ACH2002 – Introdução à Ciência da Computação 2 – Prof. Fábio Nakano.
Sugestão de solução para a prova 1 aplicada à turma 02.

1-) **(1,5pt)** Escreva o código recursivo que calcula a raiz quadrada de n. Ele deve implementar a sequência

$x_{i+1}=\frac{1}{2}(x_i+\frac{n}{x_i})$ e calcular novos termos enquanto $\varepsilon=|x_{i+1}-x_i|>EPS$.

Sugestão: o método recebe x_i como parâmetro, calcula x_{i+1} e ε e age de acordo com o necessário. (Apenas números positivos serão passados ao método).

```
import java.lang.Math.*;

class Raiz {
	public static final double EPS=1e-10;
	public static double raizrec (double x0, double n) {
		// este é o método:
		double proxx=0.5*(x+(n/x));
		if (Math.abs(proxx-x)<EPS) return proxx;
		return raizrec (proxx, n);
	}
	public static double raiz (double n) {
		return raizrec (1, n);
	}
	public static void main (String[] args) {
		System.out.println (raiz (1.44));
	}
}
```

2-) Dado o código abaixo, obtenha a recorrência que descreve sua complexidade de tempo **(1pt)** e explique a que corresponde cada um de seus termos **(1pt)**. Note que o algoritmo é de ordenação por inserção!!

```
class InsertionSort {
	int[] Array;
	void ordena (int IElemento) {  // o fim é o final do array mm.
		System.out.println (IElemento);
		if (IElemento<(Array.length-1)) {
			ordena (IElemento+1);
			insere (IElemento);
		}
	}

	/** Insere o elemento de índice IElemento na posicao "certa"
	  * do array ordenado. */
	void insere (int IElemento) {
		int Elemento=Array[IElemento];
		int i=IElemento;
		while ((i<(Array.length-1))&&(Elemento>Array[i+1])) {
			i++;
			Array[i-1]=Array[i];
		}
		Array[i]=Elemento;
	}
	/** Métodos auxiliares fornecidos por completude. */
	InsertionSort (int[] Arr) {
		Array=Arr;
	}

	void imprime () {
		for (int i=0;i<Array.length;i++)
			System.out.print (Array[i] + " ");
		System.out.println ();
	}
	public static void main (String[] args) {
		int[] Arr={98, 56, 33, -4, -98, 100, 7};
		InsertionSort IS=new InsertionSort (Arr);
		IS.ordena(0);
		IS.imprime();
	}
}
```

O método ordena não tem nenhum loop e tem chamadas aos métodos ordena e insere. A chamada (recursiva) a ordena é feita com o mesmo parâmetro incrementado de 1 e não é mais feita quando Ielemento>=(Array.length-1). Como a primeira chamada é feita com parametro=0 e prossegue até Array.length-1 então ocorrem n chamadas (recursivas). O método insere varre o array comparando o Elemento (a inserir) com os outros, inserindo o elemento na posição certa. Este tem um loop que se repete n-1 vezes no pior caso. Portanto a recorrência é $T(n)=T(n-1)+K_{insere}*(n-1)+K_{compara}$ onde T(n) é a complexidade de tempo de ordena com problema de tamanho n, que é igual a complexidade de ordena com problema de tamanho n-1 somado a complexidade de insere, que é proporcional a n-1, somado a uma constante que corresponde ao tempo de se executar uma comparação. Variações como $T(n)=T(n-1)+n-1+1$, que contam apenas número de comparações ou supõe que as constantes valem 1 são válidas, desde que devidamente justificadas, pois resultam na mesma complexidade de tempo.

3-) Dada a recorrência $T(n)=\begin{cases} k\ para\ n=1 \\ 2*T(\frac{n}{3})+k\ para\ n>1 \end{cases}$ **, prove por indução** que a fórmula fechada que a descreve é $T(n)=(2^{\log_3(n)+1}-1)k$. **(base 0,3pt, passo 1,2pt)**

base – para n=1, pela definição T(1)=k e pela fórmula $T(1)=(2^1-1)*k=k$
passo: para o próximo valor de n, que é o triplo do valor corrente:

Pela definição: $T(3*n)=2*T(\frac{3*n}{3})+k=2*T(n)+k$

Aplicando a hipótese de indução: $2*T(n)+k=2*((2^{\log_3(n)+1}-1)k)+k$

Distribuindo 2 e simplificando: $2*((2^{\log_3(n)+1}-1)k)+k=2*2^{\log_3(n)+1}*k-k$

notando que $2=2^1$ e agrupando os expoentes: $2^{\log_3(n)+1+1}*k-k$

notando que $\log_b(b)=1$: $2^{\log_3(n)+1+\log_3(1)}*k-k$

agrupando os logb e pondo k em evidência:

$2^{\log_3(n)+1+\log_3(1)}*k-k=(2^{\log_3(3*n)+1}-1)*k$

esta é a expressão da fórmula para o próximo n, portanto a fórmula está correta.

4-) Dadas f(n) e g(n) abaixo, demonstre que $f(n)\in O(g(n))$. No ítem b, use o conceito de dominância – não use a definição usando limites.

a-) **(1,0pt)** $f(n)=n*(\log(n)+sen(n));g(n)=n^2$

$\lim_{x\to\infty}(\frac{n*(\log(n)+sen(n))}{n^2})=\lim_{x\to\infty}(\frac{k*1/n+\cos(n)}{1})=\lim_{x\to\infty}(k*1/n+\cos(n))$ e
ste limite está entre -1 e +1, logo $f(n)\in O(g(n))$.

b-) **(1,0pt)** $f(n)=500*n^2+10000;g(n)=2^n$ **use dominância!!**

$500*n^2+10000\leqslant c*2^n$ para algum valor de c e n0.

