# CS481: Bioinformatics Algorithms

Can Alkan
EA224
calkan@cs.bilkent.edu.tr

**http://www.cs.bilkent.edu.tr/~calkan/teaching/cs481/**

# Reminder

- The TA will hold a few recitation sessions for the students from non-CS departments
    - Quick version of CS201 and CS202
    - Details of big-oh notation
    - Basic data structures
    - Email your schedules to ekayaaslan@gmail.com

# Computational complexity (basic)

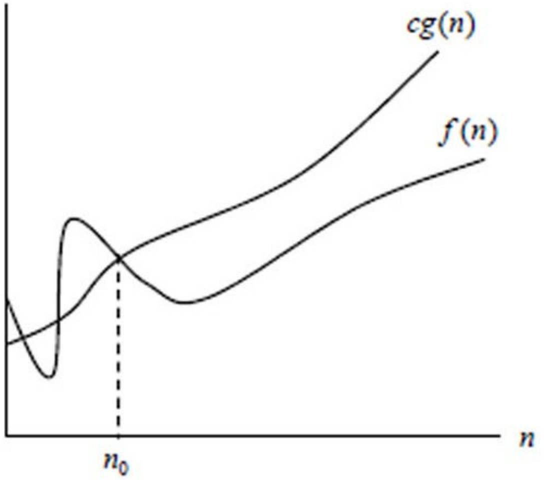
- When we develop or use an algorithm, we would like to know how its run time and memory requirements will scale with respect to data size
- Big-O Notation, and its counterparts: Limiting behavior of a function
    - **O(f(x)): Upper bound**
    - Ω(f(x)): Lower bound
    - Θ(f(x)): Tight bound

# Bounds

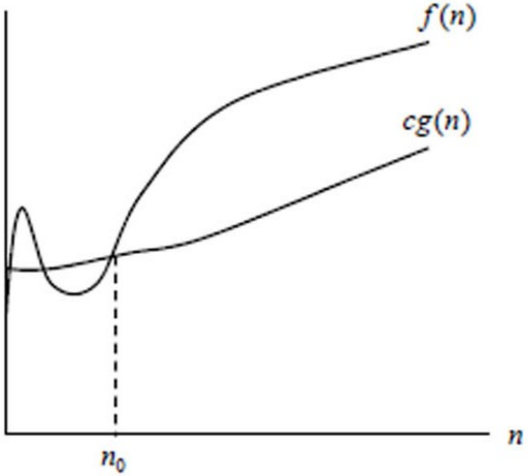
- $f(x)$ is $O(g(x))$ if there are positive real constants $c$ and $x_0$ such that $f(x) \leq cg(x)$ for all values of $x \geq x_0$.
- $f(x)$ is $\Omega(g(x))$ if there are positive real constants $c$ and $x_0$ such that $f(x) \geq cg(x)$ for all values of $x \geq x_0$.
- $f(x)$ is $\Theta(g(x))$ if $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$

