

# ECS32B

Introduction to Data Structures

Sorting

Lecture 19

# Announcements

- Homework 4 due tomorrow May 14 at 11:59pm
- SLAC tonight from 6:30 - 9:00 pm Hutchison 73.
- Midterms are graded.



# Implementing a Map

- Arguably the most useful built in Python type is the **dictionary** where a **key** is used to look up an associated data **value**.
- We also refer to this idea as a **map**.
- Our textbook implements a Map ADT using a class called HashTable.

# Homework 4 Problem 3

## Improve the Map ADT

- Fix insertion, implemented by the put method, by increasing the table size when it becomes full. This is similar to the goal of maintaining a roughly constant load factor, which is key to the  $O(1)$  complexity of hashing.
- Facilitate the uniform distribution of values in the hash table to minimize collisions. Hash table sizes will be prime numbers.

# Why prime?

10	None	None	None	None	20	None	None	None	None
0	1	2	3	4	5	6	7	8	9

- We want to hash 10, 20, 30, 40, 50 into the hash table above using  **$h(\text{item}) = \text{item} \% \text{size}$**  as our hash function and linear probing with a skip value of 5 for collision resolution.
- The hash function maps all these items to the same slot.
- We can only insert 10 and 20, because only one extra slot is available for collision resolution.
- There is no place to put 30 in this table!

# Why prime?

10	None	None	None	None	20	None	None	None	None
0	1	2	3	4	5	6	7	8	9

- The problem here is the skip value evenly divides the table.
- Whenever this happens you don't have access to all the slots during the collision resolution.
- The simplest solution is to choose a **prime number** for the table size. No skip value greater than 1 will evenly divide into a prime number.

# Why prime?

- For the values 10, 20, 30, 40, 50, there are no collisions because they will now map to different slots.
- Here's 11, 22, 33, 44, 55 in the new table. These items hash to slot 0, but the skip value 5 no longer evenly divides the table.

11	None	None	None	44	22	None	None	None	55	33
0	1	2	3	4	5	6	7	8	9	10

# The HashTable a.k.a Map Abstract Datatype

**HashTable()** Create a new, empty map. It returns an empty map collection.

**put(key,val)** Add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.

**get(key)** Given a key, return the value stored in the map or None otherwise.

**del** Delete the key-value pair from the map using a statement of the form `del map[key]`.

**len()** Return the number of key-value pairs stored in the map.

**in** Return True for a statement of the form `key in map`, if the given key is in the map, False otherwise.



# HashTable Class

```
class HashTable:  
    def __init__(self):  
        self.size = 11  
        self.slots = [None] * self.size  
        self.data = [None] * self.size
```

hash table for keys



mirrored table for values

# HashTable Class

This method implements the simple hash function

```
def hashfunction(self, key, size):  
    return key % size
```



this only works for integers

# HashTable Class

This method implements simple **linear probing** with a skip value of 1 for collision resolution.

```
def rehash(self, oldhash, size):  
    return (oldhash+1)%size
```



skip value = 1

# HashTable Class


The put method implements putting a key into the hashtable

```
def put(self, key, data):  
    hashvalue = self.hashfunction(key, len(self.slots))  
  
    if self.slots[hashvalue] == None:  
        self.slots[hashvalue] = key  
        self.data[hashvalue] = data  
    else:  
        ...
```

Get hash value



Easy case  
slot is empty



Tricky case  
slot is not empty



# HashTable Class

The put method cont.

```
def put(self, key, data):
    :
    else:
        if self.slots[hashvalue] == key:
            self.data[hashvalue] = data #replace
        else:
            nextslot = self.rehash(hashvalue, len(self.slots))
            while self.slots[nextslot] != None and \
                  self.slots[nextslot] != key:
                nextslot = self.rehash(nextslot, len(self.slots))

            if self.slots[nextslot] == None:
                self.slots[nextslot] = key
                self.data[nextslot] = data
            else:
                self.data[nextslot] = data #replace
```


Tricky case  
slot is not empty



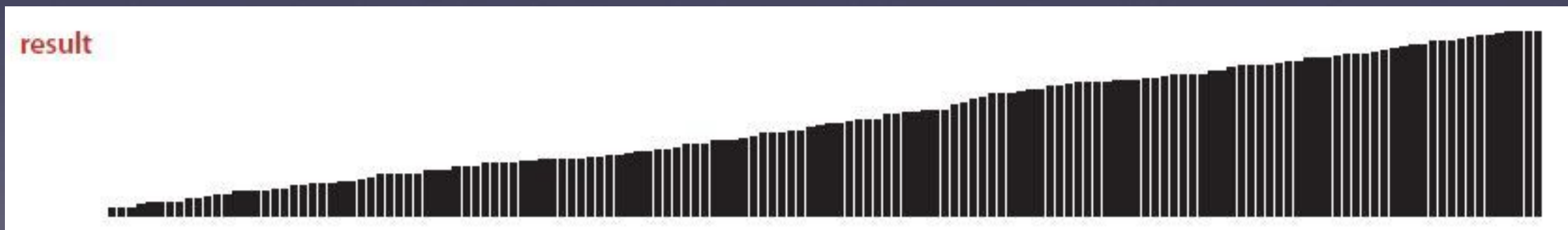
Due to duplicate key?



Linear probing  
for empty slot  
or  
duplicate key



# Sorting



# Sorting and searching

Computers are essential for keeping track of large quantities of data and finding little pieces of that data on demand.

Searching for and finding data when necessary is made much easier when the data is sorted in some way, so computer scientists have thought a lot about how to **sort** things.

Finding medical records, banking information, income tax returns, driver's license information...even names in a phone book...all depends on the information being sorted.

# Sorting and searching

There is a lot of information out there to be searched. So computer scientists have spent a lot of time thinking about how to sort efficiently. It is a classic textbook problem.

Does the efficiency of one approach to sorting vs. another make a difference? Yes.

Here's an example of a well-known, simple, but not very efficient sorting algorithm that's easily implemented using a Python list.



# Selection sort

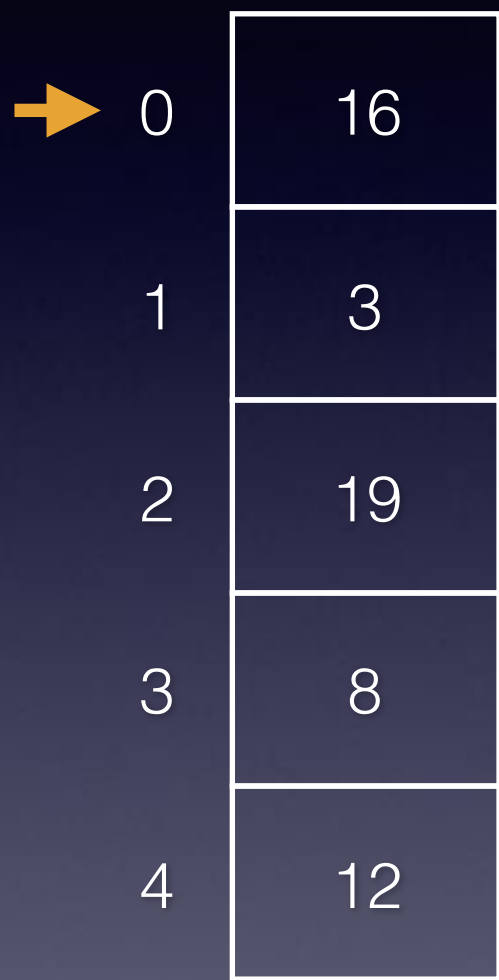
0	16
1	3
2	19
3	8
4	12

Let's say we want to sort the values in the list at the left in increasing order.

One way to approach the problem would be to use an algorithm called **selection sort**.

It will become obvious why it's called this.

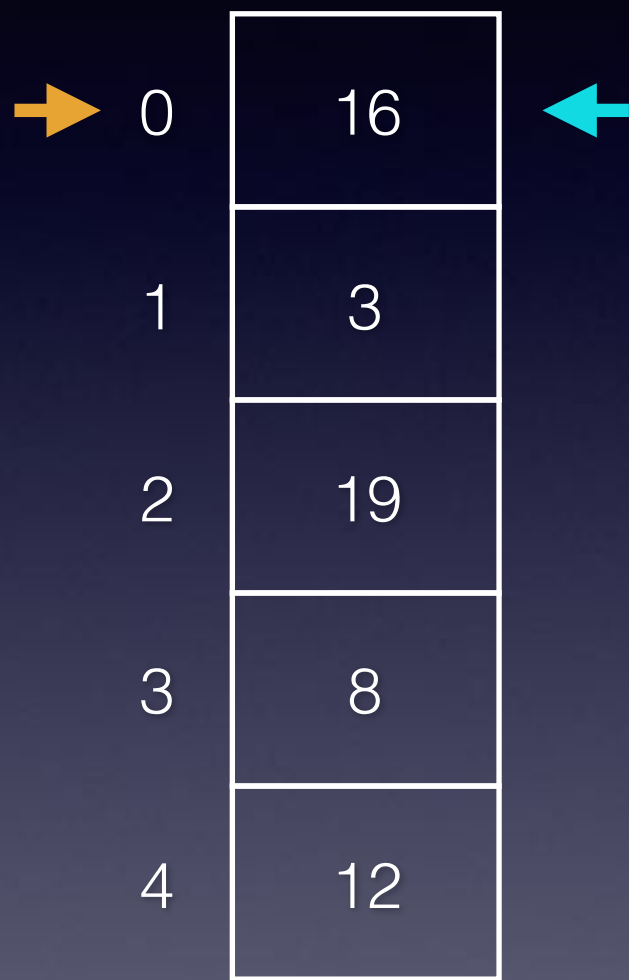
# Selection sort



→ 0	16
1	3
2	19
3	8
4	12

We start by setting a pointer to the first element in the list; this is where the smallest value in the list will be placed.

