

Algorithmen und Datenstrukturen

Aufgabe 2

1

1

Aufgabenstellung

In dieser Aufgabe werden Algorithmen zum Sortieren implementiert. Dabei sind die Algorithmen/Strategien zu verwenden, die in der Vorlesung vorgestellt wurden! Die wesentliche Aufgabe besteht darin, das Kernkonzept der Algorithmen zu extrahieren. Die Korrektheit des Transfers

"Konkretes, technisches Konzept **im Beispiel** (oft Java) <-->

Kernkonzept des Algorithmus **im Entwurf** <-->

Konkrete, technische Realisierung/Implementierung **in Erlang**"

ist nachzuweisen.

Sofern der in der Vorlesung vorgestellte Algorithmus mittels einem Speicherplatzkonzept, z.B. array, technisch realisiert wurde, ist dies so zu transformieren bzw. zu verallgemeinern, dass das Kernkonzept extrahiert wird! Die Begründung (Nachweis der Korrektheit) dazu ist im Entwurf aufzuführen. Eine Simulation des Speicherplatzkonzeptes array mittels Listen ist nicht zulässig, da bei allen hier zu realisierenden Algorithmen ein array nicht zum Kernkonzept gehört!

Technische Vorgabe: die Zahlen sind in der Erlang-Liste [] gehalten und zu sortieren.

2

2

Aufgabenstellung

Implementieren Sie bitte:

- **Insertion Sort:** Die Schnittstelle: `insertionS:insertionS(<Liste>)`.
- **Radixsort:** Die Schnittstelle: `radixS:radixS(<Liste>,<Digit>)`. `<Digit>` gibt dabei die maximal vorkommende Stelligkeit der Zahlen in an.
- **Introsort:** Die Schnittstelle: `introS:introS(<pivot-methode>,<Liste>,<switch-num>)`
`<pivot-methode>`-Werte sind: left/middle/right/median/random.
 Technische Vorgabe: Wechsel zu Insertion Sort ist flexibel über die `<switch-num>` zu gestalten. Ein Wechsel zu Heapsort findet statt, wenn die Rekursionstiefe $2 \cdot \log_2(n)$ überschreitet.
- **Heap Sort:** Die Schnittstelle `heapS:heapS(<Liste>)`. (Für das Kernkonzept verwenden Sie bitte die in der VL vorgestellte Animation)! Da das Verfahren mit einem Max-Heap arbeitet, ist dieser als eigene ADT zu realisieren: `heap.erl`. Die Schnittstellen sind
`heap:create/0` erstellt einen leeren Heap,
`heap:isEmpty/1` gibt an, ob der Heap leer ist, Return ist ein Boolean.
`heap:insert/2, heap:insert(<Heap>,<Element>)` fügt ein Element in den Heap ein und gibt den modifizierten Heap zurück.
`heap:pop/1` löscht das Wurzelement des Heap und gibt den modifizierten Heap zurück.
`heap:top/1` gibt das Wurzelement des Heap zurück.

Achtung: der Heap ist stets kompakt zu halten, d.h. die unterste Ebene ist von links nach rechts besetzt, alle davor liegenden Ebenen sind komplett besetzt. Dies ist beim Einfügen und löschen eines Elementes zu beachten.

3

3

Aufgabenstellung

- eine **Testumgebung**, die die Laufzeit misst. Unterschiedliche Einstellungen, z.B. Anzahl der Zahlen, Strategie bei Quicksort etc, sollen möglich sein und gegeneinander getestet werden. Sie können auch stattdessen die Programme verwenden: `zeitinsertionS` (für Insertion Sort), `zeitradixS` (für Radixsort), `zeitintroS` (für Introsort) und `zeitheapS` (für Heap Sort) (Aufruf jeweils mit `zeit*S:messung()`). Die Zeitmessung / Laufzeittests sind (mit Zufallszahlen, aufsteigend bzw. absteigend sortierten Zahlen) so durchzuführen, dass sie aussagekräftig sind (Einheit: ms!).
- Bei Introsort ist die beste **"Wechselgrösse"** durch Tests zu ermitteln. Dazu können Sie das Programm verwenden: `zeitInSwitch` (Aufruf mit `zeitInSwitch:messung()`).

Beachten Sie: Zu Lernzwecken sind alle benötigten Hilfsfunktionen selbst zu implementieren, insbesondere `[..|..]` oder Indexzugriffe auf Tupel oder Listen bzw. das Modul `lists`: und andere ist nicht zulässig. Zugriff auf Listen und Tupel ist nur mit Pattern (z.B. `[Head|Tail]`) zulässig. Fragen Sie bitte im Zweifel unbedingt nach! Algorithmen auf den Basisstrukturen müssen Sie also bitte selbst implementieren. Die Verwendung anderer, komplexerer Datenstrukturen wie etwa `dict` oder `sets` sind daher auch nicht zulässig.

