Chapter 6

Sorting and Efficient Searching

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. We also discuss **Binary Searching** ... a more efficient way of searching through sorted data

6.1 Sorting

In addition to searching, *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 (or arrange) 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.

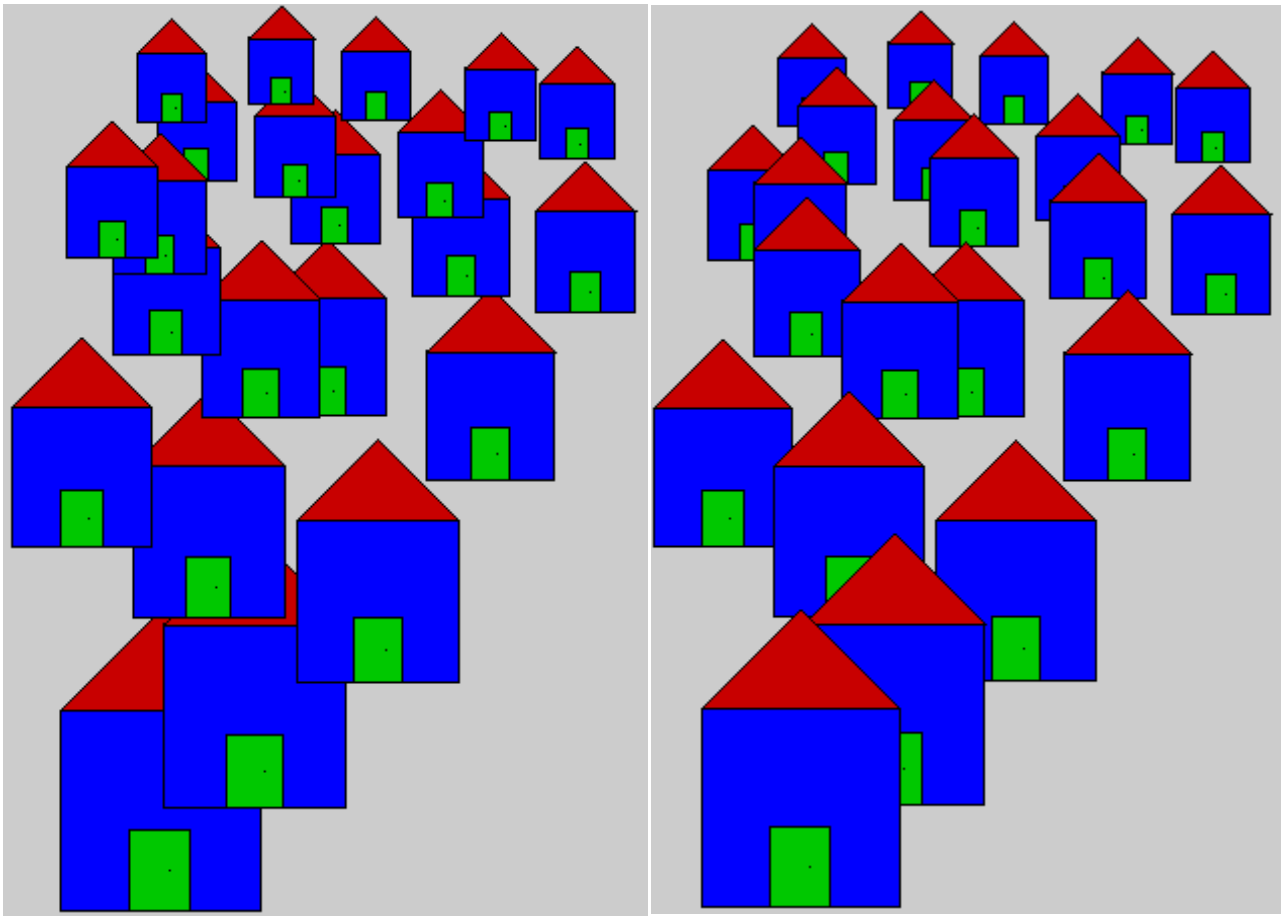
- DVDs are sorted by category on **Netflix** (e.g., comedy, drama, new releases) so that we can easily find what we are looking for.

- 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.

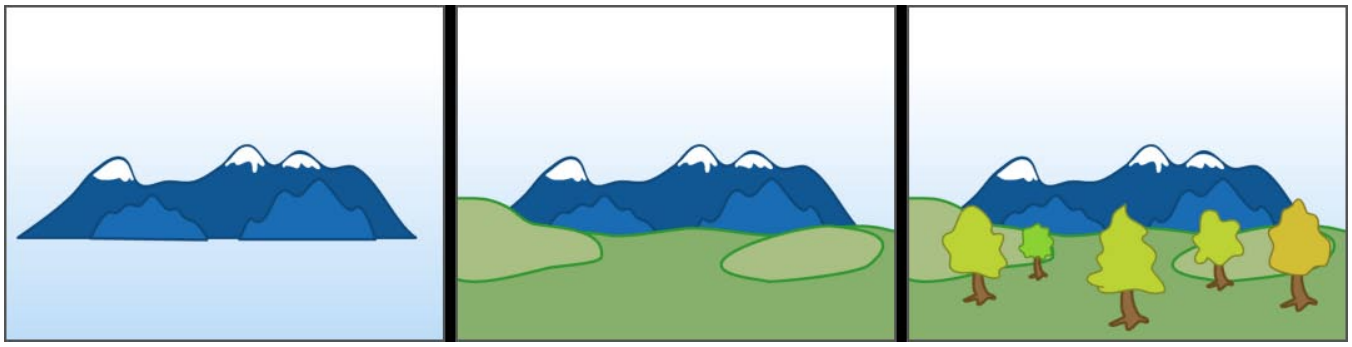
- **Ballots** can be sorted so that we can determine easily who had the most votes.

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. 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) original (given) order (b) sorted order

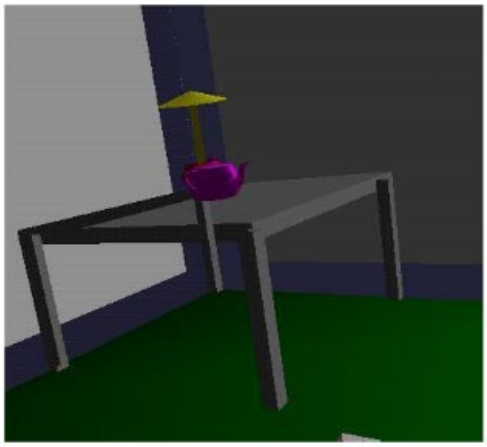
Notice how the houses in (a) are not drawn realistically in terms of proper perspective. In (b), however, the houses are displayed from back to front. That is, those with a smaller y-value (i.e., topmost houses) are displayed first. Thus, sorting the houses by their y-coordinate and then displaying them 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:



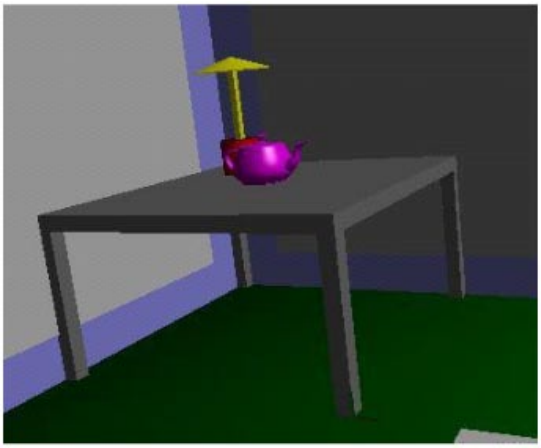
In 3D computer graphics (e.g., in a 3D game) most objects are made up of either triangular faces or quad surfaces joined together to represent 3D objects and surfaces. The triangles must be displayed in the correct order with respect to the current viewpoint. That is, far triangles need to be displayed 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, QuickSort
Selection Sorts	Selection Sort, Heap Sort, Cartesian Tree Sort, Tournament Sort
Insertion Sorts	Insertion Sort, Shell Sort, Tree Sort, Library Sort
Merge Sorts	Merge Sort, Polyphase Merge Sort, Strand Sort
Non-Comparison Sorts	Bucket Sort, Burst Sort, Counting Sort, Pigeonhole 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.

Just to give you a feel for how much of an impact our choice in algorithm can make, consider sorting **N** items using an $\mathbf{N^2}$ sorting algorithm, vs. an **N·log(N)** sorting algorithm. The table below shows how fast the numbers can grow. Notice how much time (i.e., Run Time) that it would take to sort, given that we can process 1000 items per second:

N	N^2 Comparisons	N^2 Run Time	N·log(N) Comparisons	N·log(N) Run Time
10	100	0.1 sec	33	0.03 sec
20	400	0.4 sec	86	0.09 sec
50	2.5k	2.5 sec	282	0.28 sec
100	10k	10 sec	664	0.66 sec
200	40k	40 sec	1.5k	1.5 sec
500	250k	**4** min	4.5k	4.5 sec
1,000	1M	**16** min	10k	10 sec
10,000	100M	**27** hours	133k	**2.2** min
50,000	2.5G	**29** days	780k	**13** min
100,000	10G	**116** days	1.66M	**27** min
1,000,000	1T	**31** years !	19.9M	**5.5** hours

It is also possible to compare algorithms in terms of:

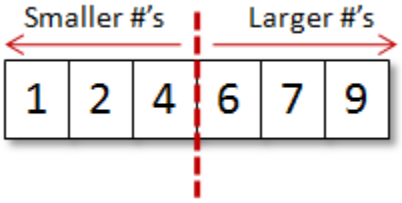
1. how many times a pair of items in the list are swapped (i.e., change positions).
2. how much memory (or other computer resources) is required to complete the sort.
3. how simple the algorithm is to implement.

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.

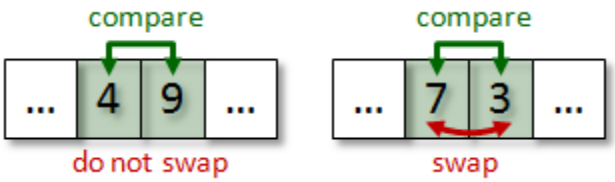
6.2 Bubble Sort

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. In a list that is sorted in increasing order, the smallest numbers are in the first half of the list, while the largest numbers are in the second half of the list:



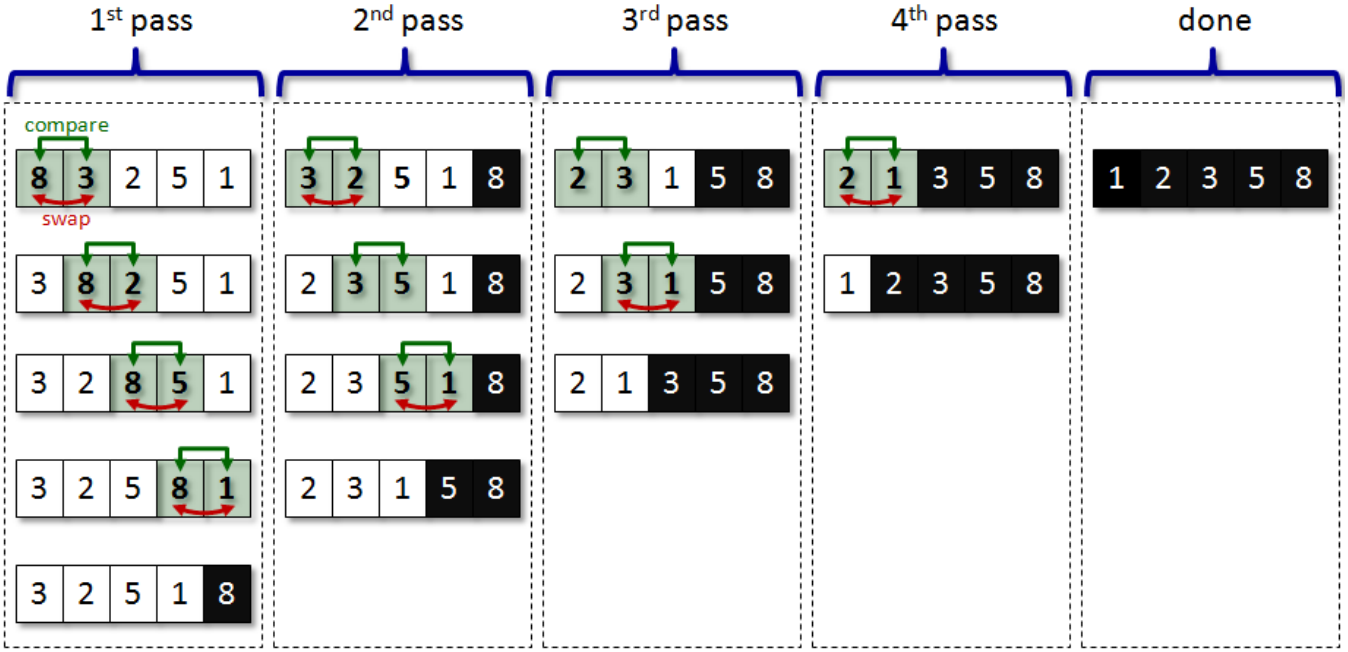
By considering this principle, we can actually draw conclusions as to relative positions of adjacent numbers in the list as we are trying to sort. For example, assume that we examine any two adjacent numbers in the list. If the number on the left is *smaller* than the one on the right (e.g., 4 and 9 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/top/left* of the heavier item. It does this by making multiple passes (i.e., multiple iterations or "rounds") through the list, each time *moving*/*sinking* the heaviest item towards its final position at the *end*/*bottom*/*right* 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 **N-1** (i.e., 4 in this example) 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
        N:        the size of the array

