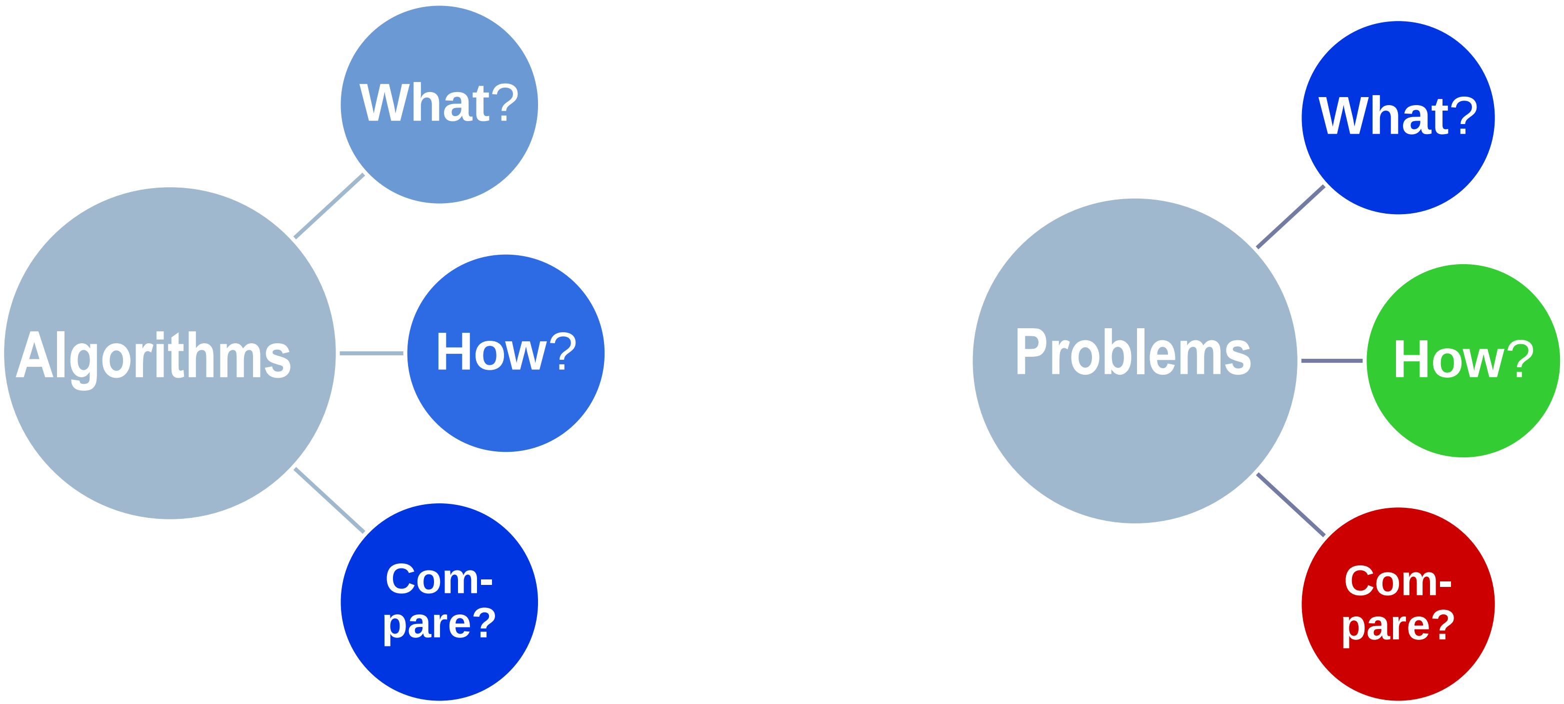
Complexity

Department of Computer Science, Tsinghua University

Complexity



Algorithm Complexity

- Let $T_A(n)$ = the computational cost for solving an problem instance of size n by algorithm A.
  - Shorted as $T(n)$

```
candidate = NIL
count = 0
for i = 1 to n
    if count == 0
        candidate = A[i]
    if candidate == A[i]
        count = count + 1
    else count = count − 1
```

Pre-requisite

```
Type Exponentiation(Type x, int n) // n is even.
{
        int m = n; Type power = 1, z = x;        q=0;
        while (m > 0) {
                while (!(m%2)) {
                        m /= 2; z *= z;        q=q+1;
                }
                m--; power *= z;
        }
        return power;
}
```

Hint

Hint: $b_k b_{k-1} \cdots b_1$ is the binary representation of n, $x^n = x^{\sum_{q=0}^{k} b_q 2^q} = (x)^{b_0} * x^{2b_1} * x^{4b_2} * \cdots * x^{2^k b_k}$

What?