Hier die verwendeten links/url's:

Sortieralgorithmen: <https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>

Heapsort: <https://visualgo.net/en/heap> (Einstellung: **create(A) - (N log N)!!**)

4

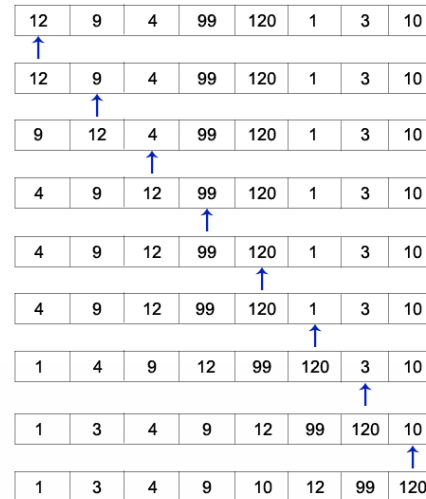
4

Insertion Sort

```

InsertionSort() {
  for ( int i = 2; i ≤ N ; i++ ) {
    int j = i;
    Datensatz t = a[i];
    int k = t.key;
    while(a[j-1].key > k) {
      a[j] = a[j-1];
      j = j-1;
    }
    a[j] = t;
  }
}

```



5

5

Radixsort (Bucketsort)

Nutzt **Zahldarstellung** der Schlüssel anstelle von Vergleichen

Idee: zuerst nach dem letzten Zeichen sortieren

Schlüssel k = Wort über Alphabet mit m Elementen

Beispiel:

$F = 434, 528, 154, 176, 783, 204, 351, 218, 900$

$$k = (k_l, k_{l-1}, \dots, k_1, k_0)_m = \sum_{i=0}^l k_i m^i$$

$$k_i \in \{0, \dots, m-1\}$$

Verteilen nach Ziffer 3:

900	351		783	154		176		218	
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9

Sammeln: 900, 351, 783, 434, 154, 204, 176, 528, 218

Verteilen nach Ziffer 2:

204	218	528	434		154		176	783	
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9

Sammeln: 900, 204, 218, 528, 434, 351, 154, 176, 783

Verteilen nach Ziffer 1:

	176	218							
	154	204	351	434	528		783		900
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9

Sammeln: 154, 176, 204, 218, 351, 434, 528, 783, 900

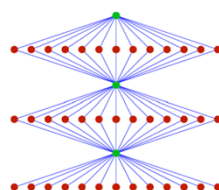
$m = 10$ Dezimalzahlen

$m = 2$ Binärzahlen

$m = 26$ Zeichenketten über $\{a, \dots, z\}$

Funktion: $z_m(k, i) = k_i$

Hier: Sortieren durch
Fachverteilung



6

6

Radixsort (Bucketsort)

Wiederhole für jede Stelle der Schlüssel, wobei mit der letzten Stelle begonnen (LSD (engl. least significant digit)) und mit der ersten Stelle geendet wird:

1. Partitionierungsphase

Teile die Daten in die vorhandenen Fächer auf, wobei für jedes Zeichen des zugrundeliegenden Alphabets ein Fach zur Verfügung steht. In welches Fach der gerade betrachtete Schlüssel gelegt wird, wird durch das an der gerade betrachteten Stelle stehende Zeichen bestimmt.

2. Sammelphase

Lege die Daten wieder auf einen Stapel, wobei zuerst alle Daten aus dem Fach mit der niedrigsten Wertigkeit eingesammelt werden, und die Reihenfolge der darin befindlichen Elemente nicht verändert werden darf.

7

7

Radixsort (Bucketsort)

```
1 def sort(array, base=10):
2     if not array: # array is empty
3         return
4     iteration = 0
5     max_val = max(array) # identify largest element
6     while base ** iteration <= max_val:
7         buckets = [[] for num in range(base)]
8         for elem in array:
9             digit = (elem // (base ** iteration)) % base
10            buckets[digit].append(elem)
11        pos = 0
12        for bucket in buckets:
13            for elem in bucket:
14                array[pos] = elem
15                pos += 1
16        iteration += 1
```

8

8

Heap Sort

Sortieren mit Hilfe eines Max-Heaps:

- ♦ **Phase 1:** Einfügen der zu sortierenden Liste in einen anfangs leeren Max-Heap (**reHeap_up**)
- ♦ **Phase 2:** Iteriertes Extrahieren des Maximums (**reHeap_down**):
Entnahme des aktuellen Maximums aus dem Max-Heap.
- ♦ Phase 1 kann in Zeit $O(n)$ ausgeführt werden.
- ♦ Phase 2 wird n -Mal ausgeführt, jede Ausführung kostet höchstens $O(\log n)$ Schlüsselvergleiche.
- ♦ Gesamtlaufzeit von Heapsort: $O(n \log n)$
- ♦ **Heapeigenschaft:** Ein Baum von Schlüsseln ist ein **Max-Heap**, wenn für alle Teilbäume gilt: die Wurzel ist größer oder gleich dem linken und rechten Nachfolger.

