Gabarito

Questão 1)

Passo Base: Primeiro, vamos verificar se a fórmula é válida para n = 1: 1 = (1 x 2) / 2 =1

portanto, a fórmula é verdadeira para n = 1.

Hipótese de indução:

Agora, vamos supor que a fórmula seja válida para um número inteiro positivo k, isto é:

$1 + 2 + ... + k = (k (k+1)) / 2$

Passo de Indução:

Precisamos provar que a fórmula é verdadeira para k + 1, ou seja:

$1 + 2 + ... + k + (k+1) = ((k+1) ((k+1) +1)) / 2$

Vamos começar somando $1 + 2 + ... + k + (k+1)$:

$1 + 2 + ... + k + (k+1) = (k (k+1)) / 2 + (k+1)$

$1 + 2 + ... + k + (k+1) = (k (k+1) + 2(k+1)) / 2$

$1 + 2 + ... + k + (k+1) = ((k+1) (k+2)) / 2$

Substituindo $1 + 2 + ... + k$ por $(k (k+1)) / 2$ temos:

$(k (k+1)) / 2 + k + 1 = ((k+1) (k+2)) / 2$

$(K^2 + K)/2 + k + 1 = ((k+1) (k+2)) / 2$

$(K^2 + K)/2 + k + 1 = ((k+1) (k+2)) / 2$

$K^2 + K + 2k + 2 = k^2 + 2k + k + 2$

$K^2 + 3k + 2 = k^2 + 3k + 2$

Portanto, a fórmula é válida para k + 1, e a prova está completa.

Assim, podemos concluir que a fórmula $1 + 2 + ... + n = (n (n+1))/2$ é verdadeira para todo número inteiro positivo n.

Questão 2

a) Para encontrar a solução da recorrência $T(n) = 2T(n-1) + 1$ com $T(0) = 1$, podemos iterar a recorrência para os primeiros valores de n e encontrar um padrão.

$T(0) = 1$

$T(1) = 2T(0) + 1 = 2*1 + 1 = 3$

$T(2) = 2T(1) + 1 = 2*2 + 1 = 7$

$T(3) = 2T(2) + 1 = 2*7 + 1 = 15$

Podemos ver que a cada iteração, o valor de T(n) é sempre uma potência de 2 menos 1.

Podemos generalizar a solução como:

$T(n) = 2^n + (2^{(n-1)} + 2^{(n-2)} + ... + 2^1 + 2^0)$

Sendo assim, chegamos à equação:

$T(n) = 2^{n+1}-1$

Esta é a solução da recorrência deduzida a partir dos 4 primeiros valores de T(n).

b) Iterando a recorrência quatro vezes, obtemos:

$T(n) = 2T(n-1) + 1$

$T(1) = 2(2T(n-2) + 1) + 1 = 4T(n-2) + 3 =$

$T(2) = 4(2T(n-3) + 1) + 3 = 8T(n-3) + 7$

$T(3) = 8(2T(n-4) + 1) + 7 = 16T(n-4) + 15$

Podemos observar um padrão emergindo, onde cada termo é uma potência de 2 multiplicada pelo termo anterior, com um número constante adicionado. Podemos escrever isso como:

$T(n) = 2^k * T(n-k) + (2^k - 1)$

Para k = n, temos:

$T(n) = 2^n * T(0) + (2^n - 1)$

$T(n) = 2^n + (2^n - 1)$

$T(n) = 2^{(n+1)} - 1$

Portanto, temos a solução da recorrência é $T(n) = 2^{(n+1)} – 1$, que é a mesma solução encontrada no item **(a)**.

Questão 3

a) Conforme Cormen, cap. 25, pág. 558, Se o grafo possuir arestas de peso negativo não se deve usar o algoritmo de Dijkstra. O algoritmo de Dijkstra não lida bem com arestas com pesos negativos em um grafo, pois tal situação pode gerar um ciclo. Em geral, é necessário utilizar outro algoritmo, como o algoritmo de Bellman-Ford, para lidar com grafos que possuem arestas com pesos negativos.

b) *Segundo Cormen, seção 22.1, página 485 "**grafos ponderados*** são grafos nos quais cada aresta tem um ***peso*** associado, normalmente dado por uma ***função peso*** *w* : *E* → ". Ou seja, um grafo ponderado é um grafo em que é atribuído um peso a cada aresta.
Além disso em um grafo dirigido, as arestas têm uma direção associada, o que significa que elas representam uma relação unidirecional entre dois vértices.