1.      repeat N-1 times {
2.          for each location i from 0 to N-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
        N:        the size of the array

1.      for each pass p from N-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 swaps, 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
        N:        the size of the array

1.      for each pass p from N-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

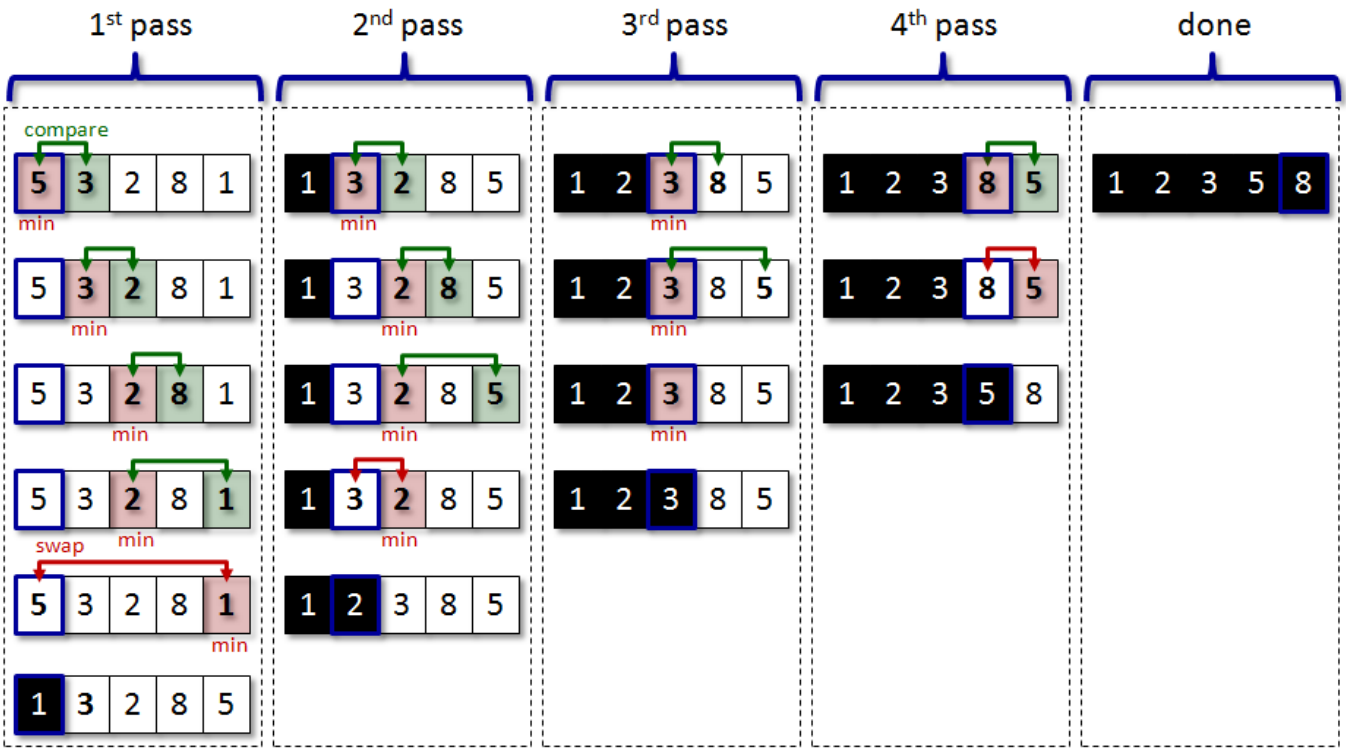
The **selection sort** algorithm is a natural kind of sorting technique. It is also easy to implement but like the Bubble Sort, it also has a bad worst-case complexity of around N^2, so it is not an efficient algorithm when **N** becomes large.

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 then 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:

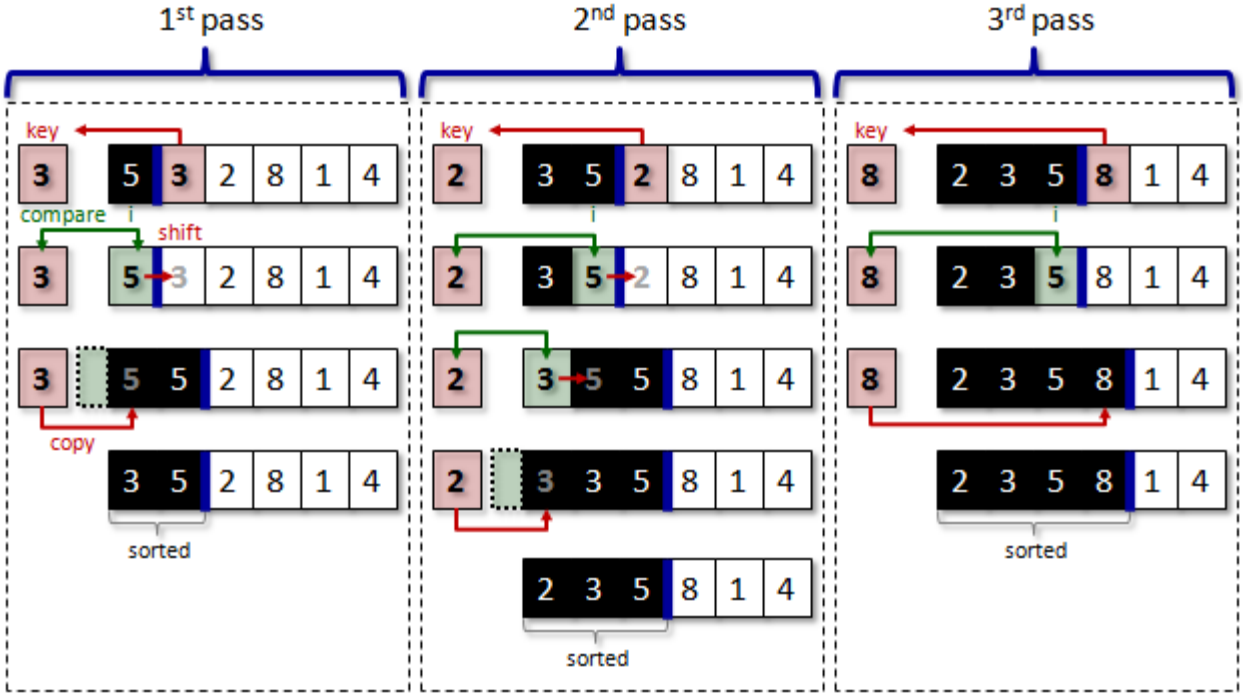
```
Algorithm: SelectionSort
        items:    the array containing the items to sort
        N:        the size of the array

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

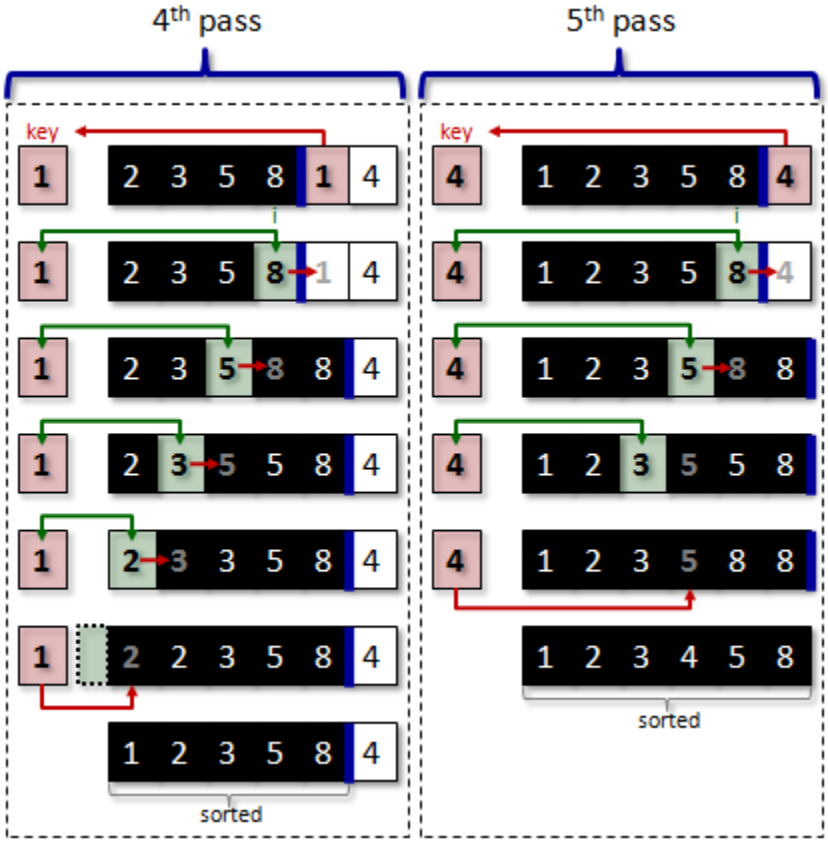
6.4 Insertion Sort

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 $\mathbf{N^2}$, 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
        N:        the size of the array

1.      for each pass p from 1 to N-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

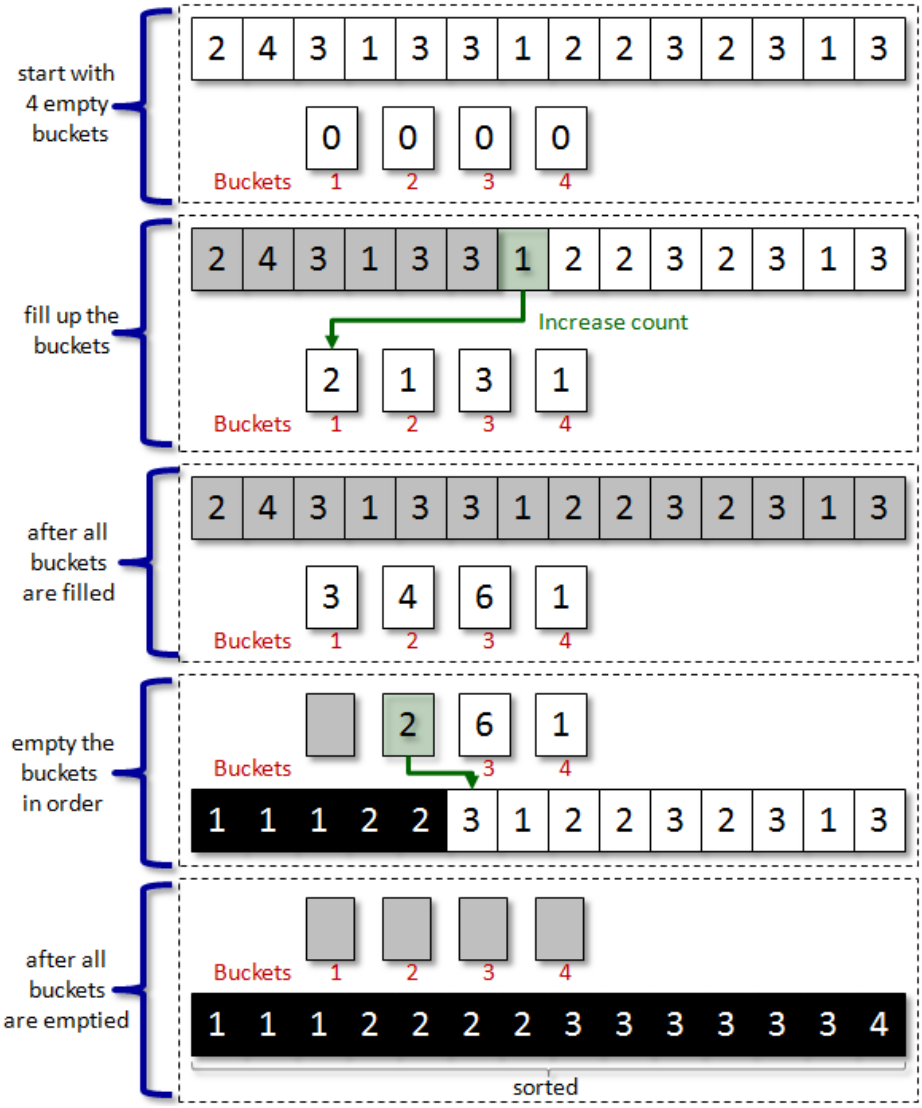
Insertion Sort takes about half the speed of a Selection Sort ... due to the ability of the **while** loop to exit earlier. So, in general, the Insertion Sort is a little more efficient.

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 here. 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: CountingSort
        items:    the array containing the items to sort
        N:        the size of the array
        b:        the number of bins to use

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

2.      for each item i from 0 to N-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
            }
        }
```

