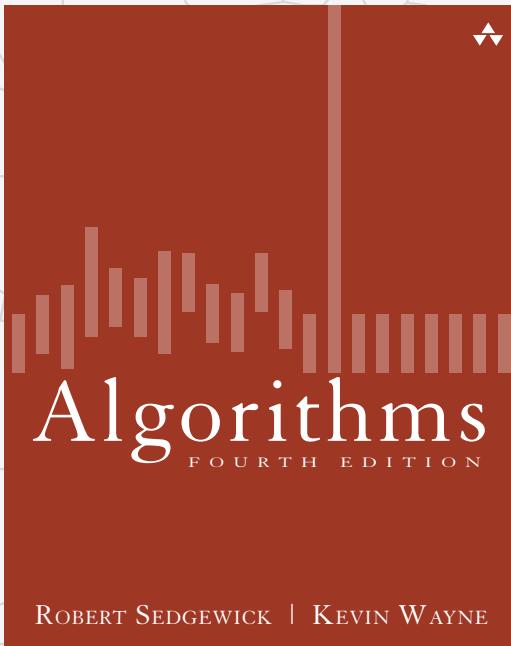


# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

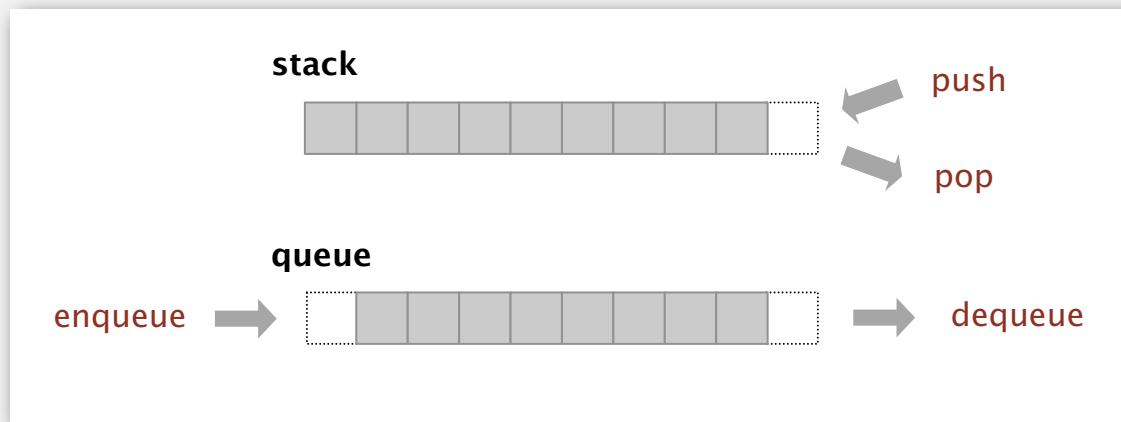
- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Stacks and queues

---

## Fundamental data types.

- Value: collection of objects.
- Operations: **insert**, **remove**, **iterate**, test if empty.
- Intent is clear when we insert.
- Which item do we remove?



Stack. Examine the item most recently added. ← LIFO = "last in first out"

Queue. Examine the item least recently added. ← FIFO = "first in first out"

# Client, implementation, interface

---

Separate interface and implementation.

Ex: stack, queue, bag, priority queue, symbol table, union-find, ....

Benefits.

- Client can't know details of implementation ⇒ client has many implementation from which to choose.
- Implementation can't know details of client needs ⇒ many clients can re-use the same implementation.
- Design: creates modular, reusable libraries.
- Performance: use optimized implementation where it matters.

**Client:** program using operations defined in interface.

**Implementation:** actual code implementing operations.

**Interface:** description of data type, basic operations.

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Stack API

---

Warmup API. Stack of strings data type.

```
public class StackOfStrings
```

StackOfStrings()	<i>create an empty stack</i>
------------------	------------------------------

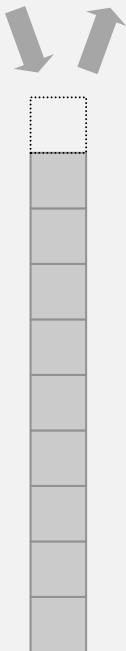
void push(String item)	<i>insert a new string onto stack</i>
------------------------	---------------------------------------

String pop()	<i>remove and return the string most recently added</i>
--------------	---

boolean isEmpty()	<i>is the stack empty?</i>
-------------------	----------------------------

int size()	<i>number of strings on the stack</i>
------------	---------------------------------------

push    pop



Warmup client. Reverse sequence of strings from standard input.

## Stack test client

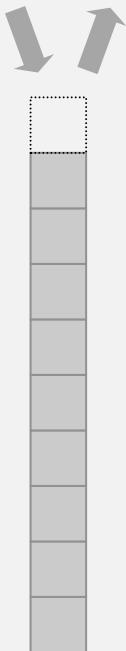
---

Read strings from standard input.

- If string equals "-", pop string from stack and print.
- Otherwise, push string onto stack.

push    pop

```
public static void main(String[] args)
{
    StackOfStrings stack = new StackOfStrings();
    while (!StdIn.isEmpty())
    {
        String s = StdIn.readString();
        if (s.equals("-")) StdOut.print(stack.pop());
        else             stack.push(s);
    }
}
```

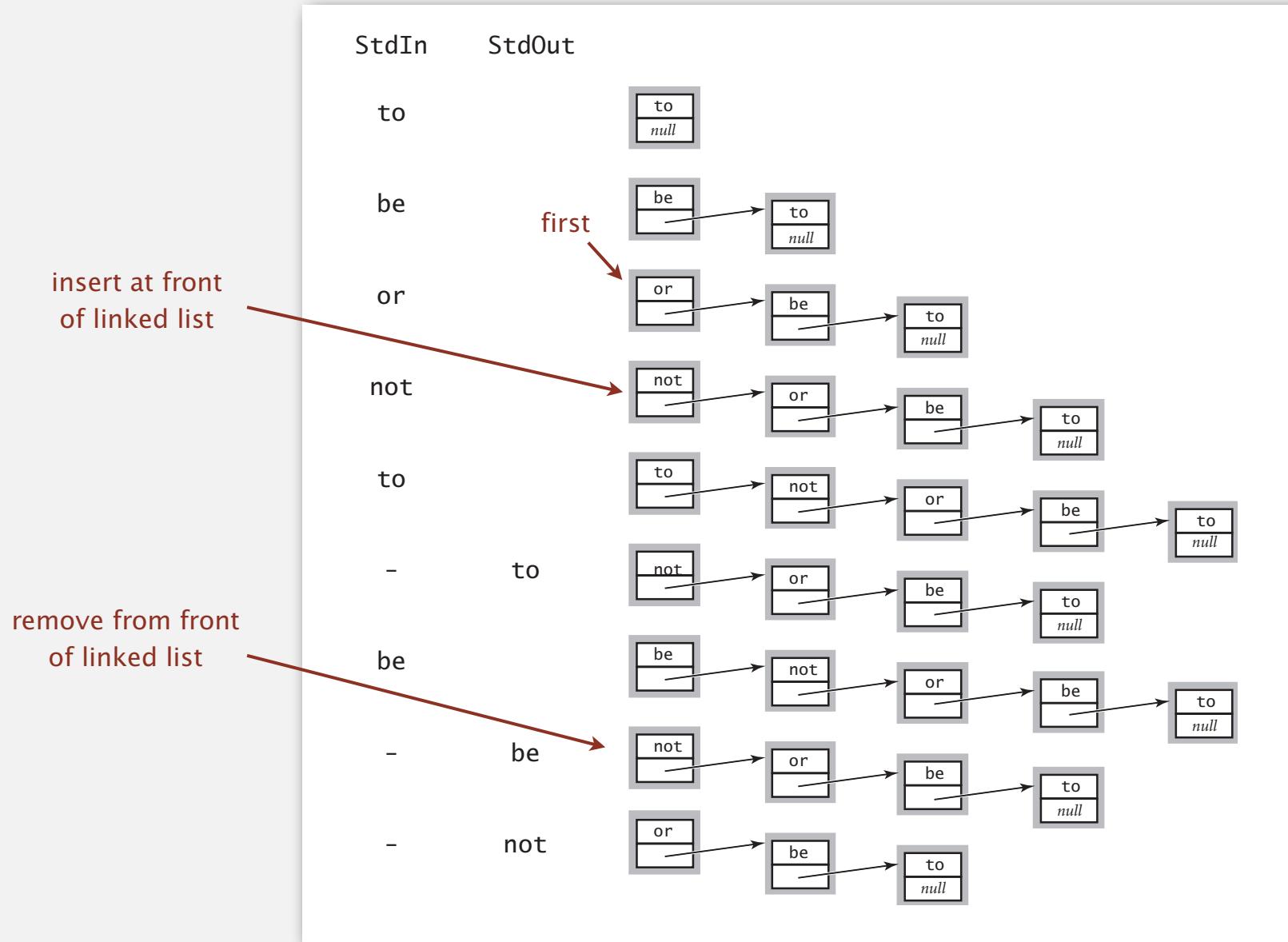


```
% more tobe.txt
to be or not to - be - - that - - - is

% java StackOfStrings < tobe.txt
to be not that or be
```

# Stack: linked-list representation

Maintain pointer to first node in a linked list; insert/remove from front.



# Stack pop: linked-list implementation

inner class

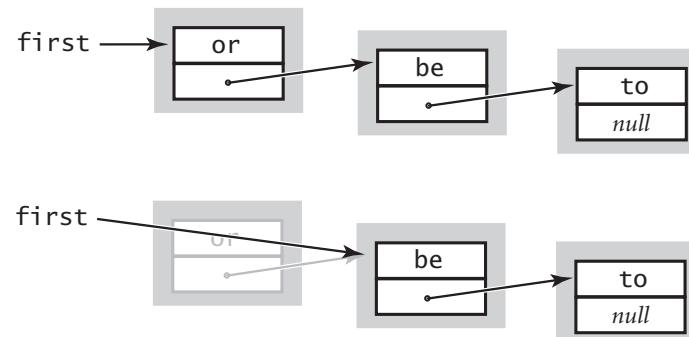
```
private class Node  
{  
    String item;  
    Node next;  
}
```

save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



return saved item

```
return item;
```

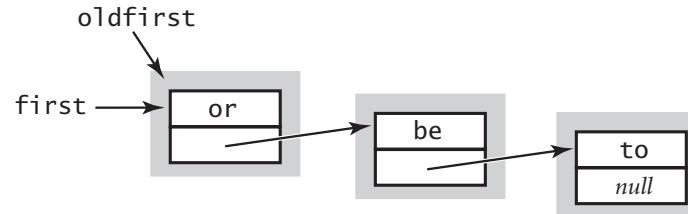
# Stack push: linked-list implementation

inner class

```
private class Node  
{  
    String item;  
    Node next;  
}
```

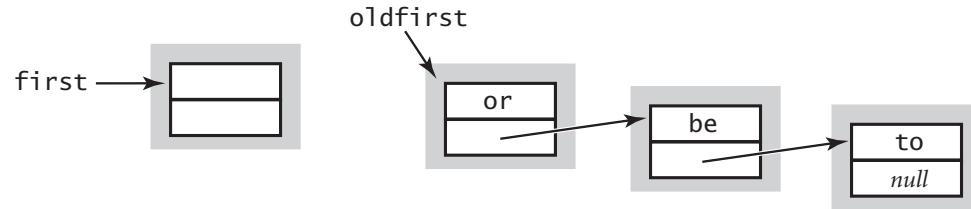
save a link to the list

```
Node oldfirst = first;
```



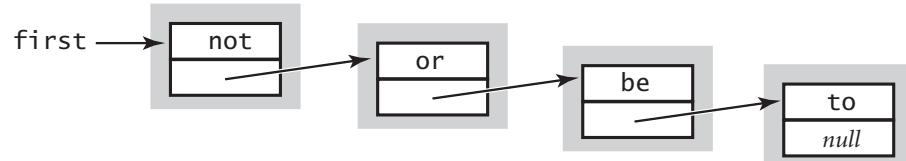
create a new node for the beginning

```
first = new Node();
```



set the instance variables in the new node

```
first.item = "not";  
first.next = oldfirst;
```



# Stack: linked-list implementation in Java

```
public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

    public boolean isEmpty()
    {   return first == null;   }

    public void push(String item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

private inner class  
(access modifiers don't matter)



# Stack: linked-list implementation performance

---

Proposition. Every operation takes constant time in the worst case.

Proposition. A stack with  $N$  items uses  $\sim 40N$  bytes.

```
inner class  
private class Node  
{  
    String item;  
    Node next;  
}
```



Remark. This accounts for the memory for the stack  
(but not the memory for strings themselves, which the client owns).

# Stack: array implementation

---

## Array implementation of a stack.

- Use array  $s[]$  to store  $N$  items on stack.
- $\text{push}()$ : add new item at  $s[N]$ .
- $\text{pop}()$ : remove item from  $s[N-1]$ .

$s[]$	to	be	or	not	to	be	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
	0	1	2	3	4	5	6	7	8	9
							$N$			$\text{capacity} = 10$

Defect. Stack overflows when  $N$  exceeds capacity. [stay tuned]

# Stack: array implementation

```
public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public FixedCapacityStackOfStrings(int capacity)
    {   s = new String[capacity]; }
```

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

```
public void push(String item)
{   s[N++] = item; }
```

use to index into array;  
then increment N

```
public String pop()
{   return s[--N]; }
```

```
}
```

a cheat  
(stay tuned)



decrement N;  
then use to index into array

# Stack considerations

---

## Overflow and underflow.

- Underflow: throw exception if pop from an empty stack.
- Overflow: use resizing array for array implementation. [stay tuned]

Null items. We allow null items to be inserted.

Loitering. Holding a reference to an object when it is no longer needed.

```
public String pop()
{   return s[--N]; }
```

loitering

```
public String pop()
{
    String item = s[--N];
    s[N] = null;
    return item;
}
```

this version avoids "loitering":  
garbage collector can reclaim memory  
only if no outstanding references

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

## Stack: resizing-array implementation

---

**Problem.** Requiring client to provide capacity does not implement API!

**Q.** How to grow and shrink array?

**First try.**

- `push()`: increase size of array `s[]` by 1.
- `pop()`: decrease size of array `s[]` by 1.

**Too expensive.**

- Need to copy all items to a new array.
- Inserting first  $N$  items takes time proportional to  $1 + 2 + \dots + N \sim N^2/2$ .

  
infeasible for large  $N$

**Challenge.** Ensure that array resizing happens infrequently.

# Stack: resizing-array implementation

Q. How to grow array?

A. If array is full, create a new array of **twice** the size, and copy items.

"repeated doubling"

```
public ResizingArrayStackOfStrings()
{   s = new String[1]; }

public void push(String item)
{
    if (N == s.length) resize(2 * s.length);
    s[N++] = item;
}

private void resize(int capacity)
{
    String[] copy = new String[capacity];
    for (int i = 0; i < N; i++)
        copy[i] = s[i];
    s = copy;
}
```

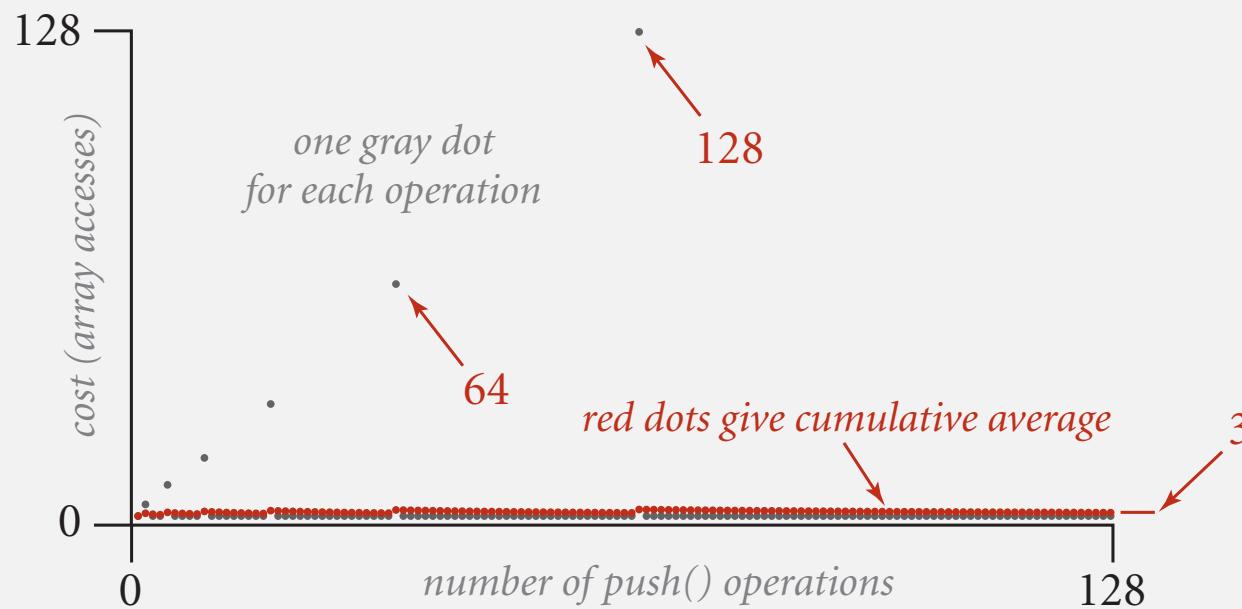
see next slide

Consequence. Inserting first  $N$  items takes time proportional to  $N$  (not  $N^2$ ).

# Stack: amortized cost of adding to a stack

Cost of inserting first  $N$  items.  $N + (2 + 4 + 8 + \dots + N) \sim 3N.$

↑  
1 array access  
per push                           ↑  
k array accesses to double to size k  
(ignoring cost to create new array)



# Stack: resizing-array implementation

---

Q. How to shrink array?

First try.

- `push()`: double size of array `s[]` when array is full.
- `pop()`: halve size of array `s[]` when array is **one-half full**.

Too expensive in worst case.

- Consider push-pop-push-pop-... sequence when array is full.
- Each operation takes time proportional to  $N$ .

$N = 5$	to	be	or	not	to	null	null	null
---------	----	----	----	-----	----	------	------	------

$N = 4$	to	be	or	not
---------	----	----	----	-----

$N = 5$	to	be	or	not	to	null	null	null
---------	----	----	----	-----	----	------	------	------

$N = 4$	to	be	or	not
---------	----	----	----	-----

# Stack: resizing-array implementation

---

Q. How to shrink array?

Efficient solution.

- `push()`: double size of array `s[]` when array is full.
- `pop()`: halve size of array `s[]` when array is **one-quarter full**.

