COMP2521 24T1
Sorting Algorithms (I)
Introduction to Sorting Algorithms

Kevin Luxa

cs2521@cse.unsw.edu.au

sorting
properties of sorting algorithms
• Sorting enables faster searching
  • Binary search
• Sorting arranges data in useful ways (for humans and computers)
  • For example, a list of students in a tutorial
• Sorting provides a useful intermediate for other algorithms
  • For example, duplicate detection/removal, merging two collections
• 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
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
Example: Each student has an ID and a name

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5151515</td>
<td>John</td>
</tr>
<tr>
<td>5012345</td>
<td>Jane</td>
</tr>
<tr>
<td>3456789</td>
<td>Bob</td>
</tr>
<tr>
<td>5050505</td>
<td>Alice</td>
</tr>
<tr>
<td>5555555</td>
<td>John</td>
</tr>
<tr>
<td>5432109</td>
<td>Andrew</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3456789</td>
<td>Bob</td>
</tr>
<tr>
<td>5012345</td>
<td>Jane</td>
</tr>
<tr>
<td>5050505</td>
<td>Alice</td>
</tr>
<tr>
<td>5151515</td>
<td>John</td>
</tr>
<tr>
<td>5432109</td>
<td>Andrew</td>
</tr>
<tr>
<td>5555555</td>
<td>John</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5050505</td>
<td>Alice</td>
</tr>
<tr>
<td>5432109</td>
<td>Andrew</td>
</tr>
<tr>
<td>3456789</td>
<td>Bob</td>
</tr>
<tr>
<td>5012345</td>
<td>Jane</td>
</tr>
<tr>
<td>5151515</td>
<td>John</td>
</tr>
<tr>
<td>5555555</td>
<td>John</td>
</tr>
</tbody>
</table>
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 $(hi - lo + 1)$
- $C$: the number of comparisons between items
- $S$: the number of times items are swapped
Properties of Sorting Algorithms

Properties:
- Stability
- Adaptability
- In-place
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.
Motivation
Sorting
Analysis
Properties
Stability
Adaptability
In-place
Programming

Properties of Sorting Algorithms

Stability

Example: Each card has a value and a suit

A stable sort on value:
Example: Each card has a value and a suit

Example of an unstable sort on value:
When is stability important?

- When sorting the same array multiple times on different keys
- Some sorting algorithms rely on this, for example, radix sort
Example: Array of first names and last names

<table>
<thead>
<tr>
<th></th>
<th>Alice Wunder</th>
<th>Andrew Bennett</th>
<th>Jake Renzella</th>
<th>Alice Hatter</th>
<th>Andrew Taylor</th>
<th>John Shepherd</th>
</tr>
</thead>
</table>

Sort by last name:

<table>
<thead>
<tr>
<th></th>
<th>Andrew Bennett</th>
<th>Alice Hatter</th>
<th>Jake Renzella</th>
<th>John Shepherd</th>
<th>Andrew Taylor</th>
<th>Alice Wunder</th>
</tr>
</thead>
</table>

Then sort by first name (using stable sort):

|     | Alice Hatter | Alice Wunder | Andrew Bennett | Andrew Taylor | Jake Renzella | John Shepherd |
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
• 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
Warning!

Just because a sorting algorithm sorts sorted input faster than it sorts random input, *does not necessarily mean* that it is adaptive.
Example of data for non-adaptive sorting algorithm:

![Graph showing time vs. input size for random and sorted inputs](image-url)
Example of data for adaptive sorting algorithm:
• 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);
```
Item is a typedef, which is a way to give a new name to a type.

For example, if we want to sort integers:

```c
typedef int Item;
```

For example, if we want to sort strings:

```c
typedef char *Item;
```
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:

```c
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
```
When sorting structs:

```c
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)
```
https://forms.office.com/r/5c0fb4tvMb