# Efficiency;
# Classifying problems;

Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.

## Searching & Sorting Algorithms

- We have learned that if ***n*** is the size of our list of data:
  - The running time of **sequential search** is proportional to **n**
  - The running time of **binary search** is proportional to **$\log_2 n$**
  - The running time of **selection sort** is proportional to **$n^2$**
  - The running time of **quick sort** is proportional to **$n \log_2 n$**

- For example if binary search takes 1 seconds to search for an element in a list with 500,000 elements, then
  - a list with 1,000,000 elements → roughly ___________
  - a list with 10,000,000 elements→ roughly ___________
  - a list with 100,000,000 elements→ roughly ___________

## Searching & Sorting Algorithms

- We have learned that if ***n*** is the size of our list of data:
  - The running time of **sequential search** is proportional to **$n$**
  - The running time of **binary search** is proportional to **$\log_2 n$**
  - The running time of **selection sort** is proportional to **$n^2$**
  - The running time of **quick sort** is proportional to **$n \log_2 n$**

- For example if binary search takes 1 seconds to search for an element in a list with 500,000 elements, then
  - a list with 1,000,000 elements → roughly 2 seconds
  - a list with 10,000,000 elements→ roughly ___________
  - a list with 100,000,000 elements→ roughly ___________

1,000,000 = **2** * 500,000

**$\log_2 n \sim 1$ sec -> $\log_2 2n = \log_2 2 + \log_2 n \sim 1 + 1 = 2$ sec**

## Searching & Sorting Algorithms

- We have learned that if ***n*** is the size of our list of data:
  - The running time of **sequential search** is proportional to **$n$**
  - The running time of **binary search** is proportional to **$\log_2 n$**
  - The running time of **selection sort** is proportional to **$n^2$**
  - The running time of **quick sort** is proportional to **$n \log_2 n$**

- For example if binary search takes 1 seconds to search for an element in a list with 500,000 elements, then
  - a list with 1,000,000 elements → roughly 2 seconds
  - a list with 10,000,000 elements→ roughly 5.32 seconds
  - a list with 100,000,000 elements→ roughly ___________

10,000,000 = **20** * 500,000

**$\log_2 n \sim 1$ sec -> $\log_2 20n = \log_2 20 + \log_2 n \sim 4.32 + 1 = 5.32$ sec**

## Searching & Sorting Algorithms

- We have learned that if ***n*** is the size of our list of data:
  - The running time of **sequential search** is proportional to **n**
  - The running time of **binary search** is proportional to **$\log_2 n$**
  - The running time of **selection sort** is proportional to **$n^2$**
  - The running time of **quick sort** is proportional to **$n \log_2 n$**

- For example if binary search takes 1 seconds to search for an element in a list with 500,000 elements, then
  - a list with 1,000,000 elements → roughly 2 seconds
  - a list with 10,000,000 elements→ roughly 5.32 seconds
  - a list with 100,000,000 elements→ roughly 8.64 seconds

100,000,000 = **200** * 500,000

**$\log_2 n \sim 1$ sec -> $\log_2 200n = \log_2 200 + \log_2 n \sim 7.64 + 1 = 8.64$ sec**

## Algorithm Analysis

- Computer scientists characterize an algorithm's efficiency by specifying its *growth function*.
  - the function to which its running time is roughly proportional

- We've seen several different growth functions:

```
log₂n      # binary search
n          # sequential/linear search
n log₂n    # quicksort
n²         # selection sort
```

- Others include:

```
cⁿ         # exponential growth
n!         # factorial growth
```

- CS 112 develops a mathematical formalism for these functions.

## How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n?
  - assume the growth function gives the exact # of operations
  - assume that each operation requires 1 $\mu$sec (1 x $10^{-6}$ sec)

| growth function | problem size (n) | | | | | |
|---|---|---|---|---|---|---|
| | 10 | 20 | 30 | 40 | 50 | 60 |
| n | | | | | | |
| $n^2$ | | | | | | |
| $n^5$ | | | | | | |
| $2^n$ | | | | | | |

## How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n?
  - assume the growth function gives the exact # of operations
  - assume that each operation requires 1 $\mu$sec (1 x $10^{-6}$ sec)

| growth function | problem size (n) | | | | | |
|---|---|---|---|---|---|---|
| | 10 | 20 | 30 | 40 | 50 | 60 |
| n | .00001 s | | | | | |
| $n^2$ | .0001 s | | | | | |
| $n^5$ | **.1 s** | | | | | |
| $2^n$ | **.001 s** | | | | | |

Note for small data sets we **may** get a misleading result of the algorithm's efficiency!

