**1ª Avaliação de Introdução à Análise de Algoritmos**

**Prof. Glauber Cintra**

Envie essa lista pelo Classroom até 07/08/2020 às 15:30h.

1. **(0,5 pontos)** Numere as funções abaixo, a partir de 1, em ordem estritamente crescente de dominação assintótica. Se duas funções f e g são tais que f ∈ o(g) então f deve ter um número menor do que g. Se f ∈ Θ(g) então f e g devem ter o mesmo número.

( 6 ) 22n ( 2 ) log10 (n) ( 5 ) n3 ( 4 ) 3n2 ( 2 ) 5log2 (n4)

( 7 ) n! ( 5 ) 5n + n3 ( 3 ) nlog2 (n2) ( 6 ) 2n+1 ( 1 ) 8

1. **(1 ponto)** Explique o significado dos termos *algoritmo*, *algoritmo computacional,* *algoritmo correto,* *algoritmo eficiente* e *tamanho da entrada de um algoritmo.*

**Algoritmo**: Sequência de instruções desenvolvidas para resolver instâncias de um problema.

**Algoritmo Computacional**: Algoritmo criado apenas com instruções bem definidas, não ambíguas.

**Algoritmo Correto**: Algoritmo que resolve corretamente todas as instâncias do problema para qual ele foi desenvolvido.

**Algoritmo Eficiente**: Algoritmo que requer a execução de uma quantidade de instruções elementares limitada por um polinômio no tamanho da entrada.

**Tamanho da Entrada**: Quantidade de bits para representar os dados de uma entrada.

1. **(1 ponto)** Indique quais são o melhor caso e o pior caso do algoritmo abaixo e sua região crítica. Qual a complexidade temporal desse algoritmo no pior e no melhor caso? Qual a complexidade espacial desse algoritmo? O algoritmo é eficiente? É de cota inferior? Justifique suas respostas.

Algoritmo Inserção

Entrada: um vetor v com n posições (indexado a partir do zero)

Saída: o vetor v em ordem crescente

para i = 1 até n - 1

pivo = v[ i ]

j = i - 1

enquanto j >= 0 e v[ j ] > pivo

v[ j + 1 ] = v[ j ]

j = j – 1

v[ j + 1 ] = pivo

devolva v

**Melhor caso**: Se o vetor já estiver todo ordenado em ordem crescente.

**Pior caso**: Se o vetor estiver ordenado em ordem decrescente.

**Região crítica**: “enquanto j >= 0 e v[ j ] > pivo” pois essa instrução irá comparar se o elemento da posição v[ i ], que foi passado para a variável “pivo”, é menor que a posição anterior, que no caso é v[ j ] e j = i – 1, e com isso fazer a alteração de posição, e sairá comparando “pivo” as posições anteriores até que ache uma maior ou chegue na primeira posição do vetor, e com isso sair do laço.

**Complexidade temporal / melhor caso**: Já que o vetor já está todo ordenado o algoritmo só irá verificar cada posição e devolver o próprio vetor, ou seja levará tempo Θ(n).

**Complexidade temporal / pior caso**: No pior caso o algoritmo irá comparar n posições e o laço condicional pode chegar a repetir também n vezes, com isso temos que que a complexidade temporal do pior caso é Θ(n2).

*“Quantas vezes a função insercao compara x com um elemento do vetor? Mais precisamente, quantas vezes, no máximo, a função insercao executa a comparação v[i] > x? Esse número está diretamente relacionado com a sequência de valores de i ao longo da execução da função. Para cada valor de j, a variável i assume, no pior caso, os valores j-1,  . . . ,  0 . A seguinte tabela mostra esses valores explicitamente:*

*j i*

*1 0 1*

*2 1 0 2*

*3 2 1 0 3*

*. . . . .*

*n-1 n-2 n-3 .. 1 0 n-1*

*A terceira coluna da tabela dá o número de diferentes valores de i na linha. Portanto, o número de execuções da comparação v[i] > x é, no pior caso, igual à soma da terceira coluna. Essa soma é n(n-1)/2, ou seja, um pouco menos que a metade de n2 .”*

*Referência:* [*https://www.ime.usp.br/~pf/algoritmos/aulas/ordena.html*](https://www.ime.usp.br/~pf/algoritmos/aulas/ordena.html)

**Complexidade espacial**: O espaço requerido é O(1), pois há somente declarações de variáveis escalares.

**Cota inferior**: Não, pois há algoritmos que requerem tempo menor que O(n2), como é o caso do algoritmo mergesort que tem como complexidade temporal O(.

**Algoritmo eficiente**: O algoritmo inserção é eficiente, pois o a função que limita o tamanho da entrada é n que seria o tamanho do vetor, ou seja Θ(n).

