CS 375 – Analysis of Algorithms

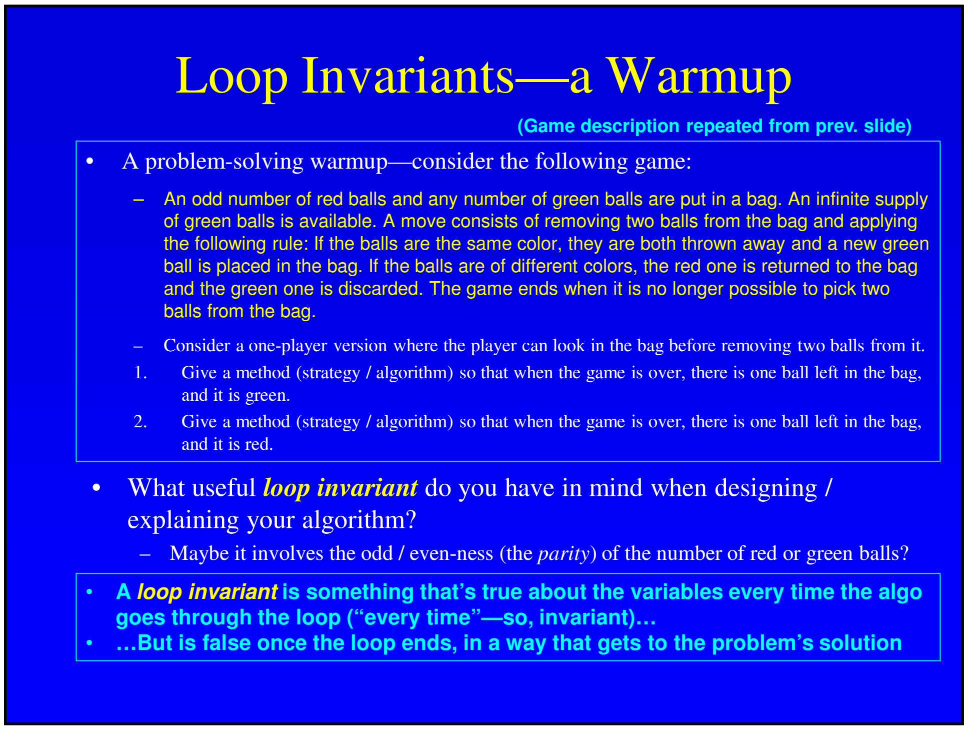
Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- Grading update:
  - SA1 returned already
  - PS0 grading update
- I welcome feedback about grader feedback—let's talk!
- Problem Set 1 due Oct. 5
  - See note about *break* statements—please avoid them in CS375
  - Please get started early and ask questions early!
- Project 1 due end of day today!
  - (Please don't count on my availability for much help by email after I leave the office this afternoon
- Another SA may be out soon—I'll email if so