```
public String pop()
{
    String item = s[--N];
    s[N] = null;
    if (N > 0 && N == s.length/4) resize(s.length/2);
    return item;
}
```

Invariant. Array is between 25% and 100% full.

# Stack: resizing-array implementation trace

push()	pop()	N	a.length	a[]							
				0	1	2	3	4	5	6	7
			0	1	null						
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or		null			
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	null	null	null
-	to	4	8	to	be	or	not	null	null	null	null
be		5	8	to	be	or	not	be	null	null	null
-	be	4	8	to	be	or	not	null	null	null	null
-	not	3	8	to	be	or	null	null	null	null	null
that		4	8	to	be	or	that	null	null	null	null
-	that	3	8	to	be	or	null	null	null	null	null
-	or	2	4	to	be	null	null				
-	be	1	2	to	null						
is		2		to	is						

Trace of array resizing during a sequence of push() and pop() operations

## Stack resizing-array implementation: performance

**Amortized analysis.** Average running time per operation over a worst-case sequence of operations.

**Proposition.** Starting from an empty stack, any sequence of  $M$  push and pop operations takes time proportional to  $M$ .

	best	worst	amortized
construct	1	1	1
push	1	$N$	1
pop	1	$N$	1
size	1	1	1

**order of growth of running time  
for resizing stack with  $N$  items**

doubling and halving operations

## Stack resizing-array implementation: memory usage

**Proposition.** Uses between  $\sim 8N$  and  $\sim 32N$  bytes to represent a stack with  $N$  items.

- $\sim 8N$  when full.
- $\sim 32N$  when one-quarter full.

```
public class ResizingArrayStackOfStrings
{
    private String[] s;
    private int N = 0;
    ...
}
```

8 bytes (reference to array)  
24 bytes (array overhead)  
8 bytes x array size  
4 bytes (int)  
4 bytes (padding)  
4 bytes (padding)

**Remark.** This accounts for the memory for the stack  
(but not the memory for strings themselves, which the client owns).

## Stack implementations: resizing array vs. linked list

---

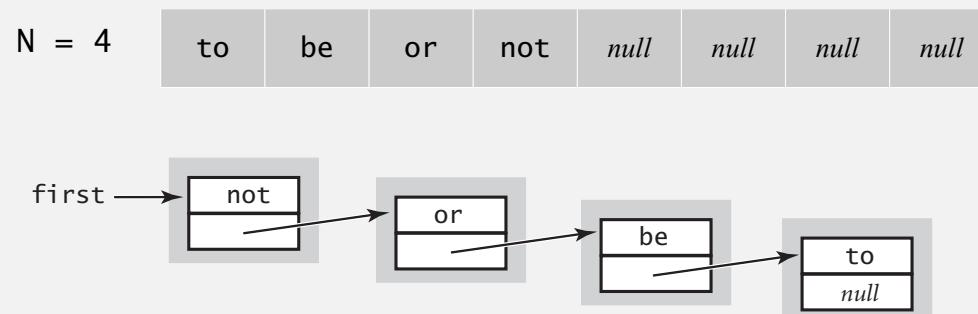
**Tradeoffs.** Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

### Linked-list implementation.

- Every operation takes constant time in the **worst case**.
- Uses extra time and space to deal with the links.

### Resizing-array implementation.

- Every operation takes constant **amortized** time.
- Less wasted space.



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Queue API

```
public class QueueOfStrings
```

```
    QueueOfStrings()
```

*create an empty queue*

```
    void enqueue(String item)
```

*insert a new string onto queue*

```
    String dequeue()
```

*remove and return the string  
least recently added*

```
    boolean isEmpty()
```

*is the queue empty?*

```
    int size()
```

*number of strings on the queue*

enqueue

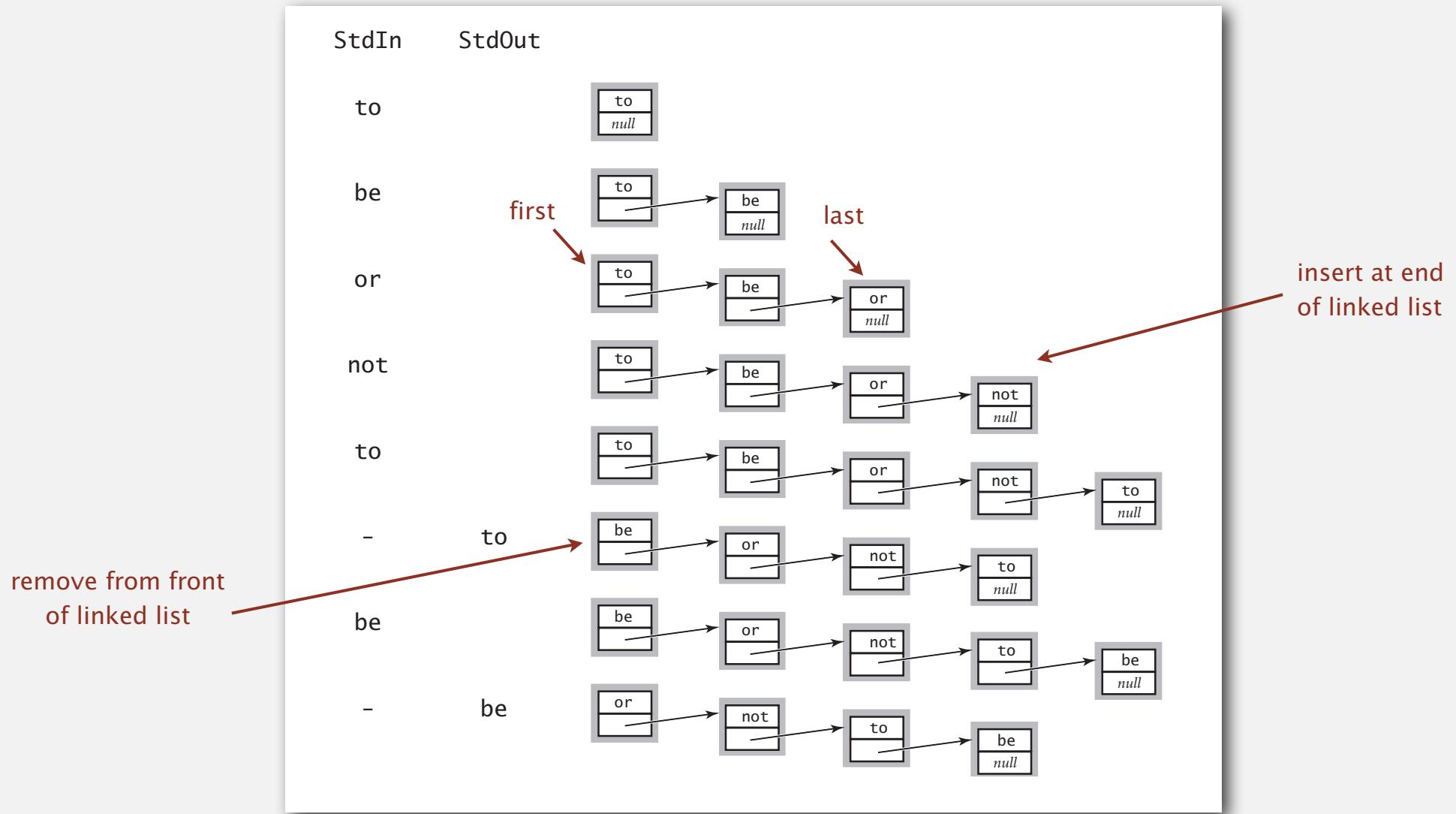


dequeue



# Queue: linked-list representation

Maintain pointer to first and last nodes in a linked list;  
insert/remove from opposite ends.



# Queue dequeue: linked-list implementation

## inner class

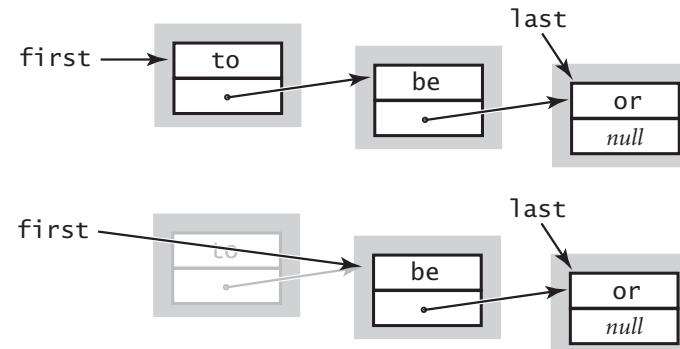
```
private class Node  
{  
    String item;  
    Node next;  
}
```

### save item to return

```
String item = first.item;
```

### delete first node

```
first = first.next;
```



### return saved item

```
return item;
```

**Remark.** Identical code to linked-list stack pop().

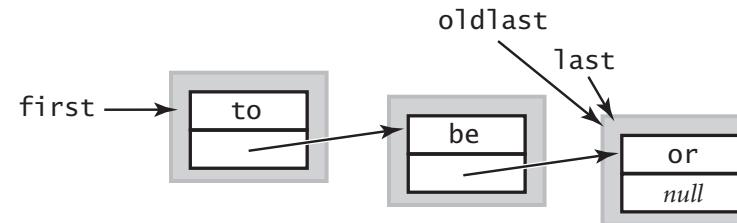
# Queue enqueue: linked-list implementation

## inner class

```
private class Node  
{  
    String item;  
    Node next;  
}
```

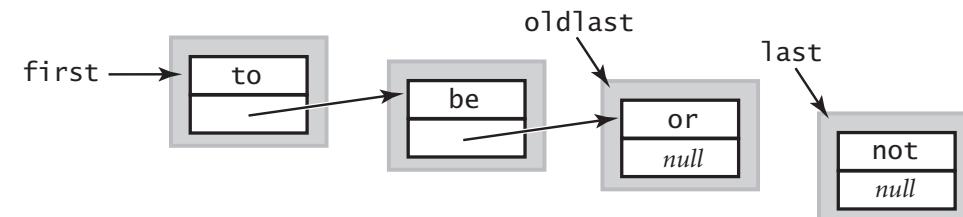
### save a link to the last node

```
Node oldlast = last;
```



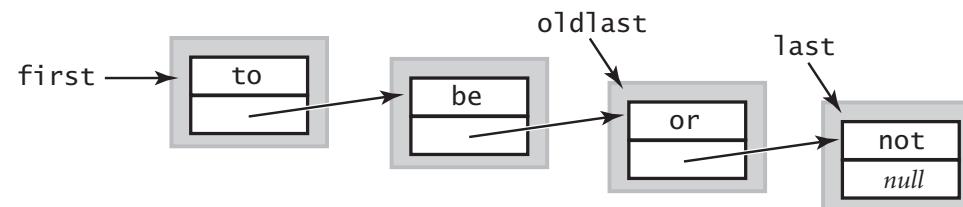
### create a new node for the end

```
last = new Node();  
last.item = "not";
```



### link the new node to the end of the list

```
oldlast.next = last;
```



# Queue: linked-list implementation in Java

```
public class LinkedQueueOfStrings
{
    private Node first, last;

    private class Node
    { /* same as in StackOfStrings */ }

    public boolean isEmpty()
    { return first == null; }

    public void enqueue(String item)
    {
        Node oldlast = last;
        last = new Node();
        last.item = item;
        last.next = null;
        if (isEmpty()) first = last;
        else           oldlast.next = last;
    }

    public String dequeue()
    {
        String item = first.item;
        first     = first.next;
        if (isEmpty()) last = null;
        return item;
    }
}
```

special cases for empty queue

# Queue: resizing array implementation

---

## Array implementation of a queue.

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update head and tail modulo the capacity.
- Add resizing array.



Q. How to resize?

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

## Parameterized stack

---

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 1. Implement a separate stack class for each type.

- Rewriting code is tedious and error-prone.
- Maintaining cut-and-pasted code is tedious and error-prone.

@#\$\*! most reasonable approach until Java 1.5.



# Parameterized stack

---

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 2. Implement a stack with items of type Object.

- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```
StackOfObjects s = new StackOfObjects();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

run-time error



## Parameterized stack

---

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 3. Java generics.

- Avoid casting in client.
- Discover type mismatch errors at compile-time instead of run-time.

The diagram shows a code snippet within a gray rectangular box. Two red arrows point from the text "type parameter" to the generic type parameters "`<Apple>`" in the first two lines of the code. A third red arrow points from the text "compile-time error" to the line `s.push(b);`, indicating that this is where a compile-time error would occur due to type mismatch.

```
Stack<Apple> s = new Stack<Apple>();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = s.pop();
```

Guiding principles. Welcome compile-time errors; avoid run-time errors.

# Generic stack: linked-list implementation

```
public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

    public boolean isEmpty()
    {   return first == null;   }

    public void push(String item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

```
public class Stack<Item>
{
    private Node first = null;

    private class Node
    {
        Item item;
        Node next;
    }

    public boolean isEmpty()
    {   return first == null;   }

    public void push(Item item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public Item pop()
    {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
```

generic type name

# Generic stack: array implementation

```
public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public ..StackOfStrings(int capacity)
    {   s = new String[capacity];   }

    public boolean isEmpty()
    {   return N == 0;   }

    public void push(String item)
    {   s[N++] = item;   }

    public String pop()
    {   return s[--N];   }
}
```

the way it should be

```
public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    {   s = new Item[capacity];   }

    public boolean isEmpty()
    {   return N == 0;   }

    public void push(Item item)
    {   s[N++] = item;   }

    public Item pop()
    {   return s[--N];   }
}
```

@#\$\*! generic array creation not allowed in Java

# Generic stack: array implementation

```
public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public ..StackOfStrings(int capacity)
    {   s = new String[capacity];   }

    public boolean isEmpty()
    {   return N == 0;   }

    public void push(String item)
    {   s[N++] = item;   }

    public String pop()
    {   return s[--N];   }
}
```

the way it is

```
public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    {   s = (Item[]) new Object[capacity];   }

    public boolean isEmpty()
    {   return N == 0;   }

    public void push(Item item)
    {   s[N++] = item;   }

    public Item pop()
    {   return s[--N];   }
}
```

the ugly cast

## Unchecked cast

---

```
% javac FixedCapacityStack.java
```

Note: FixedCapacityStack.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

```
% javac -Xlint:unchecked FixedCapacityStack.java
```

FixedCapacityStack.java:26: warning: [unchecked] unchecked cast

found : java.lang.Object[]

required: Item[]

```
    a = (Item[]) new Object[capacity];
```

^

1 warning

## Generic data types: autoboxing

---

Q. What to do about primitive types?

Wrapper type.

- Each primitive type has a **wrapper** object type.
- Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast between a primitive type and its wrapper.

```
Stack<Integer> s = new Stack<Integer>();
s.push(17);           // s.push(Integer.valueOf(17));
int a = s.pop();     // int a = s.pop().intValue();
```

Bottom line. Client code can use generic stack for **any** type of data.

