Sorting Algorithms and the Game of Life

Richard Cotton

1 Introduction

In this report, I shall discuss and evaluate the comparable efficiencies of three different sorting algorithms. The purpose for this is to see which algorithm is preferable to use for differing lengths and types of arrays. This will allow us to investigate how increasing the dimension of the arrays affects the time in which the arrays are sorted. This generalisation is important, especially within three dimensional problems which can be used when modelling environmental and physical problems, for example on Earth.

We shall begin with the bubblesort algorithm before investigating the quicksort and heapsort algorithms by looking at their respective sorting speeds. We shall then abstract the sorting algorithms to two dimensions, sorting tuples of integers using lexicographical ordering. Following this, we shall implement a simulation called 'Conway's Game of Life' which, from an initial condition of cells and a set of rules regarding neighbouring cells, evolves over time creating interesting patterns and interactions between the living cells. This again has important applications within the real world, mainly in areas such a small scale biology and physics problems, as well as evolution problems within computer science such as viral infection and malware problems. [2]

2 Sorting Algorithms

2.1 Bubblesort

The bubblesort is one of the simplest sorting algorithms which can be implemented within C++. We take the input of an unsorted array of length n with numbers of type double. Beginning with the first element, we compare this with the second element. If the value of the first element is larger than the second, we swap the two. We then compare the second and third elements, swapping if necessary. This continues until we have compared all adjacent pairs of elements once.

After this has been done, the largest element of the array will be located in nth position with the rest of the $(n-1)$ elements still to be sorted. We then repeat the process with the remaining elements until the array is fully sorted.

A C++ implementation of the bubblesort can be seen in appendix A.1. We run this code starting with a randomly generated unsorted array of length 100 and find the time it takes for the bubblesort algorithm to fully sort the array. We continue in this way by generating lists of length 200, 300, 400 etc. up to a maximum array size of 5000 elements and timing how long it takes to sort them. At each array length size, we run 100 instances of the bubblesort and take an average. This is done to ensure a randomly generated array is not created in a state of 'close completion' which would give a faster than expected sorting time. These results are saved in a file and plotted using Octave. The output of this can be seen in Figure 1.

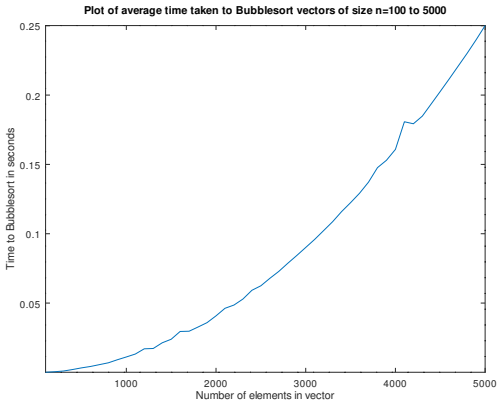

Figure 1: Plot of bubblesort for varying vector sizes n

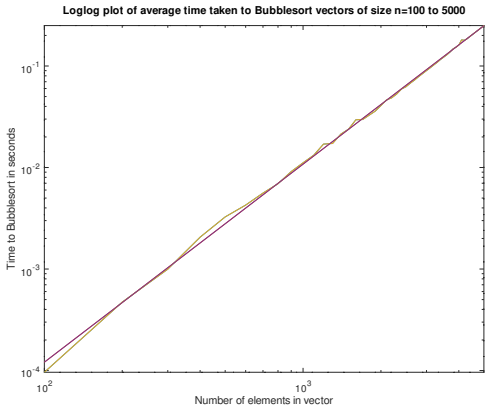

Figure 2: Loglog plot of bubblesort for varying vector sizes n

We also produce a loglog plot of Figure 1 to determine the form of the bubblesort algorithm. In doing this, we can plot a line of best fit and determine the slope of the loglog plot as is shown in Figure 2. In this case, we find the line to be of gradient 2 and thus, determine that the form of the algorithm complexity to be $\mathcal{O}(n^2)$.

