Midterm Review

Data Structures and Algorithms

Warm Up

Imagine you are asked to test a method in a class that implements a Binary Search Tree. This method should be able to take in any node and then return back the height of the given tree.

```
public int treeHeight(TreeNode node)
```

What are some unit tests you would write to validate this method works like it should?

What would you name those unit tests?

What are some examples of trees you would pass into the method to test if it can handle all appropriate cases?

Logistics

50 minutes

8.5 x 11 in note page, front and back

Review tonight

4:30-6:30 in MLR 301

Extra TA office hours

No office hours on Friday

ADTs vs Data Structures

Data Structure

- *A way of organizing and storing related data points*
- An object that implements the functionality of a specified ADT
- Describes exactly how the collection will perform the required operations
- Examples: LinkedIntList, ArrayIntList

Algorithm

- A series of precise instructions used to perform a task
- Examples from CSE 14X: binary search, merge sort, recursive backtracking

Abstract Data Type (ADT)

- *A definition for expected operations and behavior*
- A mathematical description of a collection with a set of supported operations and how they should behave when called upon
- Describes what a collection does, not how it does it
- Can be expressed as an interface
- Examples: List, Map, Set

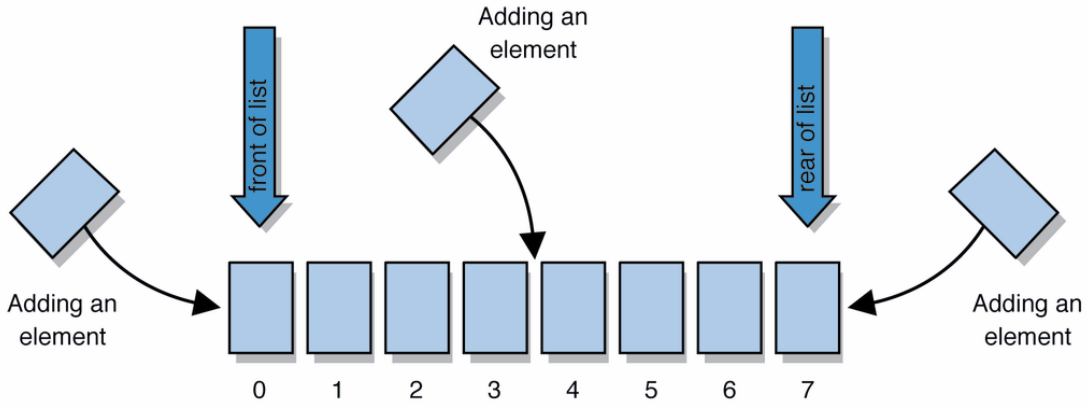
List ADT

list: stores an ordered sequence of information.

- Each item is accessible by an index.
- Lists have a variable size as items can be added and removed

Supported Operations:

- **get(index):** returns the item at the given index
- **set(value, index):** sets the item at the given index to the given value
- **append(value):** adds the given item to the end of the list
- **insert(value, index):** insert the given item at the given index maintaining order
- **delete(index):** removes the item at the given index maintaining order
- **size():** returns the number of elements in the list



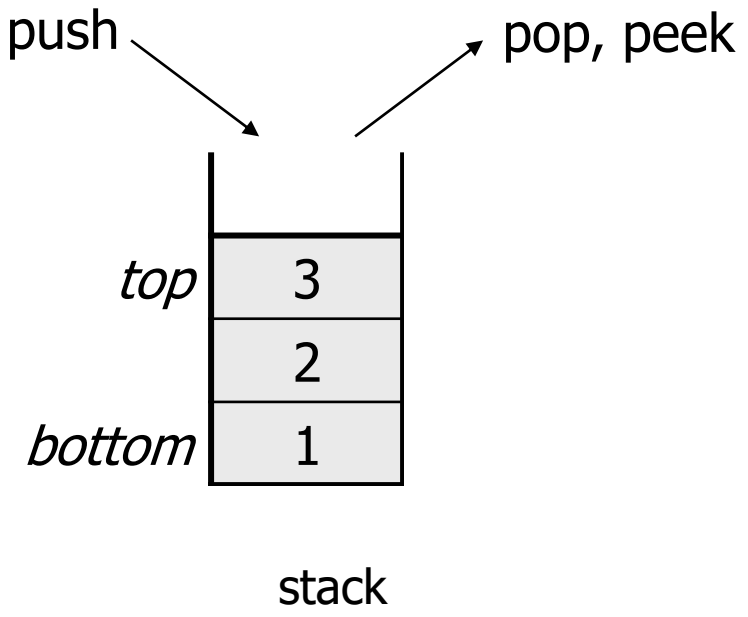
Stack ADT

stack: A collection based on the principle of adding elements and retrieving them in the opposite order.