- Not Enough! Different instants of the same size may lead to very different computational costs.

Worst-case running time : (usually)

- $T(n)$ = the maximum running time of an algorithm on any input of size n.

Average-case running time: (sometimes)

- $T(n)$= the expected running time of an algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case running time: (rare)

- $T(n)$ = the minimum running time of an algorithm on any input of size n.

How?

- Measure the running time?
  - Implemented by different people, language, complier…
  - Different: architecture, operating systems…

- Need to compare on an ideal platform or model:
  - Human, language, machine-independent
  - Describe algorithms directly and precisely

RAM

- RAM Model = Random Access Machine
  - Each instruction takes a constant amount of time.
  - Read/Write Memory cell | Arithmetic | Comparison | goto | call
  - No concurrent operations
- Why RAM model
  - Simplification and abstraction of computing platform
  - Assess the efficiency of algorithms by running time
  - **Machine-independent time, to compare the efficiency of algorithms**
    - Running time $\propto$ # of basic operations

Asymptotic Analysis

- We are more interested in ***big enough*** problems.
  - Asymptotic Analysis: when the problem size is big enough, how does computational cost grow? Look at the ***growth*** of $T(n)$ as n is growing bigger.

- Can be used to compare the complexity of two algorithms: "When n gets large enough, a $\theta(n^2)$ algorithm *always* beats a $\theta(n^3)$ algorithm."

Complexity

Algorithms

What?

- Best case
- Average case
- Worst case

How?

- RAM Model
- Asymptotic analysis

Com-pare?

- Ω, θ, and O

Problems

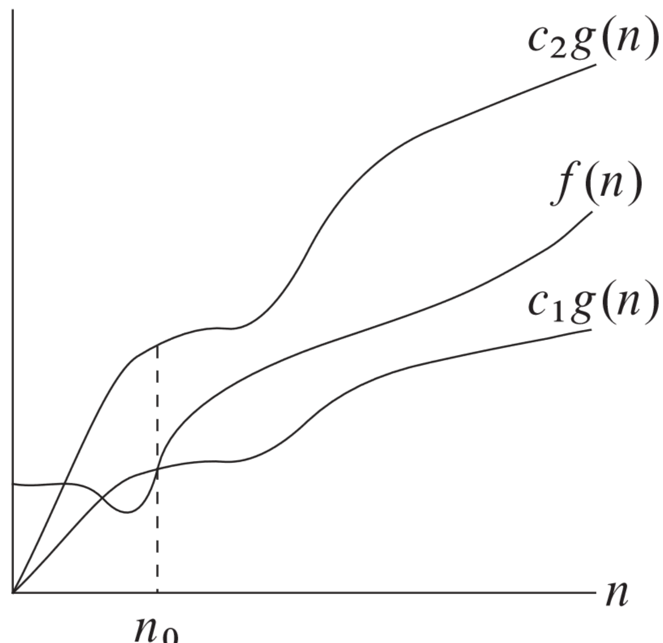
What?

How?

Com-pare?

- Θ-notation

$f(n) = \theta(g(n))$, iff there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$



$f(n) = \Theta(g(n))$

Θ-notation

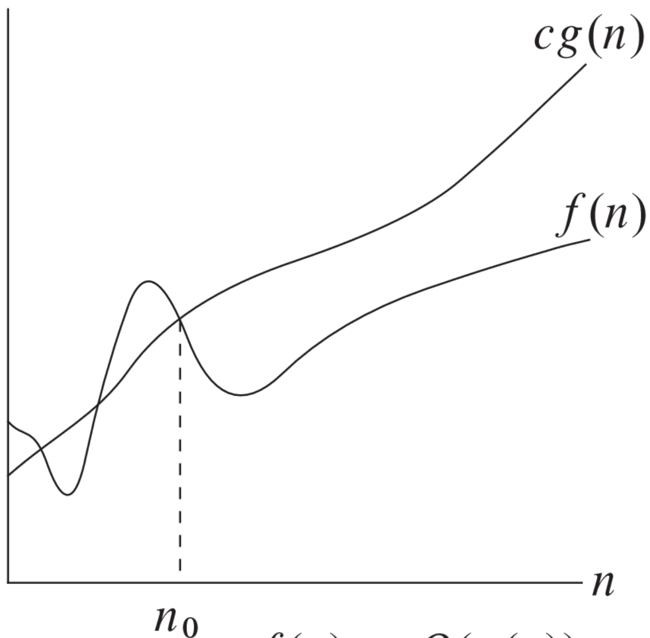
- Example: $f(n) = \frac{1}{2}n^2 - 3n,\ g(n) = n^2$
- Show that $f(n) = \theta(g(n))$
- We need to show that there exist positive constants c_1, c_2, and n_0 such that
  - $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ for all $n \geq n_0$.

