CMSC 132:
Object-Oriented Programming II

Algorithmic Complexity II

Department of Computer Science

University of Maryland, College Park

Overview

- Critical sections
- Comparing complexity
- Types of complexity analysis

Analyzing Algorithms

- Goal
  - Find asymptotic complexity of algorithm
- Approach
  - Ignore less frequently executed parts of algorithm
  - Find critical section of algorithm
  - Determine how many times critical section is executed as function of problem size

Critical Section of Algorithm

- Heart of algorithm
- Dominates overall execution time
- Characteristics
  - Operation central to functioning of program
  - Contained inside deeply nested loops
  - Executed as often as any other part of algorithm
- Sources
  - Loops
  - Recursion

Critical Section Example 1

- **Code (for input size n)**

```
1. A
2. for (int i = 0; i < n; i++) {
3.     B
4. }
5. C
```

critical section

- **Code execution**
  - **A ⇒ once**
  - **B ⇒ n times**
  - **C ⇒ once**
- **Time ⇒ 1 + n + 1 = O(n)**

Critical Section Example 2

- Code (for input size n)

```
1. A
2. for (int i = 0; i < n; i++) {
3.    B
4.    for (int j = 0; j < n; j++) {
5.       C
6.    }
7. } D
```

critical section

- Code execution
  - A $\Rightarrow$ once
  - B $\Rightarrow$ n times
  - C $\Rightarrow$ n^2 times
  - D $\Rightarrow$ once
- Time $\Rightarrow 1 + n + n^2 + 1 = O(n^2)$

Critical Section Example 3

- Code (for input size n)

```
1. A
2. for (int i = 0; i < n; i++) {
3.     for (int j = i+1; j < n; j++) {
4.         B
5.     }
6. }
```

critical section

- Code execution
  - A $\Rightarrow$ once
  - B $\Rightarrow$ ½ n (n-1) times
- Time $\Rightarrow 1 + \frac{1}{2} n^2 = O(n^2)$

Critical Section Example 4

- **Code (for input size n)**

```
1. A
2. for (int i = 0; i < n; i++) {
3.    for (int j = 0; j < 10000; j++) {
4.       B
5.    }
6. }
```

critical section

- **Code execution**
  - A $\Rightarrow$ once
  - B $\Rightarrow$ 10000 n times
- **Time $\Rightarrow$ 1 + 10000 n = O(n)**

Critical Section Example 5

- **Code (for input size n)**

```
1. for (int i = 0; i < n; i++) {
2.    for (int j = 0; j < n; j++)
3.       A
4. for (int i = 0; i < n; i++)
5.    for (int j = 0; j < n; j++)
6.       B
```

critical sections

- **Code execution**
  - A $\Rightarrow$ n^2 times
  - B $\Rightarrow$ n^2 times
- **Time** $\Rightarrow n^2 + n^2 = O(n^2)$

Critical Section Example 6

- Code (for input size n)
  1. $i = 1$
  2. while ($i < n$) {
  3. $\quad$ A
  4. $\quad i = 2 \times i$

     }
  5. B

critical section

- Code execution
  - $A \Rightarrow \log(n)$ times
  - $B \Rightarrow 1$ times
- Time $\Rightarrow \log(n) + 1 = O(\log(n))$

Critical Section Example 7

- **Code (for input size n)**
  1. **DoWork (int n)**
  2. **if (n == 1)**
  3. **A**
  4. **else {**
  5. **DoWork(n/2)**
  6. **DoWork(n/2)**
  7. **}**

critical sections

- **Code execution**
  - **A $\Rightarrow$ 1 times**
  - **DoWork(n/2) $\Rightarrow$ 2 times**
- **Time(1) $\Rightarrow$ 1** **Time(n) = 2 $\times$ Time(n/2) + 1**

Recursive Algorithms

- Definition
  - An algorithm that calls itself
- Components of a recursive algorithm
  1. Base cases
     - Computation with no recursion
  2. Recursive cases
     - Recursive calls
     - Combining recursive results

Recursive Algorithm Example

- **Code (for input size n)**

```
DoWork (int n)
   if (n == 1)
      A
   else {
      DoWork(n/2)
      DoWork(n/2)
   }
```

base case

recursive cases

Comparing Complexity

- Compare two algorithms
  - f(n), g(n)
- Determine which increases at faster rate
  - As problem size n increases
- Can compare ratio

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

  - If ∞, f() is larger
  - If 0, g() is larger
  - If constant, then same complexity

