COMP2521 23T3

Sorting Algorithms (I)

Kevin Luxa

`cs2521@cse.unsw.edu.au`

properties of sorting algorithms
elementary sorting algorithms

Motivation

- 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

- 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`
- Complex 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):

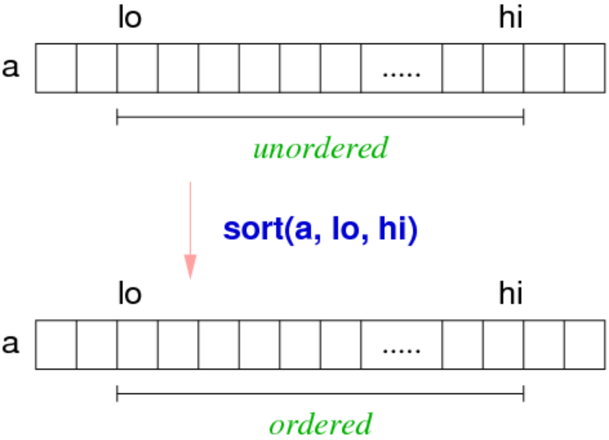
3456789 Bob	5012345 Jane	5050505 Alice	5151515 John	5432109 Andrew	5555555 John

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

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

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`.

Sorting

(More) Formally

Pre-conditions:
array `a[N]` of `Items`
`lo`, `hi` are valid indices on `a`
(roughly, $0 \leq \texttt{lo} < \texttt{hi} \leq \texttt{N} - 1$)

Post-conditions:
array `a[lo..hi]` contains the same values as before the sort
$\texttt{a}[\texttt{lo}] \leq \texttt{a}[\texttt{lo}+1] \leq \texttt{a}[\texttt{lo}+2] \leq \cdots \leq \texttt{a}[\texttt{hi}]$

Properties of Sorting Algorithms

Properties:

- Stability
- Adaptability
- In-place

Properties of Sorting Algorithms

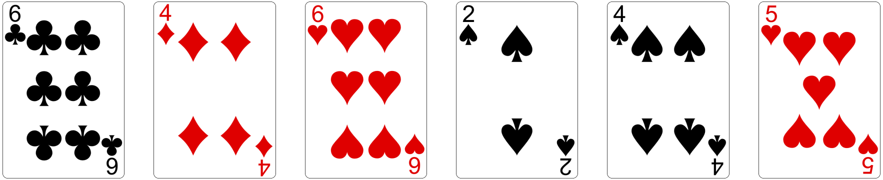
Stability

- 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.

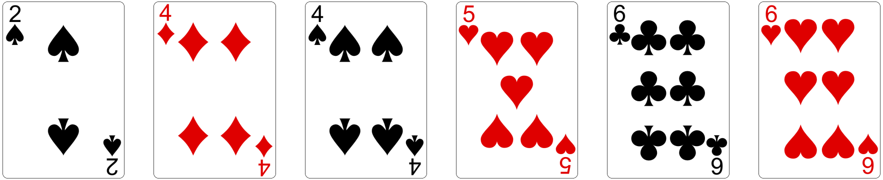
Properties of Sorting Algorithms

Stability

Example: Each card has a value and a suit



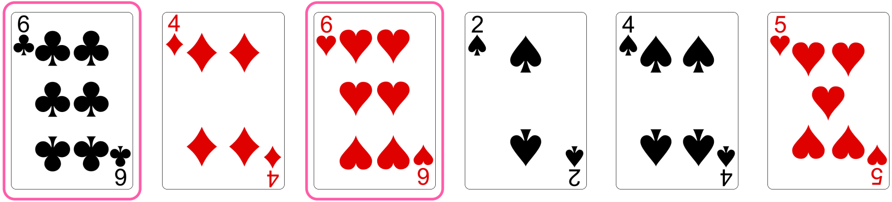
A stable sort on value:



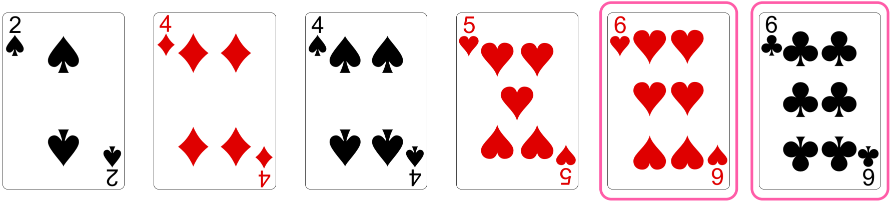
Properties of Sorting Algorithms

Stability

Example: Each card has a value and a suit



Example of an unstable sort on value:



Properties of Sorting Algorithms

Stability

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

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

Properties of Sorting Algorithms

Adaptability

- An adaptive sorting algorithm takes advantage of existing order in its input
    - Time complexity of an adaptive sorting algorithm will be better for sorted or nearly-sorted inputs
- Can be a useful property, depending on whether nearly sorted inputs are common

Properties of Sorting Algorithms

In-place

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

A Concrete Framework

```
// we deal with generic `Item's
typedef int Item;

// abstractions to hide details of items
#define key(A) (A)
#define lt(A, B) (key(A) <  key(B))
#define le(A, B) (key(A) <= key(B))
#define ge(A, B) (key(A) >= key(B))
#define gt(A, B) (key(A) >  key(B))

// Sort a slice of an array of Items
void sort(Item a[], int lo, int hi);
```

A Concrete Framework

This framework can be adapted by...
defining a different data structure for `Item`;
defining a method for extracting sort keys;
defining a different ordering (`less`);
defining a different swap method for different `Item`

```
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)
```

Analysis of Sorting Algorithms

In analysing sorting algorithms:

- 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

(We usually aim to minimise C and S.)

Cases to consider for input order:

- random order: `Items` in `a[lo..hi]` have no ordering
- sorted order: $\texttt{a}[\texttt{lo}] \leq \texttt{a}[\texttt{lo} + 1] \leq \cdots \leq \texttt{a}[\texttt{hi}]$
- reverse-sorted order: $\texttt{a}[\texttt{lo}] \geq \texttt{a}[\texttt{lo} + 1] \geq \cdots \geq \texttt{a}[\texttt{hi}]$

Examples of Sorting Algorithms

Elementary sorting algorithms:

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

More efficient sorting algorithms:

- Merge sort
- Quick sort

Non-comparison-based sorting algorithms:

- Radix sort
- Key-indexed counting sort

Selection Sort

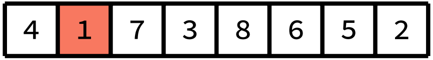
Method:

- Find the smallest element, swap it with the first element
- Find the second-smallest element, swap it with the second element
- ...
- Find the second-largest element, swap it with the second-last element

Each iteration improves the “sortedness” of the array by one element

Selection Sort

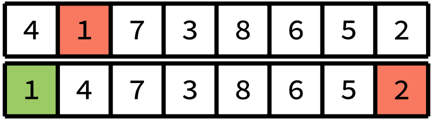
Example



4	1	7	3	8	6	5	2

Selection Sort

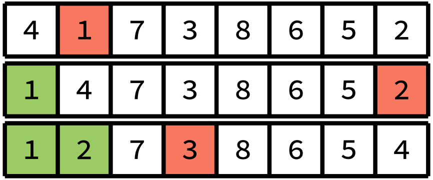
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2

Selection Sort

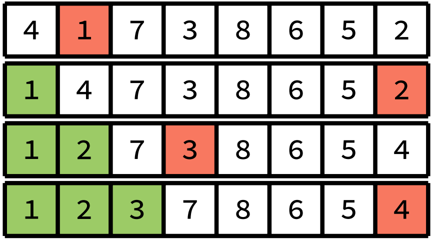
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	2	7	3	8	6	5	4