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

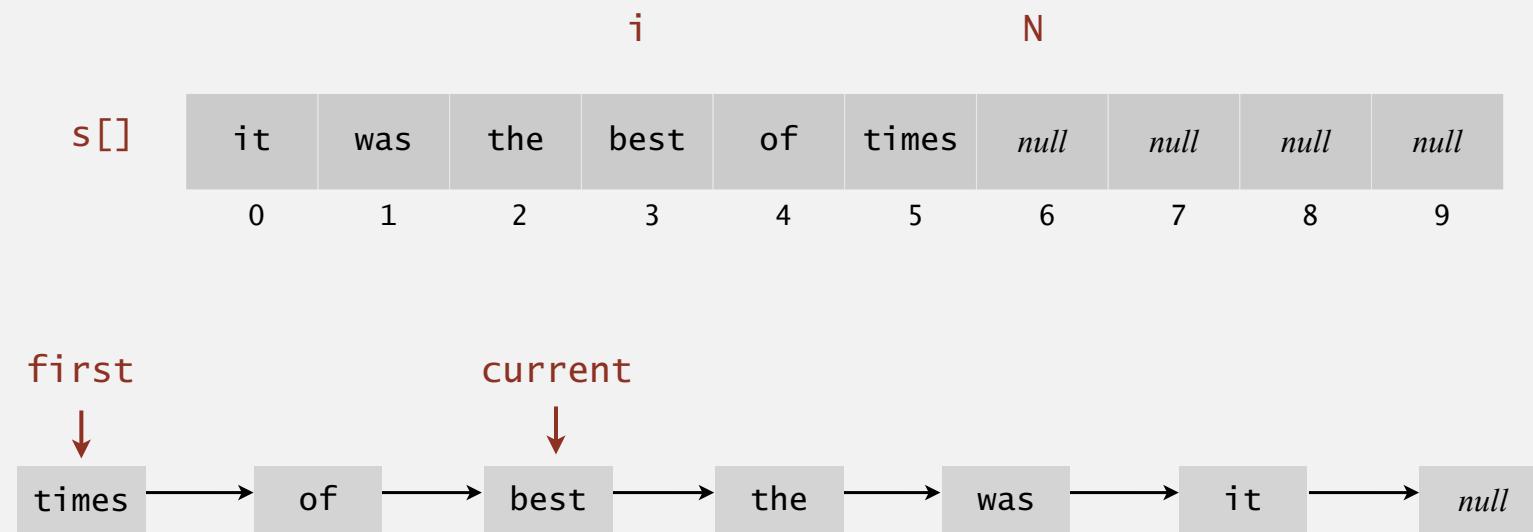
---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Iteration

---

**Design challenge.** Support iteration over stack items by client, without revealing the internal representation of the stack.



**Java solution.** Make stack implement the `java.lang.Iterable` interface.

# Iterators

---

Q. What is an **Iterable** ?

A. Has a method that returns an **Iterator**.

## Iterable interface

```
public interface Iterable<Item>
{
    Iterator<Item> iterator();
}
```

Q. What is an **Iterator** ?

A. Has methods `hasNext()` and `next()`.

## Iterator interface

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove(); ← optional; use at your own risk
}
```

Q. Why make data structures **Iterable** ?

A. Java supports elegant client code.

## “foreach” statement (shorthand)

```
for (String s : stack)
    StdOut.println(s);
```

## equivalent code (longhand)

```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
    String s = i.next();
    StdOut.println(s);
}
```

## Stack iterator: linked-list implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

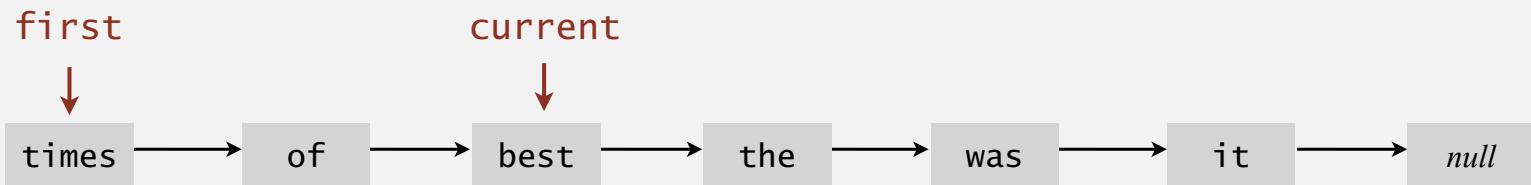
        public boolean hasNext() { return current != null; }

        public void remove() { /* not supported */ }

        public Item next()
        {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```

Diagram annotations for the ListIterator code:

- An arrow points from the `remove()` method to the note: "throw UnsupportedOperationException".
- An arrow points from the `next()` method to the note: "throw NoSuchElementException if no more items in iteration".



## Stack iterator: array implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator()
    { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() { return i > 0; }
        public void remove()    { /* not supported */ }
        public Item next()      { return s[--i]; }
    }
}
```

s[]	it	was	the	best	of	times	null	null	null	null
	0	1	2	3	4	5	6	7	8	9

i

N

# Bag API

Main application. Adding items to a collection and iterating (when order doesn't matter).

```
public class Bag<Item> implements Iterable<Item>
```

```
    Bag()
```

*create an empty bag*

```
    void add(Item x)
```

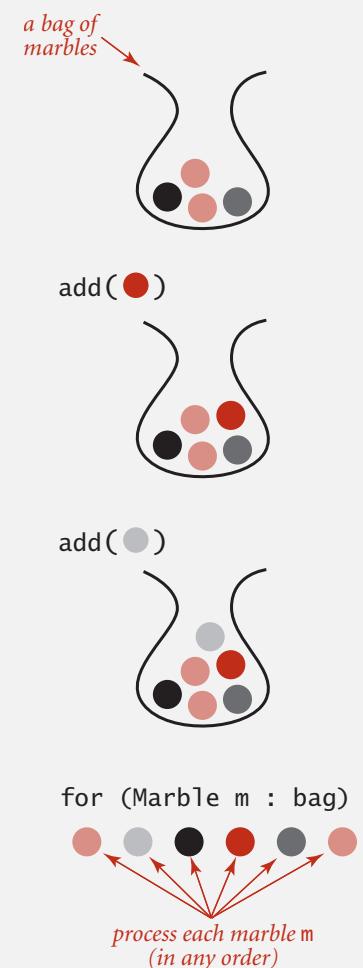
*insert a new item onto bag*

```
    int size()
```

*number of items in bag*

```
    Iterable<Item> iterator()
```

*iterator for all items in bag*



Implementation. Stack (without pop) or queue (without dequeue).

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Java collections library

---

**List interface.** `java.util.List` is API for an sequence of items.

<code>public interface List&lt;Item&gt; implements Iterable&lt;Item&gt;</code>	
<code>List()</code>	<i>create an empty list</i>
<code>boolean isEmpty()</code>	<i>is the list empty?</i>
<code>int size()</code>	<i>number of items</i>
<code>void add(Item item)</code>	<i>append item to the end</i>
<code>Item get(int index)</code>	<i>return item at given index</i>
<code>Item remove(int index)</code>	<i>return and delete item at given index</i>
<code>boolean contains(Item item)</code>	<i>does the list contain the given item?</i>
<code>Iterator&lt;Item&gt; iterator()</code>	<i>iterator over all items in the list</i>
<code>...</code>	

**Implementations.** `java.util.ArrayList` uses resizing array;

`java.util.LinkedList` uses linked list. 

caveat: only some operations are efficient

# Java collections library

---

## `java.util.Stack`.

- Supports push(), pop(), and and iteration.
- Extends `java.util.Vector`, which implements `java.util.List` interface from previous slide, including, get() and remove().
- Bloated and poorly-designed API (why?)



`java.util.Queue`. An interface, not an implementation of a queue.

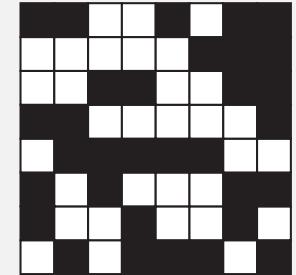
**Best practices.** Use our implementations of Stack, Queue, and Bag.

## War story (from Assignment 1)

---

Generate random open sites in an  $N$ -by- $N$  percolation system.

- Jenny: pick  $(i, j)$  at random; if already open, repeat.  
Takes  $\sim c_1 N^2$  seconds.
- Kenny: create a `java.util.ArrayList` of  $N^2$  closed sites.  
Pick an index at random and delete.  
Takes  $\sim c_2 N^4$  seconds.



Why is my program so slow?



Kenny

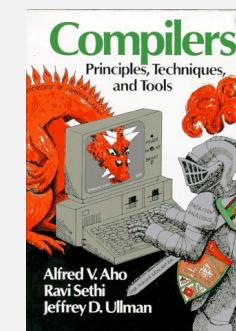
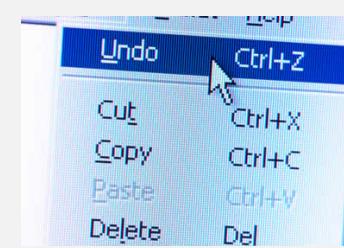
Lesson. Don't use a library until you understand its API!

This course. Can't use a library until we've implemented it in class.

# Stack applications

---

- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.
- ...



# Function calls

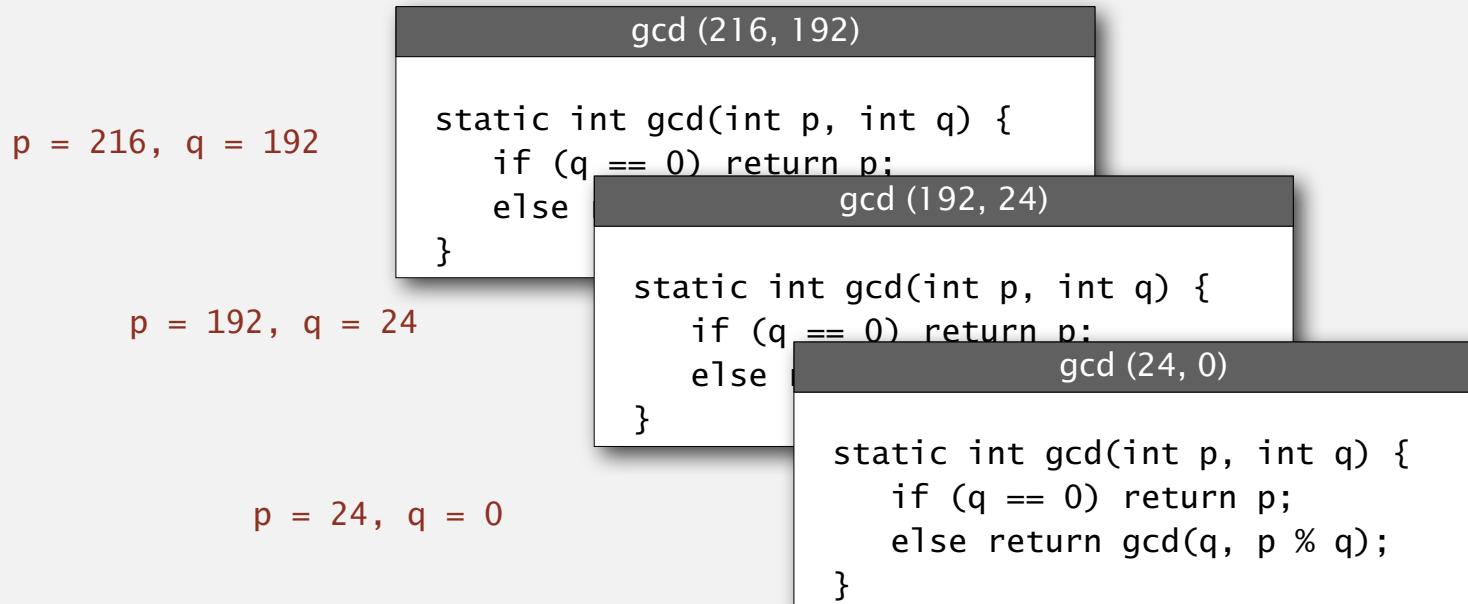
---

How a compiler implements a function.

- Function call: **push** local environment and return address.
- Return: **pop** return address and local environment.

**Recursive function.** Function that calls itself.

**Note.** Can always use an explicit stack to remove recursion.



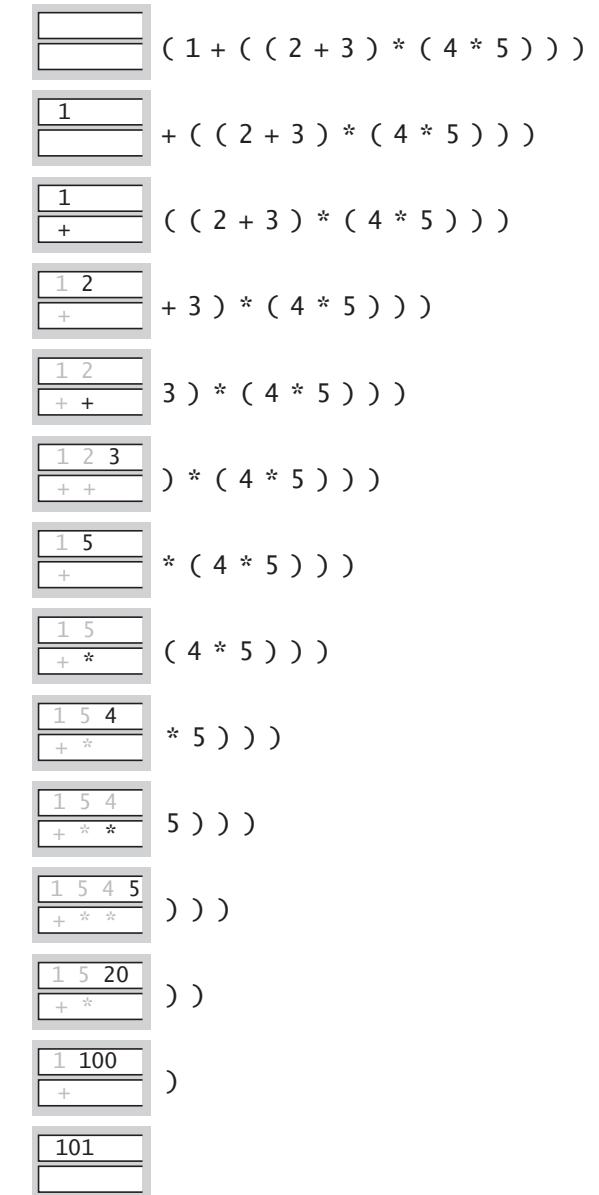
# Arithmetic expression evaluation

Goal. Evaluate infix expressions.

( 1 + ( ( 2 + 3 ) \* ( 4 \* 5 ) ) )

operand                    operator

value stack  
operator stack



Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parenthesis: ignore.
- Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!

# Dijkstra's two-stack algorithm demo

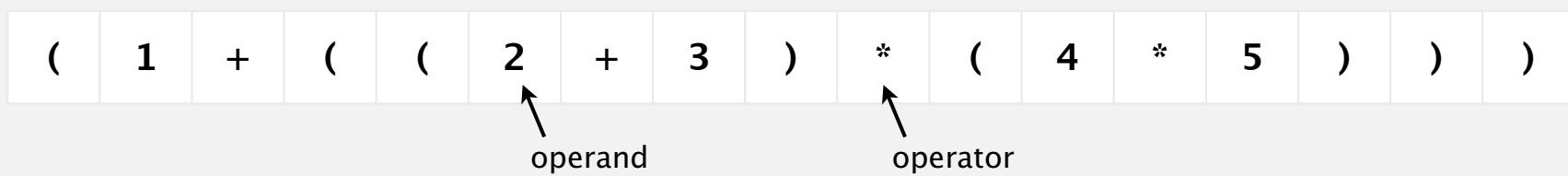
---



**infix expression**  
**(fully parenthesized)**

value stack

operator stack



# Arithmetic expression evaluation

---

```
public class Evaluate
{
    public static void main(String[] args)
    {
        Stack<String> ops = new Stack<String>();
        Stack<Double> vals = new Stack<Double>();
        while (!StdIn.isEmpty()) {
            String s = StdIn.readString();
            if      (s.equals("("))           ;
            else if (s.equals("+"))   ops.push(s);
            else if (s.equals("*"))   ops.push(s);
            else if (s.equals(")"))
            {
                String op = ops.pop();
                if      (op.equals("+")) vals.push(vals.pop() + vals.pop());
                else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
            }
            else vals.push(Double.parseDouble(s));
        }
        StdOut.println(vals.pop());
    }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

## Correctness

---

Q. Why correct?

A. When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

```
( 1 + ( 2 + 3 ) * ( 4 * 5 ) ) )
```

as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

Repeating the argument:

```
( 1 + ( 5 * 20 ) )  
( 1 + 100 )  
101
```

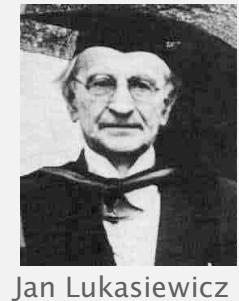
Extensions. More ops, precedence order, associativity.

# Stack-based programming languages

---

**Observation 1.** Dijkstra's two-stack algorithm computes the same value if the operator occurs **after** the two values.

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```



**Observation 2.** All of the parentheses are redundant!

