ANY CAROLINY SOUZA SILVA
MATRÍCULA: 201900016796
LISTA 1 - PAA - TURMA 03

1. Prove por indução matemática que a soma dos cubos dos n primeiros números inteiros positivos, $n \geq 1$, é igual ao quadrado da soma destes números.
Variável de indução: n, a quantidade de termos.
Caso base: considerando que a soma dos termos dos n primeiros inteiros positivos é $\frac{n(n+1)}{2}$, para $n_0 = 1$ temos

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

Hipótese de indução: supondo-se que a fórmula seja verdadeira para um valor arbitrário $n = k$, temos

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

Caso geral: estendendo a veracidade da asserção para $n = k + 1$, deseja-se provar que

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

Usando o lado esquerdo da igualdade temos
$1^3 + 2^3 + \ldots + (k+1)^3 =$
$= 1^3 + 2^3 + \ldots + k^3 + (k+1)^3$ *(associatividade da soma)*
$= (1^3 + 2^3 + \ldots + k^3) + (k+1)^3$ *(hipótese de indução)*

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

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

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

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

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

2. Seja T uma árvore binária cheia de altura h, h ≥ 0. Prove por indução matemática que T tem $n = 2^{h+1} - 1$ nós. Uma árvore binária é dita cheia quando todos os nós ou são folhas ou possuem exatamente dois filhos.

Variável de indução: h, a altura da árvore binária.
Caso base: para $n_0 = 0$, a altura da árvore é $n = 2^{0+1} - 1$ nós, isto é $n = 2 - 1 => n = 1$ nó, ou seja, o único nó presente na árvore é a raiz.
Hipótese de indução: supondo que a fórmula seja verdadeira para $h = k$, sendo k um

inteiro arbitrário, temos que a árvore possui $n = 2^{k+1} - 1$ nós.
Caso geral: deseja-se estender a veracidade da fórmula para uma árvore de altura $h = k + 1$. Ou seja deseja-se provar que a árvore binária tem $n = 2^{k+2} - 1$ nós.
Utilizando a hipótese de indução, ao aumentar a altura da árvore em uma unidade, devem ser acrescidos dois nós filhos para cada nó do nível h = k. E sabendo que para o nível k da árvore binária existem 2^k nós, temos
$n = 2^{k+1} - 1 + 2.2^k$
$n = 2^{k+1} - 1 + 2^{k+1}$
$n = 2^{k+1}(1 + 1) - 1$
$n = 2.2^{k+1} - 1$
$n = 2^{k+2} - 1$ como queríamos demonstrar.

4. Dado um vetor de inteiros de n elementos, elabore um algoritmo para determinar quantos destes números são negativos.

(a) Estruturação da solução por indução (fraca).
Variável de indução: n, o tamanho do vetor.
Caso base: $n_0 = 0$, ou seja, para o vetor vazio não há nenhum elemento negativo, obviamente verdade.
Hipótese de indução: assumindo-se que sabemos determinar a quantidade de valores negativos para um vetor de tamanho $n = k - 1$.
Caso geral: deseja-se determinar a quantidade de valores negativos para o tamanho $n = k$.
Estratégia: testar para todos os $k - 1$ elementos se eles são negativos se utilizando da hipótese de indução, a medida que se acumula a quantidade de condições verdadeiras em uma variável. Por fim, incrementa-se uma unidade a variável acumulada para o caso do termo $n = k$ ser negativo, caso contrário, o valor acumulado não é alterado.

(b) derivação do algoritmo recursivo em pseudo-linguagem a partir da solução do item (a)

```
algoritmo contaNegativos(v, n)  {

{- Entrada: v um vetor de inteiros e n o tamanho deste vetor
Saída: valor inteiro representando a quantidade de valores negativos -}
        neg := 0
        se n < 0 -- Caso base
                então retorne neg

        neg := contaNegativos(v, n-1) -- HI

        se v[n] < 0 -- Caso geral
                então neg := neg+1
```

```
        retorne neg
 }
```

5. Dada uma árvore binária de inteiros e dois limites de intervalo, inferior e superior, determine a soma dos elementos da árvore que estão neste intervalo de valores (intervalo fechado).

