Chapter 5

Lists, Arrays and Searching

What is in This Chapter ?

When solving problems, we often deal with data that has been collected together. We often must sift through collections of information to find answers. This chapter discusses how data can be collected together into ***Arrays*** and also the various ways that we can ***search*** through the data efficiently to find what we want.

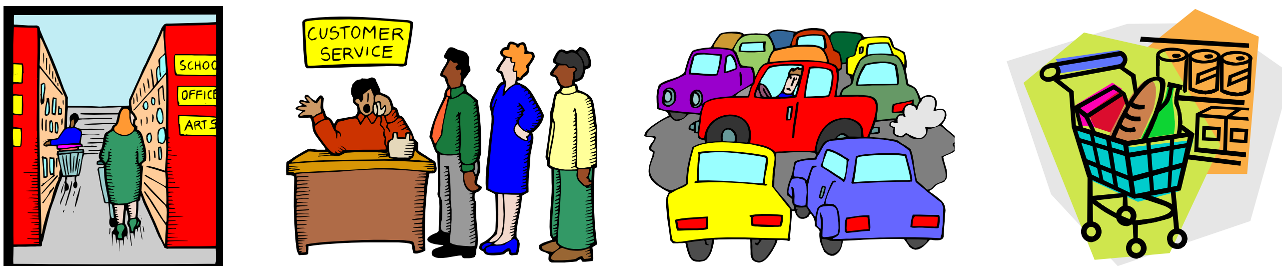
5.1 Collecting Data Using Lists

As we have seen in the last section, it is easy to define our own data structures. In fact, it is such common practice to define data structures that there is an entire 2nd year course on Data Structures and Algorithms. We will not discuss the deeper concepts related to data structures in this course.

Obviously, we can define a particular data structure to suit our own needs. There are, however, some common well-known kinds of data structures that are fundamental to the development of more useful algorithms. Such data structures have "similar patterns" of what they store and how they are used. When we have two or more particular data structures that fall under the same "pattern" of usage, we group all such data structures into a category and denote such a category as an ***abstract data type*** (ADT).

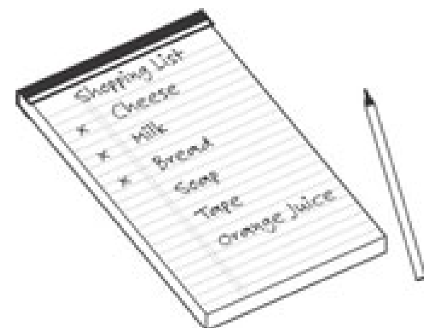
One such abstract data type is called a ***Collection*** which represents a group of objects that are treated as a single entity. Collections appear in real-life situations such as:

- storing products on shelves in a store
- maintaining information on customers
- keeping track of cars for sale, for rental or for servicing
- a personal collection of books, CDs, DVDs, cards, etc...
- maintaining a shopping cart of items to be purchased from a website



A more specific kind of collection is called an ***Ordered Collection*** or ***List***. The concept of a list is intuitive to most of us. We know it as a bunch of items that are placed in some kind of order (e.g., ordered in a first-come-first-served basis, or perhaps sorted alphabetically, etc..).

We may think of a list written on a piece of paper, perhaps of items that need to be picked up at the grocery store. Even though the order of the items is unimportant, it nevertheless has an order (perhaps arbitrary) to the items. Sometimes, a list may also be sorted, perhaps alphabetically or by price or date. Such a kind of ordering would make the list a ***Sorted List***.



For the remainder of this discussion, we will assume that the word **List** represents an unordered list of items, unless otherwise stated.

When using a list, there are certain standard functions and procedures that are expected to be available. That is, we should be able to create a new list, add something to the list, remove something from the list etc… One of the most common operations to perform on a list is to iterate (or traverse) through its elements one-by-one in order. Let us look at some examples that build up some lists of items and then do something simple with the list data.

Example:

Consider working as an assistant at a library. Imagine that someone was handing books to you and you needed to place them side-by-side on an empty shelf. What represents the *collection* data structure in this situation ?

The shelf is the collection, and in fact … it is an ordered list since the books will eventually have an order once they are on the shelf (i.e., in the list). Assuming that we had to find the empty shelf on our own, how would we write an algorithm for stocking the shelf, assuming that there is a shelf with enough space for all the books ?

Algorithm: PlaceBooks

```
1.    shelf ← new List
2.    while there are more books {
3.        book ← get the next book
4.        add book to shelf
      }
```

The algorithm is simple and logical, but it does not indicate where to add the books on the shelf. It may not matter. In this case, step 4 may simply require adding the book to the end of the shelf, beside the last one added.

However, if the books are to be sorted by their call numbers, then the order will matter. It will require us to insert the book at a specific position on the shelf. We can be a little more specific in our code by making use of the call number:

Algorithm: PlaceSortedBooks

```
1.    shelf ← new List
2.    while there are more books {
3.        book ← get the next book
4.        callNumber ← look at the book's call number
5.        insert book into shelf at the appropriate callNumber location
      }
```

Of course, step 5 can always be clarified further, but the algorithm as shown is understandable at a high level.

Example:

Now assume that we want to transfer all the books from an existing shelf to an empty shelf of the same size. How would we adjust the algorithm to do this, assuming that we are given two parameters to the algorithm representing the shelves ?

```
Algorithm: TransferBooks
    shelf1:     shelf containing some books to be transferred
    shelf2:     an empty shelf to place the books onto

1.      for each book on shelf1 {
2.          remove book from shelf1
3.          add book to shelf2
        }
```

The algorithm is straight-forward, and again … a similar version for sorted books can be written.

Example:

The above example assumes that there is enough space on **shelf2** to hold the books that were on **shelf1**. How could we adjust the algorithm to account for the situation where **shelf2** is smaller than **shelf1** ?

```
Algorithm: TransferLimitedBooks
    shelf1:     shelf containing some books to be transferred
    shelf2:     an empty shelf to place the books onto

1.      for each book on shelf1 {
2.          if (there is space on shelf2) then {
3.              remove book from shelf1
4.              add book to shelf2
            }
        }
```

As we will see later, there are more details to work out when we begin to program, but for now, it is important to understand how lists can be used in other important examples.

5.2 List Searching Algorithms

In real life, we are often faced with the problem of sifting through information (i.e., data) to determine whether or not a particular type of value lies within the information. We call this the ***searching*** problem, since we are searching through data for a (possibly partial) solution to our problem at hand.

For example, we may look through a list to:

1. determine the existence of a piece of data (e.g., check if a person is on a list)
2. verify that all items satisfy a condition (e.g., is anyone missing from work today)
3. pick an appropriate candidate for our problem (e.g., find an available seat on the bus)
4. determine the most efficient solution (e.g., detect the shortest line at Walmart)
   etc..

> A ***linear search*** (or ***sequential search***) is a method of checking every one of the elements in a list, one at a time and in sequence, until a value is found that satisfies some particular condition (e.g., until a *match* is found).

When developing algorithms, the simplest approach is often called a "**brute force**" approach, implying that there is a lack of human sensibility in the solution. A "brute force" algorithm is one that is often easy to come up with, but that does not usually consider efficiency or any form of ingenuity.

A linear search is the simplest kind of search since it involves naively looking through the elements in turn for the one that matches the criteria. In all the examples below, we will assume that the items that we are searching through are stored as an array (explained later).

Example:

Consider determining whether or not a person is on a list, perhaps to allow them admittance into a special event. How would you develop an algorithm to do this using a linear search ?

Assume for example that we wanted to find "Patty O. Lantern" on the list shown here. It may be obvious that all we need to do is to start at the top of the list and then work our way down towards the bottom, comparing names along the way.

But how do we explain this procedure with an algorithm ?
Here is one way …

```
Algorithm: IsOnList
        names:              the list (e.g., array) to search through
        searchName:         the name to search for

1.      found ← false
2.      for each name in the names list {
3.          if (name = searchName) then
4.              found ← true
        }
5.      print found
```

Name
Bob E. Pins
Sam Pull
Mary Me
Jim Class
Patty O. Lantern
Robin Banks
Shelly Fish
Barb Wire
Tim Burr
Dwayne D. Pool
Hugh Jarms
Ilene Dover
Frank N. Stein
Ruth Less

Notice that the **for** loop checks all the people on the list one-by-one. Notice the use of the Boolean variable **found**. This is called a boolean ***flag*** in computer science. It is analogous to the little flag on mailboxes that the mailperson lifts up to let people know when their mail has arrived.

Notice how the *flag* is down (i.e., **false**) before the loop starts and that it only gets raised (i.e., set to **true**) when the **name** matches the **searchName**.

The above algorithm always takes ***linear time*** … that is, in the worst case it may require us to check all of the names on the list (i.e., the last name on the list matches).

In practice however, it is possible to exit the loop when a match is found. After all, why keep looking through the rest of the names when we have already found it. Consider using a **while** loop where the *flag* is the loop condition:

```
Algorithm: IsOnList2
        names:              the list (e.g., array) to search through
        searchName:         the name to search for

1.      found ← false
2.      while ((found = false) AND (there are more names to check)) {
3.          name ← get the next name in the names list
4.          if (name = searchName) then
5.              found ← true
        }
6.      print found
```

Notice now how the **false** value at the beginning ensures that we enter into the **while** loop. Then, once the name is found, the *flag* is set and the loop will stop right away. This is more efficient in practice, although in the worst case, it may still require us to check all of the people on the list. Notice as well that the loop ends when no more names are available.

Example:

Name	Age
Bob E. Pins	25 ✓
Sam Pull	24 ✓
Mary Me	31 ✓
Jim Class	54 ✓
Patty O. Lantern	62 ✓
Robin Banks	18 ✓
Shelly Fish	17 ×
Barb Wire	21
Tim Burr	26
Dwayne D. Pool	47
Hugh Jarms	36
Ilene Dover	42
Frank N. Stein	13
Ruth Less	71

Consider now determining whether or not all items in a list satisfy some condition. For example, how would we write an algorithm that determines whether or not **everyone** on a list is 18 years of age or older ? This is a kind of verification algorithm that should return **false** if just one "under age" person is found.

Likely we'll begin as before checking each person's age one-by-one. But what are we looking for ? We are looking for someone who is under the age of 18.

Do we need to check every person, or can we jump out of the loop at any time ? Here is the algorithm:

```
Algorithm: AllAdults
        people:         the list (e.g., array) to search through

1.      allAdults ← true
2.      for each person in the people list {
3.              if (person.age < 18) then
5.                      allAdults ← false
        }
6.      print allAdults
```

The structure of this search algorithm is quite similar to our previous example. Notice however, that we began with an assumption that all in the list are adults and only set the *flag* to **false** when there is someone found who is under age. Do you understand why we had to switch the *flag* around like this ?

Example:

Now let us go back to our algorithm that searches for a name on a list. Perhaps to get into the special event, we need to show our ID at the door so that our age can be verified.

How would we adjust the algorithm so that it didn't simply return **true** or **false** whether or not a person is on the list but in fact returned something about the person, such as their **age** ?

What is different ? Do we still need a boolean *flag* ?

Here is a solution:

```
Algorithm: FindAgeOfPerson
        people:          the list (e.g., array) to search through
        searchName:      the name to search for

1.      for each person in the people list {
2.          if (person.name = searchName) then
3.              age ← person.age
        }
4.      print age
```

Name	Age
Bob E. Pins	25
Sam Pull	24
Mary Me	31
Jim Class	54
Patty O. Lantern	62
Robin Banks	18
Shelly Fish	17
Barb Wire	21
Tim Burr	26
Dwayne D. Pool	47
Hugh Jarms	36
Ilene Dover	42
Frank N. Stein	13
Ruth Less	71

Notice now that we did not use a boolean *flag* but instead we stored the **age** of the person that we found in the list so that we could print it later. Of course, if all we wanted to do was print the **age**, we could have done it within the loop:

```
Algorithm: FindAgeOfPerson2
        people:          the list (e.g., array) to search through
        searchName:      the name to search for

1.      for each person in the people list {
2.          if (person.name = searchName) then
3.              print (person.age)
        }
```

We can also even adjust the code to use a **while** loop so as to allow the loop to exit early:

```
Algorithm: FindAgeOfPerson3
        people:          the list (e.g., array) to search through
        searchName:      the name to search for

1.      found ← false
2.      while ((found = false) AND (there are more people to check))  {
3.          person ← get the next person in the people list
4.          if (person.name = searchName) then {
5.              print (person.age)
6.              found ← true
            }
        }
```

Notice now that the algorithm is efficient in that it prints out the **age** and quits the loop when the person has been found.

One problem, however, with our above solutions is that they do not handle the situation in which the person is not found in the list. In the 2nd and 3rd case, this may not be a problem since nothing is printed. In the 1st case however, the **age** is printed on the last line. But this **age** is only set when the person is found. What is the **age** set to when the person is not found ? This depends on the computer language being used. Often, the **age** will default to zero, and so **0** will be printed. It might be a good idea to specify the "default" age at the top of the algorithm … perhaps **age = -1** or something like that to indicate that it is invalid.

Example:

Another very common task when searching through a set of values is to find the maximum or minimum. Can you write an algorithm to find the age of the oldest person out of a set of people ?

What do you need to keep track of as you are figuring out who has the maximum age ?

```
Algorithm: FindOldest
        people:              the list (e.g., array) to search through

1.      oldestSoFar ← 0
2.      for each person in the people list {
3.              if (person.age > oldestSoFar) then
4.                      oldestSoFar ← person.age
        }
5.      print oldestSoFar
```

Notice how the **oldestSoFar** variable is used as a "record keeper" to remember the **age** of the person who is currently the oldest as we search systematically through the list. We only change that value once we find someone older.

To find the youngest person, the algorithm is nearly identical. Do you know what changes ?

```
Algorithm: FindYoungest
        people:         the list (e.g., array) to search through

1.      youngestSoFar ← 200
2.      for each person in the people list {
3.              if (person.age < youngestSoFar) then
4.                      youngestSoFar ← person.age
        }
5.      print youngestSoFar
```

As you can see, the code is basically the same, except for the minor change in the condition as well as the initial starting value for the **youngestSoFar**. Why did we set this to such a large number like **200** ? What would have happened if we left it at **0** ? Do you understand ?

It is important to choose a number that we know to be larger than any other possible age of the people that we are searching through. In our times, 200 is a reasonable number since all people will have an age lower than this. However, just a few thousand years ago, people lived longer than 200 years old. For example, if you recall the historical account of Noah (the man that built the ark with all the animals on it) … history records that he lived 950 years old! People actually lived a long time back then .. the oldest recorded was 969 (Noah's grandfather whose name was Methuselah).

How would our algorithm perform if used on people who lived that long ? Well, likely all people will be older than 200 years and therefore the algorithm would never set the **youngestSoFar** variable and so the answer would always be 200 … which is wrong.

So, it is always "better to be safe than sorry" by choosing a very large value that we know to be above all possible values in our problem. Recorded history tells us therefore that any number 1000 or more would be sufficient to ensure that at least one of our people are below that age. Some computer languages have predefined constants representing the maximum integer or minimum integer. Processing and JAVA, for example, have constants that represent the largest and smaller integers that can be stored: **Integer.MAX_VALUE** and **Integer.MIN_VALUE**.

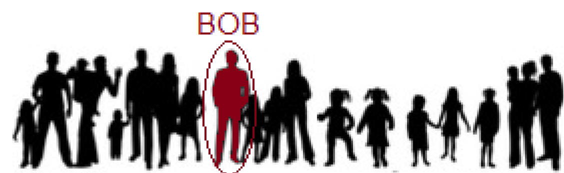
These would be good value to use if we are trying to find maximum and minimum numerical values. Whenever we find ourselves in need of determining the largest/maximum or smallest/minimum value of a list, the templates above will always work.

Sometimes we need to keep track of additional information as we find the maximum and minimum. For example, what if we wanted to know the **name** of the oldest person, not just the age ? What do you need to keep track of as you are figuring out who has the maximum age ?

Algorithm: FindNameOfOldest

people: the list (e.g., array) to search through

```
1.    oldestAge ← 0
2.    oldestName ← "unknown"
3.    for each person in the people list {
4.        if (person.age > oldestAge) then {
5.            oldestAge ← person.age
6.            oldestName ← person.name
          }
      }
7.    print oldestName
```

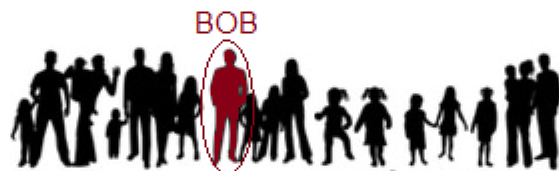


Notice how both the **age** and the **name** of oldest person is maintained throughout the loop. The **oldestAge** is used to compare against each person's age while the **oldestName** is simply stored as the name of the person who had the **oldestAge** so far.

As before, we would have to choose some kind of default **oldestName** for the situation in which no oldest person is found. However, this would only occur when all people are 0 years of age or more likely when there is nobody on the list. Provided that we choose a proper starting **oldestAge**, then the default **oldestName** will only be an issue in the case where the list is empty.

We could actually simplify the code to make use of the entire **person** data structure (with the **name**, **age**, etc..) as opposed to just the **name** as follows:

```
Algorithm: FindNameOfOldest2
        people:         the list (e.g., array) to search through

1.      oldest ← the first person in the people list
2.      for each person in the people list {
3.          if (person.age > oldest.age) then
4.              oldest ← person
        }
5.      print oldest.name
```



Notice now that only one variable is needed because it is an entire data structure that contains the **name** & **age** of the person who is the **oldest**. (Remember, a data structure is really just a grouping together of data). In fact, if we wanted, on line 5 we could access or print out any (or all) of that **oldest** person's information because it is readily available within the data structure.

Notice that line 1 set the **oldest** to initially be the first person in the list. Beginning with the first value in a list is a typical when finding a maximum or minimum value as opposed to beginning with a value of **null**. That is because the very first person in the list would be compared in line **3** with **oldest.age**. If oldest is **null**, then **oldest.age** would be undefined and the code would not make sense.

Example:

In our examples above, we did not consider the situation in which people have the same name and/or the same age. For example, consider what happens when we are searching for a name in a list that has duplicates.

In the case where we are trying to find out whether a particular person is on the list, the algorithm need not worry. Also, if we are trying to determine whether or not everyone on the list is an adult, duplicates do not affect the algorithm either.

Consider people playing a game one or more times and recording their score on a list. In this case, a person may appear more than once and their associated score is likely unique each time.

What would be the result of the following algorithm if the **searchName** is "Patty O. Lantern":

```
Algorithm: FindScoreOfPerson
        people:         the list (e.g., array) to search through
        searchName:     the name to search for

1.      for each person in the people list {
2.           if (person.name = searchName) then
3.                score ← person.score
        }
4.      print score
```

Name	Score
Hugh Jarms	36
Bob E. Pins	25
Shelly Fish	17
Sam Pull	24
Mary Me	31
Jim Class	54
Patty O. Lantern	62
Robin Banks	18
Shelly Fish	17
Barb Wire	21
Patty O. Lantern	31
Tim Burr	26
Dwayne D. Pool	47
Hugh Jarms	31
Ilene Dover	42
Frank N. Stein	13
Patty O. Lantern	16
Ruth Less	71

Which score gets printed ? The last one does. Careful examination of the code shows that the **score** variable is set three times for the list shown above, the 1st and 2nd values are overwritten by the 3rd value of **16**.

We can actually adjust the algorithm to list all the scores for that person by moving the print procedure from step 5 to step 4 like this: **print (**score of **person)**.

Example:

How could we alter the algorithm so that it displayed the average score for the person ?

```
Algorithm: AverageScoreOfPerson
        people:         the list (e.g., array) to search through
        searchName:     the name to search for

1.      score ← 0
2.      count ← 0
3.      for each person in the people list {
4.           if (person.name = searchName) then {
5.                score ← score + person.score
6.                count ← count + 1
             }
        }
7.      print score / count
```

Name	Score
Hugh Jarms	36
Bob E. Pins	25
Shelly Fish	17
Sam Pull	24
Mary Me	31
Jim Class	54
Patty O. Lantern	62
Robin Banks	18
Shelly Fish	17
Barb Wire	21
Patty O. Lantern	31
Tim Burr	26
Dwayne D. Pool	47
Hugh Jarms	31
Ilene Dover	42
Frank N. Stein	13
Patty O. Lantern	16
Ruth Less	71

+

Notice how the **score** is totaled each time we notice the person in the list. Also, do you understand why we needed to also maintain a **count** of how many times the person's name appeared on the list ? It is necessary in order to compute the average.

Example:

How about finding the highest score where there are multiple people with the same score ?

If you remember, the first step to problem solving is to *understand* the problem. So what should the answer be ? Well, three people in our example here have the highest score. What should the output be ? We have some choices:

1. Print the first person who received the highest score.
2. Print the last person who tied the highest score.
3. Print all three who have the highest score.

