UC Davis, ECS20, Spring 2019

Discrete Mathematics for Computer Science

Prof. Raissa D'Souza
(slides adopted from Michael Frank and Haluk Bingöl)

Lecture 12

Complexity of Algorithms 3.2-3.3

Recall Lec 10 on algorithms How long does each algorithm take to run?

- How do the number of operations scale with input length *n*?
- Worst-case complexity
  - e.g. Search list of length *n* and the item to find is in position a_n
- Best-case complexity
  - e.g. Search list of length *n* and the item to find is in position a_1
- Average-case complexity / Typical-case complexity

Average case complexity

- e.g., for an unordered list of length *n*
- With probability 1/n the element is in position a_j for *j* ranging from 1 to n.
- Let M(j) denote the number of operations needed if the desired item is in position a_j
- Average case:

$$(1/n) \sum_{j=1}^{n} M(j)$$

Specifying an algorithm: Our Pseudocode Language:

- **procedure** *name*(*argument: type*)
- *variable* := *expression*
- **begin** *statements* **end**
- {*comment*}
- **if** *condition* then *statement* [else *statement*]
- **for** *variable* := *initial value* to *final value* *statement*
- **while** *condition* *statement*
- **return** *expression*

procedure *procname*(*arg*: *type*)

- Often just say "*procname(arg)*" without the type.

- Declares that the following text defines a procedure named *procname* that takes inputs (*arguments*) named *arg* which are data objects of the type *type*.
  - Example:
    procedure *maximum*(*L*: list of integers)
    [statements defining *maximum*…]

variable := *expression*

- An *assignment* statement evaluates the expression *expression*, then reassigns the variable *variable* to the value that results.
  - Example assignment statement:
    v := 3*x*+7 (If *x* is 2, changes *v* to 13.)
  - Also use
  - *v* <- 3x + 7

Informal statement

- Sometimes we may write a statement as an informal English imperative, if the meaning is still clear and precise: *e.g.*, "swap *x* and *y*"
- Keep in mind that real programming languages never allow this.
- When we ask for an algorithm to do so-and-so, writing "Do so-and-so" isn't enough!
  - Break down algorithm into detailed steps.

Swap two values

- V1 := 10
- V2 := 20

```
badswap (V1, V2)
        V1 = V2
        V2 = V1        (What ends up being the value of V2?)

swap (V1, V2)
        temp := V1
        V1 = V2
        V2 = temp    (This one works)
```

begin *statements* **end**

- Groups a sequence of statements together:

begin
  statement 1
  statement 2
  …
  statement n
end

Curly braces {} are used instead in many languages.

- Allows the sequence to be used just like a single statement.
- Might be used:
  - After a **procedure** declaration.
  - In an **if** statement after **then** or **else**.
  - In the body of a **for** or **while** loop.

{*comment*}

- Not executed (does nothing).
- Natural-language text explaining some aspect of the procedure to human readers.
- Also called a *remark* in some real programming languages, *e.g.* BASIC.
- Example, might appear in a *max* program:
  - {Note that *v* is the largest integer seen so far.}

if *condition* **then** *statement*

- Evaluate the propositional expression *condition*.
  - If the resulting truth value is **True**, then execute the statement *statement*;
  - otherwise, just skip on ahead to the next statement after the **if** statement.

- Variant: **if** *cond* **then** *stmt1* **else** *stmt2*
  - Like before, but iff truth value is **False**, executes *stmt2*.

while *condition statement*

- *Evaluate* the propositional (Boolean) expression *condition*.
- If the resulting value is **True**, then execute *statement*.
- Continue repeating the above two actions over and over until finally the *condition* evaluates to **False**; then proceed to the next statement.

while *condition statement*

- Also equivalent to infinite nested **if**s, like so:

```
if condition
    begin
        statement
        if condition
            begin
                statement
                ...(continue infinite nested if's)
            end
    end
```

for *var* := *initial* to *final* *stmt*

- *Initial* is an integer expression.
- *Final* is another integer expression.
- **Semantics:** Repeatedly execute *stmt*, first with variable *var* := *initial*, then with *var* := *initial*+1, then with *var* := *initial*+2, *etc.*, then finally with *var* := *final*.
- **Question:** What happens if *stmt* changes the value of var, or the value that *initial* or *final* evaluates to?

