

Collections, Part One

Outline for Today

- ***Container Types***
 - Holding lots of pieces of data.
- ***The Vector type***
 - Storing sequences.
- ***Reference Parameters***
 - A key part of C++ programming.
- ***Recursion on Vectors***
 - A problem with cell towers.

Organizing Data

- From CS106A or your own experience, you're probably seen basic data structures like
 - Arrays / lists:
 - Java ArrayList, Python lists, JavaScript arrays.
 - Associative arrays / Maps:
 - Java HashMap, Python dict, JavaScript objects.
- These structures are the bread and butter of programming.

Container Types

- A ***collection class*** (or ***container class***) is a data type used to store and organize data in some form.
- These are types like Java's ArrayList and HashMap, Python's list and dict, and JavaScript's array and object.
- Our next three lectures are dedicated to exploring different collections and how to harness them appropriately.
- Later in the quarter, we'll see how these types work and analyze their efficiencies. For now, let's just focus on how to use them.

Vector

Vector

- The **Vector** is a collection class representing a list of things.
 - Similar to Java's `ArrayList` type and to arrays in JavaScript and Python.
- Syntax is pretty friendly:
 - Read/write an element: **`vec[index]`**
 - Append elements: **`vec += a, b, c, d;`**
 - Remove elements: **`vec.remove(index);`**
 - Check the size: **`vec.size()`**

Range-Based For Loops

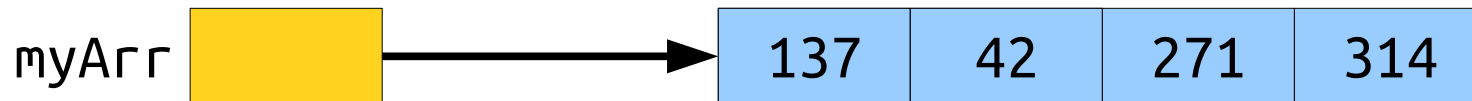
- In C++, you can iterate over all the items of a container (string, Vector, etc.) using this syntax:

```
for (type var: container) {  
    // do something with var  
}
```

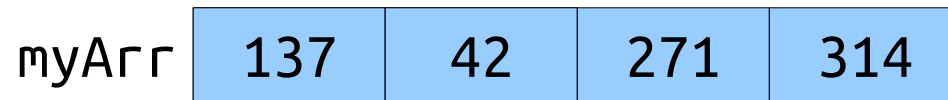
- This visits each element of the container in sequence. At each iteration, **var** represents the currently-visited element.

Objects in C++

- In most programming languages, object variables are *references*.
- The variable isn't the object; it just says where to look for that object.



- C++ is different. In C++, object variables *literally are* the objects.



- While C++ does have a **new** keyword, we won't be using it until later in the quarter.

Pass-by-Value

- In C++, objects are passed into functions by *value*. The function gets its own local copy of the argument to work with.
 - There's a cool nuance where this isn't 100% true; come talk to me after class if you're curious!
- Don't just take my word for it - watch what happens!

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
  
    growUp(moonlight);  
  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
void growUp(Vector<string> characters) {  
    characters += "Paula";  
    characters[0] = "Chiron";  
}
```

characters

"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight "Little" "Teresa" "Kevin"

```
void growUp(Vector<string> characters) {  
    characters += "Paula";  
    characters[0] = "Chiron";  
}
```

characters "Little" "Teresa" "Kevin"

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
void growUp(Vector<string> characters) {  
    characters += "Paula";  
    characters[0] = "Chiron";  
}
```

characters

"Little"	"Teresa"	"Kevin"	"Paula"
----------	----------	---------	---------

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
void growUp(Vector<string> characters) {  
    characters += "Paula";  
    characters[0] = "Chiron";  
}
```

characters

"Little"	"Teresa"	"Kevin"	"Paula"
----------	----------	---------	---------

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
void growUp(Vector<string> characters) {  
    characters += "Paula";  
    characters[0] = "Chiron";  
}
```

characters

"Chiron"	"Teresa"	"Kevin"	"Paula"
----------	----------	---------	---------


```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
void growUp(Vector<string> characters) {  
    characters += "Paula";  
    characters[0] = "Chiron";  
}
```

characters

"Chiron"	"Teresa"	"Kevin"	"Paula"
----------	----------	---------	---------

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
  
    growUp(moonlight);  
  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"
----------	----------	---------

Pass-by-Reference

- In C++, there's the option to pass parameters into function ***by reference***.
- This means that the actual argument itself gets sent into the function, not a copy of it.
- To declare a function that takes an argument by reference, put an ampersand (&) after the type of the argument.