What would the algorithm look like in each case ? For the first case, the format is identical to the one that found the oldest person:

```
Algorithm: HighestScore
        people:         the list (e.g., array) to search through

1.      highestScore ← 0
2.      highestName ← "unknown"
3.      for each person in the people list {
4.              if (person.score > highestScore) then {
5.                      highestScore ← person.score
6.                      highestName ← person.name
                }
        }
7.      print highestName
```

Name	Score
Hugh Jarms	36
Bob E. Pins	25
Shelly Fish	47
Sam Pull	24
Mary Me	91
Jim Class	54
Patty O. Lantern	62
Robin Banks	18
Shelly Fish	17
Barb Wire	21
Patty O. Lantern	91
Tim Burr	26
Dwayne D. Pool	47
Hugh Jarms	86
Ilene Dover	91
Frank N. Stein	13
Patty O. Lantern	16
Ruth Less	71

For the 2nd case, we just need to change the condition in step 5 to (**score** >= **highestScore**). The 3rd case is a little trickier. Can you explain why the following code will NOT work ?

```
highestScore ← 0
highestName ← "unknown"
for each person in the people list {
        if (person.score >= highestScore) then {
                highestScore ← person.score
                print (person.name)
        }
}
```

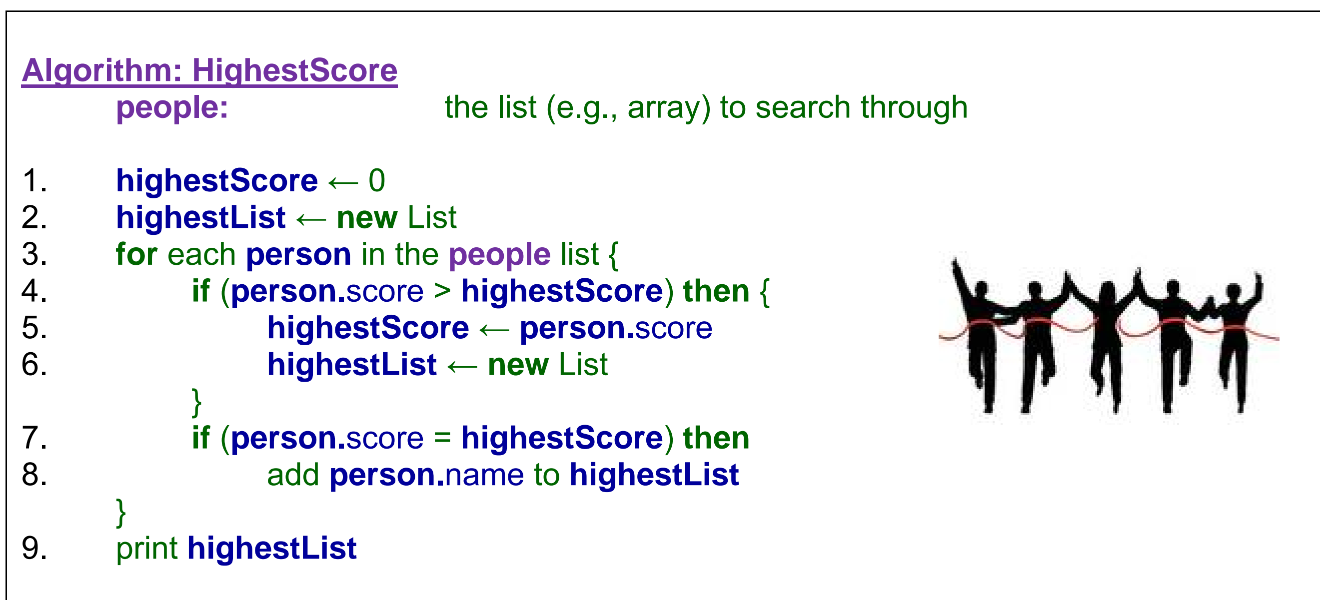
This algorithm would print out every person whose score exceeded the current high score as we went through the list. Here would be the names printed:

```
Hugh Jarms, Shelly Fish, Mary Me, Patty O. Lantern, Ilene Dover
```

Now, while the last three names are correct, the first two do not share the same high score. What is wrong with the algorithm ? We cannot simply print out the person whenever they "beat" the currently best high score. We need to know that the currently high score is the actual highest score before we print. So, we will need to print at the end, but somehow remember all those who had the highest score.

To do this, we will need to keep a new list of people who share the high score. When do we add to the list ? Do we ever need to reset the list ? Think about it.

Here is a revised (and correct) solution:



```
Algorithm: HighestScore
        people:          the list (e.g., array) to search through

1.      highestScore ← 0
2.      highestList ← new List
3.      for each person in the people list {
4.          if (person.score > highestScore) then {
5.              highestScore ← person.score
6.              highestList ← new List
            }
7.          if (person.score = highestScore) then
8.              add person.name to highestList
        }
9.      print highestList
```

Notice that whenever we find a new highest score, the **highestList** of people who had the previous highest score is emptied out (step 7). Then in step 9, this person is added to that newly emptied **highestList**. Whenever someone else is found who shares that highest score, he/she is simply added to the **highestList**.

Example:

Here is a tougher one now. What if we were on our way to an autoshow but before we left our roommate asked us (for some strange/unknown reason) to determine the most popular color of car at the autoshow.

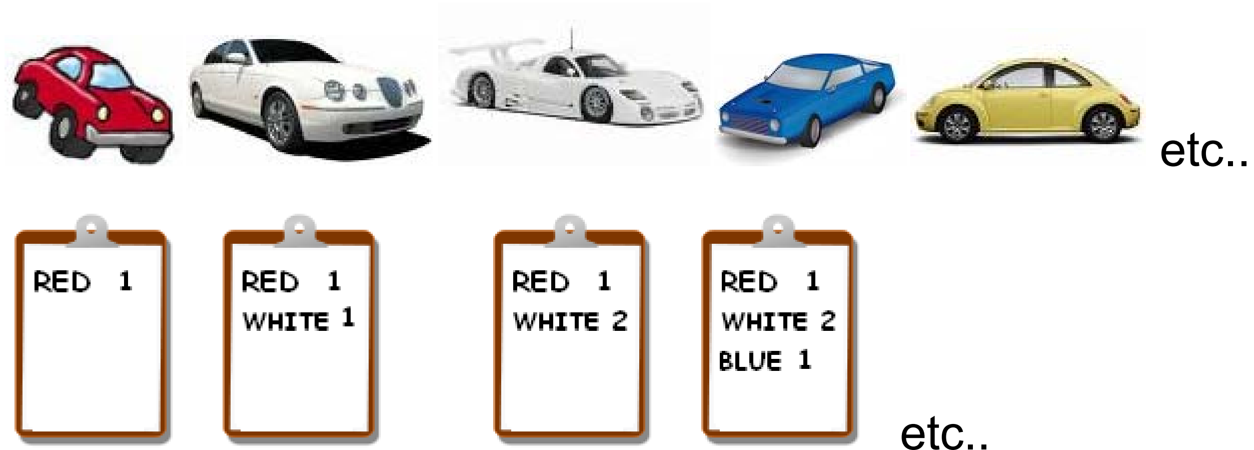
Now how do we approach the problem ?

Well think of real life. Assuming that there were hundreds of cars and that your memory is not perfect, you would likely bring with you a piece of paper (perhaps on a clipboard) so that you can keep track of the colors of cars that you find.

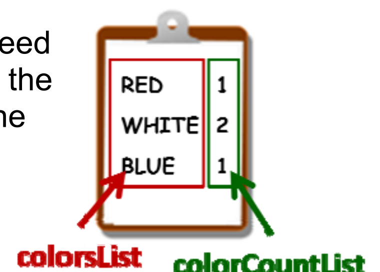
When you enter the autoshow and see the first car, you would look at its **color** and then likely write down the color on the paper and maybe put the number 1 beside it.

Assume that you went to the next car and that it was a different color. You would write that new color down too along with a count of 1. If you find a car with the same color again, you would just increment its count.

Below is a picture showing how your clipboard gets updated as you encounter car colors in the order of red, white, white, blue, etc..:



If we were to take this approach in our programming solution, then we need to realize that each color that we write down on our clipboard will have a single number associated with it at all times (representing the **count** of the number of times that color appeared).



The clipboard/paper itself represents our list of colors so we would need a list to store that. In fact, since we need a list of **counts** along with the list of **colors**, we will need ***two*** lists… one to store the **colors** and one to store the **counts**.

At this point, we can do a partial algorithm that will count up the number of times that each color appears at the show as follows:

```
Algorithm: CarColorCount
        cars:                   the list (e.g., array) to search through

1.      colorsList ← new List
2.      colorCountList ← new List
3.      for each car in the cars list {
4.          if (car.color is not in the colorsList) then
5.              add car.color to colorsList
6.          add 1 to colorCountList for the color corresponding to car.color
        }
```

Once this part of the algorithm has been completed, we have two lists filled in showing the colors at the autoshow and their corresponding counts.

To find the most popular color, we simply need to find out which of these colors has the largest value (i.e., the maximum count). We will assume that there are no duplicates (or that only the first one with the maximum count is the answer).

The solution would then be to first count the colors using the above algorithm and then apply a max/min algorithm similar to the one that found the name of the oldest person as we did before:

```
Algorithm: MostPopularColor
        cars:                   the list (e.g., array) to search through

1.      perform algorithm CarColorCount()
2.      bestCountSoFar ← 0
3.      bestColorSoFar ← unknown
4.      for each color in the colorsList {
5.          count ← count in colorCountList corresponding to color
6.          if (count > bestCountSoFar) then {
7.              bestCountSoFar ← count
8.              bestColorSoFar ← color
            }
        }
9.      print bestColorSoFar
```

Notice that the structure is similar to the standard min/max template except that we performed a kind of pre-processing stage in step 1 to count the colors.

Alternatively, we could have combined the color counting and max-finding sub-algorithms together:

```
Algorithm: CombinedMostPopularColor
        cars:                   the list (e.g., array) to search through

1.      colorsList ← new List
2.      colorCountList ← new List
3.      bestCountSoFar ← 0
4.      bestColorSoFar ← unknown

5.      for each car in the cars list {
6.              if (car.color is not in the colorsList) then
7.                      add car.color to the colorsList
8.              add 1 to colorCountList for the color corresponding to car.color
9.              count ← count in colorCountList corresponding to car.color

10.             if (count > bestCountSoFar) then {
11.                     bestCountSoFar ← count
12.                     bestColorSoFar ← car.color
                }
        }
13.     print bestColorSoFar
```

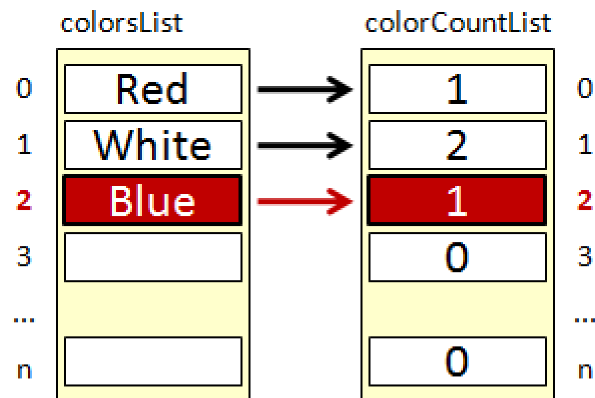
5.3 Arrays

In most of our list examples, we were not overly concerned about particular positions of items in the list. That is, we usually iterated through the items in the list one at a time, but were not concerned about the number of the item within the list. When we used a **while** loop, we simply asked to "get the next item in the list". Therefore, we were essentially treating the list as if it was unordered (i.e., as if the order did not matter):

```
for each person in the people list { … }
for each car in the cars list { … }
name ← get the next name in the names list
```

However, in our last example, we wanted to maintain a count for each color in the list. When we encountered a particular color, we needed to find that color in the **colorsList** and add 1 to the corresponding color in the **colorCountList** →

To do this, we need to make sure that we update the correct counter each time. So, we need to know the position of the color in the **colorsList** and update the counter at the ***SAME*** position in the **colorCountList**.

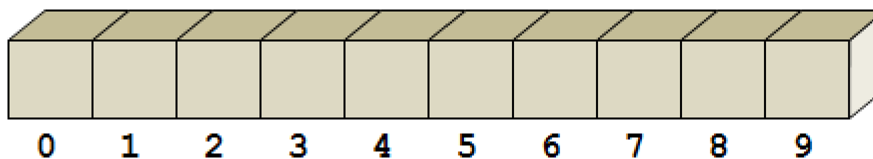


A list in which the order of the items is important to know, is called an ***ordered list***. Some programming languages have pre-defined data types to represent ordered lists with convenient methods such as **add**, **remove**, **insertItemAtPosition**, **getItemAtPosition**, etc... However, not all languages have such a convenient data type readily available.

All the popular programming languages do, however, have a fixed data type called an ***array***.

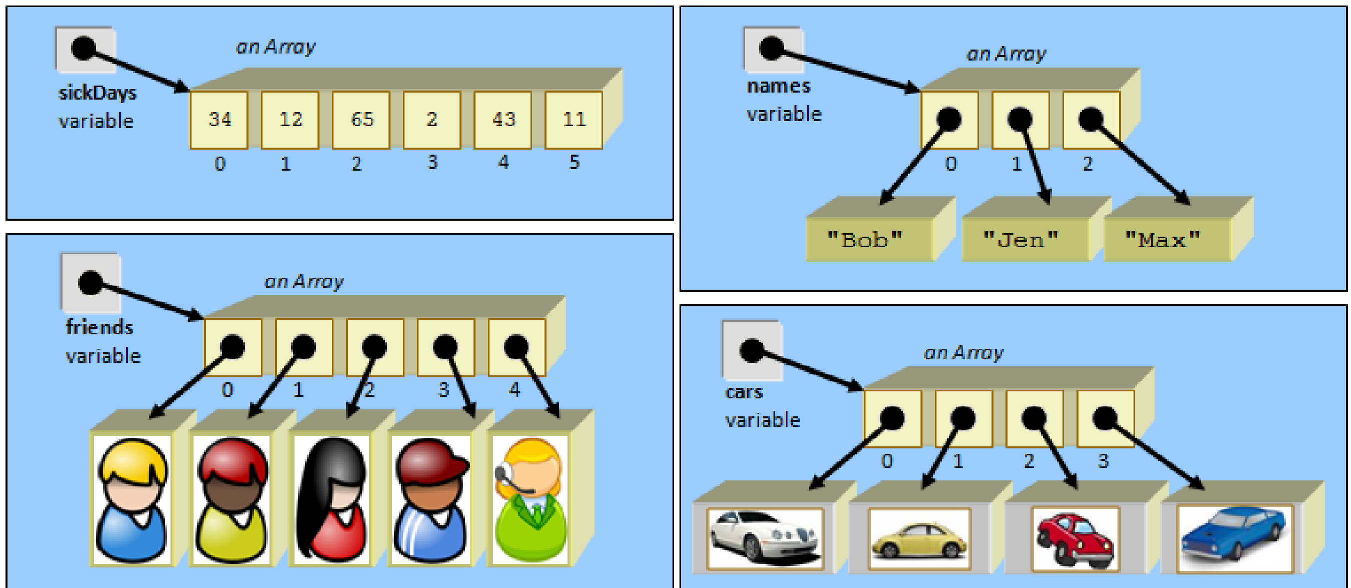
*An **array** is a collection of data items that can be selected by indices (or locations).*

So, arrays are a means of “gluing” a bunch of variables side-by-side in some specified order.



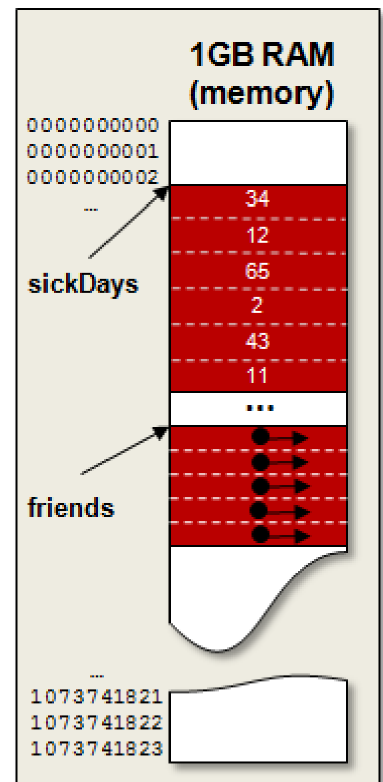
Each item (also known as ***element***) in an array is stored at a location which is specified by an **index**. The index is an integer that identifies the location within the sequence of items. Indices start at 0. Arrays are fixed-sized collections in that they can hold a fixed number of items … that is … they don’t grow or shrink … they have a fixed size. In the above image, the array holds exactly 10 items, so the indices go from **0** through **9**. An index of **10** or higher would be out of the array’s “bounds”, and would therefore be invalid. We often refer to the “size” of the array as its **length**. The size/or length of the array is NOT the number of items that we have put inside it, rather, it is the ***capacity*** of the array (i.e., the maximum number of items that we can put into it). We will use **anArray.length** in our pseudocode to indicate the length of an array which is stored in the **anArray** variable.

Arrays themselves are data types that store a particular kind of item within them. Each item is understood to be of the same type (e.g., all integers, all floats, all Strings, all Cars, etc..). Here are some examples of arrays that hold integers, Strings, Person objects and Car objects:



Notice how each item in the array is of the same type. The array data structure is quite efficient in how it actually stores the items in memory. Once allocated in memory, the array takes up a contiguous (i.e., connected without a break) sequence of memory locations.

If the data being store is simple primitive-like data, then it is stored directly in the array one item after another. The **sickDays** variable above, for example, represents an array with simple integers that would be store sequentially in memory as shown here: →



If, however, the array's data items are a more complex data type, then it is stored differently in a way that depends on the programming language.

For example, in JAVA/Processing, pre-existing objects may be created and then the array would store simply pointers (e.g., memory addresses) to those objects. So an array of items of type **Person**, would store a sequence of pointers to the individual **Person** objects, as shown here in the **friends** array: →

Alternatively, in languages like C for instance, the array would simply reserve contiguous space for the entire series of Person data types by calculating the size of a person data structure and multiplying by the size of the array.

Because array data is stored and accessible in contiguous memory locations, we can actually refer to all of the data by using a single variable name! That is nice. We can use just one variable to refer to all the items, even if there are thousands of them.

Of course, when we want to access a particular item, we need to indicate which one we want. So, we need to supply its index. Also, each time we want to put something into the array we also need to supply the index location.

So, we will use square brackets (i.e., **[ ]**) whenever we refer to a particular item in the array. Thus, we will use the array's variable name, followed by square brackets and place the index number within the brackets as follows:

```
theArray[5]
```

Example:

Consider for example our previous algorithm for determining whether or not someone's name (i.e., **searchName**) is on a list (i.e., **names**). The left side of the table below shows the code as was used with *unordered* lists, while the right side indicates the *array* version.

Unordered List Version	Array Version

found ← **false**
for each **name** in the **names** list {
    if (**name** = **searchName**) **then**
        found ← **true**
}
print **found**
	found ← **false**
	for index **i** from **0** to **names**.length-1 {
	    if (**names[i]** = **searchName**) **then**
	        found ← **true**
	}
	print **found**

Notice a couple of things with the above code. We are specifying in the **for** loop the index number (stored in variable **i**) of the items in the array. Then, this index is used in the **if** statement condition to access the name at position **i** in the **names** array. You may notice that there is no **name** variable to store the name. Instead, we simply use the entire array and indicate the name by means of the square brackets and index **i**.

So you can see that the code is essentially the same, except that we now need to specify the location of each item in the list by means of its index.

Example:

Recall the example that determined the person with the highest score in the unordered list. Again, below is a comparison of the unordered list version with the array version. In the example, **people** is the parameter containing **Person** data structures (i.e., objects), which each have **score** and **name** attributes.

<table>
<tr><th>Unordered List Version</th><th>Array Version</th></tr>
<tr><td><pre>
highestScore ← 0
highestName ← “unknown”
for each person in the people list {
    if (person.score > highestScore) then {
        highestScore ← person.score
        highestName ← person.name
    }
}
print highestName
</pre></td><td><pre>
highestScore ← 0
highestName ← “unknown”
for index i from 0 to people.length-1 {
    person ← people[i]
    if (person.score > highestScore) then {
        highestScore ← person.score
        highestName ← person.name
    }
}
print highestName
</pre></td></tr>
</table>

Notice that the code is almost identical again, except that we require the index. In order to keep the code simpler, we added a line to set a **person** variable to the element of the array at position **i**. This variable was not necessary. We could have substituted **people[i]** for **person** as follows:

```
for index i from 0 to people.length-1 {
    if (people[i].score > highestScore) then {
        highestScore ← people[i].score
        highestName ← people[i].name
    }
}
```

The point is that each time we want to refer to a different person, we need to supply the index.

