Datenstrukturen und Algorithmen
Übung 01 – Grundlagen

Musterlösung

Grundlagen: Summen, Produkte und Induktion

1. Leiten Sie eine einfache Formel für $\sum_{k=1}^{n}(5k+1)$ her. **(1 Punkt)**

Bei der Vereinfachung wird zuerst die Linearitätseigenschaft ausgenutzt.

$$\sum_{k=1}^{n}(5k+1) = \sum_{k=1}^{n}5k + \sum_{k=1}^{n}1 = n + 5\sum_{k=1}^{n}k = (\star)$$

Die Summe $\sum_{k=1}^{n}k$ ist eine arithmetische Reihe und hat den Wert

$$\sum_{k=1}^{n}k = \frac{n(n+1)}{2}$$

Dies führt zu einer weiteren Vereinfachung:

$$\begin{aligned}(\star) &= n + 5\sum_{k=1}^{n}k \\ &= n + 5\frac{n(n+1)}{2} \\ &= n + \frac{5}{2}n^2 + \frac{5}{2}n \\ &= \boxed{\frac{7}{2}n + \frac{5}{2}n^2}\end{aligned}$$

2. Werten Sie das Produkt $\Pi_{k=1}^{n}3\cdot 2^k$ aus. **(1 Punkt)**

Das Produkt kann in zwei Produkte zerlegt werden:

$$\Pi_{k=1}^{n}3\cdot 2^k = (\Pi_{k=1}^{n}3)\cdot(\Pi_{k=1}^{n}2^k) = (\star)$$

Das erste Produkt ist

$$\Pi_{k=1}^{n}3 = 3^n$$

also

$$(\star) = 3^n\Pi_{k=1}^{n}2^k.$$

Ausgeschrieben sieht das zweite Produkt wie folgt aus:

$$2^1\cdot 2^2\cdot 2^3\cdot 2^4\cdot\ldots\cdot 2^n = (\star\star)$$

Benutzt man die Potenzregeln, kann das Produkt als Term mit einer Summe geschrieben werden:

$$(\star\star) = 2^{\sum_{k=1}^{n} k} = (\star\star\star)$$

Hier findet sich die arithmetische Reihe wieder, also

$$(\star\star\star) = 2^{\frac{n(n+1)}{2}}$$

Folglich enspricht unser ursprüglicher Ausdruck

$$\Pi_{k=1}^{n} 3 \cdot 2^k = \boxed{3^n \cdot 2^{\frac{n(n+1)}{2}}}$$

3. Zeigen Sie, dass die Reihe $\sum_{k=1}^{n} \frac{1}{k^2}$ durch eine von n unabhängige Konstante nach oben beschränkt ist. Beweisen Sie dies aber nicht direkt, sondern beweisen Sie zu erst folgendes Hilfsresultat per Induktion und benutzen Sie es um die Aussage zu beweisen:

$$\sum_{k=1}^{n} \frac{1}{k^2} \leq 1 + 1 - \frac{1}{n}$$

Strukturieren Sie den induktiven Beweis so, wie in der Vorbesprechung vorgestellt. Es könnte ausserdem nützlich sein, dass $\frac{1}{n} = \frac{1}{n+1} + \frac{1}{n(n+1)}$; dies gilt wegen $\frac{1}{n} - \frac{1}{n+1} = \frac{n+1-n}{n(n+1)} = \frac{1}{n(n+1)}$. **(1 Punkt)**

Aus dem Hilfsresultat folgt direkt

$$\sum_{k=1}^{n} \frac{1}{k^2} \leq 2,$$

da $-1/n$ immer negativ ist. Wir zeigen das Hilfsresultat per vollständiger Induktion:

Induktionsanfang: $n = 1$:

$\sum_{k=1}^{1} \frac{1}{k^2} = 1 \leq 1 + 1 - \frac{1}{1}$ ✓.

