Übungen Grundlagen der Informatik, Blatt 5

Prof. Dr. Sascha Hauke

Wintersemester 2023/24

1 Einfaches Sortieren

1.1 Selection Sort

Das Funktionsprinzip von Selection Sort ist ganz einfach. Man nehme eine Liste von (vergleichbaren) Elementen, teile diese in zwei Teile, einen sortierten (zu Anfang leer) und einen unsortierten, auf und gehe wie folgt vor:

- suche das kleinste Element der unsortierten Teilliste
- füge es hinten an die sortierte Teilliste an
- dadurch wird der unsortierte Teil in jedem Schritt ein Element kleiner, der sortierte ein Element größer
- wiederhole, bis unsortierter Teil leer

Der unten aufgeführte Algorithmus verdeutlicht die technische Umsetzung in Python (hier wird das Anfügen durch eine Vertauschung des kleinsten Elements mit dem an der gerade betrachteten Stelle realisiert).

```
# Input: Eine Liste mit vergleichbaren Elementen
# Output: Die Liste sortiert (als Effekt)
def selectionSort(alist):

    for fillslot in range(0,len(alist)-1):
        positionOfMin=fillslot

        for location in range(fillslot+1,len(alist)-1):
            if alist[location]<alist[positionOfMin]:
                positionOfMin = location

        temp = alist[fillslot]
        alist[fillslot] = alist[positionOfMin]
        alist[positionOfMin] = temp
```

- Gegeben sei die Liste [8,4,7,3,5,2]. Führen Sie Selection Sort auf dieser Liste aus und geben die Liste nach jedem Schleifendurchlauf der äußeren Schleife an!
- Sortieren Sie die Eingabe "Ein_Beispiel" mit Selection Sort. Geben Sie die Zwischenschritte nach jedem Schleifendurchlauf der äußeren Schleife an!

- Überlegen Sie: Angenommen, die Eingabeliste hätte die Länge n. Wie oft muss im schlimmsten Fall der Vergleich $alist[location] < alist[positionOfMin]$ durchgeführt werden? Hängt das von der Reihenfolge der Elemente in der Eingabeliste ab? Was können Sie über den Unterschied von Best- und Worst-Case sagen? Welche Komplexitätsklasse bietet sich für eine Worst-Case-Abschätzung bzgl. der notwendigen Vergleichsoperationen an?

1.2 Insertion Sort

Recherchieren Sie die Funktionsweise von Insertion Sort.

Der unten aufgeführte Algorithmus verdeutlicht die technische Umsetzung in Python.

```
# Input: Eine Liste mit vergleichbaren Elementen
# Output: Die Liste sortiert (als Effekt)
def insertionSort(alist):

        for index in range(1,len(alist)):

    currentvalue = alist[index]
    position = index

    while position>0 and alist[position-1]>currentvalue:
        alist[position]=alist[position-1]
        position = position-1

    alist[position]=currentvalue
```

- Gegeben sei die Liste [8,4,7,3,5,2]. Führen Sie Insertion Sort auf dieser Liste aus und geben die Liste nach jedem Schleifendurchlauf der äußeren Schleife an!
- Sortieren Sie die Eingabe "Ein_Beispiel" mit Insertion Sort. Geben Sie die Zwischenschritte nach jedem Schleifendurchlauf der äußeren Schleife an!
- Grob nach Komplexitätsklassen abgeschätzt, wieviele Durchläufe der inneren Schleife werden im besten und wieviele im schlimmsten Fall benötigt? Überlegen Sie, wie die Eingabeliste im besten und im schlimmsten Fall sortiert wäre!

2 Effizientes Sortieren

2.1 Quicksort

Recherchieren Sie die Funktionsweise von Quicksort.

Der unten aufgeführte Algorithmus verdeutlicht die technische Umsetzung in Python.

```
# Function to find the partition position
def partition(array, low, high):

    # choose the rightmost element as pivot
    pivot = array[high]

    # pointer for greater element
    i = low - 1
```

```
    # traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:

            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1

            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])

    # Swap the pivot element with the greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])

    # Return the position from where partition is done
    return i + 1

# function to perform quicksort


def quickSort(array, low, high):
    if low < high:

        # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)

        # Recursive call on the left of pivot
        quickSort(array, low, pi - 1)

        # Recursive call on the right of pivot
        quickSort(array, pi + 1, high)
```

Die gezeigte Implementierung nutzt das Verfahren in Cormen et al. *Introduction to Algorithms* – zu finden im Online Katalog der Hochschulbibliothek.

- Führen Sie Quicksort auf die Liste [7,4,1,9,8,2,3,5,6] aus und stellen Sie dies als Diagramm dar. Achten Sie darauf, wie die jeweiligen Teillisten der Elemente größer als das Pivotelement angeordnet sind (hier gibt es ein kleines Detail – Bonus, wenn Sie es entdecken).
- Sortieren Sie die Eingabe „Ein_Beispiel“ mit Quicksort.
- Wie würden Sie Quicksort modifizieren, damit in absteigender statt in aufsteigender Reihenfolge sortiert wird?