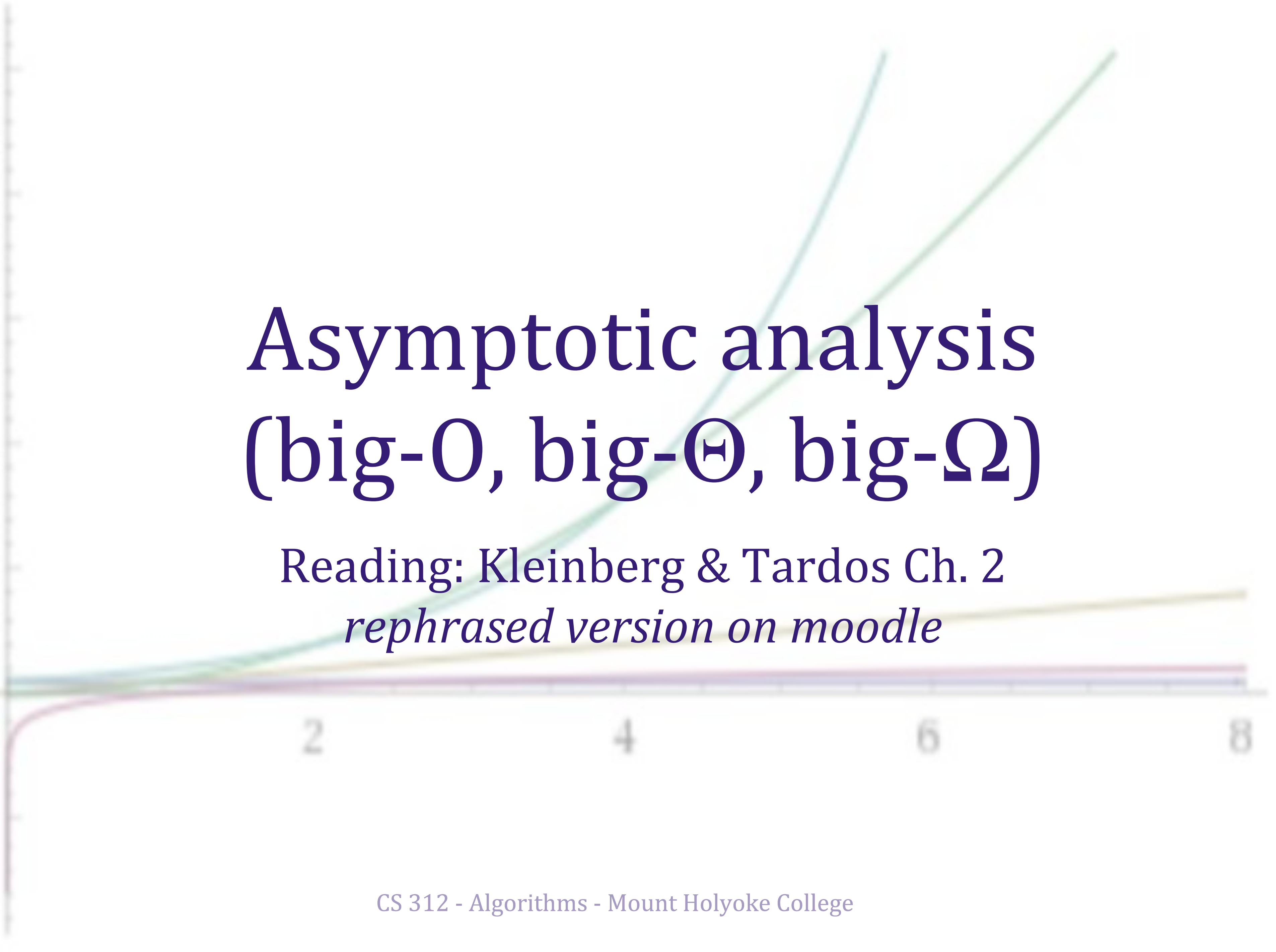


Asymptotic analysis (big-O, big-Θ, big-Ω)

Reading: Kleinberg & Tardos Ch. 2

rephrased version on moodle

Why asymptotic analysis?

- What is the running time of this algorithm?
- How many "primitive steps" are executed for an input array *A* of size *n*?