Induktionsschritt:

Induktionsvoraussetzung (IV): Gelte die Aussage für ein $n \in \mathbb{N}, n > 0$. Dann:

$$\begin{aligned}
\sum_{k=1}^{n+1} \frac{1}{k^2} &= \sum_{k=1}^{n} \frac{1}{k^2} + \frac{1}{(n+1)^2} \\
&\overset{\text{(IV)}}{\leq} 1 + 1 - \frac{1}{n} + \frac{1}{(n+1)^2} \\
&\overset{(\star)}{=} 1 + 1 - \left(\frac{1}{n+1} + \frac{1}{n \cdot (n+1)}\right) + \frac{1}{(n+1)^2} \\
&= 1 + 1 - \frac{1}{n+1} + \underbrace{\frac{1}{(n+1)^2} - \frac{1}{n \cdot (n+1)}}_{\geq 0} \\
&\leq 1 + 1 - \frac{1}{n+1}.
\end{aligned}$$

($(\star)$ ist hierbei die Anwendung des Hinweises aus der Aufgabenstellung.)

Somit ist die Aussage für $n + 1$ gezeigt.

Mit vollständiger Induktion folgt, dass das Hilfsresultat für alle $n \in \mathbb{N}$ gilt.

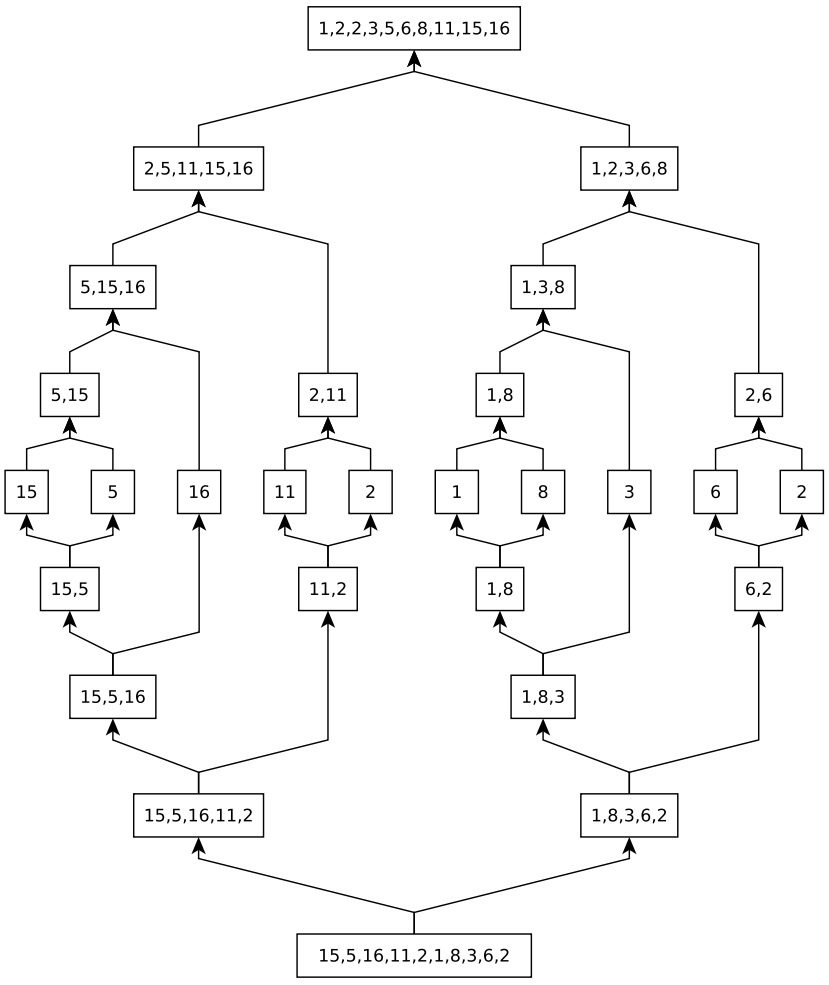
Theoretische Aufgaben

1. Illustrieren Sie den Ablauf von Sortieren durch Einfügen (INSERTION SORT) anhand einer Skizze. Verwenden Sie die Eingabe $\langle 15,5,16,11,2,1,8,3,6,2\rangle$. **(1 Punkt)**