9

9

ADT Heap

Objektmenge: heap, boolean, number

Operationen:

- ♦ **create/0:** $\rightarrow \text{heap}$
- ♦ **isEmpty/1:** $\text{heap} \rightarrow \text{boolean}$
- ♦ **insert/2:** $\text{heap} \times \text{number} \rightarrow \text{heap}$
- ♦ **top/1:** $\text{heap} \rightarrow \text{number}$
- ♦ **pop/1:** $\text{heap} \rightarrow \text{heap}$

10

10

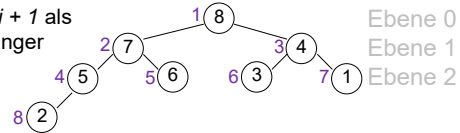
ADT Heap

Ein Heap wird durch das Tupel $\{<tree>, <pos>\}$ dargestellt, wobei

$<tree>$ der Baum des Heap ist.

$<pos>$ gibt die erste freie Position im Baum des Heap an, startet mit 1.

- **create/0**: erstellt einen initialen Heap
- **isEmpty/1**: prüft, ob der Heap leer ist
- **insert/2**: fügt ein Element in den Heap ein, wobei die Wurzel eines Baums stets größer ist, als alle anderen Elemente in den unteren Teilbäumen. Der Baum ist stets kompakt zu halten.
- **top/1**: liefert das größte Element des Heaps, also die Wurzel.
- **pop/1**: löscht das größte Element des Heaps und behält die kompakte Struktur des Heaps sowie dessen Sortierung bei.
- Um den Heap kompakt zu halten, ist die Kodierung der Stellen im Baum wie folgt vorzunehmen:
 - Ebene i hat 2^i Knoten (evtl. außer der untersten Ebene)
 - Knoten sind von oben nach unten und von links nach rechts nummeriert.
 - Knoten i hat Knoten $2i$ als linken und Knoten $2i + 1$ als rechten Nachfolger und Knoten $\lfloor i/2 \rfloor$ als Vorgänger (außer bei der Wurzel).



11

11

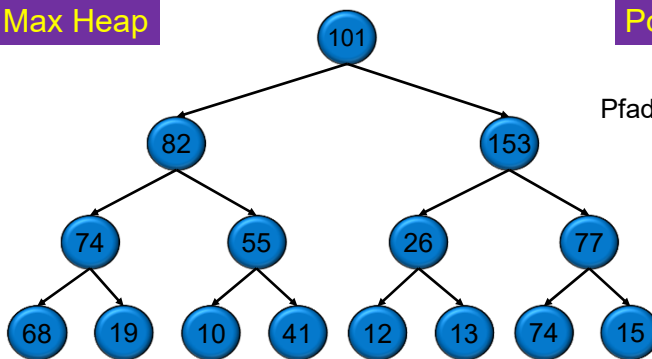
ADT Heap

Max Heap

Position

16

Pfad: [1,1,1,1]



Einzufügen: 75

12

12

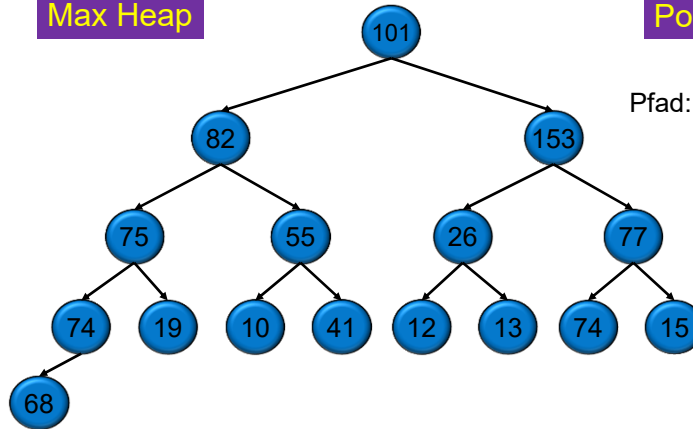
ADT Heap

Max Heap

Position

17

Pfad: [l,l,l,r]



13

13

Introsort

```

// Funktion zum Durchführen von Introsort für das angegebene Array
void introsort(int a[], int *begin, int *end, int maxdepth)
{
    // Einfügesortierung durchführen, wenn die Partitionsgröße 16 oder kleiner ist
    if ((end - begin) < 16) {
        insertionsort(a, begin - a, end - a);
    }
    // Heapsort ausführen, wenn die maximale Tiefe 0 ist
    else if (maxdepth == 0) {
        heapsort(begin, end + 1);
    }
    else {
        // andernfalls Quicksort durchführen
        int pivot = randPartition(a, begin - a, end - a);
        introsort(a, begin, a + pivot - 1, maxdepth - 1);
        introsort(a, a + pivot + 1, end, maxdepth - 1);
    }
}

```

Hier: Wechsel zu **Insertion Sort** ist flexibel zu gestalten und als Teilaufgabe zu untersuchen, was die beste Größe ist. Ein Wechsel zu **Heapsort** findet statt, wenn die Rekursionstiefe (maxdepth) $2 \cdot \log_2(n)$ überschreitet.

