Algorithms and their Applications CS2004 (2020-2021)

Dr Mahir Arzoky

4.1 Time Complexity and Asymptotic Notation

Notices

The Assessment Brief (Task #1 and #2) is released!

Previously On CS2004…

- So far we have looked at:
  - Concepts of Computation and Algorithms
  - Comparing algorithms
  - Some mathematical foundation

Time Complexity and Asymptotic Notation

- ❑ Within this lecture we will discuss:
  - ❑ The Selection sort algorithm
  - ❑ The Binary search algorithm
  - ❑ Big-Oh notation
  - ❑ This builds upon the core topic of counting **Primitive Operations** we covered two weeks ago...

Why Sorting?

- Sorting is one of the most common tasks in data analysis
- Examples:
  - Print out a collection of employees sorted by salary
  - Print out a list of names in alphabetical order
- There are many sorting algorithms
- The **Selection Sort** algorithm repeatedly finds the smallest element in the unsorted tail region of a list and moves it to the front

Selection Sort algorithm – Part 1

- Sorting an Array of Integers…
- Array in original order:

11	**9**	**17**	**5**	**12**

- Find the smallest and swap it with the first element:

5	**9**	**17**	**11**	**12**

Selection Sort algorithm – Part 2

- ❑ Find the next smallest
- ❑ It is already in the correct place

5	**9**	**17**	**11**	**12**

- ❑ Find the next smallest and swap it with the first element of the unsorted portion

5	**9**	**11**	**17**	**12**

Selection Sort algorithm – Part 3

- Repeat

5	9	11	12	17

- When the unsorted portion is of length 1, we are done

5	9	11	12	17

How Fast is the Algorithm?

With an array of size n, count how many primitive operations are needed:

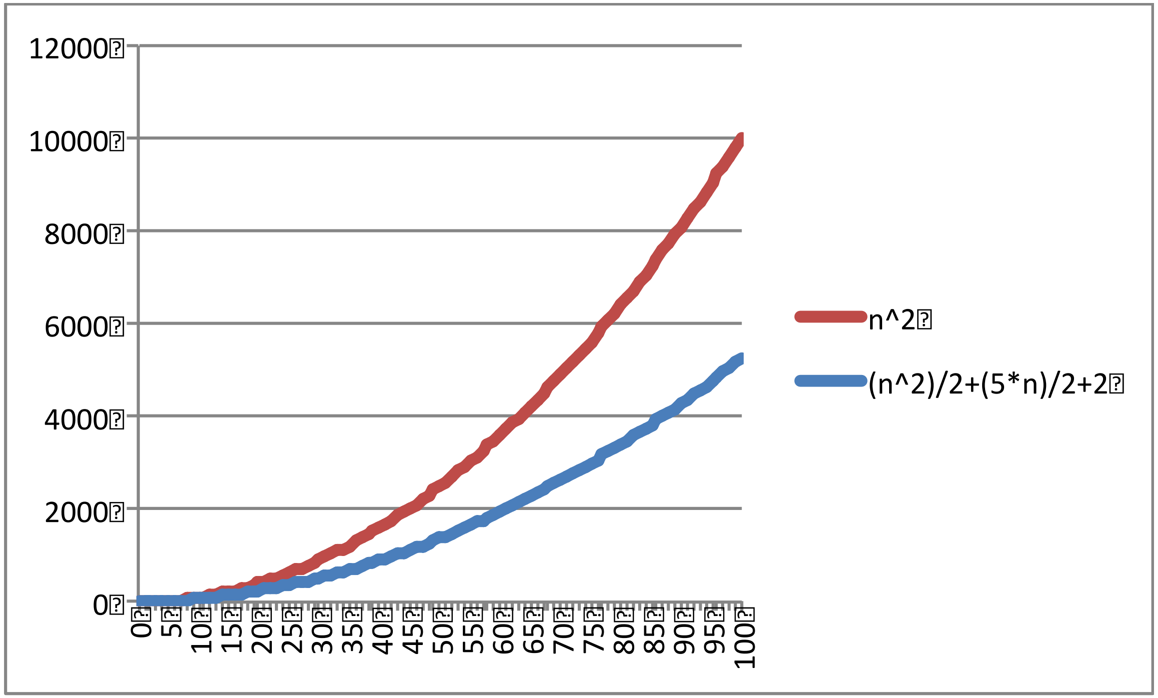
- To find the smallest, visit n elements + 2 operations for the swap
- To find the next smallest, visit (n-1) elements + 2 operations for the swap
- The last term is 2 elements visited to find the smallest + 2 operations for the swap

