1 Introduction

This project involves the implementation of the sorting algorithms Bubblesort, Quicksort and Heapsort in C++ with reference to some partial order. The algorithms will be used on variables of type `MVector`, a class that stores a vector of doubles. With the power of abstraction and class inheritance, the algorithms will also work on variables of type `CoordinateArray`, a class that stores a vector of ordered unsigned integer pairs.

Sorting is widely used today, most notably in the fields of commercial computing (sorting items on a shopping website) and the searching problem (e.g. binary search requires a sorted list) [7]. We will use the sorting algorithms to simulate John Conway's the Game of Life in C++, where sorting and searching will help us to quickly determine the state of the live cells for a generation.

2 Classes

Remark. All classes are included in the appendix due to the size of the code listings. Members of a class will be referenced with the code listing and line range if they are required.

The first class that was implemented for this project was the `MVector` class (Listing 10), that stores a vector of doubles. Various member functions are also implemented so we can use the vector in other calculations, but perhaps the most important member functions included are `size`, a function which returns the size of a vector; `swap`, a function which swaps the positions of 2 elements in a vector; and `cmp`, a comparison operator (defining a partial order) for 2 elements in a vector. These are used to construct the functions which implement the 3 sorting algorithms mentioned in the introduction, and so a question we might ask ourselves is "can we use these sorting algorithms on more types of objects?".

This is precisely what the class `SortableContainer` (Listing 9) allows us to do. `SortableContainer` is an abstract base class that contains pure virtual functions for `size`, `swap` and `cmp`. Hence any class we define that contains an object we wish to sort must be a derived class of `SortableContainer`, and so the class `MVector` inherits from our abstract base class.

Next we define a structure `IntegerCoordinate` (Listing 11) which stores an ordered pair of unsigned integers. This structure is used to define the class `CoordinateArray` (Listing 12), which stores a vector of type `IntegerCoordinate` and inherits from `SortableContainer`. The `size` and `swap` functions are similar to the functions of the same name in the `MVector` class, but the comparison function `cmp` compares elements of the vector lexicographically, which is defined as follows:

$$(u, v) \leq (x, y) \quad \Longleftrightarrow \quad u < x \quad \vee \quad (u = x \wedge v \leq y)$$

The `CoordinateArray` class was constructed so it can be used to implement the Game of Life.

`Life` is the class (Listing 13) that implements the Game of Life and stores 2 `CoordinateArray` vectors - one containing the positions of the live cells of the current generation and the other containing the positions of the live cells for the next generation - and an integer that stores current generation of the game from the initial configuration.

3 The sorting algorithms

We define a sequence of elements $v_0, v_1, ..., v_{n-1}$ to be sorted if $v_0 \leq v_1 \leq ... \leq v_{n-1}$, where $\leq$ is a partial order. Hence a solution to the sorting of some sequence of elements $v_0, ..., v_{n-1}$ is to find a permutation of $\{0, 1, ..., n-1\}$ σ such that $v_{\sigma(0)} \leq v_{\sigma(1)} \leq ... \leq v_{\sigma(n-1)}$. In this section we will implement 3 algorithms which will solve the sorting problem in C++ and test them to find which one is most efficient.

3.1 Bubblesort

Bubblesort is a simple algorithm that iterates over a vector and compares the elements so that for a vector $(v_0, ..., v_{n-1})$ of size n, if $v_{j+1} \leq v_j$ then we swap the positions of v_j and v_{j+1}, correcting the ordering.

```cpp
// takes a vector v and sorts it according to the bubble sort algortihm
void bubble(SortableContainer &v)
{
    for (int i = 0; i < v.size() - 1; i++)
    {
        bool sorted = true; // value for exiting loop
        for (int j = 0; j < v.size() - 1 - i; j++)
            /* assuming last i indices have been sorted, swap v_j, v_j+1 if
            v_j+1 < v_j */
            if (v.cmp(j + 1, j))
                v.swap(j, j + 1);
                sorted = false; // swap occured so not sorted
        if (sorted) break; // if no swaps occur it is sorted
    }
}
```

Listing 1: `bubble` function for implementing Bubblesort algorithm

Listing 1 shows the code that implements the bubble sort algorithm on an object of type `SortableContainer`. At any pass over an index `i`, the algorithm places the largest element in the last possible position (i.e. at the end of the subsequence $(v_0, v_1, ..., v_{n-1-i})$, assuming the last i indices have already been sorted). Also note that the algorithm is terminated if at any pass over an index `i` no swaps occur, as this means the program is already sorted.

Now we want to calculate the average time taken to sort an object of types `MVector` and `CoordinateArray` using the bubble sort algorithm for vectors of various sizes. The average times will be calculated by taking the mean of the time taken to sort 10 vectors of size n ($n \in \{10, 20, ..., 10000\}$). The sample size has been chosen to be 10 as it seems like a good trade off between attaining a more precise average while making sure the computation time isn't too high. While I feel this is sufficient, it may capture more best/worst case scenarios for smaller n, where best cases occur when the vector is already sorted, and worst cases occur when it requires the maximum number of passes [8].

```cpp
int main()
{
    std::srand(std::time(NULL));
    std::ofstream bubbletime;
    bubbletime.open("bubbletime.csv");
    bubbletime << "size,avg time (MVector),avg time (CoordinateArray)\n";
    for (int n = 1; n <= 1000; n++) /* calculate average time for vectors of size
    up to 1000 */
    {
        double avgtime_v = 0.0, avgtime_c = 0.0;
        for (int i = 1; i <= 10; i++) // calculate time for 10 vectors of size n
        {
            MVector v(n*10);
            v.initialise_random(0.0, 10000.0); // random MVector of size n
            double StartTime = Timer(); // start timer
            bubble(v); // sort
            double EndTime = Timer(); // end timer
            avgtime_v += (EndTime - StartTime); // add time taken to avgtime
        }
        avgtime_v /= 10.0; // divide avgtime by 10, thus getting the average time
        for (int i = 1; i <= 10; i++) // calculate time for 10 vectors of size n
        {
            CoordinateArray c(n*10);
            c.InitialiseRandom(0, 1000, 0, 1000); // random CoordinateArray size n
            double StartTime = Timer(); //start timer
            bubble(c); // sort
            double EndTime = Timer(); //end timer
            avgtime_c += (EndTime - StartTime); // add time taken to avgtime
        }
        avgtime_c /= 10.0; // divide avgtime by 10, thus getting the average time
        // write the size of vector and time taken to sort to a csv file
        bubbletime << n << "," << avgtime_v << "," << avgtime_c << "\n";
    }
    bubbletime.close();
    return 0;
}
```

Listing 2: Writing the average times to a csv file

Listing 2 shows the code which calculates the average time taken to sort vectors of size n and of type `MVector` and `CoordinateArray` and stores them into a csv file, which will be used to produce graphs in Matlab.