# Selection sort



We start by setting a pointer to the first element in the list; this is where the smallest value in the list will be placed.

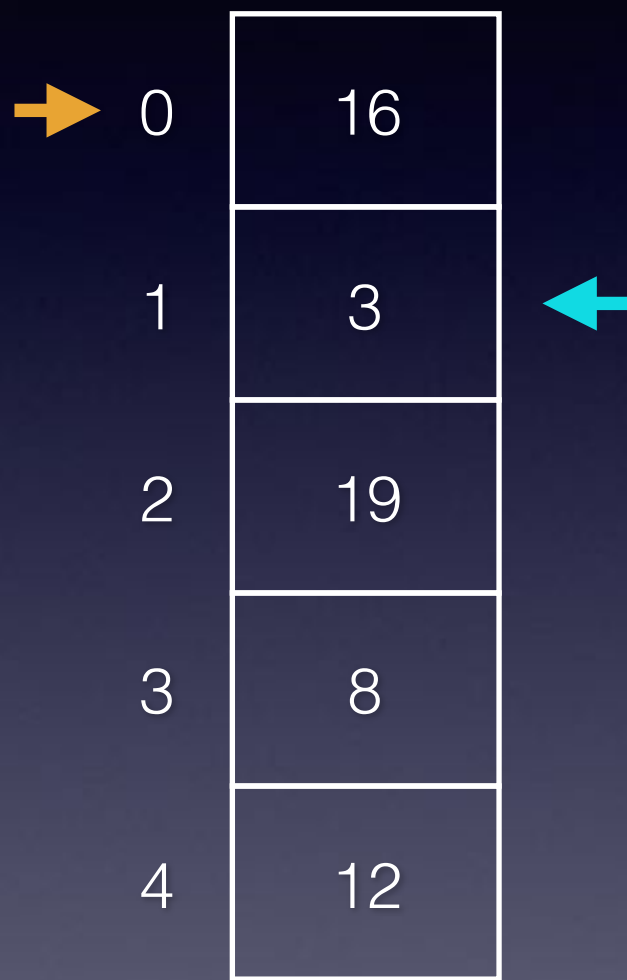
Then we'll look at every value in this unsorted list and find the minimum value.

First we note the smallest value seen so far.

The smallest value  
so far is 16

Its index is 0

# Selection sort



The smallest value  
so far is 3

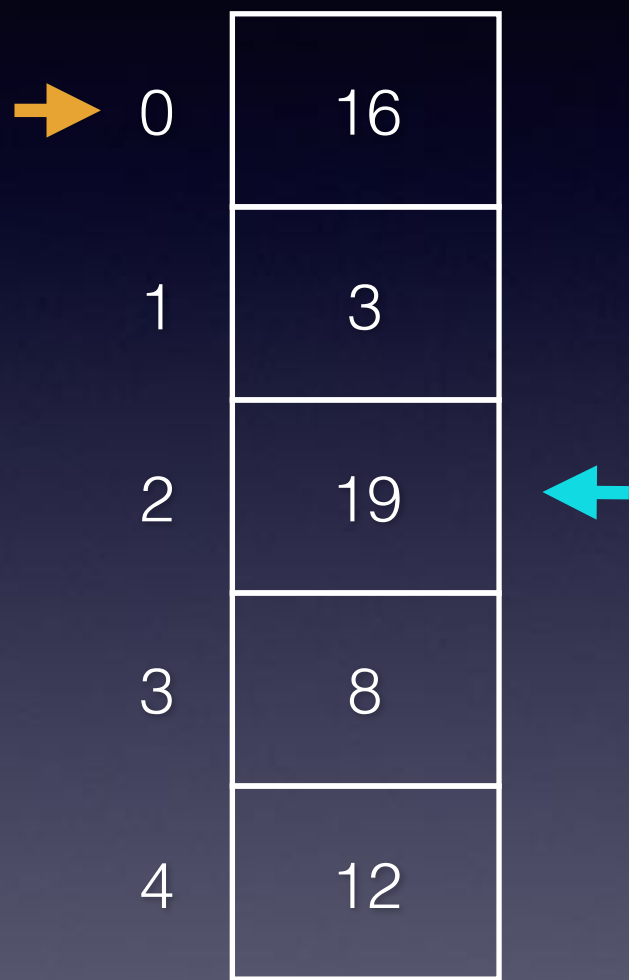
Its index is 1

We start by setting a pointer to the first element in the list; this is where the smallest value in the list will be placed.

Then we'll look at every value in this unsorted list and find the minimum value.

As we traverse the list, if we see a smaller value we update the smallest value seen so far.

# Selection sort



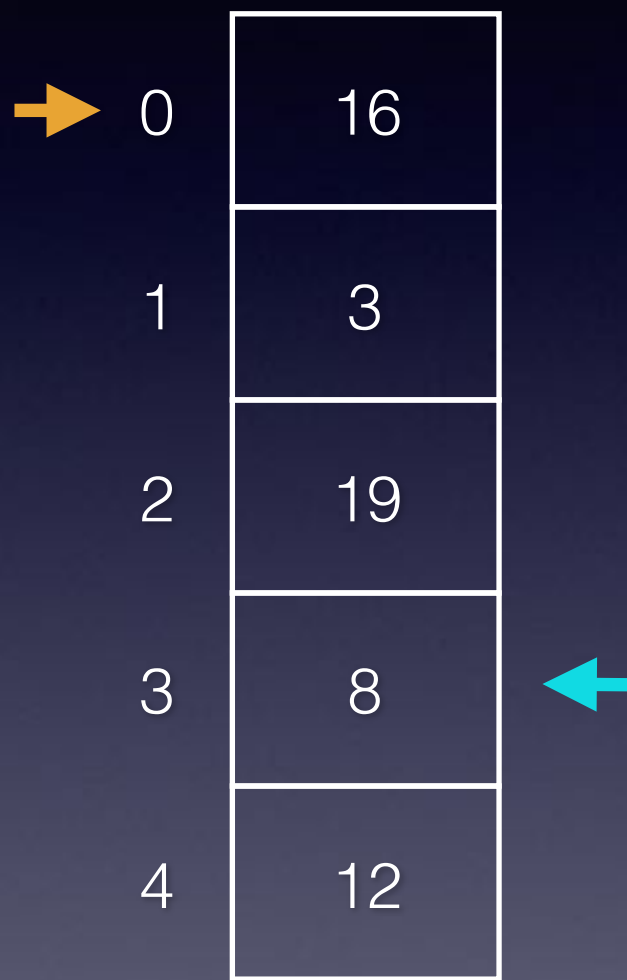
We start by setting a pointer to the first element in the list; this is where the smallest value in the list will be placed.

Then we'll look at every value in this unsorted list and find the minimum value.

The smallest value  
so far is 3

Its index is 1

# Selection sort



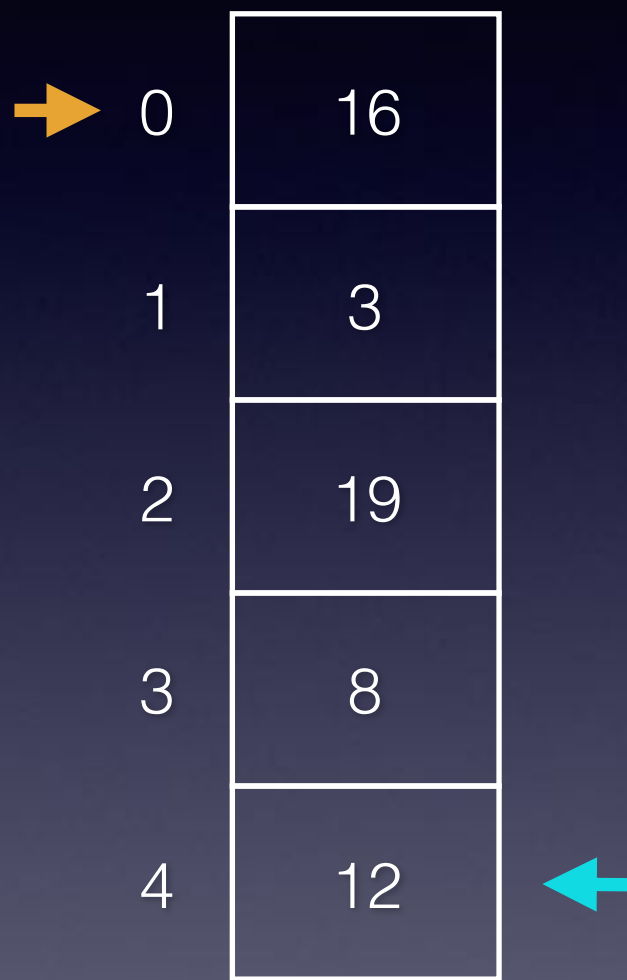
We start by setting a pointer to the first element in the list; this is where the smallest value in the list will be placed.

Then we'll look at every value in this unsorted list and find the minimum value.