We can look at the time complexity of the implemented algorithm as we would expect the theoretical complexity of the algorithm to be the same as the computed one from our simulations. Again looking at the bubblesort code in appendix A.1, we can see that the algorithm revolves around two nested for loops. There are $(n-1)$ comparisons on the first pass through the array, $(n-2)$ on the second and so on until there is only one comparison between the remaining two elements. Therefore, the algorithm has a time complexity of $\mathcal{O}(n^2)$. This matches with our inspection from Figure 2.

2.2 Quicksort

The next sorting algorithm of interest is the quicksort. This is a divide and conquer algorithm, so called as it breaks down the problem into smaller and smaller problems, works on them individually and then combines the results to form the final solution. This type of algorithm can be defined recursively allowing for much faster sorting in general.

We begin by choosing a random element as the pivot for the rest of the elements to be compared to. We then move all elements smaller than the pivot to the left of the pivot and all elements larger than the pivot to the right. After this is done, we consider the two sublists either side of the pivot and sort them using the quicksort algorithm. This is done until all sublists are completely sorted and they are then combined to produce the final solution.

A C++ implementation of the quicksort algorithm can be found in appendix A.2. We again run this at intervals of 100 elements up to 5000, with each interval having 100 random arrays created and sorted using the quicksort algorithm. The times are recorded and averaged, resulting in plot seen in Figure 3.

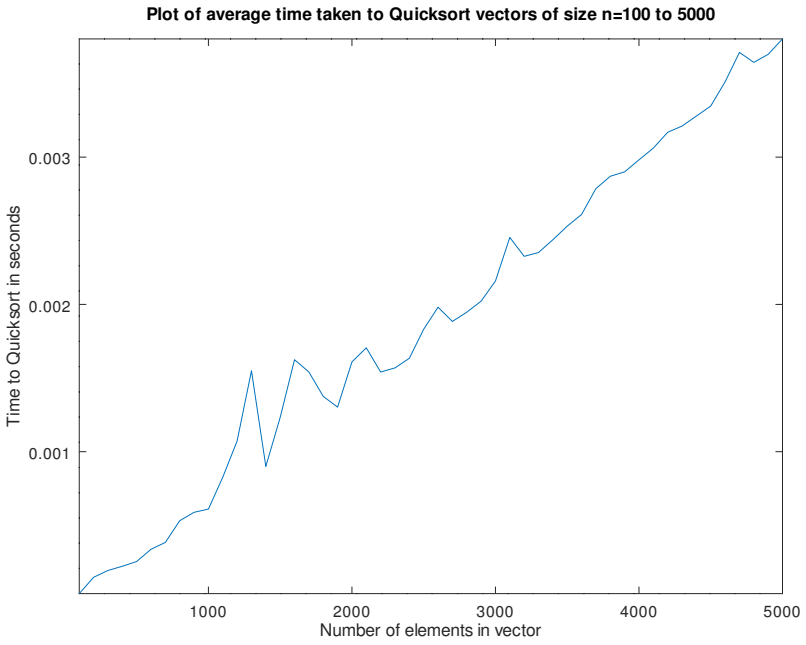

Figure 3: Plot of quicksort for varying vector lengths n

Upon visual inspection of the graph, we can see that the graph appears to have a much more linear slope compared to that of Figure 1. Thus, we may expect a function close to the form of n, compared to that of an exponential curve as seen from the bubblesort algorithm.

There are many sources which prove that the quicksort algorithm has complexity of $\mathcal{O}(n\log(n))$, for example Shaffer [1]. We outline the proof as follows.

