Runtime Analysis: O notation

• \( T(N) \) is \( O(F(N)) \) if there are positive constants \( c \) and \( N_0 \) such that \( T(N) \leq c \cdot F(N) \) for all \( N \geq N_0 \).

• \( T(N) \) is bounded by a multiple of \( F(N) \) from above for every big enough \( N \).
Adding and multiplying functions

- **Rule for sums** (e.g. - two consecutive blocks of code): If \( T_1(N) = O(F(N)) \) and \( T_2(N) = O(G(N)) \) then \( T_1 + T_2 = O(\max(F(N),G(N))) \). The biggest contribution dominates the sum.

- **Rule for products** (e.g. - an inner loop run by an outer loop): If \( T_1(N) = O(F(N)) \) and \( T_2(N) = O(G(N)) \) then \( T_1 \times T_2 = O(F(N) \times G(N)) \).

- Example: \( (n^2+2n+17) \times (2n^2+n+17) = O(n^2 \times n^2) = O(n^4) \). (remember to ignore all but the leading term).

- If we sum over a large number of terms, we multiply the number of terms by the estimated size of one term.

- Example: Sum of \( i \) from 1 to \( N \). Average size of an element: \( N/2 \). There are \( N \) terms so the sum is \( O(n^2) \). Exact term: \( N \times (N-1)/2 \).
Loops (cont.)

- Work from inside out:
  - Calculate the body of inner loop (constant – an if statement and three assignments).
  - Estimate the number of passes of the inner loop: n-i passes.
  - Estimate the number of passes of the outer loop: n passes. Each pass counts n,n-1,n-2,...1.
  - Overall 1+2+3+...+n passes of constant operations: n*(n-1)/2 = O(n^2).

- This is not the fastest sorting algorithm but it's simple and works in-place. Good for small size input. We'll go back to sorting later in the course.
Recursive functions

• Recursive functions perform some operations and then call themselves with a different (usually smaller) input.

• Example: factorial.

```c
int factorial (int n)
{
  if(n<=1) return 1;
  return n*factorial(n-1);
}
```
Runtime analysis

• O(1) operations before recursive call – if statement and a multiplication.
• So: \( T(n) = c + T(n-1) \).
• Similarly: \( T(n-1) = c + T(n-2) \) \( \Rightarrow \) \( T(n) = 2c + T(n-2) \).
• After \( n \) such equations we reach \( T(1) = k \) (just the if-statement).
• \( T(n) = (n-1)c + k = O(n) \).
• Iterative function performs the same.
Ill-behaved recursion

• Recall power2 from homework 1
  ```
  int power2(int n) {
    if (n==0) return 1;
    return power2(n-1)+power2(n-1);
  }
  ```
• The call tree looks like a big binary tree of height n, so $T(N)$ is exponential.
• But double recursion is not bad, as long as we split the work too!
• Example: Merge sort – sort recursively two halves of an array and merge. Call recursively twice, but on different input! The work is split between recursive calls in a smart way.
APIs and encapsulation

- **Application Programming Interface (API)**: a group of function specifications that are meant to provide some service or services to programs, clients of the API.
- **Data encapsulation**: Hiding the data of the implementation of the API functions so well that the client (programs using the API) can’t get at it by dotting into its variables.
- The clients **have** to use the API functions to get their work done.
Data encapsulation

Implementation details and class variables

API

Implementation details are hidden from the client, who has to use the API.
Implementation details and class variables are hidden from the client, who has to use the API.
Bank account example

- Example – bank account.
- Construct an account:
  
  ```java
  BankAccnt ba =
      new BankAccnt(“JJ”, 234, 100);
  ```

- Modify balance:
  
  ```java
  ba.withdraw(20); ba.deposit(10);
  ```

- Access balance:
  
  ```java
  ba.balance; // Incorrect!!
  ba.getBalance(); // Correct!!
  ```

- The API is the method specification. An API lies between two bodies of code, the client code and the API implementation code.
• A Java interface can describe and codify API actions, or a subset of them. Here is an interface for account objects:

```java
public interface Account {
    // withdraw amt from this BA
    int withdraw(int amount);
    // deposit amt to this BA
    void deposit(int amount);
    // return balance for this BA
    int getBalance();
}
```

• Note an interface can’t include constructor(s)
• So an API = constructors + interface methods (possibly + others)

Note: S&W has APIs, but doesn’t set up interfaces for them.
Any class that has a balance going up and down can implement this interface:

```java
public class BankAccount implements Account {
    // constructor - create a BA
    public BankAccount(String nm, int _id, int bal) {...}
    // Account API functions
    public int withdraw(int amt) {...}
    public void deposit(int amt) {...}
    public int getBalance() {...}
    // Fields—all private
    private int id;
    private String name;
    private int balance;
}
```
Bank account example (cont.)