```
1 2 3 + 4 5 * * +
```

**Bottom line.** Postfix or "reverse Polish" notation.

**Applications.** Postscript, Forth, calculators, Java virtual machine, ...

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

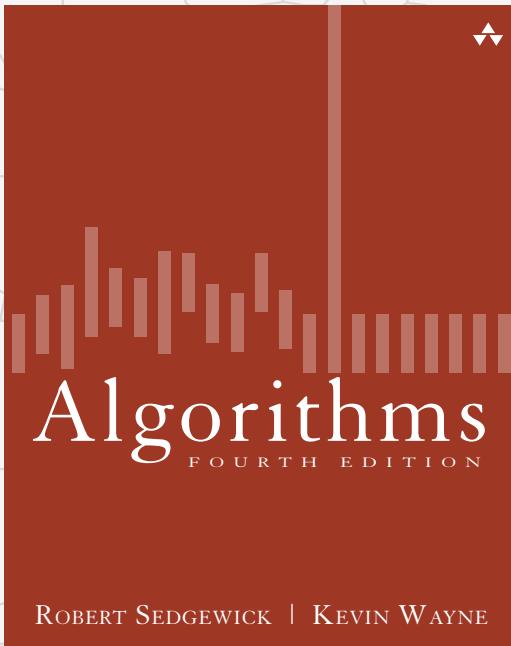
## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

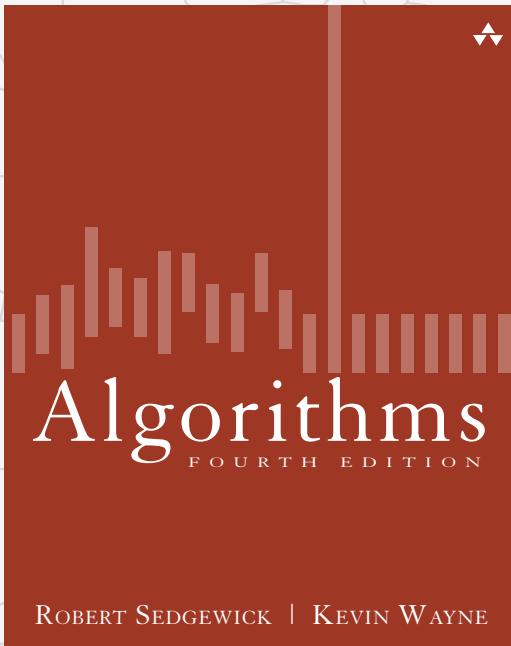


<http://algs4.cs.princeton.edu>

## 1.3 BAGS, QUEUES, AND STACKS

---

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*



<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

---

- ▶ *rules of the game*
- ▶ *selection sort*
- ▶ *insertion sort*
- ▶ *shellsort*
- ▶ *shuffling*
- ▶ *convex hull*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

---

- ▶ *rules of the game*
- ▶ *selection sort*
- ▶ *insertion sort*
- ▶ *shellsort*
- ▶ *shuffling*
- ▶ *convex hull*

# Sorting problem

---

Ex. Student records in a university.

Chen	3	A	991-878-4944	308 Blair
Rohde	2	A	232-343-5555	343 Forbes
Gazsi	4	B	766-093-9873	101 Brown
Furia	1	A	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman

item →

key →

Sort. Rearrange array of  $N$  items into ascending order.

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

# Sample sort client 1

---

Goal. Sort **any** type of data.

Ex 1. Sort random real numbers in ascending order.

seems artificial, but stay tuned for an application

```
public class Experiment
{
    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);
        Double[] a = new Double[N];
        for (int i = 0; i < N; i++)
            a[i] = StdRandom.uniform();
        Insertion.sort(a);
        for (int i = 0; i < N; i++)
            StdOut.println(a[i]);
    }
}
```

```
% java Experiment 10
0.08614716385210452
0.09054270895414829
0.10708746304898642
0.21166190071646818
0.363292849257276
0.460954145685913
0.5340026311350087
0.7216129793703496
0.9003500354411443
0.9293994908845686
```

## Sample sort client 2

---

Goal. Sort **any** type of data.

Ex 2. Sort strings from file in alphabetical order.

```
public class StringSorter
{
    public static void main(String[] args)
    {
        String[] a = In.readStrings(args[0]);
        Insertion.sort(a);
        for (int i = 0; i < a.length; i++)
            StdOut.println(a[i]);
    }
}
```

```
% more words3.txt
bed bug dad yet zoo ... all bad yes
```

```
% java StringSorter words3.txt
all bad bed bug dad ... yes yet zoo
```

## Sample sort client 3

---

Goal. Sort **any** type of data.

Ex 3. Sort the files in a given directory by filename.

```
import java.io.File;
public class FileSorter
{
    public static void main(String[] args)
    {
        File directory = new File(args[0]);
        File[] files = directory.listFiles();
        Insertion.sort(files);
        for (int i = 0; i < files.length; i++)
            StdOut.println(files[i].getName());
    }
}
```

```
% java FileSorter .
Insertion.class
Insertion.java
InsertionX.class
InsertionX.java
Selection.class
Selection.java
Shell.class
Shell.java
ShellX.class
ShellX.java
```

# Callbacks

---

**Goal.** Sort **any** type of data.

**Q.** How can `sort()` know how to compare data of type `Double`, `String`, and `java.io.File` without any information about the type of an item's key?

**Callback = reference to executable code.**

- Client passes array of objects to `sort()` function.
- The `sort()` function calls back object's `compareTo()` method as needed.

**Implementing callbacks.**

- Java: **interfaces**.
- C: **function pointers**.
- C++: **class-type functors**.
- C#: **delegates**.
- Python, Perl, ML, Javascript: **first-class functions**.

# Callbacks: roadmap

## client

```
import java.io.File;
public class FileSorter
{
    public static void main(String[] args)
    {
        File directory = new File(args[0]);
        File[] files = directory.listFiles();
        Insertion.sort(files);
        for (int i = 0; i < files.length; i++)
            StdOut.println(files[i].getName());
    }
}
```

## object implementation

```
public class File
implements Comparable<File>
{
    ...
    public int compareTo(File b)
    {
        ...
        return -1;
        ...
        return +1;
        ...
        return 0;
    }
}
```

## Comparable interface (built in to Java)

```
public interface Comparable<Item>
{
    public int compareTo(Item that);
}
```

key point: no dependence  
on File data type

## sort implementation

```
public static void sort(Comparable[] a)
{
    int N = a.length;
    for (int i = 0; i < N; i++)
        for (int j = i; j > 0; j--)
            if (a[j].compareTo(a[j-1]) < 0)
                exch(a, j, j-1);
            else break;
}
```

## Total order

---

A **total order** is a binary relation  $\leq$  that satisfies:

- Antisymmetry: if  $v \leq w$  and  $w \leq v$ , then  $v = w$ .
- Transitivity: if  $v \leq w$  and  $w \leq x$ , then  $v \leq x$ .
- Totality: either  $v \leq w$  or  $w \leq v$  or both.

**Ex.**

- Standard order for natural and real numbers.
- Chronological order for dates or times.
- Alphabetical order for strings.
- ...

violates totality: `(Double.NaN <= Double.NaN)` is false



an intransitive relation

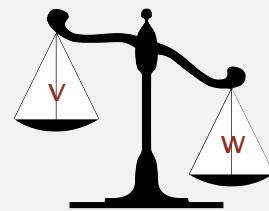
Surprising but true. The `<=` operator for `double` is not a total order. (!)

# Comparable API

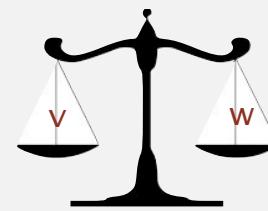
---

Implement `compareTo()` so that `v.compareTo(w)`

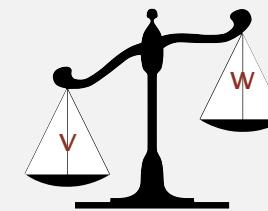
- Is a total order.
- Returns a negative integer, zero, or positive integer if `v` is less than, equal to, or greater than `w`, respectively.
- Throws an exception if incompatible types (or either is `null`).



less than (return -1)



equal to (return 0)



greater than (return +1)

Built-in comparable types. Integer, Double, String, Date, File, ...

User-defined comparable types. Implement the Comparable interface.

# Implementing the Comparable interface

Date data type. Simplified version of java.util.Date.

```
public class Date implements Comparable<Date>
{
    private final int month, day, year;

    public Date(int m, int d, int y)
    {
        month = m;
        day   = d;
        year  = y;
    }
}
```

```
public int compareTo(Date that)
{
    if (this.year < that.year) return -1;
    if (this.year > that.year) return +1;
    if (this.month < that.month) return -1;
    if (this.month > that.month) return +1;
    if (this.day   < that.day ) return -1;
    if (this.day   > that.day ) return +1;
    return 0;
}
```

only compare dates  
to other dates

## Two useful sorting abstractions

---

Helper functions. Refer to data through compares and exchanges.

Less. Is item v less than w ?

```
private static boolean less(Comparable v, Comparable w)
{  return v.compareTo(w) < 0;  }
```

Exchange. Swap item in array a[] at index i with the one at index j.

```
private static void exch(Comparable[] a, int i, int j)
{
    Comparable swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```

# Testing

---

**Goal.** Test if an array is sorted.

```
private static boolean isSorted(Comparable[] a)
{
    for (int i = 1; i < a.length; i++)
        if (less(a[i], a[i-1])) return false;
    return true;
}
```

**Q.** If the sorting algorithm passes the test, did it correctly sort the array?

**A.**

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

---

- ▶ *rules of the game*
- ▶ *selection sort*
- ▶ *insertion sort*
- ▶ *shellsort*
- ▶ *shuffling*
- ▶ *convex hull*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

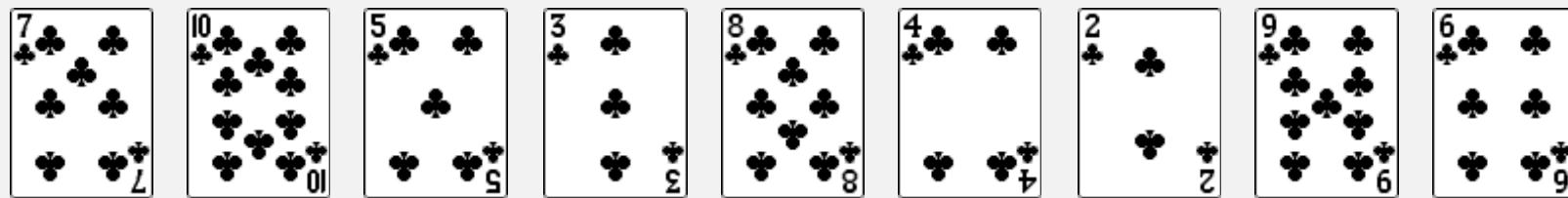
---

- ▶ *rules of the game*
- ▶ ***selection sort***
- ▶ *insertion sort*
- ▶ *shellsort*
- ▶ *shuffling*
- ▶ ***convex hull***

## Selection sort demo

---

- In iteration  $i$ , find index  $\text{min}$  of smallest remaining entry.
- Swap  $a[i]$  and  $a[\text{min}]$ .



initial



# Selection sort

---

Algorithm. ↑ scans from left to right.

Invariants.

- Entries to the left of ↑ (including ↑) fixed and in ascending order.
- No entry to right of ↑ is smaller than any entry to the left of ↑.



# Selection sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```



- Identify index of minimum entry on right.

```
int min = i;
for (int j = i+1; j < N; j++)
    if (less(a[j], a[min]))
        min = j;
```



- Exchange into position.

```
exch(a, i, min);
```



## Selection sort: Java implementation

---

```
public class Selection
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        for (int i = 0; i < N; i++)
        {
            int min = i;
            for (int j = i+1; j < N; j++)
                if (less(a[j], a[min]))
                    min = j;
            exch(a, i, min);
        }
    }

    private static boolean less(Comparable v, Comparable w)
    { /* as before */ }

    private static void exch(Comparable[] a, int i, int j)
    { /* as before */ }
}
```

## Selection sort: mathematical analysis

Proposition. Selection sort uses  $(N-1) + (N-2) + \dots + 1 + 0 \sim N^2/2$  compares and  $N$  exchanges.

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X

Trace of selection sort (array contents just after each exchange)

Running time insensitive to input. Quadratic time, even if input is sorted.  
Data movement is minimal. Linear number of exchanges.

## Selection sort: animations

## 20 random items

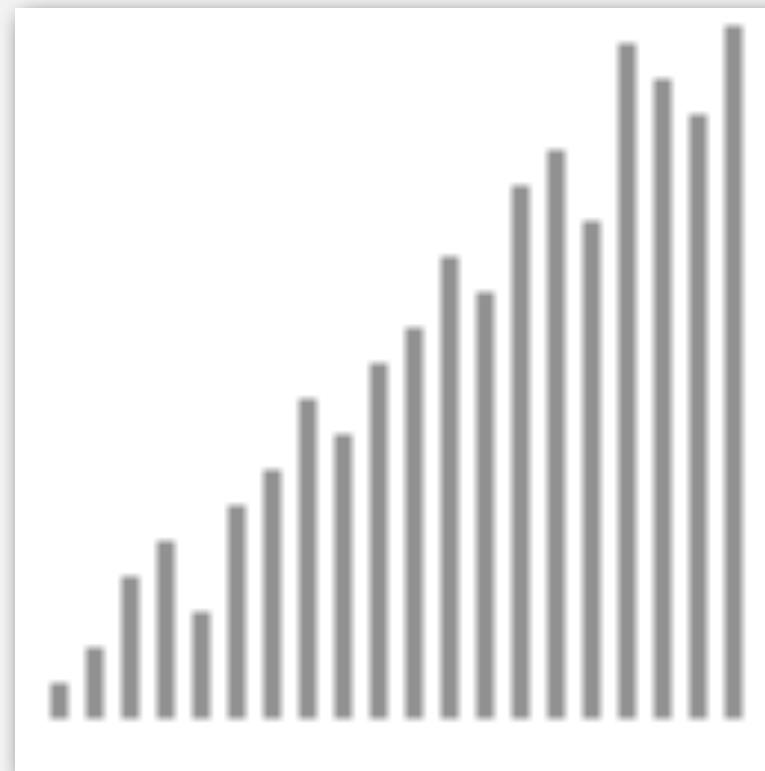





<http://www.sorting-algorithms.com/selection-sort>

## Selection sort: animations

## 20 partially-sorted items



- The legend consists of three horizontal bars and their corresponding labels:

  - A red triangle pointing upwards followed by the text "algorithm position in final order".
  - A black bar followed by the text "not in final order".
  - A grey bar followed by the text "not in final order".

<http://www.sorting-algorithms.com/selection-sort>

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

---

- ▶ *rules of the game*
- ▶ ***selection sort***
- ▶ *insertion sort*
- ▶ *shellsort*
- ▶ *shuffling*
- ▶ ***convex hull***

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

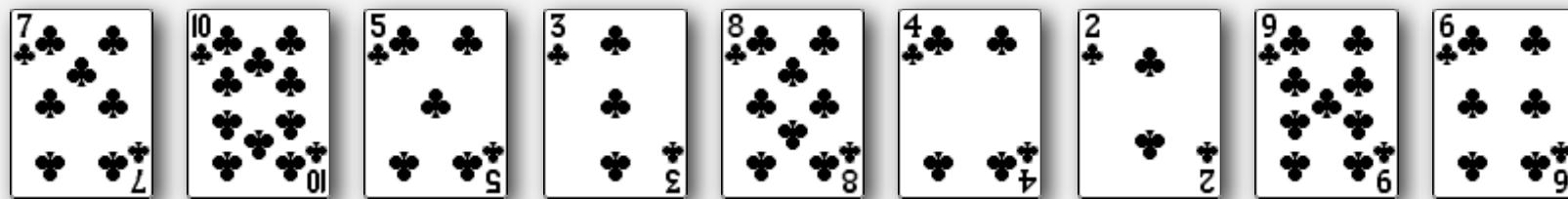
---

- ▶ *rules of the game*
- ▶ *selection sort*
- ▶ *insertion sort*
- ▶ *shellsort*
- ▶ *shuffling*
- ▶ *convex hull*

## Insertion sort demo

---

- In iteration  $i$ , swap  $a[i]$  with each larger entry to its left.



## Insertion sort

---

Algorithm.  $\uparrow$  scans from left to right.

### Invariants.

- Entries to the left of  $\uparrow$  (including  $\uparrow$ ) are in ascending order.
- Entries to the right of  $\uparrow$  have not yet been seen.



## Insertion sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```



- Moving from right to left, exchange  $a[i]$  with each larger entry to its left.

```
for (int j = i; j > 0; j--)  
    if (less(a[j], a[j-1]))  
        exch(a, j, j-1);  
    else break;
```



## Insertion sort: Java implementation

---

```
public class Insertion
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        for (int i = 0; i < N; i++)
            for (int j = i; j > 0; j--)
                if (less(a[j], a[j-1]))
                    exch(a, j, j-1);
                else break;
    }

    private static boolean less(Comparable v, Comparable w)
    { /* as before */ }

    private static void exch(Comparable[] a, int i, int j)
    { /* as before */ }
}
```

## Insertion sort: mathematical analysis

**Proposition.** To sort a randomly-ordered array with distinct keys, insertion sort uses  $\sim \frac{1}{4} N^2$  compares and  $\sim \frac{1}{4} N^2$  exchanges on average.

