Iterative Listen

Sven Eric Panitz

7. April 2021

Inhaltsverzeichnis

1 Eine Reihung als Grundlage einer Listeklasse

Zum Einüben der Rekursion auf rekursiven Strukturen haben wir in einer Aufgabe sehr viele kleine rekursive Funktionen umgesetzt. Wir haben bereits im ersten Semester eine Umsetzung von Listen auf Grundlage einer Reihung, in der die Listenelemente gespeichert werden, gesehen. Zur weiteren Übung, sollen in dieser Aufgabe die Methoden der rekursiven Liste nun für Array-basierte Liste umgesetzt werden.

Statt lauter rekursiver Funktionen, die über die rekursive Struktur laufen, bis eine leere Liste erreicht ist, werden nun lauter iterative Methoden geschrieben, in denen zumeist eine Schleife mit einem Index über die Liste iteriert.

Zunächst einmal die Imports unserer Umsetzung. Wir wollen eine Reihe von Funktionen höherer Ordnung implementieren, so dass wir uns ein paar der wichtigen funktionalen Schnittstellen implementieren.

Java: AL

```
package name.panitz.util;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.Consumer;
import java.util.Comparator;
```

Die eigentliche Implementierung ist die generische Klasse `AL`. Sie ist generisch gehalten über die Elemente der Liste. `AL` steht dabei für *Array List*.

Java: AL

```
public class AL<E>  {
```

Intern benötigt sie zwei Felder. Eine Zahl, die anzeigt, wie viele Elemente in der Liste enthalten sind, und die eigentliche Reihung, in der die Elemente gespeichert sind.

Java: AL

```
private int size = 0;
private Object[]store = new Object[10];
```

Man mache sich klar, dass die Länge der Reihung `store.length` in der Regel größer ist als die Länge der Liste `size`. Die Länge der Reihung stellt die derzeitige Kapazität der Liste dar, die `size` hingegen den tatsächlichen Füllgrad der Liste.

Die elemantare Testfunktion, ob die Liste überhaupt ein Elenegt enthält, lässt sich über das feld `size` direkt ausdrücken.

Java: AL

```
public boolean isEmpty(){return size==0;}
```

Somit lässt sich auch die Funktion `length`, die in der rekursiven Implementierung ein komplettes Durchzählen der Elemente vornehmen musste, direkt über das Feld `size` ermitteln.

Java: AL

```
public int length(){
  return size;
}
```

Ebenso hat man einen direkten Zugriff auf die Elemente über den Index.

Java: AL

```
public E get(int i){
  if (i>=size||i<0) throw new IndexOutOfBoundsException();
  return (E)store[i];
}
```

Diese beiden Funktionen, haben anders als in der rekursiven Lösung einen konstanten und keinen linearen Aufwand. Anders als in der rekursiven Lösung ist es also nicht schädlich diese beiden Methoden zur Umsetzung anderer Funktionen aufzurufen.

In der rekursiven Umsetzung gab es nur ein Elemente, auf das direkt zugegriffen werden konnte, nämlich das erste Element, auf das mit der Selektorfunktion `head` zugegriffen werden konnte. Diese lässt sich natürlich auch für die Klasse `AL` leicht implementieren.

Java: AL

```
public E head(){return get(0);}
```

Konstruktoraufruf Da wir keinen Konstruktor für die Klasse geschrieben haben, gibt es nur den leeren Konstruktor, der entsprechend eine leere Liste erzeugt. In Analogie zu den rekursiven Listen, definieren wir eine statische Methode `nil()`zum Erzeugen einer leeren Liste.

Java: AL

```
public static <E>AL<E> nil(){return new AL<>();}
```

Hinzufügen von Elementen Anders als bei den rekursiven Listen, haben wir nur ein Objekt vorliegen, in dem alle Listenelemente gebündelt vorliegen. Bei den rekursiven Listen hatte jedes Element sein eigenes Listenobjekt, dass verkette war mit den Listenobjekten der Restliste.

Es ergibt sich eine vollkommen andere Arbeitsweise mit den Listen. Während bei den rekursiven Liste jeweils vorne ein weiteres Listenobjekt mit dem Konstruktor `Cons` angehängt wurde, sehen wir für die Klasse `AL` eine modifizierende Methode vor, die der Liste ein weiteres Element am Ende hinzu fügt.