## How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n?
  - assume the growth function gives the exact # of operations
  - assume that each operation requires 1 $\mu$sec (1 x $10^{-6}$ sec)

| growth | problem size (n) | | | | | |
|---|---|---|---|---|---|---|
| function | 10 | 20 | 30 | 40 | 50 | 60 |
| $n$ | .00001 s | .00002 s | | | | |
| $n^2$ | .0001 s | .0004 s | | | | |
| $n^5$ | .1 s | **3.2 s** | | | | |
| $2^n$ | .001 s | **1.0 s** | | | | |

Note for small data sets we **may** get a misleading result of the algorithm's efficiency!

## How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n?
  - assume the growth function gives the exact # of operations
  - assume that each operation requires 1 $\mu$sec (1 x $10^{-6}$ sec)

| growth | problem size (n) | | | | | |
|---|---|---|---|---|---|---|
| function | 10 | 20 | 30 | 40 | 50 | 60 |
| $n$ | .00001 s | .00002 s | .00003 s | | | |
| $n^2$ | .0001 s | .0004 s | .0009 s | | | |
| $n^5$ | .1 s | 3.2 s | **24.3 s** | | | |
| $2^n$ | .001 s | 1.0 s | **17.9 min** | | | |

But eventually….

# How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n?
  - assume the growth function gives the exact # of operations
  - assume that each operation requires 1 μsec (1 x $10^{-6}$ sec)

| growth | problem size (n) | | | | | |
|---|---|---|---|---|---|---|
| function | 10 | 20 | 30 | 40 | 50 | 60 |
| n | .00001 s | .00002 s | .00003 s | .00004 s | | |
| $n^2$ | .0001 s | .0004 s | .0009 s | .0016 s | | |
| $n^5$ | .1 s | 3.2 s | 24.3 s | **1.7 min** | | |
| $2^n$ | .001 s | 1.0 s | 17.9 min | **12.7 days** | | |

The inefficiency of the algorithm becomes apparent!

# How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n?
  - assume the growth function gives the exact # of operations
  - assume that each operation requires 1 μsec (1 x $10^{-6}$ sec)

| growth | problem size (n) | | | | | |
|---|---|---|---|---|---|---|
| function | 10 | 20 | 30 | 40 | 50 | 60 |
| n | .00001 s | .00002 s | .00003 s | .00004 s | .00005 s | |
| $n^2$ | .0001 s | .0004 s | .0009 s | .0016 s | .0025 s | |
| $n^5$ | .1 s | 3.2 s | 24.3 s | 1.7 min | **5.2 min** | |
| $2^n$ | .001 s | 1.0 s | 17.9 min | 12.7 days | **35.7 yrs** | |

..**painfully** apparent!!

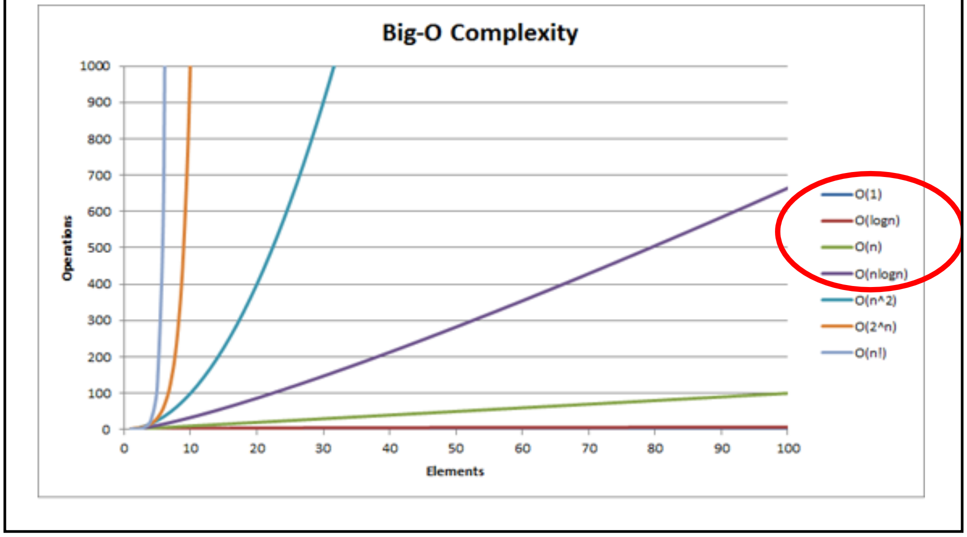
## How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n?
  - assume the growth function gives the exact # of operations
  - assume that each operation requires 1 μsec (1 x $10^{-6}$ sec)