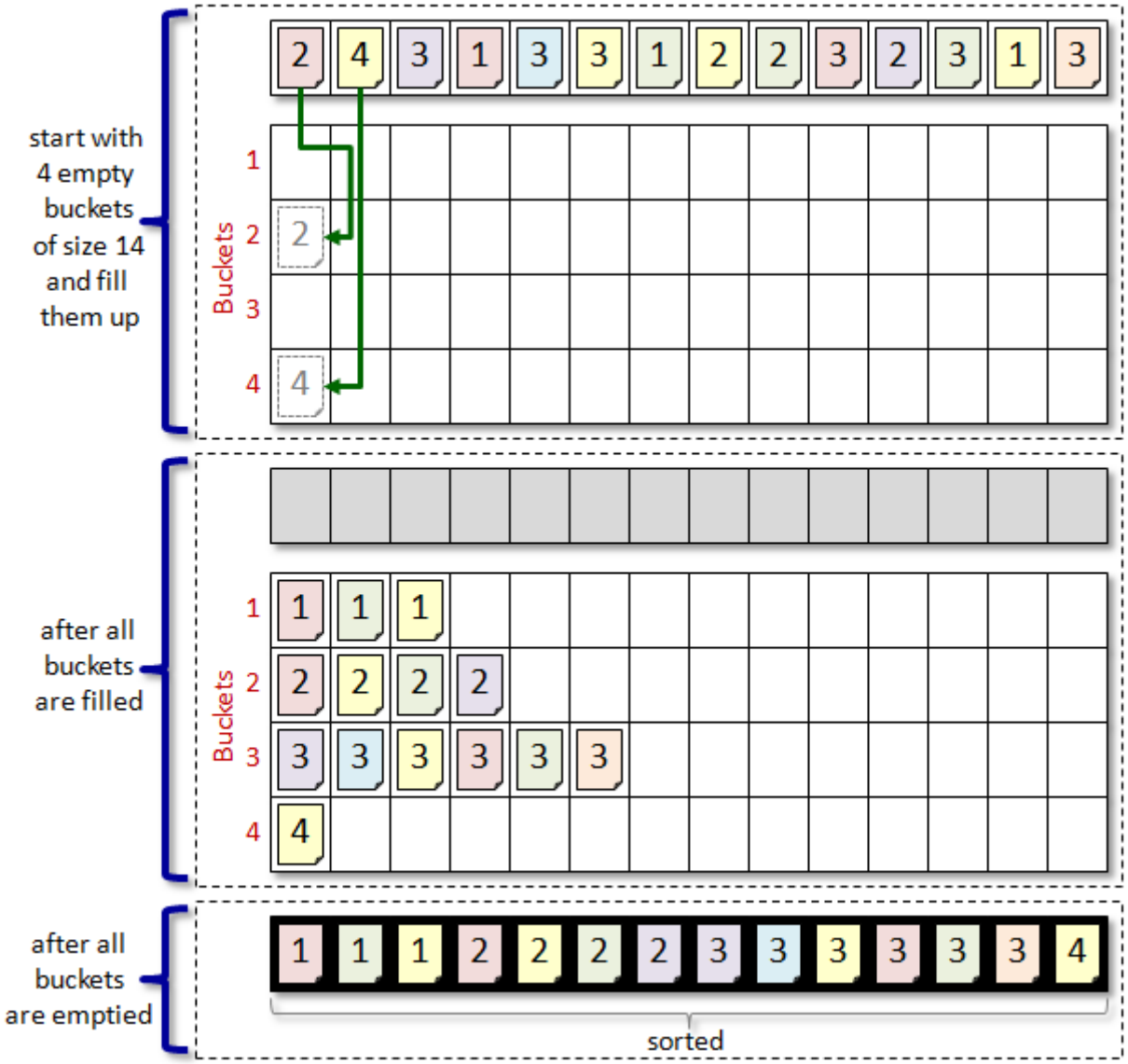
Notice in steps 2 and 3 how the bins array simply stores a count of how many items had that bin's value. Then in steps 4 through 8 we simply go through each bin and fill in the items array with the correct number of items from the bins.

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. On the next page is a diagram that shows 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.

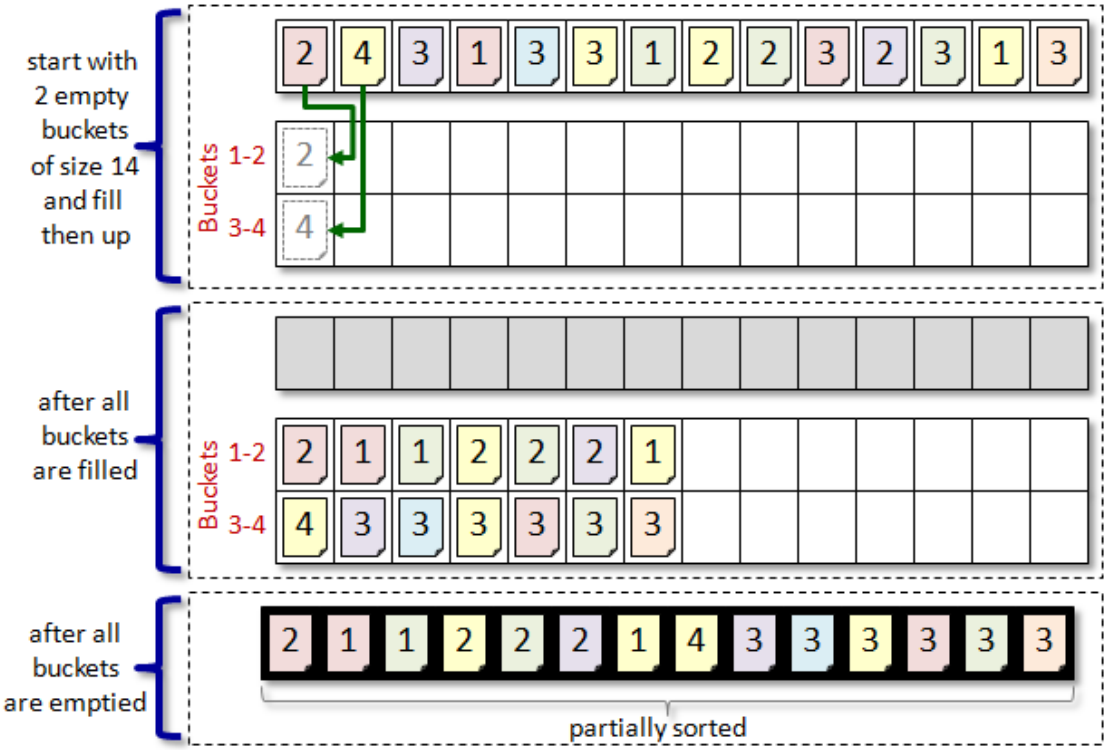


In our exam paper example, the 2D array would need to have space to store **101·1000** = **101,000** exam papers, although only **1,000** exam papers would actually be stored!!! That is about **99%** of wasted space! Of course, there are ways to fix this, as you will learn in your 2nd year here in computer 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:

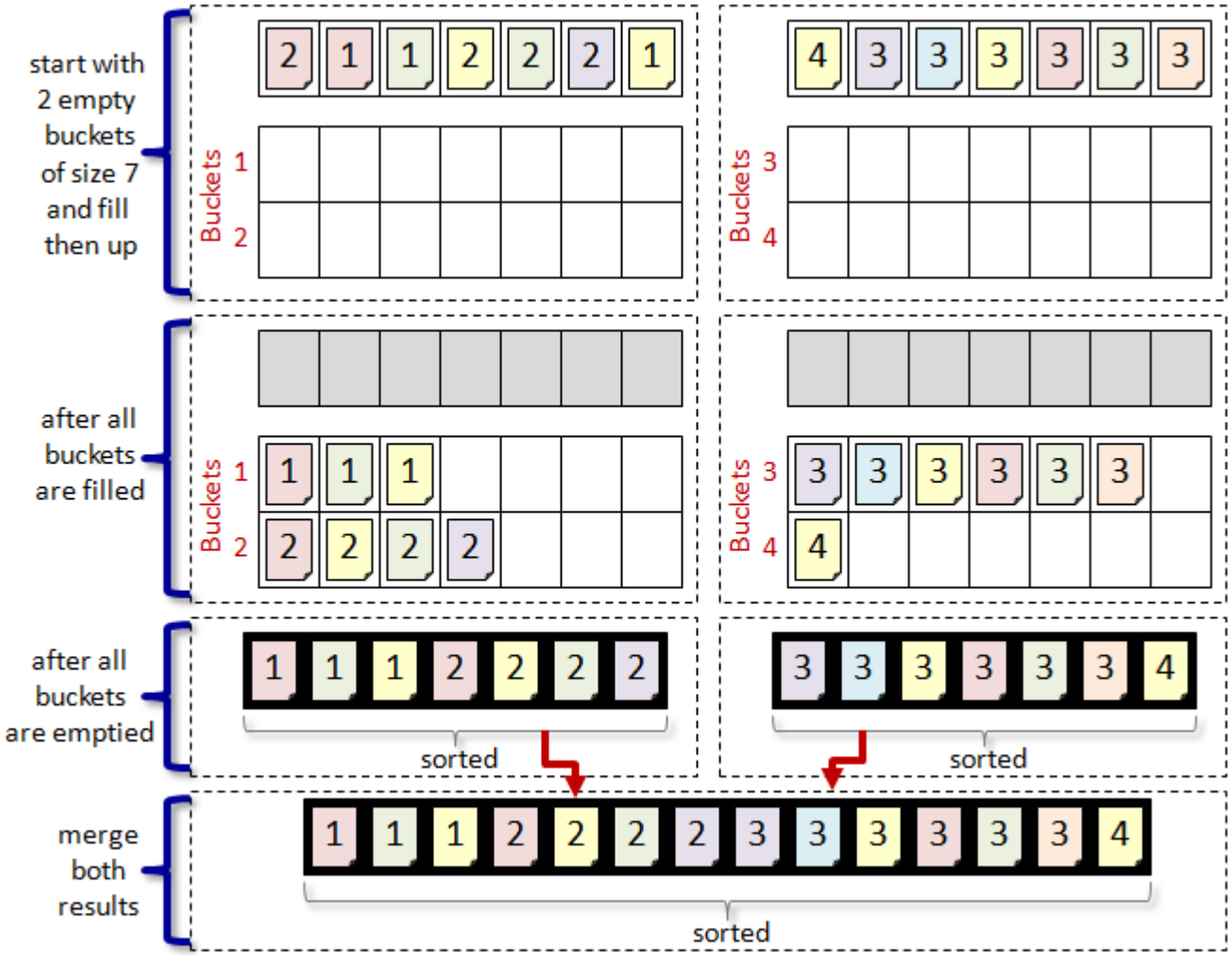
```
Algorithm: BucketSort1
        items:    the array containing the items to sort
        N:        the size of the array
        b:        the number of bins to use

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

3.      for each item i from 0 to N-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:

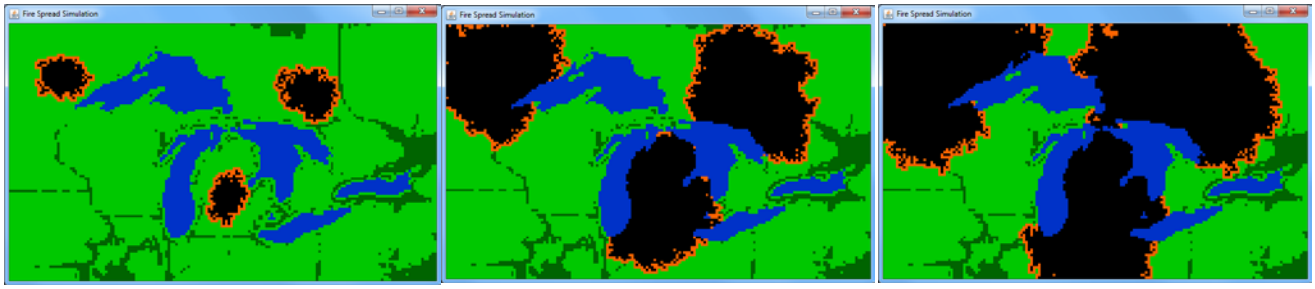
```
Algorithm: BucketSort2
        items:    the array containing the items to sort
        N:        the size of the array
        b:        the number of bins to use

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

3.      for each pass i from 0 to N-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 there are at least 2 bins. It can be very efficient when there are many equal items.