The number of operations:

- $(n + 2) + [(n-1) + 2] + [(n-2) + 2] + \ldots + (1+ 2) + 2$
- This can be simplified to
  - $n^2/2 + 5n/2 + 2$
- $5n/2 + 2$ is small compared to $n^2/2$ - so let's ignore it
- Also ignore the 1/2 - use the *simplest* expression of the class
- The number of visits is of the order n^2

The Big-Oh notation

- The number of visits is of the order n^2
- Using Big-Oh notation:
  - The number of visits is $O(n^2)$



Search Algorithms

- Searching Algorithms: check for an element from any data structure where it is stored
- Classed into two categories:
  - Sequential Search e.g. linear search
    - The list is traversed sequentially, and every element is checked
    - The list does NOT need to be sorted
  - Interval Search e.g. binary search
    - A 'divide and conquer' algorithm
    - The list MUST be sorted

Example: Binary Search

- Task: search for a key in a **sorted** list (e.g. 21)
- First check the middle list element

1	5	6	12	21	23	26	42	46	51	58

- If the key matches the middle element, we are done!
- If the key is less than the middle element, the key must be in the first half (otherwise in second half)

$21 < 23$

1	5	6	12	21	23	26	42	46	51	58

Example: Binary Search

- Repeat process

		21 > 6			21 < 23					
1	5	6	12	21	23	26	42	46	51	58

- Until you find the key (e.g. 21) or you know that the key is not in the list

		21 > 6	21 > 12		21 < 23					
1	5	6	12	21	23	26	42	46	51	58

Binary Search VS Linear Search

- Binary search is an $O(\log_2(n))$ algorithm
  - n elements → $n/2$ elements → $n/4$ elements → .... → 1 element
- Linear search algorithm of order $O(n)$
- Which algorithm is faster?
  - Binary search algorithm is much faster, *But* it only works on sorted data....
  - Binary search examples?
    - Spell checkers, phone books, dictionaries…

Binary search VS Linear search

- We have an unsorted Array with a million elements, and we would like to find a particular element…
- Question: Would it be faster to use Binary search (with Selection sort) *or* Linear search?
  - Linear search: $O(n)$
  - Binary search + Selection sort: $O(\log n) + O(n^2) = O(n^2)$
  - So linear search is faster!?

Recap

- It's important to define the run time of an algorithm without experimental studies!
  - Because of factors including processor speed, disk speed, compiler etc…
- Algorithmic complexity tells us how slow or fast an algorithm performs
- $T(n)$ will be used to denote the time an algorithm takes to execute – time versus input size n
- We measure $T(n)$ by counting the number of primitive operations

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in Big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size, $\mathrm{T}(n)$
  - We express this function with Big-Oh notation