Jede Zeile gibt an, wie das Array **vor** dem Schleifendurchlauf mit dem angegebenem j aussieht.

j = 2:	15	5	16	11	2	1	8	3	6	2
j = 3:	5	15	16	11	2	1	8	3	6	2
j = 4:	5	15	16	11	2	1	8	3	6	2
j = 5:	5	11	15	16	2	1	8	3	6	2
j = 6:	2	5	11	15	16	1	8	3	6	2
j = 7:	1	2	5	11	15	16	8	3	6	2
j = 8:	1	2	5	8	11	15	16	3	6	2
j = 9:	1	2	3	5	8	11	15	16	6	2
j = 10:	1	2	3	5	6	8	11	15	16	2
	1	2	2	3	5	6	8	11	15	16

2. Illustrieren Sie den Ablauf von Sortieren durch Mischen (MERGE SORT) ähnlich wie in Figure 2.4 im Buch. Verwenden Sie dieselbe Eingabe wie oben. Wird ein Feld $A[p...r]$ ungerader Länge in zwei Hälften geteilt soll die untere Hälfte ein Element grösser sein als die obere. **(1 Punkt)**



Hinweis: Es reicht aus, den Merge-Teil darzustellen.

3. Wir betrachten das folgende *Suchproblem*:

Eingabe: Ein Feld von Zahlen $A = \langle a_1, a_2, \ldots a_n \rangle$ und ein Wert v.
Ausgabe: Ein Index i, so dass $v = A[i]$ oder den speziellen Wert NIL, falls v nicht in A vorkommt.

(a) Schreiben Sie Pseudocode, welcher das Problem löst, indem das Feld Element um Element durchlaufen wird, bis v gefunden oder das Ende des Feldes erreicht wird.

(b) Zeigen Sie, dass Ihr Algorithmus korrekt ist, indem Sie eine Schleifeninvariante beweisen. Das heisst: formulieren Sie eine Schleifeninvariante, „beweisen" sie die Invariante, indem Sie je in einigen Sätzen die Initialisierungseigenschaft, die Fortsetzungseigenschaft und die Terminierungseigenschaft begründen. Benutzen Sie dann die Schleifeninvariante um die Korrektheit des Algorithmus zu beweisen. (Vgl. auch Kapitel 2.1 im Buch)

(c) Was ist die Worst-Case-Laufzeit dieses Algorithmus?

(1 Punkt)

```
LINEAR-SEARCH(A,v)
1  i := 1
2  while i ≤ A.length
3
4        if A[i] == v
5              return i
6        i := i + 1
7
8  return NIL
```

Schleifeninvariante Es bezeichne n die Länge der Eingabe A. Wir wählen

$$I(A,i,v) := v \notin A[1..i-1].$$

Initialisierung: Am Anfang der Schleife ist $i = 1$, damit ist $A[1..i-1] = A[1..0] = \emptyset$ leer und die Invariante trivial erfüllt ✓.

Fortsetzung: Angenommen, die Invariante ist erfüllt am Anfang der Iteration i. Falls $A[i] == v$, so wird die Schleife abgebrochen. Ansonsten ist $v \neq A[i]$, zusammen mit der Invariante $v \notin A[1..i-1]$ gilt somit $v \notin A[1..i]$. Nach Inkrementierung von i gilt also $v \notin A[1..i-1]$, die Invariante ist weiter erfüllt ✓.

Terminierung: Fallunterscheidung: Fall **(a)** Die Schleife terminiert per *return* in Iteration i. Am Anfang der Iteration galt die Invariante, und da A, i und v unverändert blieben, ist die Invariante weiter erfüllt. Fall **(b)**: Die Schleife terminiert regulär mit $i = n + 1$; in diesem Fall haben wir bereits gezeigt, dass die Schleifeninvariante erfüllt ist ✓.