| growth function | problem size (n) | | | | | |
|---|---|---|---|---|---|---|
| | 10 | 20 | 30 | 40 | 50 | 60 |
| n | .00001 s | .00002 s | .00003 s | .00004 s | .00005 s | .00006 s |
| $n^2$ | .0001 s | .0004 s | .0009 s | .0016 s | .0025 s | .0036 s |
| $n^5$ | .1 s | 3.2 s | 24.3 s | 1.7 min | 5.2 min | 13.0 min |
| $2^n$ | .001 s | 1.0 s | 17.9 min | 12.7 days | **35.7 yrs** | **36,600 yrs** |

..**excruciatingly** apparent!!

## Algorithm Analysis



We use selection sort to sort a list of length 10,000,
and it takes 2 seconds to complete the task.

If we now use selection sort to sort a list of length 80,000, roughly how long should it take?

A. 2 seconds

B. 128 seconds

C. 16 seconds

D. 256 second

E. it is impossible to predict

F. none of these

We use selection sort to sort a list of length 10,000,
and it takes 2 seconds to complete the task.

If we now use selection sort to sort a list of length 80,000, roughly how long should it take?

A. 2 seconds

B. 128 seconds

C. 16 seconds

D. 256 second

E. it is impossible to predict

F. none of these

- 10,000 → 80,000
- the list is 8x longer.
- The growth function of selection sort is proportional to $n^2$
- thus, running time is ~64x longer.

80,000 = **8** * 10,000

**$(n)^2 \sim 2$ sec -> $(8n)^2 = 64n^2 \sim 64*2 = 128$ sec**

We use **the best algorithms** to search and find an element in the following list of length 5,000,000, and it takes 1 second to complete the task.

lst = [2, 10000, -759, 450, 3, 10, …]

If we now use **the best algorithms** to search and find another element in the following list of length 20,000,000, roughly how long should it take?

lst = [20, 1010, 759, -401, 9, -10, …]

A. 4 seconds

B. 3 seconds

C. 128 seconds

D. 5 seconds

E. 64 seconds

D. it is impossible to predict

**Available Algorithms:**

- sequential search ~ $n$
- binary search ~ $\log_2 n$
- selection sort ~ $n^2$
- quick sort ~ $n \log_2 n$

$\log_2 4n = \log_2 4 + \log_2 n = 2 + \log_2 n$

We use **the best algorithms** to search and find an element in the following list of length 5,000,000, and it takes 1 second to complete the task.

lst = [2, 10000, -759, 450, 3, 10, …]

If we now use **the best algorithms** to search and find another element in the following list of length 20,000,000, roughly how long should it take?

lst = [20, 1010, 759, -401, 9, -10, …]

**Available Algorithms:**

- sequential search ~ $n$
- binary search ~ $\log_2 n$
- selection sort ~ $n^2$
- quick sort ~ $n \log_2 n$

$\log_2 4n = \log_2 4 + \log_2 n = 2 + \log_2 n$

As list is not sorted, we cannot use binary search directly. So, we have two options:

1. Using **sequential search**.
2. Using **quick sort** to sort it and then **binary search**

We use **the best algorithms** to search and find an element in the following list of length 5,000,000, and it takes 1 second to complete the task.

lst = [2, 10000, -759, 450, 3, 10, …]

If we now use **the best algorithms** to search and find another element in the following list of length 20,000,000, roughly how long should it take?

lst = [20, 1010, 759, -401, 9, -10, …]

**Available Algorithms:**

- sequential search ~ n
- binary search ~ $\log_2 n$
- selection sort ~ $n^2$
- quick sort ~ $n \log_2 n$

**$\log_2 4n = \log_2 4 + \log_2 n = 2 + \log_2 n$**

1. Using sequential search

n ~ 1 sec

20,000,000 = 4 * 5,000,000

n ~ 1 sec -> **4n ~ 4 sec**

It takes 4 seconds to find the element using sequential search.