The Big-Oh Notation

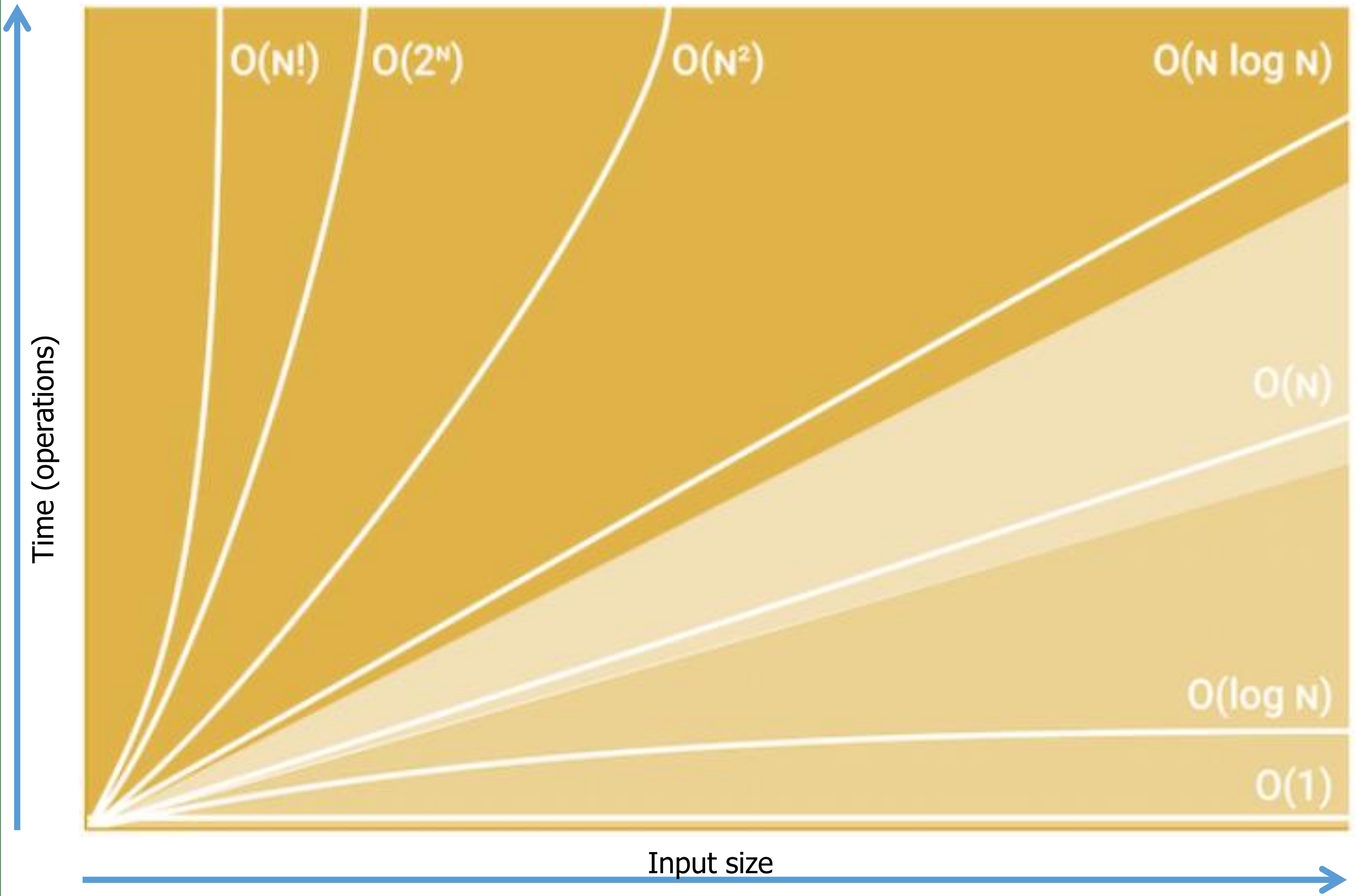
- Big-Oh Notation defines an upper bound of an algorithm (worst-case)
- The runtime in terms of how quickly it grows relative to the input – as the input gets larger
- E.g. for Insertion Sort
  - It takes **linear time** at best case
  - It takes **quadratic time** at worst case
  - We take worst-case: the time complexity of Insertion Sort is O(n^2)
  - This also covers linear time…

The Big-Oh Runtime Analysis

- **Step 1**: Find out the input and what *n* represents
- **Step 2**: Calculate the primitive operations of the algorithm in terms of *n*
- **Step 3**: Drop the lower-order terms
- **Step 4**: Remove all constant factors

Asymptotic Algorithm Analysis

- Example:
  - The algorithm `ArrayMax` executes at most $T(n) = 8n - 5$ primitive operations
  - We say that algorithm `ArrayMax` "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped, we can disregard them when counting primitive operations
  - We do not have to be 100% accurate as long as we get the powers of n correct
  - I.e. we do not miscount n^3 as n^2 etc…
  - Confusing $7n$ for $5n$ will not matter…


