What is a Sorting Function?

Fritz Henglein

Department of Computer Science
University of Copenhagen
Email: henglein@diku.dk

WG 2.8 2008, Park City, June 15-22, 2008

Outline

1 Sorting algorithms

- Literature definitions
- What is a sorting criterion?
- Properties of sorting algorithms

Outline

Outline

1. Sorting algorithms
   - Literature definitions
   - What is a sorting criterion?
   - Properties of sorting algorithms
2. Permutation functions
   - Consistency with ordering relation
   - Local consistency
   - Parametricity
   - Stability
3. Representing orders
   - Isomorphisms
   - Structure preservation

Outline

The sorting problem

Cormen, Leiserson and Rivest (1990):

> *"**Input:** A sequence of n numbers $\langle a_1, \ldots, a_n \rangle$.*
> ***Output:** A permutation (reordering) $\langle a'_1, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq a'_n$.'*
> *The input sequence is usually an n-element array, although it may be represented in some other fashion.*
> *[. . .]*

Knuth (1998):

> *"[Given records with keys $k_1 \ldots k_N$.] The goal of sorting is to determine a permutation $p(1)p(2) \ldots p(N)$ of the indices $\{1, 2, \ldots, N\}$ that will put the keys into nondecreasing order [.]"*

Questions

- Can you only sort numbers? What about strings? Sets? Trees? Graphs?
- Is $\leq$ fixed by data type of elements?
- What kind of relation is $\leq$? Total order on elements, in particular antisymmetric?
- Permutation as output or just permuted output?
- ...

Sorting criterion: Total order?

- A sorting algorithm permutes input sequences for a certain explicitly given or implicitly understood *sorting criterion*: its output elements have to be in some given “order”.
- What does it mean to be “in order”?

Sorting criterion, first attempt: A total order specification.

Sorting criterion: Key order?

- Sorting algorithms operate on *records* and sort them according to their *keys*.
- E.g. addresses sorted according to their last names.
- More generally, records sorted according to a *key function*;
- E.g. words in dictionary sorted according to their *signature* (characters in ascending lexicograhic order).
- Total order on the *key* domain, but *not* on the records!

Definition (Key order)

A *key order* for set S is a pair consisting of a total order $(K, \leq_K)$, and a function $key : S \to K$.

Sorting criterion, second attempt: A key order specification.

Key orders too concrete

- *Different* key orders may be *equivalent* for sorting purposes.
- E.g. sorting strings with the key function mapping all letters in a word to upper case, or another key function mapping all letters to lower case.
- Both key orders define the same *total preorder*

Sorting criterion: Total preorder!

Definition (Total preorder, order, ordering relation)

An *total preorder (order)* (S, R) is a set S together with a binary relation $R \subseteq S \times S$ that is

- *transitive*: $\forall x, y, z \in S : (x, y) \in R \wedge (y, z) \in R \Longrightarrow (x, z) \in R$; and
- *total*: $\forall x, y \in S : (x, y) \in R \vee (y, x) \in R$

Sorting criterion, definition: A total preorder specification.

Permutation versus permuted input

Functionality of a sorting function:

- Input: A sequence of elements, e.g. *["foo", "bar", "foo"]*
- Output: A *permutation*, $[2, 3, 1]$ or *permuted input elements*: *["bar", "foo", "foo"]*
- Not equivalent! Permutation provides more "intensional" information.

What if we are only interested in permuted elements?
Unnecessary burden on algorithms designer ("too concrete specification") to have to return a permutation, not just the permuted elements. So:
Property 1 (Permutativity): A sorting algorithm permutes its input: it transforms, possibly destructively, an input sequence into a rearranged sequence containing the same elements.

