BEFORE WE START

If the input to a function call on level \( i \) is \((\frac{n}{3^i})\), and we have this recurrence, what level \( i \) is the base case?

\[
T(n) = \begin{cases} 
4 & \text{if } n \leq 1 \\
T\left(\frac{n}{3}\right) + n & \text{otherwise}
\end{cases}
\]
Announcements

• EX1 (Algo Analysis I) due TONIGHT 11:59pm PDT
  - You can use late days on exercises, just like projects!

• P2 (Maps) and EX2 (Algo Analysis II) released today

• Don’t forget to fill out the P2 Partner Form!
  - Even if you want the default, please confirm for us by filling it out!
    - https://courses.cs.washington.edu/courses/cse373/tools/20su/partner/p2/

• Summations Reference published (on course calendar under Wednesday’s lecture)
P2: Maps

• Implement everyone’s good pal: the Hash Map!
• Like P1, look at multiple data structures under a single ADT
  - But this time, we have the algorithmic analysis tools to reason about more complicated situations (especially Case Analysis!)
• 3 Parts:
  - ArrayMap
  - ChainedHashMap
  - Experiments
• Start early! In particular, the ChainedHashMap iterator can take a long time!
STUDENT FEEDBACK

• THANK YOU for letting us know how optional review questions could be more helpful for you! Don’t stop here: your feedback & ideas are how we make this the best course it can be!

• Post-Lecture Optional Review Questions:
  - New We’ll publish solutions at the same time as problems. Use however you prefer!
  - New Reflection: what’s one conception you cleared up, or one question you still have?
  - Extra credit: No points, but doing lots can round up your GPA 0.1 (completion only, not graded on correctness)
  - No deadline: Complete anytime during the quarter. Recommendation: before next lecture
Announcements

• Regarding the fall F-1 online classes visa situation:
  - “The Allen School stands with our international students and is *vehemently opposed* to the planned visa changes that would upend lives and put people at risk during a pandemic. This action goes against our values as a school, a campus community, and a nation, and it should not stand. I want you all to know that school leadership, the University of Washington, and the broader higher education and computing communities are doing everything within our power to try to *prevent these changes* from taking effect.” – Magdalena Balazinska (Director, Paul G. Allen School)

• We know this is a stressful time, and you may need flexibility to work on things that aren’t this class
  - Effective immediately, we’re giving everyone two extra late days
    - Apply to P1, EX1, whatever you need. Everyone now has 9 for the quarter.
  - P1 and EX1 late cutoffs are now *5 days after the due date*
  - Next week, we’ll offer increased OH coverage and 1:1 meetings availability

• These changes are designed to give flexibility, but we know it’s not a one-size-fits-all situation. Please reach out if you would benefit from further accommodations – this class should not be a burden as you handle more important things.
Welcome to the Data Structures Part™

• We’re now armed with a toolbox stuffed full of analysis tools
  - Wednesday was the last algorithmic analysis lecture
  - It’s time to apply this theory to more practical topics!

• Today, we’ll take our first deep dive using those tools on a data structure: Hash Maps!
Learning Objectives

After this lecture, you should be able to...

1. Compare the relative pros/cons of various Map implementations, especially given a design like the ones we cover today

2. Trace operations in a Separate Chaining Hash Map on paper (such as insertion, getting an element, resizing)

3. Implement a Separate Chaining Hash Map in code (P2)

4. Differentiate between the “worst” and “in practice” runtimes of a Separate Chaining Hash Map, and describe what assumptions allow us to consider the “in practice” case
Lecture Outline

1. ArrayMap
   - FASTER: Jump directly to element, only int keys

2. DirectAccessMap
   - MORE FLEXIBLE: Hash function supports any type of key

3. SimpleHashMap
   - YOUR BEST FRIEND: Addresses limitations with hash collisions, but still fast!

4. SeparateChaining HashMap

MAP ADT

Review

As seen on Project 2
Lecture Outline

MAP ADT

1. ArrayMap
   - DirectAccessMap
     - **FASTER**: Jump directly to element, only int keys

2. SimpleHashMap
   - **MORE FLEXIBLE**: Hash function supports any type of key

3. SeparateChaining HashMap
   - **YOUR BEST FRIEND**: Addresses limitations with hash collisions, but still fast!

Review
Review The Map ADT

- **Map**: an ADT representing a set of distinct keys and a collection of values, where each key is associated with one value.
  - Also known as a **dictionary**
  - If a key is already associated with something, calling `put(key, value)` replaces the old value

- **Used all over the place**
  - It’s hard to work on a big project without needing one sooner or later
  - CSE 143 introduced:
    - `Map<String, Integer> map1 = new HashMap<>();`
    - `Map<String, String> map2 = new TreeMap<>();`

