Chapter 6

Sorting

What is in This Chapter ?

Sorting is a fundamental problem-solving "tool" in computer science which can greatly affect an algorithm's efficiency. Sorting is discussed in this chapter as it pertains to the area of computer science. A few sorting strategies are discussed (i.e., bubble sort, selection sort, insertion sort and counting sort) and briefly compared. Finally, sorting is applied to the problem of simulating a fire spreading across a forest.

6.1 Sorting

In addition to searching lists, *sorting* is one of the most fundamental "tools" that a programmer can use to solve problems.

> ***Sorting*** *is the process of arranging items in some sequence and/or in different sets.*

In computer science, we are often presented with a list of data that needs to be sorted. For example, we may wish to sort a list of people. Naturally, we may imagine a list of people's names sorted by their last names. This is very common and is called a ***lexicographical*** (or *alphabetical*) sorting. The "way" in which we compare any two items for sorting is defined by the ***sort order***. There are many other "sort orders" to sort a list of people. Depending on the application, we would choose the most applicable sorting order:

- sort by **ID numbers**
- sort by **age**
- sort by **height**
- sort by **weight**
- sort by **birth date**
- etc..

A list of items is just one obvious example of where sorting is often used. However, there are many problems in computer science in which it is less obvious that sorting is required. However, sorting can be a necessary first step towards solving some problems efficiently. Once a set of items is sorted, problems usually become easier to solve. For example,

- **Phone books** are sorted by name so it makes it easier to find someone's number.

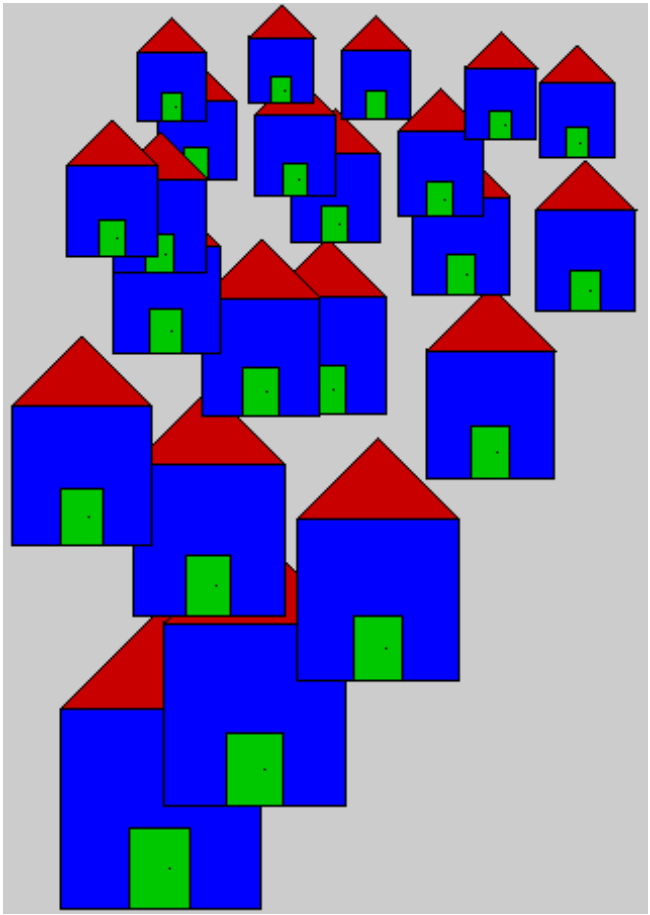
- DVDs are sorted by category at the **video store** (e.g., comedy, drama, new releases) so that we can easily find what we are looking for.

- A pile of many **trading cards** can be sorted in order to make it easier to find and remove the duplicates.

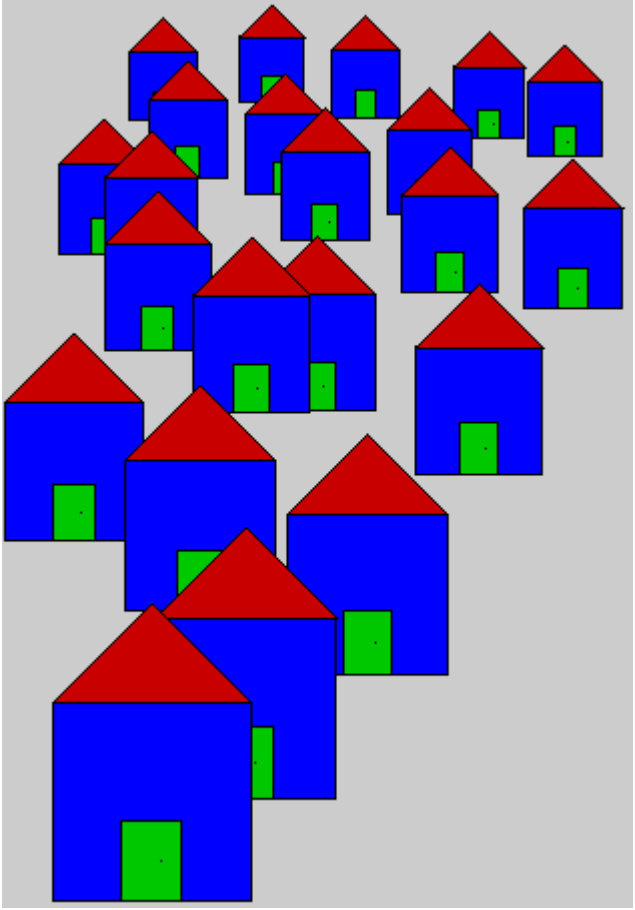
- Incoming **emails** are sorted by date so that we can read and respond to them in order of arrival.

- **Ballots** can be sorted so that we can determine easily who had the most votes.

Sorting is also an important tool in computer graphics. For example, scenes in a computer-generated image may draw various objects, but the order that the objects are drawn is important. Consider drawing houses as we did earlier in the course. Here is an example of drawing them in (a) the order in which they were added to the program, and (b) in sorted order from back to front:

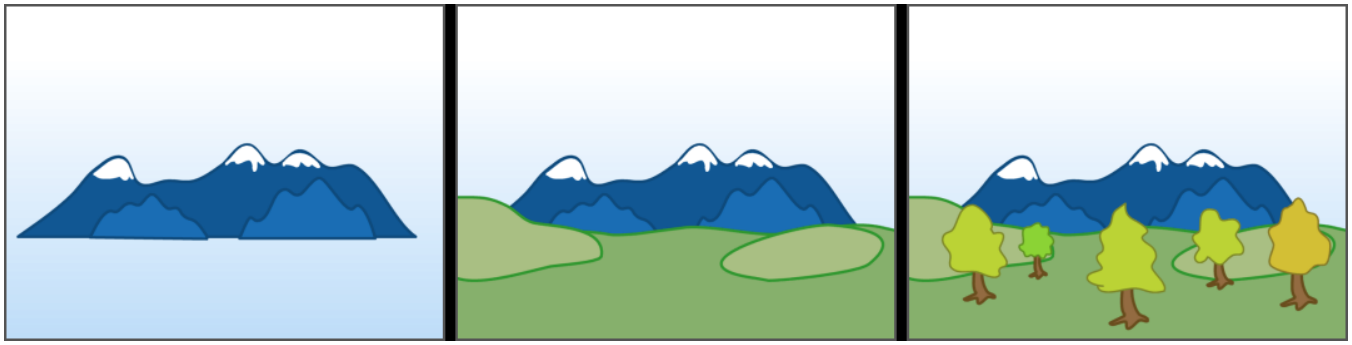


(a) normal order



(b) sorted order

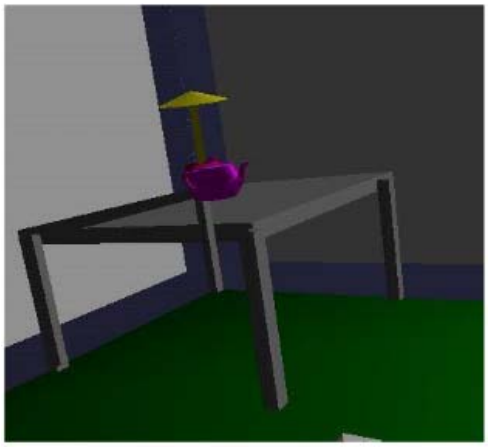
Notice how the houses in (a) are not drawn realistically in terms of proper perspective. In (b), however, the houses are displayed from back to front. That is, those with a smaller y-value (i.e., topmost houses) are displayed first. Thus, sorting the houses by their y-coordinate and then displaying them in that order will result in a proper image. This idea of drawing (or painting) from back to front is called the ***painter's algorithm*** in computer science. Painters do the same thing, as they paint background scenery before painting foreground objects:



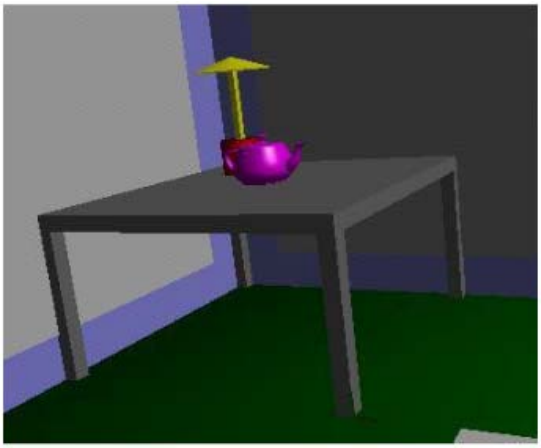
In 3D computer graphics (e.g., in a 3D game) most objects are made up of either triangular faces or quad surfaces joined together to represent 3D objects and surfaces. The triangles must be displayed in the correct order with respect to the current viewpoint. That is, far triangles need to be displayed before close ones. If this is not done correctly, some surfaces may be displayed out of order and the image will look wrong:

(a) wrong display order

(b) correct depth-sorted order



(a) wrong display order



(b) correct depth-sorted order

Thus, sorting all surfaces according to their depth (i.e., distance from the viewpoint) is necessary in order to properly render (i.e., display) a scene. The idea of displaying things in the correct order is commonly referred to as ***hidden surface removal***.

So, you can see that sorting is necessary in many applications in computer science.

Not only is sorting sometimes necessary to obtain correct results, but the ability to sort data efficiently is an important requirement for **optimizing** algorithms which may require data to be in sorted order to work correctly.

Therefore, if we can sort quickly, this can speed up search times and it can even allow our 3D games to run faster and more smoothly. Because sorting is such a fundamental tool in computer science which underlies many algorithms, many different "ways" of sorting have been developed ... each with their own advantages and disadvantages.

How many ways are there to sort ? Many.

For example, here is a table of just some types of sorting algorithms:

Sorting Style	Algorithms
Exchange Sorts	Bubble Sort, Cocktail Sort, Odd-even Sort, Comb Sort, Gnome Sort, QuickSort
Selection Sorts	Selection Sort, Heap Sort, Smooth Sort, Cartesian Tree Sort, Tournament Sort, Cycle Sort
Insertion Sorts	Insertion Sort, Shell Sort, Tree Sort, Library Sort
Merge Sorts	Merge Sort, Polyphase Merge Sort, Strand Sort
Non-Compairon Sorts	Bead Sort, Bucket Sort, Burst Sort, Counting Sort, Pigeonhole Sort, Proxmap Sort, Radix Sort

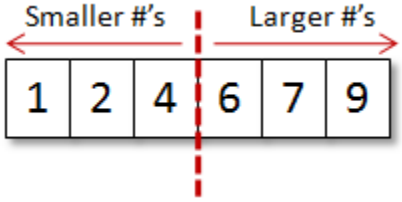
Why are there so many ways to sort ? These algorithms vary in their computational complexity. That is, for each algorithm, we can compute the (1) ***worst***, (2) ***average*** and (3) ***best*** case behavior in terms of how many times we need to compare two items from the list during the sort. Given a list of **n** items, a "good" sorting algorithm would require in the order of **n·log(n)** comparisons, while a "bad" sorting algorithm could require $\mathbf{n^2}$ or more comparisons.

It is also possible to compare algorithms in terms of:
1. how many times a pair of items in the list are swapped (i.e., change positions).
2. how much memory (or other computer resources) is required to complete the sort.
3. how simple the algorithm is to implement.