1. **(1 ponto)** Escreva um algoritmo que receba duas matrizes de ordem *n*x*m* e devolva a soma dessas matrizes. Prove que seu algoritmo é correto e determine sua complexidade temporal e espacial. Seu algoritmo é de cota inferior? É de cota superior?

Algoritmo: somaMatriz

Entrada: matriz M1 e M2 com n linhas e m colunas

Saída: a soma das matrizes

soma [n,m]

Para i = 0 até n – 1

Para j = 0 até m – 1

Soma[i,j] = M1[i,j] + M2[i, j]

Devolva soma

**Corretude**: Note que o laço mais externo faz o controle das linhas de 0 até *n-1* e o laço mais interno controla as colunas de 0 até *m-1*, também se pode notar que o algoritmo só irá mudar de linha após o laço interno percorrer toda a coluna da linha *i*, com isso irá para próxima linha, *i+1*, e fazer o mesmo processo de percorrer toda a coluna. E dentro do laço mais interno há a instrução de somar, ou seja começando da posição [i=0**,**j=0] das duas matrizes ele irá somar os elementos presentes nessas posições e colocar na matriz *soma* que também está na posição [i=0**,**j=0], em seguida pular para coluna seguinte, [i=0**,**j=1], e fará esse processo até que chegue na última coluna onde ele sairá do laço interno e pulará para linha seguinte e irá repetir o mesmo processo de somar só que agora a partir da posição [i=1**,**j=0] e assim sucessivamente até que some todos os elementos das matrizes, ou seja até que os contadores do laço cheguem na posição n-1 e m-1, e as somas sejam colocadas na matriz soma o que é correto.

**Complexidade Temporal**: O algoritmo percorre as n linhas e m colunas da matriz, logo a complexidade temporal é O(nm).

**Complexidade Espacial**: O espaço requerido é O(1), pois há somente declarações de variáveis escalares.

**Cota inferior**: O algoritmo é de cota inferior, pois é fácil perceber que todo algoritmo correto para esse problema requer tempo Ω(nm) e o algoritmo em questão requer tempo O(nm).

**Cota superior**: O algoritmo é de cota superior, pois a melhor maneira conhecida e também correta de somar matrizes é com esse algoritmo que gasta tempo O(nm). *E vale ressaltar que todo algoritmo, correto e conhecido, de cota inferior também é um algoritmo de cota superior*.

1. **(1 ponto)** Resolva as seguintes fórmulas de recorrência:
   1. T(n) = T(n+ n, T(1) = 1

Supondo n = 2^k, ∀ k ≠ 0 e ∈ N ⇒ **⎣**n/2**⎦** = **⎣**2(k-1)**⎦** = 2(k-1). Com isso podemos eliminar a função piso pois ela não será utilizada de a cordo com a nossa suposição já que sempre resultará em um número natural.

Utilizando o método da soma

*Equação 0*: 1\*T(n)= T(n/2) + n

*Equação 1*: 2\*T(n/2) = 2T(n/4) + 2()

*Equação 2*: 4\*T(n/4) = 4T(n/8) + 4()

*Equação 3*: 8\*T(n/8) = 8T(n/16) + 8()

....

*Equação k – 1* : 2(k-1) \* T() = T() + \*()

*Equação k*: 2k \* T() e como pressupus que n é igual 2k temos que T()= T(1) = 1.

Ao somar todas as equações ficamos com:

T(n) = n + n + n + n +... +2k + 1 ⇒ T(n) = k\*n + 2k + 1

Ainda baseado na suposição de n = 2k ⇒ k = ⇒ T(n) = n\* + n + 1. No final temos que T(n) ∈ O(n)

* 1. T(n) = 2T(n – 1) + n, T(1) = 1

Utilizando o método da soma

*Equação 0*: T(n)= T(n - 1) + n

*Equação 1*: T(n - 1) = T(n - 2) + (n – 1)

*Equação 2*: T(n - 2) = T(n - 3) + (n – 2)

*Equação 3*: T(n - 3) = T(n - 4) + (n – 3)

....

*Equação n – 1*: T(2) = T(1) + 2

*Equação n*: T(1) = 1

Ao somar todas as equações ficamos com:

T(n) = n + (n – 1) + (n – 2) + (n – 3) + ...+ 2 + 1

Aplicando a fórmula da soma de uma P.A obtêm-se: ()\*n = () ⇒ T(n) ∈ O(n²)

1. Considere o seguinte algoritmo:

Algoritmo enigma

Entrada: n (natural)

se n = 0

devolva 0

se não

devolva enigma(n – 1) + n + n – 1

1. **(1 ponto)** Determine a complexidade temporal e espacial do algoritmo *enigma* (mostre os cálculos realizados para determinar tais complexidades). Esse algoritmo é eficiente? Para que serve esse algoritmo? É de cota inferior? Justifique suas respostas.

