relação à linguagem usada. Ela agrega informações que não podem ser facilmente obtidas nas etapas anteriores, como a análise léxica e a análise sintática. Um exemplo de cenário é a verificação da consistência com relação ao uso de tipos, muito comum em algumas linguagens de programação.

IV - Afirmativa correta.

JUSTIFICATIVA. Existem vários tipos de melhorias e otimizações possíveis. Muitas vezes, o objetivo da otimização vai ser uma escolha do programador. Há cenários em que o programador pode querer priorizar o desempenho do programa final, mesmo que isso implique ~~um~~ uso maior de memória. Em outras situações, o programador pode querer a situação inversa: priorizar a memória em relação ao desempenho (velocidade de execução).

Alternativa correta: E.

3. Indicações bibliográficas

- AHO, A. V. et al. *Compiladores: princípios, técnicas e ferramentas*. 2. ed. São Paulo: Pearson Addison, 2008.
- BARBOSA, C. S. et al. *Compiladores*. Porto Alegre: SAGAH, 2021.
- COOPER, K.; TORCZON, L. *Construindo compiladores*. 2. ed. Rio de Janeiro: Elsevier, 2014.

Questões 4 e 5

Questão 4.[4]

Observe o código abaixo escrito na linguagem C.

```
1   #include <stdio.h>
2   #define TAM 10
3   int funcao1(int vetor[], int v){
4        int i;
5        for (i = 0; i < TAM; i++){
6             if (vetor[i] == v)
7                  return i;
8       }
9        return -1;
10  }
11  int funcao2(int vetor[], int v, int i, int f){
12    int m = (i + f) / 2;
13    if (v == vetor[m])
14         return m;
15    if (i >= f)
16         return -1;
17    if (v > vetor[m])
18         return funcao2(vetor, v, m+1, f);
19    else
20         return funcao2(vetor, v, i, m-1);
21  }
22  int main(){
23    int vetor[TAM] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
24    printf("%d - %d", funcao1(vetor, 15), funcao2(vetor, 15, 0, TAM-1));
25    return 0;
26  }
```

A respeito das funções implementadas, avalie as afirmativas.

I. O resultado da impressão na linha 24 é: 7 - 7.

II. A função `funcao1`, no pior caso, é uma estratégia mais rápida do que a `funcao2`.

III. A função `funcao2` implementa uma estratégia iterativa na concepção do algoritmo.

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.

[4]Questão 20 – Enade 2021.

Questão 5.[5]

Existe um grande número de implementações para algoritmos de ordenação. Um dos fatores a serem considerados, por exemplo, é o número máximo e médio de comparações que são necessárias para ordenar um vetor com **n** elementos. Diz-se também que um algoritmo de ordenação é estável se ele preserva a ordem de elementos que são iguais. Isto é, se tais elementos aparecem na sequência ordenada na mesma ordem em que estão na sequência inicial. Analise o algoritmo abaixo, onde **A** é um vetor e "**i**, **j**, **lo** e **hi**" são índices do vetor.

```
algoritmo ordena(A, lo, hi):
      se lo < hi então:
            p := particao(A, lo, hi)
            ordena(A, lo, p - 1)
            ordena(A, p + 1, hi)

algoritmo particao(A, lo, hi):
      pivot := A[hi]
      i := lo
      repita para j := lo até hi:
            se A[j] < pivot então:
                  troca A[i] com A[j]
                  i := i + 1
      troca A[i] com A[hi]
      return i
```

Com relação ao algoritmo apresentado, avalie as afirmativas.

I. O algoritmo precisa de um espaço adicional O(n) para a pilha de recursão.

II. O algoritmo apresentado é um algoritmo de ordenação recursivo e estável.

III. O algoritmo precisa, em média, de O(n log n) comparações para ordenar n itens.

IV. O uso do primeiro elemento do vetor como "*pivot*" é mais eficiente que usar o último.

É correto apenas o que se afirma em

A. I e III.

B. II e IV.

C. III e IV.

D. I, II e III.

E. I, II e IV.

[5]Questão 32 – Enade 2021.

1. Introdução teórica

1.1. Linguagem C

A linguagem de programação C é uma linguagem de programação de propósito geral, criada por Dennis Ritchie no início da década de 1970 no laboratório Bell da AT&T. Ela foi desenvolvida como uma evolução da linguagem de programação B e influenciou muitas outras linguagens modernas, como C++, C#, Java e Python.