---

**MAP ADT**

<table>
<thead>
<tr>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of keys, Collection of values</td>
<td>put(key, value) add value to collection, associated with key</td>
</tr>
<tr>
<td>Count of keys</td>
<td>get(key) return value associated with key</td>
</tr>
<tr>
<td></td>
<td>containsKey(key) return if key is associated</td>
</tr>
<tr>
<td></td>
<td>remove(key) remove key and associated value</td>
</tr>
<tr>
<td></td>
<td>size() return count</td>
</tr>
<tr>
<td></td>
<td>clear() remove all</td>
</tr>
<tr>
<td></td>
<td>iterator() get an iterator</td>
</tr>
</tbody>
</table>
**Review** Implementing a Map with an Array

### MAP ADT

**State**
- Set of keys, Collection of values
- Count of keys

**Behavior**
- `put(key, value)` add value to collection, associated with key
- `get(key)` return value associated with key
- `containsKey(key)` return if key is associated
- `remove(key)` remove key and associated value
- `size()` return count

### ArrayMap<K, V>

**State**
- `Pair<K, V>[]` data

**Behavior**
- `put(key, value)` find key, overwrite value if there. Otherwise create new pair, add to next available spot, grow array if necessary
- `get(key)` scan all pairs looking for given key, return associated item if found
- `containsKey(key)` scan all pairs, return if key is found
- `remove(key)` scan all pairs, replace pair to be removed with last pair in collection
- `size()` return count of items in dictionary

### Big-Oh Analysis – (if key is the last one looked at / not in the dictionary)

- `put()` O(n) linear
- `get()` O(n) linear
- `containsKey()` O(n) linear
- `remove()` O(n) linear
- `size()` O(1) constant

### Big-Oh Analysis – (if the key is the first one looked at)

- `put()` O(1) constant
- `get()` O(1) constant
- `containsKey()` O(1) constant
- `remove()` O(1) constant
- `size()` O(1) constant

---

**Example**

```
put('b', 97)
put('e', 20)
```

```
('a', 1)  ('b', 97)  ('c', 3)  ('d', 4)  ('e', 20)
```

---

**Map ADT State**
- Set of keys, Collection of values
- Count of keys

**Map ADT Behavior**
- `put(key, value)` add value to collection, associated with key
- `get(key)` return value associated with key
- `containsKey(key)` return if key is associated
- `remove(key)` remove key and associated value
- `size()` return count

**ArrayMap<K, V> State**
- `Pair<K, V>[]` data

**ArrayMap<K, V> Behavior**
- `put(key, value)` find key, overwrite value if there. Otherwise create new pair, add to next available spot, grow array if necessary
- `get(key)` scan all pairs looking for given key, return associated item if found
- `containsKey(key)` scan all pairs, return if key is found
- `remove(key)` scan all pairs, replace pair to be removed with last pair in collection
- `size()` return count of items in dictionary
**Review Implementing a Map with Linked Nodes**

### MAP ADT

**State**
- Set of keys, Collection of values
- Count of keys

**Behavior**
- `put(key, value)`: add value to collection, associated with key
- `get(key)`: return value associated with key
- `containsKey(key)`: return if key is associated
- `remove(key)`: remove key and associated value
- `size()`: return count

### LinkedMap<K, V>

**State**
- `front`
- `size`

**Behavior**
- `put(key, value)`: if key is unused, create new with pair, add to front of list, else replace with new value
- `get(key)`: scan all pairs looking for given key, return associated item if found
- `containsKey(key)`: scan all pairs, return if key is found
- `remove(key)`: scan all pairs, skip pair to be removed
- `size()`: return count of items in dictionary

** contieneKeys(char 'c')
- get('d')
- put('b', 20)

### Big O Analysis

- **if key is the last one looked at / not in the dictionary**
  - `put()`: $O(n)$ linear
  - `get()`: $O(n)$ linear
  - `containsKey()`: $O(n)$ linear
  - `remove()`: $O(n)$ linear
  - `size()`: $O(n)$ linear

- **if the key is the first one looked at**
  - `put()`: $O(1)$ constant
  - `get()`: $O(1)$ constant
  - `containsKey()`: $O(1)$ constant
  - `remove()`: $O(1)$ constant
  - `size()`: $O(1)$ constant
Could we do better?

• **put, get, and remove** have \( \Theta(n) \) runtimes. Could we use a \( \Theta(1) \) operation to improve?