Each time the array of length n is partitioned, there is an equally likely chance that the pivot lies in any position within the sorted array, thus partitioning the array into two subarrays of lengths $(0, n-1), (1, n-2)$ etc. Thus the average time taken satisfies the recurrence relation

$$T(n) = cn + \frac{1}{n}\sum_{k=0}^{n-1}[T(k) + T(n-1-k)], \quad T(0) = T(1) = c, \quad c \in \mathbb{R}.$$

We can see that the terms within the summation produce the same values as we step through the terms. Thus the relation can be rewritten as

$$T(n) = cn + \frac{2}{n}\sum_{k=0}^{n-1} T(k), \quad T(0) = T(1) = c.$$

We can now solve this recurrence relation by canceling out the summation terms. In doing this,

$$nT(n) = cn^2 + 2\sum_{k=1}^{n-1} T(k) \tag{1}$$

$$(n+1)T(n+1) = c(n+1)^2 + 2\sum_{k=1}^{n} T(k). \tag{2}$$

Subtracting (1) from (2),

$$\begin{aligned}(n+1)T(n+1) - nT(n) &= c(n+1)^2 - cn^2 + 2T(n)\\ (n+1)T(n+1) &= c(2n+1) + (n+2)T(n)\\ T(n+1) &= \frac{c(2n+1)}{n+1} + \frac{n+2}{n+1}T(n).\end{aligned}$$

We can now expand the recurrence relation to find,

$$\begin{aligned}T(n+1) &\leq 2c + \frac{n+2}{n+1}T(n)\\ &= 2c + \frac{n+2}{n+1}\left(2c + \frac{n+1}{n}T(n-1)\right)\\ &= 2c\left(1 + \frac{n+2}{n+1} + \frac{n+2}{n+1}\frac{n+1}{n} + \cdots + \frac{n+2}{n+1}\frac{n+1}{n}\cdots\frac{3}{2}\right)\\ &= 2c\left(1 + (n+2)\left(\frac{1}{n+1} + \frac{1}{n} + \cdots \frac{1}{2}\right)\right)\\ &= 2c + 2c(n+2)(H_{n+1} - 1)\end{aligned}$$

where H_{n+1} is the Harmonic series which is of order $\mathcal{O}(\log(n))$. Thus the overall complexity is of order $\mathcal{O}(n\log(n))$.

2.3 Heapsort

We also want to implement and investigate one final algorithm called the heapsort. We begin our analysis by defining a structure called a binary tree. This is a data structure which consists of a root node at the top of the tree, with a maximum of two child nodes emanating from each subsequent node as the tree descends. We also define a leaf to be a node which has no children. We can then define a heap, a special case of a binary tree where the value at each node is larger than or equal to both of its child nodes. The heapsort algorithm uses this heap property to its advantage when sorting arrays.

The algorithm is executed as follows. We take in a randomised, unsorted array as the input to the algorithm. We can think of this input as a binary tree where given the index of an element i, the element at index $(2i+1)$ is the left child and the element at index $(2i+2)$ is the right child. We then run the `heap_from_root` function which recursively steps through each parent node and checks whether the children nodes are larger, swapping if necessary. This creates a heap from the input of an unsorted array. We then work through elements of the heap, moving the root to the appropriate place and heapsorting this reduced heap which excludes the root. This continues until

all elements have been considered and sorted.

Once again, we implement the algorithm in C++ as found in appendix A.3, running the code at intervals of 100 up to 5000 elements with 100 random instantiations. These are then heapsorted and the average time plotted in Figure 4.