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        sum += $A[i]*A[j]$
```

Why asymptotic analysis?

- What is the running time of this algorithm?
- How many "primitive steps" are executed for an input array A of size n?

How is this stored?

How is this accessed?

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        sum += $A[i]*A[j]$
```

Random access machine model of computation

- Primitive step
  - “Simple” instruction (such as +, *, -, =, if):
  - Memory access

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        sum += $A[i]*A[j]$
```

Counting steps precisely...

- Primitive step
  - "Simple" instruction (such as +, *, -, =, if):
  - Memory access

$\boxed{T(n) = 15n^2 + 8n + 3 + 1}$

```
sum = 0                            1 step
n = length of array A              2 steps
for i = 1 to n
    for j = 1 to n
        sum += A[i]*A[j]           8 steps
```

assign i

outer loop test and increment

assign j

inner loop test and increment

n times ⟶ 1 + n*(8+7) steps

n times ⟶ 1 + n*((1 + 15n) + 7) steps

Counting steps precisely… is painful & platform-dependent

- Primitive step
  - "Simple" instruction (such as +, *, -, =, if):
  - Memory access

$$\boxed{\mathbf{T(}\boldsymbol{n}\mathbf{) = 15}\boldsymbol{n}^{\mathbf{2}}\mathbf{ + 8}\boldsymbol{n}\mathbf{ + 4}}$$

For large values of n, $\mathrm{T}(n)$ is less than some multiple of n^2.

We say $\mathrm{T}(n)$ is $\mathbf{O}(\boldsymbol{n}^{\mathbf{2}})$ and typically don't care about other terms, as the "quadratic" term dominates as n grows larger.

```
sum = 0
n = length of array A
for i = 1 to n
    for j = 1 to n
        sum += A[i]*A[j]
```

Why asymptotic analysis?

- Analyze pseudocode with each line as a single step
- Capture “efficiency” of process
    - simplified, yet rigorous
- $T(n) = 15n^2 + 8n + 4$ is $\mathbf{O(n^2)}$ *and* $\Omega(n^2)$ *and* $\Theta(n^2)$...