14

14

Quicksort

```

quicksort(int ilinks, int irechts) {
    if (ilinks < irechts) {
        int i = quickSwap(ilinks,irechts);
        quickSort(ilinks,i-1);
        quickSort(i+1,irechts); }
    }

    int quickSwap(int ilinks, int irechts) {
        int i = ilinks;
        int j = irechts-1;
        int pivot = a[irechts].key;

        while(i <= j) {
            while((a[i].key ≤ pivot) && (i < irechts)) i++;
            // a[i].key > pivot
            while((ilinks ≤ j) && (a[j].key > pivot)) j--;
            // a[j].key ≤ pivot
            if ( i < j ) swap(i,j); }

        swap(i,irechts); //Pivotelement in die Mitte tauschen
        return i;}

```

15

15

Quicksort

- Die Wahl eines passenden **Pivot-Wertes** p (Vergleichswerts) für die Partitionierung des Feldes a ist offensichtlich **von entscheidender Bedeutung** für die Effizienz von Quicksort.
- Optimal** wäre ein Element, das a in **zwei annähernd gleich große Teile** partitioniert: Dies müsste ein, dem Wert nach, mittleres Element von a sein. Um ein solches Element zu bestimmen, wäre jedoch ein Aufwand erforderlich, der die Laufzeitvorteile von Quicksort wieder zunichte machen würde.
- In den bekannten Implementierungen beschränkt man sich daher auf die Inspektion von drei naheliegenden Elementen von a (**Median of 3**):
 - $a[li]$ das linke Element von a ;
 - $a[mid]$ das mittlere Element von a mit $mid = (li+re)/2$;
 - $a[re]$ das rechte Element von a ;

Man kann entweder eines dieser drei Elemente nehmen und die anderen unberücksichtigt lassen – oder aber unter den drei genannten Werten dasjenige auswählen, welches dem Wert nach in der Mitte liegt.
- Eine weitere Möglichkeit wäre eine **randomisierte** (zufällige) **Wahl** des Pivot-Elementes: randomisiertes Quicksort (Paradigma Zufallsstichproben)

16

16

Durchzuführende Tests

```

6> testSORT:testSORT(420000,'bsp.log').
Achtung, die Zeitmessungen/Tests werden nebenläufig gesteuert.
Ende des Tests ist durch ...*S.done.. zu erkennen.
Test beginnt...
<0.174.0>
...pong-3-left...
...pong-3-random...
...pong-3-middle...
...pong-3-right...
...pong-3-median...
...ping-3-middle...
...ping-3-random...
...ping-3-median...
...pong-1...
...InS.middle.done.
Introsort middle Zufallszahlen Wechsel bei 16,
2| Diff: 744 ms Sortiert: true
Introsort middle aufsteigend 420000 Zahlen Start
rtiert: true
Introsort middle absteigend 420000 Zahlen Start
rtiert: true
...InS.random.done.
Introsort random Zufallszahlen Wechsel bei 16,
5| Diff: 687 ms Sortiert: true
Introsort random aufsteigend 420000 Zahlen Start
rtiert: true
Introsort random absteigend 420000 Zahlen Start
rtiert: true

```

bsp.log
 heapS.beam
 heapS.erl
 insertionS.beam

```

Introsort middle [...] Diff: 449 ms Sortiert: true
Introsort random [...] Diff: 687 ms Sortiert: true
Introsort random [...] Diff: 587 ms Sortiert: true
Introsort random [...] Diff: 644 ms Sortiert: true
Introsort median [...] Diff: 817 ms Sortiert: true
Introsort median [...] Diff: 523 ms Sortiert: true
Introsort median [...] Diff: 690 ms Sortiert: true
Radix Sort Zufallszahlen 420000 Zahlen
Startzeit: 08.07 14:06:57,588|
Endezeit: 08.07 14:06:59,139|
Diff: 1551 ms
Sortiert: true
Radix Sort aufsteigend 420000 Zahlen
Startzeit: 08.07 14:06:59,145|
Endezeit: 08.07 14:07:00,356|
Diff: 1212 ms
Sortiert: true
Radix Sort absteigend 420000 Zahlen
Startzeit: 08.07 14:07:00,361|
Endezeit: 08.07 14:07:01,667|
Diff: 1306 ms
Sortiert: true

```

17


17


Laufzeitmessung

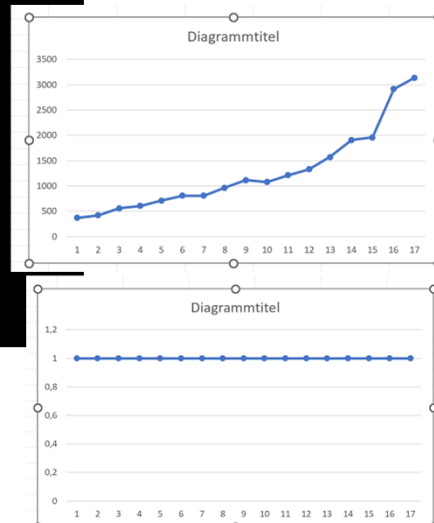
```

4> zeitintroS:messung(420000,120000,17,1,rand,left,16).
"Elemente: 780000"
"introS Laufzeit i.D.: 604ms left"
" "
"Elemente: 1260000"
"introS Laufzeit i.D.: 969ms left"
" "
"Elemente: 1740000"
"introS Laufzeit i.D.: 1327ms left"
" "
"Elemente: 2220000"
"introS Laufzeit i.D.: 2912ms left"
" "
ok
5>