Sorting two elements

- Imagine you have a sorting algorithm, but nobody has told you the ordering relation $\leq$ (the sorting criterion).
- You are given two distinct input elements x_1, x_2.
- Apply the sorting algorithm to $[x_1, x_2]$ and to $[x_2, x_1]$.
- Assume in both cases the result is $[x_1, x_2]$.
- What can you conclude about the ordering relation between x_1 and x_2?
  - We know for sure: $x_1 \leq x_2$!
  - But what about $x_2 \leq x_1$?
  - We would like to conclude that $x_2 \not\leq x_1$.

Locality

Property 2 (locality): A sorting algorithm can be used as a decision procedure for the order (S, R) it sorts according to: Given x_1, x_2 run it on $x_1 x_2$ and on $x_2 x_1$. If at least one of the results is $x_1 x_2$ then $R(x_1, x_2)$ holds; otherwise it does *not* hold.

Satellite data (keys and records)

Cormen, Leiserson, Rivest (1990):

> *"Each record contains a key, which is the value to be sorted [sic!], and the remainder of the record consists of satellite data, which are usually carried around with the key. In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well."*

Note: Satellite data may be empty.
Property 3 (obliviousness): A sorting algorithm only copies and moves satellite data without inspecting them.

A sorting algorithm may be stable

- For some applications it is important that equivalent input elements—e.g. records with the same key—are returned in the same order as in the input.
- Example: Individual sorting steps in least-significant-digit (LSD) radix sorting.
- So, a final property that some, but not all sorting algorithms have is *stability*.

Property 4 (stability): A stable sorting algorithm returns equivalent elements in the same relative order as they appear in the input.

Summary: Properties of sorting algorithms

A sorting algorithm:

1. operates on sequences and permutes them such that they obey a given total preorder;
2. decides the given ordering relation;
3. treats order-equivalent elements obliviously;
4. outputs order-equivalent elements in the same relative order as they occur in the input, if it is required to be stable.

All these properties can be formulated as extensional properties of the input/output behavior *for a given total preorder*.

That is not the question!

- Question is: What is a sorting function? (Period. No order given.)
- Why is this interesting?
  - Message at last WG2.8 meeting: Don't use binary comparison function to provide access to ordering relation of an ordered type: algorithmic bottleneck!
  - Use *n*-ary sorting function (or variant, such as discriminator): no algorithmic bottleneck, no leaking of representation information.
  - But how do we know that the exposed function is a "sorting" function? We are not given an order to start with!
- General problem: We are used to *defining* sorting *given* an order. We now want to reverse this: *define* order *given* sorting function.

Permutation function

What makes a function f a sorting function?
Let us formulate some intrinsic requirements: properties f must have without reference to any *a priori* order.

Definition (Permutation function)

A function f is a *permutation function* if $f : S^* \to S^*$ and $f(\vec{x})$ is a permutation of $\vec{x}$ for all $\vec{x} \in S$.

Requirement 1: f must be a permutation function.

Consistency with ordering relation

Definition (Consistency with ordering relation)

Let $f : S^* \to S^*$ be a permutation function. We say f and ordering relation R on S are *consistent* with each other if for all $y_1 y_2 \ldots y_n = f(x_1 x_2 \ldots x_n)$ we have $R(y_i, y_{i+1})$ for all $1 \leq_\omega i <_\omega n$.

Requirement (turns out to be trivial): f must be consistent with *some* ordering relation R.

One permutation function, many ordering relations

- Each permutation function f is consistent with *many* ordering relations, in particular the trivial one: $S \times S$.
- $S \times S$ is the the *biggest* (least informative) relation: it may relate x_1, x_2 even though f never outputs them in that relative order.
- Does f have a *smallest* (most informative) relation?

Canonically induced ordering relation

Definition (Canonically induced ordering relation)

Let $f : S^* \to S^*$ be a permutation function. We call R the *canonically induced ordering relation* of f if

1. f is consistent with R and
2. for all R' that f is consistent with we have $R \subseteq R'$.

We write $\leq_f$ for R and $\equiv_f$ for the equivalence relation induced by $\leq_f$.