Apesar de ser bastante antiga, devido à versatilidade e ao desempenho, a linguagem C ainda é amplamente utilizada em uma variedade de aplicações, incluindo sistemas operacionais, drivers de dispositivos, compiladores, jogos e aplicativos de sistemas embarcados, entre outras.

Conhecida pela eficiência e pela proximidade ao hardware, a linguagem C é adequada para a programação de sistemas e para o desenvolvimento de software de baixo nível. Ela fornece um conjunto de instruções simples, uma sintaxe clara e um modelo de programação procedural, permitindo aos programadores um controle preciso sobre a manipulação de memória e operações que envolvem o hardware da máquina. Exemplo dessa eficiência pode ser notada no uso de aritmética de ponteiros, manipulação direta de memória, funções de biblioteca padrão e uma abordagem modular que incentiva a reutilização de código.

A portabilidade é outra característica fundamental, pois programas escritos em C podem ser compilados e executados em diferentes plataformas com relativamente poucas modificações.

1.2. Complexidade de algoritmos

Estimar a complexidade é fundamental para entender como o desempenho de um algoritmo se comporta à medida que variamos o tamanho da sua entrada. Podemos avaliar a complexidade de um algoritmo em termos do tempo de execução (chamada de complexidade temporal) ou em termos do consumo de memória (chamada de complexidade espacial).

Um questionamento bastante comum quando começamos a estudar o assunto de complexidade de algoritmos é o seguinte: por que não simplesmente executar um programa que implemente um dado algoritmo, medir o seu tempo de execução (ou tamanho em memória) e considerar isso como uma medida de desempenho? Por que devemos fazer análises teóricas e matemáticas complexas, em vez de uma simples medida?

Existem várias razões para a abordagem teórica tradicional, mas as medidas de desempenho de um programa também podem ser úteis. Contudo, elas são limitadas: em primeiro lugar, elas estão ligadas a dado equipamento, dado sistema operacional, dada linguagem de programação e até mesmo a dado estilo de programação. Assim, o tempo de execução de um programa real depende de vários fatores além do algoritmo que ele está implementando. Isso significa que é difícil de comparar, de forma honesta, medidas feitas em máquinas diferentes ou em linguagens diferentes, mesmo que o algoritmo em si seja o mesmo.

Outro ponto importante é que o interesse na caracterização da complexidade dos algoritmos está não apenas em identificar o seu tempo de execução para uma situação específica, mas em entender como o tempo de execução cresce com o tamanho da entrada. O foco principal está em caracterizar qual é o tipo de crescimento do algoritmo (por exemplo, do tempo de execução) conforme a entrada aumenta, sem necessariamente calcular o tempo exato.

A análise da complexidade foca apenas no algoritmo em si, sem se preocupar com a linguagem de programação específica, o sistema operacional ou o hardware no qual ele vai ser executado. Dessa forma, ao avaliarmos um algoritmo, temos uma estimativa segura e durável da complexidade de um algoritmo, que pode ser utilizada para comparação com outros algoritmos diferentes, mas que executam a mesma tarefa.

Uma das etapas na avaliação da complexidade de um algoritmo está em encontrar um modelo de computação específico, por exemplo, uma máquina de Turing determinística. Depois, devemos calcular o número de passos que essa máquina utiliza para processar uma entrada de tamanho n.

Contudo, muitas vezes um algoritmo pode levar tempos diferentes para processar duas entradas com o mesmo tamanho. Por exemplo, suponha que um algoritmo está buscando um item em um vetor não ordenado, varrendo o vetor elemento por elemento, do início até o fim. É possível que o item procurado esteja no início do vetor (terminando a busca muito rapidamente), no meio do vetor ou no fim do vetor (o pior caso possível). Assim, devemos decidir qual é o tipo de análise que queremos fazer: do pior caso (uma das análises mais utilizadas), do caso médio ou do melhor caso. Além disso, é possível também fazer análises estatísticas do algoritmo.

Uma vez que decidimos o tipo de análise, devemos obter uma função que caracteriza o tempo de execução (ou o espaço, caso estejamos interessados no consumo de memória) em função do tamanho da entrada. Como o nosso interesse está em entender o crescimento

dessa função com a entrada, utilizamos uma notação matemática especial, que foi desenvolvida especificamente para esse tipo de análise.

