	UNIVERSIDADE ESTADUAL DE SANTA CRUZ **DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA** **Prof. Hélder Almeida – E-mail: hcalmeida@uesc.br**	
Curso: Ciência da Computação		**Período:** 2021.1
Disciplina: Estrutura de Dados		**Código:** CET0077
Alunos: Sala toda		**Matrículas:** várias
Nota:	**Atividade Ordenação**	**Data:**

Questão 01.

1. (0,5) Uma das possíveis formas de se descrever a complexidade de um algoritmo é a chamada **Notação-Big-O**, que é definida da seguinte forma: **T(n) = O(f(n))** se existem constantes **c** e **n0** tais que **T(n) <= c.f(n)** quando **n > n0**. Explique o que você entendeu por esta definição.*(máximo de 500 caracteres)*

   A definição mostra que existe uma constante c que faz com que f(n) seja no máximo igual a T(n), sempre que existir um n maior que n0, ou seja o custo de f(n) nunca será pior do que o seu pior caso T(n).

Questão 02.

2. (1,0) Vimos que existem **duas formas de se implementar listas**: como conjuntos de dados fisicamente adjacentes, através da utilização de **vetores** e como conjuntos de dados apenas logicamente interligados, mas sem adjacência física, através da utilização de **listas encadeadas**. As listas encadeadas possuem vantagens de economia de memória bastante óbvias, uma vez que é somente utilizada a memória realmente necessária para armazenar um determinado conjunto de dados. Do ponto de vista de complexidade, discutimos em sala de aula também vantagens e desvantagens dos dois modelos. Explique **qual dos dois modelos é melhor em termos de complexidade de tempo** de acesso e explique por que isto ocorre, exemplificando. Diga qual é a complexidade média de tempo de acesso a um dado em uma lista com vetor de n dados e uma lista simplesmente encadeada com n dados. Justifique.*(máximo de 1500 caracteres)*

   Para o tempo de acesso, um vetor, quando é alocado estaticamente num bloco sequencial de memória, ao definir um índice de acesso a um dos elementos do bloco, o acesso é obtido através de um cálculo simples obtido da soma unitária do endereço do primeiro elemento mais o produto do índice pelo tamanho do tipo de dado primitivo ou abstrato. Essa é uma complexidade O(1).

   No caso da lista encadeada. Os índices não são colocados dessa forma, sendo necessário varrer o vetor linearmente para obter acesso ao dado numa certa posição, ou seja, O(n).

Para a pesquisa linear de dados, ambos têm complexidade O(n) já que os algoritmos lineares precisarão varrer o vetor n vezes.

*Como exemplo, para acessar o item de índice 5 no vetor estático de 10 posições Ve, basta fazer Ve[5] e a aplicação retornará em O(1) a soma do endereço de hex(Ve[0]) + hex(5*sizeof(Ve[0])).*

Para acessar o item de índice 5 no vetor dinâmico em modo de lista encadeada de 10 posições alocadas, o algoritmo acessará sequencialmente de forma iterativa ou recursiva os ponteiros para o próximo elemento até que chegue no elemento 5.

Dentre as vantagens e desvantagens de uso, listas dinâmicas são flexíveis e entregam liberdade para o operador do código em situações que podem variar em quantidade. Vetores têm implementação mais simples e servem para situações que estão sob o controle do operador de maneira finita, são problemas que estão normalmente ligados a ações definitivas dentro da própria aplicação.

Questão 03.

3. (0,5) Para o cálculo da complexidade de algoritmos **não recursivos**, existe um conjunto de instruções bastante simples de serem seguidas. Cite e descreva-as. *(máximo de 500 caracteres)*

Determina-se a operação básica de maior frequência na execução do algoritmo (operação dominante), desconsiderando as constantes e elementos menores dos cálculos. Para estruturas iterativas considerar:

Laços do-while/for e outros: soma da complexidade de todas as instruções dentro do laço multiplicado pelo número de iterações. Quando aninhados multiplica-se pelo produto dos tamanhos dos laços.

Funções: complexidade da função x o número de vezes que ela é chamada.

If/Then/Else: Seu tempo nunca é maior que o tempo de execução da condição + o tempo de execução do termo dominante.

Questão 04.

4. (1,5) Algoritmos recursivos são bem mais difíceis de se analisar com respeito ao seu comportamento assintótico (complexidade de tempo). Quando nós tentamos descrever a complexidade de tempo de um algoritmo recursivo utilizando as regras acima, acabamos obtendo uma fórmula também recursiva, que nós chamamos de relação de recorrência. Para resolver essa fórmula existe um conjunto de regras matemáticas que, provavelmente, você ainda não viu. Mas você pode, para alguns problemas, “estimar” a complexidade de execução **Big-O** de um algoritmo recursivo com base em algumas de suas características sem a necessidade de resolver um problema matemático mais complexo.

a. Explique que tipos de problemas ou algoritmos recursivos costumam ter complexidade da ordem de **O(nlog(n))**. *(máximo de 500 caracteres)*