O(N!)
O(2^N)
O(N^2)
O(N log N)
O(N)
O(log N)
O(1)
Time (operations)
Input size

Rank From Fast to Slow…

Before…

- $\mathrm{T}(n) = n^4$
- $\mathrm{T}(n) = n \log n$
- $\mathrm{T}(n) = n^2$
- $\mathrm{T}(n) = n^2 \log n$
- $\mathrm{T}(n) = n$
- $\mathrm{T}(n) = 2^n$
- $\mathrm{T}(n) = \log n$
- $\mathrm{T}(n) = n + 2n$

After…

1. $\mathrm{T}(n) = \log n \equiv \mathrm{O}(\log n)$
2. $\mathrm{T}(n) = n \equiv \mathrm{O}(n)$
2. $\mathrm{T}(n) = n + 2n \equiv \mathrm{O}(n)$
4. $\mathrm{T}(n) = n \log n \equiv \mathrm{O}(n \log n)$
5. $\mathrm{T}(n) = n^2 \equiv \mathrm{O}(n^2)$
6. $\mathrm{T}(n) = n^2 \log n \equiv \mathrm{O}(n^2 \log n)$
7. $\mathrm{T}(n) = n^4 \equiv \mathrm{O}(n^4)$
8. $\mathrm{T}(n) = 2^n \equiv \mathrm{O}(2^n)$

But Can We Do Better?

- There are algorithms that have a computational complexity of $O(2^n)$ (this is very poor: $2^{50} \approx 10^{15}$)

- But ......

- How do we know there **isn't** a better algorithm which only takes polynomial time anyway?

Polynomial Time

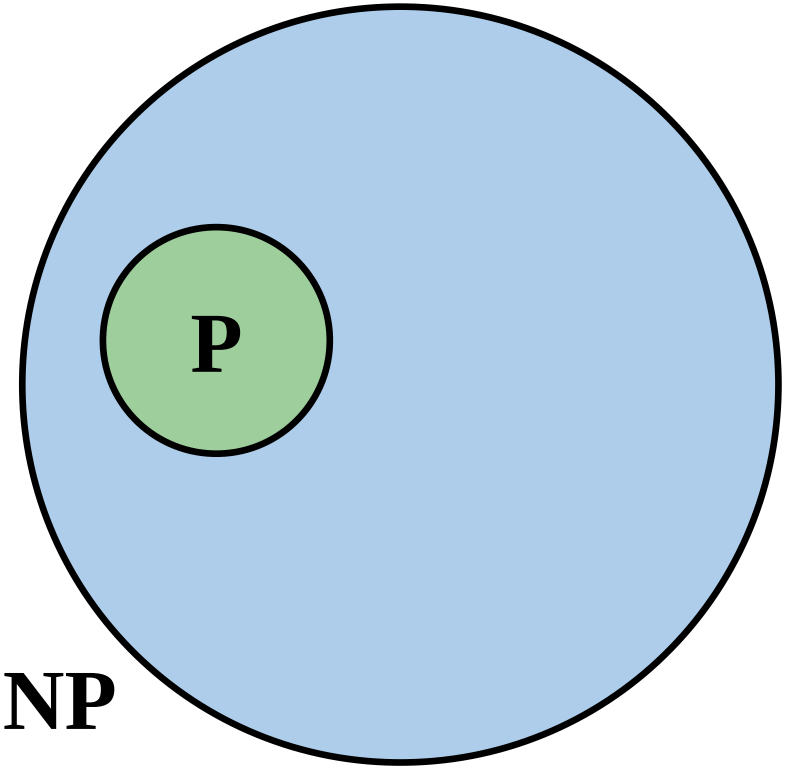
An algorithm is solvable in **polynomial time** if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer k, n being the complexity of the input.

The Classes of Algorithms – Part 1

- Algorithms are divided up into a number of classes (classifications):
  - Computable
    - Problems that **can** be solved using a computer
    - e.g. $f(x) = x + 1$
  - Non-Computable
    - Problems that **cannot** can be solved using a computer
    - e.g. Halting Problem
    - Can we find a program that can predict whether any other program and its input will halt or run forever…

