Abgabe: **29.05.2016** (bis 23:59 Uhr)

Aufgabe 6.1 (P) MergeSort

Sortieren Sie die Zahlenfolge $523, 126, 67, 1, 500, 34, 21, 229, 9, 123, 13$ mit MergeSort. Geben Sie für jede Rekursionsebene jeweils für das Aufspalten der Teilsequenzen und für das Verschmelzen der sortierten Teilsequenzen einen Zwischenschritt an (d.h. bei dieser Eingabesequenz insgesamt circa acht Zwischenschritte), sodass Ihr Vorgehen nachvollzogen werden kann.

Wie viele Rekursionsebenen gibt es im Allgemeinen bei MergeSort (wobei wir den initiale Aufruf von MergeSort nicht als eigene Rekursionsebene zählen)? In welcher Größenordnung liegt asymptotisch der Aufwand für jede Rekursionsebene?

Aufgabe 6.2 (P) Laufzeit Mergesort

In dieser Aufgabe machen wir eine Laufzeitanalyse von Mergesort. In der Vorlesung wurde die Laufzeit rekursiv formuliert und das Mastertheorem verwendet, um zu zeigen, dass die Laufzeit von Mergesort in $\mathcal{O}(n \log n)$ liegt.

Die rekursive Formulierung der Laufzeit lautet

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n) \\ T(1) &= \Theta(1) \end{aligned} \tag{1}$$

Zeigen Sie ohne Verwendung des Master-Theorems, dass die Worst-Case-Laufzeit von Mergesort angewandt auf Zahlenfolgen, deren Längen Zweierpotenzen sind, in $\mathcal{O}(n \log n)$ liegt.

(a) Verwenden Sie dazu die Methode des iterativen Einsetzens.

(b) Verwenden Sie dazu vollständige Induktion.

Aufgabe 6.3 (P) Die perfekte Tabelle

Konstruieren Sie eine statische perfekte Hashtabelle für die Elemente:

$$\begin{array}{cccccc} (16,10,11) & (8,2,15) & (7,12,8) & (1,10,3) & (13,11,14) & (6,11,14) \\ (7,3,16) & (2,2,8) & (10,5,15) & (7,3,14) & (2,10,1) & (14,11,6) \end{array}$$

Jedes Element x besteht aus den Stellen (x_0, x_1, x_2). Verwenden Sie jeweils passend eine der Hashfunktionen:

$$\begin{array}{c} \left(\sum_{i=0}^{2} 2^i x_i\right) \mod 17 \\ \left(\sum_{i=0}^{2} a_i x_i\right) \mod 7 \text{ mit } \mathbf{a} = (0,0,1) \text{ oder } \mathbf{a} = (6,6,2) \\ \left(\sum_{i=0}^{2} a_i x_i\right) \mod 3 \text{ mit } \mathbf{a} = (1,0,0) \text{ oder } \mathbf{a} = (0,2,2). \end{array}$$

Zur Erinnerung: Beim statischen perfekten Hashing wird in der ersten Stufe so lange eine Hashfunktion $h : \text{Key} \to \{0, \ldots, \lceil\sqrt{2}cn\rceil - 1\}$ aus einer c-universellen Familie $H_{\lceil\sqrt{2}cn\rceil}$ gezogen, bis für die gezogene Hashfunktion die Zahl $C(h)$ der Kollisionen maximal $\sqrt{2}n$ beträgt. Letzteres ist eine hinreichende Voraussetzung dafür, dass die zweite Stufe in erwartet linearer Laufzeit ablaufen kann. Die Laufzeit für die erste Stufe ist ebenfalls erwartet linear.

Alle Schlüssel, die durch h auf dieselbe Position abgebildet werden, gehören zum selben Bucket. Wenn b_ℓ Schlüssel zu Bucket B_ℓ mit $\ell \in \{0, \ldots, \lceil\sqrt{2}cn\rceil - 1\}$ gehören, dann hat das Bucket die Größe $m_\ell = cb_\ell(b_\ell - 1) + 1$.

In der zweiten Stufe wird für jedes Bucket B_ℓ so lange eine Hashfunktion h_ℓ einer c-universellen Familie H_{m_ℓ} gezogen, bis h_ℓ die Schlüssel von Bucket B_ℓ injektiv abbildet.

Bei dieser Aufgabe sind die Hashfunktionen bereits gegeben und müssen nicht erst gezogen werden.

Aufgabe 6.4 (P) Bad Hash

Wir betrachten ein Negativbeispiel für eine Hashfunktion h, die auf einem String $s = s_1 \ldots s_n$ der Länge n bestehend aus ASCII-Zeichen arbeitet:

$$h(s) := \sum_{i=1}^{n} \text{Anzahl der Einsen in der Binärdarstellung von } s_i.$$

Nehmen Sie an, dass jedes der 256 Zeichen in dem Text gleichwahrscheinlich vorkommt.