Algoritmos de complexidade logarítmica O(nlog(n)) são aqueles que dividem um problema em subproblemas, resolvendo-os de forma independente para depois combinar os resultados. Um bom exemplo, são os algoritmos que tratam da ordenação de um conjunto de números, como os do Heap Sort e Merge Sort.

b. Quais problemas com soluções recursivas possuem, geralmente, complexidade da ordem de **O(log(n))?** Cite dois exemplos. *(máximo de 500 caracteres).*

Algoritmos com complexidade logarítmica O(log(n)) são aqueles que costumam dividir um problema em problemas menores (dividir para conquistar). Como os algoritmos de 'Busca binária' e 'Encontrando o maior / menor número em uma árvore de pesquisa binária'.

c. Quais os problemas recursivos costumam ter complexidade exponencial **O(d^n)**? Cite um exemplo. *(máximo de 500 caracteres)*

Algoritmos que não se sabe exatamente uma estrutura para se encontrar a resposta , e é feita a busca por força bruta até encontrar um possível resultado , Um exemplo é a torre de Hanoi.

Questão 05.

5. (1,0) Entre os algoritmos de ordenação mais utilizados, um dos que possui menor complexidade é o **Insertion Sort**, considerando o seu melhor caso, **O(n)**. Podemos dizer que nenhum outro algoritmo poderá atingir uma complexidade melhor? Justifique. *(máximo de 500 caracteres)*

Resposta -> Dentro dos algoritmos de ordenação mais utilizados e vistos em sala, nenhum algoritmo consegue ter um "melhor caso" do que o Insertion Sort (O(n)) que é uma complexidade linear, que por definição do Big O não considera constantes multiplicativas. Existe um caso específico do BubbleSort que no melhor caso (vetor classificado) a complexidade é também O(n). Esse caso leva em consideração uma flag switched (boolean) e no melhor caso é necessário apenas n-1 comparações (nenhuma troca).

Questão 06.

6. (1,0) Dois algoritmos A e B possuem complexidade **O(n^5)** e **O(2^n)**, respectivamente. Você utilizaria o algoritmo B ao invés do A. Se sim, em qual caso? Exemplifique. *(máximo de 500 caracteres)*

Na maioria dos casos, os algoritmos com complexidade O(2^n) são algoritmos que utilizam força bruta, que são normalmente os de menos eficiência. No entanto, nós poderíamos utilizá-los em algoritmos muito complexos nos quais ainda não se encontrou alguma outra

forma mais eficiente de serem programados, como no caso do algoritmo recursivo das Torres de Hanoi. Também podemos utilizar algoritmos com tal complexidade na área de Cyber Segurança (ex. Força Bruta) para testar sistemas.

Questão 07.

7. (5,0) Analise o pior caso dos algoritmos abaixo:

a.	`int func(int n) {`
	`  int i, r;`
	`  r = 1;`
	`  for (i = 1; i <= n; i++)`
	`  r = r * n;`
	`  return r;`
	`}`
		PIOR CASO:
		O(n)

b.	`int func(int n) {`
	`  int a, b, c, r;`
	`  b = n;`
	`  c = n;`
	`  r = 1;`
	`  while (b >= 1) {`
	`  a = b % 2;`
	`  if (a == 1)`
	`  r = r * c;`
	`  c = c * c * c;`
	`  b = b / 2;`
	`  }`
	`  return r;`
	`}`
		PIOR CASO:
		$O(log(n))$

c.	`int thisOneIsTricky(int * A, int n) {`
	`  if (n < 12) {`
	`  return (A[0]);`
	`  }`
	`  int y, i, j, k;`
	`  for (i = 0; i < n / 2; i++) {`
	`  for (j = 0; j < n / 3; j++) {`
	`  for (k = 0; k < n; k++) {`
	`  A[k] = A[k] - A[j] + A[i];`
	`  }`
	`  }`
	`  }`
	`  y = thisOneIsTricky(A, n - 5);`
	`  return y;`
	`}`
		PIOR CASO:
		$O(n^4)$

d.	`void transpor(int ** M, int n) {`
	`  int i, j, temp;`
	`  for (i = 1; i < n; i++) {`
	`  for (j = 0; j < i; j++) {`
	`  if (M[i][j] != M[j][i]) {`
	`  temp = M[i][j];`
	`  M[i][j] = M[j][i];`
	`  M[j][i] = temp;`
	`  }`
	`  }`
	`  }`
	`}`
		PIOR CASO:
		$O(n^2)$

e.	`void transpor(int ** M, int n) {`
	`  int i, j, temp;`
	`  for (i = 0; i <= (n - 1) / 2; i++) {`
	`  for (j = i; j <= n - 2 - i; j++) {`
	`  temp = M[n - 1 - j][i];`
	`  M[n-1-j][i] = M[n-1-i][n-1-j];`
	`  M[n-1-i][n-1-j] = M[j][n-1-i];`
	`  M[j][n - 1 - i] = M[i][j];`
	`  M[i][j] = temp;`
	`  }`
	`  }`
	
	`}`
		PIOR CASO:
		$O(n * log(n))$