Korrektheit Falls das *return i*-Statement erreicht wird, ist v an Stelle i im Array erhalten. Falls die Schleife komplett durchläuft, ist $i = n + 1$, und zusammen mit der Schleifeninvariante folgt $v \notin A[1..i] = A[1..n+1-1]$, also $v \notin A$. Somit ist NIL der korrekte Rückgabewert ✓.

Laufzeit Die Schleife wird im worst case genau n mal durchlaufen, wobei n die Länge der Eingabe A ist. Somit ist die Worstcase-Laufzeit in $\Theta(n)$, der Algorithmus befindet sich somit in $\mathcal{O}(n)$.

4. Das Suchproblem oben kann effizienter gelöst werden, wenn das Eingabefeld sortiert ist. In diesem Fall kann das Element in der Mitte des Feldes mit v verglichen werden, und die eine Hälfte des Feldes kann von der Suche ausgeschlossen werden. BINÄRE SUCHE ist ein Algorithmus, der diese Idee wiederholt anwendet, indem die verbleibende Hälfte des Feldes jeweils wieder halbiert wird.

   (a) Schreiben Sie Pseudocode für binäre Suche, der iterativ (nicht rekursiv) abläuft.

   (b) Zeigen Sie, dass Ihr Algorithmus korrekt ist, indem Sie eine Schleifeninvariante beweisen.

   (c) Erklären Sie, warum die Worst-Case-Laufzeit des Verfahrens $\Theta(\log n)$ ist.

(1 Punkt)

BINARY-SEARCH(A, v)

```
1  // A muss aufsteigend sortiert sein
2  l := 1 // lower bound
3  u := A.length // upper bound
4  while l ≤ u
5
6      m := l + ⌈(u−l)/2⌉ // ⌈·⌉: aufrunden
7      if A[m] < v
8          l := m + 1
9      elseif v < A[m]
10         u := m − 1
11     else // v == A[m]
12         return m
13
14 return NIL
```

Schleifeninvariante Es bezeichne n die Länge der Eingabe A. Wir wählen

$$I(A, v, u, l) := v \notin A[1..l-1] \wedge v \notin A[u+1..n].$$

Initialisierung: Zu Anfang der Schleife ist $l = 1$ und $u = n$, die Invariante entspricht also $I(A, v, u, l) = v \notin A[1..0] \wedge v \notin A[n+1..n]$., was trivialerweise erfüllt ist ✓.

Fortsetzung: Angenommen, die Invariante ist zu Beginn der Schleife erfüllt. Fallunterscheidung: Fall **(a)** $A[m] < v$: Da A aufsteigend sortiert ist, git auch $A[k] < v$ für alle $k \leq m$, somit $v \notin A[1..m]$. Mit $l' = m + 1$ (l' bezeichne den neuen Wert von l) gilt also $v \notin A[1..l'-1]$, folglich bleibt die Invariante erhalten. Fall **(b)** $v < A[m]$: Analog zu Fall (a), mit $u' = m - 1$. Fall **(c)** $A[m] == v$: Alle Variablen bleiben unverändert, somit gilt die Invariante unverändert ✓.

Terminierung: Die Schleife kann genau an den Stellen verlassen werden, für die wir bereits den Erhalt der Invariante gezeigt haben (per *return* in Fall (c), oder regulär nach Fall (a) oder (b)) ✓.

Korrektheit Fallunterscheidung: Falls das *return*-Statement erreicht wird, wurde v an Stelle m gefunden. Ansonsten muss $l > u$ gelten. Die in den beiden Ausdrücken der Invariante benannten Teilarrays $[1..l-1]$ und $A[u+1..n]$. umfassen folglich das gesamte Array, somit $v \notin A$, und NIL ist der korrekte Rückgabewert ✓.