```
sum = 0
$n$ = length of array $A$
for $i = 1$ to $n$
    for $j = 1$ to $n$
        sum += $A[i]*A[j]$
```

Asymptotic bounds

- Upper bound: $O(g(n))$
  - Strict upper bound: $o(g(n))$
- Tight bound: $\Theta(g(n))$
- Lower bound: $\Omega(g(n))$
  - String lower bound: $\omega(g(n))$

$O(g(n))$	$\leq$
$o(g(n))$	$<$
$\Theta(g(n))$	$=$
$\Omega(g(n))$	$\geq$
$\omega(g(n))$	$>$

Why **worst-case** asymptotic analysis?

Mathematical analysis of *worst-case* running time as function of *input size*

- Mathematical
  - describes algorithm
  - avoids hard-to-control experimental factors while still capturing behavior
    - (CPU, programming language, quality of implementation)
- Worst-case
  - just works…
  - "average case" appealing, but hard to analyze
- Function of *input size*
  - allows predictions -- what will happen on new input?
  - zoom out to generalized behavior

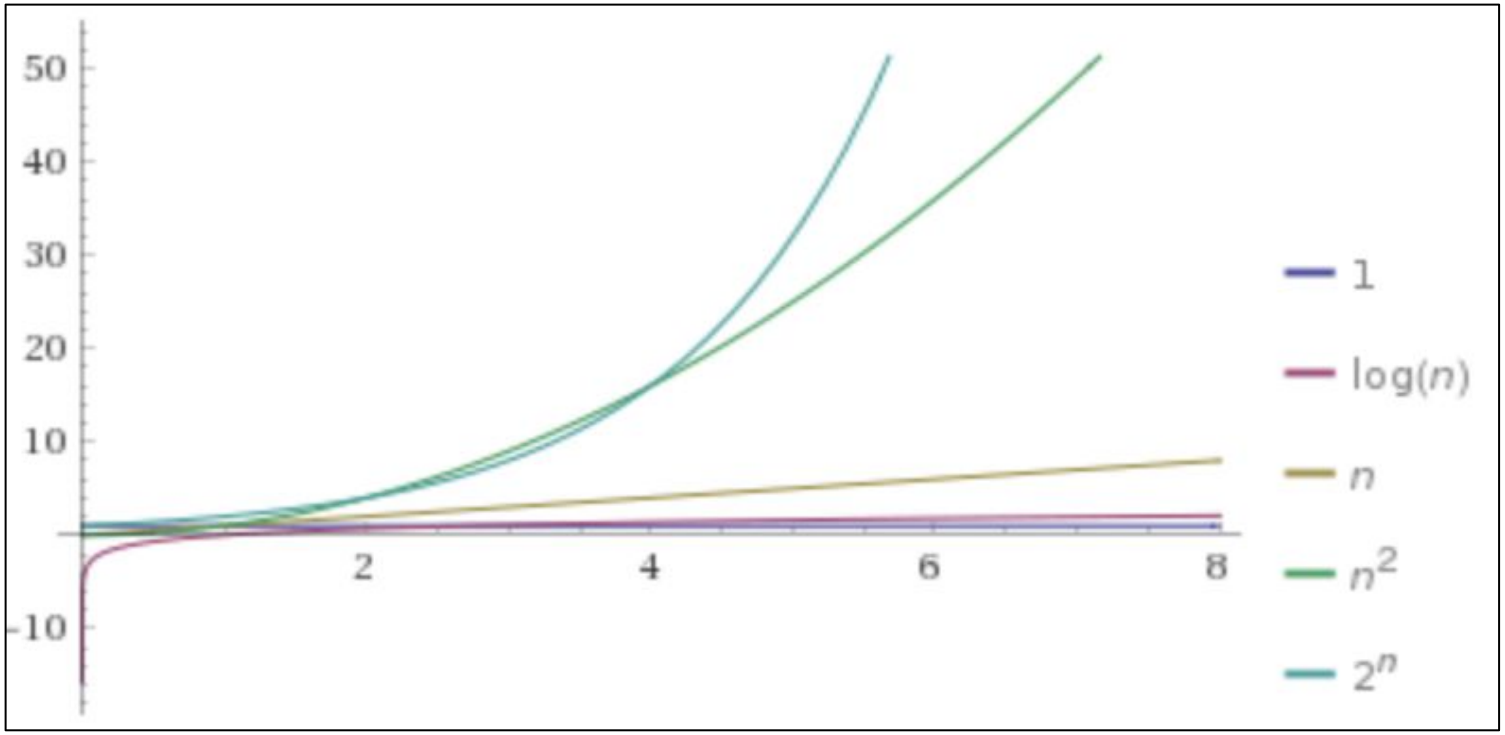
Asymptotic behavior -- what happens as *n* increases?

- How do these functions grow comparatively?
  - n^2
  - $n^2 + 1000000n$
  - n
  - n + 10000
- wolframalpha.com

plot n^2; n^2+1000000n; n from n = 0 to 100000

Common functions

$f_0(n) = 1$ vs. $f_1(n) = \log_2 n$ vs. $f_2(n) = \mathrm{n}$ vs. $f_3(n) = n^2$ vs. $f_4(n) = 2^n$

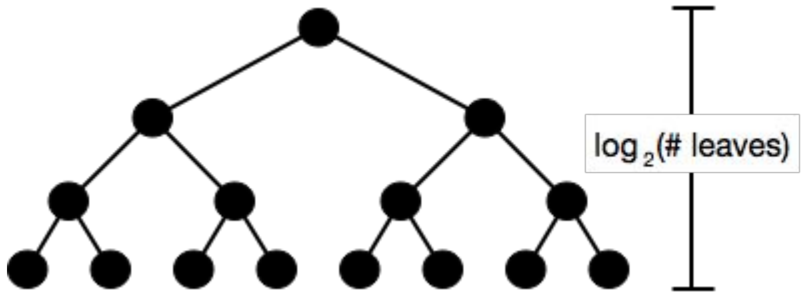


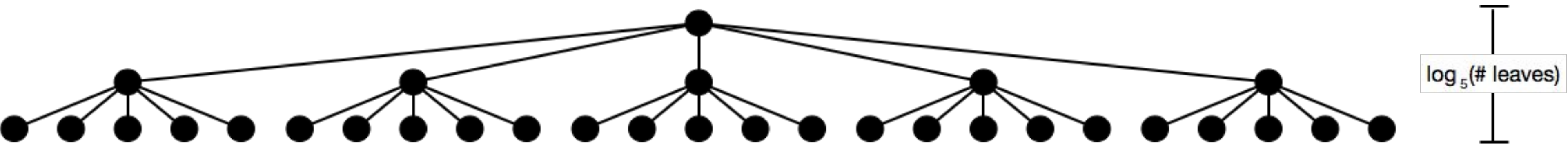
Wolframalpha.com: Plot[{1,log n, n, n^2, 2^n}, {n, 0, 8}]

Logarithm

Definition: $\log_b(a)$ is the unique number c such that $b^c=a$

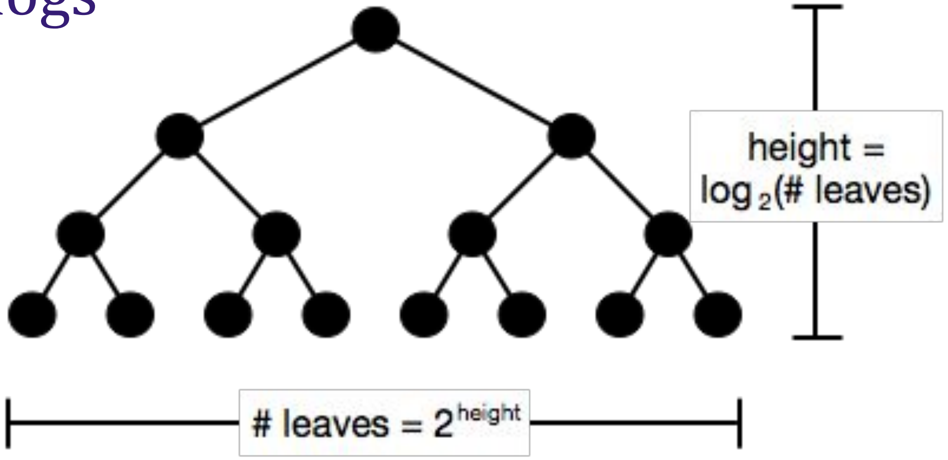
Informally: the number of times you can divide a by b until you get to 1





Properties of log

- Log of product $\longrightarrow$ sum of logs
  - $\log(xy) = \log x + \log y$
  - $\log(x^k) = k \log x$
- $\log_b(a)$ is inverse of b^a
  - $\log_b(b^n) = n$
  - $b^{\log_b n} = n$
- Logs in any two bases are proportional
  - $\log_a n = \log_a b \log_b n$
- When using big-O, it's OK not to specify base.
  - Assume $\log_2$ if not specified.



Worst-case asymptotic analysis

Asymptotic upper bound

$\mathrm{O}(g(n))$

$\leq$

Big-O: formal definition

read: “is order f(n)”

Definition: The function $T(n)$ is $O(f(n))$ if there exist constants $c \geq 0$ and $n_0 \geq 0$ such that

$$T(n) \leq c\, f(n) \text{ for all } n \geq n_0$$

We say that f is an “asymptotic upper bound” for T.

Example:

- Claim: $T(n) = 15n^2 + 8n + 4 = O(n^2)$
- Let $c = 16$, $n_0 = 9$
  - If $n = 9$, is $8n + 4 \leq n^2$?