Figure 1 contains the graph of the information output by the code in Listing 2.

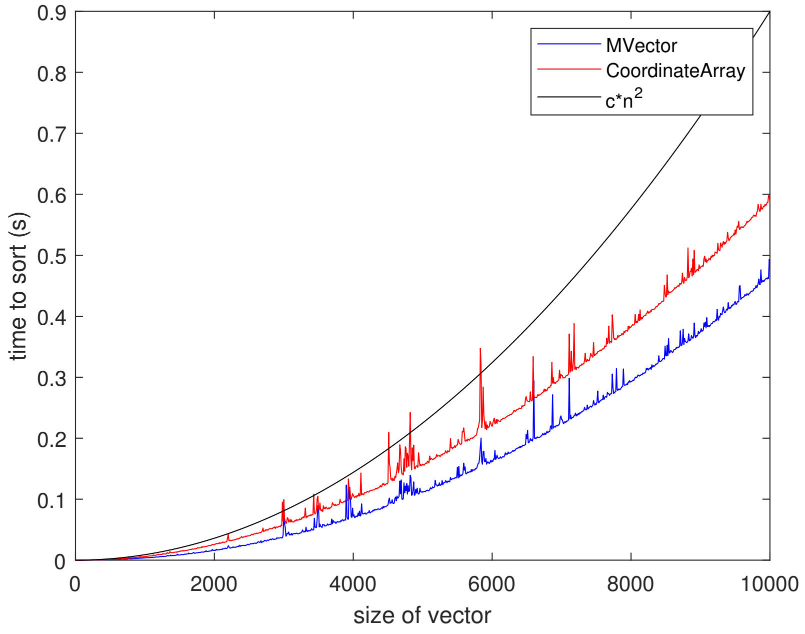

Figure 1: Time taken to sort vectors using the Bubblesort algorithm

Also included in Figure 1 is the function cn^2, where c is some constant chosen to show the time complexity of the algorithm. Recall that the time complexity of the algorithm is defined as follows: a function $f(n)$ is $\mathbf{O}(g(n))$ if there exists $n_0, c \in \mathbb{R}$ such that

$$f(n) \leq cg(n) \quad \forall n \geq n_0 \tag{1}$$

Comparing the curves regarding the sorting of vectors with the curve cn^2 we can deduce that the functional form of the curves is n^2. This agrees with the theoretical complexity of the Bubblesort algorithm which is $\mathbf{O}(n^2)$, with a best case of $\mathbf{O}(n)$ when the vector is already sorted and its worst case $\mathbf{O}(n^2)$ [5].

3.2 Quicksort

The Quicksort algorithm is a little more involved than Bubblesort but much more efficient. The algorithm involves choosing a random element of a vector, known as the pivot, and swapping positions of elements in the vector so that all elements smaller than the pivot appear before it in the vector, and all elements larger than the pivot appear after it in the vector. The pivot will now be in the correct place in the vector, and so Quicksort is then performed on the subsequences of the vector which appear either side of the pivot. Hence any function implementing the Quicksort algorithm requires a recursive call.

In fact, while choosing a random element of the vector is sufficient to implementing the Quicksort algorithm, it doesn't lead to the most efficient solution. Suppose a random number generator picks the smallest or largest element of the vector, then many swaps may have to be performed to place the element in the correct position (assuming it isn't already) and the recursive call acts on a subsequence not much smaller than the previous one, hence requiring more checks. To circumvent this, we use the 'median of three' rule [6]. This method involves choosing the first and last elements of the vector (or subsequence of a vector), and the middle element of the vector (i.e. the $\lfloor\frac{n-1}{2}\rfloor^{th}$ element of a vector $(v_0, ..., v_{n-1})$), arranging them so they are in order, and placing the median element at the beginning of the vector. Using this rule avoids all worst case scenarios and usually cuts the partitions the vector into subsequences of roughly equal size.

```
void MedOfThree(SortableContainer &v, int start, int end)
/* takes a vector v and takes 2 indices start and end, calculates the midpoint
index mid, and sorts the values so median of the three values is at the start */
{
    int mid = start + (end - start) / 2;
    if (v.cmp(end, start)) v.swap(end, start);
    if (v.cmp(mid, start)) v.swap(mid, start);
    if (v.cmp(end, mid)) v.swap(end, mid);
    v.swap(start, mid);
}
```

Listing 3: `MedOfThree` function for implementing the 'median of three' rule

Listing 3 shows the code that implements the 'median of three' rule, and will be used to find a pivot in the function that implements the Quicksort algorithm.