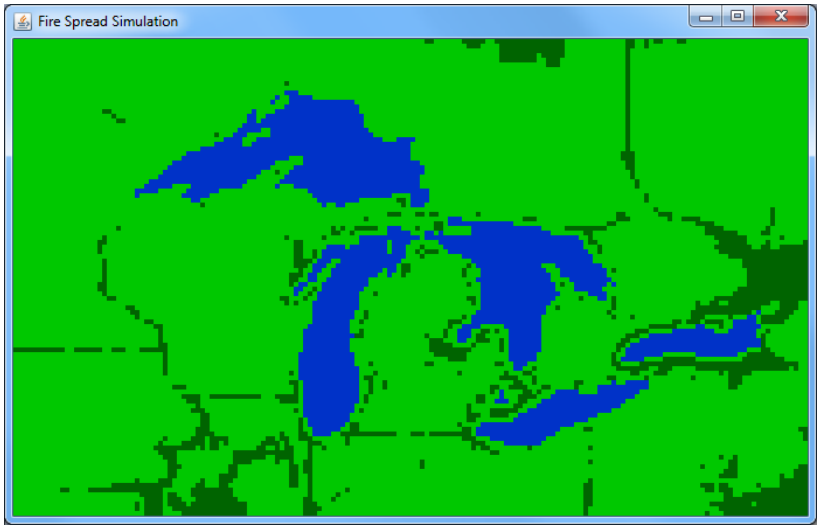
6.6 Sorting Application - Fire Spread Simulation

Consider another example of where sorting is necessary in order to simulate. Assume that we want to simulate the spread of a fire across a terrain. By simulating a fire-spread scenario, we can get a better idea as to which areas need to be evacuated first, the order in which multiple fires must be extinguished, and we can better assess the environmental impact that the fire will have. Rest assured that no animals will be harmed in this simulation :).

Here is an example of what we are trying to do. The screen snapshots show 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 in our simulation. Here is a display of the terrain that we will use. It is represented as a 2D array of 170 (width) x 102 (height) "grid cells".



We can create this array in our code, as an array of bytes as follows:

```
public static final int TERRAIN_WIDTH = 170;
public static final int TERRAIN_HEIGHT = 102;

public static byte[][] terrain = new byte[TERRAIN_HEIGHT][TERRAIN_WIDTH];
```

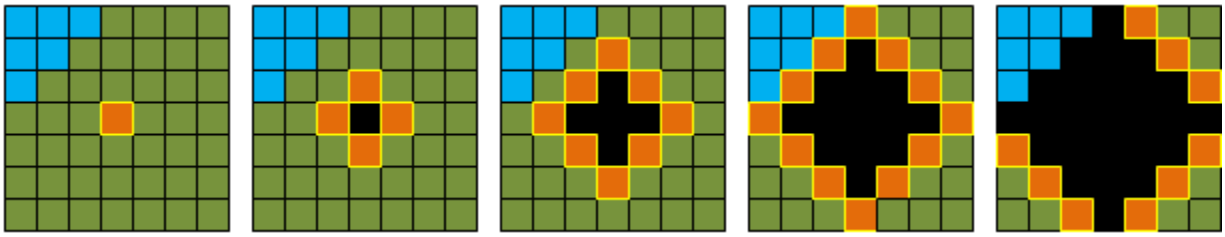
Each grid cell represents either grass, forest or water. As the fire spreads, we will need two more cell types ... one representing a cell on fire, the other representing a cell that is burned forest. We will define the following constants values so that we can use them in our code:

```
public static final byte GRASS  = 1;
public static final byte TREES  = 2;
public static final byte WATER  = 3;
public static final byte FIRE   = 4;
public static final byte BURNED = 0;
```

We can simulate the spreading of fires by changing appropriate pixels in the terrain to orange for a while and then to 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 itself a kind of *miniature fire* that needs to be spread outwards.

Therefore, we will need to keep track of this active border which will contain all locations that are currently on fire so that we can propagate the fire outwards from each of these locations. The active fire borders can be simply represented as (x,y) locations in the terrain.

We can store the currently active border in an array. But how big should it be ? Notice in image (a) above the active border starts with only one cell, but in the next images (b),(c),(d), the border increases to 4, 8 and 12. In the final image (e), the border shrinks back to size 9 since some fire was extinguished by the water. In general, the active border will shrink and grow over time. We will need to choose an array size that is big enough to hold the border, but not too big so as to waste space. Likely the size of the border will depend on the terrain size.

If the entire 107x102 cell terrain was on fire ... that would require 17,340 fire cells on the active border. 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. We will set the active order to around 5% of the terrain... at a size of about 1,000 cells ... which should be enough ... but this is just a rough number that we can change if we find that more or less space is needed.

We will use two arrays to store the active border, one to store the **X** cell value and one for the **Y** cell value. We will also need to know how many active cells are in the arrays ... so we'll need to keep the **borderSize**:

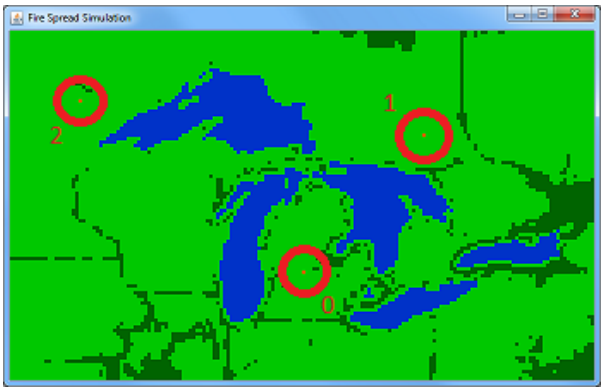
```
public static int[]    borderX;       // X value of cells along the border of the fire
public static int[]    borderY;       // Y value of cells along the border of the fire
public static int      borderSize;    // # points along the border of the fire
```

Here is the code to create the arrays:

```
borderX = new int[1000];
borderY = new int[1000];
```

To begin a fire, we just add the fire's starting location as the only active cell in the array:

```
borderX[0] = 85;
borderY[0] = 70;
borderSize = 1;
```



We can add more fires easily:

```
borderX[1] = 120;
borderY[1] = 30;
borderSize++;

borderX[2] = 20;
borderY[2] = 20;
borderSize++;
```

So now that we know the active border at any time, how do we "grow" the fires ? Here is the basic algorithm:

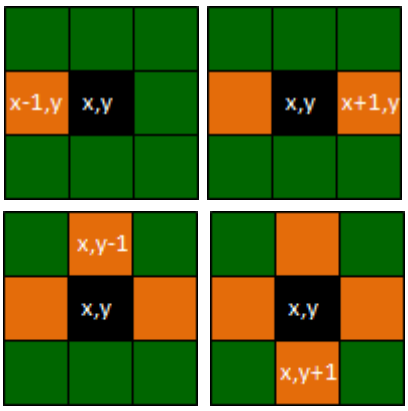
while (there are still fires on the active border) {

Get a cell from the active border
and mark it as being **BURNED**

Get the 4 neighboring cells around that cell
and mark them as being **FIRE**

Add the 4 neighboring cells to the active border

}



So, basically, each round of fire spreading (i.e., each round through the loop) simply takes just ONE cell that is on fire and then spreads that fire outwards by adding its neighboring cells to the active border. These neighboring cells will be processed at a later time. Once there are no more cells on the active border, then nothing is on fire anymore and the simulation is done.

Here is more detailed pseudocode for the algorithm which makes use of our array variables:

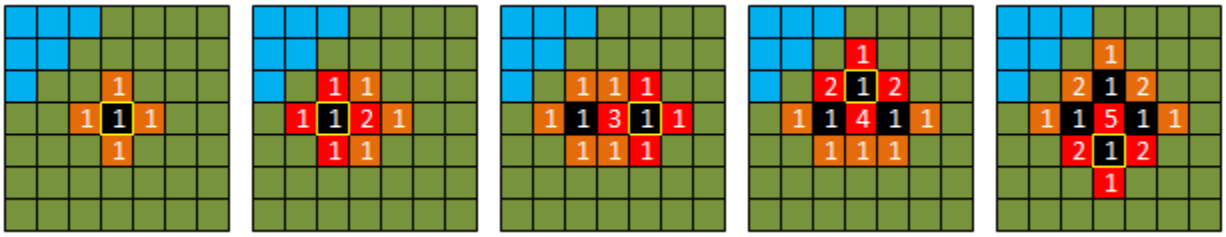
```
while (borderSize > 0) {
    // Grab a cell on the active border, and mark it as burned
    x = borderX[borderSize-1]
    y = borderY[borderSize-1]
    borderSize = borderSize - 1
    terrain[x][y] = BURNED

    // Now spread the fire in all 4 directions around cell (x,y)
    borderX[borderSize] = x-1              // Add (x-1, y) to  the active border
    borderY[borderSize] = y
    borderSize = borderSize + 1
    terrain[x-1][y] = FIRE

    borderX[borderSize] = x+1              // Add (x+1, y) to  the active border
    borderY[borderSize] = y
    borderSize = borderSize + 1
    terrain[x+1][y] = FIRE

    borderX[borderSize] = x                // Add (x, y-1) to  the active border
    borderY[borderSize] = y-1
    borderSize = borderSize + 1
    terrain[x][y-1] = FIRE

    borderX[borderSize] = x                // Add (x, y+1) to  the active border
    borderY[borderSize] = y+1
    borderSize = borderSize + 1
    terrain[x][y+1] = FIRE
}
```

We need to make sure however, that we do not process any pixels outside the valid array range. For example, if the border point is the top left cell in the terrain (i.e., (0, 0)), 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 change each **IF** statement as follows:

```
if (x > 0) {                           // make sure not to go beyond left border
    borderX[borderSize] = x-1          // Add (x-1, y) to  the active border
    borderY[borderSize] = y
    borderSize = borderSize + 1
    terrain[x-1][y] = FIRE
}

if (x < TERRAIN_WIDTH+1) {             // make sure not to go beyond right border
    borderX[borderSize] = x+1          // Add (x+1, y) to  the active border
    borderY[borderSize] = y
    borderSize = borderSize + 1
    terrain[x+1][y] = FIRE
}

if (y > 0) {                           // make sure not to go beyond top border
    borderX[borderSize] = x            // Add (x, y-1) to  the active border
    borderY[borderSize] = y-1
    borderSize = borderSize + 1
    terrain[x][y-1] = FIRE
}
```

```
if (y < TERRAIN_HEIGHT - 1) {          // make sure not to go beyond bottom border
    borderX[borderSize] = x            // Add (x, y+1) to  the active border
    borderY[borderSize] = y+1
    borderSize = borderSize + 1
    terrain[x][y+1] = FIRE
}
```

At this point, we still have a slight problem. When we extract an active border cell, we need to ensure that we don't re-propagate through the same points over and over again (i.e., through points that are on FIRE or BURNED), 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., FIRE) or if there is a burned area there (i.e., BURNED) or if there is water there (i.e., WATER).

We can write a function that determines whether or not a cell is able to be burned:

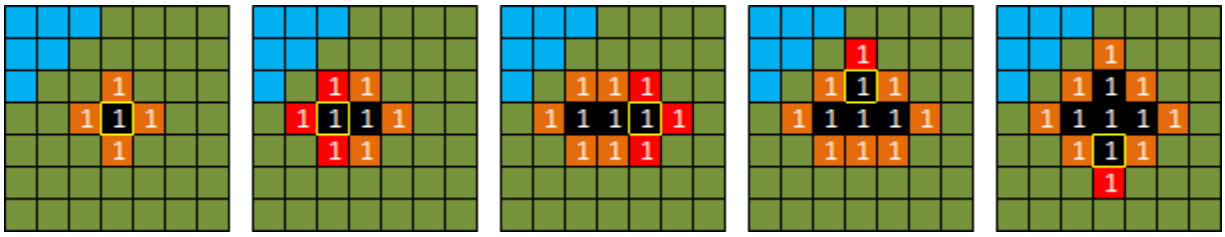
```
public static boolean isBurnable(int y, int x) {
      return (terrain[y][x] == GRASS) || (terrain[y][x] == TREES);
}
```

Then, we can make use of this function by checking before we start propagating the fire in a new direction. Here are the modifications:

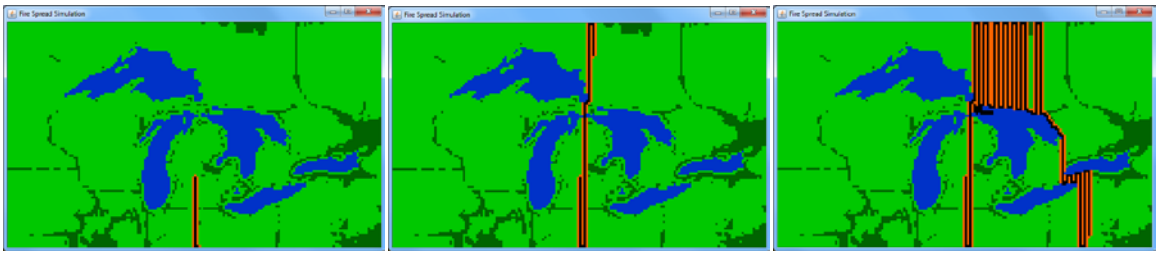
```
if ((x > 0) && isBurnable(y, x-1)) {
    borderX[borderSize] = x-1
    borderY[borderSize] = y
    borderSize = borderSize + 1
    terrain[x-1][y] = FIRE
}

if ((x < TERRAIN_WIDTH+1) && isBurnable(y, x+1)) {
    borderX[borderSize] = x+1
    borderY[borderSize] = y
    borderSize = borderSize + 1
    terrain[x+1][y] = FIRE
}

if ((y > 0)  && isBurnable(y-1, x)) {
    borderX[borderSize] = x
    borderY[borderSize] = y-1
    borderSize = borderSize + 1
    terrain[x][y-1] = FIRE
}
```

```
if ((y < TERRAIN_HEIGHT - 1)  && isBurnable(y+1, x))  {
    borderX[borderSize] = x
    borderY[borderSize] = y+1
    borderSize = borderSize + 1
    terrain[x][y+1] = FIRE
}
```