# Bounds

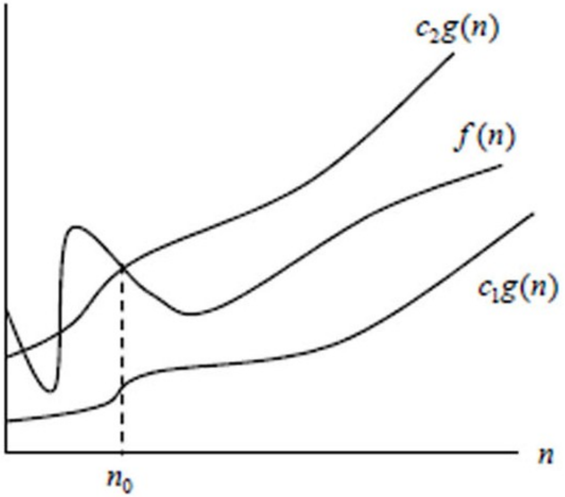
**f(n)=O(g(n))**



**f(n)=Ω(g(n))**



**f(n)=Θ(g(n))**

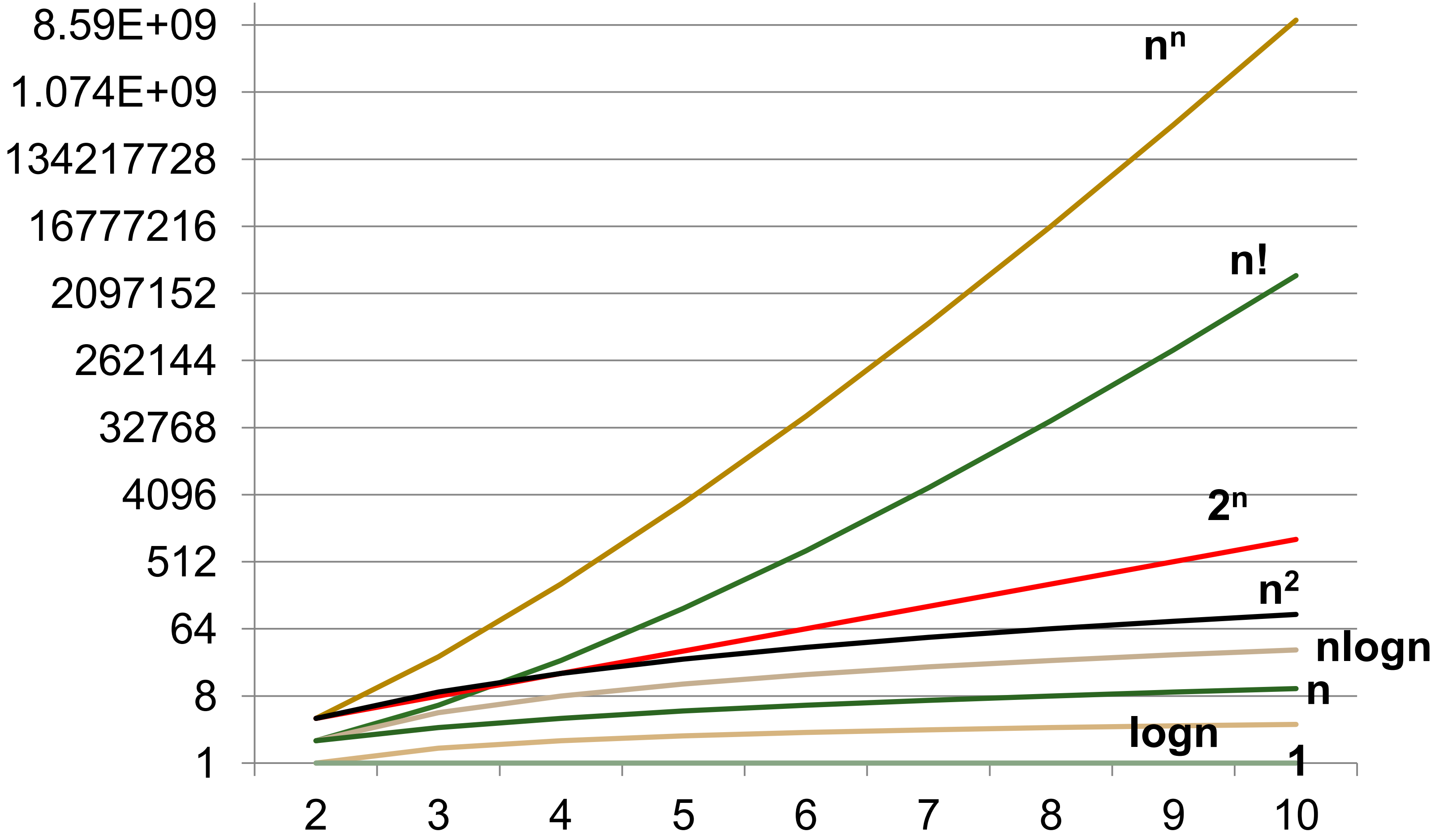


$n^2 = O(n^2)$
$n^2 + n = O(n^2)$
$n^2 + 1000n = O(n^2)$
$5000n^2 + 1000n = O(n^2)$

**Constants do not matter!**

http://meherchilakalapudi.wordpress.com/2012/09/14/data-structures-1asymptotic-analysis/

# Fast vs. slow algorithms



# Polynomial vs. exponential

- Polynomial algorithms: run time is bounded by a polynomial function (addition, subtraction, multiplication, division, non-negative integer exponents)
    - $n$, $n^2$, $n^{5000}$, etc.
- Exponential algorithms: run time is bounded by an exponential function, where exponent is n
    - $n^n$, $2^n$, etc.