As mentioned above, Quicksort requires a recursive call and any sorting step requires that at least one value be put in it's correct position in a vector, namely the pivot. Hence, we implement a function `CorrectPivotPlacement` that places the pivot chosen by the 'median of three' method in the correct position in a subsequence, places all elements with value greater than the pivot after the correct position and all elements with value smaller than the pivot before the correct position.

```
/* takes a vector v and 2 indices start and end, and places the pivot chosen by
MedOfThree in the correct position between start and end, and all elements in
subsequence greater than it above pivot and all elements less than below pivot*/
int CorrectPivotPlacement(SortableContainer &v, int start, int end)
{
    MedOfThree(v, start, end); // choose pivot and place at start
    int p_goes_here = start + 1; // index of pivot
    for (int i = start + 1; i <= end; i++)
    {
        // if v[i] < pivot, increase index of pivot by 1
        if (v.cmp(i, start))
        {
            v.swap(p_goes_here, i);
            p_goes_here++;
        }
    }
    p_goes_here--;
    v.swap(start, p_goes_here); // place pivot in correct position
    return p_goes_here; // return index of pivot
}
}
```

Listing 4: `CorrectPivotPlacement` for implementing the sorting step of Quicksort

Now we have all the functions required for sorting a vector using the Quicksort algorithm, we need a function that completely sorts the vector (i.e. computing `CorrectPivotPlacement` until the vector is completely sorted).

```
/* takes a vector and sorts it between indices start and end according to the
quicksort method*/
void quick_recursive(SortableContainer &v, int start, int end)
{
    if (start == end) { ; } // do nothing if vector is empty or 1 dim.
    else
    {
        int pivot_index = CorrectPivotPlacement(v, start, end);
        // do quicksort on either side of pivot if possible
        if (pivot_index != start) quick_recursive(v, start, pivot_index - 1);
        if (pivot_index != end) quick_recursive(v, pivot_index + 1, end);
    }
}

// wrapper for quick_recursive
void quick(SortableContainer &v) { quick_recursive(v, 0, v.size() - 1); }
```

Listing 5: `quick_recursive` and `quick` functions for implementing Quicksort

Listing 5 contains the functions that implement the Quicksort algorithm. When we say the function `quick` is a wrapper for `quick_recursive`, we mean that the the main function of `quick` is to call the subroutine `quick_recursive`. This allows us to use Quicksort easily without interfering with some of the smaller details needed for the `quick_recursive` function.

We again want to calculate the average sort times of the Quicksort algorithm for vectors of increasing sizes. The Quicksort algorithm is much quicker at sorting a vector than Bubblesort is, so to see the trends of the function we must take the sizes of a vector $(v_0, ..., v_{n-1})$ to be much larger. Hence we will use Quicksort on vectors of size n for $n \in \{1000, 2000, ..., 1000000\}$. The code which writes the average times to a csv file is functionally similar to the code in Listing 2. The only aspects of the code that are changed are the function which is called to sort the algorithm, the loops being changed to iterate over our new set of sizes n and the `fstream` variable and filenames changed to reflect the new algorithm being called. Hence the code is omitted here.

Figure 2 contains the graph of the information output by the code explained above.

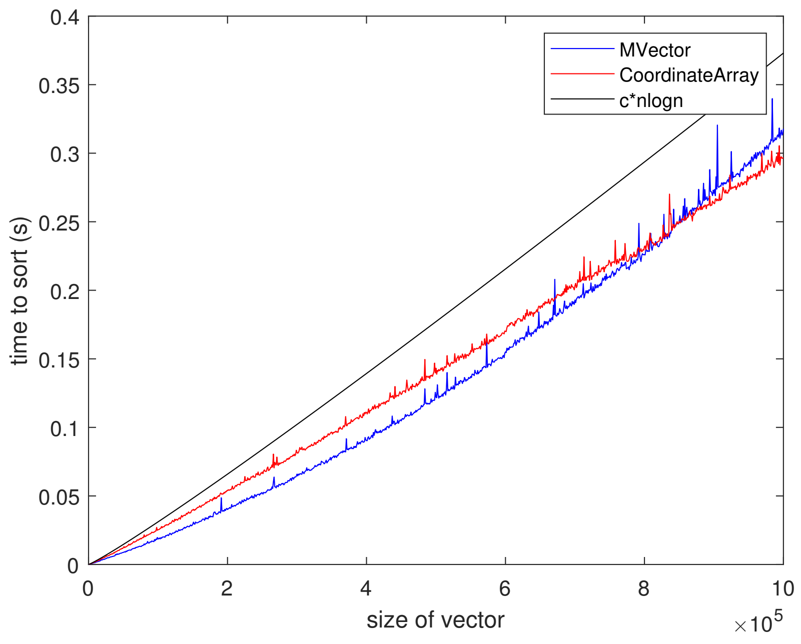

Figure 2: Time taken to sort vectors using the Quicksort algorithm

Comparing with the curve $cn \log n$ (c a constant) in Figure 2 we can deduce that the functional form of the curves is $n \log n$. Again, this agrees with the theoretical complexity of the Quicksort algorithm which is $\mathbf{O}(n \log n)$, with an equivalent best case and a worst case of $\mathbf{O}(n^2)$, which occurs when at each sorting step a maximal or minimal element is chosen as the pivot, giving only one subseqeunce to sort at the next call [5]. However, the median of three rule we implemented to choose the pivot means we always avoid the worst case scenario.

3.3 Heapsort

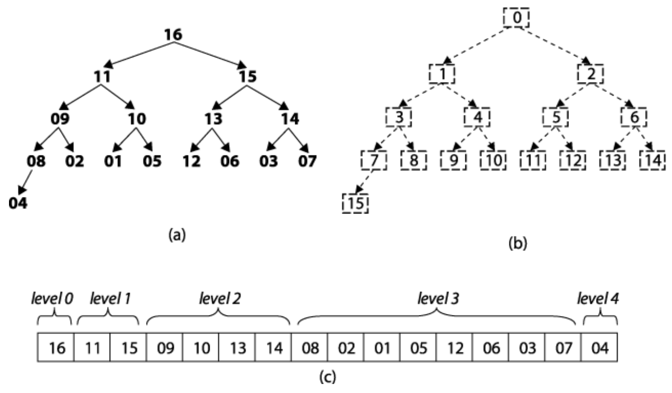

Figure 3: Visualisation of Heapsort [3]

Heapsort is a sorting algorithm which uses the idea of binary trees to sort a vector. A binary tree is a graph structure such that it is a tree in which each node has at most 2 'children' and 1 'parent'. A heap is a specialised binary tree such that the value of any node is greater than or equal to the value of its two child nodes and less than or equal to the value of it's parent node (given these nodes exist). We call the node which has no parents the root of the heap, and so any other node is a root of a smaller heap.

