UNIVERSITY OF WATERLOO

Faculty of Mathematics
Waterloo, Ontario N2L 3G1

Centre for Education in
Mathematics and Computing

Grade 6 Math Circles

14/15 November, 2017

Algorithms

Solutions

Example 1

You're asked to sort the following list using a bubble sort algorithm:

{*Elephant, Beetle, Human, Mouse*}

Pass #1

We compare (**Comparison 1**) *Elephant* and *Beetle*. Since *Elephant* is bigger, it goes to the right by one step (**Swap 1**), giving:

{*Beetle, Elephant, Human, Mouse*}

Next we compare *Elephant* and *Human* (**Comparison 2**) and we swap them since *Elephant* is bigger (**Swap 2**):

{*Beetle, Human, Elephant, Mouse*}

We keep going. Next we compare *Elephant* and *Mouse* (**Comparison 3**) and swap them since *Elephant* is bigger (**Swap 3**):

{*Beetle, Human, Mouse, Elephant*}

We are now done with our first pass through the list. Notice that the biggest item in the list is now in its final position - it 'bubbles up' to its correct position in the list.

Pass #2

We start again with the first two elements of the list we got after Swap 3 i.e. *Beetle* and *Human*. We compare these two (**Comparison 4**) but we **do not** swap them because they're already in the right order. So our list doesn't change, yet.

Remember, the reason we're comparing the two even if they were in the right order is because the computer doesn't *know* that they're in the right order unless a comparison is made.

Next we compare *Human* and *Mouse* (**Comparison 5**) and we swap them since *Human* is bigger (**Swap 4**), giving:

{*Beetle, Mouse, Human, Elephant*}

Notice that our list is now sorted. But our algorithm doesn't know this yet so it continues.

We compare *Human* and *Elephant* (**Comparison 6**) but we don't swap them again because they're in the right order.

So far we've made 2 passes, 6 comparisons and 4 swaps to get a sorted list. For our purposes, this is our final answer.

Note: Actually, the computer will pass through the list once more and make 3 comparisons and 0 swaps. This is because the computer knows that the list is sorted only if 0 swaps need to be made in a pass through the list.

To understand this, consider sorting an already sorted list using the bubble sort algorithm. If the list contains (say) 5 items, the bubble sort will still make one pass and do 4 comparisons but 0 swaps (since the list is already sorted). This is how the algorithm *knows* that a list is sorted.

Example 2

In this example, you're asked to sort the same list as in Example 1 but using a selection sort.

We start with:

{*Elephant, Beetle, Human, Mouse*}

Pass #1

In a selection sort, we still consider two adjacent list items at a time. So we start by comparing *Elephant* and *Beetle* (**Comparison 1**). Since *Elephant* is bigger, we keep it in our sub-list and drop *Beetle*.

We now compare *Elephant* and *Human* (**Comparison 2**) and do the same thing i.e. keep *Elephant* since it's bigger.

Continuing, we compare *Elephant* and *Mouse* (**Comparison 3**) and keep *Elephant* as it's bigger.

Now that we're left with no more items in the list to compare, we conclude that *Elephant* is the biggest item in the list. We swap *Elephant* and *Mouse* (**Swap 1**) to put *Elephant* in it's correct position in the list.

We're now done with Pass #1. Our list looks like this:

{*Mouse, Beetle, Human, Elephant*}

Pass #2

We do the same thing as in Pass #1 but with the next biggest item in the list. So we start by comparing *Beetle* and *Human* (**Comparison 4**). We do not start with *Elephant* because we've already sorted it (we don't need to sort it again).

Since *Human* is bigger, we keep it and compare it to *Mouse* (**Comparison 5**). Again, since *Human* is bigger and there are no more items in the list to compare against, we conclude that *Human* is the second biggest item in the list.

Since *Human* is to the left of *Elephant*, we don't do a swap.

Note: Some versions involve self-swaps i.e. in the above case, *Human* would be swapped with itself as the correct position for an item in a list is defined differently.

At the end of Pass #2, our list looks exactly the same as after Pass #1:

{*Mouse, Beetle, Human, Elephant*}

Pass #3

Our third pass just involves two list items (since two are already sorted) i.e. *Beetle* and *Mouse*. We compare them (**Comparison 6**) and keep *Mouse* because it's bigger. Since we're left with no more items in the list to compare against, we conclude that the third biggest item in the list is *Mouse*.