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



If we were to run our code, this is what we would see now:



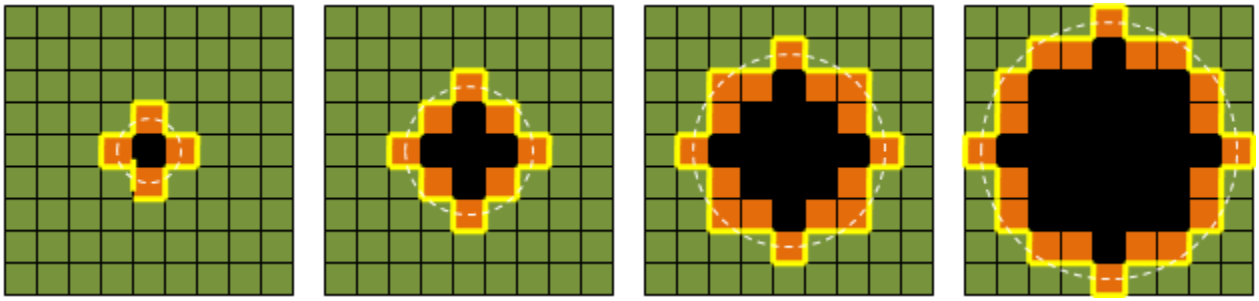
What is going on ! That's not a proper fire spread simulation! It seems that the fire only spreads downwards in a straight line, then it goes one cell to the right and then spreads only upwards in a straight line, forming a zig-zag pattern. The program even crashes after a while!

Do you understand the problem ?

We are not processing the terrain cells in the correct manner. We need to process them outwards from the starting location in all directions equally. Our current code favors going downwards first. That is because our last **IF** statement always adds the cell below (x,y)... that is (x, y+1). Because our code always extracts and processes the last border cell added (i.e., x = **borderX**[**borderSize**-1], y = **borderY**[**borderSize**-1]), we are always taking and expanding the bottom cell's first. Therefore the fire spreads down in a straight line. If we re-order the **IF** statements, we can get the fire to favor any of the 4 directions. Regardless, the result is wrong!

Clearly, this is not a realistic fire-spread model as fires do not spread in straight-line zig-zag formation!! In order to make the spreading realistic, we need to cause the fire to spread outwards by processing the cells outwards in a circular pattern from the starting fire location. This is also known as the ***wavefront propagation*** technique, which is similar to what happens when a pebble is dropped into water ... waves form from the center and move/propagate outwards.

With our terrain, we need to simulate these "waves" of fire outwards from the center:



To do this, we need to ensure that active border cells that are **closest** to the center are processed first. To do this, we would need to **sort** the border points by their distance from the starting location of the fire. This is where sorting is required to achieve the solution that we want.

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

To represent these distances, we can have each cell on the active border maintain its distance from the fire's center. We need another array to keep this cost to each cell as follows

```
public static int[]    borderCost;      // Cost of cells along the border of the fire
```

Here is the code to create the array:

```
borderCost = new float[1000];
```

We will also need to set the cost to **0** to begin the fire since the distance to the fire is zero at that cell:

```
borderCost[0] = 0;
```

Finally, we will need to remember the start location for the fire:

```
int      startX = borderX[0];
int      startY = borderY[0];
```

Here are the changes to the algorithm:

```
while (borderSize > 0) {
    x = borderX[borderSize-1]
    y = borderY[borderSize-1]
    borderSize = borderSize - 1
    terrain[x][y] = BURNED

    if ((x > 0) && isBurnable(y, x-1)) {
        borderX[borderSize] = x-1
        borderY[borderSize] = y
        borderCost[borderSize] = DIST(startX, startY, x-1, y)        // Cost from (startX, startY) to (x-1, y)
        borderSize = borderSize + 1
        terrain[x-1][y] = FIRE
```

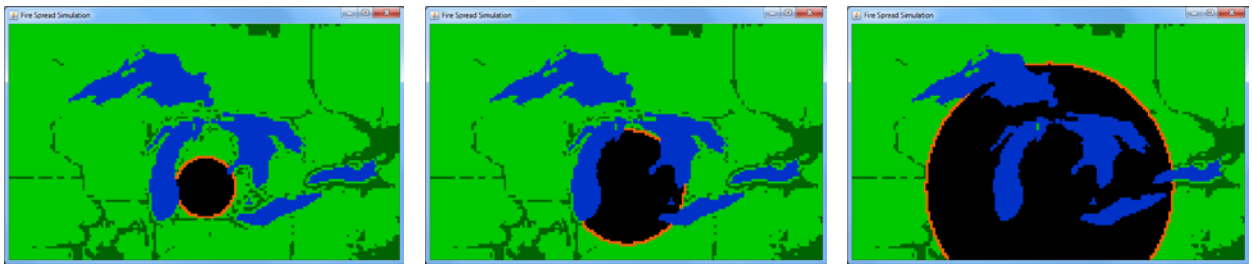
```
        }

        if ((x < TERRAIN_WIDTH+1) && isBurnable(y, x+1)) {
            borderX[borderSize] = x+1
            borderY[borderSize] = y
            borderCost[borderSize] = DIST(startX, startY, x+1, y)        // Cost from (startX, startY) to (x+1, y)
            borderSize = borderSize + 1
            terrain[x+1][y] = FIRE
        }

        if ((y > 0)  && isBurnable(y-1, x)) {
            borderX[borderSize] = x
            borderY[borderSize] = y-1
            borderCost[borderSize] = DIST(startX, startY, x, y-1)        // Cost from (startX, startY) to (x, y-1)
            borderSize = borderSize + 1
            terrain[x][y-1] = FIRE
        }

        if ((y < TERRAIN_HEIGHT - 1)  && isBurnable(y+1, x))  {
            borderX[borderSize] = x
            borderY[borderSize] = y+1
            borderCost[borderSize] = DIST(startX, startY, x, y+1)        // Cost from (startX, startY) to (x, y-1)
            borderSize = borderSize + 1
            terrain[x][y+1] = FIRE
        }

        // Resort the active border cells again (from lowest cost to largest cost) , since we just added some
        ReSort(borderX, borderY, borderCost)
    }
```

Regarding the sorting, any sorting algorithm will work, as long as the sorting is done with respect to the distances stored in each border cell. We can try a simple bubble sort. We would need to sort by the **borderCost** values. Here is a simple function to do this:

```
// BubbleSort algorithm to sort all border points.   Most of the points are
// already sorted, but we just added a few new ones by expanding one fire point.
// These (up to 8) new points need to be in the correct spot in the sorted list.
public static void BubbleSort() {
    for (int p=borderSize-1; p>=0; p--) {
        boolean madeSwap = false;
        for (int i=0; i<=p-1; i++) {
            if (borderCost[i] < borderCost[i+1]) {
                int tempX = borderX[i+1];
                int tempY = borderY[i+1];
                float tempCost = borderCost[i+1];
                borderX[i+1] = borderX[i];
                borderY[i+1] = borderY[i];
                borderCost[i+1] = borderCost[i];
                borderX[i] = tempX;
                borderY[i] = tempY;
                borderCost[i] = tempCost;
                madeSwap = true;
            }
        }
        if (!madeSwap) return;
    }
}
```

The code follows from the algorithm discussed previously.

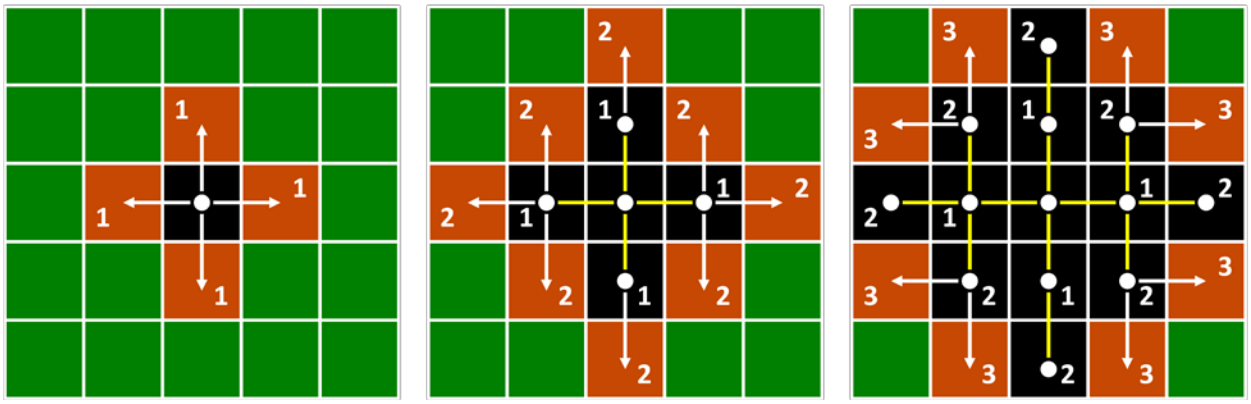
Now, the above code will produce a nice "round" fire spread ... however ... the fire is still not spreading realistically. Can you see what is wrong in the following simulation images?



Notice particularly the 3rd image. The fire border is spreading evenly as if the fire burned right through the lake at the same rate of the forest. This is not realistic, because it would take a longer time for a fire to wrap itself around to the other side of the lake. Therefore ... 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 (e.g., lakes). So, our idea of sorting solely with respect to the starting fire location will not produce a realistic fire spread simulation.

Instead, 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 cell that it spawned from, not always from the starting fire location. Besides, the above code assumes that only one fire is burning. If multiple fires started at different locations, how would we know which starting location to compute the distance from ? We don't know which fire reach that cell first.

So, we need to treat each cell on fire as a unique fire and spread the cost from that fire to the cell beside it. Notice how we need to count the costs as the fire spreads:



Each time we work on a new and larger "wave", the costs of the cells increases by one. We do this by taking the cost of the cell that we just expanded, adding one to it, and then setting that to be the cost of the new cell that we just caught on fire. Therefore, each of these lines:

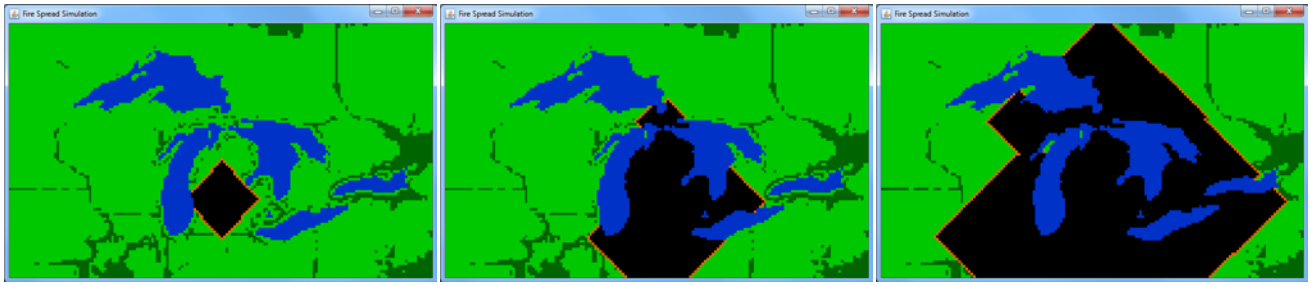
```
borderCost[borderSize] = DIST(startX, startY, x-1, y)
borderCost[borderSize] = DIST(startX, startY, x+1, y)
borderCost[borderSize] = DIST(startX, startY, x, y-1)
borderCost[borderSize] = DIST(startX, startY, x, y+1)
```

becomes this line:

```
borderCost[borderSize] = DIST(startX, startY, x, y) + 1      // Cost from (startX, startY) to (x, y) plus 1
```

This will allow each new fire cell to have a distance value that is *"1 more"* than the previous distance value.

