Übungen Grundlagen der Informatik, Blatt 4

Prof. Dr. Sascha Hauke

Wintersemester 2023/24

1 Rekursion

1.1 GGT

Betrachten Sie die Lösung des ggT-Algorithmuses auf den Vorlesungsfolien.

- Formulieren Sie eine Lösung des ggT-Algorithmuses, die einen rekursiven Aufruf anstelle der expliziten Schleifenanweisung nutzt! Schreiben Sie diesen in Pseudocode auf (oder implementieren Sie in kurz in Python)!
- Geben Sie die rekursiven Aufrufe für eine Ausführung von ggT(15,10) an!

1.2 Fibonacci-Sequenz

Die Fibonacci-Sequenz sieht folgendermaßen aus:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, ...$$

Die ersten beiden Elemente sind fib(0) = 0 und fib(1) = 1. Danach entsprechen die Elemente der Summe ihrer jeweiligen 2 Vorgängerelemente.

- Formulieren Sie einen Algorithmus in Pseudocode zur rekursiven Berechnung von Fibonacci-Zahlen!
- Betrachten Sie den Baum der rekursiven Aufrufe der Funktion fib zur Berechnung der Fibonacci-Sequenz mit fib(4) (Abbildung 1). Zeichnen Sie den Baum für fib(5)!
- Wieviele Aufrufe von fib() sind nötig, um fib(4) zu berechnen? Wieviel für fib(5)?
- Schließen Sie auf das Wachstum der Funktion bzgl. der notwendigen Aufrufe von fib()! Verdeutlichen Sie dies, indem Sie aus dem Vergleich von fib(4) und fib(5) ableiten, wieviel Aufrufe von fib() für die Funktionsaufrufe fib(6) und fib(7) benötigt werden!
- Betrachten Sie den folgenden Code, der ebenfalls die Fibonacci-Sequenz berechnet:

```
algorithm fib_it(n)
Eingabe: Natürliche Zahl n >= 0
Ausgabe: Die Fibonacci-Zahl an der Position n
IF n == 0 THEN
        RETURN 0
ELSE IF n == 1 THEN
        RETURN 1
ELSE
```

```
fib_a = 0
fib_b = 1
i = 2
WHILE i <= n  DO
        aa = fib_b
        bb = fib_a + fib_b
        fib_a = aa
        fib_b = bb
        i = i + 1

RETURN fib_b
```

Welches Programm ist effizienter? Warum?

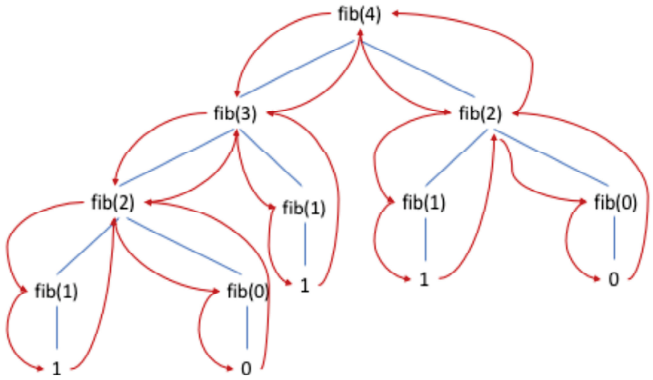

Abbildung 1: fib(4)

2 Suchen

2.1 Lineares Suchen

nehmen Sie an, sie haben eine Liste von positiven natürlichen Zahlen. Es gelte:

- Sie wissen nicht, in welcher Reihenfolge die Zahlen in der Liste stehen.
- Sie können die Liste Schritt für Schritt mit der Funktionn *nächsteZahl* abgehen: Es wird Ihnen dabei jeweils die nächste Zahl aus der Liste ausgegeben und diese aus der Liste entfernt. Es wird immer die Zahl am Anfang der Liste ausgegeben.
- Sie können mit *istLeer* überprüfen, ob die Liste leer ist (d.h., Sie am Ende der Liste angelangt sind).

Geben Sie einen Pseudocodealgorithmus an, der *wahr* zurückliefert, wenn eine bestimmte Zahl x in der Liste zu finden ist, *falsch* sonst.

Angenommen, es stünden n-viele Zahlen in der Liste:

- Wie oft müssen Sie *nächsteZahl* im besten und im schlechtesten Fall aufrufen, um zu ermitteln, ob eine bestimmte Zahl in der Liste steht?
- Wie viele Aufrufe benötigen Sie im Durchschnitt, wenn Sie davon ausgehen, daß sich die gesuchte Zahl irgendwo in der Liste befindet?

2.2 Binäres Suchen

Folgender Pseudocodealgorithmus stellt eine binäre Suche dar

```
algorithm binaersuche_rekursiv(werte, gesucht, start, ende)
Eingabe: Eine Sortierte Liste mit Werten, der gesuchte Werte, Start- und Endposition
Ausgabe: Der gesuchte Wert
    IF ende < start THEN
        RETURN 'nicht gefunden'
    mitte = (start + ende)//2
    IF werte[mitte] == gesucht THEN
        RETURN mitte
    ELSE IF werte[mitte] < gesucht THEN
        RETURN binaersuche_rekursiv(werte, gesucht, mitte+1, ende)
    ELSE
                        RETURN binaersuche_rekursiv(werte, gesucht, start, mitte-1)
```

Führen Sie die binäre Suche nach dem Element 32 auf folgender Liste aus:

$$[12, 17, 23, 24, 31, 32, 36, 37, 42, 47, 53, 55, 67, 76, 87, 89, 91, 91, 93]$$

Dokumentieren Sie die jeweiligen rekursiven Aufrufe und geben Sie die Teilliste an, auf der Sie operieren. Wieviele Aufrufe benötigen Sie, bis Sie das Element gefunden haben? Wieviele Schritte benötigen Sie, wenn Sie lineare Suche anwenden? Welche Rückschlüsse können Sie ziehen?