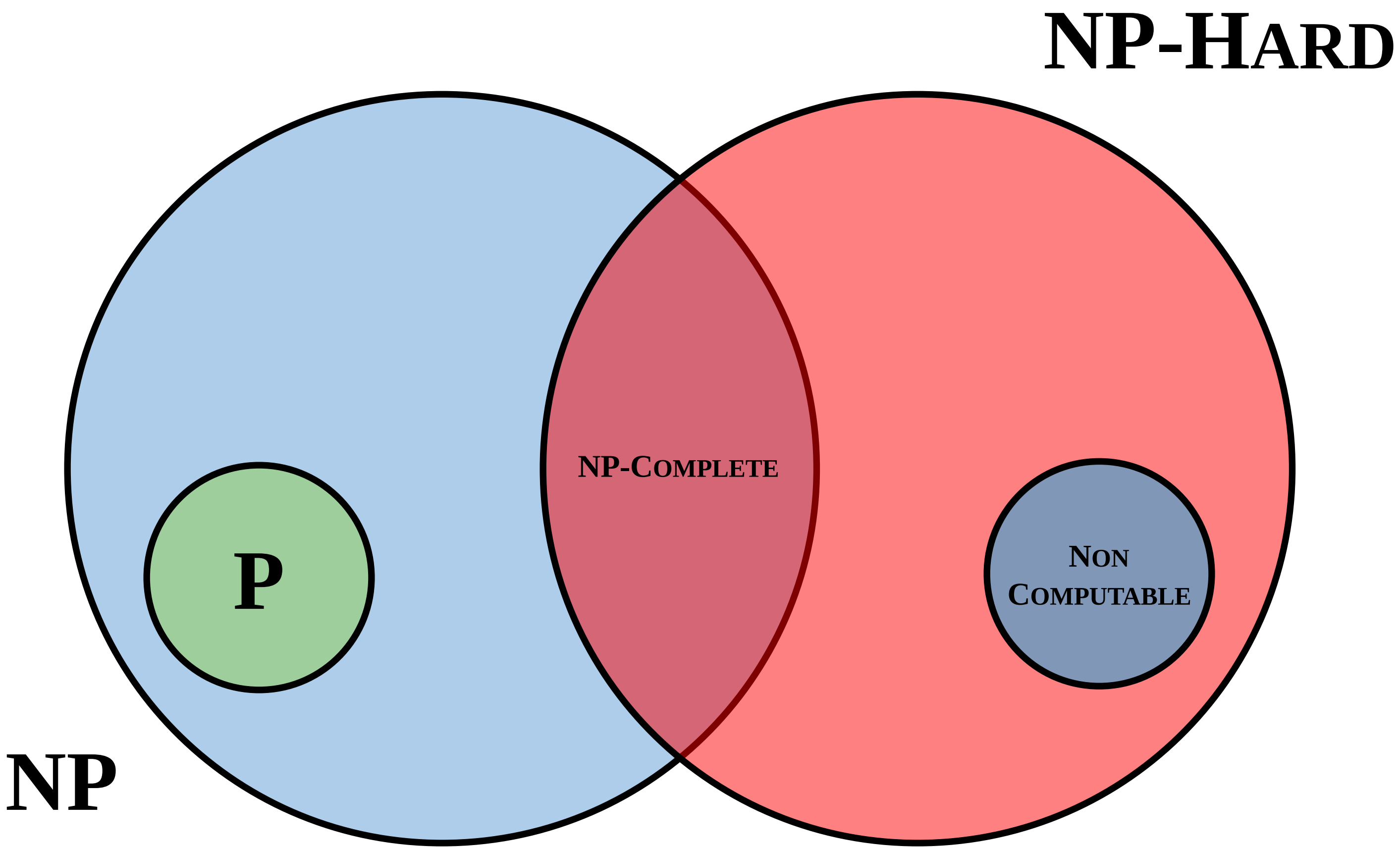
The Classes of Algorithms – Part 2

- Algorithms are divided up into a number of classes (classifications):
  - P Problems
    - Solved in a reasonable amount of time (polynomial time)
    - E.g. multiplication and sorting
  - NP Problems
    - Difficult to solve in a reasonable amount of time but easy to *verify* the solution
    - Non-deterministic algorithms
    - Problems involving decision making
    - Important class of problems e.g. job scheduling, circuit design, vehicle routing etc..