```

 ZReiheintro134.csv

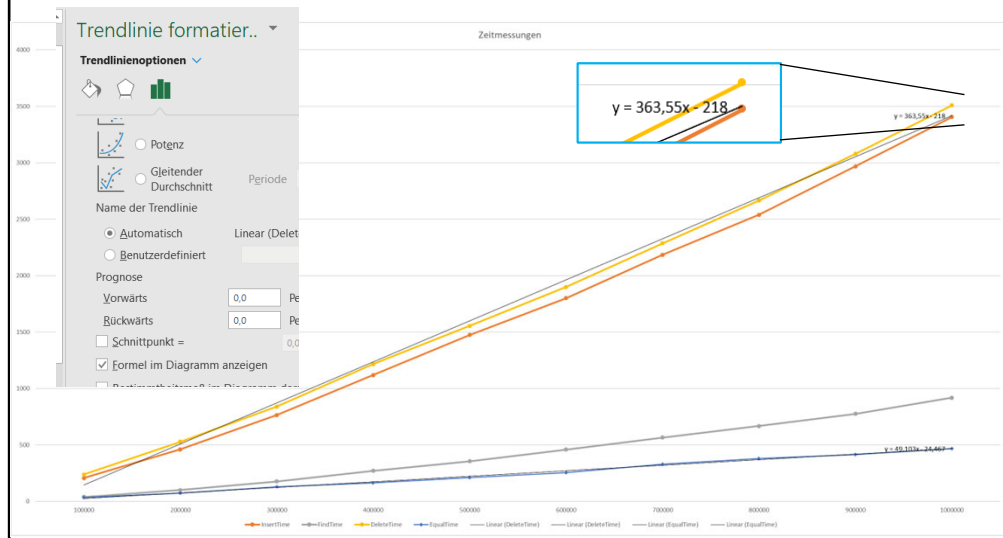
 ZReiheintroElem134.csv



18

18

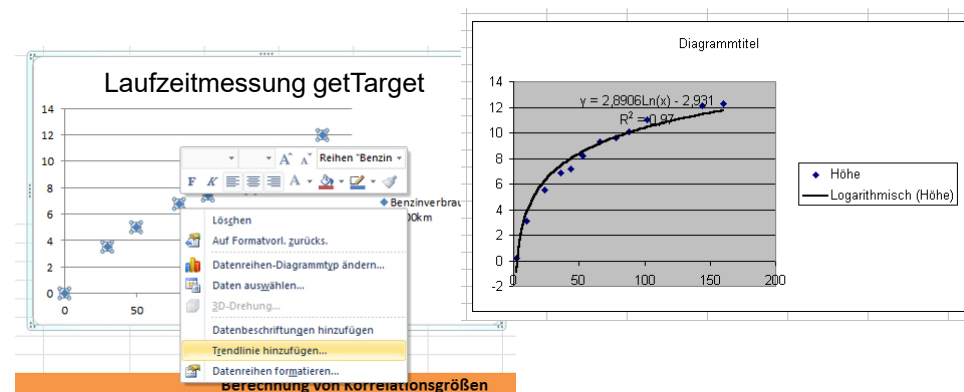
Laufzeitmessung



19

19

Laufzeitmessung



```

Start = erlang:timestamp(),
        Aufruf Algorithmus,
Ende = erlang:timestamp(),
Diffms = util:float_to_int(timer:now_diff(Ende,Start)/1000),

```

Einheit: ms!

20

20

Laufzeitmessung

```
8> heapS:profile_hsort(420000,r).
Reading trace data...
```

Spalte CNT: Gesamtzahl der Funktionsaufrufe

Spalte ACC: Gesamtzeit des Trace (mit externen Funktionen)

Spalte OWN: Summe der Ausführungszeit (ohne externe Funktionen)

```
... fprof.trace
```

2023 08:53

TRACE-Datei

7.269.137 KB

```
End of trace!
Processing data...
Creating output...
%% Analysis results:
{ analysis_options,
  [{callers, true},
   {sort, acc},
   {totals, false},
   {details, true}]}.
```

	CNT	ACC	OWN	
{ totals,	40378929	83002.318	82824.379	XXX
{ "<0.102.0>",	40378929	undefined	82824.379	XX
{ {undefined,		0.83002.318,	0.104}}	
{ {fprof,apply_start_stop,4},		0.83002.318,	0.104}}	%
{ {heapS,heapS,1},		1.83002.214,	0.002}}	
{suspend,	1,	0.000,	0.000}}	
{ {fprof,apply_start_stop,4},		1.83002.214,	0.002}}	
{ {heapS,heapS,1},		1.83002.214,	0.002}}	%
{ {heapS,reHeapdown,1},		1.52309.264,	0.001}}	
{ {heapS,reHeapup,1},		1.30692.948,	0.002}}	

```
profile_hsort (Num,Case) ->
  case Case of r -> Liste = util:randomliste (Num) ;
               us -> Liste = util:sortliste (Num) ;
               bs -> Liste = util:resortliste (Num) end,
  fprof:apply(heapS, heapS, [Liste]),
  fprof:profile(), fprof:analyse().
```