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
  
    growUp(moonlight);  
  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

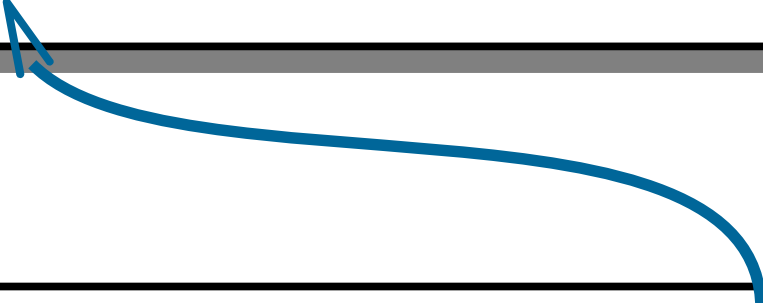
"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
void growUp(Vector<string>& characters) {  
    characters += "Paula";  
    characters[0] = "Chiron";  
}
```



```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"
----------	----------	---------

```
void growUp(Vector<string>& characters) {  
    characters += "Paula";  
    characters[0] = "Chiron";  
}
```



```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"	"Paula"
----------	----------	---------	---------

```
void growUp(Vector<string>& characters) {  
    characters += "Paula";  
    characters[0] = "Chiron";  
}
```

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

"Little"	"Teresa"	"Kevin"	"Paula"
----------	----------	---------	---------

```
void growUp(Vector<string>& characters) {  
    characters += "Paula";  
    characters[0] = "Chiron";  
}
```

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

moonlight

"Chiron"	"Teresa"	"Kevin"	"Paula"
----------	----------	---------	---------

```
void growUp(Vector<string>& characters) {  
    characters += "Paula";  
    characters[0] = "Chiron";  
}
```

```
int main() {  
    Vector<string> moonlight = { "Little", "Teresa", "Kevin" };  
    growUp(moonlight);  
    /* ... */  
}
```

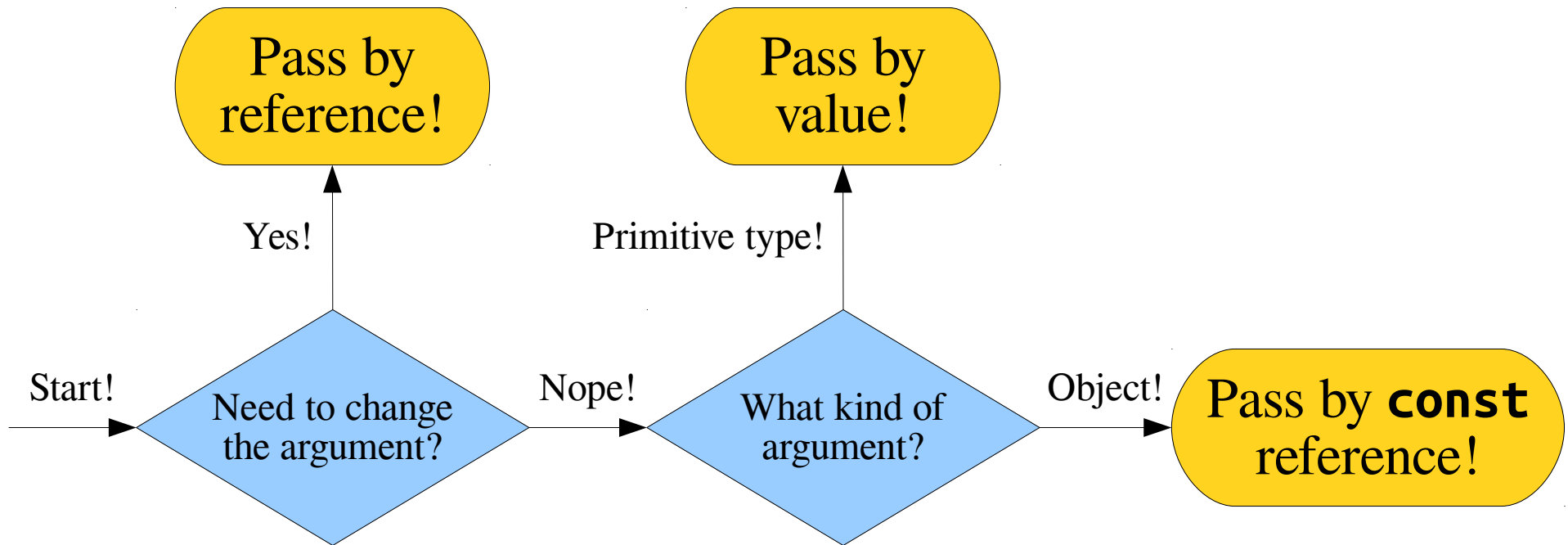
moonlight

"Chiron"	"Teresa"	"Kevin"	"Paula"
----------	----------	---------	---------

Pass-by-const-Reference

- Passing a large object into a function by value can take a *lot* of time.
- Taking parameters by reference avoids making a copy, but risks that the object gets tampered with in the process.
- As a result, it's common to have functions that take objects as parameters take their argument by ***const reference***:
 - The “by reference” part avoids a copy.
 - The “**const**” (constant) part means that the function can't change that argument.

Parameter Flowchart



This is the general convention used in C++ programming. Please feel free to ask questions about this over the course of the quarter!

Time-Out for Announcements!

Migrating to C++ Session

- We'll be holding an extra about migrating from Python or JavaScript to C++.
Details below:

Monday, January 14th
7:00PM - 8:30PM
Hewlett 102

- Feel free to stop on by!

Sections

- Discussion sections start this week!
- Forgot to sign up? The signup link will reopen on Tuesday at 5PM, and you can choose any open section time.
- This week's handout of section problems is available on the course website. Your section leader will distribute hardcopies in person.