Θ-notation

- **Asymptotic notation in equations**
  - $T(n) = 2n^2 + 3n + 1 = 2n^2 + \theta(n) = \theta\left(n^2\right)$
- **Transitivity**
  - $f(n) = \theta\big(g(n)\big)\ and\ g(n) = \theta(h(n))\ imply\ f(n) = \theta\big(h(n)\big)$
- **Reflexivity**
  - $f(n) = \theta(f(n))$
- **Symmetry**
  - $f(n) = \theta\big(g(n)\big) \Longleftrightarrow g(n) = \theta\big(f(n)\big)$

Big-O notation

- Big-O notation
  - $f(n) = O(g(n))$ iff there exist positive constants c and n_0, such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$



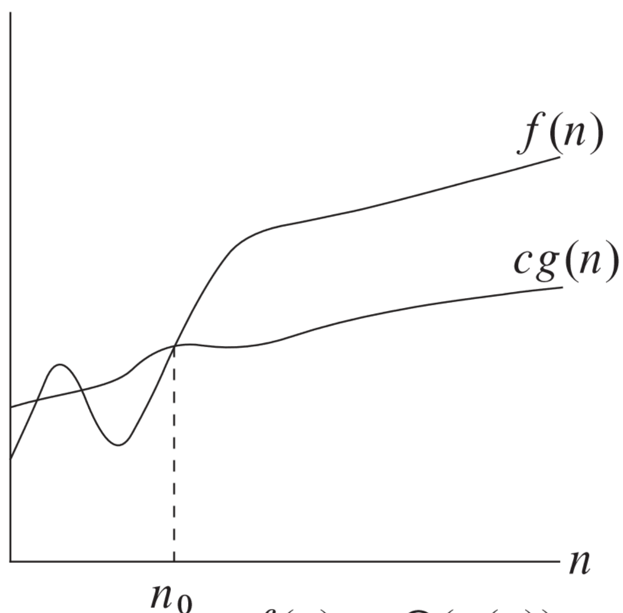
$f(n) = O(g(n))$

little-o notation

- Little-o notation (asymptotic strict upper bound)
  - $f(n) = o(g(n))$ iff for any positive constant $c > 0$, there exists a constant $n_0 > 0$, such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.
- For example:
  - $2n = o(n^2)$ but $2n^2 \neq o(n^2)$
- Relation to limit:
  - $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

Big-Ω notation

- Big-Ω notation
  - $f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0, such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$