The smallest value  
so far is 3

Its index is 1

# Selection sort



The smallest value  
so far is 3

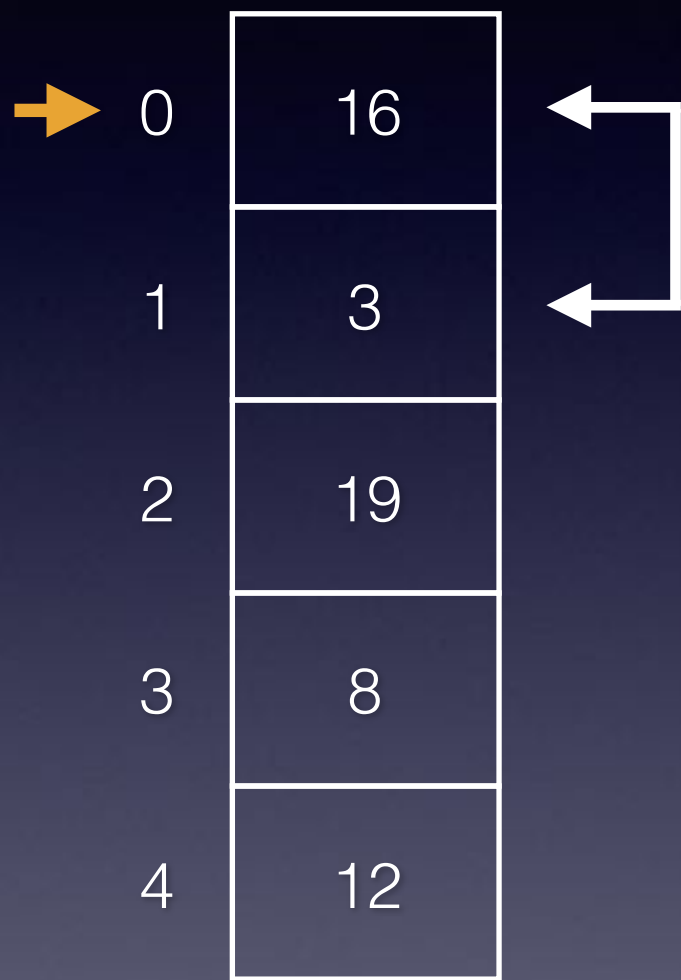
Its index is 1

We start by setting a pointer to the first element in the list; this is where the smallest value in the list will be placed.

Then we'll look at every value in this unsorted list and find the minimum value.

After we finish examining all values in the list, we know we have the minimum value.

# Selection sort



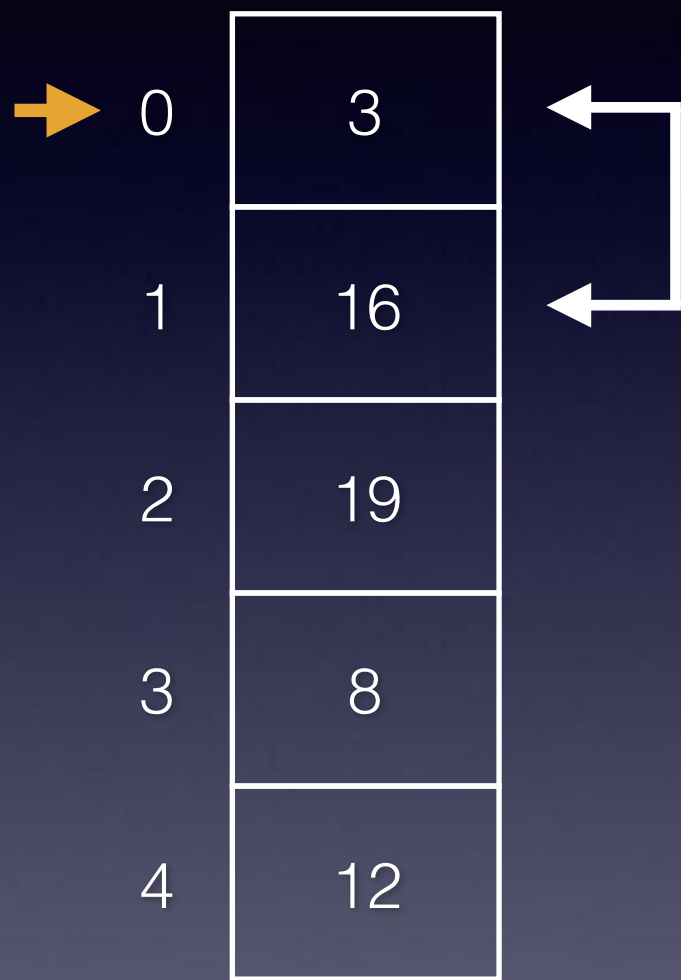
Once we've found the minimum value, we swap that value with the one we selected at the beginning.

The smallest value  
so far is 3

Its index is 1



# Selection sort

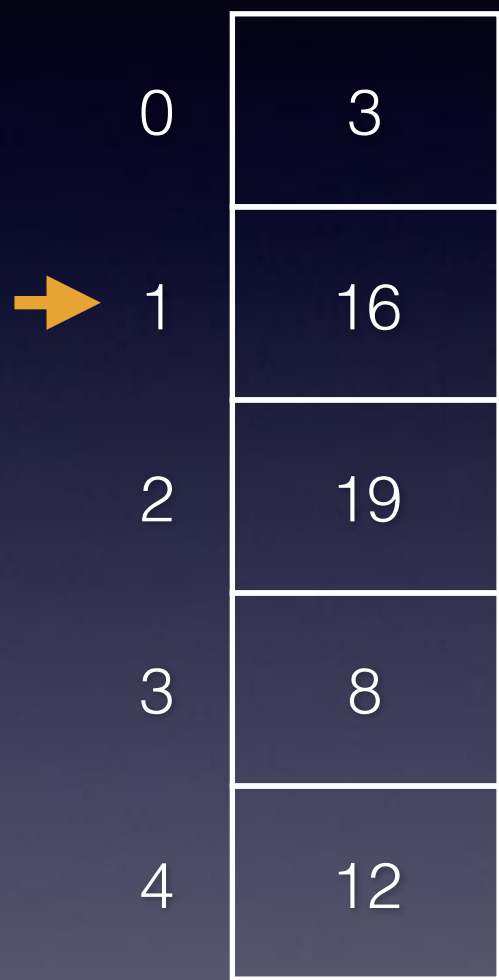


Once we've found the minimum value, we swap that value with the one we selected at the beginning.

The smallest value  
so far is 3

Its index is 1

# Selection sort



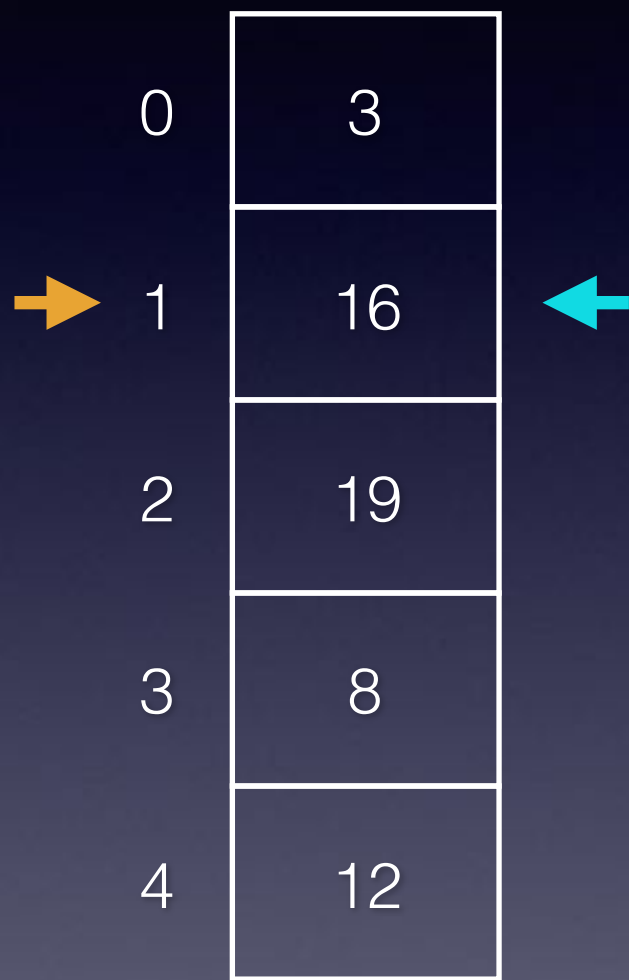
0	3
→ 1	16
2	19
3	8
4	12

At this point we know that the smallest number in the list is in the first (index 0) element in the list.

That is, the first element is sorted, and the rest of the list remains unsorted.

So now we can select the second element of the list to be the location which will hold the next smallest value in the list.

# Selection sort



The smallest value  
so far is 16

Its index is 1

At this point we know that the smallest number in the list is in the first (index 0) element in the list.