    - Check $8*9 + 4 \leq 9*9$?

Properties of big-O

- Big-O is a **relation** between two functions
- There is **no unique** function $g(n)$ s.t. $f(n) = O(g(n))$
- There are **no unique** c and n_0 to show $f(n) = O(g(n))$
- Whether $f(n) = O(g(n))$ does not depend on
  - multiplying f or g by a constant (we choose c)
  - the first 2 or 5 or 1000 etc. values (we choose n_0)

Definition: The function f(n) is O(g(n)) if there exist constants
$c \geq 0$ and $n_0 \geq 0$ such that

$$f(n) \leq c\, g(n) \text{ for all } n \geq n_0$$

Big-O: transitivity and additivity

Claim *(Transitivity)*:

If $f = O(g)$ and $g = O(h)$, then $f = O(h)$

Claim (*Additivity*)

- If $f = O(g)$, then $f + g = O(g)$
- If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$

Proofs

- Exercises below

How does additivity make life easier?

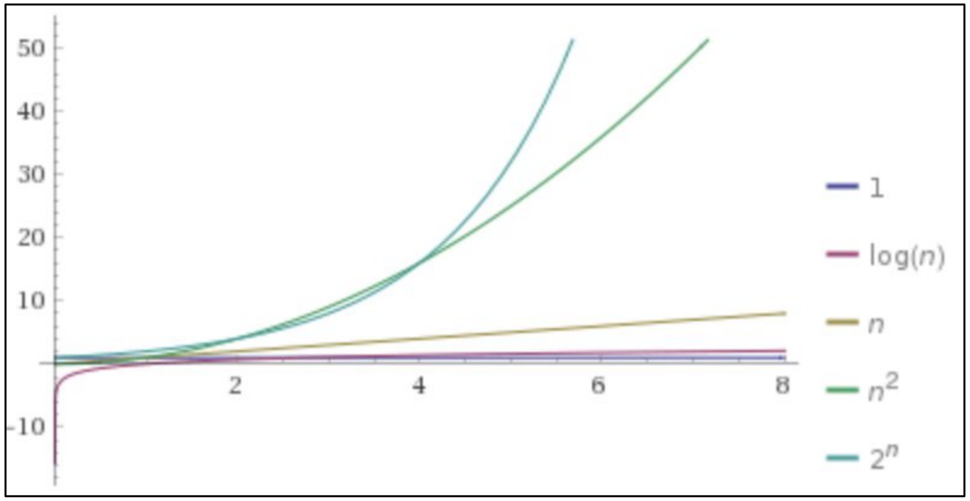
$$\boxed{\text{If } f = O(h) \text{ and } g = O(h) \text{, then } f + g = O(h)}$$

- OK to drop lower order terms
  - $2n^5 + 10n^3 + 4n \log n + 1000n = O(n^5)$
- Polynomials: Only highest-degree term matters
  - Intuitively, dominates as n grows large
  - If $a^d > 0$, then

    $a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d = O(n^d)$
- Ignore the running time of statements outside for loops

Some common shortcuts: constant < log < poly < exp

- Polynomials grow faster than logarithm of any base
  $\log_b(n) = O(n^d)$ for all $b, d > 0$
- Exponential functions grow faster than polynomials
  $n^d = O(r^n)$ for all $r > 1$



Wolframalpha.com: Plot[{1,log n, n, n^2, 2^n}, {n, 0, 8}]

Which grows faster? $n(\log n)^3$ vs. $n^{4/3}$

- divide by common factor n, simplifies to $(\log n)^3$ vs. $n^{1/3}$
- take cubic root, simplifies to: $\log n$ vs. $n^{1/9}$
- $\log n = O(n^d)$ for all d, so $\log n = O(n^{1/9})$

$\Rightarrow n(\log n)^3 = O(n^{4/3})$

Big-O: correct usage

- categorize upper bounded growth rate of functions relative to other functions
- never first
  - ~~"O(n) is the running time of my algorithm"~~
  - ~~"$O(n^2) = n$"~~
- correct usage
  - "The algorithm's running time is $O(n)$" [linear]
- not unique
  - infinitely many asymptotic upper bounds: $n = O(n^2)$, $n = O(n^3)$, …
  - try to find tightest upper bound: $n = O(n)$

Asymptotic lower bound

$$\mathbf{\Omega}(g(n))$$

$$\geq$$

Comparing algorithms

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        for $k$ = 1 to $n$
            sum += $A[i]*A[j]*A[k]$
```

