CS151 Intro to Data Structures

Abstract Data Types

Interfaces

Algorithm Analysis

Announcements

- HW02 due Tuesday October 3rd
- Lab checkoff, deadline is when corresponding HW is due

HW02 Recommendations

Each file should still be read only once

- Create only one new `Scanner` per file
- No resetting the `Scanner`s

Totals are computed only once and stored, not over and over again

All relevant stats for one name is stored in a single `Name` object

Print yearly stats in the order of the files given

HW02 Recommendations

Parse the year from the filename

Don’t store the percentages

- Computing and storing totals cost no additional loop
- Computing and storing percentages do

Do you feel like you are writing the same code twice a lot?

- Try making it into a method and call that twice, instead of repeating the lines

HW02 Recommendations

Command-line Arguments:

Array passed into `Main`: `String[] args`

```
for(int i=0; i<args.length; i++) {
  System.out.println(args[i]);
}
```

Catch the exceptions!

Quit the program

```
System.exit(0);
```

Outline

- Interfaces
- Abstract Data Types
- Algorithm Analysis

Data Structures so far

Arrays

ArrayLists (Expandable Array is a simplified version)

LinkedList

So, what is Data Structures?

Organizing data to efficiently access, manipulate, and manage that data

Data Structures – key points

Organization:

organize data to make specific operations easier:

- Arrays & LinkedLists – data is organized linearly
- Other data structures store data hierarchically

Efficiency:

optimize common operations: searching, insertion, deletion, and retrieval.

- Choosing the right data structure can significantly impact the efficiency of an algorithm.

Abstraction:

hide the underlying implementation details from the user

Choosing which to use:

different types of data have their own strengths and weaknesses.

- Understanding these helps choose when to use which data structure

Student, Faculty, TA, Professor

- Student: GPA, courses, major
- Faculty: non-student stuff
- TA:
  - is a student
  - teaching: office hours, grading
- Professor:
  - is a faculty
  - teaching: office hours, grading

Design

TA inherits from Student

Professor inherits from Faculty

TA and Professor both Teach

- Hold office hours
- Grade
- Run labs
- Lecture (some places TAs lecture)

We need a way to specify that they both do Teaching

Teachable Interface

```
public interface Teachable {
   int grade(Homework homework);

   boolean teach(int lectureNumber);

   void runOfficeHours(Student[] students);

}
```

TA

```
public class TA extends Student implements Teachable {
  // constructors and getters not shown
  public int grade(Homework homework){
    return homework.getMaxPoints();
  }
  public boolean teach(int lectureNumber){return true;}

  public void runOfficeHours(Student[] students) {
    for (Student student : students) {
       student.learnsMaterial();
    }
  }
}
```

Interface

A collection of method signatures with no bodies

No modifier - implicitly `public`

No instance variables except for constants (`static final`)

No constructors and can not be instantiated

A class implementing an `interface` must implement all methods as specified

Interfaces

An interface is not a class!

- can not be instantiated
- incomplete specification

An interface is a type

- a class that implements an interface is a subtype of the interface

A class may implement several interfaces

Interfaces

Interfaces declare features (methods) but provides no implementation

Class that implement an interface MUST implement all specified methods

Method implementations must match signatures in interface exactly

A class is what an object is

An interface is what an object does

Object Comparison

Remember, all classes are subtypes of `Object`

Methods we inherit from `Object`:

```
String toString()
boolean equals(Object)
```

A custom class must define (override) its own `equals` because Java doesn't know how

`equals` tests object equality, but what about relative ordering?

`compareTo`

`compareTo`

`compareTo` **returns an** `int`**, not a** `Boolean`

Why?

because it needs to convey three outcomes:

- `-1` if smaller compared to the parameter
- `0` if equal
- `1` if larger compared to the parameter

`Comparable` interface

The `Comparable` interface is designed for objects that have an ordering

```
public interface Comparable<T> {
  int compareTo(T o);
}
```

Data Structures – key points

Organization:

organize data to make specific operations easier:

- Arrays & LinkedLists – data is organized linearly
- Other data structures store data hierarchically

Efficiency:

optimize common operations: searching, insertion, deletion, and retrieval.