# Fast vs. Slow: Fibonacci

- Fibonacci series:
  - $F_n = F_{n-1} + F_{n-2}$
  - $F_1 = F_2 = 1$
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, …

# Two Fibonacci algoritms

```
RecursiveFibonacci($n$)
if $n = 1$ or $n = 2$
    return 1
else
    $a \leftarrow$ RecursiveFibonacci($n - 1$)
    $b \leftarrow$ RecursiveFibonacci($n - 2$)
    return $a + b$
```

**$O(2^n)$**

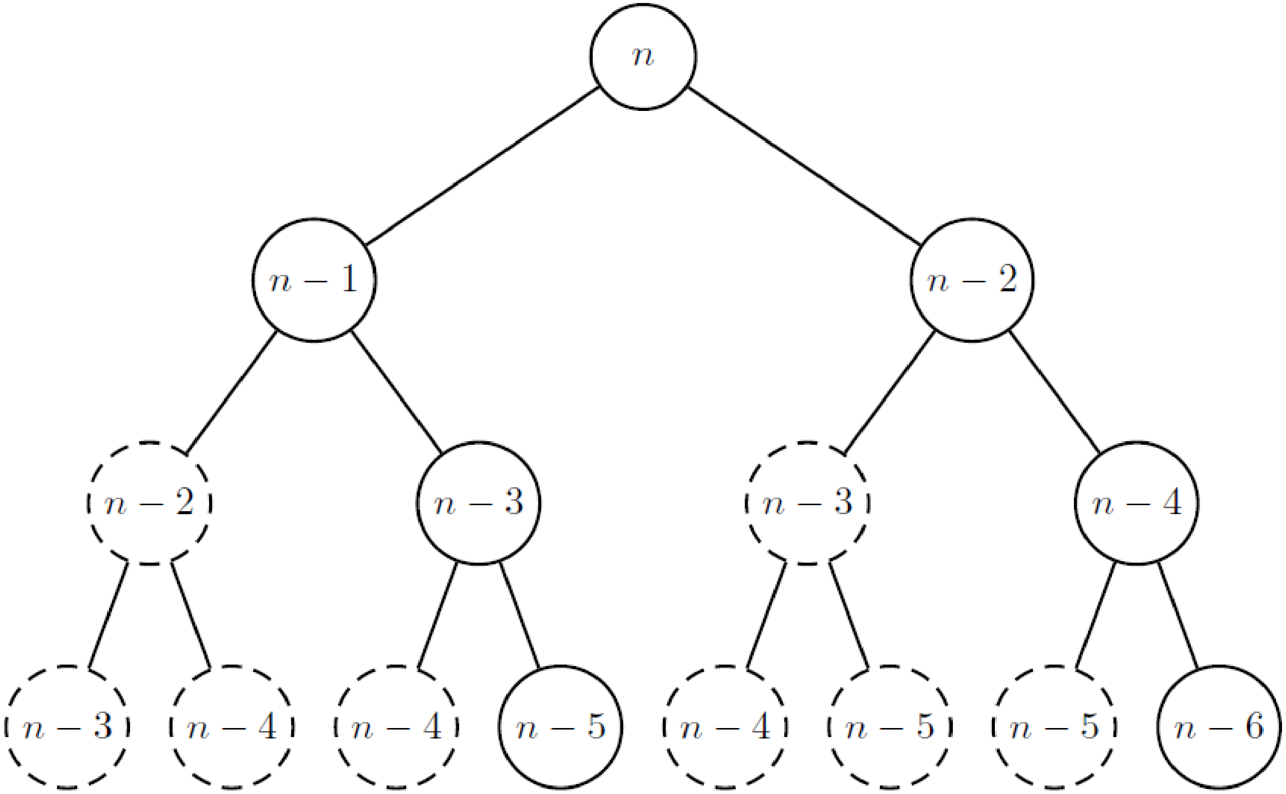
```
Fibonacci($n$)
$F_1 \leftarrow 1$
$F_2 \leftarrow 1$
for $i \leftarrow 3$ to $n$
    $F_i \leftarrow F_{i-1} + F_{i-2}$
return $F_n$
```

