TU Kaiserslautern – FB Informatik 10.12.2019
Dr. Patrick Michel

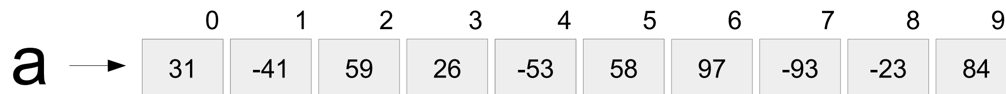
Algorithmen und Datenstrukturen (WS 2019)

Aufgabenblatt 6

zu bearbeiten bis: 16.12.19 / 18.12.19

Aufgabe 6.1 (Traveling Salesman - Theorie)
Wir betrachten die Greedy Version des Traveling Salesman Problems aus der Vorlesung. Am Beispiel haben wir gesehen, dass dieser Algorithmus schlechte Lösungen liefern kann. Deshalb wollen wir das Problem nun mit Hilfe von **Backtracking** lösen.
Skizzieren Sie in Pseudo-Code eine Routine `TSP`, welche die Route mit den geringsten Gesamtkosten mittels Backtracking berechnet. Nehmen Sie das allgemeine Schema für Backtracking Algorithmen und den Pseudocode der Greedy Variante als Ausgangspunkte.

Aufgabe 6.2 (Summen in Arrays - Praxis)
Gegeben sei ein Feld mit n (potenziell negativen) Zahlen. Suchen Sie eine zusammenhängende Teilfolge `a[start],a[start+1]...,a[end-1],a[end]`, so dass die Summe der Elemente dieser Teilfolge *maximal* ist.



In diesem Beispiel ist die Teilfolge `a[2],...,a[6]` die Lösung, mit Summe:

$$59 + 26 + (-53) + 58 + 97 = 187.$$

- Implementieren Sie einen naiven Algorithmus mit drei verschachtelten Schleifen: Eine für den Anfangsindex, eine für den Schlussindex und eine zum Summieren. Es werden alle möglichen Subbereiche (start,ende) explizit ausprobiert und der mit der maximalen Summe gewählt.

- Lösen Sie das Problem noch einmal mit Hilfe des "Divide & Conquer"-Ansatzes: Teilen Sie das Feld jeweils in zwei Hälften und lösen Sie das Problem rekursiv. Um die beste Lösung für einen Bereich zu finden, vergleichen Sie drei Lösungen: (1) die beste Lösung, die rekursiv in der linken Hälfte gefunden wurde, (2) die beste Lösung, die rekursiv in der rechten Hälfte gefunden wurde (3) die beste kombinierte Lösung die beide Bereiche überschneidet. Letzte erhalten Sie, indem Sie – von der Mitte des Bereichs aus – die maximale rechte Randsumme des linken Teils und die maximale linken Randsumme des rechten Teils finden und beide summieren.

Aufgabe 6.3 (Summen in Arrays - Theorie)

Wir vergleichen nun die Aufwände und resultierenden Komplexitätsklassen der beiden Implementierungen. Definieren Sie dazu jeweils zunächst die Aufwandsfunktion. Zählen Sie dabei die Arrayzugriffe in Abhängigkeit der Länge n des Arrays. Geben Sie dann die Komplexitätsklasse der Funktion an und begründen / argumentieren Sie.

- Bestimmen Sie die Komplexität Ihres **naiven** Algorithmus.
- Bestimmen Sie die Komplexität des **Divide & Conquer** Algorithmus.