for *var* := *initial* to *final* *stmt*

- **For** can be exactly defined in terms of **while,** like so:

```
begin
    var := initial
    while var ≤ final
        begin
            stmt
            var := var + 1
        end
end
```

Now writing Alg 1: find max

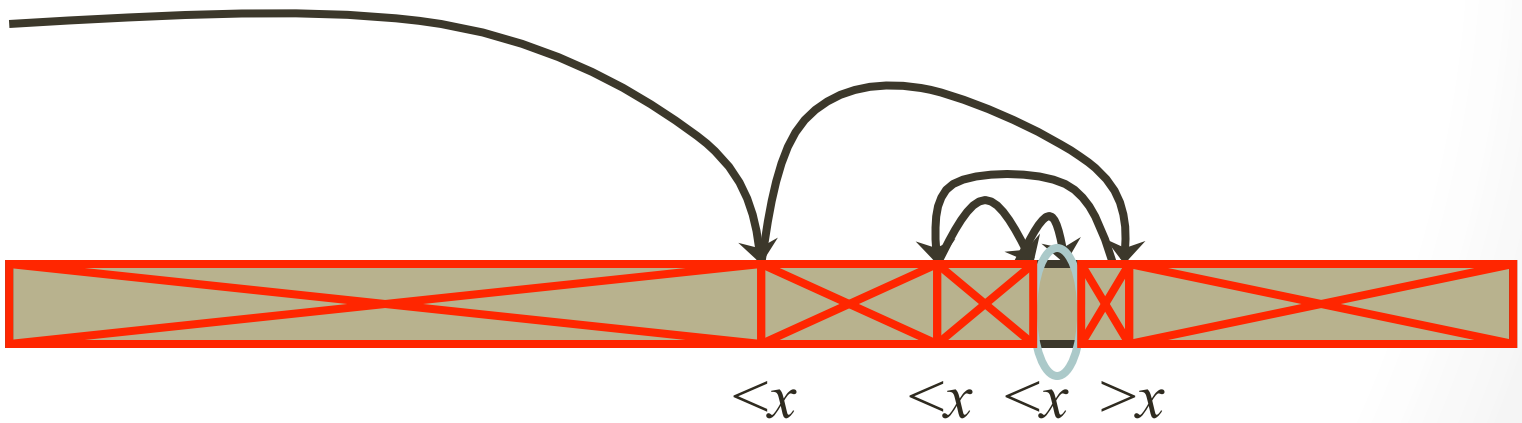
Algorithm 1 An algorithm for finding the maximum element in a finite sequence $a = \{a_1, \ldots, a_n\}$ of integers.

FIND-MAX (a)

```
max ← a_1
for i ← 2, 3, ..., n do
    if max < a_i then
        max ← a_i
    end if
end for
return max {max is the largest element.}
```

Alg. #3: Binary Search

- Basic idea: On each step, look at the *middle* element of the remaining list to eliminate half of it, and quickly zero in on the desired element.



Alg. #3: Binary Search

- **procedure** *binary search*
  (x:integer, a_1, a_2, …, a_n: distinct integers)
  i := 1 {left endpoint of search interval}
  j := n {right endpoint of search interval}
  while $i<j$ **begin** {while interval has >1 item}
  m := $\lfloor (i+j)/2 \rfloor$ {midpoint}
  if $x>a_m$ **then** i := $m+1$ **else** j := m
  end
  if $x = a_i$ **then** *location* := i **else** *location* := 0
  return *location*

