Micael Andrade Dos Santos

1. Questão

1.

Caso Base:

$$P(1) = \frac{1(1+1)^2}{4} = 1$$, válido para $P(1)$

Agora vamos supor que vale para $P(k)$. Logo,

$$1^3 + 2^3 + \dots k^3 = \frac{k^2(k+1)^2}{4} \quad \text{HIP}$$

Passo Indutivo:

Temos que, se vale para $P(k)$, deve valer para $P(k+1)$.

$$1^3 + 2^3 + \dots + k^3 + (k+1)^3 = \frac{(k+1)^2(k+2)^2}{4} \quad \text{ASSOCIATIVIDADE}$$

$$(1^3 + 2^3 + \dots + k^3) + (k+1)^3 = \frac{(k+1)^2(k+2)^2}{4} \quad \text{HIPÓTESE}$$

Álgebra

$$\frac{k^2(k+1)^2 + 4(k+1)^3}{4} = \frac{(k+1)^2(k+2)^2}{4}$$

$$\frac{[k.(k+1)]^2 + 4(k^3+3k^2+3k+1)}{4} = \frac{(k+1)^2(k+2)^2}{4}$$

$$\frac{(k^2+k)^2 + 4k^3+12k^2+12k+4}{4} = \frac{(k+1)^2(k+2)^2}{4}$$

$$\frac{k^4+2k^3+k^2+4k^3+12k^2+12k+4}{4} = \frac{(k+1)^2(k+2)^2}{4}$$

$$\frac{k^4+6k^3+13k^2+12k+4}{4} = \frac{(k^2+2k+1)(k^2+4k+4)}{4}$$

$$\frac{k^4+6k^3+13k^2+12k+4}{4} = \frac{k^4+4k^3+4k^2+2k^3+8k^2+8k+k^2+4k+4}{4}$$

$$\frac{k^4+6k^3+13k^2+12k+4}{4} = \frac{k^4+6k^3+13k^2+12k+4}{4}$$

$\blacksquare$

---

2. Questão

Casos Bases:

$n=0$, $n=1$, $n=2$: Para os três casos temos que
$n=3$.
a asserção é válida, pois

$0 \leq 1$, $1 \leq 3$, $2 \leq 9$ e $3 \leq 27$.

HIPÓTESE: Agora vamos supor que $h_j \leq 3^j$ para todos inteiros j, tal que $1 \leq j < k$.

PASSO INDUTIVO: como assumimos que a hipótese é válida para j, sendo que $1 \leq j < k$, então também vale para k. Logo $h_k \leq 3^k$

PROVA:

Por definição temos que:

$h_k = h_{k-1} + h_{k-2} + h_{k-3}$. Como $k-1$, $k-2$ e $k-3$ são menores que k, pela HIP temos $h_{k-1} \leq 3^{k-1}$, $h_{k-2} \leq 3^{k-2}$ e $h_{k-3} \leq 3^{k-3}$ $\therefore h_k = h_{k-1} + h_{k-2} + h_{k-3} \leq 3^{k-1} + 3^{k-2} + 3^{k-3} =$ $3^{k-3}(3^2+3+1) = 3^{k-3}(13)$.

$$h_k \leq 3^{k-3}(13)$$

$$h_k \leq 27(3^{k-3})$$

$$h_k \leq 3^3(3^{k-3})$$

$$\boxed{h_k \leq 3^k}$$

3. Questão

   a. Podemos estruturar o problema da seguinte forma indutiva: quando n for 1, temos um único elemento, logo ele é o maior.

`CASO BASE:` $n1 = x[1]$

`HIPÓTESE:` Suponha que conseguimos encontrar o maior valor do vetor x para $n-1$ elementos.