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        sum += $A[i]*A[j]$
```

Fact: running time is $O(n^3)$

Fact: running time is $O(n^3)$

Can we conclude both algorithms have the **same** running time?

Maybe an upper bound isn't enough?

Definition: The function $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$T(n) \geq c\, f(n) \text{ for all } n \geq n_0$$

We say that f is an "asymptotic lower bound" for T.

Informally: T grows at least as fast as f

Example

Show that the running time is $\Omega(n^2)$

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        sum += $A[i]*A[j]$
```

The function $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$T(n) \geq c\, f(n) \text{ for all } n \geq n_0$$

Example

- Claim: $T(n) = 15n^2 + 8n + 4 = \Omega(n^2)$
- Let $c = 15$, $n_0 = 0$
  - If $n \geq 0$, is $15n^2 + 8*n + 4 \geq 15n^2$?
    - Check $15*0 + 8*0 + 4 \geq 15*0$?

The function $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$T(n) \geq c\, f(n) \text{ for all } n \geq n_0$$

Comparing algorithms

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        for $k$ = 1 to $n$
            sum += $A[i]*A[j]*A[k]$
```

Fact: running time is $O(n^3)$

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        sum += $A[i]*A[j]$
```

Fact: running time is $O(n^3)$

Fact: running time is $\Omega(n^2)$

Example

The running time is $\Omega(n^3)$

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        for $k$ = 1 to $n$
            sum += $A[i]*A[j]*A[k]$
```

The function $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that

$$T(n) \geq c\, f(n) \text{ for all } n \geq n_0$$

Comparing algorithms

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        for $k$ = 1 to $n$
            sum += $A[i]*A[j]*A[k]$
```

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        sum += $A[i]*A[j]$
```

Fact: running time is $O(n^3)$

Fact: running time is $\Omega(n^3)$

Fact: running time is $O(n^3)$

Fact: running time is $\Omega(n^2)$

Example

Show that the running time is $\Omega(n^2)$

- Hard way
  - count exactly how many times the inner loop statement executes:
    $n + (n - 1) + \dots + 2 + 1$
    $= n(n + 1)/2 = \Omega(n^2)$
- Easy way
  - Ignore all loop executions where
    $i > n/2$ or $j < n/2$:
    $\rightarrow$ inner statement executes at least
    $(n/2)^2 = \Omega(n^2)$times

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = $i$ to $n$
        sum += $A[i]*A[j]$
```

Recall: big-O properties

Transitivity

If $f = O(g)$ and $g = O(h)$, then $f = O(h)$

Additivity

- If $f = O(g)$, then $f + g = O(g)$
- If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$

Do they apply to big-omega?

Transitivity:

If $f = \Omega(g)$ and $g = \Omega(h)$, is $f = \Omega(h)$?

Additivity

- If $f = \Omega(g)$, is $f + g = \Omega(g)$?
- If $f = \Omega(h)$ and $g = \Omega(h)$, is $f + g = \Omega(h)$?

Big-omega properties

Transitivity:

If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.

Additivity

- If $f = \Omega(g)$, then $f + g = \Omega(g)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$, then $f + g = \Omega(h)$.

Asymptotic tight bound

$$\Theta(g(n))$$

$$=$$

Make a sandwich!

Definition: The function $T(n)$ is $\Theta(f(n))$ if there exist constants $c_1 > 0$, $c_2 \geq 0$ and $n_0 \geq 0$ such that

$$c_1 f(n) \leq T(n) \leq c_2 f(n) \text{ for all } n \geq n_0$$

We say that *f* is an "asymptotically tight bound" for T.

Informally: T grows at the same rate as f

Make a sandwich!