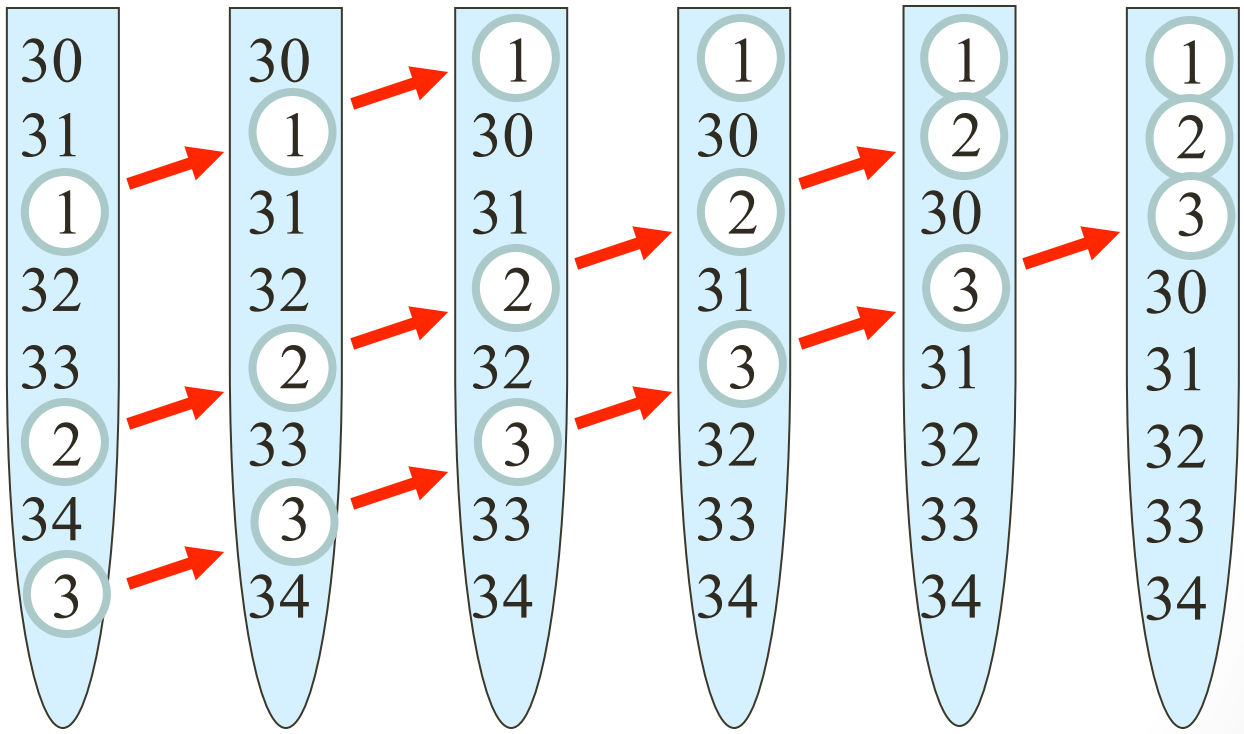
Sorting Algorithms

- Sorting is a common operation in many applications.
  - *E.g.* spreadsheets and databases
- It is also widely used as a subroutine in other data-processing algorithms.
- Two sorting algorithms shown in textbook:
  - Bubble sort
  - Insertion sort

However, these are *not* very efficient, and you should not use them on large data sets!

Bubble Sort (Alg #4)

- Smallest elements "float" up to the top of the list, like bubbles in a container of liquid.



Review §3.1: Algorithms

- Characteristics of algorithms.
- Pseudocode.
- Examples: Max algorithm, linear search & binary search algorithms, sorting.
- Intuitively we see that binary search is much faster than linear search, but how do we analyze the efficiency of algorithms formally?
- Use methods of *algorithmic complexity*, which utilize the order-of-growth concepts from §1.8.

See handout 7

- Linear search (unsorted list)
- Binary search (needs sorted list)
- Bubble sort
- Insertion sort
  - (note *"for k in 0 to -1"* does not execute)
- Greedy algorithms
  - Make the best choice given what you know.
  - Internet routing algorithms
  - Making change with minimal number of coins

How long does each algorithm take to run?

- How do the number of operations scale with input length n?

- Worst-case complexity
  - e.g. Search list of length n and the item to find is in position a_n
- Best-case complexity
  - e.g. Search list of length n and the item to find is in position a_1
- Average-case complexity / Typical-case complexity

How to we generalize this?

