CSE 143
Lecture 5

More Stacks and Queues;
Complexity (Big-Oh)

reading: 13.1 - 13.3

slides adapted from Marty Stepp and Hélène Martin
http://www.cs.washington.edu/143/

Stack/queue exercise

- A *postfix expression* is a mathematical expression but with the operators written after the operands rather than before.

  `1 + 1` becomes `1 1 +`

  `1 + 2 * 3 + 4` becomes `1 2 3 * + 4 +`

  - supported by many kinds of fancy calculators
  - never need to use parentheses
  - never need to use an = character to evaluate on a calculator

- Write a method `postfixEvaluate` that accepts a postfix expression string, evaluates it, and returns the result.
  - All operands are integers; legal operators are `+` , `-`, `*`, and `/`

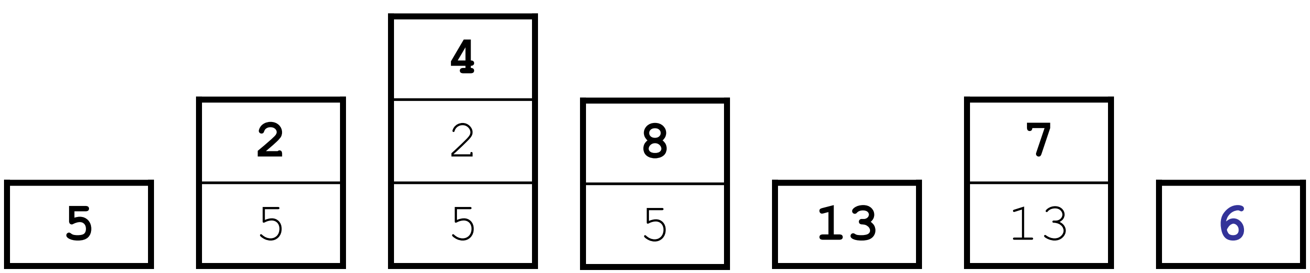
  `postFixEvaluate("5 2 4 * + 7 -")` returns `6`

Postfix algorithm

- The algorithm: Use a **stack**
  - When you see an operand, push it onto the stack.
  - When you see an operator:
    - pop the last two operands off of the stack.
    - apply the operator to them.
    - push the result onto the stack.
  - When you're done, the one remaining stack element is the result.