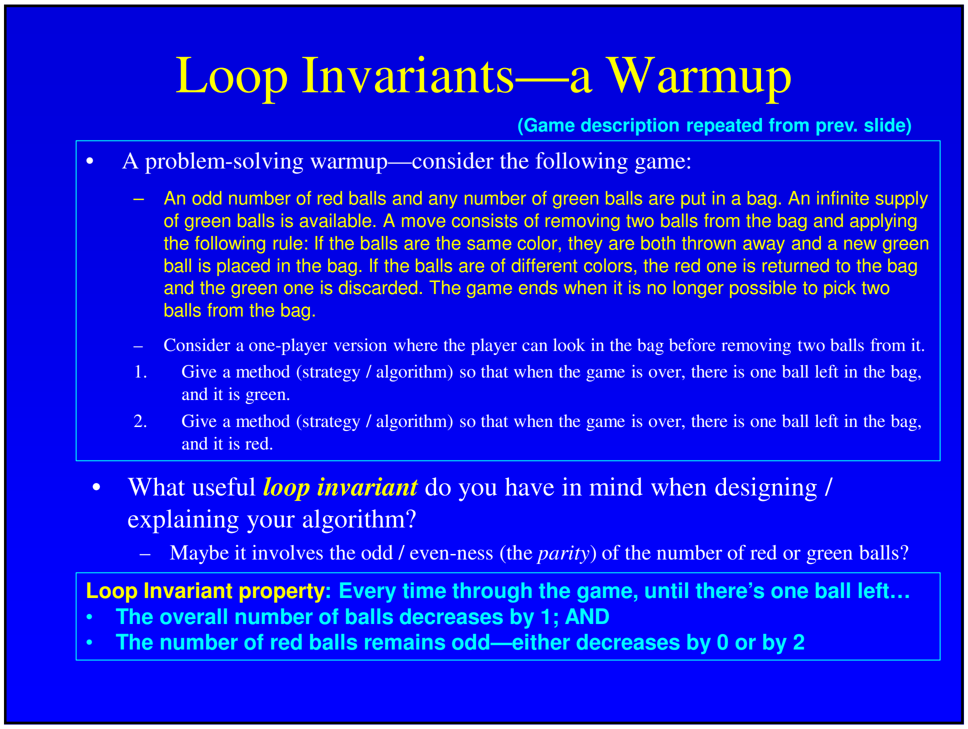


Loop Invariants—a Warmup

(Game description repeated from prev. slide)

- A problem-solving warmup—consider the following game:
  - An odd number of red balls and any number of green balls are put in a bag. An infinite supply of green balls is available. A move consists of removing two balls from the bag and applying the following rule: If the balls are the same color, they are both thrown away and a new green ball is placed in the bag. If the balls are of different colors, the red one is returned to the bag and the green one is discarded. The game ends when it is no longer possible to pick two balls from the bag.
  - Consider a one-player version where the player can look in the bag before removing two balls from it.
  1. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is green.
  2. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is red.
- What useful ***loop invariant*** do you have in mind when designing / explaining your algorithm?
  - Maybe it involves the odd / even-ness (the *parity*) of the number of red or green balls?

- **A *loop invariant* is something that's true about the variables every time the algo goes through the loop ("every time"—so, invariant)…**
- **…But is false once the loop ends, in a way that gets to the problem's solution**

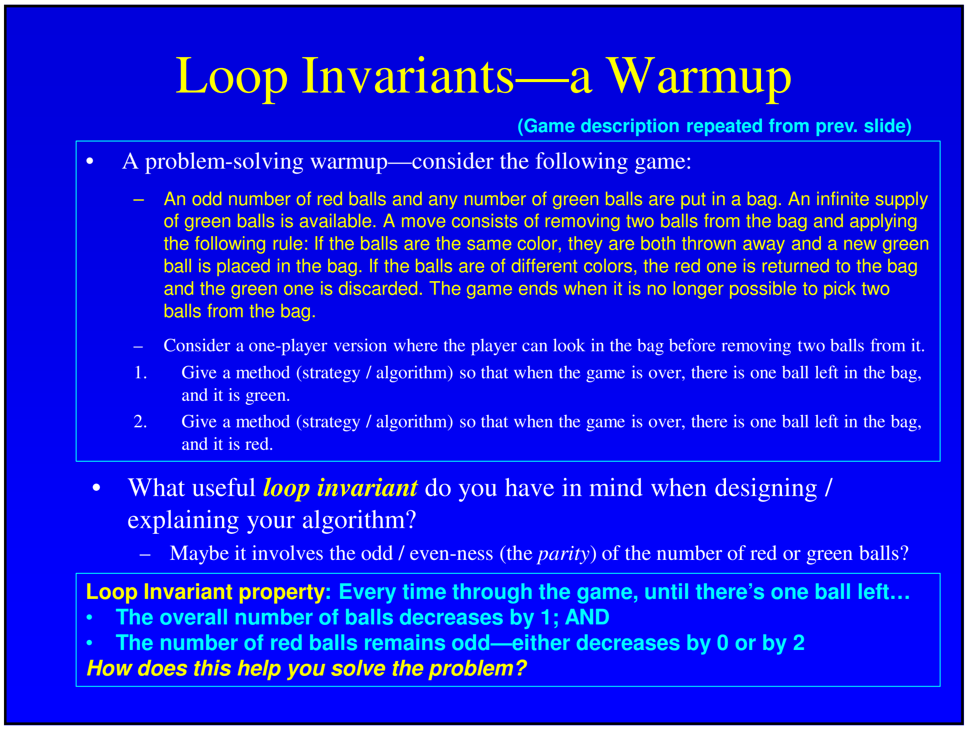


Loop Invariants—a Warmup

(Game description repeated from prev. slide)