Begründen Sie, warum Sie die Hashfunktion nicht für geeignet halten.

Aufgabe 6.5 [5 Punkte] **(H) Slowsort**

Der Slowsort-Algorithmus ist ein besonders ineffizientes Sortierverfahren mit nicht-polynomieller Laufzeit[1], welches dem Prinzip *vervielfache und kapituliere* folgt. Der Algorithmus führt dabei folgende Schritte aus:

1. Bestimmung des Maximums in der linken Hälfte des Feldes, indem der Slowsort-Algorithmus rekursiv aufgerufen wird
2. Bestimmung das Maximums in der rechten Hälfte des Feldes, indem der Slowsort-Algorithmus rekursiv aufgerufen wird
3. Wahl des größeren der beiden Maxima, welches auch das Maximum des gesamten Feldes ist, Tausch dieses Elements an den höchsten Index des Feldes
4. Rekursive Sortierung des Restfeldes mit Slowsort

In Java lässt sich der Algorithmus wie folgt implementieren:

```
1  void slowsort(int[] numbers, int from, int to) {
2    if(from >= to) // Rekursionsende bei leerem Teilfeld
3      return
4    int m = from + (to - from)/2;
5    slowsort(numbers, from, m); // Schritt 1
6    slowsort(numbers, m + 1, to); // Schritt 2
7    if(numbers[m] > numbers[to]) // Schritt 3
8      swap(numbers, m, to); // Schritt 3
9    slowsort(numbers, from, to - 1); // Schritt 4
10 }
```

Lösen Sie folgende Teilaufgaben:

a) [2.5] Stellen Sie eine Rekursionsgleichung für die Laufzeit von *Slowsort* auf. Überlegen Sie sich eine obere Schranke für die geschlossene Darstellung der Laufzeit des Algorithmus, indem Sie die Ungleichung $n - 1 \geq \frac{n}{2}$ (für $n \geq 2$) ausnutzen. Beweisen Sie die Richtigkeit Ihrer oberen Schranke durch vollständige Induktion.

b) [2] Zeigen Sie die Korrektheit von *Slowsort* durch vollständige Induktion über die Anzahl der Elemente im zu sortierenden Feld.

c) [0.5] Ist *Slowsort* ein stabiles Sortierverfahren? Beweisen Sie Ihre Antwort.

Aufgabe 6.6 [5 Punkte] **(H) Dirty Double Hashing (without Ponys)**

Für diese Aufgabe verwenden wir ein modifiziertes Double Hashing, welches beim Löschen von Elementen die abhängigen Kollisionen nicht neu hasht, sondern einfach einen "gelöscht"-Platzhalter einfügt (unten mit 'X' zu markieren).

Die Größe der Hashtabelle ist $m = 13$. Die Schlüssel der Elemente sind die Elemente selbst. Verwenden Sie die folgenden Hashfunktionen:

$$h(x, i) = (h(x) + i * h'(x)) \mod 13$$
$$h(x) = 3x \mod 13$$
$$h'(x) = 1 + x \mod 12$$

[1] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.116.9158

a) [0.5] Können bei dieser Vorgehensweise die Platzhalter beim Einfügen überschrieben werden? Begründen Sie Ihre Antwort.

b) [0.5] Unter welchen zwei Umständen kann die Find-Operation ergebnislos abbrechen?

c) [0.5] Wir fügen $n > 0$ Elemente in eine anfangs leere Hashtabelle ausreichender Größe ein und löschen diese wieder. Was ist die Worst-Case Laufzeit einer folgenden Find-Operation? Begründen Sie Ihre Antwort.

d) [3.5] Führen Sie folgende Operationen in der gegebenen Reihenfolge aus (tragen Sie auch alle überprüften Positionen ein):

Insert:	$1, 14, 2, 7$
Delete:	$1, 7$
Insert:	$7, 1$

Es soll keine dynamische Größenanpassung der Hashtabelle stattfinden.

Hinweis: Für natürliche Zahlen x gilt $3x \bmod 13 = (3(x \bmod 13)) \bmod 13$.