Definition: The function $T(n)$ is $\Theta(f(n))$ if there exist constants $c_1 > 0$, $c_2 \geq 0$ and $n_0 \geq 0$ such that

$$0 \leq c_1 f(n) \leq T(n) \leq c_2 f(n) \text{ for all } n \geq n_0$$

Example.

$T(n) = 32n^2 + 17n + 1$

- $T(n)$ is $\Theta(n^2)$
  - $c_1 = 0$, $c_2 \geq 33$ and $n_0 = 18$
- $T(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$

Make a sandwich!

Definition: The function $T(n)$ is $\Theta(f(n))$ if there exist constants $c_1 > 0$, $c_2 \geq 0$ and $n_0 \geq 0$ such that

$$0 \leq c_1 f(n) \leq T(n) \leq c_2 f(n) \text{ for all } n \geq n_0$$

Equivalently: The function $T(n)$ is $\Theta(f(n))$ if it is both $O(f(n))$ and $\Omega(f(n))$.

Comparing algorithms

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        for $k$ = 1 to $n$
            sum += $A[i]*A[j]*A[k]$
```

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        sum += $A[i]*A[j]$
```

Fact: running time is $O(n^3)$

Fact: running time is $\Omega(n^3)$

Fact: running time is $O(n^2)$

Fact: running time is $\Omega(n^2)$

Comparing algorithms

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        for $k$ = 1 to $n$
            sum += $A[i]*A[j]*A[k]$
```

running time is $\Theta(n^3)$