We use **the best algorithms** to search and find an element in the following list of length 5,000,000, and it takes 1 second to complete the task.

lst = [2, 10000, -759, 450, 3, 10, …]

If we now use **the best algorithms** to search and find another element in the following list of length 20,000,000, roughly how long should it take?

lst = [20, 1010, 759, -401, 9, -10, …]

**Available Algorithms:**

- sequential search ~ n
- binary search ~ $\log_2 n$
- selection sort ~ $n^2$
- quick sort ~ $n \log_2 n$

**$\log_2 4n = \log_2 4 + \log_2 n = 2 + \log_2 n$**

2. Using **quick sort** to sort it and then **binary search**

$(n \log_2 n + \log_2 n)$ ~ 1 sec

20,000,000 = **4** * 5,000,000

$(4n \log_2 4n + \log_2 4n) = [4n (2 + \log_2 n) + 2 + \log_2 n]$

$= 2 + 8n + (n \log_2 n + \log_2 n) + 3n \log_2 n$

$= 2 + 8n + 1 + 3n \log_2 n =$ **$3 + 8n + 3n \log_2 n$**

We use **the best algorithms** to search and find an element in the following list of length 5,000,000, and it takes 1 second to complete the task.

lst = [2, 10000, -759, 450, 3, 10, …]

If we now use **the best algorithms** to search and find another element in the following list of length 20,000,000, roughly how long should it take?

lst = [20, 1010, 759, -401, 9, -10, …]

**Available Algorithms:**

- sequential search ~ n
- binary search ~ $\log_2 n$
- selection sort ~ $n^2$
- quick sort ~ $n \log_2 n$

$\log_2 4n = \log_2 4 + \log_2 n = 2 + \log_2 n$

2. Using **quick sort** to sort it and then **binary search**

**$3 + 8n + 3n \log_2 n$ ~ ??? Seconds**

- It is not possible to compute!!!
- The only available relation is $(n \log_2 n + \log_2 n)$ ~ 1 sec

We use **the best algorithms** to search and find an element in the following list of length 5,000, and it takes 1 second to complete the task.

lst = [2, 10000, -759, 450, 3, 10, …]

If we now use **the best algorithms** to search and find another element in the following list of length 20,000, roughly how long should it take?

lst = [20, 1010, 759, -401, 9, -10, …]

A. 4 seconds

B. 3 seconds

C. 128 seconds

D. 5 seconds

E. 64 seconds

F. it is impossible to predict

- Sequential ~ n ~ 4 seconds
- Quicksort + binary ~ $n \log_2 n + \log_2 n$ ~ ??? seconds
- Which one is the answer A or F?
- Comparing Big O complexity of **n** & **$n \log_2 n$** shows that $O(n) < O(n \log_2 n)$
- Thus, $O(\mathbf{n}) < O(\mathbf{n \log_2 n + \log_2 n})$

We use **the best algorithms** to search and find an element in the following list of length 5,000, and it takes 1 second to complete the task.

lst = [2, 10000, -759, 450, 3, 10, …]

If we now use **the best algorithms** to search and find another element in the following list of length 20,000, roughly how long should it take?

lst = [20, 1010, 759, -401, 9, -10, …]

**A. 4 seconds**

B. 3 seconds

C. 128 seconds

D. 5 seconds

E. 64 seconds

F. it is impossible to predict

- Sequential ~ n ~ 4 seconds
- Quicksort + binary ~ n $\log_2 n$ + $\log_2 n$ ~ ??? seconds
- Which one is the answer A or F?
- Comparing Big O complexity of **n** & **n $\log_2$n** shows that O(n) < O(n $\log_2$n)
- Thus, O(**n**) < O(**n $\log_2$n + $\log_2$n** )
- Hence, using sequential search is the best algorithm.
- And, the answer is choice A, i.e. 4 seconds.

## Classifying Problems

- **"Easy" problems**: can be solved using an algorithm with a growth function that is a *polynomial* of the problem size, n.

  $\log_2 n$<br>
  n<br>
  n $\log_2 n$<br>
  $n^2$<br>
  $n^3$<br>
  etc.

  - we can solve large problem instances in a reasonable amount of time

## Classifying Problems

- **"Easy" problems**: can be solved using an algorithm with a growth function that is a *polynomial* of the problem size, n.

  $\log_2 n$
  $n$
  $n \log_2 n$
  $\mathbf{n^2}$
  $\mathbf{n^3}$

  But even an easy problem can be solved inefficiently!

  - we can solve large problem instances in a *reasonable* amount of time

## Classifying Problems

- **"Easy" problems**: can be solved using an algorithm with a growth function that is a *polynomial* of the problem size, n.