Das eigentliche Hinzufügen geschieht, indem am Index `size` das neue Element gespeichert wird und anschließend das Feld `size` um eins erhöht wird: `store[size++] = e`. Problematisch ist nur, wenn die Kapazität der Listenklasse ausgeschöpft ist, weil alle Indizes des Arrays bereits mit einem Element belegt sind. Dann sind die Elemente in einen neueren, größeren Array zu kopieren, der dann Platz für weitere Elemente hat.

Java: AL

```
public void add(E e){
  if (size>=store.length) enlargeStore();
  store[size++] = e;
}
```

Die Kopie der einzelnen Elemente in eine neue, größere Reihung ist in einer privaten Hilfsmethode ausgelagert.

Java: AL

```
private void enlargeStore(){
  Object[]newStore = new Object[store.length+10];
  for (int i=0;i<size;i++) newStore[i]=store[i];
  store=newStore;
}
```

Konstruktion mit variabler Argumentanzahl Ebenso wie für die rekursiven Listen sehen wir eine statische Funktion vor, mit deren Hilfe eine Liste durch Aufzählung der Elemente erzeugt werden kann.

Java: AL

```
public static <E>AL<E> of(E...es){
  AL<E> r = nil();
  for (var e:es) r.add(e);
  return r;
}
```

Stringdarstellung Die Methode `show` listet die Listenelemente in eckigen Klammern mit Komma separiert auf.

Anders als in der rekursiven Liste, nutzen wir eine Schleife, um alle Elemente in die Stringdarstellung zu integrieren.

Java: AL

```
@Override public String toString(){
  StringBuffer result = new StringBuffer("[");
  boolean first = true;
  for (var i=0;i<size;i++){
    if (first) first = false;else result.append(", ");
    result.append(store[i]);
  }
  result.append("]");
  return result.toString();
}
```

Gleichheit Anders als in der Umsetzung der rekursiven Listen, in der mit einer Record-Klasse gearbeitet wurde, müssen wir eine eigene Implementierung der Gleichheit liefern. Für Record-Klassen wird eine Gleichheitsfunktion immer direkt generiert.

Java: AL

```
@Override public boolean equals(Object o){
  if (o.getClass()!=AL.class) return false;
  var that = (AL<E>)o;
  if (this.length()!=that.length()) return false;
  for (int i=0;i<size;i++){
    if (!this.get(i).equals(that.get(i)))return false;
  }
  return true;
}
```

1.1 Aufrufe auf der JShell

Einfache Testsaufrufe lassen sich auch für diese Listenimplementierung in der JShell durchführen. Somit lässt sich dieser Abschnitt aus dem Lehrbrief über rekursive Liste mehr oder weniger direkt übernehmen.

Hierzu ist die Klasse mit dem Javacompiler zu übersetzen und die class-Dateien entsprechend der Paketstruktur in einem Ordner zu generieren:

Shell

```
javac --enable-preview -source 15 -d classes/ AL.java
```

Beim Übersetzen ist mindestens die Java-Version 15 mit angeschalteten Preview zu verwenden. Die Option `-d classes` sorgt dafür, dass in dem Ordner `classes` die generierten Klassen entsprechend der Paketstruktur gespeichert werden.

Jetzt kann man eine JShell-Session öffnen:

Shell

```
jshell  --enable-preview --class-path classes
|  Welcome to JShell -- Version 15.0.1
|  For an introduction type: /help intro

jshell>
```

Hierbei ist der Ordner, in dem die generierten Klassen liegen, als Klassenpfad anzugeben.

Es empfiehlt sich, alle statischen Eigenschaften der Schnittstelle `AL` in der JShell-Session zu importieren.

Shell

```
jshell> import static name.panitz.util.AL.*;

jshell>
```

Jetzt kann interaktiv mit den Listen gearbeitet werden.

Shell

```
jshell> nil()
$2 ==> []

jshell> cons(1,cons(2,cons(3,nil())))
$3 ==> [1, 2, 3]

jshell> of(1,2,3,4,5,6)
$4 ==> [1, 2, 3, 4, 5, 6]

jshell> of(1,2,3,4,5,6).get(2)
$5 ==> 3

jshell> of(1,2,3,4,5,6).isEmpty()
$6 ==> false

jshell> of("hallo","welt")
$7 ==> [hallo, welt]
```

2 Aufgaben

Es folgen viele kleine Aufgaben. Alle zu schreibenden Funktionen lassen sich am besten mit einer Schleife umsetzen.

Die Funktionen können sich natürlich gegenseitig aufrufen. Dieses ist vielfach gewünscht. Auch die Funktionen `length` und `get` können in dieser Umsetzung direkt verwendet werden ohne in eine Falle bezüglich des Aufwands zu geraten.