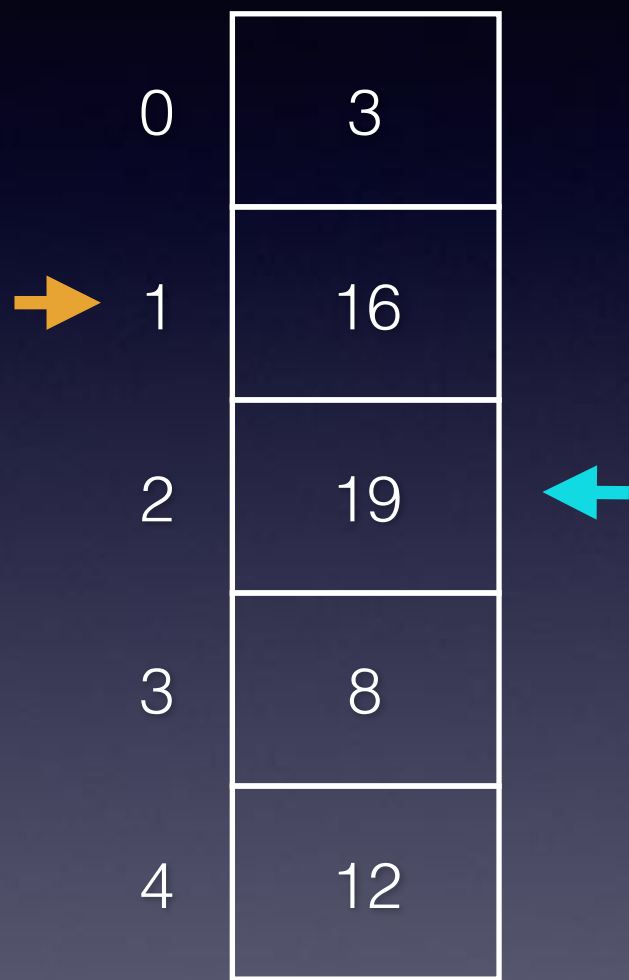
That is, the first element is sorted, and the rest of the list remains unsorted.

So now we can select the second element of the list to be the location which will hold the next smallest value in the list.

In other words, we'll do everything we just did, only we'll do it only to the unsorted part of the list -- in this case, all but the first element.

# Selection sort

So we'll look at every value in the unsorted part of the list and find the minimum value.

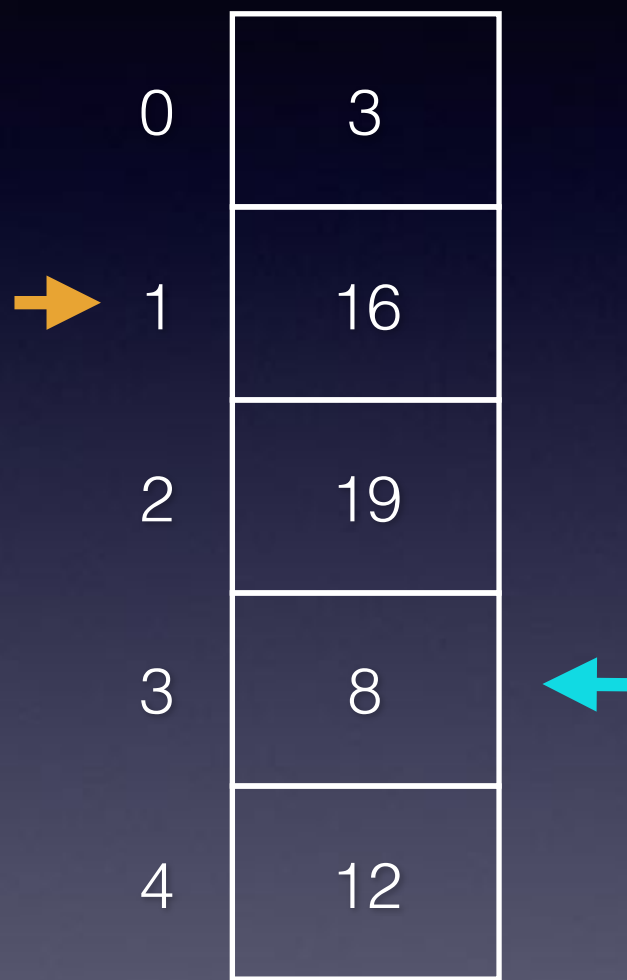


The smallest value  
so far is 16

Its index is 1

# Selection sort

So we'll look at every value in the unsorted part of the list and find the minimum value.

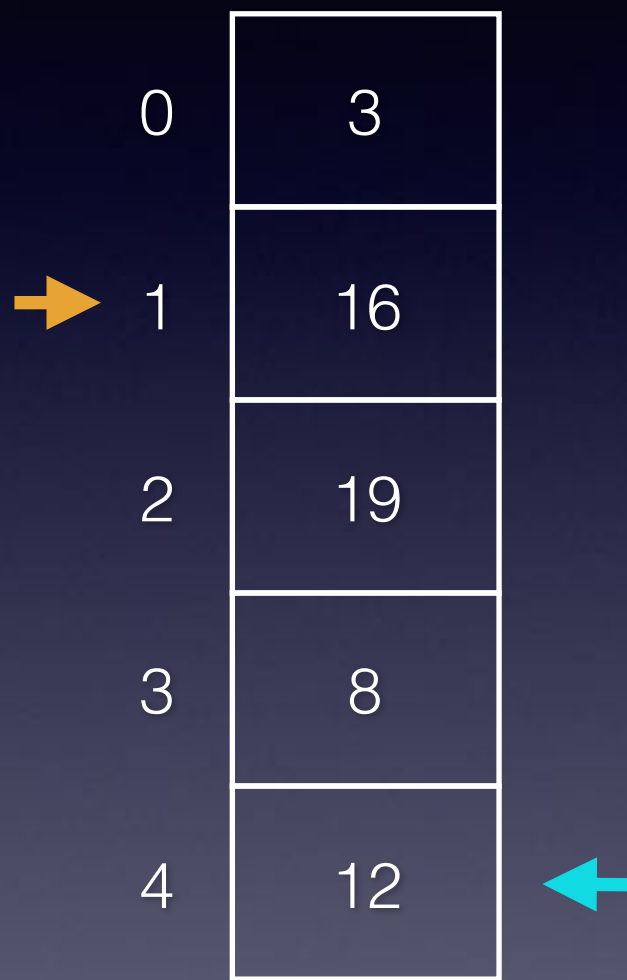


The smallest value  
so far is 8

Its index is 3

# Selection sort

So we'll look at every value in the unsorted part of the list and find the minimum value.

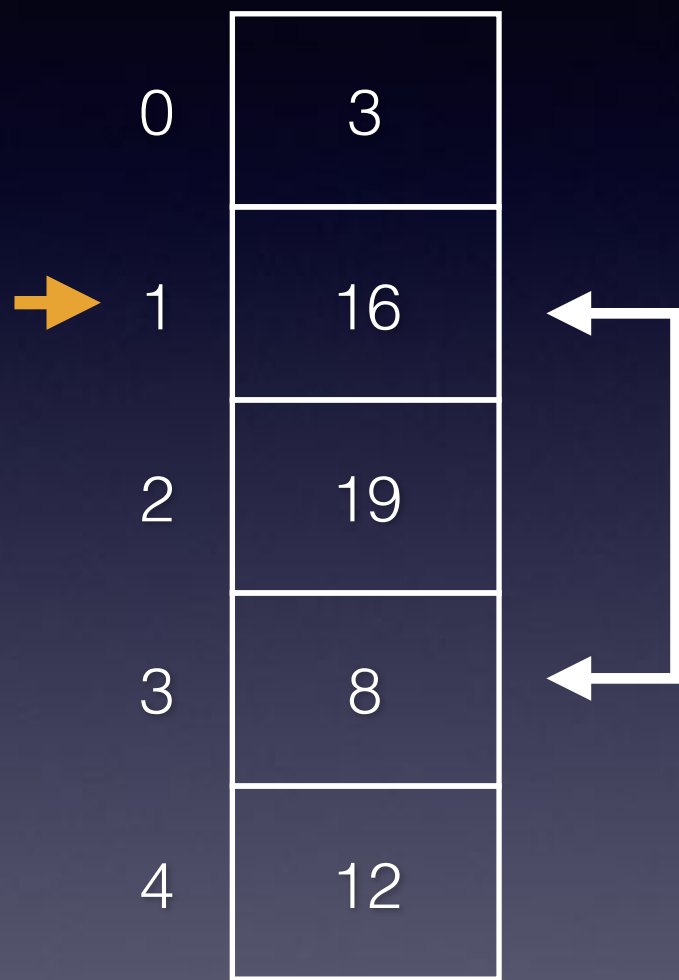


The smallest value  
so far is 8

Its index is 3

# Selection sort

Now we swap the minimum value with the selected list value -- in this case, the second element.