P is a subset of NP

The Classes of Algorithms – Part 3

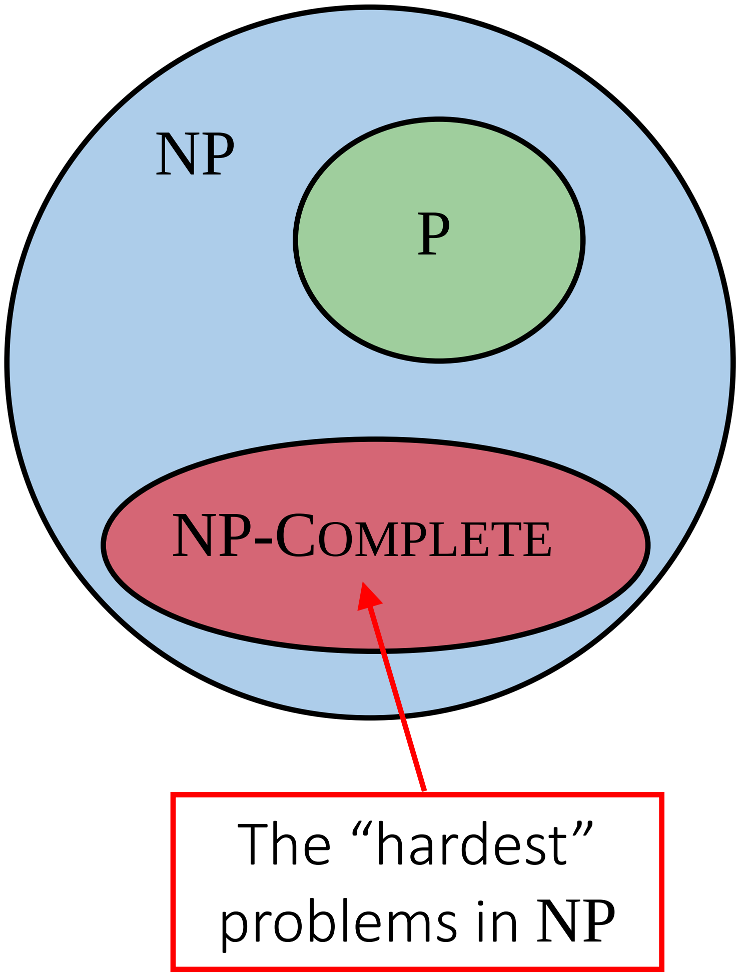


The Classes of Algorithms – Part 4

- NP-HARD Problems
  - Very, very difficult to solve problems, which cannot be done in a reasonable amount of time
  - The exhaustive search is not polynomial
  - They are very difficult to verify in polynomial time

- NP-COMPLETE Problems
  - The hardest problems in NP set
  - Complexities greater than polynomial time
  - The exhaustive search is not polynomial
  - Verifiable in polynomial time
  - No polynomial-time algorithm is discovered for any NP-complete problem
  - Nobody was able to prove that no polynomial-time algorithm exist for any of the problems

The Hardest NP Problems



If a polynomial time algorithm is found for **any** problem in NP-COMPLETE then **every** problem in NP can be solved in polynomial time

Examples of NP-Complete Problems

- The travelling salesperson problem
- Finding the shortest common superstring
- Checking whether two finite automata accept the same language
- Given three positive integers a, b, and c, do there exist positive integers (x and y) such that $ax^2 + by^2 = c$

How to be Very, Very (in)Famous!

P Vs NP

"**If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem?** This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution." .....

Clay Mathematics Institute
https://www.claymath.org/millennium-problems/p-vs-np-problem

Consequences of P=NP

- If this was found to be true, the news would make **WORLD HEADLINES!**
- Modern society as we know it would be completely and utterly **DOOMED!**
- All passwords could be cracked in polynomial time (very, very fast)
- You would not be able to secure any computer or device on the internet, wireless or mobile networks.....
- Algorithms that currently take years might take minutes!
- Proof [that NP≠P] was put forward in 2010, but has since been refuted…

This Week's Laboratory

- This laboratory is one of the worksheets you may be assessed in **Task #1 and/or #2**
- You will be implementing and comparing two algorithms
  - Computing T(n)
  - Computing O(n)
  - Running some experiments

Next Lecture

- We will look at data structures and their applications