```
sum = 0
$n$ = length of array $A$
for $i$ = 1 to $n$
    for $j$ = 1 to $n$
        sum += $A[i]*A[j]$
```

running time is $\Theta(n^2)$

Tight bounds $\rightarrow$ *different* running times!

Big-theta properties

Transitivity:

If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$

Additivity

- If $f = \Theta(g)$, then $f + g = \Theta(g)$
- If $f = \Theta(h)$ and $g = \Theta(h)$, then $f + g = \Theta(h)$

Why asymptotic analysis?

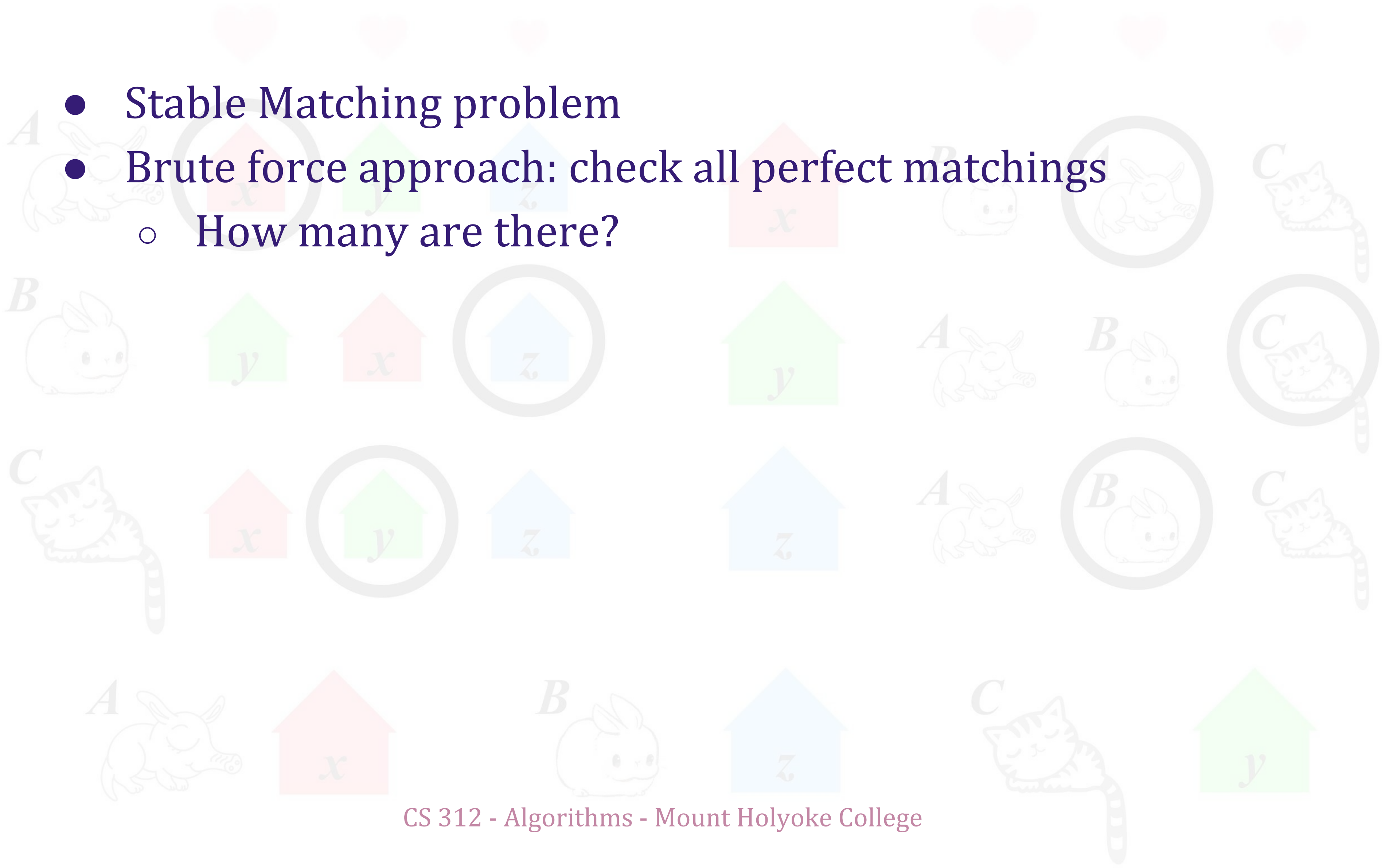
Asymptotic bounds

- Upper bound: $O(g(n))$
  - Strict upper bound: $o(g(n))$
- Tight bound: $\Theta(g(n))$
- Lower bound: $\Omega(g(n))$
  - String lower bound: $\omega(g(n))$

$O(g(n))$	$\leq$
$o(g(n))$	$<$
$\Theta(g(n))$	$=$
$\Omega(g(n))$	$\geq$
$\omega(g(n))$	$>$

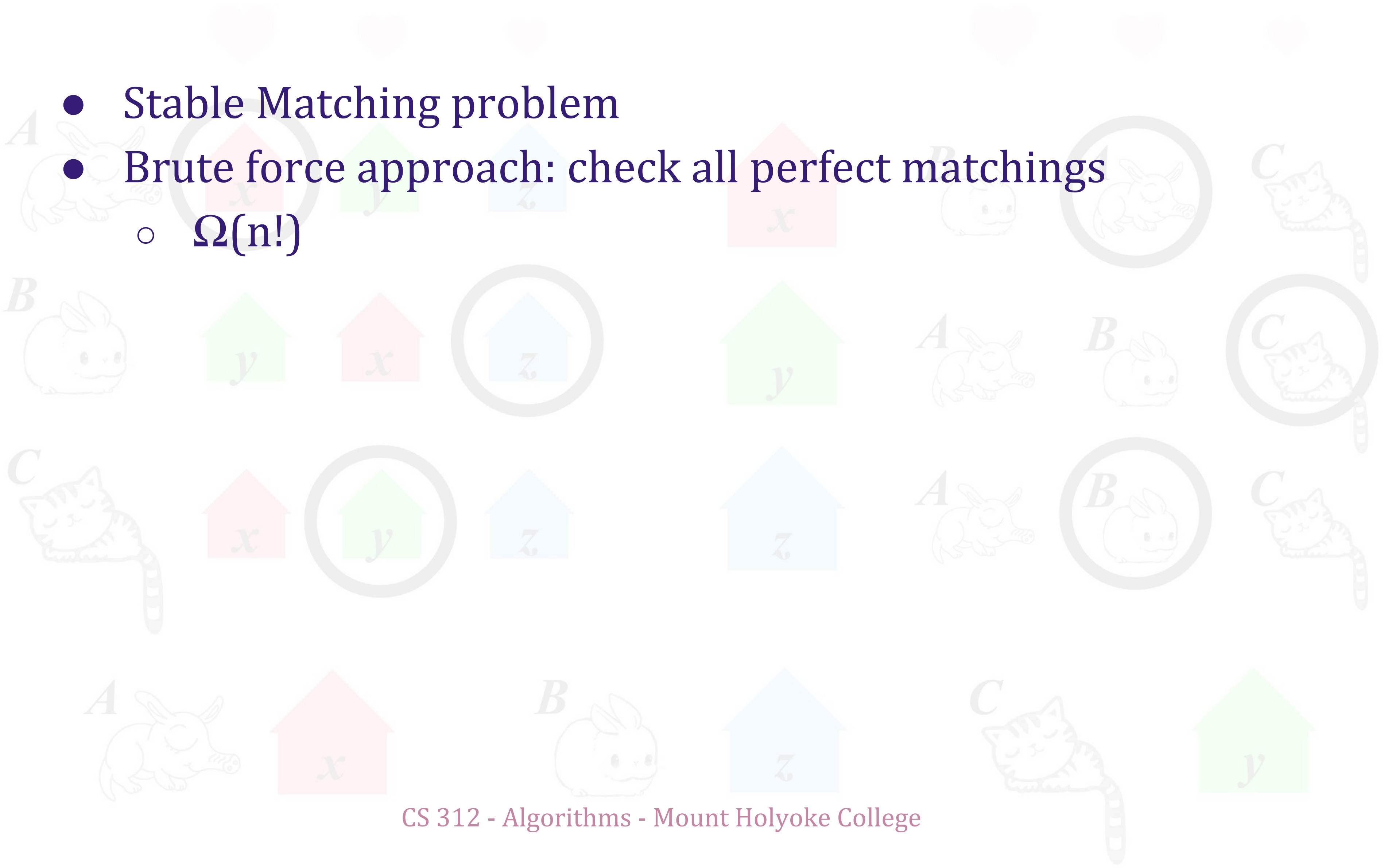
How do we capture efficiency?

- Stable Matching problem
- Brute force approach: check all perfect matchings
    - How many are there?



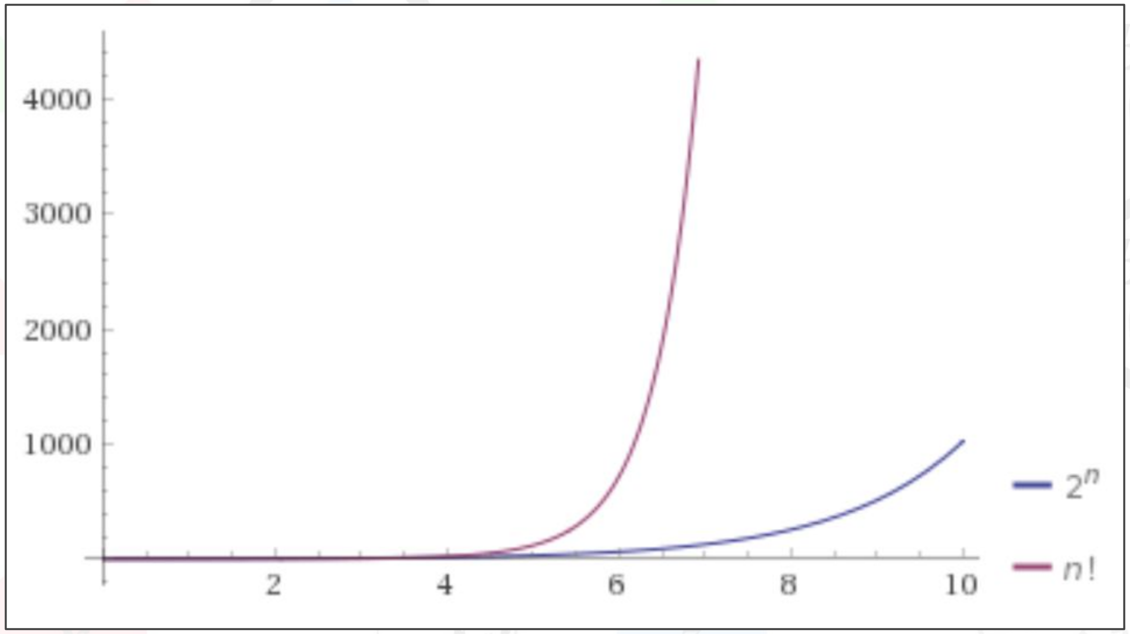
How do we capture efficiency?

- Stable Matching problem
- Brute force approach: check all perfect matchings
    - $\Omega(n!)$



How do we capture efficiency?

- Stable Matching problem
- Brute force approach: check all perfect matchings
  - $\Omega(n!) = \Omega(2^n)$



How do we capture efficiency?

- Stable Matching problem