• What about array indexing?
  - data[i] (array access) and data[i] = 2 (array update) are constant runtime!
  - What if we could jump directly to the requested key?
  - We could simplify the problem: **only allow integer keys**
Lecture Outline

1. ArrayMap
   - Review

2. DirectAccessMap
   - FASTER: Jump directly to element, only int keys

3. SimpleHashMap
   - MORE FLEXIBLE: Hash function supports any type of key

4. SeparateChaining HashMap
   - YOUR BEST FRIEND: Addresses limitations with hash collisions, but still fast!
DirectAccessMap

• put, get, and remove have $\Theta(n)$ runtimes. Could we use a $\Theta(1)$ operation to improve?

• What about array indexing?
  - data[i] (array access) and data[i] = 2 (array update) are constant runtime!
  - What if we could jump directly to the requested key?
  - We could simplify the problem: only allow integer keys

put(3, “Melissa”)
get(3)
DirectAccessMap Implementation

```java
public void put(int key, V value) {
    this.array[key] = value;
}

public boolean containsKey(int key) {
    return this.array[key] != null;
}

public V get(int key) {
    return this.array[key];
}

public void remove(int key) {
    this.array[key] = null;
}
```

### DirectAccessMap<K, V>

**State**
- data[]
- size

**Behavior**
- **put** put item at given index
- **get** get item at given index
- **containsKey** if data[] null at index, return false, return true otherwise
- **remove** nullify element at index
- **size** return count of items in dictionary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Case</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, value)</td>
<td>best</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>get(key)</td>
<td>best</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>containsKey(key)</td>
<td>best</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>
Pros and Cons of DirectAccessMap

What’s a benefit of using it? What’s a drawback?

Pros and Cons of DirectAccessMap
Pros and Cons of DirectAccessMap

• Super Fast!
  - Everything is $\Theta(1)$

• Wasted Space
  - Say we want to store 0 and 999999999. This implementation would waste all
    the space inbetween 😞

• Only Integer Keys
  - Would be nice to store any type of data 😞
  - But note what’s so useful here: being able to go quickly from key to array
    index
Can We Store Any Integer?

IDEA 1
• Create a GIANT array with every possible integer as an index
• Problems:
  - Can we allocate an array big enough?
  - Super wasteful

IDEA 2
• Create a smaller array, with a translation from integer keys into available indices
• Problems:
  - How can we construct a translation?
Hash Functions

- **Hash Function**: any function that can be used to map data of an arbitrary size to fixed-size values.
  - We want to translate from the set of all integers to the set of valid indexes in our array

\[
\begin{align*}
504 & 1 & 9002 \\
9002 \mod 10 &= 2 \quad \text{(so store it in index 2 of the array)}
\end{align*}
\]

- One simple approach: take the key and \( \mod \) (mod) it by size of the array
Lecture Outline

1. ArrayMap
   - Review
2. DirectAccessMap
   - FASTER: Jump directly to element, only int keys
3. SimpleHashMap
   - MORE FLEXIBLE: Hash function supports any type of key
4. SeparateChaining HashMap
   - YOUR BEST FRIEND: Addresses limitations with hash collisions, but still fast!
Mod: Remainder

• The \( \% \) operator computes the remainder from integer division.

\[
\begin{align*}
14 \% 4 & \text{ is } 2 \\
3 & \\
4 & \text{) 14} \\
12 & \\
2 & \\
218 \% 5 & \text{ is } 3 \\
43 & \\
5 & \text{) 218} \\
20 & \\
18 & \\
15 & \\
3 & 
\end{align*}
\]

Equivalently, to find \( a \% b \) (for \( a, b > 0 \)):

\[
\text{while}(a > b-1) \\
a -= b; \\
\text{return } a;
\]

14 \% 4 is 2

• Applications of \( \% \) operator:

- Obtain last digit of a number: \( 230857 \% 10 \) is 7
- See whether a number is odd: \( 7 \% 2 \) is 1, \( 42 \% 2 \) is 0
- Limit integers to specific range: \( 8 \% 12 \) is 8, \( 18 \% 12 \) is 6

For more review/practice, check out https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic

Limit keys to indices within array
SimpleHashMap: “% by size” as Hash Function

| put(0, “I”)       | 0 % 10 = 0 |
| put(8, “Maps”)    | 8 % 10 = 8 |
| put(11, “<3”)     | 11 % 10 = 1 |
| put(23, “Hash”)   | 23 % 10 = 3 |

index: 0 1 2 3 4 5 6 7 8 9

data: I <3 Hash Maps

```java
public void put(int key, int value) {
    data[hashToValidIndex(key)] = value;
}

public V get(int key) {
    return data[hashToValidIndex(key)];
}

public int hashToValidIndex(int k) {
    return k % this.data.length;
}
```

IMPLEMENTATION
SimpleHashMap: Collisions?!