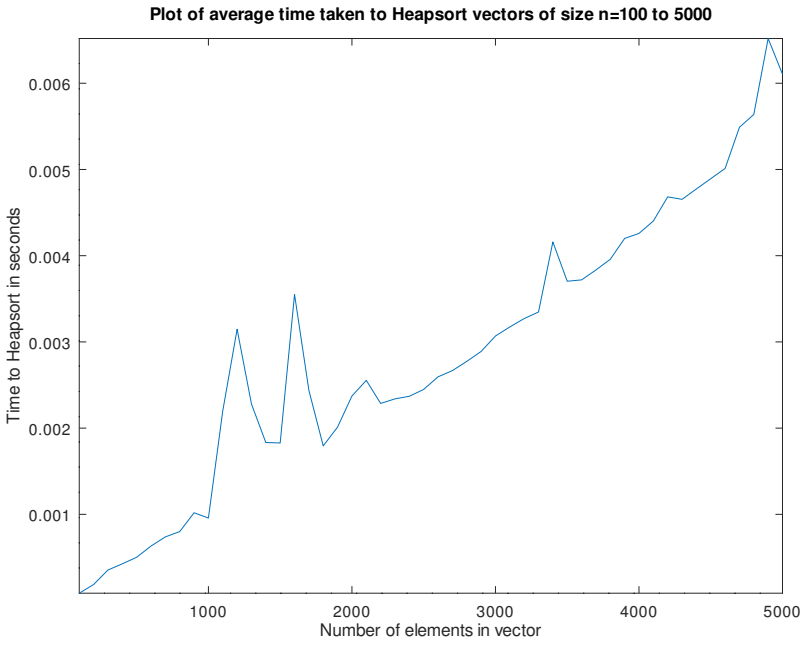

Figure 4: Plot of array length vs time taken to sort using heapsort algorithm

A visual inspection of the heapsort graph shows a similar relationship to that of the quicksort algorithm, with a slightly steeper slope, but with the same linear form of order n. We shall now show that the heapsort algorithm runs with complexity $\mathcal{O}(n\log(n))$.

Consider a binary tree with n elements. Then the number of layers this tree contains is equal to $\log(n)$. When calling the `heap_from_root` function, we would need to make some multiple of $\log(n)$ comparisons in the worst case scenario. This occurs when the root node and one of the leaf nodes need to be swapped. This operation is completed for $n/2$ elements, thus the time taken is of order $\mathcal{O}(n\log(n))$.

We then take the elements in turn from the heap, move the root and then sort the reduced heap until only one element remains. Again we have the worst case scenario of swapping a root and a leaf of time $\log(n)$. This is done n times, once for each element in the array. Thus the time taken is again of order $\mathcal{O}(n\log(n))$. Therefore, the total time taken is of order $\mathcal{O}(n\log(n))$. [3]

2.4 Algorithm Comparison

There are a number of considerations which should be analysed before choosing which algorithm to use for a given array of unsorted data. These include:

- Size of array
- Running time

- Memory constraints
- Partially and completely sorted array inputs

Therefore, the sorting algorithm used for a given data set must be chosen to play to the advantages of each. For example, we would recommend using the bubblesort algorithm for arrays of a smaller size (i.e. $n \leq 30$) as it is very efficient for these sizes. However, we would not use this for larger sets as its $\mathcal{O}(n^2)$ time complexity implies that the efficiency decreases exponentially.

In general, I would recommend the quicksort algorithm given a random array of unknown elements. This is due to the time complexity of the algorithm of order $\mathcal{O}(n\log(n))$ which is significantly more efficient compared to bubblesort. There is only a marginal difference in timings between the quicksort and heapsort for large n so either may be used in an efficient manner. However, when comparing Figures 3 and 4, we can see that quicksort is approximately twice as fast as heapsort for $n = 5000$. This is reasonable for our purposes as most arrays will be smaller in size than this.

3 Sorting Coordinate Arrays

Now we have successfully implemented three sorting algorithms for arrays in one dimension, we can abstract this to any number of dimensions we wish. For our purposes, these shall only be extended to the two-dimensional case, where the arrays contain pairs of integer coordinates. We cannot use the intuitive previous definition when comparing elements within the array as there is no obvious way of size comparison in two dimensions. Thus, we must define a lexicographical ordering on sets of coordinates given by

$$(X, Y) < (A, B) \quad \text{if} \quad X < A \quad \text{or} \quad (X = A \text{ and } Y < B).$$

We can then implement this in C++ as seen in appendix A.4 using the `cmp` function. From this, we calculate the time to sort n integer coordinates where n begins at 100 and increments in 100s up to a maximal size of 5000. This is done 100 times for each array length containing tuples and an average time is taken. These are then sorted using the bubble, quick and heapsort algorithms.

