COMP2521 25T3

Sorting Algorithms (I)
Introduction to Sorting Algorithms

Sim Mautner

cs2521@cse.unsw.edu.au

sorting
properties of sorting algorithms

Motivation

- Sorting enables faster searching
  - Binary search
- Sorting provides a useful intermediate for other algorithms
  - For example, duplicate detection/removal, merging two collections

Sorting

- Sorting involves arranging a collection of items in order
  - Arrays, linked lists, files
- Items are sorted based on some property (called the **key**), using an ordering relation on that property
  - Numbers are sorted numerically
  - Strings are sorted alphabetically

Sorting

We sort arrays of `Items`, which could be:

- Simple values: `int`, `char`, `double`
- Aggregate values: strings
- Structured values: `struct`

The items are sorted based on a key, which could be:

- The entire item, if the item is a single value
- One or more fields, if the item is a struct

Sorting

Example: Each student has an ID and a name

5151515 John	5012345 Jane	3456789 Bob	5050505 Alice	5555555 John	5432109 Andrew

Sorting by ID (i.e., key is ID):

3456789 Bob	5012345 Jane	5050505 Alice	5151515 John	5432109 Andrew	5555555 John

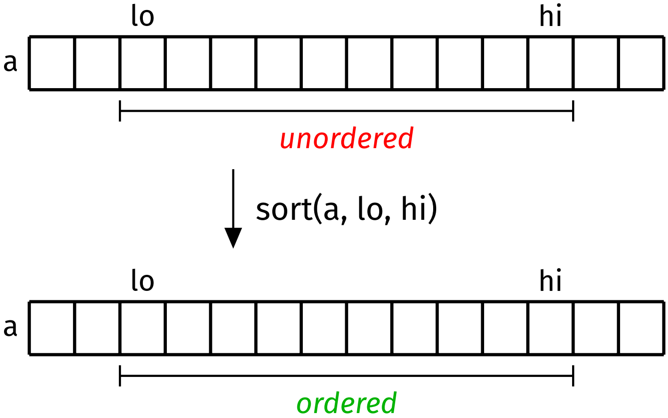
Sorting by name (i.e., key is name):

5050505 Alice	5432109 Andrew	3456789 Bob	5012345 Jane	5151515 John	5555555 John

Motivation
Sorting
Analysis
Properties
Programming

Sorting

Arrange items in array slice `a[lo..hi]` into sorted order:



To sort an entire array of size `N`, `lo == 0` and `hi == N - 1`.

Examples of Sorting Algorithms

Elementary sorting algorithms:

- Selection sort
- Bubble sort
- Insertion sort
- Shell sort

Divide-and-conquer sorting algorithms:

- Merge sort
- Quick sort

Non-comparison-based sorting algorithms:

- Radix sort
- Key-indexed counting sort

Analysis of Sorting Algorithms

Three main cases to consider for input order:

- Random order
- Sorted order
- Reverse-sorted order

When analysing sorting algorithms, we consider:

- n: the number of items ($\texttt{hi} - \texttt{lo} + 1$)
- C: the number of comparisons between items
- S: the number of times items are swapped

Properties of Sorting Algorithms

Properties:

- Stability
- Adaptability
- In-place

Motivation
Sorting
Analysis
Properties
Stability
Adaptability
In-place
Programming

Properties of Sorting Algorithms

Stability

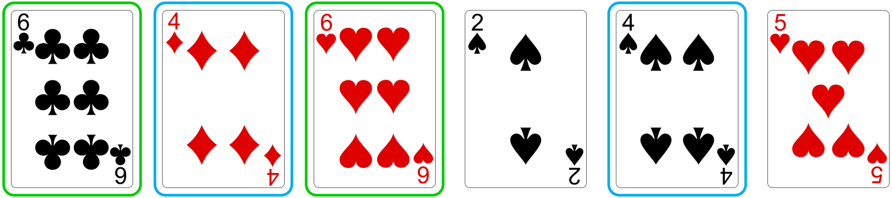
- A stable sort preserves the relative order of items with equal keys.
- **Formally:** For all pairs of items x and y where $\text{KEY}(x) \equiv \text{KEY}(y)$, if x precedes y in the original array, then x precedes y in the sorted array.