`"5 2 4 * + 7 -"`

`5      2      4      *      +      7      -`

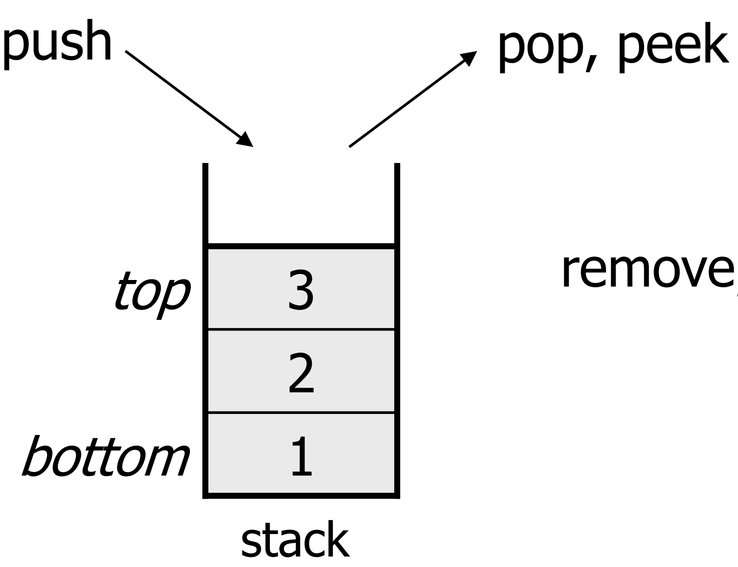


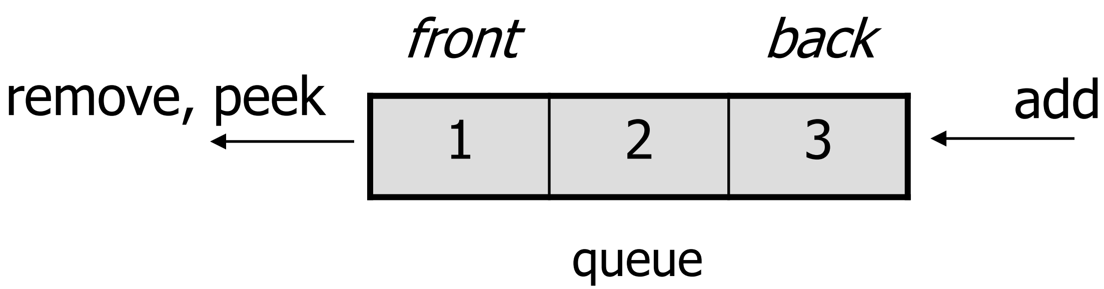
Exercise solution

```
// Evaluates the given prefix expression and returns its result.
// Precondition: string represents a legal postfix expression
public static int postfixEvaluate(String expression) {
    Stack<Integer> s = new Stack<Integer>();
    Scanner input = new Scanner(expression);
    while (input.hasNext()) {
        if (input.hasNextInt()) {      // an operand (integer)
            s.push(input.nextInt());
        } else {                       // an operator
            String operator = input.next();
            int operand2 = s.pop();
            int operand1 = s.pop();
            if (operator.equals("+")) {
                s.push(operand1 + operand2);
            } else if (operator.equals("-")) {
                s.push(operand1 - operand2);
            } else if (operator.equals("*")) {
                s.push(operand1 * operand2);
            } else {
                s.push(operand1 / operand2);
            }
        }
    }
    return s.pop();
}
```

Stack/queue motivation

- Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.
- Stacks and queues do few things, but they do them efficiently.





Runtime Efficiency (13.2)

- **efficiency**: A measure of the use of computing resources by code.
  - can be relative to speed (time), memory (space), etc.
  - most commonly refers to run time

- Assume the following:
  - Any single Java statement takes the same amount of time to run.
  - A method call's runtime is measured by the total of the statements inside the method's body.
  - A loop's runtime, if the loop repeats N times, is N times the runtime of the statements in its body.

Efficiency examples

```
statement1;
statement2;
statement3;
```

3

```
for (int i = 1; i <= N; i++) {
    statement4;
}
```

N

```
for (int i = 1; i <= N; i++) {
    statement5;
    statement6;
    statement7;
}
```