Selection Sort

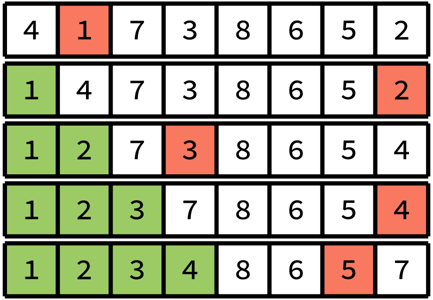
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	2	7	3	8	6	5	4
1	2	3	7	8	6	5	4

Selection Sort

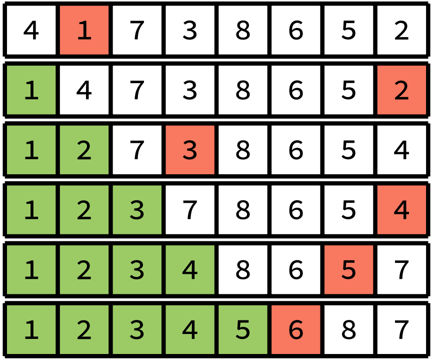
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	2	7	3	8	6	5	4
1	2	3	7	8	6	5	4
1	2	3	4	8	6	5	7

Selection Sort

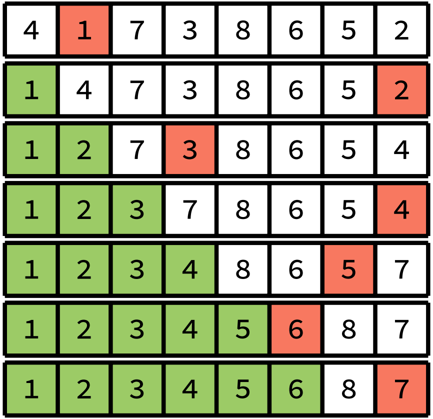
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	2	7	3	8	6	5	4
1	2	3	7	8	6	5	4
1	2	3	4	8	6	5	7
1	2	3	4	5	6	8	7

Selection Sort

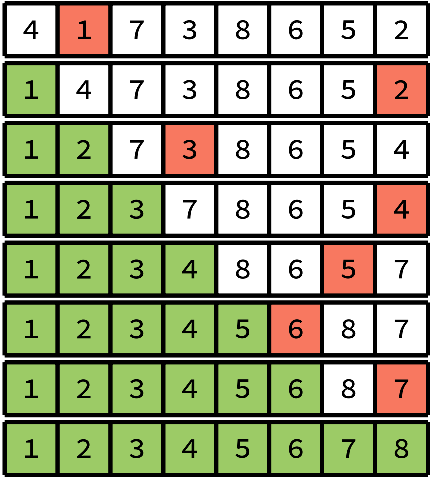
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	2	7	3	8	6	5	4
1	2	3	7	8	6	5	4
1	2	3	4	8	6	5	7
1	2	3	4	5	6	8	7
1	2	3	4	5	6	8	7

Selection Sort

Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	2	7	3	8	6	5	4
1	2	3	7	8	6	5	4
1	2	3	4	8	6	5	7
1	2	3	4	5	6	8	7
1	2	3	4	5	6	8	7
1	2	3	4	5	6	7	8

Selection Sort

C Implementation

```c
void selectionSort(Item items[], int lo, int hi) {
    for (int i = lo; i < hi; i++) {
        int min = i;
        for (int j = i + 1; j <= hi; j++) {
            if (lt(items[j], items[min])) {
                min = j;
            }
        }
        swap(items, i, min);
    }
}
```

Selection Sort

Analysis

Cost analysis:

- In the first iteration, $n-1$ comparisons, 1 swap
- In the second iteration, $n-2$ comparisons, 1 swap
- ...
- In the final iteration, 1 comparison, 1 swap
- $C = (n-1) + (n-2) + \ldots + 1 = \frac{1}{2}n(n-1) \Rightarrow O(n^2)$
- $S = n - 1$

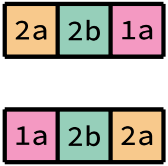
Cost is the same, regardless of the sortedness of the original array.