```
return;
```

Recursion on Vectors

Thinking Recursively

```
if (The problem is very simple) {  
    Directly solve the problem.  
    Return the solution.  
} else {  
    Split the problem into one or more  
    smaller problems with the same  
    structure as the original.  
    Solve each of those smaller problems.  
    Combine the results to get the overall  
    solution.  
    Return the overall solution.  
}
```

These simple cases
are called *base*
cases.

These are the
recursive cases.

Example: Cell Tower Purchasing

Buying Cell Towers



137

106

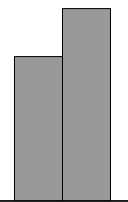
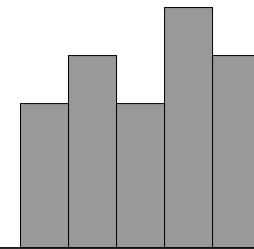
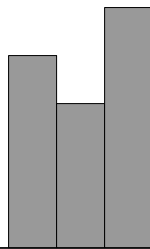
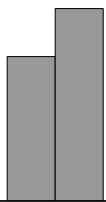
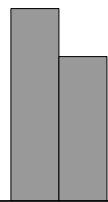
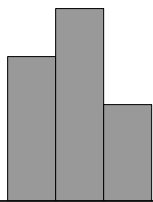
107

166

103

261

109



Buying Cell Towers



137

106

107

166

103

261

109

×

✓

×

✓

×

×

✓

Towers can't be built
in two adjacent cities.

People Covered: $106 + 166 + 109 = \mathbf{381}$.

Buying Cell Towers



137



106



107



166



103



261



109



People Covered: $137 + 107 + 103 + 109 = 456$.

Buying Cell Towers



137



106



107



166



103



261



109