Existence and uniqueness of induced ordering

Proposition

$\leq_f$ exists and is unique.
Furthermore, $x \leq_f x' \Leftrightarrow \exists \vec{y} : \vec{y} \in range(f).\, \vec{y} = \ldots x \ldots x' \ldots$; that is, $x \leq_f x'$ if and only if x occurs to the left of x' in the output of f for some input $\vec{x}$.

End of talk?

- *Every* permutation function f induces a unique "best" ordering relation $\leq_f$ consistent with f.
- When somebody asks: "You have a permutation function and say that it sorts its input. But what is the ordering relation it sorts according to?"
- We have an answer: "It is the canonically induced ordering relation, which is uniquely determined by the permutation function."
- So, is *every* permutation function a function that "sorts"?

Bad news: Undecidability

- Every permutation function f canonically induces an ordering relation $\leq_f$.
- How do we get a handle on it? Can we implement it using f?
- Bad news: It is *impossible* to *implement* $\leq_f$ using f!

Theorem (Undecidability of canonically induced ordering relation)

There exists a total, computable permutation function $f : \mathbb{N}_0^* \to \mathbb{N}_0^*$ such that its canonically induced ordering relation $\leq_f$ is recursively undecidable.

Local evidence of the ordering relation

Definition (R_f)

Define $R_f(x_1, x_2) \iff f(x_1 x_2) = x_1 x_2 \vee f(x_2 x_1) = x_1 x_2$.

$R_f(x_1, x_2)$ is "local" evidence for $x_1 \leq_f x_2$.

Proposition

Let f be a permutation function. Then $R_f \subseteq \leq_f$.

Local consistency

Definition (Local consistency)

A permutation function $f : S^* \to S^*$ is *locally consistent* if $x_1 \leq_f x_2 \Longrightarrow R_f(x_1, x_2)$ for all $x_1, x_2 \in S$.

Requirement 2: f must be locally consistent: $\leq_f \subseteq R_f$.

Parametricity: Basics

Idea: Employ relational *parametricity* (Reynolds 1983, Wadler 1990) to capture oblivious treatment of satellite data in sorting algorithms as an extensional property.

Definition (Relations respecting relations)

Let $R, R' \subseteq S \times S$ be binary relations. We say R *respects* R' if $R \subseteq R'$.

Definition (Preserving relations)

Function $f : S^* \rightarrow S^*$ *preserves* relation $R \subseteq S \times S$ if $f(x_1 x_2 \ldots x_n)\, R^*\, f(x'_1 x'_2 \ldots x'_n)$ whenever $x_1 x_2 \ldots x_n\, R^*\, x'_1 x'_2 \ldots x'_n$.

Parametricity requirement

Definition (Parametricity)

A permutation function $f : S^* \rightarrow S^*$ is *parametric* if it preserves all relations that respect $\equiv_f$.

We can now formulate the obliviousness property of sorting algorithms as a parametricity requirement.
Requirement 3: f must be parametric.

Parametricity: locality, characterization of equivalence

Lemma (Parametricity implies locality)

Let $f : S^* \to S^*$ be a parametric permutation function. Then:

1. *f is locally consistent: $x_1 \leq_f x_2$ if and only if $R_f(x_1, x_2)$.*
2. $$x \equiv_f y \iff \begin{array}{ccc} (f(x_1 x_2) = x_1 x_2 & \wedge & f(x_2 x_1) = x_2 x_1) \\ & \vee & \\ (f(x_1 x_2) = x_2 x_1 & \wedge & f(x_2 x_1) = x_1 x_2) \end{array}$$

   for all $x, y \in S$.

Sorting function: Definition

With local consistency subsumed by parametricity, we define a sorting function to be *any parametric permutation function*.

