

Lecture 2

Data Structures - overview

Comparing Algorithms: Analysis

Growth of Functions

Prof.dr. Irma Ravkic

Based on book (and more): Object-Oriented Data Structures using Java 4th Ed. by
Nell Dale

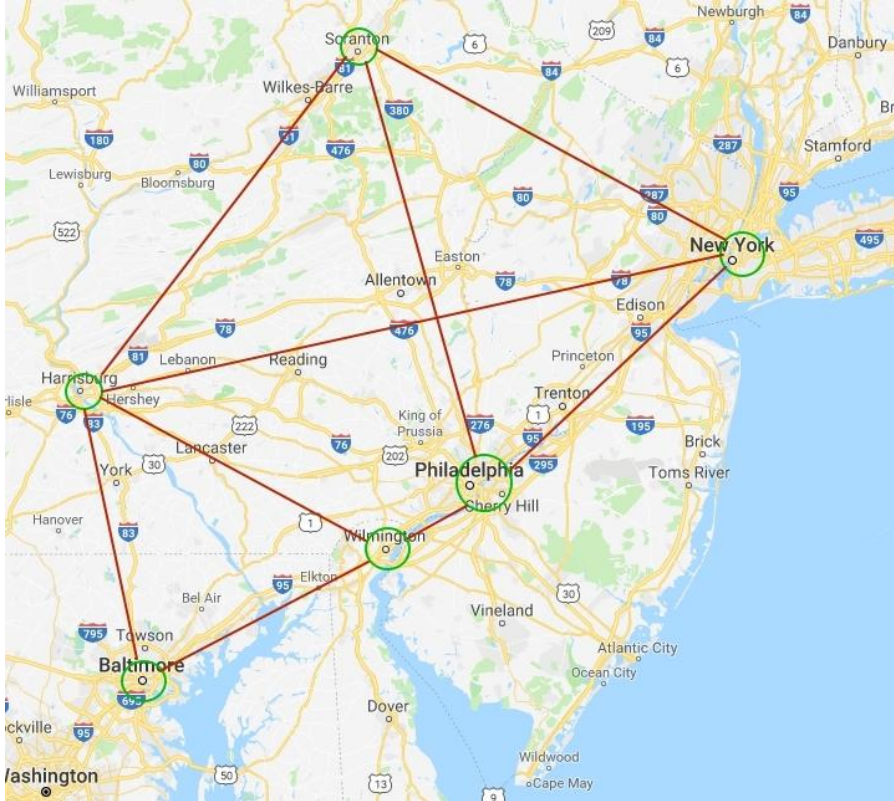
Data Structures: motivation

When you look up a course description in a catalog or a word in a dictionary, you are using **an ordered list of words** (how do you look up a word in a dictionary? You **map** a word to its translation!)

When you take a number at a delicatessen you become part of a **line/queue**

When you study the pairings in a sports tournament and try to predict which team or player will advance through all the rounds and become champion, you create a **treelike** list of predicted results

Data Structures: motivation



Graphs

For example: when representing places (as nodes) and connection between places (links) with some properties (distance)

Data Structures: motivation

Java's primitive types (int, char, float): good for representing arithmetic elements (sum of two numbers, character of keyboards)

VS.

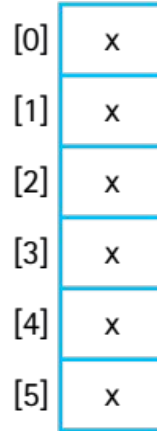
Data structures: organizational, structured data to represent data relationships

Remember: complex numbers

```
typedef struct  
{  
    double real;  
    double imag;  
} Py_complex
```

In Python, you have complex numbers as part of the language, but internally they are implemented with a C-like structure

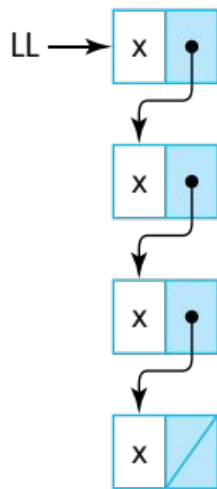
Implementation-dependent structures: arrays



[0]	x
[1]	x
[2]	x
[3]	x
[4]	x
[5]	x

You have studied and used arrays in your previous work. An array's components are accessed by using their positions in the structure. Arrays are one of the most important organizational structures. They are available as a basic language construct in most high-level programming languages. Additionally, they are one of the basic building blocks for implementing other structures.'

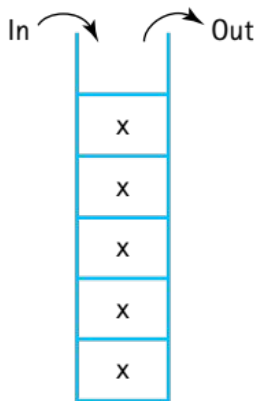
Implementation-dependent structures: linked lists



A linked list is a collection of separate elements, with each element linked to the one that follows it in the list. We can think of a linked list as a chain of elements. The linked list is a versatile, powerful, basic implementation structure and, like the array, it is one of the primary building blocks for the more complicated structures. Teaching you how to work with links and linked lists is one of the important goals of this text.

Implementation-independent structures: stack

Stack




The defining feature of a stack is that whenever you access or remove an element, you work with the element that was most recently inserted. Stacks are “last in, first out” (LIFO) structures. To see how they work, think about a stack of dishes or trays. Note that the concept of a stack is completely defined by the relationship between its accessing operations, the operations for inserting something into it or removing something from it. No matter what the internal representation is, as long as the LIFO relationship holds, it is a stack.

Implementation-independent structures: queue



Queues are, in one sense, the opposite of stacks. They are “first in, first out” (FIFO) structures. The defining feature of a queue is that whenever you access or remove an element from a queue, you work with the element that was in the queue for the longest time. Think about an orderly line of people waiting to board a bus or a group of people, holding onto their service numbers, at a delicatessen. In both cases, the people will be served in the order in which they arrived. In fact, this is a good example of how the abstract organizational construct, the queue, can have more than one implementation approach—an orderly line or service numbers.

Implementation-independent structures: sorted list

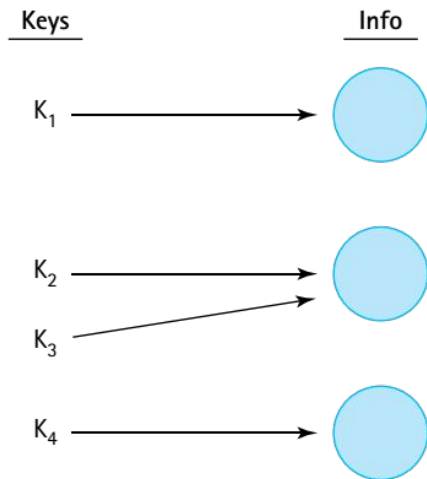


George, John, Paul, Ringo

The elements of a sorted list display a linear relationship. Each element (except the first) has a predecessor, and each element (except the last) has a successor. In a sorted list, the relationship also reflects an ordering of the elements, from “smallest” to “largest,” or vice versa.

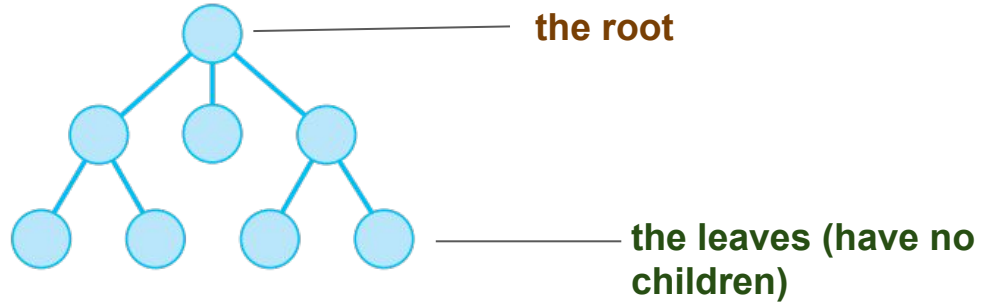
You might be thinking that an array whose elements are sorted is a sorted list—and you would be correct! As we said earlier, arrays are one of the basic building blocks for constructing other structures. But that is not the only way to implement a sorted list. We will cover several other approaches.

Implementation-independent structures: map



Maps, also known as dictionaries, tables, or associative arrays, are used to store “key”-“info” ordered pairs. Maps provide quick access to desired information when you provide an appropriate key. Consider, for example, when you enter a bank and provide a teller with your account number—within a few seconds (hopefully) the teller has access to your account information. Your account number is the “key”—it “maps” onto your account information. Although there are many ways to implement a map structure, they all must follow the same simple rules: keys are unique and a key maps onto a single information node.

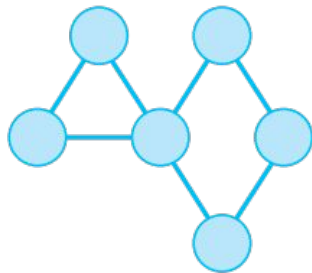
Implementation-independent structures: tree



Trees and graphs are nonlinear. Each element of a tree is capable of having many successor elements, called its children. A child element can have only one parent. Thus, a tree is a branching structure. Every tree has a special beginning element called the root. The root is the only element that does not have a parent.

Trees are useful for representing hierarchical relationships among data elements. For example, they can be used to classify the members of the animal kingdom or to organize a set of tasks into subtasks. Trees can even be used to reflect the *is-a* relationship among Java classes, as defined by the Java inheritance mechanism.