(a) estruturação da solução por indução;
Variável de indução: n, a quantidade de nós da árvore.
Caso base: para uma árvore vazia, independentemente do intervalo, a soma dos valores é 0.
Hipótese de indução: assume-se que saibamos calcular a soma dos elementos do intervalo da árvores com r nós, sendo $0 \leq r < k$.
Caso geral: deseja-se calcular a soma dos elementos no intervalo determinado de uma árvore com k nós.
Estratégia: removendo-se a raiz da árvore, são obtidas duas subárvores, a da esquerda e a da direita. Sabe-se que ambas subárvores possuem necessariamente tamanho menor que k, dado que a raiz foi removida. Deste modo, pela hipótese de indução sabemos calcular a a soma dos elementos para as subárvores e a estratégia de solução é testar para os elementos das mesmas se eles estão dentro do intervalo determinado e somá-los, por fim, incrementar o valor da raiz à soma para o caso de pertencer ao intervalo.

(b) derivação do algoritmo recursivo em pseudo-linguagem a partir da solução do item (a)

```
algoritmo addElementos(raiz, i, j) {
{- Entrada: raiz denota o nó raiz de uma árvore binária de inteiros, i
corresponde ao início do intervalo e j ao fim
Saída: valor inteiro denotando a soma dos elementos dentro do intervalo -}
        se raiz == -1 -- Caso base
                então retorne 0

        -- HI
        nosEsq = addElementos(raiz.esq,i,j)
        nosDir = addElementos(raiz.dir,i,j)

        se raiz.dado >= i && raiz.dado <= j então -- Caso geral
                retorne raiz.dado + nosEsq + nosDir
        senão então
                retorne nosEsq + nosDir
}
```

6. Dados dois vetores de inteiros ordenados em ordem crescente, gere um terceiro vetor com os dados dos vetores de entrada também ordenados em ordem crescente, mas sem elementos repetidos.

(a) descreva a ideia da sua estratégia de solução;
A estratégia de solução consiste em percorrer ambos os vetores uma vez e comparar seus elementos, se aproveitando da ordenação crescente. Assim, na comparação entre dois valores, o valor menor é previamente inserido no vetor união, a fim de assegurar que não haverá outro valor superior, já o valor maior é previamente ignorado, enquanto no caso de igualdade o valor é inserido e prossegue-se com a comparação dos valores adjacentes em ambos vetores.

(b) escreva o algoritmo em pseudo-linguagem;

```
algoritmo uniaoOrdenada(v1, v2, n1, n2) {
{- Entrada: v1 e v2 são vetores de inteiros ordenados crescentemente, n1 e n2
são seus respectivos tamanhos
Saída: vetor união dos vetores v1 e v2 -}
        i := 1
        j := 1
        u := 1

        enquanto i < n1 && j < n2 faça
                se v1[i] < v2[j] então
                        uniao[u] = v1[i]
                        u := u+1
                        i := i+1
                senão se v1[i] > v2[j] então
                        uniao[u] = v2[j]
                        u := u+1
                        j := j+1
                senão então
                        uniao[u] = v1[i]
                        u := u+1
                        i := i+1
                        j := j+1

        enquanto i < n1 faça
                uniao[u] := v1[i]
                u := u+1
                i := i+1

        enquanto j < n2 faça
                uniao[u] := v2[j]
```

```
                u := u+1
                j := j+1

        retorne uniao
}
```

(c) calcule a complexidade de tempo e espaço do algoritmo
Complexidade de espaço:
O espaço total utilizado da memória é:

- 1 inteiro para armazenar a variável i;
- 1 inteiro para armazenar a variável j;
- 1 inteiro para armazenar a variável u;
- 1 inteiro para armazenar a variável $n1$;
- 1 inteiro para armazenar a variável $n2$;
- $n1$ inteiros para armazenar os elementos de um vetor;
- $n2$ inteiros para armazenar os elementos do outro vetor.

Complexidade de espaço:
A quantidade total de memória usada pelo algoritmo *uniaoOrdenada* é $n1 + n2 + 5$, em função do tamanho da entrada.