Um dos problemas com essa notação é que ela é parecida com a notação utilizada em outras áreas da Matemática, mas o significado de vários símbolos é um pouco diferente da "Matemática tradicional". Dessa forma, devemos tomar cuidado em como interpretar corretamente essa notação. Essa análise é chamada de análise assintótica, um nome peculiar e que pode induzir a uma avaliação equivocada, mas muito utilizado na literatura específica da área de algoritmos.

Entre as diversas ferramentas e notações utilizadas na área da análise de algoritmos, destacamos a chamada notação do "O grande" (representada pela letra "O" maiúscula). Para entender o conceito matemático por trás dessa notação, vamos supor duas funções f(n) e g(n). Essas funções podem representar o tempo de pior caso de um algoritmo, por exemplo. Como elas avaliam o algoritmo em termos do tamanho de uma entrada (dado pela variável n), o domínio da função é um número natural (pensemos em um vetor: ele pode ter zero elementos, um elemento, dois elementos ou n elementos, sendo n sempre um número Natural). A expressão matemática de f(n) ou g(n) pode ser qualquer, incluindo funções exponenciais e logarítmicas, entre outras.

Vale destacar que, o tempo de execução de um programa sempre vai ser um número positivo (não faz sentido dizer que um programa levou -2 segundos para ser executado), o que significa que estamos considerando apenas números reais positivos. Dizemos isso, matematicamente, fazendo:

$$f, g: N \rightarrow R^{+}$$

Suponha que existam dois inteiros positivos c e n_0, de forma que, para todo número inteiro $n \geq n_0$, a seguinte desigualdade seja sempre válida:

$$f(n) \leq cg(n)$$

Nesse caso, dizemos que g(n) é um limitante superior assintótico para f(n) e utilizamos a seguinte notação para exprimir esse fato:

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

De um ponto de vista matemático, devemos tomar cuidado com a interpretação do símbolo de igualdade utilizado na última expressão, pois ele deve ser visto mais como uma relação de "pertencimento de conjuntos" do que uma "igualdade matemática" propriamente dita.

1.3. Algoritmo de busca linear

Algoritmos de pesquisa sequencial ou busca linear percorrem uma lista (ou vetor de entrada) elemento por elemento, comparando-os com o valor que se deseja localizar. A coincidência indica que o elemento foi encontrado; caso contrário, examina-se o próximo item.

A pesquisa é finalizada quando encontramos o valor na lista de entrada ou quando não há mais elementos para serem pesquisados, e o item buscado não foi encontrado.

Como exemplo desse processo, considere um vetor de entrada com 10 elementos: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]. Suponha que desejemos verificar se o número 5 encontra-se nesse vetor e, em caso positivo, qual é o índice da sua posição. Nesse caso, o algoritmo de busca linear deve encontrar o valor 5 na 3° rodada. Veja a sequência a seguir.

1) Verifica-se o primeiro elemento do vetor, na posição 0, com o valor 1. Como 1 não é igual a 5, a busca continua para o próximo elemento.
2) O segundo elemento do vetor, na posição 1, é igual a 3, que é diferente de 5. Dessa forma, a busca continua para o próximo elemento.
3) O terceiro elemento do vetor, na posição 2, é o valor 5, que é igual ao número sendo pesquisado (5). Logo, o algoritmo retorna a posição (índice) desse elemento (no caso, 2), terminando a execução do algoritmo.

Considere um exemplo simples de um algoritmo de busca linear em pseudocódigo:

```
algoritmo buscaLinear(vetor, numeroProcurado)
     para cada índice i de 0 até (comprimento_do_vetor - 1) faça:
          se vetor[i] é igual a numeroProcurado então:
               retorne i //índice encontrado de numeroProcurado
     retorne -1 // numeroProcurado não está no vetor
```

Observe que o algoritmo apresentado percorre cada elemento do vetor, começando do primeiro elemento (na posição zero) e prosseguindo para as demais posições, verificando se cada elemento é igual ao valor da variável numeroProcurado (um dos argumentos passados na chamada do algoritmo). Se encontrar uma correspondência, o algoritmo retorna o índice (dado pela variável i) no qual o valor correspondente a "numeroProcurado" foi encontrado. Contudo, se após percorrer todo o vetor nenhum elemento correspondente for encontrado, o

valor "-1" é retornado como uma forma de indicar que o valor procurado não está presente no vetor de entrada (uma vez que -1 não é um índice válido de um vetor). Note também que estamos considerando que o primeiro elemento do vetor é dado pelo índice zero, e, por esse motivo, o último elemento é indexado pelo valor do comprimento do vetor menos um.