- Choosing the right data structure can significantly impact the efficiency of an algorithm.

Abstraction:

hide the underlying implementation details from the user

Choosing which to use:

different types of data have their own strengths and weaknesses.

- Understanding these helps choose when to use which data structure

Abstract Data Types (ADTs)

ADT: abstraction of a data structure

ADT specifies (what but not how):

- data stored
- operations on data
- error conditions

The collective set of behaviors supported by an ADT is its public interface (API)

Why use ADT?

An ADT is a theoretical definition of a set of "behaviors”

It doesn’t specify how those behaviors are implemented

Given the same ADT, different underlying implementations are possible

Code that uses the ADT’s API does not have to change, or even be aware that the underlying implementation changes

Why ADT?

- An ADT is a theoretical definition of a set of "behaviors"
- It doesn’t specify how those behaviors are implemented
- Given the same ADT, different underlying implementations are possible
- Code that uses the ADT’s API does not have to change, or even be aware that the underlying implementation changes

Making Custom Exceptions

Often times we need to raise a custom exception

Extend `Exception` or `RuntimeException`

Subclass of `Exception` are checked exceptions – must be treated/caught

Subclass of `RuntimeException` are not checkable during compile time

```
throw new ExceptionConstrutor(msg);
```

Example

```
public class MyException extends Exception {
  public MyException (String errorMsg) {
    super(errorMsg);
  }
}
public class IllegalArgumentException extends
RuntimeException {
  public IllegalArgumentException (String errorMsg) {
    super(errorMsg);
  }
}
```

Data Structures – key points

Organization:

organize data to make specific operations easier:

- Arrays & LinkedLists – data is organized linearly
- Other data structures store data hierarchically

Efficiency:

optimize common operations: searching, insertion, deletion, and retrieval.

- Choosing the right data structure can significantly impact the efficiency of an algorithm.

Abstraction:

hide the underlying implementation details from the user

Choosing which to use:

different types of data have their own strengths and weaknesses.

- Understanding these helps choose when to use which data structure

Running Time

How long a program runs depends on

- efficiency of the algorithm/implementation
- size of input

The running time typically grows with input size

We focus on worse case analysis

- how long will it take in the worst case?

Space (Memory) Complexity

How much memory a program needs depends on

- efficiency of the algorithm/implementation
- size of input

The space requirements time typically grows with input size

We focus on worse case analysis

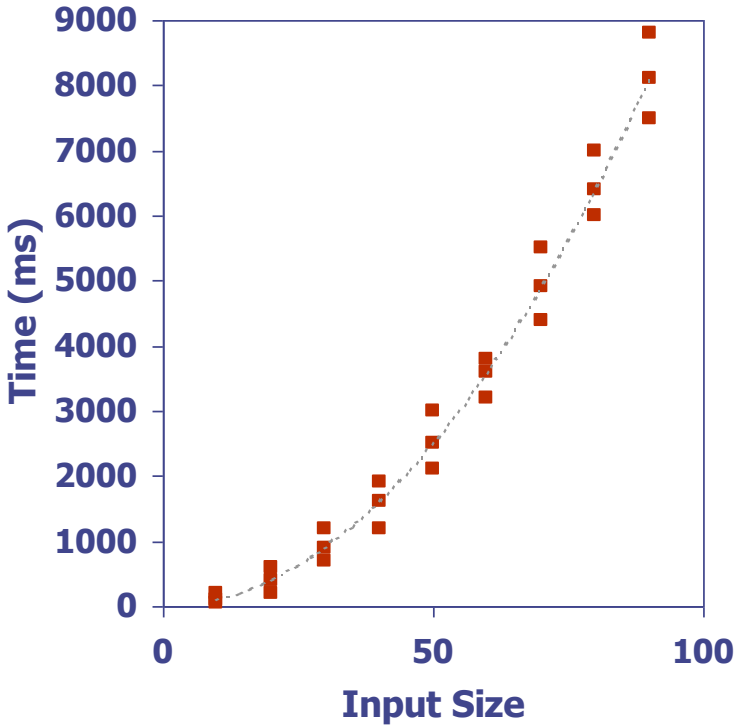
- how much space will it take in the worst case?