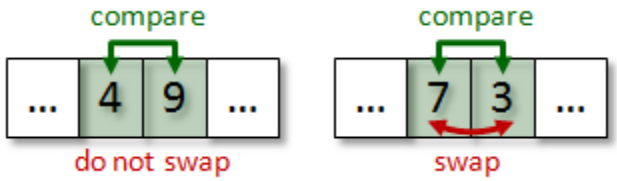
It is not the purpose of this course to make a comparison of a pile of sorting algorithms. However, it is good to get an idea as to how to write sorting routines in various ways so that you get a feel as to how various algorithms can be used to solve the same problem. We will examine a few of these now.

6.2 Bubble Sort

The **bubble sort** algorithm is easy to implement (i.e., it is easy to write the code for it). For this reason, many programmers use this strategy when they are in a hurry to write a sorting routine and when they are not worried about efficiency. The bubble sort algorithm has a bad worst-case complexity of around $\mathbf{n^2}$, so it is not an efficient algorithm when **n** becomes large.
Consider sorting some integers in increasing order. The idea behind any sorting algorithm is to make sure that the small numbers are in the first half of the list, while the larger numbers are in the second half of the list:



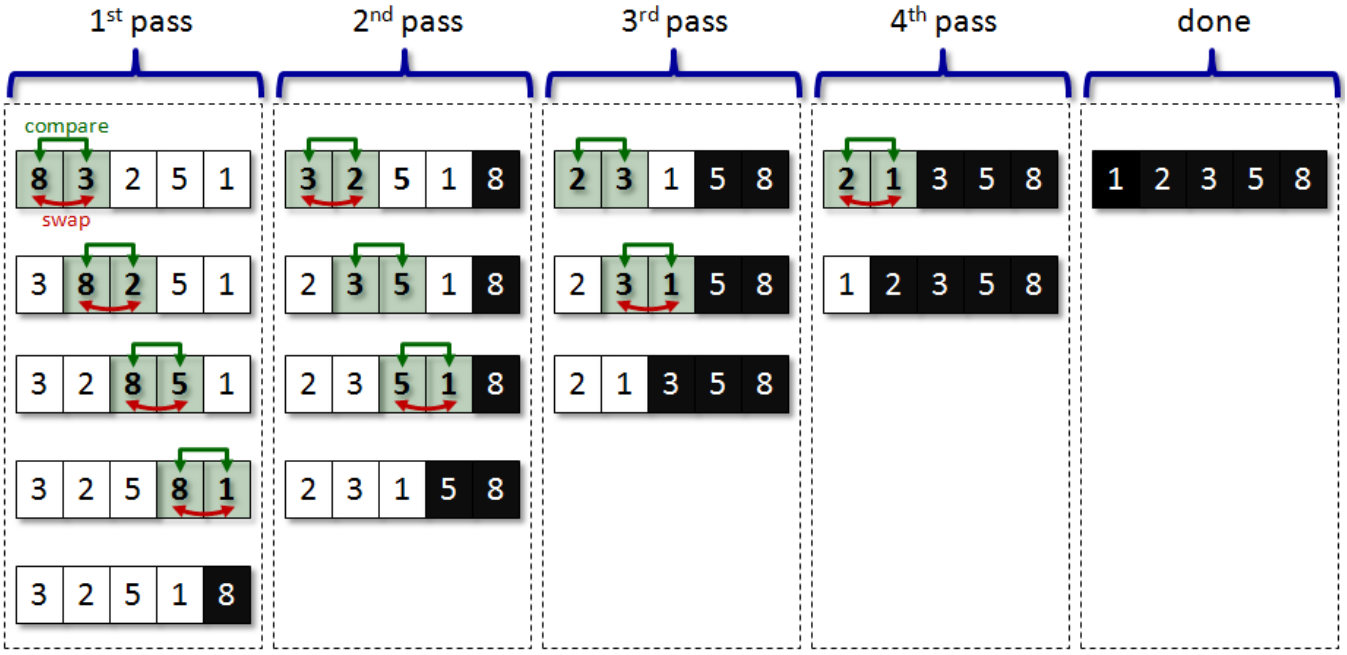
Regarding this "split" of numbers in the final sorted list, we can actually draw conclusions as to relative positions of adjacent numbers in the list as we are trying to sort. For example, assume that we examine any two adjacent numbers in the list. If the number on the left is *smaller* than the one on the right (e.g., 4 and 9 in the picture below), then relatively, these two numbers are in the correct order with respects to the final sorted outcome. However, if the number on the left is *larger* than the one on the right (e.g., 7 and 3 in the picture below), then these two numbers are out of order and somehow need to "swap" positions to be in correct order:



So, by generalizing this swapping principle to ensure "proper" ordering, we can take pairs of adjacent items in the list and swap them repeatedly until there are no more out of order. However, we need a systematic way of doing this.

The bubble sort approach is to imagine the items in a list as having a kind of "weight" in that "heavy" items sink to the bottom, while "lighter" items float (or ***bubble up***) to the top of the list. The algorithm performs the "bubbling-up" of light items by swapping pairs of adjacent items in the list such that the lighter one is ensured to be above the heavier item. It does this by making multiple passes (i.e., multiple iterations or "rounds") through the list, each time moving (or sinking) the heaviest item towards its final position at the end (or bottom) of the list.

Here is an example of how the algorithm works on a list of 5 integers:



As can be seen, during the first pass through the data, the heaviest item is moved (i.e., sinks) to the end of the list because all numbers are smaller (i.e., lighter) than it, so they "bubble-up" higher towards the top of the list. At the end of pass one, we are ensured that the largest item is at the end of the list.

During the second pass, comparisons of adjacent items are made the same way and eventually the next largest item sinks down toward the bottom. Notice that there is no need to compare the 5 with 8 at the end of the 2nd pass since the 1st pass ensured that 8 was the largest item. So the 2nd pass takes one less step to complete. The subsequent passes continue in the same manner. Once 4 passes have been completed, the list is guaranteed to be sorted.

Notice that the algorithm requires 10 comparisons of adjacent items. This is exactly twice the list size. However, as the number of items in the list grows, it is easy to see that the algorithm requires this many comparisons:

(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 ... which is: **n·(n-1)/2** comparisons.

This is a little slow, but the algorithm is simple. Can you write the code for this algorithm ? Hopefully, you can easily see the need for nested loops, as the outer loop will cover the number of passes, while the inner loop will handle the comparisons during a single pass.

Here is a straight forward implementation:

```
Algorithm: BubbleSort1
        items:          the array containing the items to sort

1.      repeat items.length-1 times {
2.          for each location i from 0 to items.length-2 {
3.              if (items[i] > items[i +1]) {
4.                  temp ← items[i +1]
5.                  items[i +1] ← items[i]
6.                  items[i] ← temp
                }
            }
        }
```

This implementation, however, always requires **(n-1)·(n-2)** comparisons. We forgot to adjust the code to eliminate one less comparison in the list each time, since each pass ensures that one more item is in its final position. To do this, we need to adjust the count for the inner loop to reflect the pass number that we are making. Here is the adjusted code:

```
Algorithm: BubbleSort2
        items:          the array containing the items to sort

1.      for each pass p from items.length-1 down to 0 {
2.          for each location i from 0 to p-1 {
3.              if (items[i] > items[i +1]) {
4.                  temp ← items[i +1]
5.                  items[i +1] ← items[i]
6.                  items[i] ← temp
                }
            }
        }
```

One more thing .... what if the list suddenly becomes sorted during the middle of the algorithm (i.e., all numbers fall into place) ? Even worse ... what if the list is *already* sorted ? We can add something to the algorithm to cause it to quit when the list is sorted. How do we know when the list is sorted ? Well ... if we go through an entire pass and did not make any swappings, then all integers must be in their correct position ... do you agree ? So we could just check to ensure that a swap was made during a pass ... and if not ... then quit:

```
Algorithm: BubbleSort3
        items:          the array containing the items to sort

1.      for each pass p from items.length-1 down to 0 {
2.              madeSwap ← false
3.              for each location i from 0 to p-1 {
4.                      if (items[i] > items[i +1]) then {
5.                              temp ← items[i +1]
6.                              items[i +1] ← items[i]
7.                              items[i] ← temp
8.                              madeSwap ← true
                        }
                }
9.              if (madeSwap is false) then
                        quit()
        }
```

6.3 Selection Sort

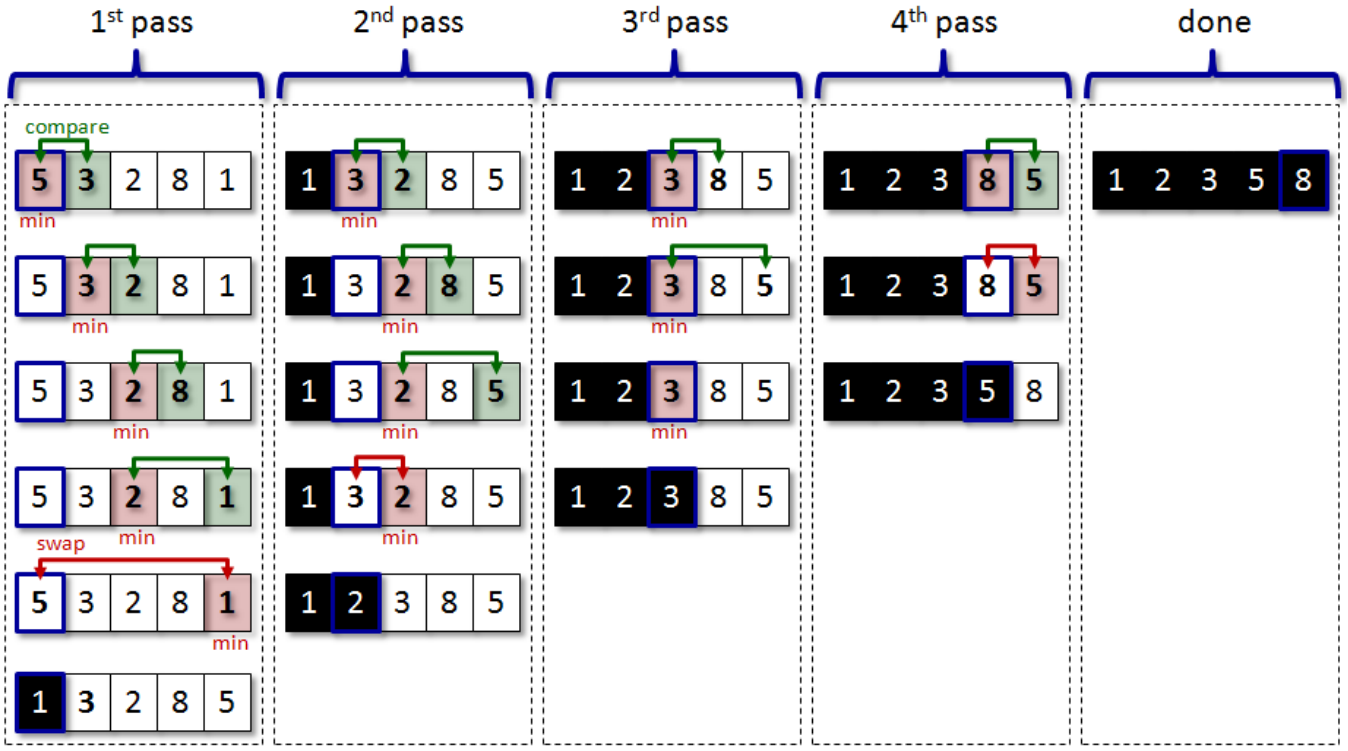
The **selection sort** algorithm is a natural kind of sorting technique. It is also easy to implement but like the Bubble Sort, it also has a bad worst-case complexity of around n^2, so it is not an efficient algorithm when **n** becomes large.

The idea behind the algorithm is simple. It is similar to the idea of stacking a set of blocks in a tower. Find the largest block, place it at the bottom. Then find the next largest and place it on top of it, then the next largest ... and so on ... with the smallest block being placed at the top.

When the data is in a list or an array, the algorithm is slightly more complicated because we need to ensure that each item is always stored somewhere in the array. So, when we find the largest item that needs to go at the end of the array... we need to swap its position with the last item in the array. So this idea of swapping positions is necessary. The algorithm itself is usually described as finding the minimum and placing it at the front of the array ... which is opposite to the block-stacking example just mentioned, but nevertheless produces the same sorted result.