Alle Funktionen können direkt in der JShell kurz getestet werden. Ein Beispielaufruf ist jeweils bei der Aufgabenstellung mit angegeben.

Vergleichen Sie jeweils die Lösungen mit denen für die rekursive Listen. Überlegen Sie, was effektiver in rekursiven und was effektiver in iterativen Listen umgesetzt werden kann.

Aufgabe 1 Wir beginnen mit den vielen kleinen Listen verarbeitenden Funktionen:

a) Schreiben Sie eine Funktion, die das letzte Element der Liste zurück gibt. Ist die Liste leer, so soll eine `NoSuchElementException` geworfen werden.

Java: AL

```java
public E last(){
  return null;   /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).last()
$13 ==> 10
```

b) Schreiben Sie die Funktion, die eine neue Liste erstellt, die mit den Elementen der this-Liste beginnt und anschließend die Elemente der that-Liste hat. Es soll also nicht die this-Liste verändert werden, sondern eine komplett neue Liste erzeugt werden.

Java: AL

```java
public AL<E> append(AL<E> that){
  AL<E> rs = nil();
  return rs;  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).append(of(431,432,431,21,76))
$4 ==> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 431, 432, 431, 21, 76]
```

c) Schreiben Sie jetzt die Funktion, die *keine* neue Liste erstellt, sondern die this-Liste verändert, indem die Elemente der that-Liste in ihr am Ende eingefügt werden.

Java: AL

```java
public void addAll(AL<E> that){
  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> var xs = of(1,2,3,4)
xs ==> [1, 2, 3, 4]

jshell> xs.add
add(      addAll(
jshell> xs.addAll(of(5,6,7,8,9))

jshell> xs
xs ==> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

d) Schreiben Sie die Funktion, die eine Teilliste erzeugt, die ohne die ersten i-Elemente aus der this-Liste entsteht. Ist i größer als die Länge, dann sei das Ergebnis die leere Liste. Ist i negativ, so sei das Ergebnis die gesamte Liste.

Java: AL

```
public AL<E> drop(int i){
  AL<E> rs = nil();
  return rs;  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).drop(6)
$9 ==> [7, 8, 9, 10]
```

Mit Hilfe der Funktion `drop` lässt sich auch die aus den rekursiven Listen bekannte Selektorfunktion `tail` umsetzen.

Java: AL

```
public AL<E> tail(){return drop(1);}
```

In dem Fall der Klasse `AL` ist diese Funktion allerdings problematisch, da sie eine komplette Kopie der Liste ohne das erste Element erstellt. Sie hat damit einen linearen Aufwand und sollte nur mit Vorsicht verwendet werden.

e) Schreiben Sie die Funktion, die eine Teilliste erzeugt, die aus den ersten i-Elemente aus der this-Liste entsteht. Ist i größer als die Länge, dann sei das Ergebnis die komplette Liste. Ist i negativ, dann sei das Ergebnis die leere Liste.

Java: AL

```
public AL<E> take(int i){
  AL<E> rs = nil();
  return rs;  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).take(6)
$10 ==> [1, 2, 3, 4, 5, 6]
```

f) Schreiben Sie die Funktion, die eine Teilliste erzeugt, die mit dem Element an dem angegebenen Index startet und die angegebene Länge hat. Sollten nicht genug Elemente in der Liste sein, so wird die maximale Anzahl der von dem Index an kommenden Elemente genommen.

Java: AL

```
public AL<E> sublist(int from, int length) {
  return nil();  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).sublist(3,4)
$11 ==> [4, 5, 6, 7]

jshell> of(1,2,3,4,5,6).sublist(3,100000)
$2 ==> [4, 5, 6]

jshell> of(1,2,3,4,5,6).sublist(10,100000)
$3 ==> []
```

g) Schreiben eine Funktion, die eine Liste mit den Elementen in umgekehrter Reihenfolge erzeugt.

Java: AL

```
public AL<E> reverse(){
  return nil();   /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).reverse()
$19 ==> [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

h) Schreiben Sie eine Funktion, die eine neue Liste erzeugt, in der zwischen den Elementen der this-Liste das übergebene Element steht.

Java: AL

```
public AL<E> intersperse(E e){
  return nil();      /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6).intersperse(42)
$2 ==> [1, 42, 2, 42, 3, 42, 4, 42, 5, 42, 6]
```

i) Schreiben Sie eine Funktion, die prüft, ob die this-Liste der Anfang der that-Liste ist.