Experimental Study

Write a program implementing the algorithm

Run it with different input sizes and compositions

Record times and plot results




```
1  long startTime = System.currentTimeMillis( );      // record the starting time
2  /* (run the algorithm) */
3  long endTime = System.currentTimeMillis( );        // record the ending time
4  long elapsed = endTime − startTime;                // compute the elapsed time
```

Limitation of Experiments

Need to implement the algorithm

Need to generate inputs that represent all cases

Comparing two algorithms requires exact same hardware and software environments

Theoretical Analysis

Uses high-level description of algorithm

pseudo-code

Characterizes running time as a function of the input size, n

Takes into account all possible inputs

Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Primitive Operations

Basic computations

- Assigning variables, adding, multiplying, boolean operators

Not looped

Assumed to take constant time

- exact constant is not important

Example

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[ ] data) {
3    int n = data.length;
4    double currentMax = data[0];          // assume first entry is biggest (for now)
5    for (int j=1; j < n; j++)             // consider all other entries
6      if (data[j] > currentMax)           // if data[j] is biggest thus far...
7        currentMax = data[j];             // record it as the current max
8    return currentMax;
9  }
```

- #3:
- #4:
- #5:
- #6:
- #7:
- #8:

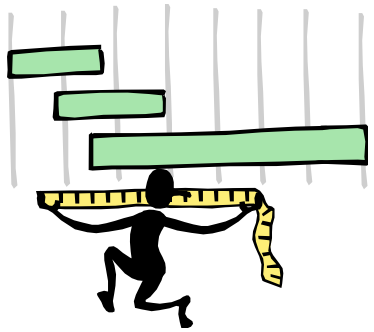
Example

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[ ] data) {
3    int n = data.length;
4    double currentMax = data[0];          // assume first entry is biggest (for now)
5    for (int j=1; j < n; j++)             // consider all other entries
6      if (data[j] > currentMax)           // if data[j] is biggest thus far...
7        currentMax = data[j];             // record it as the current max
8    return currentMax;
9  }
```

- #3: 1 op
- #4: 1 op
- #5: $2n$ ops

- #6: $n - 1$ ops,
- #7: 0 to $n - 1$ ops,
- #8: 1 op

Estimate Running Time



```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[ ] data) {
3    int n = data.length;
4    double currentMax = data[0];          // assume first entry is biggest (for now)
5    for (int j=1; j < n; j++)             // consider all other entries
6      if (data[j] > currentMax)           // if data[j] is biggest thus far...
7        currentMax = data[j];             // record it as the current max
8    return currentMax;
9  }
```

`arrayMax` executes a total of $4n + 1$ primitive operations in the worst case, $3n + 2$ in the best case

Let a = fastest primitive operation time,
b = slowest primitive operation time

Let $T(n)$ denote the worst-case time of `arrayMax`:

$T(n) \leq b(4n + 1) = O(n)$

Growth Rate of Running Time

Changing the hardware/ software environment

- Affects $T(n)$ by a constant factor, but
- Does not alter the growth rate of $T(n)$