Firstly, we want to implement a function which can create a heap structure. The idea is to compare three elements of a vector and swap them so that the largest of the three elements appears before the two others, but how do we choose the indices of the three elements? For some chosen index $i \in \{0, 1, ..., n-1\}$ of a vector of size n, choose it's children to be at $2i + 1$ and $2i + 2$. Figure 3 shows this idea to be consistent, where (a) is the visualisation of the heap, (b) shows the vector indices of the heap and (c) shows how they appear in a vector form.

```
/*takes a SortableContainer v, it's end index end and some index i, and performs
a heap sort on the 3 indices i, 2*i+1 and 2*i+2, and calls heap_from_root again
if the largest node wasn't in the correct place*/
void heap_from_root(SortableContainer &v, int i, int end)
{
    // initialise indices of the parent node and it's 2 child nodes
    int parent = i, child1 = 2 * i + 1, child2 = 2 * i + 2;
    // find the largest value among the parent and children if possible
    if (child1 < end && v.cmp(parent, child1)) { parent = child1; }
    if (child2 < end && v.cmp(parent, child2)) { parent = child2; }
    /* if the largest value wasn't in the correct place, place it at the correct
    index i and perform heap sort where the largest value was */
    if (parent != i)
    {
        v.swap(parent, i);
        heap_from_root(v, parent, end);
    }
}
```

Listing 6: `heap_from_root` function that creates a heap structure

Listing 6 contains the function `heap_from_root` which creates a heap structure from a vector v with its root at index i of the vector. The function calls itself recursively if the parent wasn't in the correct position, on the position where the parent was initially. This is done to help us create a maximal heap, i.e. a heap structure of the whole vector. The idea is to create a maximal heap from the vector, and then extract the roots one by one and place them in their correct position at the end of the index. Calling `heap_from_root` again while doing this process ensures we always choose the largest element that hasn't already been sorted.

```
// takes a SortableContainer v and sorts it according to the Heapsort method
void heap(SortableContainer &v)
{
    int end = v.size()-1;
    // build a maximal heap (i.e. biggest possible binary tree)
    for (int i = end / 2; i >= 0; i--)
    {
        heap_from_root(v, i, end);
    }
    /* largest value is at the root of heap, so place it at the ith index and call
    heap_from_root */
    for (int i = end; i >= 0; i--)
    {
        v.swap(0, i);
        heap_from_root(v, 0, i);
    }
}
```

Listing 7: `heap` function implementing the Heapsort algorithm

Listing 7 contains the function `heap` that implements the Heapsort algorithm. Note that while `quick` was a wrapper for `quick_recursive` in Listing 5, `heap` is not a wrapper for `heap_from_root` as it does more than just call the subroutine.

To calculate the average sort times of the Heapsort algorithm on vectors of various sizes, we use the same set $\{1000, 2000, ..., 10000000\}$ as used in the Quicksort situation and change the code in Listing 2 in the same way - we instead call `heap` to sort the vectors, and `fstream` variable and filenames now reflect the fact we are using the Heapsort algorithm. Hence the code which prints the average times to a csv file is omitted.

Figure 4 contains the graph of the information output by the code explained above.

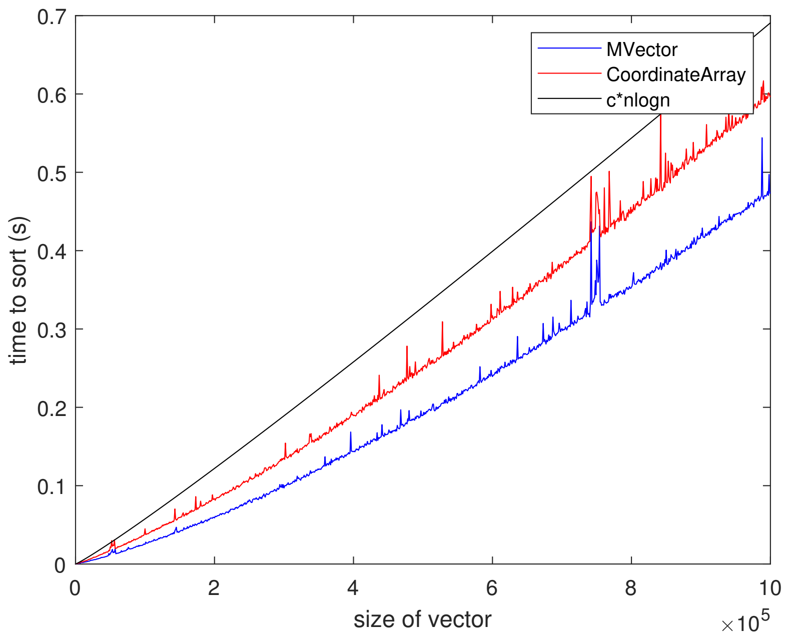

Figure 4: Time taken to sort vectors using the Heapsort algortihm

As we have done previously, by comparing with the curve $cn \log n$ we can deduce that the functional form of the curve is $n \log n$. This agrees with the theoretical complexity of the Heapsort algorithm, which is $\mathbf{O}(n \log n)$ [5]. It's best and worst case scenarios also run in $n \log n$ time, as the construction of the maximal heap and rearranging of elements always occur in any use of Heapsort.

3.4 The best sorting algorithm

The Bubblesort algorithm, while simple and having the fastest best case scenario, is too slow to sort when we don't know, ahead of time, the state of the container we are wishing to sort. It also takes ≈ 0.6 seconds to sort a vector of size 10^4, whereas the Heapsort and Quicksort algorithms sort vectors of size 10^6 in this time, meaning Bubblesort should not be used in almost all cases.

In my opinion, the best algorithm to use is Quicksort. While this may be obvious from the graphs in Figures 2 and 4 (a vector of size 10^6 can be sorted in ≈ 0.6 seconds using Heapsort and ≈ 0.3 seconds using Quicksort), the worst case scenarios that can happen using Quicksort mean we may want to consider Heapsort if we want to guarantee that the vector be sorted in some amount of time. However, the use of the median of three rule in our implementation of Quicksort means we always avoid the worst case scenario and usually choose a pivot such that the two subsequences on either side of it are of roughly equal size, giving close to optimal conditions for the Quicksort algorithm [4].