Em termos de eficiência, a busca linear tem complexidade de tempo O(n), com a variável n sendo igual ao número de elementos na lista (ou vetor) de entrada. Em grandes conjuntos de dados, outros algoritmos de busca, como a busca binária em listas ordenadas, são mais eficientes. Contudo, a busca linear é fácil de entender e implementar, sendo útil em situações simples, quando a lista é pequena ou quando os dados de entrada da lista não apresentam nenhum tipo de ordenação.

Na busca linear, o melhor caso ocorre quando o valor procurado está na posição zero (ou seja, é o primeiro elemento do vetor), pois isso resulta em um único acesso. Já o pior caso ocorre quando o valor procurado está na última posição ou não está presente no vetor de entrada, o que exige a verificação de todos os elementos do vetor.

1.4. Algoritmo de busca binária

Quando o vetor de entrada está ordenado, é possível utilizar algoritmos muito mais rápidos do que a busca linear, como o algoritmo de busca binária. Por exemplo, suponha um vetor ordenado em ordem crescente. Quando desejamos encontrar determinado número, o algoritmo deve checar o meio do vetor e comparar o valor lido com o valor do número procurado: se forem iguais, a busca encontrou o valor desejado. Se o valor lido for menor do que o número procurado, o algoritmo descarta a primeira metade do vetor (uma vez que o número procurado é maior) e repete o processo de divisão e comparação na metade restante do vetor. Caso o valor lido seja maior do que o número procurado, a segunda metade da parcela atual do vetor é descartada (porque o número procurado é menor). Esse procedimento de divisão e comparação repete-se até que o número procurado seja encontrado ou verifique-se que ele não está presente no vetor. Perceba que o algoritmo também pode ser facilmente utilizado se o vetor estiver com os seus elementos em ordem decrescente.

Como exemplo, considere um vetor ordenado de forma crescente, com 10 elementos:

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

Suponha que queremos buscar o índice do valor 5 contido nesse vetor utilizando a busca binária. Podemos perceber que a busca será satisfeita em três iterações do algoritmo, descritas a seguir.

1) Inicialmente, devemos encontrar o índice do elemento que divide o vetor de entrada ao meio. Para isso, somamos o índice da primeira posição (zero) e o índice da última posição (igual a 10-1=9), dividimos por dois e arredondamos o valor encontrado para baixo: (9+0)/2=4,5. Isso nos dá, arredondado para baixo, o valor 4. O valor encontrado na posição com índice quatro é igual a 9. Como 9 é maior do que 5, sabemos que o número 5, se estiver presente no vetor, deve estar na sua primeira metade, o que corresponde aos valores: [1, 3, 5, 7].
2) Aplicamos novamente o algoritmo na primeira metade do vetor, correspondente ao segmento dado por [1, 3, 5, 7]. Dividindo essa metade do vetor original ao meio, o valor encontrado é 3. Como 3 < 5, a segunda metade do vetor é mantida para pesquisa [5, 7].
3) Dividimos novamente o segmento restante ao meio e, como o valor encontrado nessa posição é 5 (observe que existem apenas dois elementos, os números 5 e 7), a pesquisa é finalizada.

Segue um exemplo básico de um algoritmo de busca binária em pseudocódigo.

```
algoritmo buscaBinaria(vetor, numeroProcurado)
      inicio := 0
      fim := comprimento_do_vetor - 1
      enquanto inicio <= fim, faça:
            meio := arredondaParaBaixo((inicio+fim)/2)
            se vetor[meio] é igual a numeroProcurado, então:
                  retorne meio
            se numeroProcurado < vetor[meio], então:
                  // numeroProcurado pode estar na metade inferior
                  fim := meio - 1
            senão:
                  // numeroProcurado pode estar na metade superior
                  inicio := meio + 1
      // nesse ponto sabemos que numeroProcurado não foi encontrado
      retorne - 1
```

Observe que o algoritmo começa com toda a lista e, a cada iteração do loop, compara o elemento desejado com o elemento no meio da lista. Se eles são iguais, o índice é retornado. Caso contrário, o algoritmo ajusta os limites (dados pelas variáveis "inicio" e "fim") com base na comparação para continuar a busca na metade apropriada. O processo é repetido até que o elemento seja encontrado ou a parte da lista onde ele possa estar seja reduzida a zero. Se o elemento não for encontrado, a função retorna -1 (um índice que não é válido).