// client code example:
public class TestBankAccount {
    static void main(String [] args) {
        Account ba =
        new BankAccount("JJ", 234, 100);
        ba.withdraw(20);
        ba.deposit(10);
    }
}

• Here ba is of static type Account but dynamic type BankAccount.
• We can only use methods deposit and withdraw (plus Object methods) with ba, even if BankAccount has another method defined.
Java Interfaces:

Consider the Bag API, page 121

public class Bag<Item> implements Iterable<Item>
Bag() -- constructor, not part of Java interface
void add(Item item)
boolean isEmpty()
int size()

Here is an Interface with all the methods of the API:
public interface Bag<Item> implements Iterable<Item>
{
    void add(Item item);
    boolean isEmpty();
    int size();
}

Promises method iterator<Item>()
An Interface is a contract

• Recall the PA1 Ideas slides and the story of two teams agreeing on the Tokenizer interface...

• By agreeing to this API, codified in the Java interface, the two teams have a contract between them:
  • The Tokenizer team implements the API
  • The Xref team implements the code that calls the API

• Note that the interface doesn’t capture everything about the interactions: for example, the constructor syntax is missing, and the treatment of end-of-data.
  • But it is an excellent start on the full contract.
Collections in Java

- A collection is a container of objects.
- A collection may be ordered or unordered.
- It may or may not allow duplicates.
- The Java **Collection Interface** lays down the foundation—see next slide.
- Java Collection classes we’re studying:
  - Lists, Stacks, Queues, Sets, and PriorityQueues.
  - All these implement Collection<E>
  - We also study Maps, which have their own interface, but fit in with the others nicely
JDK Collection Interface  
(somewhat simplified)

// Collection interface; the root of all collections.
public interface Collection<AnyType> extends 
    Iterable<AnyType>, java.io.Serializable
{
    int size(); // How many items are in this collection.
    boolean isEmpty(); // Is this collection empty?
    boolean contains(Object x); // Is X in collection?
    boolean add(AnyType x); // Adds x to collection.
    boolean remove(Object x); // Removes x from collection.
    void clear(); // Change collection size to zero.
    // Obtains an Iterator object to traverse collection.
    Iterator<AnyType> iterator();
    // Obtains a primitive array view of the collection.
    Object[] toArray();
    // Obtains a primitive array view of the collection.
    <OtherType> OtherType[] toArray(OtherType[] arr);
}

Additional methods that are really here, covered later: addAll, containsAll.
removeAll, retainAll.
Iterators traverse collections

// Iterator interface from JDK (Java 5+)
public interface Iterator<AnyType> {
    // Are there any items not yet iterated over
    boolean hasNext();
    // Obtains the next item in the collection
    AnyType next();
    // Remove the last item returned by next.
    // Can only be called once after next
    void remove();
}
JDK Sets

- A set contains a number of elements, with no duplicates and no order.
- A = {1, 5, 3, 96}, or B = {17, 5, 1, 96}, C={"Mary", "contrary", "quite"}. Incorrect -{"Mary", "contrary", "quite", "Mary"}.
- In Java, the Set interface is the Collection interface.
- The API isn’t sensitive to the lack of duplicates, only the implementation. The implementations in the JDK are the TreeSet and HashSet.
- They check for duplicates by using the equals method (HashSet) or compareTo (TreeSet) of the elements.
- HashSet uses hashCode() and equals() and TreeSet uses compareTo() and equals()—recall homework2 #8 on elements Point2D, etc.
Maps - definition

- Given two sets, Domain and Range, with a relation from one to another.
- Like a math function, each domain element has associated with it exactly one range element.
- Two arrows can land on the same range element, but one domain element cannot have two arrows out of it.
- S&W calls maps “symbol tables”, but it’s exactly the same idea.
Maps (cont.)