- A problem-solving warmup—consider the following game:
  - An odd number of red balls and any number of green balls are put in a bag. An infinite supply of green balls is available. A move consists of removing two balls from the bag and applying the following rule: If the balls are the same color, they are both thrown away and a new green ball is placed in the bag. If the balls are of different colors, the red one is returned to the bag and the green one is discarded. The game ends when it is no longer possible to pick two balls from the bag.
  - Consider a one-player version where the player can look in the bag before removing two balls from it.
  1. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is green.
  2. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is red.
- What useful ***loop invariant*** do you have in mind when designing / explaining your algorithm?
  - Maybe it involves the odd / even-ness (the *parity*) of the number of red or green balls?

Loop Invariant property: Every time through the game, until there's one ball left…
- **The overall number of balls decreases by 1; AND**
- **The number of red balls remains odd—either decreases by 0 or by 2**



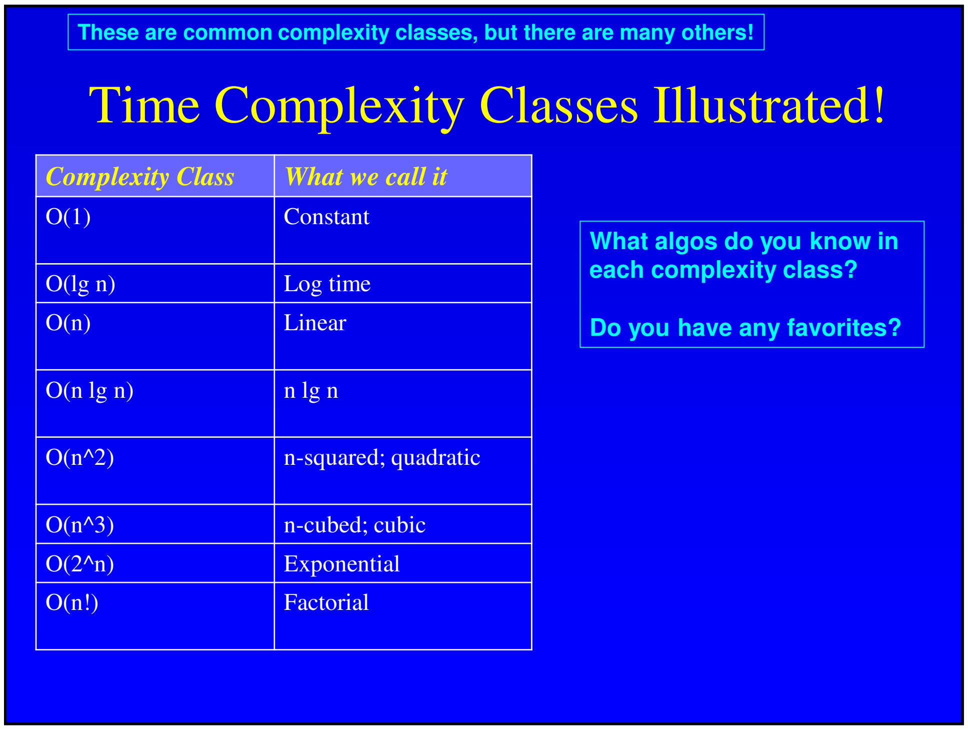
Loop Invariants—a Warmup

(Game description repeated from prev. slide)

- A problem-solving warmup—consider the following game:
  - An odd number of red balls and any number of green balls are put in a bag. An infinite supply of green balls is available. A move consists of removing two balls from the bag and applying the following rule: If the balls are the same color, they are both thrown away and a new green ball is placed in the bag. If the balls are of different colors, the red one is returned to the bag and the green one is discarded. The game ends when it is no longer possible to pick two balls from the bag.
  - Consider a one-player version where the player can look in the bag before removing two balls from it.
  1. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is green.
  2. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is red.
- What useful ***loop invariant*** do you have in mind when designing / explaining your algorithm?
  - Maybe it involves the odd / even-ness (the *parity*) of the number of red or green balls?

Loop Invariant property: Every time through the game, until there's one ball left…
- **The overall number of balls decreases by 1; AND**
- **The number of red balls remains odd—either decreases by 0 or by 2**

How does this help you solve the problem?