Example:

Now assuming that high scores can be tied, we wrote an algorithm that returned a new unordered list of all people who obtained that high score:

```
Algorithm: HighestScore
        people:             the list (e.g., array) to search through

1.      highestScore ← 0
2.      highestList ← new List
3.      for each person in the people list {
4.          if (person.score > highestScore) then {
5.              highestScore ← person.score
6.              highestList ← new List
            }
7.          if (person.score = highestScore) then
8.              add person.name to highestList
        }
9.      print highestList
```

How can we adjust this to use arrays … even creating a new array to represent the solution ? Let us first convert the code into code that assumes that the **people** list is an array:

```
highestScore ← 0
highestList ← new List
for index i from 0 to people.length-1 {
    if (people[i].score > highestScore) then {
        highestScore ← people[i].score
        highestList ← new List
    }
    if (people[i].score = highestScore) then
        add people[i].name to highestList
}
print highestList
```

Now, how do we adjust the code so that **highestList** is an array ? Recall that an array must have its capacity specified when it is created. How big should this answer be ? Think of the "worst-case-scenario" (i.e., the scenario that causes the array to be its largest size).

It is possible that everyone shares the same high score. In that case, the array would need to be the same size as the original **people** list:

```
highestList ← new Array with capacity people.length
```

How do we add people to the **highestList** ? Well, recall that this array should just be a sequence of **Person** data structures/objects. So, to add a person to the array, we need to indicate where in the array it should go … that is … we need to supply an index in the range from **0** to **highestList.length – 1**.

Logically, the first person added should go at position 0 in the array, the next one in position 1, the third in position 2, etc… So we will need to keep track of the next available location within the array. This can be done with a simple **nextLocation** integer counter which starts at 0.

Then, we simply insert into the array by simply assigning the person to the location in the array specified by **nextLocation**. Here is the code so far:

```
highestScore ← 0
highestList ← new Array with capacity people.length
nextLocation ← 0
for index i from 0 to people.length-1 {
        if (people[i].score > highestScore) then {
              highestScore ← people[i].score
              highestList ← new Array with capacity people.length
              nextLocation ← 0
        }
        if (people[i].score = highestScore) then {
              highestList[nextLocation] ← people[i].name
              nextLocation ← nextLocation + 1
        }
}
print highestList
```

Notice that we need to increase the **nextLocation** counter after we add a new person so that the next person will be added right after it. Do you know what would happen if we did not increase this counter ?

Assume that we were keeping track of the equal highest scores and suddenly we encounter a better high score. What do we do ? All of the high scores that we have added to the **highestList** are no longer highest scores.

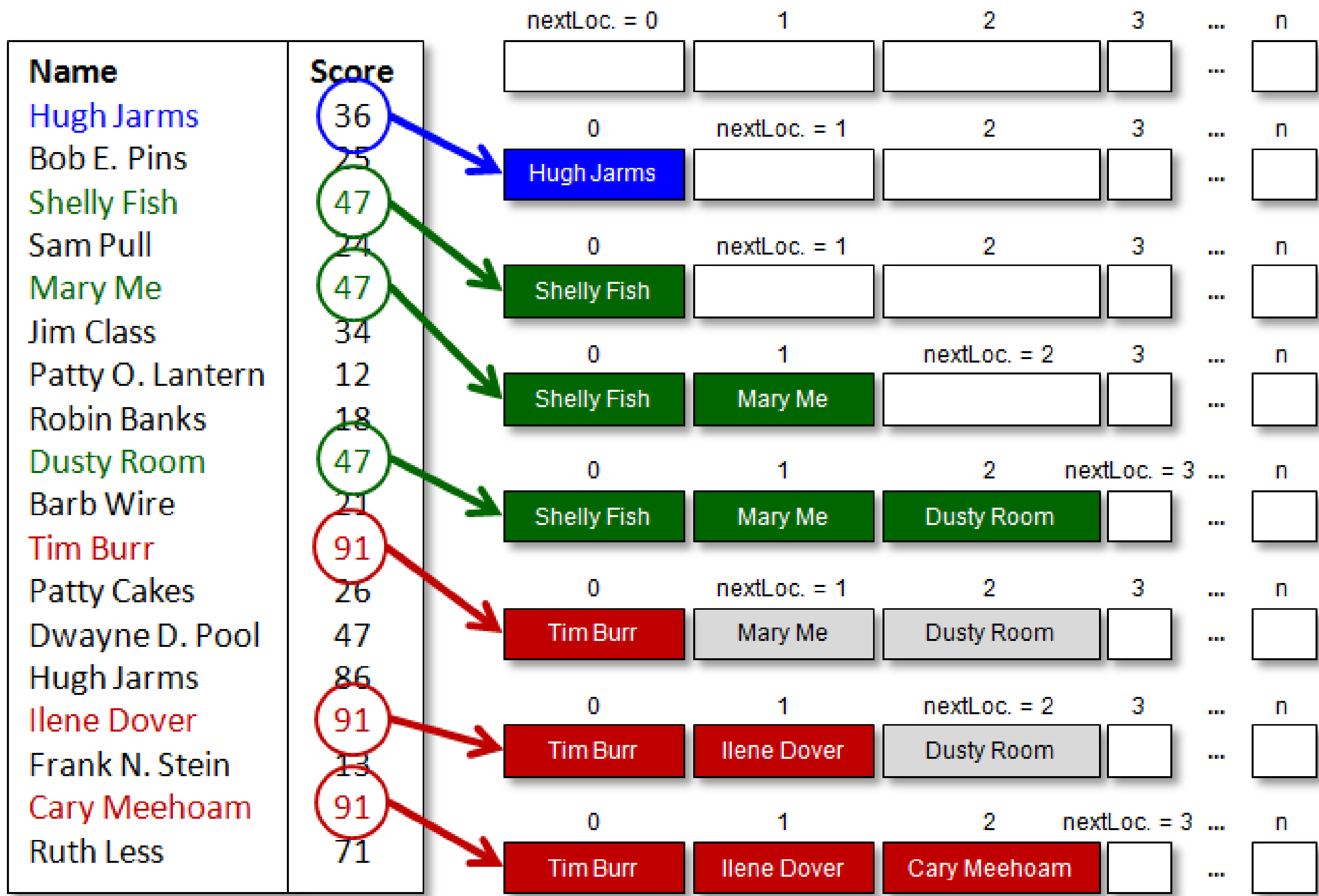
Currently, our code simply replaces the **highestList** with a brand new **highestList**. Where does the previous **highestList** go ? Well, it depends on the programming language. In Processing/JAVA, this old/garbage array will be "garbage collected". That is, the memory space that this array is using up will automatically be discarded at a later point in time by the system. However, in a language like C where WE are responsible for allocating and freeing up the memory, we must make sure to free this memory before re-assigning a new array to the **highestList** variable.

Regardless of the programming language, in both cases we would need to reset the **nextLocation** back to **0** so that new items begin to be placed at the start of the array again.

Interestingly, what would happen if we removed this line from the first **if** statement:

```
highestList ← new Array with capacity people.length
```

Well, the **highestList** would contain some high scores that are no longer valid. However, since we are resetting the **nextLocation** back to zero, all newly encountered high scores will be added to the array such that they overwrite the previous high scores. Notice in this example how the resetting of the **nextLocation** causes overwrite:



As a result, the final array may have some invalid data for all indices greater than **nextLocation**. The **nextLocation** variable actually indicates (at all times) the number of highest scores. Hence, even though the array is of size **n**, if the **nextLocation** variable has the value of **3** at the end of the loop iteration, then there are only **3** valid high scores which are located at the beginning of the array.

Example:

In the previous example, we produced a solution in which the resulting array had a size likely greater than was necessary. For example, if the **people** array had 100 people in it and only two had the highest score, the resulting array would be of size 100, yet only two values in the array would be valid. This is wasteful. How can we adjust the algorithm so that the array has a length that is exactly equal to the number of people with the highest score ?

Logically, we cannot know how many people have the highest score until after we have checked all of the people. So, we must traverse the list at least one time before knowing how big to make the array.

If we leave our code as it is now, before returning we could create a new array with the proper size (i.e., size = **nextLocation**) and then copy the items over from the big array to the other array. Then we can free the memory from the other array if necessary. We just need to append this to the end of our algorithm's code:

```
finalAnswer ← new Array with capacity nextLocation
for index i from 0 to nextLocation-1 {
        finalAnswer[i] ← highestList[i]
}
```

This code will copy the people from one array to the other. The **highestList** array will then be discarded. This is done either through automatic garbage collection (as in Processing/Java), or by manually freeing up the memory (as in C) … depending on the programming language used.

Of course, the above solution may be considered wasteful in that a potentially large array needs to be allocated and then disposed of. This is not a space-efficient solution. Alternatively, we could adjust our algorithm to follow these steps:

1. Find the highest score
2. Find out how many people have the highest score
3. Create an array of the correct size
4. Fill up the array with all those with the highest score.

Separately, steps 1, 2 and 4 each require an iteration through the people in the list:

```
highestScore ← 0                                              // STEP 1 above
for index i from 0 to people.length-1 {
        if (people[i].score > highestScore) then
              highestScore ← people[i].score
}
highestCount ← 0                                              // STEP 2 above
for index i from 0 to people.length-1 {
        if (people[i].score = highestScore) then
              highestCount ← highestCount + 1
}
highestList ← new Array with capacity highestCount            // STEP 3 above
nextLocation ← 0                                              // STEP 4 above
for index i from 0 to people.length-1 {
        if (people[i].score = highestScore) then {
              highestList[nextLocation] ← people[i].name
              nextLocation ← nextLocation + 1
        }
}
print highestList
```

While this solution is more space-efficient, it is less efficient in terms of running time, as it requires an additional iteration through the list. However, with some careful thought, we can combine steps 1 and 2. Do you know how ?

```
highestScore ← 0                                                  // STEP 1 & 2 above
highestScoreCount ← 0
for index i from 0 to people.length-1 {
        if (people[i].score > highestScore) then {
                highestScore ← people[i].score
                highestScoreCount ← 0
        }
        if (people[i].score = highestScore) then
                highestScoreCount ← highestScoreCount + 1
}
highestList ← new Array with capacity highestScoreCount           // STEP 3 above
nextLocation ← 0                                                  // STEP 4 above
for index i from 0 to people.length-1 {
        if (people[i].score = highestScore) then {
                highestList[nextLocation] ← people[i].name
                nextLocation ← nextLocation + 1
        }
}
print highestList
```

Notice that we just needed to keep track of the number of people with the current high score … the same way as **nextLocation** was used in the previous example to store the number of people with the current highest score.

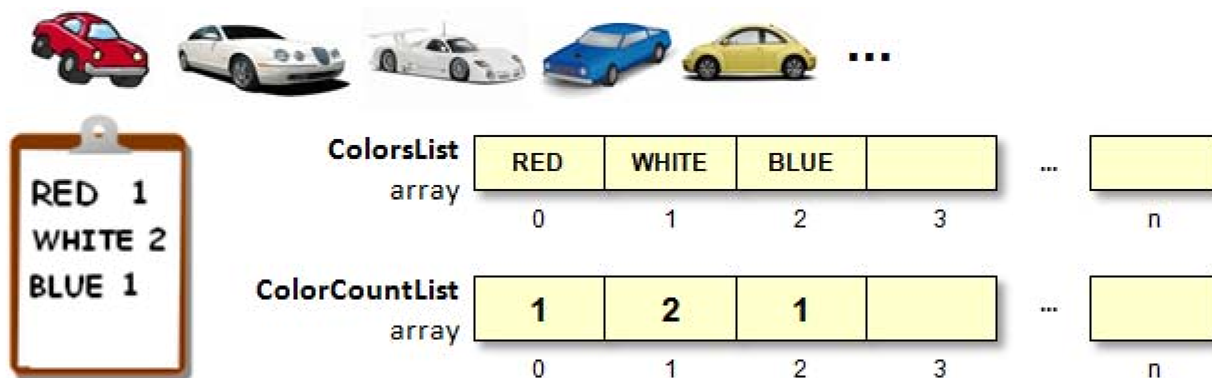
Example:

Recall our example for finding the most popular car color at the autoshow:

```
Algorithm: MostPopularColor
        cars:                    the list to search through

1.      colorsList ← new List
2.      colorCountList ← new List
3.      colorsCount ← CarColorCount()
4.      bestCountSoFar ← 0
5.      bestColorSoFar ← unknown
6.      for each color in the colorsList {
7.              count ← count in colorCountList corresponding to color
8.              if (count > bestCountSoFar) then {
9.                      bestCountSoFar ← count
10.                     bestColorSoFar ← color
                }
        }
11.     print bestColorSoFar
```

```
CarColorCount()
1.      for each car in the cars list {
2.          if (car.color is not in the colorsList) then
3.              add car.color to colorsList
4.          add 1 to colorCountList for the color corresponding to car.color
        }
5.      return the size of colorsList
```

Assume that the **colorsList** and **colorCountList** are now arrays:



The main algorithm remains largely similar, assuming that **CarColorCount()** returns the number of unique colors:

```
Algorithm: MostPopularColor
        cars:                   the list to search through

1.      colorsList ← new array with capacity cars.length
2.      colorCountList ← new array with capacity cars.length
3.      colorsCount ← CarColorCount()
4.      bestCountSoFar ← 0
5.      bestColorSoFar ← unknown
6.      for index i from 0 to colorsCount {
7.          if (colorCountList[i] > bestCountSoFar) then {
8.              bestCountSoFar ← colorCountList[i]
9.              bestColorSoFar ← colorsList[i]
            }
        }
10.     print bestColorSoFar
```

How though can we adjust the **CarColorCount()** function to use arrays ? The code for checking whether a car is in the list or not, is not as simple as this:

```
if (car.color is not in the colorsList) then …
```

We are going to have to go through the whole **colorsList** to determine if the car's color is already in there. This requires an additional loop (see steps 2 though 5 below) with some sort of boolean flag as follows:

```
CarColorCount()

1.    for each car in the cars list {
2.        found ← false
3.        for each color in the colorsList {
4.            if (car.color = color) then
5.                found ← true
          }
6.        if (not found) then
7.            add car.color to the colorsList
8.        add 1 to colorCountList for the color corresponding to car.color
      }
9.    return the size of colorsList
```

This code will add the color to the **colorList** when **found** has not be changed to **true**. The list will therefore contain the proper colors when completed.

But we should adjust the code so that the **colorsList**'s colors are accessed properly by integer indices:

```
CarColorCount()

1.    for each car in the cars list {
2.        found ← false
4.        for index j from 0 to colorsList.length-1 {
5.            if (car.color = colorsList[j]) then
5.                found ← true
          }
6.        if (not found) then
7.            add car.color to the colorsList
8.        add 1 to colorCountList for the color corresponding to car.color
      }
9.    return the size of colorsList
```

But there is a problem on line **7**. Where in the **colorsList** do we add the newly found color ? Logically, we should add it beside the last color that we added. But nowhere do we keep track of *where* the last color was added. So, we should maintain a variable that indicates where to add the next color. This variable will also serve as a counter that indicates how many colors are currently in the list. That is good, because line 9 requires us to return that value:

```
CarColorCount()

1.    colorsCount ← 0
2.    for each car in the cars list {
3.        found ← false
4.        for index j from 0 to colorsCount {
5.            if (car.color = colorsList[j]) then
6.                found ← true
          }
7.        if (not found) then {
8.            colorsList[colorsCount] ← car.color
9.            colorsCount ← colorsCount + 1
          }
10.       add 1 to colorCountList for the color corresponding to car.color
      }
11.   return colorsCount
```

Notice how the loop in line **4** needs only to go from **0** to the number of colors currently in the list (i.e., **colorsCount**), not its capacity.

We are almost done. Line 10 is the last line to fix. In order to increase the count in the **colorsCountList** we need to know the position of **car.color** in the **colorsList**. So we need to, not only determine whether or not a color is *already in* the list, but also *where* (i.e., its index) it is in the list.
So the boolean **found** flag will need to be an integer index instead:

```
CarColorCount()

1.    colorsCount ← 0
2.    for each car in the cars list {
3.        colorPosition ← -1
4.        for index j from 0 to colorsList.length-1 {
5.            if (car.color = colorsList[j]) then
6.                colorPosition ← j
          }
7.        if (colorPosition = -1) then {
8.            colorsList[colorsCount] ← car.color
9.            colorPosition ← colorsCount
10.           colorsCount ← colorsCount + 1
11.           colorCountList[colorPosition] ← 0
          }
12.       colorCountList[colorPosition] ← colorCountList[colorPosition] + 1
      }
13.   return colorsCount
```

Notice how **-1** is used to identify an invalid **colorPosition**. Also, notice how line **11** places a **0** in the **colorCountList** for all newly found colors. On line **12**, both new and repeated colors have their count incremented. That's it. The code is complete.

Example:

Consider a program that continually asks for integers from the user, and adds them to an array as they come in until **-1** is entered, and then does something interesting with the numbers that were entered. What is the structure of the algorithm ?

```
Algorithm: GetValidNumbers

1.    numbers ← new array with some initial capacity
2.    count ← 0
3.    entered ← get a number from the user
4.    while (entered is not -1) {
5.        numbers[count] ← entered
6.        count ← count + 1
7.        entered ← get a number from the user
      }
8.    do something with numbers
```

The code is straight forward. However, can you foresee a problem that may arise ? Notice that each number coming in is added to the array according to the **count** position, which is incremented each time a valid number arrives. When the loop has completed, **count** represents the number of integers that were entered and added to the array.

It is possible that we may attempt to add an item beyond the array's capacity. In such a case, the program will usually crash (or stop unexpectedly and non-gracefully).

How do we address this potentially serious problem ? Since arrays are fixed size, we cannot simply make more room within the existing array. Rather, a new *bigger* array must be created and all elements must be copied into the new array. But how much bigger ? It's up to us.

Here is how we could change the code to increase the array size by 5 each time it gets full …

```
Algorithm: GetValidNumbers

1.    numbers ← new array with some initial capacity
2.    count ← 0
3.    entered ← get a number from the user
4.    while (entered is not -1) {
5.        if (count >= numbers.length) then {
6.            tempArray ← new array with capacity numbers.length + 5
7.            for index i from 0 to numbers.length-1 {
8.                tempArray[i] ← numbers[i]
              }
9.            numbers ← tempArray
          }
10.       numbers[count] ← entered
11.       count ← count + 1
12.       enteredNumber ← get a number from the user
      }
13.   do something with numbers
```

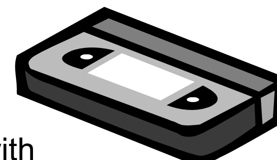
Of course, we can increase by any amount each time, perhaps even doubling the size.

Example:

Assume that we are given an array of Person objects and we need to discard (or remove from the array) all people below the age of 18.

```
Algorithm: RemoveYoungsters
      people:        the list to search through

1.    for each person in the people list {
2.        if (person.age < 18) then
3.            remove person from the people list
      }
```

How could we do this using an array ? Well, an array is fixed size, so when we remove data, the array will not get smaller. The situation is analogous to a program recorded on a videotape, we cannot actually remove the item but we can overwrite it (i.e., replace it) with a new value.

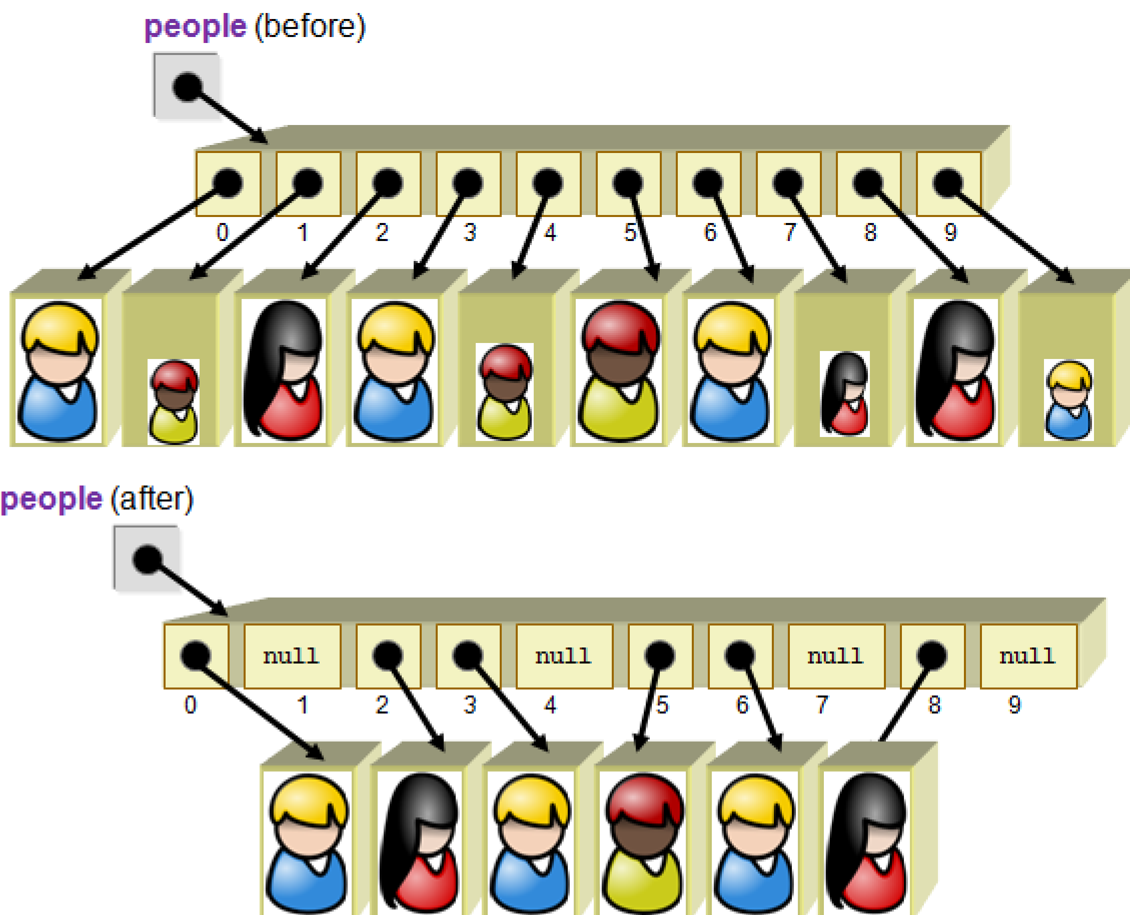


So, to delete a piece of information from an array, you usually replace it with **0** or **null**. The array will stay the same size, but the data will be deleted.