$f(n) = \Omega(g(n))$

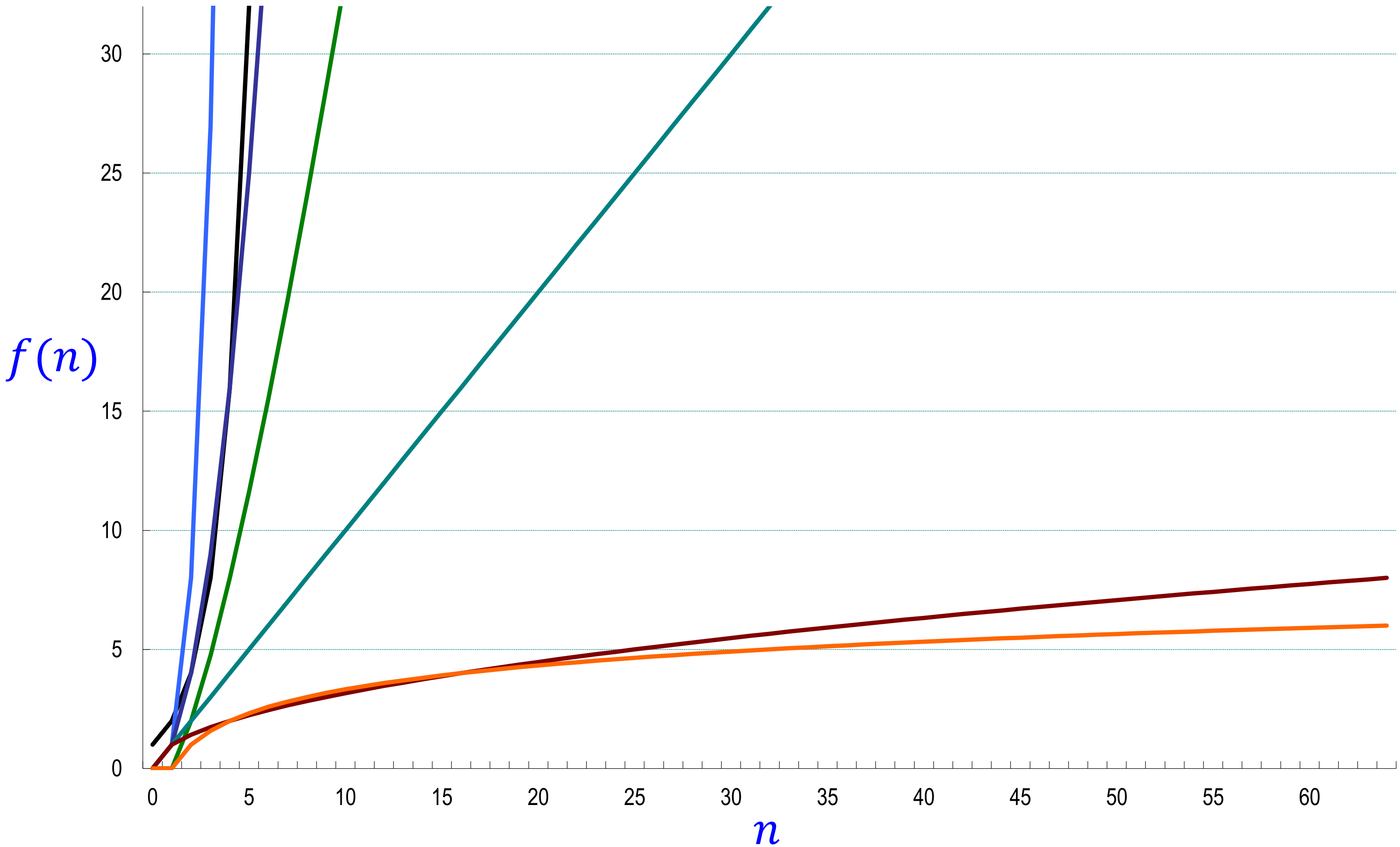
Ω, θ, and O

- The running time of A is $\Omega(g(n))$ means *no matter what particular input of size n is chosen for each value of n*, the running time on that input is at least a constant times $g(n)$, for sufficiently large n.
- In-class exercise: True or False? when you are talking about the best-case/average-case/worst-case of an algorithm A
  - The best-case running time of A is $\Omega(g(n))$ implies the running time of A is $\Omega(g(n))$.
  - The best-case running time of A is $O(g(n))$ implies the running time of A is $O(g(n))$.
  - The worst-case running time of A is $\Omega(g(n))$ implies the running time of A is $\Omega(g(n))$.
  - The worst-case running time of A is $O(g(n))$ implies the running time of A is $O(g(n))$.
- The running time of A is $\Theta(g(n))$ implies?

Speed of Growth

- Constant function
  - $2 = \Theta(1)$, $2020 = \Theta(1)$, $2020^{2020} = \Theta(1)$
- Logarithm $\theta(\log_a n) = \theta(\lg n) = \theta(\ln n)$
  - *Irrelevant to constant bases*
    - $\forall\ n > 0,\ \log_a n = \log_a b \times \log_b n = \theta(\log_b n)$
    - $\forall$ constant $a, b > 0,\ \frac{\ln a}{\ln b} = \log_a b = \theta(1)$
  - *Irrelevant to constant exponents of* n
    - $\forall$ constant $c > 0, \lg n^c = c \times \lg n = \theta(\lg n)$
- Polynomial function
  - $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = \theta\left(n^k\right), a_k > 0$
- Exponential function
  - $\forall\ c > 1, n^c = O(2^n)$ *from* $O(n^c)$ *to* $O(2^n)$ *, big step!*