A busca binária tem complexidade de tempo dada por O(log n), com n sendo o tamanho da lista (ou vetor) de entrada. Isso a torna muito eficiente, especialmente em listas grandes.

Nesse tipo de algoritmo, o melhor caso ocorre quando o valor procurado está no meio do vetor e exige apenas um acesso. O pior caso ocorre quando o vetor é continuamente dividido ao meio e o elemento procurado é o último a ser encontrado (ou, quando ele não está presente na lista).

Tendo em vista que, cada vez que o vetor é dividido, o tamanho dele se reduz à metade, é necessário saber quantas vezes determinado número (que expressa o tamanho do vetor) pode ser dividido por 2 antes de chegar a 1. O logaritmo na base 2 responde exatamente a esse questionamento. Portanto, para se calcular o pior caso de uma busca binária, basta aplicar a fórmula ($\log_2 n$).

Por fim, o caso médio de uma busca binária é expresso por ($\log_2 n$) – 1, o que significa que a sua complexidade também é dada por O(log n). Vale lembrar que, quando utilizamos a notação do "O-grande", não é necessário explicitar a base do logaritmo, uma vez que essa notação ignora fatores constantes.

1.5. Algoritmos de ordenação

Existem diversos algoritmos de ordenação. A função deles é realizar a ordenação de vetores ou listas de acordo com algum critério estabelecido, em ordem crescente ou decrescente.

Alguns dos algoritmos de ordenação mais comuns são os mencionados a seguir.

- ***Bubble Sort*** **(ordenação por bolha)**: este algoritmo compara elementos adjacentes e os troca se estiverem na ordem errada. Ele continua fazendo isso até que nenhuma troca seja necessária.
- ***Insertion Sort*** **(ordenação por inserção)**: este algoritmo constrói uma sequência ordenada de elementos um de cada vez. Para cada elemento, ele o compara com os elementos já ordenados e o insere na posição correta.
- ***Selection Sort*** **(ordenação por seleção)**: neste algoritmo, a cada passo, o elemento mínimo é selecionado da parte não ordenada da lista e trocado com o primeiro elemento não ordenado.
- ***Merge Sort*** **(ordenação por fusão)**: utiliza uma abordagem de divisão e conquista. Neste tipo de ordenação, o vetor original é dividido em duas partes, que

são ordenadas separadamente. Posteriormente, essas partes devem ser combinadas, finalizando o processo.

- ***Quick Sort* (ordenação rápida)**: também utiliza uma abordagem de divisão e conquista. Assim como em outras abordagens, o vetor (ou a lista) de entrada a ser ordenado é dividido em duas partes, mas isso é feito utilizando um elemento especial, chamado de pivô, que deve ser escolhido antes da divisão. A divisão do vetor de entrada é feita com base no pivô: uma das partes deve ficar com os elementos menores que o pivô, enquanto a outra parte deve ficar com os elementos maiores. O processo deve ser repetido em cada uma das partes, de forma recursiva, até que todo o vetor original esteja ordenado.
- ***Heap Sort* (ordenação por monte)**: utiliza uma estrutura de dados chamada *heap* ("monte" em português) para organizar os elementos. O algoritmo constrói uma "árvore de heap" (uma forma especial de árvore binária) e, em seguida, extrai os elementos na ordem desejada.
- ***Radix Sort* (ordenação por radix)**: este algoritmo ordena os elementos processando os dígitos individualmente. Ele começa pelos dígitos menos significativos e avança para os dígitos mais significativos, usando uma técnica de ordenação estável.

Cada algoritmo tem vantagens e desvantagens, além de diversos níveis de complexidade. Alguns são fáceis de serem entendidos e podem ser implementados de forma simples, enquanto outros requerem um estudo mais detalhado.

A escolha de um algoritmo de ordenação pode depender do contexto específico, como o tamanho da entrada, a distribuição dos dados e os requisitos de desempenho. Além disso, alguns algoritmos podem ser mais facilmente paralelizáveis, podem ter uma implementação mais rápida ou, alternativamente, podem ser mais facilmente implementados em dada plataforma.

