Rodrigo de Figueiredo Soares - 202401826
Lista 1 - AED1

1. Dizer que uma função $g(n)$ é $O(f(n))$ é o mesmo que dizer que, se existem c, m inteiros constantes, se $n >= m$, então $g(n) <= c*g(n)$.

2. Dizer que uma função $g(n)$ é $\Theta(f(n))$ é o mesmo que dizer que, se existem c1, c2, m inteiros constantes, se $n >= m$, então $c1*f(n) <= g(n) <= c2*g(n)$.

3. Dizer que uma função $g(n)$ é $\Omega(f(n))$ é o mesmo que dizer que, se existem c, m inteiros constantes, se $n >= m$, então $g(n) >= c*g(n)$.

4. A partir de $n = 23$, 2^n se torna maior que n^5.

```
n = 20
n^5 = 3200000
2^n = 1048576

n = 21
n^5 = 4084101
2^n = 2097152

n = 22
n^5 = 5153632
2^n = 4194304

n = 23
n^5 = 6436343
2^n = 8388608

n = 24
n^5 = 7962624
2^n = 16777216

n = 25
n^5 = 9765625
2^n = 33554432
```

Portanto, eu utilizaria o algoritmo B a partir de um tamanho de dados de entrada de 23. O algoritmo é maior para DOIS casos em que $n < 23$, que é $n = 0$ - porém, não há sentido em determinar o custo de utilizar 0 entradas, visto que, desse modo, não estaríamos utilizando o programa -; e para $n = 1$; logo, se eu fosse rodar apenas uma entrada num programa, utilizaria o algoritmo B também, ainda que, com apenas uma entrada, dificilmente seja notável a diferença entre o tempo dos dois.

5. i) O problema de reprodução de bactérias é um que costuma ser exponencial, visto que é possível prever a quantidade de bactérias presentes em uma população com base na quantidade inicial de bactérias e na quantidade de reproduções que elas realizaram, dado que, a cada reprodução, a população dobra de tamanho, portanto, teríamos $p*2^n$ bactérias, em que p é a população inicial, e n é a quantidade de reproduções realizadas.

ii) Outro problema que costuma ser exponencial é o de decaimento radioativo, em que, a cada ciclo de meia-vida de uma substância radioativa, seja possível prever quanto de massa daquele material radioativo ainda se tem, e quanto já sofreu uma meia-vida, dado que, a cada meia vida, a massa decai pela metade, portanto, teríamos m*(1/(2^n)), em que m é a massa inicial, e n é a quantidade de meias-vidas ocorridas.

6. a) $f(n) = 2n + 10 \rightarrow 2n \rightarrow n$. Logo, $O(n)$.

b) $f(n) = 1/2n * (n + 1) = 1n/2n + 1/2n = 1/2 + 1/2n \rightarrow 1/2n$. Como quanto mais aumentamos n, menor fica 1/2n, temos que o pior caso é n = 1, o que dá 1/2, logo, $O(1)$.

c) $f(n) = 1/2\ n^2 \rightarrow n^2$. Logo, $O(n^2)$.

d) $f(n) = 1/2\ n^4 – 3n^2 + 5n + 7 \rightarrow 1/2\ n^4 - 3n^2 + 5n \rightarrow 1/2\ n^4 \rightarrow n^4$. Logo, $O(n^4)$.

e) $f(n) = 7n + 3 \log_2(n) + 20 \rightarrow 7n + 3 \log_2(n)$. $n > \log_2(n)$, logo, temos $7n \rightarrow n$. Logo, $O(n)$.

f) $f(n) = n! + 5n^2 + 10 \rightarrow n! + 5n^2$. $n! > n^2$, logo, temos $n!$. Logo, $O(n!)$.

g) $f(n) = 3 * 5000^n + 1000 \rightarrow 3*5000^n \rightarrow 5000^n$. Logo, $O(5000^n)$.

h) f(n) = 10^10. Logo, $O(1)$.

7.

int i,j,k;

for(i = 0; i < N; i++) {

1 atribuição + 1 comparação + n comparações e n incrementos = $2 + 2n$

for(j=0; j < N; j++) {

n*1 atribuições + n*1 comparações + n*n comparações + n*n incrimentos = $2n + 2n^2$

R[i][j] = 0;

1 atribuição n vezes, n vezes = n^2

for(k=0; k < N; k++)

n*n*1 atribuições + n*n*1 comparações + n*n*n comparações + n*n*n incrementos = $2n^2 + 2n^3$

R[i][j] += A[i][k] * B[k][j];

1 atribuição n vezes, n vezes, n vezes = n^3

}

}

total = $n^3 + 2n^3 + 2n^2 + 2n^2 + 2n + 2n + 2 = 3n^3 + 4n^2 + 4n + 2$.

8.

```
int verifica_ordenado(int * v, int tam) {
        for(int i=0;i<tam-1;i++) {
        1 atribuição + 1 comparação + tam-1 comparações + tam-1 incrementos = 2 +
2tam - 2 = 2*tam
                if(v[i] > v[i+1]) return 0;
                (tam-1)*1 comparações
        }
        return 1;
}
```

total = tam-1 + 2*tam = 3*tam-1. Se entendermos n como o tamanho do vetor, então temos 3*n - 1.

9.

```
int busca_normal(int * v, int tam, int elem) {
        for(int i=0;i<tam;i++) {
1 atribuição + 1 comparação + tam comparações + tam incrementos = 2 + 2tam
                if(v[i] == elem) return 1;
                tam*1 comparações
        }
}
```

total = 2 + 2tam + tam = 3*tam + 2. Assumindo n como o tamanho do vetor, temos 3*n + 2.

3*n + 2 → 3*n → n. Logo, O(n).
No melhor caso, encontramos o elemento logo de cara, portanto, 1 + 1 + 1 = 3 operações, portanto, Ω(3) = Ω(1).

```
int busca_binaria(int * v, int tam, int elem) {
        int inicio = 0, meio;
        1 atribuição
        while(1) {
                meio = (inicio+tam)/2;
                1 atribuição
                if(v[meio] == elem) return 1;
                1 comparação
                if(inicio == tam) return 0;
                1 comparação
                else if(elem > v[meio]) inicio = meio+1;
                1 comparação + 1 atribuição = 2
                else if(elem < v[meio]) tam = meio-1;
                1 comparação + 1 atribuição = 2
        }
```

}
total = 1 + 1 + 1 + 2 (apenas uma das três últimas comparações podem acontecer, portanto, no pior caso, uma das que executam duas operações acontecem) = 5. Porém, essas 5 operações acontecem todas as vezes que o elemento não for encontrado, e, no pior caso, pegamos uma parte “metade” do vetor e não encontramos o elemento. Logo, temos tam/2 até que tam = 1. Logo, 1 = tam/(2^n), logo, teríamos 5 operações para cada divisão, logo, temos 5n. Isso nos dá O(n).
Para o melhor caso, temos que o elemento foi encontrado logo de cara, portanto, 1 + 1 + 1 = 3, portanto Ω(3) = Ω(1).