The selection sort approach is to always keep track of (i.e., hold on to) the smallest item as you go through the list. As the algorithm iterates through the items in the list it always compares against the smallest item being held onto and if a smaller item comes along, it them becomes the smallest item. Once the list has been checked, we then move the smallest item to the front of the list and do another round starting with the second item.

Here is an example of how the algorithm works on a list of 5 integers:



Notice that the algorithm again requires 10 comparisons of adjacent items as well as 3 swaps. As with the bubble sort, it is easy to see that the selection sort may require **n·(n-1)/2** comparisons. However, much less swaps are made with the selection sort. The selection sort may require **(n-1)** swaps, while the bubble sort can require up to **n·(n-1)/2** (e.g., when the list is in reverse order). You can see therefore, that the selection sort is a little more efficient.

The code for this algorithm is quite similar to that of the bubble sort in that it has nested loops and compares items. However, this time, we compare each item against the minimum, not against its adjacent neighbor. Also, the swap occurs outside the inner loop... not within the inner loop as with the bubble sort.

Here is a straight-forward implementation:

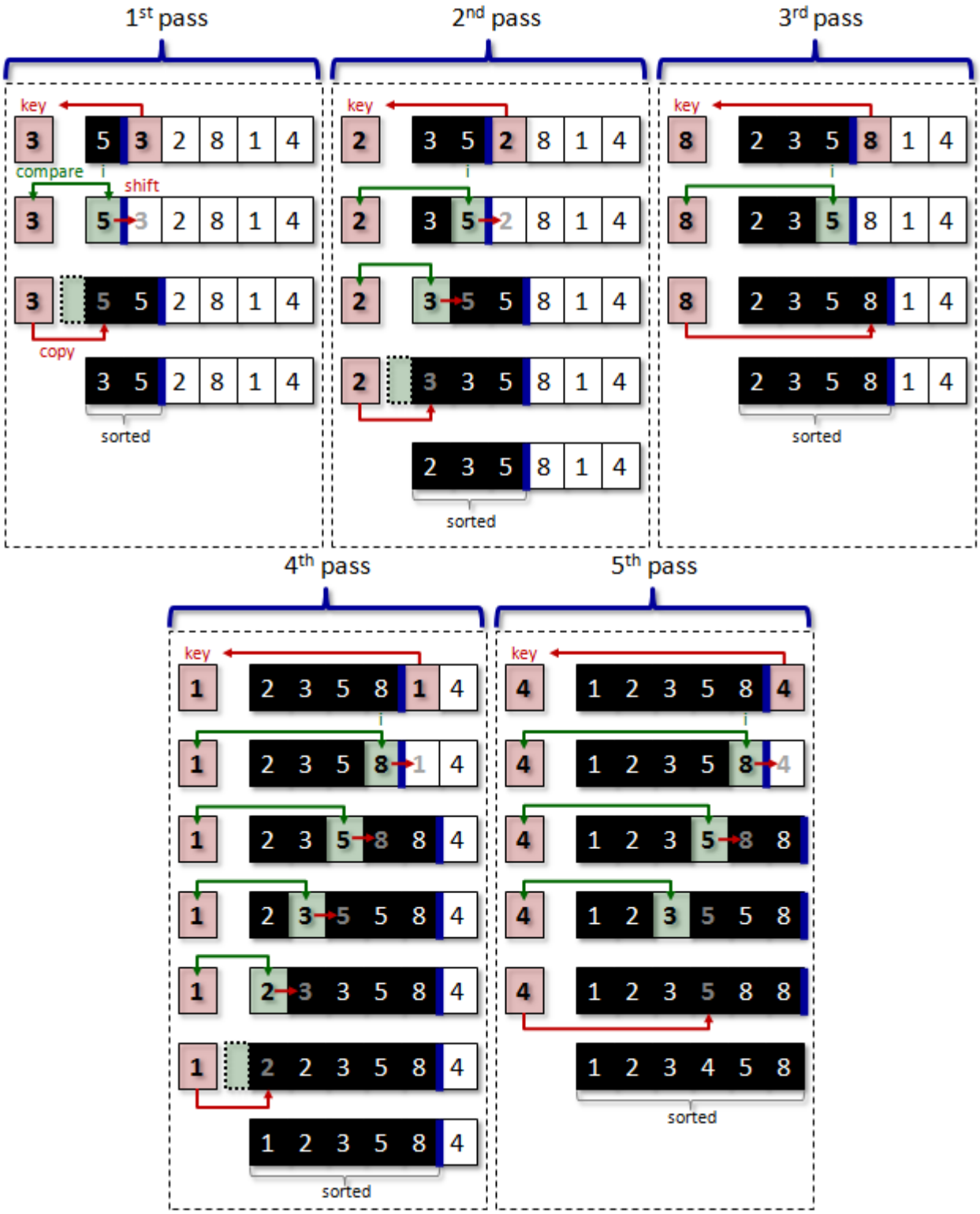
```
Algorithm: SelectionSort
        items:          the array containing the items to sort

1.      for each pass p from 0 to items.length-1 {
2.              minIndex ← p
3.              for each index i from p+1 to items.length-1 {
4.                      if (items[i] < items[minIndex]) then
5.                              minIndex ← i
                }
6.              temp ← items[p]
7.              items[p] ← items[minIndex]
8.              items[minIndex] ← temp
        }
```

6.4 Insertion Sort

The **Insertion Sort** algorithm is perhaps the most natural sorting algorithm. It is very much like the Selection Sort in that it attempts to sort by selecting one item at a time. Even though it has a bad worst-case complexity of around n^2, in general it is faster than the Selection Sort because it does not search the whole list to look for the smallest item first.

The idea behind the algorithm is simple and relates to a real-life kind of sort that we would naturally perform. Imagine on the table a "pile" of unsorted items. The idea is to repeatedly select items from the unsorted pile and place them in a newly sorted pile. So, the algorithm maintains a portion of items that are sorted (i.e., the ones at the front of the list) and a portion of items that still remain to be sorted (i.e., the ones at the back of the list). Each time an item is selected, the "sorted portion" grows by one, while the "unsorted pile" shrinks by one. After doing this **n** times, the whole list is sorted. It is similar to the idea of a librarian placing books on a shelf with "already-sorted" books.

Here is an example of how the algorithm works on a list of 6 integers:



Notice how the left side of the list always contains items in sorted order, although additional items still need to be inserted in there, so that sorted list is not complete until the last pass. Notice in all except the 3rd pass that there was a need to search backwards through the sorted portion in order to find the correct place to insert the **key** item.

Here is a straight-forward implementation. Notice the use of a **while** loop in order to allow the loop to exit quickly as soon as the **key** item is larger than the ones remaining in the sorted portion of the list:

```
Algorithm: InsertionSort
        items:          the array containing the items to sort

1.      for each pass p from 1 to items.length-1 {
2.              key ← items[p]
3.              i ← p - 1
4.              while (i >= 0) AND (items[i] > key) {
5.                      items[i +1] ← items[i]
6.                      i ← i - 1
                }
7.              items[i +1] ← key
        }
```

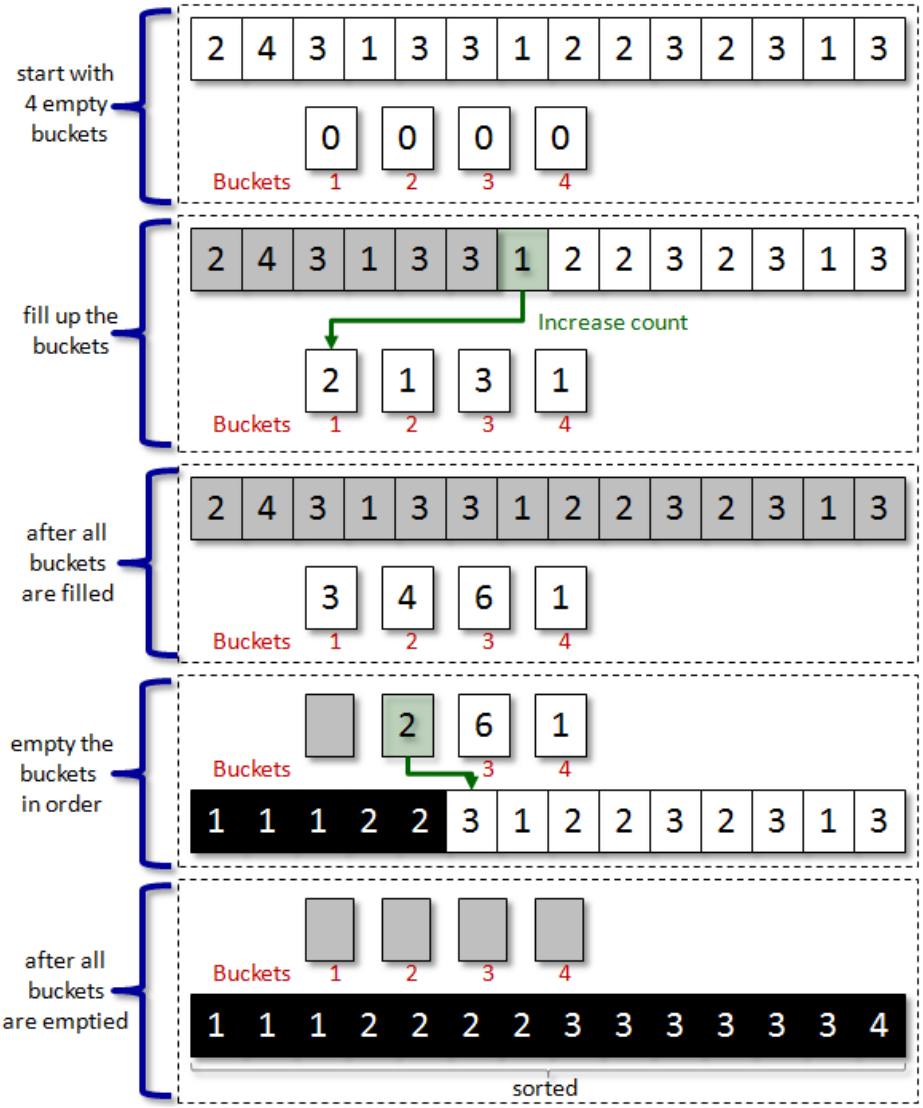
With a careful look at the code, you can see that the algorithm may need to make **n·(n-1)/2** comparisons as with the Selection Sort. Also, there may be a need to make this many swaps as well. However, in a typical scenario with an initial random arrangement of numbers, the Insertion Sort takes about half the speed of a Selection Sort ... due to the ability of the **while** loop to exit earlier. So, in general, the Insertion Sort is a little more efficient.

6.5 Bucket Sort & Counting Sort

The **Bucket Sort** algorithm is an excellent sorting algorithm for the special case in which there are many duplicate items and the items are bounded by some small maximum size. It is a simple algorithm and can run in worst-case complexity of **2·n** time (under certain situations) !

The idea of the algorithm is similar to that of what you may find at the post office. Incoming mail is quickly placed into "roughly sorted" bins (or buckets). Each item in a particular bin is "equal" in some sense of the word (e.g., same destination city, same postal code, same street, etc..). So each bin represents a partially-sorted list. Each bin can then be sorted separately, in any manner. A special case arises when multiple items are considered equal. That is, consider 1,000 student exam papers with integer grades ranging from 0% to 100% that need to be sorted by grade. You can sort them by making 101 bins representing the grades and then placing each exam in the corresponding bin according to the grade.