`CASO GERAL:` Agora precisamos encontrar o maior valor para n. Como o maior valor está no range $[1, 2, 3...k-1, k]$, basta calcular o maior valor entre $1...k-1$ e verificar se esse valor é maior do que k. Se for, então encontramos, caso contrário k é o maior valor.

b. Pseudo-code

```
algoritmo maxx(x, n, maior)
{- ENTRADA: Vetor x cujo elementos de x pertencem ao conjunto dos Reais.
SAIDA: Maior elemento de X
-}
início
se n = 1 então
    retorne maior
se (x[n-1] >= maior) então
  retorne maxx(x, n-1, x[n-1]) --x[n-1] passa a ser o maior
senão
  retorne maxx(x, n-1, maior)
fim
max(x, n, x[n]) --x[n]assume o último elemento de x como maior.
```

---

5. Questão

   a. A solução foi elaborada em duas partes para atender os itens I e II. Primeiro, temos uma função `contZeroOne` que recebe uma string e conta a quantidade de Zeros e Uns dessa string e retorna True caso essa quantidade seja igual, False caso contrário.

      Segundo, temos uma função `binary` recursiva exclusiva para verificar o item II. Essa função varre a string da esquerda para direita e realiza a contagem de 1s e 0s, caso em algun momento a quantidade de 1s exceda a quantidade 0s naquela posição, então retornamos False. Porém essa função não garante a validade do item I, logo precisamos invocar `contZeroOne` no final.

   b. Pseudo-code

```
algoritmo contZeroOne(stringBinaria, n)
{- ENTRADA: Vetor de carácteres e seu respectivo tamanho.

SAIDA: True quando a quantidade de 0s e 1s forem iguais, false caso contrário.
-}
início
totalZeros := 0
totalUns := 0
para i = 1 até n faça
    início
        se stringBinaria[i] == '0' então
            totalZeros := totalZeros + 1
        senão:
            totalUns := totalUns + 1
    fim
retorne totalUns == totalZeros
fim
```

```
algoritmo binary(stringBinaria, i, n, qt0, qt1)
{- ENTRADA: Vetor de carácteres, posição inicial da string,
tamanho da string, quantidade de 0s e quantidade de 1s,
respectivamente.

SAIDA: True se caso seja uma strnig binária válida, false caso contrário.
-}
início
    se qt1 > qt0 então
        retorne False
    senão
        se(i<=n) então
            se(x[i] = '1') então
                retorne binary(stringBinaria, i+1, n, qt0, qt1+1)
```

```
            senão
                retorne binary(stringBinaria, i+1, n, qt0+1, qt1)
    retorne contZeroOne(stringBinaria, n)
fim
```

c. Complexidade

`contZeroOne` é $O(n)$ e o espaço utilizado é proporcinal à string, ou seja, $1byte * n + 4bytes$ (para armazenar o tamanho de n.
`binary` é $O(n)$, porém ela chama `contZeroOne` no final. Logo, a complexidade total seria $O(2n)$, ou $O(n)$. O espaço consumido seria $1byte * n + 4bytes + 4bytes + 4bytes + 4bytes$.

---

6. Questão

i. Sem gastar espaço adicional

a. Podemos usar dois laços para realizar essa verificação. Um externo que fixa um valor o qual será comparado com os próximos valores.

b. Pseudo-code

```
algoritmo isDuplicated(x, n)
{- ENTRADA: Vetor x de n elementos.

SAIDA: True quando encontra um elemento duplicado, False caso contrário.
-}
início
para i := 1 até n faça
    início
        para j := i+1 até n faça
            início
                se (x[i] = x[j]) então
                    retorne True
            fim
    fim
retorne False
fim
```

c. Complexidade
Para cada valor x temos que `isDuplicated` irá percorrer $n - 1$ vez o vetor novamente.
Logo a complexidade seria, $O(n^2)$. Além disso, o espaço utilizado será proporcinal ao tamanho de do vetor de entrada $(n * 4Bytes)$

ii. Podendo gastar espaço adicional

a. Para aprimorar o tempo de execução do algoritmo anterior, podemos utilizar uma tabela Hash com tamanho n, sendo que n é tamanho de entrada do vetor. Cada chave dessa tabela será todos os valores do vetor de entrada, e o valor de cada chave será justamente o total de vezes que esse valor foi encontrado.