4 The Game of Life

The Game of Life was created by John Conway in 1970 and is a cellular automaton - a grid of some specified shape containing a collection of coloured cells that evolves according to a set of given rules [9]. Life is not a game in the traditional sense as the only input that is required is the initial configuration of live cells, hence it is considered a zero player game. Conway devised a set of rules the initial configurations should follow and a set of laws for birth, death and survival that the cells should obey [2]:

- **I** There should not be initial configurations that can be shown to grow without limit by a simple proof. However, initial configurations that *apparently* grow without limit are allowed.
- **II** There should exist simple initial configurations that grow and change for significant periods of time before ending by either ceasing to exist, entering a static configuration (no more growth or change) or entering an oscillating pattern of configurations.
- **III** A dead cell becomes alive (i.e. is born) if 3 of its 8 neighbouring cells are alive, otherwise it remains dead.
- **IV** A live cell remains alive (i.e. survives) if 2 or 3 of its 8 neighbouring cells are alive, otherwise it dies.

While **I** is not exactly required to implement Life on a computer, following it will mean we won't run into situations where the computer would eventually have insufficient memory. The rules of the initial configurations (**I**, **II**) tell us what we should expect from any 'correct' implementation of Life (i.e. an initial configuration which satisfies **I**, **II**). **III** and **IV** are where our attentions really lie, as they will allow us to implement the Game of Life in C++.

4.1 Implementing Life in C++

Remark. Unless stated otherwise, all the following functions are members of the `Life` class, and so any reference to them will include only the line ranges. Recall that the `Life` class is included in Listing 13.

To begin, Listing 8 shows the code that evolves the initial configuration

$$\{(1,1),(1,2),(2,1),(2,2)\} \tag{2}$$

according to the rules of Life. Then an explanation will be given as to what each of line of code does.

```
int main()
{
    Life Game1(4); // initalise Life with a LiveCells of size 4
    // initial configuration coordinates
    IntegerCoordinate a1(1, 1);
    IntegerCoordinate b1(1, 2);
    IntegerCoordinate c1(2, 1);
    IntegerCoordinate d1(2, 2);
    // add initial configuration to LiveCells
    Game1.LiveCells[0] = a1;
    Game1.LiveCells[1] = b1;
    Game1.LiveCells[2] = c1;
    Game1.LiveCells[3] = d1;
    // evolution of game
    while (Game1.gen <= 5206)
    {
        quick(Game1.LiveCells); // sort LiveCells
        std::cout << Game1.LiveCells << "\n"; // print current configuration
        // vector of dead cells that may become alive at next generation
        CoordinateArray PossLiveCells = Game1.DeadCells();
        quick(PossLiveCells);
        PossLiveCells.unique(); // remove repeat elements from PossLiveCells
        // add dead cells that become alive to LiveCellsNext
        Game1.born(PossLiveCells);
        Game1.die(); // add live cells that stay alive to LiveCellsNext
        Game1.tick(); // advance to next generation of game
    }
    return 0;
}
```

Listing 8: Code evolving the initial configuration (2)

Firstly, the `Life` constructor (11-17) is called which creates a vector `LiveCellsNext` of size 4 in this case, and also initalises an empty vector `LiveCellsNext` to store the live cells for the next generation, and an integer `gen` which stores the current generation of the configuration. Then the coordinates for the initial configuration (2) are initialised using the constructor for the structure `InitialCoordinate`, and then these are placed in `LiveCells`.

Now we move onto the evolution of the game. Firstly, we must sort `LiveCells` as the functions which determine the state of the cells for the next generation rely on using the `CoordinateArray` member function `find` (Listing 12, 42-54) which implements a binary search algorithm. The function `DeadCells` (42-57) returns a vector `PossLiveCells` of all dead cells that may become alive at the next generation, and to do this it uses the function `neighbours` which returns a vector of the 8 neighbours $\{(x+i, y+j) | i, j \in \{-1, 0, 1\}, i \neq j \neq 0\}$ of some live cell (x, y). Next, we must sort the vector `PossLiveCells` so we can use the `CoordinateArray` member function `unique` (Listing 12, 56-64) to remove all repeats from a sorted vector. The function `born` (60-75) adds the dead cells in `PossLiveCells` with exactly 3 neighbours in `LiveCells` to the vector `LiveCellsNext`, whereas `die` (77-94) adds the live cells with 2 or 3 neighbours to `LiveCellsNext`. Then the function `tick` (97-105) sets the configuration for the next generation - it replaces `LiveCells` with `LiveCellsNext` and resets `LiveCellsNext` to be empty, while also increasing `gen` by 1.

4.2 Block and Blinker

We now consider the initial configurations (2) and

$$\{(5, 4), (5, 5), (5, 6)\} \tag{3}$$

We will use the code above to implement the game, with the only changes being the initial configurations, hence the code is omitted.

In fact, the following output showing the first few generations of the game tell us everything we need to know about how they evolve for all time.

```
Game1:                                  Game2:
(1,1),(1,2),(2,1),(2,2),                (5,4),(5,5),(5,6),
(1,1),(1,2),(2,1),(2,2),                (4,5),(5,5),(6,5),
(1,1),(1,2),(2,1),(2,2),                (5,4),(5,5),(5,6),
(1,1),(1,2),(2,1),(2,2),                (4,5),(5,5),(6,5),
(1,1),(1,2),(2,1),(2,2),                (5,4),(5,5),(5,6),
(1,1),(1,2),(2,1),(2,2),                (4,5),(5,5),(6,5),
```

The first initial configuration is known as a 'Block', a static configuration in Life, meaning it does not evolve with time, whereas the second initial configuration is known as a 'Blinker', a oscillating configuration of periodicity 2 that flips between horizontal and vertical configurations of itself [1]. It follows that both (2) and (3) seem to obey Conway's law **II** of initial configurations of Life.

4.3 Acorn

We now consider the initial configuration 'the acorn'

$$\begin{aligned}&\{(10001, 10001), (10002, 10001), (10002, 10003),\\&(10004, 10002), (10005, 10001), (10006, 10001), (10007, 10001)\}\end{aligned} \tag{4}$$

It turns out this configuration expands into a pretty considerable size - there are 633 live cells after 5206 generations from the initial configuration. This is where the importance of the sorting algorithms comes in, as we would have needed considerably more time to compute the configuration after 5206 generations (while the size doesn't seem large compared to some of the sizes of vectors we sorted in section 2, consider the fact that we also have to sort the vector of currently dead but possibly live cells and perform searches for all the neighbours of these cells and the live cells). Figures 5 and 6 show the state of the acorn after 5206 generations, included on the next page.

