Exercícios de Revisão

1. 1 - O **Casamento de Padrõe**s é um problema clássico em ciência da computação e é aplicado em áreas diversas como pesquisa genética, editoração de textos, buscas na internet, etc. Basicamente, ele consiste em encontrar as ocorrências de um padrão P de tamanho m em um texto T de tamanho n. Por exemplo, no texto T = “PROVA DE AEDSII” o padrão P = “OVA” é encontrado na posição 3 enquanto o padrão P = “OVO” não é encontrado. O algoritmo mais simples para o casamento de padrões é o algoritmo da “Força Bruta”, mostrado abaixo. Analise esse algoritmo e responda: Qual é a função de complexidade do número de comparações de caracteres efetuadas no melhor caso e no pior caso. Dê exemplos de entradas que levam a esses dois casos. Explique sua resposta!

```
typedef char TipoTexto[MaxTamTexto];
typedef char TipoPadrao[MaxTamPadrao];

void ForcaBruta(TipoTexto T, int n, TipoPadrao P, int m) {
  int i, j, k;
  for (i = 1; i <= (n - m + 1); i++) {
      k = i;
      j = 1;
      while (T[k-1] == P[j-1] && j <= m) {
         j++;
         k++;
      }
      if (j > m)
         printf(" Casamento na posição %3d\n", i);
    }
}
```

No melhor caso, o primeiro caractere do padrão casa com o texto. Nesse caso o while é sempre falso e são feitas n-m+1 comparações. Exemplo T = “PROVA DE AEDSII” P = “XXX”

No pior caso, o padrão casa o máximo de vezes com o texto. Nesse caso, para cada iteração do for são feitas m comparações no while, totalizando m x (n-m+1) comparações. Exemplo T = “AAAAAAAAAA” P = “AAA”

2. O que será impresso pelo programa abaixo. Explique a sua resposta

```
void inicializa(int v[10], int n) {
    n = 0;
    while (n<10) {
       v[n] = n;
       n++;
    }
}

int main()
{
    int v[10], n, i;

    inicializa(v, n);
    printf("%d\n", n);
    for(i=0; i<10; i++)
       printf("%d\n", v[i]);
}
```

O valor de n é indefinido, uma vez que a passagem de parâmetros em C é feita por valor e as modificações feitas dentro na função não são refletidas no programa principal.

Já o vetor é impresso: 0 1 2 3 4 5 6 7 8 9, pois todo vetor em C é um apontador, dessa as alterações feitas no conteúdo de memória “apontado” por v são refletidas no programa principal.

3. Considere o algoritmo abaixo. O que ele faz? Qual é a função de complexidade do número de comparações de elementos do vetor no melhor caso e no pior caso? Que configuração do vetor de entrada A leva a essas duas situações? Explique / Demonstre como você chegou a esses resultados. (Dica: analise para cada valor de i quantas vezes o while é executado no melhor e no pior caso, e monte um somatório...)

```
typedef int Vetor[MAX];

void Prova1(Vetor A, int n){
  int i, j, x;

  for(i=1; i<n; i++) {
     x = A[i];
     j = i - 1;
     while ( (j >= 0) && (x < A[j]) ) {
        A[j + 1] = A[j];
        j = j - 1;
     }
     A[j + 1] = x;
  }
}
```

O Algoritmo Ordena um vetor (algoritmo da inserção).

No melhor caso o vetor já está ordenado e o while nunca é executado. Nesse caso são feitas n-1 comparações

No pior caso o vetor está inversamente ordenado e o while é feito o número máximo de vezes para cada iteração do for. Especificamente:

i	# comparações
1	1
2	2
3	3
...	...
n-1	n-1

Número total de comparações é 1 + 2 + 3 + ... + n-1 = n x (n-1) / 2

4. Sejam f(n), g(n) duas funções assintóticas positivas e a e b. Prove que as afirmativas abaixo são verdadeiras ou falsas, usando para isso as definições das notações assintóticas ou contraexemplos.

a) $2^{n+1} = O(2^n)$

Verdadeiro:
Existe c tal que:
$2^{n+1} <= c2^n$
$2.2^n <= c2^n$
C >= 2

b) $2^{2n} = O(2^n)$

Falso:
Não existe c tal que:
$2^{2n} <= c2^n$
$2^n.2^n <= c2^n$
C >= 2^n -> Não existe constante

c) f(n) + g(n) = O (Max(f(n), g(n))