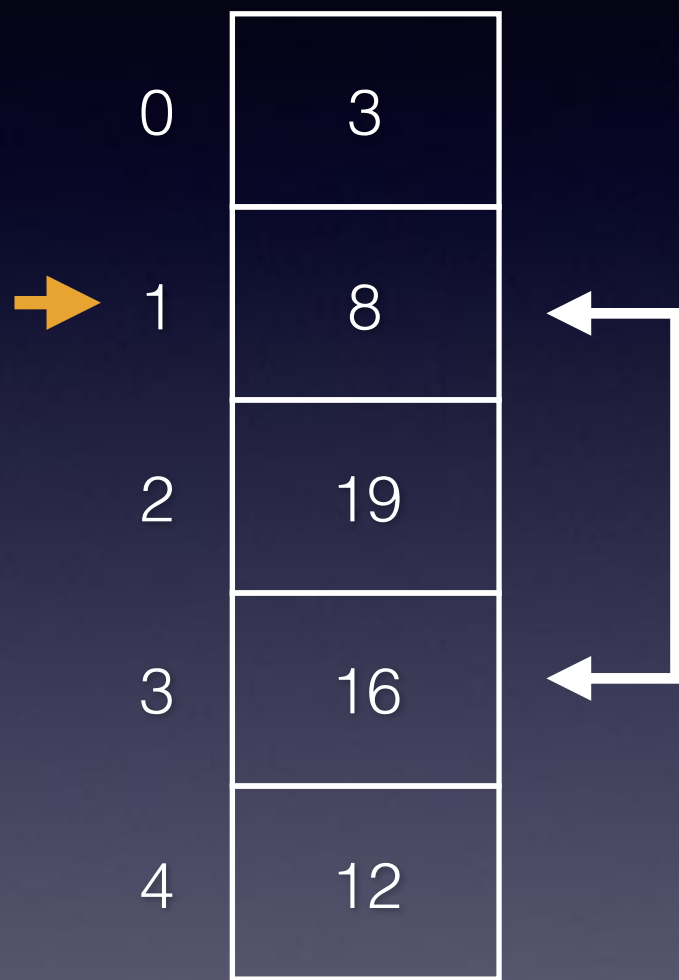


The smallest value  
so far is 8

Its index is 3

# Selection sort

Now we swap the minimum value with the selected list value -- in this case, the second element.

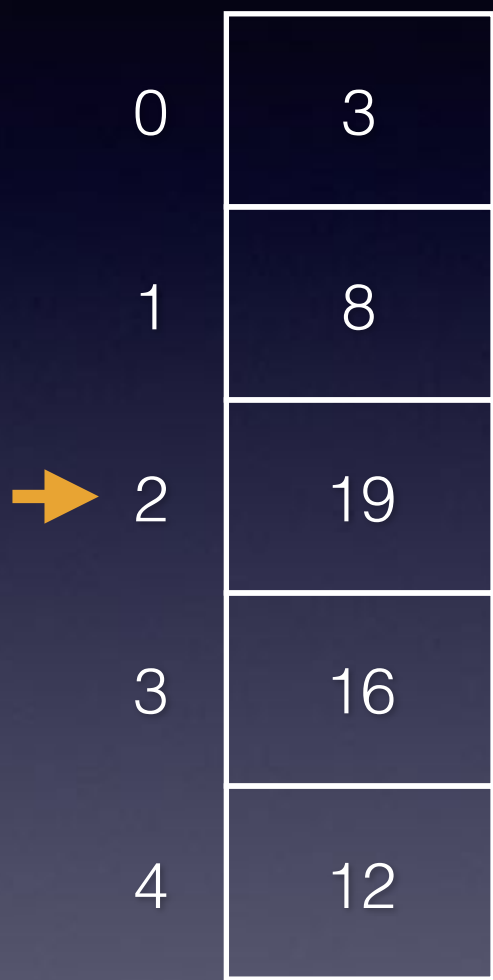


The smallest value  
so far is 8

Its index is 3



# Selection sort

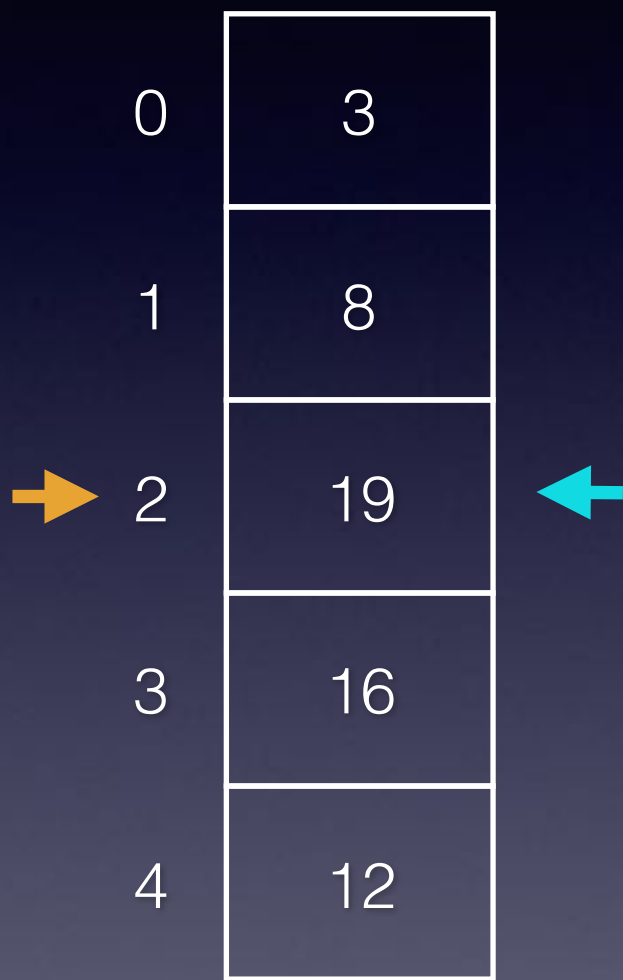


0	3
1	8
→ 2	19
3	16
4	12

Now the first two elements of the list are sorted.

We select the third element of the list (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the list for that value, just like before.

# Selection sort



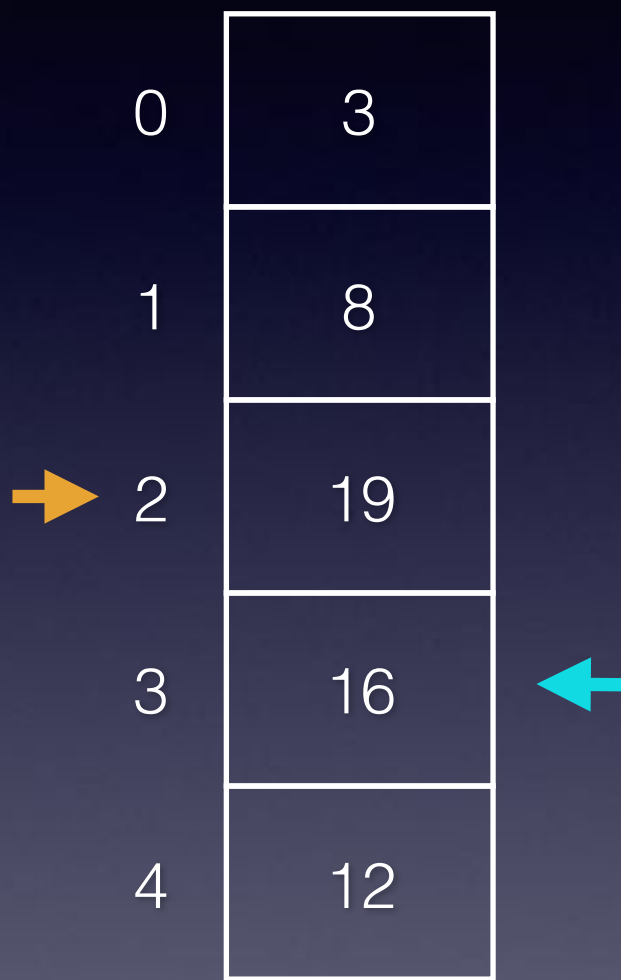
Now the first two elements of the list are sorted.

We select the third element of the list (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the list for that value, just like before.

The smallest value  
so far is 19

Its index is 2

# Selection sort



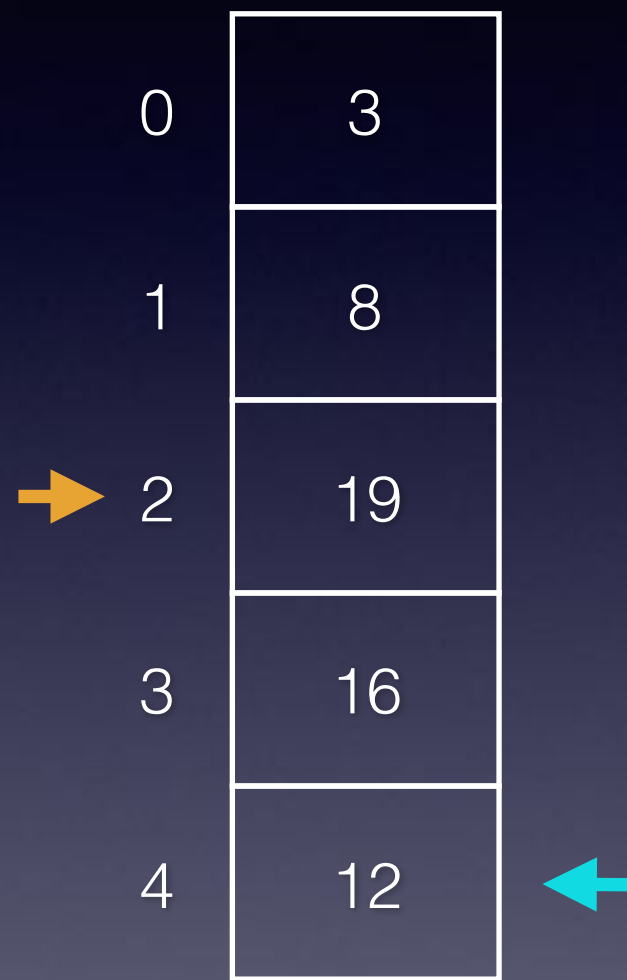
Now the first two elements of the list are sorted.

We select the third element of the list (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the list for that value, just like before.

The smallest value  
so far is 16

Its index is 3

# Selection sort



Now the first two elements of the list are sorted.