Laufzeit Im worst case ist das Element nicht in der Eingabe enthalten. In jeder Iteration wird der zu durchsuchende Bereich $A[u,\ldots,l]$ halbiert, bis er leer ist. Unter Vernachlässigung der Rundungen gilt für die Anzahl der Durchläufe m somit $n = 2^m$, also $m = \log_2 n$. Mit $T(n) = m = \log_2 n$ ergibt sich eine Worstcase-Laufzeit von $\Theta(\log_2 n)$, der Algorithmus ist also enthalten in $\mathcal{O}(\log_2 n)$.

Praktische Aufgaben

1. In der Vorlesung wurden für INSERTION SORT die Zeitkomplexität $\mathcal{O}(n^2)$ und für MERGE SORT $\mathcal{O}(n \log n)$ angegeben. Bestätigen Sie diese Grössenordnungen experimentell. Sie finden Java-Code für beide Algorithmen sowie eine Timer-Klasse im `.zip` dieser Aufgabenstellung.

   Schreiben Sie ein Java Programm zur Durchführung des Experiments. Erzeugen Sie zufällige Eingabefelder verschiedener Grösse und messen Sie jeweils die Zeit, die zum Sortieren nötig ist. Zur Zeitmessung können Sie die Klasse `Timer` verwenden, die wir zur Verfügung stellen. Beachten Sie, dass die Messung von kurzen Zeitspannen relativ ungenau sein kann.

   Als Abgabe erwarten wir den Ausdruck Ihres Testprogramms. Weiter sollen Sie eine Grafik erstellen, welche die Zeitmessungen für beide Algorithmen und verschieden grosse Felder zusammenfasst. Erwähnen Sie auch die Spezifikationen des Computers, auf denen das Experiment durchgeführt wurde. Erläutern Sie, ob Ihre Messungen der Theorie entsprechen.

   Schätzen Sie zudem auf Grund Ihrer Messungen die Laufzeit von INSERTION SORT bei der Eingabe von 10'000'000 Zahlen ab.

   Hinweise: Konsultieren Sie die Java-Dokumentation, um herauszufinden wie Zufallszahlen erzeugt werden können. Sie können die Grafik (akkurat) von Hand zeichnen oder ein Softwareprogramm wie z.B. gnuplot, R, oder Excel verwenden.

   Vergessen Sie nicht ihren Sourcecode innerhalb der Deadline über die Ilias-Aufgabenseite einzureichen.

   (3 Punkte)

Hinweis: Das hier gezeigte Programm gibt Daten auf der Standardausgabe im CSV-Format aus, das von vielen Programmen gelesen werden kann. Laufzeiten von -1 bedeuten "kein Messwert vorhanden".

```
1  import java.util.*;
2
3
4  public class Benchmark
5  {
6      long time_insertionSort(int[] arr)
7      {
8          Timer timer = new Timer();
9          Sorting.insertionSort(arr);
10         return timer.timeElapsed();
11     }
12
13     long time_mergeSort(int[] arr)
14     {
15         Timer timer = new Timer();
16         Sorting.mergeSort(arr, 0, arr.length-1);
17         return timer.timeElapsed();
18     }
19
```

```
20      long time_builtinSort(int[] arr)
21      {
22          Timer timer = new Timer();
23          Arrays.sort(arr);
24          return timer.timeElapsed();
25      }
26
27      void run() {
28          Random random = new Random();
29
            ↪  System.out.println("n_elements;ins_ms;merge_ms;builtin_ms");
30          long duration_merge_ms = 0;
31          long duration_insert_ms = 0;
32          long duration_builtin_ms = 0;
33          List<Integer> lens = new ArrayList<Integer>();
34          for (int len=20000; len <= 300000; len += 20000) {
35              lens.add(len);
36          }
37          for (int len=300000; len <= 3000000; len += 100000) {
38              lens.add(len);
39          }
40          for (int len=3000000; len <= 10000000; len += 500000) {
41              lens.add(len);
42          }
43          for (int len: lens)
44          {
45              int[] random_arr = new int[len];
46              int repetitions = 5;
47              for (int i=0; i < len; ++i) {
48                  random_arr[i] = random.nextInt();
49              }
50              duration_builtin_ms = 0;
51              duration_merge_ms = 0;
52              duration_insert_ms = 0;
53              for (int i = 0; i < repetitions; ++i) {
54                  duration_builtin_ms +=
                        ↪  time_builtinSort(random_arr.clone());
55                  duration_merge_ms +=
                        ↪  time_mergeSort(random_arr.clone());
56              }
57              duration_builtin_ms /= repetitions;
58              duration_merge_ms /= repetitions;
59
60              if (len > 300000) {
61                  duration_insert_ms = -1;
62              } else {
63                  if (len >= 100000) {
64                      repetitions = 3;
65                  } else if (len >= 200000) {
66                      repetitions = 1;
67                  }
68                  for (int i = 0; i < repetitions; ++i) {
```

```
                    duration_insert_ms +=
                     ↪  time_insertionSort(random_arr.clone());
                }
                duration_insert_ms /= repetitions;
            }



            System.out.println(String.valueOf(len)
                    + ";" + duration_insert_ms
                    + ";" + duration_merge_ms
                    + ";" + duration_builtin_ms);

        }
    }
    public static void main(String[] args)
    {
        Benchmark b = new Benchmark();
        b.run();
    }
}
```