The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm `arrayMax`

Growth Rate

n	$\log n$	n	$n \log n$	n^2	n^3	2^n

Growth Rate

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256

Growth Rate

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536

Growth Rate

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296

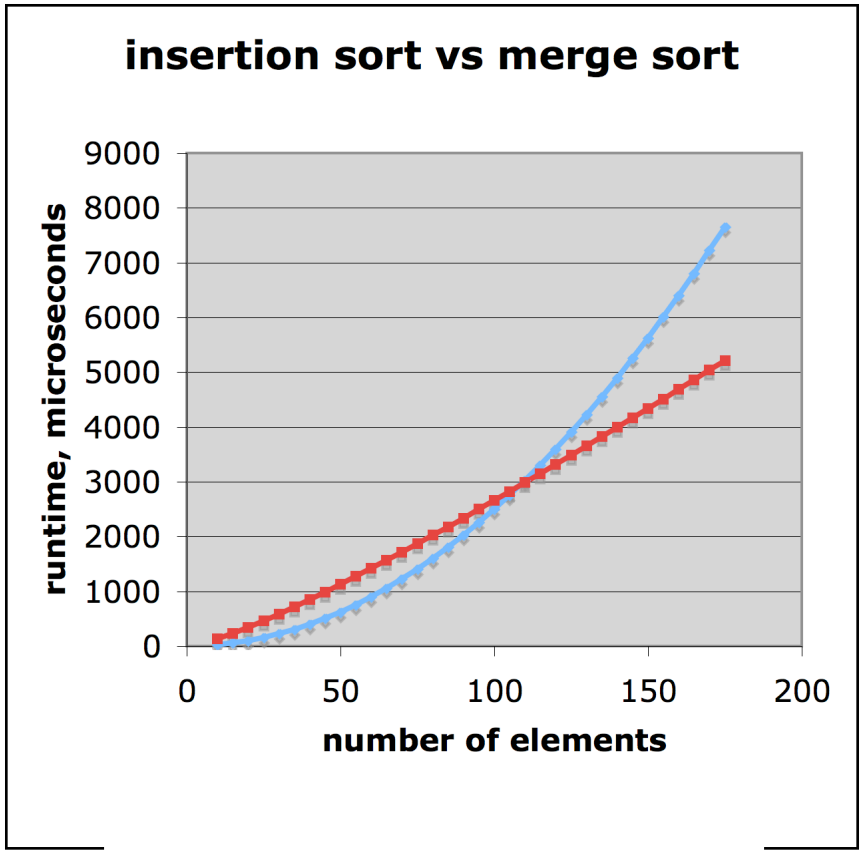
Growth Rate

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}

Growth Rate

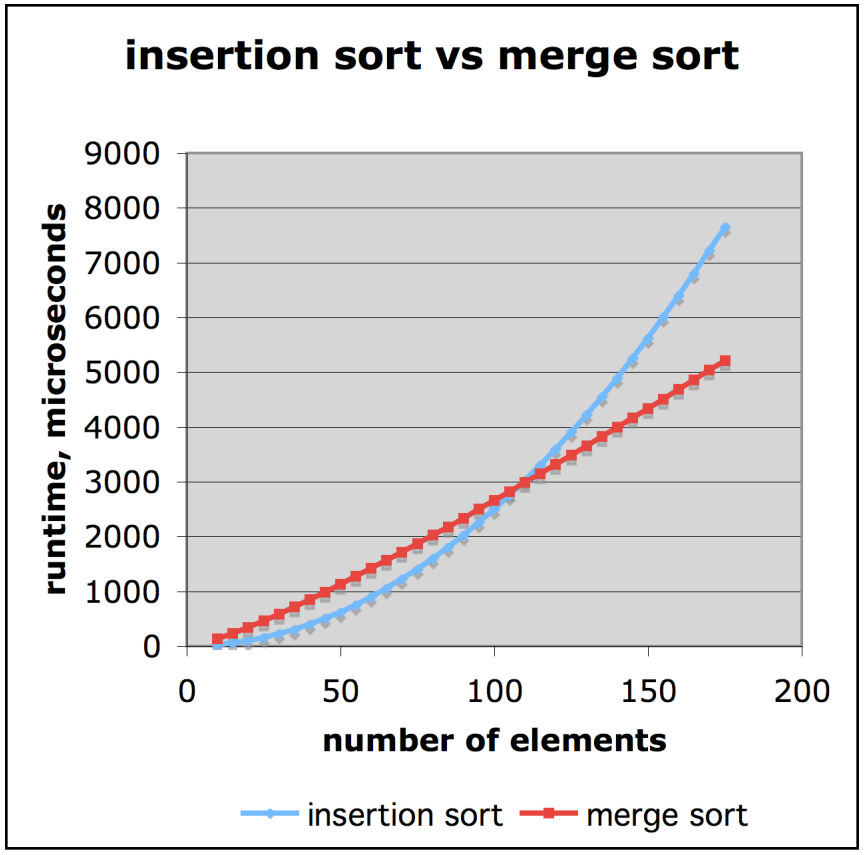
n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Comparison of Two Algorithms



