Übungen 4

# Aufgabe 1

Der gegebene Brute Force Algorithmus prüft alle nachfolgenden Werte:

|  |
| --- |
| public int maxSub(int[] data) {  int max = 0;  for (int i = 0; i < data.length; i++) {  int income = 0;  for (int j = i; j < data.length; j++) {  income += data[j];  if (income > max) {  max = income;  }  }  }  return max;  } |

Der «Algorithmus» ergibt folgende Zeiten:

|  |
| --- |
| n = 10 time: 0.000 sec  n = 20 time: 0.000 sec  >>> ratio: 1.9997703788748569  n = 40 time: 0.000 sec  >>> ratio: 2.20599379951774  n = 80 time: 0.000 sec  >>> ratio: 2.613210493441599  n = 160 time: 0.000 sec  >>> ratio: 3.6020515884872024  n = 320 time: 0.001 sec  >>> ratio: 5.732278631505024  n = 640 time: 0.001 sec  >>> ratio: 0.9676307191276355  n = 1280 time: 0.001 sec  >>> ratio: 0.5569458346127312  n = 2560 time: 0.004 sec  >>> ratio: 7.225123017797977  n = 5120 time: 0.005 sec  >>> ratio: 1.1165757436118502  n = 10240 time: 0.013 sec  >>> ratio: 2.949588011690457  n = 20480 time: 0.059 sec  >>> ratio: 4.4657789792944245  n = 40960 time: 0.219 sec  >>> ratio: 3.694984166142785  n = 81920 time: 0.893 sec  >>> ratio: 4.07018827500023  n = 163840 time: 3.482 sec  >>> ratio: 3.8995199961506417  n = 327680 time: 13.987 sec  >>> ratio: 4.017394093834135 |

Was beim Schreiben auffällt:

* Die Summenformel sieht unschön aus, da immer relativen Werte verrechnet werden müssen, um den effektiven Gewinn/Verlust eines Paares zu bestimmen
* Da die Summenformel nötig ist, können negative Werte (Aktie verliert an Wert) nicht ignoriert werden
* Das doppelt verschachtelte for erhöht die Komplexität enorm

Um die Summenformel zu eliminieren, berechne ich zuerst die absoluten Werte und vergleiche diese dann direkt (Das neue Array wird um ein Element grösser und resultiert in einer linearen Komplexität):

|  |
| --- |
| public int maxSub(int[] data) {  // Make all values absolute  int tmpvalue = 0;  int[] tmpdata = new int[data.length + 1];  for (int i = 0; i < data.length; i++) {  tmpvalue += data[i];  tmpdata[i + 1] = tmpvalue;  }  tmpdata[0] = 0;  // Search for the maximum  int max = 0;  for (int i = 0; i < tmpdata.length; i++) {  for (int j = i + 1; j < tmpdata.length; j++) {  int diff = tmpdata[j] - tmpdata[i];  if (diff > max) {  max = diff;  }  }  }  return max;  } |

Der Algorithmus ist minimal schneller, da er die Summenformel eliminiert, aber linear die absoluten Werte berechnen muss:

|  |
| --- |
| n = 10 time: 0.000 sec  n = 20 time: 0.000 sec  >>> ratio: 1.8128355295265983  n = 40 time: 0.000 sec  >>> ratio: 2.93067707632252  n = 80 time: 0.000 sec  >>> ratio: 3.976575417968032  n = 160 time: 0.000 sec  >>> ratio: 2.2662801173508282  n = 320 time: 0.001 sec  >>> ratio: 3.8655216349829264  n = 640 time: 0.001 sec  >>> ratio: 0.6862551980739768  n = 1280 time: 0.001 sec  >>> ratio: 2.076279604144532  n = 2560 time: 0.003 sec  >>> ratio: 3.152878395171969  n = 5120 time: 0.003 sec  >>> ratio: 0.942038828027392  n = 10240 time: 0.012 sec  >>> ratio: 3.8474177913764476  n = 20480 time: 0.054 sec  >>> ratio: 4.3938440121051965  n = 40960 time: 0.204 sec  >>> ratio: 3.751081008499454  n = 81920 time: 0.825 sec  >>> ratio: 4.054913459811471  n = 163840 time: 3.210 sec  >>> ratio: 3.8890906526365887  n = 327680 time: 12.907 sec  >>> ratio: 4.021222321617529 |

Der grösste Zeitverbraucher ist also die doppelt verschachtelte for Schleife (Quadratische Komplexität). Googelt man etwas herum findet man schnell heraus, dass Kadanes Algorithmus als Spezialfall zu einer linearen Komplexität führt:

|  |
| --- |
| public int maxSub(int[] data) {  int maxend = data[0];  int maxfar = data[0];  for (int i = 1; i < data.length; i++) {  maxend = Math.max(data[i], maxend + data[i]);  maxfar = Math.max(maxfar, maxend);  }  return maxfar > 0 ? maxfar : 0;  } |

Zeitverbrauch:

|  |
| --- |
| n = 10 time: 0.000 sec  n = 20 time: 0.000 sec  >>> ratio: 0.718299791048059  n = 40 time: 0.000 sec  >>> ratio: 0.3921763760238842  n = 80 time: 0.000 sec  >>> ratio: 1.2500487995315246  n = 160 time: 0.000 sec  >>> ratio: 2.0398188632104937  n = 320 time: 0.000 sec  >>> ratio: 1.5295108321212585  n = 640 time: 0.000 sec  >>> ratio: 1.7435435435435438  n = 1280 time: 0.000 sec  >>> ratio: 1.6250143529681937  n = 2560 time: 0.000 sec  >>> ratio: 2.5384479499726194  n = 5120 time: 0.000 sec  >>> ratio: 1.4010675091684701  n = 10240 time: 0.000 sec  >>> ratio: 1.9669252776508452  n = 20480 time: 0.001 sec  >>> ratio: 1.9767124152472821  n = 40960 time: 0.001 sec  >>> ratio: 1.6266370971450763  n = 81920 time: 0.001 sec  >>> ratio: 0.4817947211681034  n = 163840 time: 0.000 sec  >>> ratio: 0.5018770121031826  n = 327680 time: 0.000 sec  >>> ratio: 1.2454261113583143  n = 655360 time: 0.001 sec  >>> ratio: 1.8690739151864766  n = 1310720 time: 0.002 sec  >>> ratio: 2.1125795658446607  n = 2621440 time: 0.003 sec  >>> ratio: 2.0512618140269074  n = 5242880 time: 0.006 sec  >>> ratio: 1.9535668001268596 |

# Aufgabe 2

## Teilaufgabe A

|  |
| --- |
| package ch.fhnw.algd1.simplearraylist;  import java.util.AbstractList;  import java.util.List;  // Source: https://docs.oracle.com/javase/8/docs/api/java/util/AbstractList.html  public class SimpleArrayList<E> extends AbstractList<E> implements List<E> {  private int minArrLen = 16;  private Object[] arr = new Object[minArrLen];  private int size = 0;  @Override  public int size() {  return size;  }  @SuppressWarnings("unchecked")  @Override  public E get(int index) {  // Check the index  if (index < 0 || index >= size()) {  throw new IndexOutOfBoundsException("Invalid index");  }  return (E) arr[index];  }  @Override  public E set(int index, E element) {  // UnsupportedOperationException: Not required  // ClassCastException: Not required  // NullPointerException: Not required  // IllegalArgumentException: Not required  E old = get(index); // includes index checking  arr[index] = element;  return old;  }  @Override  public void add(int index, E element) {  // UnsupportedOperationException: Not required  // ClassCastException: Not required  // NullPointerException: Not required  // IllegalArgumentException: Not required  // Check the index  if (index < 0 || index > size()) {  throw new IndexOutOfBoundsException("Invalid index");  }  // In case no empty slot is available, double the array and copy the old content  if (arr.length == size) {  Object[] temparr = new Object[size \* 2];  System.arraycopy(arr, 0, temparr, 0, size);  arr = temparr;  }  // Shift all elements  System.arraycopy(arr, index, arr, index + 1, size - index);  // Set the element  arr[index] = element;  size++;  }  @Override  public E remove(int index) {  E old = get(index); // includes index checking  while (index + 1 < size) {  arr[index] = arr[index + 1];  index++;  }  arr[size] = null;  size--;  return old;  }  } |

## Teilaufgabe B

Gemäss JavaDoc wird beim Einfügen eines Elementes geprüft, ob die Liste genug gross ist. Ist dies nicht der Fall, wird die Liste um ein Element vergrössert und die alten Elemente werden kopiert, bevor das neue Element eingefügt wird.

Da dieser Vorgang beim einzelnen Hinzufügen enorm ineffizient ist, verdopple ich einfach die Liste beim Erreichen der Grösse und transferiere dann den alten Inhalt. Dies braucht meistens mehr Speicher, eliminiert aber die Vergrösserungs- und Kopiervorgänge.

## Teilaufgabe C

Misst man jeden Indexzugriff (Wert aus Array lesen oder auch in ein Array schreiben – jedoch ohne Allozierung beim Vergrössern) auf ein Array ergeben sich folgende Werte:

* Accesses for 128 items: 368
* Accesses for 1024 items: 3056
* Accesses for 32768 items: 98288

## Teilaufgabe D

Gemäss Teilaufgabe C circa das dreifache der finalen Elementlänge.