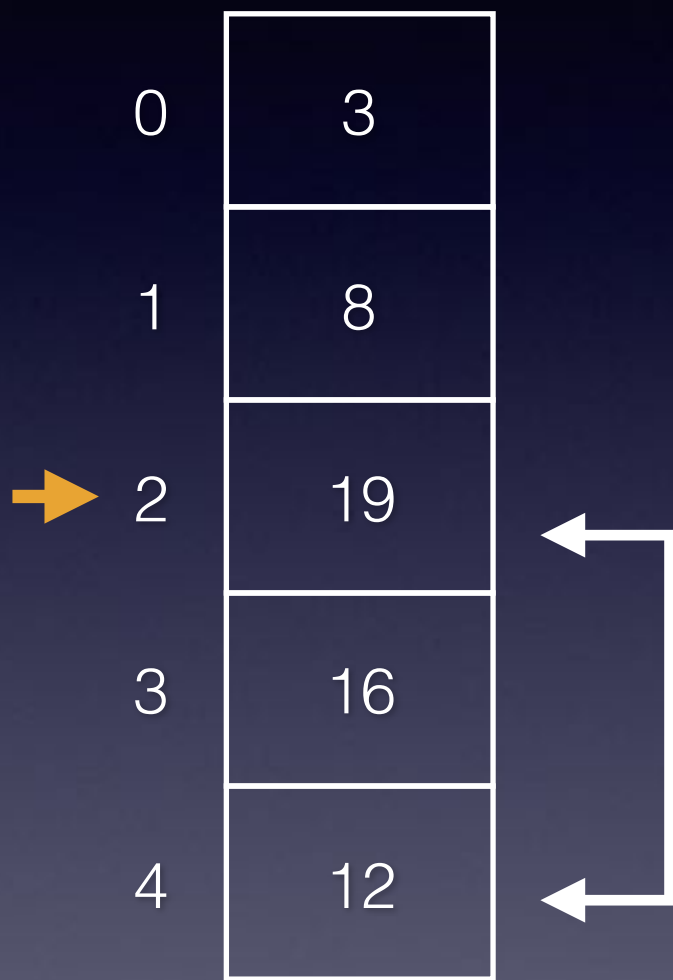
We select the third element of the list (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the list for that value, just like before.

The smallest value  
so far is 12

Its index is 4

# Selection sort

Once we've visited all the remaining list indices, again, we can swap the values...

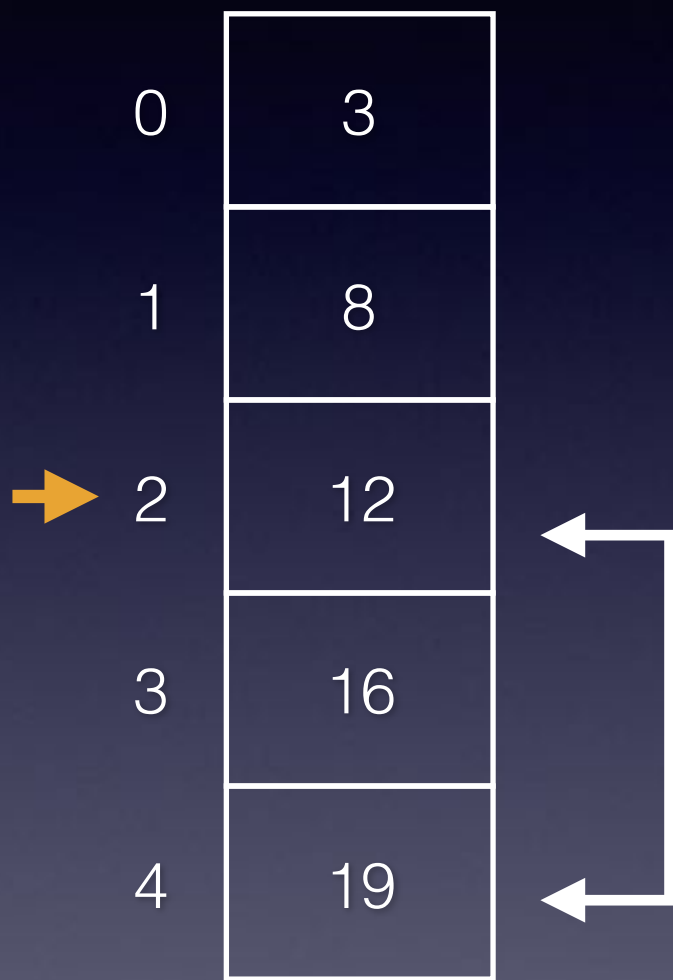


The smallest value  
so far is 12

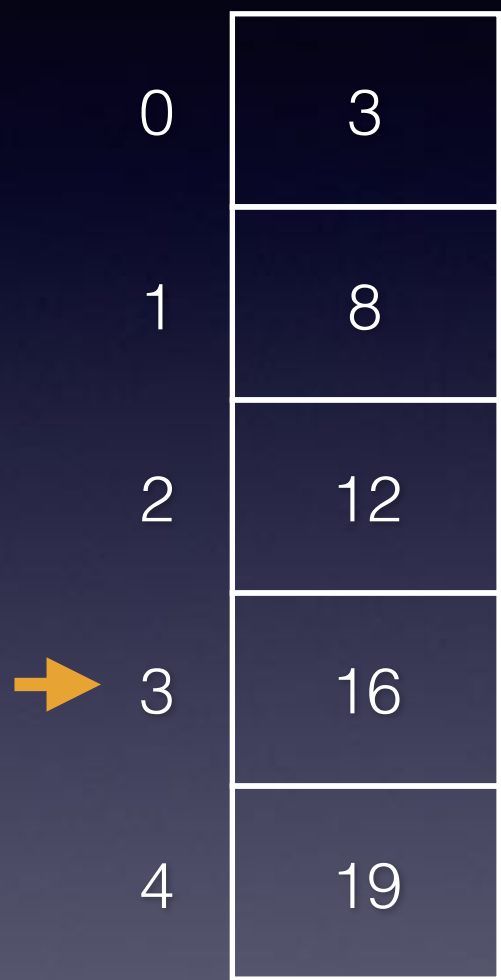
Its index is 4

# Selection sort

Once we've visited all the remaining list indices, again, we can swap the values...



# Selection sort



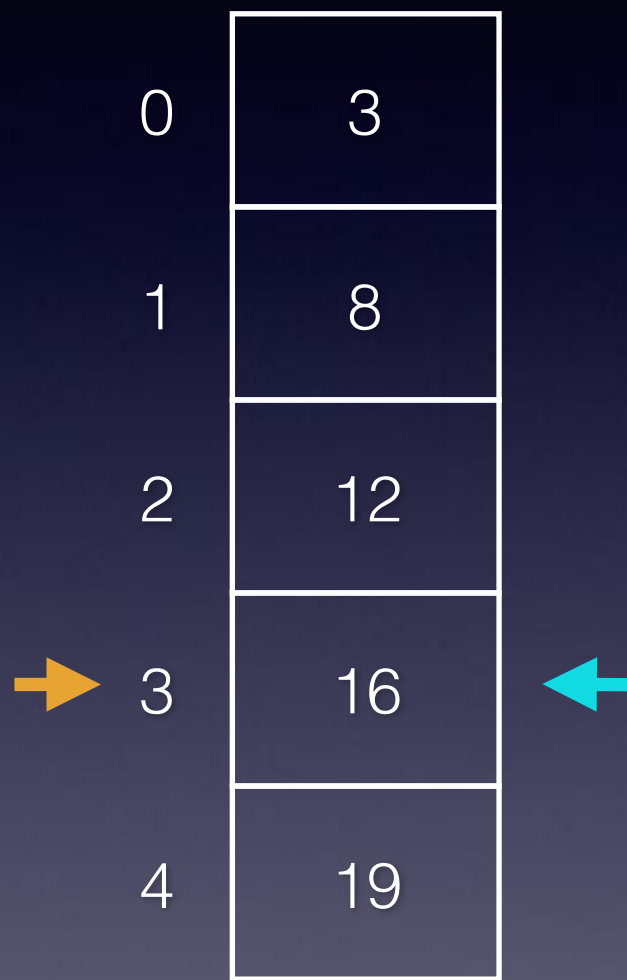
0	3
1	8
2	12
→ 3	16
4	19

Once we've visited all the remaining list indices, again, we can swap the values... and do the whole thing again.

Indices 3 and higher are unsorted.

Index 3 will be the location for the next smallest value in the unsorted part of the list.

# Selection sort



The smallest value  
so far is 16

Its index is 3

Once we've visited all the remaining list indices, again, we can swap the values... and do the whole thing again.

Indices 3 and higher are unsorted.