We swap *Mouse* with *Beetle* to put it in its correct position. Our list now looks like this:

{*Beetle, Mouse, Human, Elephant*}

Notice that our list is already sorted. But our computer running the sorting algorithm doesn't know this just yet, so it keeps going. Since we're left with only one unsorted item, we know that it must be the smallest in our list, so we check to see if it's in the right position (which it is - extreme left).

Now we're done. We've made 3 passes, 6 comparisons and 2 swaps.

It's interesting to see that our selection sort made fewer swaps than our bubble sort on the same list. This is almost always true. Since swapping two items in a list takes more resources (memory, processing power) in a computer, it's typically more time consuming than a simple comparison.

Example 3

Looking at Examples 1 and 2, when run on the same list a selection sort algorithm makes fewer swaps than a bubble sort algorithm. Since swaps take the most time and resources, the algorithm with fewer swaps will usually be faster. So the computer running the selection sort algorithm will finish faster.

Example 4

The solution is obtained by applying all 4 rules to every square on the grid exactly three times. The resulting cell arrangement is shown in Figure 1 below:

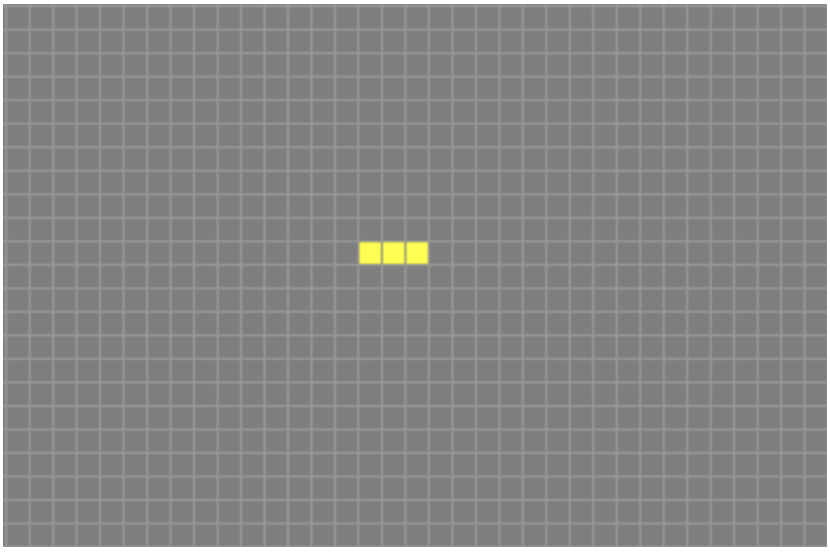

Figure 1: The initial state in Example 4 is an oscillator

This is an example of an oscillator - one of the important types of cell arrangements in the Game of Life.

Example 5

The final states are obtained by applying all 4 rules to every square on the grid exactly once and twice respectively. They are shown in Figures 2 and 3 below:

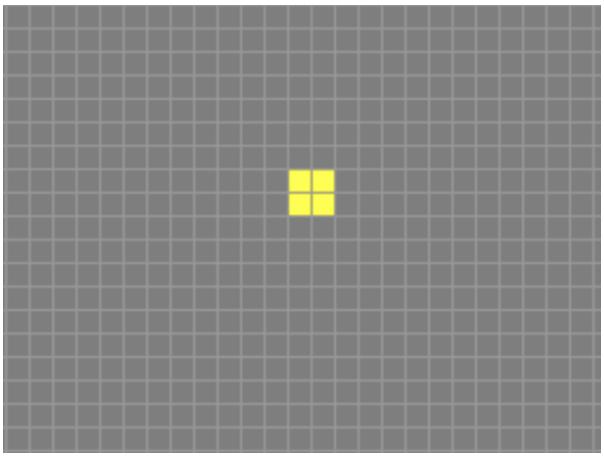

Figure 2: The final state after 1 generation

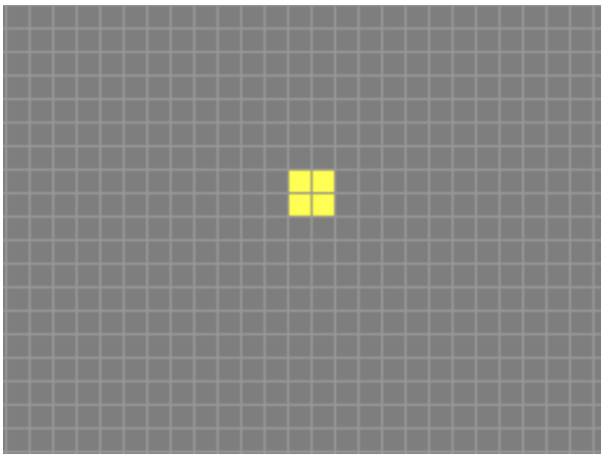

Figure 3: The final state after 2 generations

After 1 generation, the game doesn't change i.e. is stationary. Try using the rules on Figure 2 - you'll see that it doesn't change from one generation to the next. In fact, it doesn't ever change. Figure 2 is an example of a **stationary state** or **still-life**.

Problem Set

1. **Pass #1**

   We consider {5,2} and swap them (since 5 is bigger). We keep going and consider {5,4} (which we swap), and {5,1} (which we swap). Finally we consider {5,3} which we also swap. After the first pass, our list looks like this:

   $$\{2, 4, 1, 3, 5\}$$

   Since the list isn't yet sorted, we make another pass.

   Pass #2

   We consider {2,4} (no swap), {4,1} (swap) and {4,3} (swap). We finally consider {4,5} which we don't swap. This leaves our list looking like this:

   $$\{2, 1, 3, 4, 5\}$$

   Our list isn't sorted, so we keep going.

   Pass #3

   We consider {2,1} (swap), {2,3} (no swap), {3,4} (no swap) and {4,5} (no swap). This leaves us with the sorted list:

   $$\{1, 2, 3, 4, 5\}$$

2. **Pass #1**

   As with bubble sort, we consider two adjacent list items at a time. We first find the biggest item in the list and place it on the extreme right. Next we find the second biggest item... and so on.

   Starting with $\{2,\frac{1}{2}\}$ we keep 2 since it's bigger. We keep going and compare $\{2,\frac{3}{7}\}$ and $\{2,\frac{1}{100}\}$, keeping 2 each time. Finally, we compare {2,7} and keep 7 since it's bigger.

   Now technically, we swap 7 with itself (even though it's already in its correct position in the list). For now we can ignore these self-swaps as they make things complicated. Our final list looks like this:

   $$\{2, \tfrac{1}{2}, \tfrac{3}{7}, \tfrac{1}{100}, 7\}$$

After Pass #1, the biggest item in our list is sorted. Our entire list is still unsorted, so we keep going.

Pass #2

We compare $\{2,\frac{1}{2}\}$, $\{2,\frac{3}{7}\}$, $\{2,\frac{1}{100}\}$, keeping 2 each time. At the end of all three comparisons, we're left with 2 as the second biggest item in the list. So we swap 2 with $\frac{1}{100}$ giving us the following list:

$$\{\tfrac{1}{100}, \tfrac{1}{2}, \tfrac{3}{7}, 2, 7\}$$

We're still not done, so we make another pass.

Pass #3

We compare $\{\frac{1}{100}, \frac{1}{2}\}$ and keep $\frac{1}{2}$ since it's bigger. Next, we compare $\{\frac{1}{2}, \frac{3}{7}\}$ and still keep $\frac{1}{2}$ since it's bigger. So $\frac{1}{2}$ is the third biggest item in our list. To put it in its correct position, we swap it with $\frac{3}{7}$ giving us the following list:

$$\{\tfrac{1}{100}, \tfrac{3}{7}, \tfrac{1}{2}, 2, 7\}$$

Now our list is sorted. But, there's another technicality here - the algorithm continues making two more passes to sort $\frac{3}{7}$ and $\frac{1}{100}$ in that order. We'll ignore these for now, but remember that unless otherwise mentioned, self-swaps are allowed in a selection sort algorithm.

3. This question might seem silly because from the Examples we did earlier and the previous problem set questions, it's pretty obvious that the selection sort runs faster because it doesn't need as many swaps as bubble sort. But, it's not that simple.

   Look carefully at the list - only D and C are out of place. Swapping them should give a sorted list.

   Remember how in the earlier question we ignored the technicality of self-swaps in a selection sort algorithm. Well since this question asks us to decide which one to use, we can't ignore that anymore.

   A **Bubble Sort** algorithm would need to make 1 Pass, 4 comparisons and 1 swap to sort the given list.

   A **Selection Sort** algorithm would need to make 5 Passes, 10 comparisons and 5 swaps (including self-swaps) to sort the given list.