**Para que server**? É um somatório de números impares (2n – 1) + (2(n-1) -1) +...+ 5 + 3 + 1 ⇒ . O algoritmo pega um número n, multiplica por 2 e soma todos os números ímpares menores que 2n até 1.

**Complexidade temporal**: T(n) ∈ O(n²)

Transformando em numa formula de recorrência: T(n)= T(n - 1) + 2n ; T(0) = 0

Utilizando o método da soma

Equação 0: T(n)= T(n - 1) + 2n

Equação 1: T(n - 1) = T(n - 2) + (2n – 1)

Equação 2: T(n - 2) = T(n - 3) + (2n – 2)

Equação 3: T(n - 3) = T(n - 4) + (2n – 3)

....

Equação n – 1: T(1) = T(0) + c

Equação n: T(0) = c

Ao somar todas as equações ficamos com:

T(n) = 2n + (2n – 1) + (2n – 2) + (2n – 3) + ...+ 1 + 0

Aplicando a fórmula da soma de uma P.A obtêm-se: ()\*n = () ⇒ T(n) ∈ O(n²)

**Complexidade espacial**: Já que nesse algoritmo recursivo é usado a pilha de recursão o cálculo da complexidade espacial é igual ao do temporal E(n) ∈ O(n²).

**Algoritmo eficiente**: O algoritmo não é eficiente, por conta que o tamanho da entrada para representar um número natural é t = + 1 bits ⇒ n = 2t ⇒ ∈ Θ (2t).

**Cota inferior**: Não, pois há maneiras que gastam menos tempo de saber a soma dos números ímpares, que como foi mostrado no somatório que seria se eu recebesse um número n e fizesse “*devolva n\*n*” eu teria no fim a soma de todos os números ímpares de 1 até 2n, ou seja levaria tempo constante, quer dizer que a cota inferior é de Ω(1).

1. **(1 ponto)** Prove que o algoritmo é correto.

Teorema: enigma é correto.

Caso base n = 1, é um caso trivial já que ele devolverá 1, o que é correto.

Suponha agora que enigma (n) = n2(H.I).

Temos que enigma(n) = (2n – 1) + (2(n-1) -1) +...+ 5 + 3 + 1 = n2, aplicando indução em n,

(2(n+1) – 1) + (2n – 1) + (2(n-1) -1) +...+ 5 + 3 + 1 = (n+1)2. Note que + (2n – 1) + (2(n-1) -1) +...+ 5 + 3 + 1 é a nossa H.I ou seja podemos substituir por n2, fazendo 2(n+1) - 1 + n2 = n2 + 2n + 1 ⇒ n2 + 2n + 1 = n2 + 2n + 1.

Provando assim que o algoritmo é correto pois as somas dos números impares de 1 até 2n-1 é igual a n², que foi o que usamos na nossa prova por indução.

1. **(1 ponto)** Escreva um *algoritmo de cota inferior* ***recursivo*** para multiplicar os valores contidos num vetor de números. Justifique porque ele é de cota inferior.

Algoritmo multiplicaVetor

Entrada: um vetor V e a posição n

se (n = 0)

devolva V[n]

se não

devolva V[n] \* multiplicaVetor(V, n - 1)

Sabemos que a melhor maneira para resolver esse problema corretamente temos que escolher um ponto de origem, que no caso desse algoritmo seria a última posição, e multiplicar pela posição anterior e assim sucessivamente até que chegue na posição 0 do vetor. Já que todos os algoritmos que resolvem corretamente esse problema gastam tempo Ω(n), ou seja gasta no mínimo tempo linear, e o algoritmo multiplicaVetor resolve em tempo O(n), conclui-se que o algoritmo multiplicaVetor é de cota inferior.

Utilizando o método da soma

*Equação 0*: T(n)= T(n – 1) + c

*Equação 1*: T(n - 1) = T(n - 2) + c

*Equação 2*: T(n - 2) = T(n - 3) + c

*Equação 3*: T(n - 3) = T(n - 4) + c

....

*Equação n – 1*: T(1) = T(0) + c

*Equação n*: T(0) = c

Ao somar todas as equações ficamos com: T(n) = c + c + c +... + c (somar as constantes n vezes) = cn ⇒ T(n) ∈ O(n)

1. **(0,5 pontos)** Pesquise e cite o nome de um algoritmo de *cota superior* para o *problema da primalidade* (testar se um número é primo).

Existem algoritmos probabilísticos simples e eficientes que determinam se um dado número é primo e apresentam uma pequena margem de erro (por exemplo, o teste de primalidade de Fermat). Os algoritmos determinísticos conhecidos para resolver o mesmo problema ainda são consideravelmente mais lentos na prática.