People Covered: $106 + 166 + 261 = \mathbf{533}$.

Buying Cell Towers



137



106



107



166



103



261



109



People Covered: $137 + 166 + 261 = 564$.

Buying Cell Towers



99



100



99



People Covered: **100**.

Buying Cell Towers



99

100

99

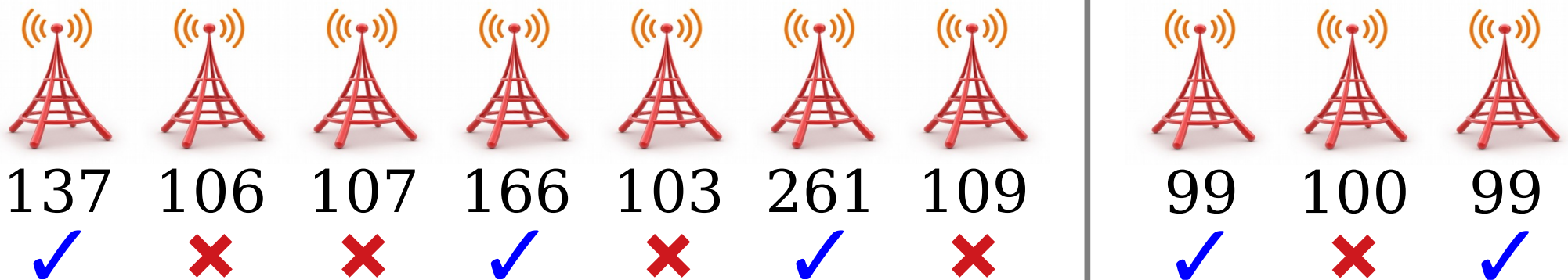


People Covered: $99 + 99 = \mathbf{198}$.

Question: Given a list of cities, what's the maximum number of people you can cover?

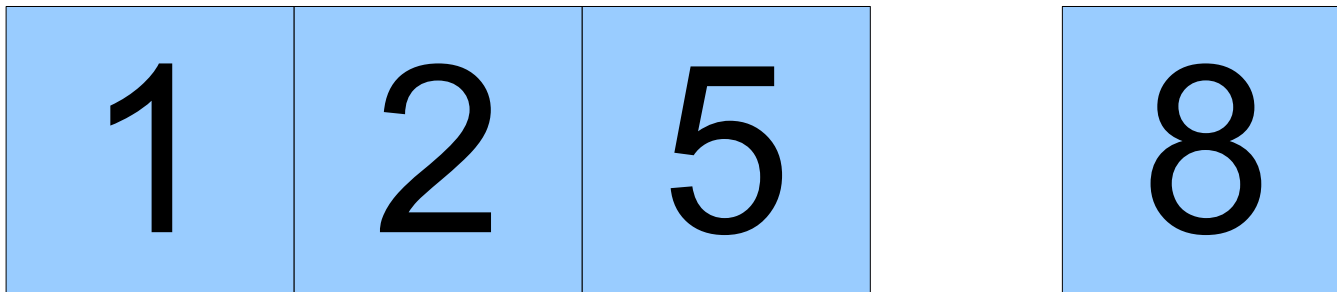
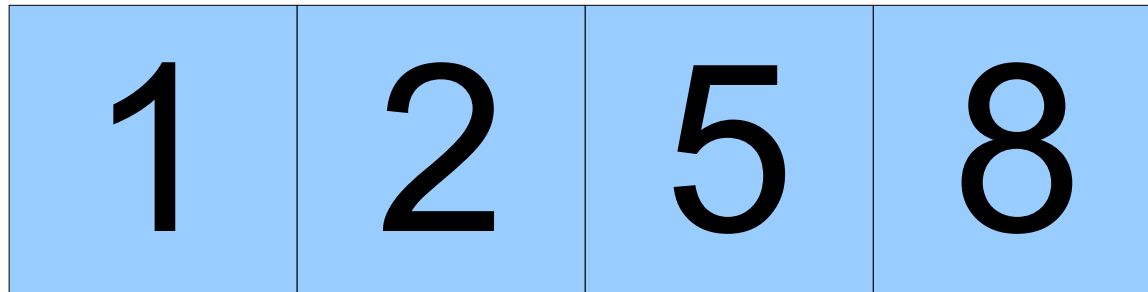
Buying Cell Towers

- Sometimes, the best option is to pick towers in an alternating pattern, but sometimes it isn't.
- Sometimes, the best option is to keep picking the most populous city and building there, but sometimes it isn't.



- How can we guarantee we always find the best solution?

Thinking Recursively



Thinking Recursively

I	B	E	X
---	---	---	---

I	B	E	X
---	---	---	---

Thinking Recursively



What are we going to do with this cell tower?



14



22



13



25



30

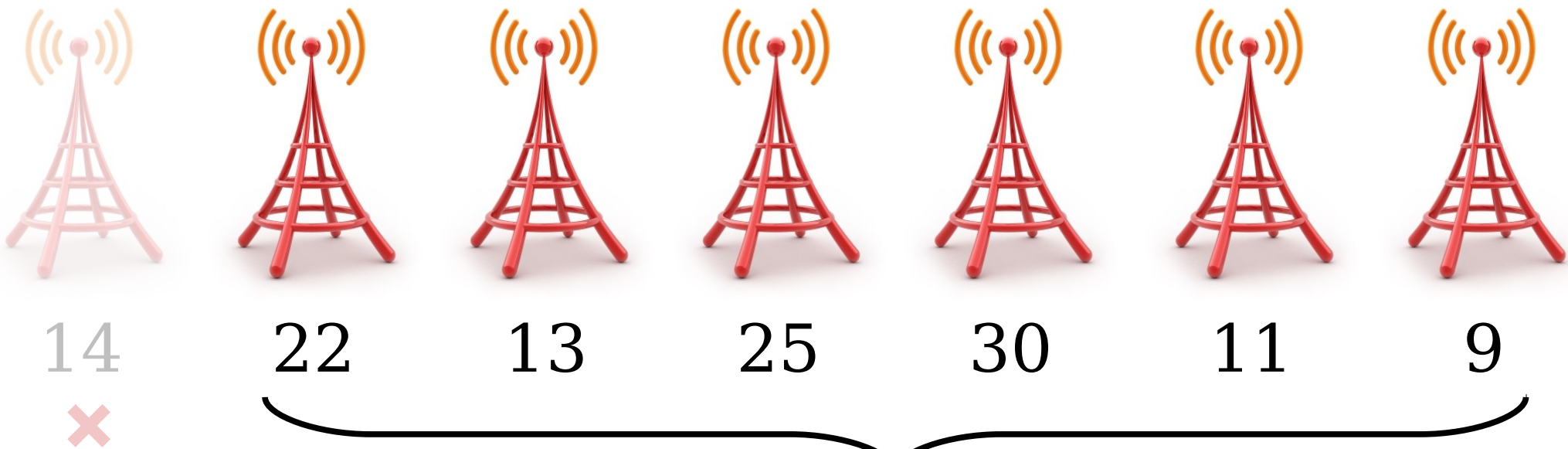


11



9

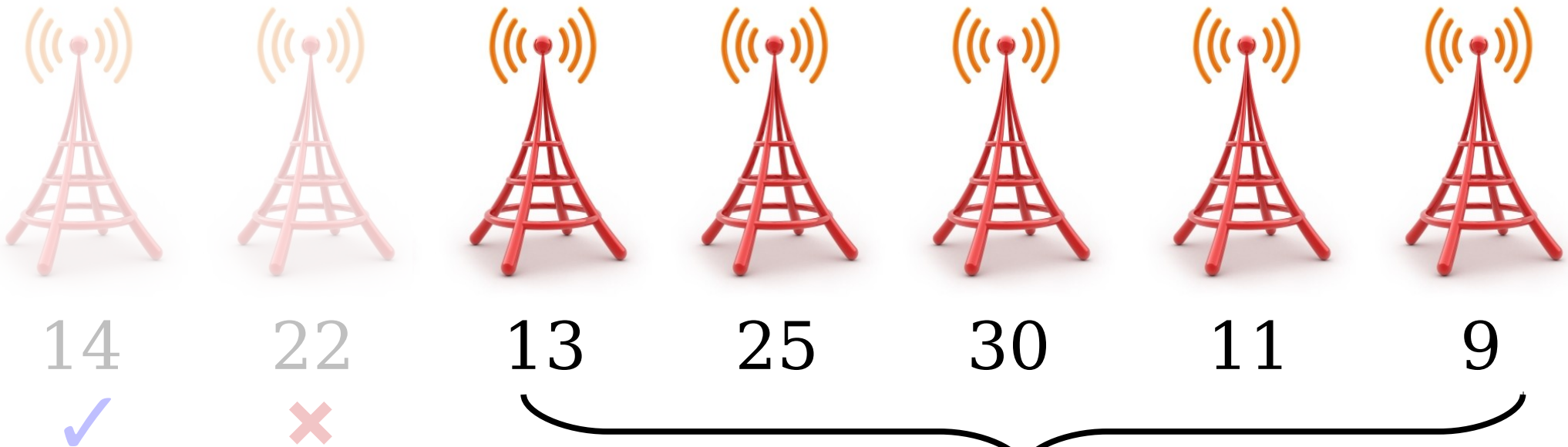
What are we going to do with this cell tower?



Option 1:
Don't purchase that tower.

Now, do whatever is best to maximize coverage to all but the first city.