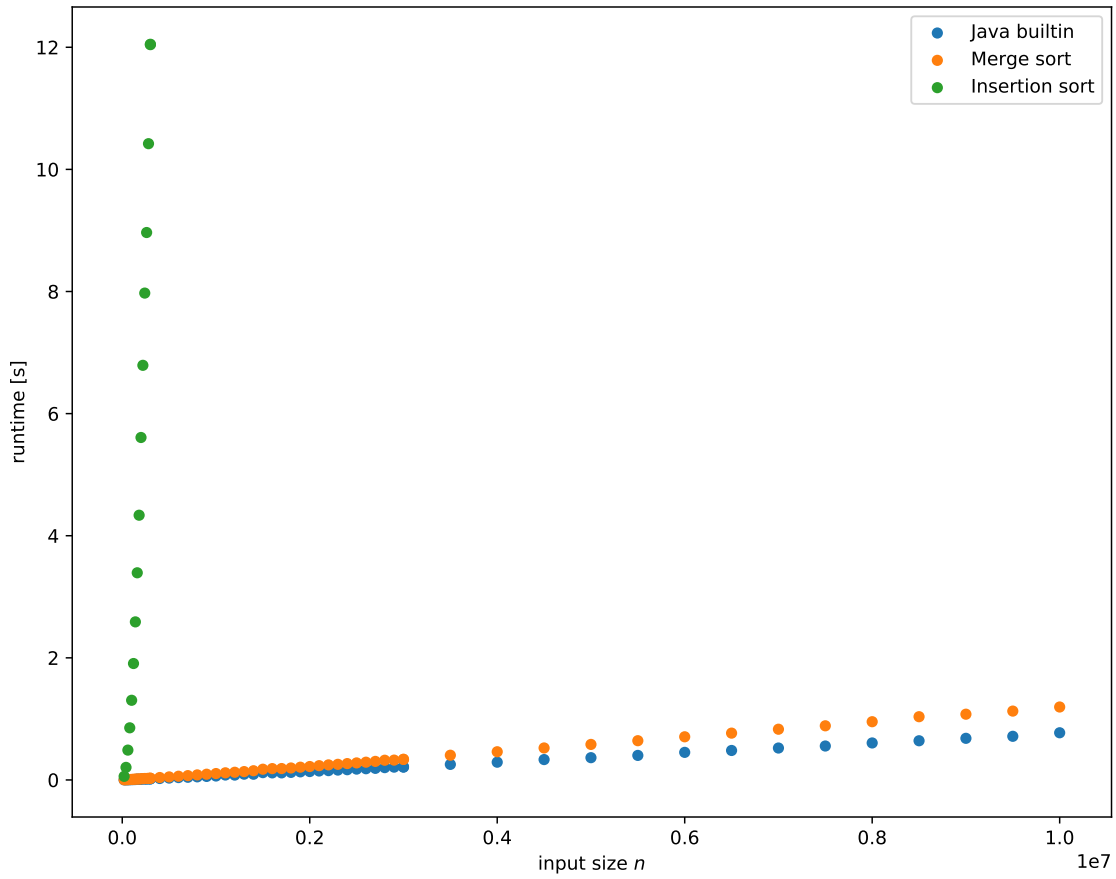
Auf einem Laptop mit einem *i7-8750H* unter Verwendung der JVM `OpenJDK 64-Bit Server VM (build 11.0.2+9-Debian-3)` erhalten wir die Ergebnisse `results.csv`.