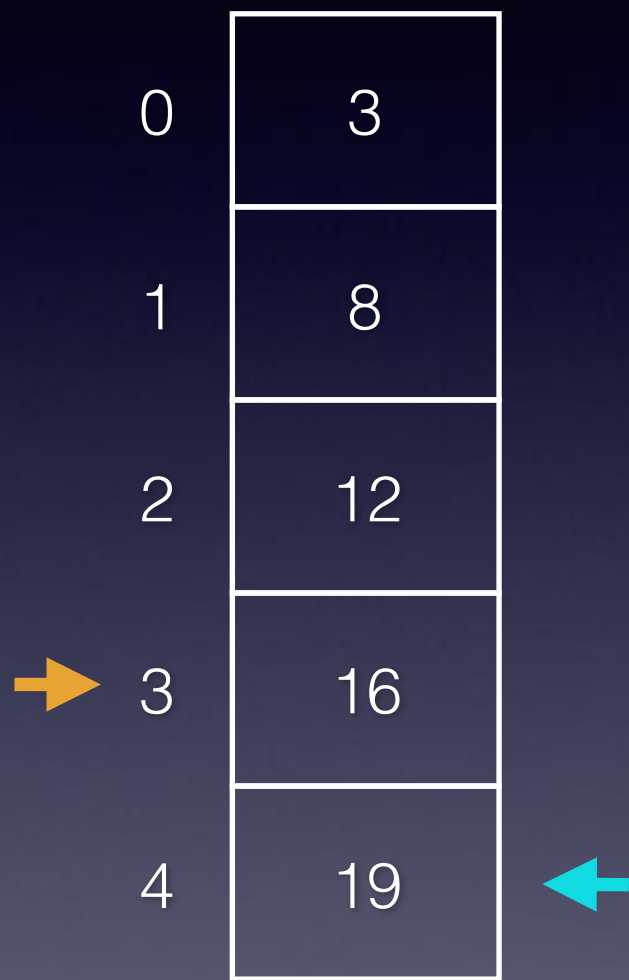
Index 3 will be the location for the next smallest value in the unsorted part of the list.

The smallest value so far is set to index 3.



# Selection sort

Once we've visited all the remaining list indices, again, we can swap the values... and do the whole thing again.

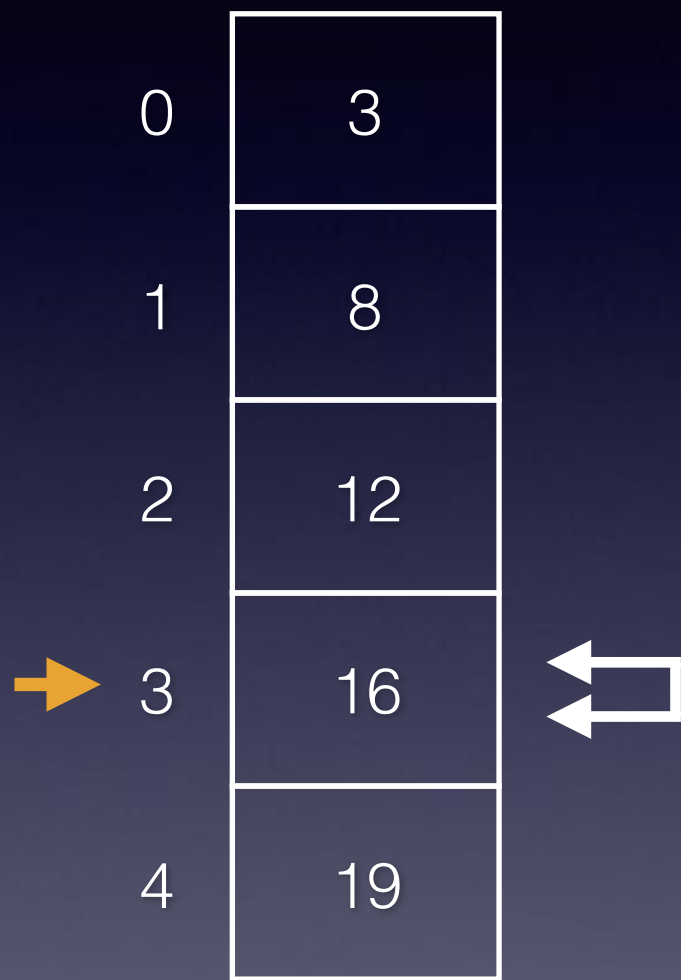


The smallest value  
so far is 16

Its index is 3

# Selection sort

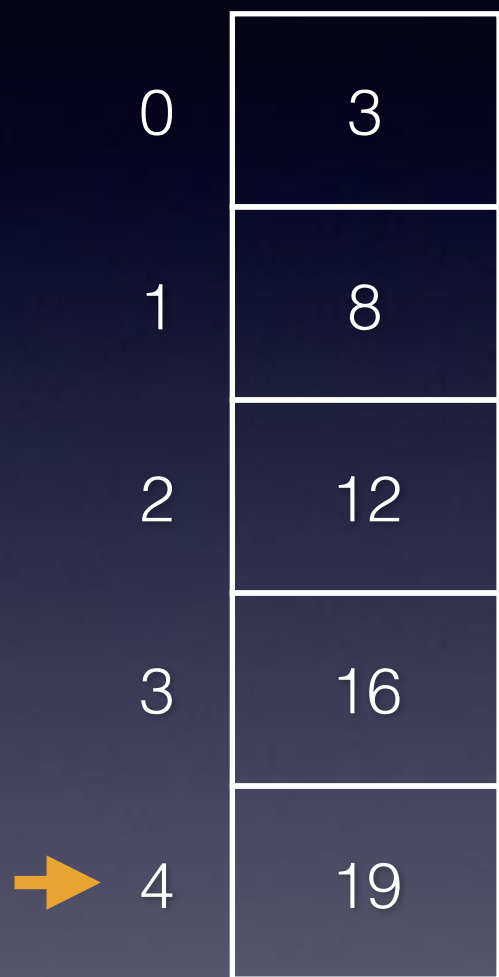
And we swap again...it's not actually necessary in this particular case, but it's what the algorithm says to do, so we do it.



The smallest value  
so far is 16

Its index is 3

# Selection sort



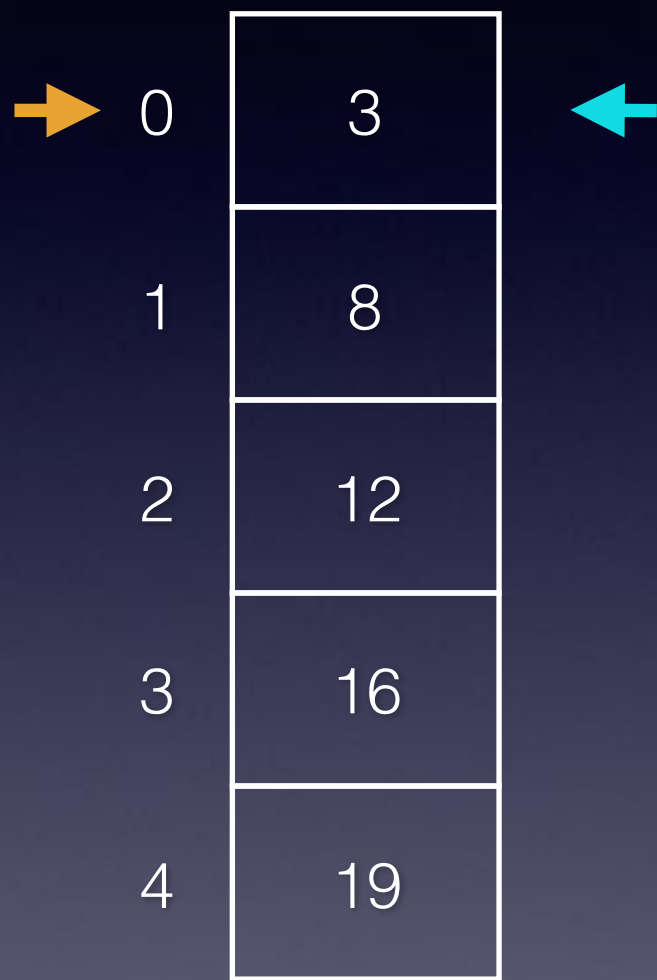
0	3
1	8
2	12
3	16
→ 4	19

Finally, we could select the last element of the list (index 4).

But since we know all of the list except for this element is already sorted, we can quickly conclude that this element is the largest value in the list, so it's already where we want it.

In other words, the list is sorted, and we're done. We don't need to select the last element.

# Selection sort

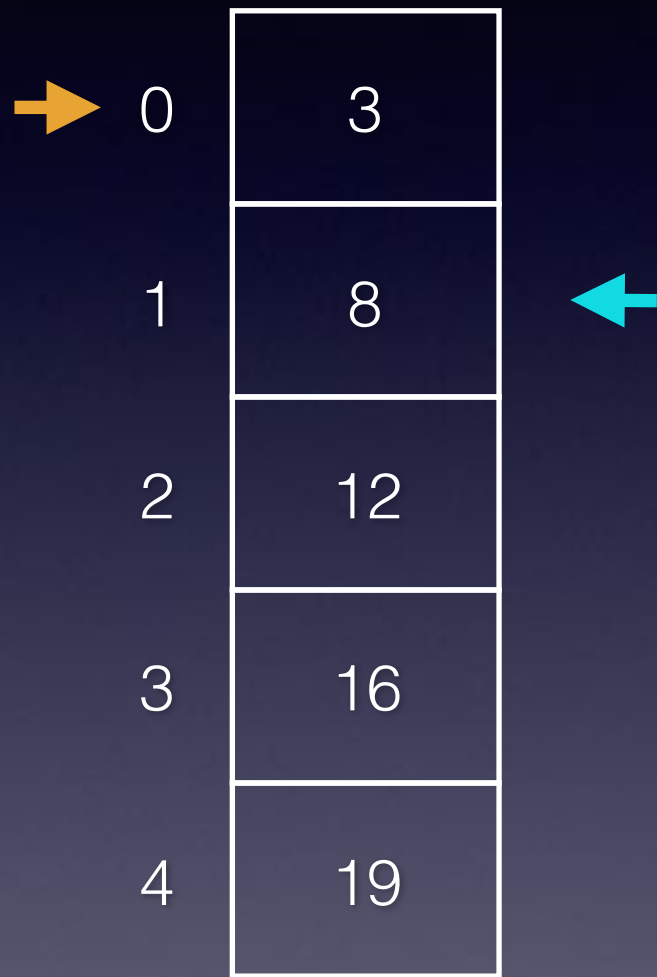


As the example proceeded, you watched the arrow on the **left** and the arrow on the **right** move down the list. The arrows represent two different variables, each one containing an index into the list.