- Last-In, First-Out ("LIFO")
- Elements are stored in order of insertion.
  - We do not think of them as having indexes.
- Client can only add/remove/examine the last element added (the "top").

basic stack operations:

- **push(item)**: Add an element to the top of stack
- **pop()**: Remove the top element and returns it
- **peek()**: Examine the top element without removing it
- **size()**: how many items are in the stack?
- **isEmpty()**: true if there are 1 or more items in stack, false otherwise



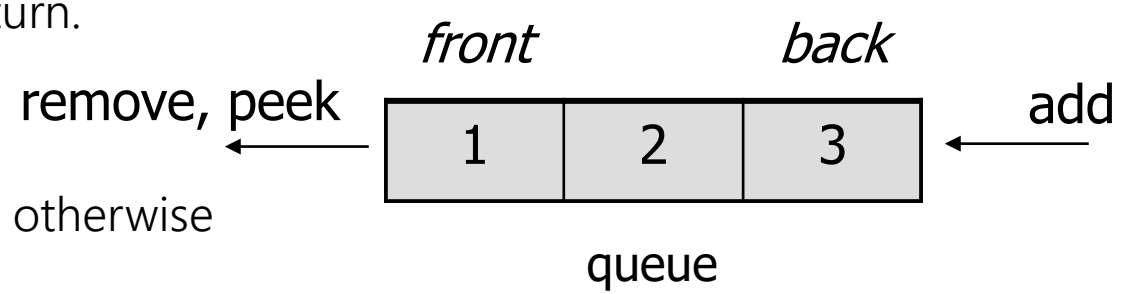
Queue ADT

queue: Retrieves elements in the order they were added.

- First-In, First-Out ("FIFO")
- Elements are stored in order of insertion but don't have indexes.
- Client can only add to the end of the queue, and can only examine/remove the front of the queue.

basic queue operations:

- **add(item):** aka "enqueue" add an element to the back.
- **remove():** aka "dequeue" Remove the front element and return.
- **peek()**: Examine the front element without removing it.
- **size():** how many items are stored in the queue?
- **isEmpty():** if 1 or more items in the queue returns true, false otherwise