Java: AL

```
public boolean isPrefixOf(AL<E> that){
  return false;     /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4).isPrefixOf(of(1,2,3,4,5,6,7,8,9,10))
$2 ==> true
```

Beachten Sie, dass die leere Liste ein Präfix von jeder Liste ist.

j) Schreiben Sie eine Funktion, die prüft, ob die this-Liste das Ende der that-Liste ist.

Java: AL

```
public boolean isSuffixOf(AL<E> that){
  return false;     /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(8,9,10).isSuffixOf(of(1,2,3,4,5,6,7,8,9,10))
$3 ==> true
```

k) Schreiben Sie eine Funktion, die prüft, ob die this-Liste in der that-Liste enthalten ist.

Java: AL

```
public boolean isInfixOf(AL<E> that){
  return false;     /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(6,7,8,9).isInfixOf(of(1,2,3,4,5,6,7,8,9,10))
$4 ==> true
```

l) Schreiben Sie eine Funktion, die eine neue Liste erzeugt, bei der das ursprüngliche Kopfelement an die letzte Stelle wandert.

Java: AL

```
public AL<E> rotate(){
  return nil();  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6).rotate()
$2 ==> [2, 3, 4, 5, 6, 1]
```

m) Schreiben Sie eine Funktion, die die Liste aller Listen erzeugt, mit der die this-Liste endet.

Java: AL

```
public AL<AL<E>> tails(){
  return of(nil());  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4).tails()
$4 ==> [[1, 2, 3, 4], [2, 3, 4], [3, 4], [4], []]
```

Aufgabe 2 Als nächstes sind einige Funktionen höherer Ordnung zu entwickeln. Eine Funktion wird als Funktion höherer Ordnung bezeichnet, wenn sie ein Argument hat, das wiederum eine Funktion ist. In Java können zwar eigentlich nicht direkt Funktionen sondern immer nur Objekte als Parameter übergeben werden, aber die funktionalen Schnittstelle, die nur eine Methode enthalten, repräsentieren Funktionen.

a) Schreiben Sie eine Funktion, die ein Konsumentenobjekt auf jedes Element der Liste anwendet. Es soll also für alle Elemente der Liste eine bestimmte Methode aufgerufen werden.

Java: AL

```
public void forEach(Consumer<? super E> con) {
    /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> var x = new int[1]
x ==> int[1] { 0 }

jshell> of(1,2,3,4,5,6,7,8,9,10).forEach(y->x[0]+=y)

jshell> x
x ==> int[1] { 55 }
```

b) Schreiben Sie eine Funktion, die genau dann wahr ergibt, wenn mindestens ein Element der Liste das übergebene Prädikat erfüllt.

Java: AL

```
public boolean containsWith(Predicate< ? super E> p) {
  return false;  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).containsWith(x->x>7 && x<10)
$5 ==> true

jshell> of(1,2,3,4,5,6,7,8,9,10).containsWith(x->x>7 && x%6==0)
$6 ==> false
```

c) Schreiben Sie jetzt eine Funktion, die testet, ob ein dem Argument gleiches Element in der Liste enthalten ist.

Java: AL

```
public boolean contains(E el) {
  return false;  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).contains(6)
$7 ==> true

jshell> of(1,2,3,4,5,6,7,8,9,10).contains(42)
$8 ==> false
```

d) Schreiben Sie jetzt eine Funktion, die von der Liste alle Elemente vorne absplittet, für die das übergebene Prädikat wahr ist.

Java: AL

```
public AL<E> dropWhile(Predicate< ? super E> p){
  return nil();  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).dropWhile(x->x<=5)
$2 ==> [6, 7, 8, 9, 10]
```

e) In der nächsten Funktionen soll eine Liste aus der Liste erzeugt werden, solange die Elemente noch das Prädikat erfüllen.

Java: AL

```
public AL<E> takeWhile(Predicate< ? super E> p){
  return nil();   /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).takeWhile(x-> x<8)
$12 ==> [1, 2, 3, 4, 5, 6, 7]
```

f) Schreiben Sie eine Funktion, die eine Liste aus allen Listenelementen erzeugt, für die das übergebene Prädiukat erfüllt ist.

Java: AL

```
public AL<E> filter(Predicate<? super E> p){
  return nil();   /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).filter(x->x%2==0)
$17 ==> [2, 4, 6, 8, 10]
```

g) Schreiben Sie eine Funktion, die eine Liste durch Anwendung der übergebenen Funktion auf jedes Listenelement erzeugt.