- insertion sort: $n^2/4$
- merge sort: $2nlogn$
- $n = 1000000$
  - insertion sort:
    70 hours
    40 minutes
  - merge sort:
    40 seconds
    0.5 seconds

Comparison of Two Algorithms



- insertion sort: $n^2/4$
- merge sort: $2nlogn$
- $n = 1000000$
  - insertion sort:
    70 hours
    40 minutes
  - merge sort:
    40 seconds
    0.5 seconds

Asymptotic Notation

Provides a way to simplify analysis

Allows us to ignore less important elements

- constant factors

Focus on the largest growth of n

Focus on the dominant term

How do these functions grow?

- $f_1(x) = 43n^2 \log^4 n + 12n^3 \log n + 52n \log n$
- $f_2(x) = 15n^2 + 7n \log^3 n$
- $f_3(x) = 3n + 4 \log_5 n + 91n^2$
- $f_4(x) = 13 \cdot 3^{2n+9} + 4n^9$
- $f_5(x) = \sum_{i=0}^{\infty} \frac{1}{2^i}$

Big O and Growth Rate

Big-O notation gives an upper bound on the growth rate

$f(n) = O(g(n))$ means :

the growth of $f(n)$ is no more than $g(n)$

$f(n)$ can be a complicated polynomial

$g(n)$ is one of a few highly-recognizable simple polynomials

Examples

- $7n + 2$
- $3n^3 + 20n^2 + 5$
- $3logn + 5$

Big-O Analysis

1. Write a polynomial in terms of input size n
   - Only loops contribute
   - Each nested factor is multiplied
   - Each sequential factor is summed

2. Simplify the polynomial
   - Identify dominant term – highest degree polynomial
   - Polynomials beat polylogs
   - Exponentials beat polynomials
   - Discard constants

Examples

```
1  /** Returns true if there is no element common to all three arrays. */
2  public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
3    for (int a : groupA)
4      for (int b : groupB)
5        for (int c : groupC)
6          if ((a == b) && (b == c))
7            return false;                    // we found a common value
8    return true;                             // if we reach this, sets are disjoint
9  }
```

```
1  /** Returns true if there is no element common to all three arrays. */
2  public static boolean disjoint2(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
3    for (int a : groupA)
4      for (int b : groupB)
5        if (a == b)                  // only check C when we find match from A and B
6          for (int c : groupC)
7            if (a == c)              // and thus b == c as well
8              return false;          // we found a common value
9    return true;                     // if we reach this, sets are disjoint
10 }
```

Summations

- Constant series

$$\sum_{i=a}^{b} 1 = \max(b - a + 1, 0)$$

- Arithmetic series

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \in \Theta(n^2)$$

- Quadratic series

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{2n^3 + 3n^2 + n}{6} \in \Theta(n^3)$$

Example

```
void triStars(int size) {
  for (int i=0;i<size;i++){
    for (int j=i;j<size;j++){
      System.out.print("*");
    }
    System.out.println();
  }
}
```

```
void mystery(int n) {
  triStars(n);
  for (int i=0; i<200; i++) {
    for (int j=0; j<n; j++) {
      triStars(n);
    }
  }
}
```

Linear Time Algorithms: $O(n)$

Process the input in a single pass spending constant time on each item

- max, min, sum, average, linear search

Any single loop

HW00 was O(1)

$O(nlogn)$ time

Frequent running time in cases when algorithms involve:

- Sorting

- Divide and conquer
  - Think binary search

Quadratic Time: $O(n^2)$

Nested loops, double loops

Your HW01: when reading in the second file - find a zipcode match in an `ExpandableArray` of n elements, n times