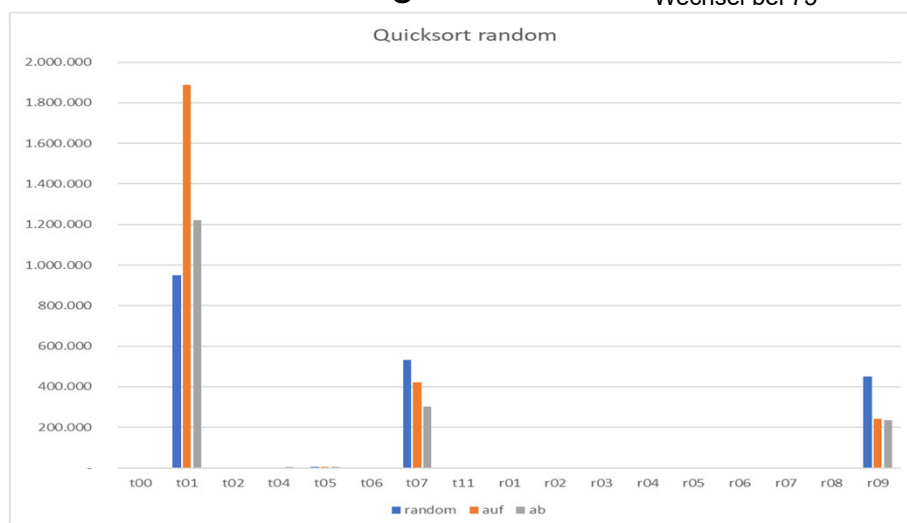
Einheit: ms!!