**$O(n)$**

# Recursion or no recursion?

**Why is it not a good idea to write recursive algorithms when you *can* write non-recursive versions?**

# Recursion tree for Fibonacci

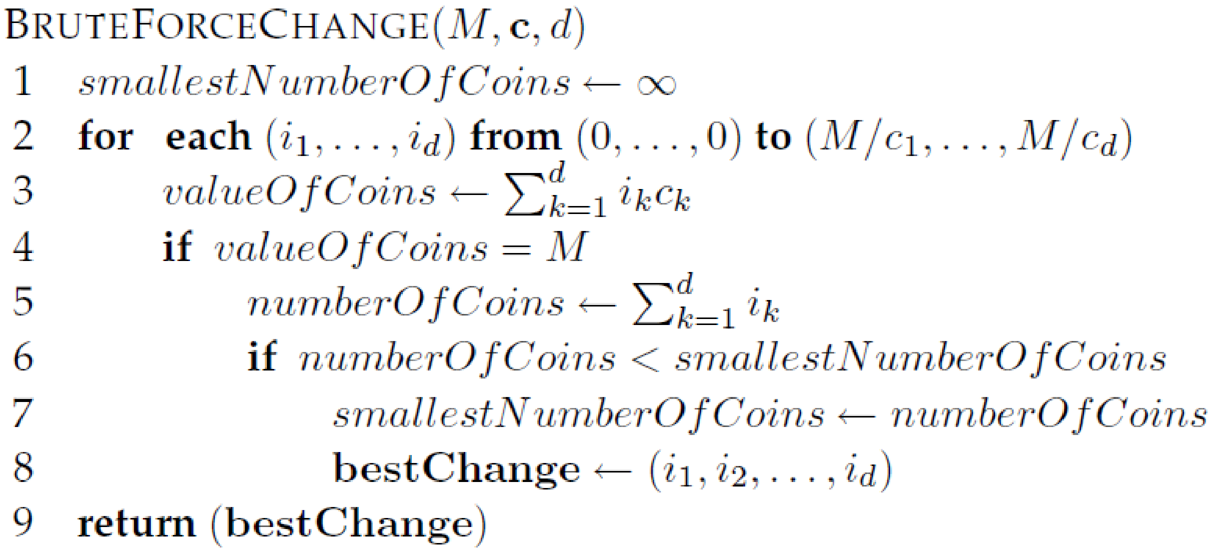


# Sample problem: Change

- Input: An amount of money M, in cents
- Output: Smallest number of coins that adds up to M
  - Quarters (25c): q
  - Dimes (10c): d
  - Nickels (5c): n
  - Pennies (1c): p
  - Or, in general, $c_1$, $c_2$, …, $c_d$ (*d* possible denominations)

# Algorithm design techniques

- **Exhaustive search / brute force**
  - Examine every possible alternative to find a solution



```
BruteForceChange($M, \mathbf{c}, d$)
    $smallestNumberOfCoins \leftarrow \infty$
    for each $(i_1, \ldots, i_d)$ from $(0, \ldots, 0)$ to $(M/c_1, \ldots, M/c_d)$
        $valueOfCoins \leftarrow \sum_{k=1}^{d} i_k c_k$
        if $valueOfCoins = M$
            $numberOfCoins \leftarrow \sum_{k=1}^{d} i_k$
            if $numberOfCoins < smallestNumberOfCoins$
                $smallestNumberOfCoins \leftarrow numberOfCoins$
                bestChange $\leftarrow (i_1, i_2, \ldots, i_d)$
    return (bestChange)
```

# Algorithm design techniques

- **Branch and bound:**
  - Omit a large number of alternatives when performing brute force

# Algorithm design techniques

- **Greedy algorithms:**
  - Choose the “most attractive” alternative at each iteration

```
BETTERCHANGE($M, \mathbf{c}, d$)
$r \leftarrow M$
for $k \leftarrow 1$ to $d$
    $i_k \leftarrow r/c_k$
    $r \leftarrow r - c_k \cdot i_k$
return $(i_1, i_2, \ldots, i_d)$
```

```
USCHANGE($M$)
$r \leftarrow M$
$q \leftarrow r/25$
$r \leftarrow r - 25 \cdot q$
$d \leftarrow r/10$
$r \leftarrow r - 10 \cdot d$
$n \leftarrow r/5$
$r \leftarrow r - 5 \cdot n$
$p \leftarrow r$
return $(q, d, n, p)$
```

