Analyzing efficiency

- **efficiency**: A measure of the use of computing resources by code.
  - most commonly refers to run time; but could be memory, etc.

- Rather than writing and timing algorithms, let's *analyze* them. Code is hard to analyze, so let's make the following assumptions:
  - Any *single Java statement* takes a constant amount of time to run.
  - The runtime of a *sequence* of statements is the sum of their runtimes.
  - An *if/else*'s runtime is the runtime of the if test, plus the runtime of whichever branch of code is chosen.
  - A *loop*'s runtime, if the loop repeats *N* times, is *N* times the runtime of the statements in its body.
  - A *method call*'s runtime is measured by the total of the statements inside the method's body.

Runtime example

```
statement1;
statement2;
```

2

```
for (int i = 1; i <= N; i++) {
    statement3;
    statement4;
    statement5;
    statement6;
}
```

$4N$

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N/2; j++) {
        statement7;
    }
}
```

$\frac{1}{2} N^2$

$\frac{1}{2} N^2 + 4N + 2$

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

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

Java collection efficiency

Method	Array List	Linked List	Stack	Queue	TreeSet /Map	[Linked] HashSet /Map	Priority Queue
`add` or `put`	O(1)	O(1)	O(1)*	O(1)*	O(log *N*)	O(1)	O(log *N*)*
`add` at index	O(*N*)	O(*N*)	-	-	-	-	-
`contains`/ `indexOf`	O(*N*)	O(*N*)	-	-	O(log *N*)	O(1)	-
`get`/`set`	O(1)	O(*N*)	O(1)*	O(1)*	-	-	O(1)*
`remove`	O(*N*)	O(*N*)	O(1)*	O(1)*	O(log *N*)	O(1)	O(log *N*)*
`size`	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

- * = operation can only be applied to certain element(s) / places

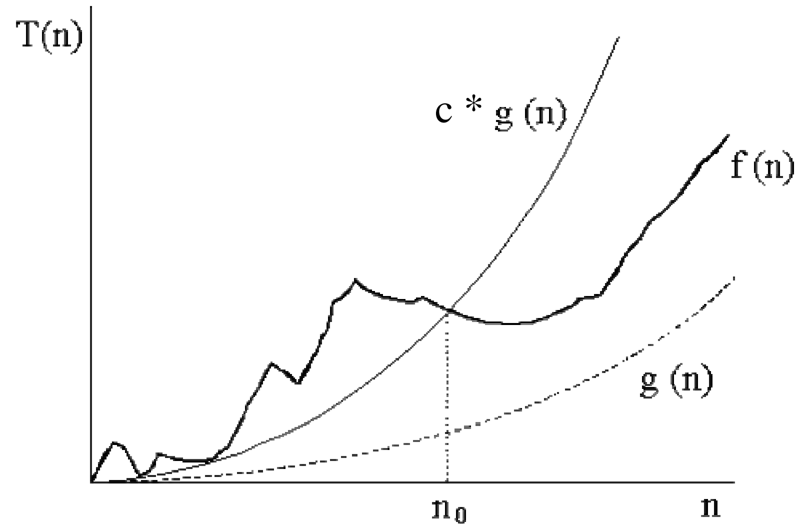
Big-Oh defined

- Big-Oh is about finding an *asymptotic upper bound*.
- Formal definition of Big-Oh:

  $f(N) = O(g(N))$, if there exists positive constants c , N_0 such that

  $f(N) \leq c \cdot g(N)$ for all $N \geq N_0$.

  - We are concerned with how f grows when N is large.
    - not concerned with small N or constant factors
  - Lingo: "$f(N)$ grows no faster than $g(N)$."



Big-Oh questions

- $N + 2 = O(N)$?
  - yes
- $2N = O(N)$?
  - yes
- $N = O(N^2)$?
  - yes
- $N^2 = O(N)$?
  - no
- $100 = O(N)$?
  - yes
- $N = O(1)$?
  - no
- $214N + 34 = O(N^2)$?
  - yes

Preferred Big-Oh usage

- Pick the tightest bound. If $f(N) = 5N$, then:

  $f(N) = O(N^5)$

  $f(N) = O(N^3)$

  $f(N) = O(N \log N)$

  $f(N) = O(N)$ $\leftarrow$ preferred

- Ignore constant factors and low order terms:

  $f(N) = O(N)$, *not* $f(N) = O(5N)$

  $f(N) = O(N^3)$, *not* $f(N) = O(N^3 + N^2 + 15)$

  - Wrong: $f(N) \leq O(g(N))$
  - Wrong: $f(N) \geq O(g(N))$
  - Right: $f(N) = O(g(N))$

A basic Big-Oh proof

- *Claim*: $2N + 6 = O(N)$.
- *To prove*: Must find c, N_0 such that for all $N \geq N_0$,

  $$2N + 6 \leq c \cdot N$$

- *Proof*: Let $c = 3$, $N_0 = 6$.

  $$2N + 6 \leq 3 \cdot N$$

  $$6 \leq N$$

Math background: Exponents

- Exponents:
  - X^Y , or "X to the Y^{th} power";
    X multiplied by itself Y times
- Some useful identities:
  - $X^A \cdot X^B = X^{A+B}$
  - $X^A / X^B = X^{A-B}$
  - $(X^A)^B = X^{AB}$
  - $X^N + X^N = 2X^N$
  - $2^N + 2^N = 2^{N+1}$

Math background: Logarithms

- Logarithms
  - *definition*: $X^A = B$ if and only if $\log_X B = A$
  - *intuition*: $\log_X B$ means:
    "the power X must be raised to, to get B"
  - In this course, a logarithm with no base implies base 2.
    $\log B$ means $\log_2 B$
- Examples
  - $\log_2 16 = 4$ (because $2^4 = 16$)
  - $\log_{10} 1000 = 3$ (because $10^3 = 1000$)

Logarithm bases

- Identities for logs with addition, multiplication, powers:
  - $\log (A \cdot B) = \log A + \log B$
  - $\log (A/B) = \log A - \log B$
  - $\log (A^B) = B \log A$

- Identity for converting bases of a logarithm:

$$\log_A B = \frac{\log_C B}{\log_C A}$$

  - example:
    $\log_4 32 = (\log_2 32) / (\log_2 4)$
    $= 5 / 2$
  - Practically speaking, this means all $\log_c$ are a constant factor away from $\log_2$, so we can think of them as equivalent to $\log_2$ in Big-Oh analysis.

More runtime examples

- What is the exact runtime and complexity class (Big-Oh)?