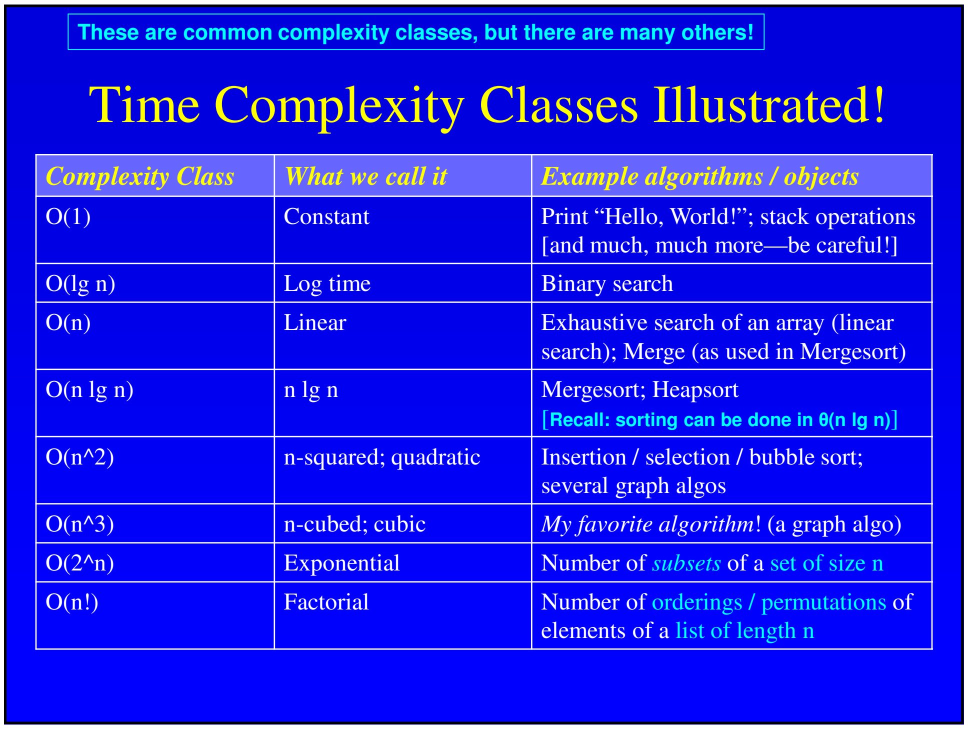
These are common complexity classes, but there are many others!

Time Complexity Classes Illustrated!

Complexity Class	*What we call it*
O(1)	Constant
O(lg n)	Log time
O(n)	Linear
O(n lg n)	n lg n
O(n^2)	n-squared; quadratic
O(n^3)	n-cubed; cubic
O(2^n)	Exponential
O(n!)	Factorial

What algos do you know in each complexity class?

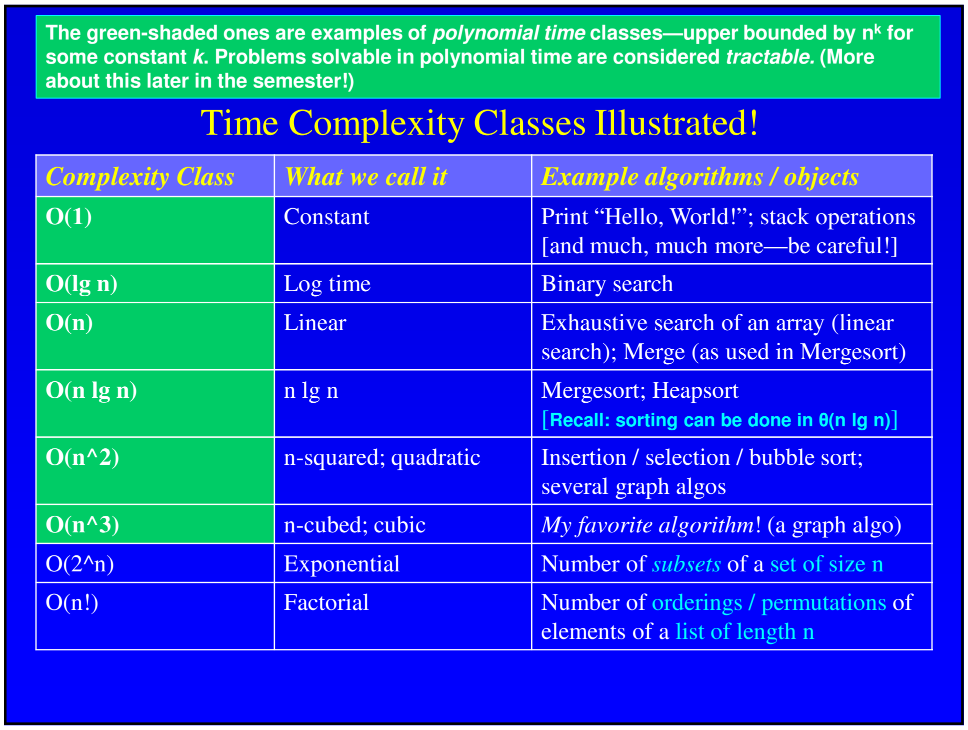
Do you have any favorites?