Definition (Sorting function)

We call a function $f : S^* \to S^*$ a *sorting function* if

1. (permutativity) it is a permutation function;
2. (parametricity) it preserves all relations that respect the equivalence relation Q_f defined by

$$Q_f(x_1, x_2) \iff \begin{array}{ccc} (f(x_1 x_2) = x_1 x_2 & \wedge & f(x_2 x_1) = x_2 x_1) \\ & \vee & \\ (f(x_1 x_2) = x_2 x_1 & \wedge & f(x_2 x_1) = x_1 x_2). \end{array}$$

Comparison-based sorting functions

A comparison-based sorting algorithm is an algorithm that is allowed to apply an inequality test (comparison function) to the elements in its input sequence, but has no other operations for operating on the input elements.

Definition (Comparison-based sorting function)

A *comparison-based sorting function* on S is a function $F : (S \times S \rightarrow Bool) \rightarrow S^* \rightarrow S^*$ that is

1. (parametricity) $F : \forall X.(X \times X \rightarrow Bool) \rightarrow X^* \rightarrow X^*$.
2. (sorting) if *lte* is a comparison function then $F(lte)$ is a permutation function consistent with (the ordering relation corresponding to) *lte*.

Comparison-based implies parametric

Theorem

Let F be a comparison-based sorting function on S. Let $lte : S \times S \rightarrow Bool$ be a comparison function. Then $f = F(lte)$ is a parametric permutation function and $x_1 \leq_f x_2 \Leftrightarrow lte(x_1, x_2) = true$.

Key-based (distributive) sorting functions

A key-based (distributive) sorting algorithm may operate on totally ordered keys using any operation whatsoever, including bit operations.

Definition (Key-based (distributive) sorting function)

A *key-based (distributive) sorting function* on S is a function $F : (S \to K) \to S^* \to S^*$ for some total order $(K, \leq_K)$ such that:

1. (parametricity): $F : \forall X.(X \to K) \to X^* \to X^*$.
2. (sorting): $F(key)$ is a permutation function that is consistent with the key order $((K, \leq_K), key)$.

Key-based implies parametric

Theorem

Let F be a key-based sorting function on S. Let $((K, \leq_K), key : S \to K)$ be a key order.
Then $f = F(key)$ is a parametric permutation function and $x_1 \leq_f x_2 \Leftrightarrow key(x_1) \leq_K key(x_2)$.

Nonexamples of sorting functions: Wrong type

Recall: “Sorting function” means “parametric permutation function”.

- Consider *sortBy* $: (X \times X \to Bool) \to X^* \to X^*$ as defined in the Haskell base library. This is a comparison-based sorting function, *not* a sorting function for the simple reason that it does not have the right type. *sortBy returns* a sorting function when applied to a comparison function.
- Consider a probabilistic or nondeterministic sorting algorithm, such as Quicksort with random selection of the pivot element. It does not implement a sorting function for the simple reason that it is not a *function*: the same input may be mapped to different outputs during different runs.

Nonexamples (continued): Not locally consistent

- Consider the permutation function of the Undecidability Theorem. It is not locally consistent and thus not parametric. Ergo it is not a sorting function. Even though it orders the input such that the output respects some ordering relation we cannot use it to *decide* the ordering relation.

Nonexamples (continued): Not parametric

- Consider the function $f : \mathbb{N}_0^* \rightarrow \mathbb{N}_0^*$ that first lists the even elements in its input and then the odd ones, in either case in the same order as in the input. f is a sorting function. Now modify f as follows:
-

$$\begin{aligned} f'(6\,8\,1\,5) &= 8\,6\,1\,5 \\ f'(\vec{x}) &= f(\vec{x}), \textit{otherwise} \end{aligned}$$

- f' is locally consistent, but not parametric.
- Intuition: f' inspects “satellite data”.

Stability

Definition (Stability)

A permutation function $f : S^* \to S^*$ is *stable* if it preserves the relative order of $\equiv_f$-equivalent elements in its input.

We can now add as a requirement on a *stable sorting function f* that it be a sorting function and that
Requirement 4: f must be stable.

Stability implies parametricity

Lemma (Stability implies parametricity)

Let $f : S^* \to S^*$ be a stable permutation function. Then:

1. *f is parametric.*
2. $x \leq_f y \Leftrightarrow f(xy) = xy$

Stable sorting function

Definition (Stable sorting function)

We call a function $f : S^* \to S^*$ a *stable sorting function* if

- (permutativity) it is a permutation function;
- (stability) it is stable.