In the case where we actually have a set of fixed-range integers that we need to store, the algorithm becomes what is known as a **Counting Sort** and is quite simple. Consider an array with 14 numbers as show below. The array is assumed to contain integers from 1 to 4. We can make 4 buckets in the form of integer counters and then simply fill up the corresponding bucket counter as we iterate through the numbers. Then, to get the sorted list, we just empty the buckets in order →



This is very simple and it only takes **2n** steps. Here is a straight-forward implementation:

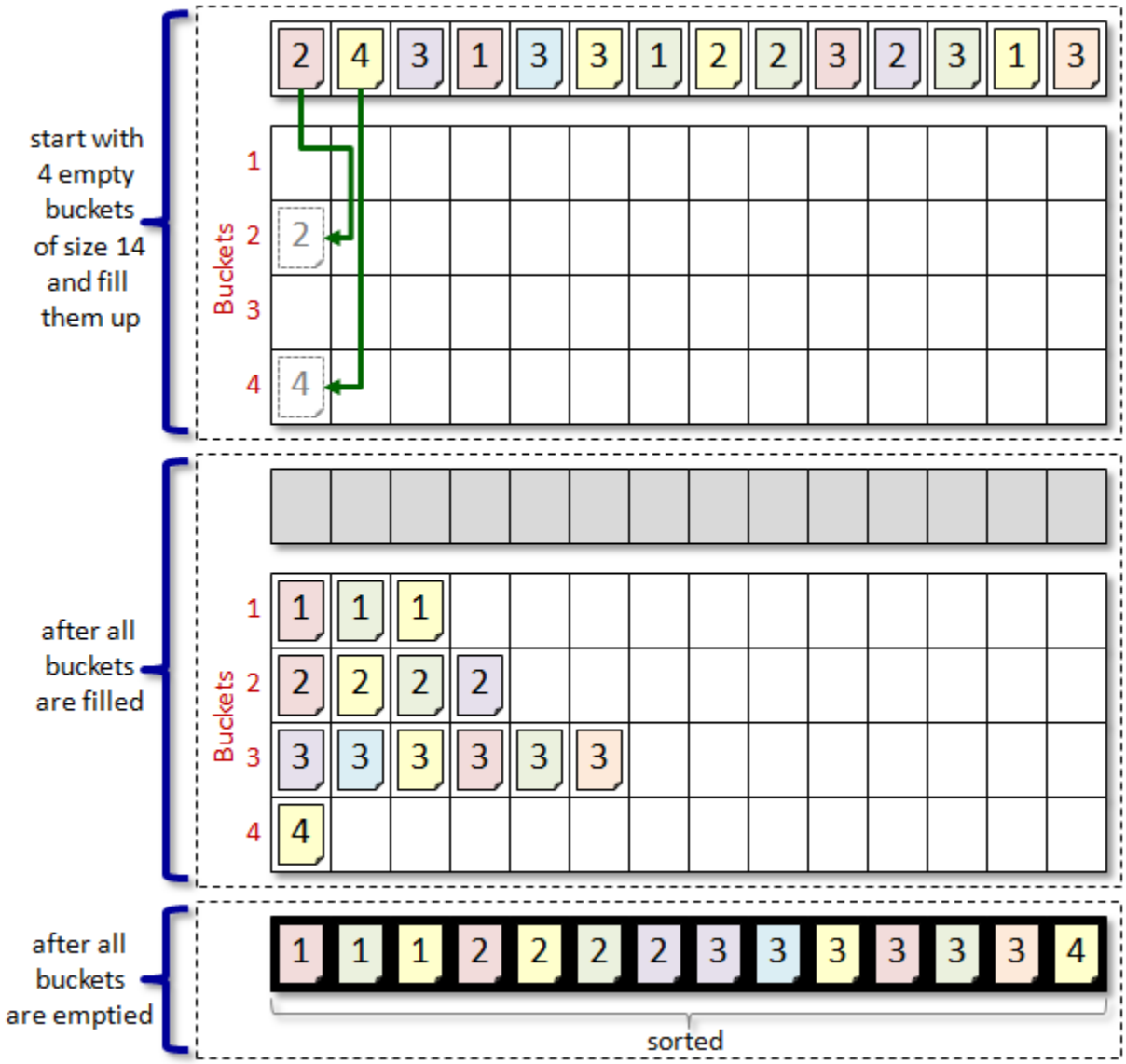
```
Algorithm: BucketSort1
        items:      the array containing the items to sort
        b:          the number of bins to use

1.      bins ← new array of size b each element set to 0

2.      for each pass i from 0 to items.length-1 {
3.          bins[items[i]-1] ← bins[items[i]-1] + 1
        }

4.      i ← 0
5.      for each bin x from 0 to b-1 {
6.          for each count c from 0 to bins[x] {
7.              items[i] ← x + 1
8.              i ← i + 1
            }
        }
```

Notice in steps 2 and 3 how the bins array simply stores a count of how many items had that bin's value. Then in steps 4 through 8 we simply go through each bin and fill in the items array with the correct number of items from the bins.

A slightly more complex situation arises when we actually need to store the items themselves in the bucket (i.e., not just counters). In our example with the exam papers, we need to store the exam papers themselves, not just the grades.

To accomplish this, we need to store more than a counter in each bin. In fact, we need to reserve space for the items themselves. How many items may fit into a bin ? Well, if all the items are equal, they will end up in the same bin!! Therefore, even though it is likely that a typical random set of items will be distributed evenly among the bins, it is possible that some bins may get very full. So, to be safe, we would need to make each bin large enough to hold all the items.

Therefore, our bins array in the above code would need to be a two-dimensional array of size **b·n** so that each of the **b** buckets can store up to **n** items. Here is what the algorithm will do:



Now, in reality, only **n** items are being stored in the buckets. Therefore **(b-1)·n** spaces in the 2D array will remain empty. This is a little bit wasteful.

In our exam paper example, the 2D array would need to have space to store 101·1000 = 101,000 exam papers, although only 1,000 exam papers would actually be stored!!! That is about 99% of wasted space! Of course, there are ways to fix this, as you will learn in your 2nd year here in computer science. We can, for example, make initially small buckets based on "estimates" as to how many items we expect to fall into any given bucket ... and then *grow* the buckets as they become full. We will not discuss this further here. However, be aware that there is often a trade-off between runtime complexity and storage space.

Here is the adjusted code to handle the storage of items instead of counters:

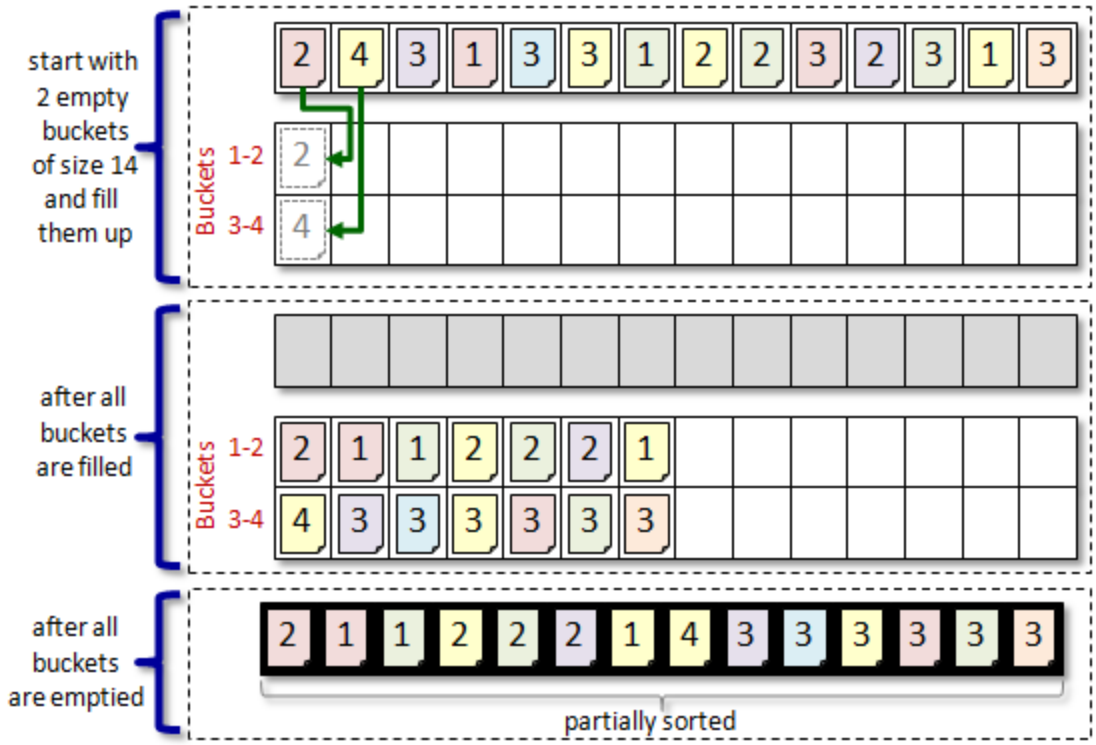
```
Algorithm: BucketSort2
        items:          the array containing the items to sort
        b:              the number of bins to use

1.      bins ← new array of b empty arrays each of size items.length
2.      binCount ← new array of b counters initially set to 0

3.      for each pass i from 0 to items.length-1 {
4.          binID ← getBinFor(items[i])
5.          bins[binID][binCount[binID]] ← items[i]
6.          binCount[binID]← binCount[binID] + 1
        }
7.      i ← 0
8.      for each bin binID from 0 to b-1 {
9.          for each item c from 0 to binCount[binID]  {
10.             items[i] ← bins[binID][c]
11.             i ← i + 1
            }
        }
```

The above code assumes that the order of any two exam papers with the same grade is arbitrary/unimportant and thus do not need to be re-arranged in any way.

However, the algorithm can be generalized by allowing less bins. For example, considering our exam paper example, if we use only 10 bins instead of 101, this would significantly reduce the storage requirements.

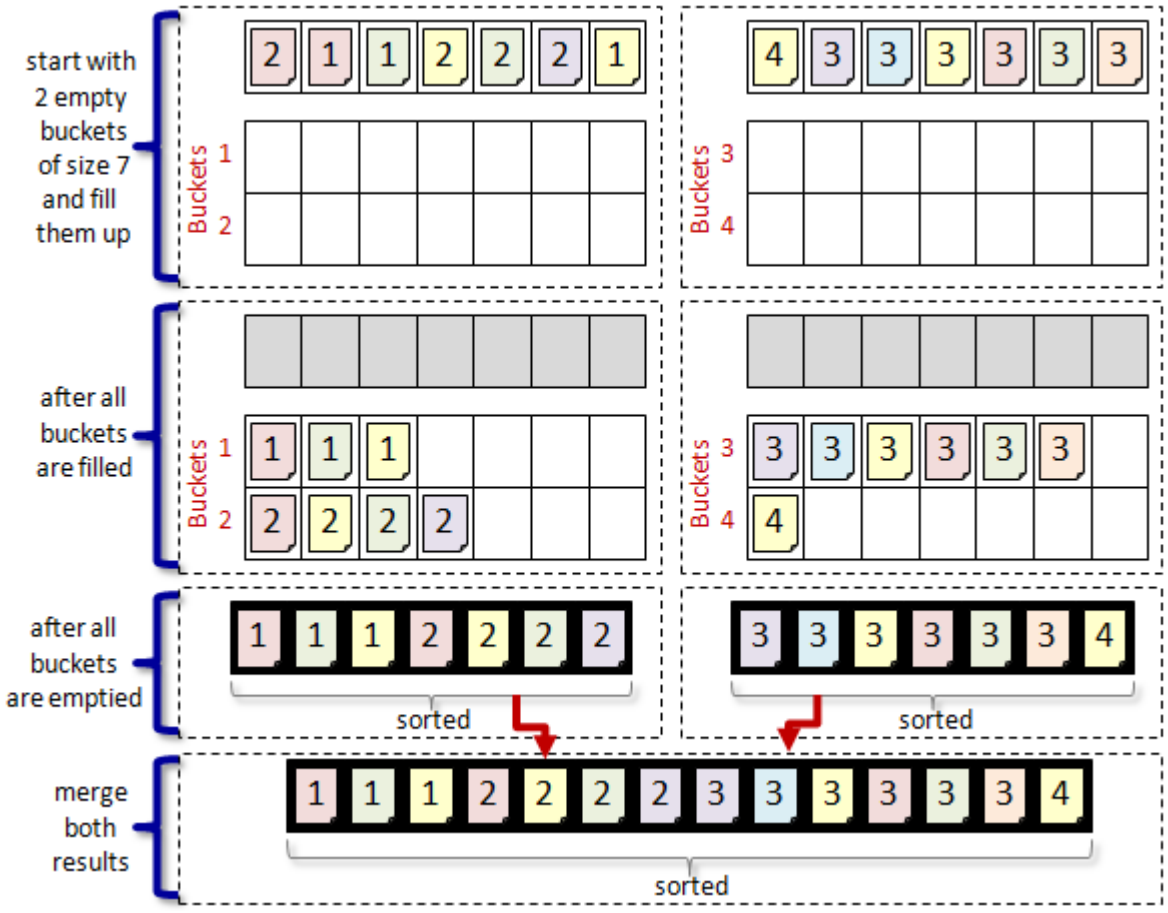
We could, for example, put all grades in the 70%-79% range into one bin, all grades in the 80%-89% range into another bin, etc..