# Algorithm design techniques

- **Dynamic Programming:**
    - Break problems into subproblems; solve subproblems; merge solutions of subproblems to solve the real problem
    - Keep track of computations to avoid recomputing values that you already solved
    - *Dynamic programming table*

# DP example: Rocks game

- Two players
- Two piles of rocks with $p_1$ rocks in pile 1, and $p_2$ rocks in pile 2
- In turn, each player picks:
  - One rock from either pile 1 or pile 2; OR
  - One rock from pile 1 and one rock from pile2
- The player that picks the last rock wins

# DP algorithm for Player 1

- Problem: $p_1 = p_2 = 10$
- Solve more general problem of $p_1 = n$ and $p_2 = m$
- It's hard to directly calculate for n=5 and m=6; we need to solve smaller problems

# DP algorithm for Player 1

*pile2*

*pile1*

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | W | | | | | | | | | |
| 1 | W | W | | | | | | | | | |
| 2 | | | | | | | | | | | |
| 3 | | | | | | | | | | | |
| 4 | | | | | | | | | | | |
| 5 | | | | | | | | | | | |
| 6 | | | | | | | | | | | |
| 7 | | | | | | | | | | | |
| 8 | | | | | | | | | | | |
| 9 | | | | | | | | | | | |
| 10 | | | | | | | | | | | |

**Initialize; obvious win for Player 1 for 1,0; 0,1 and 1,1**

# DP algorithm for Player 1

| | *pile2* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | | W | L | | | | | | | | |
| *pile1* | 1 | W | W | | | | | | | | | |
| | 2 | L | | | | | | | | | | |
| | 3 | | | | | | | | | | | |
| | 4 | | | | | | | | | | | |
| | 5 | | | | | | | | | | | |
| | 6 | | | | | | | | | | | |
| | 7 | | | | | | | | | | | |
| | 8 | | | | | | | | | | | |
| | 9 | | | | | | | | | | | |
| | 10 | | | | | | | | | | | |

**Player 1 cannot win for 2,0 and 0,2**

# DP algorithm for Player 1

*pile2*

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | | W | L | | | | | | | | |
| *pile1* | 1 | W | W | W | | | | | | | | |
| | 2 | L | W | | | | | | | | | |
| | 3 | | | | | | | | | | | |
| | 4 | | | | | | | | | | | |
| | 5 | | | | | | | | | | | |
| | 6 | | | | | | | | | | | |
| | 7 | | | | | | | | | | | |
| | 8 | | | | | | | | | | | |
| | 9 | | | | | | | | | | | |
| | 10 | | | | | | | | | | | |

**Player 1 can win for 2,1 if he picks one from pile2**

**Player 1 can win for 1,2 if he picks one from pile1**

# DP algorithm for Player 1

*pile2*

*pile1*

**Player 1 can win for 2,1 if he picks one from pile2**

**Player 1 can win for 1,2 if he picks one from pile1**

# DP algorithm for Player 1

*pile2*

*pile1*

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | W | L | | | | | | | | |
| 1 | W | W | W | | | | | | | | |
| 2 | L | W | L | | | | | | | | |
| 3 | | | | | | | | | | | |
| 4 | | | | | | | | | | | |
| 5 | | | | | | | | | | | |
| 6 | | | | | | | | | | | |
| 7 | | | | | | | | | | | |
| 8 | | | | | | | | | | | |
| 9 | | | | | | | | | | | |
| 10 | | | | | | | | | | | |