Speed of Growth



Annotate on each line from the following functions: $\lg n, \sqrt{n}, n, n\lg n, n^2, n^3, 2^n$

Hierarchy of Complexity

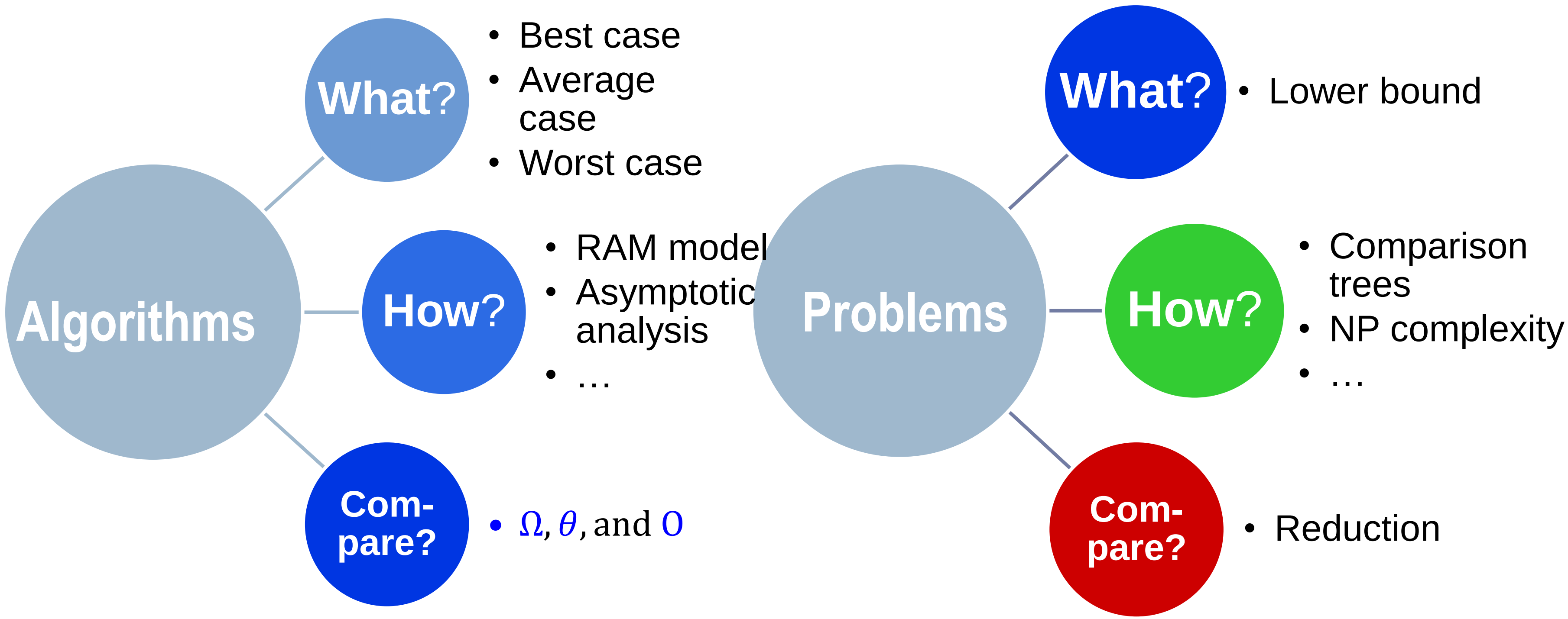
$O(1)$	constant	You are lucky!	Basic operations to data structures
$O(\lg n)$	logarithm	Close to constant, and not rare	Binary Search, insertion/deletion in a dictionary
$O(n)$	linear	A good goal	Traversal of tree, graph
$O(n \lg n)$		Quite often	Sorting, Huffman code
$O(n^2)$	quadratic	Pair-wise operations of input	Dijkstra Algorithm
$O(n^3)$	cubic	Often seen in DP	Floyd-Warshall
$O(n^c)$	polynomial	P problem = solvable by a polynomial algorithm	
$O(2^n)$	Exponential	Trivial algorithms to many NP optimization problems	
...		...	