Implementation-independent structures: tree



A graph is made up of a set of elements, usually called nodes or vertices, and a set of edges that connect the vertices. Unlike with trees, there are no restrictions on the connections between the elements. Typically, the connections, or edges, describe relationships among the vertices. In some cases, values, also called weights, are associated with the edges to represent some feature of the relationship. For example, the vertices may represent cities and the edges may represent pairs of cities that are connected by airplane routes. Values of the edges could represent the distances or travel times between cities.

Basic structuring mechanisms

- *Memory*
- *References*
- *Arrays*

Memory

address word

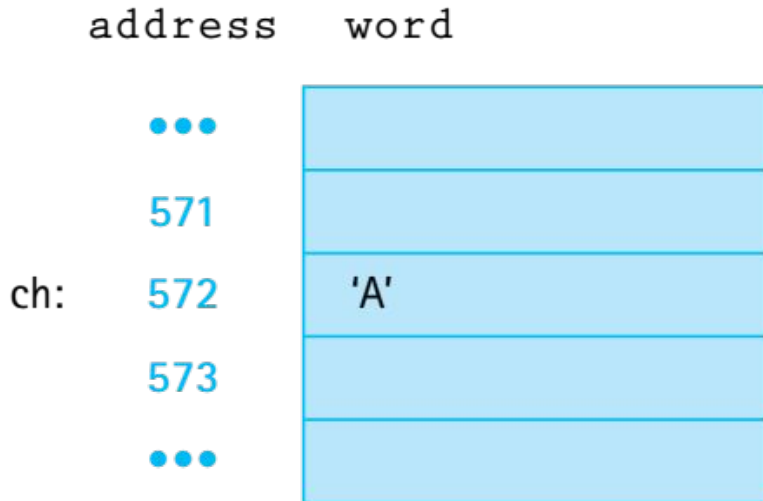
0	
1	
2	
3	
...	

All programs and data are held in memory

A **variable** in our program corresponds to a **memory location**. The compiler handles the translation so that every time the code references the same variable, the system uses the same memory location.

Addressing modes: direct

Corresponds how primitive types are used in Java: the memory location associated with the variable holds the value of the variable



Java code:

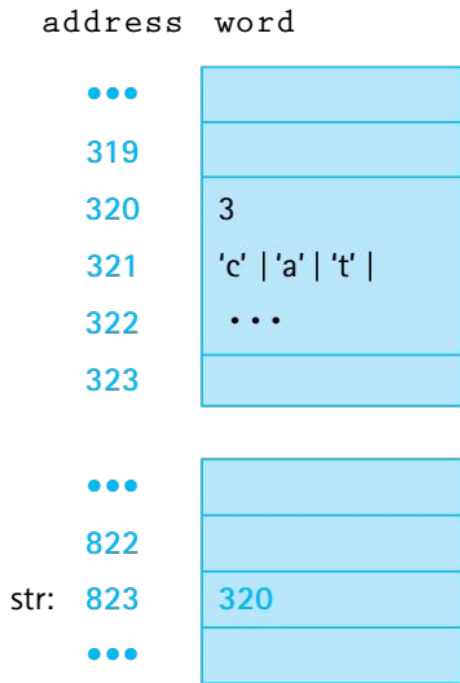
```
char ch = 'A';
```

Abstract view:

```
ch: 'A'
```

Addressing modes: indirect

The memory location associated with the variable **holds the address** of the location the holds the value of the variable



Java code:

```
String str = "cat";
```

Abstract view:

str → "cat"

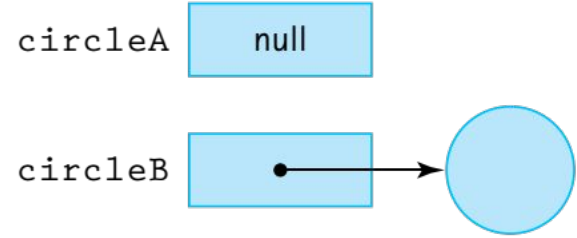
References

```
Circle circleA;
```

```
Circle circleB = new Circle(8);
```

Reserves memory space for a variable of class **Circle**

Also creates an object of class **Circle** and places the reference to that object in **circleB** variable



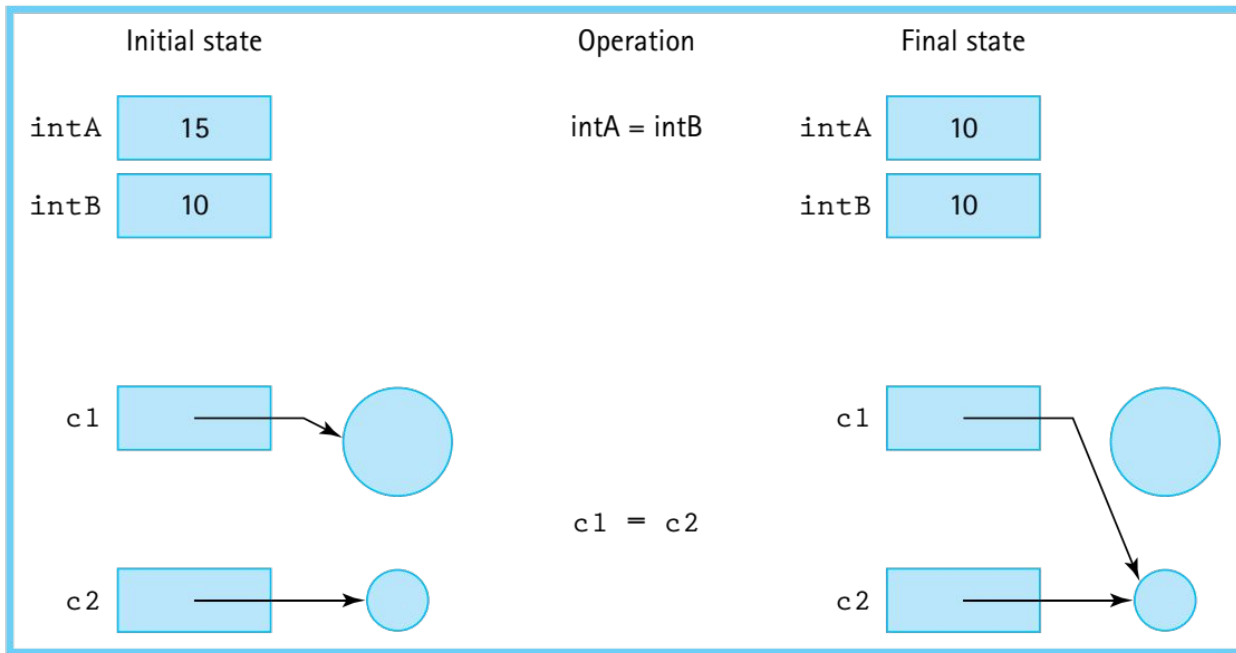
- The reference itself is a memory address
- Sometimes called links, pointers, addresses
- No address assigned? **Null reference**
- Can also explicitly assign null reference: `circleA = null;`

Reference types versus Primitive types

Primitive types (such as int) are handled “**by value**”

Non-primitive types (arrays, classes) are handled “**by reference**”

This distinction is very important! See the image below!



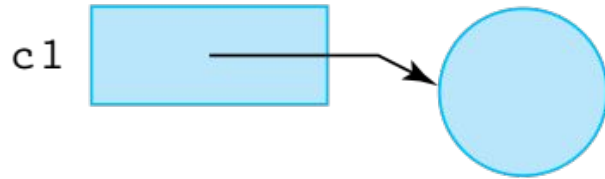
Aliases

When we assign a value of non-primitive type to non-primitive type we actually don't copy the value of the object, but the reference

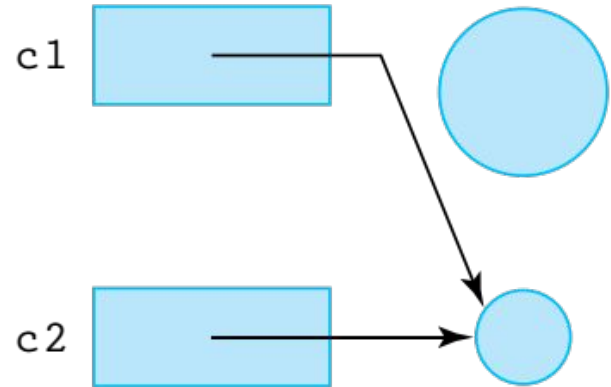
This means we will have two names for two different objects: **aliases**

Good programmers avoid aliases!

If you want to copy a non-primitive object you must use a copy constructor!



`c1 = c2`



Garbage

Memory space that has been allocated, but is not accessible to the program is called **garbage**

```
Circle c1;  
for (n = 1; n <= 100; n++)  
{  
    Circle c1 = new Circle(n);  
    // Code to initialize and use c1 goes here.  
}
```

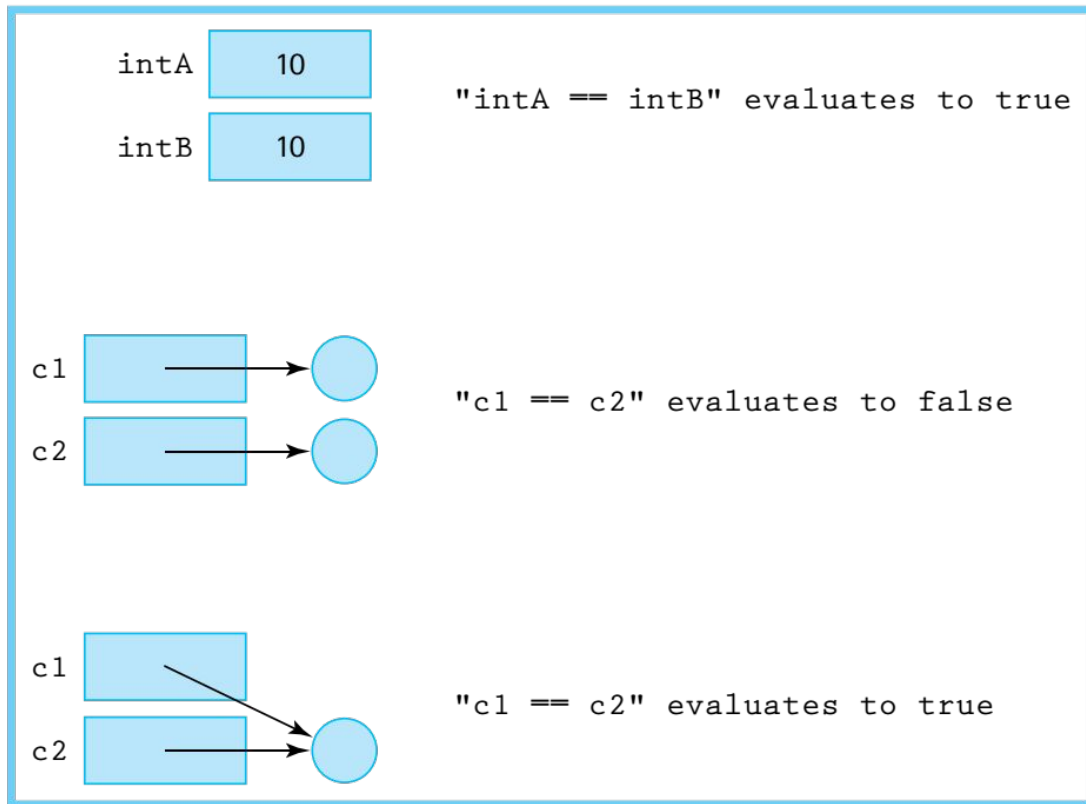
Only one object accessible by c1 reference after the loop is finished. Other 99 are garbage.

Garbage collection: a **Java** process that identifies the unreachable objects and deallocates the memory (**dynamic memory management**) - not all OOP languages have this (e.g., C++)

Comparing objects

Comparison operator (==) doesn't work on non-primitive types as you would think: **you're comparing addresses not values!**

== for non-primitive types is only true for aliases!



Parameters

Java allows only passing “by value” of arguments to methods

As a consequence, passing an object variable as an argument causes the receiving method to create an alias of the object. If the method uses the alias to make changes to the object, then when the method finishes, an access via the original variable finds the object in its modified state.

Java Note

All java arguments are “passed by value.” If the argument is of a primitive type, it represents the value of the primitive. If the argument is of a reference type, then it represents the address of the object.

Arrays (in Java)

Basic structuring construct. An array allows the programmer to access a sequence of locations using an **indexed** approach

Arrays in Java are non-primitive type (handled by reference just like objects)! Be careful with aliases, comparisons and method parameters

Arrays must be instantiated with new (like all other objects)

```
int[] numbers = new int[10];
```

Arrays (in Java)

Question 1: What are the initial values in an array instantiated by using new?

Answer: for primitives their **default values**, for objects initial values are **null**

Question 2: Can you provide initial values for an array?

Answer: Yes. `int numbers[] = {5, 32, -23, 57, 1, 0, 27, 13, 32, 32};`

Question 3: What if we go out of bounds of an array? E.g., its size is 9 and we access index 9 (we go from index 0 in Java)

Answer: It would be out-of-bounds exception. Java takes care you don't access/modify a location outside of the bounds of arrays (unlike C++, for instance)

Arrays of Objects (in Java)

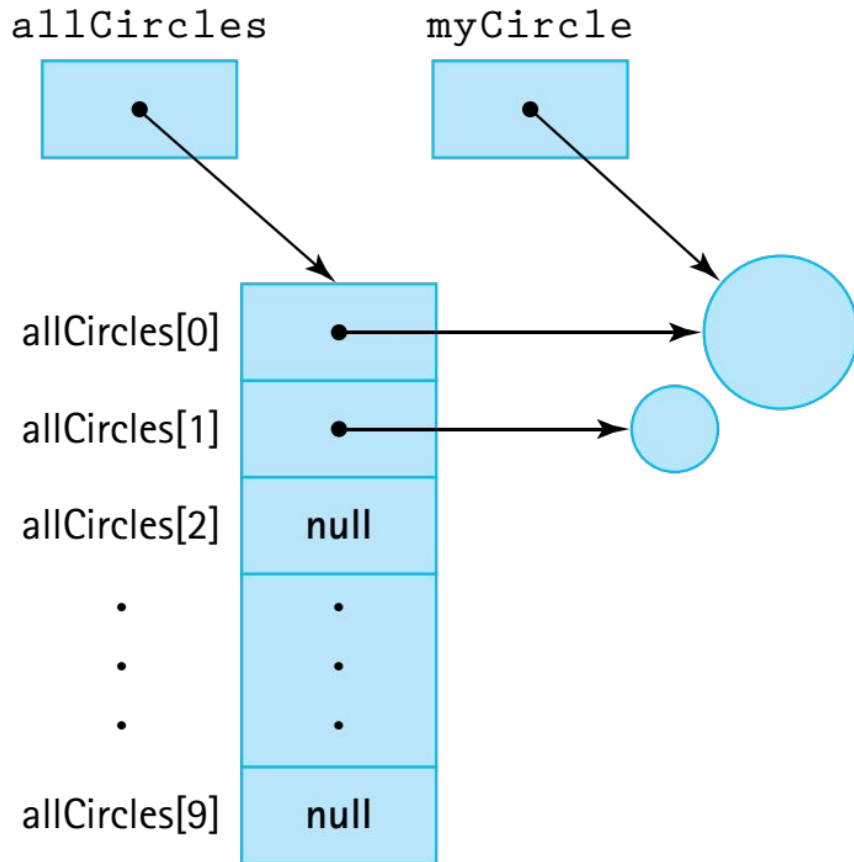
```
Circle[] allCircles = new Circle[10];
```

This means allCircles is an array that can hold 10 references to Circle objects. What are the diameters of the circles? We do not know yet. The array of circles has been instantiated, but the Circle objects themselves have not.

```
Circle[] allCircles = new Circle[10];  
allCircles[0] = myCircle;  
allCircles[1] = new Circle(4);
```

Arrays of Objects (in Java)

```
Circle[] allCircles = new Circle[10];  
allCircles[0] = myCircle;  
allCircles[1] = new Circle(4);
```



Comparing algorithms:

Order of growth analysis - big O

- *Time efficiency*
- *Complexity cases*
- *Size of input*
- *Comparison*
- *Common orders of growth*
- *Space complexity*

Motivation #1

Scenario 1:

You've got a file on a hard drive and you need to send it to your friend who lives across the country. You need to do it as fast as possible. How should you send it?

E-mail? Yes, if it's a small file

What if the file is REALLY large? Deliver by plane?

Motivation #1

Scenario 1:

You've got a file on a hard drive and you need to send it to your friend who lives across the country. You need to do it as fast as possible. How should you send it?

Send one 1TB of data

- 47 hours to finish uploading at a speed of 50 Mbps (assuming your connection never drops or slows down)
- 1 day at 100 Mbps
- But what if you have 2 TB of content to upload, or 4 TB, or 10 TB? Even at a 100-Mbps sustained data-transfer rate, you will need a mind-boggling 233 hours to move 10 TB of content!

Motivation #1

Scenario 1:

You've got a file on a hard drive and you need to send it to your friend who lives across the country. You need to do it as fast as possible. How should you send it?

Obviously different methods you use influence the time of the process

- **Electronic Transfer:** $O(s)$, s is the size of the file. The time to transfer the file increases linearly with the size of the file (a simplification)
- **Airplane transfer:** $O(1)$, as the size of the file increases it won't take longer to take the file to your friend. The time is **constant** (for reasonable files you would send your friend).

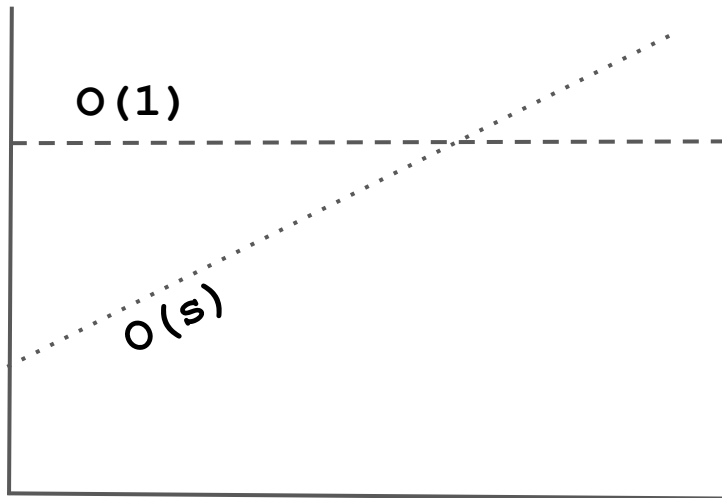
Motivation #1

Scenario 1:

You've got a file on a hard drive and you need to send it to your friend who lives across the country. You need to do it as fast as possible. How should you send it?

- Electronic Transfer: $O(s)$
- Airplane transfer: $O(1)$

No matter how big the constant is, and how slow the linear increase is, linear will at some point surpass constant.



Motivation #2

Alice: "I'm thinking of a number between 1 and 1,000."

Bob: "Is it 1?"

Alice: "No ... it's higher."

Bob: "Is it 2?"

Alice: "No ... it's higher."

Bob: "Is it 3?"

Alice: rolls her eyes ...

Motivation #2

Alice: "I'm thinking of a number between 1 and 1,000."

Bob: "Is it 1?"

Alice: "No ... it's higher."

Bob: "Is it 2?"

Alice: "No ... it's higher."

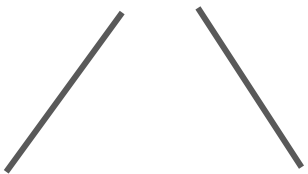
Bob: "Is it 3?"

Alice: rolls her eyes ...

A valid algorithm (sequential search), but very inefficient

Motivation #2

Should help you with determining which of two algorithms requires fewer resources to accomplish a certain task



How much
time does it
take?

How much
space
(memory)
does it take?

Trade-offs: *sorting a deck of cards numbered 1–300: sitting on a bus with these cards and have to sort them while holding them in your hands versus if you are standing in front of a table large enough to hold all 300 of them. In this situation you can look at each card just once and place it in its correct spot on the table. The extra space afforded by the table allows for a more time-efficient sorting algorithm.*

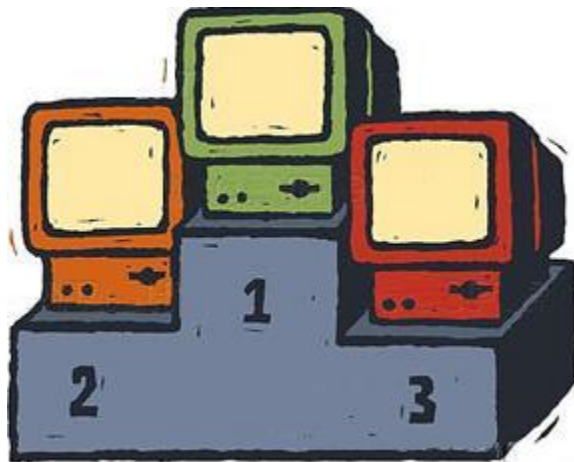
Time efficiency

Code algorithms and then compare execution times? The one that finishes sooner is more time efficient, right?

Time efficiency

Code algorithms and then compare execution times? The one that finishes sooner is more time efficient, right?

(problem: it is indeed more time efficient on a particular computer, but possibly not in general!)



Time efficiency



Code algorithms, run them, then compare execution times? The one that finishes sooner is more time efficient, right?

(problem: it is indeed more time efficient on a particular computer, but possibly not in general!)

when we change the hardware and software used to implement an algorithm, we can closely approximate the time required to solve a problem of size N (**input size**) by multiplying the previous time required by a constant. E.g., supercomputer faster than PC, but the speed factor doesn't depend on the input size

Big-O notation



estimate the number of operations an algorithm uses as its input grows

estimate the growth of a function without worrying about constant multipliers or smaller order terms

- do not have to worry about the hardware and software used to implement an algorithm
- assume that the different operations used in an algorithm take the same time, which simplifies the analysis considerably

Big-O notation

DEFINITION 1

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is read as “ $f(x)$ is big-oh of $g(x)$.”]

Remark: Intuitively, the definition that $f(x)$ is $O(g(x))$ says that $f(x)$ grows slower than some fixed multiple of $g(x)$ as x grows without bound.

Big-O notation: math proof example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

1. Set $k = 1$


$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

Big-O notation: math proof example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

1. Set $k = 1$
2. We have then that $x > 1$ (Remember our definition)



$|f(x)| \leq C|g(x)|$
whenever $x > k$. [This is

Big-O notation: math proof example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

1. Set $k = 1$
2. We have then that $x > 1$ (Remember our definition)
3. When $x > 1$ we also have that $x^2 > x$
So we have the chain: $x^2 > x > 1$

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

Big-O notation: math proof example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

1. Set $k = 1$
2. We have then that $x > 1$ (Remember our definition)
3. When $x > 1$ we also have that $x^2 > x$
So we have the chain: $x^2 > x > 1$
4. When we have $x > 1$ it also holds $2x < 2x^2$ (multiply both sides by $2x$)

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

Big-O notation: math proof example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

1. Set $k = 1$
2. We have then that $x > 1$ (Remember our definition)
3. When $x > 1$ we also have that $x^2 > x$
So we have the chain: $x^2 > x > 1$
4. When we have $x > 1$ it also holds $2x < 2x^2$ (multiply both sides by $2x$)
5. When we have $1 < x$ it also holds that $1 < x < x^2$

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

Big-O notation: math proof example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

1. Set $k = 1$
2. We have then that $x > 1$ (Remember our definition)
3. When $x > 1$ we also have that $x^2 > x$
So we have the chain: $x^2 > x > 1$
4. When we have $x > 1$ it also holds $2x < 2x^2$ (multiply both sides by $2x$)
5. When we have $1 < x$ it also holds that $1 < x < x^2$

So following all these chain inequalities we have that:

$$0 \leq \underbrace{x^2 + 2x + 1}_{f(x)} \leq \underbrace{x^2 + 2x^2 + x^2}_{Cg(x^2)} = 4x^2$$

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

Big-O notation: math proof example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

1. Set $k = 1$
2. We have then that $x > 1$ (Remember our definition)
3. When $x > 1$ we also have that $x^2 > x$
So we have the chain: $x^2 > x > 1$
4. When we have $x > 1$ it also holds $2x < 2x^2$ (multiply both sides by $2x$)
5. When we have $1 < x$ it also holds that $1 < x < x^2$

So following all these chain inequalities we have that:

$$0 \leq \underbrace{x^2 + 2x + 1}_{f(x)} \leq \underbrace{x^2 + 2x^2 + x^2}_{Cg(x^2)} = 4x^2$$

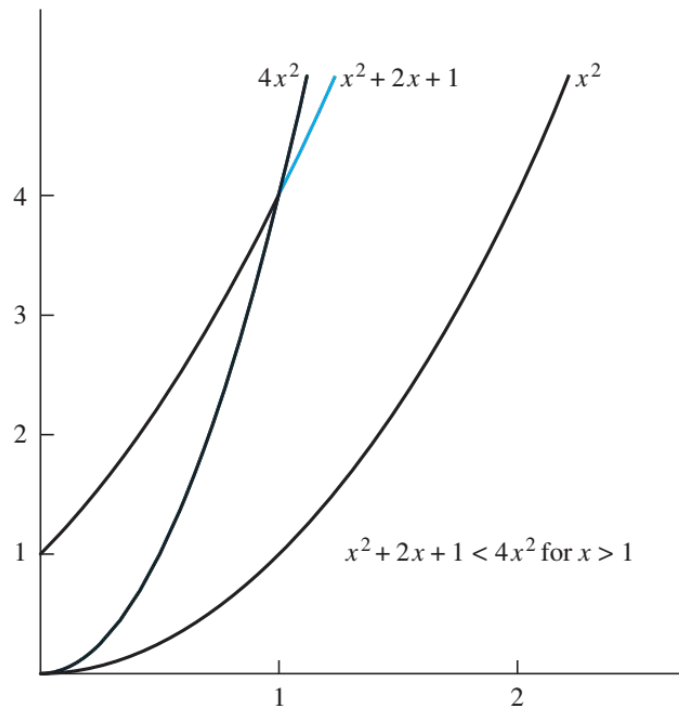
$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

So we found a constant $C = 4$
and constant $k = 1$ for which
our inequality holds

Big-O notation: math proof example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.



The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in blue.

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

So we found a constant $C = 4$ and constant $k = 1$ for which our inequality holds

FIGURE 1 The Function $x^2 + 2x + 1$ is $O(x^2)$.

Big-O notation: math proof example

Show that $7x^2$ is $O(x^3)$.

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

Big-O notation: math proof example

Show that $7x^2$ is $O(x^3)$.

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

1. Set $k = 1$

Big-O notation: math proof example

Show that $7x^2$ is $O(x^3)$.

1. Set $k = 1$
2. We have then that $x > 1$ (Remember our definition)

$|f(x)| \leq C|g(x)|$
whenever $x > k$. [This is

Big-O notation: math proof example

Show that $7x^2$ is $O(x^3)$.

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

1. Set $k = 1$
2. We have then that $x > 1$ (Remember our definition)
3. When $x > 1$ we also have that $x^3 > x^2 > x$

Big-O notation: math proof example

Show that $7x^2$ is $O(x^3)$.

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

1. Set $k = 1$
2. We have then that $x > 1$ (Remember our definition)
3. When $x > 1$ we also have that $x^3 > x^2 > x$
4. Also: $7x^3 > 7x^2 > 7x$

Big-O notation: math proof example

Show that $7x^2$ is $O(x^3)$.

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

1. Set $k = 1$
2. We have then that $x > 1$ (Remember our definition)
3. When $x > 1$ we also have that $x^3 > x^2 > x$
4. Also: $7x^3 > 7x^2 > 7x$
5. So, $f(x) = 7x^2 < 7x^3$ with **$C = 7$**

Big-O notation: math proof example

Show that n^2 is not $O(n)$.

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

1. To show that n^2 is not $O(n)$ we must prove that we cannot find constants C and k such that $n^2 \leq Cn$

Big-O notation: math proof example

Show that n^2 is not $O(n)$.

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

1. To show that n^2 is not $O(n)$ we must prove that we cannot find constants C and k such that $n^2 \leq Cn$
2. Assume the opposite that there are constants C and k such that (1) $n^2 \leq Cn$

Big-O notation: math proof example

Show that n^2 is not $O(n)$.

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

1. To show that n^2 is not $O(n)$ we must prove that we cannot find constants C and k such that $n^2 \leq Cn$
2. Assume the opposite that there are constants C and k such that (1) $n^2 \leq Cn$
3. If $n > 0$ we divide by n and get $n \leq C$

Big-O notation: math proof example

Show that n^2 is not $O(n)$.

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

1. To show that n^2 is not $O(n)$ we must prove that we cannot find constants C and k such that $n^2 \leq Cn$
2. Assume the opposite that there are constants C and k such that (1) $n^2 \leq Cn$
3. If $n > 0$ we divide by n and get $n \leq C$
4. No matter what C and k are, the inequality $n \leq C$ cannot hold **for all n** even when with $n > k$.

Big-O notation: math proof example

Show that n^2 is not $O(n)$.

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is

1. To show that n^2 is not $O(n)$ we must prove that we cannot find constants C and k such that $n^2 \leq Cn$
2. Assume the opposite that there are constants C and k such that (1) $n^2 \leq Cn$
3. If $n > 0$ we divide by n and get $n \leq C$
4. No matter what C and k are, the inequality $n \leq C$ cannot hold **for all n** even when with $n > k$.

E.g., set $k = 1$, I can always find some $n > 2$ that will be greater than any constant we set. For example, if $C = 10,000$, I can find n being 10,001

Big-O notation: math proof example (method 2)

Prove that $f(n) = n^2 + 2n + 1$ is $O(n^2)$

1. Choose $k = 1$
2. Assuming $n > 1$ find/derive a C such that
$$\frac{f(n)}{g(n)} \leq \frac{C g(n)}{g(n)} = C$$
3. This shows that $n > 1$ implies $f(n) \leq C g(n)$.
 - Keep in mind: $n > 1$ implies $1 < n$, $n < n^2$, $n^2 < n^3$, . . .
 - “Increase” numerator to “simplify” fraction

Big-O notation: math proof example (version 2)

Prove that $f(n) = n^2 + 2n + 1$ is $O(n^2)$

1. Choose $k = 1$

2. Assuming $n > 1$ find/derive a C such that $\frac{f(n)}{g(n)} \leq \frac{C g(n)}{g(n)} = C$

$$\frac{f(n)}{g(n)} = \frac{n^2 + 2n + 1}{n^2} < \frac{n^2 + 2n^2 + n^2}{n^2} = 4$$

Choose $C=4$. Note that $2n < 2n^2$ and $1 < n^2$.

$$n^2 + 2n + 1 \leq 4n^2 \text{ whenever } n > 1$$

Thus, $n^2 + 2n + 1$ is $O(n^2)$

Time efficiency: programming example

Hi-Lo Sequential Search

```
Set guess to 0
do
  Increment guess by 1
  Announce guess
while (guess is not correct)
```

Alice: "I'm thinking of a number between 1 and 1,000."

Bob: "Is it 1?"

Alice: "No ... it's higher."

Bob: "Is it 2?"

Alice: "No ... it's higher."

Bob: "Is it 3?"

Alice: rolls her eyes ...

Complexity cases: sequential search example

Hi-Lo Sequential Search

```
Set guess to 0
do
  Increment guess by 1
  Announce guess
while (guess is not correct)
```

Best case complexity: complexity when we're lucky (e.g. Alice thinks of 1)

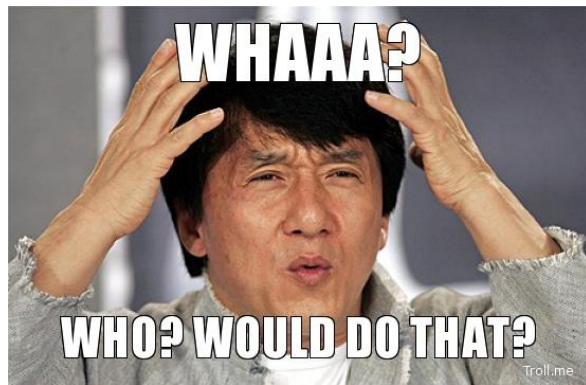
Average case complexity: the average number of steps required, considering all possible inputs. $(1 + 1000) / 2 = 500.5$ guesses

Worst case complexity: the highest number of steps that an algorithm would require. E.g., if Alice thinks number 1000. The most interesting measure!

Size of input: sequential search example

Alice: "OK. I'm thinking of a number between 1 and 1,000,000."

Bob: blinks



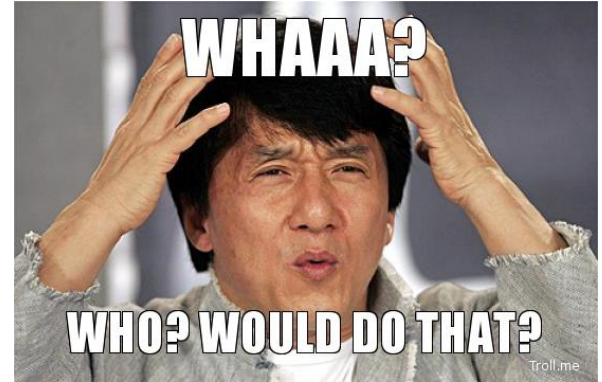
Size of input: sequential search example

Alice: "OK. I'm thinking of a number between 1 and 1,000,000."

Bob: blinks

What is the worst case now for our sequential search?

If the game is to guess a number between 1 and N , the size of the input is N , and for the sequential search algorithm, the worst case number of guesses required is also N .



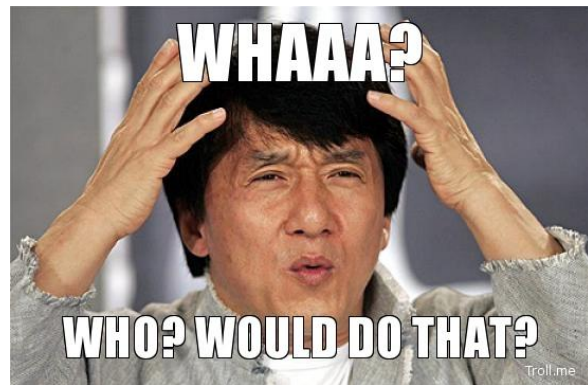
Size of input: sequential search example

Alice: "OK. I'm thinking of a number between 1 and 1,000,000."

Bob: blinks

What is the worst case now for our sequential search?

If the game is to guess a number between 1 and N , the size of the input is N , and for the sequential search algorithm, the worst case number of guesses required is also N .



it makes sense to speak of an algorithm's efficiency in terms of the input size, and to use that size as a parameter when describing the efficiency of the algorithm.

Comparing algorithms

Carlos: "What's up?"

Bob: "Alice wants me to guess a number between 1 and 1,000,000. No way."

Carlos: "Hmmm. I'll try. Is it 500,000?"

Alice: "No, it's lower."

Carlos: "Is it 250,000?"

Alice: "No, it's higher."

Carlos: "Is it 375,000?"

...

How is this approach different from the previous approach?

Comparing algorithms

Carlos: "What's up?"

Bob: "Alice wants me to guess a number between 1 and 1,000,000. No way."

Carlos: "Hmmm. I'll try. Is it 500,000?"

Alice: "No, it's lower."

Carlos: "Is it 250,000?"

Alice: "No, it's higher."

Carlos: "Is it 375,000?"

...

How is this approach different from the previous approach?

Binary search! Eliminate half of the numbers each time!

Binary search: complexity?

Hi-Lo Binary Search(N)⁴

```
Set range to 1 . . . N
do
  Set guess to middle of range
  Announce guess
  if (guess was too high)
    Set range to first half of range
  if (guess was too low)
    Set range to second half of range
while (guess is not correct)
```

Each time an incorrect guess is made, the remaining range of possible numbers is cut in half. So, another way of asking this is “How many times can you reduce N by half, before you get down to 1?”

Binary search: complexity?

Hi-Lo Binary Search(N)⁴

```
Set range to 1 . . . N
do
  Set guess to middle of range
  Announce guess
  if (guess was too high)
    Set range to first half of range
  if (guess was too low)
    Set range to second half of range
while (guess is not correct)
```

Each time an incorrect guess is made, the remaining range of possible numbers is cut in half. So, another way of asking this is “How many times can you reduce N by half, before you get down to 1?” **$\log_2 N$**

⁴ Recall that $\log_2 N$ is the power that you raise 2 to, in order to get N . For example, $\log_2 8 = 3$ because $2^3 = 8$. But another way of looking at this is to consider that $\log_2 N$ is the number of times you can cut N in half before reaching 1. We can cut 8 in half 3 times: $8 \rightarrow 4 \rightarrow 2 \rightarrow 1$.

Binary search: complexity?

Carlos: "What's up?"

Bob: "Alice wants me to guess a number between 1 and 1,000,000. No way."

Carlos: "Hmmm. I'll try. Is it 500,000?"

Alice: "No, it's lower."

Carlos: "Is it 250,000?"

Alice: "No, it's higher."

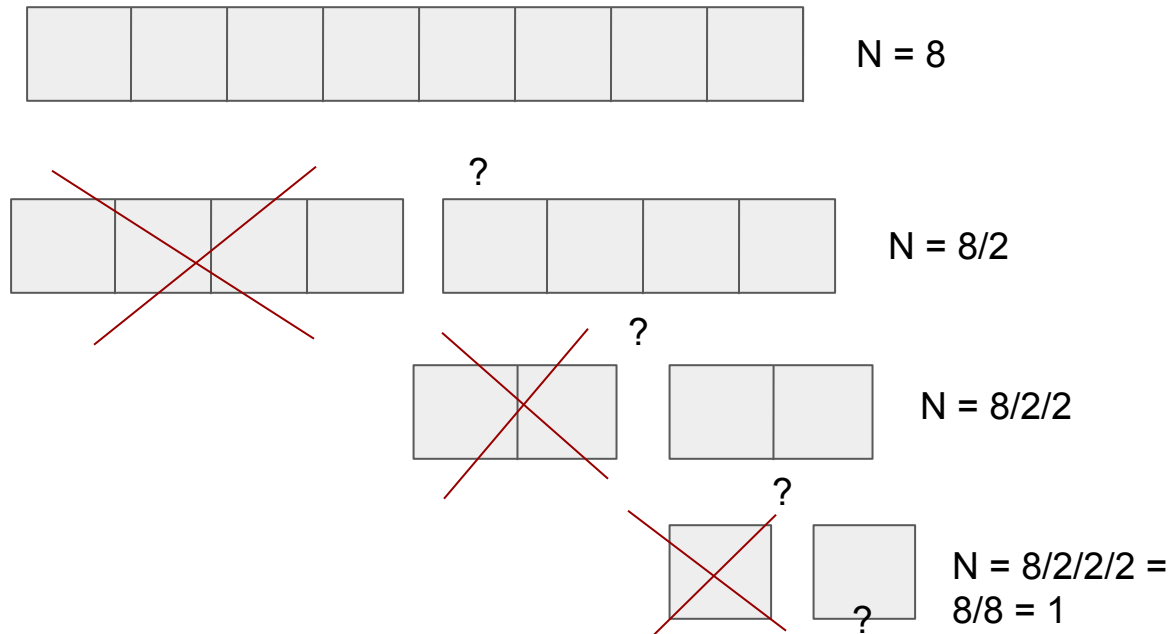
Carlos: "Is it 375,000?"

...

Each time an incorrect guess is made, the remaining range of possible numbers is cut in half. So, another way of asking this is "How many times can you reduce N by half, before you get down to 1?" $\log_2 N$

Note: for an input size of 1,000,000 this equates to 20 guesses, as opposed to 1,000,000 guesses for the sequential search

Binary search: why is $O(\log_2 N)$?



How many guesses we need? 4 (we also need to guess the final element)

$$8/2^3 = 1$$

$$N/2^k = 1$$

$$N = 2^k$$

Which means $k = \log_2 N$

Order of growth of our two search algorithms

Size	Sequential Search	Binary Search
N	$2N + 1$ steps	$5 \log_2 N + 3$ steps
2	5	8
4	9	13
8	17	18
16	33	23
32	65	28
1,024	2,049	53
1,000,000	2,000,001	98
1,000,000,000	2,000,000,001	148

each time a guess is made the value is incremented and announced. In the worst case this will require two steps (increment, announce) for each guess plus the one initial step resulting in a total of **$2N + 1$** steps

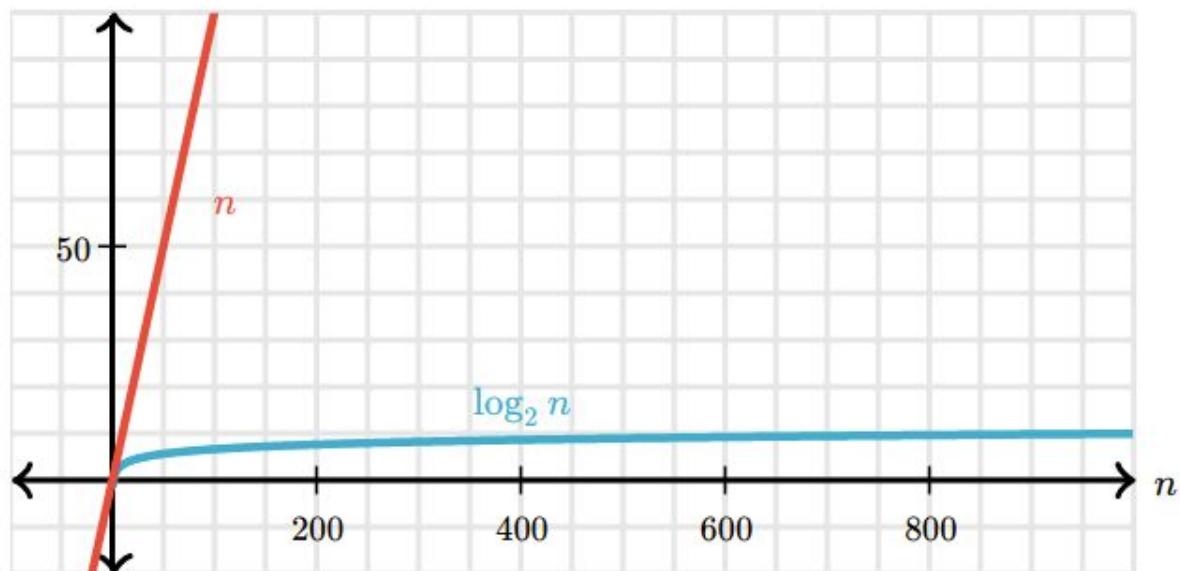
set the low value and high value of the range, for each guess **add** together the low value and high value, **divide**, **round**, **announce** the guess, and **adjust** the range (5 operations) + 2 initial steps and 1 final step

Order of growth of our two search algorithms

Size	Sequential Search	Binary Search
N	$2N + 1$ steps	$5 \log_2 N + 3$ steps
2	5	8
4	9	13
8	17	18
16	33	23
32	65	28
1,024	2,049	53
1,000,000	2,000,001	98
1,000,000,000	2,000,000,001	148

These detailed counts of steps don't give us much information

Compare n vs $\log_2 n$ below:



Order of growth of our two search algorithms

Size	Sequential Search	Binary Search
N	$2N + 1$ steps	$5 \log_2 N + 3$ steps
2	5	8
4	9	13
8	17	18
16	33	23
32	65	28
1,024	2,049	53
1,000,000	2,000,001	98
1,000,000,000	2,000,000,001	148

VS

Size	Sequential Search	Binary Search
N	N steps	$\log_2 N$ steps
2	2	1
4	4	2
8	8	3
16	16	4
32	32	5
1,024	1,024	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30

In computer science the order of growth is reported for algorithm analysis

Perhaps the number of steps needed for the sequential search is $f(N) = 2N + 1$, but we say that it is “**Order of Growth N**” or even **$O(N)$** read as “**Oh of N**” or “**Order N**”.

$2N^5 + N^2 + 37$ is $O(N^5)$
 $2N^2 \log_2 N + 3N^2$ is $O(N^2 \log_2 N)$
 $1 + N^2 + N^3 + N^4$ is $O(N^4)$

Demo: selection sort

Given an unsorted list of elements, the algorithm **scans** through the list and finds the smallest element. It then selects that element and **swaps** it with the first element. It **scans** the list again to find the second smallest element, again selecting it and **swapping** it with the second element. As the algorithm **repeatedly selects** the next smallest element and **swaps** it into its “correct” position, the sorted section of the list grows larger and the unsorted section of the list grows smaller, until eventually the entire list is sorted.

Selection Sort(values[0 . . . N - 1])

for current going from 0 to $N - 2$

 Set minIndex to index of smallest unsorted element

 Swap the elements at indices current and minIndex

Demo: selection sort

Selection Sort(values[0 . . . N - 1])

for current going from 0 to $N - 2$

 Set minIndex to index of smallest unsorted element

 Swap the elements at indices current and minIndex

How do we find the smallest element?

How do we swap two elements?

How many loops do we need? Why?

What is the best case, average case and worst case complexity?

What is the order of growth?

Demo: selection sort

(a)

	values			
[0]	126			
[1]	43	①	<126?	YES
[2]	26	②	<43?	YES
[3]	1	③	<26?	YES
[4]	113	④	<1?	NO

**How many
comparisons in the
first iteration?**

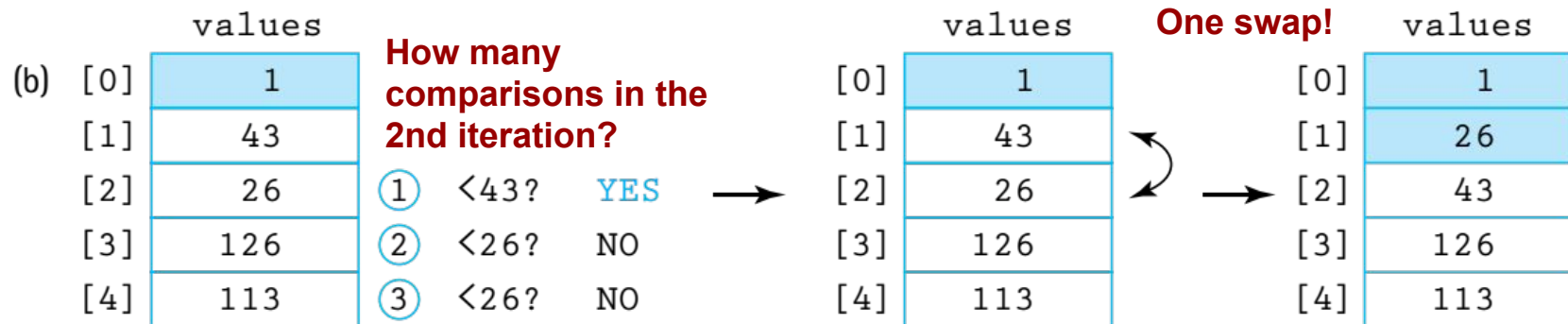
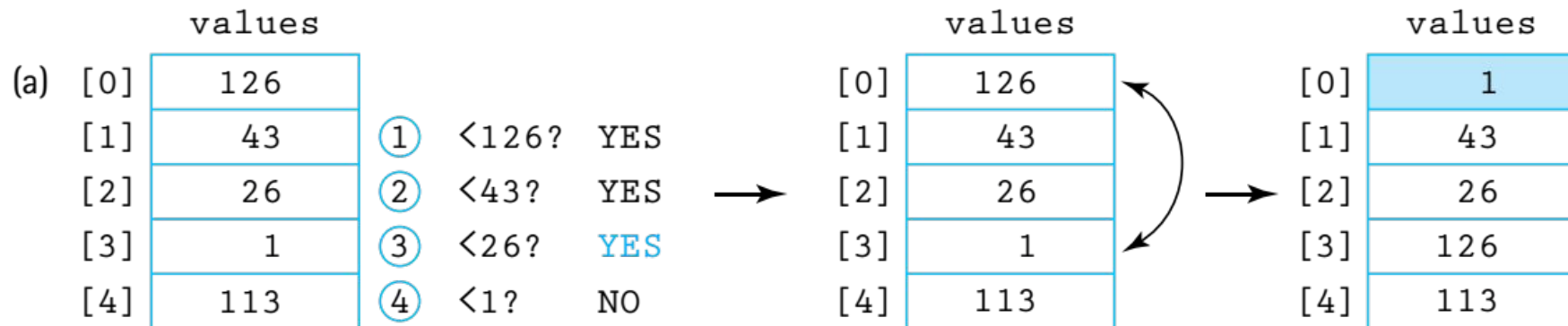


One swap!

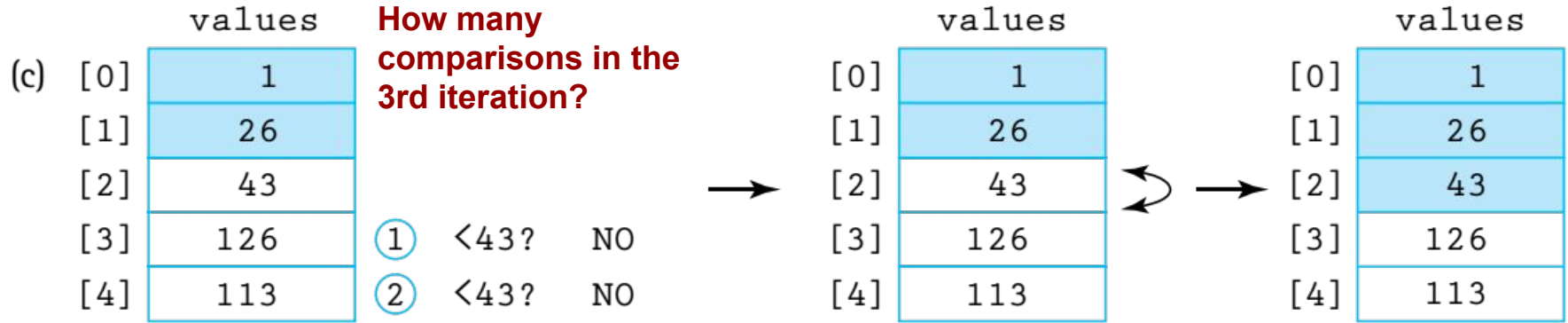
	values			
[0]	126			
[1]	43			
[2]	26			
[3]	1			
[4]	113			

	values
[0]	1
[1]	43
[2]	26
[3]	126
[4]	113

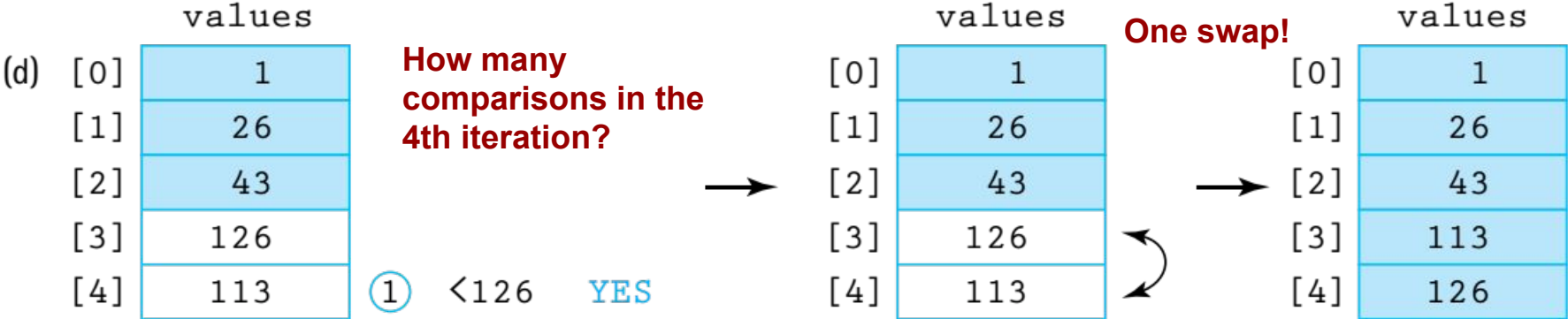
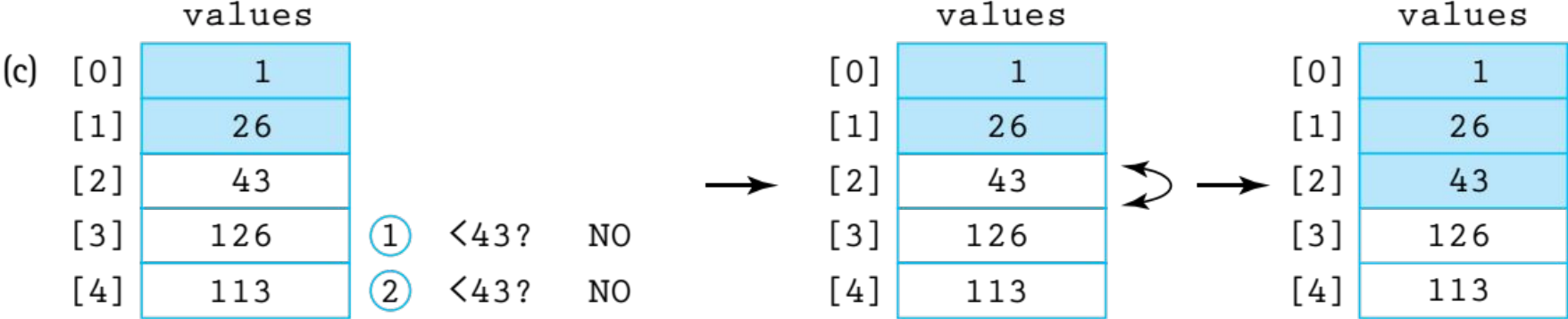
Demo: selection sort



Demo: selection sort



Demo: selection sort



Demo: selection sort

Number of elements = 5

Number of comparisons = 10

Number of swaps = 4

(doesn't give us a general
answer; what is the number of
elements changes)

Demo: selection sort

Number of elements = 5

Number of comparisons = 10

Number of swaps = 4

(doesn't give us a general answer; what is the number of elements changes)

Selection Sort(values[0 . . . N - 1])⁷

for current going from 0 to $N - 2$

 Set minIndex to current

 for check going from (current + 1) to ($N - 1$)

 if (values[check] < values[minIndex])

 Set minIndex to check

 Swap the elements at indices current and minIndex

Demo: selection sort

Number of elements (N) = 5

Number of comparisons = 10

Number of swaps = 4

(doesn't give us a general answer; what is the number of elements changes)



N - any number of elements

For each element in the range of 0 to $N - 2$ (the last element will always be sorted by default) we perform a different number of comparisons (depending on current)

Demo: selection sort

Number of elements (N) = 5

Number of comparisons = 10

Number of swaps = 4

(doesn't give us a general answer; what is the number of elements changes)



N - any number of elements

For each element in the range of 0 to $N - 2$ (the last element will always be sorted by default) we perform a different number of comparisons (depending on current)

For current = 0 we have (?) comparisons

For current = 1 we have (?) comparisons

...

For current = $N - 1$ we have (?) comparisons

For current = $N - 2$ we have (?) comparisons

Demo: selection sort

Number of elements (N) = 5

Number of comparisons = 10

Number of swaps = 4

(doesn't give us a general answer; what is the number of elements changes)



N - any number of elements

For each element in the range of 0 to $N - 2$ (the last element will always be sorted by default) we perform a different number of comparisons (depending on current)

For current = 0 we have **$N - 1$** comparisons

For current = 1 we have **$N - 2$** comparisons

...

For current = $N - 1$ we have **2** comparisons

For current = $N - 2$ we have **1** comparisons

What is the total of comparisons for N elements?

Demo: selection sort

Number of elements (N) = 5

Number of comparisons = 10

Number of swaps = 4

(doesn't give us a general answer; what is the number of elements changes)



N - any number of elements

For each element in the range of 0 to N - 2 (the last element will always be sorted by default) we perform a different number of comparisons (depending on current)

What is the total of comparisons for N elements?

$(N - 1) + (N - 2) + (N - 3) + \dots + 2 + 1$

or $N(N-1)/2$

Demo: selection sort

N - any number of elements
For each element in the range of 0 to $N - 2$ (the last element will always be sorted by default) we perform a different number of comparisons (depending on current)

What is the total of comparisons for N elements?

**$(N - 1) + (N - 2) + (N - 3) + \dots + 2 + 1$
or $N(N-1)/2$**

Order of growth?

Demo: selection sort

N - any number of elements
For each element in the range of 0 to $N - 2$ (the last element will always be sorted by default) we perform a different number of comparisons (depending on current)

What is the total of comparisons for N elements?

**$(N - 1) + (N - 2) + (N - 3) + \dots + 2 + 1$
or $N(N-1)/2$**

Order of growth?

$1/2N^2 - 1/2N$

What is the fastest increasing part?

Demo: selection sort

N - any number of elements
For each element in the range of 0 to $N - 2$ (the last element will always be sorted by default) we perform a different number of comparisons (depending on current)

What is the total of comparisons for N elements?

$(N - 1) + (N - 2) + (N - 3) + \dots + 2 + 1$
or $N(N-1)/2$

Order of growth?

$$1/2N^2 - 1/2N$$

What is the fastest increasing part?

$$1/2N^2$$

(we can discard the constant)

And we are left with $O(N^2)$

Demo: selection sort

N - any number of elements
For each element in the range of 0 to $N - 2$ (the last element will always be sorted by default) we perform a different number of comparisons (depending on current)

Hint: you see two nested loops in an algorithm with same conditions, most likely will have N^2 order of growth

What is the total of comparisons for N elements?

$(N - 1) + (N - 2) + (N - 3) + \dots + 2 + 1$
or $N(N-1)/2$

Order of growth?

$1/2N^2 - 1/2N$

What is the fastest increasing part?

$1/2N^2$

(we can discard the constant)

And we are left with $O(N^2)$

Common orders of Growth

$O(1)$ is called “**bounded time.**” The amount of work is bounded by a constant and is not dependent on the size of the problem. Initializing a sum to 0 is $O(1)$. Although bounded time is often called constant time, the amount of work is not necessarily constant. It is, however, bounded by a constant.

Common orders of Growth

$O(1)$ is called “**bounded time.**” The amount of work is bounded by a constant and is not dependent on the size of the problem. Initializing a sum to 0 is $O(1)$. Although bounded time is often called constant time, the amount of work is not necessarily constant. It is, however, bounded by a constant.

$O(\log_2 N)$ is called “logarithmic time.” The amount of work depends on the logarithm, in base 2, of the size of the problem. Algorithms that successively cut the amount of data to be processed in half at each step, like the binary search algorithm, typically fall into this category. Note that in the world of computing we often just say “log N” when we mean $\log_2 N$. The base 2 is assumed.

Common orders of Growth

$O(N)$ is called “linear time.” The amount of work is some constant times the size of the problem. Algorithms that work through all the data one time to arrive at a conclusion, like the sequential search algorithm, typically fall into this category.

Common orders of Growth

$O(N)$ is called “linear time.” The amount of work is some constant times the size of the problem. Algorithms that work through all the data one time to arrive at a conclusion, like the sequential search algorithm, typically fall into this category.

$O(N \log_2 N)$ is called (for lack of a better term) “N log N time”. Algorithms of this type typically involve applying a logarithmic algorithm N times. The better sorting algorithms, such as Quicksort have N log N complexity.

Common orders of Growth

$O(N)$ is called “linear time.” The amount of work is some constant times the size of the problem. Algorithms that work through all the data one time to arrive at a conclusion, like the sequential search algorithm, typically fall into this category.

$O(N \log_2 N)$ is called (for lack of a better term) “N log N time”. Algorithms of this type typically involve applying a logarithmic algorithm N times. The better sorting algorithms, such as Quicksort have N log N complexity.

$O(N^2)$ is called “**quadratic time.**” Algorithms of this type typically involve applying a linear algorithm N times. Most simple sorting algorithms, such as the Selection Sort algorithm, are $O(N^2)$ algorithms.

Common orders of Growth

This pattern of increasingly time complex algorithms continues with $O(N^2 \log_2 N)$, $O(N^3)$, $O(N^3 \log_2 N)$, and so on

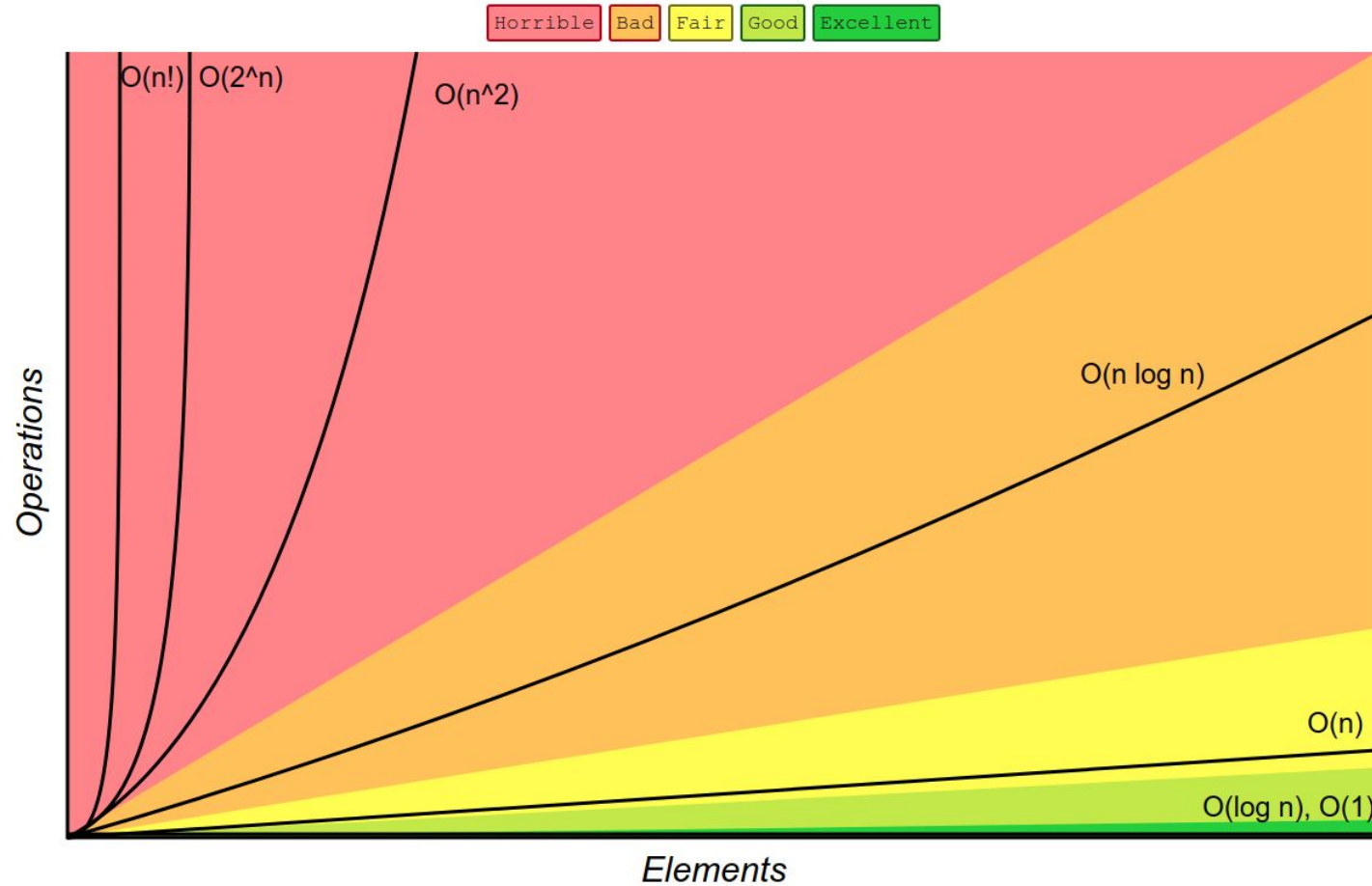
$O(2^N)$ is called “**exponential time.**” These algorithms are extremely costly and require more time for large problems than any of the polynomial time algorithms previously listed. An example of a problem for which the best known solution is exponential is the traveling salesman problem—given a set of cities and a set of roads that connect some of them, plus the lengths of the roads, find a route that visits every city exactly once and minimizes total travel distance.

Common orders of Growth

Table 1.3 Comparison of Rates of Growth

N	$\log_2 N$	$N \log_2 N$	N^2	N^3	2^N
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4,096	65,536
32	5	160	1,024	32,768	4,294,967,296
64	6	384	4,096	262,144	approximately 20 billion billion
128	7	896	16,384	2,097,152	It would take a fast computer a trillion billion years to execute this many instructions
256	8	2,048	65,536	16,777,216	Do not ask!

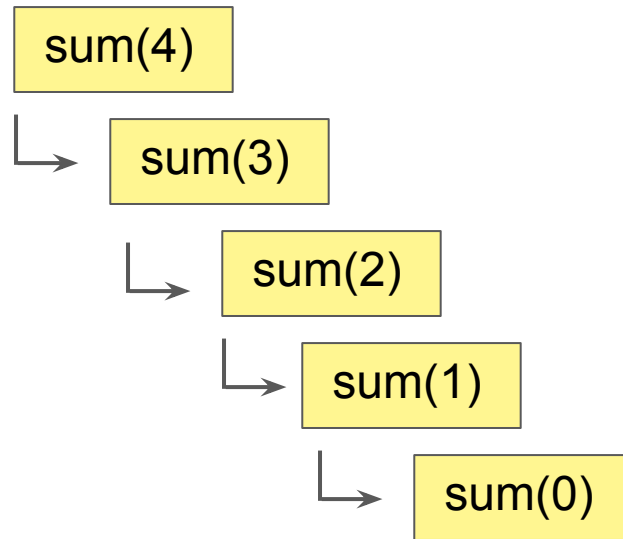
Big-O Complexity Chart



Space complexity: $O(n)$ space/time recursion example

```
int sum (int n) {  
    if (n <= 0) {  
        return 0;  
    }  
    return n + sum(n - 1);  
}
```

Each call adds a level to the stack:



Space complexity: recursion example (however)

```
int pairSumSequence(int n){
    int sum = 0;
    for(int i = 0; i < n; i++) {
        sum += pairSum(i, i+1);
    }
    return sum;
}

int pairSum (int a, int b) {
    return a + b;
}
```

Even though you have n calls doesn't mean it takes $O(n)$ space!

There will be roughly $O(n)$ calls to `pairSum`. However those calls do not exist simultaneously on the call stack, so you need only **$O(1)$** space.