- In general we do not care about the exact number of operations required. We want to estimate or bound this by some function of *n*. (Where *n* is the length of the input)

Growth of functions (§3.2)

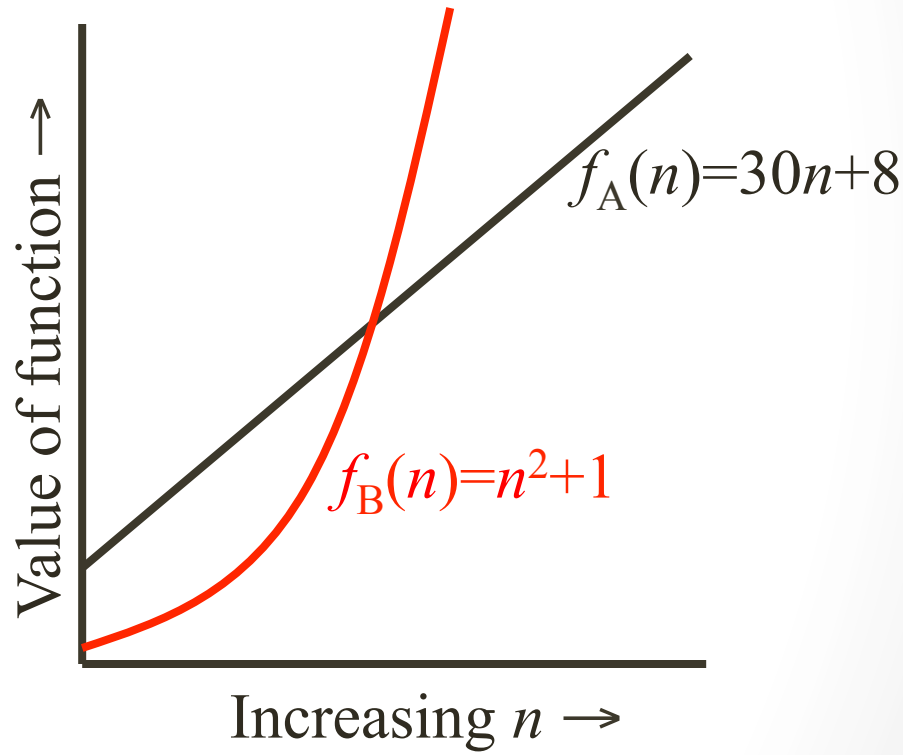
- For functions over numbers, we often need to know a rough measure of *how fast a function grows*.

- If $f(x)$ is *faster growing* than $g(x)$, then $f(x)$ always eventually becomes larger than $g(x)$ *in the limit* (for <u>large enough</u> values of x).

- Useful in engineering for showing that one design *scales* better or worse than another.

Visualizing Orders of Growth

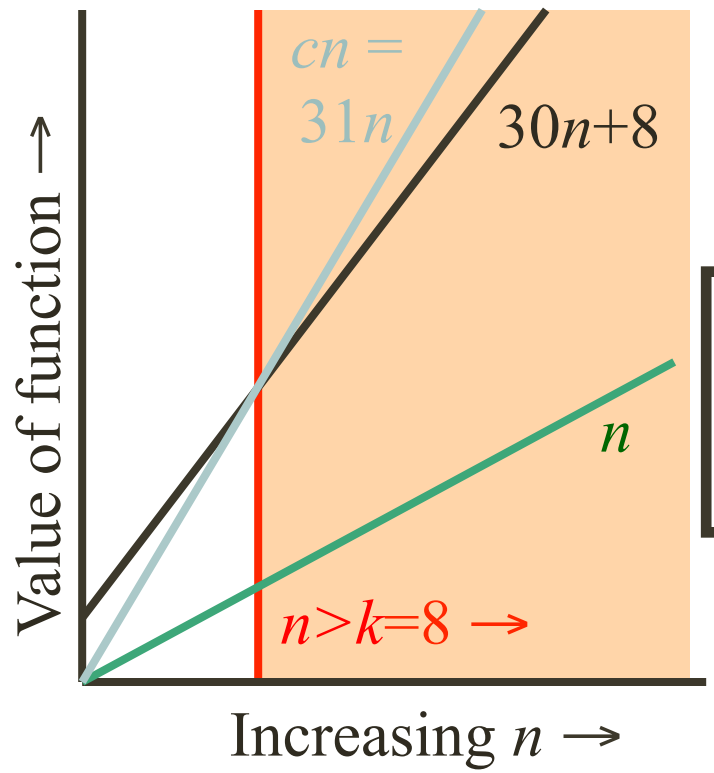
- On a graph, as you go to the right, the faster-growing function always eventually becomes the larger one...