Algorithm: RemoveYoungstersArrayVersion
    people: the array to search through

1. **for** index **i** from **0** to **people.**length-1 {
2.   **if** (**people**[**i**].age < 18) **then**
3.   **people**[**i**] ← **null**
    }

This will replace all underage people with **null**, but the result array will be filled with “gaps” or holes:



The holes may present two problems. First, we don’t know how many people are left in the array. Second, when looping through the array we will encounter **null** objects that we need to deal with properly.

To handle the issue regarding the amount of valid data remaining in the array, we can always maintain a variable indicating the number of such valid elements as follows:

```
Algorithm: RemoveYoungstersArrayVersion2
        people:         the array to search through

1.      count ← 0
2.      for index i from 0 to people.length-1 {
3.              if (people[i].age < 18) then
4.                      people[i] ← null
5.              otherwise
6.                      count ← count + 1
        }
```

This **count** variable will indicate the number of people who are 18 or over. The code above assumes that the array was filled with people.

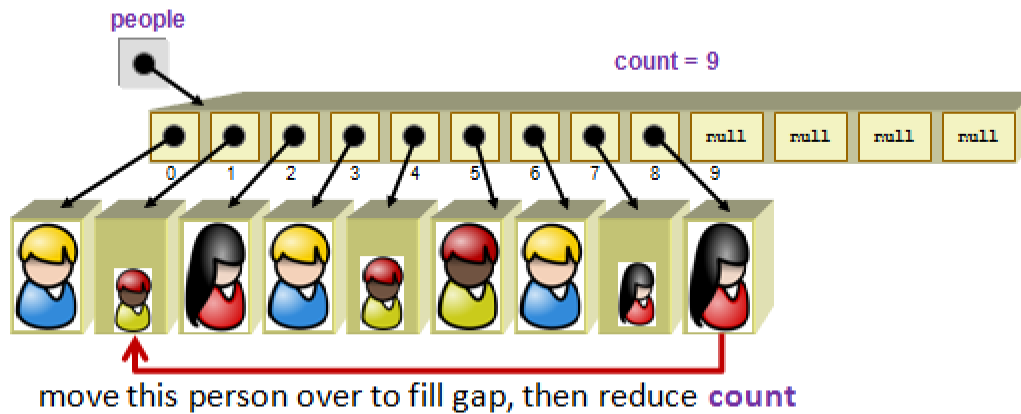
The second issue of being able to handle the holes implies that we check for a hole each time:

```
Algorithm: RemoveYoungstersArrayVersion3
        people:         the array to search through

1.      count ← 0
2.      for index i from 0 to people.length-1 {
3.              if (people[i] is not null) then {
4.                      if (people[i].age < 18) then
5.                              people[i] ← null
6.                      otherwise
7.                              count ← count + 1
                }
        }
```

However, it is often unpleasant to continually check for “holes” in the array like this. In fact, in a typical application we are likely more often going to need to traverse through elements of an array for display purposes than to search through the array to remove data. Therefore, it would be more advantageous to “fill-in” the hole each time so that the valid array elements are at the front-most part of the array at all times.

Assume that we have such a valid array in which all the elements are at the front-most part of the array. Assume then, that we always have a count as to how many people are stored in the array at all times.

Whenever we decide to remove a person, we can grab the person from the end of the array and place it in that spot, then reduce our counter by one to indicate that we have one less piece of data. We can even place **null** in the last location of the array to erase the data that was there.

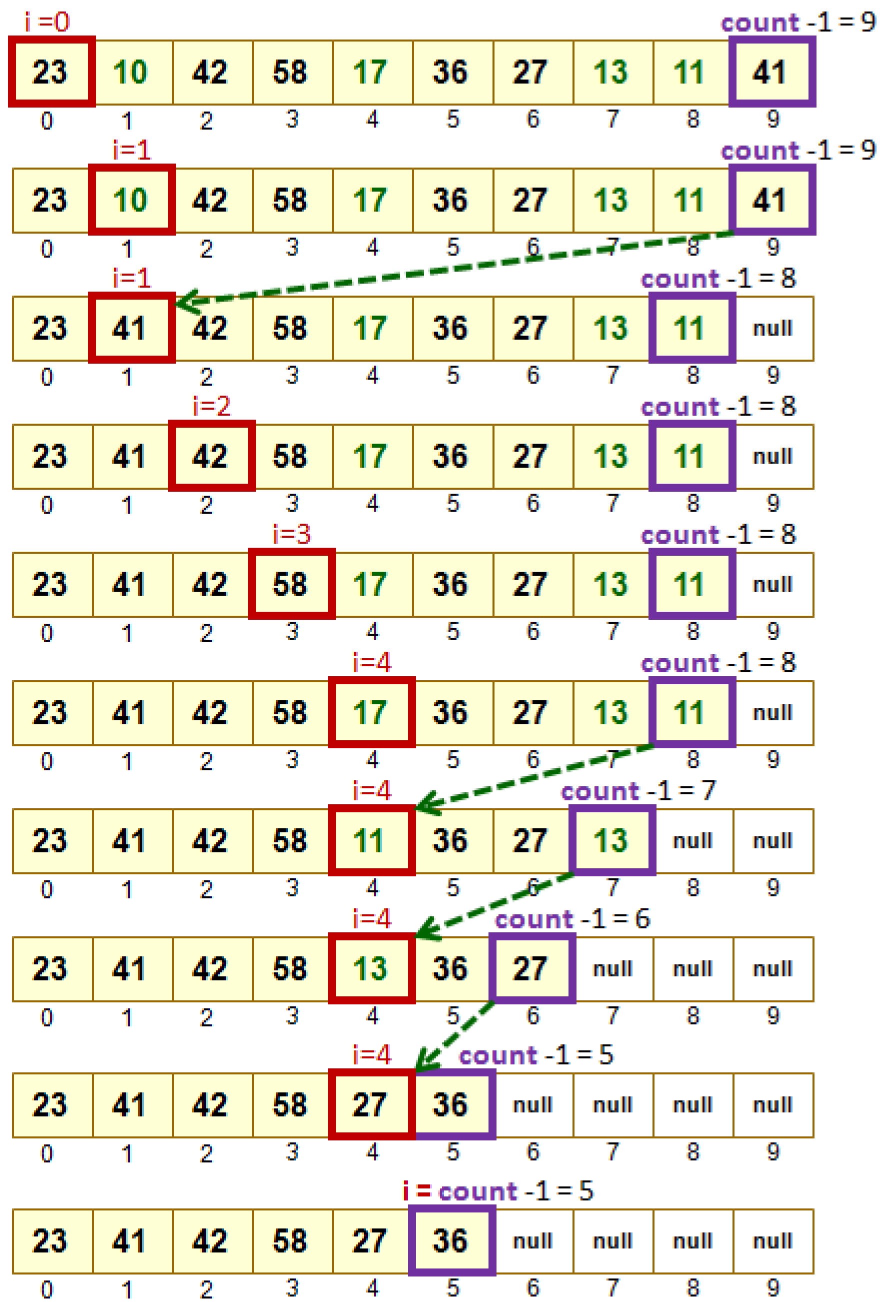


However, it is possible that the last person in the array that we try to move over is also underage. Therefore, we could go ahead and move it over, but make sure to check the age of the person that we moved over as well before we move on to the next index in the array. Here is the code:

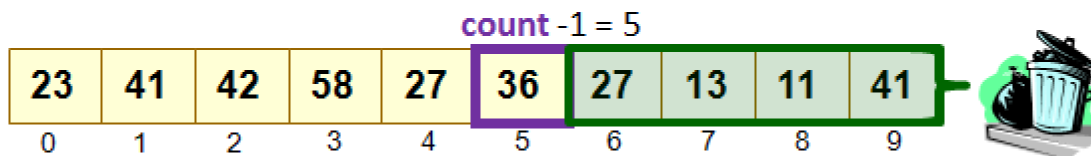
Algorithm: RemoveYoungstersArrayVersion4

```
        people:       the array to search through
        count:        number of people in the array

1.      for index i from 0 to count – 1 {
2.          if (people[i].age < 18) then {
3.              people[i] ← people[count – 1]
4.              people[count – 1] ← null
5.              count ← count - 1
6.              i ← i – 1      // decrease index so that next time we check this new item
            }
        }
```

Here is a picture of what is happening (the ages are shown in the array instead of the person in order to save space):

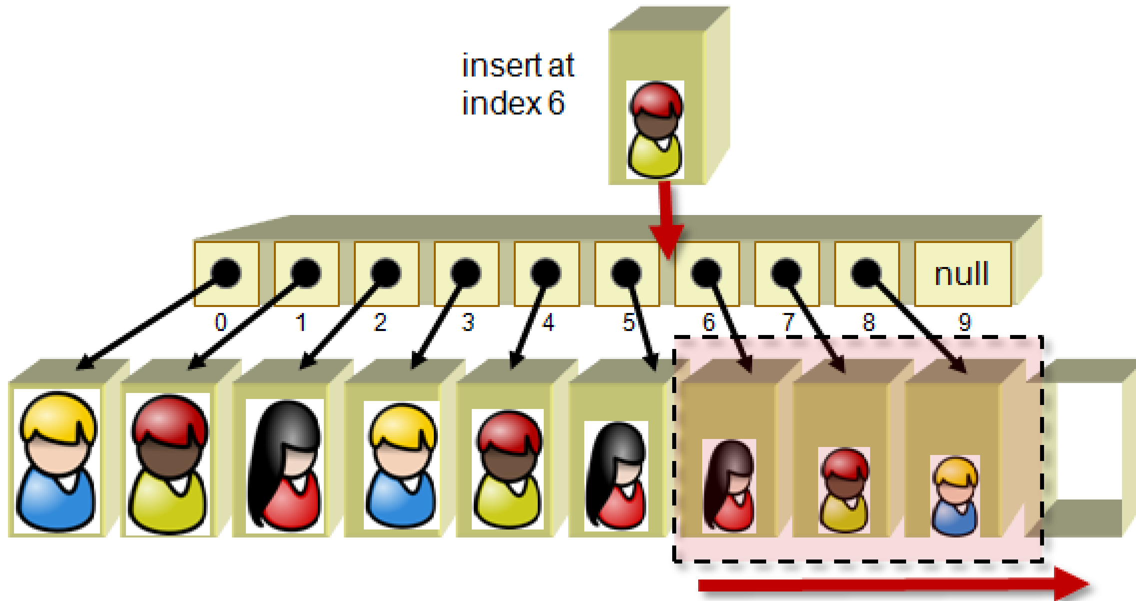


Notice how the count continually decreases as elements are removed from the array. It is not actually necessary to ***move*** the item to the open "holes". The item may simply be ***copied*** over, leaving "garbage" at the end of the array:

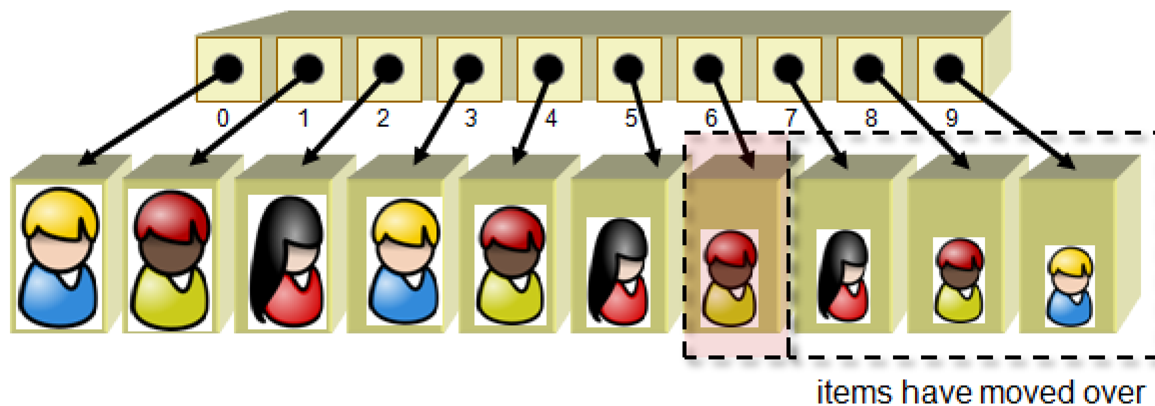


Example:

Assume that we already had a list of people sorted by their ages. How can we write code that will ***insert*** a new person into the appropriate location in the array so that it remains sorted ?



To insert at some index position (e.g., **i**) we must shift all array items from indices **i** onwards one position further in the array:



Here is the code to do this:

```
Algorithm: ArrayInsert
        people:         the array of sorted people
        newPerson:      the new person to insert into the array
        count:          number of items currently in the array

1.      if (count < people.length) then {
2.              index ← 0
3.              while (index < count) {
4.                      if (people[index].age < newPerson.age) then {
5.                              for index i from count-1 down to index {   // count backwards
6.                                      people[i+1]  ← people[i]
                                }
7.                              people[index]  ← newPerson
8.                              count  ← count + 1
9.                              index ← count       // Need this in order to quit
                        }
10.                     index ← index + 1           // Need this to check next position
                }
        }
```

Notice how the search for the insertion position continues until a person in the array has an age less than that of the person being inserted. Then every person after that "found" position is moved over to the right and the new person is simply added to that newly found position.

5.4 Efficient Searching Using Arrays

We have discussed linear searching and how it sometimes requires us to iterate through all the items in a list 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 kinda 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 an example of how the algorithm works if we were looking for the name **Green**:

Abraham	**Abraham**	Abraham	Abraham	Abraham
Bryant	Bryant	Bryant	Bryant	Bryant
Davidson	Davidson	Davidson	Davidson	Davidson
Flanders	**Flanders**	Flanders	Flanders	Flanders
Green	Green	**Green**	**Green**	**Green**
Johnson	Johnson	**Johnson**	**Johnson**	Johnson
Kent	**Kent**	**Kent**	Kent	Kent
Lemaire	Lemaire	Lemaire	Lemaire	Lemaire
Matthews	Matthews	Matthews	Matthews	Matthews
Orion	Orion	Orion	Orion	Orion
Peters	Peters	Peters	Peters	Peters
Robinson	Robinson	Robinson	Robinson	Robinson
Stevens	Stevens	Stevens	Stevens	Stevens
Thompson	Thompson	Thompson	Thompson	Thompson
Vernon	Vernon	Vernon	Vernon	Vernon
Walsh	Walsh	Walsh	Walsh	Walsh

Here is a more formal algorithm:

```
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.

Of course, this algorithm works for any collection of sorted items. Assume that the items are stored in an array. We can make the algorithm even more formal and compact as shown below. This pseudocode assumes that the items in the array can be compared using a **<** operator:

```
Algorithm: BinarySearch
        items:          the array to search through
        searchItem:     the item you are looking for

1.      s ← 0
2.      e ← items.length - 1
3.      location ← -1
4.      while ((location < 0) and (s <= e)) {
5.          m ← (s + e) / 2
6.          if (items[m] = searchItem) then
7.              location ← m
8.          else {
9.              if (items[m] < searchItem) then
10.                 s ← m + 1
11.             else
12.                 e ← m - 1
            }
        }
13.     print location
```

The code has been adjusted to return the **location** of the found item, not just a boolean showing whether or not it was found. Of course, if the item is not found, it will return **-1**.

Notice that it loops through the items one at a time and returns the same result. However, this code is simpler than the binary search. In order to see the advantage of the binary search, let us look at a real example.

Example:

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:

2	3	5	7	7	12	14	19	21	24	28	32	32	45	48	49
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Here is the result of applying the algorithm:

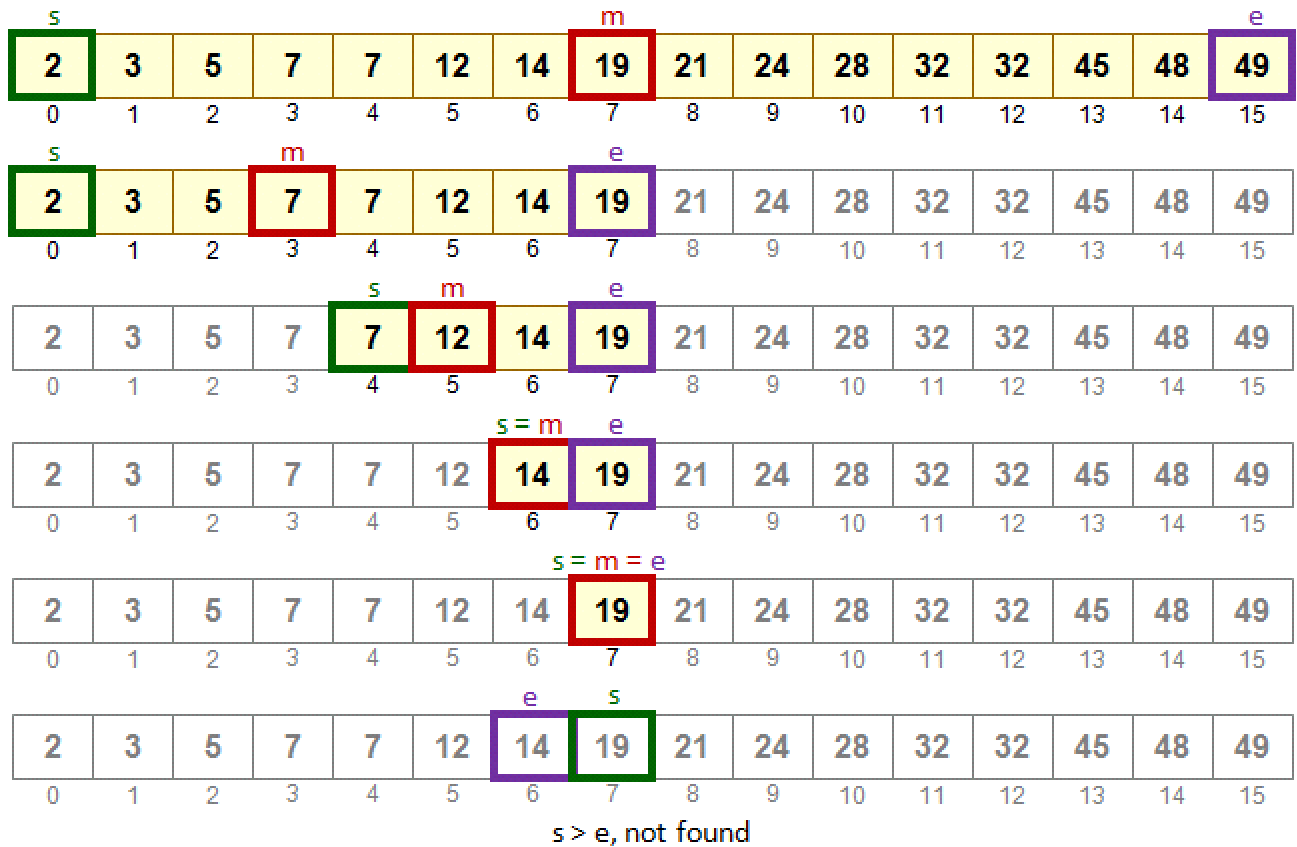
s							m								e
2	3	5	7	7	12	14	19	21	24	28	32	32	45	48	49
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

								s			m				e
2	3	5	7	7	12	14	19	21	24	28	32	32	45	48	49
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

								s	m	e					
2	3	5	7	7	12	14	19	21	24	28	32	32	45	48	49
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

									location = 9						
2	3	5	7	7	12	14	19	21	24	28	32	32	45	48	49
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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



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:

```
Algorithm: LinearSearch
        items:          the array to search through
        searchItem:     the item you are looking for

