Training für Programmierwettbewerbe - Exam Questions

Hanno Barschel (174761)

1 Introduction

(1) Zeit Komplexität von drei Funktionen:

- **foo(.)** hat eine Zeitkomplexität von $\mathcal{O}(n)$ (wobei n die Länge des gegebenen Vektors ist) und eine Platzkomplexität von $\mathcal{O}(n)$.

  Das folgt direkt daraus, dass wir genau 2 Vorschleifen die jeweils den gesamten Vektor durchgehen haben also $\mathcal{O}(2n) = \mathcal{O}(n)$.

- **print_ pairs(.)** hat eine Zeitkomplexität von $\mathcal{O}(n^2)$ und eine Platzkomplexität von $\mathcal{O}(n)$.

  Für jedes Element des Vektors ruft die Funktion nochmal jedes Element des Vektors auf und printed beide als Paar aus. Somit kommt eine Zeitkomplexität von $\mathcal{O}(n^2)$ zustande.

- **print_ unordered_ pairs(.)** hat eine Laufzeitkomplexität von $\mathcal{O}(n^2)$ und eine Platzkomplexität von $\mathcal{O}(n)$.

  Für jedes Element des Vektors rufen wir alle Folgeelemente auf. Das sind es $n - (i + 1)$ Aufrufe für Element i (i startet bei 0). Damit kommen wir auf eine Laufzeit von

  $$\sum_{i=0}^{n-1}(n - (i + 1)) = n^2 - \sum_{i=1}^{n} i = n^2 - \frac{n \cdot (n + 1)}{2} = n^2 - \frac{n^2}{2} - \frac{1}{2} \in \mathcal{O}(n^2).$$

- **all_ fib(.)** hat eine Zeitkomplexität von $\mathcal{O}(n)$ und eine Platzkomplexität von $\mathcal{O}(n)$.

  Die for-Schleife printed uns im Prinzip alle Fibonacci Zahlen bis n aus. Sobald wir eine Null in $memo[i]$ finden sind aber bereits $memo[i-1]$ und $memo[i-2]$ berechnet, somit müssen wir diese nur addieren und haben keine wirklich tiefe Rekursion.

(2) Algorithmen Beispiele:

- $\mathcal{O}(1)$: Für eine gegebene Hashtabelle und einen Index gibt der Algorithmus den Wert für den Index zurück.
- $\mathcal{O}(\log(n))$: In einer sortierten Liste ein gegebenes Element suchen.
- $\mathcal{O}(n)$: Der Algorithmus berechnet zu einem gegeben Array die Summe aller Elemente und gibt diese zurück.
- $\mathcal{O}(n \cdot \log(n))$: Ein guter Sortier Algorithmus (Merge Sort).
- $\mathcal{O}(n^2)$: Quicksort (worst-case Analyse).

(3) Ersteinmal natürlich ordentlich zuhören wenn die Frage gestellt ist. Dabei sollten wir uns alles merken und auch auf Besonderheiten achten. Dann wählen wir ein geeignetes Beispiel. Dabei sollten wir immer im Hintergrund behalten, dass oft Spezialfälle besonders zu betrachten sind. Anschließend schreiben wir einen brute force Algorithmus auf und betrachten dessen Zeit sowie Platzkomplexität. Jetzt optimieren wir diesen noch indem wir das Ganze möglichst einfach machen, überflüssige und doppelte Arbeit vermeiden. Mit diesen Betrachtungen verbessern wir den Brutforce Algorithmus schließlich noch.

(4) Gegeben sei eine Liste mit Natürlichen Zahlen. Gesucht sei die maximale Differenz zweier Werte im Array wobei der vordere (steht weiter vorne im Array) Wert kleiner als der hintere sein muss. Wir iterieren mittels zweier indexe, tmp_min und tmp die am Anfang beide auf 0 gesetzt werden. Gleichzeitig haben wir die Variable $max\_diff = 0$ in der die bisher gefundene maximale Differenz gespeichert wird.

Nun beginnen wir mittels einer for-Schleife mit Variablen tmp über den Array zu iterieren: Ist der Wert im Array an Stelle tmp kleiner oder gleich als der an der Stelle tmp_min so setzen wir tmp_min auf tmp. Ist tmp größer so berechnen wir die Differenz zwischen tmp und tmp_min. Ist diese größer als max_diff so ersetzen wir diese, ansonsten inkrementieren wir tmp einfach weiter.

(5) Ich würde es mit einer Hashmap checken: Wir testen erst einmal ob die Strings die gleiche Länge haben (wenn nicht dann return 0). Danach erstellen wir 2 Hashmaps (für jedes Wort eine). Anschließend gehen wir mittels einer for-Schleife beide strings durch und für jeden Buchstaben inkrementieren wir den zugehörigen Hashwert. Abschließend vergleichen wir beide Hashmaps.

$return(hashmap\_string1 == hashmap\_string2)$.