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 left). However, you may notice in the successive images above that the fire will spread in a diamond-like pattern. Here is what it would look like as we continued this process:

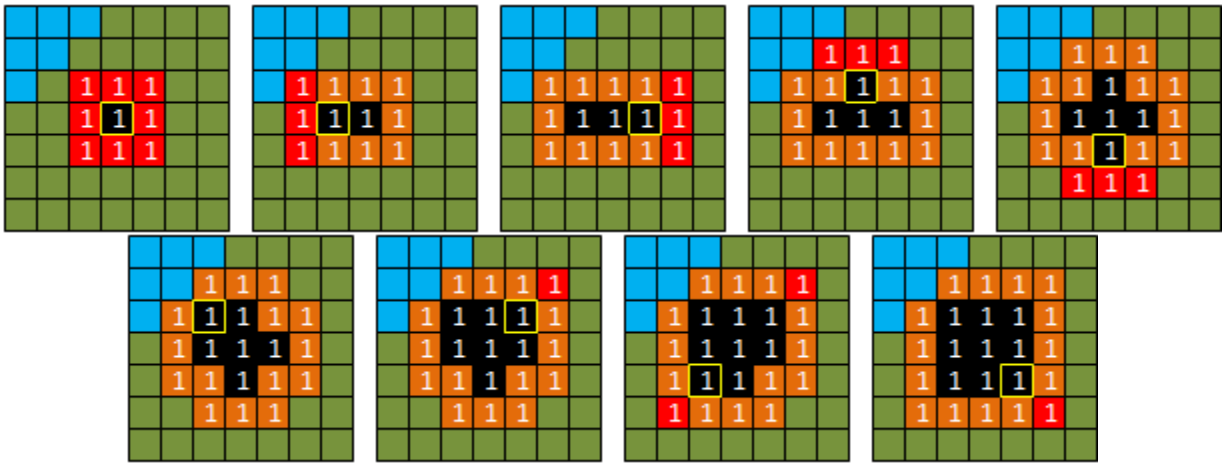


This pattern is called an *artifact* of the simulation.

> *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.*

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***.

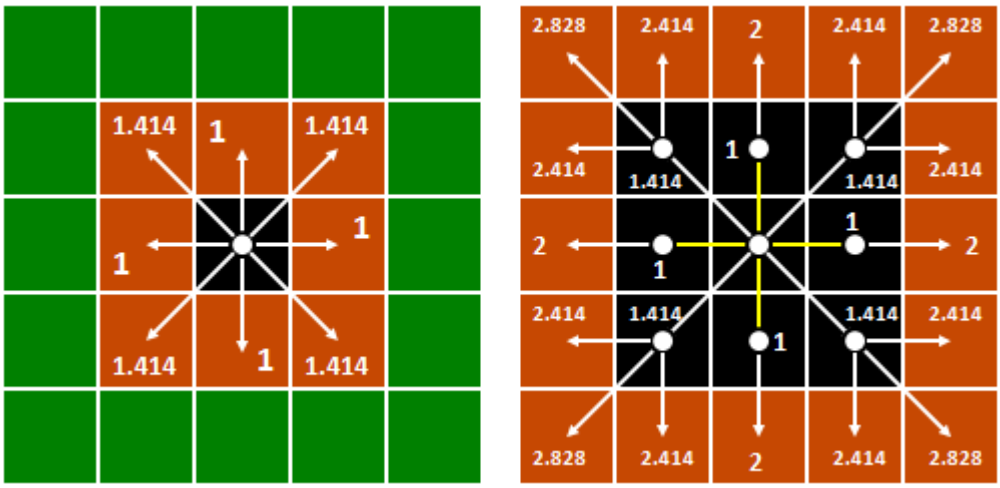
The diamond-shape can be adjusted to an octagonal shape if a ***Moore neighborhood*** is used. A Moore neighborhood includes the diagonal pixels around a pixel. To use this *8-pixel* update model, we would need to add additional code to add border points to the 8 surrounding points. Here is how the fire would spread in this case:



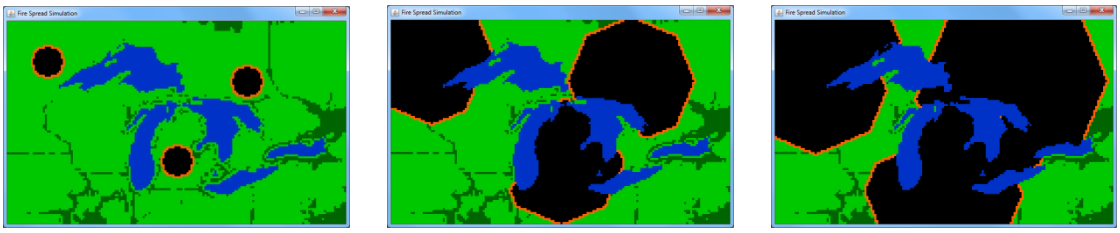
To accomplish this, we simply need to add the following 4 sets of **IF** statements to the end of our existing code ... making a total of 8 **IF** statements:

```
if ((x > 0) && isBurnable(y-1, x-1)) {                    // Upper left
    borderX[borderSize] = x-1
    borderY[borderSize] = y-1
    borderCost[borderSize] = DIST(startX, startY, x, y) + 1.414
    borderSize = borderSize + 1
    terrain[x-1][y-1] = FIRE
}

if ((x < TERRAIN_WIDTH+1) && isBurnable(y-1, x+1)) {      // Upper right
    borderX[borderSize] = x+1
    borderY[borderSize] = y-1
    borderCost[borderSize] = DIST(startX, startY, x, y) + 1.414
    borderSize = borderSize + 1
    terrain[x+1][y-1] = FIRE
}

if ((y > 0)  && isBurnable(y+1, x-1)) {                   // Bottom left
    borderX[borderSize] = x-1
    borderY[borderSize] = y+1
    borderCost[borderSize] = DIST(startX, startY, x, y) + 1.414
    borderSize = borderSize + 1
    terrain[x-1][y+1] = FIRE
}

if ((y < TERRAIN_HEIGHT - 1)  && isBurnable(y+1, x+1))  {  // Bottom right
    borderX[borderSize] = x+1
    borderY[borderSize] = y+1
    borderCost[borderSize] = DIST(startX, startY, x, y) + 1.414
    borderSize = borderSize + 1
    terrain[x+1][y+1] = FIRE
}
```

Notice that the cost to the diagonals is actually 1.414 ... which is the square root of 2. This is the diagonal distance from the middle of the center cell to the middle of the diagonal cell. Here is how the costs now spread:



If you run the code, the result would now be growth that is ***octagonal*** ... a more realistic approximation, although the octagonal shape is still somewhat "artificial-looking". Notice though, how the fire still bends nicely around the lakes:



We can make the fire 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:

```
borderCost[borderSize] = DIST(startX, startY, x, y) + 1
```

becomes this line of code:

```
borderCost[borderSize] = DIST(startX, startY, x, y) + 1 + rand(3)
```

where **rand(3)** is a randomly chosen number which is either 0, 1 or 2.

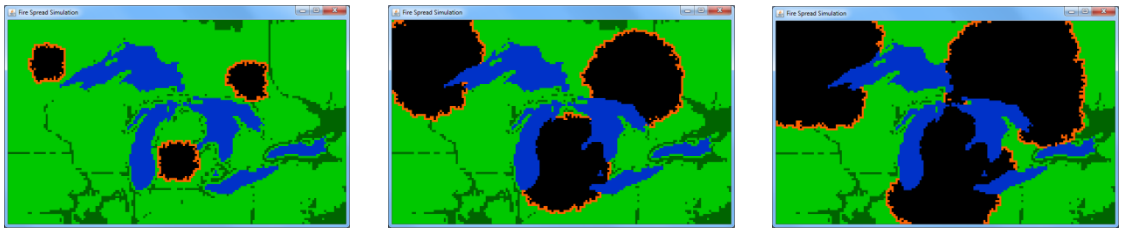
And this line:

```
borderCost[borderSize] = DIST(startX, startY, x, y) + 1.414
```

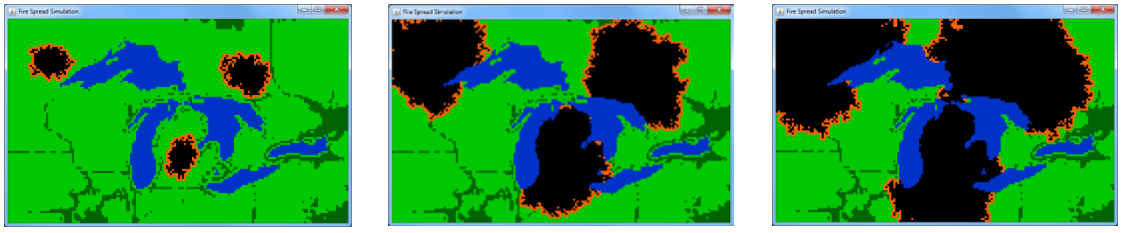
becomes this line of code:

```
borderCost[borderSize] = DIST(startX, startY, x, y) + 1.414 + rand(3)
```

Here is the result of this effort:



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:



For added realism, 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...

Below is the final code. It makes use of a **TerrainPanel** class to display the terrain. This is a topic for COMP1406 and is not discussed further in this course. The terrain itself is hard-coded into a String array (many lines have been omitted here but the full code is online).

```
import java.awt.*;         // Needed for window and graphics (explained in COMP1406)
import javax.swing.*;      // Needed for window and graphics (explained in COMP1406)

// This application displays an image
public class FireSpreadSimulation {
    // This variable stores the panel that displays the fire
    // (This is a topic discussed in COMP1406 course)
    public static TerrainPanel   terrainPanel;

    // These are the codes that indicate what is at each position in the array
    public static final byte GRASS  = 1;
    public static final byte TREES  = 2;
    public static final byte WATER  = 3;
    public static final byte FIRE   = 4;
    public static final byte BURNED = 0;

    // This is the 170 x 102 cell terrain where each cell corresponds to
    // a terrain type as is shown with the above 5 possible byte codes.
    public static String[] terrainData = {
      "11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
       11221111112222211222222222222222111111111111221121221112211111111111111111111111111111",

          ... etc ... many, many lines of code have been omitted to save space ...

      "11111111111111111111111111122221111222222222222221111111111112221111111111111111111111
       11111111111111111111111111111111111112111111111111111111111111112111222222222222222222"
    };

    // These variables store the width and height of the terrain
    public static int terrainWidth = terrainData[0].length();
    public static int terrainHeight = terrainData.length;

    // This is the terrain itself, which is made from the above String array.
    public static byte[][] terrain = new byte[terrainHeight][terrainWidth];


    // This code represets the active border of all fires
    public static int[]    borderX;       // points along the border of the fire
    public static int[]    borderY;       // points along the border of the fire
    public static float[]  borderCost;    // points along the border of the fire
    public static int      borderSize;    // # points along the border of the fire
```