1.      i ← 0
3.      location ← -1
4.      while ((location < 0) and (i <= items.length - 1)) {
5.          i ← i + 1
6.          if (items[i] = searchItem) then
7.              location ← i
        }
13.     print location
```

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 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 **$\log_2$(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.

5.5 Arrays in Processing

Recall that in Processing & Java, variables are defined by specifying their type (primitive or object) as well as a label (the variable's name). Array variables are defined similarly, but with square **[ ]** brackets. Below is a table showing how to declare variables that hold a single value verses variables that hold arrays (or multiple values):

Single-value variables	Array variables

`boolean hungry;`
`int days;`
`char gender;`
`float amount;`
`double weight;`
 	
`Car myCar;`
`FullName myName;`
`Address myHomeAddress;`
`BankAccount myAccount;`
	`boolean[] hungry;`
	`int[] days;`
	`char[] gender;`
	`float[] amount;`
	`double[] weight;`
	
	`Car[] myCar;`
	`FullName[] myName;`
	`Address[] myHomeAddress;`
	`BankAccount[] myAccount;`

So you can see that arrays are declared similarly to regular variables. Note that the square brackets may appear either with the type (as shown) or with the variable's name as follows:

```
int   days[];
Car   myCar[];
```

Now these array variable declarations simply reserve space to store array objects, but it actually does not create the array object.

So, the following code would print out **null** because the variable is not yet initialized:

```
int[]   days;
println(days);
```

Notice above that we did **not** use the square brackets **[]** when you are ***using*** the array variable in your code ... you only use the brackets when we ***define*** the variable.

To ***use*** this variable (which is just like any other variable), we must give it a value. What kind of value should it have ? An array of course. But it may surprise you to know that we do not call a constructor to create arrays. Instead, we use a special (similar) syntax as follows:

```
new ArrayType[ArraySize]
```

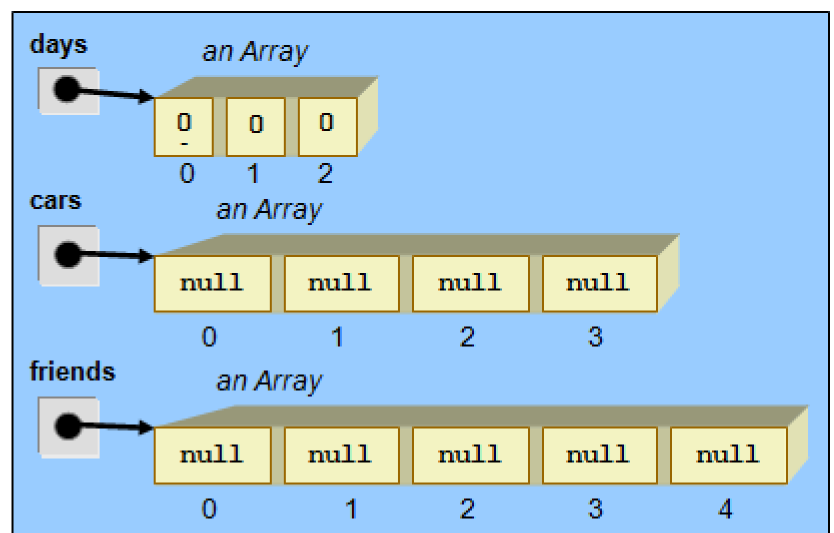
This is the template to create an array that can hold up to *ArraySize* values of type *ArrayType*. Remember that arrays are fixed size, so you cannot enlarge them later. If you are unsure how many items that you want to put into the array, you should chose an over-estimate (i.e., a maximum bound) for its size. Here are some examples of how to give a value to our 6 array variables by creating some fixed-size arrays:

```
int[]          days;
double[]       weights;
String[]       names;
Car[]          rentals;
Person[]       friends;

days = new int[30];        // creates array that can hold 30 ints
weights = new double[100];// creates array that can hold 100 doubles
names = new String[3];     // creates array that can hold 3 String objects
rentals = new Car[500];    // creates array that can hold 500 Car objects
friends = new Person[50]; // creates array that can hold 50 Person objects
```

Once we create arrays using the code above, the arrays themselves simply reserve enough space to hold the number of objects (or data types) that you specified. However, it does not create any of those objects!! The array is NOT initialized with new objects in each location. Newly created arrays are filled with 0 for numbered arrays, character 0 (i.e., the **null** character) for **char** arrays, **false** for **boolean** arrays and **null** for Object-type arrays.

Hence, the following code produces the result shown in the picture here:

```
int[]       days;
Car[]       cars;
Person[]    friends;

days = new int[3];
cars = new Car[4];
friends = new Person[5];
```



Notice that the arrays which hold object types are simply filled with **null** values … there are no **Car** objects nor **Person** objects created in the above code.

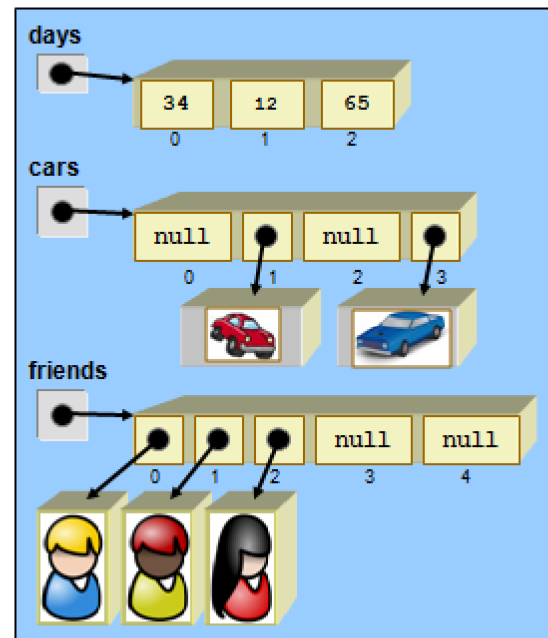
At any time, if we would like to ask an array how big it is (i.e., its size or capacity), we can access one of its special attributes called **length** as follows:

```
println(days.length);      // displays 3
println(cars.length);      // displays 4
```

Remember that the **length** of an array is its overall capacity, it is not the number of elements that you put into the array.

Values are actually assigned to an array using the **=** operator just as with other variables. Here is an example of how to fill in our arrays:

```
int[]       days;
Car[]       cars;
Person[]    friends;

days = new int[3];
cars = new Car[4];
friends = new Person[5];

days[0] = 34;
days[1] = 12;
days[2] = 65;
cars[1] = new Car("Red");
cars[3] = new Car("Blue");
friends[0] = new Person(...);
friends[1] = new Person(...);
friends[2] = new Person(...);
```



The picture here shows the result from this code →

Notice that we can insert an object at any location in the array, provided that the index is valid.

The following two lines of code would produce an **ArrayIndexOutOfBoundsException**:

```
days[3] = 87;                   // Error: index 3 is out of range
cars[10] = new Car("Yellow");   // Error: index 10 is out of range
```

A very common mistake made when learning to use arrays is to declare the array variable, but forget to create the array and then try to use it.   For example, look at this code:

```
Person[]  friends = new Person[10];
println(friends[0].firstName);    // Error!
println(friends[0].age);          // Error!
println(friends[0].gender);       // Error!
```

Although the above code does *create* an array that can *hold* **Person** objects, it actually never creates any **Person** objects.   Hence, the array is filled with 10 values of **null**.

The code will produce a **NullPointerException** because **friends[0]** is **null** here and we are trying to access the attributes of a **null** object. We are really doing this: **null.firstName** … which makes no sense.

Assigning individual values to an array like this can be quite tedious, especially when the array is large. Sometimes, in fact, we already know the numbers that we want to place in an array (e.g., we are using some fixed table or matrix of data that is pre-defined). In this case, to save on coding time JAVA allows us to assign values to an array at the time that we create it. This is done by using braces **{ }**. In this case, neither the **new** keyword, the **type** nor the **size** of the array are specified. Instead, we supply the values on the same line as the declaration of the variable. Here are some examples:

```
int[]       ages = {34, 12, 45};
double[]    weights = {4.5, 23.6, 84.124, 78.2, 61.5};
boolean[]   retired = {true, false, false, true};
String[]    names = {"Bill","Jennifer","Joe"};
char[]      vowels = {'a', 'e', 'i', 'o', 'u'};
```

Here, the array's size is automatically determined by the number of values that you specify within the braces, each value separated by a comma. So the sizes of the arrays above are 3, 5, 4, 3 and 5, respectively.

Objects may also be created and assigned during this initialization process as follows …

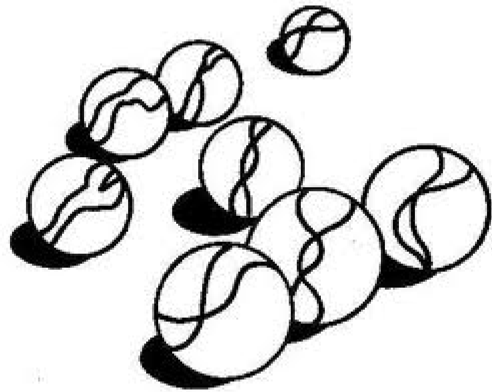
```
Person[]   people = {new Person("Hank", "Urchif", 19, 'M', false),
                     new Person("Don", "Beefordusk", 23, 'M', false),
                     new Person("Holly", "Day", 67, 'F', true),
                     null,
                     null};
```

Here we are actually creating three specific **Person** objects and inserting them into the first three positions in the **people** array. Notice that you can even supply a value of **null**, for example if you wish to leave some extra space at the end for future information. Hence the people array above has a capacity (i.e., length) of **5**.

Example:

Recall our example in which a ball was thrown around the window. How can we adjust the code so that **multiple balls** are able to be thrown around ?

```
final int   RADIUS = 40;
final float ACCELERATION = 0.10;

Ball        b;     // the Ball
boolean     grabbed;

class Ball {       // as defined earlier
      ...
}

void setup()() {
  size(600,600);
  aBall = new Ball(width/2, height/2,
                   random(TWO_PI), 10);
  grabbed = false;
}

void draw() { ... }
void mousePressed() { ... }
void mouseReleased() { ... }
```



To begin, we will need to store an array of balls. We can adjust **b** to be an array **balls**. Then in the **setup()** method, we need to create the array and fill it up with some balls:

<table>
<tr><th>One Ball Version</th><th>Array of Balls Version</th></tr>
<tr><td><pre>Ball   b;   // the Ball
boolean   grabbed;
...
void setup() {
  ...
  aBall = new Ball(width/2, height/2,
                   random(TWO_PI), 10);
  grabbed = false;
}</pre></td><td><pre>Ball[]   balls;   // lotsa balls
boolean   grabbed;
...
void setup() {
  ...
  balls = new Ball[10];
  for (int i=0; i<balls.length; i++)
    balls[i] = new Ball(width/2, height/2,
                        random(TWO_PI), 10);
  grabbed = false;
}</pre></td></tr>
</table>

Notice how we need to use a loop to fill in the array at each location. If we did not do this, the array would be filled with **null**. Each ball is made as before, and placed in the array at consecutive locations.

The **draw()** procedure will now need to draw ALL the balls. So we will need to loop through them all in order to ensure that all get displayed. Here is how the code gets adjusted:

One Ball Version

```
void draw() {
  background(0,0,0);
  ellipse(b.x, b.y, 2*RADIUS, 2*RADIUS);

  if (!grabbed) {
    b.x = b.x + int(b.speed * cos(b.direction));
    b.y = b.y + int(b.speed * sin(b.direction));
  }
  else {
    b.x = mouseX;
    b.y = mouseY;
  }

  b.speed = max(0, b.speed - ACCELERATION);

  if ((b.x + RADIUS >= width)||(b.x - RADIUS <= 0))
    b.direction = PI - b.direction;
  if ((b.y + RADIUS >= height)||(b.y - RADIUS <= 0))
    b.direction = - b.direction;
}
```

Array of Balls Version

```
void draw() {
  background(0,0,0);
  for (int i=0; i<balls.length; i++) {
    ellipse(balls[i].x, balls[i].y, 2*RADIUS, 2*RADIUS);

    if (!grabbed) {
      balls[i].x = balls[i].x + int(balls[i].speed * cos(balls[i].direction));
      balls[i].y = balls[i].y + int(balls[i].speed * sin(balls[i].direction));
    }
    else {
      balls[i].x = mouseX;
      balls[i].y = mouseY;
    }

    balls[i].speed = max(0, balls[i].speed - ACCELERATION);

    if ((balls[i].x + RADIUS >= width)||(balls[i].x - RADIUS <= 0))
      balls[i].direction = PI - balls[i].direction;
    if ((balls[i].y + RADIUS >= height)||(balls[i].y - RADIUS <= 0))
      balls[i].direction = - balls[i].direction;
  }
}
```

Notice how the **for** loop wraps around the code. Also, notice how each reference to **b** is simply replaced by **balls[i]** to indicate the particular ball in the array.

However, there is a problem in regards to grabbing the ball. What ball are we grabbing ? Should we be able to grab any ball ? Probably.

We will need to make a change so that instead of remembering ***whether or not we grabbed*** a ball, we should remember ***which ball we grabbed***, if any at all.

Therefore, we will change our **grabbed** boolean to be of type Ball, so that we can keep track of the particular ball that was grabbed:

```
boolean     grabbed;
```
will become
```
Ball   grabbed;
```

In the **setup()** procedure, we will set it to **null**, since we have not grabbed any balls when the program first starts.

```
grabbed = false;
```
will become
```
grabbed = null;
```

How do we change the **if (!grabbed)** line in the **draw()** procedure ? We want to move each ball except the one that was grabbed. So we move the ball at location **i** as long as it is not the grabbed one. So we need to change this line to **if (grabbed != balls[i])** in the **draw()** procedure.

Likewise, we need to adjust the **mousePressed()** procedure to search through all the balls and find out which one was grabbed, then remember it as follows:

One Ball Version

```
void mousePressed() {
  if (dist(b.x, b.y, mouseX, mouseY) < RADIUS)
    grabbed = true;
}
```

Array of Balls Version

```
void mousePressed() {
  for (int i=0; i<balls.length; i++) {
    if (dist(balls[i].x, balls[i].y, mouseX, mouseY) < RADIUS)
      grabbed = balls[i];
  }
}
```

Likewise, we need to adjust the **mouseReleased()** procedure to “un-remember” the grabbed ball as follows:

One Ball Version

```
void mouseReleased() {
  if (grabbed) {
    b.direction = atan2(mouseY - pmouseY, mouseX - pmouseX);
    b.speed = int(dist(mouseX, mouseY, pmouseX, pmouseY));
  }
  grabbed = false;
}
```

Array of Balls Version

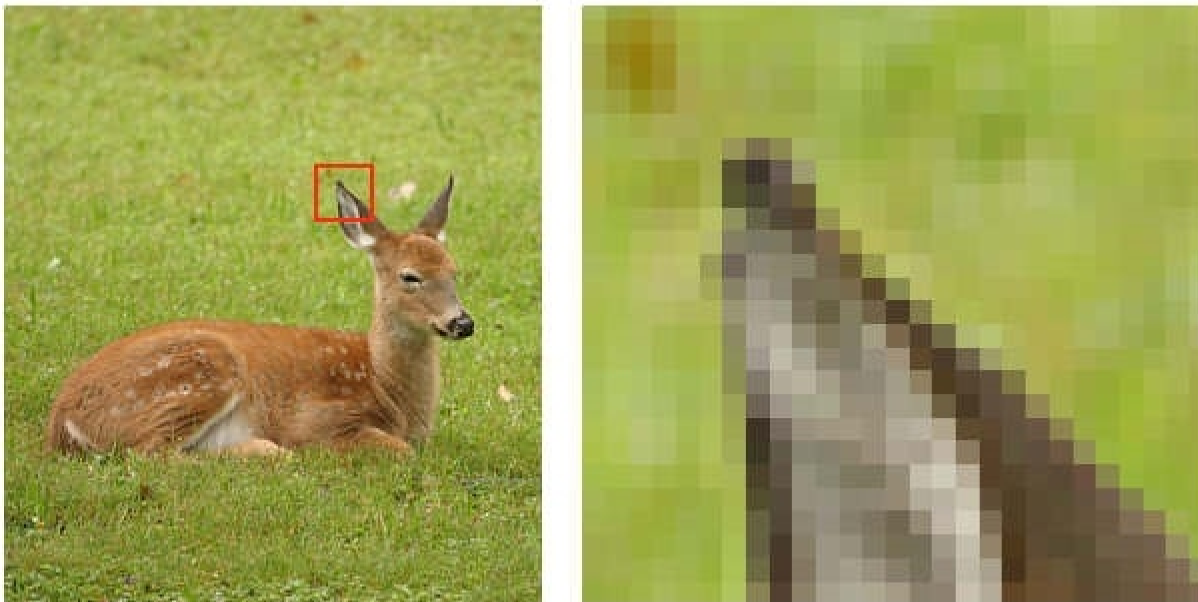
```
void mouseReleased() {
  if (grabbed != null) {
    grabbed.direction = atan2(mouseY - pmouseY, mouseX - pmouseX);
    grabbed.speed = int(dist(mouseX, mouseY, pmouseX, pmouseY));
  }
  grabbed = null;
}
```

Notice how we now use the grabbed variable to set the direction and speed of the grabbed ball according to where we throw it.

5.6 2D Arrays

In all of our search-related examples so far we assumed that we had a list of items that we are searching through. A list of items is considered 1-dimensional in that we travel one way through the list (although this could be done forwards or backwards). Sometimes, however, we need to perform 2-dimensional searches by traveling up, down, left or right through data that is usually arranged in a grid.

One particular class of problems that apply to 2-dimensional data is called ***image processing***. That is, there are many algorithms that process and manipulate images. All images can be stored in pixels arranged in an (x,y) grid:



Sometimes the images are compressed and stored differently (e.g., jpg) but in our examples, we will assume that the data is in what is known as a "raw" format where each pixel (representing a dot of the image) is accessible and modifiable by a unique (x,y) location in the image. Therefore, we can imagine that the image is stored in a 2-dimensional array as rows and columns of pixels. Therefore, we will access a particular pixel in the **image** by using this notation: **image**[**row**][**column**]

Here, each pixel requires two indices…one to represent the row (starting at zero), the other to represent the column (starting at 0 as well). While there are many ways to represent pixels in an image, we will assume for our examples that each pixel value is either stored as a:

- **grayscale** value: Usually a number from 0 to 255 representing the *amount of "gray"* at that location in the image.
- **RGB** value: A set of 3 numbers (each from 0 to 255) representing the *amount of "red", "green" and "blue"* at that location in the image.

Example:

How would we print out the grayscale values of a gray scale image stored in an array with **R** rows and **C** columns ? What does the loop structure look like ?