**Pf.** Expect each entry to move halfway back.

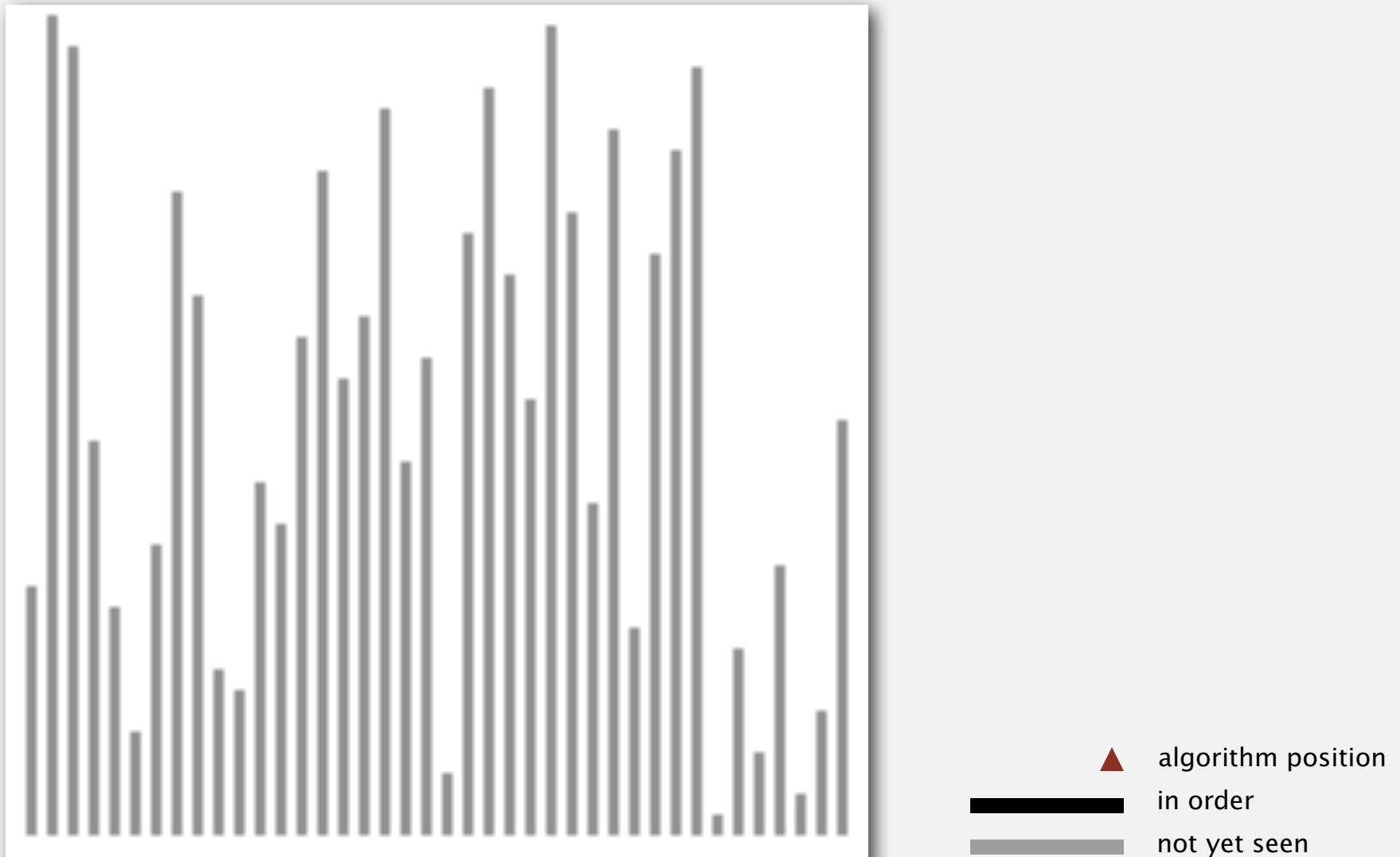
		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Trace of insertion sort (array contents just after each insertion)

## Insertion sort: trace

## Insertion sort: animation

## 40 random items



<http://www.sorting-algorithms.com/insertion-sort>

## Insertion sort: best and worst case

---

**Best case.** If the array is in ascending order, insertion sort makes  $N-1$  compares and 0 exchanges.

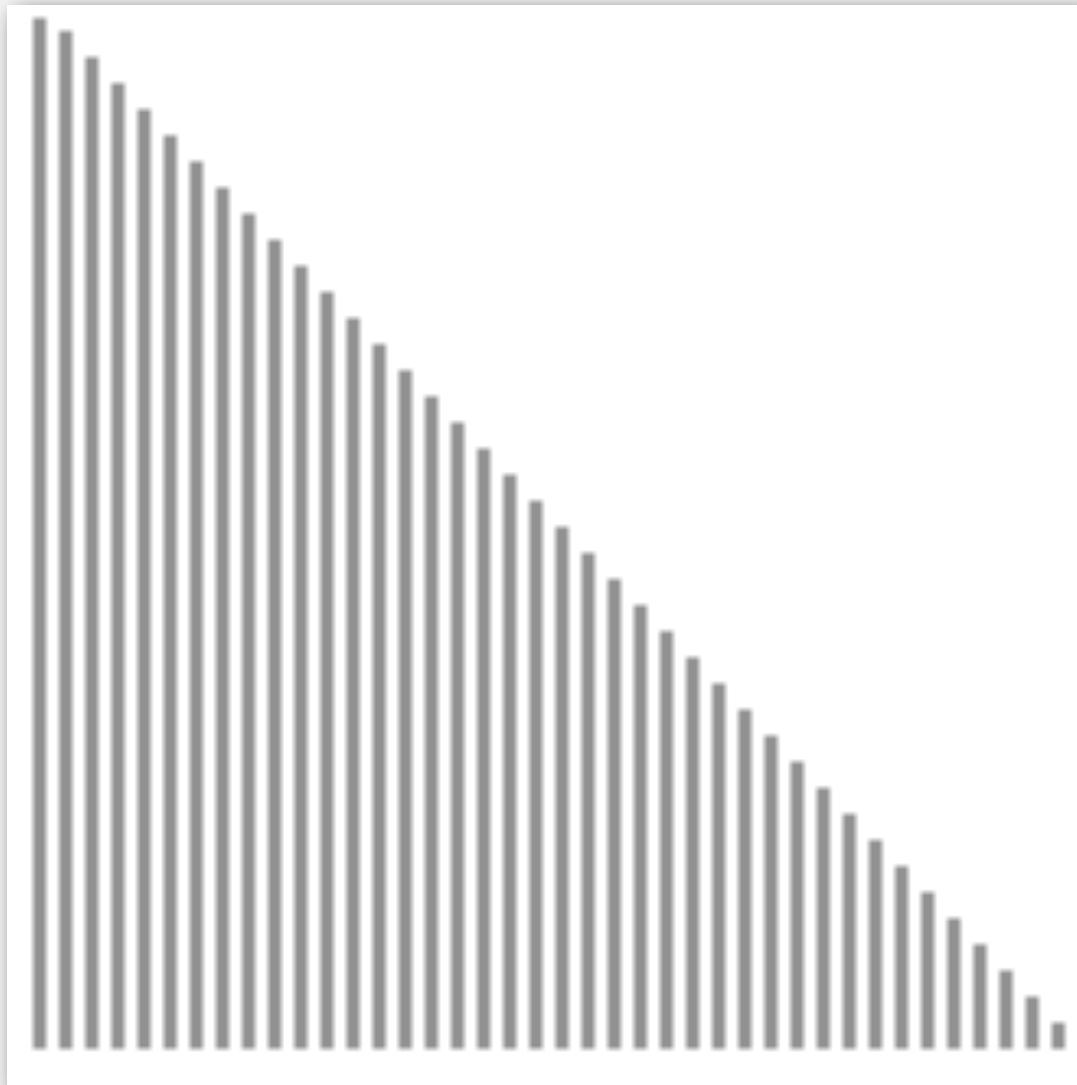
A E E L M O P R S T X

**Worst case.** If the array is in descending order (and no duplicates), insertion sort makes  $\sim \frac{1}{2} N^2$  compares and  $\sim \frac{1}{2} N^2$  exchanges.

X T S R P O M L E E A

## Insertion sort: animation

## 40 reverse-sorted items

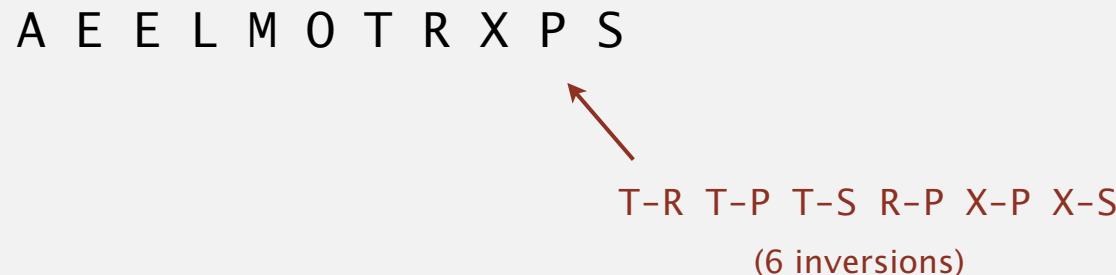


<http://www.sorting-algorithms.com/insertion-sort>

## Insertion sort: partially-sorted arrays

---

Def. An **inversion** is a pair of keys that are out of order.



Def. An array is **partially sorted** if the number of inversions is  $\leq c N$ .

- Ex 1. A subarray of size 10 appended to a sorted subarray of size  $N$ .
- Ex 2. An array of size  $N$  with only 10 entries out of place.

Proposition. For partially-sorted arrays, insertion sort runs in linear time.

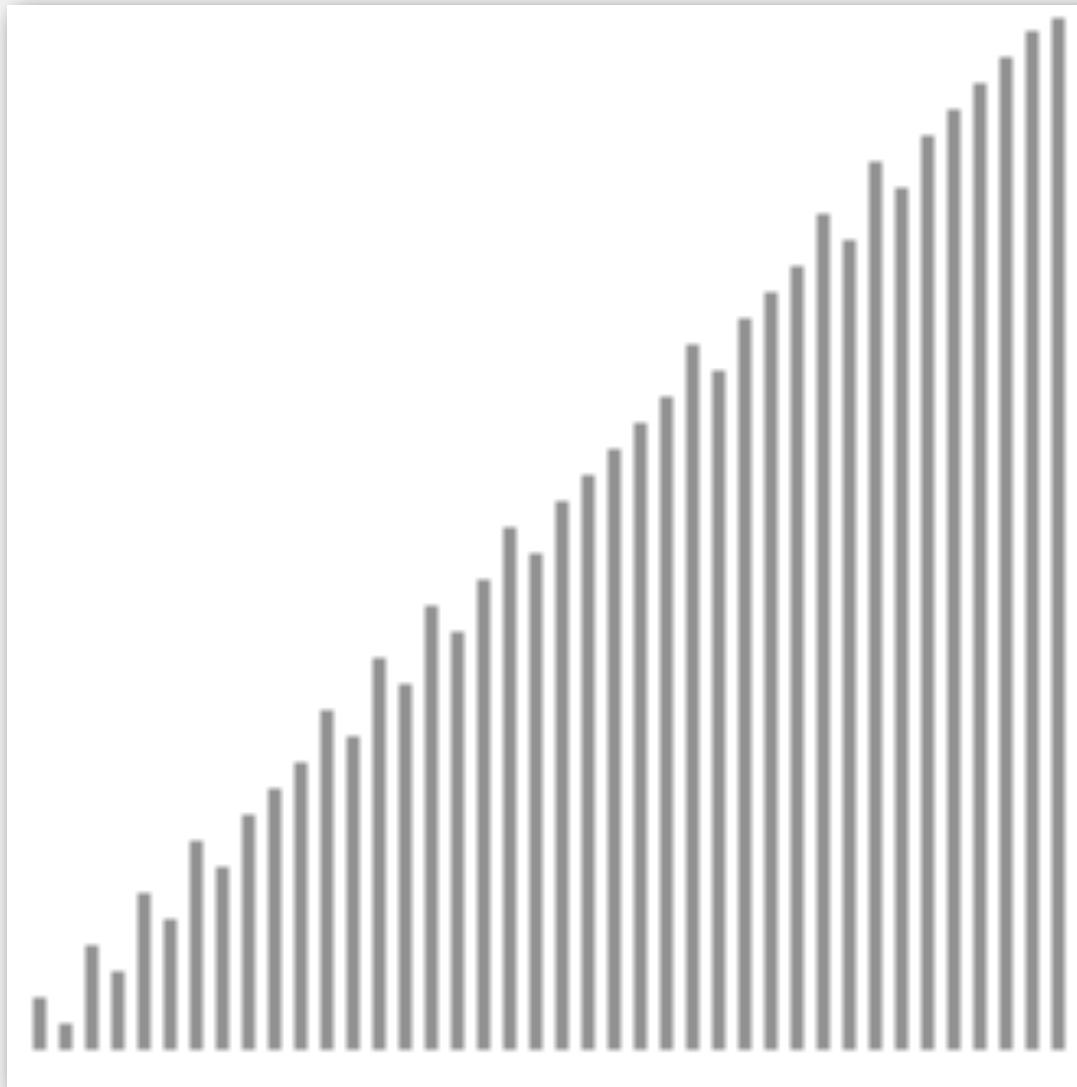
Pf. Number of exchanges equals the number of inversions.

$$\text{number of compares} = \text{exchanges} + (N - 1)$$

↑

## Insertion sort: animation

## 40 partially-sorted items



<http://www.sorting-algorithms.com/insertion-sort>

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

---

- ▶ *rules of the game*
- ▶ *selection sort*
- ▶ *insertion sort*
- ▶ *shellsort*
- ▶ *shuffling*
- ▶ *convex hull*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

---

- ▶ *rules of the game*
- ▶ *selection sort*
- ▶ *insertion sort*
- ▶ ***shellsort***
- ▶ *shuffling*
- ▶ *convex hull*

# Shellsort overview

Idea. Move entries more than one position at a time by *h*-sorting the array.

an *h*-sorted array is *h* interleaved sorted subsequences



Shellsort. [Shell 1959] *h*-sort array for decreasing sequence of values of *h*.

<b>input</b>	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
<b>13-sort</b>	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
<b>4-sort</b>	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
<b>1-sort</b>	A	E	E	E	H	L	L	M	O	P	R	S	S	T	X	

# **h**-sorting

---

How to *h*-sort an array? Insertion sort, with stride length *h*.

## 3-sorting an array

M	O	L	E	E	X	A	S	P	R	T
E	O	L	M	E	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T

## Why insertion sort?

- Big increments  $\Rightarrow$  small subarray.
- Small increments  $\Rightarrow$  nearly in order. [stay tuned]

# Shellsort example: increments 7, 3, 1

---

input

S O R T E X A M P L E

7-sort

S O R T E X A M P L E  
M O R T E X A S P L E  
M O R T E X A S P R L E  
M O L T E X A S P R E  
M O L E E X A S P R T

3-sort

M O L E E X A S P R T  
E O L M E X A S P R T  
E E L M O X A S P R T  
E E L M O X A S P R T  
A E L E O X M S P R T  
A E L E O X M S P R T  
A E L E O P M S X R T  
A E L E O P M S X R T  
A E L E O P M S X R T

1-sort

A E L E O P M S X R T  
A E L E O P M S X R T  
A E L E O P M S X R T  
A E E L O P M S X R T  
A E E L O P M S X R T  
A E E L O P M S X R T  
A E E L M O P S X R T  
A E E L M O P S X R T  
A E E L M O P R S X T  
A E E L M O P R S T X

result

A E E L M O P R S T X

# Shellsort: intuition

Proposition. A  $g$ -sorted array remains  $g$ -sorted after  $h$ -sorting it.

7-sort										3-sort									
S	O	R	T	E	X	A	M	P	L	E	M	O	L	E	E	X	A	S	P
M	O	R	T	E	X	A	S	P	L	E	E	O	L	M	E	X	A	S	P
M	O	R	T	E	X	A	S	P	L	E	E	E	L	M	O	X	A	S	P
M	O	L	T	E	X	A	S	P	R	E	E	E	L	M	O	X	A	S	P
M	O	L	E	E	X	A	S	P	R	T	A	E	L	E	O	X	M	S	P
											A	E	L	E	O	X	M	S	P
											A	E	L	E	O	P	M	S	R
											A	E	L	E	O	P	M	S	X
											A	E	L	E	O	P	M	S	R
											A	E	L	E	O	P	M	S	T
											A	E	L	E	O	P	M	S	X
											A	E	L	E	O	P	M	S	R
											A	E	L	E	O	P	M	S	T

still 7-sorted

Challenge. Prove this fact—it's more subtle than you'd think!

## Shellsort: which increment sequence to use?

---

Powers of two. 1, 2, 4, 8, 16, 32, ...

No.

Powers of two minus one. 1, 3, 7, 15, 31, 63, ...

Maybe.

→  $3x + 1$ . 1, 4, 13, 40, 121, 364, ...

OK. Easy to compute.

Sedgewick. 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, ...