The result is that the array would be partially sorted, but not complete. Here is our example again but with 2 buckets instead of 4:



Notice that there is much less wasted space, but the end-result is that the array is not sorted. A solution would be to sort each bucket before filling up the array again. Depending on the size, we could use any sorting technique to sort the buckets.

We could, for example, use a bucket sort again on the two buckets:



The actual merging of the sorted buckets is easy again, as we simply use the same code as before. The speed of the algorithm will depend on the kind of sorting technique that we use.

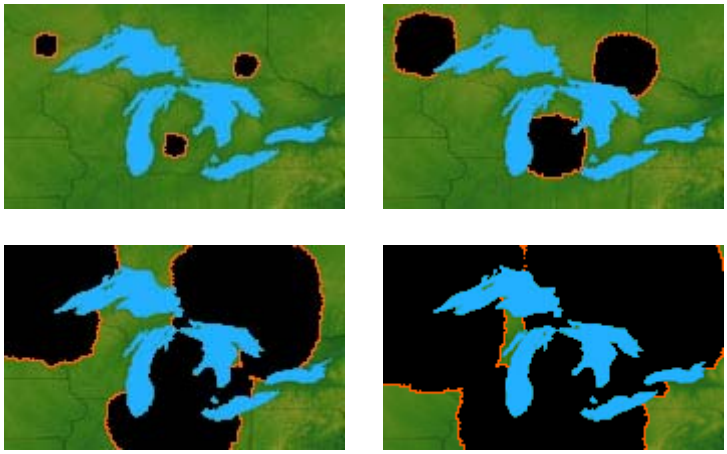
Here is the code for the general **Bucket Sort** where the buckets need to be sorted:

```
Algorithm: BucketSort3
        items:          the array containing the items to sort
        b:              the number of bins to use

1.      bins ← new array of b empty arrays each of size items.length
2.      binCount ← new array of b counters initially set to 0

3.      for each pass i from 0 to items.length-1 {
4.              binID ← getBinFor(items[i])
5.              bins[binID][binCount[binID]] ← items[i]
6.              binCount[binID]← binCount[binID] + 1
        }
7.      for each bin binID from 0 to b-1 {
8.              sort(bins[binID])       // use any algorithm...will affect runtime though
        }
9.      i ← 0
10.     for each bin binID from 0 to b-1 {
11.             for each item c from 0 to binCount[binID]  {
12.                     items[i] ← bins[binID][c]
13.                     i ← i + 1
                }
        }
```

In summary, with a large number of items and enough bin space (i.e., storage space), then a **Counting Sort** is the best that we can hope for since it minimizes the number of steps needed to be made in order to sort. However, remember that it only works well if there are a lot of items that are equal. The more general **Bucket Sort** is used when the bin size is greater than one and this can also be very efficient when there are many equal items.

6.6 An Example - Fire Spreading Simulation

Consider another example of where sorting is necessary. Assume that we want to simulate the spread of a fire across a terrain. Here is an example that shows three separate fires spreading across a forested area, leaving charred remains behind:



To make all of this happen, we first need to understand how the forest & lakes are represented, stored and displayed in our simulation. The following processing code displays an image of forest and lakes from a file called "smallLakes.png":

```
PImage     terrain;   // image representing the forests and lakes

void setup() {
  terrain = loadImage("smallLakes.png");
  size(terrain.width, terrain.height);
  image(terrain, 0, 0);
  loadPixels();
}

void draw() {
  updatePixels();
}
```

In the code above, the **terrain** variable is of type **PImage** which is the data type for storing images in Processing. You must call the **loadImage()** function before you can use the image, supplying the name of the image file that you want to load (which must be of type **.gif**, **.jpg**, **.tga**, or **.png**).

Once loaded, you can access the **PImage** object which contains fields for the **width** and **height** of the image, as well as an array called **pixels[]** which contains the values for every pixel in the image. You can use the **width** and **height** values to decide how big to make your window.

The **image(anImage, x, y)** procedure will draw **anImage** onto the window with the top/left of the image starting at the given **x** and **y** coordinate on the window (although in Processing there

is a minimum window width & height so if the image is very small, it may be centered in the window).

Once the image has been displayed, you can call the **loadPixels()** procedure to load the pixel data from the display window into a pre-defined Processing array called **pixels**. The **loadPixels()** procedure must always be called before reading from or writing to **pixels**. We can use the pixels array to examine and modify any pixel in the image (i.e., to determine whether or not it is grass (green) or water (blue) and to set it to show a fire (orange) or a burnt area (black).

Finally, the **updatePixels()** procedure will take the (possibly modified) data from the **pixels** array and draw the image again on the window. During our simulation, we will do this repeatedly in the draw procedure to show visually that the fires are spreading.

So, how then do we represent the fires ? Well, a fire can be shown by simply changing a pixel to orange. Say, for example that we wanted the pixel at location (85,70) in the image to be on fire. We simply need to change the correct pixel in the **pixels** array to orange. Since the pixels array is one-dimensional, we need to determine the index in the array that represents (85,70) in the image.

The **pixels** array stores each row of the image one after another. So the first row has **terrain.width** pixels in it and these are the first pixels in the array representing the pixels in which y = 0. Hence, if we want the 71st row (i.e., y = 70, since y starts at 0), then we need to multiply the width of the image by the row (i.e., 70) to bypass the first 70 rows.

Therefore, to calculate the index of (85,70) we need to use this formula:

```
pixels[85 + (70*terrain.width)]
```

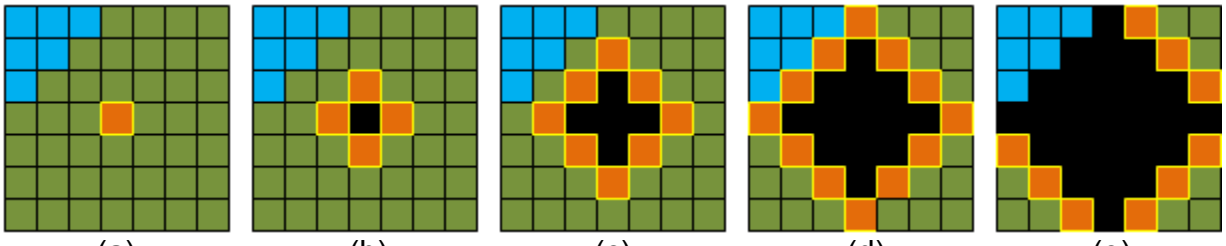
In general, the pixel at position (x, y) in the image can be set to orange as follows:

```
pixels[x + (y*terrain.width)]  = color(255, 100, 0);   // lotsa red & a bit of green = orange
```

So, we can simulate the fires by changing the appropriate pixels to orange for a while and then black afterwards to indicate a burned area.

But to simulate the fire spreading process properly, we need to make the fires grow outwards from their starting locations. But how can we do this ?

Intuitively, green pixels close to the starting location need to "catch fire" before ones that are further away. Here, for example is a single fire spreading outwards:



(a) (b) (c) (d) (e)

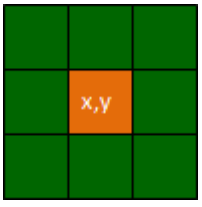
Notice that as the fire spreads ... there is a "wall" of fire that expands outwards, while the interior burned portions remain black. Intuition again tells us that we only need to consider this "wall" (also known as the ***active border***) of the fire in order to determine the spread-pattern of the fire. That is, each orange pixel is a miniature fire that needs to be spread outwards.

Therefore, we will need to maintain this active border of all locations that are currently on fire so that we can propagate the fire outwards from each of these locations. These border locations can be simply represented as (x,y) points and to begin ... we can add the first fire pixel location as the only border point upon startup. Here is how we can add code to do this:

```
PImage     terrain;      // The image with forest and lakes
Point      start;        // start location of fire
Point[]    border;       // points along active border of fire
int        borderSize;   // # points along active border of fire

// Data structure to represent a point
class Point {
  int x, y;
  Point(int ix, int iy) {
     x = ix;
     y = iy;
  }
}

void setup() {
  terrain = loadImage("smallLakes.png");
  size(terrain.width, terrain.height);
  image(terrain, 0, 0);
  loadPixels();

  border = new Point[1000];
  border[0] = new Point(85,70);
  borderSize = 1;
}
```

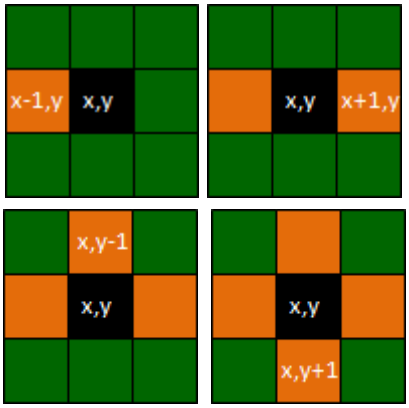
Notice that the border array begins with one point, but that it is able to hold a lot more as the border grows. The number 1000 is somewhat arbitrary, but it is big enough to hold a lot of border points. If the entire 107x102 pixel image was on fire ... that would require 17,340 border pixels potentially. However, we must remember that as the border grows it does not get very large since the previous border points will change to black as the fire consumes the forest. So, 1000 points is reasonable in our example, although with larger images a larger border would be necessary.

So now that we know the active border pixels, how do we "grow" the fires ? Well, for each pixel on the active border, we just need to look at the neighboring pixels surrounding it (i.e., see picture (b) on the previous page). Assume that the pixels are stored in a 2-dimensional image called **image** and that each location **image[x][y]** is either colored green, blue, orange or black.



Given that a **image[x][y]** is on the active border of the fire, here is the idea for spreading the fire from point (x,y):

```
image[x][y] ← BLACK   // make burnt now

image[x-1][y] ← ORANGE        // burn left now

image[x+1][y] ← ORANGE        // burn right now

image[x][y-1] ← ORANGE        // burn up now

image[x][y+1] ← ORANGE        // burn down now
```



We will need to make sure that all of these "new" fires are added to the active border for the next round of spreading. So, we need to repeat the process of extracting a fire location from the active border, processing it (i.e., spread outwards from here) and then repeat.

Here is the algorithm that we have so far now:

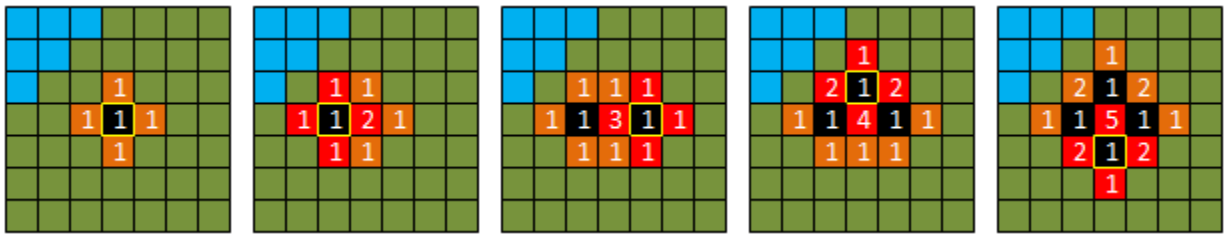
```
Algorithm: FireSpread
    image:    the image containing forests and lakes
    start:    point representing the starting location of the fire

1.  borders = new array of points, initially empty
2.  borderSize = 0
3.  borders[0] = start
4.  while (borderSize > 0) {
5.      x ← borders[borderSize-1].x
6.      y ← borders[borderSize-1].y
7.      image[x][y] ← BLACK                    // burnt
8.      borderSize ← borderSize - 1

9.      borders[borderSize] ← new Point (x-1, y)
10.     borderSize ← borderSize + 1
11.     image[x-1][y] ← ORANGE

12.     borders[borderSize] ← new Point (x+1, y)
13.     borderSize ← borderSize + 1
14.     image[x+1][y] ← ORANGE

15.     borders[borderSize] ← new Point (x, y-1)
16.     borderSize ← borderSize + 1
17.     image[x][y-1] ← ORANGE

18.     borders[borderSize] ← new Point (x, y+1)
19.     borderSize ← borderSize + 1
20.     image[x][y+1] ← ORANGE
    }
```

We need to make sure however, that we do not process any pixels outside the valid range. For example, if the border point is the top left pixel in the image, then we cannot try to propagate the fire upwards nor leftwards because the point (-1,-1) would be out of range. Therefore, we need to add some "bounds-checking" to ensure that this does not happen.