De forma geral, algoritmos como o *Merge Sort* e o *Quick Sort* costumam ser eficientes em conjuntos de dados grandes (exceto em alguns casos específicos). Alguns algoritmos mais simples, como, por exemplo, o *Bubble Sort*, são raramente utilizados em programas reais, sendo mais usados mais como um tópico no estudo de algoritmos, devido à simplicidade e à facilidade de análise e de implementação.

2. Resolução das questões e análise das afirmativas

Questão 4

O programa apresentado no enunciado da questão aborda dois métodos de pesquisa de determinado valor dentro de um vetor.

A funcao1 é uma busca linear e recebe dois parâmetros:

- o primeiro parâmetro é um vetor com números inteiros em ordem crescente;
- o segundo parâmetro é o valor que se deseja localizar nesse vetor.

Essa função retorna o índice da posição do valor desejado dentro do vetor, caso esse valor esteja presente no vetor de entrada, ou -1, caso contrário. Lembre-se de que -1 não é um índice válido de um vetor (que deve ser sempre um valor maior ou igual a zero). Dessa forma, ao retornar um valor de índice inválido, a função é capaz de indicar que o elemento não foi encontrado no vetor de entrada.

A funcao2 é uma busca binária e recebe quatro parâmetros:

- o primeiro parâmetro é um vetor de números inteiros ordenados;
- o segundo parâmetro é o valor que se deseja localizar;
- o terceiro parâmetro é o índice inicial do vetor;
- o quarto parâmetro é o índice final.

A função retorna o índice da posição do valor procurado dentro do vetor ou -1 caso não o localize.

I - Afirmativa correta.

JUSTIFICATIVA. Na linha 23 do programa, o vetor de 10 posições (0 a 9) é definido com os valores 1, 3, 5, 7, 9, 11, 13, 15, 17, 19. A linha 24 exibe como resultado uma impressão com os resultados das funções 1 e 2 na busca do valor 15 dentro do vetor.

Tendo em vista que a função 1 é uma busca linear (consulta um elemento por vez) e o item 15 está na posição 7 (lembre-se de que, na linguagem C, a primeira posição de um vetor é a posição zero), o resultado impresso na linha 24 para a primeira função é 7.

Já na função 2, que implementa uma busca binária, o vetor é dividido ao meio continuamente até que o valor 15 seja encontrado na 2ª rodada.

1) Dividimos o vetor ao meio (com m igual a 4). Como o valor encontrado é 9, que é menor do que 15, a segunda metade do vetor é mantida para pesquisa: [11, 13, 15, 17, 19]. O algoritmo mostra que o valor 9 está na posição 4 do vetor.

2) Dividimos o vetor ao meio. Como o valor encontrado é 15, a pesquisa é finalizada na segunda rodada (nas linhas 13 e 14). O algoritmo mostra que o valor 15 está na posição 7 do vetor.

Portanto, os valores impressos na linha 24 são 7 e 7, representando, respectivamente, a rodada na qual o valor 15 foi encontrado pela funcao1 e a posição do valor 15 no vetor obtido pela funcao2.

II - Afirmativa incorreta.

JUSTIFICATIVA. No caso da funcao1 (que corresponde a uma busca linear), o vetor de entrada deve ser percorrido elemento por elemento. O pior caso ocorre quando o elemento buscado não está no vetor de entrada ou está na última posição. Nesses casos, o vetor deve ser completamente percorrido, levando a um número de comparações igual ao tamanho do vetor de entrada. Já no caso da funcao2 (busca binária que divide o vetor ordenado), o pior caso pode ser estimado pela fórmula $\lfloor(n\,) + 1\rfloor$, ou seja, o valor de $(n\,) + 1$ arredondado para baixo. Dessa forma, podemos fazer:

$$\lfloor(10\,) + 1\rfloor = \lfloor 3{,}322 + 1\rfloor = \lfloor 4{,}322\rfloor = 4$$

Na prática, isso significa que são necessárias 4 chamadas da função funcao2 no pior caso. Logo, a função mais eficiente (rápida) no pior caso é a função funcao2, e não a função funcao1, como indicado na afirmativa.

III - Afirmativa incorreta.

JUSTIFICATIVA. A função 2 não é iterativa, mas sim recursiva. Uma função recursiva, ao contrário de uma função iterativa, é aquela que chama a si mesmo em tempo de execução. Isso pode ser visto nas linhas 18 e 20, nas quais há chamadas para a própria funcao2.