### IMPLEMENTATION

```java
public void put(int key, int value) {
    data[hashToValidIndex(key)] = value;
}

public V get(int key) {
    return data[hashToValidIndex(key)];
}

public int hashToValidIndex(int k) {
    return k % this.data.length;
}
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>I</td>
<td>&lt;3</td>
<td>Hash</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Maps</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Example

- `put(0, "I")`  
  \[0 \mod 10 = 0\]
- `put(8, "Maps")`  
  \[8 \mod 10 = 8\]
- `put(11, "<3")`  
  \[11 \mod 10 = 1\]
- `put(23, "Hash")`  
  \[23 \mod 10 = 3\]
- `put(20, "We")`  
  \[20 \mod 10 = 0\]
Lecture Outline

1. ArrayMap
   - FASTER: Jump directly to element, only int keys

2. DirectAccessMap

3. SimpleHashMap
   - MORE FLEXIBLE: Hash function supports any type of key

4. SeparateChaining HashMap
   - YOUR BEST FRIEND: Addresses limitations with hash collisions, but still fast!
Handling Collisions

• Two common strategies to handle collisions:

1. Separate Chaining
   "Chain" together multiple values stored in a single bucket

2. Open Addressing
   If a bucket is taken, find a new bucket using some strategy:
   Linear Probing
   Quadratic Probing
   Double Hashing

We’ll focus on separate chaining this quarter, much more common in practice

Bonus topic beyond the scope of the class
Separate Chaining

• If two values want to live in the same index, let’s just let them be roommates!

• Each index is a “bucket”
  - Linked Nodes are a common implementation for these bucket “chains”

• When item x hashes to index h:
  - If bucket at h is empty, create new list with x
  - Else, add x to the list
Separate Chaining

• If two values want to live in the same index, let’s just let them be roommates!

• Each index is a “bucket”
  - Linked Nodes are a common implementation for these bucket “chains”

• When item x hashes to index h:
  - If bucket at h is empty, create new list with x
  - Else, add x to the list

• But if multiple keys can hash to the same index, need to store the key too!
Separate Chaining

• Implementation of get/put/containsKey very similar

PSEUDOCODE

```java
public boolean get(int key) {
    int bucketIndex = key % data.length;
    for (int i = 0; i < data.length; i++) {
        if (data[i] != null && data[i].key == key) {
            return data[i].value;
        }
    }
    return null;
}
```

Let’s analyze the runtime. First, are there different possible states for this HashMap to make the code faster or slower, assuming n key/value pairs are already stored?
Separate Chaining Worst Case

- It’s possible that everything hashes to the same bucket by chance!
  - get would take $\Theta(n)$ time 😞

- Consider get(51)
  - Use hash function ($\% 10$) to get index (5)
  - Check every element in bucket for key 51

- We’ve lost that $\Theta(1)$ runtime

PSEUDOCODE