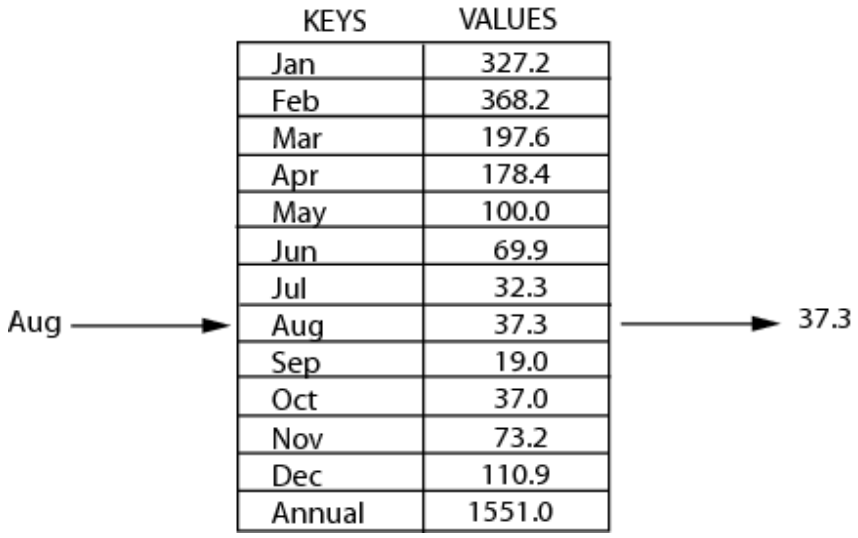
Map ADT

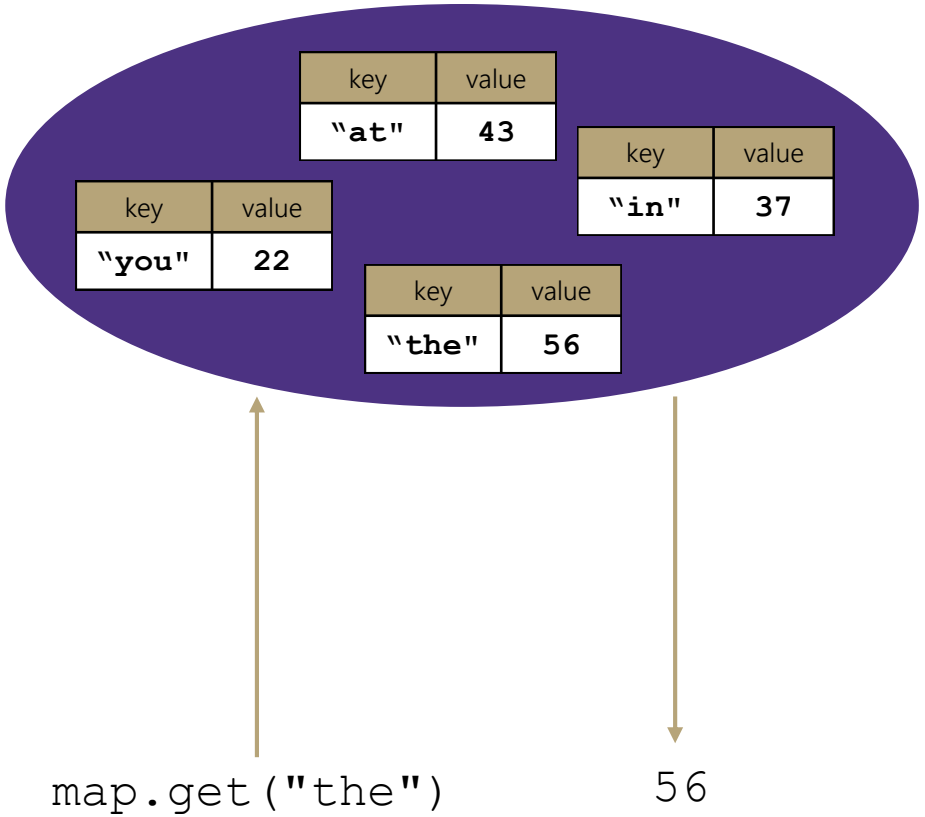
map: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.

- a.k.a. "dictionary", "associative array", "hash"

operations:

- **put**(*key*, *value*): Adds a mapping from a key to a value.
- **get**(*key*): Retrieves the value mapped to the key.
- **remove**(*key*): Removes the given key and its mapped value.





Iterators

iterator: a Java interface that dictates how a collection of data should be traversed.

Behaviors:

hasNext() – returns true if the iteration has more elements

next() – returns the next element in the iteration

```
while (iterator.hasNext()) {
    T item = iterator.next();
}
```

Testing and Debugging

Expected behavior
- The main use case scenario
- Does your code do what it should given friendly conditions?

Forbidden Input
- What are all the ways the user can mess up?

Empty/Null
- Protect yourself!
- How do things get started?

Boundary/Edge Cases
- First
- last

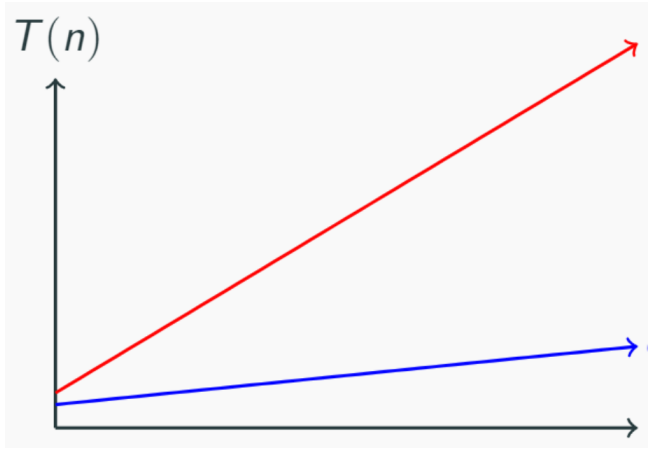
Scale
- Is there a difference between 10, 100, 1000, 10000 items?

