MAC0121 ALGORITMOS E ESTRUTURAS DE DADOS I

FOLHA DE SOLUÇÃO

Nome: Leonardo Heidi Almeida Murakami **NUSP:**11260186

Assinatura

Sua assinatura atesta a autenticidade e originalidade de seu trabalho e que você se compromete a seguir o código de ética da USP em suas atividades acadêmicas, incluindo esta atividade.

Exercício: Exercicio Teórico I - MAC0121 **Data:** 03/10/2024

SOLUÇÃO

1. Exercício 1

1.1. Explicação do funcionamento do algoritmo.

Podemos dividir o funcionamento do algoritmo da seguinte forma

$$\text{funcao}(n) = \begin{cases} n - 10 & \text{para } n > 100 \\ \text{funcao}(\text{funcao}(n + 11)) & \text{para } n \leq 100 \end{cases}$$

Para qualquer valor de n maior que 100, temos que a funcao retornara $n - 10$, para outros valores, a funcao crescerá o valor de n até atingir o valor de 101 e retornar como valor final 91.

1.2. Provando que o caso base é atingido para qualquer inteiro n.

Para provar que esta função sempre chegará a seu caso base, provaremos que, para qualquer valor n inicial, depois de um numero finito de chamadas recursivas, que este valor atingirá um valor superior a 100

(1) Primeiro devemos notar que, para qualquer $n > 100$, temos o caso base, dado que a função retorna um valor imediatamente
(2) Para $n \leq 100$, vamos verificar o que acontece em cada chamada recursiva:
   - Na chamada interna, n é acrescido de 11.
   - Se este valor for ≤ 100, será acrescido novamente de 11 na próxima chamada.
   - Este valor é decrescido, numa chamada, de no máximo 10
   - Logo, n sempre terá seu valor, previsivelmente crescente e será > 100 após uma quantidade finita de iterações
(3) Logo, sabemos que, existe um k tal que $n + 11k > 100$ para $n < 100$

1.3. **Calculando o numero de chamadas para** $funcao(n)$**.**

Podemos concluir, a partir do funcionamento da função, que a quantidade de chamadas para a $funcao$ pode ser calculada pela seguinte formula

$$\text{chamadas}(n) = \begin{cases} 1 & \text{para } n > 100 \\ \text{chamadas}(n+1) + 2 & \text{para } n \leq 100 \end{cases}$$

2. Exercício 2

2.1. Calculando f(1,6).

Vamos analisar o comportamento da função para funcao$(1, 6)$:

```
public static int funcao(int a, int b) {
   if (b == 0)
      return 0;
   else
      return (a + funcao(a, b-1));
}
```

A função é recursiva e tem o seguinte comportamento:

- Quando $b = 0$, a função retorna 0.
- Caso contrário, ela soma a ao valor retornado pela chamada recursiva funcao$(a, b - 1)$.

Agora, vamos simular a execução para funcao$(1, 6)$:

- funcao$(1, 6)$ chama funcao$(1, 5)$ e adiciona 1 ao resultado.
- funcao$(1, 5)$ chama funcao$(1, 4)$ e adiciona 1 ao resultado.
- funcao$(1, 4)$ chama funcao$(1, 3)$ e adiciona 1 ao resultado.
- funcao$(1, 3)$ chama funcao$(1, 2)$ e adiciona 1 ao resultado.
- funcao$(1, 2)$ chama funcao$(1, 1)$ e adiciona 1 ao resultado.
- funcao$(1, 1)$ chama funcao$(1, 0)$, que retorna 0 (caso base).

Agora, a função começa a retornar os valores somados:

- funcao$(1, 1)$ retorna $1 + 0 = 1$.
- funcao$(1, 2)$ retorna $1 + 1 = 2$.
- funcao$(1, 3)$ retorna $1 + 2 = 3$.
- funcao$(1, 4)$ retorna $1 + 3 = 4$.
- funcao$(1, 5)$ retorna $1 + 4 = 5$.
- funcao$(1, 6)$ retorna $1 + 5 = 6$.

Portanto, o resultado de funcao$(1, 6)$ é 6.

2.2. Provando que a função termina para qualquer a e b.

Vamos analisar o comportamento da função:

- O caso base ocorre quando $b = 0$, onde a função retorna 0 e não realiza mais chamadas recursivas.
- Em cada chamada recursiva, o valor de b é decrementado em 1 $(b - 1)$.

Assim, independentemente do valor de a, o valor de b sempre diminui até atingir o caso base $b = 0$. Como b diminui a cada iteração e é um valor inteiro, eventualmente b chegará a 0, o que faz com que a função termine para qualquer valor de b. Como atingir o caso base independe do valor de a, a função sempre termina.