- A mapping is a collection like other collections we are studying, lists, stack, queues, and sets.
- However, in Java a Map has its own interface separate from Collection.
- Note that not every collection of pairs makes a proper map: M qualifies as a map only if the collection of keys has no duplicates, i.e., constitutes a set.
- Recall Homework 2, showing that the JDK Map API and S&W’s ST API are extremely similar.
public interface Map<KeyType,ValueType> extends Serializable
{
    // Returns the number of keys in this map.
    int size();
    // Tests if this map is empty.
    boolean isEmpty();
    // Tests if this map contains a given key.
    boolean containsKey( KeyType key );
    // Returns the value in the map associated with the key.
    ValueType get( KeyType key );
    // Adds the key value pair to the map, overriding the
    // original value if the key was already present.
    ValueType put( KeyType key, ValueType value );
    // Remove the key and its value from the map.
    ValueType remove( KeyType key );
    // Removes all key value pairs from the map.
    void clear();
    // Returns the keys in the map.
    Set<KeyType> keySet();
    // Returns the values in the map. There may be duplicates.
    Collection<ValueType> values();
    // Return a set of Map.Entry objects corresponding to
    Set<Entry<KeyType,ValueType>> entrySet();
}
The Map interface: helper interface for entrySet()

```java
/**
 * The interface used to access the key/value pairs in a map.
 * From a map, use entrySet().iterator to obtain an iterator
 * over a Set of pairs. The next() method of this iterator
 * yields objects of type Map.Entry.
 */

public interface Entry<KeyType,ValueType> extends Serializable
{
    // Obtains this pair's key.
    KeyType getKey();

    // Obtains this pair's value.
    ValueType getValue();

    // Change this pair's value.
    ValueType setValue(ValueType newValue);
}
```
Map Apps/Clients

- FrequencyCounter: important Map app example that shows how to convert from S&W “symbol tables” to JDK Maps. See S&W page 372 and the JDK-based FrequencyCounter.java in TestMap.zip.

- PA1 used a Map from String to List of Integers to tabulate tokens (Java identifiers) in a Java source.

- Homework3 Pseudocode for a Map app, “instant resume scorer”.

- PA2’s TestMap and TestMapPerf are simple Map apps.
Collection types: The **List**

- A *List* is an ordered sequence of elements: $a_0, a_1, a_2, \ldots, a_{n-1}$.

- The List interface, extending the Collection interface, is on the next slide.
One interface can extend another interface, meaning that its methods are added to the other interface, so here a List has size(), isEmpty(), contains(), etc. from Collection, plus these three methods.
Collection types: The List Interface

ListIterator, an iterator that can back up along the list

```java
public interface ListIterator<AnyType> extends Iterator<AnyType>
{
    // Tests if there are more items in the collection
    // when iterating in reverse.
    boolean hasPrevious( );
    // Obtains the previous item in the collection
    // when traversing in reverse.
    AnyType previous( );
    // Remove the last item returned by next or previous.
    // Can only be called once after next or previous.
    void remove( );
}
```
Collection types: The List Interface

- The two most important classes that implement the List interface are **LinkedList** and **ArrayList**.
- We call them “concrete classes”: we can *instantiate* them, e.g., do “new LinkedList()”
- They have different performance for large lists.
- Both have extra methods over and above the List interface, plus constructors of course.
- Recall the homework2 problem on the ArrayList’s `trimToSize()` method, not in the List interface.
Mental Model of a List

• A 4-element LinkedList has refs in both directions, a “doubly linked list”

• We can get(0), ..., get(3) and access any particular object ref. We can set(0, b) and replace the object at 0 with b.
• What happens if we set(4, b)? (Fails)
• To grow the list we need to use add(Object x), but where does it go?? (adds to the end of the list)
• This is fast because the LinkedList tracks the end-of-list
• Recall many more slides on list behavior for class
Performance of LinkedList vs. ArrayList

- For ArrayList of size n
  - Get, set are very fast, O(1)
  - Append-type add is fast most of the time. If it involves array expansion, it is expensive, O(n).
  - Delete is expensive unless if it is at the end.

- For LinkedList of size n
  - Get, set depends on the index position
  - get(1) is done by two next’s down the list from the beginning of the list, and get(n-2) is done by two previous next’s from the end of the list
  - Most expensive is get(n/2)
  - Delete/add is easy once the right spot in the list is located. Remove in an iterator is O(1), but the larger task may involve O(n) next’s to get the iterator positioned.
Map Implementation: **Hashing** – quick intro

Hash function properties:

1. Map key elements to integers.
2. Fast to calculate.
3. Try to minimize *collisions* – when different keys hash to the same value.