Black Box
Behavior only – ADT requirements
From an outside point of view
Does your code uphold its contracts with its users?
Performance/efficiency

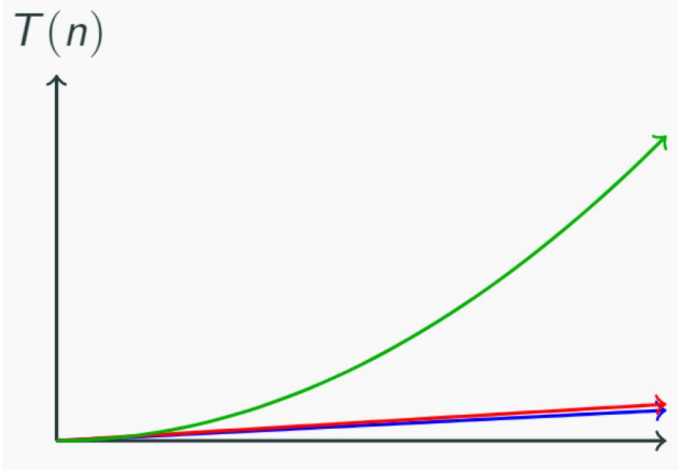
White Box
Includes an understanding of the implementation
Written by the author as they develop their code
Break apart requirements into smaller steps
“unit tests” break implementation into single assertions