A stable sorting algorithm *always* performs a stable sort.

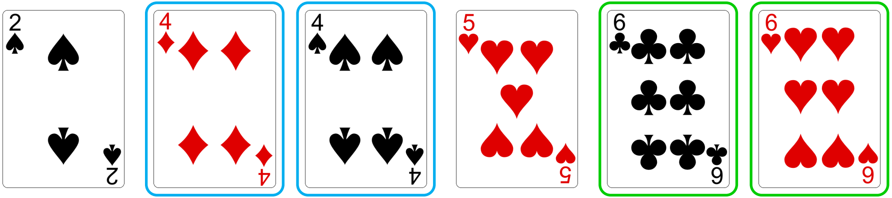
Properties of Sorting Algorithms

Stability

Example: Each card has a value and a suit



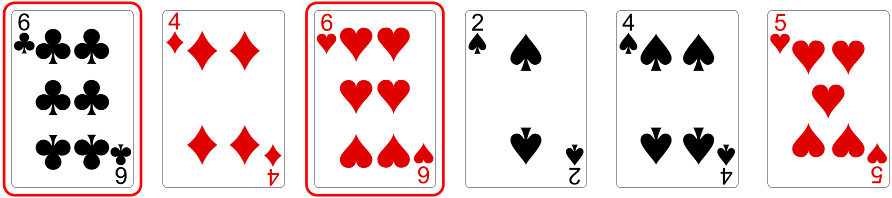
A stable sort on value:



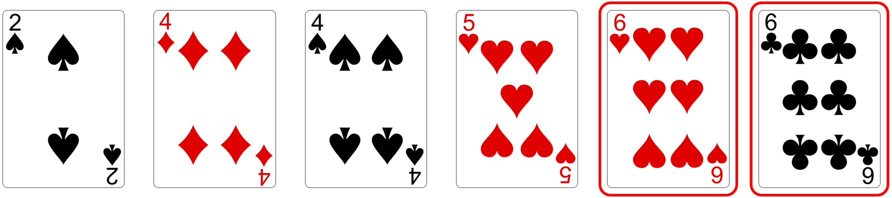
Properties of Sorting Algorithms

Stability

Example: Each card has a value and a suit



Example of an unstable sort on value:



Motivation
Sorting
Analysis
Properties
Stability
Adaptability
In-place
Programming

When is stability important?

- When sorting the same array multiple times on different keys
    - Some sorting algorithms rely on this, for example, radix sort

Properties of Sorting Algorithms

Stability

Example: Array of first names and last names

Alice Wunder	Andrew Bennett	Jake Renzella	Alice Hatter	Andrew Taylor	John Shepherd

Sort by last name:

Andrew Bennett	Alice Hatter	Jake Renzella	John Shepherd	Andrew Taylor	Alice Wunder

Then sort by first name (using stable sort):

Alice Hatter	Alice Wunder	Andrew Bennett	Andrew Taylor	Jake Renzella	John Shepherd

Motivation
Sorting
Analysis
Properties
Stability
Adaptability
In-place
Programming

Properties of Sorting Algorithms

Stability

Stability doesn't matter if...

- All items have unique keys
  - Example: Sorting students by ID
- The key is the entire item
  - Example: Sorting an array of integer values

Motivation
Sorting
Analysis
Properties
Stability
Adaptability
In-place
Programming

Properties of Sorting Algorithms

Adaptability

- An adaptive sorting algorithm takes advantage of existing order in its input
    - The nature of the algorithm allows sorted or nearly-sorted inputs to be sorted *much* quicker than other inputs

Properties of Sorting Algorithms

Adaptability

Warning!