We need to place conditional statements around lines 9-11, 12-14, 15-17 and 18-20 as follows:

```
9.          if (x > 0) then {                       // make sure not gone beyond left border
10.             borders[borderSize] ← new Point (x-1, y)
11.             borderSize ← borderSize + 1
12.             image[x-1][y] ← ORANGE
            }

13.         if (x < image.width-1) then {           // make sure not gone beyond right border
14.             borders[borderSize] ← new Point (x+1, y)
15.             borderSize ← borderSize + 1
16.             image[x+1][y] ← ORANGE
            }
17.         if (y > 0) then {                       // make sure not gone beyond top border
18.             borders[borderSize] ← new Point (x, y-1)
19.             borderSize ← borderSize + 1
20.             image[x][y-1] ← ORANGE
            }
21.         if (y < image.height-1) then {          // make sure not gone beyond bottom border
22.             borders[borderSize] ← new Point (x, y+1)
23.             borderSize ← borderSize + 1
24.             image[x][y+1] ← ORANGE
            }
        }
```

At this point, we still have a slight problem. When we extract a border point, we need to ensure that we don't re-propagate through the same points over and over again (i.e., through points that are ORANGE or BLACK), otherwise we will never have an end to our fires and fires can restart. Below, for example, shows how we spread out from the 4 fires around the center, showing the number of times a location is added as a fire to the active border:



It may come as a surprise to you that 5 separate fires end up starting at the center location ... even after that location has been burned. We need to ensure that we do not add points to the border if there is already a fire there (i.e., ORANGE) or if there is a burned area there (i.e., BLACK) or if there is water there (i.e., BLUE).

To do this, we need to again modify lines 9, 13, 17 and 21 as follows:

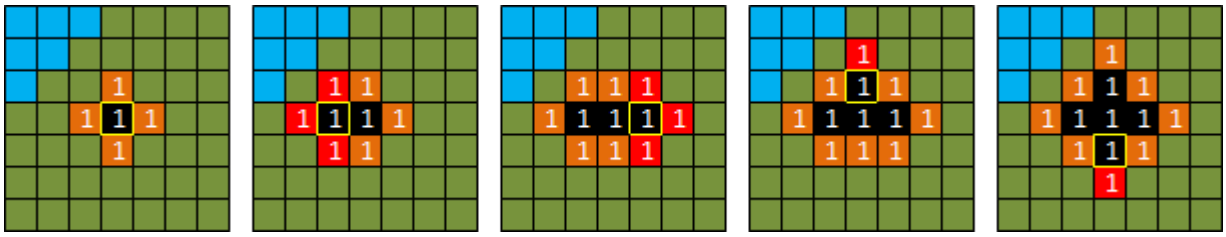
```
9.      if ((x > 0) AND (image[x-1][y] is GREEN)) then {
```

```
13.     if ((x < image.width-1) AND (image[x+1][y] is GREEN)) then {
```

```
17.     if ((y > 0) AND (image[x][y-1] is GREEN)) then {
```

```
21.     if ((y < image.height-1) AND (image[x][y+1] is GREEN)) then {
```

As a result, we do not repeatedly add fires to the same location:

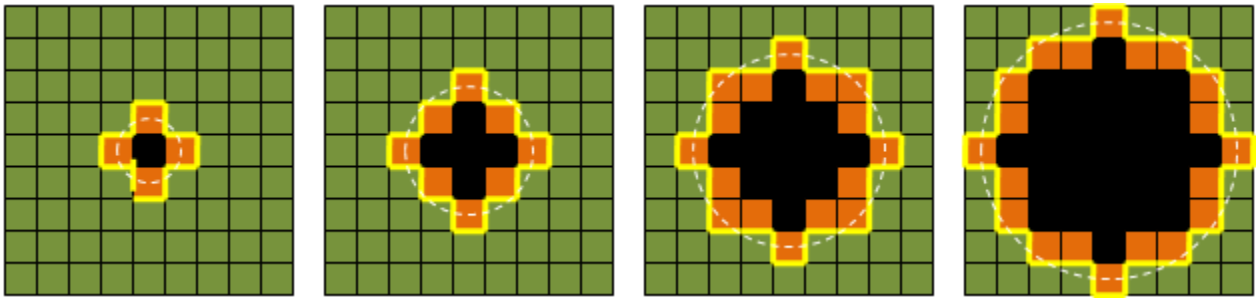


At this point, the active border will begin by growing from 1 point to having 4 new points in it, representing 4 new fires (as shown above). However, you may notice in the image above that the fire will spread in a diamond-like pattern. Here is what it would look like as we continued this process:

This pattern is called an *artifact* of the simulation.

> *An **artifact** is something that appears in a scientific result that is not a true feature of thing being studied, but instead a result of the experimental or analysis method, or observational error.*

Clearly, this is not a realistic fire-spread model as fires do not spread in diamond-shape formation. In order to make the spreading more realistic, we need to cause the fire to spread outwards by processing the pixels outwards in a circular pattern from the starting fire location:



To do this, we need to ensure that active border points that are closest to the center are processed first. So, we need to sort the border points by their distance to the starting location of the fire.

One way of doing this is to compute the distance from the center of the fire (i.e., the initial starting location) to each point on the active border. Then, we can sort the points on the active border according to their distance to the fire center and make sure that the next point to be processed is the one that is closest to the center.

To do this, we would need to keep track of the distance to each point on the active border which would be set to the distance from the starting fire location. To represent these distances, we can have each point on the active border maintain its distance from the fire's center. In processing, we could re-define the Point data structure as shown here. Here are the changes to the algorithm:

```
class Point {
  int x, y, distance;
  Point(int ix, int iy, float d) {
      x = ix;
      y = iy;
      distaince = d;
  }
}
```

Algorithm: FireSpread(image, start)

```
1.    borders = new array of points, initially empty
2.    borderSize = 0
3.    borders[0] = start
4.    while (borderSize > 0) {
5.        x ← borders[borderSize-1].x
6.        y ← borders[borderSize-1].y
7.        image[x][y] ← BLACK
8.        borderSize ← borderSize - 1
9.        if ((x > 0) AND (image[x-1][y] is GREEN)) then {
10.           borders[borderSize] ← new Point (x-1, y, distance from start to (x-1, y))
11.           borderSize ← borderSize + 1
12.           image[x-1][y] ← ORANGE
          }
13.       if ((x < image.width-1) AND (image[x+1][y] is GREEN)) then {
14.           borders[borderSize] ← new Point (x+1, y, distance from start to (x+1, y))
15.           borderSize ← borderSize + 1
16.           image[x-1][y] ← ORANGE
          }
17.       if ((y > 0) AND (image[x][y-1] is GREEN)) then {
18.           borders[borderSize] ← new Point (x, y-1, distance from start to (x, y-1))
19.           borderSize ← borderSize + 1
20.           image[x][y-1] ← ORANGE
          }
21.       if ((y < image.height-1) AND (image[x][y+1] is GREEN)) then {
22.           borders[borderSize] ← new Point (x, y+1, distance from start to (x, y+1))
23.           borderSize ← borderSize + 1
24.           image[x][y+1] ← ORANGE
          }
25.       sort borders by distances from largest to smallest
      }
```

Any sort algorithm will suffice, as long as the sorting is done with respect to the distances stored in each border point.

The above code will produce a nice "round" fire spread ... however ... the fire is still not spreading realistically. Can you see what is wrong in the following images:

Notice particularly the 2nd image. See how the fire quickly spreads around the lake on the left side of the fire border circle ? This is not realistic as the fire must spread from the bottom left side of the lake upwards....and this takes time. So, the point is... we cannot assume that the fire should be processed with respect to the distance from the center of the fire's starting point. Instead, we must adjust the code so that the fire "bends" properly around obstacles.

To do this, we need to adjust our computation so that each new fire spreads time-wise relative to the fire that spawned it. That is, the cost of a fire should be with respect to the distance from the border point that it started from. Therefore, we need to adjust lines 10, 14, 18 and 22 as follows:

```
10.    borders[borderSize] ← new Point (x-1, y, distance from (x, y) + 1)
```

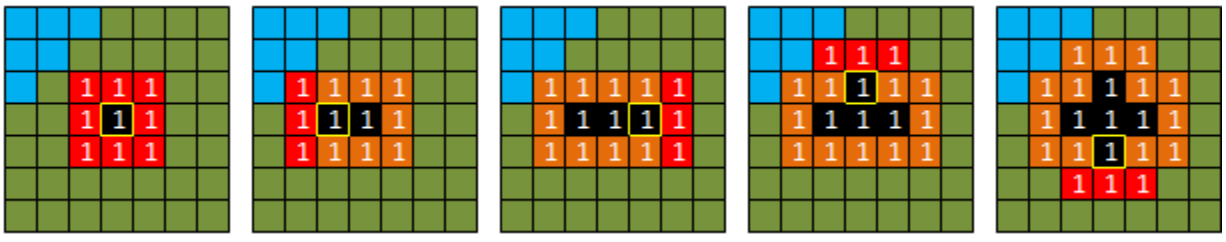
```
14.    borders[borderSize] ← new Point (x+1, y, distance from (x, y) +1)
```

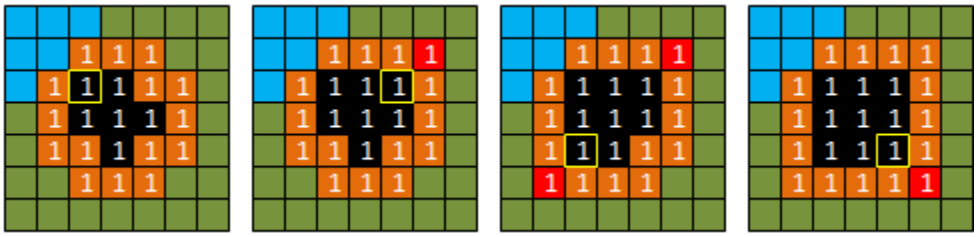
```
18.    borders[borderSize] ← new Point (x, y-1, distance from (x, y) + 1)
```

```
22.    borders[borderSize] ← new Point (x, y+1, distance from (x, y) + 1)
```

The above code allows each new fire point to have a distance value that is 1 more than the previous distance value.

However, oddly enough, the code will still produce a result that has a diamond-shaped pattern in it !!! This diamond-shaped artifact is a result of the strategy of only updating the 4 pixel neighborhood around each pixel. This is called the ***von Neuman neighborhood***. The diamond-shape can be adjusted to an octagonal shape if a ***Moore neighborhood*** is used. A Moore neighborhood includes the diagonal pixels around a pixel. To use this 8-pixel update model, we would need to add additional code to add border points to the 8 surrounding points:





To accomplish this, we simply need to add the following code to the end of our existing code:

```
25.       if ((x > 0) AND (image[x-1][y-1] is GREEN)) then {
26.           borders[borderSize] ← new Point (x-1, y-1, distance from (x, y) + 1.417))
27.           borderSize ← borderSize + 1
28.           image[x-1][y-1] ← ORANGE
          }
29.       if ((x < image.width-1) AND (image[x+1][y-1] is GREEN)) then {
30.           borders[borderSize] ← new Point (x+1, y-1, distance from (x, y) + 1.417))
31.           borderSize ← borderSize + 1
32.           image[x+1][y-1] ← ORANGE
          }
33.       if ((y > 0) AND (image[x-1][y+1] is GREEN)) then {
34.           borders[borderSize] ← new Point (x-1, y+1, distance from (x, y) + 1.417))
35.           borderSize ← borderSize + 1
36.           image[x-1][y+1] ← ORANGE
          }
37.       if ((y < image.height-1) AND (image[x+1][y+1] is GREEN)) then {
38.           borders[borderSize] ← new Point (x+1, y+1, distance from (x, y) + 1.417))
39.           borderSize ← borderSize + 1
40.           image[x+1][y+1] ← ORANGE
          }

41.       sort borders by distances from largest to smallest
      }
```

Notice that the cost to the diagonals is actually 1.417 ... which is the square root of 2. This is the diagonal distance from the middle of the center pixel to the middle of the diagonal pixel.