These are common complexity classes, but there are many others!

Time Complexity Classes Illustrated!

Complexity Class	*What we call it*	*Example algorithms / objects*
O(1)	Constant	Print “Hello, World!”; stack operations [and much, much more—be careful!]
O(lg n)	Log time	Binary search
O(n)	Linear	Exhaustive search of an array (linear search); Merge (as used in Mergesort)
O(n lg n)	n lg n	Mergesort; Heapsort [Recall: sorting can be done in θ(n lg n)]
O(n^2)	n-squared; quadratic	Insertion / selection / bubble sort; several graph algos
O(n^3)	n-cubed; cubic	*My favorite algorithm*! (a graph algo)
O(2^n)	Exponential	Number of *subsets* of a set of size n
O(n!)	Factorial	Number of orderings / permutations of elements of a list of length n



The green-shaded ones are examples of *polynomial time* classes—upper bounded by n^k for some constant *k*. Problems solvable in polynomial time are considered *tractable.* (More about this later in the semester!)

Time Complexity Classes Illustrated!

Complexity Class	*What we call it*	*Example algorithms / objects*
O(1)	Constant	Print "Hello, World!"; stack operations [and much, much more—be careful!]
O(lg n)	Log time	Binary search
O(n)	Linear	Exhaustive search of an array (linear search); Merge (as used in Mergesort)
O(n lg n)	n lg n	Mergesort; Heapsort [**Recall: sorting can be done in θ(n lg n)**]
O(n^2)	n-squared; quadratic	Insertion / selection / bubble sort; several graph algos
O(n^3)	n-cubed; cubic	*My favorite algorithm*! (a graph algo)
O(2^n)	Exponential	Number of *subsets* of a set of size n
O(n!)	Factorial	Number of orderings / permutations of elements of a list of length n

DISCLAIMER: There is no such tour. This is a gratuitous, made-up example.
But Vai and Grasso are both *great* guitar players—that part is true!

The CS375 Guitar Genius Tour!

- Guitarists Steve Vai and Pasquale Grasso—both faves of your CS375 Prof.—are finally going on tour together!
  - There are *n* possible venues they could play on their tour
  - They could play any number of them, from 0 to *n*

Steve Vai (from cover of album *Inviolate*)

Pasquale Grasso
(from video for *It Don't Mean A Thing…*)

DISCLAIMER: There is no such tour. This is a gratuitous, made-up example.
But Vai and Grasso are both *great* guitar players—that part is true!

The CS375 Guitar Genius Tour!

- Guitarists Steve Vai and Pasquale Grasso—both faves of your CS375 Prof.—are finally going on tour together!
  - There are *n* possible venues they could play on their tour
  - They could play any number of them, from 0 to *n*
  - They have a list ordering of the *n* venues in mind—the order in which it makes sense to travel to them
    - So, if the ordering is $V_1, V_2, \ldots, V_n$ from first to last, they would never play V_y before V_x if $y > x$. Larger numbers are always later in the ordering.
    - But they could skip, or play, any or all venues

DISCLAIMER: There is no such tour. This is a gratuitous, made-up example.
But Vai and Grasso are both *great* guitar players—that part is true!

The CS375 Guitar Genius Tour!