Verdadeiro:
Considere h(n) = Max(f(n), g(n)) ou seja h(n) = f(n) se f(n) >= g(n) ou h(n) = f(n) se f(n) < g(n)
Queremos provar que: f(n) + g(n) <= c . h(n)
Pela definição de h temos que h(n) >= f(n) e h(n) >= g(n) é sempre verdadeiro.
Logo somando os dois termos temos:
h(n) >= f(n)
h(n) >= g(n)

2h(n) >= f(n) + g(n). Logo c = 2 satisfaz a equação.

d) A notação θ é simétrica, ou seja, f(n) = θ(g(n)) se e somente se g(n) = θ(f(n))

Se f(n) = θ(g(n)) então existem constantes c1 e c2 tais que:
c1g(n) <= f(n) <= c2g(n)
c1g(n) <= f(n) -> g(n) <= (1/c1)f(n)
f(n) <= c2g(n) -> (1/c2)f(n) <= g(n).
Logo:
(1/c2)f(n) <= g(n) <= (1/c1)f(n)
g(n) = θ(f(n))
Mesmo raciocínio para provar o outro lado do *se e somente se*

5. Implemente uma função recursiva para computar o valor de 2^n

Considerando que n seja maior ou igual a 0

```
int expon(int n) {
    if (n == 0)
        return 1;
    else
        return 2*expon(n-1);
}
```

6. Resolva a seguinte questão sobre recursividade:

   a. Escreva uma **função recursiva** `int Palindromo(int esq, int dir, char palavra[])` que testa se uma determinada palavra é um palíndromo e retorna 1 em caso positivo e 0 em caso negativo. Um palíndromo é uma palavra que é lida da mesma forma da esquerda para direita ou da direita para esquerda (ex. ovo, arara). A palavra é passada para o função através de um vetor de caracteres limitada pelos os índices `esq` e `dir`, por exemplo: `Palindromo(0,4,"arara")`

```
int Palindromo(int esq, int dir, char palavra[])
{
    if (dir <= esq)
        return 1;
    else if(palavra[esq] != palavra[dir])
        return 0;
    else
        return Palindromo(esq+1,dir-1,palavra);
}
```

   b. Calcule qual é a **função de complexidade** para o número de comparações de caracteres da sua função no melhor caso e no pior caso. Para isso, **determine e resolva** a equação de recorrência dessa função recursiva. Qual é a **ordem de complexidade** de sua função?

No melhor caso, a primeira comparação é falsa e a o função retorna. Logo T(n) = 1;

No pior caso, quando a palavra é um palíndromo, são feitas n/2 comparações (considere n par apara simplificar). Esse resultado é obtido resolvendo a seguinte equação de recorrência:

T(n) = 1 + T(n-2); se n>=2

T(n) = 0; se n<2

Fazendo a expansão de termos

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

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

...

T(2) = 1 + T(0)

T(0) = 0

Logo T(n) = 1 + 1 + 1 + ... + 1 (n/2 vezes) = 1 . n/2

   c. Qual seria a complexidade de uma implementação não recursiva dessa mesma função? Qual das duas implementações vocês escolheria? Justifique.

A função não recursiva teria a mesma complexidade de tempo, mas o custo de memória seria maior devido aos registros em pilhados na pilha de ativação a cada chamada recursiva. Dessa forma, é melhor escolher a versão não recursiva.

7. O que faz a função abaixo? Explique o seu funcionamento.

```
int f(int a, int b) { // considere a > b
    if (b == 0)
       return a;
    else
       return f(b, a % b); //o operador % fornece o resto da divisão
}
```

A função acha o maior divisor comum de dois números. Ele faz isso através do método das divisões sucessivas. A cada passo o divisor passa a ser o numerador e o resto da divisão passa a ser o novo divisor. Isso é feito até que o resto seja zero, e nesse caso o numerador é retornado com resultado.
Por exemplo: a = 81, b = 42

```
81 42
42 39
39 3
3 0
MDC = 3
```

8. Vários algoritmos em computação usam a técnica de "Dividir para Conquistar": basicamente eles fazem alguma operação sobre todos os dados, e depois dividem o problema em sub-problemas menores, repetindo a operação. Uma equação de recorrência típica para esse tipo de algoritmo é mostrada abaixo. Resolva essa equação de recorrência usando o Teorema Mestre.

```
T(n) = 2T(n/2) + n;

T(1) = 1;
```

Pelo teorema Mestre

a = 2

b = 2

f(n) = n

$n^{\log_b a} = n$

logo $f(n) = \Theta(n^{\log_b a})$ -> caso 2 do teorema mestre. Nesse caso, $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$

Além destes exercícios sugiro os seguintes exercícios do livro texto (segunda edição):

Cap. 1: 2, 5, 7, 17, 18.

Cap. 2: 5, 6