Selection Sort

Properties

Selection sort is unstable

- Due to long-range swaps
- Example:



2a	2b	1a

1a	2b	2a

Selection Sort

Properties

Unstable
Due to long-range swaps

Non-adaptive
Performs same steps, regardless of sortedness of original array

In-place
Sorting is done within original array; does not use temporary arrays

Bubble Sort

Method:

- Make multiple passes from left (`lo`) to right
- On each pass, swap any out-of-order adjacent pairs
- Elements “bubble up” until they meet a larger element
- Stop if there are no swaps during a pass
    - This means the array is sorted

Bubble Sort

Example

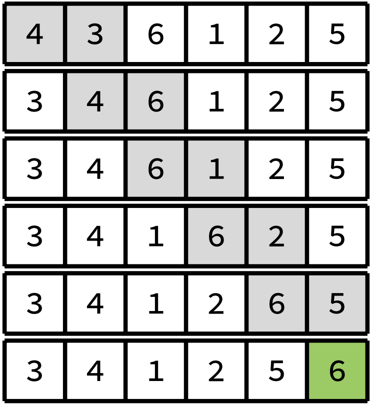
Example

4	3	6	1	2	5

Bubble Sort

Example

First pass

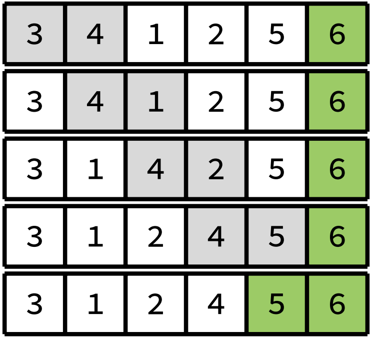


4	3	6	1	2	5
3	4	6	1	2	5
3	4	6	1	2	5
3	4	1	6	2	5
3	4	1	2	6	5
3	4	1	2	5	6

Bubble Sort

Example

Second pass

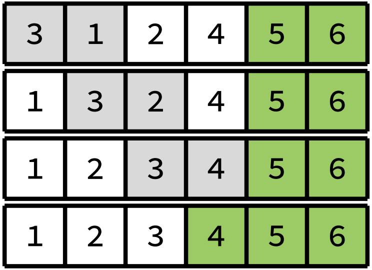


3	4	1	2	5	6
3	4	1	2	5	6
3	1	4	2	5	6
3	1	2	4	5	6
3	1	2	4	5	6

Bubble Sort

Example

Third pass

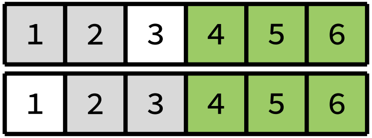


3	1	2	4	5	6
1	3	2	4	5	6
1	2	3	4	5	6
1	2	3	4	5	6

Bubble Sort

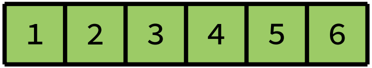
Example

Fourth pass



1	2	3	4	5	6
1	2	3	4	5	6

No swaps made; stop



1	2	3	4	5	6

Bubble Sort

C Implementation

```c
void bubbleSort(Item items[], int lo, int hi) {
    for (int i = hi; i > lo; i--) {
        bool swapped = false;
        for (int j = lo; j < i; j++) {
            if (gt(items[j], items[j + 1])) {
                swap(items, j, j + 1);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```

Bubble Sort

Analysis

Best case: Array is sorted

- Only a single pass required
- $n - 1$ comparisons, no swaps
- Best-case time complexity: $O(n)$

1	2	3	4	5	6

Bubble Sort

Analysis

Worst case: Array is reverse-sorted

- $n - 1$ passes required
  - First pass: $n - 1$ comparisons
  - Second pass: $n - 2$ comparisons
  - ...
  - Final pass: 1 comparison
- Total comparisons: $(n-1) + (n-2) + \ldots + 1 = \frac{1}{2}n(n-1)$
- Every comparison leads to a swap $\Rightarrow$ $\frac{1}{2}n(n-1)$ swaps