3N

4N + 3

Efficiency examples 2

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        statement1;
    }
}

for (int i = 1; i <= N; i++) {
    statement2;
    statement3;
    statement4;
    statement5;
}
```

N^2

$4N$

$N^2 + 4N$

- How many statements will execute if N = 10? If N = 1000?

Algorithm growth rates (13.2)

- We measure runtime in proportion to the input data size, N.
  - **growth rate**: Change in runtime as N changes.

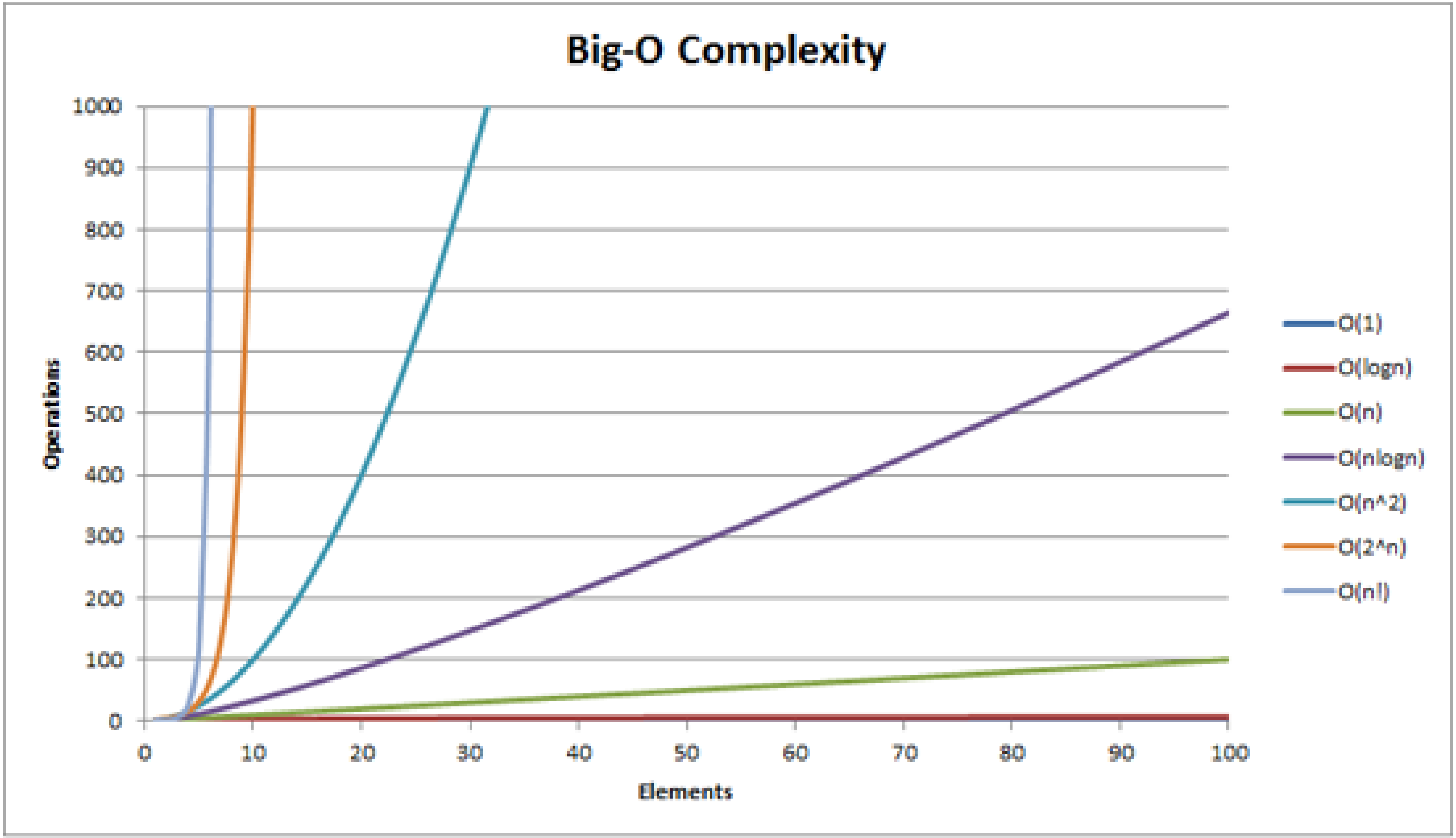
- Say an algorithm runs $\mathbf{0.4N^3 + 25N^2 + 8N + 17}$ statements.
  - Consider the runtime when N is *extremely large* .
  - We ignore constants like 25 because they are tiny next to N.
  - The highest-order term (N^3) dominates the overall runtime.
  - We say that this algorithm runs "on the order of" N^3.
  - or $\mathbf{O(N^3)}$ for short ("**Big-Oh** of N cubed")

Complexity classes

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Class	Big-Oh	If you double N, ...	Example
constant	$O(1)$	unchanged	10ms
logarithmic	$O(\log_2 N)$	increases slightly	175ms
linear	$O(N)$	doubles	3.2 sec
log-linear	$O(N \log_2 N)$	slightly more than doubles	6 sec
quadratic	$O(N^2)$	quadruples	1 min 42 sec
cubic	$O(N^3)$	multiplies by 8	55 min
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	$5 * 10^{61}$ years

Complexity classes



From http://recursive-design.com/blog/2010/12/07/comp-sci-101-big-o-notation/ - post about a Google interview

Collection efficiency

- Efficiency of various operations on different collections:

Method	ArrayList	SortedIntList	Stack	Queue
`add` (or `push`)	O(1)	O(N)	O(1)	O(1)
`add(`**index**`,` **value**`)`	O(N)		-	-
`indexOf`	O(N)	O(?)	-	-
`get`	O(1)	O(1)	-	-
`remove`	O(N)	O(N)	O(1)	O(1)
`set`	O(1)	O(1)	-	-
`size`	O(1)	O(1)	O(1)	O(1)

Sequential search

- What is its complexity class?