b. Pseudo-code

```
algoritmo isDuplicatedWithHash(x, n)
{- ENTRADA: Vetor x de n elementos.

SAIDA: True quando encontra um elemento duplicado, False caso contrário.
-}
início
hashTable := Null
para i := 1 até n faça
```

```
    início
        hashTable[i] = 0;
    fim

para i:=1 até n faça
    hashTable[x[i]] := hashTable[x[i]] +  1
    se(hashTable[x[i]]) >= 2 então
        retorne True
retorne False
fim
```

c. Complexidade

`isDuplicatedWithHash` apesar de necessitar de uma tabela Hash para armazenar todos os valores x, temos que em tempo de excução ela é melhor que a primeira abordagem. Pois gasta $O(n)$ tempo no pior caso. $n * 4Bytes + hashTable * n$

---

7. Questão

i. Vetor desordenado

a. Nessa caso $\forall e \in A$ devemos verificar se $e \in B$, para realizar essa verificação podemos usar uma busca simples para percorrer o conjunto B. Podemos utilizar uma `Flag` que irá ter seu valor como false caso o mesmo elemento esteja tanto em A quanto em B.

b. Pseudo-code

```
algoritmo AcomplementoBSimple(A, An,B,Bn)
{- ENTRADA: Vetor A de An elementos e vetor B com Bn elemetos.

SAIDA: Um vetor W talque w = A-B
-}
início
    W := []
    para i:=1 até An faça
      inicio
        eDiferente := True
        para j:=1 até Bn faça
          inicio
            se(A[i] == B[j]) então
                eDiferente = False
                pare
          fim
        se(eDiferente) então
            W := W + A[i]
        fim
    retorne W
fim
```

c. Complexidade

Sabemos que para cada elemento em A devemos percorrer B para verificar se existe elementos iguais. Logo a complexidade seria $O(n^2)$, visto que as outras instruções é $O(1)$.
Espaço necessário $An * 4bytes + Bn * 4Bytes + WAn * 4Bytes$

ii. Vetores estão ordenados

a. A estratégia é a mesma da anterior, porém ao invés de realizar uma busca simples, podemos realizar uma busca binária pois temos a garantia que os vetores estão ordenados. O algoritmo consiste em duas funções `AcomplementoBBinary` a qual retorna um novo vetor com A-B e `bi_search` a qual será utilizada para realizar a verificação.

b. Pseudo-code

```
algoritmo bi_search(B, Bn, x)
{- ENTRADA: Vetor B com Bn elementos e um elemento x.

SAIDA: True caso x esteja em B, false caso contrário.
-}
início
  baixo := 1
  alto := Bn
      enquanto(baixo < alto) faça
          meio := div((baixo+alto), 2) -- Divisão inteira
          se(B[meio] = element) então
              return True --Existe
          se não
            se (element > B[meio]):
              baixo := meio + 1
            senão:
              alto := meiodef AcomplementoBBinary(A: int, An: int, B: int, Bn: int) -> list:  # O(n.log(n))
    W = []
    for i in range(0, An):  # O(n)
        if(bi_search(B, A[i]) == None):  # O(log(n))
            W.append(A[i])
    return W
      return False
fim
```

```
algoritmo AcomplementoBBinary(A, An, B, Bn)
{- ENTRADA: Vetor A com An e um vetor B com Bn elementos.

SAIDA: Um vetor W, tal que w = A-B
-}
início
    W := []
    para i =: 1 até An faça
        se (!bi_search(B, Bn, A[i])) então --Não encontrou
            W := w + A[i]
    return W
fim
```

c. Como a complexidade da buscabinária é $O(log(n))$, lém disso `complementoBBinary` possui $O(n)$, logo a complexidade total seria $O(n * log(n))$.
Temos que a capacidade de armazenamento exigida seria $An * 4bytes + Bn * 4bytes + WAn * 4bytes + 4bytes + 4bytes + 4bytes$