Java: AL

```
public <R> AL<R> map(Function<? super E, ? extends R> f){
  return nil();    /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).map(x->x*x)
$18 ==> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Aufgabe 3 In dieser Aufgabe sollen Funktionen geschrieben werden, die auch mit Tupelobjekte arbeiten.

Hierzu sei zunächst die Record-Klasse für Paare von Objekten definiert.

Java: AL

```
static public record Pair<A,B>(A fst,B snd){
  @Override public String toString(){return "("+fst()+", "+snd()+")";}
}
```

a) Schreiben Sie die Funktion, die die Elemente der this-Liste paarweise mit den Elementen der that-Liste zu einer Liste aus Paaren verknüpft.

Java: AL

```
public <B> AL<Pair<E,B>> zip(AL<B> that){
  return nil();   /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6).zip(of("A","B","C","D"))
$2 ==> [(1, A), (2, B), (3, C), (4, D)]
```

b) Schreiben Sie die Funktion, die ein Paar aus zwei Listen erzeugt. Die erste Liste soll der Präfix der Elemente sein, die das Prädikat erfüllen, die zweite Liste, die Teilliste vom ersten Element an, das das Prädikat nicht erfüllt.

Java: AL

```
public Pair<AL<E>,AL<E>> span(Predicate<? super E> p){
  return new Pair<>(nil(),nil());   /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,1,2,3).span(x->x<4)
$3 ==> ([1, 2, 3], [4, 5, 6, 1, 2, 3])
```

c) Schreiben Sie die Funktion, die ein Paar aus zwei Listen erzeugt. Das erste Element des Paares soll die Liste aller Elemente sein, die das Prädikat erfüllen, die zweite Liste die Liste aller Elemente, die es nicht erfüllen.

Java: AL

```
public Pair<AL<E>,AL<E>> partition(Predicate<? super E> p){
  return new Pair<>(nil(),nil());   /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,1,2,3).partition(x-> x<5)
$3 ==> ([1, 2, 3, 4, 1, 2, 3], [5, 6])
```

Aufgabe 4 In dieser Aufgabe geht es um Sortierungen.

a) Schreiben Sie die Funktion, die prüft, ob die Liste aufsteigend sortiert ist. Die Sortiereigenschaft wird dabei als Comparator-Objekt übergeben. `Comparator` ist eine funktionale Standardschnittstelle mit der abstrakten Methode `compare`, die zwei Elemente vergleicht. Ist das erste Argument kleiner als das zweite, so ist das Ergebnis eine negative Zahl, ist es größer, dann eine positive Zahl. Sind beide Elemente in der Ordnung gleich, so ist das Ergebnis die Zahl 0.

Java: AL

```
public boolean isSorted(Comparator<? super E> cmp){
  return false;   /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,435,4,2345,33,2,3,2).isSorted((x,y)->x-y)
$2 ==> false

jshell> of(1,2,3,4,5).isSorted((x,y)->x-y)
$3 ==> true
```

b) Sie werden in diesem Semester im Modul ›Algorithmen und Datenstrukturen‹ unterschiedliche Algorithem zum Sortieren kennenlernen. Deren Laufzeitverhalten werden Sie dort analysieren. Dort werden Sie zunächst die Elemente auf Arrays sortieren. Für unsere Listen bietet sich an, den sogenannten *quicksort*-Algorithmus zu implementieren.

Für die leere Liste sei das Ergebnis die leere Liste.

Ansonsten: Partitionieren Sie die tail-Liste, nach den Elementen, die im übergebenen Comparator kleiner sind als das head-Element.

Sie erhalten zwei Teillisten, die der Elemente, die kleiner sind als das erste Element und die der Elemente, die größer oder gleich sind zum ersten Element.

Sortieren Sie rekursiv die beiden Ergebnislisten der Partitionierung.

Erzeugen Sie eine Ergebnisliste, in dem sie die Sortierung der kleineren mit der Sortierung der größeren Elemente zusammenhängen und dazwischen noch das head-Element fügen.

Schreiben Sie die Funktion, die die Liste mit Hilfe des Quicksort-Algorithmus sortiert.

Auch dieses lässt sich in drei Zeilen realisieren.

Java: AL

```
public AL<E> qsort(Comparator<? super E> cmp){
  return nil();     /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,4,5,435,4,2345,33,3,453,423,22,0).qsort((x,y)->x-y)
$2 ==> [0, 1, 2, 3, 4, 4, 5, 22, 33, 423, 435, 453, 2345]
```

Java: AL

```
}
```