Complexidade de tempo:
Visto que os vetores de entrada são sempre percorridos em sua totalidade, a complexidade de tempo no pior caso considerando, por exemplo, que $n1$ seja o vetor de tamanho maior é dada por
$T(n1) = 3T_{:=} + (T_{<} + T_{\&\&} + T_{<} + T_{><=}) + (T_{<} + T_{<})$
$= c_1 + \sum_{i=1}^{n1}(c_2) + c_3$
$= c_1 + (n1 - 1 + 1)c_2 + c_3$
$= c_1 + n1c_2 + c_3$
$= n1$
Considerando n como o tamanho do vetor de entrada maior, ou os tamanhos dos vetores caso fossem iguais, temos que a função que descreve a complexidade de tempo do algoritmo *uniaoOrdenada* é $T(n) = n$, que é linear em função do tamanho da entrada, deste modo, $T(n) = O(n)$.

7. Dados dois vetores de inteiros A e B, gere um terceiro vetor representando o vetor interseção dos vetores originais, nas duas situações abaixo. O vetor de interseção não pode ter elementos duplicados.

(a) A e B estão ordenados, em ordem decrescente;

(a) descreva a ideia da sua estratégia de solução;
A estratégia de solução consiste em percorrer ambos os vetores uma vez e comparar

seus elementos, se aproveitando da ordenação decrescente. Assim, na comparação entre dois valores, se o valor do 1º vetor é superior ao do 2º, deve-se percorrer o 2º em busca de valores menores até encontrar um valor possivelmente igual. Para o caso de encontrar um valor igual este é inserido no vetor interseção.

(b) escreva o algoritmo em pseudo-linguagem;

```
algoritmo intersecaoOrdenada(v1, v2, n1, n2) {
{- Entrada: v1 e v2 são vetores de inteiros ordenados decrescentemente, n1 e
n2 são seus respectivos tamanhos
Saída: vetor intersecao dos vetores v1 e v2 -}
        i := 1
        j := 1
        k := 1

        enquanto i < n1 && j < n2 faça
                se v1[i] < v2[j] então
                        j := j+1
                senão se v1[i] > v2[j] então
                        i := i+1
                senão então
                        intersecao[k] = v1[i]
                        k := k+1
                        i := i+1
                        j := j+1

        retorne intersecao
}
```

(c) calcule a complexidade de tempo e espaço do algoritmo
O espaço total utilizado da memória é:

- 1 inteiro para armazenar a variável i;
- 1 inteiro para armazenar a variável j;
- 1 inteiro para armazenar a variável k;
- 1 inteiro para armazenar a variável $n1$;
- 1 inteiro para armazenar a variável $n2$;
- $n1$ inteiros para armazenar os elementos de um vetor;
- $n2$ inteiros para armazenar os elementos do outro vetor.

Complexidade de espaço:
A quantidade total de memória usada pelo algoritmo *intersecaoOrdenada* no pior caso é $n1 + n2 + 5$, em função do tamanho da entrada.

Complexidade de tempo:
Considerando que $n1$ seja o vetor de tamanho menor, para o pior caso a complexidade de tempo é dada por
$T(n) = 3T_{:=} + (T_{<} + T_{\&\&} + T_{<} + T_{<} + T_{>} + T_{=})$
$= c_1 + (\sum_{i=1}^{n1})c_2$
$= c_1 + (n1 - 1 + 1)c_2 + c_3$
$= c_1 + n1c_2 + c_3$
$= n1$
Considerando n como o tamanho do menor vetor e abstraindo as constantes temos que a função que descreve a complexidade de tempo do algoritmo *intersecaoOrdenada* é $T(n) = n$, que é linear em função do tamanho da entrada, deste modo, $T(n) = O(n)$.

(b) A e B estão em ordem arbitrária

(a) descreva a ideia da sua estratégia de solução;
A estratégia de solução consiste em percorrer um dos vetores uma vez enquanto o outro é percorrido para todos os elementos do primeiro. Assim, realiza-se uma comparação entre os valores do primeiro vetor com todos os valores do segundo até que possivelmente se encontre igualdades que possam ser inseridas no vetor interseção.