```
public int indexOf(int value) {
    for (int i = 0; i < size; i++) {
        if (elementData[i] == value) {
            return i;
        }
    }
    return -1;    // not found
}
```

N

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

- On average, "only" N/2 elements are visited
  - 1/2 is a constant that can be ignored

Binary search

- **binary search** successively eliminates half of the elements.
  - *Algorithm:* Examine the middle element of the array.
    - If it is too big, eliminate the right half of the array and repeat.
    - If it is too small, eliminate the left half of the array and repeat.
    - Else it is the value we're searching for, so stop.

  - Which indexes does the algorithm examine to find value **42**?
  - What is the runtime complexity class of binary search?

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

min mid max

Binary search runtime

- For an array of size N, it eliminates ½ until 1 element remains.

  N, N/2, N/4, N/8, ..., 4, 2, 1

  – How many divisions does it take?

- Think of it from the other direction:
  – How many times do I have to multiply by 2 to reach N?

  1, 2, 4, 8, ..., N/4, N/2, N

  – Call this number of multiplications "x".

  $2^x = N$

  $\mathbf{x = \log_2 N}$

- Binary search is in the **logarithmic** complexity class.

Max subsequence sum

- Write a method `maxSum` to find the largest sum of any contiguous subsequence in an array of integers.
  - Easy for all positives: include the whole array.
  - What if there are negatives?

index	0	1	2	3	4	5	6	7	8
value	2	1	-4	10	15	-2	22	-8	5

Largest sum: 10 + 15 + -2 + 22 = 45

  - (Let's define the max to be 0 if the array is entirely negative.)

- Ideas for algorithms?

Algorithm 1 pseudocode

```
maxSum(a):
    max = 0.
    for each starting index i:
        for each ending index j:
            sum = add the elements from a[i] to a[j].
            if sum > max,
               max = sum.

    return max.
```

index	0	1	2	3	4	5	6	7	8
value	2	1	-4	10	15	-2	22	-8	5

Algorithm 1 code

- What complexity class is this algorithm?
  - **$O(N^3)$**. Takes a few seconds to process 2000 elements.

```
public static int maxSum1(int[] a) {
    int max = 0;
    for (int i = 0; i < a.length; i++) {
        for (int j = i; j < a.length; j++) {
            // sum = add the elements from a[i] to a[j].
            int sum = 0;
            for (int k = i; k <= j; k++) {
                sum += a[k];
            }
            if (sum > max) {
                max = sum;
            }
        }
    }
    return max;
}
```

Flaws in algorithm 1

- Observation: We are redundantly re-computing sums.
  - For example, we compute the sum between indexes 2 and 5:
    a[2] + a[3] + a[4] + a[5]
  - Next we compute the sum between indexes 2 and 6:
    a[2] + a[3] + a[4] + a[5] + a[6]
  - We already had computed the sum of 2-5, but we compute it again as part of the 2-6 computation.
  - Let's write an improved version that avoids this flaw.

Algorithm 2 code

- What complexity class is this algorithm?
  - **$O(N^2)$**. Can process tens of thousands of elements per second.

```
public static int maxSum2(int[] a) {
    int max = 0;
    for (int i = 0; i < a.length; i++) {
        int sum = 0;
        for (int j = i; j < a.length; j++) {
            sum += a[j];
            if (sum > max) {
                max = sum;
            }
        }
    }
    return max;
}
```

A clever solution

- *Claim 1* : A max range cannot start with a negative-sum range.

i ... j	j+1 ... k
< 0	sum(j+1, k)
sum(i, k) < sum(j+1, k)	

- *Claim 2* : If sum(i, j-1) ≥ 0 and sum(i, j) < 0, any max range that ends at j+1 or higher cannot start at any of i through j.

i ... j-1	j	j+1 ... k
≥ 0	< 0	sum(j+1, k)
< 0		sum(j+1, k)
		sum(?, k) < sum(j+1, k)

  – Together, these observations lead to a very clever algorithm...

Algorithm 3 code

- What complexity class is this algorithm?
  - **O(N)**. Handles many millions of elements per second!

```
public static int maxSum3(int[] a) {
    int max = 0;
    int sum = 0;
    int i = 0;
    for (int j = 0; j < a.length; j++) {
        if (sum < 0) {    // if sum becomes negative, max range
            i = j;        // cannot start with any of i - j-1
            sum = 0;      // (Claim 2)
        }
        sum += list[j];
        if (sum > max) {
            max = sum;
        }
    }
    return max;
}
```