Mit linearer Least-Squares-Regression (siehe Code unten) erhalten wir folgende Ergebnisse für die Laufzeit:

Algorithmus	Best fit	R^2
Java builtin	$4.7326 \times 10^{-9} \cdot n \log n + 0.0019005$	0.99961
MERGE-SORT	$7.4787 \times 10^{-9} \cdot n \log n + 0.0030111$	0.99975
INSERTION-SORT	$1.3429 \times 10^{-10} \cdot n^2 + 0.018555$	0.99917

Wenn wir auf diese Weise die Laufzeit von INSERTION-SORT für 1×10^7 Elemente extrapolieren, erhalten wir etwa 3h 44m erwartete Laufzeit (zum Vergleich: Merge sort benötigt für diese Eingabelänge nur 1.2s).

Laufzeit-Plot:



Für die Analyse der Ergebnisse und die Erstellung des plots haben wir das folgende Python-Script verwendet:

```
#!/usr/bin/env python3
# coding: utf-8

# Read results.csv
import csv
lens = []
dur_builtin = []
dur_merge = []
dur_insert = []
with open("results.csv") as fp:
    reader = csv.reader(fp, delimiter=";")
    next(reader, None) # skip headers
    for row in reader:
        l,i,m,b = tuple(map(int, row))
        lens.append(l)
        dur_builtin.append(b/1000.)
        dur_merge.append(m/1000.)
        if i > -1:
            dur_insert.append(i/1000.)


# Perform linear regression
from scipy import stats
```

```
from math import log

n2 = [n**2 for n in lens]
nlogn = [n * log(n) for n in lens]

for (n, rt) in [("merge", dur_merge), ("builtin", dur_builtin)]:
    slope, intercept, r_value, p_value, std_err =
     ↪  stats.linregress(
        nlogn,
        rt)

    print("T_{}: {:.5} * nlogn + {:.5}; R^2 = {:.5}".format(n,
     ↪  slope, intercept,r_value**2))


slope, intercept, r_value, p_value, std_err = stats.linregress(
    n2[:len(dur_insert)],
    dur_insert)

print("T_insertion: {:.5} * n^2 + {:.5}; R^2 =
 ↪  {:.5}".format(slope, intercept,r_value**2))
n=1e7
expected_s = n**2*slope+intercept

from datetime import datetime, timedelta
t = timedelta(seconds=expected_s)
print("Expected time for insertion sort of {} items:
 ↪  {}".format(n, t))


import matplotlib.pyplot as plt

plt.figure(figsize=(10,8))
plt.xlabel("input size $n$")
plt.ylabel("runtime [s]")
shape=25
plt.scatter(lens, dur_builtin, s=shape, label="Java builtin")
plt.scatter(lens, dur_merge, s=shape, label="Merge sort")
plt.scatter(lens[:len(dur_insert)], dur_insert, s=shape,
 ↪  label="Insertion sort")
plt.legend()
plt.show()
plt.savefig("runtime.pdf", bbox_inches="tight")
```