Um argumento possível é escolher uma função h(n), fácil de analisar e que cresce mais rápido que f (tem que demonstrar isso) e demonstrar que h cresce menos que g. Desta forma, como f <=h<=g então f<=g.

$500*n^2+10000\leqslant h(n)=(500+10000)*n^2$ É verdade para qualquer n>0 pois equivale a $10000\leqslant(10000)*n^2$. O lado esquerdo é uma constante e o lado direito aumenta em 10000 a cada incremento de uma unidade em n **.**

? $(10000+500)*n^2\leqslant c*2^n$?
Se escolhermos c=(10000+500), então a desigualdade se reduz a $n^2\leqslant 2^n$ para um valor de n=1, a desigualdade já se verifica. Se incrementarmos n em uma unidade a desigualdade fica: $(n+1)^2\leqslant 2^{(n+1)}$ desenvolvendo temos $n^2+2*n+1\leqslant 2^n+2^n$ como por hipótese (de indução) $n^2\leqslant 2^n$ então precisamos nos preocupar apenas se $2*n+1\leqslant 2^n$ Note que quando n aumenta em uma unidade, o lado esquerdo aumenta em duas unidades enquanto o lado direito dobra, ou seja, para n>1, o lado direito cresce mais rápido que o lado esquerdo. Fica claro que $2*n+1\leqslant 2^n$ é verdadeiro e portanto $(10000+500)*n^2\leqslant c*2^n$ para c=(500+10000) e n>n0>1.

5-) **(1,5pt)** A recorrência $T(n)=\left\{\begin{matrix} k\ para\ n=1 \\ 2*T(\frac{n}{3})+\log(n)\ para\ n>1\end{matrix}\right\}$ descreve a complexidade de tempo de um algoritmo. Use o teorema mestre para demonstrar a que classe de complexidade o algoritmo pertence.

$a=2,b=3,f(n)=\log(n)$

Note que $0{,}5<\log_3 2<1$ (quanto é log na base 3 da raiz quadrada de 3? raiz de 3 é menor que 2?)

? $\log(n)\in\Theta(n^{\log_3 2})$? ou $0<\lim_{x\to\infty}(\frac{\log(n)}{n^{\log_3 2}})<\infty$? Não pois esse limite tende a zero, portanto não é o caso 2.

? $\log(n)\in O(n^{\log_3 2-\epsilon})$ Há várias soluções: a mais simples é tomar $0<\epsilon<\log_3 2$, por exemplo 0,1 e calcular o limite. Outra seria calcular o limite propagando épsilon e concluir que $0<\epsilon<\log_3 2$ satisfaz a condição.

$\lim_{x\to\infty}(\frac{\log(n)}{n^{\log_3 2-\epsilon}})<\infty$ Aplicando L'Hospital:

$\lim_{x \to \infty} \left(\frac{\frac{k}{n}}{(\log_3 2 - \epsilon) * n^{\log_3 2 - \epsilon - 1}} \right) < \infty$ simplificando:

$\lim_{x \to \infty} \left(\frac{k}{(\log_3 2 - \epsilon) * n^{\log_3 2 - \epsilon}} \right) < \infty$

a desigualdade é verdade quando $\log_3 2 - \epsilon > 0$.

6-) **(1,5pt)** A sequência de operações nas questões 1 a 5 apresentam uma forma de criar e analisar algoritmos. Explique como cada passo se conecta ao outro a fim de cumprir seu objetivo **(1pt)**. Você deseja resolver um problema e tem dois algoritmos para isso. Um com complexidade O(log(n)) e outro O(n^2). **Demonstre** qual dos dois é melhor (**0,5pt**).

A questão 1 pede para que um problema seja resolvido com um programa, ou que uma solução matemática seja implementada como um programa. Deseja-se que esse programa seja eficiente, então deve-se analisar sua eficiência. Um dos aspectos da eficiência é rapidez na execução. É possível medir diretamente o tempo de execução, mas essa medida é influenciada por fatores indesejáveis como arquitetura e organização da máquina, outros processos rodando na máquina, eventos gerados pelo usuário, … Para abstrair esses elementos, utiliza-se análise de complexidade (assintótica). Para empregar essa técnica, deve se extrair do código uma função matemática que devidamente escalada serve como limitante para o tempo de execução. Em algoritmos recursivos, esta função pode ser obtida em termos de uma recorrência, em que cada parcela corresponde a execução de algum trecho do código (questão 2). Essa recorrência pode ser desenvolvida (questão 3) e sua classe de complexidade determinada (questão 4) , ou testada usando o teorema mestre, também determinando sua classe de complexidade (questão 5). Dois algoritmos diferentes que resolvem o mesmo problema podem pertencer a classes de complexidade diferentes, considera-se melhor o que for mais eficiente.
Nesta questão, comparar as complexidades corresponde a dizer que uma função cresce mais rápido que a outra. Neste caso:

$\lim_{x \to \infty} \left(\frac{\log(n)}{n^2} \right) = 0$ logo $\log(n) \in o(n^2)$ e portanto escolhe-se o algoritmo com complexidade O(log(n))