What are we going to do with this cell tower?

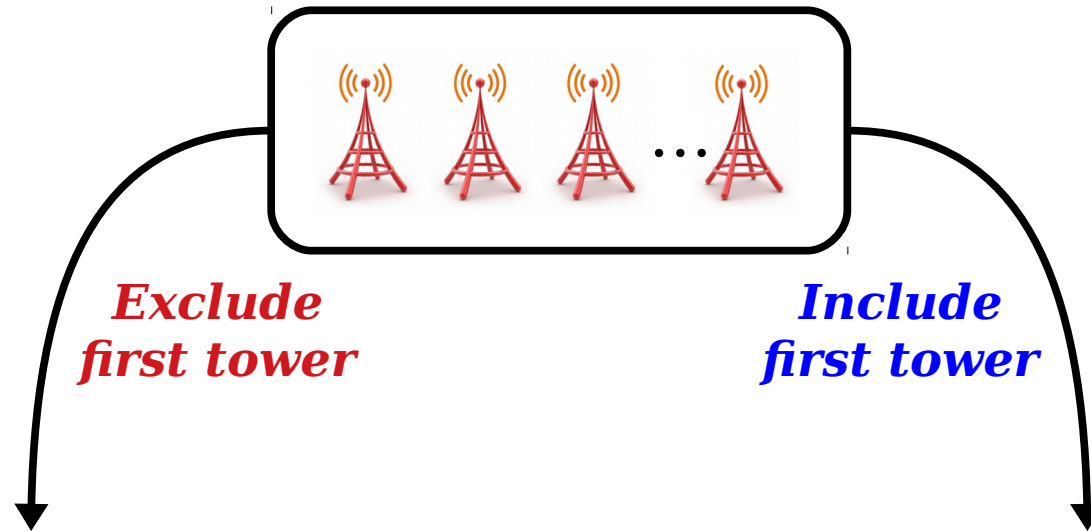


Option 2:
Purchase that cell tower.

Now, do whatever is best to maximize coverage to all but the first two cities.

A Fork in the Road

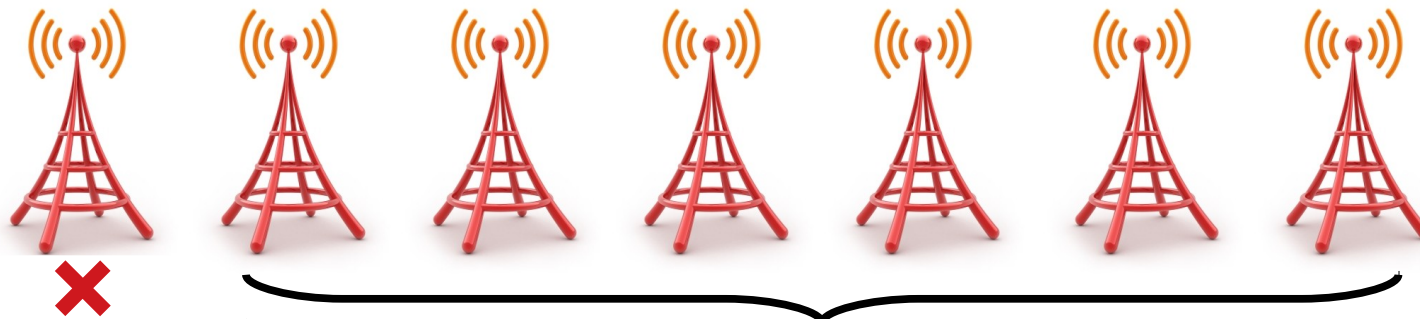
- If there is at least one cell tower, we have a choice to make about what to do with it.



- One of these options leads to the best outcome, but we can't know for sure what it is without further exploration.
- ***Key Idea:*** Try both options, and take whichever one is better.

A Recursive Solution

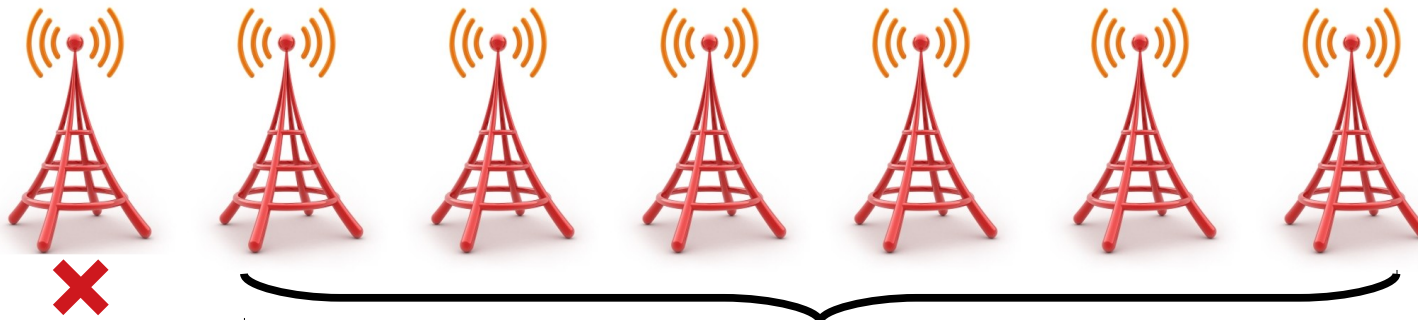
- **Base Case:** There are no cell towers.
 - How many people can you provide coverage to in this case?
- **Recursive Case:** There is at least one cell tower.
 - Option 1: Exclude the first tower.



`cities.subList(1, cities.size() - 1)`

A Recursive Solution

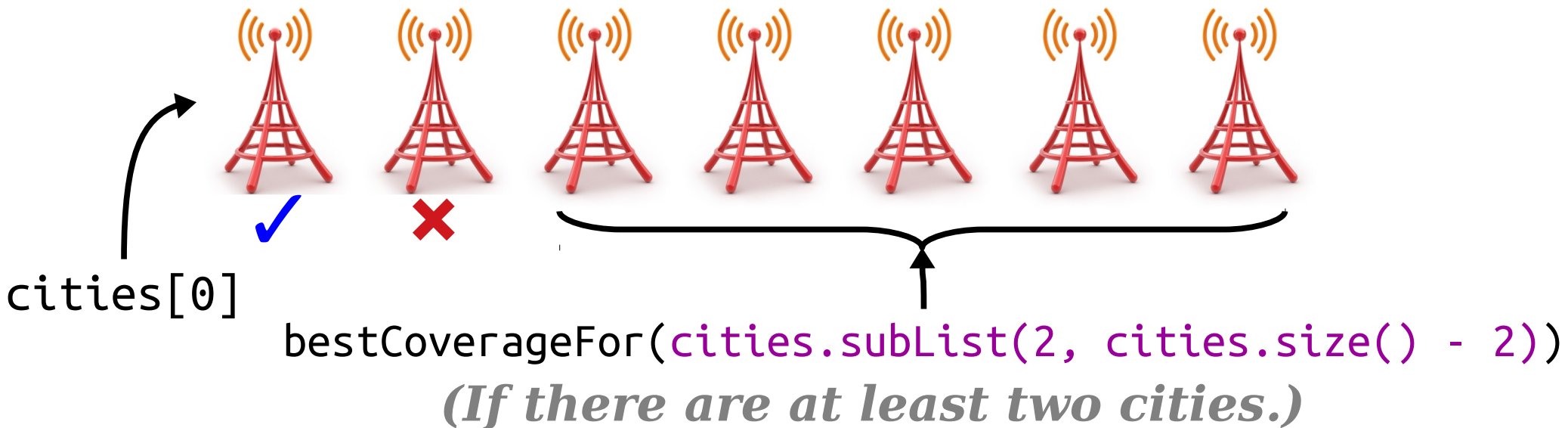
- **Base Case:** There are no cell towers.
 - How many people can you provide coverage to in this case?
- **Recursive Case:** There is at least one cell tower.
 - Option 1: Exclude the first tower.



`bestCoverageFor(cities.subList(1, cities.size() - 1))`