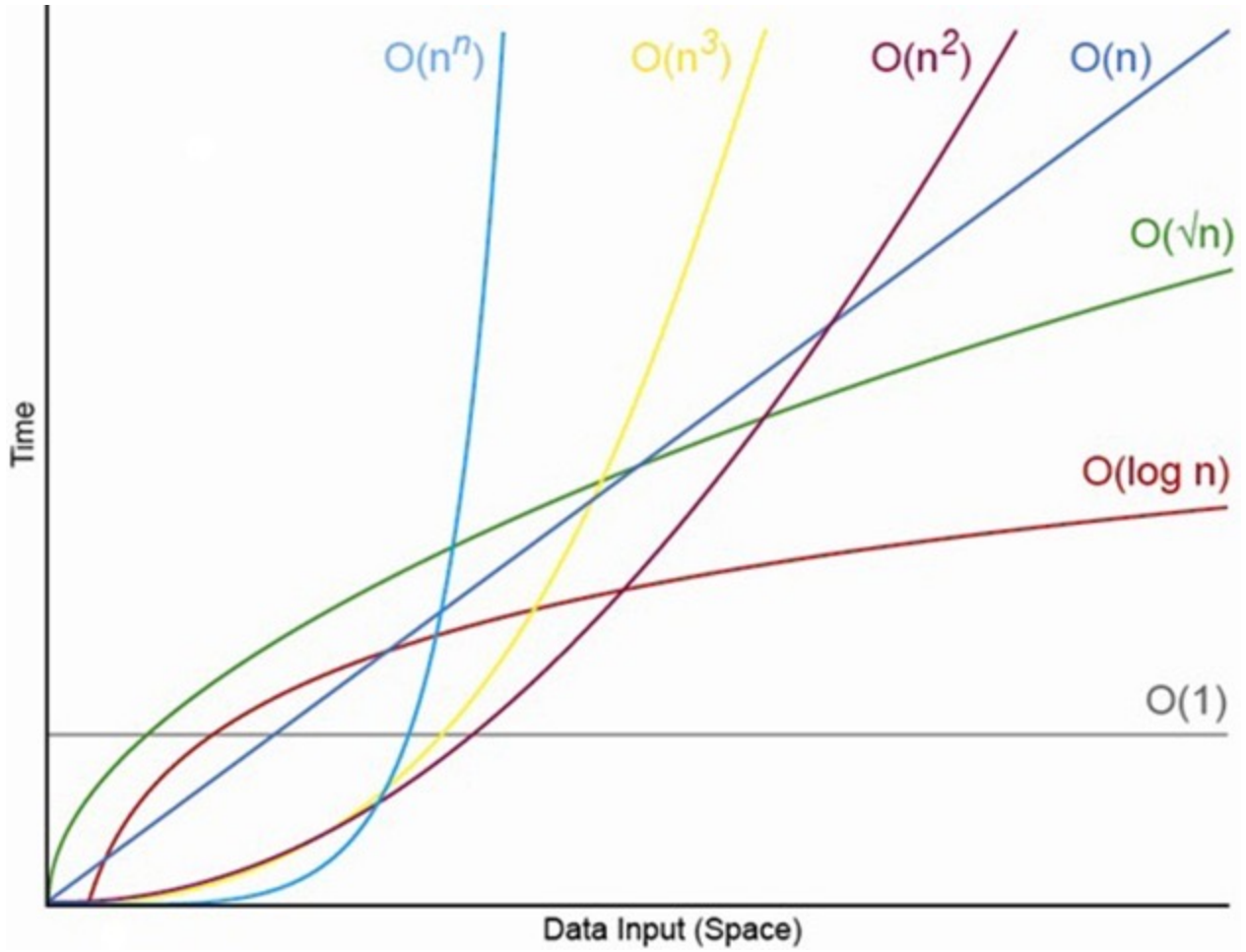
Processing all pairs of elements

Slow Times

All subsets of n elements of size k: $O(n^k)$

All subsets of n elements (power set): $O(2^n)$

All permutations of n elements: $O(n!)$


O(n^n)
O(n^3)
O(n^2)
O(n)
O(√n)
O(log n)
O(1)
Time
Data Input (Space)