```
Algorithm: PrintGrayValues
        image:          the 2D array representing the image
        R:              number of rows in the image
        C:              number of columns in the image

1.      for each row r from 0 to R-1 {
2.          for each column c from 0 to C-1 {
3.              print image[r][c]
            }
        }
```

Nested **for** loops are usually everybody's favorite control structure for iterating through the elements of a 2-dimensional array.

The above code assumes that the value of the array is indeed the grayscale value. How would the code differ if the image was stored in RGB format ? This depends on how the data is actually stored. For example, it is possible that, instead of a single grayscale byte value per pixel, the array may actually hold a data structure with accessible r, g, b values:

```
define Pixel to be made of {
        .red
        .green
        .blue
}
```

In this case, we would simply need to access the appropriate values:

```
Algorithm: PrintRGBValues
        image:          the 2D array of Pixel objects representing the image
        R:              number of rows in the image
        C:              number of columns in the image

1.      for each row r from 0 to R-1 {
2.          for each column c from 0 to C-1 {
3.              print image[r][c].red
4.              print image[r][c].green
5.              print image[r][c].blue
            }
        }
```

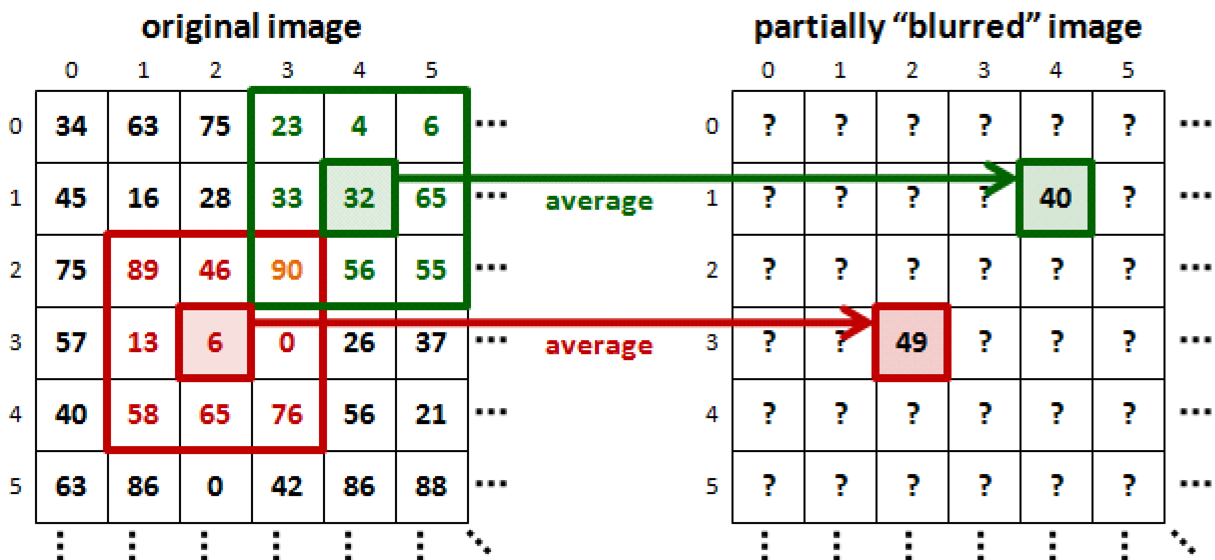
In some programming languages, however, the pixels may be stored as large 3-byte or 4-byte integers that encodes the RGB values in some manner (sometimes including gray or black level as a 4th byte). In this case, the programming language will often provide functions for extracting the relevant data and thus the code may look something like this:

```
Algorithm: PrintRGBValues
        image:          the 2D array containing the image
        R:              number of rows in the image
        C:              number of columns in the image

1.      for each row r from 0 to R-1 {
2.          for each column c from 0 to C-1 {
3.              print redAmount(image[r][c])
4.              print greenAmount(image[r][c])
5.              print blueAmount(image[r][c])
            }
        }
```

Regardless of the exact manner in which the color values are accessed, the processing of the image data occurs in a similar manner.

Example:

A common operation in image processing is that of "blurring" an image. There are multiple ways to blur an image. One way is simply to replace a pixel's value with a new value that corresponds to the average values of the pixels around it (including itself). A simple "box blur" would examine the pixels in the rows/cols that are above/below the pixel to be blurred. The number of rows and columns to use in the operation is determined by a parameter called the **window size**. Each pixel in the blurred image has a value that represents the average of the pixels within the window around it from the original image:

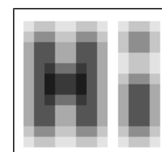


Imagine taking the following grayscale image and computing a blur for it. Here is the original image containing pixels that are either white (i.e., 255) or black (i.e., 0):

255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	0	0	255	255	0	0	255	255	255	0	0	255	255
255	255	0	0	255	255	0	0	255	255	255	0	0	255	255
255	255	0	0	255	255	0	0	255	255	255	255	255	255	255
255	255	0	0	255	255	0	0	255	255	255	255	255	255	255
255	255	0	0	0	0	0	0	255	255	255	0	0	255	255
255	255	0	0	0	0	0	0	255	255	255	0	0	255	255
255	255	0	0	255	255	0	0	255	255	255	0	0	255	255
255	255	0	0	255	255	0	0	255	255	255	0	0	255	255
255	255	0	0	255	255	0	0	255	255	255	0	0	255	255
255	255	0	0	255	255	0	0	255	255	255	0	0	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255

Here is the image after blurring:

255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	227	198	198	227	227	198	198	227	255	227	198	198	227	255
255	198	142	142	198	198	142	142	198	255	198	142	142	198	255
255	170	85	85	170	170	85	85	170	255	198	142	142	198	255
255	170	85	85	170	170	85	85	170	255	227	198	198	227	255
255	170	85	57	113	113	57	85	170	255	227	198	198	227	255
255	170	85	28	57	57	28	85	170	255	198	142	142	198	255
255	170	85	28	57	57	28	85	170	255	170	85	85	170	255
255	170	85	57	113	113	57	85	170	255	170	85	85	170	255
255	170	85	85	170	170	85	85	170	255	170	85	85	170	255
255	170	85	85	170	170	85	85	170	255	170	85	85	170	255
255	198	142	142	198	198	142	142	198	255	198	142	142	198	255
255	227	198	198	227	227	198	198	227	255	227	198	198	227	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255



You can see that the black pixels becomes lighter due to the fact that they have incorporated the white pixels around it in its new average value. Similarly, some of the white pixels became gray due to the averaging of nearby black pixels.

How could we write the code to do this assuming that the image is grayscale and is stored in an array ? We need to go through each pixel one-by-one…so a nested **for** loop is necessary. As we move through the pixels, we need to find the average of all pixels in the window around it and then set the new image's pixel to that averaged value. Here is the basic idea, although some details still need to be worked out:

```
Algorithm: BasicBlurGrayscaleImage
        image:          the 2D array representing the image
        R:              number of rows in the image
        C:              number of columns in the image
        windowSize:     width & height of the averaging window

1.      blurredImage ← new Array with capacity image.length
2.      for each row r from 0 to R-1 {
3.          for each column c from 0 to C-1 {
4.              sum ← 0
5.              for each pixel p in the window around and including image[r][c] {
6.                  sum ← sum + p
                }
7.              blurredImage[r][c] = sum / windowSize²
            }
        }
8.      image ← blurredImage
```

The loop on line 5 is not clear. How do we find each **p** pixel ?

One way is to consider the window pixels as being "offsets" in the array with respect to the pixel's location. So, for example, to get the new value for the pixel at row **3**, column **3** (see below) with a window size **w** of 5, we can consider offsets of **–w/2** to +**w/2** in both the rows and columns (which corresponds to **-2** to **+2** when **w=5**) as shown here →

	0	1	2	3	4	5	6
0	2		-w/2		+w/2		6
1	23	45	16	28	33	32	
2	32	75	89	46	90	56	-w/2
3	24	57	13	6	0	26	
4	44	40	58	65	76	56	+w/2
5	67	63	86	0	42	86	
6	56	69	22	27	73	56	92

Thus, we can use a **for** loop to iterate through the w^2 pixels in the window:

```
for each rowOffset from –w/2 to +w/2 {
    for each colOffset from –w/2 to +w/2 {
        … image[r+rowOffset][c+colOffset]
    }
}
```

Of course, there will be some issues along the borders of the image because adding the offsets will cause image coordinates that are less than zero or greater than the image's width or height →

-w/2

-w/2

+w/2

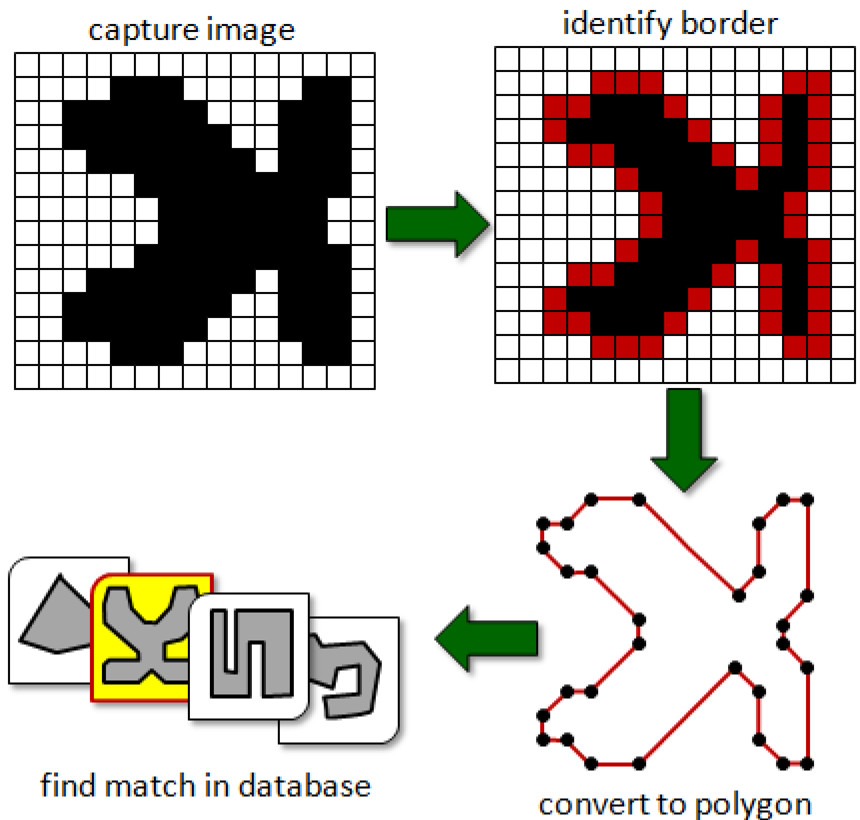
	0	1	2	3	4	5	6
0	2	34	63	75	23	4	6
1	23	45	16	28	33	32	65
2	32	75	89	46	90	56	55
3	24	57	13	6	0	26	37
4	44	40	58	65	76	56	21
5	67	63	86	0	42	86	88
6	56	69	22	27	73	56	92

So we will have to check for these "boundary" cases. We can either ignore them (i.e., don't average any pixels that are along such a border), or we can adjust the computation so that it also counts how many pixels are within the boundary and averages just those instead. We'll take the 2nd approach. So we will need to count how many valid pixels are within the window and use that to get the average. Here is the completed solution, making sure that the nested window loops are snuggled nicely within our outer nested loops:

```
Algorithm: BlurGrayscaleImage
        image:          the 2D array representing the image
        R:              number of rows in the image
        C:              number of columns in the image
        windowSize:     width/height of the averaging window

1.      blurredImage ← new Array with capacity image.length
2.      for each row r from 0 to R-1 {
3.          for each column c from 0 to C-1 {
4.              sum ← 0
5.              count ← 0
6.              for each rowOffset from –w/2 to +w/2 {
7.                  for each colOffset from –w/2 to +w/2 {
8.                      if (((r+rowOffset) >= 0) AND (r+rowOffset) < R) AND
                           ((c+colOffset) >= 0) AND ((c+colOffset) < C)) then {
9.                          sum ← sum + image[r+rowOffset][c+colOffset]
10.                         count ← count + 1
11.                     }
12.                 }
13.             }
14.             blurredImage[r][c] = sum / count
            }
        }
15.     image ← blurredImage
```

Example:

Imagine some machine parts moving along a conveyor belt. Assume that a robot must pick up a particular type part to assemble some product. Before the parts reach the robot, a camera takes images of the moving parts. The image is processed to a black and white silhouette showing the shape of a part. We would like to process the image to find the ***edges*** of the part so that we can compare it to a database of "known" parts and identify the shape for pickup by the robot arm.

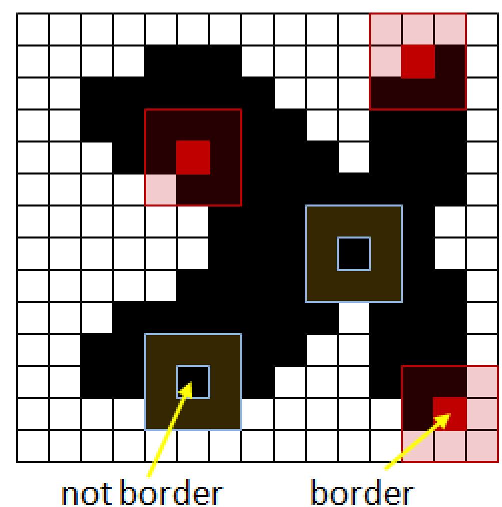


Let us consider the portion of this problem that determines the border (i.e., edge) of the object in the silhouette image.

There are many strategies for doing edge detection in an image. However, we will use an approach that computes a *chain* of pixels along an object's boundary. This will allow us to easily convert the pixel chain into a polygon and then compare with the polygons stored in the database.

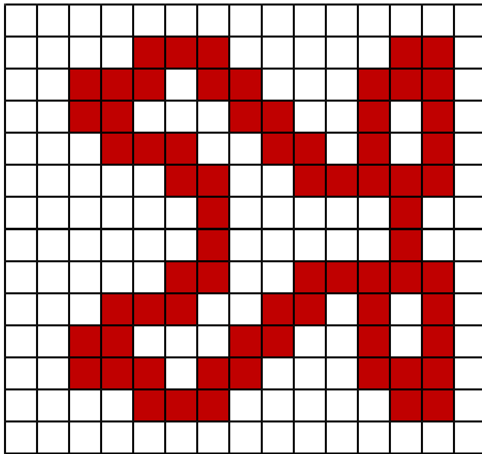
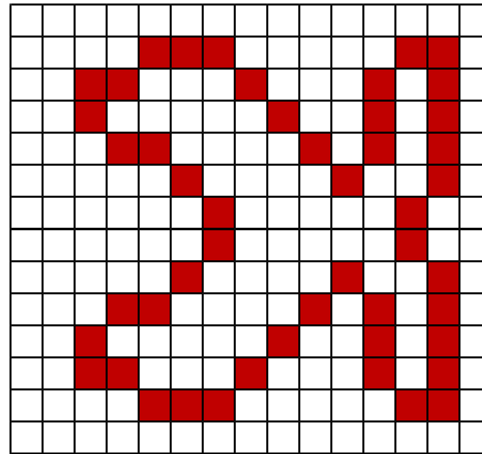
For now, assume that the image is given to us so that the entire part is within the boundaries of the image and that only one part is in the image … and the image is black and white (i.e., binary). How do we find the border pixels ?

One approach may be to simply process the image by looking at each black pixel and specifying that it is a border pixel if it is not completely surrounded by **8** black pixels all around it as shown here →



The result is that we could identify pixels that are clearly inside the part and those that are along the border.

Below, on the left, is the result. On the right is the "thinner" result that would be obtained if we "relaxed" our conditions a little and assume that a pixel is on the border only if the **4** pixels above or below are white (ignoring diagonals):

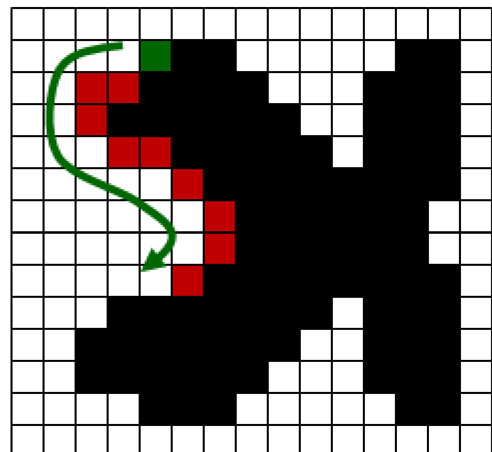



Here is the code that takes the black and white image and colors the border pixels red:

```
Algorithm: BordersGrayscaleImage
        image:          2D array representing the image
        R:              # rows in image
        C:              # columns in image

1.      blurredImage ← new Array with capacity image.length
2.      for each row r from 0 to R-1 {
3.          for each column c from 0 to C-1 {
6.              if (image[r][c] is black) then {
6.                  if ((image[r+1][c] is white) OR (image[r-1][c] is white) OR
8.                     (image[r][c+1] is white) OR (image[r][c-1] is white)) then
9.                          image[r+1][c] ← red
13.             }
            }
        }
```

The code above properly determines the border pixels. However, there is a problem. It does not give us the order of the pixels…so we cannot easily form a polygon (which has vertices and edges).

A better approach would be to start at some "beginning" edge pixel and then somehow trace around the border of the shape until we get back at the beginning pixel. →

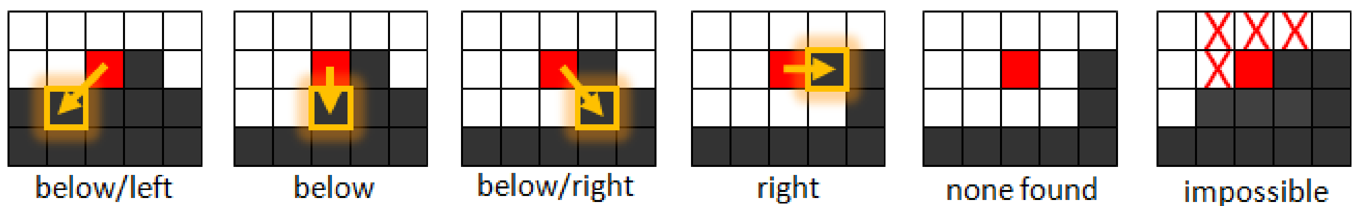


What pixel do we start the tracing at ? It does not matter, but for simplicity, we can choose the top-most/left-most pixel with some code like this:

Algorithm: FindStart

```
        image:      2D array representing image
        R:          # rows in image
        C:          # columns in image

1.      for each row r from 0 to R-1 {
2.          for each column c from 0 to C-1 {
3.              if (image[r][c] is black) then {
4.                  startR← r
5.                  startC← c
                    quit with startR, startC as starting pixel
                }
            }
        }
```

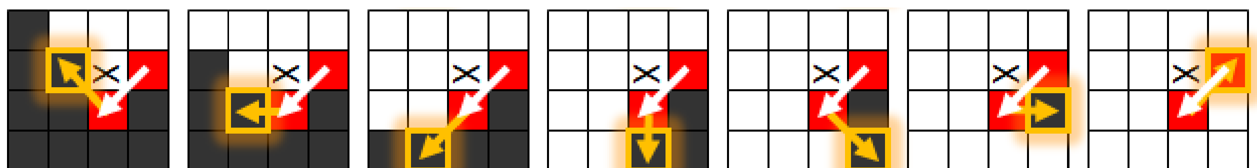
Now that we have the starting pixel, how do we start tracing ? Well, let us assume that we will trace the border counter-clockwise around the shape. Where should the next border point be ? Here are the possibilities:



The next pixel along the border depends on the shape itself. Above, we see that there are 5 possibilities … the 5th one being a very special case where the black pixel that we found was in fact a "stand-alone" black pixel … perhaps noise from the incoming data. The last image shows impossible situations … given that we started at the top left black pixel to begin with.

You may notice that all possibilities are below or to the right. You may also notice that we can simply check these 4 possibilities in sequence. Does the order matter ? Yes. We need to start by checking below/left first. Why ? Look at the last image above. What if we checked the below/right pixel first ? It is clearly not on the border of the shape.

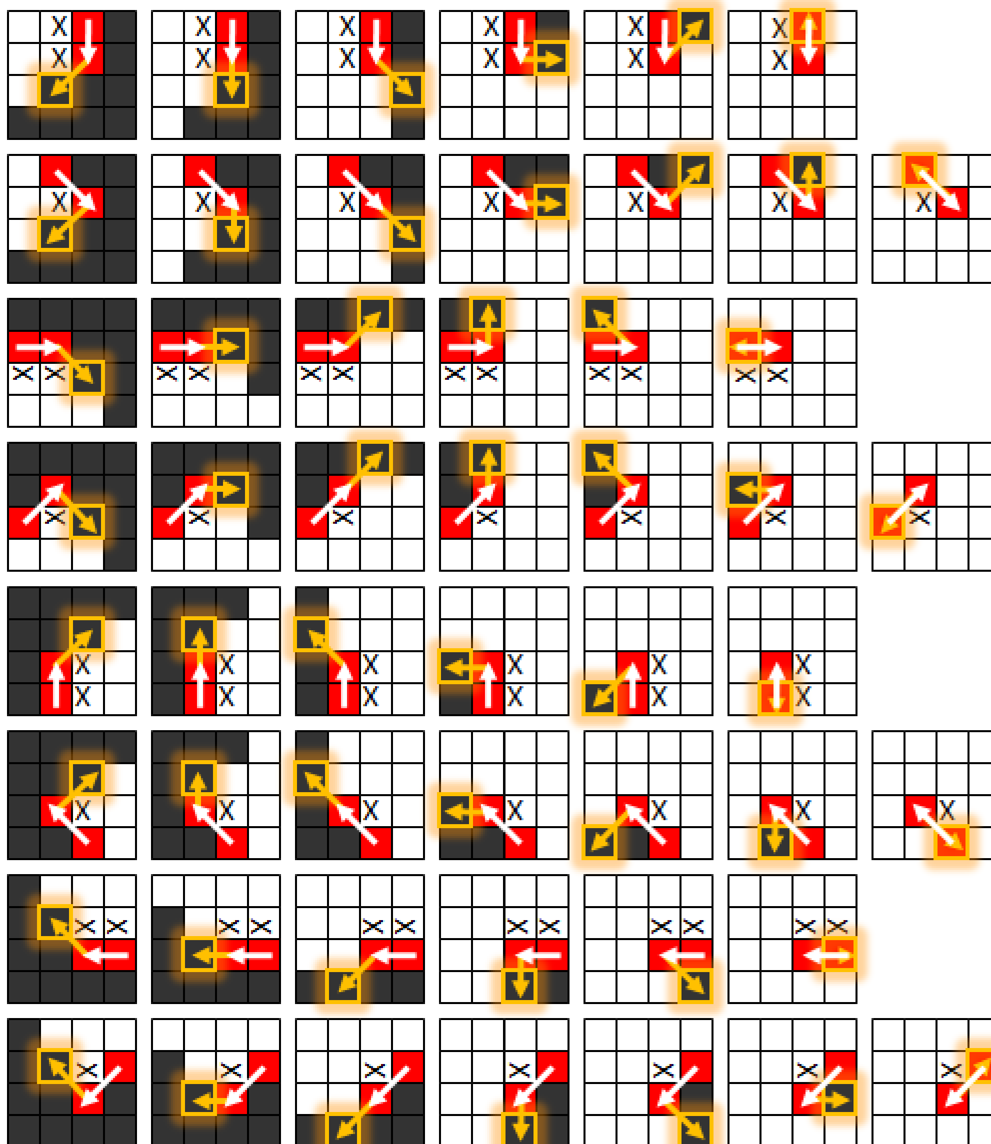
Assume then that we ended up finding a black pixel in the below/left position. We have our next point along the border. But then what ? We need to do a similar process again:



Notice now that there are more possibilities (7, in fact). So how do we know how to choose the next pixel ? We need to look around the last added pixel, again in a counter-clockwise order … this time starting with the upper-left pixel….until we find a black pixel. Notice that the pixel above (shown as X) is not a possible "next-pixel" because it would have been chosen beforehand in the border tracing.

So, the first time, we started looking in the bottom left first, the next time we started in the upper left … it is not consistent. In fact, the pixel that we start looking at first (in our trace around the last added border pixel) depends on the direction that we came from.

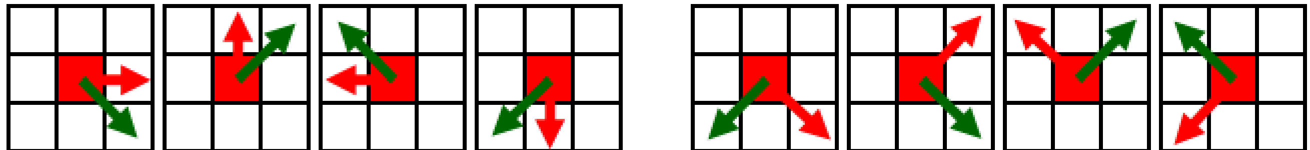
It is not as straight-forward and intuitive as it seems. Lets draw some possibilities:



It looks like a lot of possibilities, but every second set of pictures is just a 90° rotated version of a previous set. When solving problems in computer science, it is often necessary to write out these "test cases" and to try and make sure that you cover every possible solution. In this case, there are a small number of possibilities for each situation, so this is feasible. However, in some problems, it is not possible to figure out all possible situations. In time, you will get to

know which kinds of problems will have a small fixed number of possibilities and which problems are too difficult (or too time consuming) to determine all possible solutions.

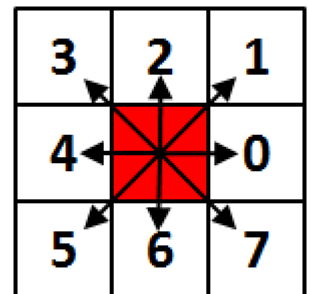
Given each of the 8 possible incoming directions, the leftmost pictures above show the direction to the next border pixel. Here is the series of images showing (red arrow) the ***incoming direction*** (from previous border pixel) and the corresponding ***start direction*** (to next potential border pixel) in which to start looking for the next border pixel:



So, to get the next pixel in the border, we could just use 8 **if** statements to figure it out:

```
if (incomingDirection is right) then
        startDirection ← lowerRight
if (incomingDirection is up) then
        startDirection ← upperRight
if (incomingDirection is left) then
        startDirection ← upperLeft
if (incomingDirection is down) then
        startDirection ← lowerLeft

if (incomingDirection is lowerRight) then
        startDirection ← lowerLeft
if (incomingDirection is upperRight) then
        startDirection ← lowerRight
if (incomingDirection is uperLeft) then
        startDirection ← upperRight
if (incomingDirection is lowerLeft) then
        startDirection ← upperLeft
```

However, there is an easier way. Consider numbering the pixels around the a pixel as shown here to the right →



In this case, the leftmost 4 pictures above will assign the start direction to be: **(*incomingDirection* + 7) modulo 8** and the rightmost pictures above will assign the start direction to be: **(*incomingDirection* + 6) modulo 8**. In both cases, the **modulo 8** ensures that the maximum value is **7**. It accounts for the "wrap around" effect.

Interestingly, the 4 left pictures above all have even numbered incoming directions, while the others are odd numbered. So, we can use this code instead of the 8 **if** statements above:

```
if (incomingDirection is even) then
        startDirection ← (incomingDirection + 7) modulo 8
otherwise
        startDirection ← (incomingDirection + 6) modulo 8
```

How do we check if a number is even or odd ? We can divide by 2 and check if the remainder is 0 or 1. So, **incomingDirection modulo 2 = 0** means that the **incomingDirection** is even.

OK, so we know the direction in which to start searching for the next border pixel. How do we actually do this search ? Well, we need to search counter clockwise until we find a black pixel, or until we checked all the 6 or 7 valid positions around the previous border pixel. In fact, we can check all 8 positions…it does not really matter, but 7 is sufficient to cover both cases.

But wait. How do we even check the pixel in the **startDirection** ? Where is that pixel in the array with respect to the last border pixel ? We need a way to convert the **startDirection** into an actual row and column in the array. Assume that the last border pixel is a row **r**, column **c**. Notice here, how the offsets are set up according to position (**r**, **c**) →

(r-1, c-1)	(r-1, c)	(r-1, c+1)
(r, c-1)	(r, c)	(r, c+1)
(r+1, c-1)	(r+1, c)	(r+1, c+1)

Again, we could set up a set of 8 if statements

```
if (startDirection is 0) then {
        startR ← r
        startC ← c + 1
}
if (startDirection is 1) then {
… etc…
```

However, there is a shorter way. We can set up 2 arrays of constant offsets and then access them using the **startDirection** as an index. This is called a ***lookup table***.

Here are the tables:

```
rowOffsets = new array with numbers: [0, -1, -1, -1, 0, 1, 1, 1]
colOffsets = new array with numbers:  [1, 1, 0, -1, -1, -1, 0, 1]
```

Then we simply compute the next potential border pixel as:

```
image[r + rowOffsets[startDirection]][c + colOffsets[startDirection]]
```

Now all we need to do is to loop counter-clockwise around the last border pixel, looking for a black pixel, starting with the **incomingDirection**. Here is a function that takes the last-added pixel (**r**, **c**) in the image along the border that is being traced and a starting **incomingDirection** and then determines the next pixel in the image.

The code returns a new point which represents the location of the next pixel along the border during the tracing procedure. It also updates the **incomingDirection** to be the last **startingDirection** in order to get ready for the next pixel along the border trace.

```
Algorithm: FindNextPixel
      image:                2D array representing the image
      incomingDirection:    direction coming in to this pixel
      r:                    row# of pixel in image
      c:                    column# of pixel in image

1.    rOff = new array with numbers: [0, -1, -1, -1, 0, 1, 1, 1]
2.    cOff = new array with numbers:  [1, 1, 0, -1, -1, -1, 0, 1]
3.    if (incomingDirection modulo 2 is 0) then
4.          startDirection ← (incomingDirection + 7) modulo 8
      otherwise
5.          startDirection ← (incomingDirection + 6) modulo 8

6.    count ← 0
7.    found ← false
8.    while (count < 8 AND found is false) {
9.          if (image[r + rOff[startDirection]][c + cOff[startDirection]] is black) then
10.               found ← true
            otherwise {
11.               count ← count + 1
12.               startDirection ← (startDirection + 1) modulo 8
            }
      }
13.   incomingDirection ← startDirection
14.   if (found is false) then
15.         return null
      otherwise
16.         return new point(r+rOff[startDirection], c+cOff[startDirection])
```

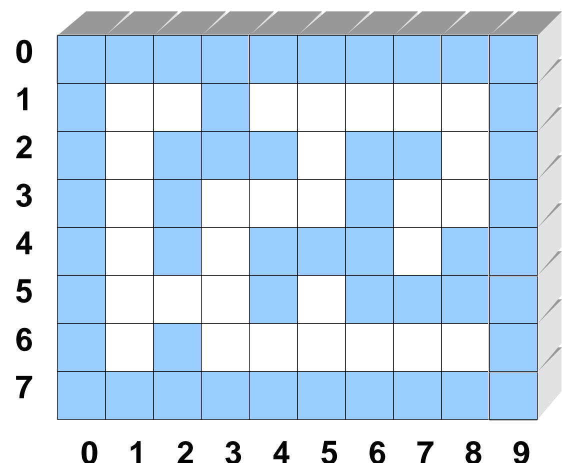
Now we have everything that we need for our algorithm. We know how to find the starting pixel and then to determine the direction to get the next pixel. We can do this until the start pixel is found again. We would need to set the **incomingDirection** to 7 as a start, since we started looking for the first black pixel from the top/left in the image. Here is the completed code:

```
Algorithm: TraceBorder
        image:          2D array representing the image
        R:              # rows in image
        C:              # columns in image

1.      polygon ← new polygon with no points
2.      startPixel ← null
3.      for each row r from 0 to R-1 {
4.          for each column c from 0 to C-1 {
5.              if ((startPixel is null) AND (image[r][c] is black)) then {
6.                  startPixel ← (r, c)
7.                  add startPixel to polygon
                }
            }
        }
8.      if (startPixel is not null)) then {       // handles case where image is all white
9.          direction ← 7
10.         done ← false
11.         while (done is false) {
12.             nextPixel ← findNextPixel(image, direction, r, c)
13.             if ((nextPixel is null) OR (nextPixel is same as startPixel)) then
14.                 done ← true
                otherwise
15.                 add nextPixel to polygon
            }
        }
16.     return polygon
```

Notice that lines 2 through 7 simply find the starting pixel (if there is one), while lines 9 through 15 do the tracing along the border. Each time, a new border pixel (i.e., **nextPixel**) is added to the polygon until the original **startPixel** is reached again. The code will also handle the case where the **startPixel** is the only black pixel…in this case **nextPixel** will be **null**.

Example:

The examples that we have seen so far are related to image processing. However, 2D arrays can be used for any situation in which data is arranged in a grid. For example, consider representing the following maze by using arrays →



In Processing, we could represent this maze by using a 2D array of **ints** indicating whether or not there is a wall at each location in the array (i.e., 1 for wall, 0 for open space).

Notice how we can do this by using the quick array declaration with the braces **{ }** as we did with one-dimensional (i.e., 1D) arrays:

```
int[][]      maze = {{1,1,1,1,1,1,1,1,1,1},
                     {1,0,0,1,0,0,0,0,0,1},
                     {1,0,1,1,1,0,1,1,0,1},
                     {1,0,1,0,0,0,1,0,0,1},
                     {1,0,1,0,1,1,1,0,1,1},
                     {1,0,0,0,1,0,1,1,1,1},
                     {1,0,1,0,0,0,0,0,0,1},
                     {1,1,1,1,1,1,1,1,1,1}};
```

This array represents a grid with 8 rows and 10 columns. Notice that there are more brace characters than with 1D arrays. Each row is specified by its own unique braces and each row is separated by a comma. In fact, each row is itself a 1D array.

We could display this array quite simply by iterating through the rows and columns:

```
for (int row=0; row<8; row++) {
    for (int col=0; col<10; col++)
       print(maze[row][col]);
    println();
}
```

```
1111111111
1001000001
1011101101
1010001001
1010111011
1000101111
1010000001
1111111111
```

Of course, we may want to display the maze with nicer-looking characters:

```
for (int row=0; row<8; row++) {
    for (int col=0; col<10; col++) {
        if (item == 1)
           print('*');
        else
           print(' ');
    }
    println();
}
```

```
**********
*  *     *
* *** ** *
* *   *  *
* * *** **
*   * ****
* *      *
**********
```

How though, would we draw the maze (as shown in the picture above) on the window in processing ? We would need to decide upon how big each square would be and then loop through drawing the squares in the appropriate color.

Here is a more complete program (assuming that the **maze** variable is set as shown earlier):

```
int  GRID_SIZE = 20;

void setup() {
  size(GRID_SIZE*10,GRID_SIZE*8);
  stroke(0);    // use a black border
}

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

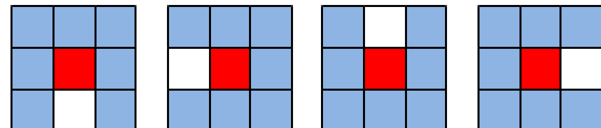
```
void drawMaze() {
  for (int row=0; row<8; row++) {
    for (int col=0; col<10; col++) {
        if (maze[row][col] == 1)
           fill(150,200,255);    // light blue
        else
          fill(255, 255, 255);   // white
        rect(col*GRID_SIZE, row*GRID_SIZE, GRID_SIZE, GRID_SIZE);
    }
  }
}
```

Notice how the **GRID_SIZE** is an adjustable parameter which is used as a scalar to make the maze appear larger or smaller.

Example:

Some mazes contain “dead-ends”. A “dead-end” is any location in the maze that has only one way to get into it (i.e., it is surrounded by 3 walls). Using the previous maze example, how can we adjust the code so that dead-ends are colored red ?

We need to identify a dead-end by checking the grid locations around it. There are exactly 4 cases that we should check as shown here →



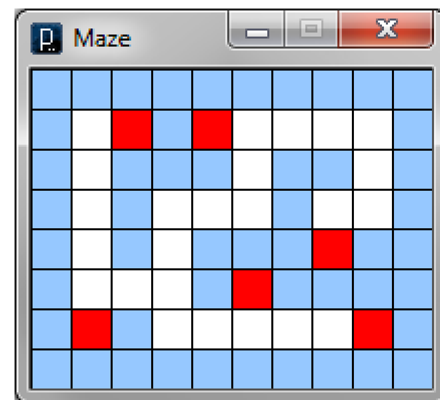
Given a particular grid location **maze[r][c]** that is an open space (i.e., **maze[r][c] = 0**) we can determine if it is a dead-end like this:

```
if (((maze[r-1][c] is a wall) AND (maze[r][c-1] is a wall) AND (maze[r][c+1] is a wall)) OR
   ((maze[r-1][c] is a wall) AND (maze[r+1][c] is a wall) AND (maze[r][c+1] is a wall)) OR
   ((maze[r+1][c] is a wall) AND (maze[r][c-1] is a wall) AND (maze[r][c+1] is a wall)) OR
   ((maze[r-1][c] is a wall) AND (maze[r+1][c] is a wall) AND (maze[r][c-1] is a wall))) then
       maze[r][c] is a dead end, so paint it red
```

This code checks each of the 4 cases by checking the grid locations above, below, to the right and to the left of location (r,c). Can we do this in a simpler way, given that all maze locations are represented as numbers either 0 or 1 ?

Yes. We can add up the maze locations of the 4 directions around (r,c). If their sum is 3, then this is a dead-end.

```
count ← maze[r-1][c] + maze[r+1][c] + maze[r][c-1] + maze[r][c+1]
if (count is 3) then
      maze[r][c] is a dead end, so paint it red
```



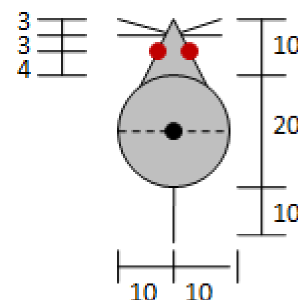
Of course, this assumes that **(r,c)** is not along the border, otherwise we may be trying to examine locations that are outside of the maze boundaries (e.g., r-1<0, c-1<0, r+1>7, c+1>7).

Of course, in our maze example, there are no open (i.e., white) locations along the border, so we would not need to worry about going beyond the boundaries. If however, the maze had openings along the border, we would have to check for such boundary issues like this:

```
if ((r>0) AND (r<7) AND (c>0) AND (c<9)) then {
        count ← maze[r-1][c] + maze[r+1][c] +
                    maze[r][c-1] + maze[r][c+1]
        if (count is 3) then
                maze[r][c] is a dead end, so paint it red
}
```

Example:

Assume that we had an underlying maze as in our previous example and that we wanted to move an object (e.g., a small rat) around in the maze by clicking on successive adjacent maze locations to move the rat.



To begin, here is what our rat will look like →

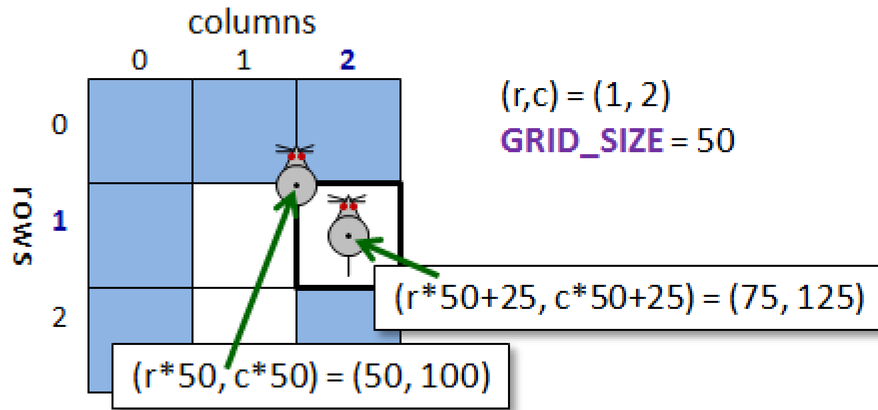
As we learned earlier in the course, the code for drawing this rat at a particular location is as easy as drawing a combination of circles, a triangle and some lines.

However, we would like to keep track of the rat's location in terms of the row and column that it lies in within the maze. Therefore, we will assume that the rat's position is recorded as a **(row,col)** in the maze and that we need to calculate the **(x,y)** position based on this row and column.

The **x** and **y** position of the rat's center (shown as the black dot above) can be computed as:

```
x ← col * GRID_SIZE  + (GRID_SIZE /2)
y ← row * GRID_SIZE + (GRID_SIZE /2)
```

The **GRID_SIZE** is the width and height (in pixels) of each square grid location. So, multiplying the row and column by the **GRID_SIZE** gives us the **(x,y)** location of the top left corner of the grid location. However, we likely want to draw the rat in the center of the grid location, so we need to add half of the **GRID_SIZE**:



Here is the code, then, for drawing the rat:

```
int  ratRow, ratCol;

void drawRat() {
  int   x = ratCol * GRID_SIZE + GRID_SIZE/2;
  int   y = ratRow * GRID_SIZE + GRID_SIZE/2;
  fill(150, 150, 150);  // gray
  triangle(x, y-20, x+10, y, x-10, y);      // head
  line(x, y+10, x, y+20);                   // tail
  line(x, y-17, x-10, y-17);                // whisker1
  line(x, y-17, x-10, y-20);                // whisker2
  line(x, y-17, x+10, y-17);                // whisker3
  line(x, y-17, x+10, y-20);                // whisker4
  ellipse(x, y, 20, 20);                    // body
  fill(255, 0, 0);  // red
  ellipse(x-3, y-14, 3, 3);                 // eye1
  ellipse(x+3, y-14, 3, 3);                 // eye2
}
```

How can we write the program that draws the rat at the location clicked on ?

Well, we know how to draw the maze and the rat… we just need to handle mouse-press events. At any time, the **draw()** method should always draw the rat at its current location in the grid. We only need to update that location when the user clicks on a new location.

```
int  GRID_SIZE = 50;
int  ratRow = 1, ratCol = 1;

void setup() { ... }
void drawMaze() { ... }
void drawRat() { ... }

void draw() {
  drawMaze();
  drawRat();
}

void mousePressed() {
      ... write code here ...
}
```

So, what do we do when the mouse is pressed ? We simply determine the **ratRow** and **ratCol** that the user clicked on. Once we change these variables to the new location, the **draw()** procedure should automatically draw the rat at the new location.

Determining the **(row, column)** from the **(mouseX, mouseY)** is not difficult. Recall that we found the **(x,y)** to draw the rat by multiplying the columns and rows by the **GRID_SIZE**. To do the reverse, we simply divide by the **GRID_SIZE**:

```
void mousePressed() {
  ratRow = mouseY / GRID_SIZE;
  ratCol = mouseX / GRID_SIZE;
}
```

Of course, we need to make sure that we move the rat to an "open" grid location. The rat should be allowed to move to any grid location that is not a wall:

```
void mousePressed() {
  int newRow = mouseY/GRID_SIZE;
  int newCol = mouseX/GRID_SIZE;
  if (maze[newRow][newCol] == 0) {
     ratRow = newRow;
     ratCol = newCol;
  }
}
```

If we are not simply selecting a location for the rat, but in fact steering it around the maze, we will want to ensure that we only move the rat to an adjacent grid location (i.e., up, down, left or right). Given that the rat is at location (r,c), the valid new locations are (r-1, c), (r+1, c), (r, c-1) and (r, c+1). Of course (r,c) is also a valid location, but the rat does not need to change locations in that case. Here is the code:

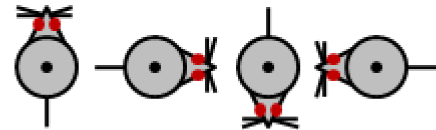
```
void mousePressed() {
  int newRow = mouseY/GRID_SIZE;
  int newCol = mouseX/GRID_SIZE;
  if (maze[newRow][newCol] == 0) {
     if (((newRow == ratRow+1) && (newCol == ratCol)) ||
         ((newRow == ratRow-1) && (newCol == ratCol)) ||
         ((newRow == ratRow) && (newCol == ratCol+1)) ||
         ((newRow == ratRow) && (newCol == ratCol-1))) {
        ratRow = newRow;
        ratCol = newCol;
     }
  }
}
```

Can this code be reduced ? Yes. We can make use of the **abs** function by examining the difference between the old row and the new row (same for columns). If the difference between the old and new row values is 1 and the column difference is 0, then this is a valid new position. Likewise, if the difference in columns is 1 and the rows is zero, then this too is a good position. So, we can add up the differences in rows and columns and make sure that the value is **1**.

```
void mousePressed() {
  int newRow = mouseY/GRID_SIZE;
  int newCol = mouseX/GRID_SIZE;
  if (maze[newRow][newCol] == 0) {
     if (abs(newRow-ratRow) + abs(newCol-ratCol) == 1) {
        ratRow = newRow;
        ratCol = newCol;
     }
  }
}
```

Example:

How can we adjust this program so that the rat turns to face the direction that it just travelled ?