1. Operation: Insert(1):

0	1	2	3	4	5	6	7	8	9	10	11	12

2. Operation: Insert(14):

0	1	2	3	4	5	6	7	8	9	10	11	12

3. Operation: Insert(2):

0	1	2	3	4	5	6	7	8	9	10	11	12

4. Operation: Insert(7):

0	1	2	3	4	5	6	7	8	9	10	11	12

5. Operation: Delete(1):

0	1	2	3	4	5	6	7	8	9	10	11	12

6. Operation: Delete(7):

0	1	2	3	4	5	6	7	8	9	10	11	12

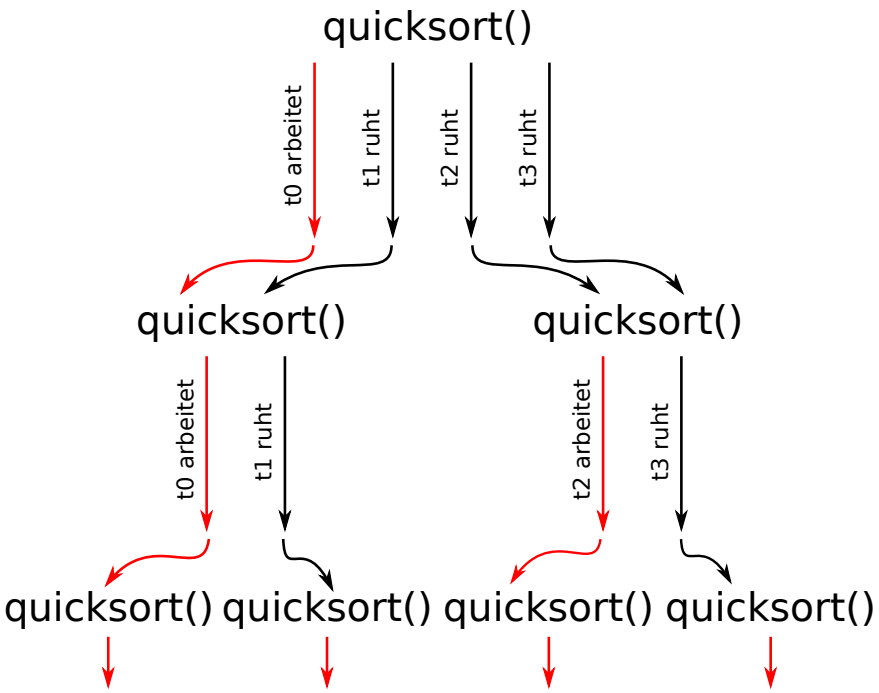

Abbildung 1: Parallelität auf Rekursionsebenen

7. Operation: Insert(7):

0	1	2	3	4	5	6	7	8	9	10	11	12

8. Operation: Insert(1):

0	1	2	3	4	5	6	7	8	9	10	11	12

Aufgabe 6.7 [10 Punkte] **(H) Programmieraufgabe: Pasquicklina**

Der Quicksort Algorithmus ist ein Sortierverfahren, welches – ähnlich zu MergeSort – durch eine *Divide and Conquer*-Strategie eine sehr gute Laufzeit aufweist. In dieser Aufgabe geht darum, eine parallele Version von Quicksort zu implementieren. Gehen Sie dabei wie folgend beschrieben vor.

Sie finden eine Vorlage für Ihre Implementierung im Ordner `pasquicklina-angabe/`. Sie dürfen während der Bearbeitung die gegebenen Klassen beliebig erweitern und neue Klassen hinzufügen, müssen aber das in der Angabe gegebene Interface nach außen wahren. Während der gesamten Aufgabe dürfen Sie davon ausgehen, dass die Anzahl an Threads eine Potenz von 2 ist. Als Pivotelement wählen wir jeweils dasjenige Element, welches den kleinsten Index (im folgenden auch das *linkeste* Element genannt) im zu sortierenden Feldbereich hat.

a) Implementieren Sie die Methode `void quicksort(int[] numbers)` der Klasse `Pasquicklina` zunächst ohne Threads, also auch ohne paralleles Sortieren. Achten Sie bei Ihrer Implementierung darauf, beim Einsortieren der Elemente auf je einer Seite des Pivot-Elements das effiziente Verfahren zu nutzen, welches in der Vorlesung beschrieben

wurde. Sie sollten den Algorithmus vollständig verstanden haben, bevor Sie damit beginnen, ihn zu parallelisieren.

b) Erweitern Sie Ihre Implementierung in der Methode `void quicksort(int[] numbers, int threads)` nun um eine erste Form von Parallelität. In diesem ersten Schritt sollen lediglich die verschiedenen rekursiven Aufrufe von unterschiedlichen Threads bearbeitet werden. Auf der obersten Rekursionsebene trägt also nur ein Thread bei; beim rekursiven Abstieg teilt sich die Menge der Threads in zwei Hälften, wobei jeweils eine Hälfte dem entsprechenden rekursiven Abstieg folgt (vgl. Abbildung 1). Um das weitere Vorgehen vorzubereiten, sollten Sie – wie in der Abbildung dargestellt – sämtliche Threads vor dem ersten Aufruf starten und für die Rekursion aufteilen.

