Outline

- Modeling Running Time of Algorithm
- Formalizing Complexity: The Big-O Notation
- In-Class Exercises
- A Potential Pitfall
- Conclusion
In Class Exercises

- For each of the following, find $T(n)$ and $O(f(n))$

```java
public void doTaskA(int n) {
    int j = 0;
    int k = n;
    while (j < k) {
        System.out.println("A");
        j++;
        k--;
    }
}

public void doTaskB(int n) {
    for (int i = 0; i < 2000000000; i++) {
        for (int j = 0; j < n; j++) {
            System.out.println("B");
            System.out.println("B");
        }
    }
}

public void doTaskC(int n) {
    this.doTaskB(n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            this.doTaskA(n);
        }
    }
}
```

2-Step Method:

1. Determine $T(n)$
2. Identify largest term in $T(n)$, and use for $f(n)$. Remove any coefficient and lesser terms, and claim $T(n) = O(f(n))$
Recall: Linear Search

- For **Linear Search** we determined that in the worst case, \( T(n) = n \)
- So linear search is in \( O(n) \). *(Can we do better?)*

```java
public static int linear_search(int[] x, int key) {
    for (int i = 0; i < x.length; i++) {
        if (x[i] == key) {
            return i;
        }
    }
    // key not found
    return -1;
}
```
Example: Binary Search

- Linear Search is fastest known algorithm for searching (for now)
  - Items can be randomly placed in the list.
  - Thus, you *have* to visit every item. *(What if we didn't?)*

- Now assume list is sorted in increasing order
  - To find an item, do we *still* have to visit every one?
Example: Binary Search

- **Precondition**: Input list must be sorted
  - What guarantees can we make if the list isn't sorted?

```java
public int binarySearch(int[] list, int key) {
    int left = 0
    int right = list.length - 1;
    while (left <= right) {
        int mid = (left + right) / 2; //find midpoint
        if (key == list[mid]) {
            return mid; // found the key! return the index and stop!
        }
        else if (key > list[mid]) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }
    // didn't find the key! return -1
    return -1;
}
```
Analysis of Binary Search

- Let's try to analyze the worst case:
  - Worst case is observed when...?
    - Estimated number of simple statements in the worst case?
  - Intuition: You have $n$ elements in the list
    - The list size shrinks by half after each comparison

- Visualize the worst case for binary search (on the board)
Analysis of Binary Search

- Binary search is $O(\log_2 n)$ or, simply $O(\log n)$
  - Computer scientists usually assume base of logs is 2
  - Overtakes linear search as "fastest" search algorithm!
    - With a precondition

- Logarithmic-time algorithms are an important class of runtimes
  - Exhibited when you can "half" the problem size with each iteration

- *But we can do even better than log-time... (rarely)*
Better Than $O(\log n)$

- What about these types of algorithms:

```java
public static void doTask(int n) {
    for (int i = 0; i < 1000000; i++) {
        System.out.println(n);
    }
}

public static boolean isOdd(int x) {
    int remainder = x % 2;
    return (remainder == 1);
}
```

- Note: the number of simple statements is independent of input size
  - When this is the case, the algorithm is said to be "Constant-Time"
  - Big-O notation is: $O(1)$
Finally, a Classification Scheme

- Important running-time classifications

<table>
<thead>
<tr>
<th>$O(f(n))$</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant-time</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>Logarithmic-time</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear-time</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>Log-linear-time</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic-time</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>Cubic-time</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>Exponential-time</td>
</tr>
<tr>
<td>$O(n!)$</td>
<td>Factorial-time</td>
</tr>
</tbody>
</table>
## Summary of Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>The input size to some algorithm. It is assumed that $n \geq 0$.</td>
</tr>
<tr>
<td>$T(n)$</td>
<td>The number of statements an algorithm takes as a function of the number of inputs, $n$.</td>
</tr>
<tr>
<td>$f(n)$</td>
<td>Any function of $n$. Generally, $f(n)$ represents a simpler function than $T(n)$.</td>
</tr>
<tr>
<td>$O(f(n))$</td>
<td>The set of functions that grow no faster than $f(n)$. We say $T(n) = O(f(n))$ to indicate that the growth of $T(n)$ is bounded by the growth of $f(n)$ by a constant factor.</td>
</tr>
</tbody>
</table>
Outline

- Modeling Running Time of Algorithm
- Formalizing Complexity: The Big-O Notation
- In-Class Exercises
- A Potential Pitfall
- Conclusion
Potential Pitfall

Consider the following code. We can reasonably claim $T(n) = 1 + 2n$

```java
public static void multi_print(int n) {
    int i = 0;
    while (i < n) {
        System.out.println(n);
        i++;
    }
}
```

Okay, it's clearly $O(n)$, but....