$f_B(n) > f_A(n)$, for $n>k$

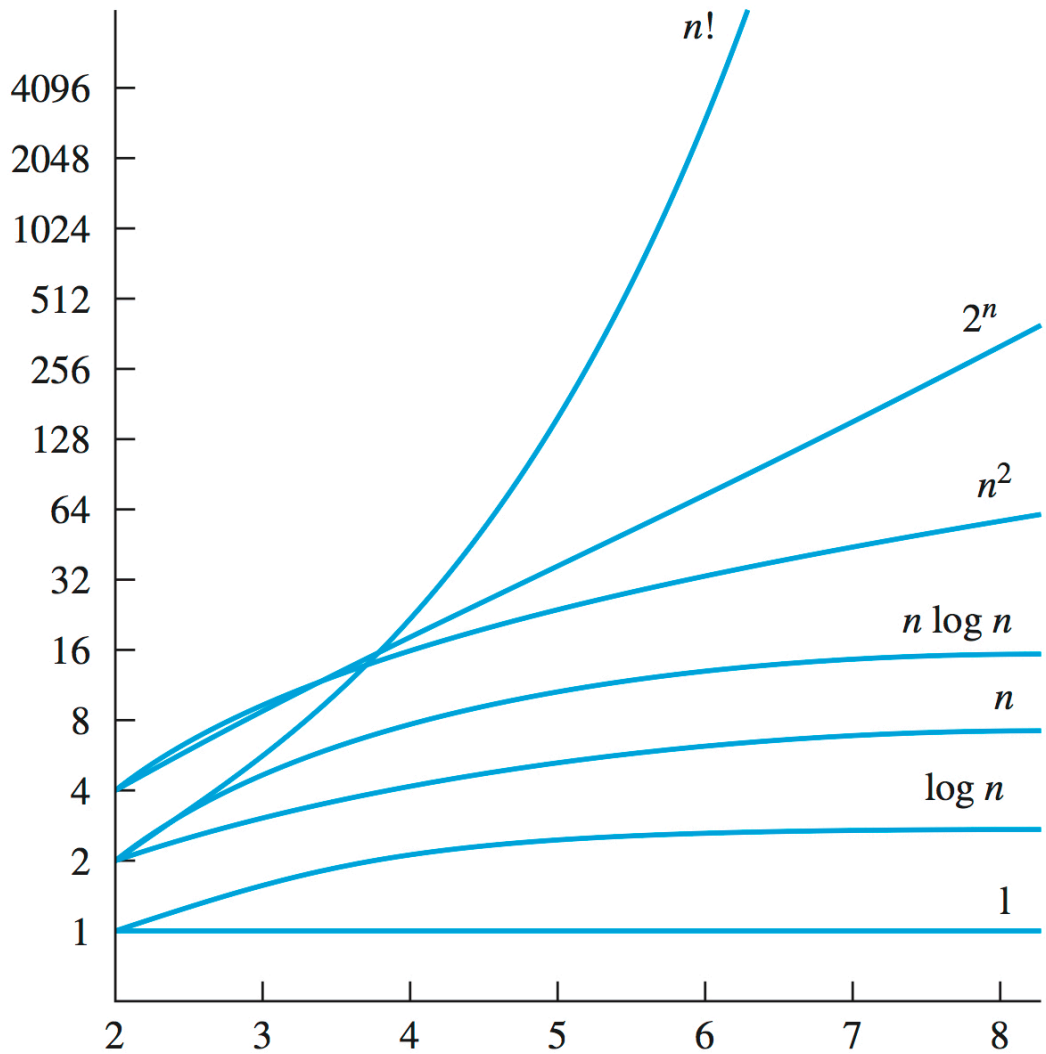
Another example, graphically

- Note $30n+8$ isn't less than n *anywhere* ($n>0$).
- It isn't even less than $31n$ *everywhere*.
- But it *is* less than $31n$ <u>everywhere to the right of $n=8$</u>.