- Guitarists Steve Vai and Pasquale Grasso—both faves of your CS375 Prof.—are finally going on tour together!
  - There are n possible venues they could play on their tour
  - They could play any number of them, from 0 to n
  - They have a list ordering of the n venues in mind—the order in which it makes sense to travel to them
    - So, if the ordering is V_1, V_2, …, V_n from first to last, they would never play V_y before V_x if $y > x$. Larger numbers are always later in the ordering.
    - But they could skip, or play, any or all venues

- **We want to figure out the best tour itinerary for them. For now, we'll use a *brute force* method of checking all possible tour itineraries.**
- ***How many possible itineraries are there?***

Exhaustive Search (Brute Force)

- One common category of algorithm is called *exhaustive search.* As just a couple of examples:
  - Find if an element is a member of a collection [search problem]
  - Find the optimal element in a collection—e.g., most valuable item (or items) to take as prizes [optimization problem]
  - Find the best way of ordering all elements in a collection—e.g., best path through a network [optimization problem]
- Exhaustive search is a kind of *brute force* algorithm
- Design is straightforward
  - Look through all elements in the collection, see if they meet criteria [search, optimality, …]

Exhaustive Search (Brute Force)

See Levitin (recommended, not required textbook), pg. 28

Notes about Exhaustive Search / Brute Force algos:

- Applicable to a very wide variety of problems
  - It may be the only algo design paradigm where it's more difficult to find problems to which it *can't* be applied than problems to which it can!
- For some important problems, brute force algos are practical, efficient, and commonly used—with no limitation on input
  - Linear search, matrix multiplication, etc.
- Sometimes, it's more expensive to design / implement / maintain a clever solution than to just use brute force
  - Depends, sometimes, on how many times the algo / solution will be run

Exhaustive Search (Brute Force)

Notes about Exhaustive Search / Brute Force algos:

- They're often not the most asymptotically efficient solution we know of…
  - But more times than you'd think, they are!
  - More about that later in the semester!
- Because of their straightforward design, they're often easier than other algos to:
  - understand and explain to others
  - reason about (correctness / complexity)
  - maintain / extend / alter
- They can be a good first step to a more efficient solution (More about that later in the semester, too!)

Something To Consider:

When you're presenting a non-brute force algo as a preferred solution, you might…

- **give reasons that it's better than brute force**
- **and say how much better it is (e.g., by asymptotic complexity analysis!)**

Exhaustive Search (Brute Force)

- How many subsets are there of a given set *S*?
  - Say, for notation, *S* has *n* elements

Vocab: The set of all subsets of S is called the *power set* of S

- How many orderings (or *permutations*) are there of all elements in a list L = [a_1, …, a_n]?