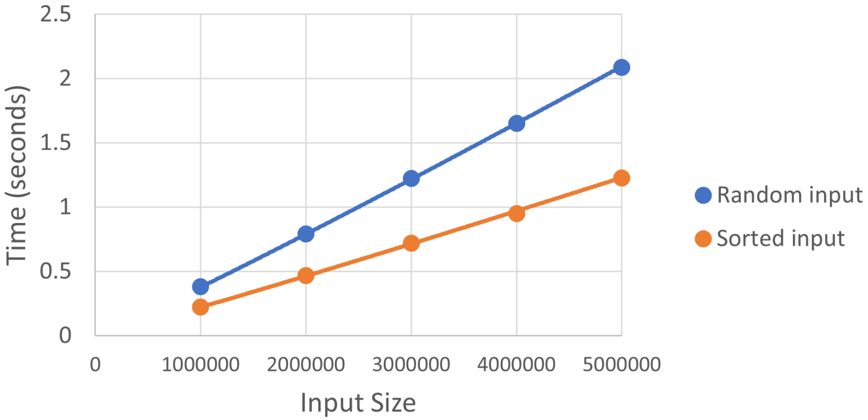
Just because a sorting algorithm
sorts sorted input faster than it sorts random/reverse-sorted input,
does not mean that it is adaptive.

Motivation
Sorting
Analysis
Properties
Stability
Adaptability
In-place
Programming

Properties of Sorting Algorithms

Adaptability

Example of data for non-adaptive sorting algorithm:

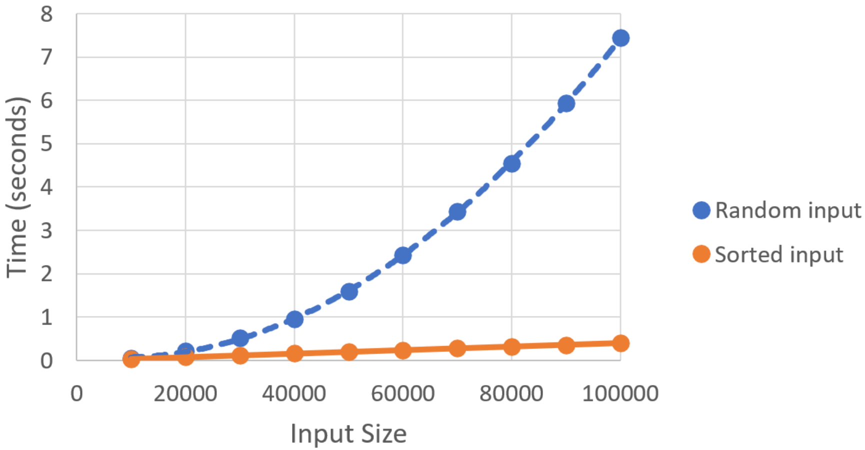


Motivation
Sorting
Analysis
Properties
Stability
Adaptability
In-place
Programming

Properties of Sorting Algorithms

Adaptability

Example of data for adaptive sorting algorithm:



Motivation
Sorting
Analysis
Properties
Stability
Adaptability
In-place
Programming

Properties of Sorting Algorithms

In-place

- An in-place sorting algorithm sorts the data within the original structure, without using temporary arrays

Implementing Sorting Algorithms

Generic sort function:

```
void sort(Item a[], int lo, int hi);
```

Helper function to swap elements at indices `i` and `j`:

```
void swap(Item a[], int i, int j);
```

Implementing Sorting Algorithms

`Item` is a `typedef`,
which is a way to give a new name to a type.

For example, if we want to sort integers:

```
typedef int Item;
```

For example, if we want to sort strings:

```
typedef char *Item;
```

Motivation
Sorting
Analysis
Properties
Programming

Implementing Sorting Algorithms

We also define macros which indicate
(1) how to extract keys from an item, and
(2) how items should be compared.

For example, when sorting integers:

```
typedef int Item;

#define key(A) (A)
#define lt(A, B) (key(A) <  key(B)) // less than
#define le(A, B) (key(A) <= key(B)) // less than or equal to
#define ge(A, B) (key(A) >= key(B)) // greater than or equal to
#define gt(A, B) (key(A) >  key(B)) // greater than
```

Motivation
Sorting
Analysis
Properties
Programming

Implementing Sorting Algorithms

When sorting structs:

```
typedef struct {
    char *name;
    char *course;
} Item;

#define key(A) (A.name)
#define lt(A, B) (strcmp(key(A), key(B)) <  0)
#define le(A, B) (strcmp(key(A), key(B)) <= 0)
#define ge(A, B) (strcmp(key(A), key(B)) >= 0)
#define gt(A, B) (strcmp(key(A), key(B)) >  0)
```