```
int sum = 0;
for (int i = 1; i <= N; i += c) {
    sum++;
}
```

  - Runtime = N / c = O(N).

```
int sum = 0;
for (int i = 1; i <= N; i *= c) {
    sum++;
}
```

  - Runtime = $\log_c N$ = O(log N).

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.

  $N, N/2, N/4, N/8, \ldots, 4, 2, 1$

  - 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?

    $1, 2, 4, 8, \ldots, N/4, N/2, N$

  - Call this number of multiplications "x".

    $2^x = N$

    $\boldsymbol{x = \log_2 N}$

- Binary search is in the **logarithmic** (O(log N)) complexity class.

Math: Arithmetic series

- Arithmetic series notation *(useful for analyzing runtime of loops)*:

$$\sum_{i=j}^{k} Expr$$

  - the sum of all values of *Expr* with each value of *i* between *j*--*k*

- Example:

$$\sum_{i=0}^{4} 2i+1$$

= (2(0) + 1) + (2(1) + 1) + (2(2) + 1) + (2(3) + 1) + (2(4) + 1)
= 1 + 3 + 5 + 7 + 9
= 25

Arithmetic series identities

- sum from 1 through *N* inclusive:

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} = O(N^2)$$

- Intuition:
  - sum = 1 + 2 + 3 + ... + (*N*-2) + (*N*-1) + *N*
  - sum = (1 + *N*) + (2 + *N*-1) + (3 + *N*-2) + ... // rearranged

    // *N*/2 pairs total

- sum of squares:

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} = O(N^3)$$

Series runtime examples

- What is the exact runtime and complexity class (Big-Oh)?

```
int sum = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N * 2; j++) {
        sum++;
    }
}
```

  - Runtime = $N \cdot 2N = O(N^2)$.

```
int sum = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= i; j++) {
        sum++;
    }
}
```

  - Runtime = $N (N + 1) / 2 = O(N^2)$.