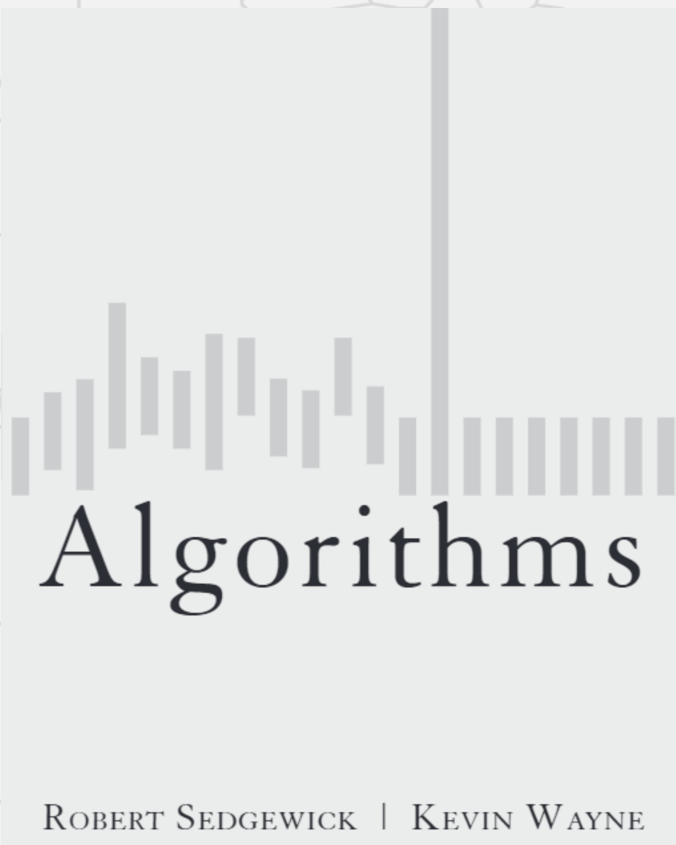

http://algs4.cs.princeton.edu

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, for each operation.
- Array accesses to insert first N items = $\underbrace{N}_{\text{1 array access per push}} + (\underbrace{2 + 4 + \ldots + 2(N-1)}_{\text{2(k–1) array accesses to expand to size k (ignoring cost to create new array)}}) \sim \overbrace{N^2}^{\text{infeasible for large N}}$.

Challenge. Ensure that array resizing happens infrequently.

Stack: resizing-array implementation

"repeated doubling"

Q. How to grow array?

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

```
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;
}
```

Array accesses to insert first N = 2^i items. $N + (2 + 4 + 8 + \ldots + N) \sim 3N.$

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: performance

Amortized analysis. Starting from an empty data structure, 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

doubling and
halving operations

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

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;   ←—— 8 bytes × array size
   private int N = 0;
   …
}
```

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.