```java
public boolean get(int key) {
    int bucketIndex = key % data.length;
    for (Pair pair : data[bucketIndex]) {
        if (pair.key == key) {
            return pair.value;
        }
    }
    return null;  // if we get here
}
```
Separate Chaining Best Case

• However, if everything is spread evenly across the buckets, get takes $\Theta(1)$

• Consider $\text{get}(22)$
  - Use hash function ($\% 10$) to get index (2)
  - Check the single element in bucket for key 22 – a constant time operation!

• Key to a successful Hash Map implementation: how can we keep the buckets as close to this distribution as possible?
Separate Chaining... In Practice

• A well-implemented separate chaining hash map will stay very close to the best case
  - Most of the time, operations are fast. Rarely, do an expensive operation that restores the map close to best case.

• How to stay close to best case?
  - Good distribution & Resizing!

• We can describe the “in-practice” case as what almost always happens:
  - (1) items are fairly evenly distributed
  - (2) assume resizing doesn’t occur
    - This is similar to the concept of “amortized”

<table>
<thead>
<tr>
<th>Operation</th>
<th>Case</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, value)</td>
<td>best</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td></td>
<td>In-practice</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>get(key)</td>
<td>In-practice</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>remove(key)</td>
<td>best</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td></td>
<td>In-practice</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>
Resizing

- The runtime to scan each bucket is creeping up
  - If we don’t intervene, our in-practice runtime is going to hit $\Theta(n)$
    - number of buckets is a constant, so $n / (# \text{ buckets})$ is $\Theta(n)$
Resizing

Don’t forget to re-distribute your keys!

How to Resize:
1. Expand the buckets array
2. For every element in the old hash table, re-distribute!
   Recompute its position by taking the mod with the new length

If we just expand the buckets array, several values are hashed in the wrong place

As seen on Project 2
When to Resize?

• In ArrayList, we were forced to resize when we ran out of room
  - In SeparateChainingHashMap, never *forced* to resize, but we want to make sure the buckets don’t get too long for good runtime

• How do we quantify “too full”?  
  - Look at the average bucket size: number of elements / number of buckets

\[
\lambda = \frac{n}{c}
\]

<table>
<thead>
<tr>
<th>LOAD FACTOR $\lambda$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$: total number of key/value pairs</td>
</tr>
<tr>
<td>$c$: capacity of the array (# of buckets)</td>
</tr>
</tbody>
</table>

| $\lambda$ = 1.6 |

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,red)</td>
<td>(6,pink)</td>
<td>(22,tan)</td>
<td>(7,blue)</td>
<td>(77,aqua)</td>
</tr>
<tr>
<td>(8,lilac)</td>
<td>(53,puce)</td>
<td>(4,orange)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\lambda = \frac{8}{5} = 1.6$
When to Resize?

• In ArrayList, we were forced to resize when we ran out of room
  - In SeparateChainingHashMap, never forced to resize, but we want to make sure the buckets don’t get too long for good runtime

• How do we quantify “too full”?  
  - Look at the average bucket size: number of elements / number of buckets

• If we resize when $\lambda$ hits some constant value like 1:
  - We expect to see 1 element per bucket: constant runtime!
  - If we double the capacity each time, the expensive resize operation becomes less and less frequent

LOAD FACTOR $\lambda$

$n$: total number of key/value pairs  
$c$: capacity of the array (# of buckets)

$$\lambda = \frac{n}{c}$$
Hashing

• What about non-integer data?
  - Remember the definition -- **Hash Function**: any function that can be used to map data of an arbitrary size to fixed-size values.

  ![Hash Function Example]

• Considerations for Hash Functions:
  1. **Deterministic** – same input should generate the same output
  2. **Efficient** – reasonable runtime
  3. **Uniform** – inputs spread “evenly” across output range
Hashing

**Implementation 1: Simple aspect of values**
public int hashCode(String input) {
    return input.length();
}

**Implementation 2: More aspects of value**
public int hashCode(String input) {
    int output = 0;
    for(char c : input) {
        out += (int)c;
    }
    return output;
}

**Implementation 3: Multiple aspects of value + math!**
public int hashCode(String input) {
    int output = 1;
    for (char c : input) {
        int nextPrime = getNextPrime();
        out *= Math.pow(nextPrime, (int)c);
    }
    return Math.pow(nextPrime, input.length());
}

**Pro:** super fast  
**Con:** lots of collisions!

**Pro:** still really fast  
**Con:** some collisions

**Pro:** few collisions  
**Con:** slower, gigantic integers
Hashing

• Fortunately, experts have made most of these design decisions for us!
  - All objects in Java have a `hashCode()` method that does some magic to make a “good” hash for any object type (e.g. String, ArrayList, Scanner)
  - The built-in `hashCode()` has a good distribution/not a lot of collisions

• More precisely, `hashCode()` just gets us an int representation: *then* we % by size

1. call `key.hashCode()` to get int representation of object

2. Mod (%) by the number of buckets to get our index

- "Melissa"
- "Joyce"
- "Howard"
Review Iterators

• **Iterator**: a Java interface that dictates how a collection of data should be traversed. Can only move forward and in a single pass.

<table>
<thead>
<tr>
<th>Iterator Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior</td>
</tr>
<tr>
<td>hasNext() – true if elements remain</td>
</tr>
<tr>
<td>next() – returns next element</td>
</tr>
</tbody>
</table>

**hasNext()** – returns true if the iteration has more elements yet to be examined

**next()** – returns the next element in the iteration and moves the iterator forward to next item

Two ways to use an iterator in Java:

```java
ArrayList<Integer> list;

Iterator itr = list.iterator();
while (itr.hasNext()) {
    int item = itr.next();
}
```

```java
ArrayList<Integer> list;

for (int i : list) {
    int item = i;
}
```
P2 Reminders

• Implementing an iterator for a Hash Map is complex!
  - You need to iterate through the elements of a bucket, but when you reach the end of the chain, have to move to the next bucket
  - “you’re not iterating over some linear data structure, you’re playing 2D chess”
    – Howard Xiao

• Start early! P2 available for over 1.5 weeks, but for good reason!
  - Especially the ChainedHashMap iterator

• Remember to read the entire Tips section of the instructions!