Grundzüge DS & Alg (WS14/15)

Lösungsvorschlag zu Aufgabenblatt 4

Aufgabe 1

- INSERTION-SORT: Ja, Es wird immer das erste, kleinste Element an die neue Liste angehängt.
- QUICK-SORT: Hängt davon ab ob PARTITION stabil ist.
- MERGE-SORT: Ja, Splitten, sowie Mergen ist stabil.
- HEAP-SORT: Nein. (Beachte HEAPIFY!).

Aufgabe 2

a)

Die Idee ist folgende: Wir speichern alle Elemente e mit e $\geq$ x in einem externen Speicher. Alle Elemente e $<$ x werden im Array komprimiert. Zum Schluss werden alle grösseren Elemente hinten in das Array kopiert.

Etwas formaler als Pseudo-Code:

```
1  Input: Array A[], linke Grenze i, rechte Grenze j, Pivotelement x
2  Output: Array A[], das wie in der Aufgabenstellung sortiert ist
3
4  readPointer = i;
5  writePointer = i; //alles links vom writePointer ist
6                                                  // kleiner als x
7  //EXT: externer Speicher
8
9  while (readPointer <= j)
10         if(A[readPointer] > x)
11                 EXT.addLast(A[readPointer]) //Element sichern
12
13         else
14                 A[writePointer] = A[readPointer]; //komprimieren
15                 writePointer++;
16
17         readPointer++;
18
19 p = writePointer;
20 // copy back
21 for(int i = 0; i < EXT.length; i++)
22         A[writePointer] = EXT[i];
```

```
23          writePointer++;
24
25  return p;
```

b)

Wir arbeiten rekursiv. Zunächst wird das Array genau in der Mitte m geteilt und rekursiv partitioniert mit dem Pivot x. Nun hat es die folgende Form: A[l ... p_1 ... m (m+1)... p_2 ... r]. l und r sind die linke und rechte Grenze fürs Partitionieren, p_1, p_2 sind die rekursiv errechneten Stellen der Pivotelemente und m ist die Mitte. Es gilt also $\forall l \leq i \leq p_1 : A[i] < x \wedge \forall p_1 < i \leq m : A[i] \geq x \wedge \forall m < i \leq p_2 : A[i] < x \wedge \forall p_2 < i \leq r : A[i] \geq x$.

Wir müssen nun nur noch die Teile von p_1 bis m und von m bis p_2, nennen wir sie B und C, vertauschen. Dafür rotieren wir sie zuerst selbst und bekommen B^R, C^R. Danach rotieren wir die Sequenz insgesamt und bekommen $(B^R, C^R)^R$, was dasselbe ist wie C B.

Dies ist stabil. Nehmen wir induktiv an, dass der rekursive Aufruf stabil ist. Dann ist auch das Zusammenfügen stabil.

Laufzeit: man kann ein Array mit n Elementen einfach in $O(n)$ rotieren. Also ist die Laufzeit: $f(n) = c \cdot n + 2f(n/2)$. und $f(1) = O(1)$, so $f(n) = O(n \log n)$.

Man kann kompliziertere Lösungen mit besserer Laufzeit in der Literatur[1] finden.

Aufgabe 3

(a)
Die Idee des Algorithmus SortLastPosition ist es, Listen von Elementen in X zu finden, für die die ersten k-1 Elemente x' des Tupels x gleich sind. Hat man eine Menge von Tupeln mit gleichem Anfang $(x_k, ..., x_2)$ gefunden, kann auf diese Menge ein CountingSort-Algorithmus ausgeführt werden. Dadurch werden die Teilmengen mit gleichem x' sortiert. Diese sortierten Teillisten müssen dann nur noch konkateniert werden um die vollständig sortierte Menge X zu erhalten.

SortLastPosition macht sich dabei zu nutze, dass das Array X bereits bezüglich x' vorsortiert ist. Es müssen nur direkt nebeneinander liegende Elemente verglichen werden. Sollten zwei Elemente $X[i]$ und $X[i+1]$ nicht den gleichen Anfang x' haben, wird mit $cPos$ (der Position, an der als nächstes ein Element einsortiert werden muss) überprüft ob direkt das kleinere Element in unser Resultat-Array B geschrieben werden kann. Ist $cPos$ kleiner als i, waren die Elemente $X'[cPos]$ bis $X'[i]$ gleich und wir sortieren mit Sort den Array $X[cPos...i]$ an die Position $B[cPos...i]$. $cPos$ kann nun auf die Position $i + 1$ aktualisiert werden.