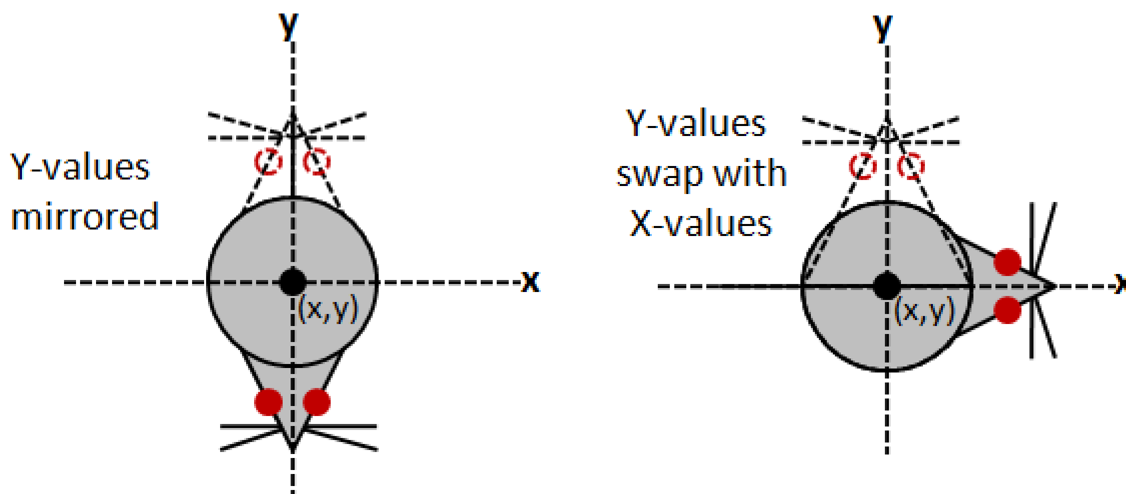
Since the rat travels in just 4 directions, we need to display it in one of 4 possible positions as shown here.

The only difference in the above code will be with respect to the **drawRat()** procedure. Recall that this code draws the rat centered around the **x** and **y** location of the center of a grid square, which is computed from the **ratRow** and **ratCol** variables:

Here is the code for drawing the rat facing upwards:

```
void drawRat() {
  int   x = ratCol * GRID_SIZE + GRID_SIZE/2;
  int   y = ratRow * GRID_SIZE + GRID_SIZE/2;
  fill(150, 150, 150);  // gray
  triangle(x, y-20, x+10, y, x-10, y);      // head
  line(x, y+10, x, y+20);                   // tail
  line(x, y-17, x-10, y-17);                // whisker1
  line(x, y-17, x-10, y-20);                // whisker2
  line(x, y-17, x+10, y-17);                // whisker3
  line(x, y-17, x+10, y-20);                // whisker4
  ellipse(x, y, 20, 20);                    // body
  fill(255, 0, 0);  // red
  ellipse(x-3, y-14, 3, 3);                 // eye1
  ellipse(x+3, y-14, 3, 3);                 // eye2
}
```

What changes when we want the rat to face downwards ? In reality, it is only the **y** offset values that need to be negated (i.e., mirrored with respect to the (x,y) origin). However, when we want to face the mouse right or left, the X and Y values of the various points will need to be swapped:

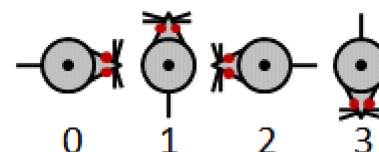


So, for example, looking at the code for drawing the triangle, here is how it changes depending on the direction that we want to draw the rat:

```
triangle(x+20, y, x, y-10, x, y+10);        // facing right
triangle(x, y-20, x+10, y, x-10, y);        // facing upwards
triangle(x-20, y, x, y+10, x, y-10);        // facing left
triangle(x, y+20, x-10, y, x+10, y);        // facing downwards
```

Since there are many coordinates that need to change like this (i.e., the line endpoints and ellipse centers), we will need to change the constants throughout the **drawRat()** procedure depending on the direction of the rat.

The simplest way to write the code is to duplicate the code 4 times (i.e., once for each direction). This, however, is a lot of duplication. We can do better. We can number the directions as shown here →



We would thus need to create the following variable to store the rat's current direction:

```
int  direction;    // 0 = right, 1 = up, 2 = left, 3 = down
```

Then, a better way to write the code would be to take advantage of the fact that directions 0 and 2 have negated offsets as well as directions 1 and 3. So, we can combine two of these by creating a multiplication factor **f** with a value of either 1 or -1 as follows:

```
void drawRat() {
  int   f = 1;
  int   x = ratCol*GRID_SIZE + (GRID_SIZE /2);
  int   y = ratRow*GRID_SIZE + (GRID_SIZE /2);
  fill(150, 150, 150);    // gray

  if ((direction == 1) || (direction == 2))
    f = -1;

  if ((direction == 1) || (direction == 3)) {
    triangle(x, y+f*20, x+10, y, x-10, y);
    line(x, y-f*10, x, y-f*20);      // tail
    line(x, y+f*17, x-10, y+f*17);  // whisker1
```

```
        line(x, y+f*17, x-10, y+f*20);  // whisker2
        line(x, y+f*17, x+10, y+f*17);  // whisker3
        line(x, y+f*17, x+10, y+f*20);  // whisker4
        ellipse(x, y, 20, 20);
        fill(255, 0, 0);   // red
        ellipse(x-3, y+f*14, 3, 3);
        ellipse(x+3, y+f*14, 3, 3);
      }
      else {
        triangle(x+f*20, y, x, y+10, x, y-10);
        line(x-f*10, y, x-f*20, y);  // tail
        line(x+f*17, y, x+f*17, y-10);  // whisker1
        line(x+f*17, y, x+f*20, y-10);  // whisker2
        line(x+f*17, y, x+f*17, y+10);  // whisker3
        line(x+f*17, y, x+f*20, y+10);  // whisker4
        ellipse(x, y, 20, 20);
        fill(255, 0, 0);   // red
        ellipse(x+f*14, y-3, 3, 3);
        ellipse(x+f*14, y+3, 3, 3);
      }
    }
```

This is not too bad as it requires half the code than if we were to duplicate the drawing code 4 times. Perhaps, an even better way may be to create a 2D array of these constant offsets where each row of constants refers to a rat direction. Here are two such arrays representing the **x** and **y** offsets, respectively:

```
int[][] xOff = {{20, 0, 0, -10, -20, 17, 17, 17, 20, 17, 17, 17, 20, 14, 14},
                {0, 10, -10, 0, 0, 0, -10, 0, -10, 0, 10, 0, 10, -3, 3},
                {-20, 0, 0, 10, 20, -17, -17, -17, -20, -17, -17, -17, -20, -14, -14},
                {0, -10, 10, 0, 0, 0, 10, 0, 10, 0, -10, 0, -10, 3, -3}};

int[][] yOff = {{0, 10, -10, 0, 0, 0, -10, 0, -10, 0, 10, 0, 10, -3, 3},
                {-20, 0, 0, 10, 20, -17, -17, -17, -20, -17, -17, -17, -20, -14, -14},
                {0, -10, 10, 0, 0, 0, 10, 0, 10, 0, -10, 0, -10, 3, -3},
                {20, 0, 0, -10, -20, 17, 17, 17, 20, 17, 17, 17, 20, 14, 14}};
```

Now we can easily make use of these offsets within a simpler procedure as follows:

```
void drawRat() {
  int   x = ratCol * GRID_SIZE + GRID_SIZE/2;
  int   y = ratRow * GRID_SIZE + GRID_SIZE/2;
  fill(150, 150, 150);  // gray
  int[]  xs = xOff[direction];
  int[]  ys = yOff[direction];
  triangle(x+xs[0], y+ys[0], x+xs[1], y+ys[1], x+xs[2], y+ys[2]);
  line(x+xs[3], y+ys[3], x+xs[4], y+ys[4]);      // tail
  line(x+xs[5], y+ys[5], x+xs[6], y+ys[6]);      // whisker1
  line(x+xs[7], y+ys[7], x+xs[8], y+ys[8]);      // whisker2
  line(x+xs[9], y+ys[9], x+xs[10], y+ys[10]);    // whisker3
  line(x+xs[11], y+ys[11], x+xs[12], y+ys[12]); // whisker4
  ellipse(x, y, 20, 20);
  fill(255, 0, 0);   // red
  ellipse(x+xs[13], y+ys[13], 3, 3);
  ellipse(x+xs[14], y+ys[14], 3, 3);
}
```

As you can see, now no code is duplicated, but it does require some extra offset numbers.

Interestingly, if we have a fixed number of directions, we can simply determine the **x** and **y** offsets for each additional direction and then increase the number of rows in the **xOff** and **yOff** arrays. Alternatively, we can calculate the **x** and **y** offsets for an arbitrary direction by using trigonometry. Then, we would not need the offset arrays. However, our code would become more complex and slower, requiring trigonometric calculations for each point of the rat.

All that remains to do is to adjust the **direction** as the user clicks on the rat's new location. Recall the code for handling mouse presses. When we find the new location to move to, we can simply compare the row and column that we are arriving at with the row and column we are leaving and this will determine the direction:

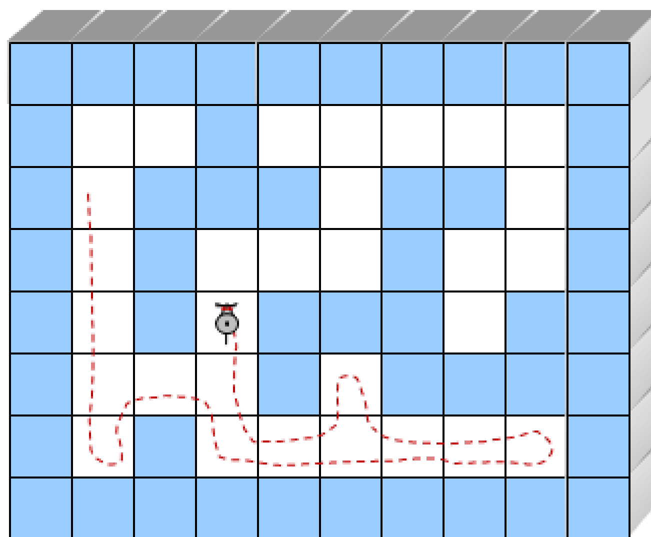
```
void mousePressed() {
  int newRow = mouseY/GRID_SIZE;
  int newCol = mouseX/GRID_SIZE;
  if (maze[newRow][newCol] == 0) {
     if (abs(newRow-ratRow) + abs(newCol-ratCol) == 1) {
        if (newRow > ratRow) direction = 3;
        else if (newRow < ratRow) direction = 1;
        else if (newCol > ratCol) direction = 0;
        else if (newCol < ratCol) direction = 2;
        ratRow = newRow;
        ratCol = newCol;
     }
  }
}
```

Example:

As a final example, how can we get the rat to travel through the maze on its own ? Assume that we simply want the rat to continuously travel around the maze without stopping. How can we do this ?

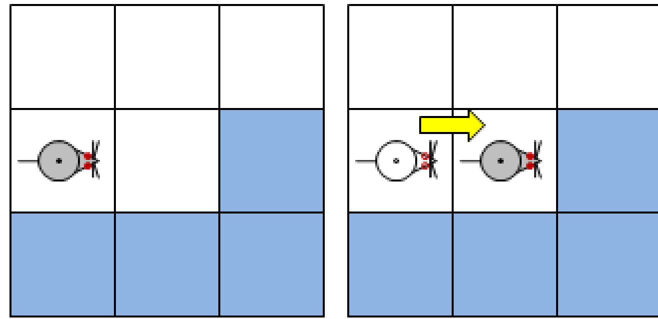
One well-known method of traveling through a maze is that of using the "right-hand rule". The right hand rule says that as long as we keep our right hand touching a wall as we walk, we will find a solution through the maze.

Does this work for all mazes ? I guess it depends on what we call a "maze".

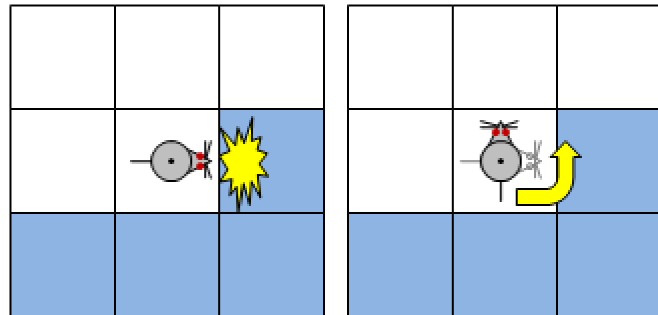


Mazes with inner loops cannot necessarily be solved using this strategy … it would depend on where the rat began following the walls.

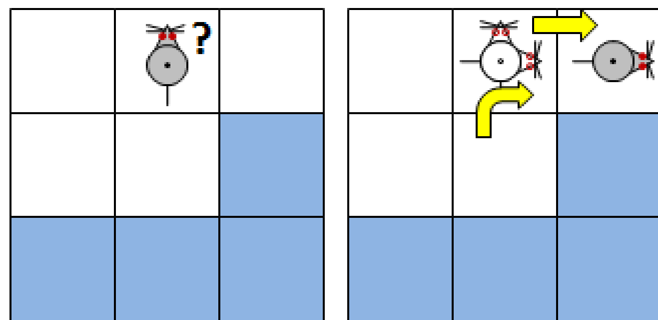
Let us consider how the rat will move around in the maze to follow the "right-hand rule". Assume that the rat has its "right hand" on a wall. If it is able to move forward, it should simply do so as shown here →



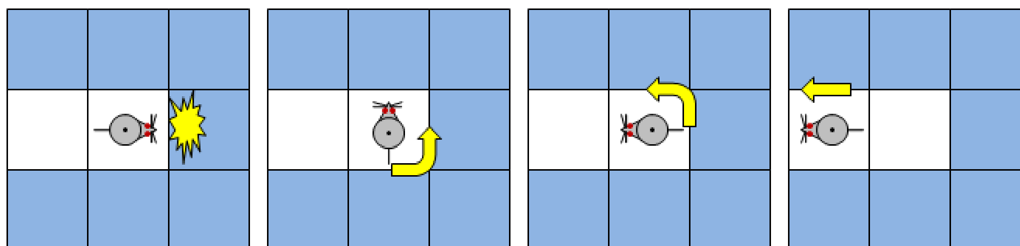
If however, the rat encounters a wall in front of it, then it should simply turn left →



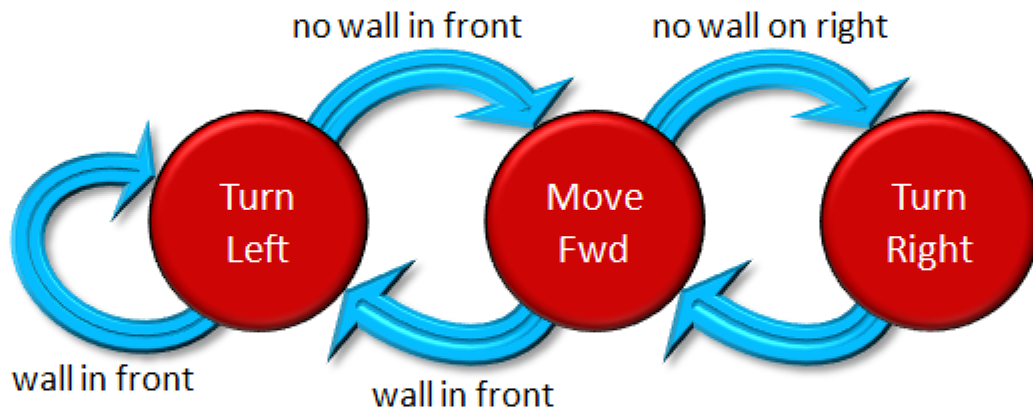
Then the rat will continue moving forwards as it now has its right hand on the wall again. It is possible that while traveling along it may lose contact with the wall →



In this case, notice that the rat needs to regain contact again with the wall on its right. First, it must turn right. However, the wall will still not be on the right of the rat after turning. Therefore it also needs to move forward. Then, it will be back-on-track again. These three cases actually encompass all situations. For example, if the rat ends up in a "dead-end", these three cases will get it out again.



So, how can we write an algorithm for traveling through this maze ? We simply follow the wall-following model that we developed and produce a state machine diagram:



The code follows directly from the state diagram:

Algorithm: TravelMaze

```
1.      repeat {
2.          if (there is a wall on the right) then {
3.              if (there is a wall in front) then
4.                  turnLeft()
5.              otherwise
6.                  moveForward()
            }
7.          otherwise {
8.              turnRight()
9.              moveForward()
10.         }
        }
```

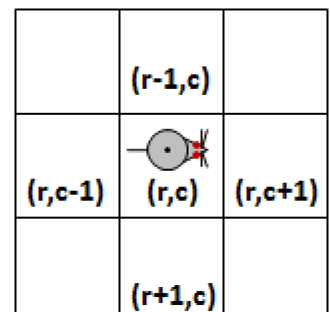
The code is straight forward. However, some details have been left out. For example, how do we know if there is a wall on the right or not ? It depends on the rat's current location in the maze as well as its current direction.

Assume that the rat has the direction and location defined in variables as before:

```
int  ratRow, ratCol, direction;
```

Also, assume that the maze has either 1 or 0 at each (row, column) location representing walls or open spaces, respectively.

How do we determine whether or not there is a "wall on the right" of the rat ? We would need to look at the position to its right in the maze. Assuming that the rat is at position **(r, c)**, then the relative positions around it are shown here →



So, we could write code like this:

```
if (((direction is 0) AND (maze[ratRow+1][ratCol] is 1)) OR
   ((direction is 1) AND (maze[ratRow][ratCol+1] is 1)) OR
   ((direction is 2) AND (maze[ratRow-1][ratCol] is 1)) OR
   ((direction is 3) AND (maze[ratRow][ratCol-1] is 1))) then {
```

To check whether or not there is a wall to the right of the rat's location. A similar check can be made for walls in front of the rat.

How do we make a turn ? We simply increase or decrease the direction. We can do this as follows:

```
direction ← (direction + 1) modulo 4;        // to turn left
direction ← (direction + 3) modulo 4;        // to turn right
```

As with checking maze locations right or ahead, moving forward will depend on the direction of the rat as well:

```
if (direction is 0) ratCol ← ratCol + 1
else if (direction is 1) ratRow ← ratRow - 1
else if (direction is 2) ratCol ← ratCol - 1
else if (direction is 3) ratRow ← ratRow + 1
```

So, we can merge all of this code together to come up with a complete solution. The solution here has duplicated code. It is possible to reduce this code quite easily by making use of procedures. See if you can shrink the code.

```
Algorithm: TravelMaze
        maze:                   the maze to travel through

1.      ratRow, ratCol ← any valid starting location in the maze with wall on right
2.      direction ← any valid starting direction in the maze with wall on right

3.      repeat {
4.          if (((direction is 0) AND (maze[ratRow+1][ratCol] is 1)) OR
5.              ((direction is 1) AND (maze[ratRow][ratCol+1] is 1)) OR
6.              ((direction is 2) AND (maze[ratRow-1][ratCol] is 1)) OR
7.              ((direction is 3) AND (maze[ratRow][ratCol-1] is 1))) then {

8.              if (((direction is 0) AND (maze[ratRow][ratCol+1] is 1)) OR
9.                  ((direction is 1) AND (maze[ratRow-1][ratCol] is 1)) OR
10.                 ((direction is 2) AND (maze[ratRow][ratCol-1] is 1)) OR
11.                 ((direction is 3) AND (maze[ratRow+1][ratCol] is 1))) then

12.                 direction ← (direction + 1) modulo 4;        // turn left

                otherwise {
13.                 if (direction is 0) then
14.                     ratCol ← ratCol + 1
15.                 otherwise if (direction is 1) then
16.                     ratRow ← ratRow – 1
17.                 otherwise if (direction is 2) then
18.                     ratCol ← ratCol – 1
19.                 otherwise if (direction is 3) then
20.                     ratRow ← ratRow + 1
                }
            }
            otherwise {
21.             direction ← (direction + 3) modulo 4;        // turn right
22.             if (direction is 0) then
23.                 ratCol ← ratCol + 1
24.             otherwise if (direction is 1) then
25.                 ratRow ← ratRow – 1
26.             otherwise if (direction is 2) then
27.                 ratCol ← ratCol – 1
28.             otherwise if (direction is 3) then
29.                 ratRow ← ratRow + 1
            }
        }
```