Asymptotic Analysis

asymptotic analysis: how the runtime of an algorithm grows as the data set grows

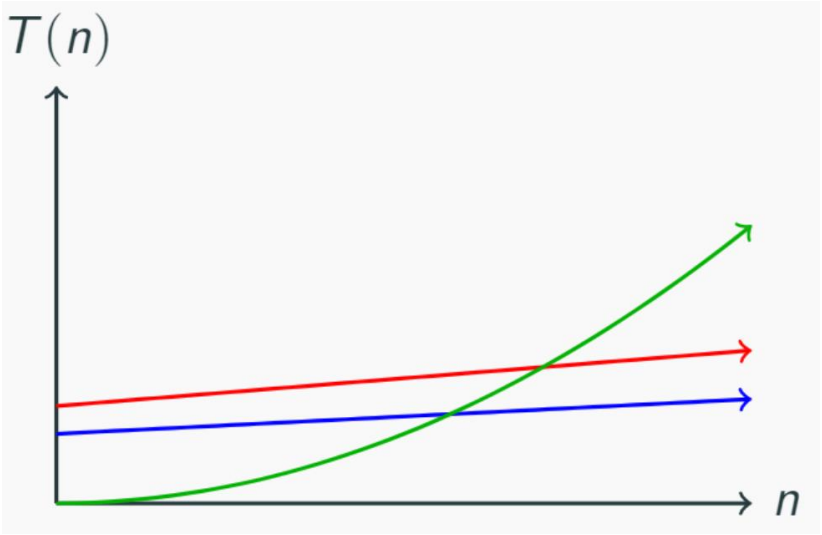


n and 4n look very different up close



n and 4n look the same over time

n^2 eventually dominates n

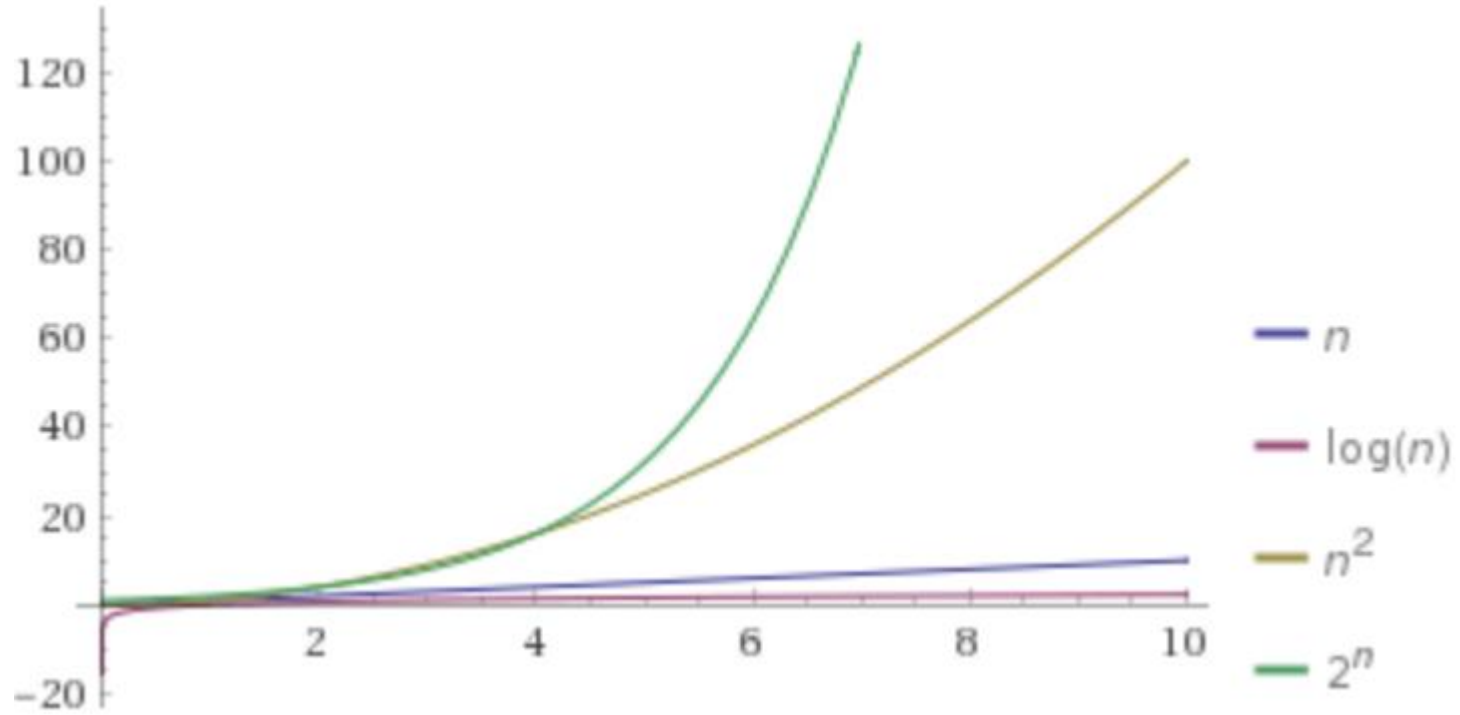


n^2 doesn't start off dominating the linear functions
It eventually takes over…

A function f(n) is dominated by g(n) when there exists two constants c > 0 and n_0 > 0 such that for all values of n ≥ n_0 f(n) ≤ c * g(n)

Domination in Graphs

Dominated by
f(n) ∈ O(g(n))



Dominates
f(n) ∈ Ω(g(n))

O, Ω, Θ Definitions

O(f(n)) is the "family" or "set" of all functions that are dominated by f(n)

f(n) ∈ O(g(n)) when f(n) <= g(n)

Ω(f(n)) is the family of all functions that dominate f(n)

f(n) ∈ Ω(g(n)) when f(n) >= g(n)

Θ(f(n)) is the family of functions that are equivalent to f(n)

We say f(n) ∈ Θ(g(n)) when both

f(n) ∈ O(g(n)) and f(n) ∈ Ω (g(n)) are true

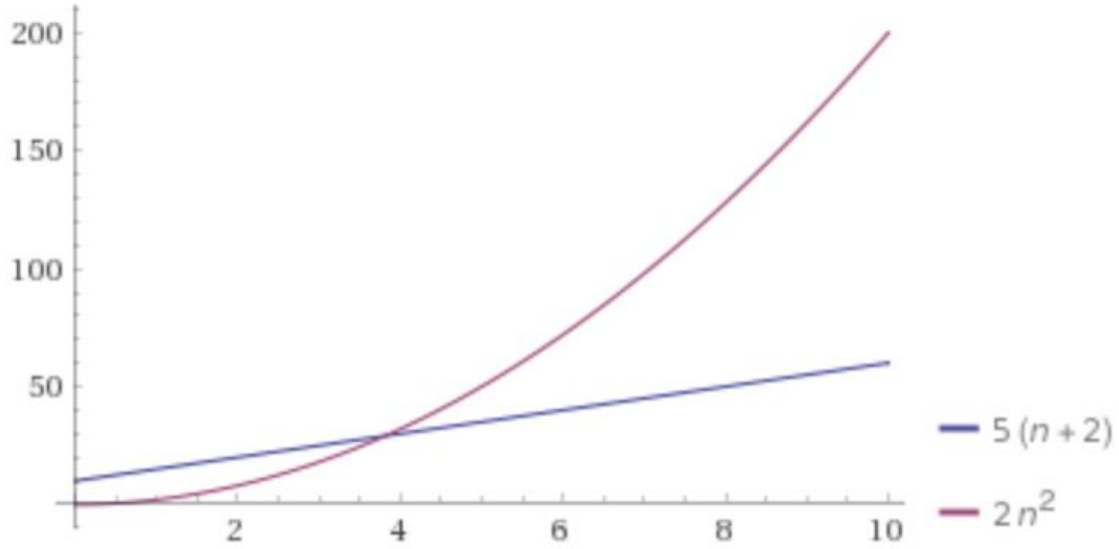
For in-depth explanation see https://piazza.com/class/jfbjq4r4em21sn?cid=188