Problem Complexity

- Problem complexity vs. Algorithm complexity
  - How do you know your algorithm is the fastest one for a problem?

- Given a problem and a function $f(n)$, if $f(n)$ is the lower bound of *any* algorithm that solves the problem *in the worst case*, we call $f(n)$ is **the lower bound of the problem**.
  - Ideal case: both $f(n)$ and the algorithm achieving $f(n)$ are known.

Problem Complexity



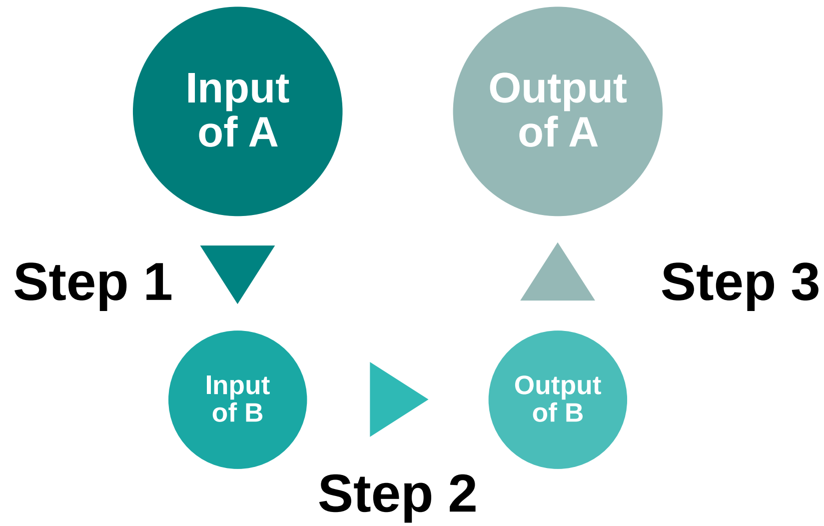
Reduction $\propto$

- Reduction is a way to compare the complexity of two problems.
- Given two problems A and B, we reduce A to B as follows:
  1. Convert the input of A into a suitable input to problem B.
  2. Solve problem B.
  3. Convert the output of B into a correct solution to problem A.

In order to achieve a $g(n)$ time reduction, Steps 1 and 3 must be performed in $\theta(g(n))$ time. We say problem A can be reduced to problem B in $g(n)$ time, and denote it by

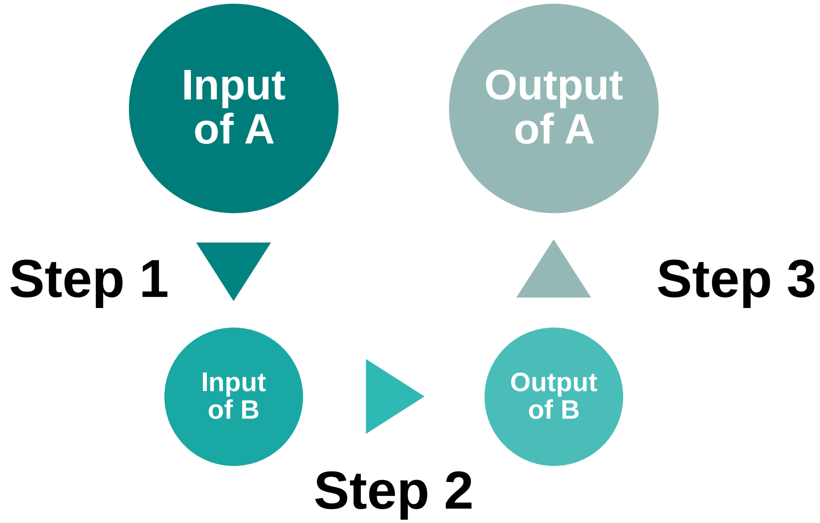
$$A \propto_{g(n)} B$$

$A \propto_{g(n)} B$



- For example: solving linear equations $\propto_c$ solving quadratic equations.
  - For any linear equation $ax + b = 0$, we can transform it to $0x^2 + ax + b = 0$ in constant time, whose solution provides a solution to $ax + b = 0$.
- Homework: Majority Element $\propto_n$ Sorting.