**Player 1 cannot win for 2,2**

**Any move causes his opponent to go to W state**

# DP “moves”

**When you are at position $(i,j)$**

***Go to:***

**Pick from pile 1:** $(i-1, j)$

**Pick from pile 2:** $(i, j-1)$

**Pick from both piles 1 and 2:** $(i-1, j-1)$

# DP final table

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | W | L | W | L | W | L | W | L | W | L |
| 1 | W | W | W | W | W | W | W | W | W | W | W |
| 2 | L | W | L | W | L | W | L | W | L | W | L |
| 3 | W | W | W | W | W | W | W | W | W | W | W |
| 4 | L | W | L | W | L | W | L | W | L | W | L |
| 5 | W | W | W | W | W | W | W | W | W | W | W |
| 6 | L | W | L | W | L | W | L | W | L | W | L |
| 7 | W | W | W | W | W | W | W | W | W | W | W |
| 8 | L | W | L | W | L | W | L | W | L | W | L |
| 9 | W | W | W | W | W | W | W | W | W | W | W |
| 10 | L | W | L | W | L | W | L | W | L | W | L |

**Also keep track of the choices you need to make to achieve W and L states: *traceback table***

# Algorithm design techniques

- **Divide and conquer:**
  - Split, solve, merge
    - Mergesort
- **Machine learning:**
  - Analyze previously available solutions, calculate statistics, apply most likely solution
- **Randomized algorithms:**
  - Pick a solution randomly, test if it works. If not, pick another random solution

# Tractable vs intractable

- Tractable algorithms: there exists a solution with O(f(n)) run time, where f(n) is *polynomial*
- *P* is the set of problems that are **known** to be *solvable* in polynomial time
- *NP* is the set of problems that are *verifiable* in polynomial time
  - *NP: “non-deterministic polynomial”*

$$P \subset NP$$

# NP-hard

- *NP-hard:* non-deterministic polynomial hard
  - Set of problems that are "*at least as hard as the hardest problems in NP*"
  - There are no known polynomial time optimal solutions
  - There *may* be polynomial-time *approximate* solutions

# NP-Complete

- A *decision problem* C is in NPC if :
  - C is in NP
  - Every problem in NP is **reducible** to C in polynomial time

That means: if you could solve any NPC problem in polynomial time, then you can solve all of them in polynomial time

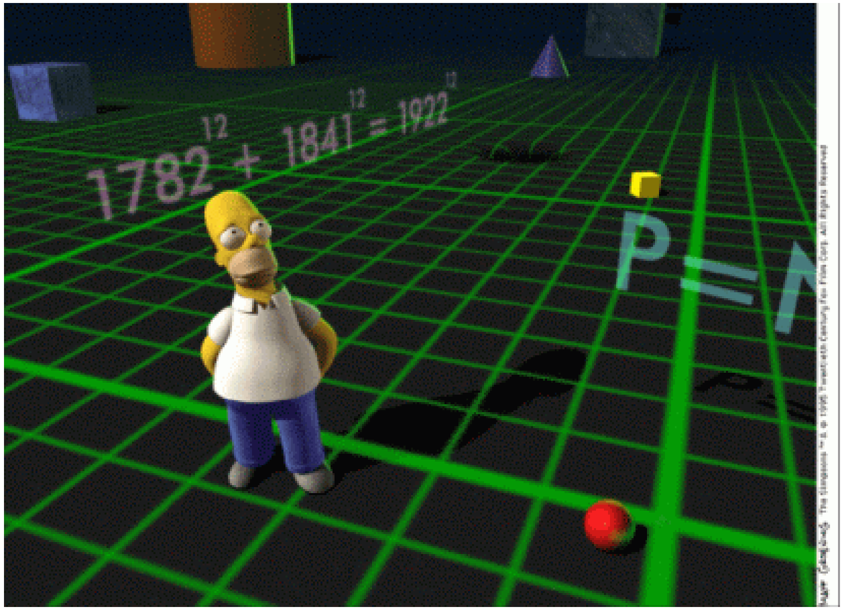
*Decision problems*: outputs “yes” or “no”