- Brute force approach: check all perfect matchings
  - $\Omega(n!) = \Omega(2^n)$

- Propose-and-Reject
  - $O(n^2)$ rounds, each round $O(1)$, assuming good data structures
  - How does it do better than brute force?
  - Question: Is it $\Omega(n^2)$?

What is considered efficient?

Definition: an algorithm runs in *polynomial time* if its running time is $O(n^d)$ for some constant d

Polynomial time

Polynomial time

$O(n^d)$

- $f_1(n) = n$
- $f_2(n) = 4n + 100$
- $f_3(n) = n\log(n) + 2n + 20$
- $f_4(n) = 0.01n^2$
- $f_5(n) = n^2$
- $f_6(n) = 20n^2 + 2n + 3$

Non-polynomial time

$\omega(n^d)$

- $f_7(n) = 2^n$
- $f_8(n) = 3^n$
- $f_9(n) = n!$

Why polynomial?

- Good definition of **efficiency**
  - almost all *practically* efficient algorithms have this property
  - usually distinguishes an algorithm from a “brute force”approach

- Threshold for saying an algorithm is not efficient, or that no efficient algorithm exists
  - Paired with asymptotic analysis (strict lower bound ω)

Exponential time

Definition: an algorithm is *exponential time* if it is $O(2^{n^k})$ for some $k > 0$.

Note: this technically implies polynomial time algorithms are exponential.

Generally, though, we talk about the tightest upper bound and may refer to polynomial and exponential time algorithms as if they are different classes.