c) A fila de prioridades em um algoritmo de Dijkstra pode ser implementada de forma mais eficiente em um grafo esparso utilizando um heap binário ou um heap de Fibonacci. Ambas as estruturas de dados mantêm a propriedade de heap, permitindo que os elementos sejam acessados em tempo logarítmico. Essa mudança na estrutura de dados em relação à implementação padrão resulta em uma melhoria significativa na complexidade de tempo do algoritmo de Dijkstra. Enquanto no algoritmo padrão o tempo de execução será $O(V^2)$, onde v é o número de vértices do grafo, a implementação da fila de prioridade mínima por heap binário mínimo produz, em grafos esparsos, um tempo de execução de O (A log V), onde A é o número de arestas e V o número de vértices. A implementação da fila de prioridades utilizando o heap de Fibonacci, embora seja uma implementação mais complexa, possui um tempo de execução de O(V log V + A).

d) Em um grafo denso, onde o número de arestas é proporcional a V^2, o tempo de execução do algoritmo de Dijkstra pode se tornar impraticável, pois a complexidade do algoritmo seria, usando um heap binário para a implementação da fila de prioridade mínima, de $O(V^3 \log V)$ e caso se utilize um heap de Fibonacci, seria de $O(V^3)$.

Questão 4

a) *Para provar que $2^{(n+1)} = O(2^n)$, precisamos encontrar uma constante positiva c e um valor de n_0 tal que:*

$$2^{(n+1)} <= c * 2^n, \text{ para todo } n >= n0$$

Podemos reescrever $2^{(n+1)}$ como $2 * 2^n$, e assim:
$2 * 2^n <= c * 2^n$
Dividindo ambos os lados por 2^n, temos:
$2 <= c$
Portanto, podemos escolher $c = 2$ e $n_0 = 1$ (ou qualquer outro valor de n_0 que seja adequado) para provar que $2^{n+1} = O(2^n)$. Isso porque:
$2^{n+1} = 2 * 2^n <= 2 * 2^n$, para todo $n >= 1$
Assim, a desigualdade é satisfeita para uma constante $c = 2$ e $n_0 = 1$, e, portanto, 2^{n+1} é $O(2^n)$.

b) *A afirmativa é falsa.*

Para provar que $2^{2n} \neq O(2^n)$, é necessário assumir que existe uma constante c, $n_0 > 0$ de tal forma que:
$0 \leq 2^{2n} \leq c. 2^n$, para $n \geq n_0$

Podemos provar que $2^n * 2^n$ não é $O(2^n)$ ao mostrar que, para qualquer constante c e valor de n_0, existe um valor de n maior ou igual a n_0 para o qual a desigualdade
$2^{2n} <= c * 2^n$ não é satisfeita.

Suponha que 2^{2n} seja $O(2^n)$, o que significa que existe uma constante positiva c e um valor de n_0 tal que $2^{2n} <= c * 2^n$, para todo n maior ou igual a n_0.

Dividindo ambos os lados da desigualdade por 2^n, temos:
$2^n <= c$
Isso implica que 2^n é limitado superiormente por uma constante c.

Portanto,a afirmação é claramente falsa, já que 2^n cresce exponencialmente com n e não é limitado superiormente por qualquer constante, pois não existe uma constante c que é maior do que 2^n para todo n. Portanto, 2^{2n} não é $O(2^n)$.

Questão 5

a) *Em linguagens de programação, l-values e r-values são usados para distinguir entre diferentes tipos de expressões e valores. Um l-value é um valor que se refere a uma localização de memória, enquanto um r-value é um valor que é computado e não se refere a uma localização de memória. A distinção entre l-values e r-values é um conceito importante em linguagens de programação, e é usado em diversas tarefas de programação, como a operações de atribuições, avaliação de expressões e gerenciamento de memória.*

b) *Em operações de atribuição, o lado esquerdo de uma atribuição é sempre um l-value, pois deve representar a localização de memória onde o valor será armazenado; enquanto o lado direito é um r-value, pois deve representar o valor que será armazenado na localização de memória. Por*

exemplo, na instrução de atribuição X = Y, X é um l-value e Y é um r-value. O valor de Y é computado e armazenado na localização de memória representada por X.