c) Die obige Implementierung weist noch große Einschränkungen bezüglich der Parallelität auf – insbesondere arbeitet auf der obersten Rekursionsebene lediglich ein einziger Thread, dies macht alleine bereits einen linearen Anteil der Laufzeit aus. Hier soll nun nachgebessert werden, sodass die Threads die Aufteilung bezüglich des Pivotelements p möglichst gemeinsam berechnen. Dazu wird das Feld bei k Threads in k *Schienen* aufgeteilt, wobei die i-te Schiene alle Elemente mit Index x des Feldbereiches enthält, für die $(x-1)\%k = i$ gilt (vgl. Abbildung 2). Jedem Thread wird nun eine Schiene zugewiesen, die er auf die bekannte Weise bezüglich des Pivotelements sortiert. Haben alle Threads ihre jeweiligen Schienen am Pivotelement ausgerichtet, so wissen wir, dass das Pivotelement zwischen Index $l = min(\{left(i) \ : \ i \in \text{Schienen}\})$ und $r = max(\{right(i) \ : \ i \in \text{Schienen}\})$ einsortiert werden muss und alle Indices $x \leq l$ bzw. $x \geq r$ bereits am Pivotelement ausgerichtet sind[2]. Es muss also noch ein einzelner Thread den den Bereich $]l; r[$ am Pivotelement auf die übliche Art und Weise ausrichten, bevor rekursiv abgestiegen werden kann.

d) **(Zusatzaufgabe ohne Bewertung und Musterlösung)** Es bleibt schließlich bei obiger Herangehensweise noch der Bereich $]l; r[$ übrig, der von nur einem einzigen Thread bezüglich des Pivotelements sortiert werden muss. Hier gibt es weitere Parallelisierungsmöglichkeiten:

- Innerhalb des Bereiches $]l; r[$ sind die Elemente schon auf ihren jeweiligen Schienen ausgerichtet. Ein Thread kann nun jeweils zwei Schienen wählen und durch Vertauschen von Elemente ihre Längen anpassen. Dieses Vorgehen wird iterativ wiederholt, bis alle Schienen eine ähnliche Länge haben und der Aufwand für die Synchronisation der Threads den Zeitgewinn überschreitet.
- Die Anzahl der Threads kann halbiert werden; dadurch ergeben sich für die restlichen Threads kürzere Schienen. Der Bereich $]l; r[$ kann so sukzessive verkleinert werden.

Weitere Ideen sind sehr willkommen!

Beachten Sie folgende **Hinweise**:

- Die Angabe enthält die Klasse `Barrier`, die eine Barriere für Threads implementiert. Eine Barriere wird mit einer bestimmten Threadzahl initialisiert, die im Konstruktor übergeben werden muss. Mit der Barrriere lässt sich ein Synchronisationspunkt implementieren, den die Threads nur gemeinsam überwinden können. Dazu rufen die Threads die Methode `barrierWait()` auf; sie werden daraufhin pausiert, bis die konfigurierte Anzahl an Threads die Barriere erreicht haben. Die gegebene Barriere ist mehrfach benutzbar.

[2] $left(i)$ bzw. $right(i)$ bezeichnen den jeweils letzten ausgerichteten Index. $left(i)$ ist also der *rechteste* Index x auf Schiene i, bei dem `numbers[x]` $\leq p$ gilt.

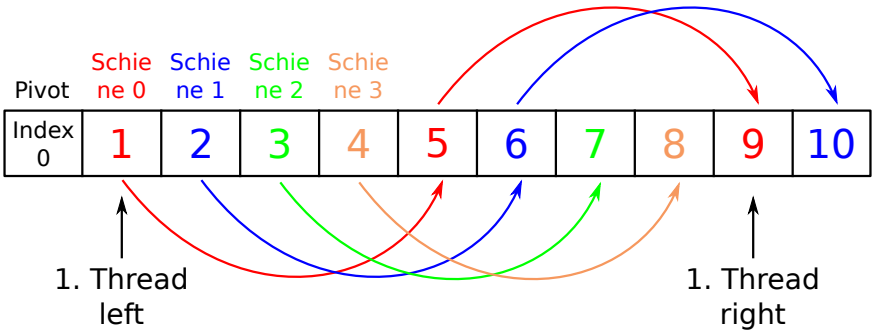

Abbildung 2: Schienen

- Synchronisation zwischen Threads kostet viel Zeit. Achten Sie daher in Ihrer Implementierung darauf, Synchronisation so sparsam wie möglich einzusetzen!