Alternativa correta: A.

Questão 5

O algoritmo apresentado é uma implementação do *Quicksort*. Trata-se de um algoritmo de ordenação que utiliza uma estratégia do tipo "dividir para conquistar". Esse tipo de algoritmo divide um problema "grande" em subproblemas menores, até que a solução de cada problema seja muito simples. Geralmente, esse processo pode ser feito de forma recursiva.

O *Quicksort* baseia-se na ideia de que, em uma lista ordenada, cada elemento conta com a seguinte propriedade: todos vizinhos à esquerda são menores ou iguais ao seu valor, enquanto todos os vizinhos à direita são maiores ou iguais a esse valor.
A etapa de particionamento é feita escolhendo um elemento da lista chamado de pivô. Os elementos da lista são reordenados de maneira que haja duas partições: uma com elementos menores que o pivô e a outra com os elementos maiores. Esquematicamente, em termos dos índices da lista, temos o que segue.

Depois de cada rodada do algoritmo, o pivô já se encontra na sua posição final. Posteriormente, o *Quicksort* é aplicado de forma recursiva nas duas partições resultantes. O processo é então repetido até que todos os elementos da lista estejam ordenados.

I - Afirmativa correta.
JUSTIFICATIVA. O algoritmo apresentado é recursivo, ou seja, utiliza uma função que chama a si mesma diversas vezes. Cada chamada de uma função implica a criação de um quadro adicional na pilha de execução (a não ser em casos especiais, como em algumas otimizações que podem ser feitas no caso de *tail recursion*, ou recursão em cauda), levando ao cenário descrito na afirmativa.

II - Afirmativa incorreta.
JUSTIFICATIVA. O *Quicksort* é um algoritmo de ordenação recursivo que realiza trocas de posição dos elementos baseadas na comparação de cada elemento com o pivô. Contudo, a versão apresentada no enunciado não é estável.

III - Afirmativa correta.
JUSTIFICATIVA. A função `ordena`, de natureza recursiva e que adota uma estratégia "dividir para conquistar", tem uma complexidade de tempo média O(log n). Para cada nível de chamadas da função `ordena`, teremos no máximo n elementos processados, como podemos observar pelo laço `repita` dentro da função `particao` (que é chamada pela função `ordena`). Dessa forma, o algoritmo como um todo, precisa executar uma média de O(n log n) comparações.

IV - Afirmativa incorreta.

JUSTIFICATIVA. Existem diversas implementações do *Quicksort*, com uma série de estratégias para a escolha do pivô. Para a implementação apresentada, o pivô deve ser tomado como o último e não o primeiro elemento da lista.

Alternativa correta: A.

3. Indicações bibliográficas

- CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN C. *Algoritmos: teoria e prática*. 3. ed. Rio de Janeiro: Campus, 2012.
- DALE, N.; LEWIS, J. *Ciência da Computação*. 4. ed. Rio de Janeiro: LTC, 2011.
- GOODRICH, M. T.; TAMASSIA, R. *Estruturas de dados e algoritmos em Java*. 5. ed. Porto Alegre: Bookman, 2013.
- JOHNSONBAUGH, R. *Discrete Mathematics*. 4. ed. Upper Saddle River: Prentice Hall, 1997.
- LINTZMAYER, C. N.; MOTA, G. O. *Análise de algoritmos e estruturas de dados*. Disponível em <https://www.ime.usp.br/~mota/livros/livro_AAED.pdf>. Acesso em 19 set. 2022.
- SEDGEWICK, R.; WAYNE, K. *Algorithms*. 4. ed. Upper Saddle River: Addison-Wesley, 2011.
- SIPSER, M. *Introdução à teoria da computação*. São Paulo: Cengage Learning, 2015.
- ZIVIANI, N. *Projeto de algoritmos: com implementações em Java e C++*. São Paulo: Thomson Learning, 2007.

ÍNDICE REMISSIVO

Questão 1	Elicitação de requisitos. Engenharia de software. Personas. Cenários.
Questão 2	Rede de computadores. Switches. Roteadores. Bridges. TCP/IP
Questão 3	Compilador. Linguagem de programação. Código-fonte. Código de máquina.
Questão 4	Programação. Linguagem C. Busca linear. Busca binária. Recursividade.
Questão 5	Algoritmo de ordenação. Programação. *Quicksort*. Pivô. Funções.