5 Conclusion

Throughout this report we have implemented the sorting algorithms Bubblesort, Quicksort and Heapsort in C++ and abstracted them so they work with all classes that are derived from the abstract base class `SortableContainer` which contains the pure virtual functions that are required for the sorting problem. We also deduced that the Quicksort algorithm is the best choice for efficiently sorting a container, while one may choose to use Heapsort if they require that the sorting be completed in some given amount of time. We then used the sorting algorithms to implement the Game of Life, where Quicksort became particularly useful in determining the evolution of the acorn, an explosive initial configuration compared to that of the block or the blinker.

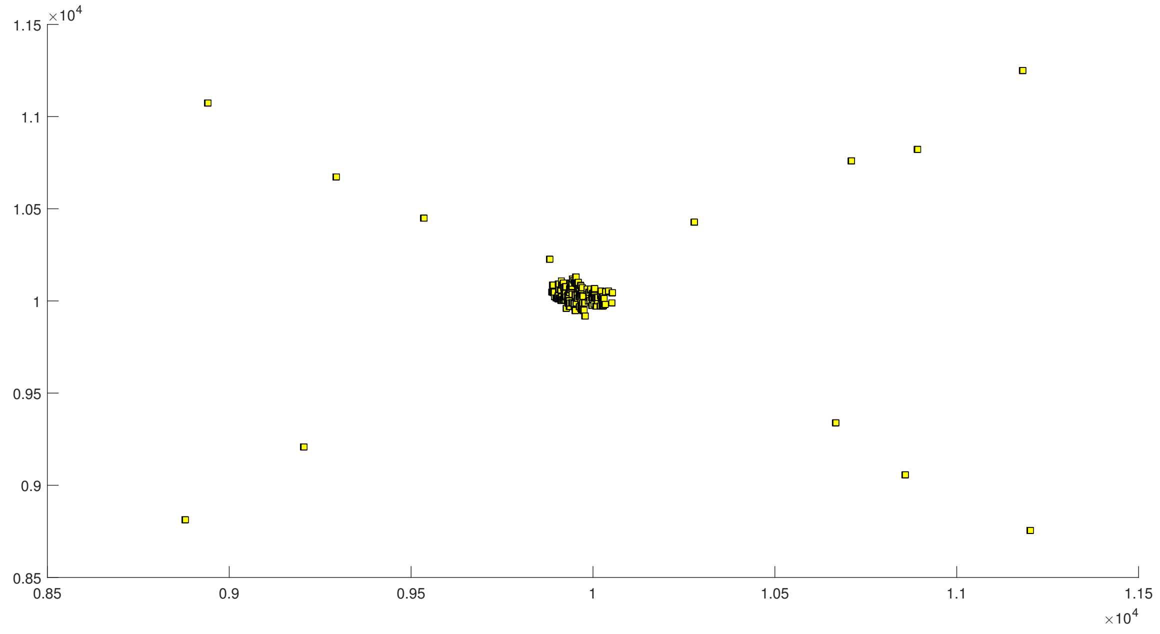

Figure 5: The state of the acorn after 5206 generations

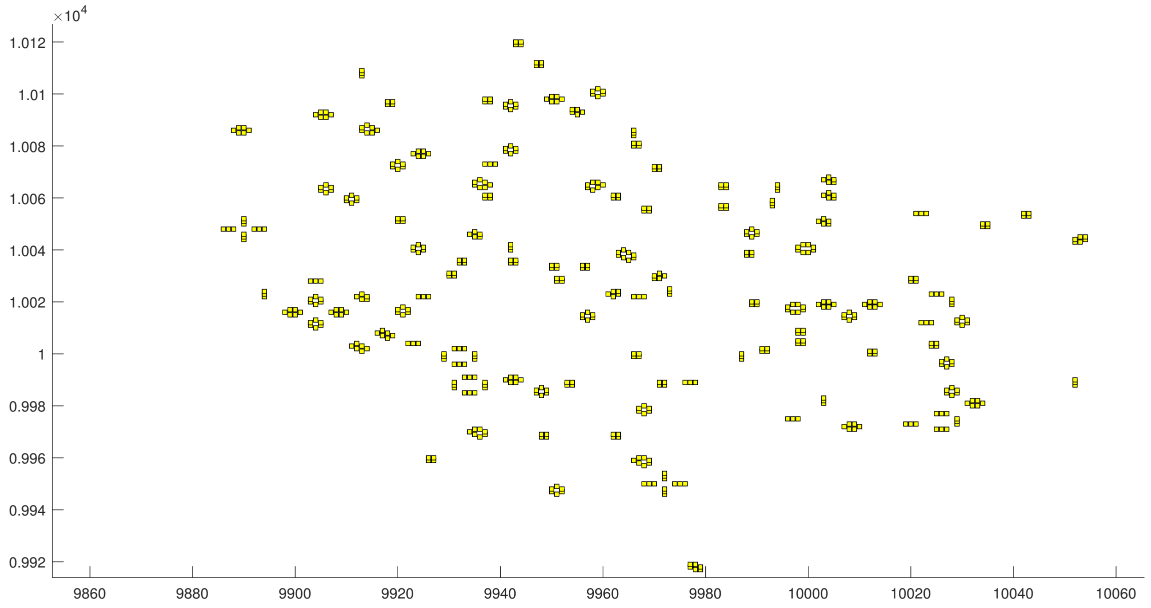

Figure 6: A zoom in on the large cluster of the acorn after 5206 generations

6 Appendix

```
// Abstract Base Class for Sorting
class SortableContainer
{
public:
    // pure virtual comparison function
    virtual bool cmp(int i, int j) const = 0;
    // pure virtual size function
    virtual int size() const = 0;
    // pure virtual "swap 2 elements in vector" function
    virtual void swap(int i, int j) = 0;
};
```