Good. Tough to beat in empirical studies.

merging of  $(9 \times 4^i) - (9 \times 2^i) + 1$   
and  $4^i - (3 \times 2^i) + 1$

# Shellsort: Java implementation

```
public class Shell
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;

        int h = 1;
        while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, ...
        ← 3x+1 increment sequence

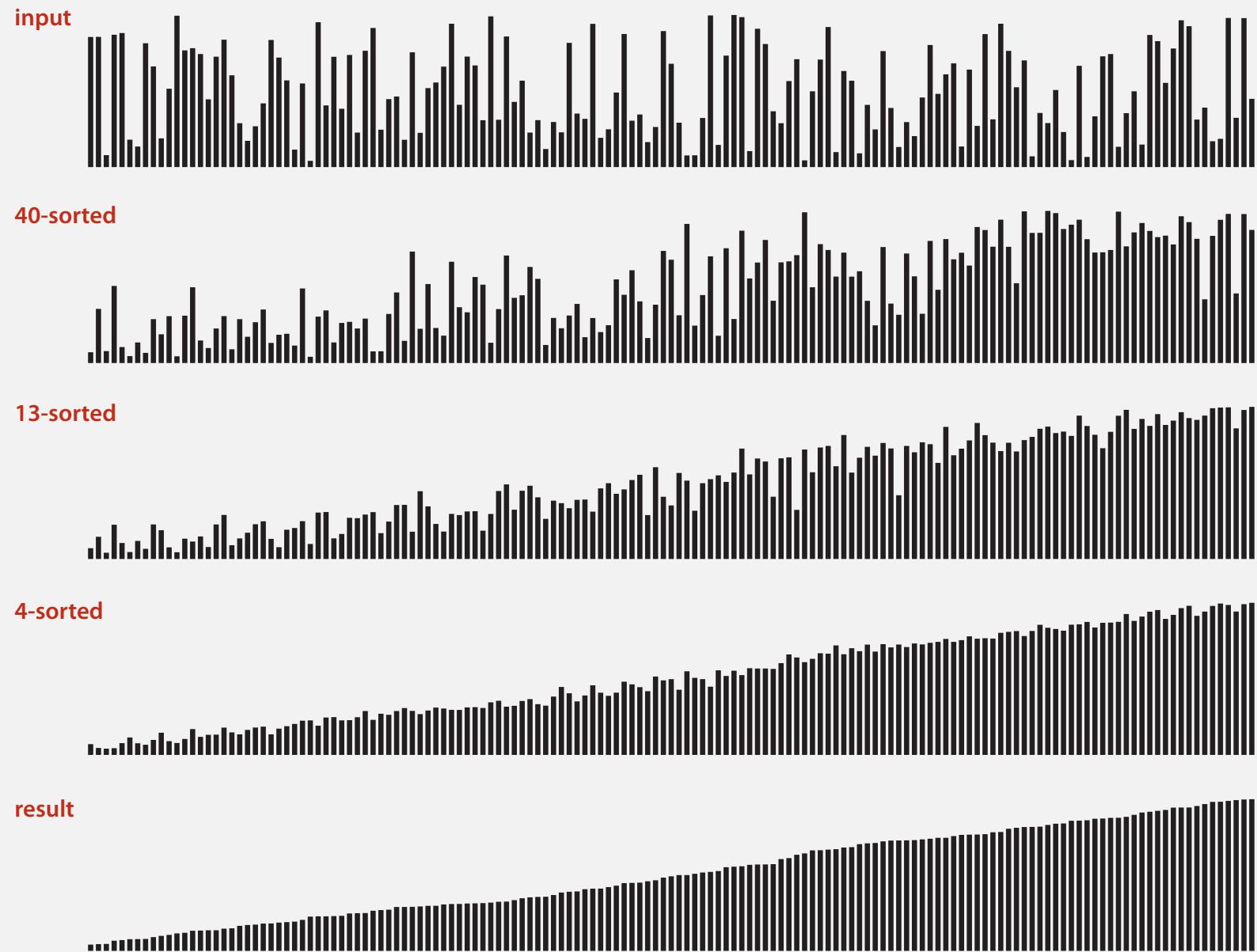
        while (h >= 1)
        { // h-sort the array.
            for (int i = h; i < N; i++)
            {
                for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
                    exch(a, j, j-h);
            }
            ← insertion sort

            h = h/3;
        }
        ← move to next increment
    }

    private static boolean less(Comparable v, Comparable w)
    { /* as before */ }
    private static void exch(Comparable[] a, int i, int j)
    { /* as before */ }
}
```

# Shellsort: visual trace

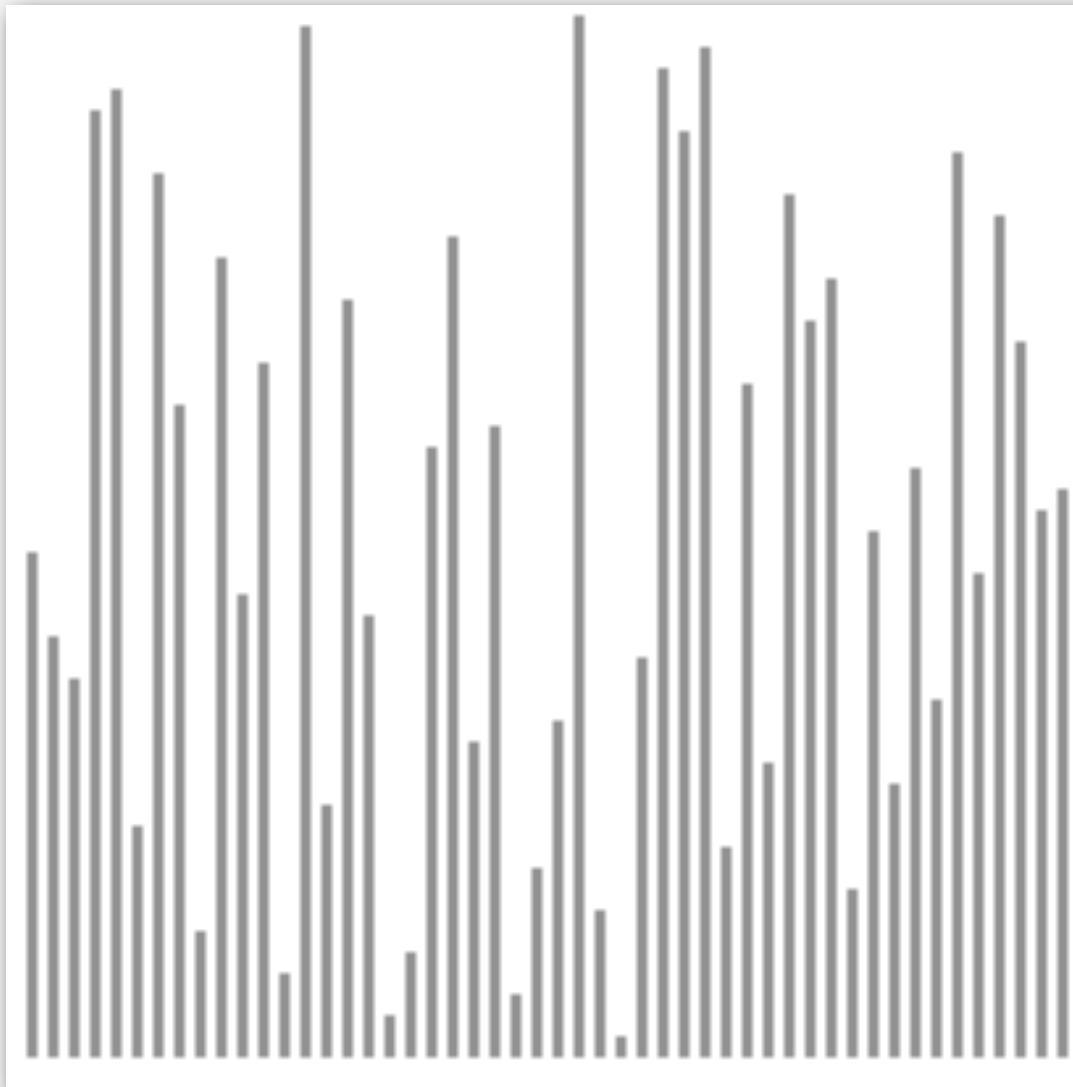
---



# Shellsort: animation

---

50 random items

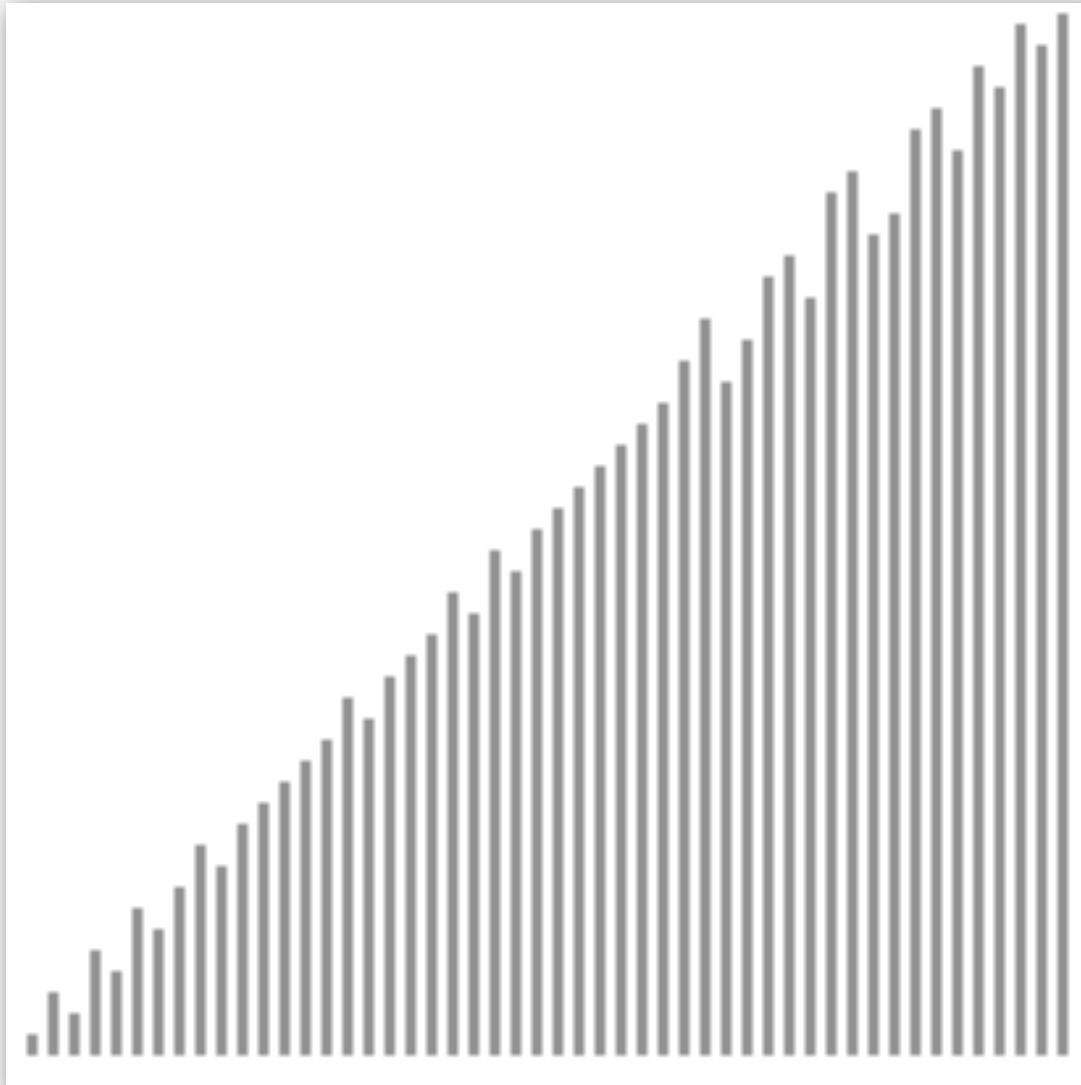


<http://www.sorting-algorithms.com/shell-sort>

- ▲ algorithm position
- ▬ h-sorted
- ▬ current subsequence
- ▬ other elements

# Shellsort: animation

## 50 partially-sorted items



<http://www.sorting-algorithms.com/shell-sort>

- The legend consists of four entries, each with a colored bar followed by text:

  - A red triangle pointing upwards: "algorithm position"
  - A solid black horizontal bar: "h-sorted"
  - A solid grey horizontal bar: "current subsequence"
  - A solid light grey horizontal bar: "other elements"

## Shellsort: analysis

---

**Proposition.** The worst-case number of compares used by shellsort with the  $3x+1$  increments is  $O(N^{3/2})$ .

**Property.** Number of compares used by shellsort with the  $3x+1$  increments is at most by a small multiple of  $N$  times the # of increments used.

N	compares	$N^{1.289}$	$2.5 N \lg N$
5,000	93	58	106
10,000	209	143	230
20,000	467	349	495
40,000	1022	855	1059
80,000	2266	2089	2257

measured in thousands

**Remark.** Accurate model has not yet been discovered (!)

# Why are we interested in shellsort?

---

Example of simple idea leading to substantial performance gains.

## Useful in practice.

- Fast unless array size is huge (used for small subarrays).
- Tiny, fixed footprint for code (used in some embedded systems).
- Hardware sort prototype.

bzip2, /linux/kernel/groups.c

uClibc

Simple algorithm, nontrivial performance, interesting questions.

- Asymptotic growth rate?
- Best sequence of increments? ← open problem: find a better increment sequence
- Average-case performance?

Lesson. Some good algorithms are still waiting discovery.

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

---

- ▶ *rules of the game*
- ▶ *selection sort*
- ▶ *insertion sort*
- ▶ ***shellsort***
- ▶ *shuffling*
- ▶ *convex hull*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

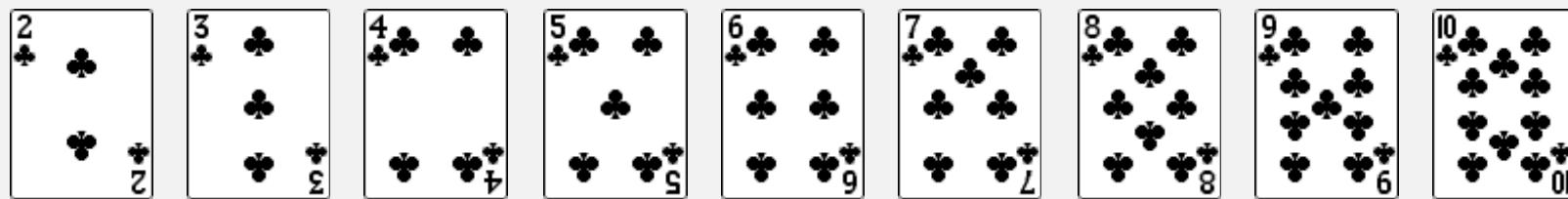
---

- ▶ *rules of the game*
- ▶ *selection sort*
- ▶ *insertion sort*
- ▶ *shellsort*
- ▶ ***shuffling***
- ▶ *convex hull*

# How to shuffle an array

---

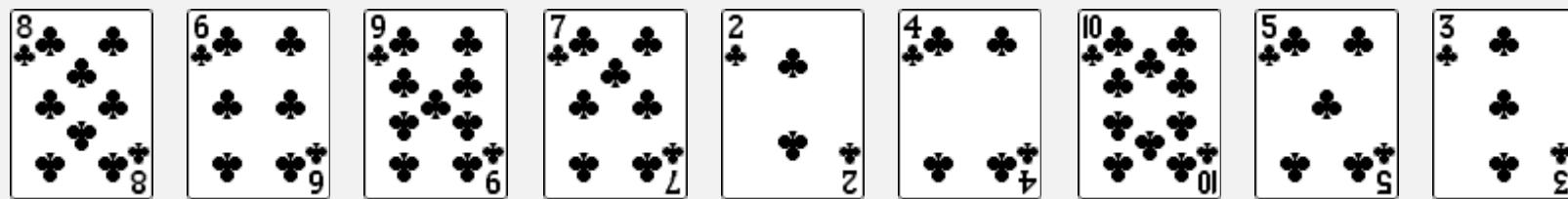
**Goal.** Rearrange array so that result is a uniformly random permutation.



