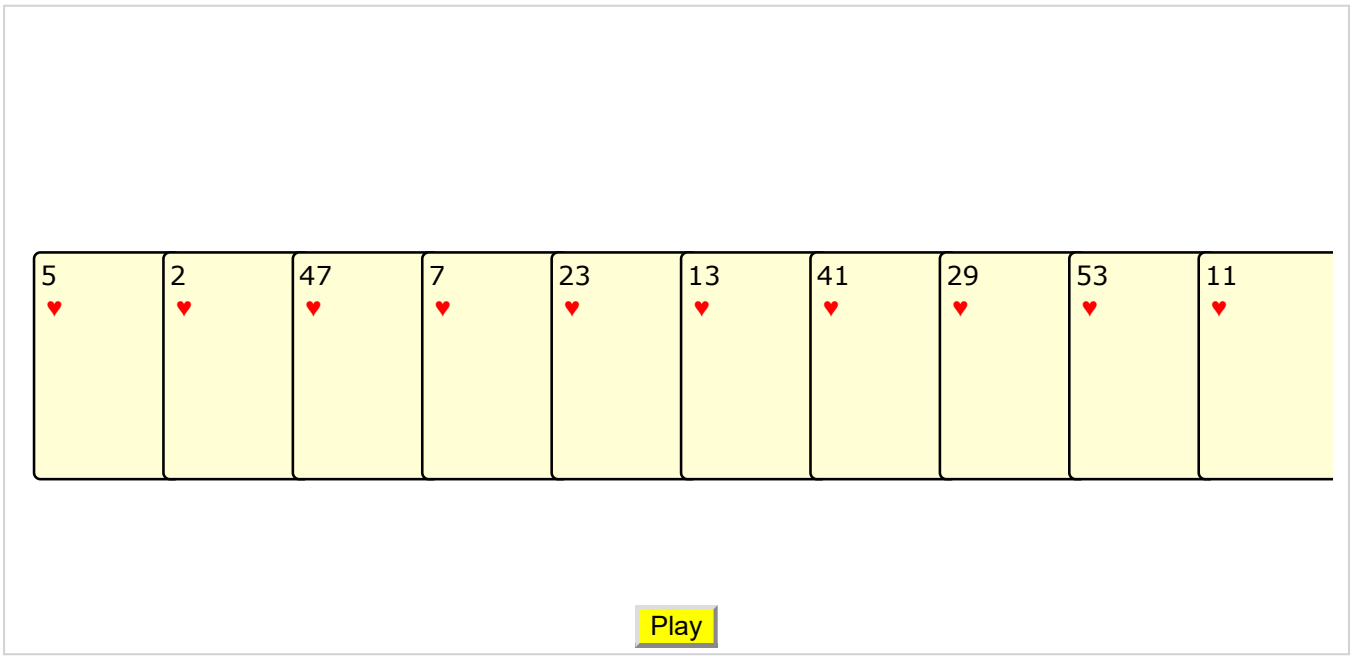
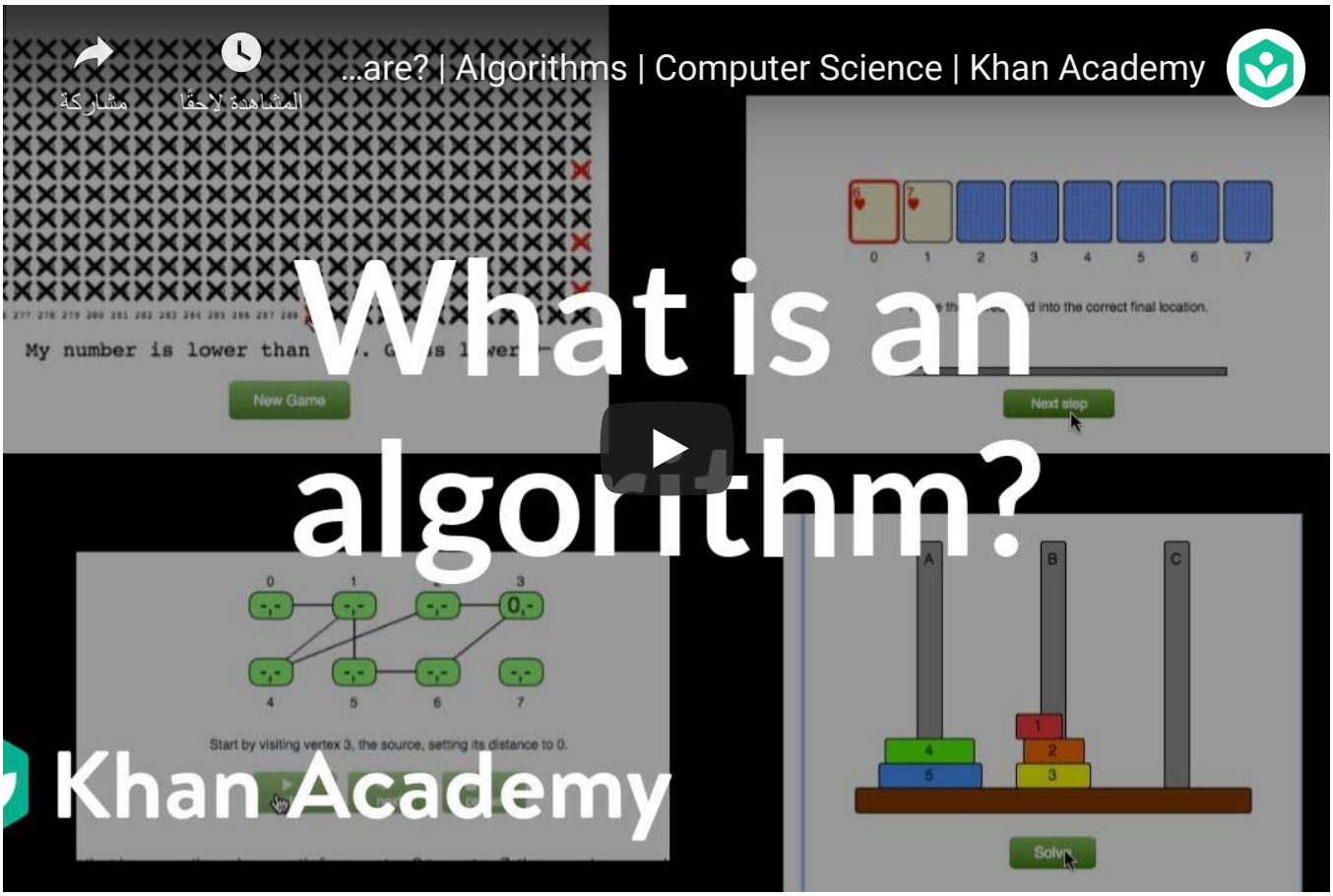
What is an algorithm and why should you care?

Algorithms are a fundamental part of computer programs. Whether you are multiplying two numbers or processing images sent by spacecrafts billions of mile away, you need algorithms to solve the problem at hand correctly and efficiently. Without going into any details, let's watch an algorithm in action. Click **Play** to see how are we sorting a deck of cards with the help of an algorithm known as **Insertion Sort** (which we will look in detail soon).

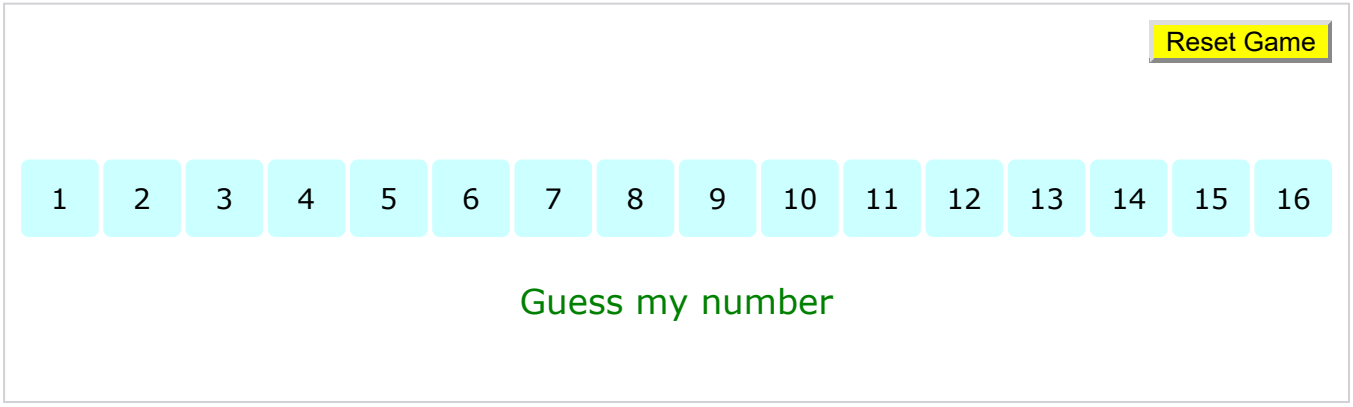


Now, let's look at a quick video about what an algorithm is.


...are? | Algorithms | Computer Science | Khan Academy
مشاركة
المشاهدة لاحقًا
What is an algorithm?
My number is lower than
New Game
Next step
Start by visiting vertex 3, the source, setting its distance to 0.
Khan Academy

A Guessing Game

Let's play a little game to give you an idea of how different algorithms for the same problem can have wildly different efficiencies. The computer is going to randomly select an integer from 1 to 16. You have to guess the number by making guesses until you find the number that the computer chose. Let's begin.



Maybe you guessed 1, then 2, then 3, then 4, and so on, until you guessed the right number. We call this approach **linear search**, because you guess all the numbers as if they were lined up in a row. It would work. But what is the highest number of guesses you could need? If the computer selects 16, you would need 16 guesses. Then again, you could be really lucky, which would be when the computer selects 1 and you get the number on your first guess. How about on average? If the computer is equally likely to select any number from 1 to 16, then on average you'll need 8 guesses.

But you could do something more efficient than just guessing 1, 2, 3, 4, ..., right? Since the computer tells you whether a guess is too low, too high, or correct, you can start off by guessing 15. If the number that the computer selected is less than 15, then because you know that 15 is too high, you can eliminate all the numbers from 15 to 30 from further consideration. If the number selected by the computer is greater than 15, then you can eliminate 1 through 15. Either way, you can eliminate about half the numbers. On your

next guess, eliminate half of the remaining numbers. Keep going, always eliminating half of the remaining numbers. We call this halving approach

binary search, and no matter which number from 1 to 30 the computer has selected, you should be able to find the number in at most 5 guesses with this technique.

Here, try it for a number from 1 to 300. You should need no more than 9 guesses.

Reset Game

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220
221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240
241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260
261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280
281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300

Guess my number

How many guesses did it take you to find the number this time? Why should you never need more than 9 guesses? (Can you think of a mathematical explanation)?

We'll return to binary search, and we'll see how you can use it to efficiently search for an item in an array. But first, let's look at an algorithm for a trickier problem.

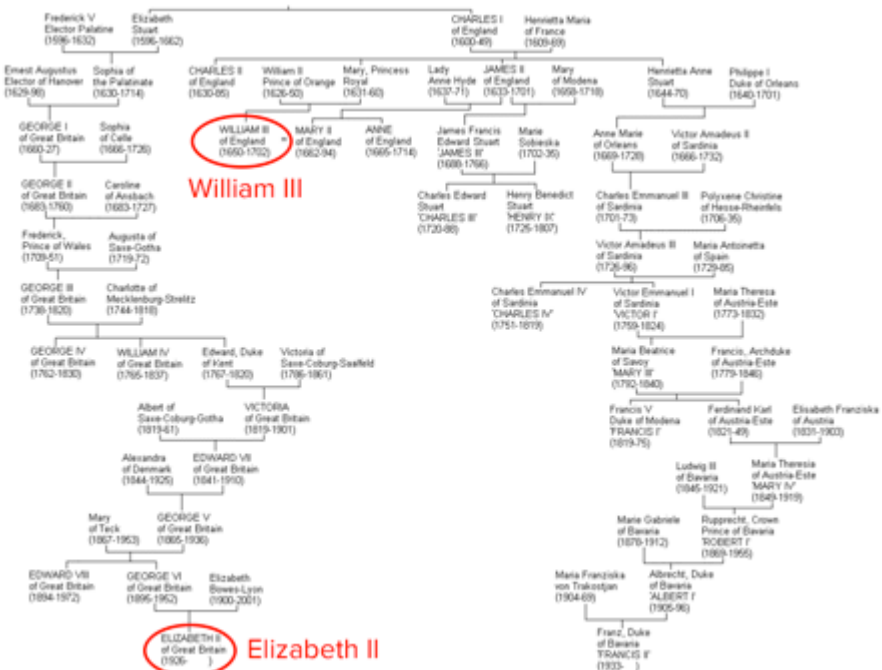
Route-finding

Sometimes, very different-sounding problems turn out to be similar when you think about how to solve them. What do Pac-Man, the royal family of Britain, and driving to Orlando have in common? They all involve route-finding or path-search problems:

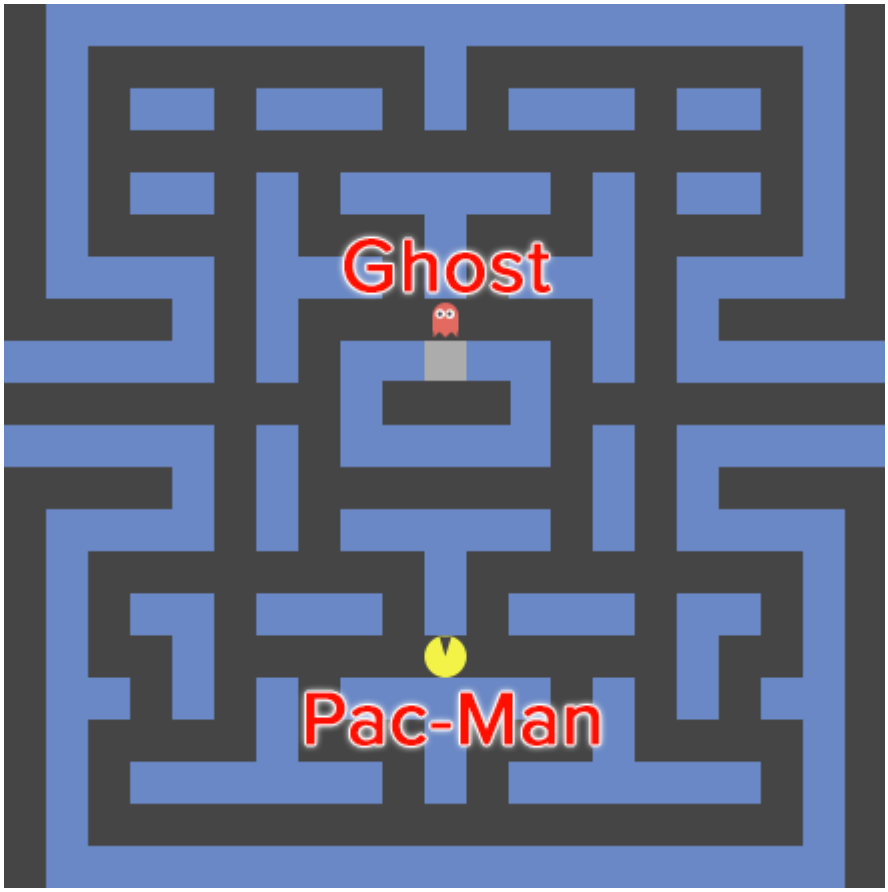
- How is the current Prince William related to King William III, who endowed the College of William and Mary in 1693?
- What path should a ghost follow to get to Pac-Man as quickly as possible?
- What's the best way to drive from Dallas, Texas to Orlando, Florida?

We have to be given some information to answer any of these questions.

For example, a family tree of the royal family of Britain would show connections between people who were directly related. Prince William is the son of Charles Philip Arthur Windsor. Charles is the son of Queen Elizabeth II. The problem is to find a short chain on the family tree connecting Prince William and William III, using these direct connections. As you can see from the tree below, it might take quite a few connections.



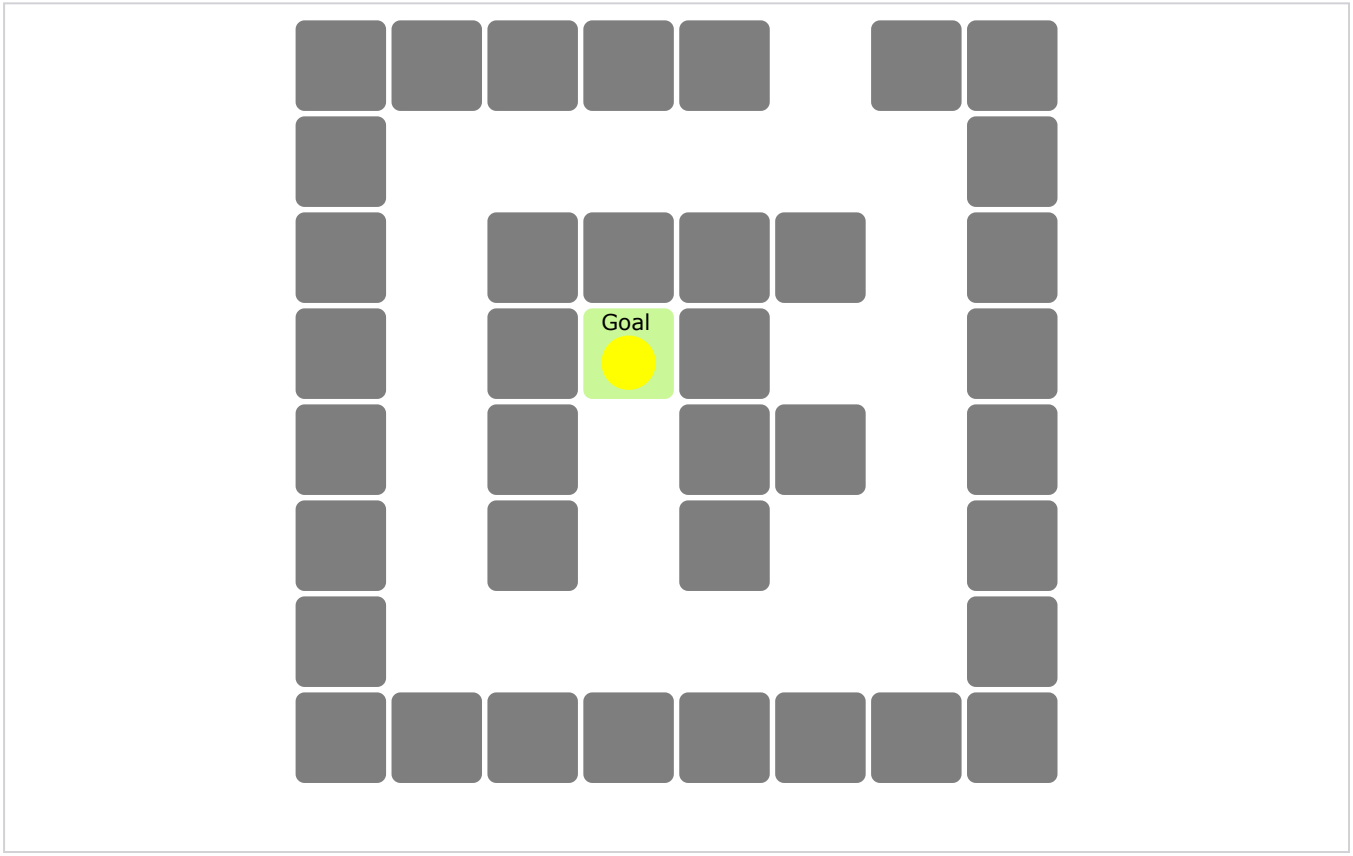
For Pac-Man, we need a map of the maze. This map shows connections between adjacent open squares in the maze—or lack of connections, if there is a wall in between—and the problem is to find a path along black squares that leads the ghost to Pac-Man.



In order to find a driving route from Dallas to Orlando, we might use a map of the United States, showing connections, roads, between nearby cities. No single road directly connects Dallas to Orlando, but several sequences of roads do.

Exploring a maze

Let's look more deeply at something like Pac-Man, a computer game in which the main character is controlled by clicking on destinations in a maze. The game is below. Try clicking on a few locations to move the character, represented by the yellow circle, to the goal, represented by the green square.



Notice how the character moved to the goal? To make that happen, the program needs to determine the precise set of movements that the character should follow to get to where the user clicked and then animate those movements. There may be multiple paths for the character to follow, and the program needs to choose the best of those paths.

Before deciding on an algorithm, the movement rules first need to be established: walls are made of gray squares and legal locations to travel are empty. In each step, the character can move from one square to an adjacent square. This character, like a chess rook, cannot move diagonally.

Here's the idea behind the algorithm that this program uses: move 1 square closer to the goal—the place the user clicked on—in each step. But what does "closer to the goal" mean? Traveling in a straight line toward the goal will often cause the character to smack into a wall. The algorithm needs to determine which of the surrounding squares are indeed "closer to the goal",

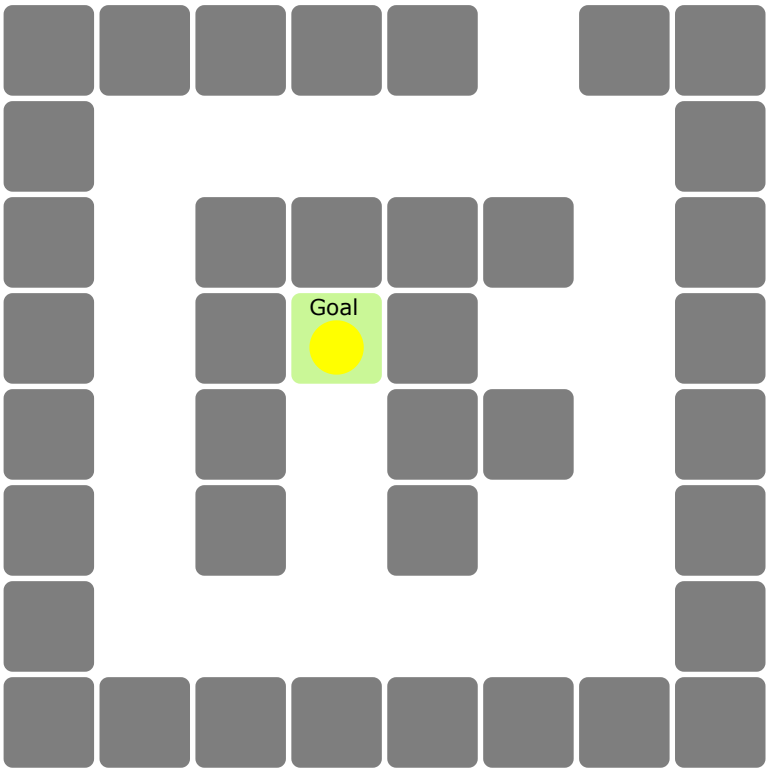
and we can do that by assigning a "cost" to each square that represents the

minimum number of steps the character would have to take to get from that vertex to the goal. Here's an algorithm for assigning a cost to each square:

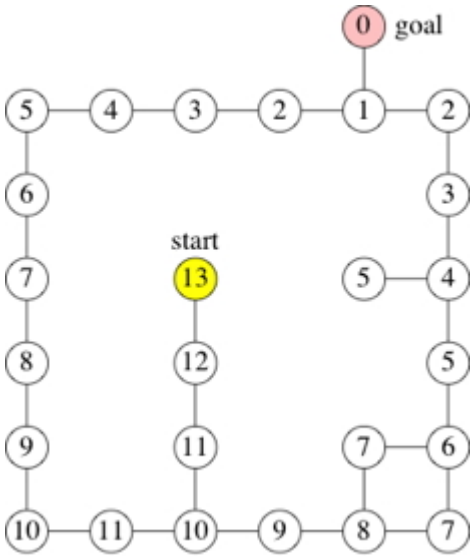
1. Start on the goal square. How far is the goal from the goal? Zero steps, mark the goal with the number 0.
2. Find all squares in the maze that are exactly one step away from the goal. Mark them with the number 1. In this maze, if the goal is the exit square, then there is only one square that is exactly one step away.
3. Now find all squares in the maze that are exactly two steps away from the goal. These squares are one step away from those marked 1 and have not yet been marked. Mark these squares with the number 2.
4. Mark all squares in the maze that are exactly three steps away from the goal. These squares are one step away from those marked 2 and have not yet been marked. Mark these squares with the number 3.
5. Keep marking squares in the maze in order of increasing distance from the goal. After marking squares with the number **k**, mark with the number **k+1** all squares that are one step away from those marked **k** and have not yet been marked.

Eventually, the algorithm marks the square where the character starts. The program can then find a path to the goal by choosing a sequence of squares from the start such that the numbers on the squares always decrease along the path. If you view the number as the height of the square, it would be like going downhill.

You can play through the cost-marking algorithm below.



What if the user were trying to get from the start square to the goal? Using the square-marking algorithm, the start square is 13 steps away from the goal. Here's a picture showing the connections between possible locations for the character, the start, the goal, and the shortest distance of each location from the goal:



There's a square immediately to the south of the start (row four, column three) that is only 12 steps from the goal. So the first move is "south". South of that square is an 11. South again. South again to a 10. Then east to a 9. East twice more to a 7, then north five times to a 2. Finish up by going west once, to

a 1, and finally north once, to the goal.

We won't discuss exactly how to implement this maze search algorithm right now, but you might find it fun to think about how you might represent the maze and the character and how you might implement the algorithm (of course in a programming language of your choice).

Binary Search

Binary search is an efficient algorithm for finding an item from an ordered list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one. We used binary search in the guessing game in the introductory tutorial.

One of the most common ways to use binary search is to find an item in an array. For example, the Tycho-2 star catalog contains information about the brightest 2,539,913 stars in our galaxy. Suppose that you want to search the catalog for a particular star, based on the star's name. If the program examined every star in the star catalog in order starting with the first, an algorithm called linear search, the computer might have to examine all 2,539,913 stars to find the star you were looking for, in the worst case. If the catalog were sorted alphabetically by star names, binary search would not have to examine more than 22 stars, even in the worst case.

The next few articles discuss how to describe the algorithm carefully, how to implement the algorithm in JavaScript, and how to analyze efficiency.

Pseudocode for binary search

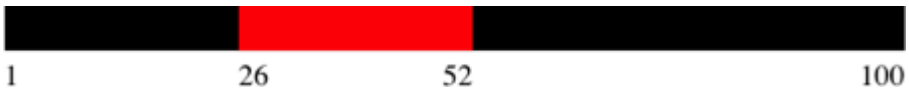
When describing an algorithm to a fellow human being, an incomplete description is often good enough. Some details may be left out of a recipe for a cake; the recipe assumes that you know how to open the refrigerator to get the eggs out and that you know how to crack the eggs. People might intuitively know how to fill in the missing details, but computer programs do not. That's why we need to describe computer algorithms completely.

In order to implement an algorithm in a programming language, you will need to understand an algorithm down to the details. What are the inputs to the problem? The outputs? What variables should be created, and what initial

values should they have? What intermediate steps should be taken to compute other values and to ultimately compute the output? Do these steps repeat instructions that can be written in simplified form using a loop?

Let's look at how to describe binary search carefully. The main idea of binary search is to keep track of the current range of reasonable guesses. Let's say that I'm thinking of a number between one and 100, just like the **guessing game**. If you've already guessed 25 and I told you my number was higher, and you've already guessed 81 and I told you my number was lower, then the numbers in the range from 26 to 80 are the only reasonable guesses. Here, the red section of the number line contains the reasonable guesses, and the black section shows the guesses that we've ruled out.

In each turn, you choose a guess that divides the set of reasonable guesses into two ranges of roughly the same size. If your guess is not correct, then I tell you whether it's too high or too low, and you can eliminate about half of the reasonable guesses. For example, if the current range of reasonable guesses is 26 to 80, you would guess the halfway point, **(26+80)/2**, or **53**. If I then tell you that 53 is too high, you can eliminate all numbers from 53 to 80, leaving 26 to 52 as the new range of reasonable guesses, halving the size of the range.



For the guessing game, we can keep track of the set of reasonable guesses using a few variables. Let the variable min be the current minimum reasonable guess for this round, and let the variable max be the current maximum reasonable guess. The input to the problem is the number n, the highest possible number that your opponent is thinking of. We assume that the lowest possible number is one, but it would be easy to modify the algorithm to take the lowest possible number as a second input.

Here's a pseudocode description of binary search:

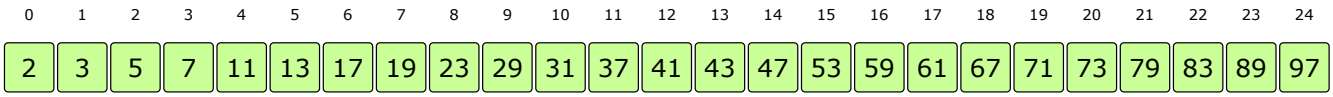
1. Let min = 1 and max = n.
2. Guess the average of max and min, rounded down so that it is an integer

2. Guess the average of max and min, rounded down so that it is an integer.
3. If you guessed the number, stop. You found it!
4. If the guess was too low, set min to be one larger than the guess.
5. If the guess was too high, set max to be one smaller than the guess.
6. Go back to step two.

We could make this pseudocode even more precise by clearly describing the inputs and the outputs for the algorithm and by clarifying what we mean by instructions like "guess a number" and "stop." But this will do for now.

Implementing Binary Search of an Array

Let's think about binary search on a sorted array. Many programming languages already provide methods for determining whether a given element is in an array and, if it is, its location. But we want to implement it ourselves, to understand how you can implement such methods. Here's an array of the first 25 prime numbers, in order:



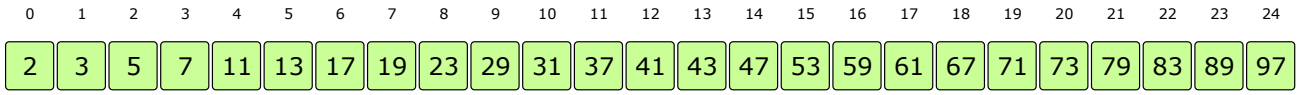
Suppose we want to know whether the number 67 is prime. If 67 is in the array, then it's prime.

We might also want to know how many primes are smaller than 67. If we find the position of the number 67 in the array, we can use the position to figure out how many smaller primes exist.

The position of an element in an array is known as its index. Array indices start at 0 and count upwards. If an element is at index 0 then it is the first element in the array. If an element is at index 3, then it has 3 elements which come before it in the array.

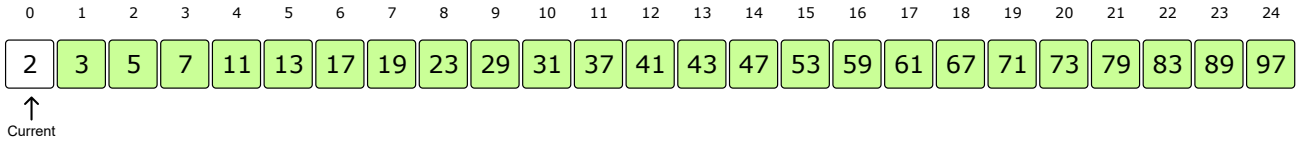
Looking at the example below, we can read the array of prime numbers from left to right, one at a time, until we find the number 67—in the green box—and see that it is at array index 18. Looking through the numbers in order like this is a **linear search**.

Once we know that the prime number 67 is at index 18, we can identify that it is a prime. We can also quickly identify that there are 18 elements which come before 67 in the array, meaning that there are 18 prime numbers smaller than 67.



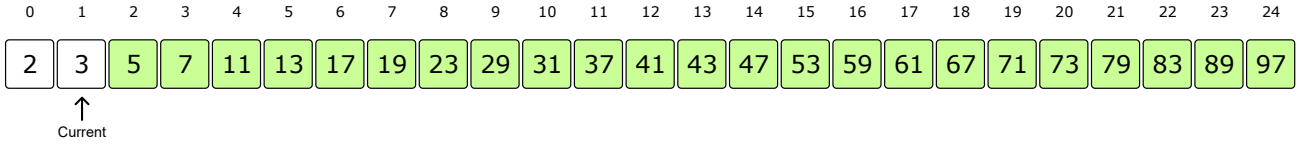
Search 67

1 of 20



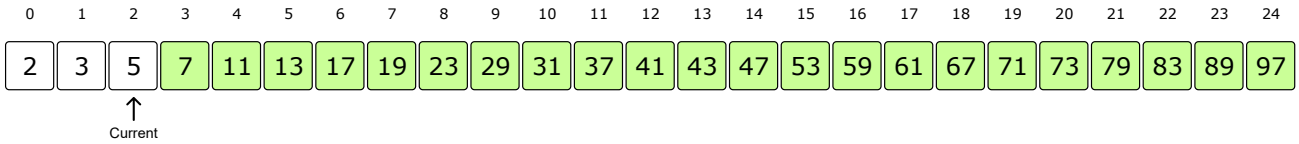
67 > 2

2 of 20



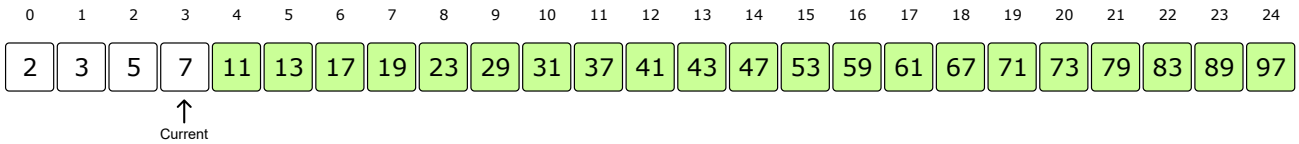
67 > 3

3 of 20



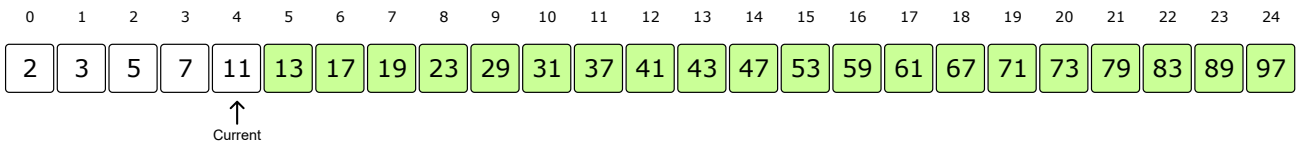
67 > 5

4 of 20



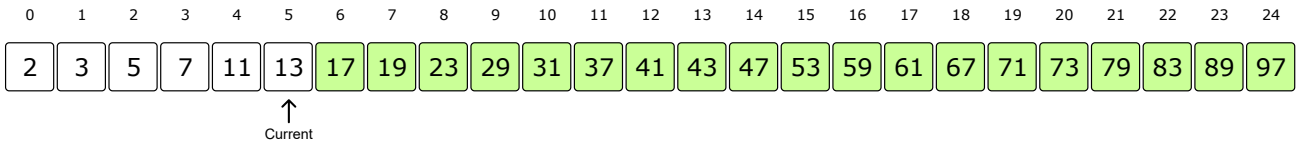
67 > 7

5 of 20



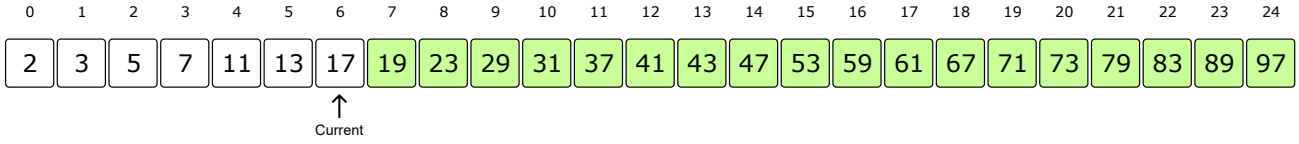
67 > 11

6 of 20



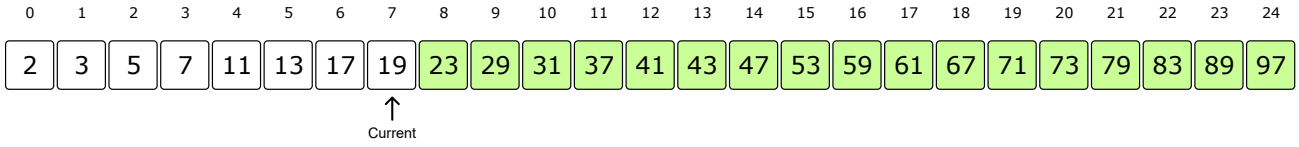
67 > 13

7 of 20



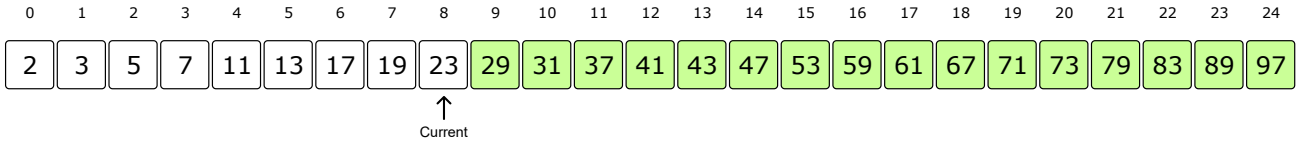
67 > 17

8 of 20



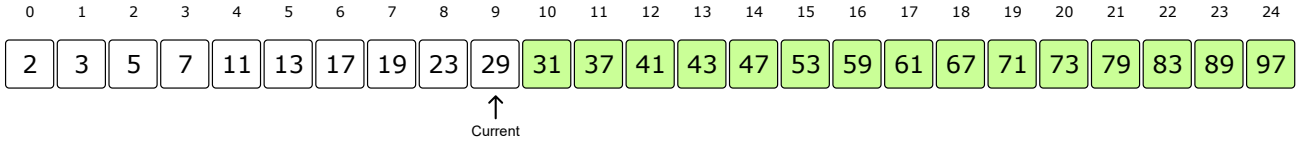
67 > 19

9 of 20



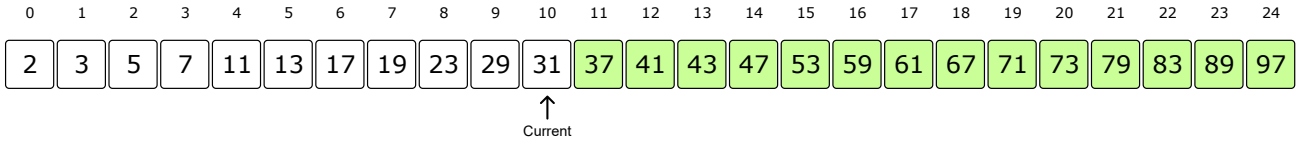
67 > 23

10 of 20

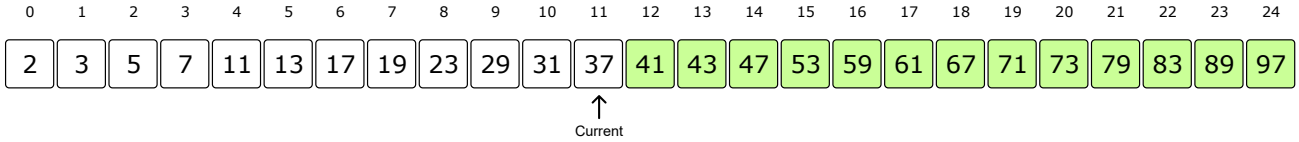


67 > 29

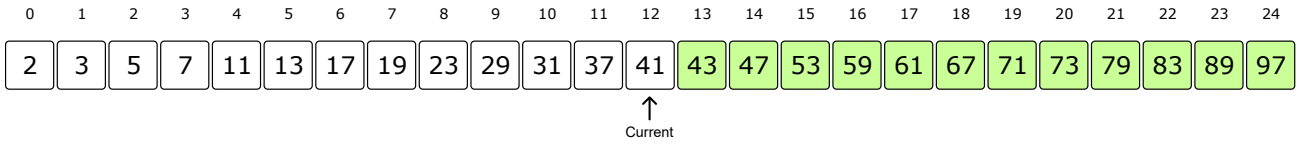
11 of 20



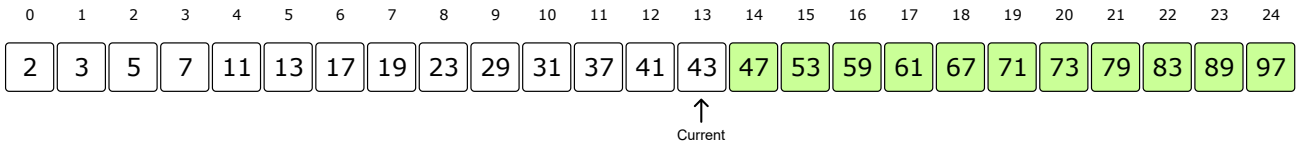
67 > 31



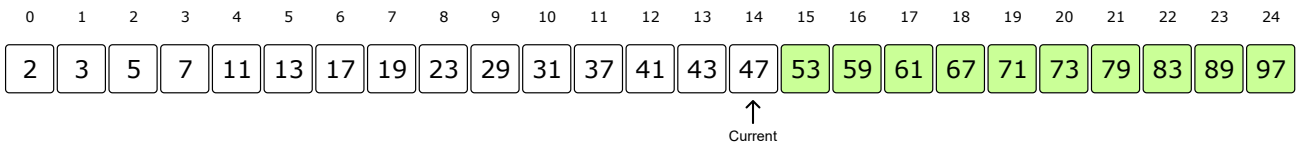
67 > 37



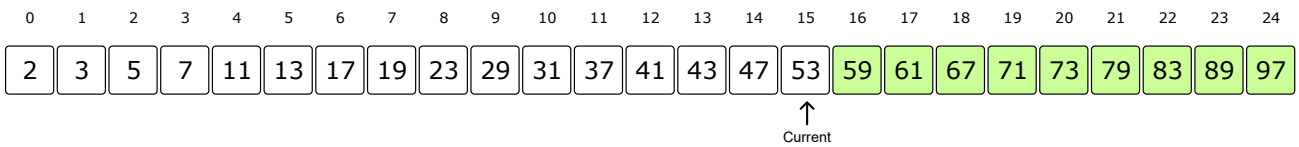
67 > 41



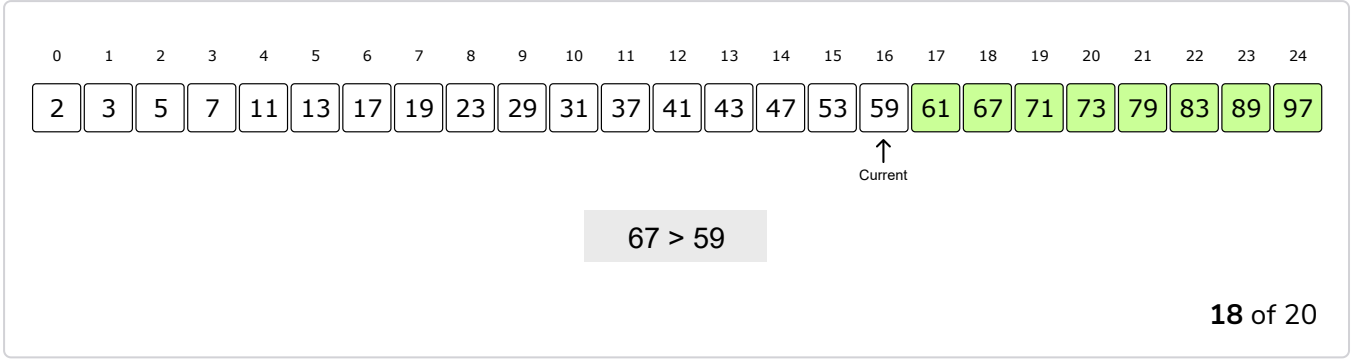
67 > 43

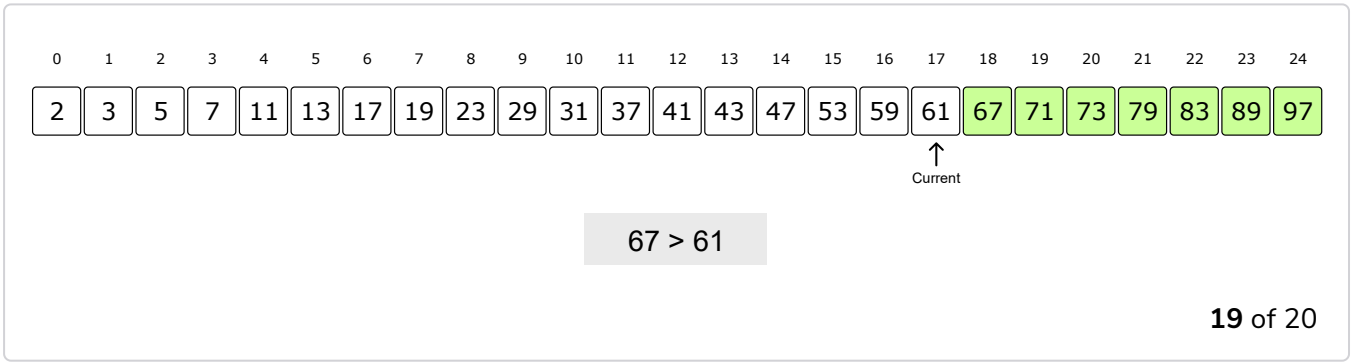


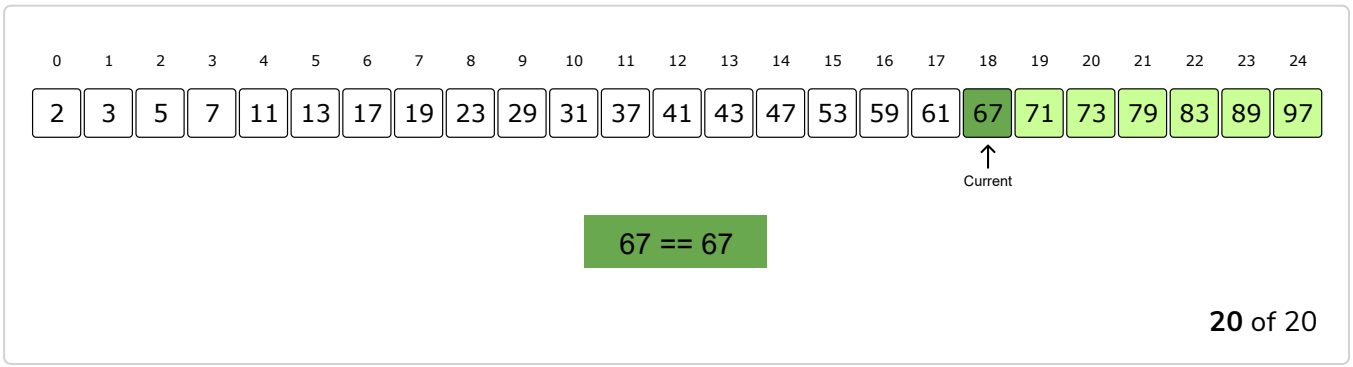
67 > 47



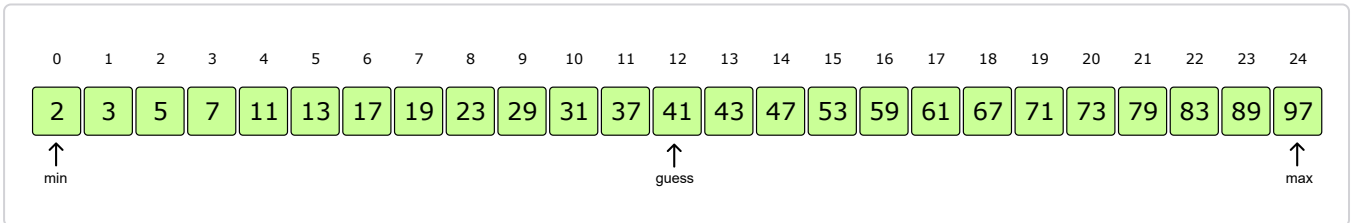
67 > 53







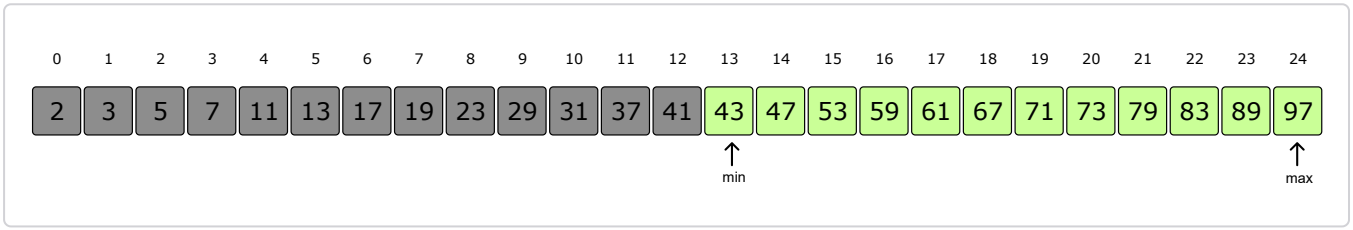
Did you see how many steps that took? A binary search might be more efficient. Because the array primesprimes contains 25 numbers, the indices into the array range from 0 to 24. Using our pseudocode from before, we start by letting **min** = **0** and **max**=**24**. The first guess in the binary search would therefore be at index 12, which is **(0** + **24)** / **2**. Is **primes[12]** equal to 67? No, **primes[12]** is **41**.



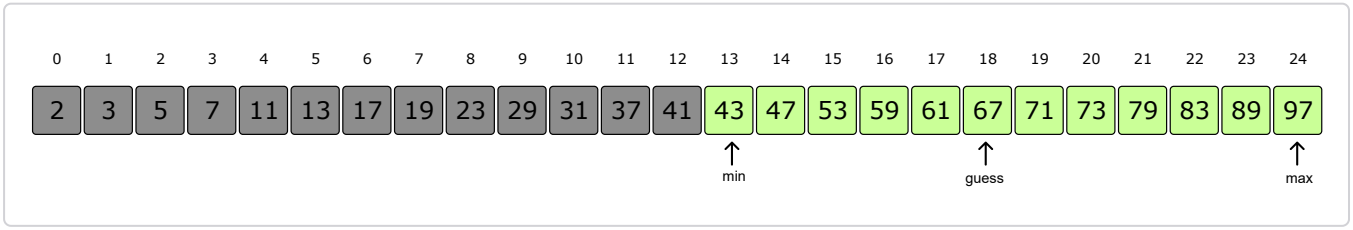
Is the index we are looking for higher or lower than 12? Since the values in the array are in increasing order, and **41** < **67**, the value 67 should be to the

right of index 12. In other words, the index we are trying to guess should be

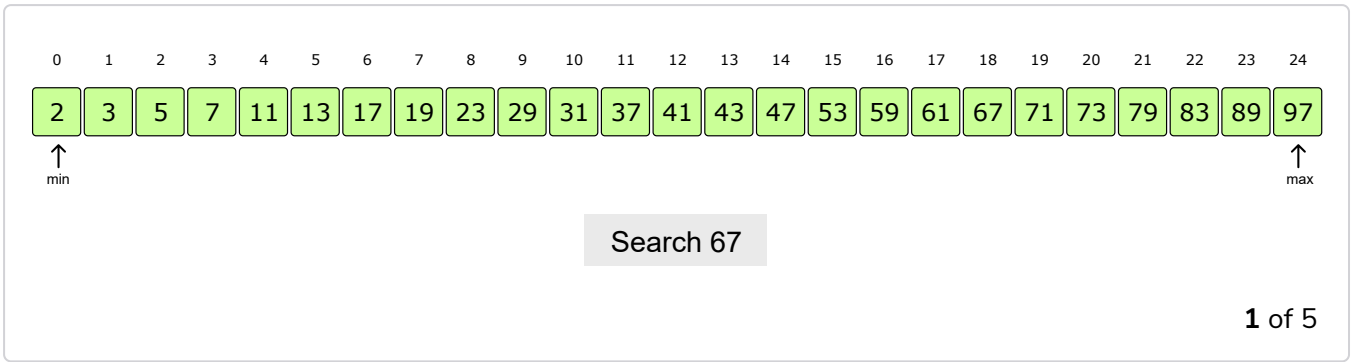
greater than 12. We update the value of **min** to **12** + **1**, or 13, and we leave **max** unchanged at 24.

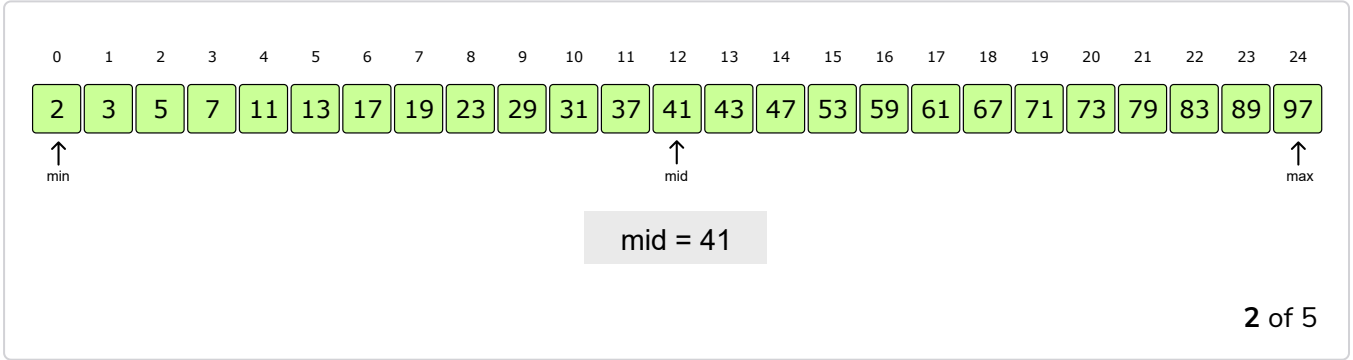


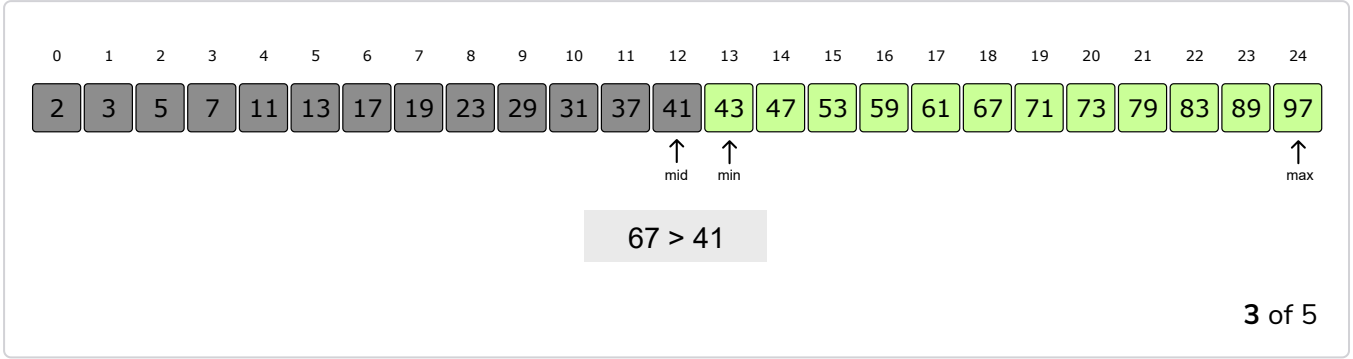
What's the next index to guess? The average of 13 and 24 is 18.5, which we round down to 18, since an index into an array must be an integer. We find that **primes[18]** is **67**.

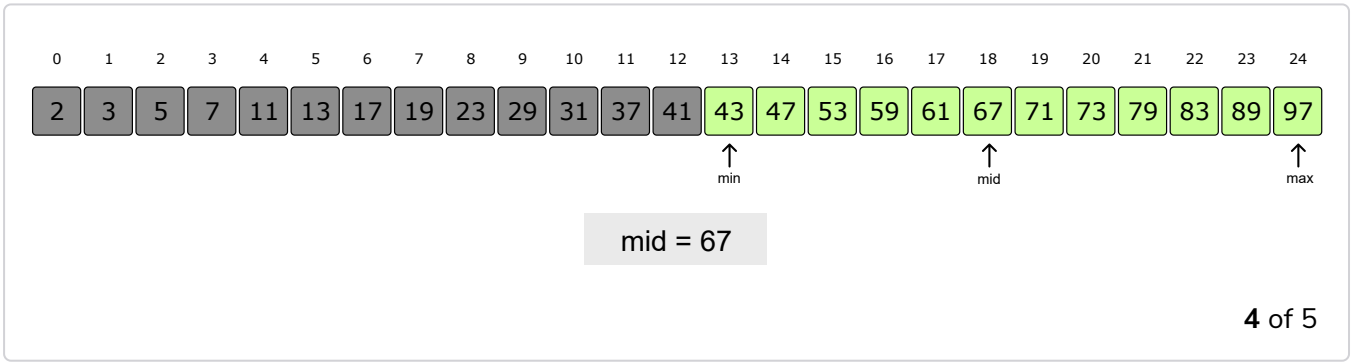


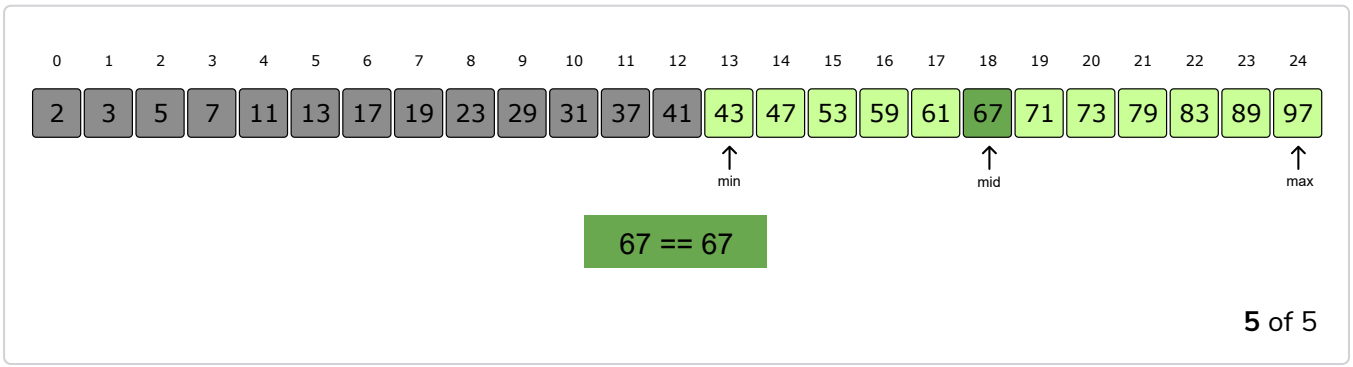
The binary search algorithm stops at this point, since it has found the answer. It took only two guesses, instead of the 19 guesses that linear search would have taken. You can step through that again in the visualization below:











Pseudocode

We just described the binary search algorithm in English, stepping through one example. That's one way to do it, but a human language explanation can vary in quality. It can be too short or too long, and—most importantly—it's not always as precise as it should be. We could jump to showing you binary search in a programming language like JavaScript or Python, but programs contain lots of details due to requirements imposed by the programming language and because programs have to handle errors caused by bad data, user error, or system faults. These details can make it hard to understand the underlying algorithm from studying just the code. That's why we prefer to describe algorithms in something called **pseudocode**, which mixes English with features that you see in programming languages.

Below is the pseudocode for binary search, modified for searching in an array

Below is the pseudocode for binary search, modified for searching in an array. The inputs are the array, which we call arrayarray; the number **n** of elements

in **array**; and **target**, the number being searched for. The output is the index in **array of target**:

1. Let **min** = **0** and **max=n−1**.
2. Compute **guess** as the average of **max** and **min**, rounded down so that it is an integer.
3. If **array[guess]** equals **target**, then stop. You found it! Return **guess**.
4. If the guess was too low, that is, array[guess] < target, then set **min** = **guess** + **1**.
5. Otherwise, the guess was too high. Set **max** = **guess - 1**.
6. Go back to step two.

Implementing Pseudocode

We'll alternate between English, pseudocode, and an actual programming language like Javascript or Python in these tutorials, depending on the situation. As a programmer, you should learn to understand pseudocode and be able to turn it into your language of choice. Even though we're using JavaScript here, it should be straightforward for you to implement pseudocode using other languages.

How would we turn the pseudocode in the example above into a JavaScript/Python program? We should create a function because we're writing code that accepts an input and returns an output, and we want that code to be reusable for different inputs. The parameters to the function—let's call it **binarySearch**—will be the array and target value, and the return value of the function will be the index of the location where the target value was found.

Now let's go into the body of the function and decide how to implement our plan. Step six says to go back to step two. That sounds like a loop. Should it be a for-loop or a while-loop? If you really wanted to use a for-loop, you could, but the indices guessed by binary search don't go in the sequential order that would make a for-loop convenient. We might first guess the index 12 and then 18, based on some computations. Because of this, a while-loop is the better choice.

There's also an important step missing in the pseudocode that doesn't matter for the guessing game, but does matter for the binary search of an array. What would happen if the number you are looking for is not in the array? Let's start by figuring out what index the **binarySearch** function should return in this case. It should be a number that cannot be a legal index into the array. We'll use -1, since that cannot be a legal index into any array—actually, any negative number would do.

The target number isn't in the array if there are no possible guesses left. In our example, suppose that we're searching for the target number 10 in the **primes** array. If it were there, 10 would be between the values 7 and 11, which are at indices 3 and 4. If you trace out the index values for **min** and **max** as the **binarySearch** function executes, you would find that they eventually get to the point where **min** = **3** and **max**=**4**. The guess is then index 3—since **(3** + **4)** / **2** = **3.5** and we round down—and **primes[3]** is less than 10, so that **min** becomes 4. With both **min** and **max** equaling 4, the guess must be index 4, and **primes[4]** is greater than 10. Now **max** becomes 3. What does it mean for **min** to equal 4 and **max** to equal 3? It means that the only possible guesses are at least 4 and at most 3. There are no such numbers! At this point, we can conclude that the target number, 10, is not in the **primes** array, and the **binarySearch** function would return -1. In general, once **max** becomes strictly less than **min**, we know that the target number is not in the sorted array.

Here is modified pseudocode for binary search that handles the case in which the target number is not present:

1. Let **min** = **0** and **max**=**n**−**1**.
2. If **max**<**min**, then stop; **target** is not present in **array**. Return -1.
3. Compute **guess** as the average of **max** and **min**, rounded down so that it is an integer.
4. If **array[guess]** equals **target**, then stop. You found it! Return **guess**.
5. If the guess was too low, that is, **array[guess]** < **target**, then set **min** = **guess** + **1**.
6. Otherwise, the guess was too high. Set **max** = **guess - 1**.
7. Go back to step two.

7. Go back to step two.

Now that we've thought through the pseudocode together, try implementing binary search yourself. It's fine to look back at the pseudocode. In fact, it's a good thing because then you'll have a better grasp of what it means to convert pseudocode into a program.

Challenge: Binary Search

Complete the doSearch function so that it implements a binary search, following the pseudo-code below (this pseudo-code was described in the previous article):

1. Let min = 0 and max = n-1.
2. If max < min, then stop: target is not present in array. Return -1.
3. Compute guess as the average of max and min, rounded down (so that it is an integer).
4. If array[guess] equals target, then stop. You found it! Return guess.
5. If the guess was too low, that is, array[guess] < target, then set min = guess + 1.
6. Otherwise, the guess was too high. Set max = guess - 1.
7. Go back to step 2.

Java | Python | C++ | JS

```
import java.util.Arrays;
import java.lang.Integer;

class Solution {
  // Returns either the index of the location in the array,
  // or -1 if the array did not contain the targetValue
  public static int doSearch(int[] array, int targetValue) {
    int min = 0;
        System.out.println(Arrays.toString(array));
    int max = array.length - 1;
    int guess;


        return -1;
  }
};
```

Running Time of Binary Search

We know that linear search on an array of nn elements might have to make as many as nn guesses. You probably already have an intuitive idea that binary search makes fewer guesses than linear search. You even might have perceived that the difference between the worst-case number of guesses for linear search and binary search becomes more striking as the array length increases. Let's see how to analyze the maximum number of guesses that binary search makes.

The key idea is that when binary search makes an incorrect guess, the portion of the array that contains reasonable guesses is reduced by at least half. If the reasonable portion had 32 elements, then an incorrect guess cuts it down to have at most 16. Binary search halves the size of the reasonable portion upon every incorrect guess.

If we start with an array of length 8, then incorrect guesses reduce the size of the reasonable portion to 4, then 2, and then 1. Once the reasonable portion contains just one element, no further guesses occur; the guess for the 1-element portion is either correct or incorrect, and we're done. So with an array of length 8, binary search needs at most four guesses.

What do you think would happen with an array of 16 elements? If you said that the first guess would eliminate at least 8 elements, so that at most 8 remain, you're getting the picture. So with 16 elements, we need at most five guesses.

By now, you're probably seeing the pattern. Every time we double the size of the array, we need at most one more guess. Suppose we need at most **m** guesses for an array of length **n**. Then, for an array of length **2n**, the first guess cuts the reasonable portion of the array down to size **n**, and at most **m** guesses finish up, giving us a total of at most **m+1** guesses.

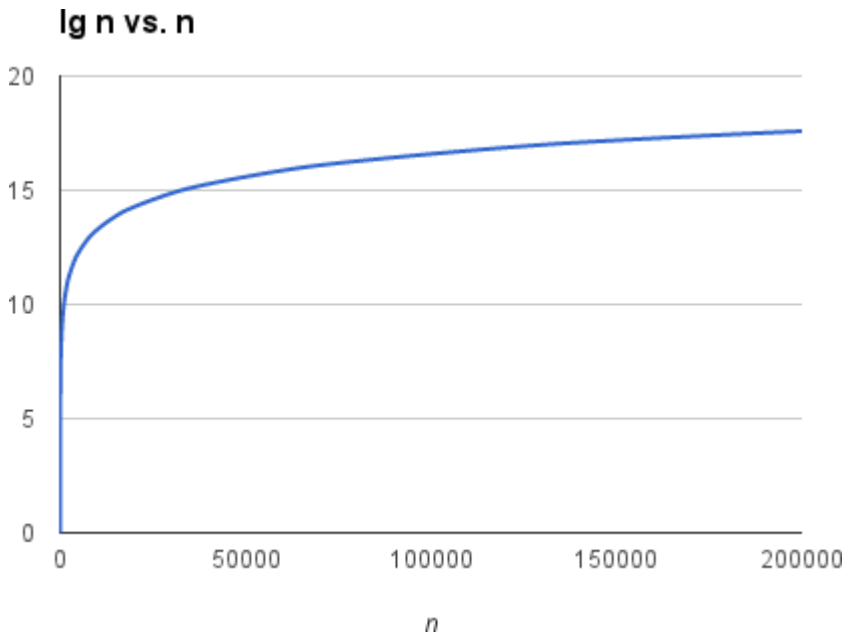
Let's look at the general case of an array of length **n**. We can express the

Let's look at the general case of an array of length **n**. We can express the number of guesses, in the worst case, as "the number of times we can repeatedly halve, starting at **n**, until we get the value 1, plus one." But that's inconvenient to write out. Fortunately, there's a mathematical function that means the same thing as the number of times we repeatedly halve, starting at **n**, until we get the value 1: the ***base-2 logarithm*** of **n**. We write it as **lg n**. (You can learn more about logarithms here.)

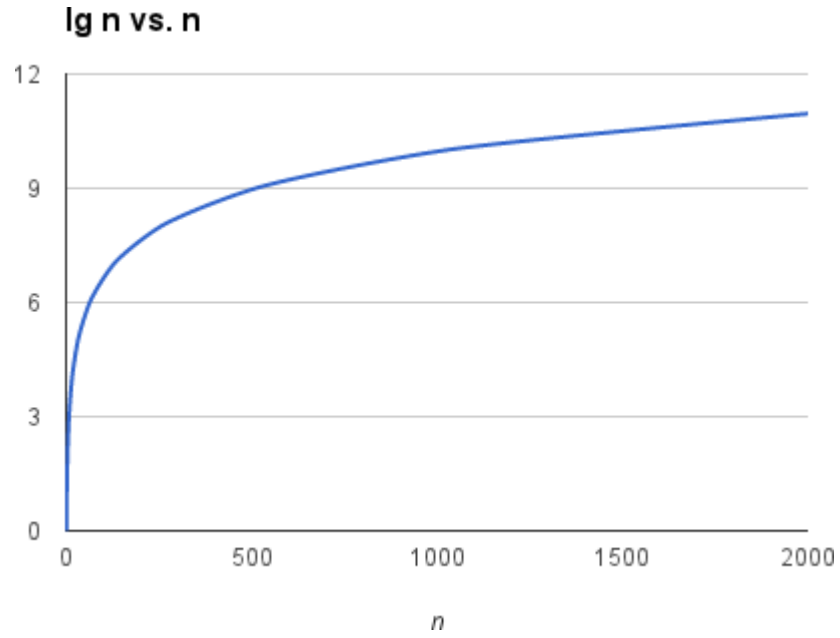
Here's a table showing the base-2 logarithms of various values of ***n***:

n	lg n
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
1,048,576	20
2,097,152	21

We can view this same table as a chart:

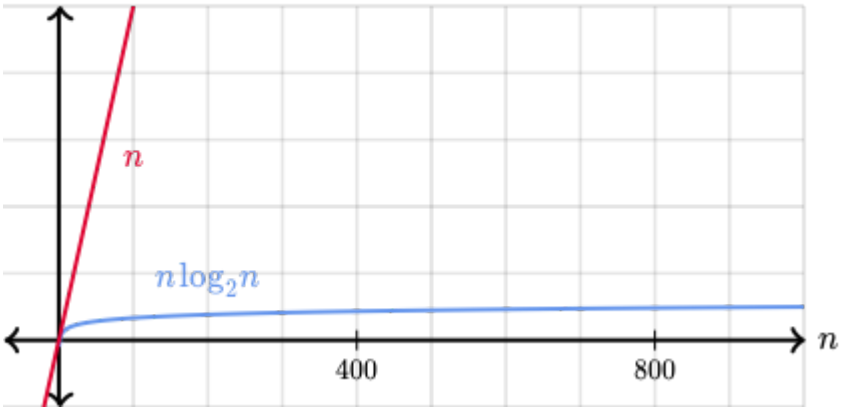


Zooming in on smaller values of n:



The logarithm function grows very slowly. Logarithms are the inverse of exponentials, which grow very rapidly, so that if **lgn=x**, then **n** = 2^{x}. For example, because **lg 128** = 7, we know that 2^{7} = **128**.

When **n** is not a power of 2, we can just go up to the next higher power of 2. For an array whose length is 1000, the next higher power of 2 is 1024, which equals 2^{10} . Therefore, for a 1000-element array, binary search would require at most 11 (10 + 1) guesses. For the Tycho-2 star catalog with 2,539,913 stars, the next higher power of 2 is 2^{22} (which is 4,194,304), and we would need at most 23 guesses. Much better than linear search! Compare them below:



In the next tutorial, we'll see how computer scientists characterize the running times of linear search and binary search, using a notation that distills the most important part of the running time and discards the less important parts.

Quiz: Running time of binary search

Compute how many steps binary search would take to find an item in arrays of various sizes.

1

32 teams qualified for the 2014 World Cup. If the names of the teams were arranged in sorted order (an array), how many items in the array would binary search have to examine to find the location of a particular team in the array, in the worst case?

2

What is lg(32), the base-2 logarithm of 32?

3

You have an array containing the prime numbers from 2 to 311 in sorted order: [2, 3, 5, 7, 11, 13, ..., 307, 311]. There are 64 items in the array. About how many items of the array would binary search have to examine before concluding that 52 is not in the array, and therefore not prime?

4

In 2013, there were 193 member states in the United Nations. If the names of these states were sorted alphabetically in an array, about how many names would binary search examine to locate a particular name in the array, in the worst case?

5

The 2014 "Catalogue of Life" contains about 1580000 names of species. If these names were sorted in an array, in the worst case, how long would it take to use binary search to find the name of a particular species in the array?

Check Answers

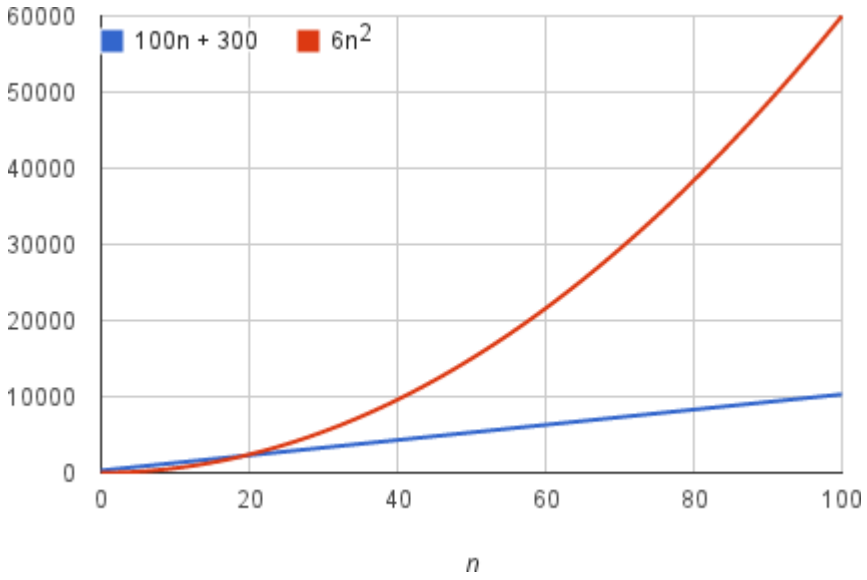
Introduction to Asymptotic notation

So far, the way we analyzed linear search and binary search has been by counting the maximum number of guesses we need to make. But what we really want to know is how long these algorithms take. We're interested in time, not just guesses. The running times of linear search and binary search include the times to make and check guesses, but there's more to these algorithms than making and checking guesses.

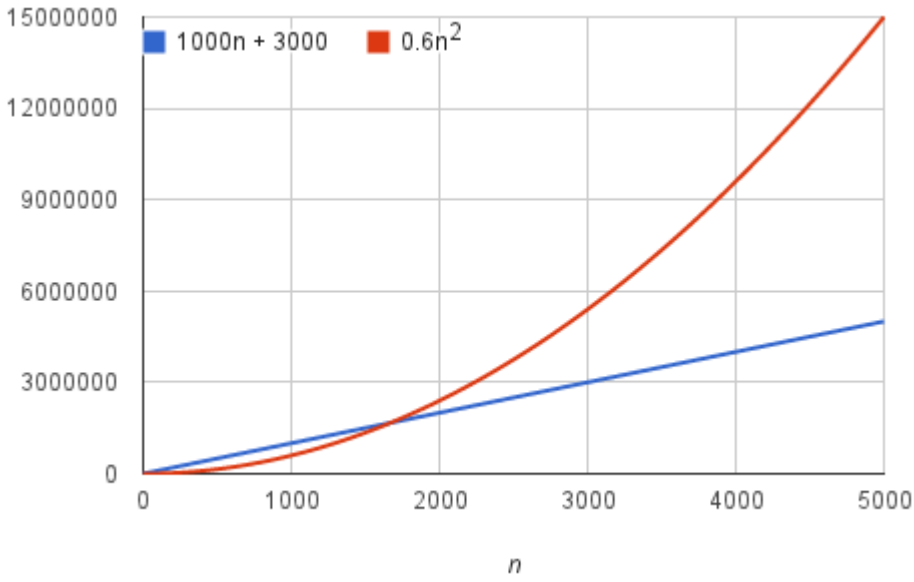
The running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm. And that depends on the speed of the computer, the programming language, and the compiler that translates the program from the programming language into code that runs directly on the computer, as well as other factors.

Let's think about the running time of an algorithm more carefully. We use a combination of two ideas. First, we determine how long the algorithm takes, in terms of the size of its input. This idea makes intuitive sense, doesn't it? We've already seen that the maximum number of guesses in linear search and binary search increases as the length of the array increases. Or think about a GPS. If it knew about only the interstate highway system, and not about every little road, it should be able to find routes more quickly, right? So we think about the running time of the algorithm as a function of the size of its input.

The second idea is that we focus on how fast this function grows with the input size. We call that the **rate of growth** of the running time. To keep things manageable, we simplify the function to distill the most important part and cast aside the less important parts. For example, suppose that an algorithm, running on an input of size $\mathbf{n}$, takes $\mathbf{6n^2 + 100n + 300}$ machine instructions. The $\mathbf{6n^2}$ term becomes larger than the remaining terms, $\mathbf{100\,n + 300}$, once $\mathbf{n}$ becomes large enough (20 in this case). Here's a chart showing values of $\mathbf{6n^2}$ and $\mathbf{100n + 300}$ for values of $\mathbf{n}$ from **0 to 100.**



We would say that the running time of this algorithm grows as $\mathbf{n^2}$, dropping the coefficient **6** and the remaining terms **100n** + **300**. It doesn't really matter what coefficients we use; as long as the running time is $\mathbf{an^2}$ + **bn** + **c**, for some numbers **a** > **0**, **b**, and **c**, there will always be a value of n for which $\mathbf{an^2}$ is greater than **bn** + **c**, and this difference increases as **n** increases. For example, here's a chart showing values of $\mathbf{0.6n^2}$ + **1000n** + **3000**, so that we've reduced the coefficient of $\mathbf{n^2}$ by a factor of **10** and increased the other two constants by a factor of **10**.



The value of **n** at which $\mathbf{0.6n^2}$ becomes greater than **1000n** + **3000** has increased, but there will always be such a crossover point, no matter what the constants.

By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time—its rate of growth

focus on the important part of an algorithm's running time—its rate of growth—without getting mired in details that complicate our understanding. When

we drop the constant coefficients and the less significant terms, we use asymptotic notation. We'll see three forms of it: **big-Θ** notation, **big-O** notation, and **big-Ω** notation.

Big-θ (Big-Theta) notation

Let's look at a simple implementation of linear search (in the language of your choice):

JS JS | C++ | Python

```
var doLinearSearch = function(array, targetValue) {
  for (var guess = 0; guess < array.length; guess++) {
    if (array[guess] === targetValue) {
        return guess;  // found it!
    }
  }

  return -1;  // didn't find it
};
```

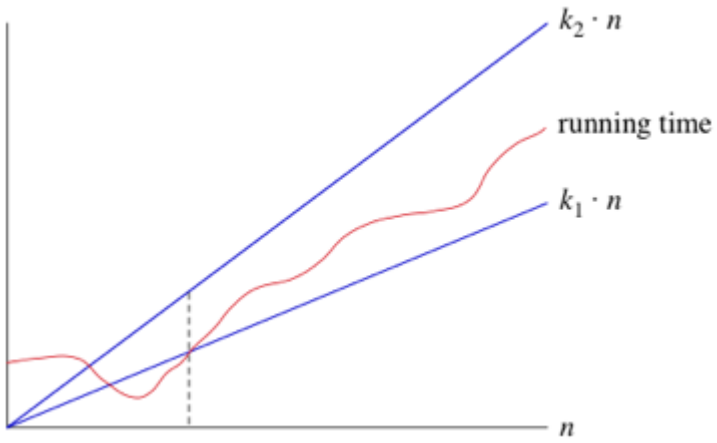
Let's denote the size of the array by **n**. The maximum number of times that the for-loop can run is **n**, and this worst case occurs when the value being searched for is not present in the array. Each time the for-loop iterates, it has to do several things: compare guess with **n**; compare **array[guess]** with **targetValue;** possibly return the value of **guess**; and increment **guess**. Each of these little computations takes a constant amount of time each time it executes. If the for-loop iterates **n** times, then the time for all **n** iterations is $c_1 \cdot n$, where c_1 is the sum of the times for the computations in one loop iteration. Now, we cannot say here what the value of c_1 is, because it depends on the speed of the computer, the programming language used, the compiler or interpreter that translates the source program into runnable code, and other factors. This code has a little bit of extra overhead, for setting up the for-loop (including initializing guess to **0**) and possibly returning **-1** at the end. Let's call the time for this overhead c_2, which is also a constant. Therefore, the total time for linear search in the worst case is $c_1 \cdot n + c_2$.

As we've argued, the constant factor c_1 and the low-order term c_2 don't tell us about the rate of growth of the running time. What's significant is that the

worst-case running time of linear search grows like the array size **n**. The

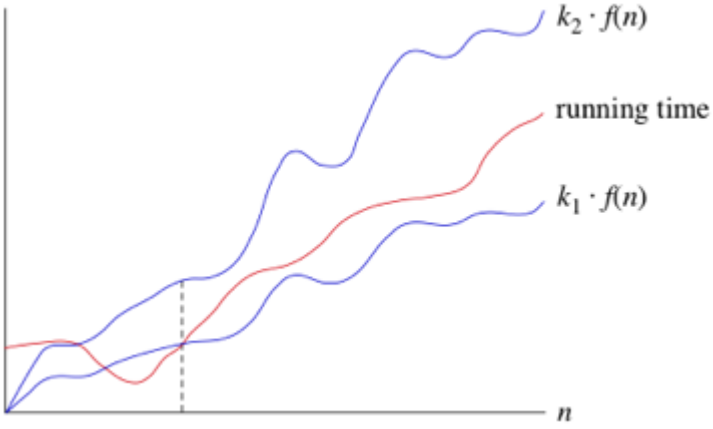
notation we use for this running time is **Θ(n)**. That's the Greek letter "**theta**," and we say "**big-Theta of n**" or just "**Theta of n**."

When we say that a particular running time is **Θ(n)**, we're saying that once **n** gets large enough, the running time is at least $\mathbf{k_1 \cdot n}$ and at most $\mathbf{k_2 \cdot n}$ for some constants $\mathbf{k_1}$ and $\mathbf{k_2}$. Here's how to think of **Θ(n)**:



For small values of n, we don't care how the running time compares with $\mathbf{k_1 \cdot n}$ or $\mathbf{k_2 \cdot n}$. But once **n** gets large enough—on or to the right of the dashed line—the running time must be sandwiched between $\mathbf{k_1 \cdot n}$ and $\mathbf{k_2 \cdot n}$. As long as these constants $\mathbf{k_1}$ and $\mathbf{k_2}$ exist, we say that the running time is **Θ(n)**.

We are not restricted to just **n** in **big-Θ** notation. We can use any function, such as $\mathbf{n^2}$, **n lgn**, or any other function of **n**. Here's how to think of a running time that is **Θ(f(n))** for some function **f(n)**:



Once **n** gets large enough, the running time is between $\mathbf{k_1 \cdot f(n)}$ and $\mathbf{k_2 \cdot f(n)}$.

In practice, we just drop constant factors and low-order terms. Another advantage of using **big-Θ** notation is that we don't have to worry about which

time units we're using. For example, suppose that you calculate that a running time is $\mathbf{6n^2 + 100n + 3006}$ microseconds. Or maybe it's milliseconds. When you use **big-Θ** notation, you don't say. You also drop the factor **6** and the low-order terms **100n + 300**, and you just say that the running time is $\mathbf{\Theta(n^2)}$.

When we use **big-Θ** notation, we're saying that we have an asymptotically tight bound on the running time. "Asymptotically" because it matters for only large values of **n**. "Tight bound" because we've nailed the running time to within a constant factor above and below.

Functions in Asymptotic Notation

When we use asymptotic notation to express the rate of growth of an algorithm's running time in terms of the input size $\mathbf{n}$, it's good to bear a few things in mind.

Let's start with something easy. Suppose that an algorithm took a constant amount of time, regardless of the input size. For example, if you were given an array that is already sorted into increasing order and you had to find the minimum element, it would take constant time, since the minimum element must be at index 0. Since we like to use a function of n in asymptotic notation, you could say that this algorithm runs in $\mathbf{\Theta(n^0)}$ time. Why? Because $\mathbf{n^0 = 1}$, and the algorithm's running time is within some constant factor of 1. In practice, we don't write $\mathbf{\Theta(n^0)}$, however; we write $\mathbf{\Theta(1)}$.

Now suppose an algorithm took $\mathbf{\Theta(\log_{10} n)}$ time. You could also say that it took $\mathbf{\Theta(\lg n)}$ time (that is, $\mathbf{\Theta(\log_2 n)}$ time). Whenever the base of the logarithm is a constant, it doesn't matter what base we use in asymptotic notation. Why not? Because there's a mathematical formula that says

$$\log_a n = \frac{\log_b n}{\log_b a}$$

for all positive numbers $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{n}$. Therefore, if $\mathbf{a}$ and $\mathbf{b}$ are constants, then $\mathbf{\log_a n}$ and $\mathbf{\log_b n}$ differ only by a factor of $\mathbf{\log_b a}$, and that's a constant factor which we can ignore in asymptotic notation.

Therefore, we can say that the worst-case running time of binary search is $\mathbf{\Theta(\log_a n)}$ for any positive constant $\mathbf{a}$. Why? The number of guesses is at most $\mathbf{\lg n + 1}$, generating and testing each guess takes constant time, and setting up and returning take constant time. As a matter of practice, we write that binary search takes $\mathbf{\Theta(\lg n)}$ time because computer scientists like to think in powers of $\mathbf{2}$ (and there are fewer characters to write than if we wrote $\mathbf{\Theta(\log_2 n)}$.)

There is an order to the functions that we often see when we analyze algorithms using asymptotic notation. If **a** and **b** are constants and **a** < **b**, then a running time of $\Theta(n^a)$ grows more slowly than a running time of $\Theta(n^b)$. For example, a running time of $\Theta(n)$, which is $\Theta(n^1)$, grows more slowly than a running time of $\Theta(n^2)$. The exponents don't have to be integers, either. For example, a running time of $\Theta(n^2)$ grows more slowly than a running time of $\Theta(n^{2\sqrt{n}})$, which is $\Theta(n^{2.5})$.

Logarithms grow more slowly than polynomials. That is, $\Theta(\lg n)$ grows more slowly than $\Theta(n^a)$ for any positive constant **a**. But since the value of **lg n** increases as **n** increases, $\Theta(\lg n)$ grows faster than $\Theta(1)$.

Here's a list of functions in asymptotic notation that we often encounter when analyzing algorithms, listed from slowest to fastest growing. This list is not exhaustive; there are many algorithms whose running times do not appear here:

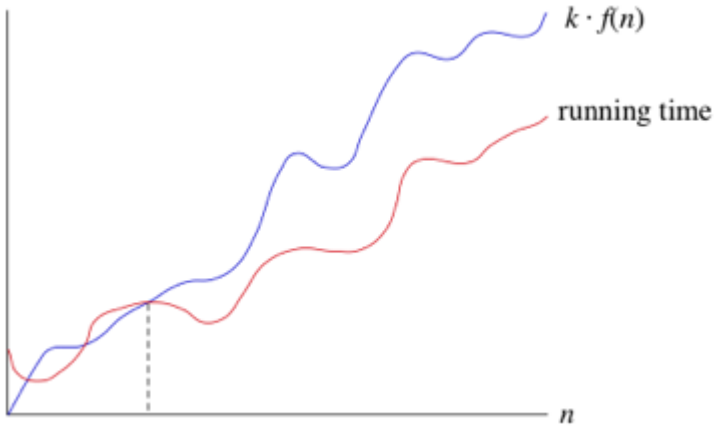
1. $\Theta(1)$
2. $\Theta(\lg n)$
3. $\Theta(n)$
4. $\Theta(n \lg n)$
5. $\Theta(n^2)$
6. $\Theta(n^2 \lg n)$
7. $\Theta(n^3)$
8. $\Theta(2^n)$

Note that an exponential function a^n where $a > 1$, grows faster than any polynomial function n^b where **b** is any constant.

Big-O notation

We use big-Θ notation to asymptotically bound the growth of a running time to within constant factors above and below. Sometimes we want to bound from only above. For example, although the worst-case running time of binary search is **Θ(lg n)**, it would be incorrect to say that binary search runs in **Θ(lg n)** time in all cases. What if we find the target value upon the first guess? Then it runs in **Θ(1)** time. The running time of binary search is never worse than **Θ(lg n)**, but it's sometimes better. It would be convenient to have a form of asymptotic notation that means "the running time grows at most this much, but it could grow more slowly." We use "big-O" notation for just such occasions.

If a running time is **O(f(n))**, then for large enough **n**, the running time is at most **k**·**f(n)** for some constant **k**. Here's how to think of a running time that is **O(f(n))**:



We say that the running time is "big-O of **f(n)**" or just "O of **f(n)**." We use big-O notation for **asymptotic upper bound**s, since it bounds the growth of the running time from above for large enough input sizes.

Now we have a way to characterize the running time of binary search in all cases. We can say that the running time of binary search is always **O(lg n)**. We can make a stronger statement about the worst-case running time: it's **Θ(lg n)**.

But for a blanket statement that covers all cases, the strongest statement we can make is that binary search runs in **O(lg n)** time.

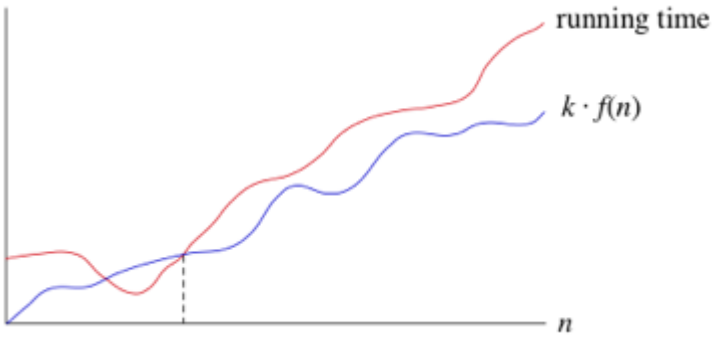
If you go back to the definition of big-Θ notation, you'll notice that it looks a lot like big-O notation, except that big-Θ notation bounds the running from both above and below, rather than just from above. If we say that a running time is **Θ(f(n))** in a particular situation, then it's also **O(f(n))**. For example, we can say that because the worst-case running time of binary search is **Θ(lg n)**, it's also **O(lg n)**. The converse is not necessarily true: as we've seen, we can say that binary search always runs in **O(lg n)** time but not that it always runs in **Θ(lg n)** time.

Because big-O notation gives only an asymptotic upper bound, and not an asymptotically tight bound, we can make statements that at first blush seem incorrect, but are technically correct. For example, it is absolutely correct to say that binary search runs in **O(n)** time. That's because the running time grows no faster than a constant times n. In fact, it grows slower. Think of it this way. Suppose you have 10 dollars in your pocket. You go up to your friend and say, "I have an amount of money in my pocket, and I guarantee that it's no more than one million dollars." Your statement is absolutely true, though not terribly precise. One million dollars is an upper bound on 10 dollars, just as O(n) is an upper bound on the running time of binary search. Other, imprecise, upper bounds on binary search would be $\mathbf{O(n^2)}$, $\mathbf{O(n^3)}$ and $\mathbf{O(2^n)}$. But none of **Θ(n)**, $\mathbf{\Theta(n^2)}$, $\mathbf{\Theta(n^3)}$, and $\mathbf{\Theta(2^n)}$ would be correct to describe the running time of binary search in any case.

Big-Ω (Big-Omega) notation

Sometimes, we want to say that an algorithm takes at least a certain amount of time, without providing an upper bound. We use big-Ω notation; that's the Greek letter "omega."

If a running time is **Ω(f(n))**, then for large enough **n**, the running time is at least **k·f(n)** for some constant **k**. Here's how to think of a running time that is **Ω(f(n))**:



We say that the running time is "big-Ω of **f(n)**." We use big-Ω notation for **asymptotic lower bounds**, since it bounds the growth of the running time from below for large enough input sizes.

Just as **Θ(f(n))** automatically implies **O(f(n))**, it also automatically implies **Ω(f(n))**. So we can say that the worst-case running time of binary search is **Ω(lgn)**. We can also make correct, but imprecise, statements using big-Ω notation. For example, just as if you really do have a million dollars in your pocket, you can truthfully say "I have an amount of money in my pocket, and it's at least 10 dollars," you can also say that the worst-case running time of binary search is **Ω(1)**, because it takes at least constant time.

Quiz: Asymptotic notation

1

For the functions, n^k and c^n, what is the asymptotic relationship between these functions? Assume that k >= 1 and c > 1 are constants.

COMPLETED 0%

1 of 2

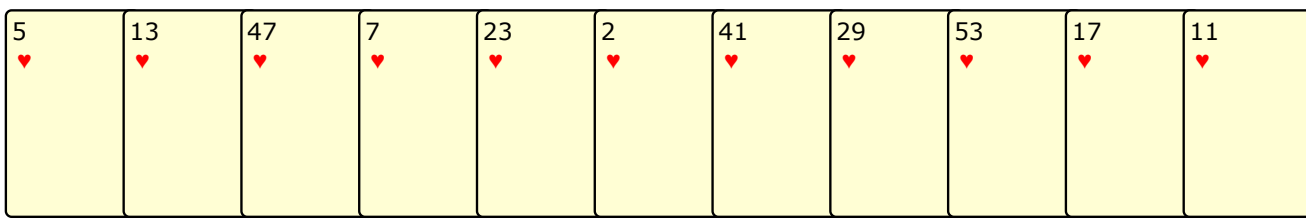
Sorting

Sorting a list of items into ascending or descending order can help either a human or a computer find items on that list quickly, perhaps using an algorithm like binary search. Most Programming languages have built-in sorting methods. It works on arrays of numbers, or even on arrays of strings:

```
animals = ["gnu", "zebra", "antelope", "aardvark", "yak", "iguana"];
animals.sort();
```

Even though languages have built-in sorting method, sorting is a great example of how there may be many ways to think about the same problem, some perhaps better than others. Understanding sorting is a traditional first step towards mastery of algorithms and computer science.

You'll implement a particular sorting algorithm in a moment. But as a warmup, here is a sorting problem to play with. You can swap any pair of cards by clicking on one card, and then the other. Swap cards until the cards are sorted with with smallest card on the left.



Challenge: Implement Swap

Do you know how Swap Function works? Implement to prove yourself!

WE'LL COVER THE FOLLOWING

Swap Function

A key step in many sorting algorithms (including selection sort) is swapping the location of two items in an array. Here's an Empty Function that needs to Swap the elements of a given array, as per the specified indices.

Function Prototype:

```
public static void swap(int[] array, int firstIndex, int secondIndex);
```

where *array* is the input int array and firstIndex/ secondIndex specify the integers to swap.

Output:

Returns the Updated Array

Sample Input

```
arr1 = [9,4,7,1,2,6,5]

firstIndex = 2
secondIndex = 0
```

Sample Output

```
arr1 = [7,4,9,1,2,6,5]
```

Explanation

You are provided with an `array` of integers, along with the first and second index. You need to return the input array with the integers at the position of `firstIndex` and `secondIndex` *Swapped!*

Coding Exercise

Take a close look and design a step-by-step algorithm first before jumping on implementation. This problem is designed for your practice, so try to solve it on your own first. If you get stuck, you can always refer to the solution provided in the solution section. Good Luck!

Java

```
class Solution {
  public static void swap(int[] array, int firstIndex, int secondIndex) {


    return;
  }
}
```

Selection Sort Pseudocode

There are many different ways to sort the cards. Here's a simple one, called selection sort, possibly similar to how you sorted the cards above:

Find the smallest card. Swap it with the first card.

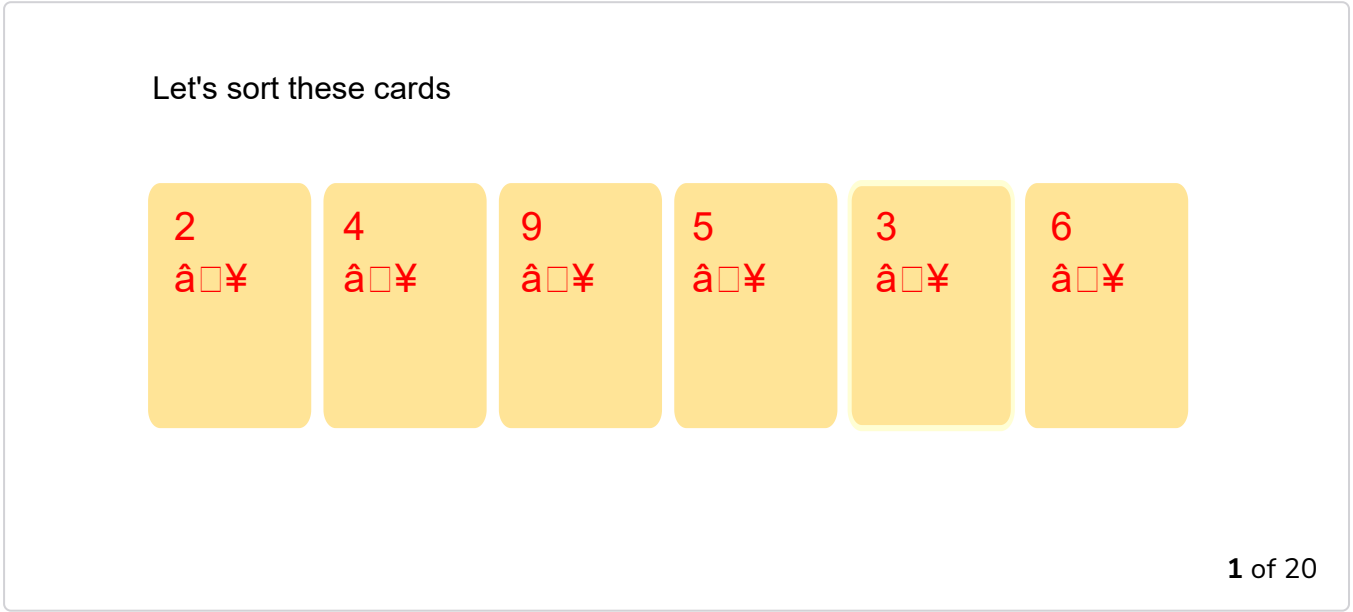
Find the second-smallest card. Swap it with the second card.

Find the third-smallest card. Swap it with the third card.

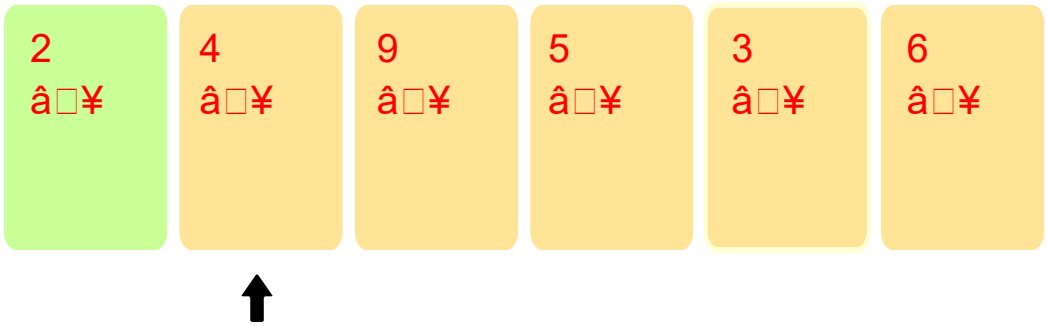
Repeat finding the next-smallest card, and swapping it into the correct position until the array is sorted.

This algorithm is called selection sort because it repeatedly selects the next-smallest element and swaps it into place.

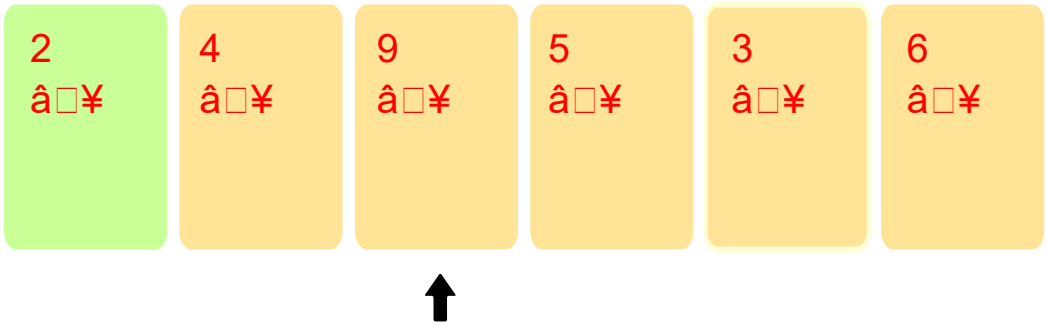
You can see the algorithm for yourself below. Start by using "Step" to see each step of the algorithm, and then try "Automatic" once you understand it to see the steps all the way through.



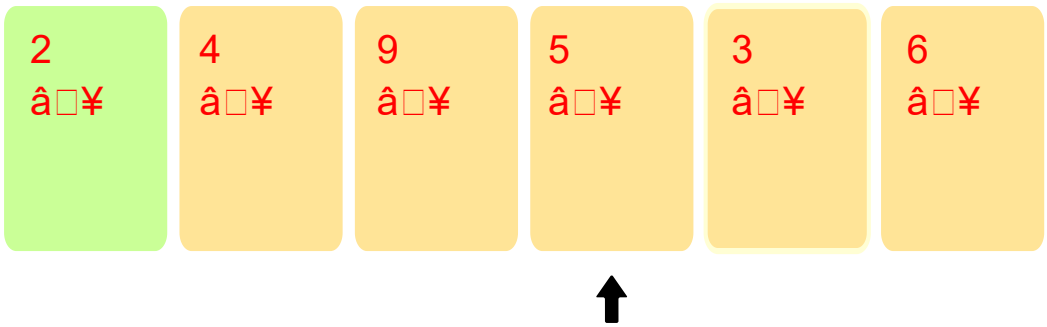
Finding next smallest



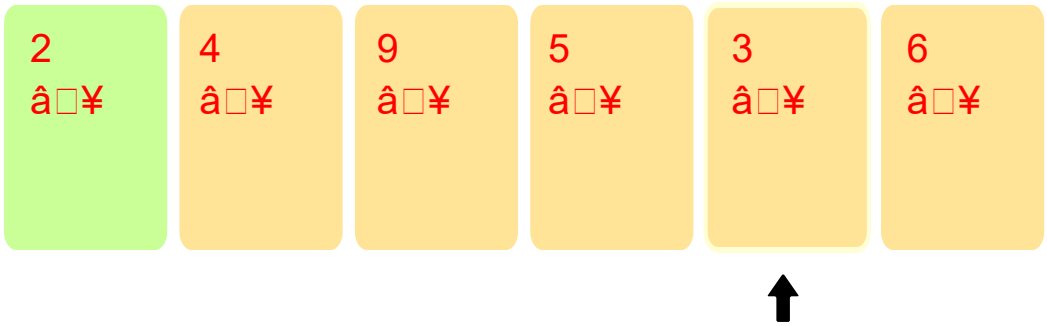
Finding next smallest



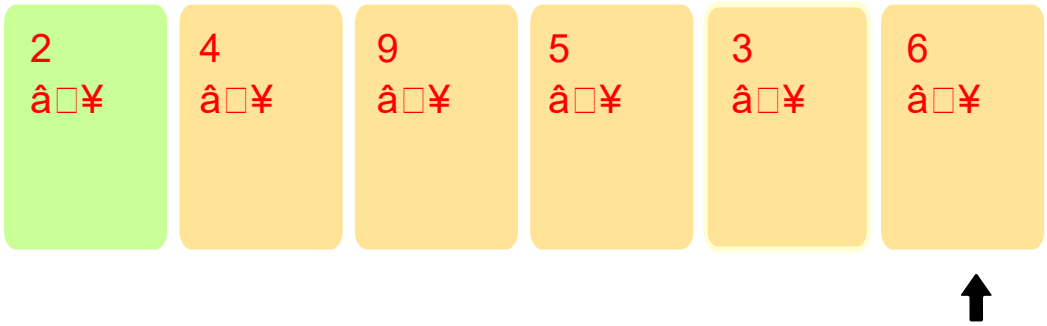
Finding next smallest



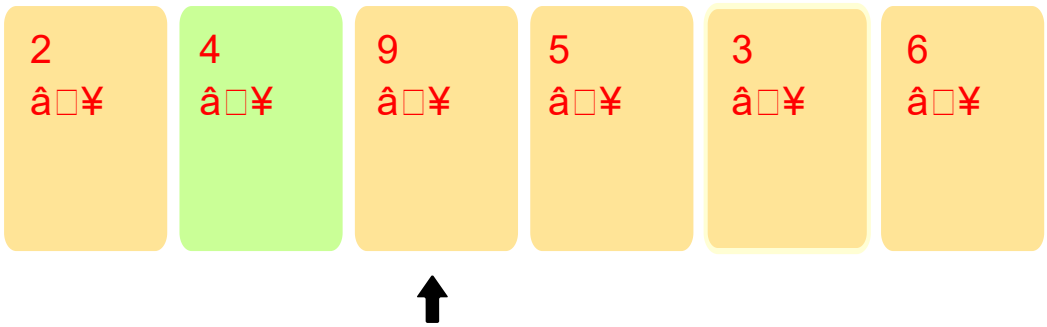
Finding next smallest



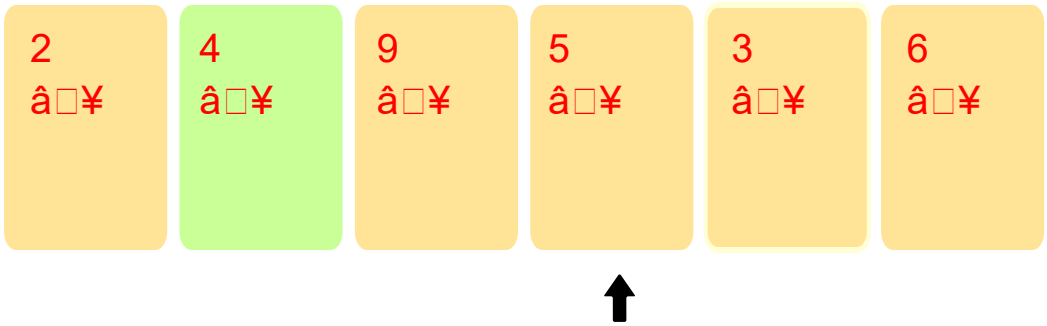
No card smaller than 2 so it stays where it is.



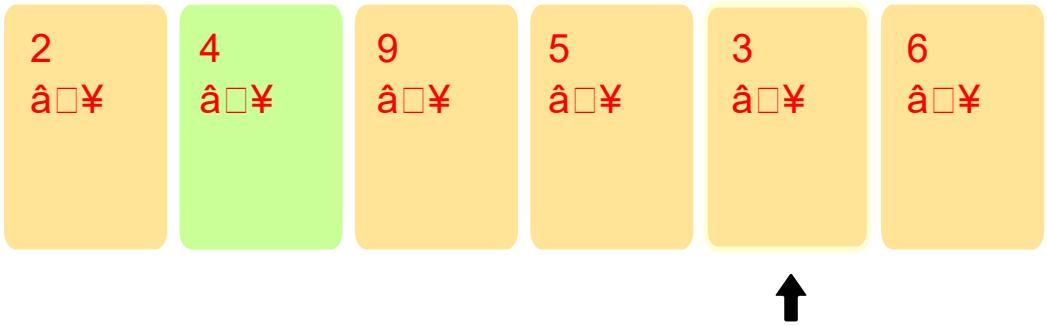
Finding next smallest



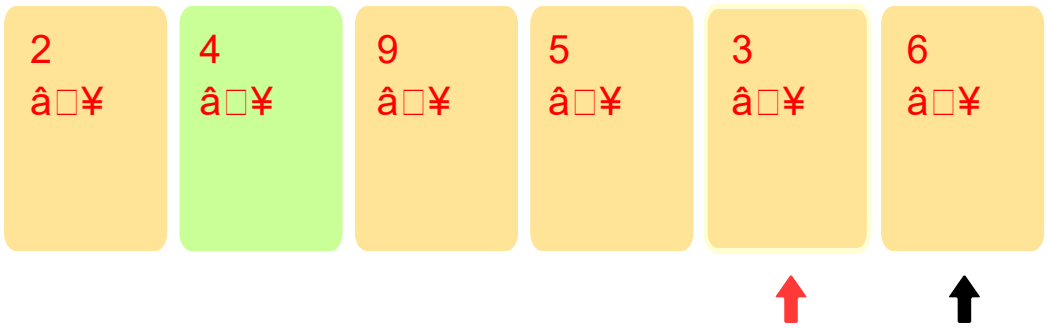
Finding next smallest



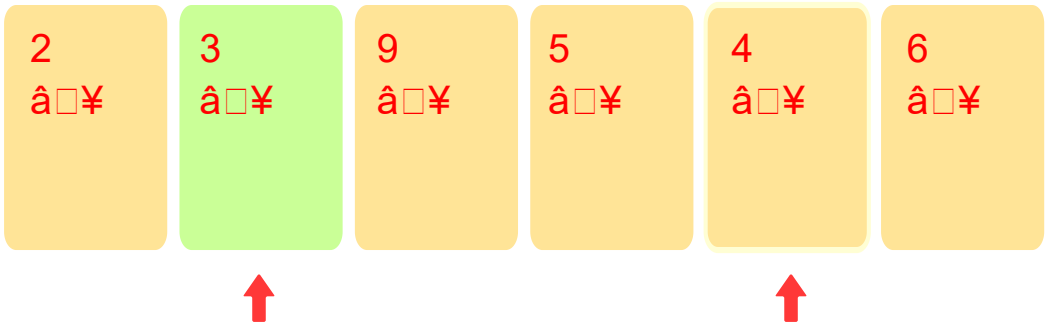
Finding next smallest



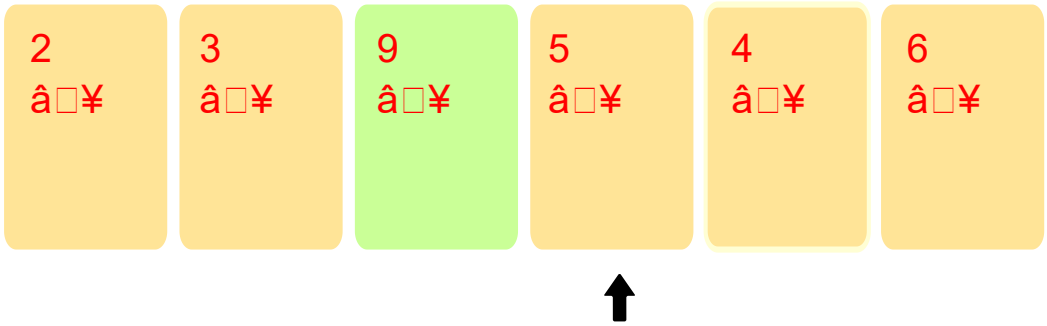
Remember 3 as it's the smallest we have seen and proceed



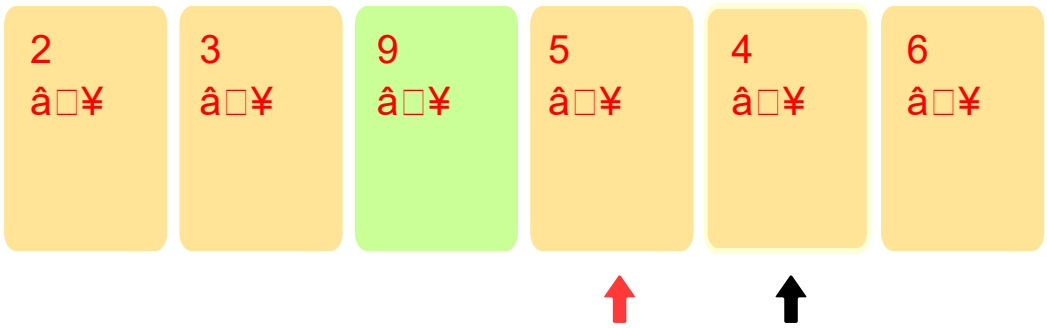
Swap 3 and 4



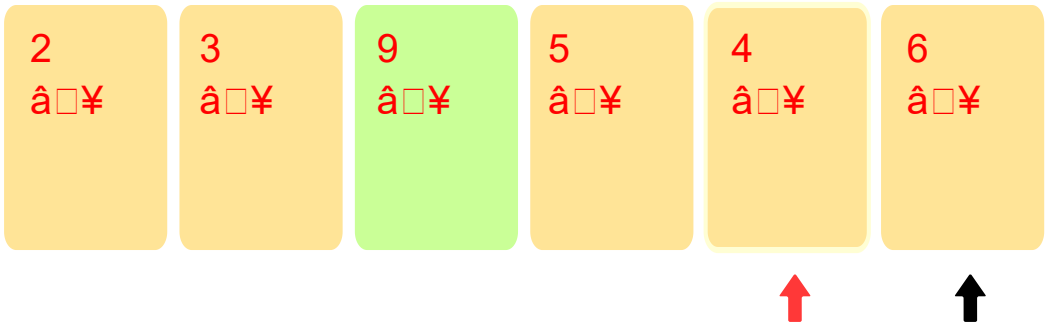
Finding next smallest



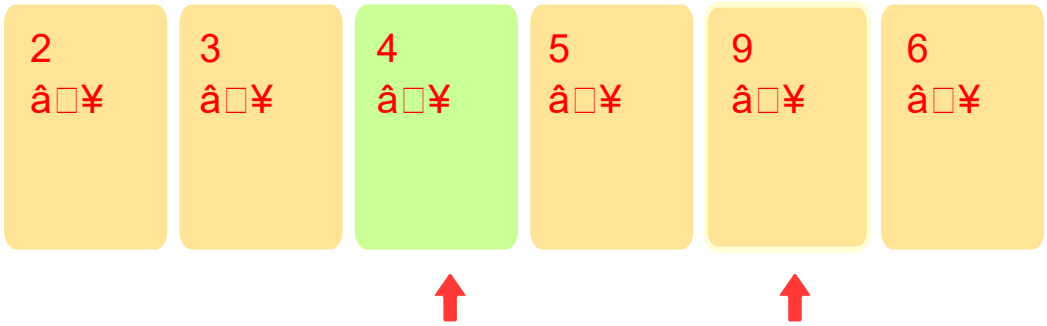
Remembering 5 as the next smallest



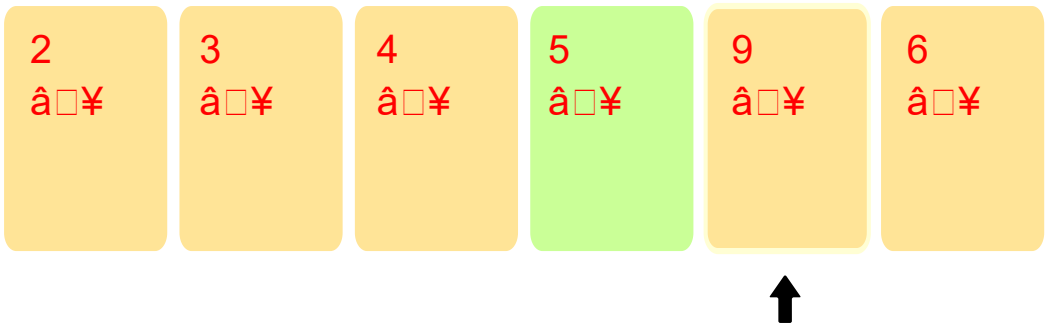
Remembering 4 as the next smallest



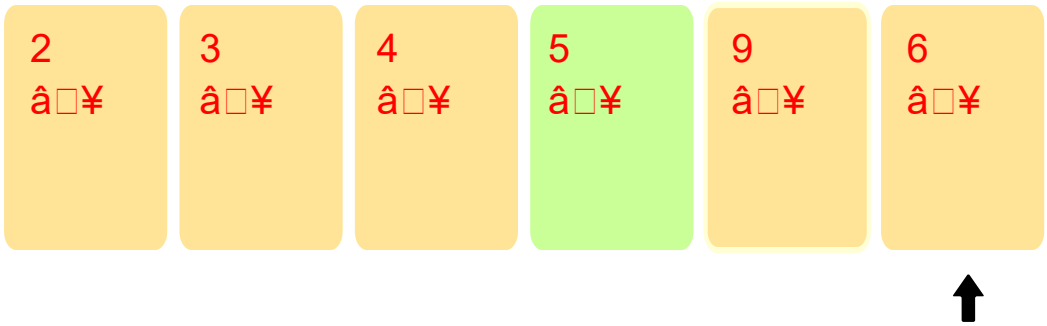
Swap 9 and 4



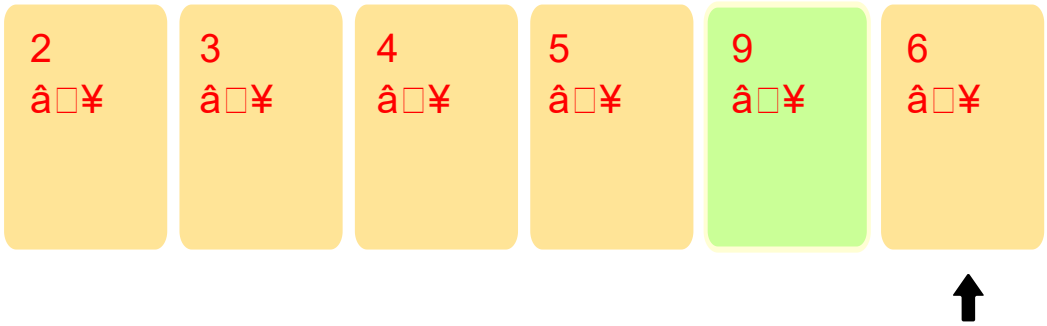
Finding next smallest



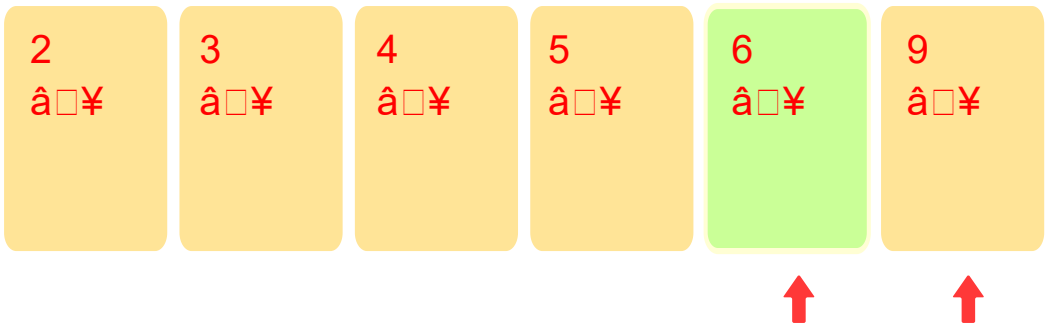
Finding next smallest

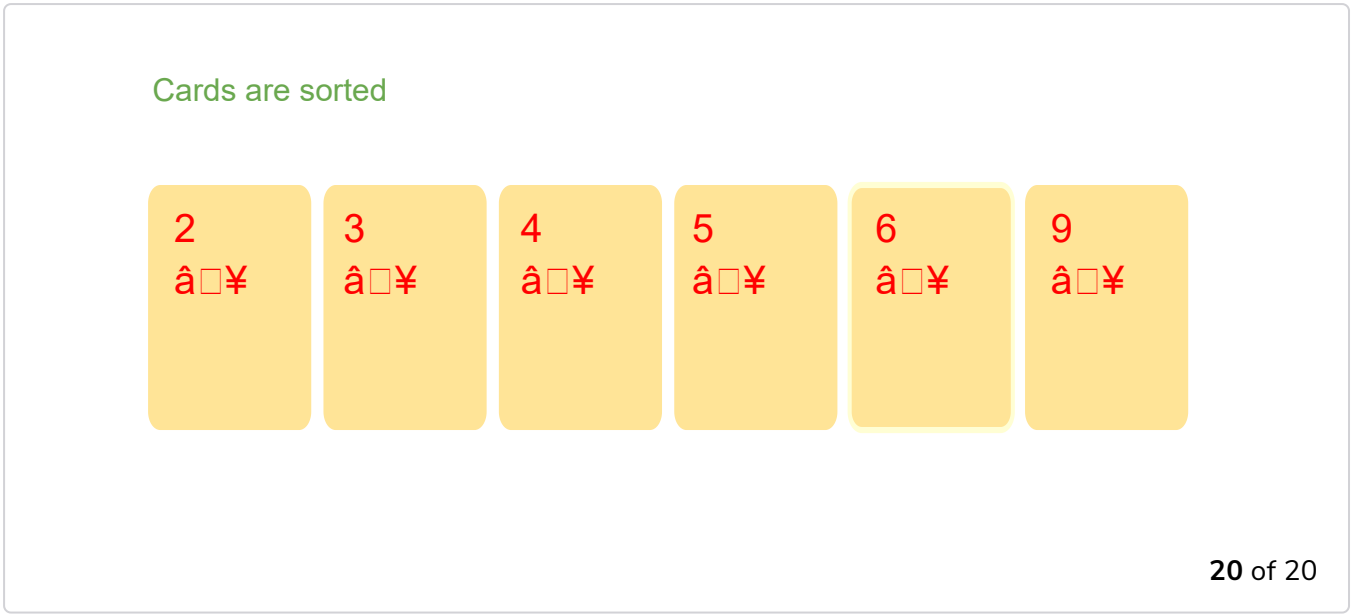


5 stays where it is. Finding next smallest.



Swapping 6 and 9.





After seeing it for yourself, what do you think aout this algorithm? What parts of it seem to take the longest? How do you think it would perform on big arrays? Keep those questions in mind as you go through and implement this algorithm.

Finding the index of the minimum element in a subarray

One of the steps in selection sort is to find the next-smallest card to put into its correct location. For example, if the array initially has values [13, 19, 18, 4, 10], we first need to find the index of the smallest value in the array. Since 4 is the smallest value, the index of the smallest value is 3.

Selection sort would swap the value at index 3 with the value at index 0, giving [4, 19, 18, 13, 10]. Now we need to find the index of the second-smallest value to swap into index 1.

It might be tricky to write code that found the index of the second-smallest value in an array. I'm sure you could do it, but there's a better way. Notice that since the smallest value has already been swapped into index 0, what we really want is to find the smallest value in the part of the array that starts at index 1. We call a section of an array a **subarray**, so that in this case, we want the index of the smallest value in the subarray that starts at index 1. For our example, if the full array is [4, 19, 18, 13, 10], then the smallest value in the

subarray starting at index 1 is 10, and it has index 4 in the original array. So index 4 is the location of the second-smallest element of the full array.

Try out that strategy in the next challenge, and then you'll have most of what you need to implement the whole selection sort algorithm.

Challenge: Find minimum in Subarray

Finish writing the function indexOfMinimum, which takes an array and a number startIndex, and returns the index of the smallest value that occurs with index startIndex or greater. If this smallest value occurs more than once in this range, then return the index of the leftmost occurrence within this range.

Java | Python | C++ | JS

```
class Solution {
  public static int indexOfMinimum(int[] array, int startIndex) {
      // Set initial values for minValue and minIndex,
      // based on the leftmost entry in the subarray:
      int minValue = array[startIndex];
      int minIndex = startIndex;

      // Loop over items starting with startIndex,
      // updating minValue and minIndex as needed:


      return minIndex;
        }
}
```

Challenge: Implement Selection Sort

Selection sort loops over positions in the array. For each position, it finds the index of the minimum value in the subarray starting at that position. Then it swaps the values at the position and at the minimum index. Write selection sort, making use of the swap and indexOfMinimum functions.

Java | Python | C++ | JS

```
class Solution {
  static void swap(int[] array, int firstIndex, int secondIndex) {
    int temp = array[firstIndex];
    array[firstIndex] = array[secondIndex];
    array[secondIndex] = temp;
    return;
  }

  static int indexOfMinimum(int[] array, int startIndex) {
    int minValue = array[startIndex];
    int minIndex = startIndex;

    for(int i = minIndex + 1; i < array.length; i++) {
      if(array[i] < minValue) {
        minIndex = i;
        minValue = array[i];
      }
    }
    return minIndex;
  };

  public static void selectionSort(int[] array) {
    // Write this method
        return;
  }
}
```

Analysis of Selection Sort

Selection sort loops over indices in the array; for each index, selection sort calls **indexOfMinimum** and swap. If the length of the array is ***n***, there are ***n*** indices in the array.

Since each execution of the body of the loop runs two lines of code, you might think that **2n** lines of code are executed by selection sort. But it's not true! Remember that **indexOfMinimum** and swap are functions: when either is called, some lines of code are executed.

How many lines of code are executed by a single call to swap? In the usual implementation, it's three lines, so that each call to swap takes constant time.

How many lines of code are executed by a single call to **indexOfMinimum**? We have to account for the loop inside indexOfMinimum. How many times does this loop execute in a given call to **indexOfMinimum**? It depends on the size of the subarray that it's iterating over. If the subarray is the whole array (as it is on the first step), the loop body runs **n** times. If the subarray is of size 6, then the loop body runs 6 times.For example, let's say the whole array is of size 8 and think about how selection sort works.

1. In the first call of indexOfMinimum, it has to look at every value in the array, and so the loop body in **indexOfMinimum** runs 8 times.
2. In the second call of **indexOfMinimum**, it has to look at every value in the subarray from indices 1 to 7, and so the loop body in **indexOfMinimum** runs 7 times.
3. In the third call, it looks at the subarray from indices 2 to 7; the loop body runs 6 times.
4. In the fourth call, the subarray from indices 3 to 7; the loop body runs 5 times.

5. ...

6. In the eighth and final call of **indexOfMimimum**, the loop body runs just 1 time.

If we total up the number of times the loop body of **indexOfMinimum** runs, we get 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 36 times.

Side note: Computing summations from 1 to n

How do you compute the sum 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 quickly? Here's a trick. Let's add the numbers in a sneaky order. First, let's add 8 + 1, the largest and smallest values. We get 9. Then, let's add 7 + 2, the second-largest and second-smallest values. Interesting, we get 9 again. How about 6 + 3? Also 9. Finally, 5 + 4. Once again, 9! So what do we have?

$$(8+1)+(7+2)+(6+3)+(5+4)=9+9+9+9$$

$$=4 * 9$$

$$=36$$

There were four pairs of numbers, each of which added up to 9. So here's the general trick to sum up any sequence of consecutive integers:

1. Add the smallest and the largest number.
2. Multiply by the number of pairs.

What if the number of integers in the sequence is odd, so that you cannot pair them all up? It doesn't matter! Just count the unpaired number in the middle of the sequence as half a pair. For example, let's sum up 1 + 2 + 3 + 4 + 5. We have two full pairs (1 + 5 and 2 + 4, each summing to 6) and one "half pair" (3, which is half of 6), giving a total of 2.5 pairs. We multiply **2.5 * 6 = 15**, and we get the right answer.

What if the sequence to sum up goes from 1 to **n**? We call this an arithmetic series. The sum of the smallest and largest numbers is **n + 1**. Because there are **n** numbers altogether, there are n/2n/2 pairs (whether **n** is odd or even). Therefore, the sum of numbers from 1 to **n** is **(n + 1)(n / 2)**, which equals $n^2/2$

+ n/2. Try out this formula for **n** = **5** and **n** = **8**.

Asymptotic running-time analysis for selection sort

The total running time for selection sort has three parts:

1. The running time for all the calls to **indexOfMinimum**.
2. The running time for all the calls to swap.
3. The running time for the rest of the loop in the **selectionSort** function.

Parts 2 and 3 are easy. We know that there are **n** calls to **swap**, and each call takes constant time. Using our asymptotic notation, the time for all calls to swap is **Θ(n)**. The rest of the loop in **selectionSort** is really just testing and incrementing the loop variable and calling **indexOfMinimum** and swap, and so that takes constant time for each of the nn iterations, for another **Θ(n)** time.

Adding up the running times for the three parts, we have $\mathbf{\Theta(n^2)}$ for the calls to **indexOfMinimum**, **Θ(n)** for the calls to swap, and **Θ(n)** for the rest of the loop in selectionSort. The $\mathbf{\Theta(n^2)}$ term is the most significant, and so we say that the running time of selection sort is $\mathbf{\Theta(n^2)}$. Notice also that no case is particularly good or particularly bad for selection sort. The loop in indexOfMinimum will always make n^2 + n/2 iterations, regardless of the input. Therefore, we can say that selection sort runs in $\Theta(n^2)$ time in all cases.

Let's see how the $\mathbf{\Theta(n^2)}$ running time affects the actual execution time. Let's say that selection sort takes approximately $\mathbf{n^2/10^6}$ seconds to sort **n** values. Let's start with a fairly small value of **n**, let's say **n** = **100**. Then the running time of selection sort is about $\mathbf{100^2/10^6}$ = **1/100** seconds. That seems pretty fast. But what if **n** = **1000**? Then selection sort takes about $\mathbf{1000^2/10^6}$ =**1** second. The array grew by a factor of 10, but the running time increased 100 times. What if **n**=**1,000,000**? Then selection sort takes $\mathbf{1{,}000{,}000^2/10^6}$=**1,000,000** seconds, which is a little more than 11.5 days. Increasing the array size by a factor of 1000 increases the running time a million times!

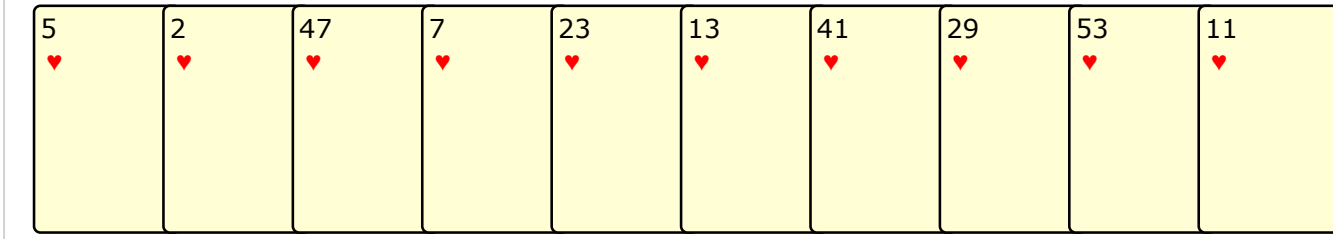
1000 increases the running time a million times.

Insertion Sort

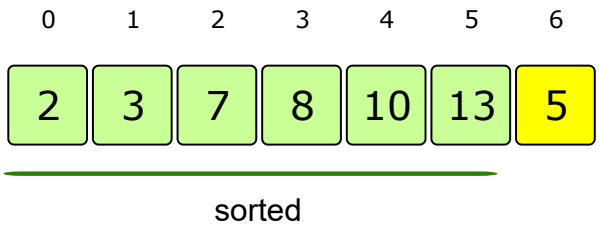
There are many different ways to sort. As selection sort runs, the subarray at the beginning of the array is sorted, but the subarray at the end is not. Selection sort scans the unsorted subarray for the next element to include in the sorted subarray.

Here's another way to think about sorting. Imagine that you are playing a card game. You're holding the cards in your hand, and these cards are sorted. The dealer hands you exactly one new card. You have to put it into the correct place so that the cards you're holding are still sorted. In selection sort, each element that you add to the sorted subarray is no smaller than the element already in the sorted subarray. But in our card example, the new card could be smaller than some of the cards you're already holding, and so you go down the line, comparing the new card against each card in your hand, until you find the place to put it. You insert the new card in the right place, and once again, your hand holds fully sorted cards. Then the dealer gives you another card, and you repeat the same procedure. Then another card, and another card, and so on, until the dealer stops giving you cards.

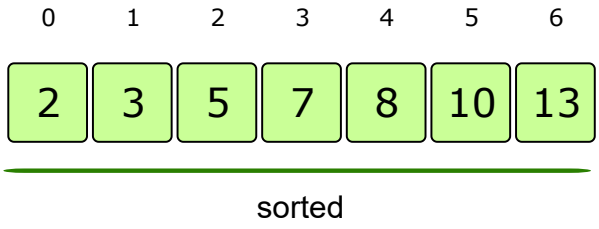
This is the idea behind insertion sort. Loop over positions in the array, starting with index 1. Each new position is like the new card handed to you by the dealer, and you need to insert it into the correct place in the sorted subarray to the left of that position. Here's a visualization that steps through that:



In terms of arrays, imagine that the subarray from index 0 through index 55 is already sorted, and we want to insert the element currently in index 6 into this sorted subarray, so that the subarray from index 0 through index 6 is sorted. Here's how we start:

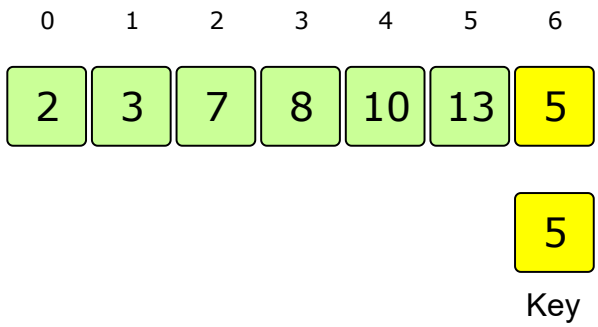


And here's what the subarray should look like when we're done:

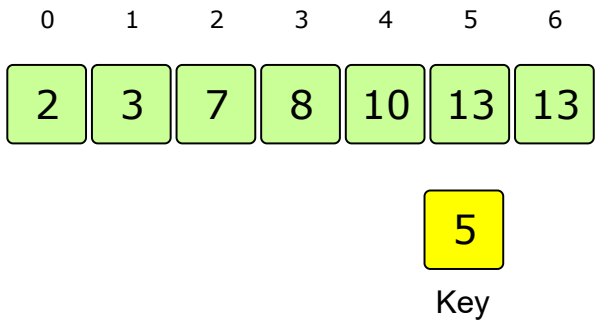


To insert the element in position 6 into the subarray to its left, we repeatedly compare it with elements to its left, going right to left. Let's call the element in position 6 the **key**. Each time we find that the key is less than an element to its

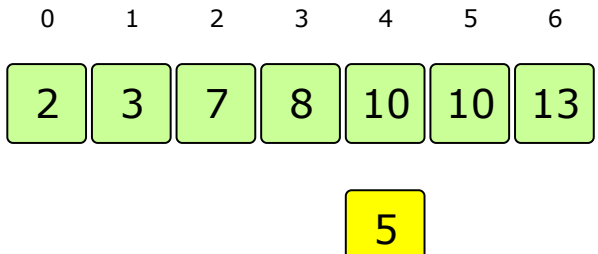
left, we slide that element one position to the right, since we know that the key will have to go to that element's left. We'll need to do two things to make this idea work: we need to have a **slide** operation that slides an element one position to the right, and we need to save the value of the key in a separate place (so that it doesn't get overridden by the element to its immediate left). In our example, let's pull the element at index 6 into a variable called **key**:



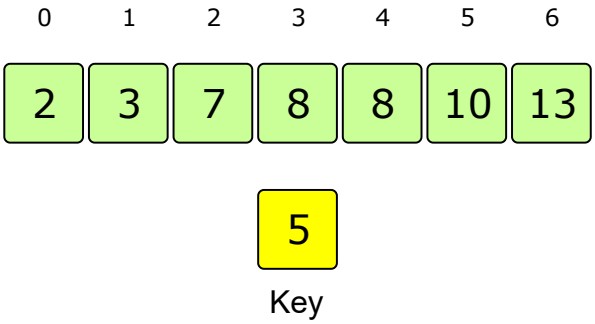
Now, we compare key with the element at position 5. We find that key (5) is less than the element at position 5 (13), and so we slide this element over to position 6:



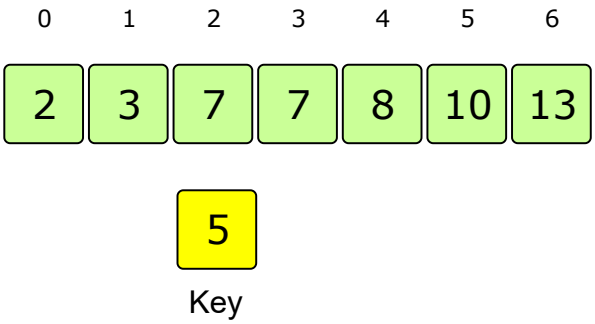
Notice that the slide operation just copies the element one position to the right. Next, we compare key with the element at position 4. We find that key (5) is less than the element at position 4 (10), and we slide this element over:



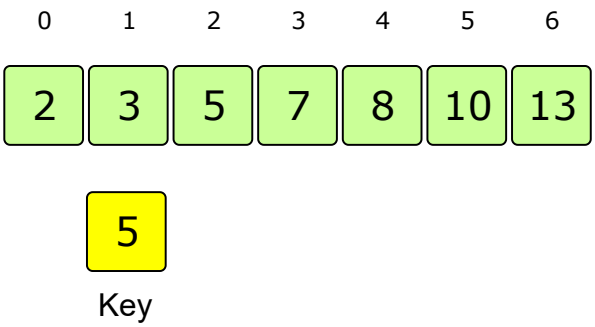
Next, we compare key with the element at position 3, and we slide this element over:



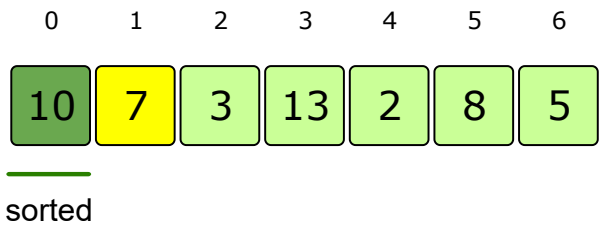
The same happens with the element at position 2:



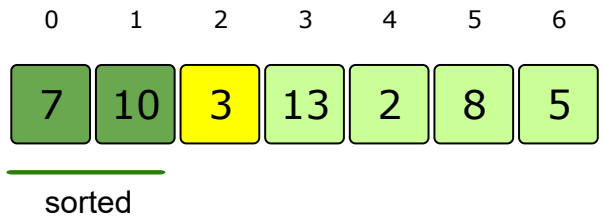
Now we come to the element at position 1, which has the value 3. This element is less than key, and so we do not slide it over. Instead, we drop key into the position immediately to the right of this element (that is, into position 2), whose element was most recently slid to the right. The result is that the subarray from index 0 through index 6 has become sorted:



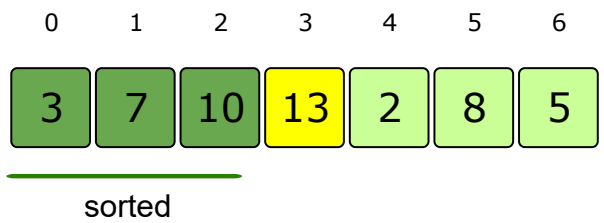
Insertion sort repeatedly inserts an element in the sorted subarray to its left. Initially, we can say that the subarray containing only index 0 is sorted, since it contains only one element, and how can a single element not be sorted with respect to itself? It must be sorted. Let's work through an example. Here's our initial array:



Because the subarray containing just index 0 is our initial sorted subarray, the first key is in index 1. (We'll show the sorted subarray in red, the key in yellow, and the part of the array that we have yet to deal with in blue.) We insert the key into the sorted subarray to its left:

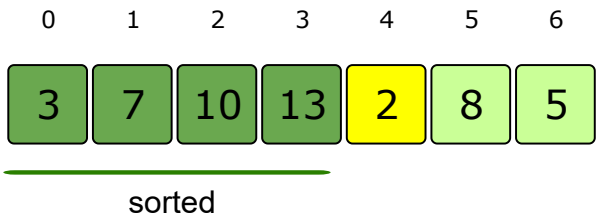


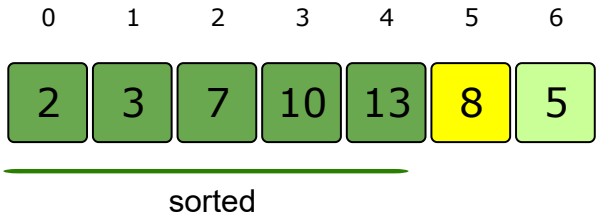
Now the sorted subarray runs from index 0 through index 1, and the new key is in index 2. We insert it into the sorted subarray to its left:

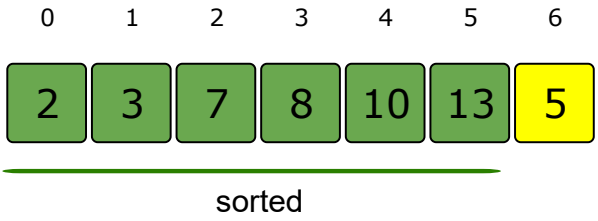


We keep going, considering each array element in turn as the key, and

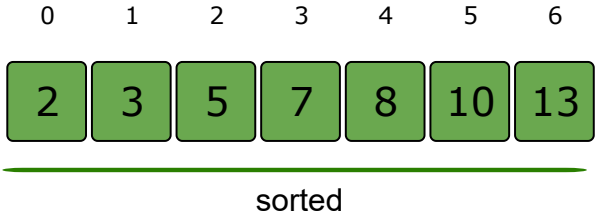
inserting it into the sorted subarray to its left:







Once we've inserted that rightmost element in the array, we have sorted the entire array:

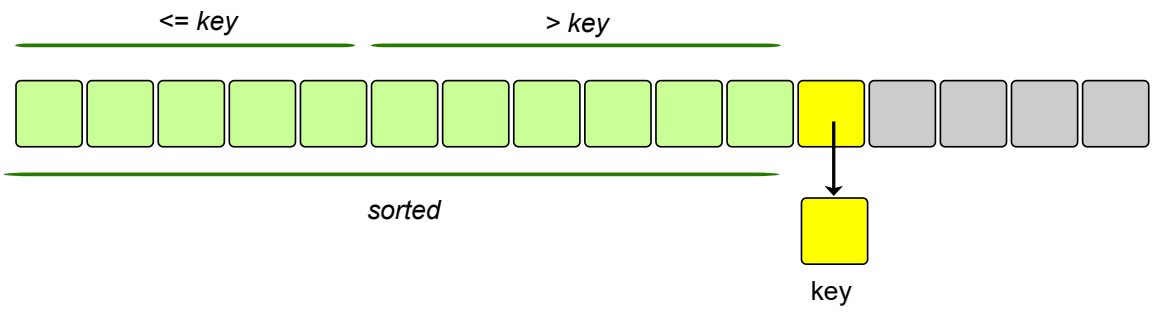


A couple of situations that came up in our example bear a little more scrutiny: when the key being inserted is less than all elements to its left (as when we inserted keys 2 and 3), and when it's greater than or equal to all elements to its left (as when we inserted key 13). In the former case, every element in the subarray to the left of the key slides one position to the right, and we have to

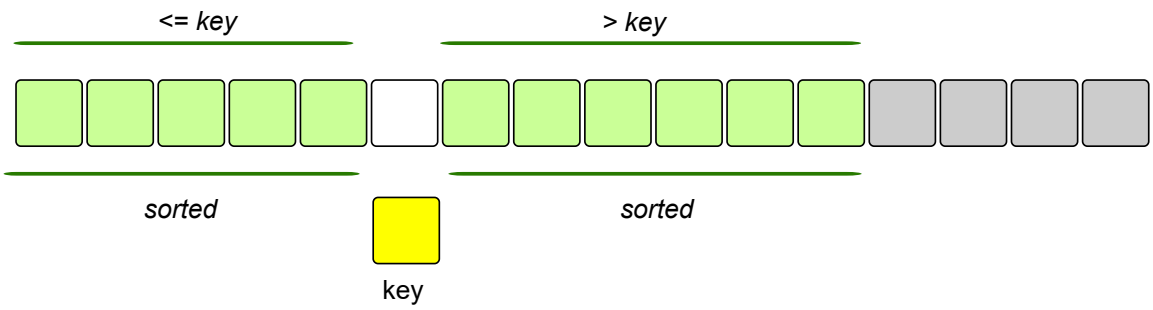
stop once we've run off the left end of the array. In the latter case, the first time we compare the key with an element to its left, we find that the key is already in its correct position relative to all elements to its left; no elements slide over and the key drops back into the position in which it started.

Inserting a value into Sorted Subarray

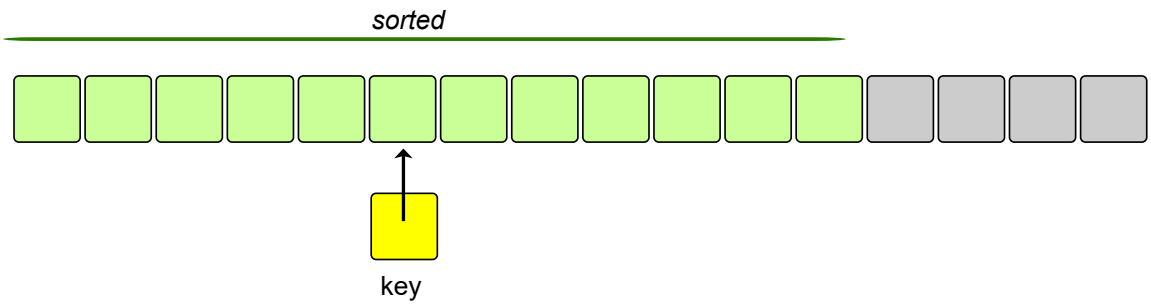
The main step in insertion sort is making space in an array to put the current value, which is stored in the variable key. As we saw above, we go through the subarray to the left of key's initial position, right to left, sliding each element that is greater than key one position to the right. Once we find an element that is less than key, or equal to key, we stop sliding and copy key into the vacated position just to the right of this element. (Of course, the position is not truly vacated, but its element was slid over to the right.) This diagram shows the idea:



1 of 3



2 of 3


sorted
key

Challenge: Insert a Value Before an Index in Sorted Order

WE'LL COVER THE FOLLOWING

Problem Statement:

In this challenge, you have to implement the insert function which will be taking three parameters as inputs: `array`, `rightIndex`, and `value`.

Explanation:

Before the insert function is called:

- the elements from `array[0]` to `array[rightIndex]` are sorted in ascending order.
- the `value` is in the array at an index from `array[rightIndex]` to `array[end]`.

After calling the insert function:

- the `value` from the array is removed from its original position and it is then inserted between `array[0]` to `array[rightIndex+1]`, maintaining the ascending order.

In order to do this, the insert function will need to make room for `value` by

moving items that are greater than `value` to the right. It should start at `rightIndex`, and stop when it finds an item that is less than or equal to `value`, or when it reaches the beginning of the array. Once the function has made room for the value, it can write it to the array.

Function Prototype:

```
void insert(int[] array, int rightIndex, int value)
```

Sample Input:

```
array = [2, 3, 5, 7, 11, 13, 9, 6]
rightIndex = 5
value = 9
```

Sample Output:

```
[2, 3, 5, 7, 9, 11, 13, 6]
```

Coding Exercise:

Understand the problem first, before implementing it. If you get stuck anywhere, you are free to refer to the solution. Good Luck!

Java | Python | C++ | JS

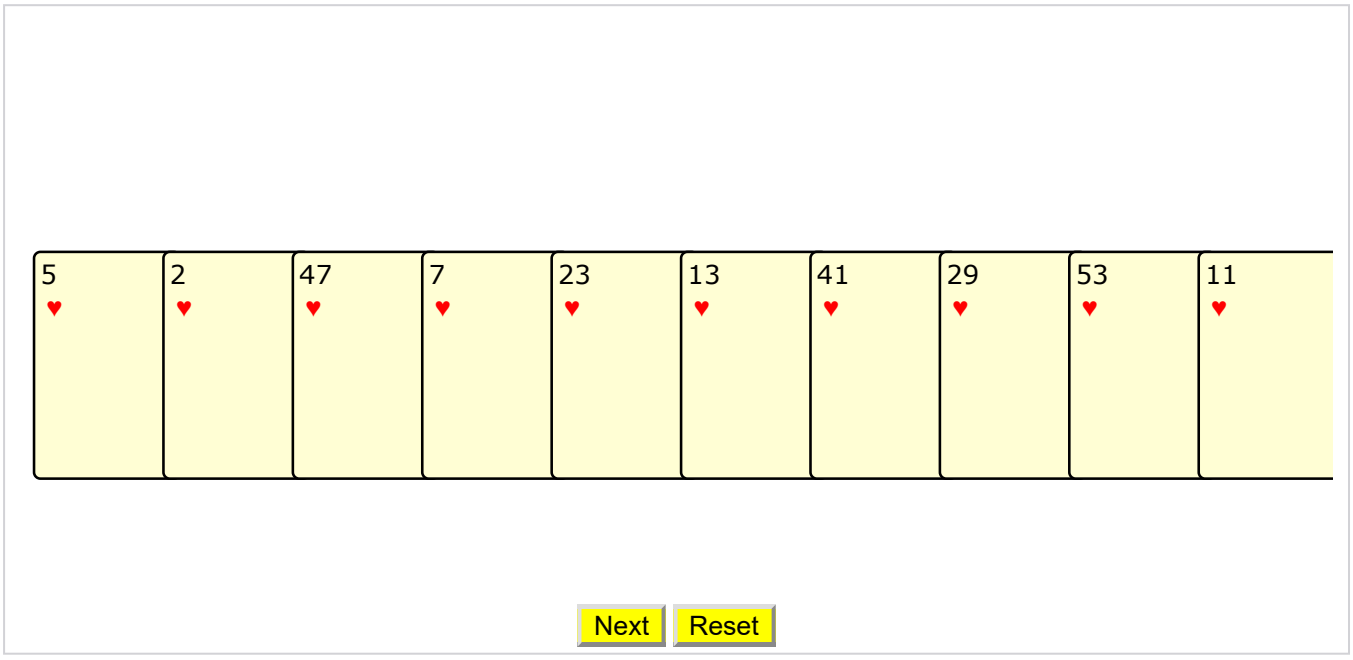
```
class Solution {
  public static void insert(int[] array, int rightIndex, int value) {
    // write this method

  }
};
```

Insertion Sort pseudocode

Now that you know how to insert a value into a sorted subarray, you can implement insertion sort:

1. Call insert to insert the element that starts at index 1 into the sorted subarray in index 0.
2. Call insert to insert the element that starts at index 2 into the sorted subarray in indices 0 through 1.
3. Call insert to insert the element that starts at index 3 into the sorted subarray in indices 0 through 2.
4. ...
5. Finally, call insert to insert the element that starts at index n-1n−1 into the sorted subarray in indices 0 through **n−2**.

As a reminder, here's the visualization that steps through the algorithm on a deck of cards:



Challenge: Implement Insertion Sort

Insertion sort loops over items in the array, inserting each new item into the subarray before the new item.

Write insertion sort, making use of the insert function that was written in the previous challenge.

Java | Python | C++ | JS

```
class Solution {
  static void insert(int[] array, int rightIndex, int value) {
    int j = rightIndex;
    for(;
        j >= 0 && array[j] > value;
        j--) {
      array[j + 1] = array[j];
    }
    array[j + 1] = value;
  }

  public static void insertionSort(int[] array) {
    //Write this method

    return;
  }
}
```

Analysis of Insertion Sort

Like selection sort, insertion sort loops over the indices of the array. It just calls insert on the elements at indices $\mathbf{1,2,3,\ldots,n-1}$. Just as each call to **indexOfMinimum** took an amount of time that depended on the size of the sorted subarray, so does each call to insert. Actually, the word "**does**" in the previous sentence should be "**can**," and we'll see why.

Let's take a situation where we call insert and the value being inserted into a subarray is less than every element in the subarray. For example, if we're inserting 0 into the subarray [2, 3, 5, 7, 11], then every element in the subarray has to slide over one position to the right. So, in general, if we're inserting into a subarray with $\boldsymbol{k}$ elements, all $\boldsymbol{k}$ might have to slide over by one position. Rather than counting exactly how many lines of code we need to test an element against a key and slide the element, let's agree that it's a constant number of lines; let's call that constant $\boldsymbol{c}$. Therefore, it could take up to $\boldsymbol{c \cdot k}$ lines to insert into a subarray of $\boldsymbol{k}$ elements.

Suppose that upon every call to insert, the value being inserted is less than every element in the subarray to its left. When we call insert the first time, $\mathbf{k=1}$. The second time, $\mathbf{k=2}$. The third time, k=3k=3. And so on, up through the last time, when $\mathbf{k=n-1}$. Therefore, the total time spent inserting into sorted subarrays is

$$\mathbf{c \cdot 1 + c \cdot 2 + c \cdot 3 + \cdots c \cdot (n-1) = c \cdot (1 + 2 + 3 + \cdots + (n-1))}$$

That sum is an arithmetic series, except that it goes up to n−1 rather than n. Using our formula for arithmetic series, we get that the total time spent inserting into sorted subarrays is

$$\mathbf{c \cdot (n-1+1)((n-1)/2) = cn^2/2 - cn/2}$$

Using big-Θ notation, we discard the low-order term $\mathbf{cn/2}$ and the constant factors $\mathbf{c}$ and 1/2, getting the result that the running time of insertion sort, in

this case, is $\Theta(n^2)$.

Can insertion sort take less than $\mathbf{\Theta(n^2)}$ time? The answer is yes. Suppose we have the array [2, 3, 5, 7, 11], where the sorted subarray is the first four elements, and we're inserting the value 11. Upon the first test, we find that 11 is greater than 7, and so no elements in the subarray need to slide over to the right. Then this call of insert takes just constant time. Suppose that every call of insert takes constant time. Because there are n-1n−1 calls to insert, if each call takes time that is some constant cc, then the total time for insertion sort is $\mathbf{c \cdot (n-1)}$, which is $\mathbf{\Theta(n)}$, not $\mathbf{\Theta(n^2)}$.

Can either of these situations occur? Can each call to **insert** cause every element in the subarray to slide one position to the right? Can each call to insert cause no elements to slide? The answer is yes to both questions. A call to insert causes every element to slide over if the key being inserted is less than every element to its left. So, if every element is less than every element to its left, the running time of insertion sort is $\mathbf{\Theta(n^2)}$. What would it mean for every element to be less than the element to its left? The array would have to start out in reverse sorted order, such as [11, 7, 5, 3, 2]. So a reverse-sorted array is the worst case for insertion sort.

How about the opposite case? A call to **insert** causes no elements to slide over if the key being inserted is greater than or equal to every element to its left. So, if every element is greater than or equal to every element to its left, the running time of insertion sort is $\mathbf{\Theta(n)}$. This situation occurs if the array starts out already sorted, and so an already-sorted array is the best case for insertion sort.

What else can we say about the running time of insertion sort? Suppose that the array starts out in a random order. Then, on average, we'd expect that each element is less than half the elements to its left. In this case, on average, a call to insert on a subarray of **k** elements would slide **k/2** of them. The running time would be half of the worst-case running time. But in asymptotic notation, where constant coefficients don't matter, the running time in the average case would still be $\mathbf{\Theta(n^2)}$, just like the worst case.

What if you knew that the array was "almost sorted": every element starts out at most some constant number of positions, say 17, from where it's supposed

to be when sorted? Then each call to insert slides at most 17 elements, and the

time for one call of insert on a subarray of $\mathbf{k}$ elements would be at most 17·c. Over all $\mathbf{n-1}$ calls to insert, the running time would be $\mathbf{17 \cdot c \cdot (n-1)}$, which is $\mathbf{\Theta(n)}$, just like the best case. So insertion sort is fast when given an almost-sorted array.

To sum up the running times for insertion sort:

- Worst case: $\mathbf{\Theta(n^2)}$.
- Best case: $\mathbf{\Theta(n)}$.
- Average case for a random array: $\mathbf{\Theta(n^2)}$
- "Almost sorted" case: $\mathbf{\Theta(n)}$.

If you had to make a blanket statement that applies to all cases of insertion sort, you would have to say that it runs in $\mathbf{O(n^2)}$ time. You cannot say that it runs in $\mathbf{\Theta(n^2)}$ time in all cases, since the best case runs in $\mathbf{\Theta(n)}$ time. And you cannot say that it runs in $\mathbf{\Theta(n)}$ time in all cases, since the worst-case running time is $\mathbf{\Theta(n^2)}$.

Recursion

Have you ever seen a set of Russian dolls? At first, you see just one figurine, usually painted wood, that looks something like this:

You can remove the top half of the first doll, and what do you see inside? Another, slightly smaller, Russian doll!

You can remove that doll and separate its top and bottom halves. And you see yet another, even smaller, doll:

And once more:

And you can keep going. Eventually you find the teeniest Russian doll. It is just one piece, and so it does not open:

We started with one big Russian doll, and we saw smaller and smaller Russian dolls, until we saw one that was so small that it could not contain another.

What do Russian dolls have to do with algorithms? Just as one Russian doll has within it a smaller Russian doll, which has an even smaller Russian doll within it, all the way down to a tiny Russian doll that is too small to contain another, we'll see how to design an algorithm to solve a problem by solving a smaller instance of the same problem, unless the problem is so small that we can just solve it directly. We call this technique **recursion**.

Recursion has many, many applications. In this module, we'll see how to use recursion to compute the factorial function, to determine whether a word is a palindrome, to compute powers of a number, to draw a type of fractal, and to solve the ancient Towers of Hanoi problem. Later modules will use recursion to solve other problems, including sorting.

The factorial function

For our first example of recursion, let's look at how to compute the factorial function. We indicate the factorial of n by **n!**. It's just the product of the integers **1** through **n**. For example, **5!** equals **1·2·3·4·5**, or **120**. (Note: Wherever we're talking about the factorial function, all exclamation points refer to the factorial function and are not for emphasis.)

You might wonder why we would possibly care about the factorial function. It's very useful for when we're trying to count how many different orders there are for things or how many different ways we can combine things. For example, how many different ways can we arrange n things? We have n choices for the first thing. For each of these n choices, we are left with n-1 choices for the second thing, so that we have **n·(n−1)** choices for the first two things, in order. Now, for each of these first two choices, we have **n−2** choices for the third thing, giving us **n·(n−1)·(n−2)** choices for the first three things, in order. And so on, until we get down to just two things remaining, and then just one thing remaining. Altogether, we have **n·(n−1)·(n−2)⋯2·1** ways that we can order nn things. And that product is just **n!** (**n** factorial), but with the product written going from **n** down to **1** rather than from **1** up to **n**.

Another use for the factorial function is to count how many ways you can choose things from a collection of things. For example, suppose you are going on a trip and you want to choose which T-shirts to take. Let's say that you own **n** T-shirts but you have room to pack only **k** of them. How many different ways can you choose **k** T-shirts from a collection of **n** T-shirts? The answer (which we won't try to justify here) turns out to be

$$\frac{n!}{k!\cdot(n-k)!}$$

The factorial function is defined for all positive integers, along with 0. What value should 0! have? It's the product of all integers greater than or equal to **1**

and less than or equal to **0**. But there are no such integers. Therefore, we

define **0!** to equal the identity for multiplication, which is **1**. (Defining **0!** = **1** meshes nicely with the formula for choosing **k** things out of **n** things. Suppose that we want to know how many ways there are to choose n things out of **n** things. That's easy, because there is only one way: choose all **n** things. So now we know that, using our formula, **n!/(n!·(n−n)!)** should equal **1**. But **(n-n)!** is **0!**, so now we know that **n!/(n!·0!)** should equal 1. If we cancel out the **n!** in both the numerator and denominator, we see that **1/(0!)** should equal **1**, and it does because **0!** equals **1**.)

So now we have a way to think about **n!**. It equals **1** when **n** = **0**, and it equals **1·2⋯(n−1)·n** when **n** is positive.

Challenge: Iterative factorial

Finish the provided factorial function, so that it returns the value **n!**.

Your code should use a for loop to compute the product **1 * 2 * 3 * ... * n**. If you write the code carefully, you won't need a special case for when **n** equals **0**.

Java | Python | C++ | JS

```
class Solution {
  public static int factorial(int n) {
    int result = 0;

    // Implement this method

    return result;
  }
}
```

Recursive factorial

For positive values of **n**, let's write **n!** as we did before, as a product of numbers starting from n and going down to **1: n!** = **n**·**(n−1)**⋯**2**·**1**. But notice that **(n−1)**⋯**2**·**1** is another way of writing **(n−1)!**, and so we can say that **n!** = **n**·**(n−1)!**. Did you see what we just did? We wrote **n!** as a product in which one of the factors is **(n−1)!**. We said that you can compute **n!** by computing **(n−1)!** and then multiplying the result of computing **(n−1)!** by **n**. You can compute the factorial function on n by first computing the factorial function on **n−1**. We say that computing **(n-1)!** is a subproblem that we solve to compute **n!**.

Let's look at an example: computing **5!**.

- You can compute **5!** as **5**·**4!**.
- Now you need to solve the subproblem of computing **4!**, which you can compute as **4**·**3!**.
- Now you need to solve the subproblem of computing **3!**, which is **3**·**2!**.
- Now **2!**, which is **2**·**1!**.
- Now you need to compute **1!**. You could say that **1!** equals **1**, because it's the product of all the integers from **1** through **1**. Or you can apply the formula that **1!** = **1**·**0!**. Let's do it by applying the formula.
- We defined **0!** to equal **1**.
- Now you can compute **1!** = **1**·**0!**=**1**.
- Having computed **1!** = **1**, you can compute **2!** = **2**·**1!** = **2**.
- Having computed **2!** = **2**, you can compute **3!** = **3**·**2!** = **6**.
- Having computed **3!** = **6**, you can compute **4!** = **4**·**3!** = **24**.
- Finally, having computed **4!** = **24**, you can finish up by computing **5!** = **5**·**4!** = **120**.

So now we have another way of thinking about how to compute the value of n!, for all nonnegative integers n:

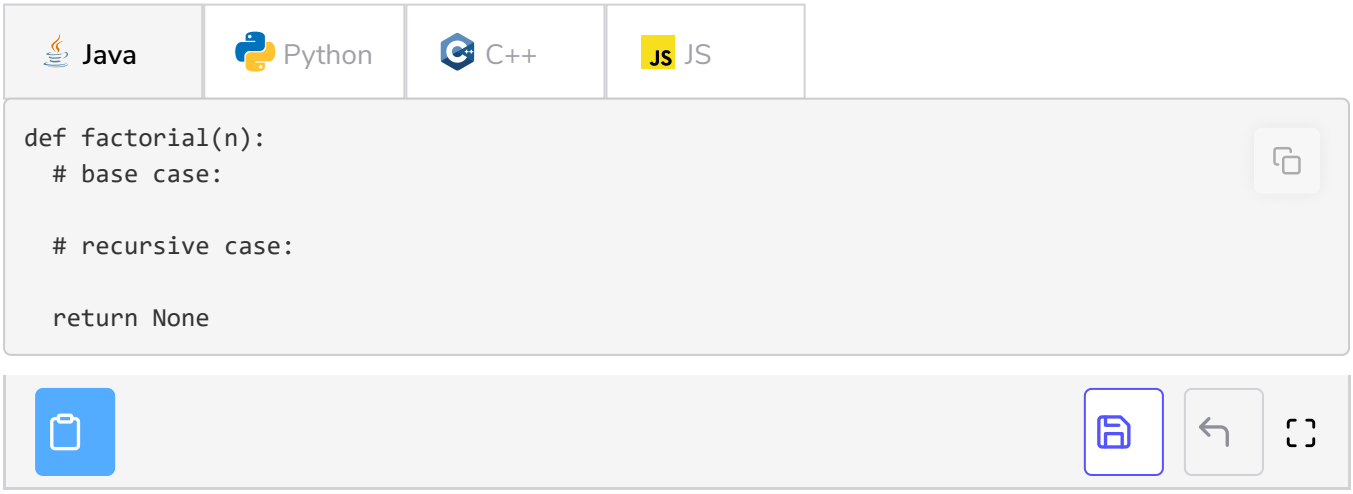
- If **n** = **0**, then declare that **n!** = **1**.
- Otherwise, **n** must be positive. Solve the subproblem of computing **(n−1)!**, multiply this result by **n**, and declare **n!** equal to the result of this product.

When we're computing **n!** in this way, we call the first case, where we immediately know the answer, the **base case**, and we call the second case, where we have to compute the same function but on a different value, the **recursive case**.

Challenge: Recursive factorial

Finish the provided factorial function, so that it returns the value **n!**.

Your code should use a for loop to compute the product **1 * 2 * 3 * ... * n**. If you write the code carefully, you won't need a special case for when **n** equals **0**.



Properties of recursive algorithms

Here is the basic idea behind recursive algorithms:

> To solve a problem, solve a subproblem that is a smaller instance of the same problem, and then use the solution to that smaller instance to solve the original problem.

When computing **n!**, we solved the problem of computing **n!** (the original problem) by solving the subproblem of computing the factorial of a smaller number, that is, computing **(n-1)!** (the smaller instance of the same problem), and then using the solution to the subproblem to compute the value of **n!**.
In order for a recursive algorithm to work, the smaller subproblems must eventually arrive at the base case. When computing **n!**, the subproblems get smaller and smaller until we compute **0!**. You must make sure that eventually, you hit the base case.
For example, what if we tried to compute the factorial of a negative number using our recursive method? To compute **(-1)!**, you would first try to compute **(−2)!**, so that you could multiply the result by **−1**. But to compute **(−2)!**, you would first try to compute **(−3)!**, so that you could multiply the result by **−2**. And then you would try to compute **(−3)!**, and so on. Sure, the numbers are getting smaller, but they're also getting farther and farther away from the base case of computing **0!**. You would never get an answer.

Even if you can guarantee that the value of **n** is not negative, you can still get into trouble if you don't make the subproblems progressively smaller. Here's an example. Let's take the formula **n!** = **n**·**(n−1)!** and divide both sides by **n**, giving **n!**/**n** = **(n−1)!**. Let's make a new variable, **m**, and set it equal to **n**+**1**. Since our formula applies to any positive number, let's substitute **m** for **n**, giving **m!**/**m** = **(m−1)!**. Since **m** = **n** + **1**, we now have **(n+1)!** / **(n+1)** = **(n+1-1)!**. Switching sides and noting that **n+1-1** = **n** gives us **n!** = **(n+1)!** / **(n+1)**. This formula leads us to believe that you can compute **n!** by first computing

(n+1)! and then dividing the result by **n+1**. But to compute **(n+1)!**, you would

have to compute **(n+2)!**, then **(n+3)!**, and so on. You would never get to the base case of 0. Why not? Because each recursive subproblem asks you to compute the value of a larger number, not a smaller number. If **n** is positive, you would never hit the base case of 0.

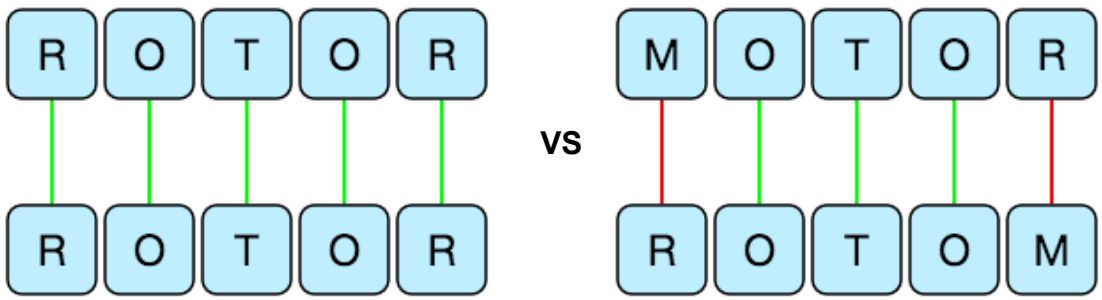
We can distill the idea of recursion into two simple rules:

1. Each recursive call should be on a smaller instance of the same problem, that is, a smaller subproblem.
2. The recursive calls must eventually reach a base case, which is solved without further recursion.

Let's go back to the Russian dolls. Although they don't figure into any algorithms, you can see that each doll encloses all the smaller dolls (analogous to the recursive case), until the smallest doll that does not enclose any others (like the base case).

Using recursion to determine whether a word is a palindrome

A **palindrome** is a word that is spelled the same forward and backward. For example, *rotor* and *redder* are palindromes, but *motor* is not.

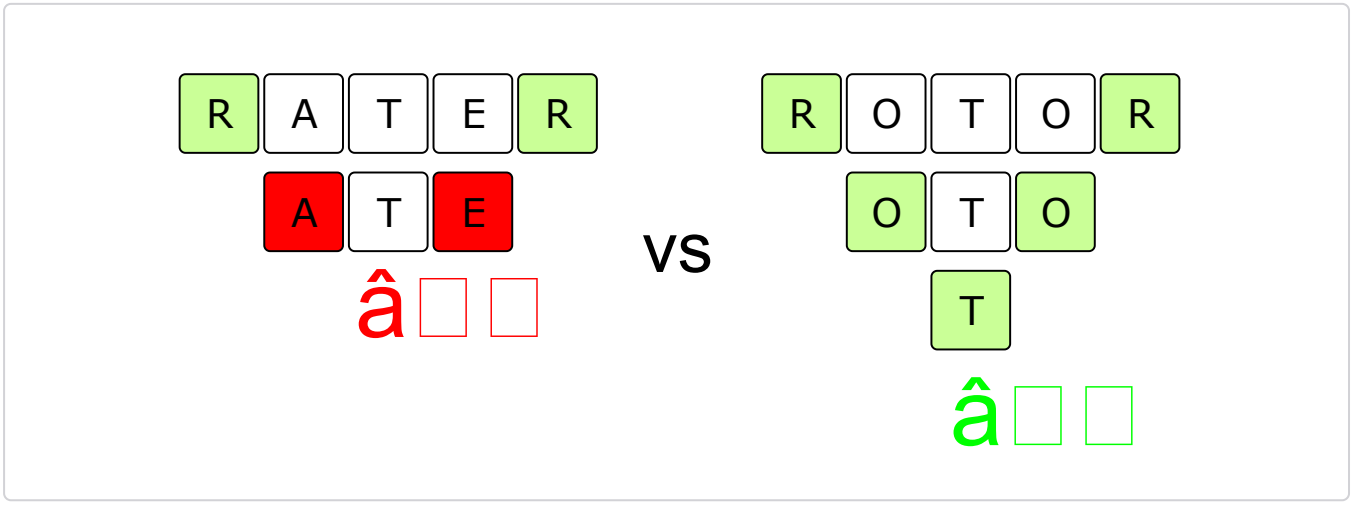


How can you use recursion to determine whether a word is a palindrome? Let's start by understanding what's a base case. Consider the word a. It's a palindrome. In fact, we don't have to think of palindromes as actual words in the English language (or whatever language you'd like to consider). We can think of a palindrome as just any sequence of letters that reads the same forward and backward, such as xyzyzyx. We call a sequence of letters a **string**. So we can say that any string containing just one letter is by default a palindrome. Now, a string can contain no letters; we call a string of zero letters an **empty string**. An empty string is also a palindrome, since it "reads" the same forward and backward. So now let's say that any string containing at most one letter is a palindrome. That's our base case: a string with exactly zero letters or one letter is a palindrome.

What if the string contains two or more letters? Here's where we'll have our recursive case. Consider the palindrome rotor. Its first and last letters are the same, and this trait must hold for any palindrome. On the other hand, if the first and last letters are not the same, as in motor, then the string cannot

possibly be a palindrome. So now we have a way to declare that a string is not a palindrome: when its first and last letters are different. We can think of this situation as another base case, since we have the answer immediately. Going back to when the first and last letters are the same, what does that tell us? The string might be a palindrome. Then again, it might not be. In the string rater, the first and last letters are the same, but the string is not a palindrome. Suppose we strip off the first and last letters, leaving the string ate. Then the first and last letters of this remaining string are not the same, and so we know that rater is not a palindrome.

So here's how we can recursively determine whether a string is a palindrome. If the first and last letters differ, then declare that the string is not a palindrome. Otherwise, strip off the first and last letters, and determine whether the string that remains—the subproblem—is a palindrome. Declare the answer for the shorter string to be the answer for the original string. Once you get down to a string with no letters or just one letter, declare it to be a palindrome. Here's a visualization of that for two words that we discussed:



How would we describe that in pseudocode?

- If the string is made of no letters or just one letter, then it is a palindrome.
- Otherwise, compare the first and last letters of the string.
    - If the first and last letters differ, then the string is not a palindrome.
    - Otherwise, the first and last letters are the same. Strip them from the string, and determine whether the string that remains is a palindrome. Take the answer for this smaller string and use it as the answer for the original string.

Challenge: is a string a palindrome?

In this challenge, you'll make it so the isPalindrome() function returns true if the provided string is a palindrome, and false otherwise. Here are the cases that we need to handle.

Base case #1

Start by implementing the first base case: if the length of the string is 0 or 1, isPalindrome() should return true.

Base case #2

If the first and last characters of the string are different, then we know immediately that the string is not a palindrome.

Recursive case

Finally, write the recursive case. Remove the first and last characters from the string and call isPalindrome function with the remaining string.

Python C++ JS JS

```
def isPalindrome(str):
  # base case #1


  # base case #2


  # recursive case

  return None
```

Computing powers of a number

Although most languages have a builtin pow function that computes powers of a number, you can write a similar function recursively, and it can be very efficient. The only hitch is that the exponent has to be an integer.

Suppose you want to compute $\mathbf{x^n}$, where **x** is any real number and **n** is any integer. It's easy if **n** is **0**, since $\mathbf{x^0 = 1}$ no matter what **x** is. That's a good base case.

So now let's see what happens when **n** is positive. Let's start by recalling that when you multiply powers of **x**, you add the exponents: $\mathbf{x^a \cdot x^b = x^{a+b}}$ for any base **x** and any exponents **a** and **b**. Therefore, if **n** is positive and even, then $\mathbf{x^n = x^{n/2} \cdot x^{n/2}}$. If you were to compute $\mathbf{y = x^{n/2}}$ recursively, then you could compute $\mathbf{x^n = y \cdot y}$. What if **n** is positive and odd? Then $\mathbf{x^n = x^{n-1} \cdot x}$, and **n-1** either is **0** or is positive and even. We just saw how to compute powers of **x** when the exponent either is 0 or is positive and even. Therefore, you could compute $\mathbf{x^{n-1}}$ recursively, and then use this result to compute $\mathbf{x^n = x^{n-1} \cdot x}$. What about when **n** is negative? Then $\mathbf{x^n = 1/x^{-n}}$, and the exponent **-n** is positive. So you can compute $\mathbf{x^{-n}}$ recursively and take its reciprocal. Putting these observations together, we get the following recursive algorithm for computing $\mathbf{x^n}$:

- The base case is when $\mathbf{n = 0}$, and $\mathbf{x^0 = 1.}$
- If **n** is positive and even, recursively compute $\mathbf{y = x^{n/2}}$, and then $\mathbf{x^n = y \cdot y}$. Notice that you can get away with making just one recursive call in this case, computing $\mathbf{x^{n/2}}$ just once, and then you multiply the result of this recursive call by itself.
- If **n** is positive and odd, recursively compute $\mathbf{x^{n-1}}$, so that the exponent either is **0** or is positive and even. Then, $\mathbf{x^n = x^{n-1} \cdot x}$
- If **n** is negative, recursively compute $\mathbf{x^{-n}}$, so that the exponent becomes

positive. Then, $\mathbf{x^n = 1/x^{-n}}$.

Challenge: Recursive Powers

Write a recursive function **power(x, n)** that returns the value of $\mathbf{x^n}$ (assume that **n** is an integer). Here are the 4 following cases that you need to handle.

1. Base Case

Start by writing the base case. $\mathbf{x^0 = 1}$ for any value of **x**.

2. Recursive case: n is odd

In this step, write the recursive case for which n is odd. Assume you have a function isOdd() to check if n is odd.

3. Recursive case: n is even

In this step, write the recursive case for which n is even. Assume you have a function isEven() to check if n is even.

4. Recursive case: n is negative

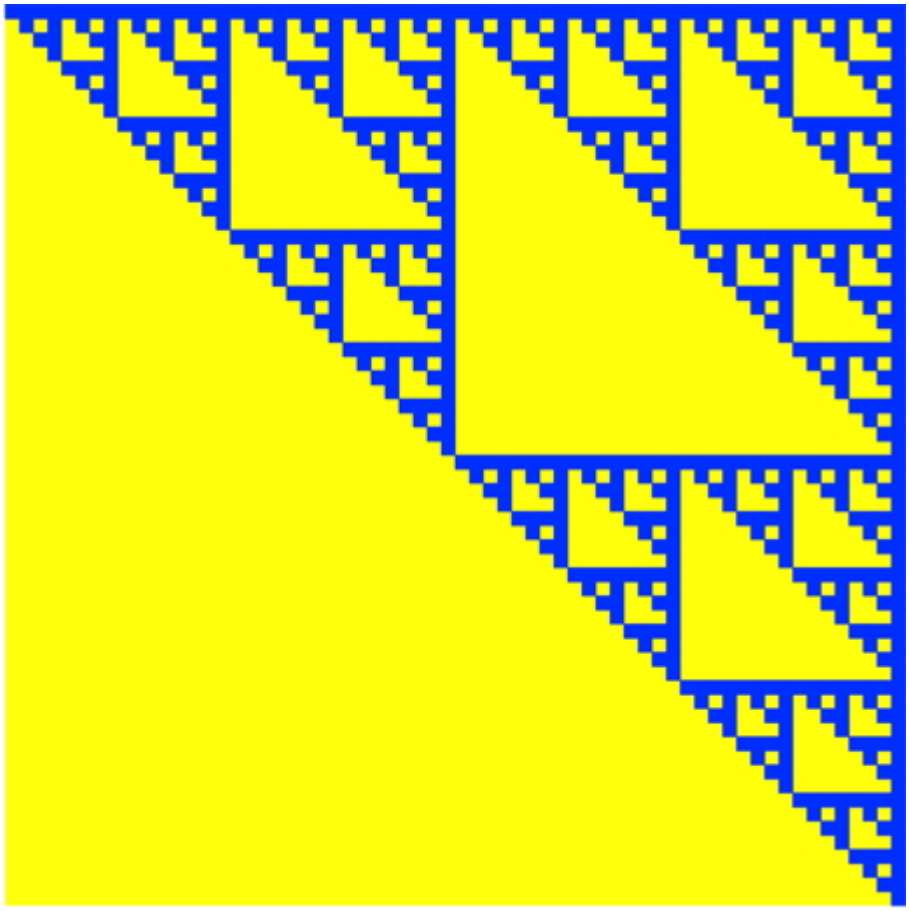
In this step, write the recursive case for which **n** is negative. Compute **x** raised to **-n** recursively, and return the reciprocal of that number.

Python | C++ | JS

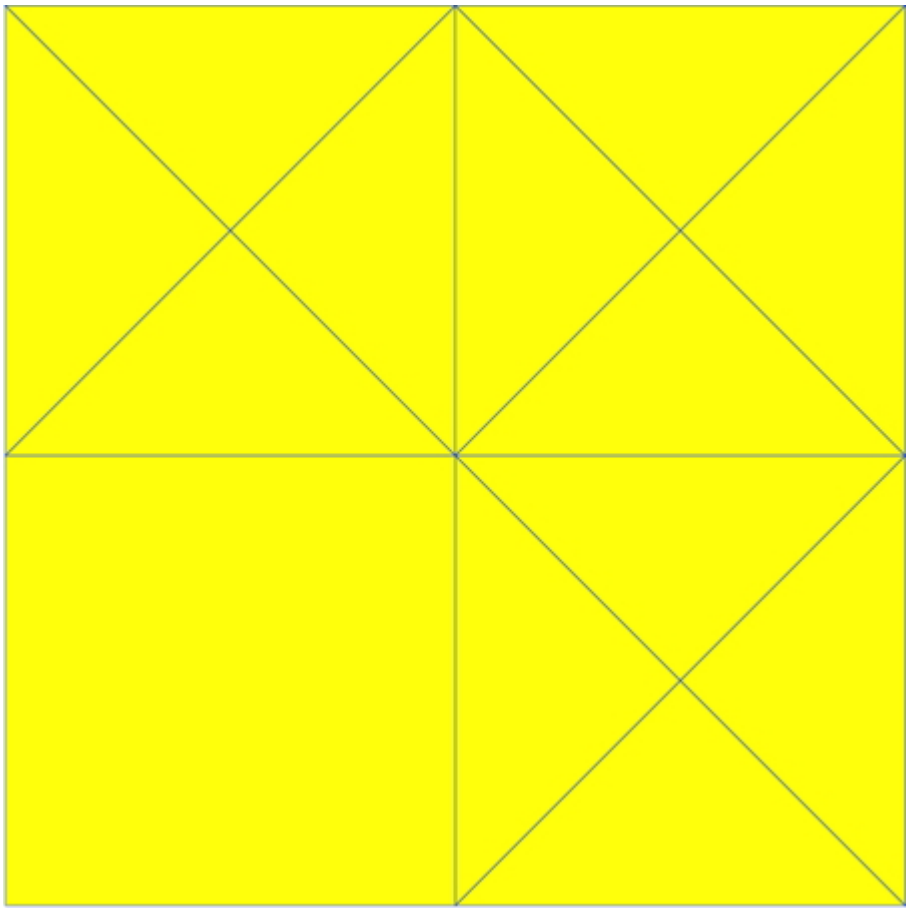
```
def power(x, n):
  # base case

  # recursive case: n is negative

  # recursive case: n is odd

  # recursive case: n is even
  return None
```

The Sierpinksi gasket

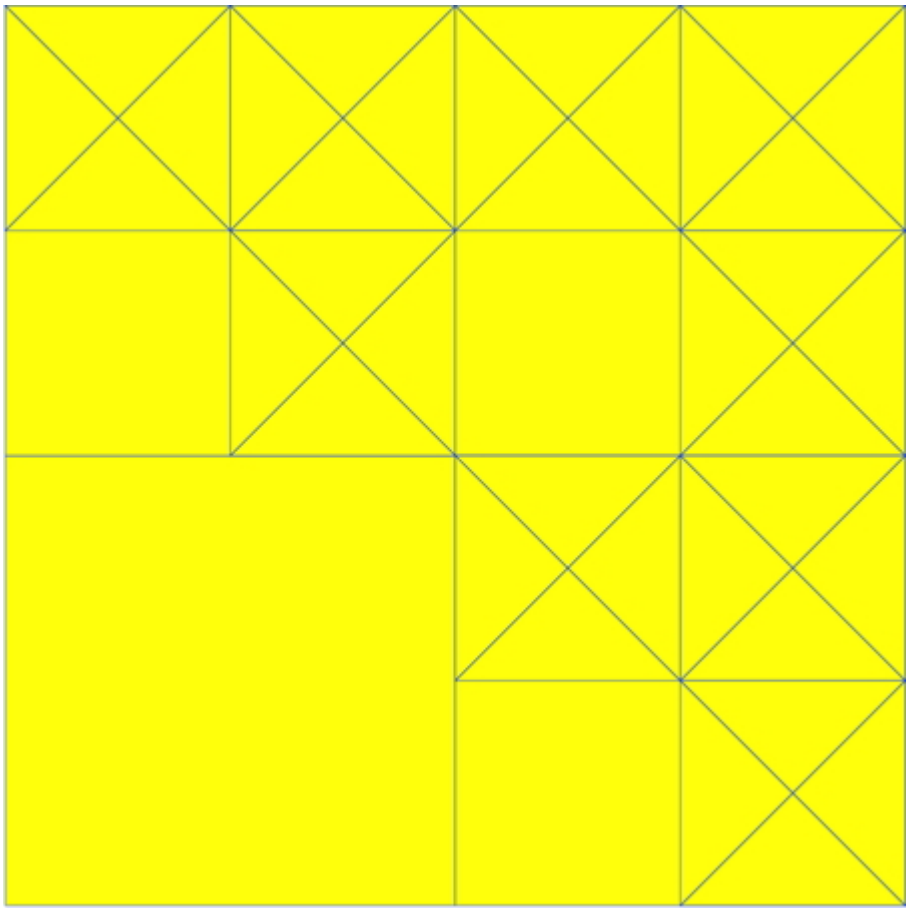
So far, the examples of recursion that we've seen require you to make one recursive call each time. But sometimes you need to make multiple recursive calls. Here's a good example, a mathematical construct that is a fractal known as a **Sierpinski gasket**:



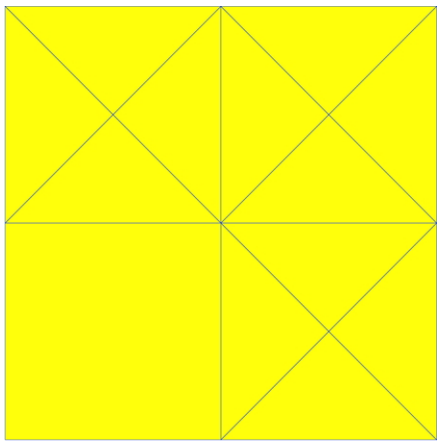
As you can see, it's a collection of little squares drawn in a particular pattern within a square region. Here's how to draw it. Start with the full square region, and divide it into four sections like so:



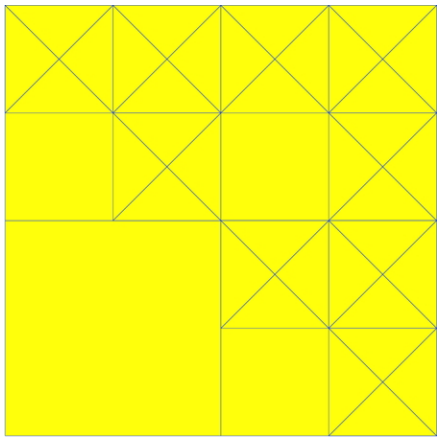
Take the three squares with an × through them—the top left, top right, and bottom right—and divide them into four sections in the same way:



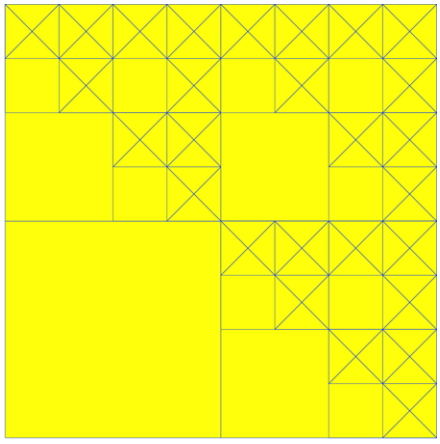
Keep going. Divide every square with an × into four sections, and place an × in the top left, top right, and bottom right squares, but never the bottom left. Once the squares get small enough, stop dividing. If you fill in each square with an × and forget about all the other squares, you get the Sierpinski gasket. Step through the animation to see it in action.



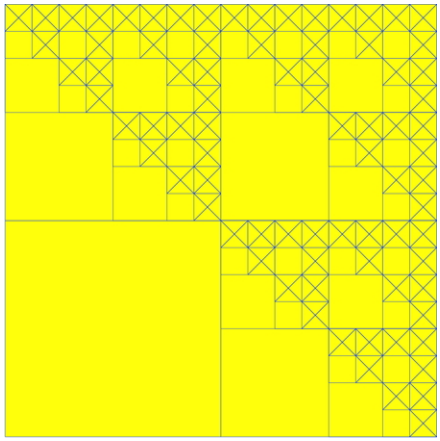
1 of 7



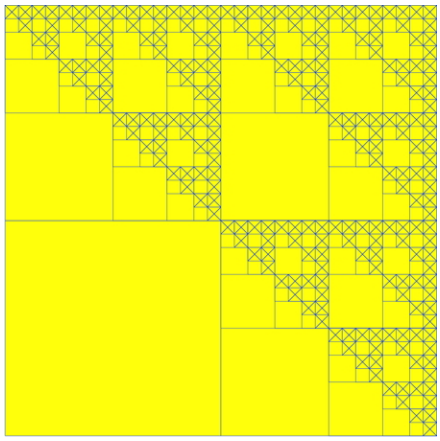
2 of 7



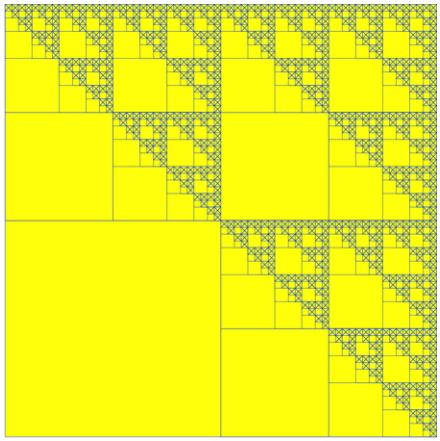
3 of 7



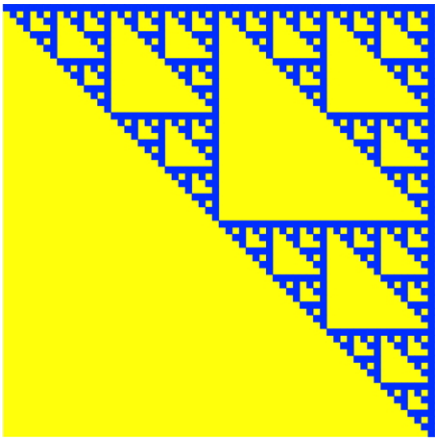
4 of 7



5 of 7



6 of 7



To summarize, here is how to draw a Sierpinski gasket in a square:

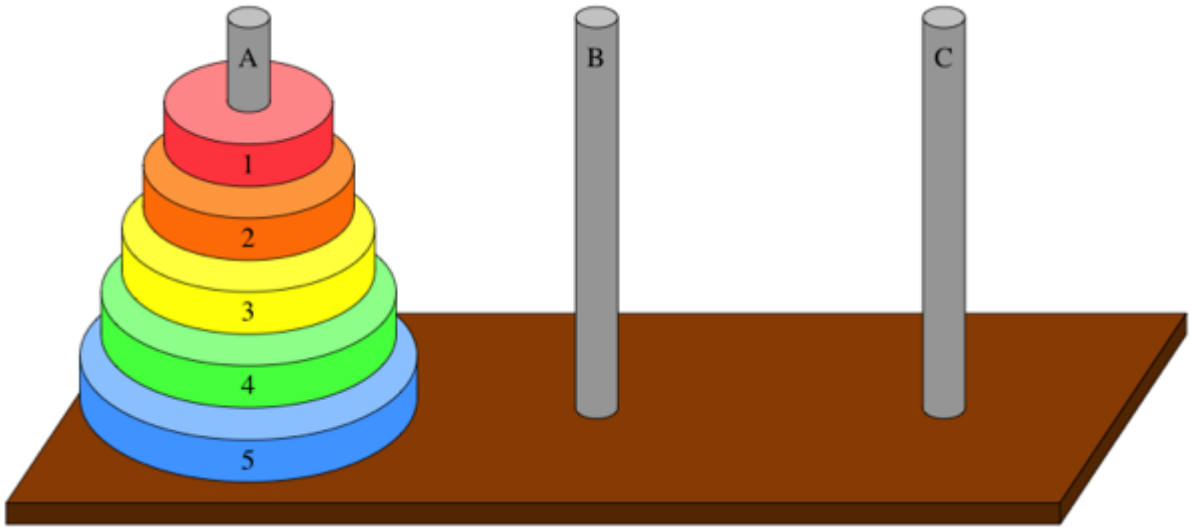
1. Determine how small the square is. If it's small enough to be a base case, then just fill in the square. You get to pick how small "small enough" is.
2. Otherwise, divide the square into upper left, upper right, lower right, and lower left squares. Recursively "solve" three subproblems:
    1. Draw a Sierpinski gasket in the upper left square.
    2. Draw a Sierpinski gasket in the upper right square.
    3. Draw a Sierpinski gasket in the lower right square.

You need to make not just one, but three recursive calls. That is why we consider drawing a Sierpinski gasket to exhibit multiple recursion.

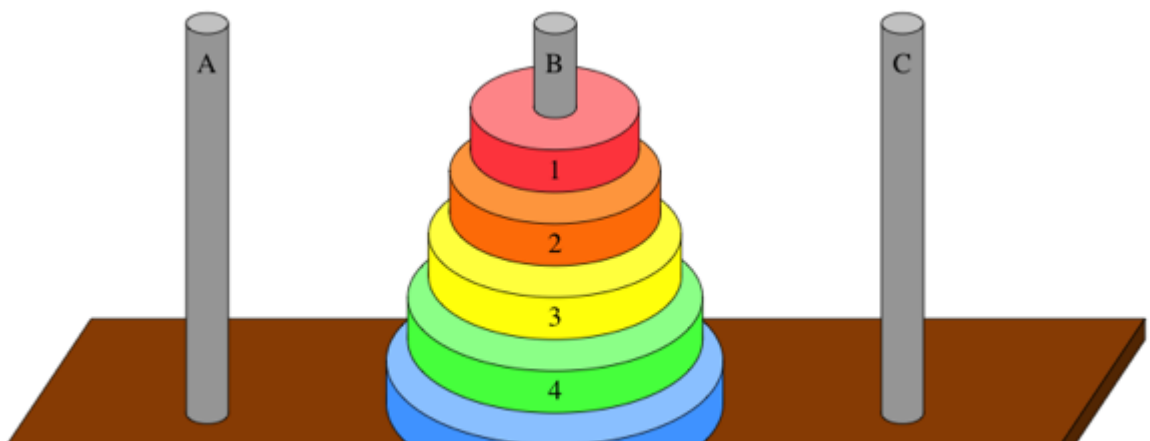
You can choose any three of the four squares in which you recursively draw Sierpinski gaskets. The result will just come out rotated by some multiple of 90 degrees from the drawing above. (If you recursively draw Sierpinski gaskets in any other number of the squares, you don't get an interesting result.)

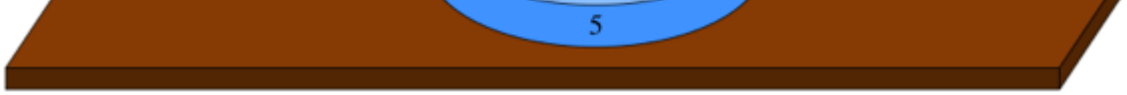
Towers of Hanoi

If you've gone through the tutorial on recursion, then you're ready to see another problem where recursing multiple times really helps. It's called the ***Towers of Hanoi***. You are given a set of three pegs and **n** disks, with each disk a different size. Let's name the pegs A, B, and C, and let's number the disks from 1, the smallest disk, to **n**, the largest disk. At the outset, all **n** disks are on peg A, in order of decreasing size from bottom to top, so that disk **n** is on the bottom and disk 1 is on the top. Here's what the Towers of Hanoi looks like for ***n*** = ***5*** disks:



The goal is to move all **n** disks from peg A to peg B:





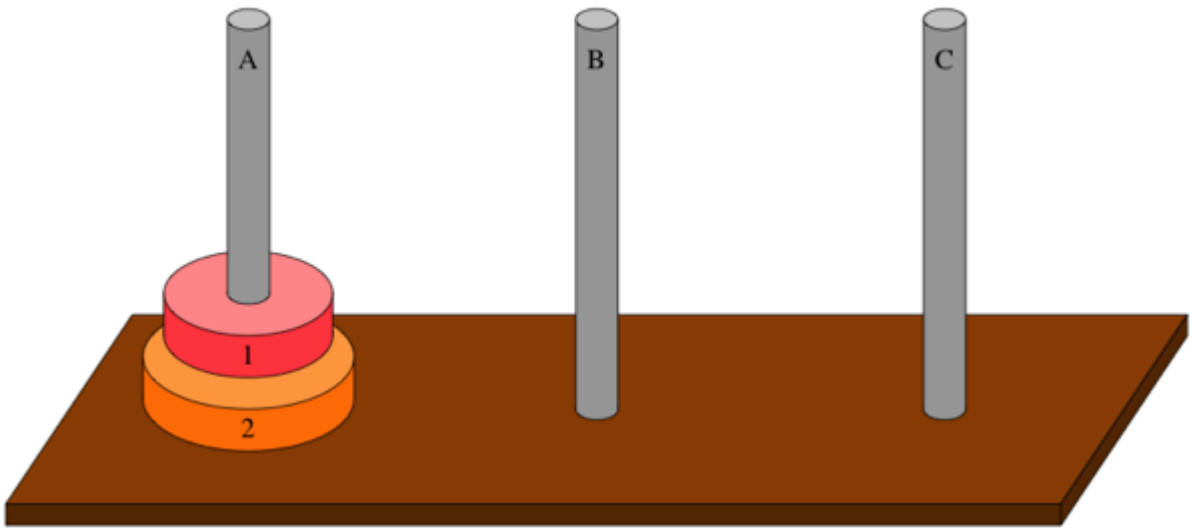
Sounds easy, right? It's not quite so simple, because you have to obey two rules:

1. You may move only one disk at a time.
2. No disk may ever rest atop a smaller disk. For example, if disk 3 is on a peg, then all disks below disk 3 must have numbers greater than 3.

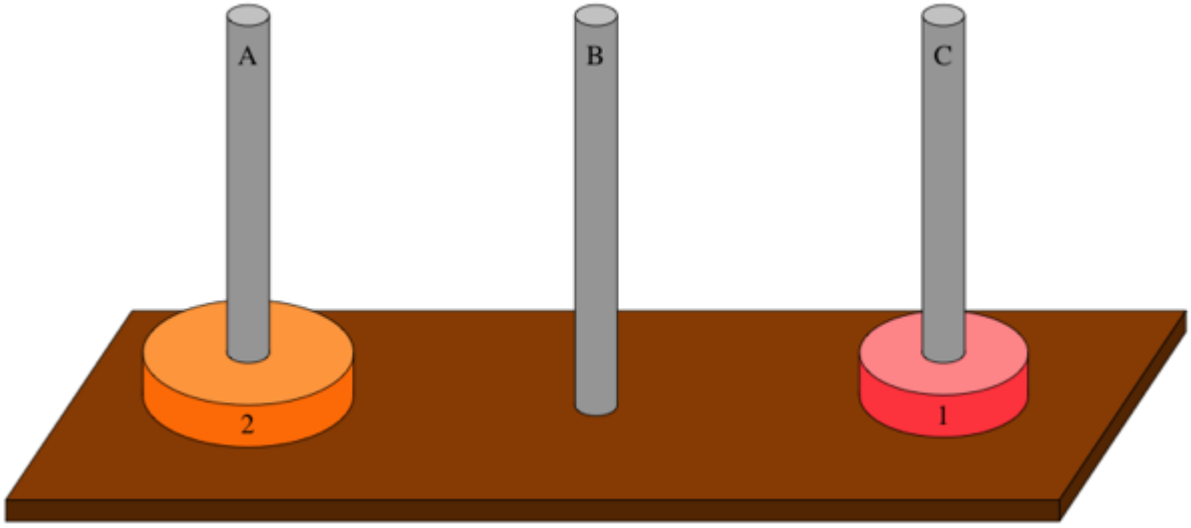
You might think that this problem is not terribly important. *Au contraire!* Legend has it that somewhere in Asia (Tibet, Vietnam, India—pick which legend on the Internet you like), monks are solving this problem with a set of 64 disks, and—so the story goes—the monks believe that once they finish moving all 64 disks from peg A to peg B according to the two rules, the world will end. If the monks are correct, should we be panicking in the streets?

First, let's see how to solve the problem recursively. We'll start with a really easy case: one disk, that is, **n=1**. The case of **n=1** will be our base case. You can always move disk 1 from peg A to peg B, because you know that any disks below it must be larger. And there's nothing special about pegs A and B. You can move disk 1 from peg B to peg C if you like, or from peg C to peg A, or from any peg to any peg. Solving the Towers of Hanoi problem with one disk is trivial, and it requires moving only the one disk one time.

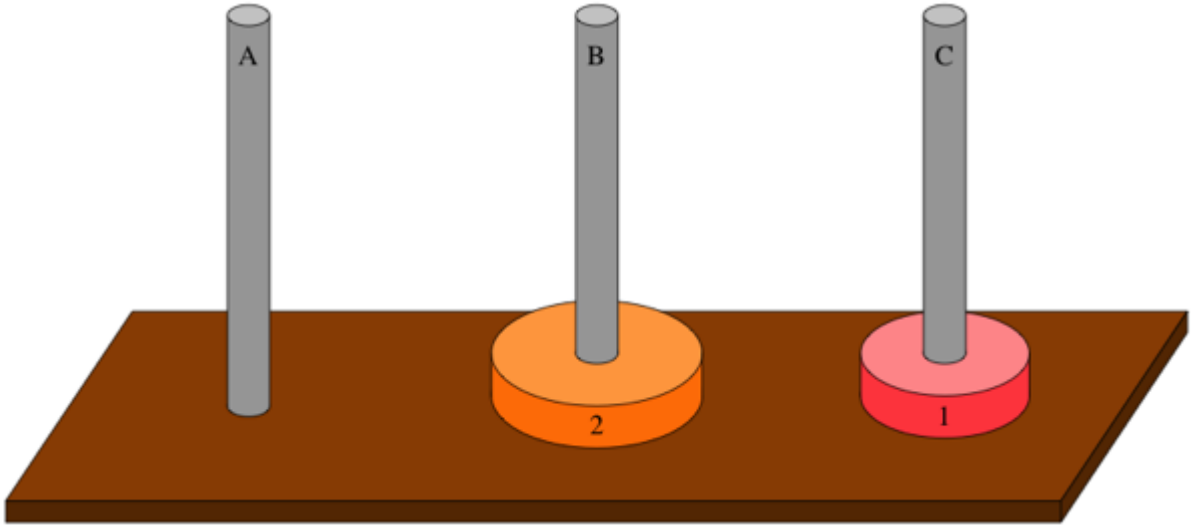
How about two disks? How do you solve the problem when **n** = **2**? You can do it in three steps. Here's what it looks like at the start:



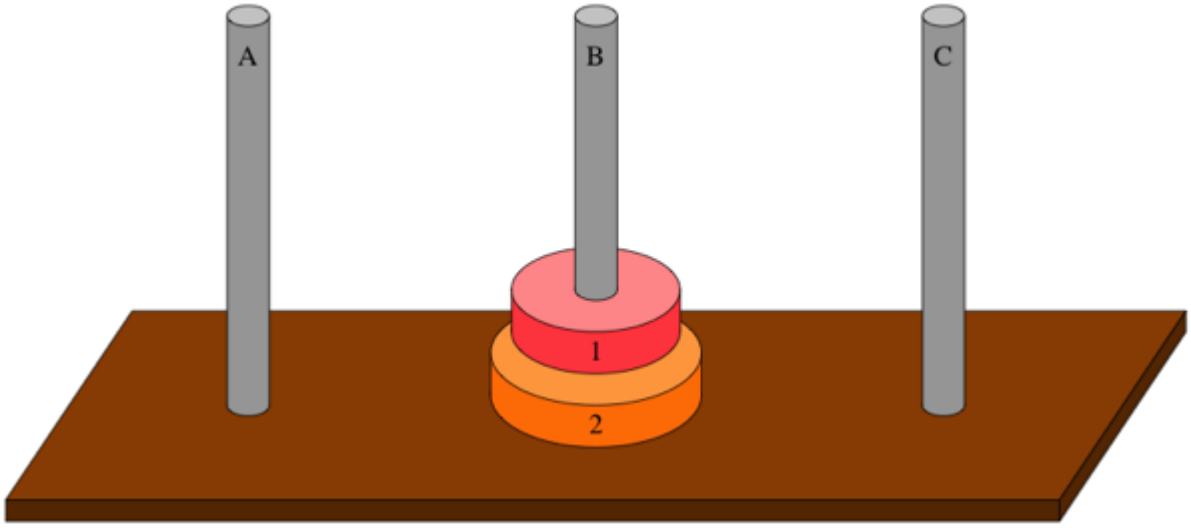
First, move disk 1 from peg A to peg C:



Notice that we're using peg C as a spare peg, a place to put disk 1 so that we can get at disk 2. Now that disk 2—the bottommost disk—is exposed, move it to peg B:

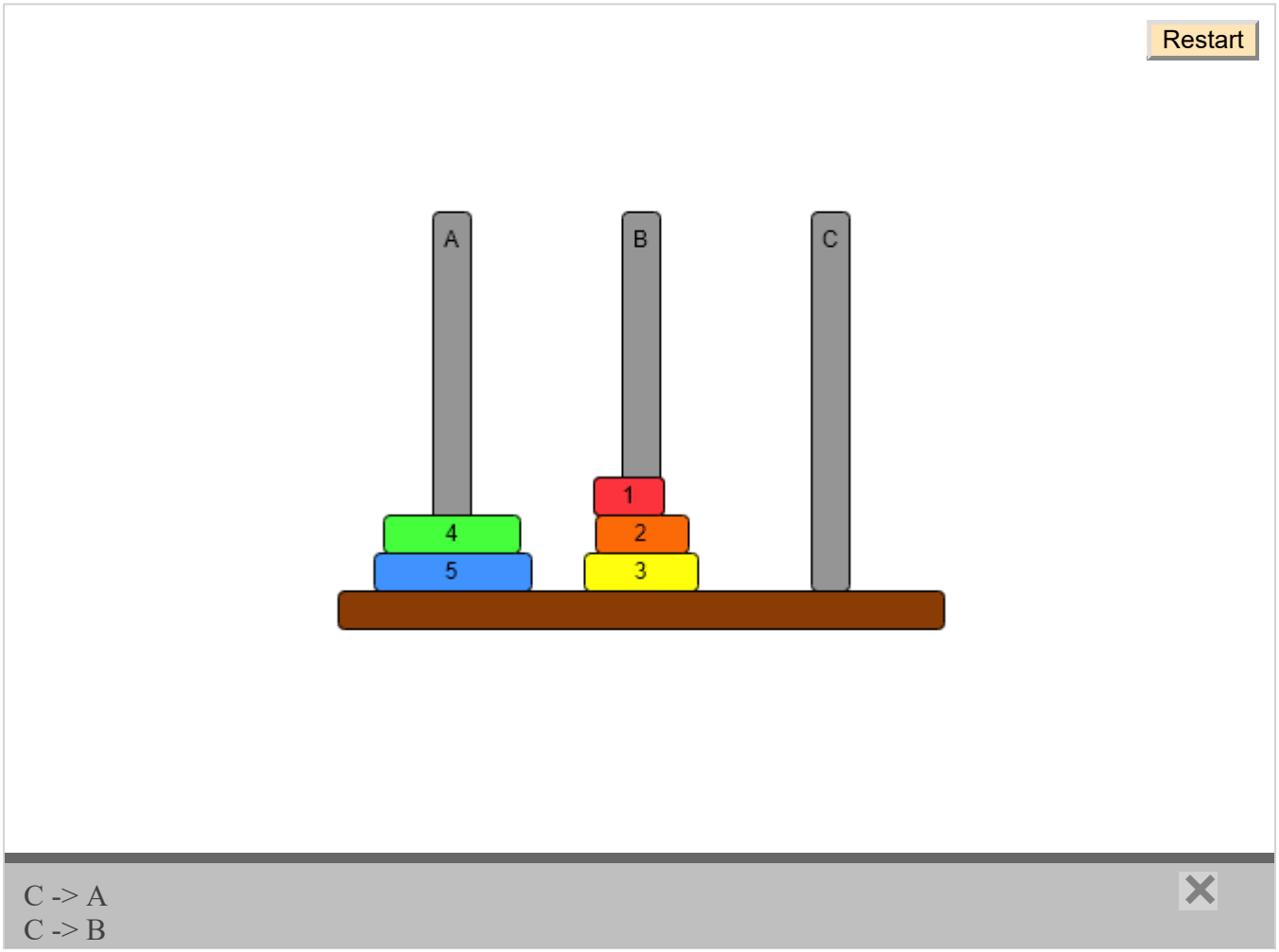


Finally, move disk 1 from peg C to peg B:

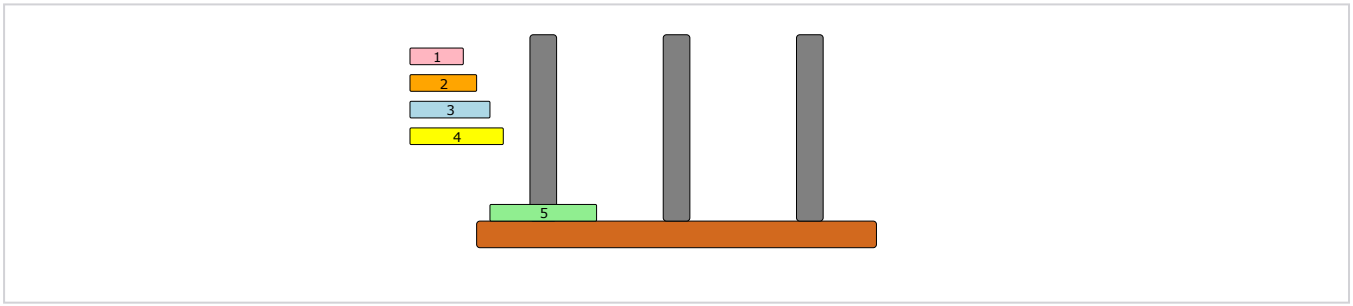


This solution takes three steps, and once again there's nothing special about moving the two disks from peg A to peg B. You can move them from peg B to peg C by using peg A as the spare peg: move disk 1 from peg B to peg A, then move disk 2 from peg B to peg C, and finish by moving disk 1 from peg A to peg C. Do you agree that you can move disks 1 and 2 from any peg to any peg in three steps? (Say "yes.")

Move three disks in Towers of Hanoi

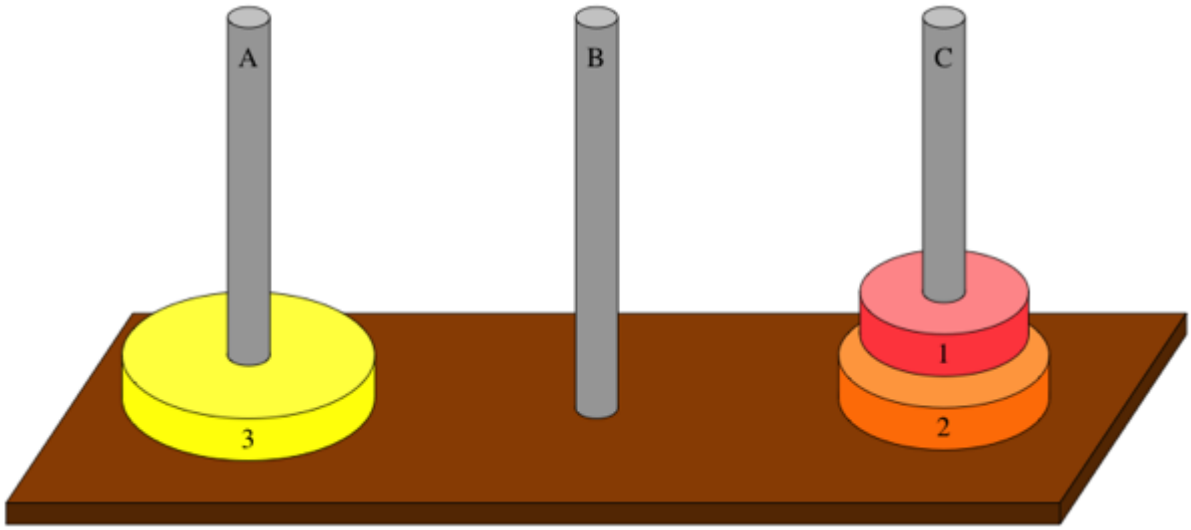


You can play around with the disks and try solving it manually.



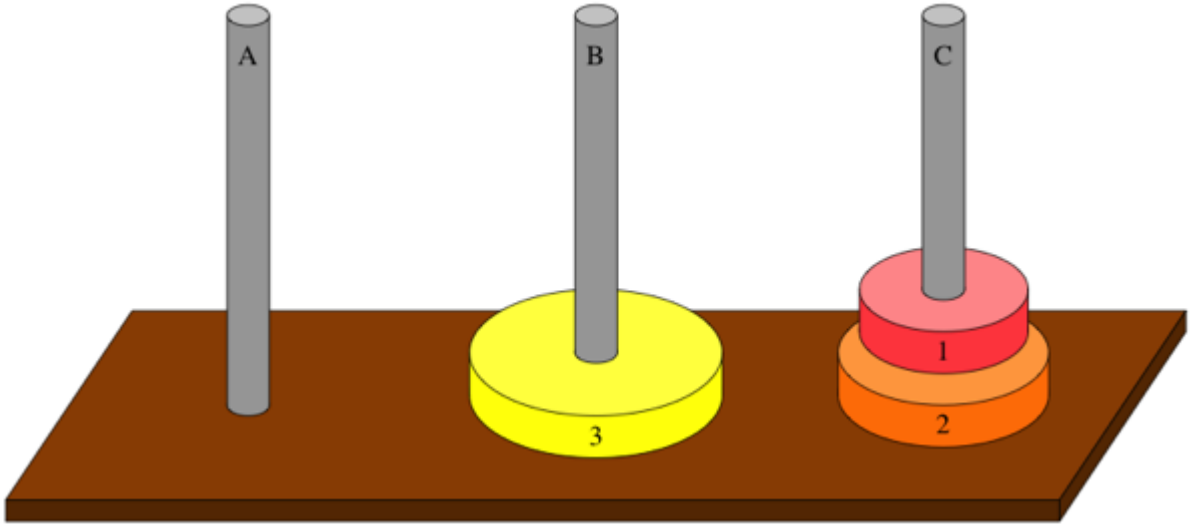
Towers of Hanoi, continued

When you solved the Towers of Hanoi for three disks, you needed to expose the bottom disk, disk 3, so that you could move it from peg A to peg B. In order to expose disk 3, you needed to move disks 1 and 2 from peg A to the spare peg, which is peg C:

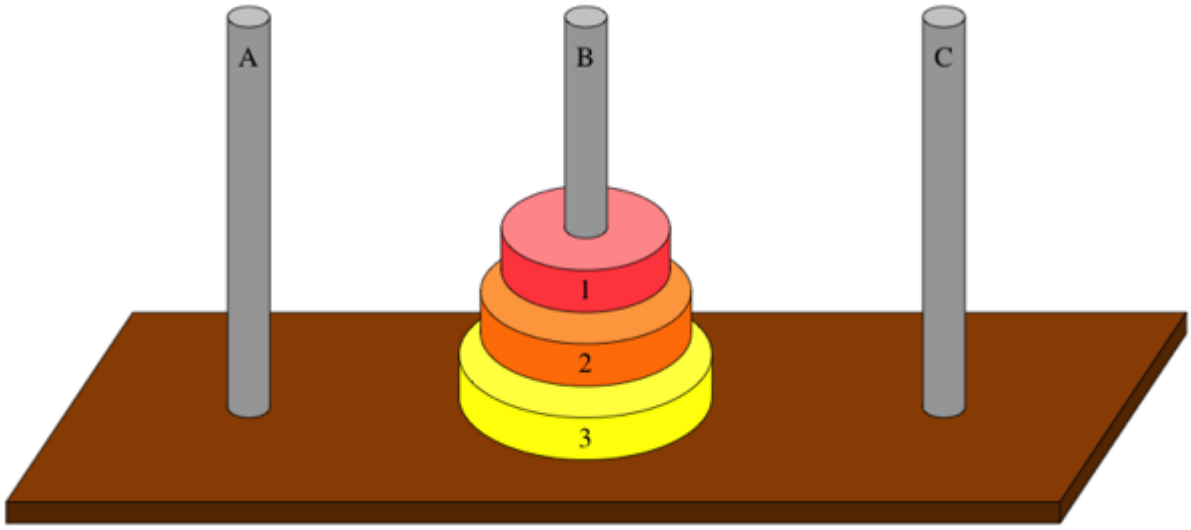


Wait a minute—it looks like two disks moved in one step, violating the first rule. But they did not move in one step. You agreed that you can move disks 1 and 2 from any peg to any peg, using three steps. The situation above represents what you have after three steps. (Move disk 1 from peg A to peg B; move disk 2 from peg A to peg C; move disk 1 from peg B to peg C.)

More to the point, by moving disks 1 and 2 from peg A to peg C, you have recursively solved a subproblem: move disks 1 through **n-1** (remember that **n = 3**) from peg A to peg C. Once you've solved this subproblem, you can move disk 3 from peg A to peg B:



Now, to finish up, you need to recursively solve the subproblem of moving disks 1 through **n-1** from peg C to peg B. Again, you agreed that you can do so in three steps. (Move disk 1 from peg C to peg A; move disk 2 from peg C to peg B; move disk 1 from peg A to peg B.) And you're done:



And—you knew this question is coming—is there anything special about which pegs you moved disks 1 through 3 from and to? You could have moved them from any peg to any peg. For example, from peg B to peg C:

- Recursively solve the subproblem of moving disks 1 and 2 from peg B to the spare peg, peg A. (Move disk 1 from peg B to peg C; move disk 2 from peg B to peg A; move disk 1 from peg C to peg A.)
- Now that disk 3 is exposed on peg B, move to it peg C.

- Recursively solve the subproblem of moving disks 1 and 2 from peg A to peg C. (Move disk 1 from peg A to peg B; move disk 2 from peg A to peg C; move disk 1 from peg B to peg C.)

No matter how you slice it, you can move disks 1 through 3 from any peg to any peg, moving disks seven times. Notice that you move disks three times for each of the two times that you recursively solve the subproblem of moving disks 1 and 2, plus one more move for disk 3.

How about when **n**=**4**? Because you can recursively solve the subproblem of moving disks 1 through 3 from any peg to any peg, you can solve the problem for **n** = **4**:

- Recursively solve the subproblem of moving disks 1 through 3 from peg A to peg C, moving disks seven times.
- Move disk 4 from peg A to peg B.
- Recursively solve the subproblem of moving disks 1 through 3 from peg C to peg B, moving disks seven times.

This solution moves disks 15 times (7 + 1 + 7) and, yes, you can move disks 1 through 4 from any peg to any peg.

At this point, you might have picked up two patterns. First, you can solve the Towers of Hanoi problem recursively. If **n** = **1**, just move disk 1. Otherwise, when **n** ≥ **2**, solve the problem in three steps:

- Recursively solve the subproblem of moving disks 1 through **n**−**1** from whichever peg they start on to the spare peg.
- Move disk **n** from the peg it starts on to the peg it's supposed to end up on.
- Recursively solve the subproblem of moving disks 1 through **n**−**1** from the spare peg to the peg they're supposed to end up on.

Second, solving a problem for **n** disks requires $2^n - 1$ moves. We've seen that it's true for **n** = **1** ($2^1 - 1 = 1$, and we needed one move), **n** = **2** ($2^2 - 1 = 3$, and three moves), **n** = **3** ($2^3 - 1 = 7$, and seven moves), and **n** = **4** ($2^4 - 1 = 15$, and 15 moves). If you can solve a problem for **n-1** disks in $2^{n-1} - 1$ moves, then you can solve a problem for **n** disks in $2^n - 1$ moves: you need $2^{n-1} - 1$ moves to recursively solve the first subproblem of moving disks 1 through **n-1**, one move to move disk **n**, and another $2^{n-1} - 1$ moves to recursively solve the second subproblem of moving disks 1 through **n-1**. If you add up the moves, you get $2^n - 1$.

Back to the monks. They're using **n** = **64** disks, and so they will need to move a disk $2^{64} - 1$ times. These monks are nimble and strong. They can move one disk every second, night and day. How long is $2^{64} - 1$ seconds? Using the rough estimate of 365.25 days per year (we're not accounting for skipping the leap year once every 400 years), that comes to 584,542,046,090.6263 years. That's 584+ billion years. The sun has only about another five to seven billion years left before it goes all supernova on us. So, yes, the world will end, but no matter how tenacious the monks may be, it will happen long before they can get all 64 disks onto peg B.

Wondering how else we can use this algorithm, besides predicting the end of the world? Wikipedia lists several interesting applications.

Challenge: Solve Hanoi recursively

Solve the base case

In this challenge, you will solve the towers of Hanoi problem for five disks, by writing a recursive function solveHanoi that will solve Hanoi for any positive number of disks.

A call to solveHanoi(numDisks,fromPeg,toPeg) should move numDisks disks from the peg fromPeg to the peg toPeg.

Start by implementing the base case of zero disks.

```
var solveHanoi = function(numDisks, fromPeg, toPeg) {
  // base case: no disks to move
  if (... == ...) {
      ...
  }
  solveHanoi(...);
  ...
};
```

Hint

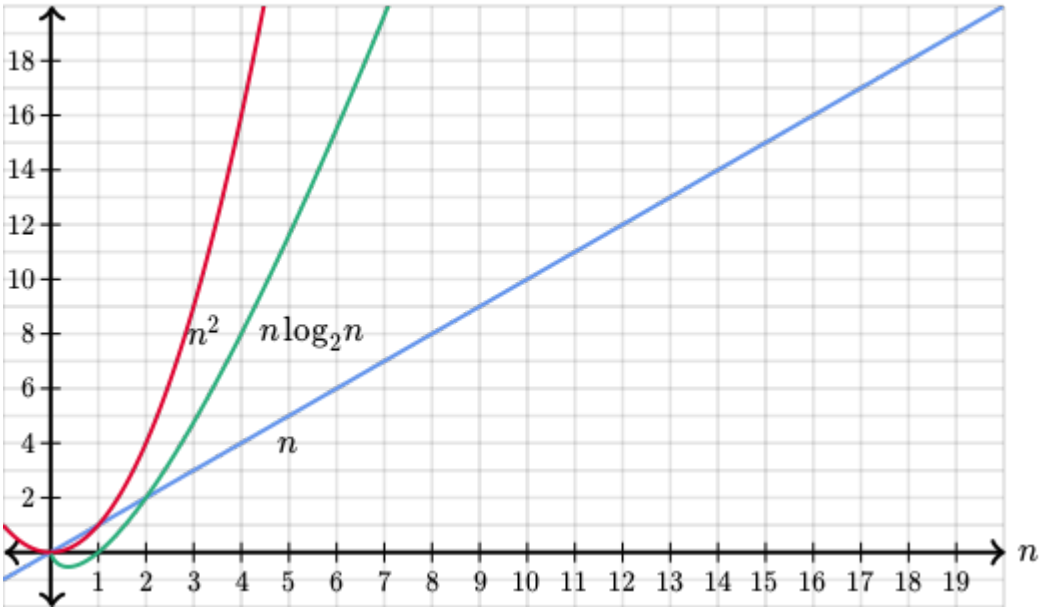
Java | Python | C++ | JS

```
class Solution {
  // You're given two helper functions.
  // 1) EdTestRunner.moveDisk(int fromPeg, int toPeg);
  //    It moves the top disk from the fromPeg to the toPeg.
  // 2) EdTestRunner.getSparePeg(int fromPeg, int toPeg);
  //    It returns the remaining peg.
  public static void solveHanoi(int disks, int fromPeg, int toPeg) {

  }
}
```

Divide and Conquer Algorithms

The two sorting algorithms we've seen so far, selection sort and insertion sort, have worst-case running times of $\Theta(n^2)$. When the size of the input array is large, these algorithms can take a long time to run. In this tutorial and the next one, we'll see two other sorting algorithms, merge sort and quicksort, whose running times ar better. In particular, merge sort runs in $\Theta(nlgn)$ time in all cases, and quicksort runs in $\Theta(nlgn)$ time in the best case and on average, though its worst-case running time is $\Theta(n^2)$. Here's a table of these four sorting algorithms and their running times:

Algorithm	Worst-case running time	Best-case running time	Average-case running time
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion sort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$



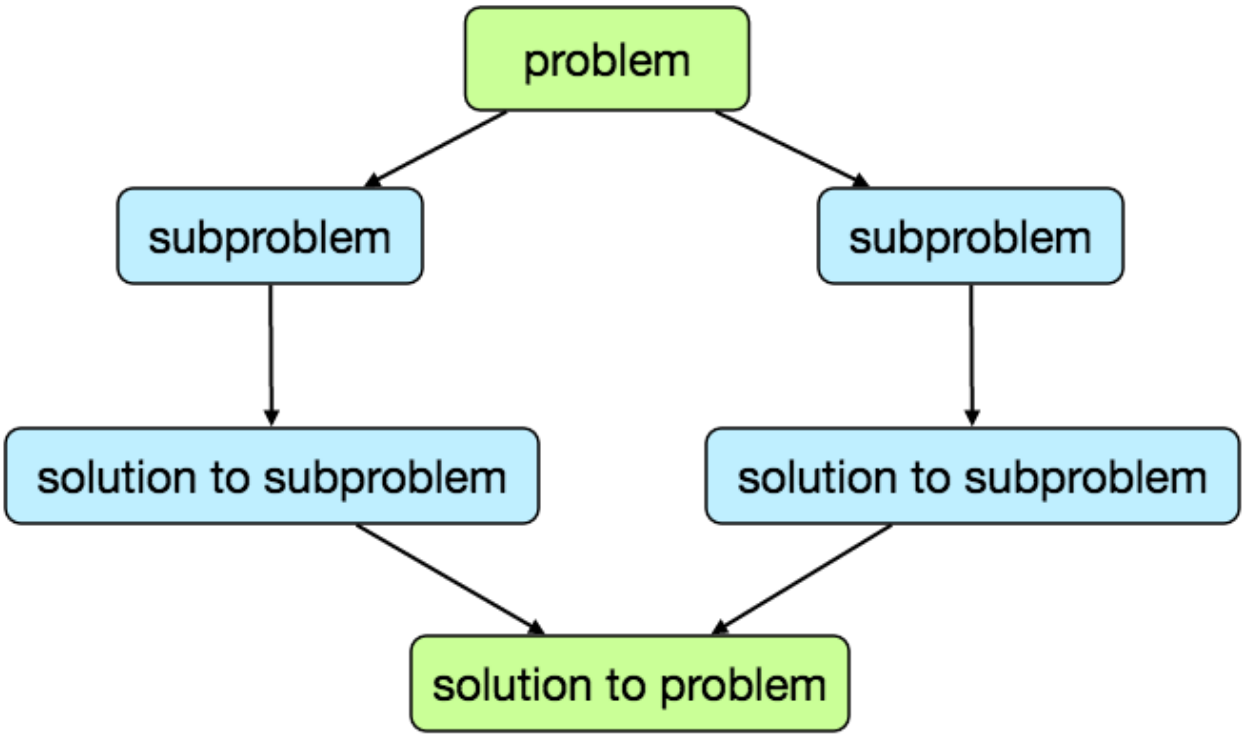
Divide-and-conquer

Both merge sort and quicksort employ a common algorithmic paradigm based on recursion. This paradigm, divide-and-conquer, breaks a problem into

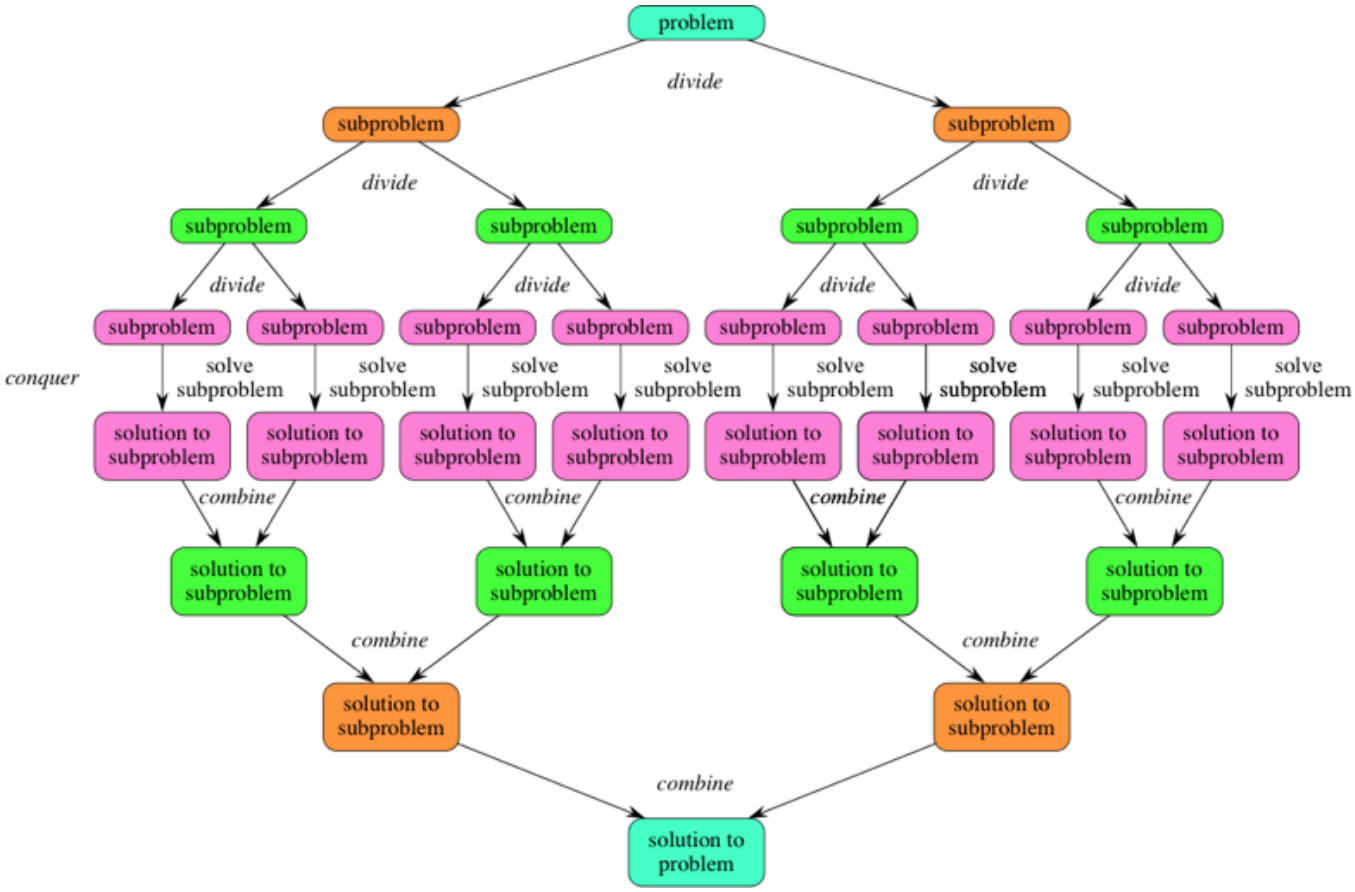
on recursion. This paradigm, divide-and-conquer, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem. Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems. You should think of a divide-and-conquer algorithm as having three parts:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

You can easily remember the steps of a divide-and-conquer algorithm as divide, conquer, combine. Here's how to view one step, assuming that each divide step creates two subproblems (though some divide-and-conquer algorithms create more than two):



If we expand out two more recursive steps, it looks like this:



Because divide-and-conquer creates at least two subproblems, a divide-and-conquer algorithm makes multiple recursive calls.

Overview of Merge Sort

Because we're using divide-and-conquer to sort, we need to decide what our subproblems are going to look like. The full problem is to sort an entire array. Let's say that a subproblem is to sort a subarray. In particular, we'll think of a subproblem as sorting the subarray starting at index pp and going through index rr. It will be convenient to have a notation for a subarray, so let's say that array[p..r] denotes this subarray of array. Note that this "two-dot" notation is not legal; we're using it just to describe the algorithm, rather than a particular implementation of the algorithm in code. In terms of our notation, for an array of **n** elements, we can say that the original problem is to sort **array[0..n-1]**.

Here's how merge sort uses divide-and-conquer:

1. Divide by finding the number **q** of the position midway between **p** and **r**. Do this step the same way we found the midpoint in binary search: add **p** and **r**, divide by 2, and round down.
2. Conquer by recursively sorting the subarrays in each of the two subproblems created by the divide step. That is, recursively sort the subarray **array[p..q]** and recursively sort the subarray **array[q+1..r]**.
3. Combine by merging the two sorted subarrays back into the single sorted subarray **array[p..r]**.

We need a base case. The base case is a subarray containing fewer than two elements, that is, when $p \geq r$, since a subarray with no elements or just one element is already sorted. So we'll divide-conquer-combine only when $p < r$.

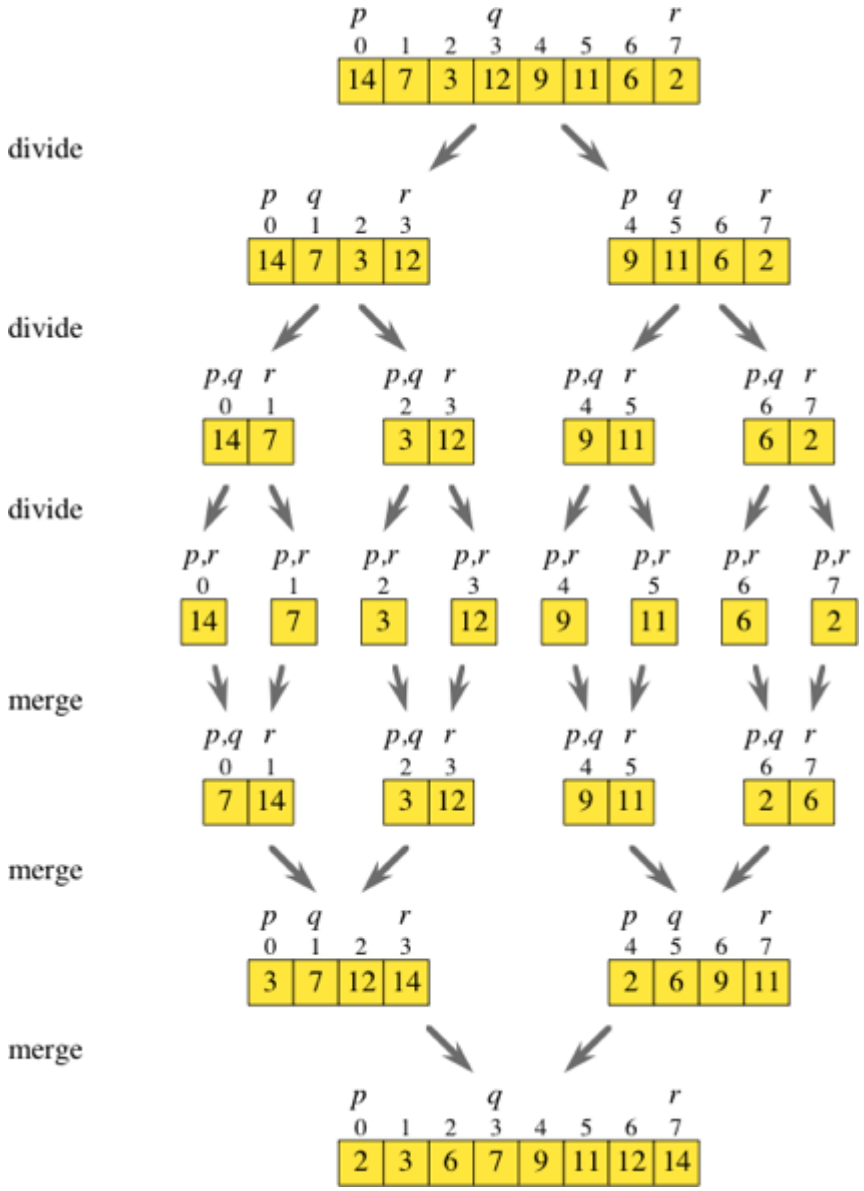
Let's see an example. Let's start with array holding [14, 7, 3, 12, 9, 11, 6, 2], so that the first subarray is actually the full array, **array[0..7] (p=0 and r=7)**. This subarray has at least two elements, and so it's not a base case.

- In the divide step, we compute **q=3**.
- The conquer step has us sort the two subarrays **array[0..3]**, which contains [14, 7, 3, 12], and **array[4..7]**, which contains [9, 11, 6, 2]. When we come back from the conquer step, each of the two subarrays is sorted: array[0..3] contains [3, 7, 12, 14] and array[4..7] contains [2, 6, 9, 11], so that the full array is [3, 7, 12, 14, 2, 6, 9, 11].
- Finally, the **combine** step merges the two sorted subarrays in the first half and the second half, producing the final sorted array [2, 3, 6, 7, 9, 11, 12, 14].

How did the subarray **array[0..3]** become sorted? The same way. It has more than two elements, and so it's not a base case. With **p=0**and **r=3**, compute **q=1**, recursively sort **array[0..1] ([14, 7])** and **array[2..3] ([3, 12])**, resulting in **array[0..3]** containing [7, 14, 3, 12], and merge the first half with the second half, producing [3, 7, 12, 14].

How did the subarray **array[0..1]** become sorted? With **p=0** and **r=1**, compute **q=0**, recursively sort **array[0..0]** ([14]) and **array[1..1]** ([7]), resulting in **array[0..1]** still containing [14, 7], and merge the first half with the second half, producing [7, 14].

The subarrays **array[0..0]** and **array[1..1]** are base cases, since each contains fewer than two elements. Here is how the entire merge sort algorithm unfolds:



Most of the steps in merge sort are simple. You can check for the base case easily. Finding the midpoint qq in the divide step is also really easy. You have to make two recursive calls in the conquer step. It's the combine step, where you have to merge two sorted subarrays, where the real work happens. For the moment, let's assume that we know how to merge two sorted subarrays efficiently and continue to focus on merge sort as a whole.

Challenge: Implement Merge Sort

The mergeSort function should recursively sort the subarray **array[p..r]** i.e. after calling mergeSort(array,p,r) the elements from index p to index r of array should be sorted in ascending order.

To remind you of the merge sort algorithm:

-If the subarray has size 0 or 1, then it's already sorted, and so nothing needs to be done.

-Otherwise, merge sort uses divide-and-conquer to sort the subarray.

Use **merge(array, p, q, r)** to merge sorted sub arrays **array[p..q]** and **array[q+1..r]**.

Java | Python | C++ | JS

```
class Solution {
  // Takes in an array that has two sorted subarrays,
  //  from [p..q] and [q+1..r], and merges the array
  static void merge(int array[], int p, int q, int r) {
      // This code has been purposefully obfuscated,
      //  as you'll write it yourself in next challenge.
      int i, j, k; int n1 = q - p + 1; int n2 =  r - q; int[] L = new int[n1]; int[] R = new
  }

  // Takes in an array and recursively merge sorts it
  public static void mergeSort(int[] array, int p, int r) {
        // Write this method


  };
}
```

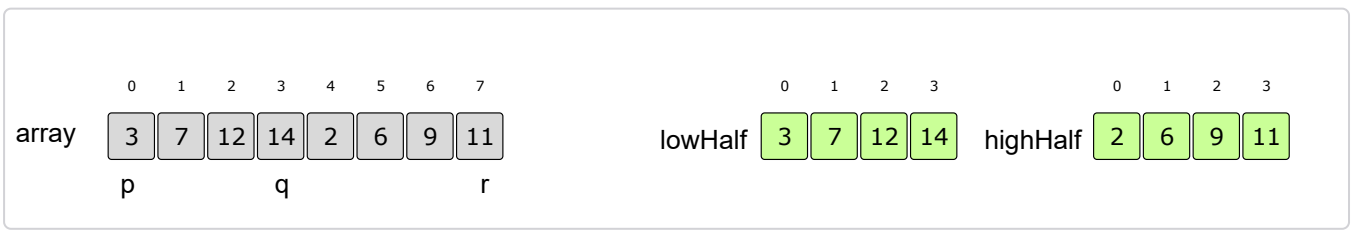
Linear-time Merging

The remaining piece of merge sort is the merge function, which merges two adjacent sorted subarrays, **array[p..q]** and **array[q+1..r]** into a single sorted subarray in **array[p..r]**. We'll see how to construct this function so that it's as efficient as possible. Let's say that the two subarrays have a total of **n** elements. We have to examine each of the elements in order to merge them together, and so the best we can hope for would be a merging time of **Θ(n)**. Indeed, we'll see how to merge a total of **n** elements in **Θ(n)** time.

In order to merge the sorted subarrays **array[p..q]** and **array[q+1..r]** and have the result in **array[p..r]**, we first need to make temporary arrays and copy **array[p..q]** and **array[q+1..r]** into these temporary arrays. We can't write over the positions in **array[p..r]** until we have the elements originally in **array[p..q]** and **array[q+1..r]** safely copied.

The first order of business in the merge function, therefore, is to allocate two temporary arrays, **lowHalf** and **highHalf**, to copy all the elements in **array[p..q]** into **lowHalf**, and to copy all the elements in **array[q+1..r]** into **highHalf**. How big should **lowHalf** be? The subarray **array[p..q]** contains **q−p+1** elements. How about **highHalf**? The subarray **array[q+1..r]** contains **r−q** elements. (In JavaScript, we don't have to give the size of an array when we create it, but since we do have to do that in many other programming languages, we often consider it when describing an algorithm.)

In our example array [14, 7, 3, 12, 9, 11, 6, 2], here's what things look like after we've recursively sorted **array[0..3]** and **array[4..7]** (so that **p=0**, **q=3**, and **r=7**) and copied these subarrays into **lowHalf** and **highHalf**:

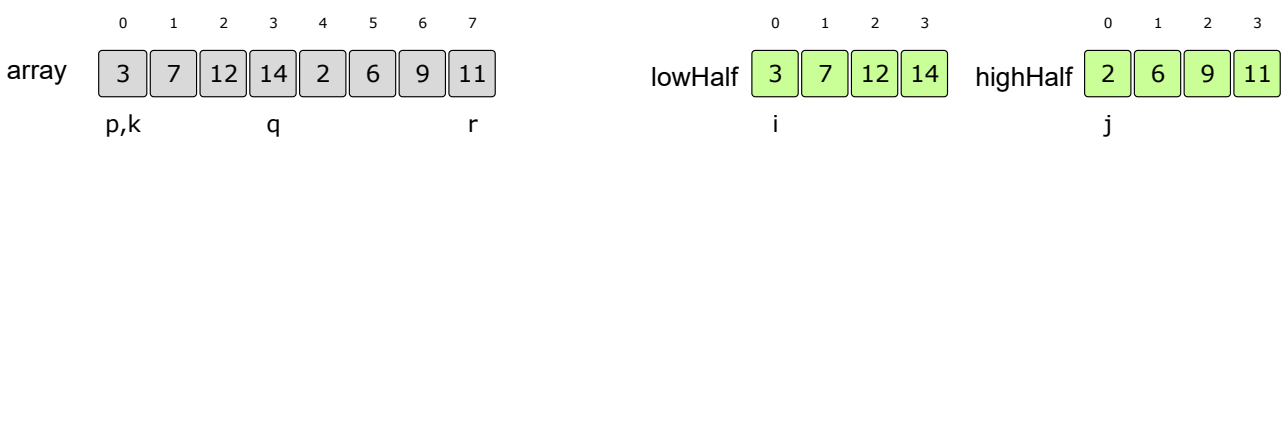


The numbers in array are grayed out to indicate that although these array positions contain values, the "real" values are now in **lowHalf** and **highHalf**. We may overwrite the grayed numbers at will.

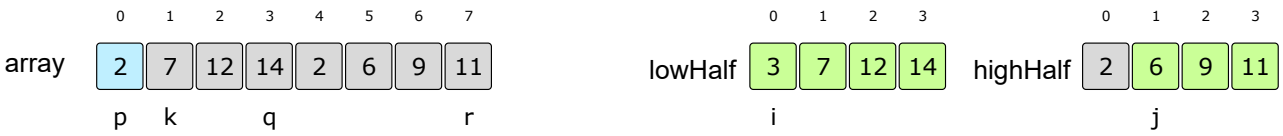
Next, we merge the two sorted subarrays, now in **lowHalf** and **highHalf**, back into **array[p..r]**. We should put the smallest value in either of the two subarrays into **array[p]**. Where might this smallest value reside? Because the subarrays are sorted, the smallest value must be in one of just two places: either **lowHalf[0]** or **highHalf[0]**. (It's possible that the same value is in both places, and then we can call either one the smallest value.) With just one comparison, we can determine whether to copy **lowHalf[0]** or **highHalf[0]** into **array[p]**. In our example, **highHalf[0]** was smaller. Let's also establish three variables to index into the arrays:

- i indexes the next element of **lowHalf** that we have not copied back into array. Initially, i is 0.
- j indexes the next element of **highHalf** that we have not copied back into array. Initially, j is 0.
- k indexes the next location in array that we copy into. Initially, k equals p.

After we copy from **lowHalf** or **highHalf** into array, we must increment (add 1 to) k so that we copy the next smallest element into the next position of array. We also have to increment i if we copied from lowHalf, or increment j if we copied from highHalf. So here are the arrays before and after the first element is copied back into array:

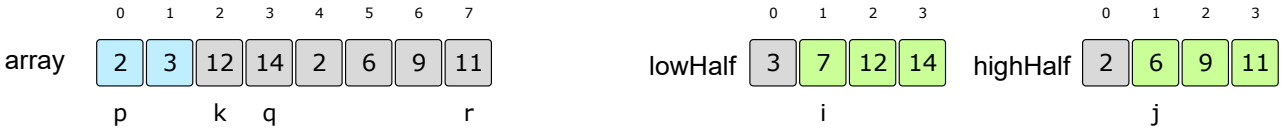


1 of 9



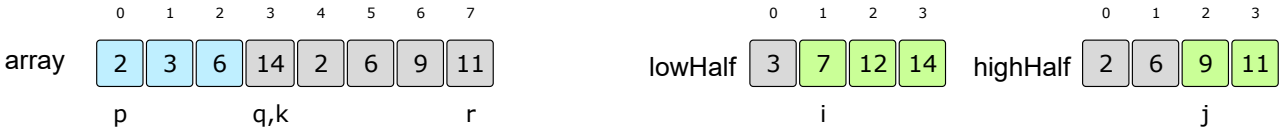
We've grayed out highHalf[0] to indicate that it no longer contains a value that we're going to consider.
The unmerged part of the highHalf array starts at index j, which is now 1.
The value in array[p] is no longer grayed out, because we copied a "real" value into it.

2 of 9



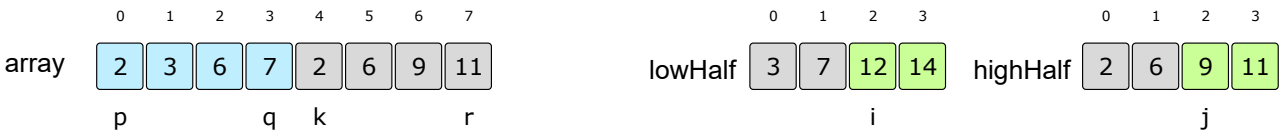
Where must the next value to copy back into array reside?
It's either first untaken element in lowHalf (lowHalf[0]) or the first untaken element in highHalf (highHalf[1])
With one comparison, we determine that lowHalf[0] is smaller, and so we copy it into array[k] and increment k and i

3 of 9

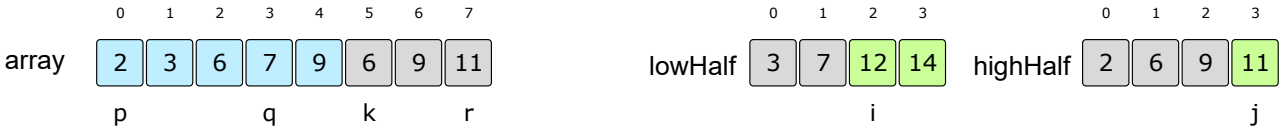


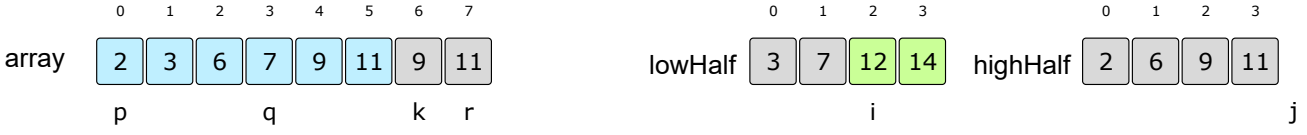
Next, we compare lowHalf[1] and highHalf[1], determining that we should copy highHalf[1] into array[k].
We then increment k and j

4 of 9

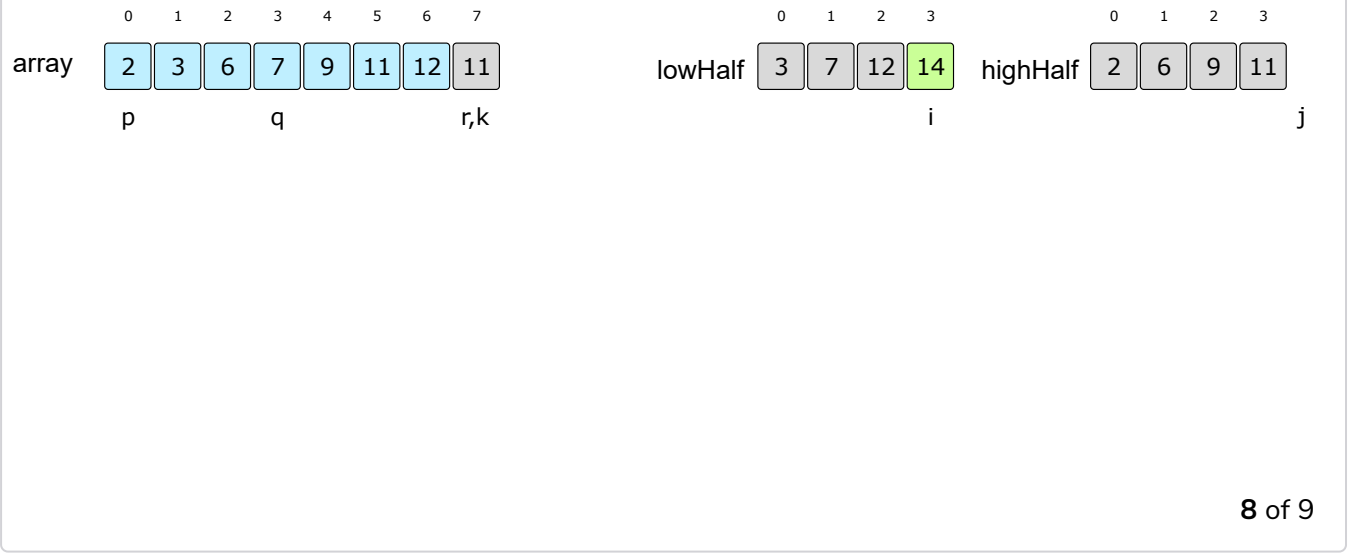


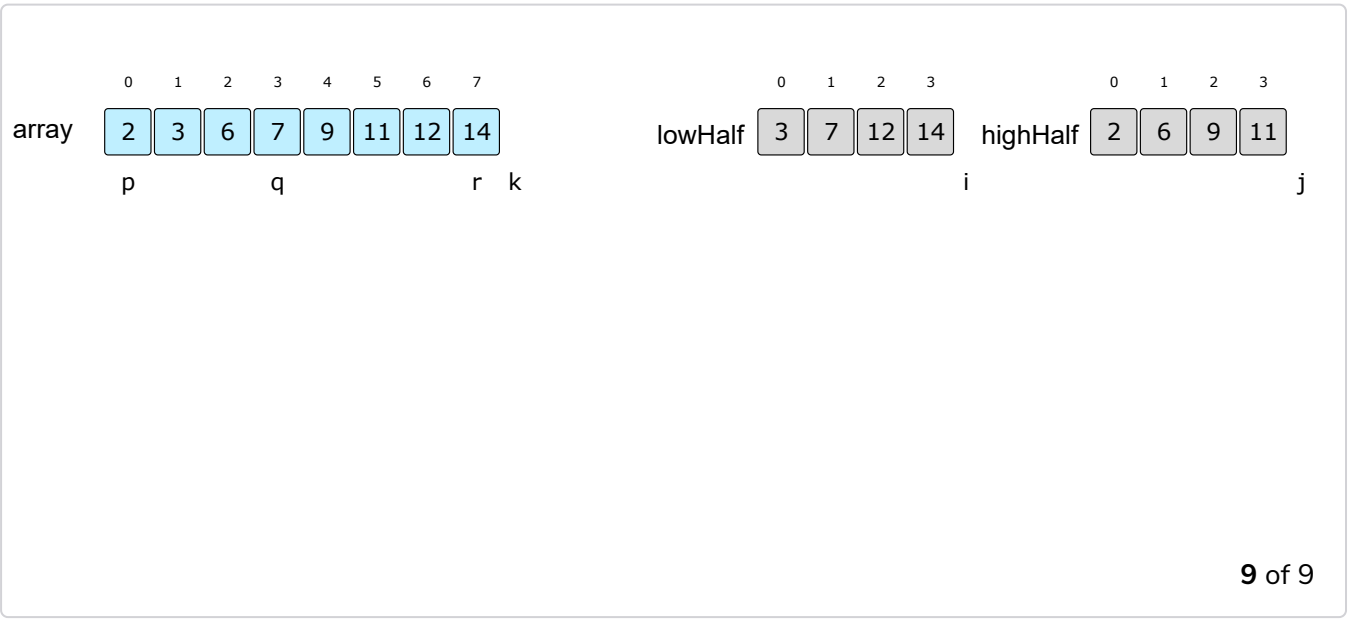
Keep going, always comparing lowHalf[i] and highHalf[j], copying the smaller of the two into array[k], and incrementing either i or j





Eventually, either all of lowHalf or all of highHalf is copied back into array. In this example, all of highHalf is copied back before the last few elements of lowHalf. We finish up by just copying the remaining untaken elements in either lowHalf or highHalf:





We claimed that merging **n** elements takes **Θ(n)** time, and therefore the running time of merging is linear in the subarray size. Let's see why this is true. We saw three parts to merging:

1. Copy each element in **array[p..r]** into either **lowHalf** or **highHalf**.
2. As long as some elements are untaken in both **lowHalf** and **highHalf**, compare the first two untaken elements and copy the smaller one back into array.
3. Once one of **lowHalf** and **highHalf** has had all its elements copied back into array, copy each remaining untaken element from the other temporary array back into array.

How many lines of code do we need to execute for each of these steps? It's a constant number per element. Each element is copied from array into either lowHalf or highHalf exactly one time in step 1. Each comparison in step 2 takes constant time, since it compares just two elements, and each element "wins" a comparison at most one time. Each element is copied back into array exactly one time in steps 2 and 3 combined. Since we execute each line of code a constant number of times per element and we assume that the subarray array[p..q] contains **n** elements, the running time for merging is indeed **Θ(n)**.

Challenge: Implement Merge

The merge function should merge the sorted subarrays in array[p..q] and array[q+1..r] into a single sorted subarray in array[p..r]. The function starts by allocating two temporary arrays, lowHalf and highHalf, and copying array[p..q] into lowHalf and array[q+1..r] into highHalf.

You should complete the function:

-Make it repeatedly compare the lowest untaken element in lowHalf with the lowest untaken element in highHalf and copy the lower of the two back into array, starting at array[p].

-Once one of lowHalf and highHalf has been fully copied back into array, the remaining elements in the other temporary array are copied back into array.

Note: use indexes i,j and k to access elements in lowHalf,highHalf,and array.

Java | Python | C++ | JS

```
class Solution {
  // Takes in an array that has two sorted subarrays,
  // from [p..q] and [q+1..r], and merges the array
  public static void merge(int[] array, int p, int q, int r) {
    // Repeatedly compare the lowest untaken element in
    // lowHalf with the lowest untaken element in highHalf
    // and copy the lower of the two back into array


    // Once one of lowHalf and highHalf has been fully copied
    // back into array, copy the remaining elements from the
    // other temporary array back into the array

  }
}
```

Analysis of Merge Sort

Given that the merge function runs in Θ(n) time when merging n elements, how do we get to showing that mergeSort runs in Θ(nlgn) time? We start by thinking about the three parts of divide-and-conquer and how to account for their running times. We assume that we're sorting a total of n elements in the entire array.

The divide step takes constant time, regardless of the subarray size. After all, the divide step just computes the midpoint qq of the indices p and r. Recall that in big-Θ notation, we indicate constant time by Θ(1).

The conquer step, where we recursively sort two subarrays of approximately n/2 elements each, takes some amount of time, but we'll account for that time when we consider the subproblems.

The combine step merges a total of n elements, taking Θ(n) time.

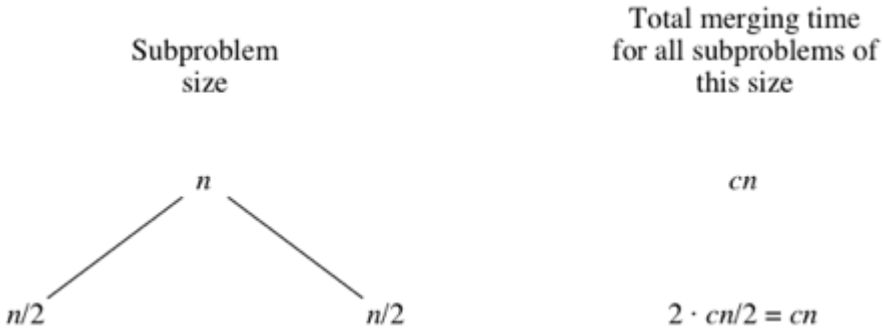
If we think about the divide and combine steps together, the Θ(1) running time for the divide step is a low-order term when compared with the Θ(n) running time of the combine step. So let's think of the divide and combine steps together as taking Θ(n) time. To make things more concrete, let's say that the divide and combine steps together take cn time for some constant c.

To keep things reasonably simple, let's assume that if n>1, then n is always even, so that when we need to think about n/2, it's an integer. (Accounting for the case in which n is odd doesn't change the result in terms of big-Θ notation.) So now we can think of the running time of mergeSort on an n-element subarray as being the sum of twice the running time of mergeSort on an (n/2)-element subarray (for the conquer step) plus cn (for the divide and combine steps—really for just the merging).

Now we have to figure out the running time of two recursive calls on n/2 elements. Each of these two recursive calls takes twice of the running time of

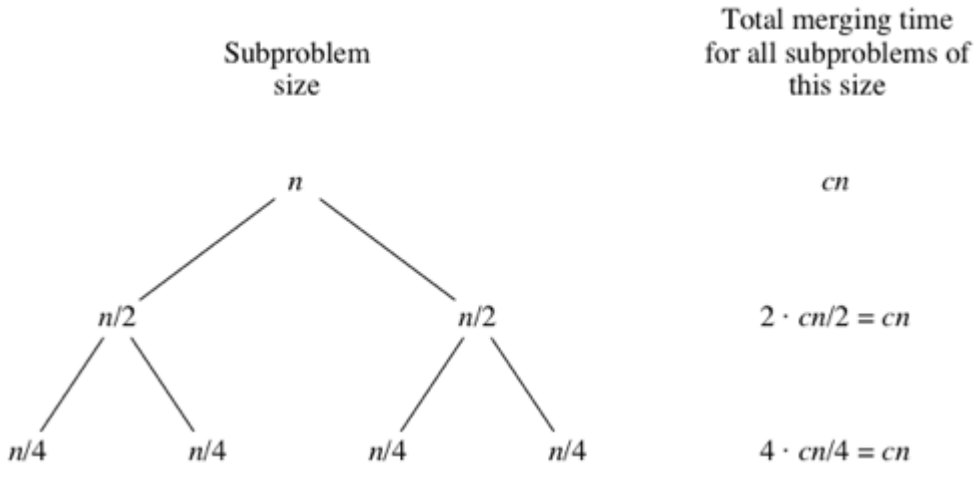
mergeSort on an (n/4)(n/4)-element subarray (because we have to halve n/2n/2) plus cn/2cn/2 to merge. We have two subproblems of size n/2, and each takes cn/2 time to merge, and so the total time we spend merging for subproblems of size n/2 is ***2·cn/2=cn***.

Let's draw out the merging times in a "tree":



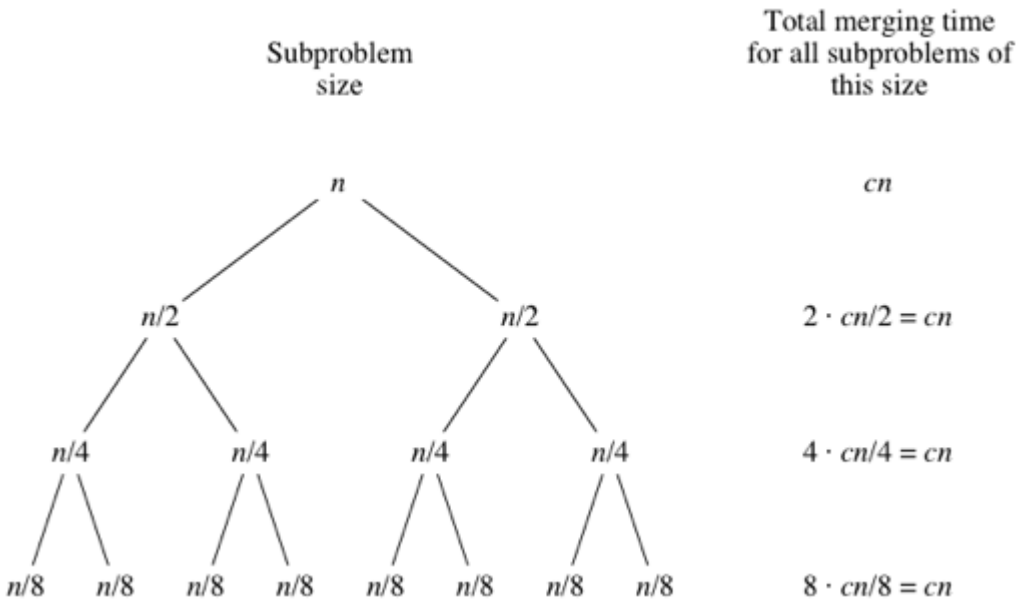
Computer scientists draw trees upside-down from how actual trees grow. A tree is a graph with no cycles (paths that start and end at the same place). Convention is to call the vertices in a tree its nodes. The root node is on top; here, the root is labeled with the n subarray size for the original n-element array. Below the root are two child nodes, each labeled n/2 to represent the subarray sizes for the two subproblems of size n/2.

Each of the subproblems of size n/2 recursively sorts two subarrays of size ((n/2)/2, or n/4. Because there are two subproblems of size n/2, there are four subproblems of size n/4. Each of these four subproblems merges a total of n/4 elements, and so the merging time for each of the four subproblems is cn/4. Summed up over the four subproblems, we see that the total merging time for all subproblems of size n/4 is ***4·cn/4=cn:***

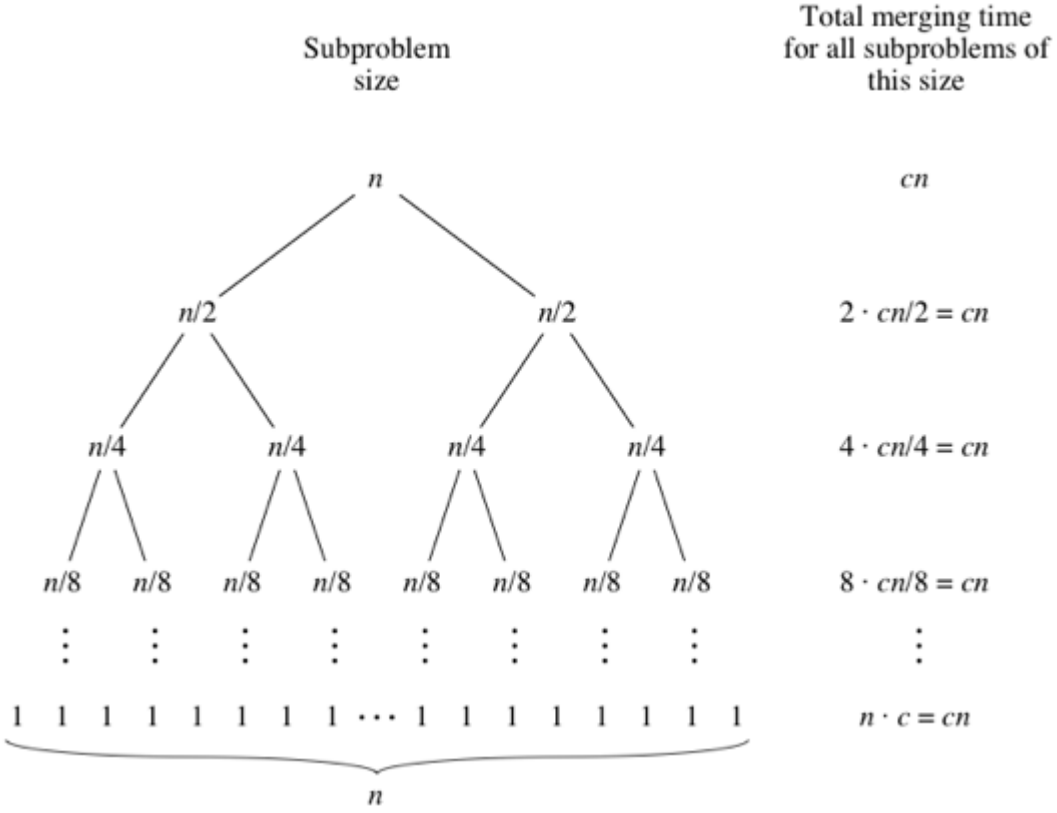


What do you think would happen for the subproblems of size n/8? There will be eight of them, and the merging time for each will be cn/8, for a total

merging time of **8·cn/8=cn**:



As the subproblems get smaller, the number of subproblems doubles at each "level" of the recursion, but the merging time halves. The doubling and halving cancel each other out, and so the total merging time is cn at each level of recursion. Eventually, we get down to subproblems of size 1: the base case. We have to spend Θ(1) time to sort subarrays of size 1, because we have to test whether p < r, and this test takes time. How many subarrays of size 1 are there? Since we started with n elements, there must be n of them. Since each base case takes Θ(1) time, let's say that altogether, the base cases take cn time:



Now we know how long merging takes for each subproblem size. The total

time for mergeSort is the sum of the merging times for all the levels. If there

are ll levels in the tree, then the total merging time is $l \cdot cn$. So what is l ? We start with subproblems of size n and repeatedly halve until we get down to subproblems of size 1. We saw this characteristic when we analyzed binary search, and the answer is $l = lgn+1$. For example, if n=8, then $lgn+1=4$, and sure enough, the tree has four levels: n = 8, 4, 2, 1. The total time for mergeSort, then, is $cn(lgn+1)$. When we use big-Θ notation to describe this running time, we can discard the low-order term (+1) and the constant coefficient (c), giving us a running time of **Θ(nlgn)**, as promised.

One other thing about merge sort is worth noting. During merging, it makes a copy of the entire array being sorted, with one half in lowHalf and the other half in highHalf. Because it copies more than a constant number of elements at some time, we say that merge sort does not work in place. By contrast, both selection sort and insertion sort do work in place, since they never make a copy of more than a constant number of array elements at any one time. Computer scientists like to consider whether an algorithm works in place, because there are some systems where space is at a premium, and thus in-place algorithms are preferred.

Overview of Quicksort

Like merge sort, quicksort uses divide-and-conquer, and so it's a recursive algorithm. The way that quicksort uses divide-and-conquer is a little different from how merge sort does. In merge sort, the divide step does hardly anything, and all the real work happens in the combine step. Quicksort is the opposite: all the real work happens in the divide step. In fact, the combine step in quicksort does absolutely nothing.

Quicksort has a couple of other differences from merge sort. Quicksort works in place. And its worst-case running time is as bad as selection sort's and insertion sort's: $\boldsymbol{\Theta(n^2)}$. But its average-case running time is as good as merge sort's: $\boldsymbol{\Theta(nlgn)}$. So why think about quicksort when merge sort is at least as good? That's because the constant factor hidden in the **big-Θ** notation for quicksort is quite good. In practice, quicksort outperforms merge sort, and it significantly outperforms selection sort and insertion sort.

Here is how quicksort uses divide-and-conquer. As with merge sort, think of sorting a subarray **array[p..r]**, where initially the subarray is **array[0..n-1]**.

1. Divide by choosing any element in the subarray **array[p..r]**. Call this element the ***pivot***. Rearrange the elements in **array[p..r]** so that all other elements in **array[p..r]** that are less than or equal to the pivot are to its left and all elements in **array[p..r]** are to the pivot's right. We call this procedure partitioning. At this point, it doesn't matter what order the elements to the left of the pivot are in relative to each other, and the same holds for the elements to the right of the pivot. We just care that each element is somewhere on the correct side of the pivot.
   As a matter of practice, we'll always choose the rightmost element in the subarray, **array[r]**, as the pivot. So, for example, if the subarray consists of [9, 7, 5, 11, 12, 2, 14, 3, 10, 6], then we choose 6 as the pivot. After partitioning, the subarray might look like [5, 2, 3, 6, 12, 7, 14, 9, 10, 11]. Let

partitioning, the subarray might look like [3, 2, 5, 6, 12, 7, 14, 9, 10, 11]. Let q be the index of where the pivot ends up.

2. Conquer by recursively sorting the subarrays **array[p..q-1]** (all elements to the left of the pivot, which must be less than or equal to the pivot) and **array[q+1..r]** (all elements to the right of the pivot, which must be greater than the pivot).

3. Combine by doing nothing. Once the conquer step recursively sorts, we are done. Why? All elements to the left of the pivot, in **array[p..q-1]**, are less than or equal to the pivot and are sorted, and all elements to the right of the pivot, in **array[q+1..r]**, are greater than the pivot and are sorted. The elements in **array[p..r]** can't help but be sorted!
   Think about our example. After recursively sorting the subarrays to the left and right of the pivot, the subarray to the left of the pivot is [2, 3, 5], and the subarray to the right of the pivot is [7, 9, 10, 11, 12, 14]. So the subarray has [2, 3, 5], followed by 6, followed by [7, 9, 10, 11, 12, 14]. The subarray is sorted.

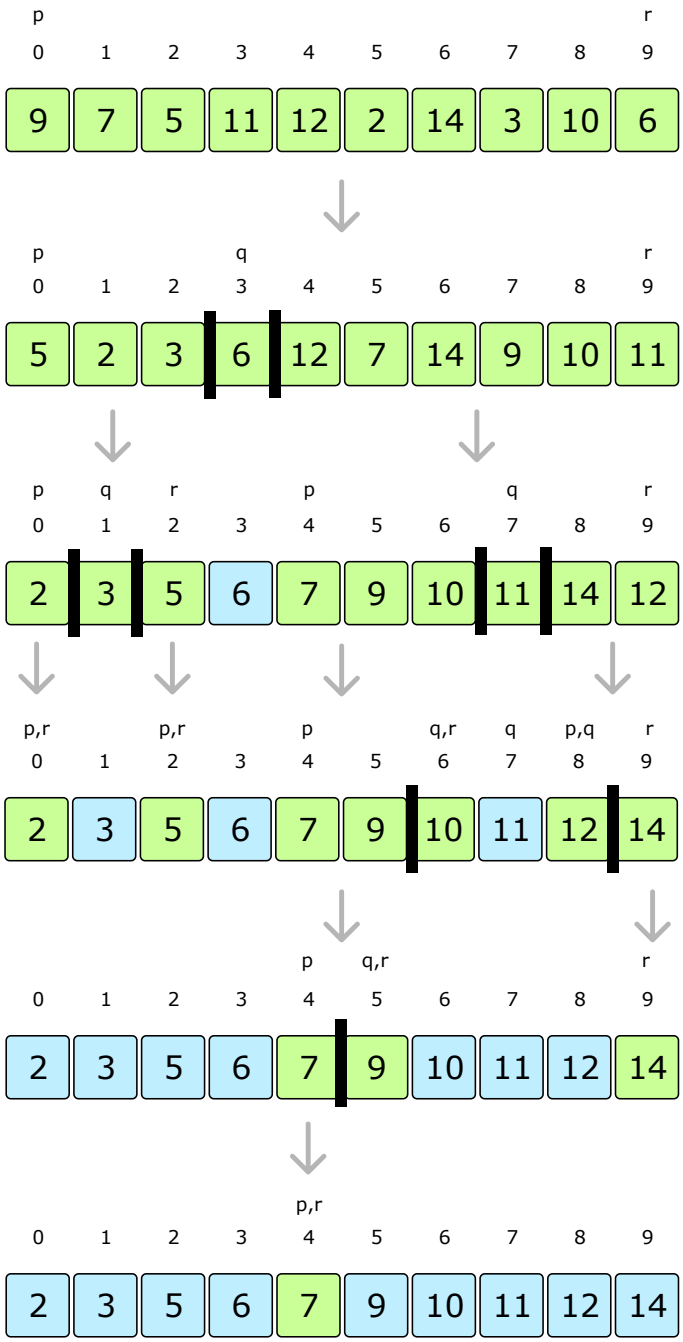
The base cases are subarrays of fewer than two elements, just as in merge sort. In merge sort, you never see a subarray with no elements, but you can in quicksort, if the other elements in the subarray are all less than the pivot or all greater than the pivot.

Let's go back to the conquer step and walk through the recursive sorting of the subarrays. After the first partition, we have have subarrays of [5, 2, 3] and [12, 7, 14, 9, 10, 11], with 6 as the pivot.

To sort the subarray [5, 2, 3], we choose 3 as the pivot. After partitioning, we have [2, 3, 5]. The subarray [2], to the left of the pivot, is a base case when we recurse, as is the subarray [5], to the right of the pivot.

To sort the subarray [12, 7, 14, 9, 10, 11], we choose 11 as the pivot, resulting in [7, 9, 10] to the left of the pivot and [14, 12] to the right. After these subarrays are sorted, we have [7, 9, 10], followed by 11, followed by [12, 14].

Here is how the entire quicksort algorithm unfolds. Array locations in blue have been pivots in previous recursive calls, and so the values in these locations will not be examined or moved again:



Challenge: Implement Quicksort

The quickSort function should recursively sort the subarray array[p..r].

- If the subarray has size 0 or 1, then it's already sorted, and so nothing needs to be done.

- Otherwise, quickSort uses divide-and-conquer to sort the subarray.

The divide step should partition the array, the conquer step should recursively quicksort the partitioned subarrays, and the combine step should do nothing.

Java | Python | C++ | JS

```
# This function partitions given array and returns
# the index of the pivot.
def partition(array, p, r):
  # Dont worry about this function. It's intentionally written with bad variable names
  # as you will implement it yourself in next challenge
  e=array
  t=p
  n=r
  def swap(e,t,n):
        r=e[t]
        e[t]=e[n]
        e[n]=r
  i=t
  s=t
  while s<n:
        if e[s]<=e[n]:
          swap(e,s,i)
          i = i + 1
        s = s + 1

  swap(e,n,i)
  return i

def quickSort(array, p, r):
  # Write method here

  return
```

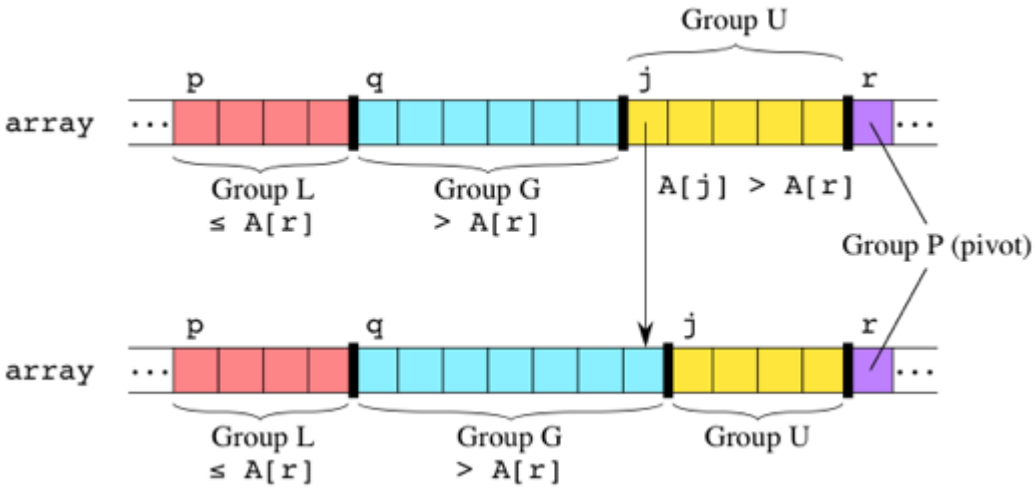
Linear-time Partitioning

The real work of quicksort happens during the divide step, which partitions subarray array[p..r] around a pivot drawn from the subarray. Although we can choose any element in the subarray as the pivot, it's easy to implement partitioning if we choose the rightmost element of the subarray, A[r], as the pivot.

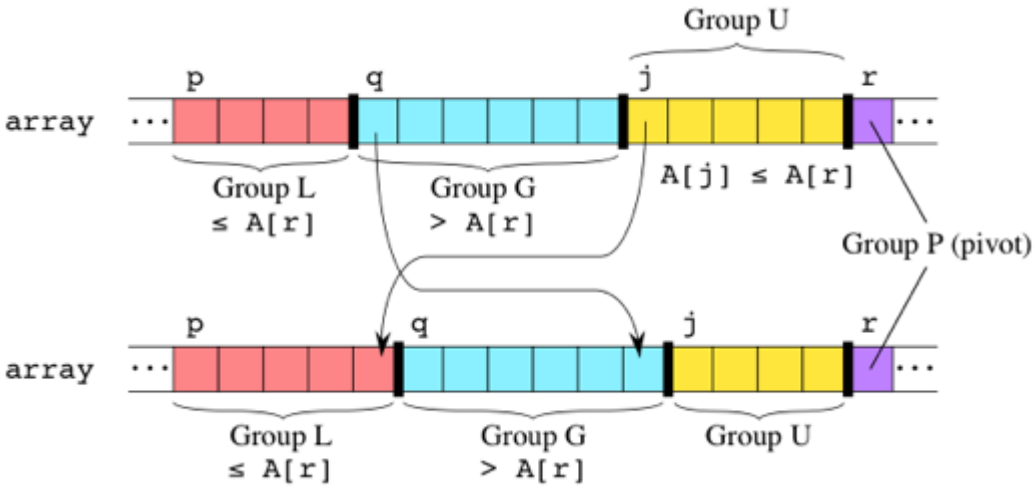
Having chosen a pivot, we partition the subarray by going through it, left to right, comparing each element with the pivot. We maintain two indices q and j into the subarray that divide it up into four groups. We use the variable name q because that index will eventually point at our pivot. We use j because it's a common counter variable name, and the variable will be discarded once we're done.

- The elements in array[p..q-1] are "group L," consisting of elements known to be less than or equal to the pivot.
- The elements in array[q..j-1] are "group G," consisting of elements known to be greater than the pivot.
- The elements in array[j..r-1] are "group U," consisting of elements whose relationship to the pivot is unknown, because they have not yet been compared.
- Finally, array[r] is "group P," the pivot.

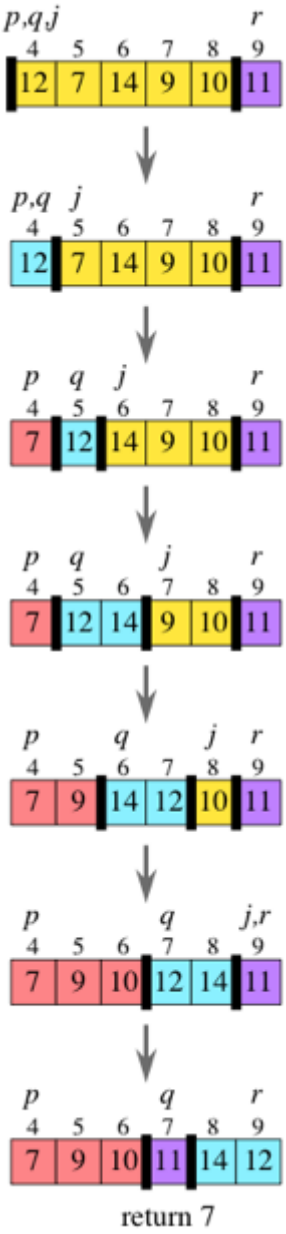
Initially, both q and j equal p. At each step, we compare A[j], the leftmost element in group U, with the pivot. If A[j] is greater than the pivot, then we just increment j, so that the line dividing groups G and U slides over one position to the right:



If instead A[j] is less than or equal to the pivot, then we swap A[j] with A[q] (the leftmost element in group G), increment j, and increment q, thereby sliding the line dividing groups L and G and the line dividing groups G and U over one position to the right:



Once we get to the pivot, group U is empty. We swap the pivot with the leftmost element in group G: swap A[r] with A[q]. This swap puts the pivot between groups L and G, and it does the right thing even if group L or group G is empty. (If group L is empty, then q never increased from its initial value of p, and so the pivot moves to the leftmost position in the subarray. If group G is empty, then q was incremented every time j was, and so once j reaches the index r of the pivot, q equals r, and the swap leaves the pivot in the rightmost position of the subarray.) The partition function that implements this idea also returns the index q where it ended up putting the pivot, so that the quicksort function, which calls it, knows where the partitions are. Here are the steps in partitioning the subarray [12, 7, 14, 9, 10, 11] in array[4..9]:



Partitioning a subarray with n elements takes **Θ(n)** time. Each element A[j] is compared once with the pivot. A[j] may or may not be swapped with A[q], q may or may not be incremented, and j is always incremented. The total number of lines executed per element of the subarray is a constant. Since the subarray has **n** elements, the time to partition is **Θ(n)**: linear-time partitioning.

Challenge: Implement Partition

The partition function should partition the subarray array[p..r] so that all elements in array[p..q-1] are less than or equal to array[q] (the pivot) and all elements in array[q+1..r] are greater than array[q], and it returns the index q of where the pivot ends up.

Use the provided swap() function for swapping.

Java | Python | C++ | JS

```
class Solution {
  // Swaps two items in an array, changing the original array
  static void swap(int[] array, int firstIndex, int secondIndex) {
      int temp = array[firstIndex];
      array[firstIndex] = array[secondIndex];
      array[secondIndex] = temp;
  };

  public static void partition(int[] array, int p, int r) {
      // Compare array[j] with array[r], for j = p, p+1,...r-1
      // maintaining that:
      //  array[p..q-1] are values known to be <= to array[r]
      //  array[q..j-1] are values known to be > array[r]
      //  array[j..r-1] haven't been compared with array[r]
      // If array[j] > array[r], just increment j.
      // If array[j] <= array[r], swap array[j] with array[q],
      //   increment q, and increment j.
      // Once all elements in array[p..r-1]
      //  have been compared with array[r],
      //  swap array[r] with array[q], and return q.

  }
}
```

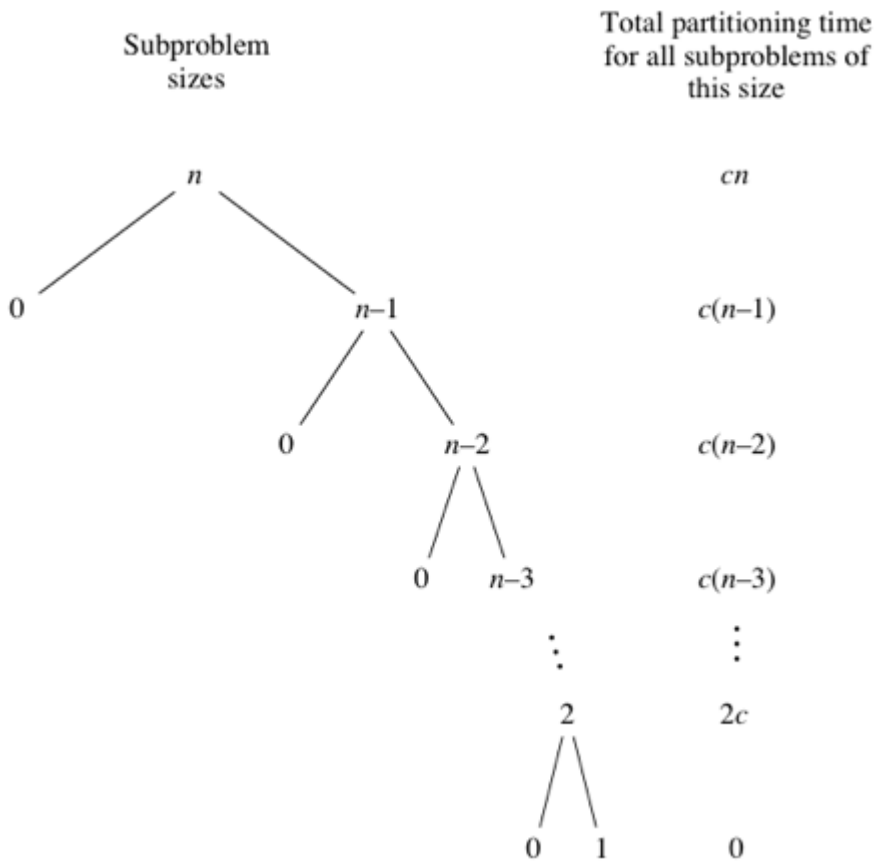
Analysis of Quicksort

How is it that quicksort's worst-case and average-case running times differ? Let's start by looking at the worst-case running time. Suppose that we're really unlucky and the partition sizes are really unbalanced. In particular, suppose that the pivot chosen by the partition function is always either the smallest or the largest element in the n-element subarray. Then one of the partitions will contain no elements and the other partition will contain $n-1$ elements—all but the pivot. So the recursive calls will be on subarrays of sizes 0 and $n-1$.

As in merge sort, the time for a given recursive call on an n-element subarray is $\Theta(n)$. In merge sort, that was the time for merging, but in quicksort it's the time for partitioning.

Worst-case running time

When quicksort always has the most unbalanced partitions possible, then the original call takes cn time for some constant c, the recursive call on $n-1$ elements takes $c(n-1)$ time, the recursive call on $n-2$ elements takes $c(n-2)$ time, and so on. Here's a tree of the subproblem sizes with their partitioning times:



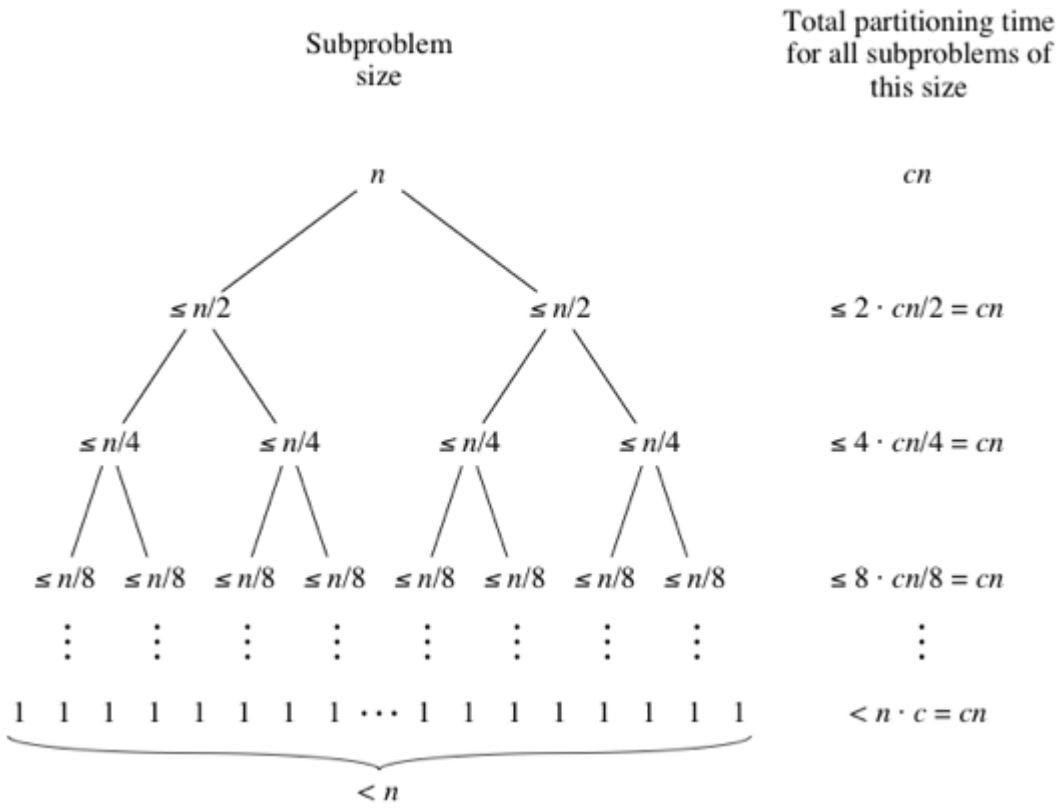
When we total up the partitioning times for each level, we get

$$cn + c(n-1) + c(n-2) + \cdots + 2c = c(n+(n-1) + (n-2) + \cdots + 2) = c((n+1)(n/2)-1)$$

The last line is because 1+2+3+⋯+n is the arithmetic series, as we saw when we analyzed selection sort. (We subtract 1 because for quicksort, the summation starts at 2, not 1.) We have some low-order terms and constant coefficients, but when we use big-Θ notation, we ignore them. In big-Θ notation, quicksort's worst-case running time is $\Theta(n^2)$.

Best-case running time

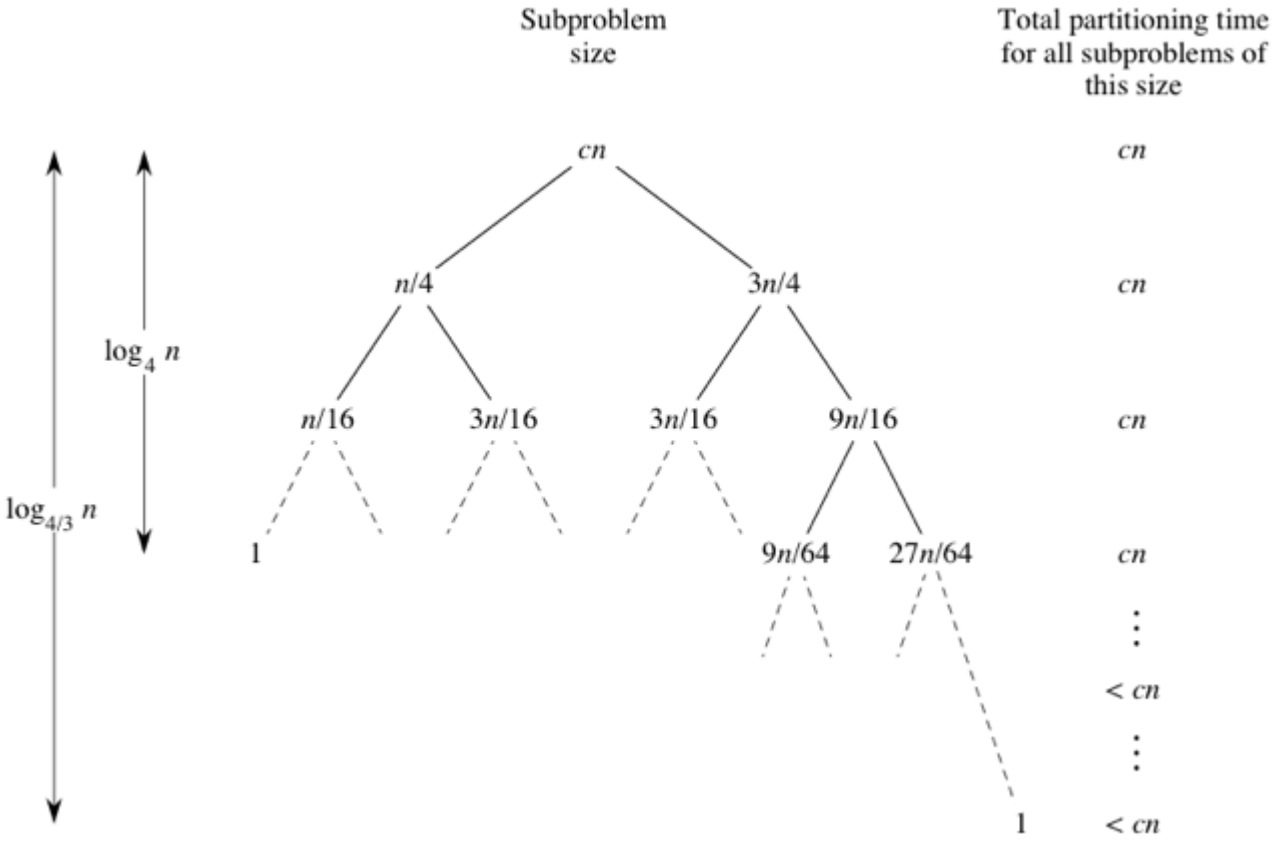
Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n-1)/2 elements. The latter case occurs if the subarray has an even number nn of elements and one partition has n/2 elements with the other having n/2-1. In either of these cases, each partition has at most n/2 elements, and the tree of subproblem sizes looks a lot like the tree of subproblem sizes for merge sort, with the partitioning times looking like the merging times:



Using big-Θ notation, we get the same result as for merge sort: Θ(nlgn).

Average-case running time

Showing that the average-case running time is also **Θ(nlgn)** takes some pretty involved mathematics, and so we won't go there. But we can gain some intuition by looking at a couple of other cases to understand why it might be O(nlgn). (Once we have O(nlgn), the Θ(nlgn) bound follows because the average-case running time cannot be better than the best-case running time.) First, let's imagine that we don't always get evenly balanced partitions, but that we always get at worst a 3-to-1 split. That is, imagine that each time we partition, one side gets 3n/4 elements and the other side gets n/4. (To keep the math clean, let's not worry about the pivot.) Then the tree of subproblem sizes and partitioning times would look like this:



The left child of each node represents a subproblem size 1/4 as large, and the right child represents a subproblem size 3/4 as large. Since the smaller subproblems are on the left, by following a path of left children, we get from the root down to a subproblem size of 1 faster than along any other path. As the figure shows, after $\mathbf{\log_4 n}$ levels, we get down to a subproblem size of 1. Why $\mathbf{\log_4 n}$ levels? It might be easiest to think in terms of starting with a subproblem size of 1 and multiplying it by 4 until we reach n. In other words, we're asking for what value of x is $\mathbf{4^x = n}$? The answer is $\log_4 n$. How about going down a path of right children? The figure shows that it takes $\mathbf{\log_{4/3} n}$ levels to get down to a subproblem of size 1. Why $\mathbf{\log_{4/3} n}$ levels? Since each right child is 3/4 of the size of the node above it (its parent node), each parent is 4/3 times the size of its right child. Let's again think of starting with a subproblem of size 1 and multiplying the size by 4/3 until we reach n. For what value of x is $\mathbf{(4/3)^x = n}$? The answer is $\mathbf{\log_{4/3} n}$.

In each of the first $\mathbf{\log_4 n}$ levels, there are n nodes (again, including pivots that in reality are no longer being partitioned), and so the total partitioning time for each of these levels is cncn. But what about the rest of the levels? Each has fewer than n nodes, and so the partitioning time for every level is at most cn. Altogether, there are $\mathbf{\log_{4/3} n}$ levels, and so the total partitioning time is O(n

\log$_{4/3}$n). Now, there's a mathematical fact that

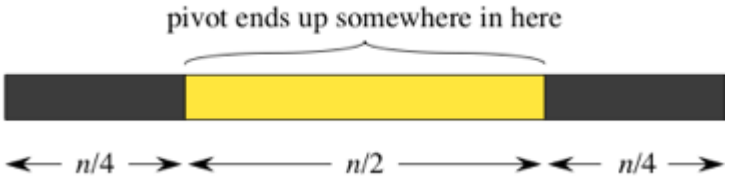
$$log_a n = \frac{log_b n}{log_b a}$$

for all positive numbers a, b, and n. Letting a = 4/3 and b = 2, we get that

$$log_{4/3} n = \frac{lgn}{lg(4/3)}$$

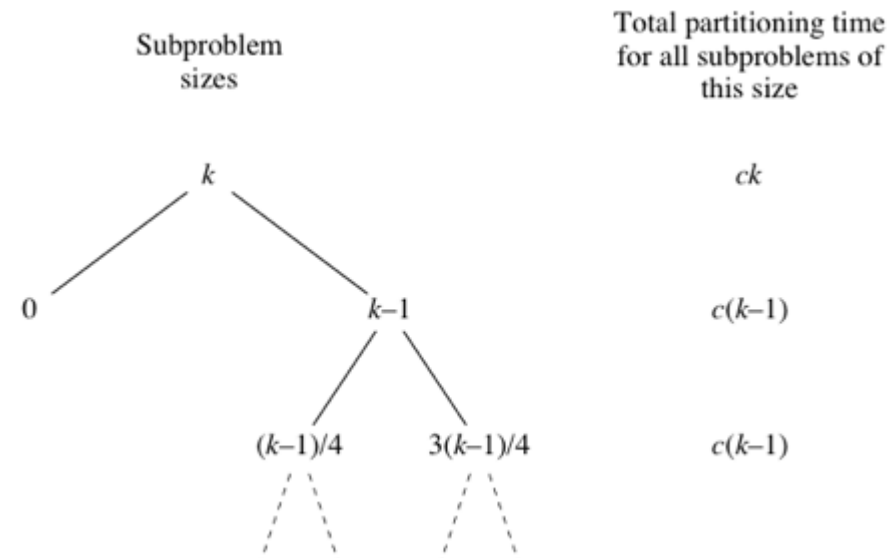
and so log$_{4/3}$n and lgn differ by only a factor of lg(4/3), which is a constant. Since constant factors don't matter when we use big-O notation, we can say that if all the splits are 3-to-1, then quicksort's running time is O(nlgn), albeit with a larger hidden constant factor than the best-case running time.

How often should we expect to see a split that's 3-to-1 or better? It depends on how we choose the pivot. Let's imagine that the pivot is equally likely to end up anywhere in an nn-element subarray after partitioning. Then to get a split that is 3-to-1 or better, the pivot would have to be somewhere in the "middle half":

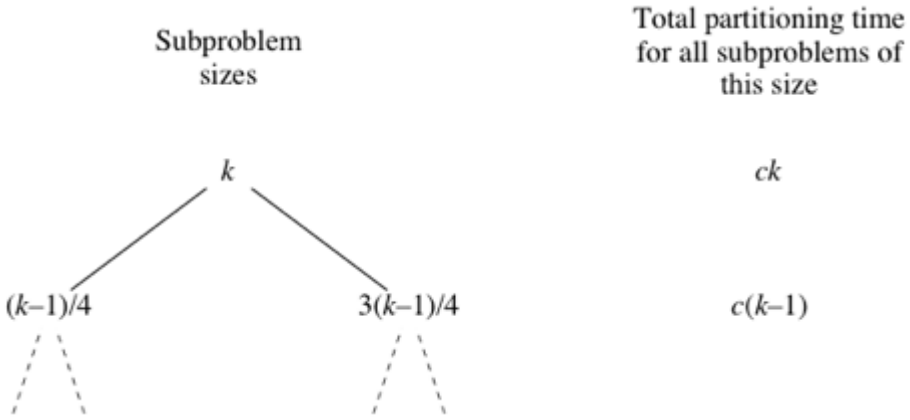


So, if the pivot is equally likely to end up anywhere in the subarray after partitioning, there's a 50% chance of getting at worst a 3-to-1 split. In other words, we expect a split of 3-to-1 or better about half the time.

The other case we'll look at to understand why quicksort's average-case running time is O(nlgn) is what would happen if the half of the time that we don't get a 3-to-1 split, we got the worst-case split. Let's suppose that the 3-to-1 and worst-case splits alternate, and think of a node in the tree with k elements in its subarray. Then we'd see a part of the tree that looks like this:



instead of like this:



Therefore, even if we got the worst-case split half the time and a split that's 3-to-1 or better half the time, the running time would be about twice the running time of getting a 3-to-1 split every time. Again, that's just a constant factor, and it gets absorbed into the big-O notation, and so in this case, where we alternate between worst-case and 3-to-1 splits, the running time is O(nlgn).

Bear in mind that this analysis is not mathematically rigorous, but it gives you an intuitive idea of why the average-case running time might be O(nlgn).

Randomized quicksort

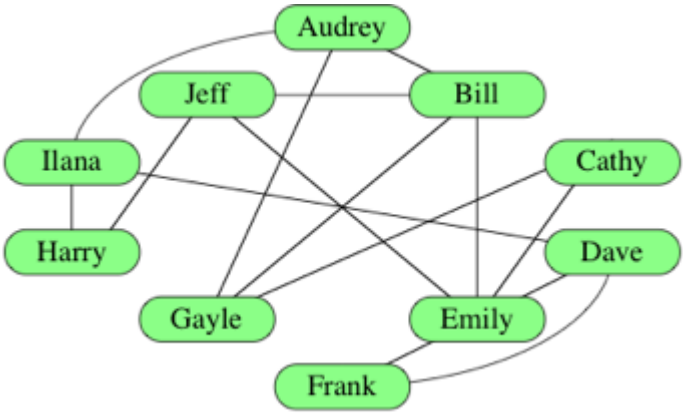
Suppose that your worst enemy has given you an array to sort with quicksort, knowing that you always choose the rightmost element in each subarray as the pivot, and has arranged the array so that you always get the worst-case split. How can you foil your enemy?

You could not necessarily choose the rightmost element in each subarray as the pivot. Instead, you could randomly choose an element in the subarray, and use that element as the pivot. But wait—the partition function assumes that the pivot is in the rightmost position of the subarray. No problem—just swap the element that you chose as the pivot with the rightmost element, and then partition as before. Unless your enemy knows how you choose random locations in the subarray, you win!

In fact, with a little more effort, you can improve your chance of getting a split that's at worst 3-to-1. Randomly choose not one, but three elements from the subarray, and take median of the three as the pivot (swapping it with the rightmost element). By the **median**, we mean the element of the three whose value is in the middle. We won't show why, but if you choose the median of three randomly chosen elements as the pivot, you have a 68.75% chance (11/16) of getting a 3-to-1 split or better. You can go even further. If you choose five elements at random and take the median as the pivot, your chance of at worst a 3-to-1 split improves to about 79.3% (203/256). If you take the median of seven randomly chosen elements, it goes up to about 85.9% (1759/2048). Median of nine? About 90.2% (59123/65536). Median of 11? About 93.1% (488293/524288). You get the picture. Of course, it doesn't necessarily pay to choose a large number of elements at random and take their median, for the time spent doing so could counteract the benefit of getting good splits almost all the time.

Describing graphs

Here's one way to represent a social network:



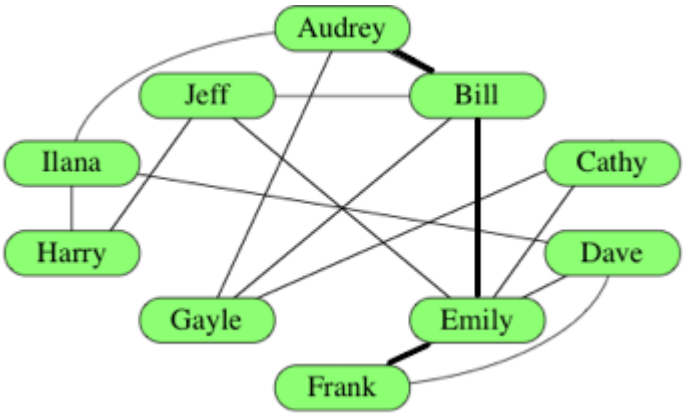
A line between the names of two people means that they know each other. If there's no line between two names, then the people do not know each other. The relationship "know each other" goes both ways; for example, because Audrey knows Gayle, that means Gayle knows Audrey.

This social network is a **graph**. The names are the **vertices** of the graph. (If you're talking about just one of the vertices, it's a **vertex**.) Each line is an **edge**, connecting two vertices. We denote an edge connecting vertices **u** and **v** by the pair **(u,v)**. Because the "know each other" relationship goes both ways, this graph is **undirected**. An undirected edge **(u,v)** is the same as **(v,u)**. Later, we'll see **directed graphs**, in which relationships between vertices don't necessarily go both ways. In an undirected graph, an edge between two vertices, such as the edge between Audrey and Gayle, is **incident** on the two vertices, and we say that the vertices connected by an edge are **adjacent** or **neighbors**. The number of edges incident on a vertex is the **degree** of the vertex.

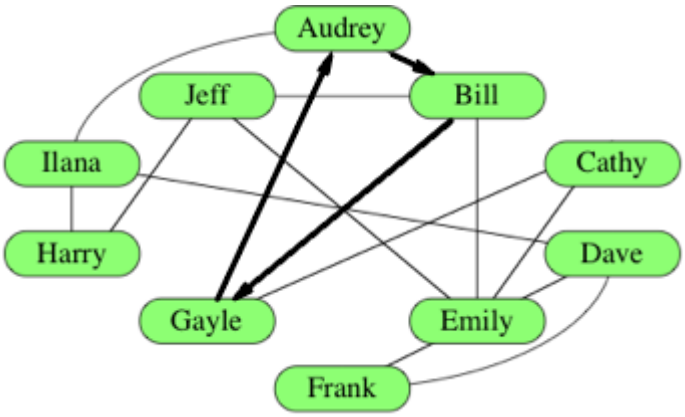
Audrey and Frank do not know each other. Suppose that Frank wanted to be introduced to Audrey. How could he get an introduction? Well, he knows Emily, who knows Bill, who knows Audrey. We say that there is a **path** of three edges between Frank and Audrey. In fact, that is the most direct way for Frank

to meet Audrey; there is no path between them with fewer than three edges.

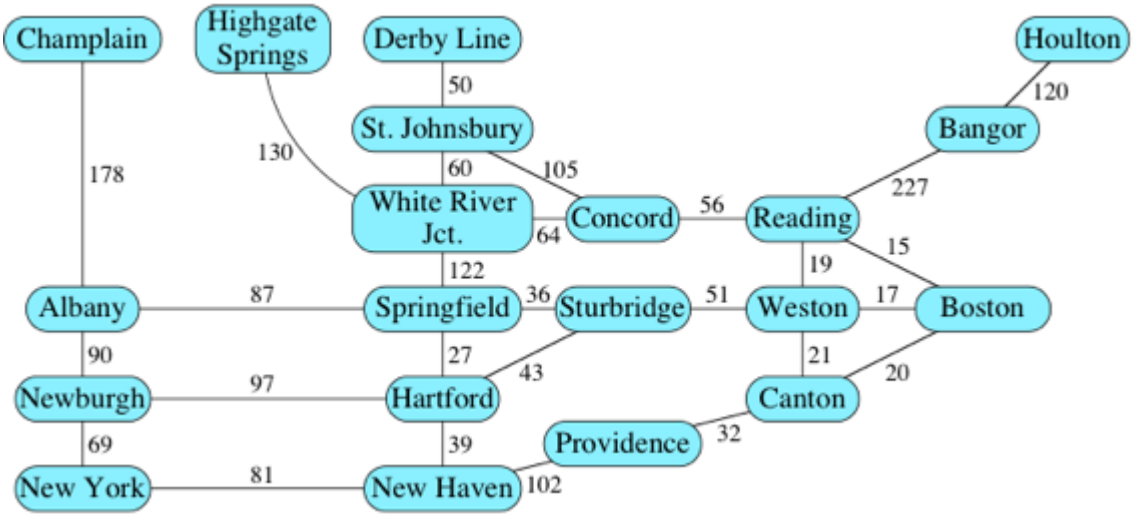
We call a path between two vertices with the fewest edges a **shortest path**. We've highlighted that particular shortest path below:



When a path goes from a particular vertex back to itself, that's a **cycle**. The social network contains many cycles; one of them goes from Audrey to Bill to Emily to Jeff to Harry to Ilana and back to Audrey. There's a shorter cycle containing Audrey, shown below: Audrey to Bill to Gayle and back to Audrey. What other cycles can you find?

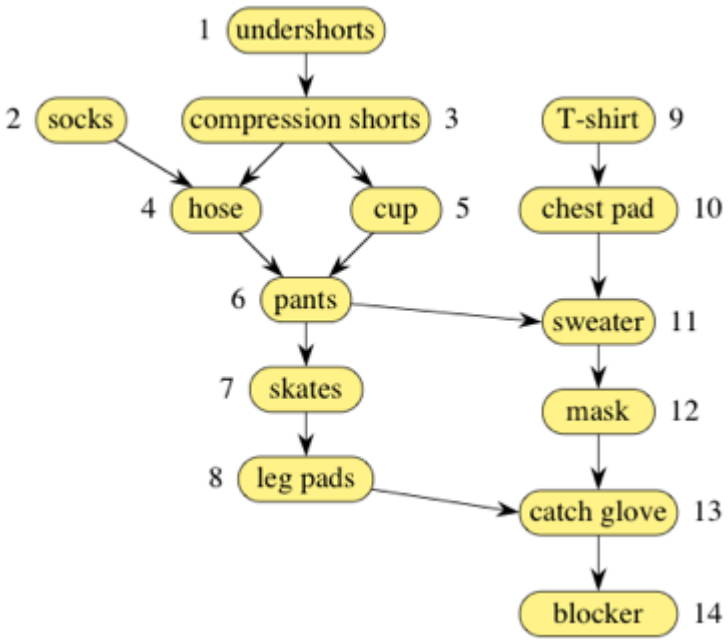


Sometimes we put numeric values on the edges. For example, in the social network, we might use values to indicate how well two people know each other. To bring in another example, let's represent a road map as a graph. Assuming that there are no one-way streets, a road map is also an undirected graph, with cities as vertices, roads as edges, and the values on edges indicating the distance of each road. For example, here's a road map, not to scale, of some of the interstate highways in the northeastern U.S., with distances next to edges:



The general term we use for a number that we put on an edge is its **weight**, and a graph whose edges have is a **weighted graph**. In the case of a road map, if you want to find the shortest route between two locations, you're looking for a path between two vertices with the minimum sum of edge weights over all paths between the two vertices. As with unweighted graphs, we call such a path a shortest path. For example, the **shortest path** in this graph from New York to Concord goes from New York to New Haven to Hartford to Sturbridge to Weston to Reading to Concord, totaling 289 miles.

The relationship between vertices does not always go both ways. In a road map, for example, there could be one-way streets. Or here's a graph showing the order in which a goalie in ice hockey could get dressed:



Now edges, shown with arrows, are **directed**, and we have a **directed graph**. Here, the directions show which pieces of equipment must be put on before

Here, the directions show which pieces of equipment must be put on before other pieces. For example, the edge from chest pad to sweater indicates that the chest pad must be put on before the sweater. The numbers next to the vertices show one of the many possible orders in which to put on the equipment, so that undershorts go on first, then socks, then compression shorts, and so on, with the blocker going on last. You might have noticed that this particular directed graph has no cycles; we call such a graph a **directed acyclic graph**, or **DAG**. Of course, we can have weighted directed graphs, such as road maps with one-way streets and road distances.

We use different terminology with directed edges. We say that a directed edge **leaves** one vertex and **enters** another. For example, one directed edge leaves the vertex for chest pad and enters the vertex for sweater. If a directed edge leaves vertex $\mathbf{u}$ and enters vertex $\mathbf{v}$, we denote it by $\mathbf{(u,v)}$, and the order of the vertices in the pair matters. The number of edges leaving a vertex is its **out-degree**, and the number of edges entering is the **in-degree**.

As you might imagine, graphs—both directed and undirected—have many applications for modeling relationships in the real world.

Graph sizes

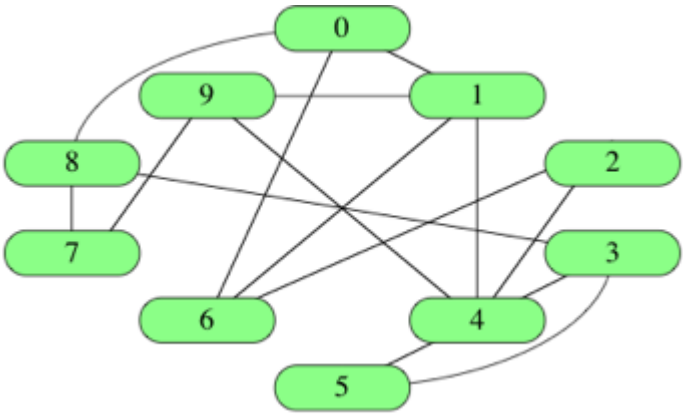
When we work with graphs, it's helpful to be able to talk about the set of vertices and the set of edges. We usually denote the vertex set by $\mathbf{V}$ and the edge set by $\mathbf{E}$. When we represent a graph or run an algorithm on a graph, we often want to use the sizes of the vertex and edge sets in asymptotic notation. For example, suppose that we want to talk about a running time that is linear in the number of vertices. Strictly speaking, we should say that it's $\mathbf{\Theta(|V|)}$, using the notation $|\cdot|$ to denote the size of a set. But using this set-size notation in asymptotic notation is cumbersome, and so we adopt the convention that in asymptotic notation, and only in asymptotic notation, we drop the set-size notation with the understanding that we're talking about set sizes. So, instead of writing $\mathbf{\Theta(|V|)}$, we write just $\mathbf{\Theta(V)}$. Similarly, instead of $\mathbf{\Theta(lg|E|)}$, we write $\mathbf{\Theta(lgE)}$.

Representing graphs

There are several ways to represent graphs, each with its advantages and disadvantages. Some situations, or algorithms that we want to run with graphs as input, call for one representation, and others call for a different representation. Here, we'll see three ways to represent graphs.

We'll look at three criteria. One is how much memory, or space, we need in each representation. We'll use asymptotic notation for that. Yes, we can use asymptotic notation for purposes other than expressing running times! It's really a way to characterize *functions*, and a function can describe a running time, an amount of space required, or some other resource. The other two criteria we'll use relate to time. One is how long it takes to determine whether a given edge is in the graph. The other is how long it takes to find the neighbors of a given vertex.

It is common to identify vertices not by name (such as "Audrey," "Boston," or "sweater") but instead by a number. That is, we typically number the $|\mathbf{V}|$ vertices from 0 to $|\mathbf{V}|\mathbf{-1}$. Here's the social network graph with its **10** vertices identified by numbers rather than names:



Edge lists

One simple way to represent a graph is just a list, or array, of $|\mathbf{E}|$ edges, which we call an **edge list**. To represent an edge, we just have an array of two vertex

numbers, or an array of objects containing the vertex numbers of the vertices that the edges are incident on. If edges have weights, add either a third element to the array or more information to the object, giving the edge's weight. Since each edge contains just two or three numbers, the total space for an edge list is **Θ(E)**. For example, here's how we represent an edge list in a programming language for the social network graph (*syntax might differ a little bit for each programming language but the concept is the same*):

```
[[0,1], [0,6], [0,8], [1,4], [1,6],
 [1,9], [2,4], [2,6], [3,4], [3,5],
 [3,8], [4,5], [4,9], [7,8], [7,9]
]
```

Edge lists are simple, but if we want to find whether the graph contains a particular edge, we have to search through the edge list. If the edges appear in the edge list in no particular order, that's a linear search through **|E|** edges. Question to think about: How can you organize an edge list to make searching for a particular edge take **O(lgE)** time? The answer is a little tricky.

Adjacency matrices

For a graph with **|V|** vertices, an adjacency matrix is a **|V|x|V|** matrix of **0s** and **1s**, where the entry in row **i** and column **j** is **1** if and only if the edge **(i,j)** is in the graph. If you want to indicate an edge weight, put it in the **row i**, **column j** entry, and reserve a special value (perhaps null) to indicate an absent edge. Here's the adjacency matrix for the social network graph:

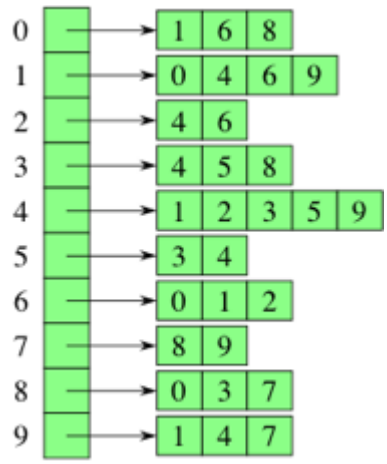
	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0

With an adjacency matrix, we can find out whether an edge is present in constant time, by just looking up the corresponding entry in the matrix. For example, if the adjacency matrix is named **graph**, then we can query whether **edge (i, j)** is in the graph by looking at **graph[i][j]**. So what's the disadvantage of an adjacency matrix? Two things, actually. First, it takes $\Theta(V^2)$ space, even if the graph is **sparse**: relatively few edges. In other words, for a sparse graph, the adjacency matrix is mostly 0s, and we use lots of space to represent only a few edges. Second, if you want to find out which vertices are adjacent to a given vertex **i**, you have to look at all $|V|$ entries in row **i**, even if only a small number of vertices are adjacent to vertex **i**.

For an undirected graph, the adjacency matrix is **symmetric**: the row **i**, column **j** entry is 1 if and only if the row jj, column ii entry is 1. For a directed graph, the adjacency matrix need not be symmetric.

Adjacency lists

Representing a graph with **adjacency lists** combines adjacency matrices with edge lists. For each vertex **i**, store an array of the vertices adjacent to it. We typically have an array of $|V|$ adjacency lists, one adjacency list per vertex. Here's an adjacency-list representation of the social network graph:



Vertex numbers in an adjacency list are not required to appear in any particular order, though it is often convenient to list them in increasing order, as in this example.

We can get to each vertex's adjacency list in constant time, because we just have to index into an array. To find out whether an edge **(i, j)** is present in the

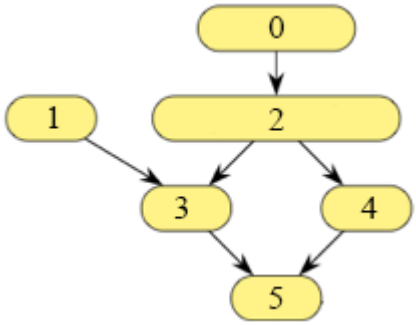
graph, we go to **i's** adjacency list in constant time and then look for **j** in **i's** adjacency list. How long does that take in the worst case? The answer is **Θ(d)**, where **d** is the degree of vertex **i**, because that's how long **i's** adjacency list is. The degree of vertex **i** could be as high as **|V| - 1** (if **i** is adjacent to all the other **|V| - 1** vertices) or as low as 0 (if **i** is isolated, with no incident edges). In an undirected graph, vertex **j** is in vertex **i's** adjacency list if and only if **i** is in **j's** adjacency list. If the graph is weighted, then each item in each adjacency list is either a two-item array or an object, giving the vertex number and the edge weight.

You can use a for-loop to iterate through the vertices in an adjacency list.

How much space do adjacency lists take? We have **|V|** lists, and although each list could have as many as |V| - 1 vertices, in total the adjacency lists for an undirected graph contain **2|E|** elements. Why **2|E|**? Each edge **(i,j)** appears exactly twice in the adjacency lists, once in **i's** list and once in **j's** list, and there are **|E|** edges. For a directed graph, the adjacency lists contain a total of **|E|** elements, one element per directed edge.

Challenge: Store a graph

Here's the graph that we will use for the following two challenges.



Challenge 1: Store an adjacency matrix

We've stored a graph, with 6 vertices indexed 0-5, as an edge list in the variable edgeList. Store the same graph, as an adjacency matrix, in the variable **adjMatrix**.

Java | Python | C++ | JS

```
import java.util.Arrays;

class Solution {
  public static int[][] edgeList = new int[][] {
    new int[] {0, 2},
    new int[] {1, 3},
    new int[] {2, 3},
    new int[] {2, 4},
    new int[] {3, 5},
    new int[] {4, 5}
  };

  // Fill in this adjMatrix to represent the graph
  public static int[][] adjMatrix = null;
}
```

Challenge 2: Store an adjacency list

Store the same graph, as an adjacency list, in the variable **adjList**.

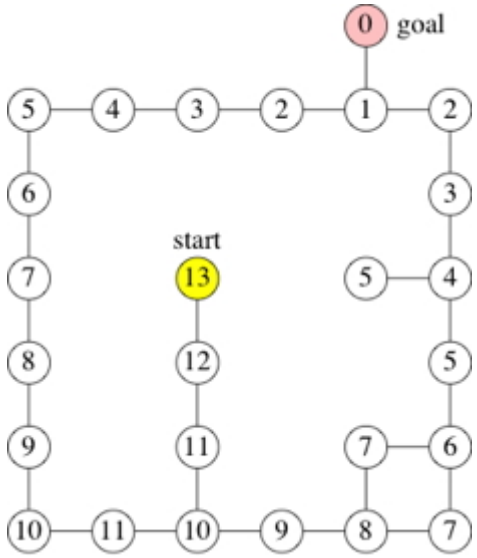
Java | Python | C++ | JS

```
edgeList = [ [0, 2], [1, 3], [2, 3], [2, 4], [3, 5], [4, 5] ];

# Fill in this adjList to represent the graph
adjList = [];
```

Breadth-first search and its uses

In the introductory tutorial, you played with having a character move through a maze to reach a goal. You started by saying that the goal is zero steps away from itself. Then you found all the squares that were one step away from the goal. Then all squares two steps away from the goal. Then three steps, and so on, until you reached the square where the character started. If you kept track of which square at distance **k** you came from to get to a square at distance **k+1**, you could backtrack to find a route from the character to the goal.

Here's the picture that we saw in the introductory tutorial:



Now you can recognize it as an undirected graph. Each vertex corresponds to a square that is not part of a wall, and each edge is incident on adjacent squares.

The route found by the above procedure has an important property: no other route from the character to the goal goes through fewer squares. That's because we used an algorithm known as **breadth-first search** to find it. Breadth-first search, also known as **BFS**, finds shortest paths from a given **source vertex** to all other vertices, in terms of the number of edges in the paths.

Here's another example of breadth-first search: the "six degrees of Kevin Bacon" game. Here, players try to connect movie actors and actresses to Kevin Bacon according to a chain of who appeared with whom in a movie. The shorter the chain, the better, and it's astounding how many actors and actresses can get to Kevin Bacon in a chain of six or fewer. As an example, take Kate Bell, an Australian actress. She was in ***MacBeth*** with Nash Edgerton in 2006; Edgerton was in ***The Matrix Reloaded*** with Laurence Fishburne in 2003; and Fishburne was in ***Mystic River*** with Kevin Bacon in 2003. Therefore, Kate Bell's "Kevin Bacon number" is 3. In fact, there are several ways to find that Kate Bell's Kevin Bacon number is 3. You can look up any actor's or actress's Kevin Bacon number at the Oracle of Bacon website.

As you might have guessed, the Oracle of Bacon website maintains an undirected graph in which each vertex corresponds to an actor or actress, and if two people appeared in the same film, then there is an edge incident on their vertices. A breadth-first search from the vertex for Kevin Bacon finds the shortest chain to all other actors and actresses.

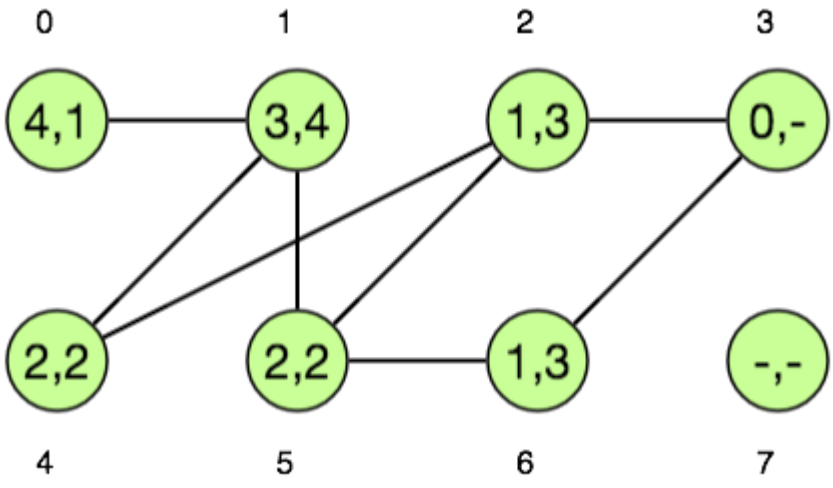
The breadth-first search algorithm

Breadth-first search assigns two values to each vertex **v**:

- A **distance**, giving the minimum number of edges in any path from the source vertex to vertex **v**.
- The **predecessor** vertex of **v** along some shortest path from the source vertex. The source vertex's predecessor is some special value, such as *null*, indicating that it has no predecessor.

If there is no path from the source vertex to vertex **v**, then **v**'s distance is infinite and its predecessor has the same special value as the source's predecessor.

For example, here's an undirected graph with eight vertices, numbered 0 to 7, with vertex numbers appearing above or below the vertices. Inside each vertex are two numbers: its distance from the source, which is vertex 3, followed by its predecessor on a shortest path from vertex 3. A dash indicates *null*:

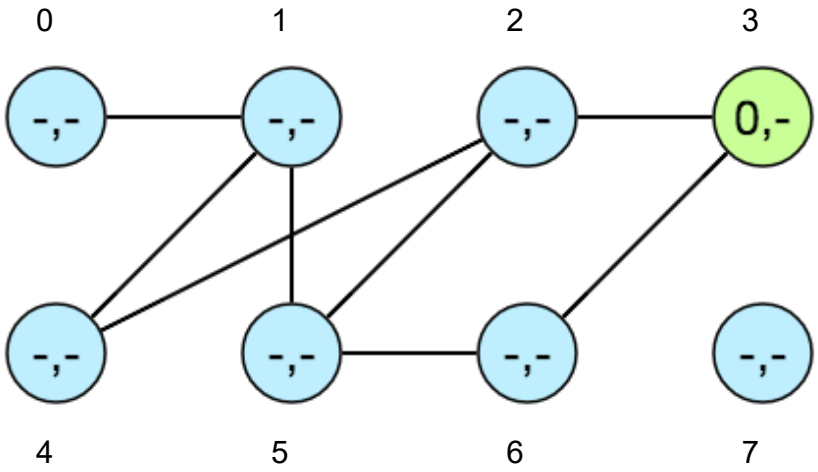


In BFS, we initially set the distance and predecessor of each vertex to the

special value (*null*). We start the search at the source and assign it a distance of 0. Then we visit all the neighbors of the source and give each neighbor a distance of 1 and set its predecessor to be the source. Then we visit all the neighbors of the vertices whose distance is 1 and that have not been visited before, and we give each of these vertices a distance of 2 and set its predecessor to be vertex from which we visited it. We keep going until all vertices reachable from the source vertex have been visited, always visiting all vertices at distance **k** from the source before visiting any vertex at distance **k+1**.

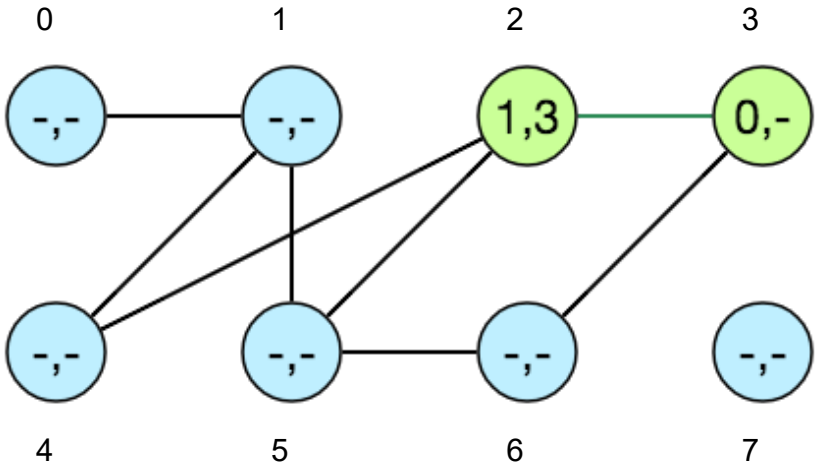
Given the example above, here are the steps plus a visualization to play through each step:

- Start by visiting vertex 3, the source, setting its distance to 0.
- Then visit vertices 2 and 6, setting their distance to 1 and their predecessor to vertex 3.
- Start visiting from vertices at distance 1 from the source, beginning with vertex 2. From vertex 2, visit vertices 4 and 5, setting their distance to 2 and their predecessor to vertex 2. Don't visit vertex 3, because it has already been visited.
- From vertex 6, don't visit vertex 5, because it was just visited from vertex 2, and don't visit vertex 3, either.
- Now start visiting from vertices at distance 2 from the source. Start by visiting from vertex 4. Vertex 2 has already been visited. Visit vertex 1, setting its distance to 3 and its predecessor to vertex 4.
- From vertex 5, don't visit any of its neighbors, because they have all been visited.
- Now start visiting from vertices at distance 3 from the source. The only such vertex is vertex 1. Its neighbors, vertices 4 and 5, have already been visited. But vertex 0 has not. Visit vertex 0, setting its distance to 4 and its predecessor to vertex 1.
- Now start visiting from vertices at distance 4 from the source. That's just vertex 0, and its neighbor, vertex 1, has already been visited. We're done!



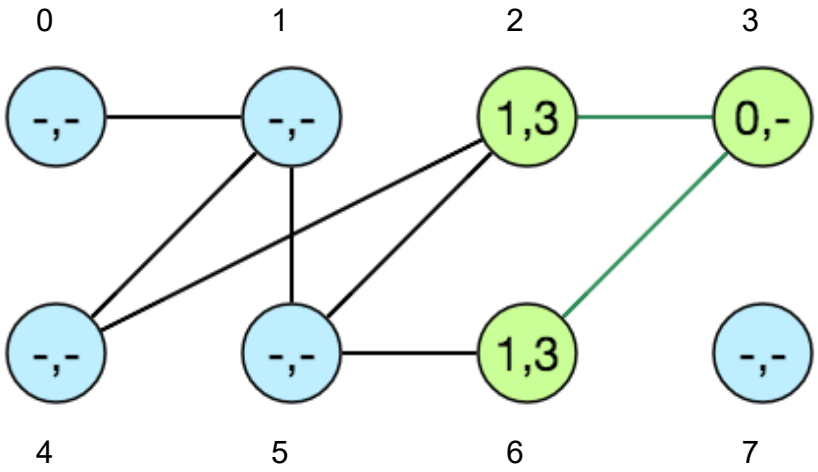
Start by visiting vertex 3, the source, setting its distance to 0.

1 of 7

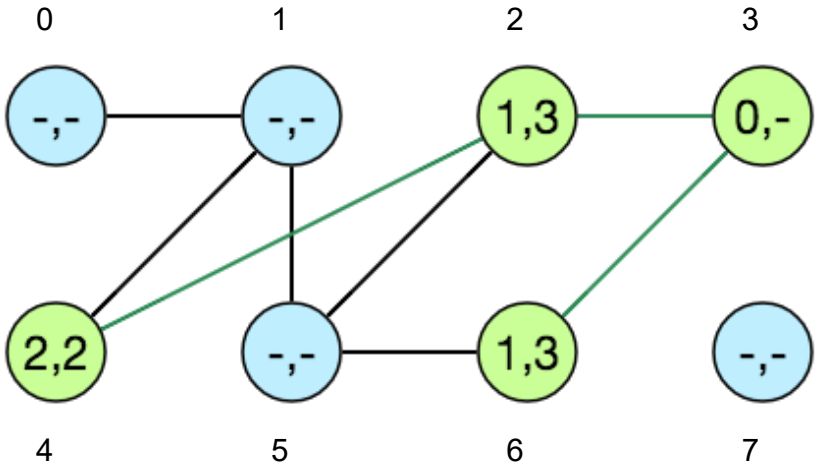


Visit vertex 2, setting its distance to 1 and its predecessor to 3.

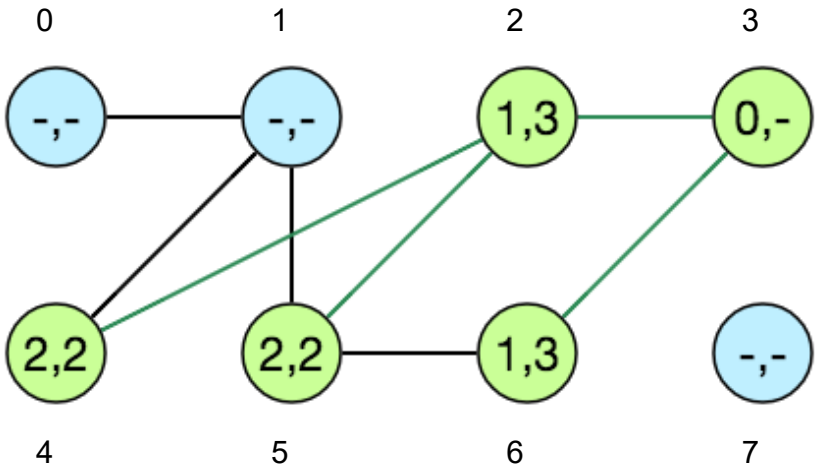
2 of 7



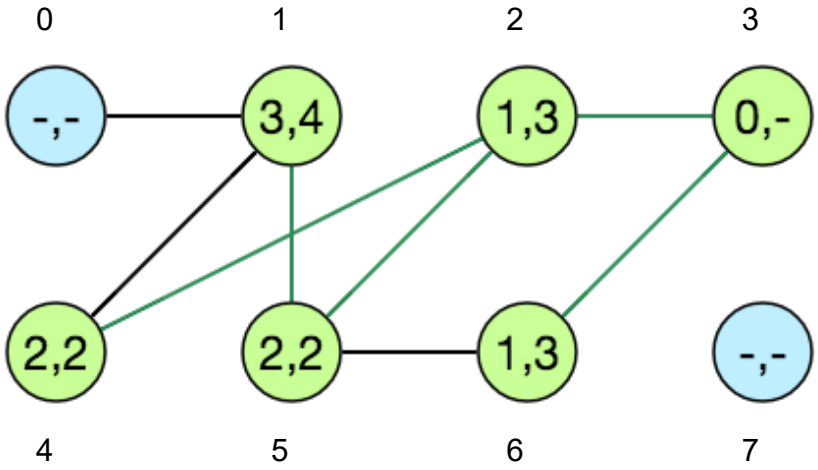
Visit vertex 6, setting its distance to 1 and its predecessor to 3.



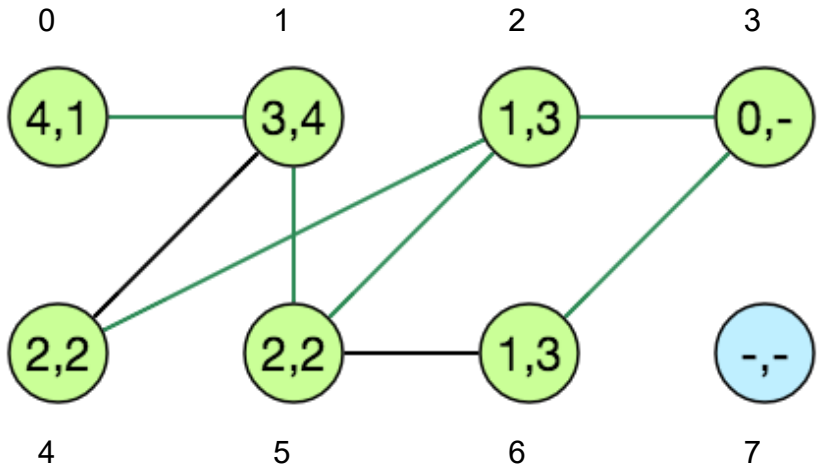
Visit vertex 4, setting its distance to 2 and its predecessor to 2.



Visit vertex 5, setting its distance to 2 and its predecessor to 2.



Visit vertex 1, setting its distance to 3 and its predecessor to 4.



Visit vertex 0, setting its distance to 4 and its predecessor to 1.

7 of 7

Notice that because there is no path from vertex 3 to vertex 7, the search never visits vertex 7. Its distance and predecessor remain unchanged from their initial values of *null*.

A couple of questions come up. One is how to determine whether a vertex has been visited already. That's easy: a vertex's distance is *null* until it has been visited, at which time it gets a numeric value for its distance. Therefore, when we examine the neighbors of a vertex, we visit only the neighbors whose distance is currently *null*.

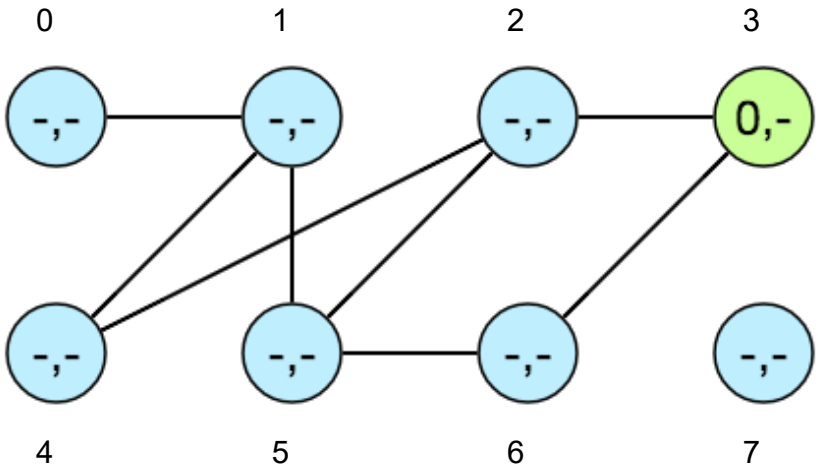
The other question is how to keep track of which vertices have already been visited but have not yet been visited from. We use a **queue**, which is a data structure that allows us to insert and remove items, where the item removed is always the one that has been in the queue the longest. We call this behavior **first in, first out**. A queue has three operations:

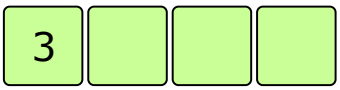
- ***enqueue(obj)*** inserts an object into the queue.
- ***dequeue()*** removes from the queue the object that has been in it the longest, returning this object.
- ***isEmpty()*** returns true if the queue currently contains no objects, and

false if the queue contains at least one object.

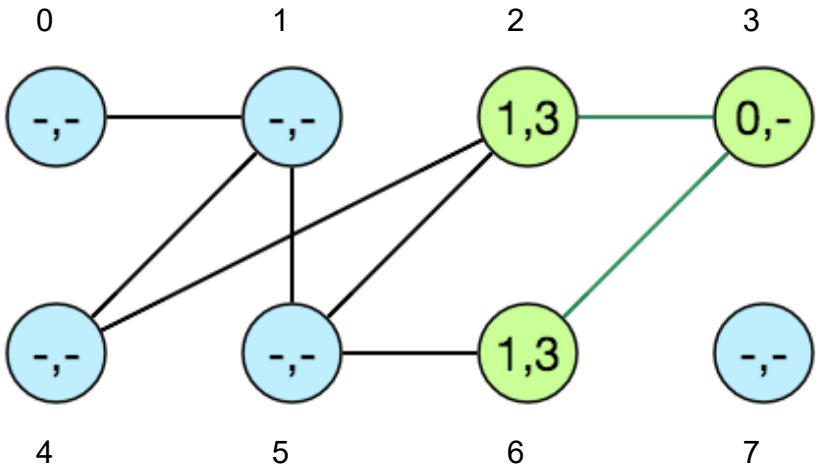
Whenever we first visit any vertex, we enqueue it. At the start, we enqueue the source vertex because that's always the first vertex we visit. To decide which vertex to visit next, we choose the vertex that has been in the queue the longest and remove it from the queue—in other words, we use the vertex that's returned from *dequeue()*. Given our example graph, here's what the queue looks like for each step, plus the previous visualization shown with the queue state:

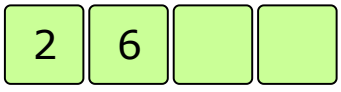
- Initially, the queue contains just vertex 3 with distance 0.
- Dequeue vertex 3, and enqueue vertices 2 and 6, both with distance 1. The queue now contains vertex 2 with distance 1 and vertex 6 with distance 1.
- Dequeue vertex 2, and enqueue vertices 4 and 5, both with distance 2. The queue now contains vertex 6 with distance 1, vertex 4 with distance 2, and vertex 5 with distance 2.
- Dequeue vertex 6, and don't enqueue any vertices. The queue now contains vertex 4 with distance 2 and vertex 5 with distance 2.
- Dequeue vertex 4, and enqueue vertex 1 with with distance 3. The queue now contains vertex 5 with distance 2 and vertex 1 with distance 3.
- Dequeue vertex 5, and don't enqueue any vertices. The queue now contains just vertex 1 with distance 3.
- Dequeue vertex 1, and enqueue vertex 0 with distance 4. The queue now contains just vertex 0 with distance 4.
- Dequeue vertex 0, and don't enqueue any vertices. The queue is now empty. Because the queue is empty, breadth-first search terminates.



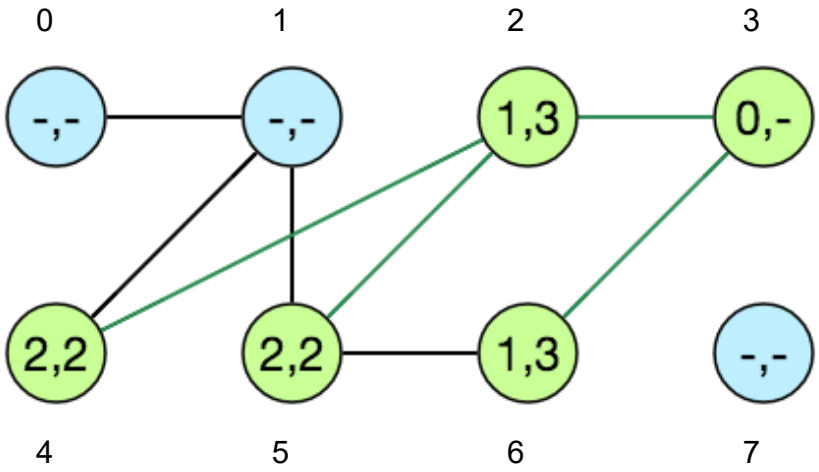
Queue 

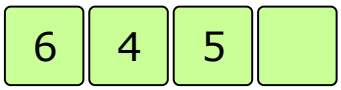
Initially, the queue contains just vertex 3 with distance 0.



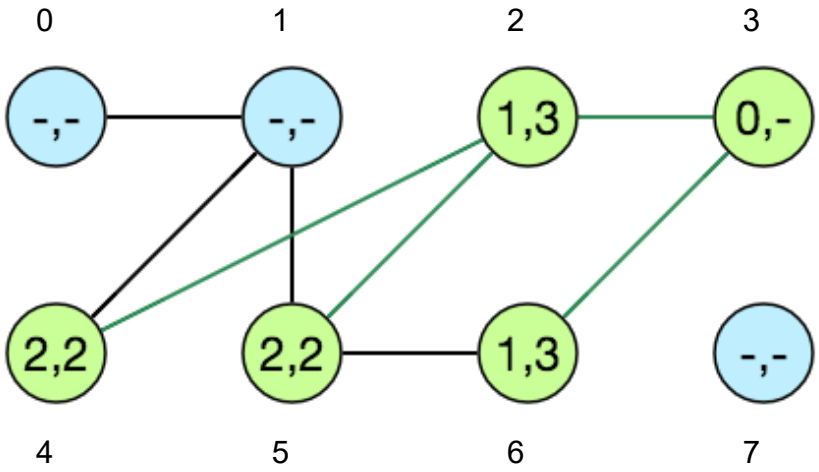
Queue 

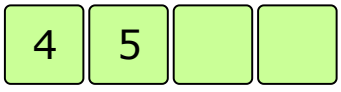
Remove 3 from the queue. Add 2 to the queue.
Add 6 to the queue.



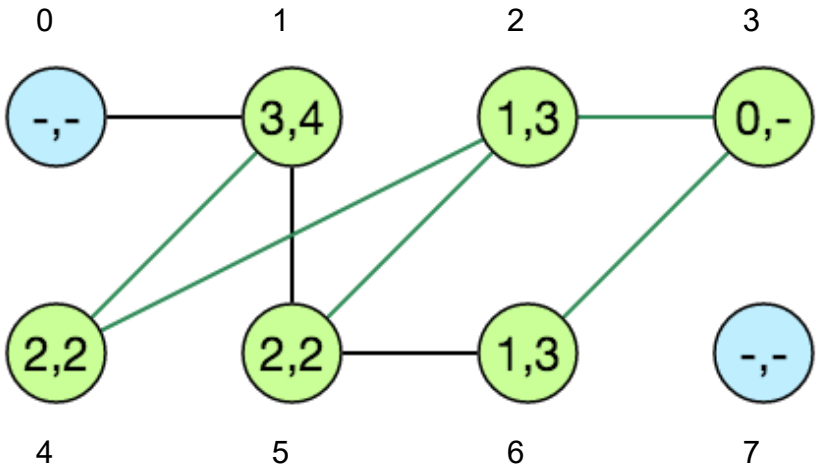
Queue 

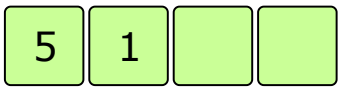
Remove 2 from the queue. Add 4 to the queue.
Add 5 to the queue.



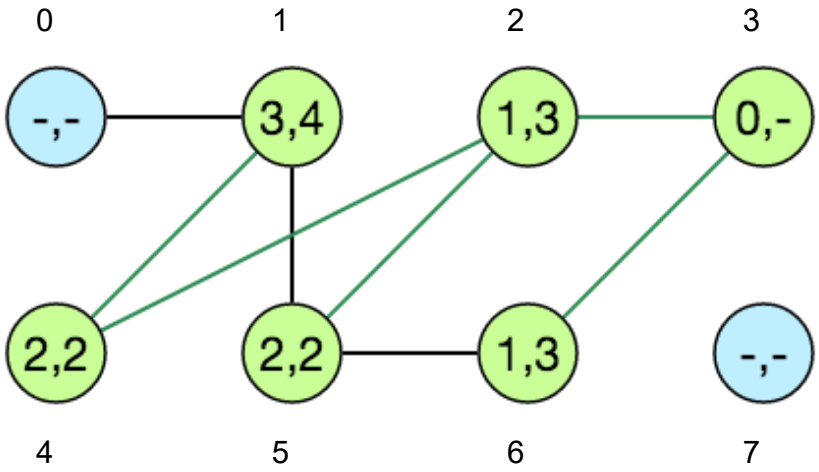
Queue 

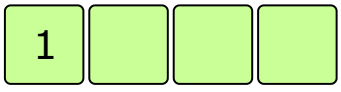
Remove 6 from the queue.
Don't enqueue any vertices.



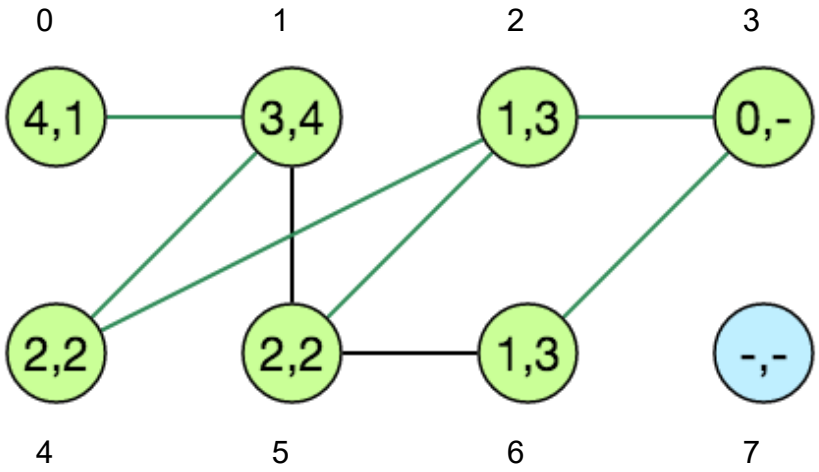
Queue 

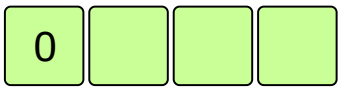
Remove 4 from the queue.
Add 1 to the queue.



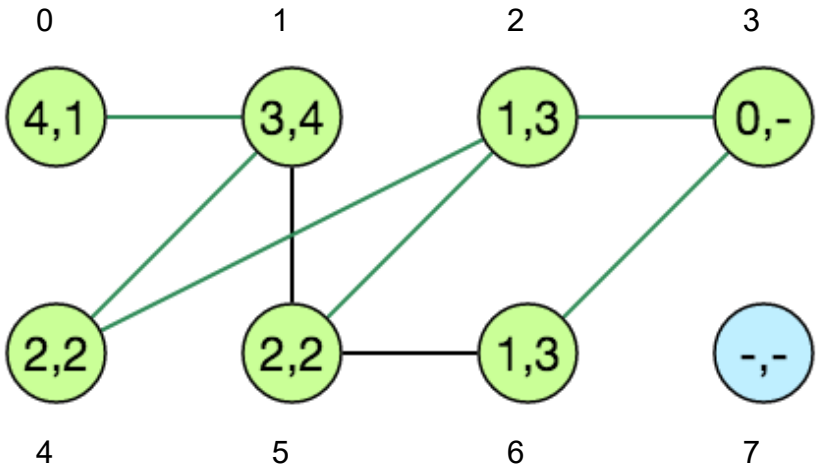
Queue 

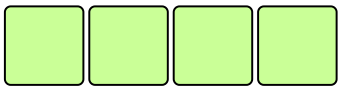
Remove 5 from the queue.
Don't enqueue any vertices.



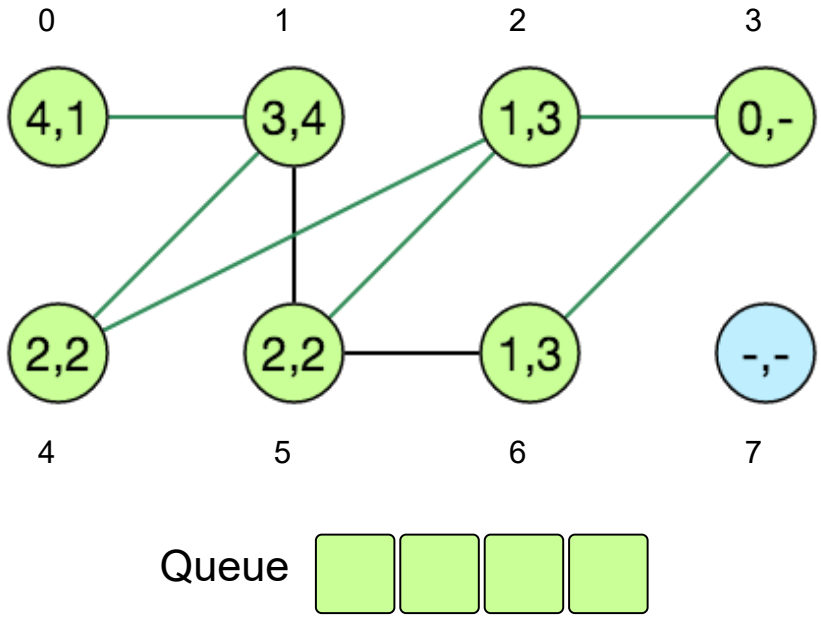
Queue 

Remove 1 from the queue.
Add 0 to the queue.



Queue 

Remove 0 from the queue.
Don't enqueue any vertices.



Notice that at each moment, the queue either contains vertices all with the same distance, or it contains vertices with distance **k** followed by vertices with distance **k+1**. That's how we ensure that we visit all vertices at distance **k** before visiting any vertices at distance **k+1**.

Challenge: Implement breadth-first search

Implement BFS

In this step, you'll finish implementing the doBFS function, which performs a breadth-first search on a graph and returns an array of objects describing each vertex.

For each vertex v, the object's distance property should be vertex v's distance from the source, and the predecessor property should be vertex v's predecessor on a shortest path from the source. If there is no path from the source to vertex v, then v's distance and predecessor should both be *null*. The source's predecessor should also be *null*.

In the starter code, the function initializes the distance and predecessor values to null, and then enqueues the source vertex. It is up to you to implement the rest of the algorithm, as described in the pseudocode.

Java | Python | C++ | JS

```
import java.util.LinkedList;
import java.util.Queue;

class BFSInfo {
  public BFSInfo() {
    this.distance = -1;
    this.predecessor = -1;
  }

  public BFSInfo(int distance, int predecessor) {
    this.distance = distance;
    this.predecessor = predecessor;
  }

  public int distance;
  public int predecessor;
};

class Solution {
  public static BFSInfo[] doBFS(int[][] graph, int source) {
    System.out.println(graph.length);
    BFSInfo[] bfsInfo = new BFSInfo[graph.length];
```

```
    bfsInfo[source] = new BFSInfo();
    bfsInfo[source].distance = 0;

    Queue<Integer> q = new LinkedList<Integer>();
    q.add(source);

    // Traverse the graph

    // As long as the queue is not empty:
    //  Repeatedly dequeue a vertex u from the queue.
    //
    //  For each neighbor v of u that has not been visited:
    //     Set distance to 1 greater than u's distance
    //     Set predecessor to u
    //     Enqueue v
    //
    //  Hint:
    //  use graph to get the neighbors,
    //  use bfsInfo for distances and predecessors

    return bfsInfo;
  }
}
```

Analysis of breadth-first search

How long does breadth-first search take for a graph with vertex set V and edge set E? The answer is $O(V+E)$ time.

Let's see what $O(V+E)$ time means. Assume for the moment that $|E| \geq |V|$, which is the case for most graphs, especially those for which we run breadth-first search. Then $|V| + |E| \leq |E| + |E| = 2 \cdot |E|$. Because we ignore constant factors in asymptotic notation, we see that when $|E| \geq |V|$, $O(V+E)$ really means $O(E)$. If, however, we have $|E| < |V|$, then $|V| + |E| \leq |V| + |V| = 2 \cdot |V|$, and so $O(V+E)$ really means $O(V)$. We can put both cases together by saying that $O(V+E)$ really means $O(\max(V,E))$. In general, if we have parameters x and y, then $O(x + y)$ really means $O(\max(x, y))$.

(Note, by the way, that a graph is **connected** if there is a path from every vertex to all other vertices. The minimum number of edges that a graph can have and still be connected is $|V|-1$. A graph in which $|E| = |V|-1$ is called a **free tree**.)

How is it that breadth-first search runs in $O(V+E)$ time? It takes $O(V)$ time to initialize the distance and predecessor for each vertex $\Theta(V)$ time, actually). Each vertex is visited at most one time, because only the first time that it is reached is its distance *null*, and so each vertex is enqueued at most one time. Since we examine the edges incident on a vertex only when we visit from it, each edge is examined at most twice, once for each of the vertices it's incident on. Thus, breadth-first search spends $O(V+E)$ time visiting vertices.

Why did we port Cormen and Balkcom's Algorithms course?

WE'LL COVER THE FOLLOWING

"Algorithms" by Dartmouth Computer Science professors Thomas Cormen and Devin Balkcom is an amazing course.

We love Khan Academy and this is in no way intended to challenge or discount the great work they are doing. We wanted to test Educative's authoring platform against a well developed course. In addition, we added challenges in multiple languages.

Here are a little bit more details.

Challenges in Multiple Programming Languages

We are launching with challenges in **Java**, **Python**, **C++** and obviously **Javascript**

If you have a favorite language that's not covered, drop us a note. If you are willing to write the challenge stubs and test cases in your favorite language, talk to us.

Testing Educative as an Authoring Platform

We wanted to see how our relatively *generic* computer science widgets would

work while porting a custom coded course. We did pretty well there but found

a couple of areas where we were seriously lacking. One thing that we realized early on was that multiple choice quizzes on Educative are primitive and so that's one area where we have room to improve.

How long did it take?

If you are curious, it took us about a week to port this course. Overall, it has been a good fun experience. Most of the time was spent in re-creating animations (of course).

How can I contribute?

We are committed to improving this. ***If you're interested in contributing or have any feedback/suggestion/criticism, drop us a note at*** *hello@educative.io*.

**An earlier version of this article didn't mention Prof. Devin Balkcom's name in the title. We appologize for that and have fixed the title.*

License