Listing 9: `SortableContainer` abstract base class

```
// Represents a mathematical vector
class MVector : public SortableContainer
{
public:
    /* constructors */
    MVector() : v(0)
    {}

    // initialise empty vector of size n
    explicit MVector(int n) : v(n)
    {}

    // initialise vector of size n with values x
    MVector(int n, double x) : v(n, x)
    {}


    /* SortableContainer functions */

    // returns true if v[i] < v[j], false otherwise
    virtual bool cmp(int i, int j) const
    {
        if (v[i] < v[j]) return true;
        else return false;
    }

    // returns no. of elements in vector
    virtual int size() const { return v.size(); }

    // swaps the places of the ith and jth elements
    virtual void swap(int i, int j)
    {
        int vtemp = v[i];
        v[i] = v[j]; v[j] = vtemp;
    }

    /* initialise random vector */

    // for an n-vector, give each place a random value in the range [xmin,xmax]
    void initialise_random(double xmin, double xmax)
    {
        size_t s = v.size();
        for (size_t i = 0; i < s; i++)
        {
            v[i] = xmin + (xmax - xmin)*rand() / static_cast<double>(RAND_MAX);
        }
    }

    /* overloaded operator functions */
    // access element (lvalue)
    double &operator[](int index)
    {
        /*if (index < 0 || index >= v.size()) // checking for errors
        {
            std::cout << index << " is not in the range of [0," <<
                v.size() - 1 << "]" << std::endl;
            exit(1);
        }
        else */
        return v[index];
    }

    // access element (rvalue)
    double operator[](int index) const
    {
        /*if (index < 0 || index >= v.size()) // checking for errors
        {
            std::cout << index << " is not in the range of [0," <<
                v.size() - 1 << "]" << std::endl;
            exit(1);
        }
        else */
        return v[index];
    }

    // print n-vector in the format v0,v1,....,vn,
    friend std::ostream& operator<<(std::ostream& os, const MVector& v);

private:
    std::vector<double> v;
};
// print MVector in the format v0,v1,....,vn,
std::ostream& operator<<(std::ostream& os, const MVector& v)
{
    for (int i = 0; i < v.size(); i++) { os << v[i] << ","; }
    return os;
}
```

Listing 10: MVector class

```
// Represents an ordered pair of integers
struct IntegerCoordinate
{
    // constructors
    IntegerCoordinate() { unsigned X = 0, Y = 0; }
    IntegerCoordinate(unsigned X_, unsigned Y_) { X = X_, Y = Y_; }
    unsigned X, Y;

    // overloaded operators

    // return true if 2 IntegerCoordinates are equal
    bool operator==(IntegerCoordinate xy)
    {
        if (X == xy.X && Y == xy.Y) return true;
        else return false;
    }

    //  return true if LHS IntegerCoordinate is less than RHS
    bool operator<(IntegerCoordinate xy)
    {
        if (X < xy.X) return true;
```

```cpp
        else if (X == xy.X && Y < xy.Y) return true;
        else return false;
    }
};
```

Listing 11: `IntegerCoordinate` structure

```cpp
// Represents a vector of ordered pairs of integers
class CoordinateArray : public SortableContainer
{
public:
    // constructors

    // initialise empty vector of IntegerCoordinate
    CoordinateArray() : v(0)
    {}

    // initialise vector of size n with each index (1,1)
    explicit CoordinateArray(int n) : v(n, IntegerCoordinate(1, 1))
    {}

    // lexicographic ordering for a vector of IntegerCoordinate
    virtual bool cmp(int i, int j) const
    {
        if (v[i].X < v[j].X) return true;
        else if (v[i].X == v[j].X && v[i].Y < v[j].Y) return true;
        else return false;
    }

    // return no. of IntegerCoordinate in vector
    virtual int size() const
    {
        return v.size();
    }

    // swap two IntegerCoordinate in vector
    virtual void swap(int i, int j)
    {
        IntegerCoordinate vtemp = v[i];
        v[i] = v[j], v[j] = vtemp;
    }

    // append IntegerCoordinate to end of vector
    void push_back(IntegerCoordinate xy)
    {
        v.push_back(xy);
    }

    // binary search for an IntegerCoordinate in vector
    bool find(IntegerCoordinate xy)
    {
        int L = 0, R = v.size() - 1; // start and end indices
        while (L <= R)
        {
            int M = L + (R - L) / 2; // midpoint index
            if (v[M] == xy) return true; // if midpoint index is xy return true
            else if (xy < v[M]) R = M - 1; // if xy < midpoint search bottom half
            else L = M + 1; // if xy > midpoint search top half
        }
        return false; // return false if xy not found
    }

    // removes repeat elements from a sorted vector of IntegerCoordinate
    void unique()
    {
        for (int i = 0; i < v.size() - 1; i++)
        {
            while (v[i] == v[i + 1])
                v.erase(v.begin() + (i+1)); // remove all values equal to v[i]
        }
    }

    // each X coord a random integer in [X_min,X_max], Y coord in [Y_min,Y_max]
    void InitialiseRandom(unsigned X_min, unsigned X_max, unsigned Y_min,
        unsigned Y_max)
    {
        size_t s = v.size();
        for (int i = 0; i < s; i++)
        {
            v[i].X = rand() % (X_max - X_min) + X_min;
            v[i].Y = rand() % (Y_max - Y_min) + Y_min;
        }
    }

    // access element (lvalue)
    IntegerCoordinate &operator[](int index)
    {
        /*if (index < 0 || index >= v.size()) // checking for errors
        {
            std::cout << index << " is not in the range of [0," <<
                v.size() - 1 << "]" << std::endl;
            exit(1);
        }
        else*/
        return v[index];
    }

    // access element (rvalue)
    IntegerCoordinate operator[](int index) const
    {
        /*if (index < 0 || index >= v.size()) // checking for errors
        {
            std::cout << index << " is not in the range of [0," <<
                v.size() - 1 << "]" << std::endl;
            exit(1);
        }
        else*/
        return v[index];
    }

    // print n-vector in the format v0,v1,....,vn,
    friend std::ostream& operator<<(std::ostream& os, const CoordinateArray& v);

private:
    std::vector<IntegerCoordinate> v;
};
```