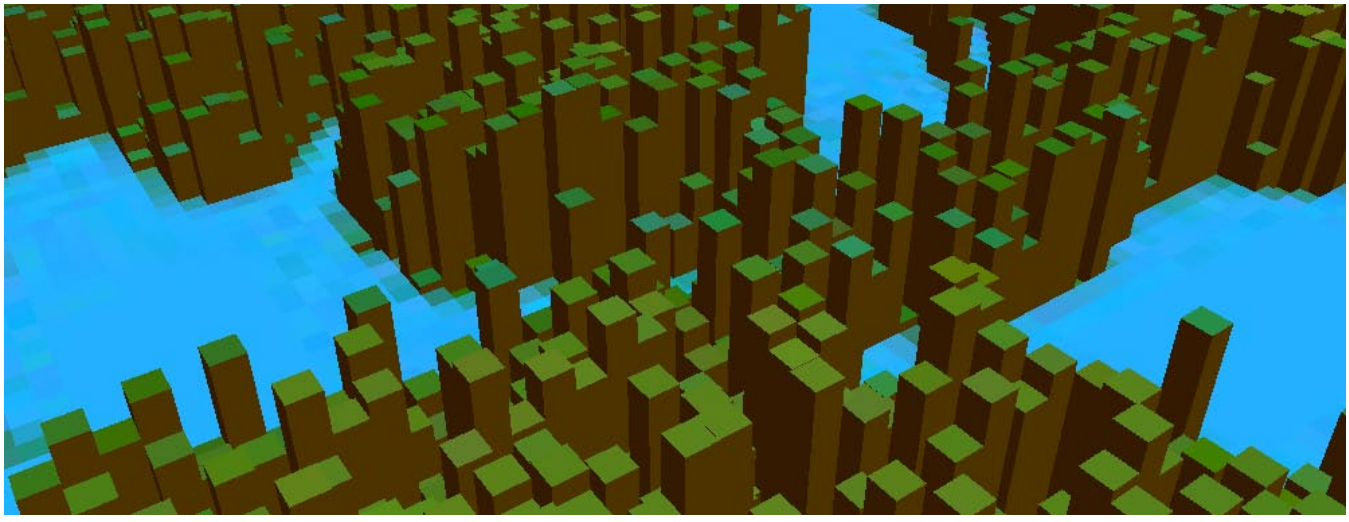
```
    public static void startSimulation() {
        // Fill up the terrain array
        for (int i=0; i<terrainHeight; i++) {
            for (int j=0; j<terrainWidth; j++) {
                terrain[i][j] = (byte)(terrainData[i].charAt(j) - 48);
            }
        }

        // Create an array to store the "border" of the fire as it spreads outwards
        borderX = new int[1000];
        borderY = new int[1000];
        borderCost = new float[1000];

        // To start, we begin with 3 small fires
        borderX[0] = 85;
        borderY[0] = 70;
        borderCost[0] = 0;
        borderX[1] = 120;
        borderY[1] = 30;
        borderCost[1] = 0;
        borderX[2] = 20;
        borderY[2] = 20;
        borderCost[2] = 0;
        borderSize = 3;

        // Now simulate the fire until it cannot spread any further
        while(borderSize > 0) {
            terrainPanel.repaint(); // This code redraws the window
            spreadFire();           // This code simulates the fire spreading
            try{ Thread.sleep(2); } catch(Exception e){}; // To slow things down a bit
        }
    }


     // Function that determines if a cell can be burned
    public static boolean isBurnable(int y, int x) {
        return (terrain[y][x] == GRASS) || (terrain[y][x] == TREES);
    }


    // Simulate the spreading of the fire in each direction from the oldest fire point
    public static void spreadFire() {

        // Grab the last border point (after sorted).  It should be the oldest
        // border of the fires ... which should be processed first
        int pX = borderX[borderSize-1];
        int pY = borderY[borderSize-1];
        float pCost = borderCost[borderSize-1];

        // Extingush the fire at this location by marking the terrain as burned
        terrain[pY][pX] = BURNED;
        borderSize--;

        // Now we will check all around this extinguished pixel to spread
        // the fire outwards in all 8 directions, if possible


        // Check left
        if ((pX > 0) && isBurnable(pY,pX-1)) {
            terrain[pY][pX-1] = FIRE;
            borderX[borderSize] = pX-1;
            borderY[borderSize] = pY;
            borderCost[borderSize] = pCost + 1 + (int)(Math.random()*3);
            borderSize++;
        }
```

```
        // Check right
        if ((pX < terrainWidth-1) && isBurnable(pY,pX+1)) {
            terrain[pY][pX+1] = FIRE;
            borderX[borderSize] = pX+1;
            borderY[borderSize] = pY;
            borderCost[borderSize] = pCost + 1 + (int)(Math.random()*3);
            borderSize++;
        }

        // Check up
        if ((pY > 0) && isBurnable(pY-1,pX)) {
            terrain[pY-1][pX] = FIRE;
            borderX[borderSize] = pX;
            borderY[borderSize] = pY-1;
            borderCost[borderSize] = pCost + 1 + (int)(Math.random()*3);
            borderSize++;
        }

        // Check down
        if ((pY < terrainHeight-1) && isBurnable(pY+1,pX)) {
            terrain[pY+1][pX] = FIRE;
            borderX[borderSize] = pX;
            borderY[borderSize] = pY+1;
            borderCost[borderSize] = pCost + 1 + (int)(Math.random()*3);
            borderSize++;
        }

        // Check left/up diagonal
        if ((pX > 0) && (pY > 0) && isBurnable(pY-1,pX-1)) {
            terrain[pY-1][pX-1] = FIRE;
            borderX[borderSize] = pX-1;
            borderY[borderSize] = pY-1;
            borderCost[borderSize] = pCost + 1.414f + (int)(Math.random()*3);
            borderSize++;
        }

        // Check right/up diagonal
        if ((pX < terrainWidth-1) && (pY > 0) && isBurnable(pY-1,pX+1)) {
            terrain[pY-1][pX+1] = FIRE;
            borderX[borderSize] = pX+1;
            borderY[borderSize] = pY-1;
            borderCost[borderSize] = pCost + 1.414f + (int)(Math.random()*3);
            borderSize++;
        }

        // Check left/down diagonal
        if ((pX > 0) && (pY < terrainHeight-1) && isBurnable(pY+1,pX-1)) {
            terrain[pY+1][pX-1] = FIRE;
            borderX[borderSize] = pX-1;
            borderY[borderSize] = pY+1;
            borderCost[borderSize] = pCost + 1.414f + (int)(Math.random()*3);
            borderSize++;
        }

        // Check right/down diagonal
        if ((pX < terrainWidth-1) && (pY<terrainHeight-1) && isBurnable(pY+1,pX+1)) {
            terrain[pY+1][pX+1] = FIRE;
            borderX[borderSize] = pX+1;
            borderY[borderSize] = pY+1;
            borderCost[borderSize] = pCost + 1.414f + (int)(Math.random()*3);
            borderSize++;
        }
        // Now that we have extended the fire along the border,
        // re-sort so that we process the one that is the "oldest"
        BubbleSort();
    }
```

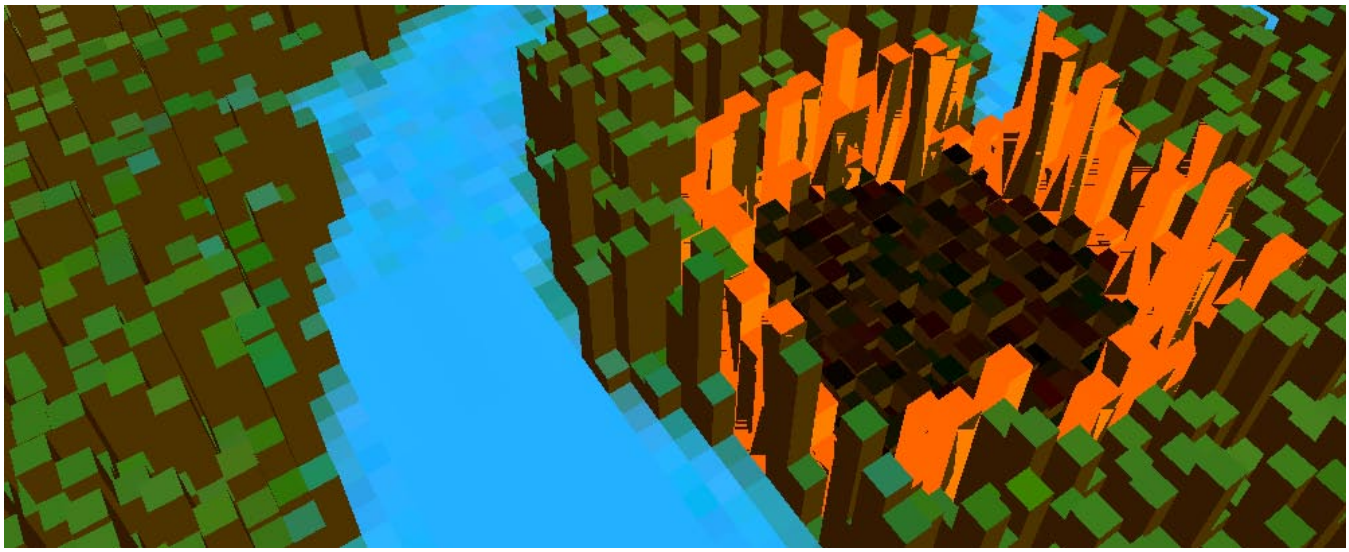
```
    // BubbleSort algorithm to sort all border points.   Most of the points are
    // already sorted, but we just added a few new ones by expanding one fire point.
    // These (up to 8) new points need to be in the correct spot in the sorted list.
    public static void BubbleSort() {
        for (int p=borderSize-1; p>=0; p--) {
            boolean madeSwap = false;
            for (int i=0; i<=p-1; i++) {
                if (borderCost[i] < borderCost[i+1]) {
                    int tempX = borderX[i+1];
                    int tempY = borderY[i+1];
                    float tempCost = borderCost[i+1];
                    borderX[i+1] = borderX[i];
                    borderY[i+1] = borderY[i];
                    borderCost[i+1] = borderCost[i];
                    borderX[i] = tempX;
                    borderY[i] = tempY;
                    borderCost[i] = tempCost;
                    madeSwap = true;
                }
            }
            if (!madeSwap) return;
        }
    }

    // Main method creates a window with a TerrainPanel and then starts the simulation
    public static void main(String args[]) {
        JFrame frame = new JFrame("Fire Spread Simulation");
        frame.add(terrainPanel = new TerrainPanel(terrain));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);

        startSimulation();
    }
}
```

For added enjoyment, we can create a 3D version of the landscape and then watch as the fire spreads, burning the trees. To do this, instead of displaying an image, we just need to create a big pile of square surfaces that corresponds to the terrain.

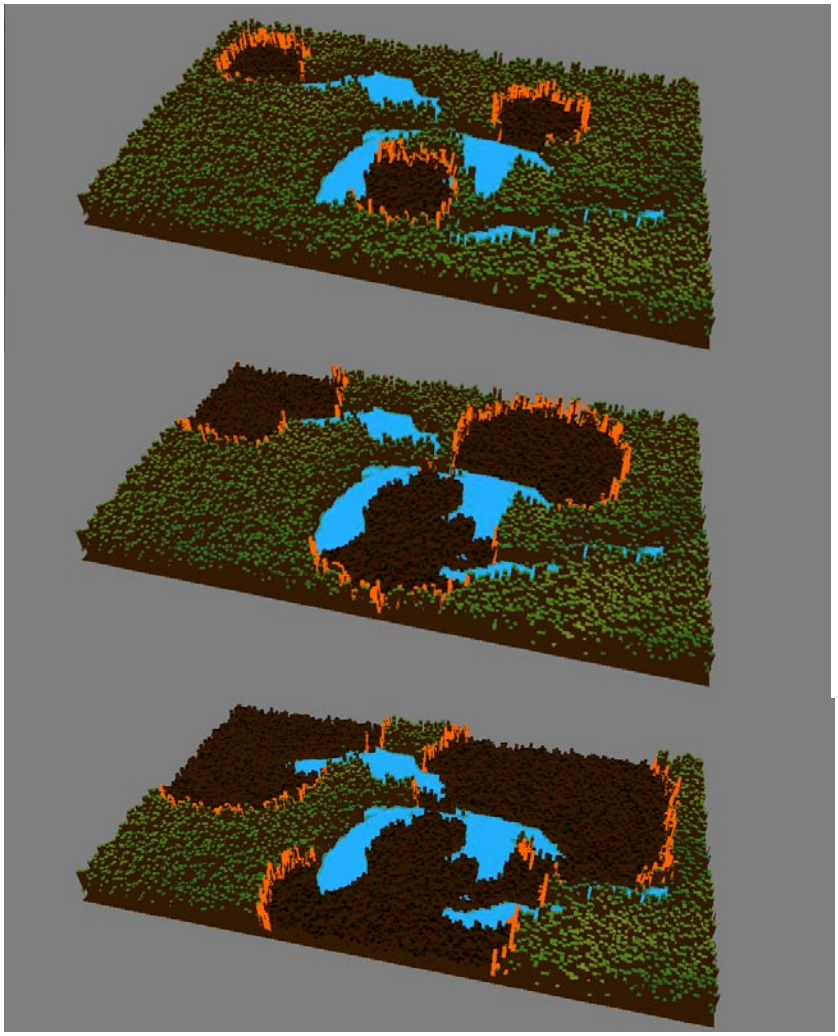
The idea is to simply take the image and assign a height value to each pixel. Water could have height 0 while the trees could have a height which is some constant value with a degree of randomness to provide variety. We could then form a 3D rectangular polyhedra for each pixel and display them ...



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



It looks pretty cool ...



6.7 Efficient Searching Using Arrays

We have discussed linear searching and you likely have realized that it sometimes requires us to iterate through all the items in an array whereas at other times we can stop the iteration once we have found something that positively answers our problem at hand. In the worst case, however, it can require a full search through all the items.

There is another very popular search strategy called a binary search:

> A ***binary search*** is an efficient method for finding a particular value in a sorted list, that consists of continually reducing the search space by half during each search step.

As stated, in order to perform a binary search it is necessary for the elements to be in sorted order. If the elements are out of order, the algorithm will not work.

The idea behind a binary search is easily understood using the analogy of looking up someone's name in the white pages of a phone book. Image that you are looking for the last name "Peterson". If you opened the book in the middle and looked at the top of the page, chances are that you would see a name that comes alphabetically before Peterson (e.g., "Lanthier") or alphabetically after it (e.g., "Ryans"). If you saw "Ryans", for example, then you know that "Peterson" comes before it in the book (since the names are sorted), so you don't bother checking the 2nd half of the book, but instead turn your attention to the first half of the book. Then, the problem starts all over again with the first half of the book in that you will now check from book from "A" to "Ryans", ignoring the portion from "Ryans" to "Z". The process continues until the name is found, each time a big chunk of the phone book is eliminated from the potential pages to search.