# How to shuffle an array

---

**Goal.** Rearrange array so that result is a uniformly random permutation.

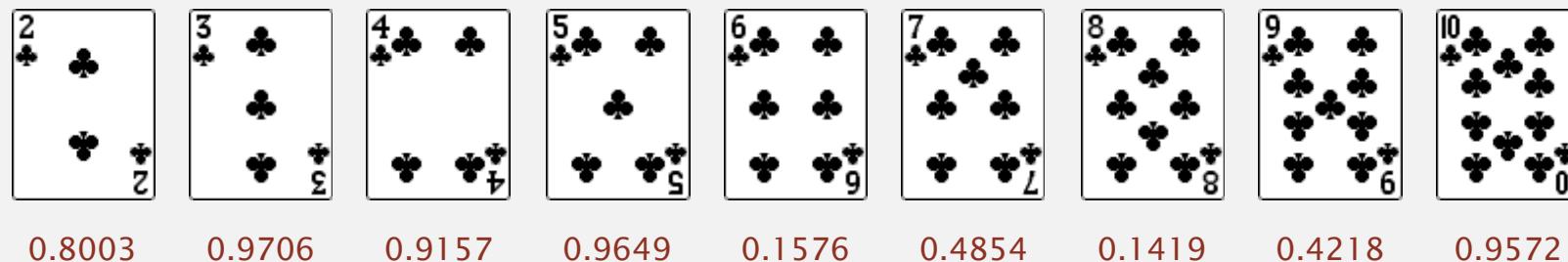


# Shuffle sort

---

- Generate a random real number for each array entry.
- Sort the array.

useful for shuffling  
columns in a spreadsheet

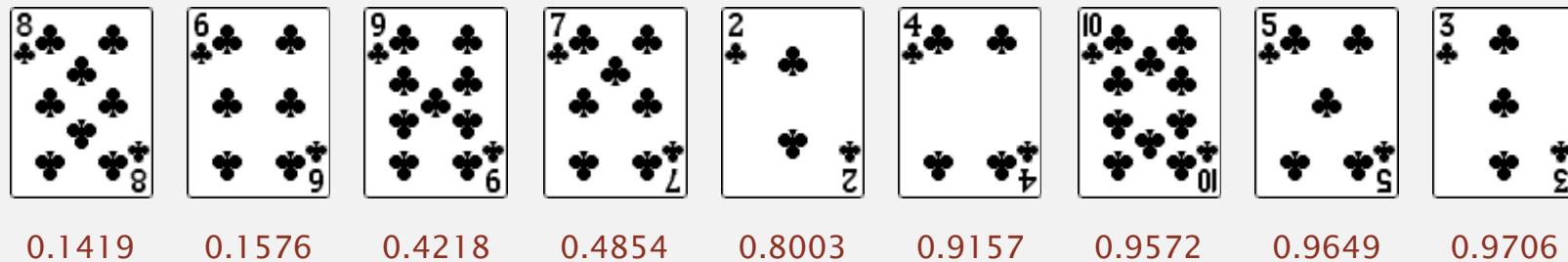


# Shuffle sort

---

- Generate a random real number for each array entry.
- Sort the array.

useful for shuffling  
columns in a spreadsheet

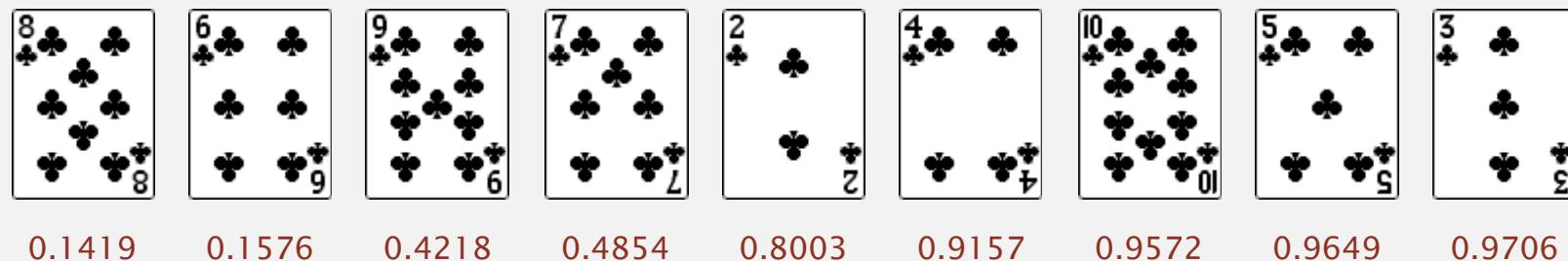


# Shuffle sort

---

- Generate a random real number for each array entry.
- Sort the array.

useful for shuffling  
columns in a spreadsheet



**Proposition.** Shuffle sort produces a uniformly random permutation  
of the input array, provided no duplicate values.

assuming real numbers  
uniformly at random

# War story (Microsoft)

Microsoft antitrust probe by EU. Microsoft agreed to provide a randomized ballot screen for users to select browser in Windows 7.

<http://www.browserchoice.eu>

## Select your web browser(s)



Google  
chrome

A fast new browser from Google. Try it now!



Safari

Safari for Windows from Apple, the world's most innovative browser.



mozilla  
Firefox

Your online security is Firefox's top priority. Firefox is free, and made to help you get the most out of the



Opera™  
browser

The fastest browser on Earth. Secure, powerful and easy to use, with excellent privacy protection.



Windows®  
Internet Explorer® 8

Designed to help you take control of your privacy and browse with confidence. Free from Microsoft.



appeared last 50% of the time

## War story (Microsoft)

---

Microsoft antitrust probe by EU. Microsoft agreed to provide a randomized ballot screen for users to select browser in Windows 7.

**Solution?** Implement shuffle sort by making comparator always return a random answer.

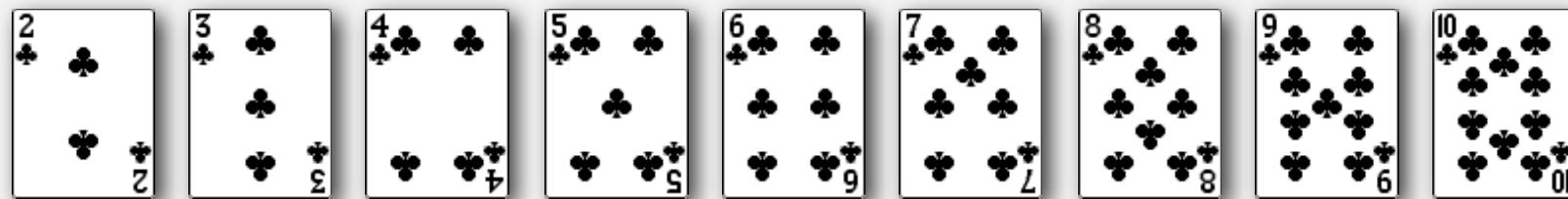
```
public int compareTo(Browser that)
{
    double r = Math.random();
    if (r < 0.5) return -1;
    if (r > 0.5) return +1;
    return 0;
}
```

← browser comparator  
(should implement a total order)

## Knuth shuffle demo

---

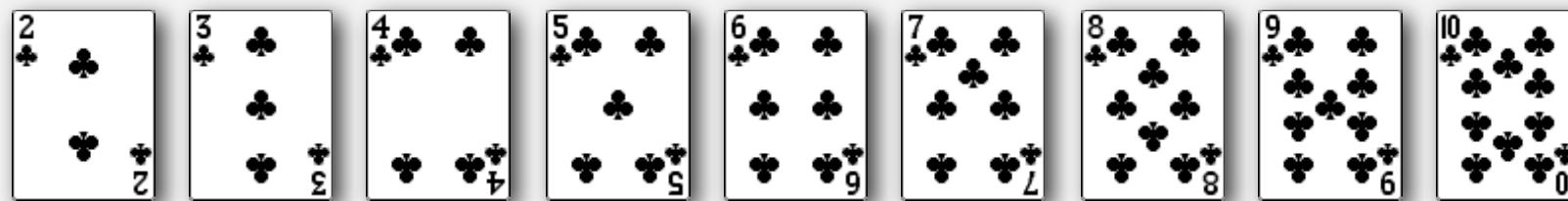
- In iteration  $i$ , pick integer  $r$  between 0 and  $i$  uniformly at random.
- Swap  $a[i]$  and  $a[r]$ .



## Knuth shuffle

---

- In iteration  $i$ , pick integer  $r$  between 0 and  $i$  uniformly at random.
- Swap  $a[i]$  and  $a[r]$ .



**Proposition.** [Fisher-Yates 1938] Knuth shuffling algorithm produces a uniformly random permutation of the input array in linear time.

→ assuming integers  
uniformly at random

# Knuth shuffle

---

- In iteration  $i$ , pick integer  $r$  between 0 and  $i$  uniformly at random.
- Swap  $a[i]$  and  $a[r]$ .

common bug: between 0 and  $N - 1$   
correct variant: between  $i$  and  $N - 1$

```
public class StdRandom
{
    ...
    public static void shuffle(Object[] a)
    {
        int N = a.length;
        for (int i = 0; i < N; i++)
        {
            int r = StdRandom.uniform(i + 1);           ← between 0 and i
            exch(a, i, r);
        }
    }
}
```

# War story (online poker)

Texas hold'em poker. Software must shuffle electronic cards.



How We Learned to Cheat at Online Poker: A Study in Software Security

<http://www.datamation.com/entdev/article.php/616221>

# War story (online poker)

---

Shuffling algorithm in FAQ at [www.planetpoker.com](http://www.planetpoker.com)

```
for i := 1 to 52 do begin
    r := random(51) + 1;           ← between 1 and 51
    swap := card[r];
    card[r] := card[i];
    card[i] := swap;
end;
```

- Bug 1. Random number r never 52  $\Rightarrow$  52<sup>nd</sup> card can't end up in 52<sup>nd</sup> place.
- Bug 2. Shuffle not uniform (should be between 1 and i).
- Bug 3. random() uses 32-bit seed  $\Rightarrow$   $2^{32}$  possible shuffles.
- Bug 4. Seed = milliseconds since midnight  $\Rightarrow$  86.4 million shuffles.

“The generation of random numbers is too important to be left to chance.”

— Robert R. Coveyou

## War story (online poker)

---

Best practices for shuffling (if your business depends on it).

- Use a hardware random-number generator that has passed both the FIPS 140-2 and the NIST statistical test suites.
- Continuously monitor statistic properties: hardware random-number generators are fragile and fail silently.
- Use an unbiased shuffling algorithm.



Bottom line. Shuffling a deck of cards is hard!

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

---

- ▶ *rules of the game*
- ▶ *selection sort*
- ▶ *insertion sort*
- ▶ *shellsort*
- ▶ ***shuffling***
- ▶ *convex hull*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

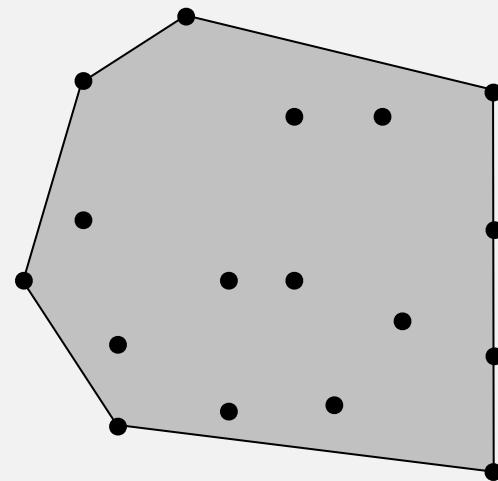
---

- ▶ *rules of the game*
- ▶ *selection sort*
- ▶ *insertion sort*
- ▶ *shellsort*
- ▶ *shuffling*
- ▶ ***convex hull***

# Convex hull

---

The **convex hull** of a set of  $N$  points is the smallest perimeter fence enclosing the points.



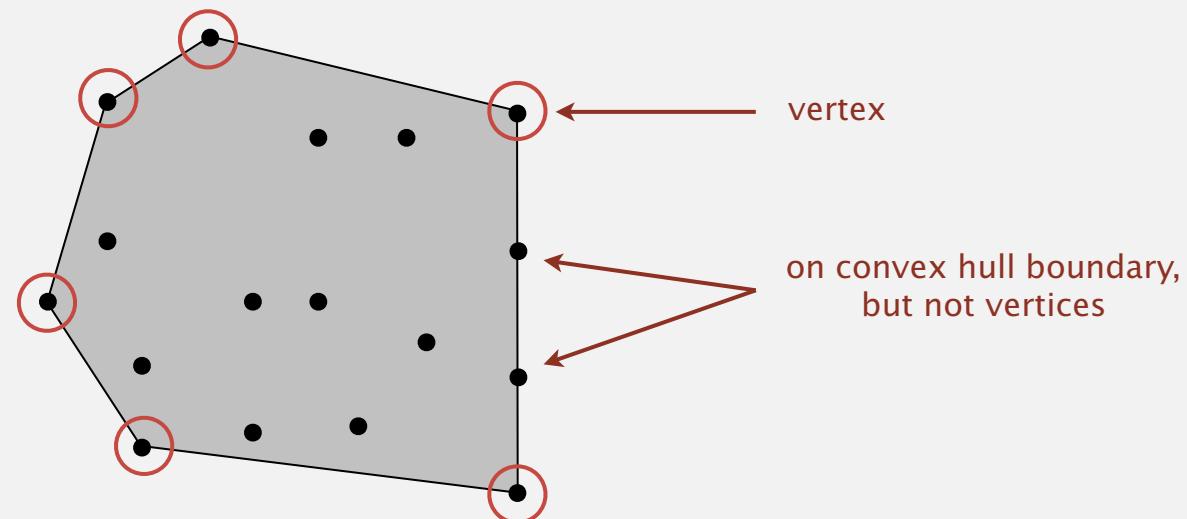
## Equivalent definitions.

- Smallest convex set containing all the points.
- Smallest area convex polygon enclosing the points.
- Convex polygon enclosing the points, whose vertices are points in set.

# Convex hull

---

The **convex hull** of a set of  $N$  points is the smallest perimeter fence enclosing the points.

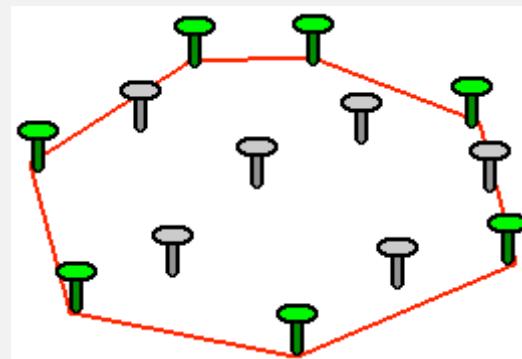


**Convex hull output.** Sequence of vertices in counterclockwise order.

## Convex hull: mechanical algorithm

---

Mechanical algorithm. Hammer nails perpendicular to plane; stretch elastic rubber band around points.

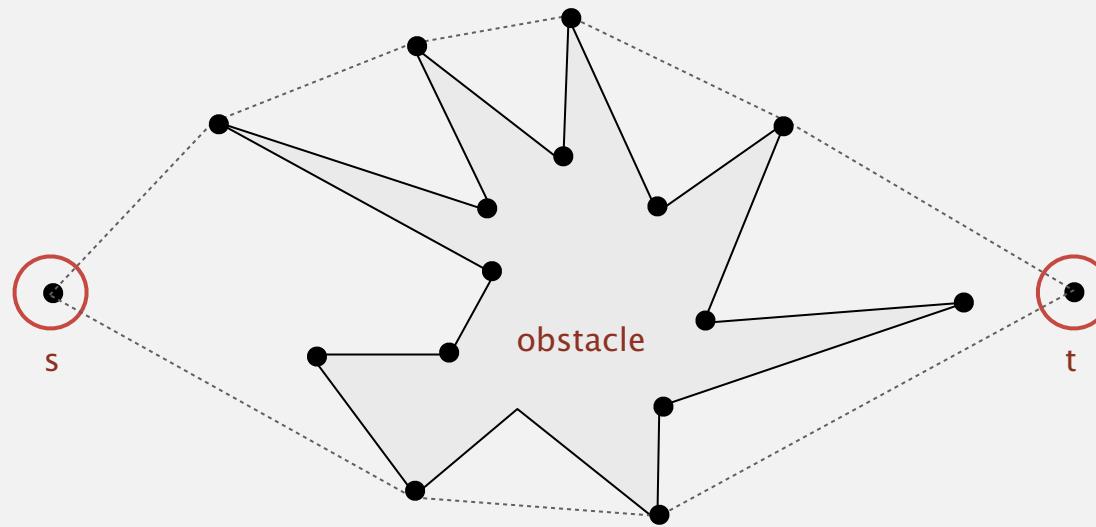


[http://www.idlcoyote.com/math\\_tips/convexhull.html](http://www.idlcoyote.com/math_tips/convexhull.html)

## Convex hull application: motion planning

---

Robot motion planning. Find shortest path in the plane from  $s$  to  $t$  that avoids a polygonal obstacle.

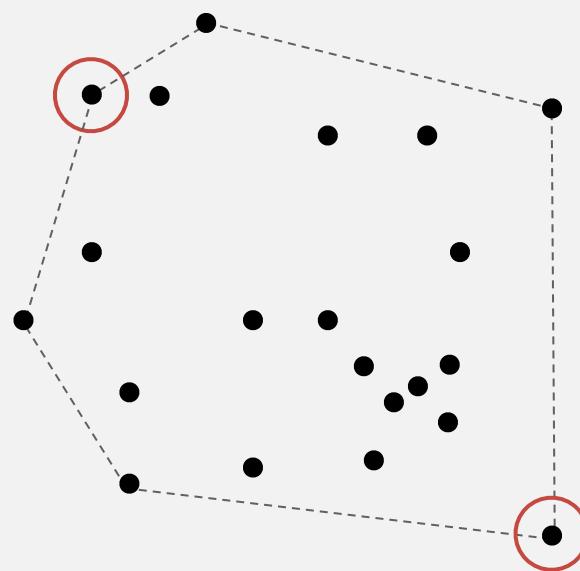


Fact. Shortest path is either straight line from  $s$  to  $t$  or it is one of two polygonal chains of convex hull.

## Convex hull application: farthest pair

---

**Farthest pair problem.** Given  $N$  points in the plane, find a pair of points with the largest Euclidean distance between them.