Listing 12: `CoordinateArray` class

```
// Represents a vector of live cells for the game of Life
class Life : public CoordinateArray
{
public:
    // Storage of the coordinates of the live cells
    CoordinateArray LiveCells;
    // generation of the game
    int gen;

    // Constructor
    Life(int n)
    {
        CoordinateArray LC(n);
        LiveCells = LC;
        CoordinateArray LiveCellsNext;
        int gen = 0;
    }

    // returns CoordinateArray of the 8 neighbours of an IntegerCoordinate
    CoordinateArray neighbours(CoordinateArray Cells, int i)
    {
        CoordinateArray v;
        // if out of index range return empty vector
        if (i < 0 || i >= Cells.size()) return v;
        else
        {
            CoordinateArray v(8);
            IntegerCoordinate BL(Cells[i].X - 1, Cells[i].Y - 1);
            IntegerCoordinate CL(Cells[i].X - 1, Cells[i].Y);
            IntegerCoordinate TL(Cells[i].X - 1, Cells[i].Y + 1);
            IntegerCoordinate BC(Cells[i].X, Cells[i].Y - 1);
            IntegerCoordinate TC(Cells[i].X, Cells[i].Y + 1);
            IntegerCoordinate BR(Cells[i].X + 1, Cells[i].Y - 1);
            IntegerCoordinate CR(Cells[i].X + 1, Cells[i].Y);
            IntegerCoordinate TR(Cells[i].X + 1, Cells[i].Y + 1);
            v[0] = BL; v[1] = CL; v[2] = TL; v[3] = BC;
            v[4] = TC; v[5] = BR; v[6] = CR; v[7] = TR;
            return v;
        }
    }

    // Returns the dead cells that may be alive at next generation (with repeats)
    CoordinateArray DeadCells()
    {
        CoordinateArray dc;
        for (int i = 0; i < LiveCells.size(); i++)
        {
            /* for each live cells neighbours, if it is a dead cell, append it to
            the CoordinateArray dc */
            CoordinateArray nbrs = neighbours(LiveCells, i);
            for (int j = 0; j < nbrs.size(); j++)
            {
                if (!LiveCells.find(nbrs[j])) dc.push_back(nbrs[j]);
            }
        }
        return dc;
    }

    // find the dead cells that become alive and add to LiveCellsNext
    void born(CoordinateArray PossLiveCells)
    {
        for (int i = 0; i < PossLiveCells.size(); i++)
        {
            int count = 0; // count for live neighbours
            CoordinateArray nbrs = neighbours(PossLiveCells, i);
            // if live neighbour found for a dead cell, add 1 to count
            for (int j = 0; j < 8; j++)
            {
                if (LiveCells.find(nbrs[j])) count++;
            }
            /* if PossLiveCells[i] has 3 live neighbours, append PossLiveCells[i]
            to LiveCellsNext */
            if (count == 3) LiveCellsNext.push_back(PossLiveCells[i]);
```

```cpp
        }
    }

    // Find the live cells that stay alive and add to LiveCellsNext
    void die()
    {
        int n = LiveCells.size();
        for (int i = 0; i < n; i++)
        {
            int count = 0; // count for live neighbours
            CoordinateArray nbrs = neighbours(LiveCells, i);
            // if a live neighbour is found for LiveCells[i], add 1 to count
            for (int j = 0; j < 8; j++)
            {
                if (LiveCells.find(nbrs[j])) count++;
            }
            /* if 2 or 3 live neighbours were found, append LiveCells[i] to
            LiveCellsNext */
            if (count == 2 || count == 3) LiveCellsNext.push_back(LiveCells[i]);
        }
    }


    /* advance time one unit, replace LiveCells with LiveCellsNext and empty
    LiveCellsNext */
    void tick()
    {
        LiveCells = LiveCellsNext;
        CoordinateArray v;
        LiveCellsNext = v;
        gen += 1;
    }

private :
    // Storage of live cells for next generation
    CoordinateArray LiveCellsNext;
};
```

Listing 13: `Life` class for implementing the Game of Life

Bibliography

[1] Conwaylife.com. (n.d.) *Conway's Game of Life.* [online] Availble at: `http://www.conwaylife.com/wiki/Conway's_Game_of_Life` [Accessed 4 Nov. 2018].

[2] Gardener, M. (1970). *Mathematical Games - The fantastic combinations of John Conway's new solitaire game "life".* [online] Web.archive.org. Available at: `https://web.archive.org/web/20090603015231/http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.html` [Accessed 14 Nov. 2018].

[3] Pollice, G., Selkow, S. and Heineman, G. (n.d.). *Algorithms in a Nutshell.* [online] O'Reilly. Available at: `https://www.oreilly.com/library/view/algorithms-in-a/9780596516246/ch04s06.html` [Accessed 14 Nov. 2018].

[4] Eecs.yorku.ca. (2011). *Quick Sort (11.2).* [online] Available at: `https://www.eecs.yorku.ca/course_archive/2010-11/W/2011/Notes/s4_quick_sort.pdf` [Accessed 14 Nov. 2018].

[5] Rowell, E. (n.d.). *Big-O Cheat Sheet.* [online] Bigocheatsheet.com. Available at: `http://bigocheatsheet.com/` [Accessed 15 Nov. 2018].

[6] Sedgewick, R. (1997). *Algorithms in C++.* 3rd ed. Boston: Addison-Wesley, pp.319 - 323.

[7] Sedgewick, R. and Wayne, K. (2011). *Algorithms.* 4th ed. Boston: Addison Wesley.

[8] Tran, T. (n.d.). *Bubble Sort!.* [online] Cs.ucsb.edu. Available at: `https://www.cs.ucsb.edu/~bboe/cs32_m12/slides/sort/bubble_01.pdf` [Accessed 12 Nov. 2018].

[9] Weisstein, E. (n.d.). *Cellular Automaton.* [online] Mathworld.wolfram.com. Available at: `http://mathworld.wolfram.com/CellularAutomaton.html` [Accessed 14 Nov. 2018].