21

21

Ergebnisse

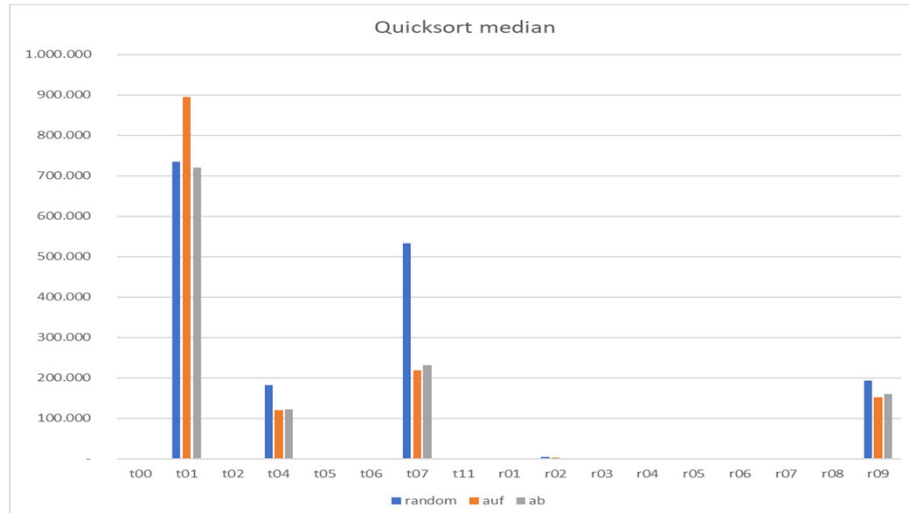
123456 zufällige Zahlen
Wechsel bei 75



22

22

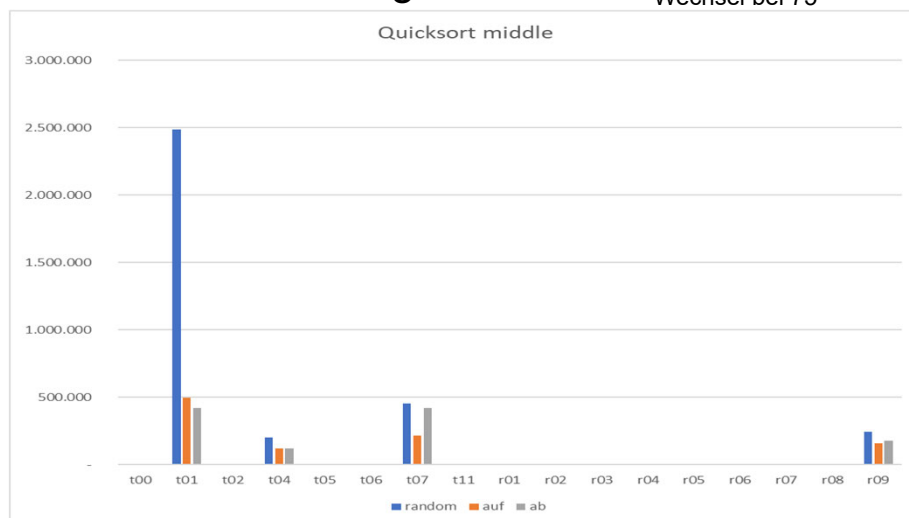
Ergebnisse

123456 zufällige Zahlen
Wechsel bei 75

23

23

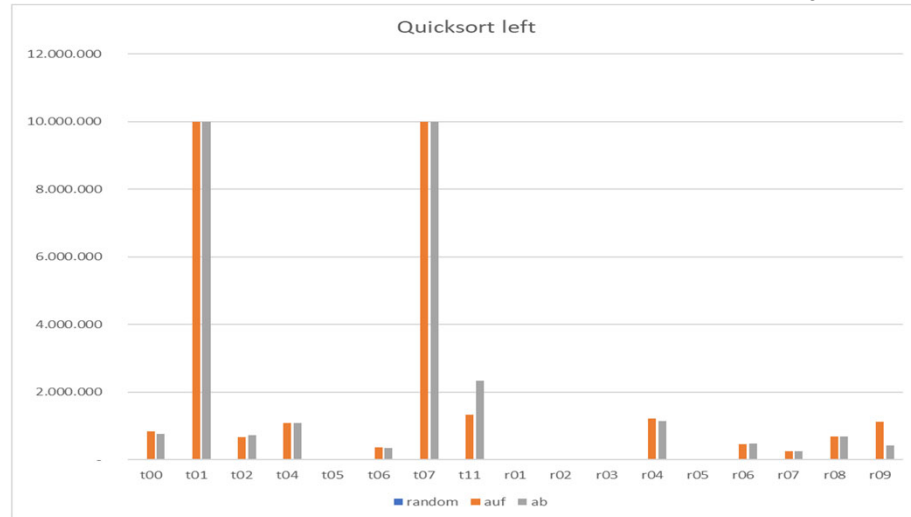
Ergebnisse

123456 zufällige Zahlen
Wechsel bei 75

24

24

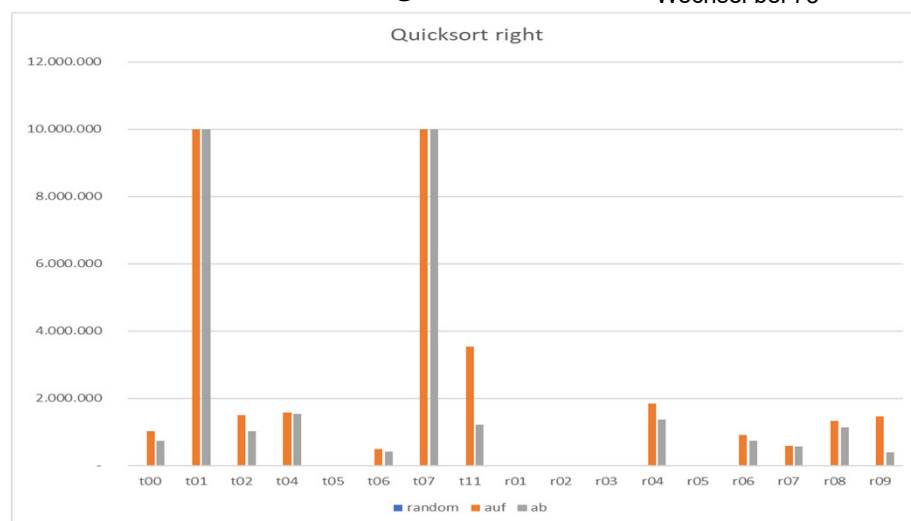
Ergebnisse

123456 zufällige Zahlen
Wechsel bei 75

25

25

Ergebnisse

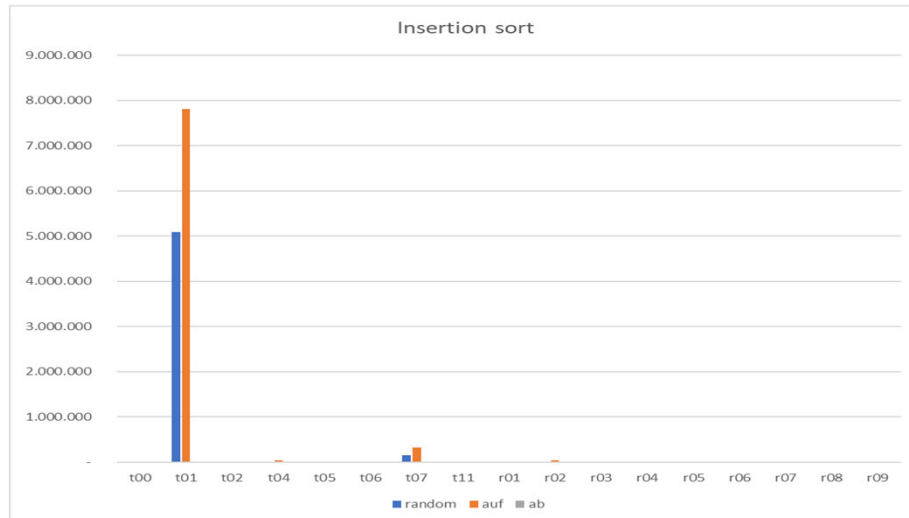
123456 zufällige Zahlen
Wechsel bei 75

26

26

Ergebnisse

12345 zufällige Zahlen

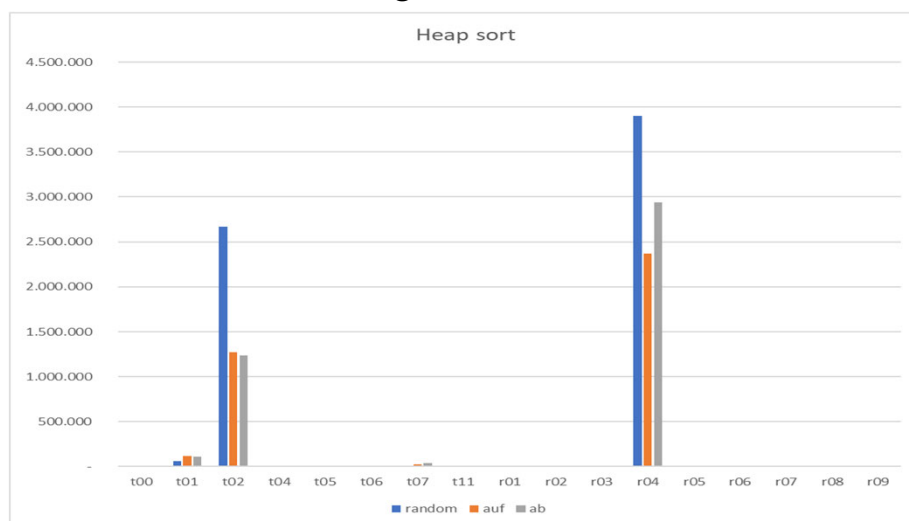


27

27

Ergebnisse

123456 zufällige Zahlen



28

28

Meine Zeiten

- ♦ **Radix Sort 2000042 Zahlen**
zufällig 8127 ms aufsteigend: 7500 ms absteigend: 7951 ms
- ♦ **Introsort middle 2000042 Zahlen**, Wechsel bei 16
zufällig: 4426 ms aufsteigend: 2132 ms absteigend: 2743 ms
- ♦ **Introsort left 2000042 Zahlen**, Wechsel bei 16
zufällig: 4453 ms aufsteigend: 17474 ms absteigend: 15977 ms
- ♦ **Introsort random 2000042 Zahlen**, Wechsel bei 16
zufällig: 4863 ms aufsteigend: 3269 ms absteigend: 3335 ms
- ♦ **Introsort median 2000042 Zahlen**, Wechsel bei 16
zufällig: 5239 ms aufsteigend: 2746 ms absteigend: 3880 ms
- ♦ **Introsort right 2000042 Zahlen**, Wechsel bei 16
zufällig: 5510 ms aufsteigend: 15906 ms absteigend: 18017 ms
- ♦ **Heap Sort 2000042 Zahlen**
zufällig: 21643 ms aufsteigend: 9245 ms absteigend: 11222 ms
- ♦ **Insertion Sort 20000 Zahlen**
zufällig: 5447 ms aufsteigend: 10932 ms absteigend: 1 ms

29

29

Abnahme