   This means that this is a rare case in which the bubble sort is actually faster than a selection sort.

So if you had to sort such a list, you'd want to pick the bubble sort algorithm.

4. (a) The next three numbers are:

$$13,\ 21,\ 34$$

   (b) The pattern is to start with 1 and 1 as the first two terms of the sequence. Then, add the previous two numbers to get the next:

$$1 + 1 = 2,\ 1 + 2 = 3,\ 2 + 3 = 5 \text{ ...etc}$$

   (c) **Step #1**

   Begin with 1, 1.

   Step #2

   Add the first two numbers to get the third term.

   Step #3

   Add the next two numbers to get the fourth term.

   Step #4

   Add the most recent two numbers in the sequence to get the next term.

   (d) You're algorithm will work in that you'll get some sequence (as shown below):

$$1, 3, 4, 7, 11, 18 \text{ ... etc}$$

   But, this is not the Fibonacci sequence. So if you started your algorithm with 1 and 3 instead of 1 and 1, you won't get the Fibonacci sequence.

5. (a) In the six blanks, write the numbers 1, 5, 10, 10, 5, 1.

   The pattern is to add the two numbers directly above each position, with 1 on the outsides of the triangle. So for example, the 4 in Row 5 is obtained by adding together the 1 and the 3 in the row above it.

   (b) **Step #1**

   Copy and paste the previous iteration of Pascal's Triangle.

   Step #2

   Put 1s one the outsides of the row. For the remaining numbers, add the two numbers directly above each position.

   (c) The answer to this question is the same as above.

6. The final state is obtained by applying all 4 rules exactly 4 times to every cell on the grid. The final state is shown in Figure 4 below:

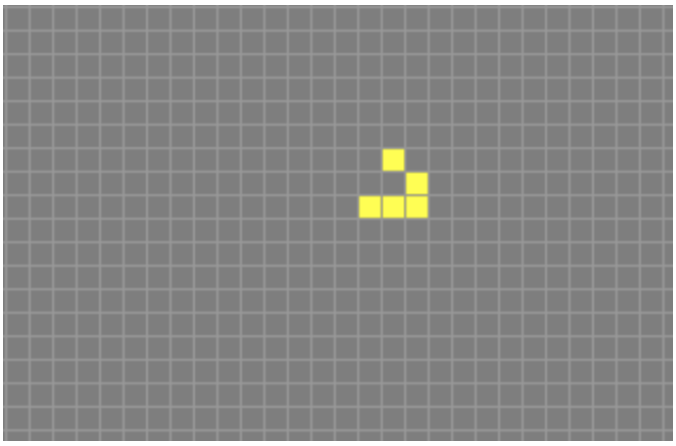

Figure 4: The final state after 4 generations

Now although the final state looks similar to the initial state, if you go through applying the rules step by step, you'll notice that the entire Glider moves diagonally down and to the right. This movement is why it's called a Glider.

7. The Toad, is an example of an oscillator. After 3 generations, the final state looks like this:

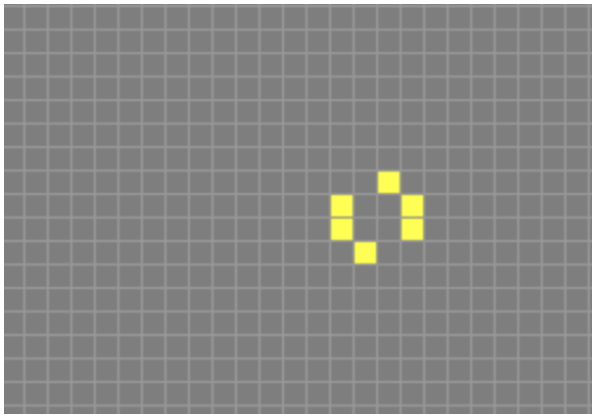

Figure 5: The final state after 3 generations

The Toad is an oscillator because it only has 2 states (the one given in the problem and the one shown in Figure 5).

8. The initial state wouldn't be an oscillator if Rule 1 were to be deleted. This is because the two live cells surrounding the *middle* live cell die in the next generation due to rule 1. If rule 1 were to be deleted, they wouldn't die and the next state wouldn't be repeating.

   To see this, start with the initial state (as given in Example 4) and apply rules 2-4 (excluding Rule 1). You'll quickly see that the number of live cells increases in the next generation and cells don't return to the initial state after 2 generations.