Mas o teste da primalidade AKS (também conhecido como teste da primalidade Agrawal-Kayal-Saxena) é até hoje considerado por muitos matemáticos como o melhor algoritmo para resolver o problema da primalidade já que ele é o primeiro algoritmo publicado para ser simultaneamente polinomial, determinístico e incondicional. O que isto significa, o tempo máximo de processamento do algoritmo pode ser expresso como um polinômio em relação ao número de dígitos no número objetivo, isto nos permite classificar o número informado como primo ou composto (ao invés de retornar um resultado probabilístico); e a sua correção não está subordinada a exatidão de uma hipótese subsidiária não provada (tal como a hipótese de Riemann).

Algoritmo AKS:

Introduzir: Inteiro n > 1

SE (n ESTÁ NA FORMA a^b COM b > 1) ENTÃO mostrar "COMPOSTO"

r := 2

while (r < n) {

SE (Mdc(n,r) NÃO É 1) ENTÃO mostrar "COMPOSTO"

SE (r É UM Nº PRIMO MAIOR QUE 2) ENTÃO {

q = MAIOR FACTOR DE r-1

SE (q > 4sqrt(r)log n) e (n(r-1)/q NÃO É IGUAL A 1(mod r)) ENTÃO SAIR DESTE CICLO(BREAK)

}

r := r+1

}

PARA a = 1 TO 2sqrt(r)log n {

SE ( (x-a)^n NÃO É (x^n-a) (mod x^r-1,n) ) ENTÃO mostrar "COMPOSTO"

}

mostrar ¨PRIMO¨

Referências: [https://pt.wikipedia.org/wiki/Teste\_de\_primalidade\_AKS#:~:text=O%20teste%20da%20primalidade%20AKS,"PRIMES%20is%20in%20P".](https://pt.wikipedia.org/wiki/Teste_de_primalidade_AKS%23:~:text=O%20teste%20da%20primalidade%20AKS,%22PRIMES%20is%20in%20P%22.)

<https://www.linux.ime.usp.br/~gervasio/ic/relatorio-ic.pdf>

<https://www.inf.ufrgs.br/~mrpritt/lib/exe/fetch.php?media=cmp155:notas-10619.pdf>

<https://cic.unb.br/~rezende/trabs/primal.htm>

<https://files.cercomp.ufg.br/weby/up/615/o/Minicurso_AKS_Manuela_Tamires.pdf?1478948693>

1. **(2 pontos)** Faça a análise amortizada do algoritmo abaixo, utilizando dois dos três métodos que estudamos, e determine o *custo amortizado* de cada chamada ao algoritmo. Informe também o custo total para realizar *n* chamadas consecutivas ao algoritmo, supondo que inicialmente a pilha está vazia.

Algoritmo empilhaSemOverflow

*Entrada: uma pilha p (implementada num vetor) e um valor x.*

*Saída: insere x em p.*

Se p.topo = p.tamanho - 1

duplique p // requer tempo linear no tamanho da pilha

p.topo = p.topo + 1

p.vetor[p.topo] = x

Se p.topo for menor que p.tamanho – 1 quer dizer que há espaço livre no vetor, isso quer dizer que levará tempo O(1) para inserir x em p. E se p.topo for igual a p.tamanho – 1, ou seja não há espaço no vetor, a operação levará tempo O(n), pois ela duplica o vetor e isso custará n\*(p.tamanho -1) com (p.tamanho -1) sendo a constante que representa o tamanho do vetor. E a cada vez que eu gasto O(n) em uma operação, eu poso fazer em seguida p.tamanho – 1 operações com custo O(1)

**Método da agregação**: Em geral se considerássemos um número arbitrário de inserções n na pilha de tamanho n, nós notaríamos que todas as operações de inserção tomariam tempo constante menos a última, que tomaria tempo O(n) para realizar a operação de dobrar o tamanho. Como existiriam n operações nós podemos pelo método da agregação concluir que a inserção de elementos em um pilha dinâmica leva : O() = O(1), tempo constante.

**Método contábil**: Vou atribuir a função “duplique p” o custo amortizado de 2\*p.tamanho, ou seja vai gerar credito de p.tamanho, e atribuir a função “p.vetor[p.topo] = x” o custo amortizado de 1, e atribuir a função “p.vetor[p.topo] = x” o custo amortizado 0. Assim, o saldo será sempre maior ou igual a zero, logo a soma dos custos amortizados é maior ou igual à soma dos custos reais. Como a soma dos custos amortizados é O(n), concluímos que o custo total das n chamadas é O(n) e o custo amortizado de cada chamada é O(1)