A Recursive Solution

- **Base Case:** There are no cell towers.
 - How many people can you provide coverage to in this case?
- **Recursive Case:** There is at least one cell tower.
 - Option 1: Exclude the first tower.
 - Option 2: Include the first tower.





137



106



107



166



103



261



109

Max coverage is **564**.



99



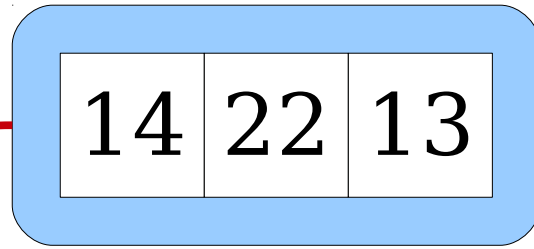
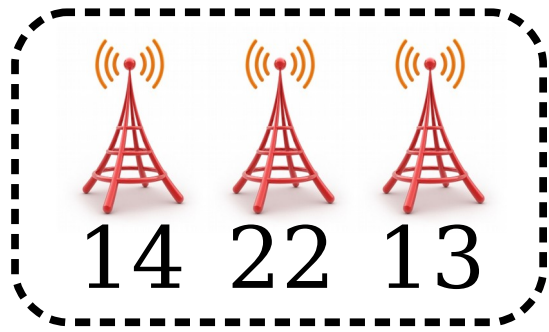
100



99

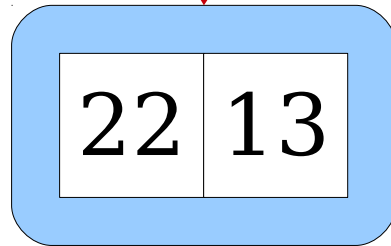
Max coverage is **198**.

How does this work?



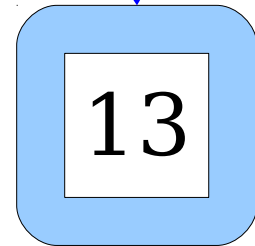
Include 14?

No

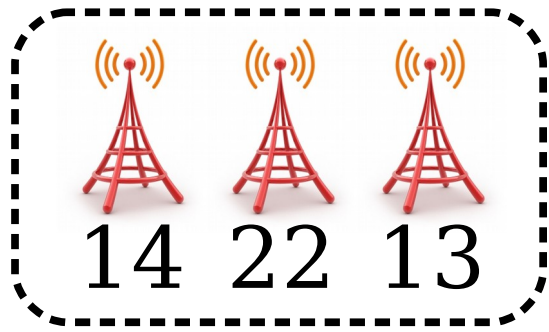


Option 1: Cover 22 people.

Yes



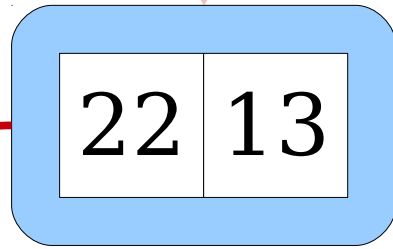
Option 2: Cover 13 people, plus the 14 people we picked earlier.



Include 14?

No

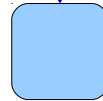
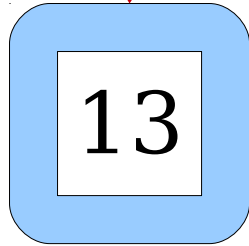
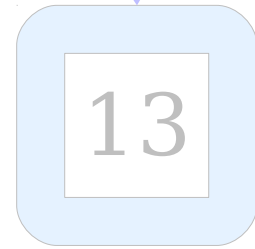
Yes



Include 22?

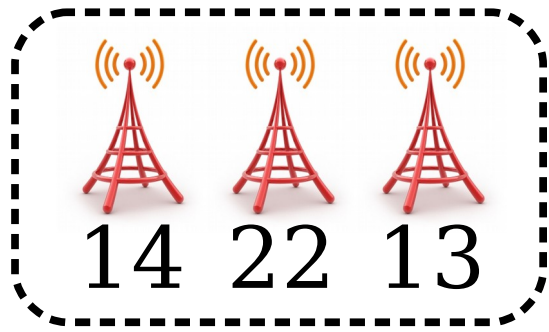
No

Yes



Option 1: Cover 13 people.

Option 2: Cover 0 people, plus the 22 people we picked earlier.



Include 14?

No

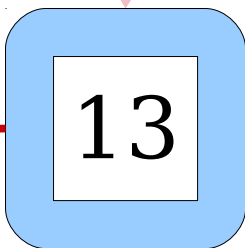
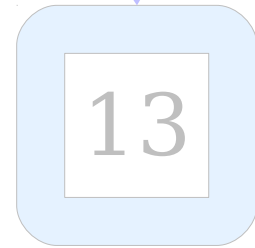
Yes



Include 22?

No

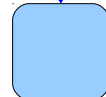