Questão 6

(a) *Uma avaliação de curto-circuito é uma técnica utilizada para a otimização da avaliação de expressões, na qual o resultado de uma expressão é determinado sem que todos os seus operandos e/ou operadores necessitem ser avaliados. Por exemplo, dada a expressão lógica "X && (Y < 10)", em que && representa o operador lógico AND, caso X seja FALSO, a expressão é falsa para qualquer valor de Y, não sendo necessário avaliar a expressão "(Y < 10)". Expressões aritméticas também podem fazer uso desse tipo de técnica. Por exemplo, a expressão (42 * X) * (Y / 42 - 1) independe do valor de (Y / 42 - 1) se X for igual a 0 (zero).*

(b) *A expressão lógica é composta por duas condições: a primeira condição verifica se a posição atual na lista (dada pela variável "posicao") é menor que o tamanho da lista, e a segunda condição verifica se o elemento na posição atual da lista é igual ou não ao valor procurado. Quando o valor procurado não existir na lista, então a variável "posicao" vai, em algum momento, assumir o valor igual ao valor de "tamanho_lista", que é uma posição inexistente na lista. Nesse caso, a primeira condição irá falhar (posicao < tamanho_lista é avaliado como FALSO) e, portanto, a segunda condição não será avaliada devido à avaliação em curto-circuito, e o laço será interrompido. Isso evita que ocorra um erro de acesso a um índice inexistente da lista.*

Questão 7

a) As linguagens regulares são aquelas que podem ser especificadas por expressões regulares ou gramáticas regulares e cujas cadeias são reconhecidas por autômatos finitos.

b) As linguagens livres de contexto são aquelas que podem ser especificadas por gramáticas livres de contexto e cujas cadeias podem ser reconhecidas pelos autômatos de pilha.

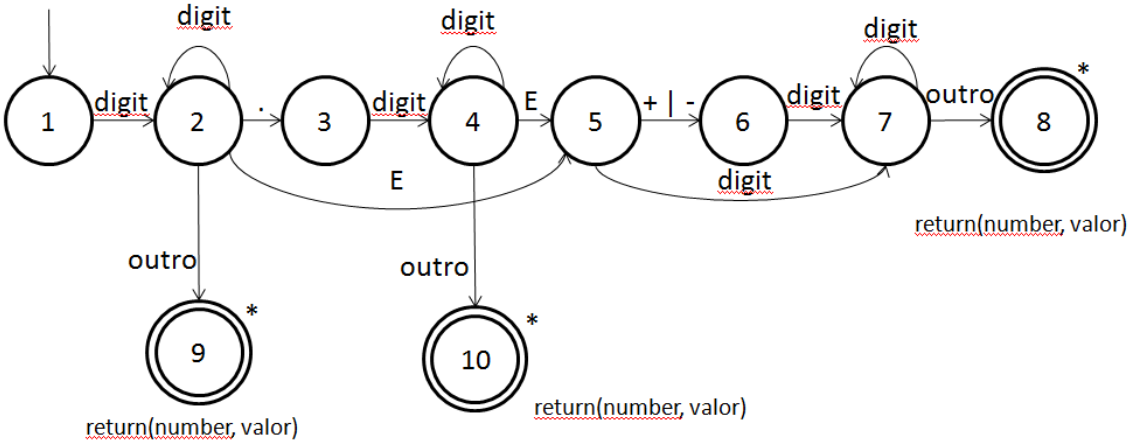
c) As linguagens recursivamente enumeráveis são aquelas que podem ser especificadas por gramáticas irrestritas e cujas cadeias podem ser reconhecidas pelas máquinas de Turing.

Questão 8

a) $P \rightarrow 0P1P \mid 1P0P \mid \lambda$

b) $P \rightarrow 0P0 \mid 1P1 \mid 0 \mid 1 \mid \lambda$

Questão 9



Questão 10

A primeira parte da compilação é denominada *front-end* ou análise. Ela é constituída pelas seguintes fases:

- Análise léxica ou *scanning*: lê o arquivo do programa fonte, caractere a caractere, e os agrupa em sequências significativas denominadas lexemas. Para cada lexema obtido do código fonte, o analisado léxico gera um *token* a ser repassada ao analisador sintático. Basicamente, o analisador léxico verificar se as sequências de caracteres presentes no código fonte casam com o padrão de formação dos *tokens* da linguagem de programação.

- Análise sintática ou *parser*: recebe como entrada os *tokens* identificados e verifica se essa cadeia de *tokens* é derivável a partir da gramática da linguagem. Dessa forma, o analisador sintático gera a árvore sintática correspondente ao programa fonte.

- Análise semântica: utiliza a tabela de símbolos e a árvore sintática gerada pelo analisador sintático para verificar se o programa está semanticamente de acordo com as especificações da linguagem fonte. As verificações realizadas pelo analisador semântico são as seguintes:

  - verificação de unicidade: verifica se os identificadores foram definidos uma única vez;

  - verificação de tipos: verifica se os tipos dos operandos nas operações são compatíveis entre si e determina os tipos resultantes nas operações;

  - verificação de classe: verifica se a categoria dos identificadores é compatível com a operação.

A segunda parte da compilação é o *back-end* ou síntese. Nesta parte, ocorre a fase de geração de código, que tem por objetivo gerar o código objeto correspondente ao código fonte analisado.