From Wikipedia
Hashing terminology

• **Keys**: each value of type keytype can be called a key. It just means that we're going to do *a look-up* using this value.

• **Hash table**: the array in use, of some size M.

• **Hash bucket or hash slot**: a subscript in the hash table array, these are numbered from 0 to M-1. M is the number of buckets.

• **Hash function**: a function from the keytype to a bucket-number: \( b = h(x) \), where \( x \) is of type keytype and \( 0 \leq b < M \) is the bucket number. We say "\( x \) hashes to \( b \)". \( h(x) \) is a computed mapping and is expected to take \( O(1) \) computation time.

• **Collision**: when two keys \( x \) and \( y \) hash to the same bucket: \( b = h(x) = h(y) \).
Collisions: Various solutions

- A collision happens when two keys hash to the same bucket, i.e., the same hash-value. What to do?

1. Separate chaining. Make the hash table an array of linked lists.

2. Linear probing. If the first spot is full, use another spot in the hash table
   - look in the next spot down/wrapped in the array.
   - We’ll look at demo at Princeton.
Separate Chaining HT: pp. 464-468

- Each hash array element is a linked list holding all the keys that hash to that bucket - all the collision participants. No further probing is needed, just list operations.
- This is the simplest way to make a hash table that works decently. Many hash tables work this way, including HashMap in the JDK, as we are seeing in PA2.
- In some cases separate chaining may be a little slower than linear probing, because it causes memory references that hop around in memory more. This could be important for very large hash tables.
Separate Chaining Data Structure
on pg. 464, and like our PA2 HashMap, and hw3 #1
Understanding Object Graphs

- This represents one `SequentialSearchST` (pg. 375) object.
- `first`: the instance variable of `SequentialSearchST` object that references the first Node object for the list.
- The box with A and 8 in little boxes: this represents one Node object for the list, with its instance variables key = “A” and val = 8. Also the arrow out of this box represents the ref called “next” of the Node object
- The Node with E and 12 has no arrow out because its next = null, the end of the list.
Linear Probing: pp. 469-474 and hw3

- If bucket $b = h(x)$ is already in use, try $b+1$, then $b+2$, etc., wrapping around to $b = 0$ if you hit $M$, the table size.
  - $b = (b + 1) \% M$.
- As soon as you find an empty spot, take it.
- On lookup, hash the key and check if it matches the one in the hash slot, if not, try the next (wrapping if necessary), etc., until you find the matching one or an empty one.
- The performance can suffer because stretches of the array get filled up and probes have to go further and further. Need to resize aggressively.
- Also, you can’t delete entries in a simple way, just removing them, because they have been used as stepping stones in other key’s probing sequences.
Example of an inner class: Stack.java, pg. 149 from slides 26-28 of HashTables07.pdf

```java
public class Stack<Item> implements Iterable<Item> {
    private Node first; // instance variable (field) of Stack class
    private int n;      // another field

    private class Node {
        Item item;
        Node next;
    }

    • This inner class inside Stack.java doesn’t bother with “private Item item;” or “private Node next;” because the whole class is private inside Stack.
    • And the variables of the inner class are always accessible to the outer class’s code anyway.
    • S&W doesn’t discuss inner classes much, just a little note on pg. 159.
```
More on Sets

• Consider set of integers: \( A = \{1, 5, 3, 96\} \), or \( B = \{17, 5, 1, 96\} \) and a set of strings, \{"Mary", "contrary", "quite"\}.

• Sets have no duplicates and order doesn't count. (vs. List allows duplicates and order does count.)

• The Set API is just the Collection API.

• So Sets have methods isEmpty, size, add, contains, remove, clear, toArray, iterator

• We will see the Collection API in fact has more methods, but first look at basic OO considerations.
Set operation examples: union

• Consider Sets made with HashSet:

```java
Set<String> a = new HashSet<String>();
```

• Union of Set<String> a and Set<String> b: we can “addAll” of b to a or vice versa, to form the union set:

```java
a.addAll(b)  // a is the union of a and b
b.addAll(a)  // turns b into the union
```

• If we don’t want to change a or b, make a new set from one of them first:

```java
Set<String> union = new HashSet<String>(a);
union.addAll(b);  // form a UNION b in own set
```
How does a Set hold its element objects?

We programmers need to keep in mind that our Set *elements* can be shared. The encapsulation of Sets covers their handling of the their element refs, but not the element objects themselves.