$$30n+8 \in O(n)$$

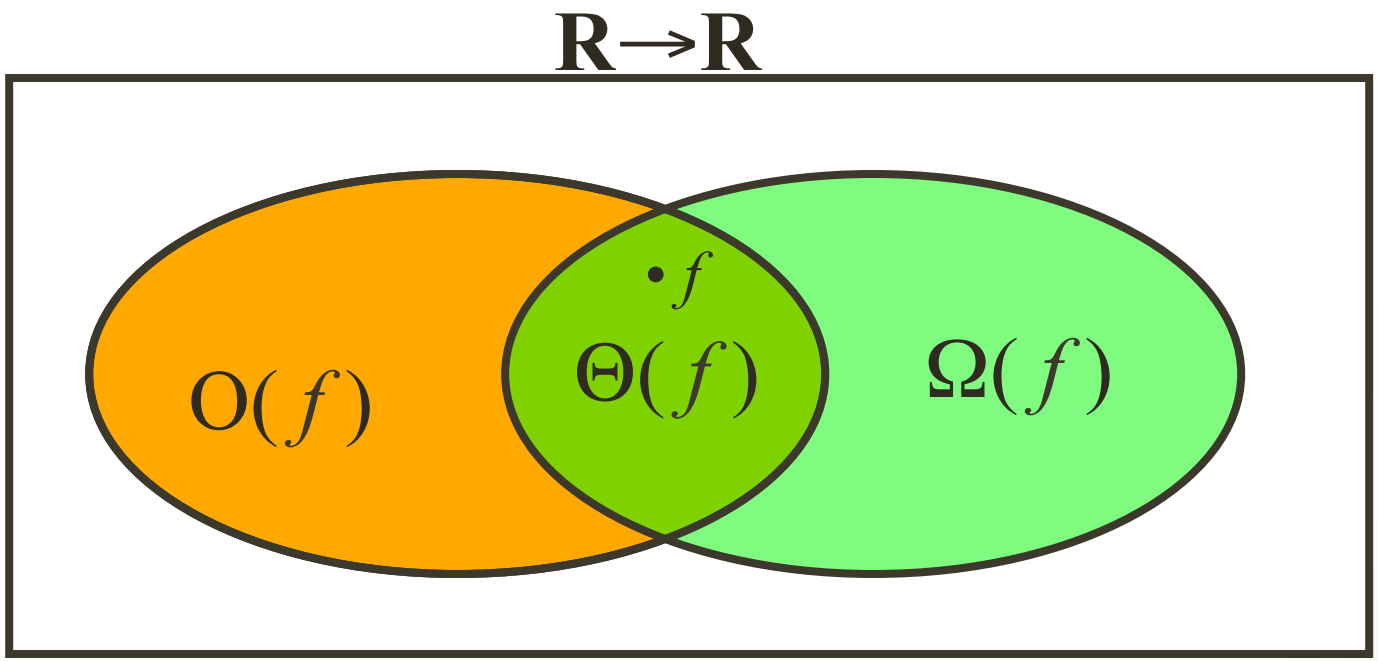
Common reference functions



Handout 8

Relations Between the Relations

- Subset relations between order-of-growth sets.



Complexity of Algorithms

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Computer time needed

TABLE 2 **The Computer Time Used by Algorithms.**

Problem Size	*Bit Operations Used*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

Computational complexity classes

- Tractable problems are polynomial complexity or less. Said to belong to computational complexity class P, for "polynomial"
- Exponential and factorial complexity are said to be "intractable".
- Special class NP (non-deterministic polynomial)

NP problems

- No polynomial time algorithm for solving the problem
- But can check/validate a potential solution in polynomial time.
- E.g. factoring large integers
- Central belief: $P \neq NP$
  - Proving this is one of the Millennium Prize problems (win $1 million!!)