3. Exercício 3

3.1. Código para calcular a sequencia de Fibonacci.

Utilizando a formula dada, temos que:

```
public class Main {
    public static int fibonacci(int n) {
        if (n == 1 || n == 2) {
            return 1;
        }
        if (n % 2 != 0) {
            int k = (n + 1) / 2;
            int fk1 = fibonacci(k);
            int fk2 = fibonacci(k - 1);
            return fk1 * fk1 + fk2 * fk2;
        } else {
            int k = n / 2;
            int fk1 = fibonacci(k + 1);
            int fk2 = fibonacci(k - 1);
            return fk1 * fk1 - fk2 * fk2;
        }
    }

    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Por favor, passe o valor de N como argumento.");
            return;
        }

        int N = Integer.parseInt(args[0]);
        for (int i = 1; i <= N; i++) {
            System.out.println("F(" + i + ") = " + fibonacci(i));
        }
    }
}
```

Este código nos permite calcular até o numero 46 da sequencia de Fibonacci (de valor 1836311903), isto ocorre devido ao overflow. Se alterarmos o código para usar o BigInteger do java de forma que o código pareça algo como:

```java
import java.math.BigInteger;

public class Main {
    public static BigInteger fibonacci(int n) {
        // Caso base
        if (n == 1 || n == 2) {
            return BigInteger.ONE;
        }
        if (n % 2 != 0) {
            int k = (n + 1) / 2;
            BigInteger fk1 = fibonacci(k);
            BigInteger fk2 = fibonacci(k - 1);
            return fk1.multiply(fk1).add(fk2.multiply(fk2));
        } else {
            int k = n / 2;
            BigInteger fk1 = fibonacci(k + 1);
            BigInteger fk2 = fibonacci(k - 1);
            return fk1.multiply(fk1).subtract(fk2.multiply(fk2));
        }
    }

    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Por favor, insira o valor de N como argumento.");
            return;
        }

        int N = Integer.parseInt(args[0]);
        for (int i = 1; i <= N; i++) {
            System.out.println("F(" + i + ") = " + fibonacci(i));
        }
    }
}
```

Com este código foi possível calcular o valor ate o numero 50000 da sequencia de Fibonacci, de valor aproximado de $1.84e + 4083$ (calculado usando outro programa)

3.2. **Prova que o de Djikstra é equivalentes a versão original.**

3.2.1. *Prova da Equivalência da Definições Alternativa de Fibonacci.*

Vamos provar por indução que as seguintes definições alternativas da função de Fibonacci são equivalentes à definição original:
Base da indução: Para $n = 0$ e $n = 1$, as definições são idênticas à original, então a equivalência é trivialmente verdadeira.
Hipótese indutiva: Assumimos que a equivalência é verdadeira para todos os valores até n, onde $n \geq 2$. Além disso, assumimos que as seguintes equações são verdadeiras:

$$\begin{aligned} F_{2n-1} &= F_n^2 + F_{n-1}^2 \\ F_{2n} &= F_n(F_{n+1} + F_{n-1}) \end{aligned}$$

Passo indutivo: Precisamos provar que a equivalência se mantém para $n + 1$. Temos dois casos a considerar:
Caso 1: $(2n + 1)$ (caso onde n é impar)
Provamos $F_{2n+1} = F_n^2 + F_{n+1}^2$:

$$\begin{aligned} F_{2n+1} &= F_{2n} + F_{2n-1} \\ &= F_n(F_{n+1} + F_{n-1}) + F_{n-1}^2 + F_n^2 \\ &= F_{n-1}(F_{n-1} + F_n) + F_n^2 + F_nF_{n+1} \\ &= F_{n+1}^2 + F_n^2. \end{aligned}$$

Caso 2: $(2n + 2)$ (caso onde n é par)
Provamos $F_{2n+2} = F_{n+1}(F_{n+2} + F_n)$:

$$\begin{aligned} F_{2n+2} &= F_{2n+1} + F_{2n} \\ &= (F_{n+1}^2 + F_n^2) + F_n(F_{n+1} + F_{n-1}) \\ &= F_{n+1}^2 + F_n^2 + F_nF_{n+1} + F_nF_{n-1} \\ &= F_{n+1}(F_{n+1} + F_n) + F_n(F_n + F_{n-1}) \\ &= F_{n+1}F_{n+2} + F_nF_{n+1} \\ &= F_{n+1}(F_{n+2} + F_n) \end{aligned}$$

Conclusão: Provamos que a definição de djikstra é equivalente à definição original para $n+1$, assumindo que são equivalentes para n. Pelo princípio da indução matemática, as definições são equivalentes para todos os números naturais.