  $\log_2 n$
  $n$
  $n \log_2 n$
  $n^2$
  $n^3$
  etc.

  - we can solve large problem instances in a reasonable amount of time

- **"Hard" problems**: their only known solution algorithm has an *exponential* or *factorial* growth function.

  $c^n$
  $n!$

  - they can only be solved exactly for small values of n

## Example of a "Hard" Problem: Map Labeling

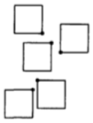
- ***Given:*** the coordinates of a set of *point features* on a map
  - cities, towns, landmarks, etc.
- ***Task:*** determine positions for the point features' *labels*

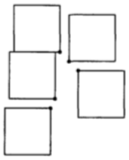
Lawrence
Lowell
95
Gloucester
Gardner
Fitchburg
Chelmsford
Billerica
Peabody
Leominster
2
Wakefield
Lynn
190
Medford
Revere
Marlborough
Waltham
Boston
Worcester
Framingham
Natick
Quincy
90
495
Randolph
146
84
Woonsocket
Franklin
95
Brockton

## Example of a "Hard" Problem: Map Labeling

- ***Given:*** the coordinates of a set of *point features* on a map
  - cities, towns, landmarks, etc.
- ***Task:*** determine positions for the point features' *labels*

Lawrence
Lowell
95
Gloucester
Gardner
Fitchburg
Chelmsford
Billerica
Peabody
Leominster
2
Wakefield
Lynn
190
Medford
Revere
Marlborough
Waltham
Boston
Worcester
Framingham
Natick
Quincy
90
495
Randolph
146
84
Woonsocket
Franklin
95
Brockton


Valid Labeling


Optimal Labeling

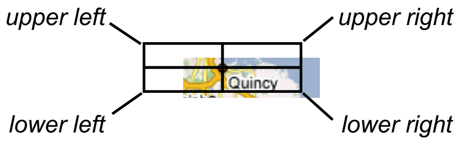
- Because the point features tend to be closely packed, we may get overlapping labels.
- ***Goal:*** find the labeling with the fewest overlaps

## Map Labeling (cont.)

- One possible solution algorithm: ***brute force!***
  - try all possible labeling
- How long would this take?

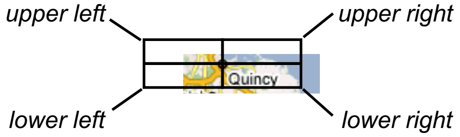
## Map Labeling (cont.)

- One possible solution algorithm: ***brute force!***
  - try all possible labelings
- How long would this take?
- Assume there are only 4 possible positions for each point's label:



## Map Labeling (cont.)

- One possible solution algorithm: ***brute force!***
  - try all possible labelings
- How long would this take?
- Assume there are only 4 possible positions for each point's label:

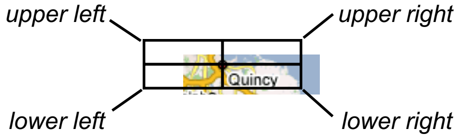


  - for n point features, there are $4^n$ possible labelings
  - thus, running time will "grow proportionally" to $4^n$

**exponential time!**

## Map Labeling (cont.)

- One possible solution algorithm: ***brute force!***
  - try all possible labelings
- How long would this take?
- Assume there are only 4 possible positions for each point's label:



  - for n point features, there are $4^n$ possible labelings
  - thus, running time will "grow proportionally" to $4^n$

**exponential time!**

  - example: 30 points → $4^{30}$ possible labelings
    - if it took 1 μsec to consider each labeling,
      **it would take over 36,000 years to consider them all!**

## Can Optimal Map Labeling Be Done Efficiently?

- In theory, a problem like map labeling could have a yet-to-be discovered efficient solution algorithm.
- How likely is this?