[1] Jyrki Katajainen, Tomi Pasanen: Stable Minimum Space Partitioning in Linear Time (1992)

Grundzüge DS & Alg (WS14/15)

Lösungsvorschlag zu Aufgabenblatt 4


```
1: procedure SortLastPosition(X, b, k)
2:     n = size(X)
3:     cPos = 1
4:     B = new Array[1...n]
5:     for i = 1 to n − 1 do
6:         if not isEqual (X[i], X[i + 1], k) then
7:             if cPos == i then
8:                 B[cPos] = X[i]
9:                 cPos = i + 1
10:            else
11:                Sort (B[cPos...i], X[cPos...i], b, k)
12:                cPos = i + 1
13:    if cPos == n then
14:        B[cPos] = X[n]
15:    else
16:        Sort (B[cPos...n], X[cPos...n], b, k)
       return B
```

```
1: procedure isEqual(x, y, k)
2:     for i = 1 to k − 1 do
3:         if x_i ≠ y_i then return false
       return true
```

```
1: procedure Sort(B[a...e], X[a...e], b, k)
2:     C = new Array[0...b − 1]
3:     for i = 0 to b − 1 do
4:         C[i] = a
5:     for i = a to e do
6:         C[X[i]_k] = C[X[i]_k] + 1
7:     for i = 1 to b − 1 do
8:         C[i] = C[i] + C[i − 1]
9:     for i = e to a do
10:        B[C[X[i]_k]] = X[i]
11:        C[X[i]_k] = C[X[i]_k] − 1
```

Die Laufzeit des Algorithmus hängt davon ab, wie viele gleiche x' es in X gibt und an welcher Stelle sich die einzelnen x' unterscheiden. Im schlechtesten Fall werden in isEqual $k-1$

Vergleiche ausgeführt und dies für alle $n-1$ Paare in X. Der Sortieralgorithmus braucht für einen Array X mit Länge c $O(c+b)$ Zeit. Wenn wir uns mit c gleich großen Teilen von n annähern erhalten wir $O(n/c(c+b)) = O(nb)$. Insgesamt wird damit eine Laufzeit von $O((k-1)*(n-1)+O(nb) = O((k+b)n)$ erreicht.

(b)
Durch die Zusatzinformation können wir uns die Vergleiche mit isEqual sparen, indem wir einfach den Wert in E ablesen. Um herauszufinden, ob ein Element in die Liste mit gleichem x' gehört, wird nur noch konstante Zeit benötigt. Dadurch erreicht man eine Laufzeit von $O(nb)$.

```
1:  procedure SortLastPosition(X, b, k)
2:      n = size(X)
3:      cPos = 1
4:      B = new Array[1...n]
5:      for i = 1 to n − 1 do
6:          if E[i] == 0 then
7:              if cPos == i then
8:                  B[cPos] = X[i]
9:                  cPos = i + 1
10:             else
11:                 Sort (B[cPos...i], X[cPos...i], b, k)
12:                 cPos = i + 1
13:     if cPos == n then
14:         B[cPos] = X[n]
15:     else
16:         Sort (B[cPos...n], X[cPos...n], b, k)
        return B
```

Grundzüge DS & Alg (WS14/15)

Lösungsvorschlag zu Aufgabenblatt 4

Aufgabe 4

Gesucht: Aus einem Feld von n Büchern, bzw deren Gewicht wollen wir die k leichtesten Bücher finden, deren Summe ≤ 50 ist, k aber maximal.

Algorithm
  Data: $A[1 \dots n]$
  Result: k lightest books with $sum \leq 50$
  $k =$ `getLightest(`$A, 1, n, 50$`);`
  return $A[1 \dots k]$;
Function `getLightest(`A, i, j, sum`)`
  if $i > j$ **then**
    return j;
  end
  $k = \text{median}(A)$;
  $m = \text{partition}(A, i, j, k)$;
  for $l = i$ **to** $m - 1$ **do**
    $s = s + A[l]$;
  end
  if $s \leq sum$ **then**
    if $s + A[m] \leq sum$ **then**
      return `getLightest(`$A, m + 1, j, sum - s - A[m]$`));`
    else
      return $m - 1$;
    end
  else
    return `getLightest(`$A, i, m - 1, sum$`);`
  end

median(A), partition(A, i, j, k) und die for-Schleife benötigen $O(n)$ Zeit. Dann wird der Algorithmus rekursiv auf einer der beiden Hälften des Feldes aufgerufen.

Sei L die Anzahl an Iterationen:

$$T(n) = b \cdot n + T(n/2) = \sum_{l=0}^{L} b \cdot n \cdot (1/2)^l$$

$$\leq \sum_{l=0}^{\infty} b \cdot n \cdot (1/2)^l = b \cdot n \cdot 2$$

Folglich benötigt der Algorithmus $O(n)$ Zeit.