Permutation functions on total orders

Corollary

Let f be a permutation function consistent with total order $(S, \leq_S)$. Then:

- *f is stable.*
- *f is parametric.*
- *f is locally consistent.*

This means: Sorting on total orders—instead of total *preorders*—is “uninteresting”.

Permutation functions and ordering relations

All permutation functions: many-many. Each permutation function is consistent with *many* ordering relations, and each ordering relation is consistent with *many* permutation functions.

Parametric permutation functions: many-one. Each parametric permutation function f is consistent with exactly *one* ordering relation R such that f is R-parametric, and each ordering relation is consistent with *many* such functions.

Stable permutation functions: one-one. Each stable permutation function f is consistent with exactly *one* ordering relation R such that f is R-stable, and each ordering relation is consistent with exactly *one* such permutation function.

Intrinsic characterization

Theorem (Local characterization of stability)

Let $f : S^* \to S^*$. The following statements are equivalent:

1. *f is a stable permutation function.*
2. *f is consistently permutative: For each sequence $x_1 \dots x_n \in S^*$ there exists $\pi \in S_{|\vec{x}|}$ such that*
   - *$f(x_1 \dots x_n) = x_{\pi(1)} \dots x_{\pi(n)}$ (permutativity);*
   - *$\forall i, j \in [1 \dots n] : f(x_i x_j) = x_i x_j \Leftrightarrow \pi^{-1}(i) \leq_\omega \pi^{-1}(j)$ (consistency).*

π^{-1} maps the *index* of an element occurrence to its *rank*. Consistency expresses that the relative order of two elements in the output of f must always be the same.

Noncharacterizations

The interesting part about the characterization is that there are numerous—at least plausible-looking— “noncharacerizations”. (Elided.)

How to provide access to an ordering relation?

Imagine we are interested in providing clients access to an ordered datatype. How should the ordering relation be offered to clients as an *operation*?

Possibilities:

- comparison function
- comparator
- sorting function (or variant such as sorting discriminator)
- ranking function (mapping to particular type with standard ordering)

Isomorphisms

Consider the following classes:

- $\mathbf{TPOrder}^{Set}$ of orders,
- $\mathbf{Ineq}^{Set}$ of comparison functions,
- $\mathbf{Comparator}^{Set}$ of comparators,
- $\mathbf{Sort}^{Set}$ of stable permutation functions.

Theorem (Order isomorphisms)

The four classes are isomorphic.

Parametric translations

Theorem

The isomorphisms mapping between inequality tests, sorting functions and comparators can be defined parametrically polymorphically:

1. $sort^{lte} : \forall X.\,(X \times X \to Bool) \to (X^* \to X^*)$;
2. $lte^{sort} : \forall^{(=)} X.\,(X^* \to X^*) \to (X \times X \to Bool)$;
3. $sort^{comp} : \forall X.\,(X \times X \to X \times X) \to (X^* \to X^*)$;
4. $comp^{sort} : \forall X.\,(X^* \to X^*) \to (X \times X \to X \times X)$;
5. $comp^{lte} : \forall X.\,(X \times X \to Bool) \to (X \times X \to X \times X)$;
6. $lte^{comp} : \forall^{(=)} X.\,(X \times X \to X \times X) \to (X \times X \to Bool)$.

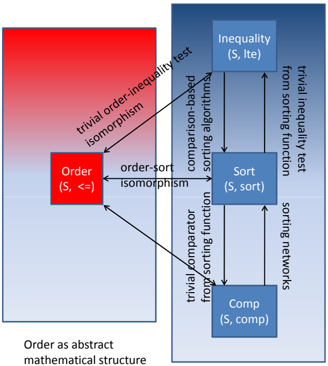
Representation independence

- Shows that comparison functions, comparators and stable sorting functions are *fully abstract* access functions for observing the ordering relation: They require and reveal nothing else about a type than its ordering relation.
- Comparison functions are definable parametrically from ranking functions $rank : T \to \mathrm{Int}$, but not conversely.
- Ranking functions compromise abstraction: There may be client code that uses the ranking function for other purposes than comparing or sorting.