- Worst-case time complexity: $O(n^2)$

6	5	4	3	2	1

Bubble Sort

Analysis

Average-case time complexity: $O(n^2)$

- Can show empirically by generating random sequences and sorting them

Bubble Sort

Properties

Stable

Comparisons are between adjacent elements only
Elements are only swapped if out of order

Adaptive

Bubble sort is $O(n^2)$ on average, $O(n)$ if input array is sorted

In-place

Sorting is done within original array; does not use temporary arrays

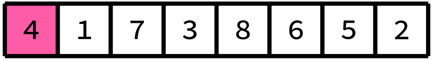
Insertion Sort

Method:

- Take first element and treat as sorted array (of length 1)
- Take next element and insert into sorted part of array so that order is preserved
  - This increases the length of the sorted part by one
- Repeat for remaining elements

Insertion Sort

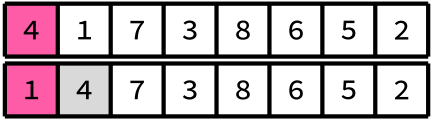
Example



4	1	7	3	8	6	5	2

Insertion Sort

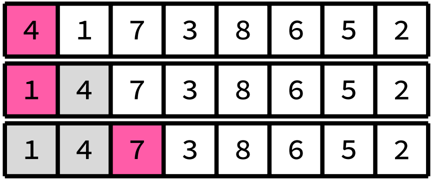
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2

Insertion Sort

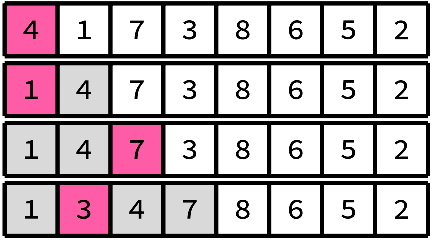
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	4	7	3	8	6	5	2

Insertion Sort

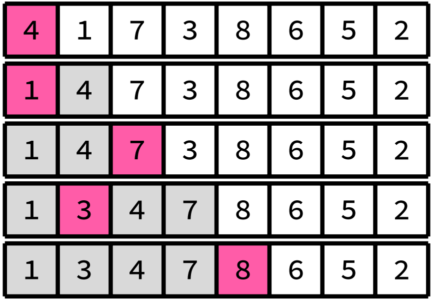
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	3	4	7	8	6	5	2

Insertion Sort

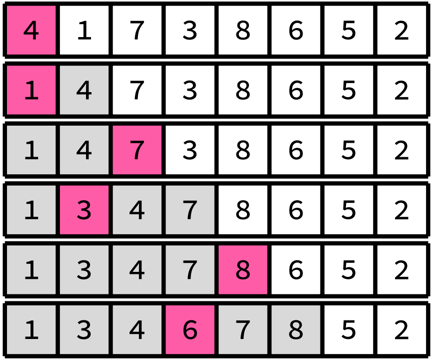
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	7	8	6	5	2

Insertion Sort

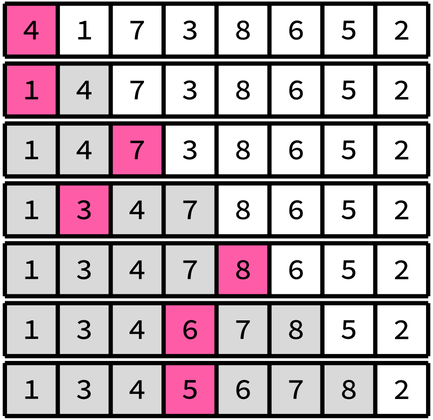
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	6	7	8	5	2

Insertion Sort

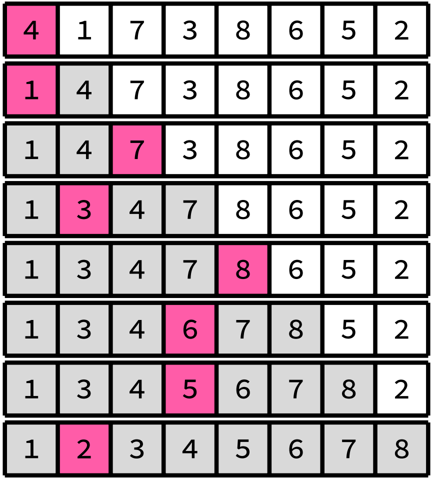
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	6	7	8	5	2
1	3	4	5	6	7	8	2

Insertion Sort

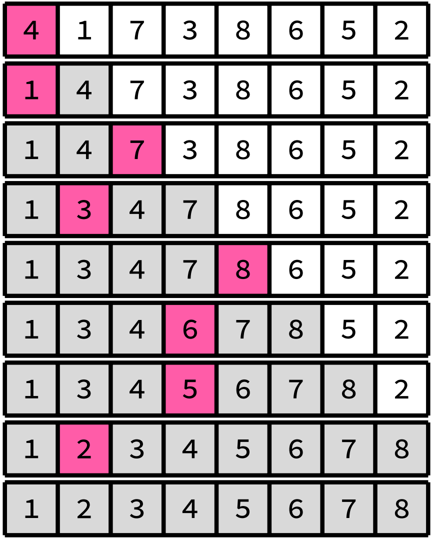
Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	6	7	8	5	2
1	3	4	5	6	7	8	2
1	2	3	4	5	6	7	8

Insertion Sort

Example



4	1	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	4	7	3	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	7	8	6	5	2
1	3	4	6	7	8	5	2
1	3	4	5	6	7	8	2
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

Insertion Sort

C Implementation