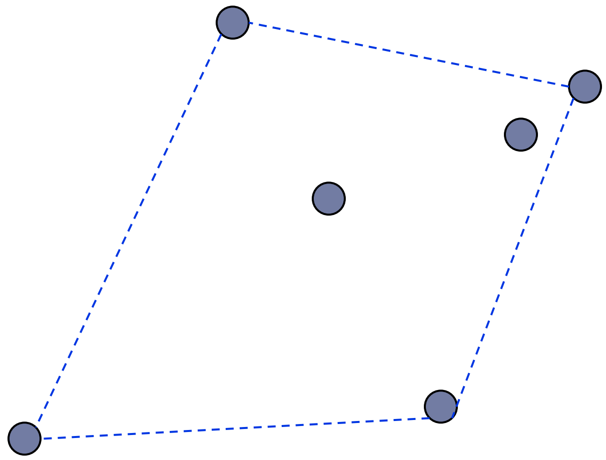
$A \propto_{g(n)} B$



- Suppose A has a lower bound of $\Omega\big(f(n)\big)$ and $g(n) = o(f(n))$, then $A \propto_{g(n)} B$ implies B has a lower bound of $\Omega\big(f(n)\big)$ as well.
  - Otherwise, we can construct a more efficient algorithm to solve A by following the reduction procedure.

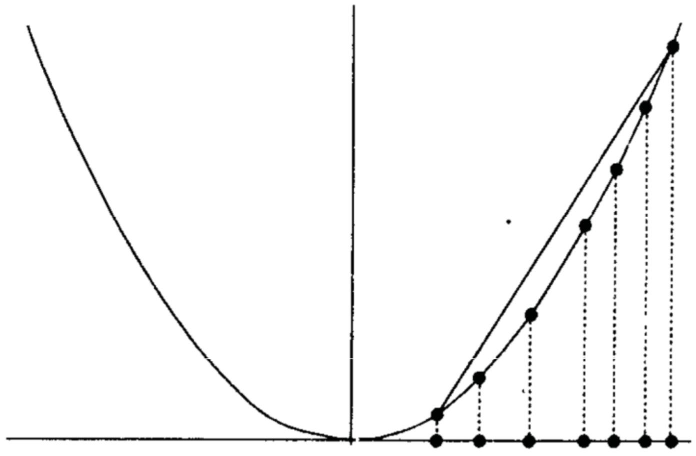
$SORTING \propto_n Convex\ Hull$

- Sorting:
  - Input: a sequence of n numbers $< a_1, a_2, \cdots, a_n >$
  - Output: a permutation of the input sequence $< a'_1, a'_2, \cdots, a'_n >$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$
- Convex Hull (p1029):
  - The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.



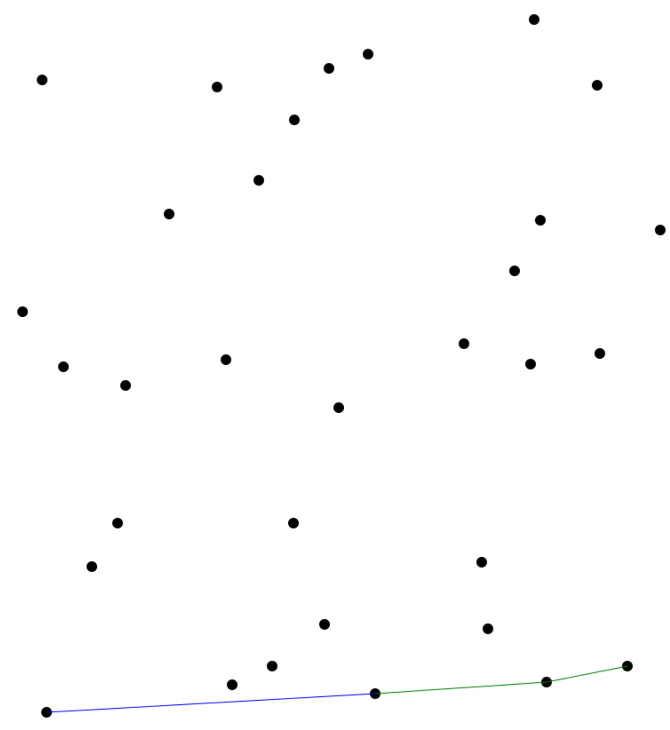
$SORTING \propto_n Convex\ Hull$

- Reduction:
  - Step1: For a sequence of n numbers $< a_1, a_2, \cdots, a_n >$, construct n points in the two dimensional plane $< (a_1, a_1^2), (a_2, a_2^2), \cdots, (a_n, a_n^2) >$, which is an input of the Convex Hull problem.
  - Step2: Using any algorithm for the Convex Hull problem to solve the constructed instance, the output will be a list of the constructed points sorted by their x-coordinates.
  - Step3: Traversing the list and outputting the fist coordinate of each point gives the correct output of the sorting problem.