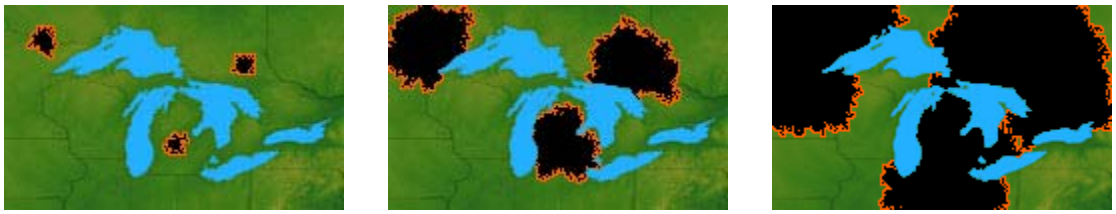
As you can see, the result is now octagonal ... a more realistic approximation, although the octagonal shape is still somewhat "artificial-looking":

However, we can make this even more realistic by adding a degree of randomness. Instead of having fixed distances as the fire spreads, we can add some random cost as well. For example, we can adjust the code as follows:

```
10.  borders[borderSize] ← new Point (x-1, y, distance(x, y) + 1 + rand(3))

14.  borders[borderSize] ← new Point (x+1, y, distance(x, y) + 1 + rand(3))

18.  borders[borderSize] ← new Point (x, y-1, distance(x, y) + 1 + rand(3))

22.  borders[borderSize] ← new Point (x, y+1, distance(x, y) + 1 + rand(3))

26.  borders[borderSize] ← new Point (x-1, y-1, distance(x, y) +1.414+ rand(3))

30.  borders[borderSize] ← new Point (x+1, y-1, distance(x, y) +1.414 + rand(3))

34.  borders[borderSize] ← new Point (x-1, y+1, distance(x, y) +1.414 + rand(3))

38.  borders[borderSize] ← new Point (x+1, y+1, distance(x, y) +1.414 + rand(3))
```

The **rand(3)** indicates a random number from 0 to 3. Here is the result:

The higher the random value, the less circular the shape will be as the fires spread. For example, if we increase the randomness to 50, then here is what we will get:



Lastly, we can even simulate wind. For example, we can have a low cost for spreading to the right and high cost for spreading left ... this would cause the fire to spread quickly rightwards:

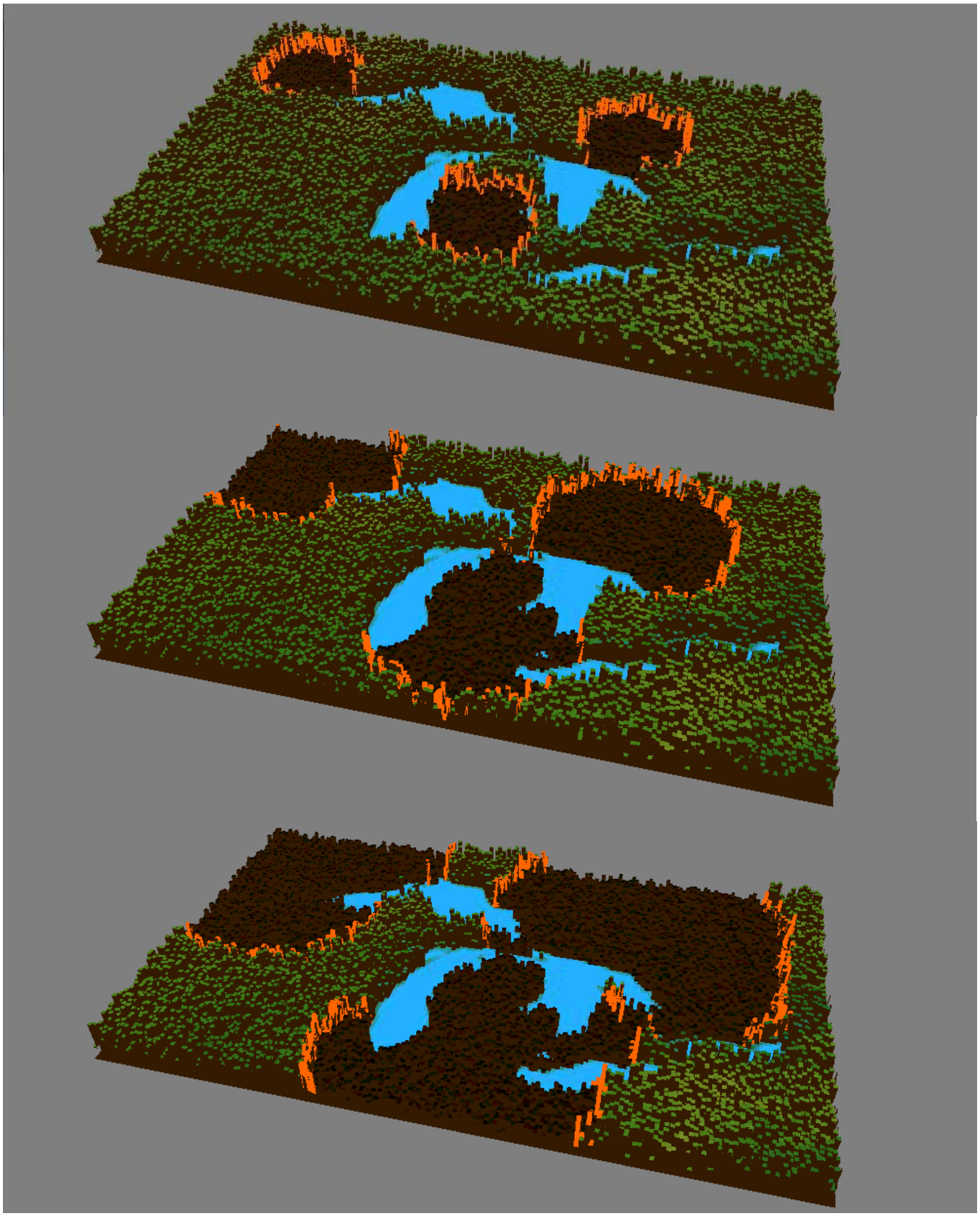
Ultimately, it would be best to produce a more realistic fire-spread model that takes into account the type of trees being burned, wind, elevation, etc...

Here is the final code in Processing:

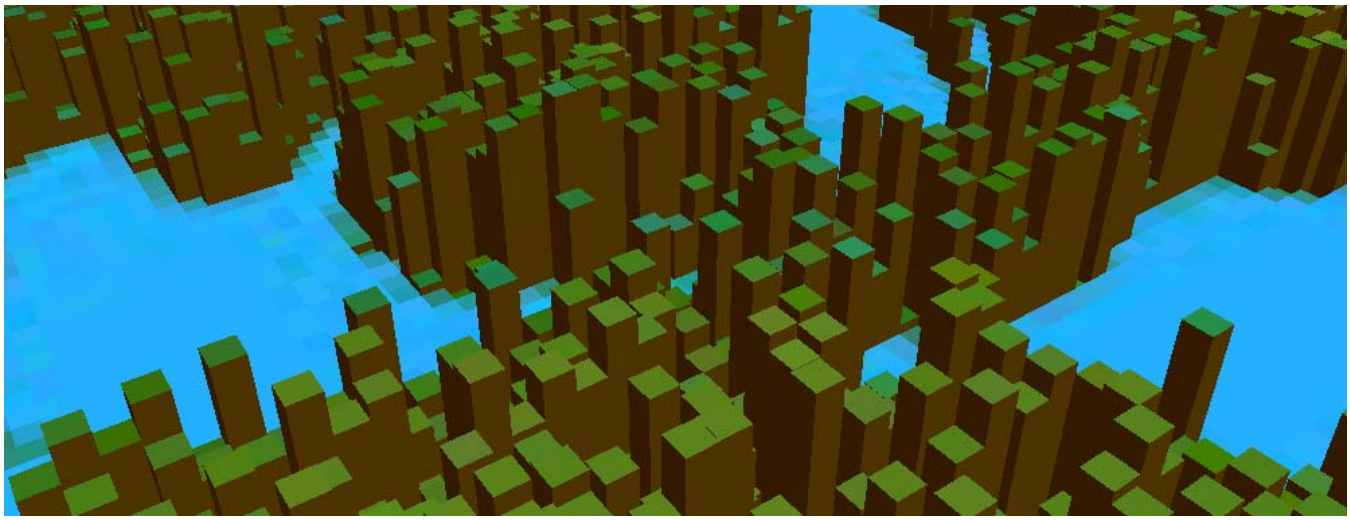
```
PImage     terrain;      // The image with the grass and lakes
Point[]    border;       // points along the border of the fire
int        borderSize;   // # points along the border of the fire

color      orange = color(255, 100, 0);
color      black = color(0, 0, 0);

// Data structure to represent a point
class Point {
  int     x, y;
  float   cost;
  Point(int ix, int iy, float c) {
    x = ix;
    y = iy;
    cost = c;
  }
}

void setup() {
  terrain = loadImage("smallLakes.png");
  size(terrain.width, terrain.height);
  image(terrain, 0, 0);
  loadPixels();
  border = new Point[1000];
  border[0] = new Point(85, 70, 0);
  border[1] = new Point(120, 30, 0);
  border[2] = new Point(20, 20, 0);
  borderSize = 3;
}

void draw() {
  if (!spreadFire()) {
    exit();         // Quit the program
  }
  updatePixels();
  println(borderSize);
}

boolean isUnburnedForest(int x, int y) {
  return (blue(get(x,y)) < 128) && (green(get(x,y)) > 80) && (red(get(x,y)) < 150);
}

boolean spreadFire() {
  if (borderSize > 0) {
    Point p = border[borderSize-1];
    pixels[p.x + (p.y*terrain.width)] = black;
    borderSize--;

    // Check left
    if ((p.x > 0) && isUnburnedForest(p.x-1,p.y)) {
      pixels[p.x-1 + (p.y*terrain.width)] = orange;
      border[borderSize++] = new Point(p.x-1, p.y, p.cost + 1 + random(3));
    }

    // Check right
    if ((p.x < terrain.width-1) && isUnburnedForest(p.x+1,p.y)) {
      pixels[p.x+1 + (p.y*terrain.width)] = orange;
      border[borderSize++] = new Point(p.x+1, p.y, p.cost + 1 + random(3));
    }
```

```
    // Check up
    if ((p.y > 0) && isUnburnedForest(p.x,p.y-1)) {
      pixels[p.x + (p.y-1)*terrain.width] = orange;
      border[borderSize++] = new Point(p.x, p.y-1, p.cost + 1 + random(3));
    }

    // Check down
    if ((p.y < terrain.height-1) && isUnburnedForest(p.x,p.y+1)) {
      pixels[p.x + (p.y+1)*terrain.width] = orange;
      border[borderSize++] = new Point(p.x, p.y+1, p.cost + 1 + random(3));
    }

    // Check left/up diagonal
    if ((p.x > 0) && (p.y > 0) && isUnburnedForest(p.x-1,p.y-1)) {
      pixels[p.x-1 + (p.y-1)*terrain.width] = orange;
      border[borderSize++] = new Point(p.x-1, p.y-1, p.cost + 1.414 + random(3));
    }

    // Check right/up diagonal
    if ((p.x < terrain.width-1) && (p.y > 0) && isUnburnedForest(p.x+1,p.y-1)) {
      pixels[p.x+1 + (p.y-1)*terrain.width] = orange;
      border[borderSize++] = new Point(p.x+1, p.y-1, p.cost + 1.414 + random(3));
    }

    // Check left/down diagonal
    if ((p.x > 0) && (p.y < terrain.height-1) && isUnburnedForest(p.x-1,p.y+1)) {
      pixels[p.x-1 + (p.y+1)*terrain.width] = orange;
      border[borderSize++] = new Point(p.x-1, p.y+1, p.cost + 1.414 + random(3));
    }

    // Check right/down diagonal
    if ((p.x < terrain.width-1) && (p.y<terrain.height-1) && isUnburnedForest(p.x+1,p.y+1)) {
      pixels[p.x+1 + (p.y+1)*terrain.width] = orange;
      border[borderSize++] = new Point(p.x+1, p.y+1, p.cost + 1.414 + random(3));
    }

    BubbleSort(border, borderSize);
    return true;
  }
  return false;
}

// BubbleSort
void BubbleSort(Point[] fire, int fireSize) {
  for (int p=fireSize-1; p>=0; p--) {
    boolean madeSwap = false;
    for (int i=0; i<=p-1; i++) {
      if (fire[i].cost < fire[i+1].cost) {
        Point temp = fire[i+1];
        fire[i+1] = fire[i];
        fire[i] = temp;
        madeSwap = true;
      }
    }
    if (!madeSwap) return;
  }
}
```