```
void insertionSort(Item items[], int lo, int hi) {
    for (int i = lo + 1; i <= hi; i++) {
        Item item = items[i];
        int j = i;
        for (; j > lo && lt(item, items[j - 1]); j--) {
            items[j] = items[j - 1];
        }
        items[j] = item;
    }
}
```

Insertion Sort

Analysis

Best case: Array is sorted

- Inserting each element requires one comparison
- $n - 1$ comparisons
- Best-case time complexity: $O(n)$

1	2	3	4	5	6	7	8

Insertion Sort

Analysis

Worst case: Array is reverse-sorted

- Inserting i-th element requires i comparisons
  - Inserting index 1 element requires 1 comparison
  - Inserting index 2 element requires 2 comparisons
  - ...
- Total comparisons: $1 + 2 + \ldots + (n - 1) = \frac{1}{2}n(n - 1)$
- Worst-case time complexity: $O(n^2)$

8	7	6	5	4	3	2	1

Insertion Sort

Analysis

Average-case time complexity: $O(n^2)$

- Can show empirically by generating random sequences and sorting them

Insertion Sort

Properties

Stable

Elements are always inserted to the right of any equal elements

Adaptive

Insertion sort is $O(n^2)$ on average, $O(n)$ if input array is sorted

In-place

Sorting is done within original array; does not use temporary arrays

Shell Sort

Bubble sort and insertion sort
really only consider *adjacent* elements.

If we make longer-distance exchanges,
can we be more efficient?

What if we consider elements that are some distance apart?

Shell sort, invented by Donald Shell

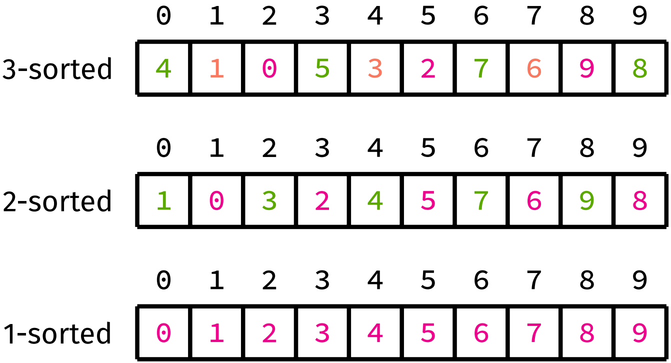
Shell Sort

Idea:

- An array is h-sorted if taking every h-th element yields a sorted array
- An h-sorted array is made up of $\frac{n}{h}$ interleaved sorted arrays
- Shell sort: h-sort the array for progressively smaller h, ending with $h = 1$

Shell Sort

Example of h-sorted arrays:



	0	1	2	3	4	5	6	7	8	9
3-sorted	4	1	0	5	3	2	7	6	9	8

	0	1	2	3	4	5	6	7	8	9
2-sorted	1	0	3	2	4	5	7	6	9	8

	0	1	2	3	4	5	6	7	8	9
1-sorted	0	1	2	3	4	5	6	7	8	9

Shell Sort

Example

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
unsorted	4	1	7	3	8	6	5	2

Shell Sort

Example

	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`	`[6]`	`[7]`
unsorted	4	1	7	3	8	6	5	2
$h = 3$ passes	3			4			5	
		1			2			8
			6			7		
3-sorted	3	1	6	4	2	7	5	8

Shell Sort

Example

	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`	`[6]`	`[7]`
unsorted	4	1	7	3	8	6	5	2
$h = 3$ passes	3			4			5	
		1			2			8
			6			7		
3-sorted	3	1	6	4	2	7	5	8
$h = 2$ passes	2		3		5		6	
		1		4		7		8
2-sorted	2	1	3	4	5	7	6	8

Shell Sort

Example

	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`[5]`	`[6]`	`[7]`
unsorted	4	1	7	3	8	6	5	2
$h = 3$ passes	3			4			5	
		1			2			8
			6			7		
3-sorted	3	1	6	4	2	7	5	8
$h = 2$ passes	2		3		5		6	
		1		4		7		8
2-sorted	2	1	3	4	5	7	6	8
$h = 1$ pass	1	2	3	4	5	6	7	8

Shell Sort

C Implementation