- ♦ Bis Donnerstagabend 20:00 Uhr, eine Woche vor dem Praktikumstermin ist bitte der Entwurf für die Aufgabe als *.pdf Dokument (Dokumentationskopf nicht vergessen!) mir per E-Mail (mit cc an den/die Teamprätner_in) zuzusenden. **Beachten Sie die Vorgaben zu einem Entwurf!**
- ♦ Am Abend vor dem Praktikum ist bis 20:00 Uhr bitte die finale Abgabe als *.zip Dokument einzusenden: bitte finaler Stand (als *.zip mit cc an Teampartner_in) per E-Mail abgeben, bestehend aus
 - dem von Ihnen entwickeltem Code zur Lösung der Aufgabe, der alle Vorgaben erfüllen muss. In den Sourcdateien ist auf den Entwurf zu verweisen, um die Umsetzung der Vorgaben zu dokumentieren.
 - den log-Dateien sowie der Bildschirmausgabe mit dem vorgegebenem Testsystem (testSORT:testSORT(<Anzahl Zahlen,<Dateiname als Atom>)). Dieser Test entbindet Sie jedoch nicht von eigenen Tests. Bei Fehlermeldungen aus dem gegebenen System melden Sie sich bitte unmittelbar mit dem gesamten eigenen Code, dem Entwurf und der Fehlermeldung bei mir per E-Mail.
 - den Ergebnissen der durchgeführten Laufzeitmessungen inklusive deren Interpretation.
- ♦ Die Abgabe wird als Abgabe gewertet, wenn der zugehörige Entwurf mindestens als ausreichend bewertet wurde und die beschriebenen Tests erfolgreich durchläuft.
- ♦ Am Tag des Praktikums findet eine Vorführung und eine Besprechung mit den Teams statt. Die Besprechung muss erfolgreich absolviert werden, um weiter am Praktikum teilnehmen zu können. Bei der Besprechung handelt es sich nicht um die Abnahme!

30

30