# NP-intermediate

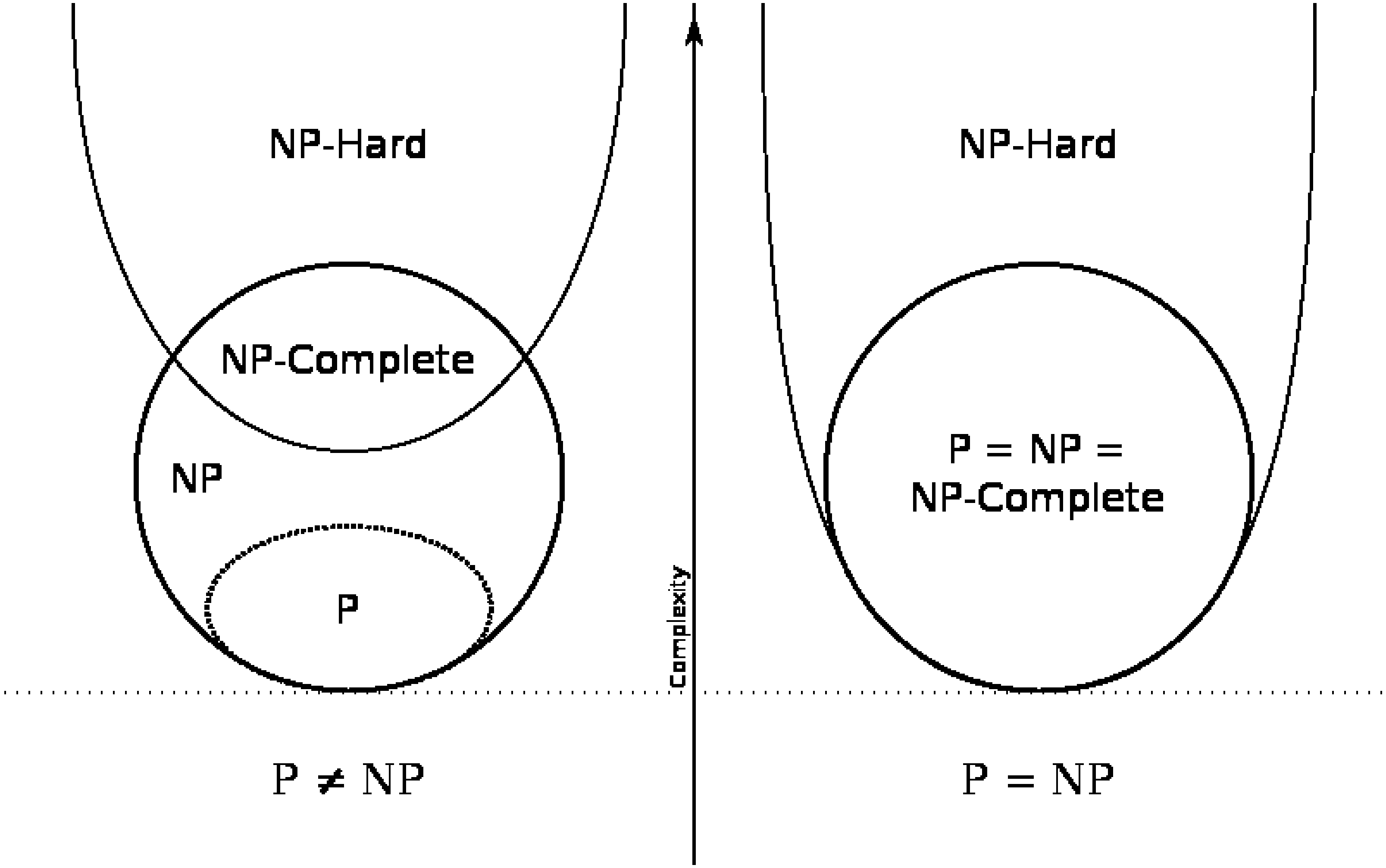
- Problems that are in NP; but not in either NPC or NP-hard

# P vs. NP

- We do not know whether P=NP or P≠NP
  - Principal unsolved problem in computer science
  - It is believed that P≠NP



# P vs. NP vs. NPC vs. NP-hard



P ≠ NP

P = NP

# Examples

- P:
  - Sorting numbers, searching numbers, pairwise sequence alignment, etc.
- NP-complete:
  - Subset-sum, traveling salesman, etc.
- NP-intermediate:
  - Factorization, graph isomorphism, etc.

# Historical reference

- The notion of NP-Completeness: Stephen Cook and Leonid Levin independently in 1971
  - First NP-Complete problem to be identified: Boolean satisfiability problem (SAT)
    - Cook-Levin theorem
- More NPC problems: Richard Karp, 1972
  - "21 NPC Problems"
- Now there are thousands….