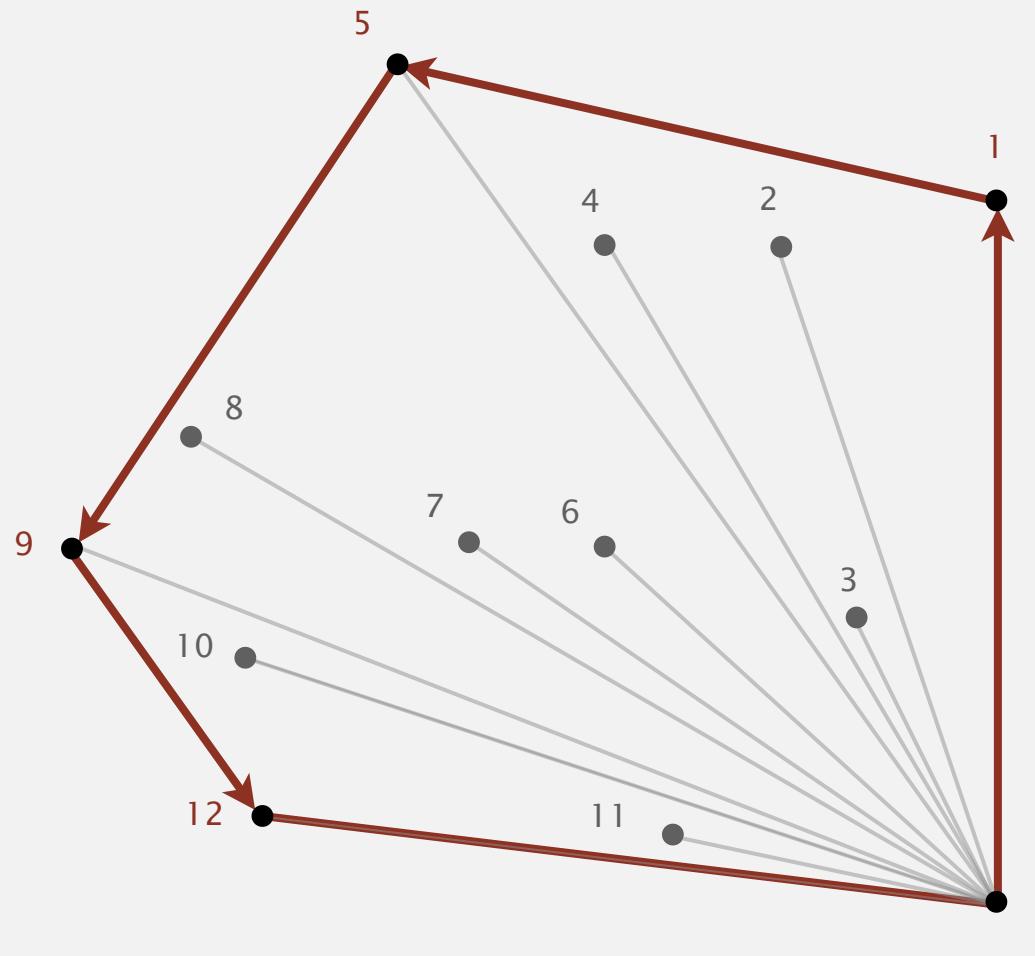
**Fact.** Farthest pair of points are extreme points on convex hull.

## Convex hull: geometric properties

---

**Fact.** Can traverse the convex hull by making only counterclockwise turns.

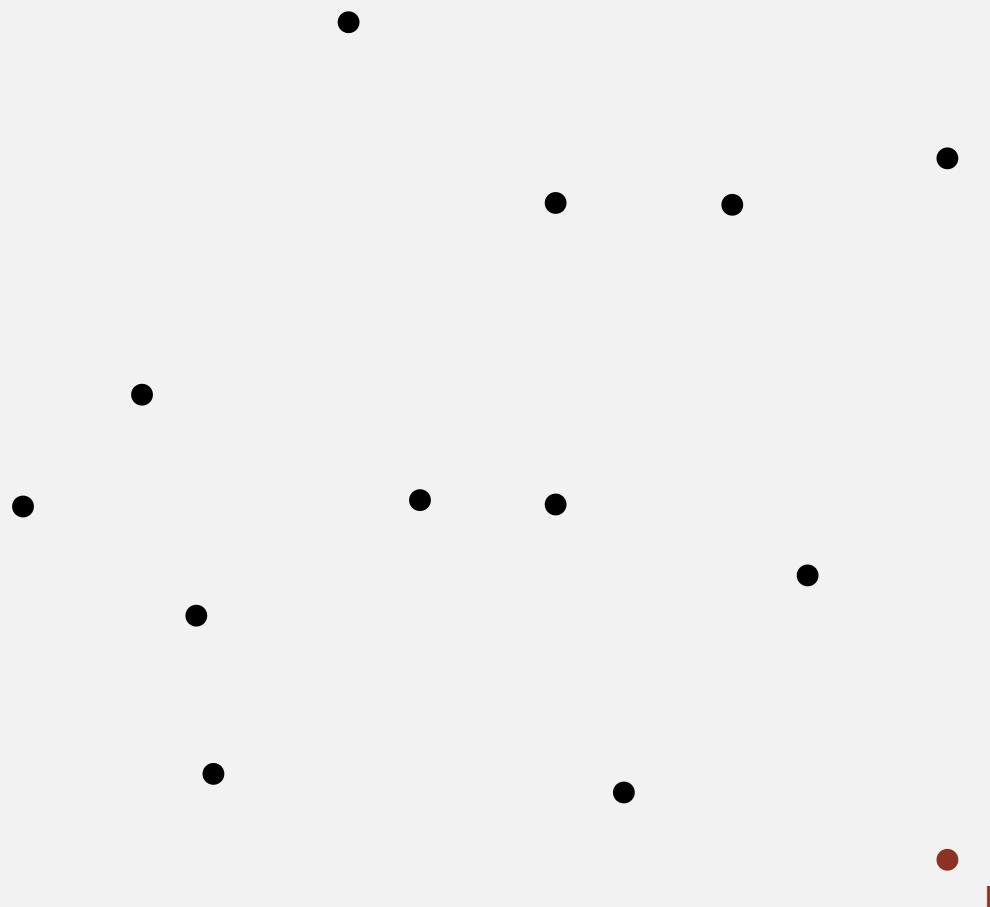
**Fact.** The vertices of convex hull appear in increasing order of polar angle with respect to point  $p$  with lowest  $y$ -coordinate.



## Graham scan demo

---

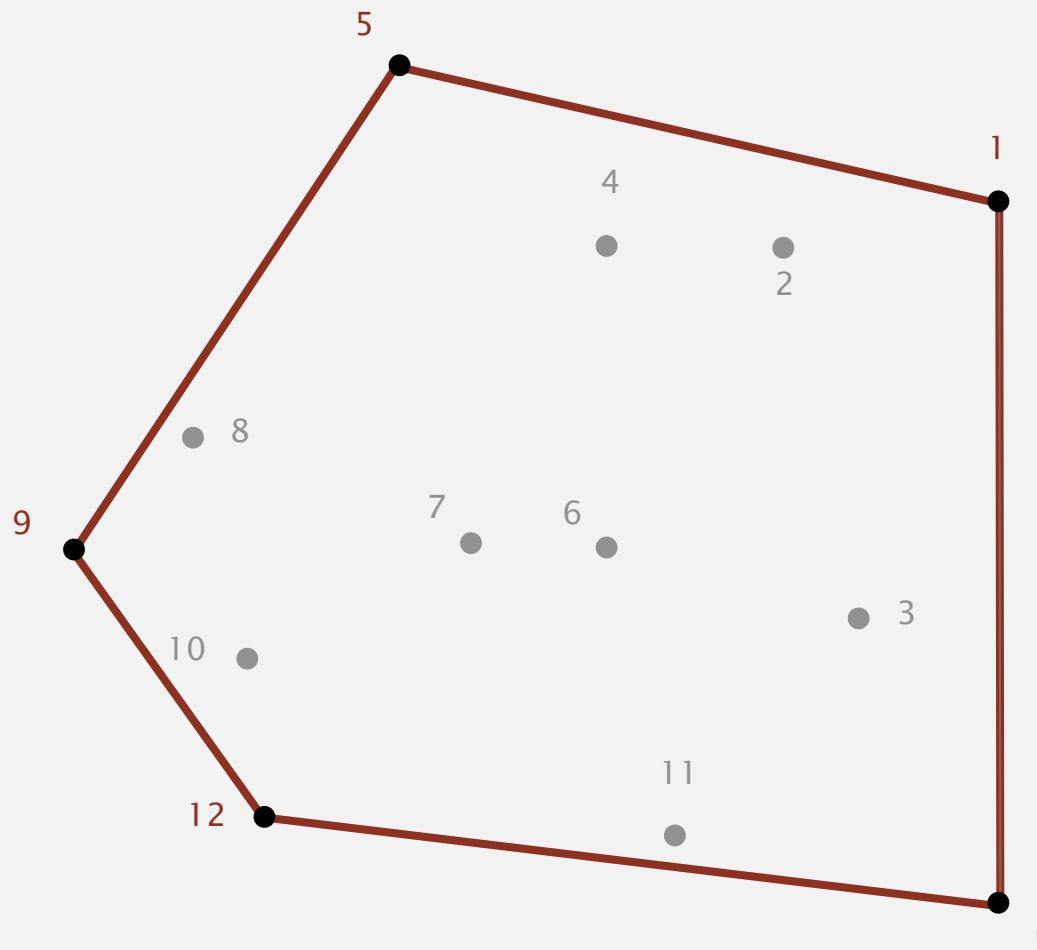
- Choose point  $p$  with smallest  $y$ -coordinate.
- Sort points by polar angle with  $p$ .
- Consider points in order; discard unless it create a ccw turn.



## Graham scan demo

---

- Choose point  $p$  with smallest  $y$ -coordinate.
- Sort points by polar angle with  $p$ .
- Consider points in order; discard unless it creates a ccw turn.



## Graham scan: implementation challenges

---

Q. How to find point  $p$  with smallest  $y$ -coordinate?

A. Define a total order, comparing by  $y$ -coordinate. [next lecture]

Q. How to sort points by polar angle with respect to  $p$  ?

A. Define a total order **for each** point  $p$ . [next lecture]

Q. How to determine whether  $p_1 \rightarrow p_2 \rightarrow p_3$  is a counterclockwise turn?

A. Computational geometry. [next two slides]

Q. How to sort efficiently?

A. Mergesort sorts in  $N \log N$  time. [next lecture]

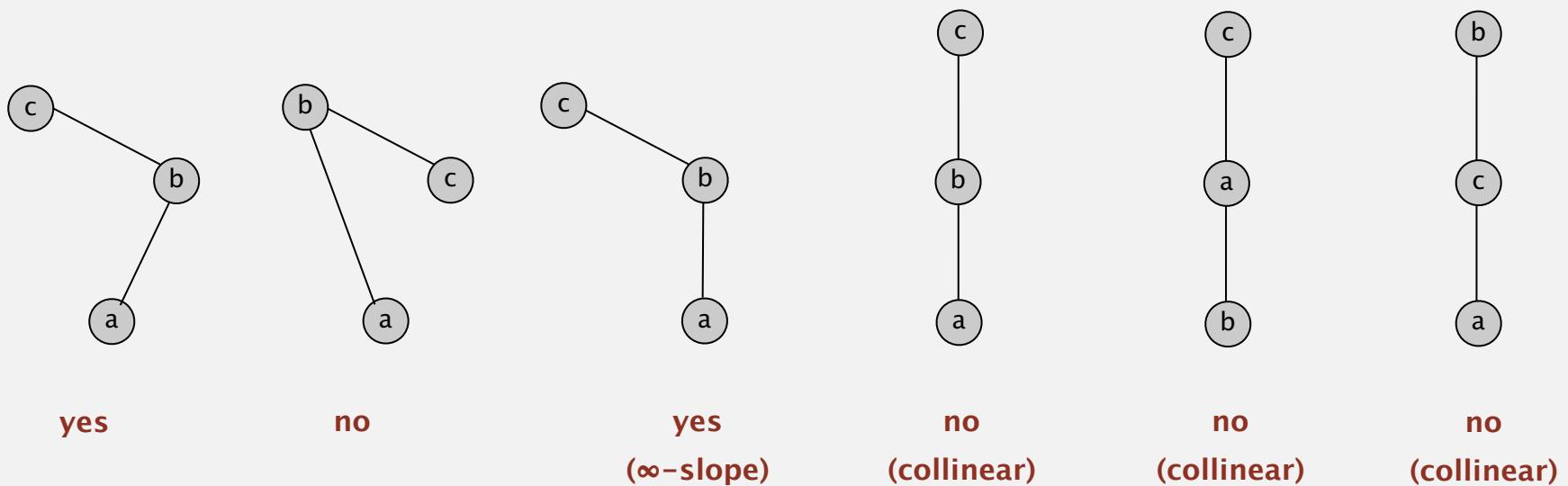
Q. How to handle degeneracies (three or more points on a line)?

A. Requires some care, but not hard. [see booksite]

## Implementing ccw

CCW. Given three points  $a$ ,  $b$ , and  $c$ , is  $a \rightarrow b \rightarrow c$  a counterclockwise turn?

is  $c$  to the left of the ray  $a \rightarrow b$



Lesson. Geometric primitives are tricky to implement.

- Dealing with degenerate cases.
- Coping with floating-point precision.

## Implementing ccw

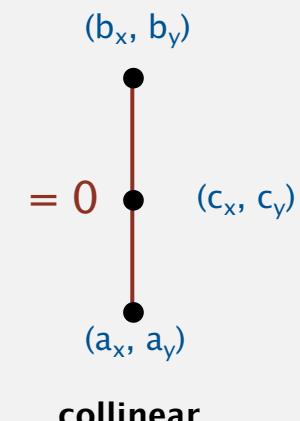
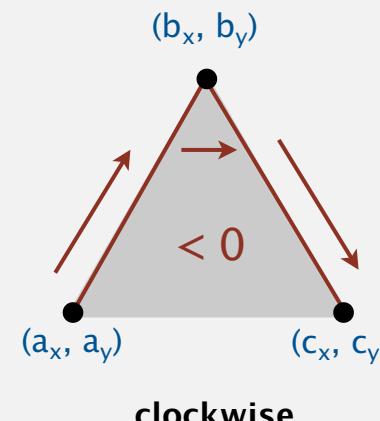
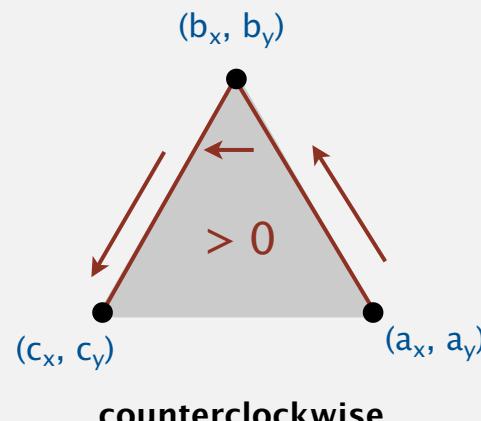
**CCW.** Given three points  $a$ ,  $b$ , and  $c$ , is  $a \rightarrow b \rightarrow c$  a counterclockwise turn?

- Determinant (or cross product) gives 2x signed area of planar triangle.

$$2 \times \text{Area}(a, b, c) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

$(b - a) \times (c - a)$

- If signed area  $> 0$ , then  $a \rightarrow b \rightarrow c$  is counterclockwise.
- If signed area  $< 0$ , then  $a \rightarrow b \rightarrow c$  is clockwise.
- If signed area  $= 0$ , then  $a \rightarrow b \rightarrow c$  are collinear.



## Immutable point data type

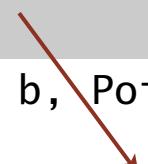
```
public class Point2D
{
    private final double x;
    private final double y;

    public Point2D(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    ...

    public static int ccw(Point2D a, Point2D b, Point2D c)
    {
        double area2 = (b.x-a.x)*(c.y-a.y) - (b.y-a.y)*(c.x-a.x);
        if      (area2 < 0) return -1; // clockwise
        else if (area2 > 0) return +1; // counter-clockwise
        else                  return 0; // collinear
    }
}
```

danger of  
floating-point  
roundoff error



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

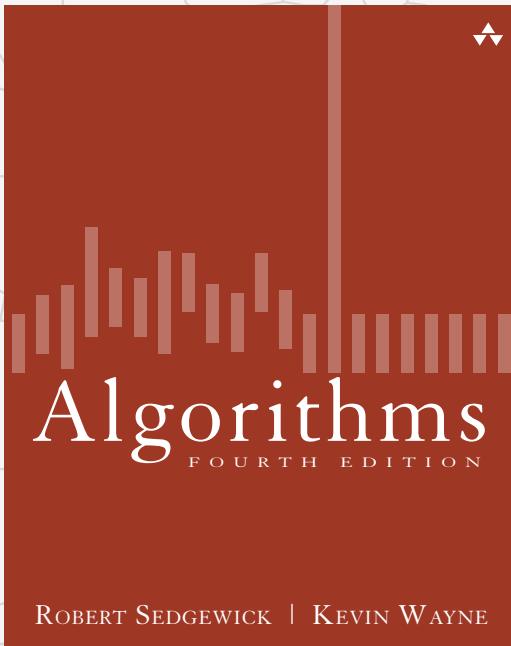
## 2.1 ELEMENTARY SORTS

---

- ▶ *rules of the game*
- ▶ *selection sort*
- ▶ *insertion sort*
- ▶ *shellsort*
- ▶ *shuffling*
- ▶ ***convex hull***

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



<http://algs4.cs.princeton.edu>

## 2.1 ELEMENTARY SORTS

---

- ▶ *rules of the game*
- ▶ *selection sort*
- ▶ *insertion sort*
- ▶ *shellsort*
- ▶ *shuffling*
- ▶ *convex hull*