Complexity Comparison Examples

- $\log(n)$ vs. $n^{1/2}$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \rightarrow \lim_{n\to\infty} \frac{\log(n)}{n^{1/2}} \rightarrow 0$$

- 1.001^n vs. n^{1000}

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \rightarrow \lim_{n\to\infty} \frac{1.001^n}{n^{1000}} \rightarrow ??$$

Not clear, use L'Hopital's Rule

Additional Complexity Measures

- Upper bound
  - Big-O ⇒ $O(\ldots)$
  - Represents upper bound on # steps
- Lower bound
  - Big-Omega ⇒ $\Omega(\ldots)$
  - Represents lower bound on # steps
- Combined bound
  - Big-Theta ⇒ $\Theta(\ldots)$
  - Represents combined upper/lower bound on # steps
  - Best possible asymptotic solution

2D Matrix Multiplication Example

- **Problem**
  - **C = A * B**

- **Lower bound**
  - **$\Omega(n^2)$** **Required to examine 2D matrix**
- **Upper bounds**
  - **$O(n^3)$** **Basic algorithm**
  - **$O(n^{2.807})$** **Strassen's algorithm (1969)**
  - **$O(n^{2.376})$** **Coppersmith & Winograd (1987)**
- **Improvements still possible (open problem)**
  - **Since upper & lower bounds do not match**

Additional Complexity Categories

- **Name** **Description**
  - **P** Deterministic polynomial time
  - **NP** Nondeterministic polynomial time
  - **PSPACE** Polynomial space
  - **EXPSPACE** Exponential space
  - **Decidable** Can be solved by finite algorithm
  - **Undecidable** Not solvable by finite algorithm

- **If a problem has a algorithm that solves it in time X, then the problem is said to be in X**
  - **e.g., matrix multiplication is in P**

Definitions

- **N → measure of problem size**
- **Growth rate → what we care about. We need to measure it**
- **Exponential Time Algorithm ($O(k^n)$)→ algorithm requires trying every possibility (very slow)**
- **Polynomial Time Algorithm ($O(n^k)$)→ algorithm can solve a problem in smart/quick way without trying every possibility.**
  - **Smart/quick → without using trial-and-error or brute force approach**
  - **A problem that can be solved in time $O(n^k)$ for some constant k is solvable in polynomial time**