(b) escreva o algoritmo em pseudo-linguagem;

```
algoritmo intersecaoArbitraria(v1, v2, n1, n2) {
{- Entrada: v1 e v2 são vetores de inteiros ordenados arbitrariamente, n1 e n2
são seus respectivos tamanhos
Saída: vetor intersecao dos vetores v1 e v2 -}
        k := 1
        para i=1 até n1 faça
                para j=1 até n2 faça
                        se v1[i] = v2[j]
                                intersecao[k] = v1[i]
                                k := k+1
        retorne intersecao
}
```

(c) calcule a complexidade de tempo e espaço do algoritmo

O espaço total utilizado da memória é:

- 1 inteiro para armazenar a variável i;
- 1 inteiro para armazenar a variável j;
- 1 inteiro para armazenar a variável k;

- 1 inteiro para armazenar a variável $n1$;
- 1 inteiro para armazenar a variável $n2$;
- $n1$ inteiros para armazenar os elementos de um vetor;
- $n2$ inteiros para armazenar os elementos do outro vetor.

Complexidade de espaço:
A quantidade total de memória usada pelo algoritmo *intersecaoArbitraria* no pior caso é $n1 + n2 + 5$, em função do tamanho da entrada.

Complexidade de tempo:
Considerando que no pior caso, ambos os vetores possuíssem o mesmo tamanho n, a complexidade de tempo é dada por
$T(n) = T_{:=} + (T_{[]} + T_{=} + T_{[]} + T_{[]} + T_{=} + T_{[]} + T_{+} + T_{:=})$
$= c_1 + (\sum_{i=1}^{n}\sum_{j=1}^{n})c_2$
$= c_1 + (\sum_{i=1}^{n1}(n - 1 + 1))c_2$
$= c_1 + ((n - 1 + 1)(n - 1 + 1))c_2$
$= c_1 + n^2c_2$
Abstraindo as constantes temos que a função que descreve a complexidade de tempo do algoritmo *intersecaoArbitraria* é $T(n) = n^2$, que é quadrática em função do tamanho da entrada, deste modo, $T(n) = O(n^2)$.

8. Dada uma expressão contendo parênteses, literais e operadores aritméticos, elabore um algoritmo para determinar se a expressão está com os parênteses balanceados, ou seja, se para cada parêntese aberto há um parêntese fechando e se os pares de parênteses estão adequadamente aninhados.

(a) descreva a ideia da sua estratégia de solução;
Tratando a expressão como uma *string*, a estratégia da solução consiste em empilhar todos os caracteres que são parênteses abertos, e desempilhar a medida que o caractere é um parêntese fechado. Caso haja a ocorrência de um parêntese fechado e a lista esteja vazia, o algoritmo encerra com retorno de expressão desbalanceada. Caso a lista não esteja vazia ao fim da iteração na expressão, o algoritmo retorna que a expressão é desbalanceada, caso contrário, é balanceada.

(b) escreva o algoritmo em pseudo-linguagem;

```
pilha() {
        topo := -1
}

ehVazia(p){
        se p.topo < 0:
                então retorne Verdadeiro
```

```
        retorne Falso
}

empilha(p, e) {
        p.topo := p.topo + 1
        p[p.topo] = e
}

desempilha(p) {
        p.topo := p.topo - 1
}

algoritmo testaExpressao(expressao, n) {
{- Entrada: expressao denota uma expressao do tipo string e n o tamanho da
mesma
Saída: booleano indicando se os parênteses estão balanceados -}
        p := pilha()
        para i=1 até n faça
                se expressao[i] = '('
                        então empilha(p, expressao[i])
                senão se expressao[i] = ')' && ehVazia(p)
                        então retorne Falso
                senão se expressao[i] = ')'
                        entao desempilha(p)

                retorne ehVazia(p)
}
```

(c) calcule a complexidade de tempo e espaço do algoritmo:

- 1 inteiro para armazenar a variável i;
- 1 inteiro para armazenar a variável n;
- n inteiros para armazenar os caracteres empilhados.

Complexidade de espaço:
A quantidade total de memória usada pelo algoritmo *testaExpressao* no pior caso é $2n + 1$, em função do tamanho da entrada.

Complexidade de tempo:
Considerando que no pior caso, ambos os vetores possuíssem o mesmo tamanho n, a complexidade de tempo é dada por
$T(n) = T_{[]} + T_{=} + T_{[]} + T_{=} + T_{\&\&} + T_{[]} + T_{=}$
$= c_1 + (\sum_{i=1}^{n})c_2$
$= (n - 1 + 1))c_2$

$= nc_2$

Abstraindo as constantes temos que a função que descreve a complexidade de tempo do algoritmo *testaExpressao* é $T(n) = n$, que é linear em função do tamanho da entrada, deste modo, $T(n) = O(n)$.