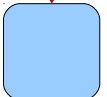
Yes



Include 13?

No

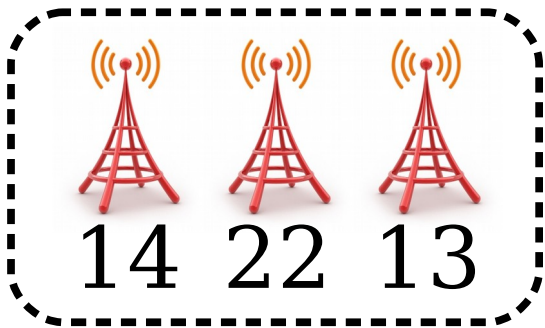
Yes



Option 2: Cover 0 people, plus the 13 from above.

Option 1:

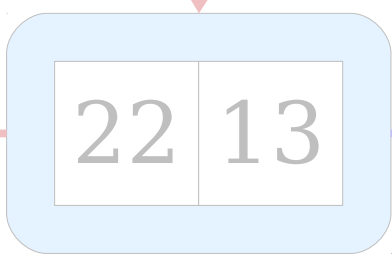
Cover 0 people.



Include 14?

No

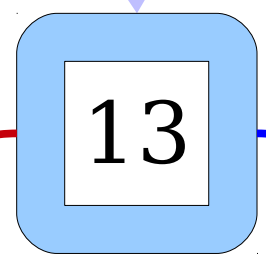
Yes



Include 22?

No

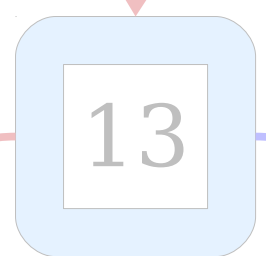
Yes



Include 13?

No

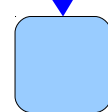
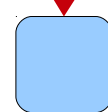
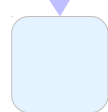
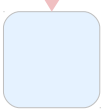
Yes



Include 13?

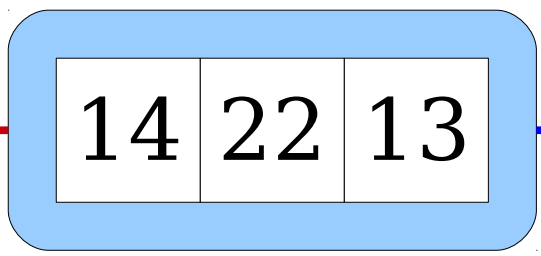
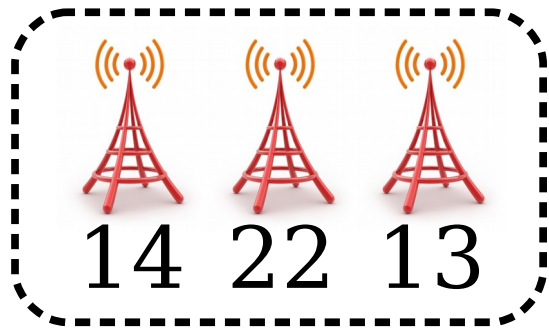
No

Yes



*Option 1:
Cover 0 people.*

*Option 2:
Cover 0
people, plus
the 13 from
above.*

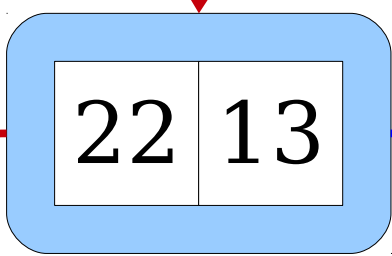


14 22 13

Include 14?

No

Yes

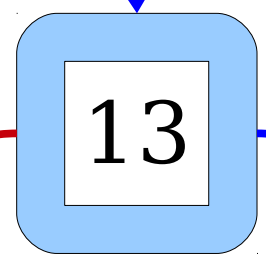


22 13

Include 22?

No

Yes

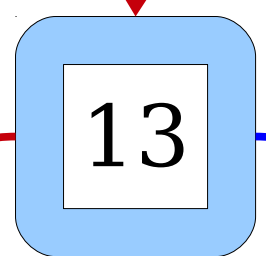


13

Include 13?

No

Yes

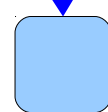
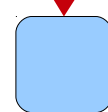
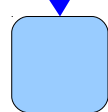
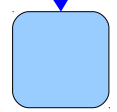
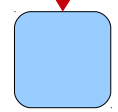


13

Include 13?

No

Yes



Wrapping Up

- If you're looking at this example and are wondering how on earth this works, no worries! That's completely normal. This way of thinking takes time to adjust to.
- Here are a few things you can do to get a better handle on how things work.
 - Draw a recursion tree for an example involving four cell towers. Follow the sort of reasoning we just did to get a sense for how this works.
 - Step through the code in the debugger. That will give you a sense of how things work in practice.
 - Tweak and change the code. Poke and prod at it to see if you see how all the pieces fit together. Form hypotheses about how things will play out differently, and then validate those hypotheses experimentally!

Your Action Items

- Start reading Chapter 5 of the textbook on the different container types.
- Work on Assignment 1.
 - Aim to complete all three recursion problems by Tuesday evening.
 - Not done by then? Don't worry! Stop by the LaIR to ask questions.
 - Start working on Flesch-Kincaid readability.
- Play around with the cell towers example. Ask questions about it! You'll learn so much more if you do.

Next Time

- ***Stacks***
 - How pancakes relate to parentheses.
- ***Queues***
 - Exploring all possible options.