The "binary" from "binary search" indicates that the data is repeatedly divided into two halves. How do we find the half-way point ? Well, if the first page is **1** and the last page is **2000**, we just do **(2000 – 1) / 2 = 999.5** which can be rounded up to **1000**. But what about when we are searching the top half of the book … perhaps from page **1000** to **2000** ? Then the same formula of **(2000 – 1000) / 2 = 500** which is not the middle. **500** is actually the number of pages from page **1000** to the middle of the **1000** to **2000** range. So we need to add the offset of the start page. The formula is therefore:

midPage = startPage + (endPage – startPage) / 2

or

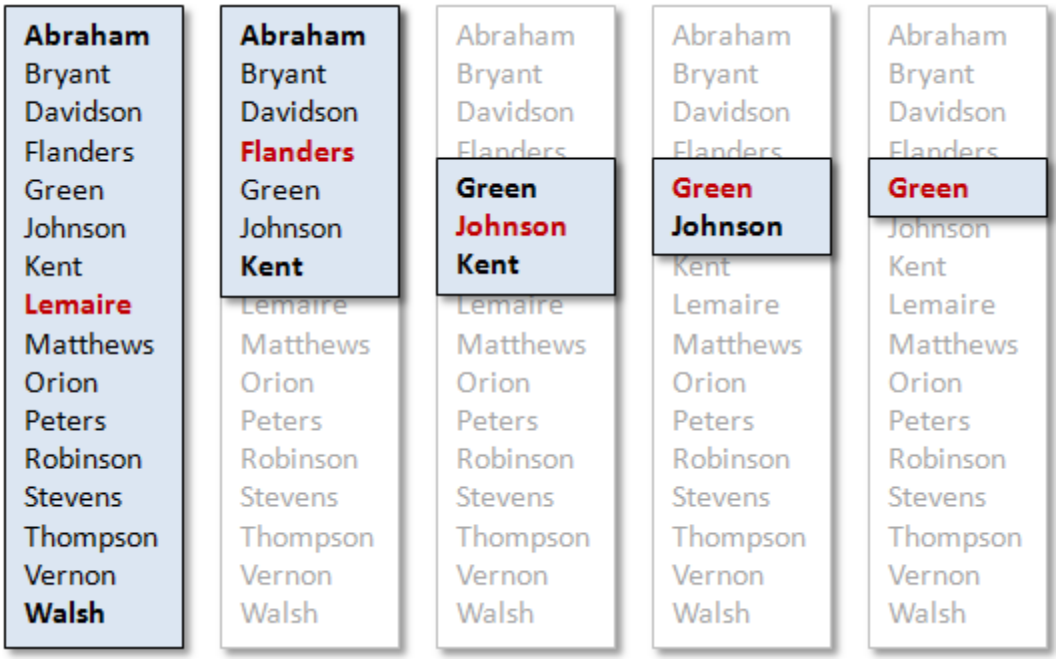
midPage = (startPage + endPage) / 2

Here is the algorithm more formally:

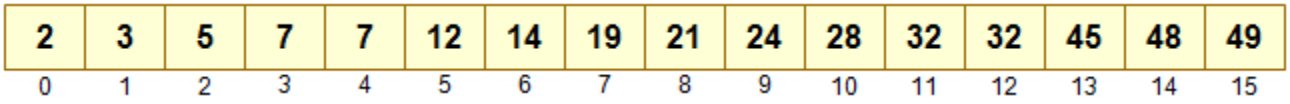
```
Algorithm: SearchPhoneBook
        phonebook:          the book to search through
        searchName:         the name you are looking for

1.      startPage ← 1
2.      endPage ← 2000      // or whatever the last page is
3.      found ← false
4.      while ((not found) and (startPage is not greater than endPage)) {
5.          midPage ← (startPage + endPage) / 2
6.          open the phonebook at midPage
7.          currentName ← name at the top of midPage
8.          if (currentName is the same as searchName) then
9.              found ← true
10.         else {
11.             if (currentName comes alphabetically before searchName) then
12.                 startPage ← midPage + 1
13.             else
14.                 endPage ← midPage - 1
            }
        }
```

Notice how the **startPage** (or **endPage**) is adjusted in line **12** (or **14**) to be one more (or less) than the **midPage**. That is because we have already checked the **midPage**, so there is no need to have it remain in the list of items to search.

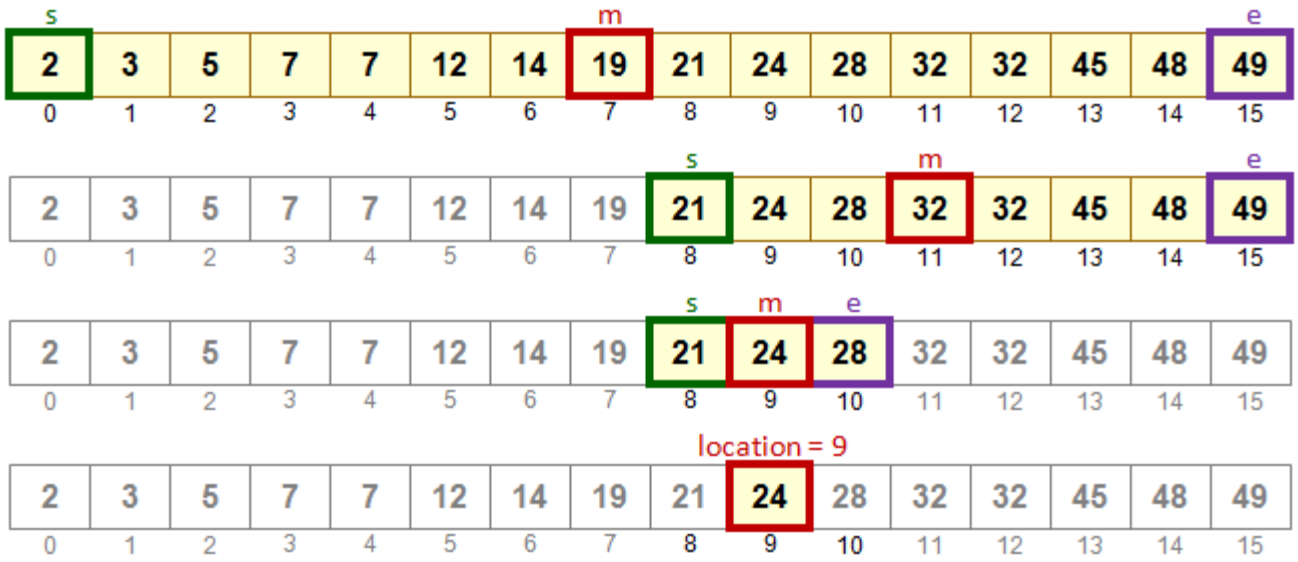
Here is an example of how the algorithm works if we were looking for the name **Green**:



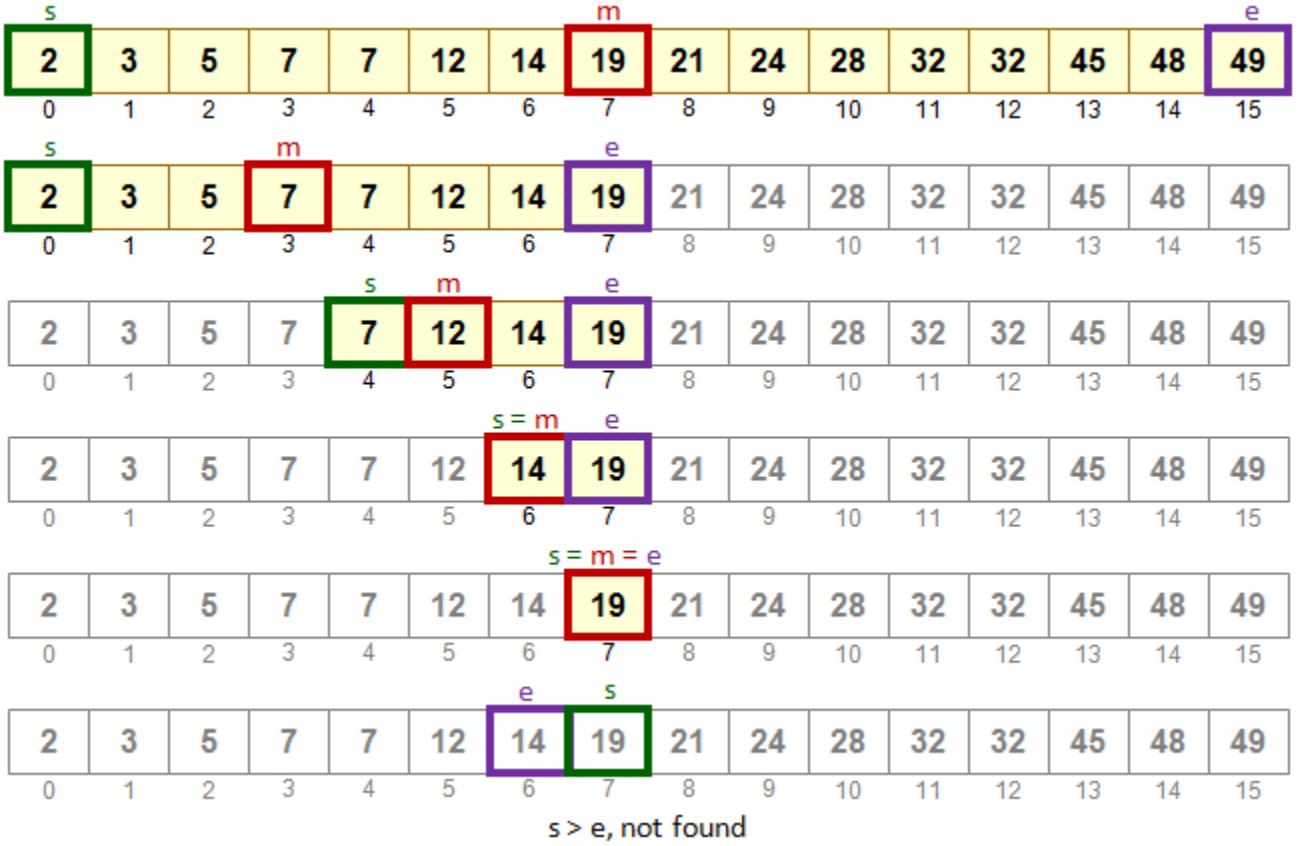
How do we write the code using arrays? Well, as an example, assume that we have the following array of numbers and that we want to determine whether or not the number **24** lies within the array:



Here is the result of applying the algorithm:



… and here is the result if we were searching for the number **18**:



The code will follow very closely to the algorithm...

```
import java.util.Scanner;

public class BinarySearchProgram {
    public static void main(String[] args) {
        int[]  items = {2, 3, 5, 7, 7, 12, 14, 19, 21, 24, 28, 32, 32, 45, 48, 49};

        System.out.print("Enter the item that you are searching for: ");
        int searchItem = new Scanner(System.in).nextInt();

        // Start searching from the beginning to the end of the array
        int     start = 0;
        int     end = items.length - 1;
        boolean found = false;

        // Keep going until start index meets end index, or unless we find it.
        while (!found && (start <= end)) {
            int middle = (start + end)/2;
            if (items[middle] == searchItem)
                found = true;
            else {
                if (items[middle] < searchItem)
                    start = middle + 1;
                else
                    end = middle - 1;
            }
        }

        if (found)
            System.out.println(searchItem + " is in the list.");
        else
            System.out.println(searchItem + " is not in the list.");
    }
}
```

You may feel that this seems more complicated than simply searching through the elements one-by-one. Yes, it is more complicated. Here is a comparative linear search:

Binary Search	Linear Search

`int start = 0;`
`int end = items.length - 1;`
`boolean found = false;`
 	
`while (!found && (start <= end)) {`
`  int middle = (start + end)/2;`
`  if (items[middle] == searchItem)`
`  found = true;`
`  else {`
`  if (items[middle] < searchItem)`
`  start = middle + 1;`
`  else`
`  end = middle - 1;`
`  }`
`}`
	`boolean found = false;`
	
	`for (int i=0; i<items.length; i++) {`
	`  if (items[i] == searchItem) {`
	`  found = true;`
	`  break;`
	`  }`
	`}`

But think about the advantages. Imagine checking the phone book for a name by always starting your search from the front of the book and checking page by page. Do you understand the advantage yet ?

In our above example, a linear search for the number 24 would require us to examine 10 array items (i.e., 2, 3, 5, 7, 7, 12, 14, 19, 21, 24), while the binary search required us to examine only 3 array items (i.e., 19, 32, 24) before the number was found.

When searching for a number that is not there (e.g., 20), the linear search would have to be exhaustive, checking all 16 items. However, the binary search required only 5 items to be examined.

Of course, the binary search code is longer and has more variables, so there is a bit of "overhead" in getting it to work. Hence, the linear search is often much better to use when a small number of items need to be searched. The advantage of the binary search comes when the array size is large. For example, in the worst case scenario an array of around **1,000,000** items would require us to search all **1,000,000** items if the item is not in the array. However, a binary search would only require a search of around $\mathbf{log_2(1{,}000{,}000)}$ ~= **20** items !!! The logarithm (base 2) comes into play because we are continually discarding ½ of the array items during each round through the loop. Can you imagine the time-savings of searching 1GB of data ? It would check only **30** items at most in place of **1,000,000,000** !!!

As you continue with your degree in computer science, you will learn much more about efficiency.