Proving Domination

$f(n) = 5(n + 2)$

$g(n) = 2n^2$

Find a c and n_0 that show that $f(n) \in O(g(n))$.



$f(n) = 5n + 10$

$5n + 10 <= 5n^2 + 10n^2$

$5n + 10 <= 15n2$ for $n >= 1$

$15n^2 <= c * 2n^2$

$c = 15/2 = 7.5$

$n_0 = 1$

O, Ω, Θ Examples

For the following functions give the simplest tight O bound

a(n) = 10logn + 5 O(logn)

b(n) = $3^n - 4n$ $O(3^n)$

c(n) = $\frac{n}{2}$ O(n)

For the above functions indicate whether the following are true or false

a(n) ∈ O(b(n)) TRUE	b(n) ∈ O(a(n)) FALSE	c(n) ∈ O(b(n)) TRUE
a(n) ∈ O(c(n)) TRUE	b(n) ∈ O(c(n)) FALSE	c(n) ∈ O(a(n)) FALSE
a(n) ∈ Ω(b(n)) FALSE	b(n) ∈ Ω(a(n)) TRUE	c(n) ∈ Ω(b(n)) FALSE
a(n) ∈ Ω(c(n)) FALSE	b(n) ∈ Ω(c(n)) TRUE	c(n) ∈ Ω(a(n)) TRUE
a(n) ∈ Θ(b(n)) FALSE	b(n) ∈ Θ(a(n)) FALSE	c(n) ∈ Θ(b(n)) FALSE
a(n) ∈ Θ(c(n)) FALSE	b(n) ∈ Θ(c(n)) FALSE	c(n) ∈ Θ(a(n)) FALSE
a(n) ∈ Θ(a(n)) TRUE	b(n) ∈ Θ(b(n)) TRUE	c(n) ∈ Θ(c(n)) TRUE

Function Modeling

```
public int mystery(int n) {
    int result = 0;
    for (int i = 0; i < n/2; i++) {
        result++;
    }
    for (int i = 0; i < n/2; i+=2) {
        result++;
    }
    result * 10;
    return result;
}
```

$$f(n) = 3 + \frac{3}{4}n$$

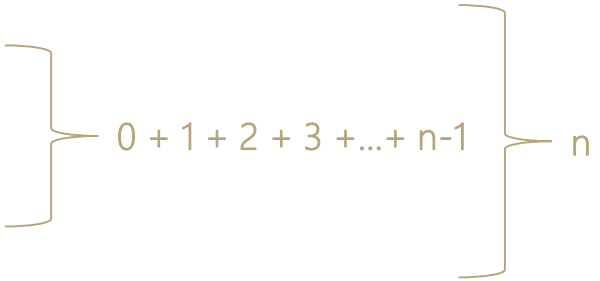
Create a mathematical model of a piece of code's runtime in relation to the given input.

You can assume basic operations all take the same constant time.

You can ignore the runtime of the operations in a for loop's header and just include how many times the for loop iterates.

Modeling Complex Loops

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("Hello!"); +c
    }
}
```



Summation
1 + 2 + 3 + 4 +… + n = $\sum_{i=1}^{n} i$

Definition: Summation

$\sum_{i=a}^{b} f(i)$ = f(a) + f(a + 1) + f(a + 2) + … + f(b-2) + f(b-1) + f(b)

T(n) = $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} c$

Function Modeling: Recursion

```
public int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
}
```

+c_1

+T(n-1)

+c_2

$$T(n) = \begin{cases} C_1 & \text{when n = 0 or 1} \\ C_2 + T(n-1) & \text{otherwise} \end{cases}$$

Definition: Recurrence

Mathematical equivalent of an if/else statement

$$f(n) = \begin{cases} runtime\ of\ base\ case\ when\ conditional \\ runtime\ of\ recursive\ case\ otherwise \end{cases}$$