Why fully abstract access matters

- Ranking function *increases* clients' ability to algorithmically exploit (representation) properties of T.
- This *decreases* T's ability to:
  - efficiently implement *other* operations,
  - ensure and exploit representation and interface invariants for correctness and (client and server) efficiency,
  - change and evolve both interface and representation of T.

Isomorphism, pictorially



Order as abstract mathematical structure

Isomorphic presentations of Order

Structure preservation

- Isomorphisms are not “structure-preserving”.
- What is the “right” structure (notion of morphism) on orders?
- Candidates: monotonic (order-preserving) and order-mapping functions
- Natural follow-up question: What is the corresponding notion of morphism on sorting structures **Sort**?
- Advance warning: Monotonic functions turn out *not* to be the “right” notion of structure for **TPOrder**!

Order-mapping and order-preserving mappings

Definition (Order-preserving, order-mapping)

Let $(S, \leq)$ and $(S', \leq')$ be orders.
We say a function $g : S \to S'$ is *order-preserving* or *monotonic* if $x \leq y \implies g(x) \leq' g(y)$ for all $x, y \in S$.
It is *order-mapping* if the implication also holds in the converse direction: $x \leq y \iff g(x) \leq' g(y)$ for all $x, y \in S$.

Sort-invariant functions

Definition (Sort-invariant function)

Let $(S, sort)$ and $(S', sort')$ be sorting structures. We say a function $g : S \to S'$ is *sort-invariant* if g commutes with *sort* and *sort'*:

$$\mathrm{Map}\, g(sort(\vec{x})) = sort'(\mathrm{Map}\, g(\vec{x})).$$

Implications

Theorem

Each order-mapping function is sort-invariant (on the induced sorting structure).
Each sort-invariant function is order-preserving (on the induced order).
The converses do not hold, however.

$$\text{order} - \text{mapping} \ \underset{\not\Longleftarrow}{\Longrightarrow} \ \text{sort} - \text{invariant} \ \underset{\not\Longleftarrow}{\Longrightarrow} \ \text{order} - \text{preserving}$$

Moral summary

- You can *first* define the notion of “order” and then, by reference to “order”, the notion of “(stable) sorting function”.
- Have shown that it is possible to *first* define the notion of “(stable) sorting function” and then, by reference to “stable sorting function”, the notion of “order”.
- The same can be done with the notion of “comparator”.
- Use intrinsic defining properties to discover when a function *cannot* be a stable sorting function for *any* order.

References

Reynolds, Types, abstraction and parametric polymorphism. Information Processing, 1983

Mitchell. Representation independence and data abstraction, POPL 1986

Wadler, Theorems for free!, FPCA 1989 (*)

Cormen, Leiserson, Rivest, Introduction to algorithms, 2d edition, 1990

Knuth, The Art of Computer Programming, volume 3: Sorting and Searching, 1998

Henglein, Intrinsically defined sorting functions. TOPPS Report D-565 (DIKU), 2007

Henglein, What is a sort function?, NWPT 2007

Henglein, What is a sorting function?, 2008, submitted to JLAP

Challenges

- Exhibit permutation function whose canonically induced ordering relation is undecidable.
- Find notion of morphism on sorting structures corresponding to total preorders with order-mapping functions.
- Figure out which algorithm was used to implement Haskell's `sortBy` function or any other comparison-parameterized sorting function.
- Passive stable sorting function checking: Observe stream of input-output pairs $(x_1, f(x_1)), \ldots, (x_i, f(x_i)), \ldots$ of a function f. Flag the first index i, if any, where it is clear that f cannot be a (stable) sorting function.