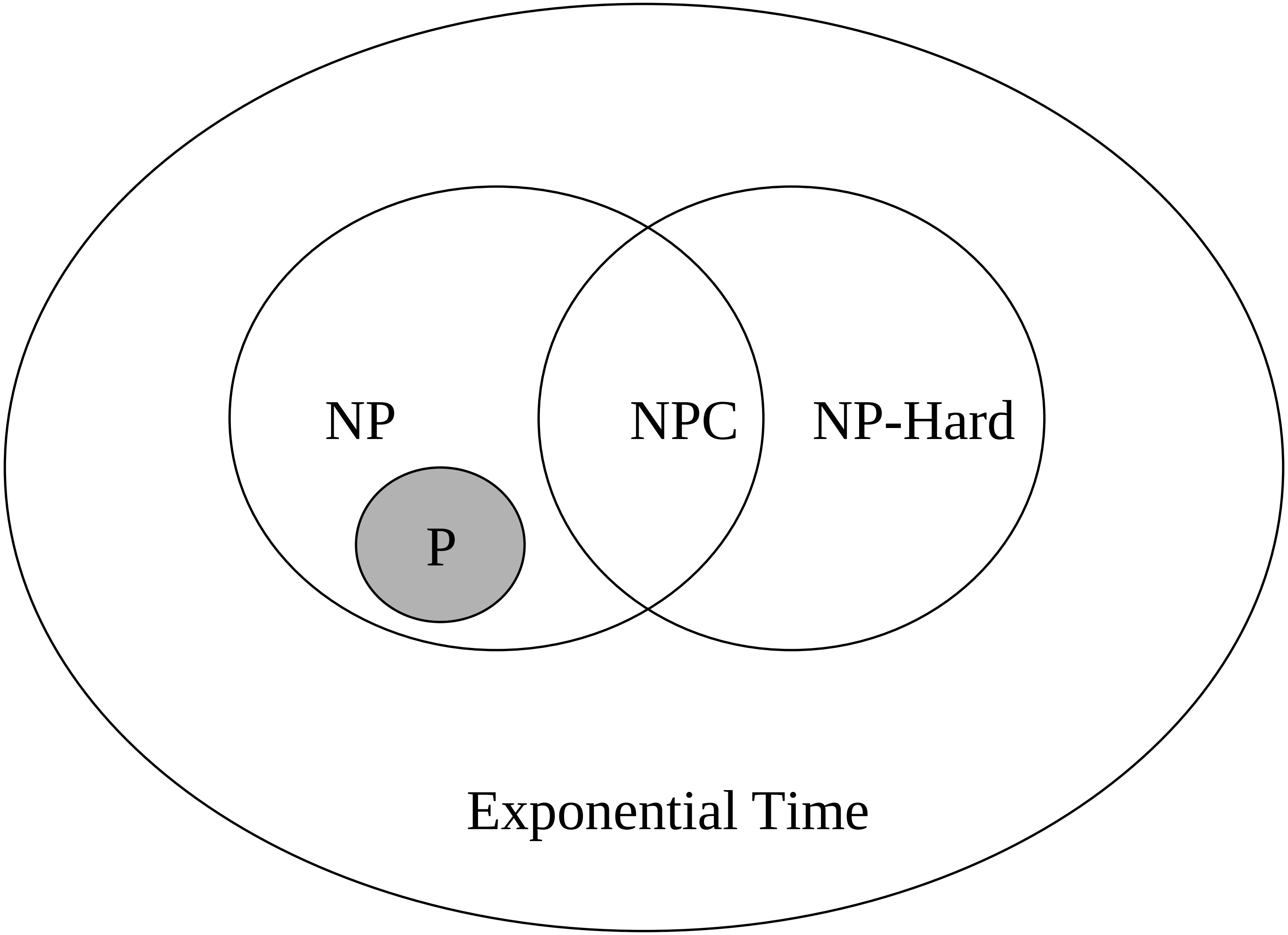
Typical Problem: TSP

- Traveling Salesman Problem (TSP)
  - Given → List of cities, distance between cities
  - Find → shortest tour visiting all cities returning to start city
- Brute force solution requires roughly 2^n steps
- If we can get it down to 2^{n-5} we are still doing brute force with some tricks
- How can we say that we are NOT doing brute force?
  - By doing it in polynomial time!
- Even n^{14} would be a real advance since its not brute force

Definitions

- **P Problem**
  - **Problem can be solved in polynomial time**
  - **P stands for polynomial and represents the set of all problems that can be solved in polynomial time.**
- **NP Problem**
  - **Solution to the problem can be verified in polynomial time**
  - **NP → does not stand for "not polynomial". Stands for nondeterministic polynomial time**
  - **Example→ Given a number, is it composite (not prime)?**
- **Being able to verify does not make the problem easy**

Type of Problems



Definitions

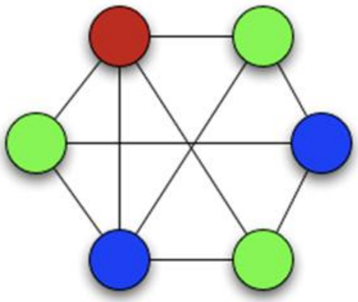
- Every P problem is an NP problem
  - Every problem in P is in NP
  - Do not confuse with P = NP
- Reduction
  - Showing one problem (X) is solvable, given a subroutine for another (Y). Solving Y solves X
- NP-hard Problem
  - Every problem in NP can be reduced to it

NPC (NP-Complete) Problem

- **Problem is both NP-hard and NP**
- **We can verify answer in polynomial time**
- **Every problem in NP reduces to it**
- **Solving any NP-Complete problem in polynomial time will solve every NP problem in polynomial time**
- **Many NPC problems, and so far no one has found a quick solution to any of them**

NP-Complete Problems

- More than 3000 NP-Complete problems
- Traveling Salesman Problem
  - Given → List of cities, distance between cities
  - Find → shortest tour visiting all cities returning to start city
- Graph Coloring



- Organizing House Accommodations

P vs NP?

- P = NP? asks:
  - Does every problem that can be verified quickly can also be solved quickly?
- No proof yet that P = NP or that P ≠ NP
- P vs NP is one of the Clay Mathematics Institute Millennium Prize Problems
  - You will receive 1 million dollars if you solve it
  - http://www.claymath.org/millennium/

References

- http://www.cs.princeton.edu/~kazad/resources/cs/npcomplete.htm
- Wikipedia
- Dr. William Gasarch
- Dr. David Mount
- Computers and Intractability: A Guide to the Theory of NP-Completeness, by Garey and D.S. Johnson
- Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein