UNIT 19. Working with Arrays: Techniques

Summary
You will learn about new ways to use arrays when solving problems. Specifically:

- You will see two techniques for having arrays that are partially filled
- You will learn about searching, and two techniques for searching arrays to see if data is contained within it

Learning Objectives
After finishing this unit, you will be able to …

- Implement a count-as-you-go partially-filled array
- Implement an impossible-value partially-filled array
- Implement a linear search on an array
- Implement a binary search on an array

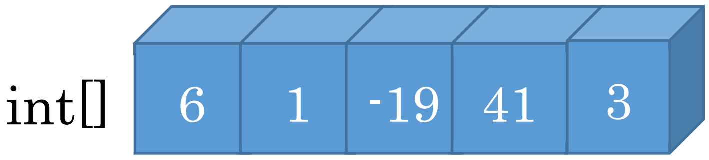
How to Proceed
- Read the unit content.
- Have a Processing window open while you read, to follow along with the examples.
- Do the sets of exercises in the **Check your Understanding** sections.
- Re-check the **Learning Objectives** once done.

19.1 **Introduction**

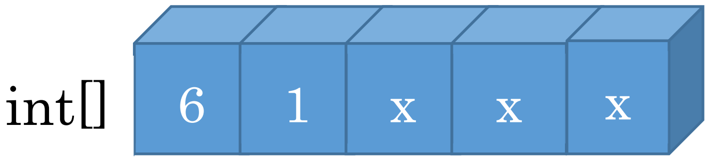
Until now in the course, each unit was specifically introducing a new programming structure, data type, technique, etc. However, you have already learned most of the basics of computer programming, and moving forward, more and more of your time will be spent on learning algorithms and data structures: ways to work with data using these tools. In this unit, we focus on learning some clever ways of working with data using arrays.

19.2 **Partially-Filled Arrays #1 – count as we go**

When we create an array, the default data is not usually that useful to us. Up until now, we have been fully populating our arrays with data before using them. That is, we set each bin to a desired value such as a position on the screen or a random value, and then start our program. For example, this is a fully populated, or fully-filled, array:

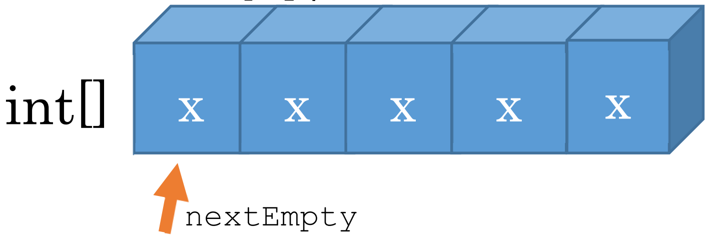


Sometimes, however, we want to have an array with data in some bins, but not in others. The most basic example of this is an array that fills up over time, but is not yet quite full:

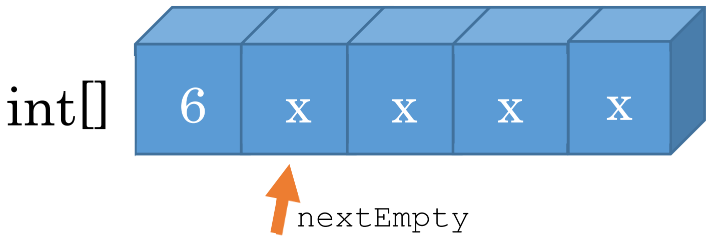


In this case, `x` marks an empty bin. The challenge in programming is to identify which bins have data, and which do not. Remember, with primitive data types there is no "no data" option (unlike `null` with objects and arrays). An integer must have a number, and with arrays, that number defaults to `0`.

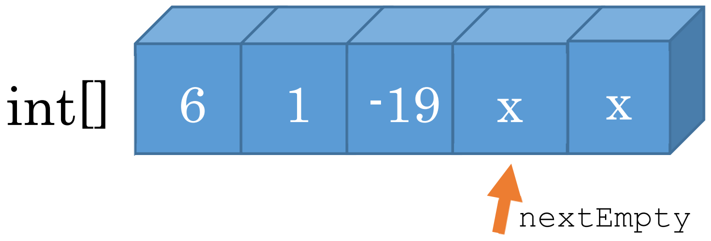
One technique is to always fill the array from the left, to the right, and use a separate integer variable to keep track of the next empty bin. At the beginning, the array is all empty, and our `nextEmpty` pointer is set to 0.



If we add a new value to the array, say a 6, then we put the data in the `nextEmpty` bin, and then move `nextEmpty` along:



And we keep going like this as we add data



We just need to be careful not to run off the edge of the array. When the array is full, `nextEmpty` will be equal to the array length. (it points to the first invalid bin number)

19.3 **Example: mouse path**

We will use an array to record a mouse movement as a series of points, and draw lines through the points. Unlike previous drawing examples, we store this in an array, which gives us power to modify or animate it. The array will start empty, and if we click, the current mouse positions get added to the array. It slowly fills up. If a keyboard key is pressed, we clear the path (empty the array).

We will need two partially-filled arrays – one for the `x` and one for the `y` coordinate. In my case, I will record up to 100 mouse positions.

```
final int HISTORY = 100;
int[] mousePathX = new int[HISTORY];
int[] mousePathY = new int[HISTORY];
```

In addition to the arrays to store the data, since they will be partially filled, we need a pointer to the next empty bin. At the beginning, the entire arrays are empty, so this points to bin 0:

```
int nextEmpty = 0;
```

Actually using these partially-filled arrays requires some changes. To add some data to them, we add the data at the next empty spot, and then move that along. In this case, if the mouse button is pressed, let's put the mouse X and Y into the next spot in the array, and move the pointer on to the next empty spot.

```
mousePathX[nextEmpty] = mouseX;
mousePathY[nextEmpty] = mouseY;
nextEmpty++;
```

In addition, before doing this, be sure to check that we are not running off the edge of the array. That is, only do this if `nextEmpty` is less than the array length. Otherwise, don't add anything.

Now, let's implement a function to draw. We have seen this before – it draws lines between the points. In this case, start from point 1, and draw a line back to the previous point, 0. Do this in a `for` loop, such that we are drawing from point `i` to `i-1`.

There is a catch here – how many lines do we draw? Previously, we just looked at the array size to setup our loop. However, here, we only want to include valid bins, so we need to know that number – we don't want to go over the entire array. In our case, the `nextEmpty` variable tells us this information. That is, if `nextEmpty` is 0, we have no data. If it's 5, we have 5 pieces of data (in bins 0...4). So, our draw function needs this information. Here, `clr` is a global variable that can change, initially set to white.

```
void drawPath(int[] x, int[] y, int bins)
{
  stroke(clr);
  for (int i = 1; i < bins; i++)
  {
    line(x[i-1], y[i-1], x[i], y[i]);
  }
}
```

When we call this in our draw block, we give `nextEmpty` as the number of bins.

Finally, we need a way to start over. Particularly given that our drawing stops if our arrays get full. How do you empty out these partially-filled arrays? All that needs to happen is for the `nextEmpty` pointer to be reset to 0, the beginning of the arrays. There is no need to empty the data out of the bins. When using the array, we use the pointer to determine how much is full. When putting data in the array, we use the pointer to determine where to put it. If a key is pressed, reset the array:

```
if (keyPressed)
{
  nextEmpty = 0;
}
```

That’s it! We used a partially filled array to hold a bit of mouse history. Ahead of time we don’t know how much we need to store so we make an array large enough, and only fill what we need. Here is my final code. I also added a function that the color used to draw changes if you press the keyboard, to illustrate that the entire path gets drawn every frame.

```
final int HISTORY = 100;
int[] mousePathX = new int[HISTORY];
int[] mousePathY = new int[HISTORY];
int nextEmpty = 0;
int clr = 255;

void setup()
{
  size(500, 500);
}

void drawPath(int[] x, int[]y, int bins)
{
  stroke(clr);
  for (int i = 1; i < bins; i++)
  {
    line(x[i-1], y[i-1], x[i], y[i]);
  }
}

void draw()
{
  background(0);
  if (mousePressed)
  {
    if (nextEmpty < mousePathX.length) {
      mousePathX[nextEmpty] = mouseX;
      mousePathY[nextEmpty] = mouseY;
      nextEmpty++;
    }
    clr = (int)random(128)+128;
  }
  if (keyPressed)
  {
    nextEmpty = 0;
  }
  drawPath(mousePathX, mousePathY, nextEmpty);
}
```

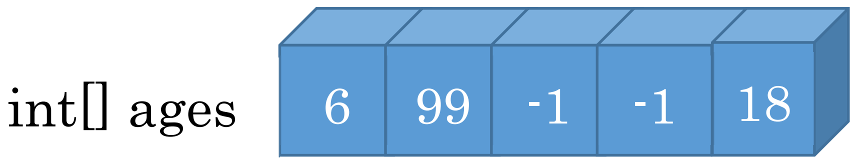
19.4 **Partially-Filled Arrays #2 – impossible values**

Sometimes, we cannot rely on an array filling up in a nice, neat order from 0 upward. What if we want to add data in the middle of the array? Or delete data in the array, leaving a hole with an empty bin? In this case, ***we can use a different technique: set the empty bins to an impossible value.***

What constitutes an impossible value? Well, it depends heavily on the context. If you are working with people's ages, `-1` is impossible. If you are working with car speeds (in km/h), maybe `1000000` is an impossible value. The general strategy is to name it as a final constant, and use that name throughout your program. In Processing, when dealing with basic canvas coordinates, `-1` is a nice impossible coordinate value, although this wouldn't work for the 3D examples we have seen.

Advanced: Another common strategy for impossible values is to use the largest or smallest possible value for your data type. When you have data that wants to use a very wide and unpredictable range, this is a good strategy as it leaves as much open room for data as possible. Most languages provide named constants for this to avoid errors. For example, in Java and Processing you have `Integer.MAX_VALUE;` and `Integer.MIN_VALUE;`. No matter what approach you use, always make sure to check for the impossible value, to avoid using it by accident resulting from a calculation. Another common approach is to use a *parallel* array – a second array, of Booleans, of exactly the same size. In this case, given data array `data` and boolean array `isEmpty`, then `data[i]` has data if `isEmpty[i]` is false.

This is how such a partially-filled array may look, e.g., for student ages:



Here, bins 2 and 3 are empty, with the other bins having data.

How would we ever end up in such a situation? That we have empty bins in the middle of an array? There are several ways. One is, what if we have an array of students, and one drops the course half way? Their bin becomes empty. What if each bin corresponds to a physical seat number in the room? Not all seats are filled, so some must be empty.

When working with these kinds of arrays, everything changes. When we want to add data to the array, we need to find an empty spot. If we want to print out the values, we have to skip over the empty bins. And so on. This takes more work, but is more flexible, as we can add or remove data at any point.

19.5 **Example: updated mouse path**

Let's change our previous mouse path to use our new partially-filled arrays technique. We will change from a path to drawing ellipses. Also, we will introduce the capability to delete points in the history – any ellipse under the mouse – to show how bins can be emptied and filled.

This will require a great deal of changes. Start by copy-pasting the previous example into processing. We will need to update everything to work with this new method. We no longer have a pointer to the next empty bin that we can rely on. After making a global for your `EMPTY` marker (set to `-1`, an impossible canvas coordinate), let's make a couple of helper functions. Let's make a function to clear an array – set all bins to empty – and do this at the beginning of the program to both the `x` and `y` arrays. (I renamed the `mousePath` to `mousePoints` for this example)

```
void setup()
{
  size(500, 500);
  clearArray(mousePointsX);
  clearArray(mousePointsY);
}

void clearArray(int[] a)
{
  for (int i = 0; i < a.length; i++)
    a[i] = EMPTY;
}
```

Now, we need a function that finds us the next empty bin so that we know where to add data. Previously we relied on a pointer that points to the end of the data in the array. Now, however, we need to go searching for one. Each time we want to store data in the array, we need to search out an empty bin, and use that one. It could be anywhere! We need a function to this. It takes an array, and tell us the index of the very first empty bin from left to right (from bin 0 up).

All that we have to do is to go through the array from the beginning until we find an empty bin. If none are empty, return `-1` (an impossible bin value). I gave this a global name `NOT_FOUND`.

```
int nextEmpty(int[] a)
{
  int next = NOT_FOUND;
  for (int i = 0; i < a.length && next == -1; i++)
  {
    if (a[i] == EMPTY)
      next = i;
  }
  return next;
}
```

Notice how in this example I modified the `for`-loop conditional to quit early if we actually found an empty bin.

Next, let's update our `drawPath` to now be `drawEllipses`. Previously, we only went up until our `nextEmpty` counter. With our new technique, we actually need to go through the entire array. The difference is, we need to make sure that each bin is not empty before we use it, simply with an `if` statement.

```
void drawEllipses(int[] x, int[] y)
{
  stroke(255);
  for (int i = 0; i < x.length; i++)
  {
    if (x[i] != EMPTY)
      ellipse(x[i], y[i], 10, 10);
  }
}
```

Now let's update the draw. First, change the logic from using the old `nextEmpty` pointer, to searching for an empty spot and using it. You still have to check if the array is full! We know that it is full if there is no empty bin available. Also, notice how we only need to check one of the arrays, the x in this case: both the `x` and `y` arrays should be matched perfectly, so an empty bin in one matches an empty bin in the other.

```
if (mousePressed)
  {
    int empty = nextEmpty(mousePointsX);
    if (empty != NOT_FOUND)
    {
      mousePointsX[empty] = mouseX;
      mousePointsY[empty] = mouseY;
    }
    clr = (int)random(128)+128;
  }
```

Now, to erase the whole array, instead of resetting the `nextEmpty` pointer, we now actually need to go through and set each bin to empty. Luckily, we have functions to do this for us already:

```
if (keyPressed)
{
  clearArray(mousePointsX);
  clearArray(mousePointsY);
}
```

Next, let's start implementing the functionality to delete ellipses. When the key is pressed, instead of starting over, let's erase any ellipses under the mouse. This poses two problems: we need to find which ellipses to delete (under the mouse), and two, we need to actually delete them from the array.

To find the ellipse, we need to go through the arrays of points, calculate the distance between the mouse and that point, and if the distance is less than the radius of the circle, it's a hit.

First make a function to return the distance between two points (we have done this before). Them, make another function `erasePoints` that goes through the array, and erases any point that is less than the ellipse radius away (under the mouse). To delete the point, all we have to do is to set the bin values to `EMPTY`. As with the other functions, at each bin, don't forget to check if it's empty!! If it is empty already, don't do the distance calculation.

```
float dist(int x, int y, int x2, int y2)
{
  float diffX = x2-x;
  float diffY = y2-y;
  return sqrt(diffX*diffX+diffY*diffY);
}

void erasePoints(int[] x, int [] y, int underX, int underY)
{
  for (int i = 0; i < x.length; i++)
    if (x[i] != EMPTY)
      if (dist(x[i], y[i], underX, underY) <= ELLIPSE_SIZE/2)
      {
        x[i] = EMPTY;
        y[i] = EMPTY;
      }
}
```

Make sure to update your `draw` to use this new function. Now, you can draw points by clicking the mouse, and erase by using the keyboard. Here is the final example:

```
final int HISTORY = 100;
int[] mousePointsX = new int[HISTORY];
int[] mousePointsY = new int[HISTORY];
final int EMPTY = -1;
final int NOT_FOUND = -1;
final int ELLIPSE_SIZE = 10;
int clr = 255;

void setup()
{
  size(500, 500);
  clearArray(mousePointsX);
  clearArray(mousePointsY);
}

void clearArray(int[] a)
{
  for (int i = 0; i < a.length; i++)
    a[i] = EMPTY;
}

int nextEmpty(int[] a)
{
  int next = NOT_FOUND;
  for (int i = 0; i < a.length && next == -1; i++)
  {
    if (a[i] == EMPTY)
      next = i;
  }
  return next;
}

void drawEllipses(int[] x, int[] y)
{
  stroke(255);
  for (int i = 0; i < x.length; i++)
  {
    if (x[i] != EMPTY)
      ellipse(x[i], y[i], ELLIPSE_SIZE, ELLIPSE_SIZE);
  }
}

float dist(int x, int y, int x2, int y2)
{
  float diffX = x2-x;
  float diffY = y2-y;
  return sqrt(diffX*diffX+diffY*diffY);
}

void erasePoints(int[] x, int []y, int underX, int underY)
{
  for (int i = 0; i < x.length; i++)
    if (x[i] != EMPTY)
      if (dist(x[i], y[i], underX, underY) <= ELLIPSE_SIZE/2)
      {
        x[i] = EMPTY;
        y[i] = EMPTY;
      }
}

void draw()
{
  background(0);
  if (mousePressed)
  {
    int empty = nextEmpty(mousePointsX);
    if (empty != NOT_FOUND)
    {
      mousePointsX[empty] = mouseX;
      mousePointsY[empty] = mouseY;
    }
    clr = (int)random(128)+128;
  }
  if (keyPressed)
  {
    erasePoints(mousePointsX, mousePointsY, mouseX, mouseY);
  }
  drawEllipses(mousePointsX, mousePointsY);
}
```

We looked at two different methods for partially-filled arrays. ***The second method is much more flexible, we are able to easily delete things anywhere and insert them anywhere in the array. However, it came at a cost – we had to do a lot more work.*** Previously, to clear the array, we just set the `nextEmpty` pointer to 0 (1 operation). Now, we need to set EVERY bin to zero (`n` operations, one per element in the array). There is also a lot more work to put an item into the array, since we need to find a bin first. In a later course, you will study this kind of analysis, regarding the **complexity** of your algorithm. There is not always a right answer, it is usually a balance of trade-offs – sometimes you use more memory to save time, or do more work to save memory, etc. As a computer scientist, your job will know which tools and techniques to use in a given situation.

19.6 **Example: Automatically Grow Array**

Let's extend the above example. One limitation of our program is that we quickly run out of array memory. One solution would be to simply make the array very large at the start – but what if we don't need it in the end? This is a waste of memory. Instead, let's make the array grow when it fills up.

How can you resize an array? Short answer – you can't! The only way to do it is to make a new array with double the size of the old one, and copy the old data over. Let's make a function called `doubleArray` which:

- Takes an array as a parameter
- Makes a new array with double the number of bins

- Copies the data from the old array to the new array
- Sets the additional bins to empty
- Returns the new array

With this function, if we run out of memory we can simply just double the array again. Make sure to try and implement this yourself. It gets tricky as you have two array boundaries – the old one, and the new one. Here is my code:

```
int[] doubleArray(int[] oldA)
{
  int[] newA = new int[oldA.length*2];
  for (int i = 0; i < oldA.length; i++)
    newA[i] = oldA[i];
  for (int i = oldA.length; i<newA.length; i++)
    newA[i] = EMPTY;
  return newA;
}
```

Update the draw code to use this now. We were already detecting if the array was full. Previously, in this case we just didn't add anything. Now, let's instead double the array so that we can always add the point. Make sure to double both the `x` and the `y` arrays.

```
  if (mousePressed)
  {
    int empty = nextEmpty(mousePointsX);
    if (empty == NOT_FOUND)
    {
      mousePointsX = doubleArray(mousePointsX);
      mousePointsY = doubleArray(mousePointsY);
      empty = nextEmpty(mousePointsX);
    }
    mousePointsX[empty] = mouseX;
    mousePointsY[empty] = mouseY;
    clr = (int)random(128)+128;
  }
```

Important! Notice how, after doubling the arrays, I called `nextEmpty` again? That's because we need to find an empty bin – `empty` was -1 so we need to find the next empty bin. Did we need to call the `nextEmpty` function again? There is a short cut to save work, do you see it?

19.7 **Searching Arrays: linear search**

In the previous example, we did quite a bit of searching on arrays. We had to search for an empty bin. We had to search for ellipses under the mouse. It turns out that searching is a very expensive operation. Imagine if we scale this up to 1 billion bins (not that unreasonable, e.g., in a high-end video game or searching on the web!). In this case, if we search for an empty bin, but there is no empty bin, we need 1 billion checks to confirm that. If the mouse doesn't touch any of the 1 billion ellipses, we need to check all 1 billion to be sure. That is slow!

There are a few ways to improve this. One way is to avoid searching – we can, for example, keep track of the next empty bin in the array cleverly so that we don't have to search for it. Another way is to be more clever with how we search.

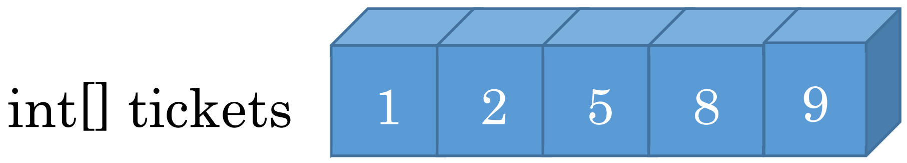
Searching is a fundamental problem of computer science, and if you stick with CS, you'll see it again and again. There are still great improvements being made to searching algorithms. Think about things like Google – Google is very fast at searching, it searches the entire web (!!!) in less than a second. This takes clever thinking, algorithms, and data structures (ways of storing data) to make it feasible.

Linear search – The simple, naïve approach to searching is called linear search. We just check every single element (like we did in the previous example). This is called linear because of you plot the search time against sample size (number of array bins, for example), you get a straight line. Twice the bins, twice the work.

You can imagine this as if you are searching a stack of randomly mixed exam papers. If I ask you to search a stack of papers to find the exam written by "Gumby", you have to search every one. If Gumby's exam isn't in the pile, you have to search *every single paper* to be 100% sure you didn't miss it.

Let's build an example to test out linear search. Let's make a list of one million lotto ticket holders. In this lottery, each ticket only has a single number, but they get large! We will start by generating 1 million ticket holders, each having a unique number. However, not all numbers are given out. That means that, when we draw a lotto number, we need to do a search to see if someone won or not.

One way to do this is to go through the array of people and hand out lotto numbers to them. We put a random space in between the numbers so that not all numbers are covered. For example:



Here, we have 5 people holding tickets, and each person has a unique number. However, as you can see, not all numbers are covered. There is no 3 or 4, 6 or 7.

We can make a function to hand out lotto numbers, and all we need is the array of people to put them into, and a global `RANDOM_SKIP` to determine how many we should skip at most. Be careful not to skip 0 as this would give us duplicate lotto numbers.

Also, it is useful to remember the largest ticket number so that we know what range of numbers we have lotto tickets in. Make the function return that number when it's done all the work.

```
final int RANDOM_SKIP = 3;

int handOutTickets(int[] people)
{
  int ticketNumber = 0;
  for (int i = 0; i < people.length; i++)
  {
    ticketNumber += (int)(random(RANDOM_SKIP)+1);
    people[i] = ticketNumber;
  }
  return ticketNumber;
}
```

This fills the array with random lotto ticket numbers. In `setup`, hand out those tickets, and `println` the array to be sure that it works as expected (no duplicates, etc.). I used a global array of tickets (1 per person), and a global `int` to keep track of the biggest ticket.

```
void setup()
{
  size(500, 500);
  biggestTicket = handOutTickets(tickets);
  println(tickets)
}
```

Once you confirm that this works, get rid of that `println`. A very large array can really wreak havoc on your computer if you try to print it.

Now that we have a data set, we can implement linear search. We can make a function called `linearSearch` which takes an array and an item, and returns the bin that the item is in. What should we do if it's not found? We can return -1 since it's an impossible value (make a final constant called `NOT_FOUND`). The algorithm is very simple:

- Assume that the item is *not* in the array – set the found index to `NOT_FOUND`
- Use a `for` loop to go through the entire array
  - if the data is found, then store the index
  - modify the `for` loop condition to quit early if the data is found
- `return` the index

```
int linearSearch(int[] data, int target)
{
  int index = NOT_FOUND;
  for (int i = 0; i < data.length && index == NOT_FOUND; i++)
  {
    if (data[i] == target)
      index = i;
  }
  return index;
}
```

Let's add a visualization to show which bins we actually looked in. We will draw a line across the screen to represent the range of tickets, with the smallest ticket on the left, and the largest on the right, and when an array bin is checked, we'll put a vertical line at the bin location to show that it's been checked.

First, use the following globals to define the line properties

```
final int CANVAS_SIZE = 500;
final int LINE_WIDTH = CANVAS_SIZE-100;
final int LINE_LEFT = CANVAS_SIZE/2 - LINE_WIDTH/2;
final int TICK_HEIGHT = 20;
```

and create two functions – one to draw the line itself at the specific `y` value:

```
void drawLine(int y)
{
  stroke(255);
  line(LINE_LEFT, y, LINE_WIDTH+LINE_LEFT, y);
}
```

And one to draw the tick mark. Since there will be more array bins than lines we need to scale it down. This function will need to know the bin number, and number of bins. This will also need to know the `y` value of the line.

```
void drawTick(int y, int bin, int binCount)
{
  float percent = bin/(float)(binCount-1);
  int x = LINE_LEFT + (int)(percent*LINE_WIDTH);
  line(x, y-TICK_HEIGHT/2, x, y+TICK_HEIGHT/2);
}
```

Finally, update your linear search to draw a tick mark at each bin checked. That is, inside the `for` loop, each time bin `i` is checked, draw a tick mark at bin `i`. (note: you will need to tell the function also the `y` coordinate of the visualization).

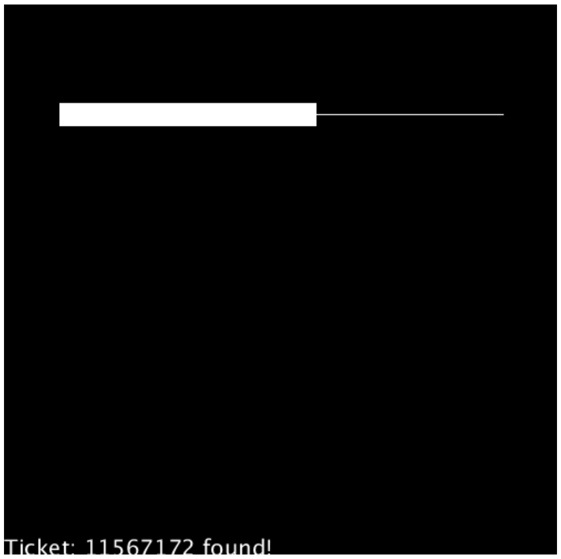
Now that we have these visualization tools, let's finish up the program. We already handed out the lotto tickets in the `setup`. Now let's pick a random number and check in the `draw` loop if anyone holds that ticket. We need a random ticket from `0..biggestTicket`, and then use linear search to check if there is a winner. It is also useful to toss out some text to give the info on the lotto draw:

```
void draw()
{
  background(0);
  int ticket = (int)random(biggestTicket+1);
  drawLine(100);
  int result = linearSearch(tickets, ticket, 100);
  String s = "Ticket: "+ticket+" ";
  if (result == NOT_FOUND)
    s += "Not found";
  else
    s += "found!";
  textSize(20);
  text(s, 0, height-1);
}
```

Try it, it should work now. What you will see, is that each time a ticket is drawn, it will put tick marks on every bin checked, tell you who (if anyone) won, and then repeat. At only 100 tickets, this is very fast. However, upgrade to 1 million tickets and try again.

What you will see now, is a lot more easy to see as it's slower. Wow this is slow. Sometimes, you'll notice that a ticket is found very quickly, and in this case, the bar of tick marks will be very small (since only a few checks had to be made). However, if the ticket was later in the array, or not found at all, it can take a long time – several seconds – to find that out.

Upgrade to 10 million people, and it gets 10 times slower.



One way to speed this up is to recognize that the tickets are in a sorted order! As you go along, if you pass the ticket number that you wanted, you can stop searching. This will help in many cases, but it will still be very slow. However, it is this recognition that the array has an order that leads to a really powerful speedup.

19.8 **Searching Arrays: binary search**

One of the fundamental approaches to searching is to make assumptions about our data. If we can have data with specific structure, then we can cleverly take advantage of that to save work.

For example, take a very large book, and search for page 237. Did you find it? How many pages were in the book, and, how many pages did you have to check? Did you have to check every single page? What if page 237 didn't exist (my kid tore it out!!). Do you have to check every page to be sure it doesn't exist? Of course not, you know that, in a book, the pages are in order, so you can be more clever. This is the essence of binary search.

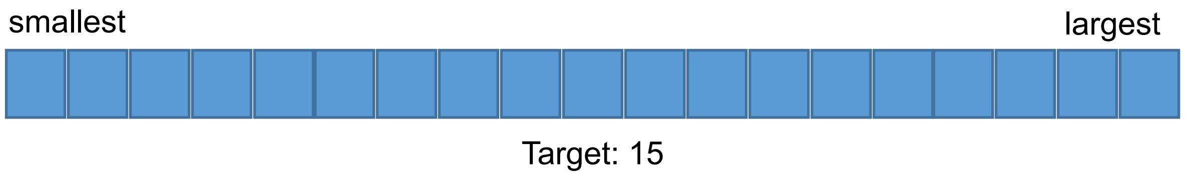
If we assume that an array of data is sorted, from smallest to largest, we can use binary search instead of linear search.

The key of how binary search makes things faster is that we end up ignoring whole sections of the array without checking them. If we can avoid checking them, we save work!

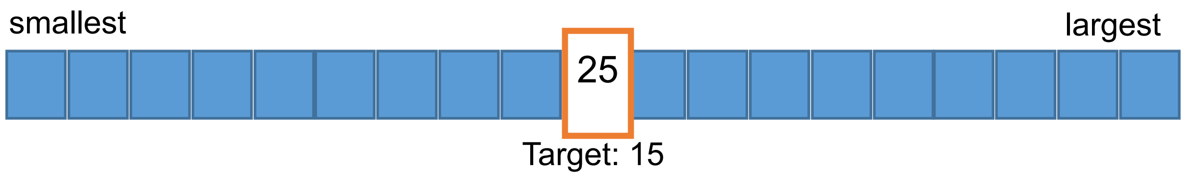
Here is the basic idea for binary search:

- We have an array of data, and a `target` we are searching for
- Assume the data in the array is sorted. If not, this won't work.
- Look at the array bin in the very middle of the array.
  - Is it our `target`? Yes? Woohoo! All done.
  - Is the data in the bin *smaller* than our `target`? Then let's immediately discount the left half of the array – no need to check those, as they are also smaller than our `target`.
  - Instead, is the data in the bin larger than our `target`? Then everything to the right is also larger, so we can discount those.
  - Repeat above. Calculate a new middle in the remaining bins

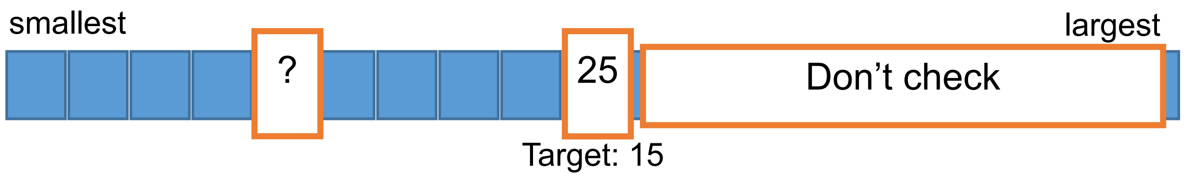
It is called binary search because, at each point, you make a binary decision – left or right. For example, here is an array, sorted. Say our `target` is 15.



First, we check the middle. The data happens to be 25:



So, since the array is sorted, we know that everything to right of 25, is either 25 or bigger. So all of those are bigger than 15, and we don't need to check them. Immediately we throw array half the away. We then repeat with a new middle of the smaller array:



Depending on this check, we'll throw away the left and right, halving our array again.

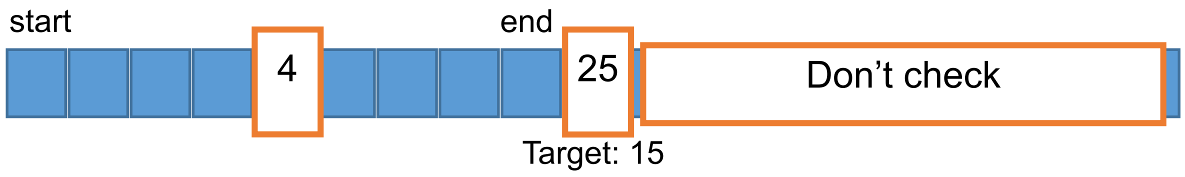
Here is the specific algorithm:

```
• Repeat:
    ➢ Set the start and end of your array. start = 0, end is length-1.
    ➢ midpoint is (start + end)/2
    ➢ Check midpoint against target
        ✧ If data > target, cut right half
            ● end = midpoint - 1
        ✧ if data < target, cut left half
            ● start = midpoint + 1
        ✧ if data = target, we're done!
```

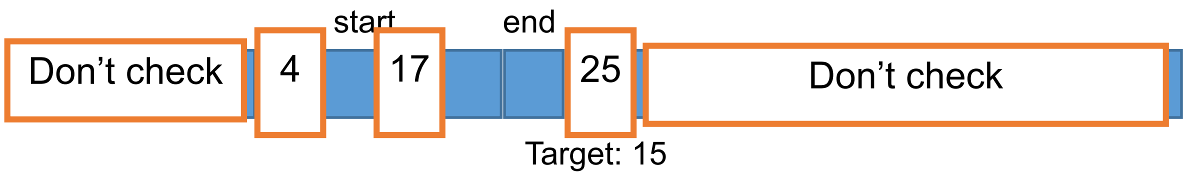
Notice that we always add or subtract 1 from the mid-point. This is important – we already checked the midpoint, so we should completely exclude it from the array.

When do we stop? We have two possibilities: a) we found the target, or b) there is no target. Knowing when we have a target is easy, but when do we know that there is no target?

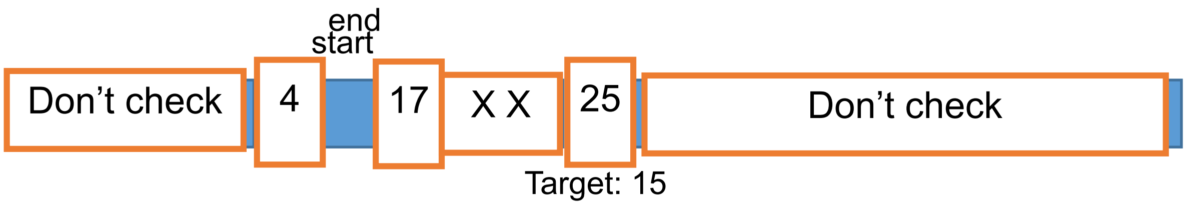
It happens like this. Eventually, `start` becomes bigger than `end`, which is illogical (`start` should always be smaller!). This happens because eventually the midpoint becomes `end` or `start` (or both!), and when we place the `start` or `end` past the midpoint, they leap frog. See, if we continue the previous example, moving start and end as we go:



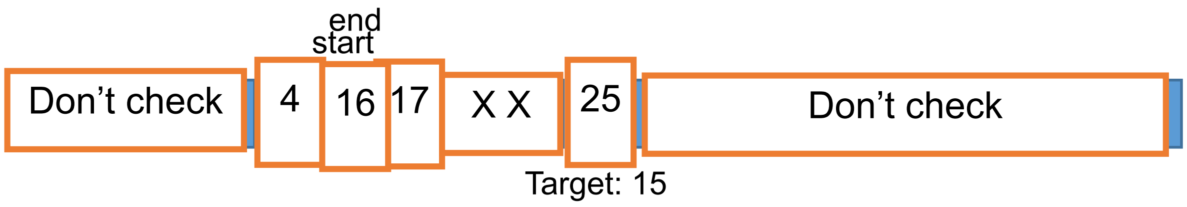
Since `midpoint` data was 4, which is smaller than the `target`, we discount the left, and move `start` one past the `midpoint`, and have a new `midpoint`



17 is bigger than `target`, so we move the `end` in. Now `end` and `start` are the same and we only have 1 bin left.



This is a 16, so we move the `end` to `midpoint - 1`, which is before start. Since `end < start`, we are done!



Notice how in an array of 19 elements, we only had to do 4 checks to determine that the item wasn't in the array! That is a lot less than linear search, where we would have had to check all 19.

Now, let's implement this in processing code, making a function called `binarySearch` that takes the array to search, the `target`, and also the `y` coordinate for plotting our progress.

First, we need the `start` and `end` variables, set to the first and last bins. We also need our `index`, assuming that our target is not found:

```
int start = 0;
int end = data.length-1;
int index = NOT_FOUND;
```

Since we don't know how may checks we need, we should use a `while` loop. We loop while `start` and `end` do not overlap, e.g., while `start <= end`, AND, while we haven't found the target yet. If either of these become true we should quit.

Each time through the loop, we calculate the `midpoint`, and look in the array at the `midpoint`. If the data is larger than the `target`, then we move the `end` point. If the data is smaller, we move the `start` point. Otherwise, we have the data and we should save the index!

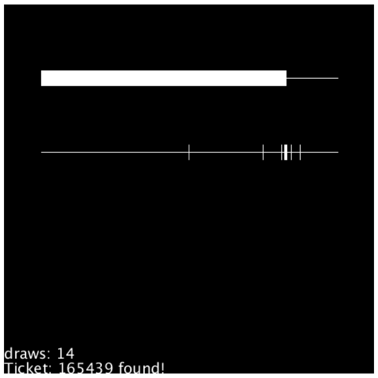
Don't forget to draw the tick marks at each bin looked at as in the previous example:

```
int binarySearch(int[] data, int target, int y)
{
  int start = 0;
  int end = data.length-1;
  int index = NOT_FOUND;
  while (start <= end && index == NOT_FOUND)
  {
    int midpoint = (start+end)/2;
    if (data[midpoint] > target)
      end = midpoint-1;
    else if (data[midpoint] < target)
      start = midpoint+1;
    else // data == target!
    index = midpoint;
    drawTick(y, midpoint, data.length);
  }
  return index;
}
```

That's it! Now we can use this instead of linear search in our `draw` loop, to see how fast it is. All that you need to do is to change the function call in `draw`. Try it starting with the 10,000,000 tickets we tried last time that really slowed down linear search.

How large can you make the search array and still keep it fast? Be careful of ticket number overflow.

Play with this. Try to add some Booleans to make it pause so you can see the result. Try comparing against linear search by plotting the binary search on a different line and running both in parallel.



```
draws: 14
Ticket: 165439 found!
```

You should see a significant time savings. Linear search becomes unreasonable soon after 10 million cases, but binary search should stay fast up until the maximum array size you can make. This is the power of algorithms, and the power of Computer Science!

19.9 **Searching Arrays: complexity**

It is very clear that binary search is faster than linear search. But how much faster is it? Doesn't it depend on the size of the dataset? Are there algorithms faster than binary search?

It turns out that in Computer Science, we have standardized measures and techniques for explaining these differences. If you stick with the program, there are whole courses on analyzing and comparing different algorithms. This may just seem mathy, but it is actually very important, as practicing computer scientists need to have the skills and knowledge at their fingertips to select and develop appropriate algorithms for your purposes.

One way to specify how fast an algorithm is, is to count the number of operations you need to perform in terms of `n` (the search space). We often look at best case, worst case, and expected case. For example, with linear search, the best case is 1 check – the first element is what we're looking for! The worst case is `n` checks – the item doesn't exist. It turns out that the expected time (average) is about `n/2`

With binary search, the best case is still 1 check (the target is right in the middle!). The worst case, however, is about $\log_2(n)$ (log base 2 n) – I'll leave it for a later class to learn how to calculate that. The expected average time is $\log_2(n)-1$ – just one check fewer than the worst case. The following table gives a breakdown:

algorithm	best case	worst case	average	10 elements	1000	1 bil.	1 septillion (24 0s)
linear	1 check	n	n/2	5	500	500mil	.5 septillion
binary	1 check	$\log_2(n)$	$\log_2(n)-1$	3	9	29	79

Wow! That's quite the difference! Linear search quickly explodes, while even at a septillion items to check, binary search only takes 79 checks.

Check your Understanding

19.10 Check Your Understanding: Exercises

Exercise 1. Use the count-as-you go partially-filled arrays technique to implement a kind of *undo* function in a drawing program. First, start with a simple drawing program as shown in the examples, using the count-as-you-go partially-filled arrays technique.

a. If a key is pressed in the keyboard, remove the *last* entry entered, so that it undoes the last drawing command.

Exercise 2. Use the impossible-value partially-filled arrays technique to implement a program that has random balls popping up on the screen. When a key is pressed, balls get erased from the left side of the screen, kind of like sweeping.

a. Make a program where, each draw, an ellipse is added and drawn at a random location. Use a partially filled array to store the ellipse location
b. If the array is full, grow it to double the size
c. If a key is pressed, erase the left-most ellipse. You can do this by finding the smallest `x` value, and erasing that ellipse. There may be several ellipses with the same value, just pick one. While the key is pressed, draw a horizontal like to simulate a broom sweep.

Exercise 3. Update Exercise 2 to use a parallel, boolean array, instead of an impossible value, to mark empty bins.

Exercise 4. Go back to Section 14.10, where you implemented a space shooter with a bad guy on the screen. Upgrade that example to have 10 bad guys, using the impossible-value partially-filled arrays technique.

Exercise 5. If binary search is much better than linear search, then why would you ever use linear search at all?

Exercise 6. The Sieve of Eratosthenes is a famous, classic method for finding prime numbers (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes). Implement the sieve to find all prime numbers up to 1,000,000. The idea is really simple, but implementing it can be a little tricky. The hard parts are that you need multiple, nested loops, and, you always need to be careful not to run off the edge of the array. Here is the algorithm

First, generate a list of numbers 1..n, where n is your maximum number (1 million, in this case). Then, starting with the number 2, *cross off* all multiples of 2 in the list. In programming, start by making an array of size `n` that has the numbers `1..n` stored in it. Then, for all `2p` while `2p < n`, (start at p or 2, 3, then up, etc.) set the array bin of `2p` to be empty.

This removes all multiples of 2 from the array, as they are clearly not prime numbers. Next, starting at 2, look in the array and find the next non-empty bin. In this case, it will be 3. Repeat the above process for removing all multiples of 3. Then, again find the next non-empty bin. 4 will be empty because it is a multiple of 2, but 5 will be in the array, so remove all multiples of 5. Be careful not to run off the edge of the array.

You stop when your search for the next non-empty bin runs off the edge of the array.

You will need several loops. Use while loops as it is non-trivial to calculate how many times you need to run the loop. Your main loop will run while you are still finding numbers to cross off multiples of, that is, while your next number to work on is less than the size of the array. Inside that loop, you need

- A `while` loop to go through the multiples of your current number, and cross them off. Start with the number 2: cross off `2p` for all `p` while `2p` is less than your maximum.
- Once that's done, a while loop to find the next number to cross off multiples of. E.g., you will find 3 the next time.
- Loop to the first step, removing all multiples of 3, then 5, etc.
- Once you are finished, the remaining numbers in the array are prime numbers. Make sure to exclude 0 if your solution used 0. Use a for loop to print out all the bins with numbers, they will be primes. Maybe start with a small size (100) first, as you can easily verify that they are primes.

Exercise 7. Make a drawing program that uses the count-as-you-go method for partially filled arrays, that lets the user draw a path, storing the points in two arrays, one for `x` and one for `y`. Make the arrays store up to 1000 points. In `draw`, draw lines through the points in order. You need a variable to remember where the next empty bin is. Be careful to check the end of the array.

a. If a key is pressed on the keyboard, delete every second point – do this by halving the size of the arrays and only keeping every second entry

b. Alternatively, if a key is pressed, double the size of the array, with new points being inserted in the middle of the existing two, as the average of both. For example, given `{1,2,3}` you would get `{1, 1.5, 2, 2.5, 3}`.

Ag Exercise 8. Make a drawing program that lets the user draw paths with the mouse, remembering the current path and the previous one drawn. They are drawn using different colors.

a. It has 2 sets of `x,y` arrays to store series' of points. One called `current`, one called `previous`
b. When the mouse is pressed, the mouse's location is added to the `current` array until the mouse button is released. If the `current` array gets full, stop adding.
c. The next time the mouse is pressed, the `current` points become the `previous` points, and we start with a fresh, clean `current`. You do not need to do any copy operation, just clever work with the variables.
d. The current path is drawn in 255 white, and the previous path is drawn in 128 grey.
e. Update the program so that when `current` gets full, it automatically

Ag Exercise 9. A limitation of binary search is that it only works in one dimension as we have learned it. What would it look like in two dimensions (e.g., pixels on a screen), three dimensions (objects in 3D space), or so forth? Look up, on Wikipedia, Quadtree and Octree.

Ag Exercise 10. Earlier in the course, we used nested `for` loops to draw a tic-tac-toe board. You now have all the pieces you need to implement a game. Try implementing a basic version of this game, using partially-filled arrays to mark a game square as either empty or having data. The logic to detect a winner is a little tricky!

Au Exercise 11. Do exercise 7 but instead with the impossible-value strategy for partially filled arrays. Part b in particular is quite tricky as you have to average a bin's neighbors.

How did you do?

Learning Objectives

How did you do? Go back to the beginning of the unit and check how you measure up to the learning objectives.