Daniel Dornelas

Exercícios - Matemática da Computação

2.1

pais ≝ Maria José, Mario Cardoso.

filhos ≝ Daniel Dornelas, Gabriel Dornelas, Melina Dornelas.

2.2

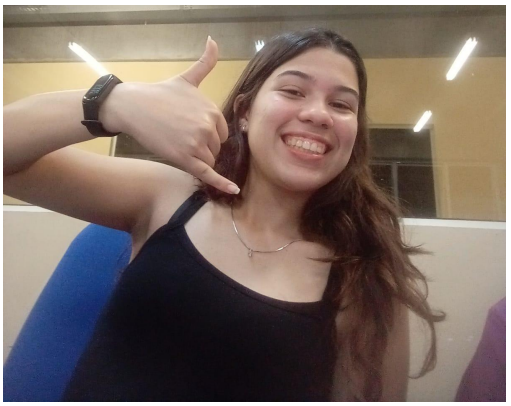
Cursos ≝ Desenvolvimento de Software Multiplataforma, Geoprocessamento, Meio Ambiente e Recursos Hídricos.

2.3

Números primos ≝ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, ...

2.4

Bianca Lucas da Silva Caçula ≝



2.5

1. Primeiro elemento ≝ 2
2. Termos da sequência numérica ≝ X_k, com X_k expresso pela fórmula $X_k = 3. X_{k-1}$.
3. Sequência numérica ≝ {2, 6, 18, 54, ...}

2.6

A. Linguagem em python:

```
from enum import Enum

class Meses(Enum):

JANEIRO = 1
FEVEREIRO = 2
MARCO = 3
ABRIL = 4
MAIO = 5
 JUNHO = 6
JULHO = 7
AGOSTO = 8
SETEMBRO = 9
OUTUBRO = 10
NOVEMBRO = 11
DEZEMBRO = 1
```

B. Linguagem em python:

```
from enum import Enum

class Dia_semana(Enum):

JANEIRO = 1
FEVEREIRO = 2
MARCO = 3
ABRIL = 4
MAIO = 5
 JUNHO = 6
JULHO = 7
AGOSTO = 8
SETEMBRO = 9
OUTUBRO = 10
NOVEMBRO = 11
 DEZEMBRO = 12
```

C. Linguagem em Javascript:

```
function fatorial(n) {
        if (n == 0 || n == 1) {
        return 1;  /* Caso base: fatorial de 0 e 1 é 1 */
        } else {
        return n * fatorial(n - 1);  /* Definição recursiva */
        }
}
```

D. Linguagem em Javascript:

```
class Graduando {

    private name: string;
    private curso: string;

    constructor(name: string, curso: string) {

        this.name = name;

        this.curso = curso;

    }

    cumprimentar_dds() {

        console.log(`Olá, meu nome é ${this.name} e tenho
${this.curso}.`);

    }

}

class Mestrando extends Graduando {

    especializacao: string;

    constructor(name: string, curso: string, especializacao:
string) {

        super(name, curso);

        this.especializacao = especializacao;

    }


    cumprimentar_dds(){

        super.cumprimentar_dds();

        console.log(`Minha especialização é
${this.especializacao}.`);

    }

}
```

2.7

```
class Veiculo {
  longitude: number;
  latitude: number;

  constructor(longitude: number, latitude: number) {
    this.longitude = longitude;
    this.latitude = latitude;
  }

  mover(mov_longi: number, mov_lat: number) {
    this.longitude += mov_longi;
    this.latitude += mov_lat;
    console.log(
      `O veículo se move nas cordenadas de latitude:
${this.latitude} e longitude ${this.longitude}.`
    );
  }
}

class Eletrico extends Veiculo {
  bateria: number;

  constructor(bateria: number, longitude: number, latitude: number)
{
    super(longitude, latitude);
    this.bateria = bateria;
```

```
  }
}


class Voador extends Eletrico {
  altitude: number;


  constructor(
    bateria: number,
    altitude: number,
    longitude: number,
    latitude: number
  ) {
    super(longitude, latitude, bateria);
    this.altitude = altitude;
  }
  voar(mov_alti: number, mov_longi: number, mov_lat: number) {
    this.altitude += mov_alti;
    this.mover(mov_longi, mov_lat);
    console.log(
      `O veículo voa nas coordenadas de latitude: ${this.latitude},
longitude ${this.longitude} e altitude ${this.altitude}.`
    );
  }
}
export{Veiculo, Eletrico, Voador}
```

Em outro arquivo TypeScript:

```
import { Veiculo, Eletrico, Voador } from "./exer_2.7_dds";


const golzinho_voador = new Voador(100, 20, 30, 20);

golzinho_voador.voar(1, 2, 3);
```

2.8

Quando uma função recursiva é chamada repetidamente e chama a si mesma um elevado número de vezes, isso pode levar ao overclock. Isso ocorre porque cada chamada à função coloca uma nova instância da função na pilha de chamadas, ocupando espaço na memória. Se o número de chamadas recursivas não for controlado adequadamente, a pilha de chamadas pode ficar sobrecarregada, levando a um overclock.

Código-fonte:

```
//POR RECURSÃO

function pa_recursiva(

  primeiro: number,

  diferenca: number,

  indice: number

): number {

  if (indice === 1) {

    return primeiro;

  } else {

    return pa_recursiva(primeiro, diferenca, indice - 1) +
diferenca;

  }

}

const primeiroTermo = 3;
const diferenca = 5;
const indiceTermo = 4;

const resultado = pa_recursiva(

  primeiroTermo,
```

```
  diferenca,
  indiceTermo
);
console.log(`O termo na posição ${indiceTermo} da progressão é:
${resultado}`);
```

```
//POR ITERAÇÃO


function pa_iterativa(a1: number, dif: number, n: number): number {
  let termo = a1;
  for (let i = 1; i < n; i++) {
      termo += dif;
  }
  return termo;
}


const termo1 = 3;
const dif = 5;
const ind = 4;


const resposta = pa_iterativa(primeiroTermo, diferenca,
indiceTermo);
console.log(`O termo na posição ${indiceTermo} da progressão é:
${resultado}`);
```

Conclusão:

A versão recursiva tende a ser mais simples de entender e escrever, pois reflete diretamente a definição matemática da PA. Mas, pode ser mais difícil de otimizar. Além disso, a versão iterativa é geralmente mais eficiente em termos de desempenho, especialmente quando se trata de sequências com um grande número de termos, porque a recursão pode criar várias chamadas à função na pilha de chamadas, logo consome mais recursos e corre risco de overclock. Também há o fato de que a versão iterativa pode ser mais fácil de manter e ajustar, especialmente quando é necessário fazer modificações no futuro. Portanto, em muitos casos a iteração é preferida devido à sua eficiência e menor risco.

2.9

```
function PG_dds(termoInicial: number, razao: number, n: number):
number {
    return termoInicial * Math.pow(razao, n - 1);
}


const n = 50;
const exemplo = PG_dds(1, 2, n)
console.log(`O termo na posição ${n} da progressão é: ${exemplo}`);
```

2.10

```
function fibonacciRecursivo_dds(n: number): number {
    if (n <= 0) {
        return 0;
    } else if (n === 1) {
        return 1;
    } else {
        return fibonacciRecursivo_dds(n - 1) +
fibonacciRecursivo_dds(n - 2);
    }
}


const indi = 6;
const result = fibonacciRecursivo_dds(indi);
console.log(`O termo na posição ${indi} da sequência de Fibonacci
é: ${result}`);
```