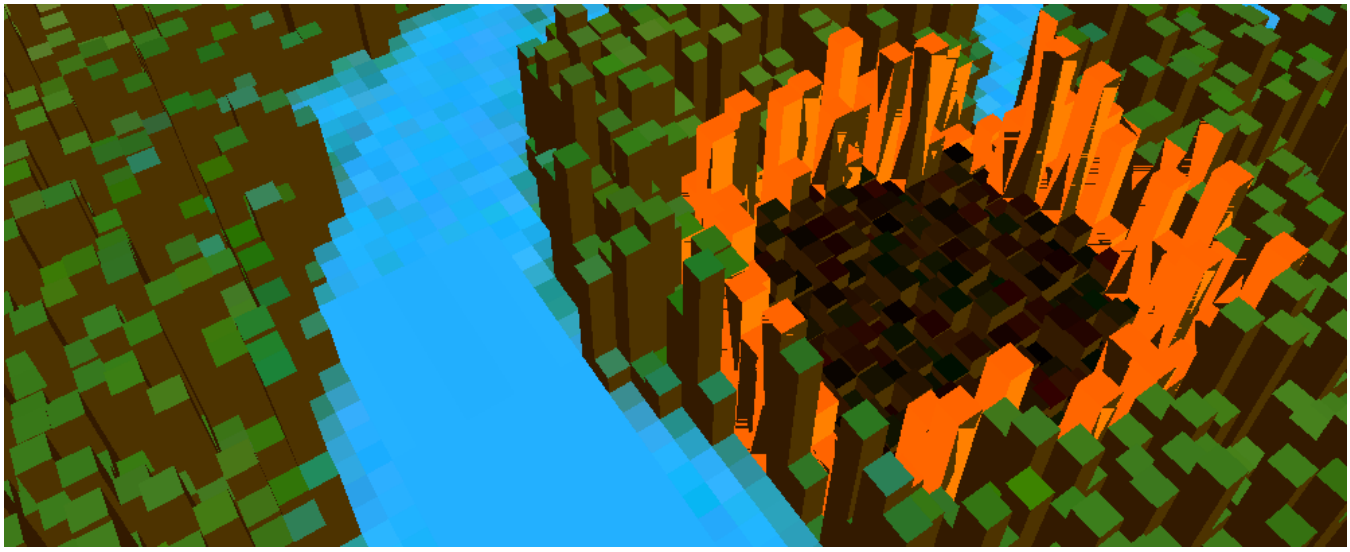
For added enjoyment, we can create a 3D version of the landscape and then watch as the fire spreads, burning the trees (rest assured that no animals were harmed in this simulation). The code to do this is quite similar. Instead of displaying an image, we just need to create a big pile of square surfaces that corresponds to the terrain.



The idea is to simply take the image and assign a height value to each pixel. Water could have height 0 while the trees could have a height which is some constant value with a degree of randomness to provide variety. We could then form a 3D rectangular polyhedra for each pixel and display them ...



As the fire spreads, we can color the rectangular polyhedra orange and then one burned, we can decrease their height to a small value to represent burnt "stubble":



Here is the code. We will not be discussing the 3D aspects of this code:

```
float     xmag, ymag = 0;         // Used for mouse interaction
float     newXmag, newYmag = 0;   // Used for mouse interaction

PImage     terrain;               // The image with the grass and lakes
int[][]    imgPixels;             // Holds the image color data
float[][]  heights;               // Holds the image "pixel" heights
int        halfWidth;             // Half the computerd width
int        halfHeight;            // Half the computerd height
int        scale;                 // Zoom factor
Point[]    border;                // points along the border of the fire
int        borderSize;            // # points along the border of the fire

color      orange = color(255, 100, 0);

// Data structure to represent a point
class Point {
  int     x, y;
  float   cost;
  Point(int ix, int iy, float c) {
```

```
    x = ix;
    y = iy;
    cost = c;
  }
}

void setup() {
  size(1000, 1000, P3D);
  noStroke();
  colorMode(RGB, 1);

  terrain = loadImage("smallLakes.png");
  imgPixels = new int[terrain.width][terrain.height];
  heights = new float[terrain.width][terrain.height];
  halfWidth = terrain.width/2;
  halfHeight = terrain.height/2;
  scale = 5;

  for (int i = 0; i < terrain.height; i++) {
    for (int j = 0; j < terrain.width; j++) {
      imgPixels[j][i] = terrain.get(j, i);
      float h = 5*random(2);
      if ((blue(imgPixels[j][i]) > 0.4))
        h = 1;
      heights[j][i] = h;
    }
  }

  border = new Point[1000];
  border[0] = new Point(85, 70, 0);
  border[1] = new Point(120, 30, 0);
  border[2] = new Point(20, 20, 0);
  borderSize = 3;

}

void draw() {
  if (!spreadFire()) {
    println("Total Time: " + millis()/1000.0 + " seconds");
    exit();
  }

  pushMatrix();

  // Code for allowing user interaction to rotate the terrain
  if (mousePressed) {
    if (mouseButton == LEFT) {
      newXmag = mouseX/float(width) * TWO_PI;
      newYmag = mouseY/float(height) * TWO_PI;
    }
    else
      scale = width/2 - mouseX;
  }

  translate(width/2, height/2, -30);
  float diff = xmag-newXmag;
  if (abs(diff) >  0.01) { xmag -= diff/4.0; }
  diff = ymag-newYmag;
  if (abs(diff) >  0.01) { ymag -= diff/4.0; }
  rotateX(-ymag);
  rotateZ(-xmag);
  scale(scale);

  // Draw the terrain
  background(0.5);
  beginShape(QUADS);
  for (int i = 0; i < terrain.width-1; i++) {
```

```
    for (int j = 0; j < terrain.height-1; j++) {
      // Set the surface color for this "pixel"
      fill(red(imgPixels[i][j]), green(imgPixels[i][j]), blue(imgPixels[i][j]));

      // Add the surface face
      vertex(i-halfWidth, j-halfHeight, heights[i][j]);
      vertex(i+1-halfWidth, j-halfHeight, heights[i][j]);
      vertex(i+1-halfWidth, j+1-halfHeight, heights[i][j]);
      vertex(i-halfWidth, j+1-halfHeight, heights[i][j]);

      // Add the remaining 4 side faces
      if (imgPixels[i][j] == orange)
        fill(1.0,0.4,0.0);
      else if (heights[i][j] < 10 )
        fill(0.2,0.1,0.0);
      else
        fill(0.2,0.6,0.0);
      vertex(i+1-halfWidth, j-halfHeight, heights[i][j]);
      vertex(i+1-halfWidth, j-halfHeight, heights[i+1][j]);
      vertex(i+1-halfWidth, j+1-halfHeight, heights[i+1][j]);
      vertex(i+1-halfWidth, j+1-halfHeight, heights[i][j]);

      if (i > 0) {
        vertex(i-halfWidth, j-halfHeight, heights[i][j]);
        vertex(i-halfWidth, j+1-halfHeight, heights[i][j]);
        vertex(i-halfWidth, j+1-halfHeight, heights[i-1][j]);
        vertex(i-halfWidth, j-halfHeight, heights[i-1][j]);
      }

      if (imgPixels[i][j] == orange)
        fill(1.0,0.5,0.0);
      else if (heights[i][j] < 10 )
        fill(0.3,0.2,0.0);
      else
        fill(0.3,0.7,0.0);
      vertex(i-halfWidth, j+1-halfHeight, heights[i][j]);
      vertex(i+1-halfWidth, j+1-halfHeight, heights[i][j]);
      vertex(i+1-halfWidth, j+1-halfHeight, heights[i][j+1]);
      vertex(i-halfWidth, j+1-halfHeight, heights[i][j+1]);

      if (j > 0) {
        vertex(i-halfWidth, j-halfHeight, heights[i][j]);
        vertex(i+1-halfWidth, j-halfHeight, heights[i][j]);
        vertex(i+1-halfWidth, j-halfHeight, heights[i][j-1]);
        vertex(i-halfWidth, j-halfHeight, heights[i][j-1]);
      }
    }
  }

  // Draw the bottom face
  fill(0.2,0.1,0);  // Dark brown
  vertex(-halfWidth,  -halfHeight,  -1);  vertex(-halfWidth,  halfHeight,  -1);
  vertex(halfWidth,  halfHeight,  -1);    vertex(halfWidth,  -halfHeight,  -1);

  endShape();
  popMatrix();
}

boolean isUnburnedForest(int x, int y) {
  return (blue(imgPixels[x][y]) < 0.5) && (green(imgPixels[x][y]) > 0.31) &&
(red(imgPixels[x][y]) < 0.59);
}

boolean spreadFire() {
  if (borderSize > 0) {
    Point p = border[borderSize-1];
    imgPixels[p.x][p.y] = color(0.2*random(1), 0.1*random(1), 0);  // random dark brown color
```

```
    heights[p.x][p.y] = random(2);      // random height for burned twigs

    borderSize--;

    // Check left
    if ((p.x > 0) && isUnburnedForest(p.x-1,p.y)) {
      imgPixels[p.x-1][p.y] = orange;
      border[borderSize++] = new Point(p.x-1, p.y, p.cost + 1 + random(3));
    }
    // Check right
    if ((p.x < terrain.width-1) && isUnburnedForest(p.x+1,p.y)) {
      imgPixels[p.x+1][p.y] = orange;
      border[borderSize++] = new Point(p.x+1, p.y, p.cost + 1 + random(3));
    }
    // Check up
    if ((p.y > 0) && isUnburnedForest(p.x,p.y-1)) {
      imgPixels[p.x][p.y-1] = orange;
      border[borderSize++] = new Point(p.x, p.y-1, p.cost + 1 + random(3));
    }
    // Check down
    if ((p.y < terrain.height-1) && isUnburnedForest(p.x,p.y+1)) {
      imgPixels[p.x][p.y+1] = orange;
      border[borderSize++] = new Point(p.x, p.y+1, p.cost + 1 + random(3));
    }
    // Check left/up diagonal
    if ((p.x > 0) && (p.y > 0) && isUnburnedForest(p.x-1,p.y-1)) {
      imgPixels[p.x-1][p.y-1] = orange;
      border[borderSize++] = new Point(p.x-1, p.y-1, p.cost + 1.414 + random(3));
    }
    // Check right/up diagonal
    if ((p.x < terrain.width-1) && (p.y > 0) && isUnburnedForest(p.x+1,p.y-1)) {
      imgPixels[p.x+1][p.y-1] = orange;
      border[borderSize++] = new Point(p.x+1, p.y-1, p.cost + 1.414 + random(3));
    }
    // Check left/down diagonal
    if ((p.x > 0) && (p.y < terrain.height-1) && isUnburnedForest(p.x-1,p.y+1)) {
      imgPixels[p.x-1][p.y+1] = orange;
      border[borderSize++] = new Point(p.x-1, p.y+1, p.cost + 1.414 + random(3));
    }
    // Check right/down diagonal
    if ((p.x < terrain.width-1)&&(p.y < terrain.height-1) && isUnburnedForest(p.x+1,p.y+1)) {
      imgPixels[p.x+1][p.y+1] = orange;
      border[borderSize++] = new Point(p.x+1, p.y+1, p.cost + 1.414 + random(3));
    }

    BubbleSort(border, borderSize);
    return true;
  }
  return false;
}


// BubbleSort
void BubbleSort(Point[] fire, int fireSize) {
  for (int p=fireSize-1; p>=0; p--) {
    boolean madeSwap = false;
    for (int i=0; i<=p-1; i++) {
      if (fire[i].cost < fire[i+1].cost) {
        Point temp = fire[i+1];
        fire[i+1] = fire[i];
        fire[i] = temp;
        madeSwap = true;
      }
    }
    if (!madeSwap) return;
  }
}
```