```
void shellSort(Item items[], int lo, int hi) {
    int size = hi - lo + 1;
    // find appropriate h-value to start with
    int h;
    for (h = 1; h <= (size - 1) / 9; h = (3 * h) + 1);

    for (; h > 0; h /= 3) {
        for (int i = lo + h; i <= hi; i++) {
            Item item = items[i];
            int j = i;
            for (; j >= lo + h && lt(item, items[j - h]); j -= h) {
                items[j] = items[j - h];
            }
            items[j] = item;
        }
    }
}
```

Shell Sort

Analysis

- Efficiency of shell sort depends on the h-sequence
- Effective h-sequences have been determined empirically
- Many h-sequences have been found to be $O(n^{\frac{3}{2}})$
  - For example: 1, 4, 13, 40, 121, 364, 1093, ...
    - $h_{i+1} = 3h_i + 1$
- Some h-sequences have been found to be $O(n^{\frac{4}{3}})$
  - For example: 1, 8, 23, 77, 281, 1073, 4193, ...

Shell Sort

Properties

Unstable
Due to long-range swaps

Adaptive
Shell sort applies a generalisation of insertion sort
(which is adaptive)

In-place
Sorting is done within original array; does not use temporary arrays

Summary of Elementary Sorts

	Time complexity			Properties	
	Best	Average	Worst	Stable	Adaptive
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	No
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Shell sort	depends	depends	depends	No	Yes

Aside: Sorting Linked Lists

Selection sort:

- Let $L =$ original list, $S =$ sorted list (initially empty)
- Repeat the following until L is empty:
  - Find the node V containing the largest value in L, and unlink it
  - Insert V at the front of S

Bubble sort:

- Traverse the list, comparing adjacent values
  - If value in current node is greater than value in next node, swap values
- Repeat the above until no swaps required in one traversal

Insertion sort:

- Let $L =$ original list, $S =$ sorted list (initially empty)
- For each node in L:
  - Insert the node into S in order

Aside: Sorting Linked Lists

Shell sort:

- Difficult to implement efficiently
- Can’t access specific index in constant time
    - Have to traverse from the beginning

Feedback

`https://forms.office.com/r/aPF09YHZ3X`