Ref from variable:

Integer \( x = 5 \);
\( a \).add(\( x \));

\( a = \{1,3,5,96\} \)

P.S. Same is true of elements of Lists, Maps, etc.
Set operation examples : union

• Consider Sets made with HashSet:

```java
Set<String> a = new HashSet<String>();
```

• Union of Set<String> a and Set<String> b: we can “addAll” of b to a or vice versa, to form the union set:

```java
a.addAll(b)  // a is the union of a and b
b.addAll(a)  // turns b into the union
```

• If we don’t want to change a or b, make a new set from one of them first:

```java
Set<String> union = new HashSet<String>(a);
union.addAll(b);  // form a UNION b in own set
```
Set Apps

- Toy example with vowels in class More on Sets: set of Character
- Monitoring users on lines, in same class: sets of Strings for user names
- Movie rentals: find the next movie to binge on, in same class: sets of String movie titles, also map of String to object containing a set of titles (but not to be covered on exam)
- PA1: Spell checker based on dictionary of words
- Instant Resume Scorer, in homework 3: Sets of words (Strings)
Set Implementations

- JDK HashSet: fast $O(1)$ add, contains, needs compatible equals, hashCode for elements
- JDK TreeSet: a little slower $O(\log N)$ add, contains, needs compatible equals, compareTo for elements, or can provide a Comparator in the constructor (made easy by Java 8+)
- Bit Vectors: not in JDK or S&W, but easy to implement, provide $O(1)$ add, contains, very fast union, intersection, etc., doing up to 64 elements per CPU instruction. Covered at the end of class More on Sets. But not on exam (FYI).
Priority Queues: only the basics

- `insert(item)` insert into proper place in PQ
- `FindMin()` return min element by priority
- `DeleteMin()` delete and return min element by priority
- `isEmpty()` and/or `size()` also needed
- `HW3`: Using JDK `PriorityQueue` to implement S&W “classic” PQ, MinPQ.
Sorting (Chapter 2)

- One of the most fundamental problems in CS.
- Problem definition: Given a sequence of elements with a well-defined order, return a sequence of the elements sorted according to this order.

Simple (insertion) Sort – runs in quadratic time (“bad sort”)
Shell sort – runs in sub-quadratic time (“pretty good sort”)
Merge sort – runs in O(NlogN) time (“good sort”)
Quick sort - runs in average O(NlogN) time (“good sort”)

Mergesort – divide and conquer

• 3 steps
  – Return if the number of items to sort is 0 or 1
  – Recursively Mergesort the first and second halves separately
  – Merge the two sorted halves into a sorted group
• This approach is called “divide and conquer”.
• Mergesort is an O(N*\log N) algorithm
Quicksort – another divide-and-conquer sort algorithm

• 4 steps to sort S:
  – Return if the number of elements in S is 0 or 1
  – Pick a “pivot” - element v in S
  – Partition S-{v} into 2 disjoint groups:
    \[ L = \{ x \in S - \{v\} \mid x \leq v \} \]
    \[ R = \{ x \in S - \{v\} \mid x \geq v \} \]
  – Return the result of Quicksort(L) followed by v followed by Quicksort(R)
JDK Arrays.sort Example

• Recall Student, with id as identifier field, and gpa as another field

• Suppose we have an array of Students students:

  Student students = [new Student(100, “Joe”, 3.2),
                     new Student(130, “Sue”, 3.9), …];

  Arrays.sort(students) sorts by id

  Arrays.sort(students,
               Comparator.comparing(Student::getGpa))

  Sorts by GPA using a Java 8 easy comparator

homework 3: calling Arrays.sort, using easy Java 8 Comparator

3/23/2021
Binary search: another divide-and-conquer algorithm

- **Definition**: Search for an element in a sorted array. Implemented, pg. 47.
- Implemented in JDK as part of the Collections API.
- Return array index where element is found or a negative value if not found.
- Idea from the book – start in the middle of the array. If the element is smaller than that, search in the smaller half. Otherwise, search in the larger half.
Subclass Inheritance

- For decades considered the bedrock of object oriented software practice, but lately less championed.
- Used in JDK HashMap and related classes, so we had to cover it a little for PA2, to warn you about the hovering AbstractMap class.
- But not enough homework on it yet, so not explicitly on exam.
Algorithm patterns

- Greedy algorithms and Dynamic Programming are too recent to be on the exam (not yet on homework)
- So the only algorithm pattern we have covered for the exam is divide and conquer, where the major examples are sorting and searching, first covered in cs210.