When you look at a selection sort procedure, it should have two loops.

Think of the outer loop as controlling the movement of the **yellow** arrow, and the inner loop as controlling the movement of the **blue** arrow.

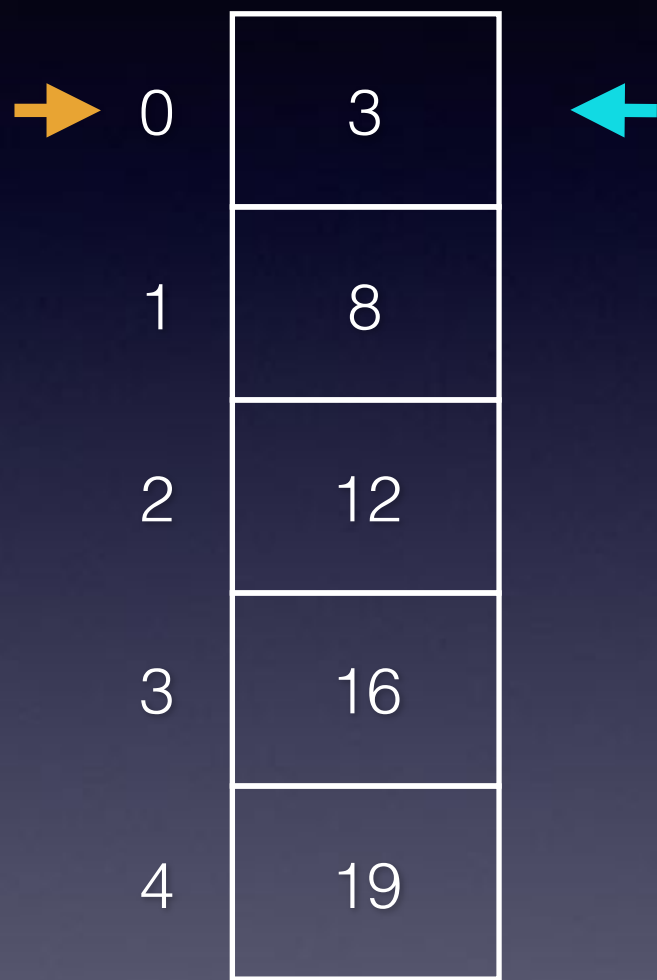
# Selection sort



0	3
1	8
2	12
3	16
4	19

```
## selection sort not from book
def selectionSort2(alist):
    for i in range(0, len(alist) - 1):
        min = i
        for j in range(i + 1, len(alist)):
            if alist[j] < alist[min]:
                min = j
        temp = alist[i]
        alist[i] = alist[min]
        alist[min] = temp
```

# Selection sort



Your textbook does this slightly differently.

The Python program in the textbook starts at the bottom of the list.

It puts the largest value in the list in the correct place, then moves upward to place the next largest value, and so on.

Whether the sorting goes from small to large or large to small, the algorithm is still selection sort.

# Selection sort

0	16
1	3
2	19
3	8
→ 4	12

```
## selection sort from book
def selectionSort(alist):
    for fillslot in range(len(alist)-1, 0, -1):
        positionOfMax = 0
        for location in range(1, fillslot + 1):
            if alist[location] > alist[positionOfMax]:
                positionOfMax = location

        temp = alist[fillslot]
        alist[fillslot] = alist[positionOfMax]
        alist[positionOfMax] = temp
```

# Now for the good stuff

The selection sort algorithm is a **comparison-based sort**, so to get some sense of the time requirements, **we count the comparisons** that must be made to complete the sorting.




# Estimating time required to sort



0	16
1	3
2	19
3	8
4	12

We can go back to the selection sort example and count the comparisons. The first pass through the list of 5 elements started with 16 being compared to 3, then 3 was compared to 19, 8, and 12. There were 4 comparisons. The value 3 was moved into the location at index 0.

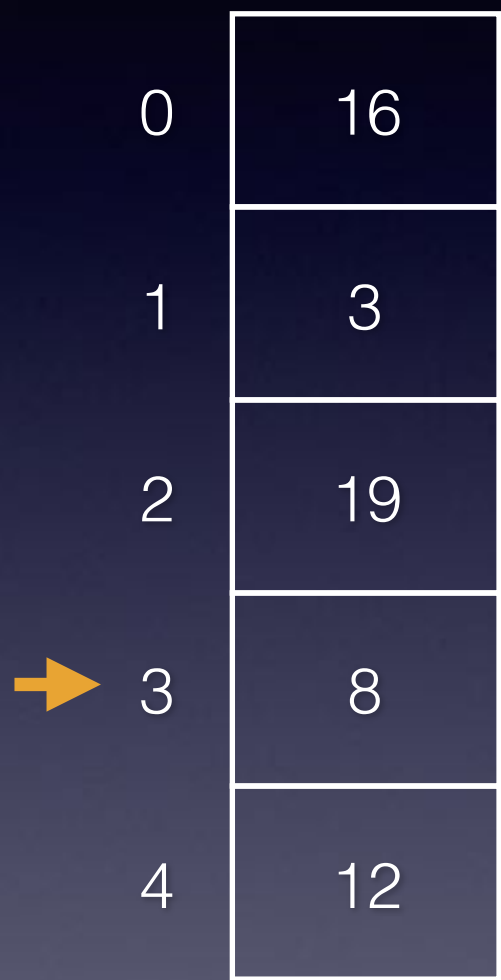
# Estimating time required to sort



0	16
1	3
2	19
3	8
4	12

Then the second pass through the list began, starting with index 1. 16 was compared to 19, then 16 was compared to 8, which became the new minimum and was compared to 12. So among 4 elements in the list, there were 3 comparisons.

# Estimating time required to sort



0	16
1	3
2	19
→ 3	8
4	12

It takes 4 passes through the list to get it completely sorted. There are 4 comparisons on the first pass, 3 comparisons on the second pass, 2 comparisons on the third pass, and 1 comparison on the last pass. That is, it takes  $4 + 3 + 2 + 1 = 10$  comparisons to sort a list of five values.

If you do this same computation on a list with six values, you'll find it takes  $5 + 4 + 3 + 2 + 1 = 15$  comparisons to sort the list. Do you see a familiar pattern?

# Estimating time required to sort

0	16
1	3
2	19
→ 3	8
4	12

For a list with n values:

$$(n-1) + (n-2) \dots + 2 + 1$$



n-1 terms

# Estimating time required to sort

0	16
1	3
2	19
→ 3	8
4	12

For a list with  $n$  values:

$$(n-1) + (n-2) \dots + 2 + 1$$

$n-1$  terms



Pair opposing pairs of terms:

$$n + n + \dots n$$

$(n-1)/2$  terms

# Estimating time required to sort

0	16
1	3
2	19
→ 3	8
4	12

For a list with  $n$  values:

$$(n-1) + (n-2) \dots + 2 + 1$$

$n-1$  terms



Pair opposing pairs of terms:

$$n + n + \dots n$$

$(n-1)/2$  terms

$$T(n) = n(n-1)/2 \text{ comparisons}$$

$$T(n) \text{ is } O(n^2)$$

# Estimating time required to sort

As  $n$ , the number of values in the list gets very big, the number of comparisons needed to sort the list grows in proportion to  $n^2$ , with the other terms becoming insignificant by comparison.

So sorting a list of 1,000 values would require approximately 1,000,000 comparisons. Similarly, sorting a list of 1,000,000 values would take approximately 1,000,000,000,000 comparisons.

As the number of values to be sorted grows, the number of comparisons required to sort them grows **much faster**.

Here are some real numbers to help you think about just how long it might take to sort some really big lists...

# Estimating time required to sort

Let's assume that your computer can make 1 billion (1,000,000,000) comparisons per second. That's a lot of comparisons in a second. And let's say your computer was using selection sort to sort the names of the people in the following hypothetical telephone books. Here's some mathematical food for thought.

phone book	number of people (N)	$N^2$	number of seconds needed to sort	
Sacramento	501,902	251,905,617,604	252	or 4.2 minutes
Canada	30,000,000	900,000,000,000,000	900,000	or 10.4 days
People's Republic of China	1,000,000,000	1,000,000,000,000,000,000	1,000,000,000	or 31.7 years
World	7,000,000,000	49,000,000,000,000,000,000	49,000,000,000	or 1554 years



# Estimating time required to sort

Fortunately, the best sorting algorithms can run in  $O(n \log_2 n)$  time instead of  $O(n^2)$  time. If  $n$  is 7,000,000,000, then  $\log_2(n)$  is just a little less than 33.

So if we round up to 33,  $n \log n$  would be 231,000,000,000 comparisons instead of 49,000,000,000,000,000,000,000 ( $n^2$  comparisons). **That's over 200 million times faster.**

If our computer can perform 1,000,000,000 comparisons per second, then sorting all the names in the whole world phone book now takes **231 seconds instead of 1554 years**. And that's why it's important to think about the efficiency of algorithms, especially as the size of your data set gets really really big.