## Can Optimal Map Labeling Be Done Efficiently?

- In theory, a problem like map labeling could have a yet-to-be discovered efficient solution algorithm.
- How likely is this?
- **Not very!**
- If you could solve map labeling efficiently, you could also solve many other hard problems!
  - the *NP-hard* problems
  - another example: the traveling salesperson problem in the reading

## Dealing With "Hard" Problems

- When faced with a hard problem, we resort to approaches that quickly find solutions that are "good enough".
- Such approaches are referred to as *heuristic* approaches.
  - heuristic = rule of thumb
  - no guarantee of getting the optimal solution
  - typically get a good solution

## Classifying Problems

- **"Easy" problems**: can be solved using an algorithm with a growth function that is a *polynomial* of the problem size, n.
  - we can solve large problem instances in a reasonable amount of time
- **"Hard" problems**: their only known solution algorithm has an *exponential* or *factorial* growth function.
  - they can only be solved exactly for small values of n

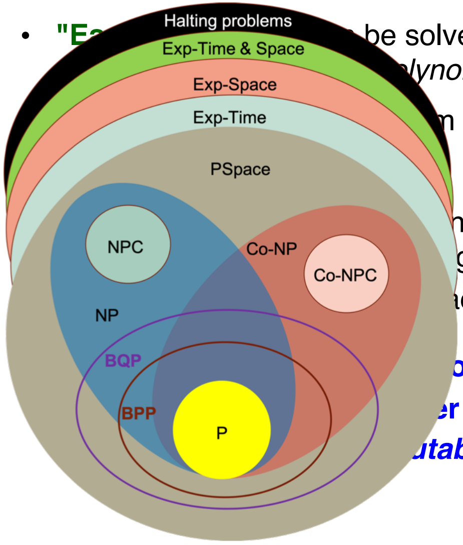
## Classifying Problems

- **"Easy" problems**: can be solved using an algorithm with a growth function that is a *polynomial* of the problem size, n.
  - we can solve large problem instances in a reasonable amount of time
- **"Hard" problems**: their only known solution algorithm has an *exponential* or *factorial* growth function.
  - they can only be solved exactly for small values of n
- **A third class: *Impossible* problems!**
  - **can't be solved, no matter how long you wait!**

## Classifying Problems

- **"Easy" problems**: can be solved using an algorithm with a growth function that is a *polynomial* of the problem size, n.
  - we can solve large problem instances in a reasonable amount of time
- **"Hard" problems**: their only known solution algorithm has an *exponential* or *factorial* growth function.
  - they can only be solved exactly for small values of n
- **A third class: *Impossible* problems!**
  - **can't be solved, no matter how long you wait!**
  - **referred to as *uncomputable* problems**

## Classifying Problems



- **"Ea** be solved using an algorithm with a *olynomial* of the problem size, n.

n instances in a

nown solution algorithm growth function.

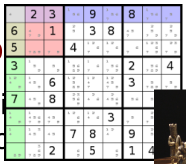
ctly for small values of n

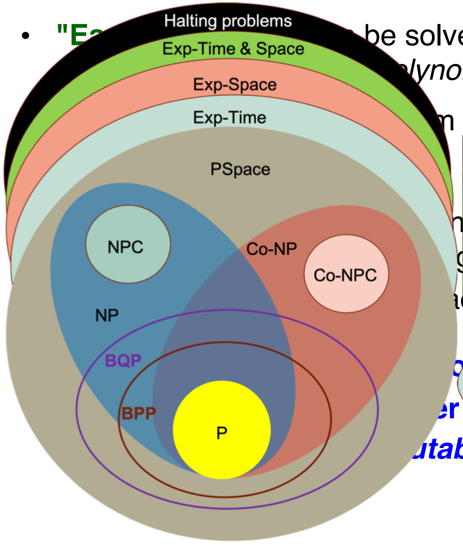
**oblems!**

**er how long you wait!**

***utable* problems**

## Classifying Pro





- **"Ea** be solved usi *olynomial* c

n instances in a

n

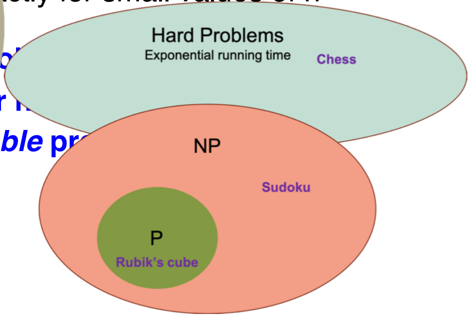
g

ctly for small values of n

**o**

**er**

***utable* pr**



# Classifying Problems

- **"Easy" problems**: can be solved using an algorithm with a growth function that is a *polynomial* of the problem size, n.
  - we can solve large problem instances in a reasonable amount of time
- **"Hard" problems**: their only known solution algorithm has an *exponential* or *factorial* growth function.
  - they can only be solved exactly for small values of n
- **A third class: *Impossible* problems!**
  - **can't be solved, no matter how long you wait!**
  - **referred to as *uncomputable* problems**