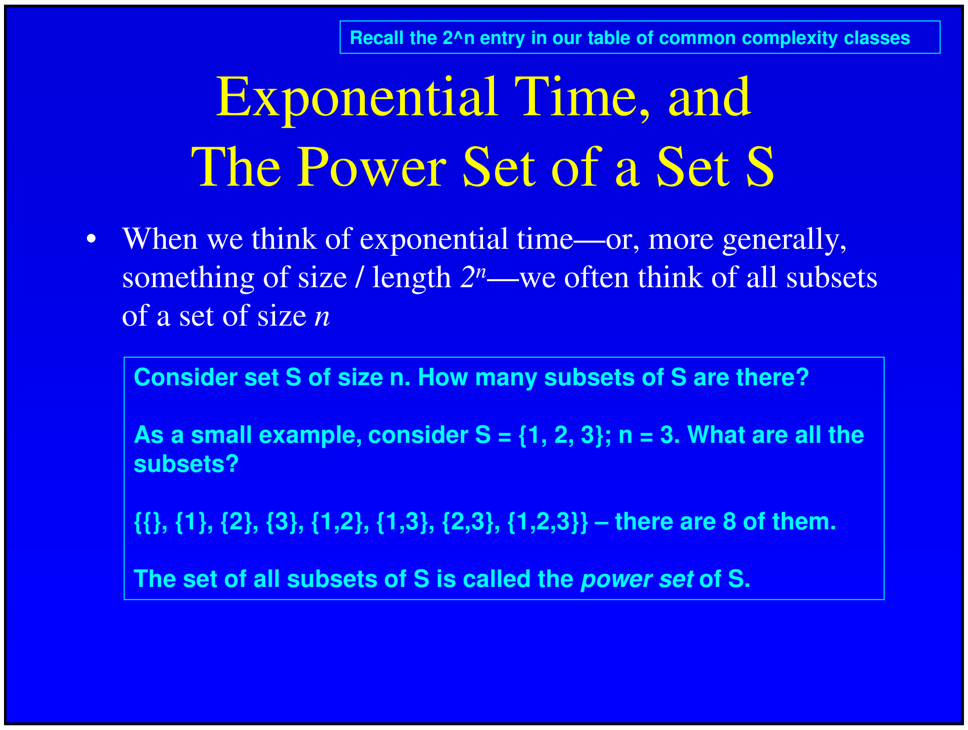
The Power Set of a Set S

Consider set S of size n. How many subsets of S are there?

As a small example, consider S = {1, 2, 3}; n = 3. What are all the subsets?

{{}, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}} – there are 8 of them.

The set of all subsets of S is called the *power set* of S.



Recall the 2^n entry in our table of common complexity classes

Exponential Time, and The Power Set of a Set S

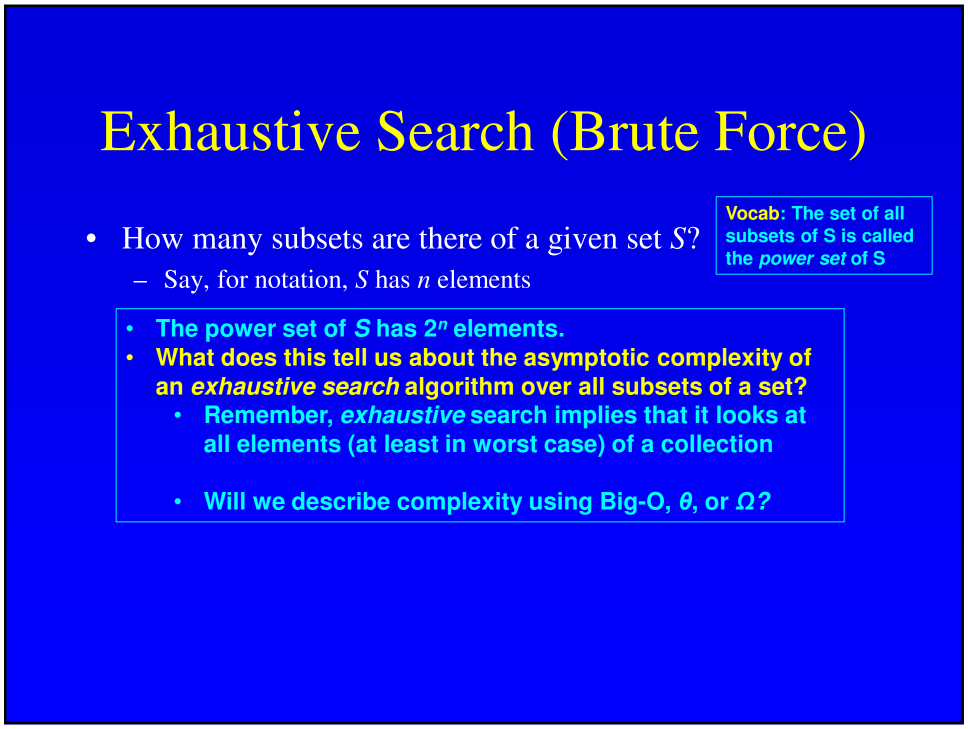
- When we think of exponential time—or, more generally, something of size / length 2^n—we often think of all subsets of a set of size n

Consider set S of size n. How many subsets of S are there?

As a small example, consider S = {1, 2, 3}; n = 3. What are all the subsets?

{{}, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}} – there are 8 of them.

The set of all subsets of S is called the *power set* of S.



Exhaustive Search (Brute Force)

- How many subsets are there of a given set S?
  - Say, for notation, S has n elements

Vocab: The set of all subsets of S is called the *power set* of S

- **The power set of *S* has 2^n elements.**
- **What does this tell us about the asymptotic complexity of an *exhaustive search* algorithm over all subsets of a set?**
  - **Remember, *exhaustive* search implies that it looks at all elements (at least in worst case) of a collection**
  - **Will we describe complexity using Big-O, *θ*, or *Ω*?**