- Would it be **wrong** to say `multi_print(..)` is $O(n^3)$? $O(n^2)$?
  - By definition of Big-O, no! They both provide upper bound to $T(n) = 1 + 2n$
  - But $O(n)$ is more precise!
Tightness of Bounds

- There is an *expectation* that we're only interested in a *tight bound*.
  - After all, we don't want to *undersell* an algorithm's efficiency
  - Example:
    - It is actually okay for us to say things like,
      - Binary search is $O(n^2)$
        » It is well-known to be tightly-bound by $O(\log_2 n)$
      - Linear search is $O(n)$
        » This is really the tightest bound on linear search
    - But, it is unsound to claim that linear search has a lower complexity than binary search! *Need to compare tight bounds.*
Outline

- Modeling Running Time of Algorithm
- Formalizing Complexity: The Big-O Notation
- In-Class Exercises
- A Potential Pitfall
- Conclusion
The Big-O notation is used to approximate and classify algorithms' use of resources.

- These resources can be time, space, network usage, etc.
  - Specifically, "time complexity" is measured as number of simple statements
- For instance, the "space complexity" of silly() is $O(1)$
  - It only introduces a constant number of variables

Why Big-O?

- Want a simple and intuitive way to compare algorithms
- Assume alg1 and alg2 solve the same problem:
  - We can claim alg2 is better than alg1 because alg1 runs in $O(n)$ and alg2 runs in $O(\log n)$
We generally only care about worst-case and average-case analyses

In practice:

- Polynomial class (and below) algorithms are desirable.
  - Constant-time algorithms are ideal, but few exist that are very useful
- Exponential (and higher) class algorithms are *usually* unacceptable!
  - Okay, when *might* they be acceptable?

Be able to: Give the tight upper-bound (in Big-O) of any given algorithm

- Except recursive ones, for now
Take a look at the following code, which adds all elements in the array:

```java
int sum = 0;
for(int i = 0; i < n; i++) {
    sum = sum + a[i];
}
```

- Is the **simple statement in the loop** constant-time?
  - Is `+` an $O(1)$ operation?
  - Are array references an $O(1)$ operation?
    - What if we assume elements at larger indices take longer to reference?
    - What if we assume all elements have constant-time reference?

- But arrays aren't the only kind of lists... *(next lecture)*
Reminders:
- Exam 1 next Tuesday 10/8
- Hwk 3 due next Wed 10/9

Last time... finished up on OOP topics
- Now seen 37/51 (71%) Java keywords

Today: Big-O notation
- Reading: K&W Chap 2.1

abstract  assert  boolean  break
byte    case    catch    char
class   const   continue default
do      double   else      enum
extends final   finally  float
for      goto    if       implements
import  instanceof int    interface
long    native   new      package
private protected public  return
short   static   strictfp super
switch  synchronized this    throw
throws  transient  try      void
volatile while    true     false
goto and const are reserved, but don't have usage.

- Now seen 37/51 (71%) keywords
- Now seen 37/49 (76%) keywords

Reminders:
- Exam 1 next Tuesday 10/8
- Hwk 3 due next Wed 10/9

Lab 5 post-mortem
- Questions about try/catch/throw?
It'sssssssssssssssssssssssssssssss Timeeeeeeeeeeeeeeeeee for the main event!!!!!!!

Starters

- 100000 matches played
  - 21 games per match to declare winner of match
- win = 2, draw = 1, loss = 0

Winners receive:

- + 5pts bonus applied to Exam I
  - (Yes, you can go over 100)
Reminders:

- Exam 1 next Tuesday 10/8
  - CWLT tutor (Lia) has copy of my review guide
- Hwk 3 due next Wed 10/9

Last time...

- Big-O notation for classifying growth of runtimes
- We've seen complexity classes: O(n), O(n^2)

Today:

- More complexity classes: O(log n), O(1)
- Binary search