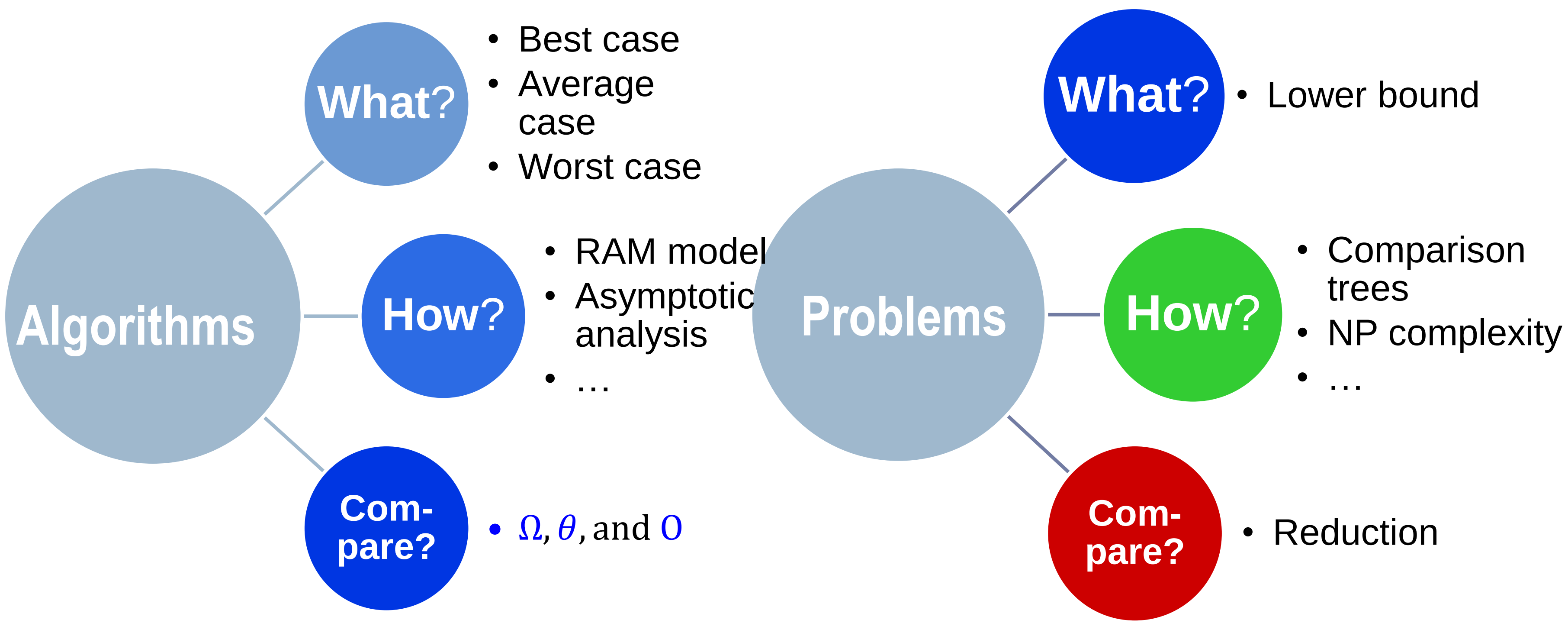
Graham's Scan

- We know the sorting problem (comparison-based) has a lower bound of $\Omega(n \lg n)$.
- $SORTING \propto_n Convex\ Hull$
- The Convex Hull problem has a lower bound of $\Omega(n \lg n)$.
- The famous Graham's Scan algorithm achieves this lower bound.



Credit: http://courses.washington.edu/css503

Summary


Algorithms
What?
Best case
Average case
Worst case
How?
RAM model
Asymptotic analysis
…
Com-pare?
Ω, θ, and O
Problems
What?
Lower bound
How?
Comparison trees
NP complexity
…
Com-pare?
Reduction