As we have increased the dimension in which the arrays store values, we can expect the sorting algorithms to have to complete more computations when executing. This results in the expected time to complete to be increased. Taking a look at our CoordinateArray class in appendix A.4, we can see that the comparison operator must get the X ordinate from the vector in every comparison and the Y ordinate in cases where $X < A$ is not true. This results in a significant increase in the time taken to complete the sorting functions.

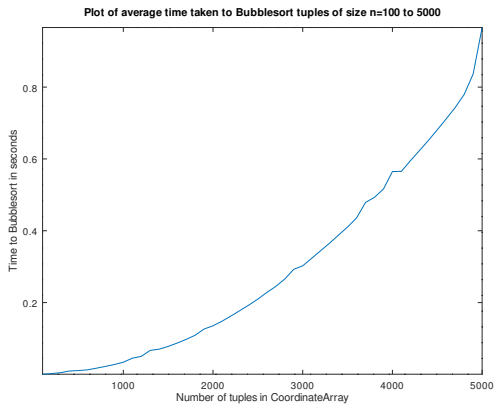

Figure 5: Plot of bubblesort for varying tuple sizes n

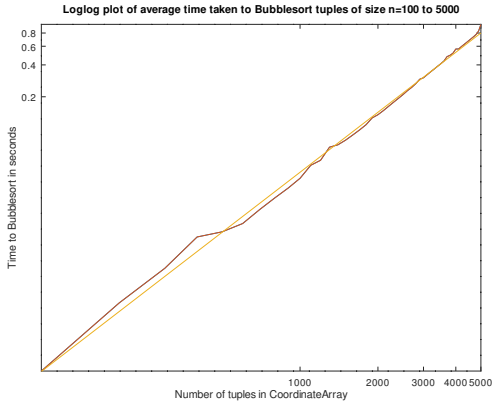

Figure 6: Loglog plot of bubblesort for varying tuple sizes n

Figures 5 and 6 show the average time taken to plot tuples of length n. We again see a similar shape to that of the one dimensional case. Figure 6 is a loglog plot, with line of best fit shown in yellow. This line of best fit has a gradient of 1.88 which implies the order of the bubblesort algorithm should be again of the form $\mathcal{O}(n^2)$. A comparison of the one-dimensional graph (Figure 1) and tuples graph (Figure 5) shows that the average time to bubblesort a list of tuples is four time that of the one-dimensional case.

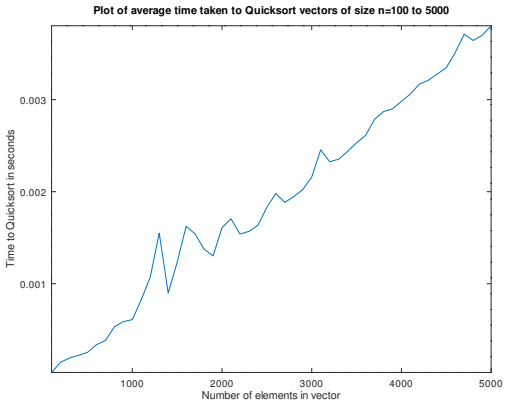

Figure 7: Plot of quicksort for varying vector sizes n

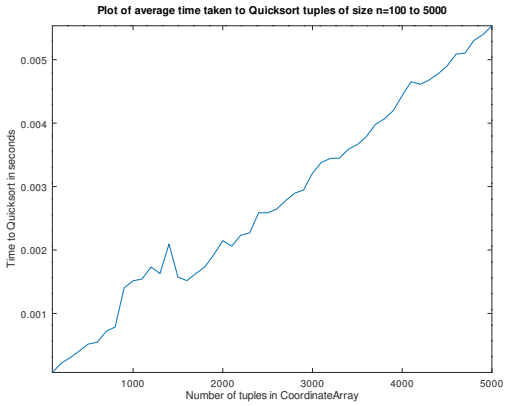

Figure 8: Plot of quicksort for varying tuple sizes n

Now looking at the quicksort graphs (Figures 7 and 8), we see that the tuples variant holds the same linear shape as expected, but results in an average time cost of twice that of the single dimension case.

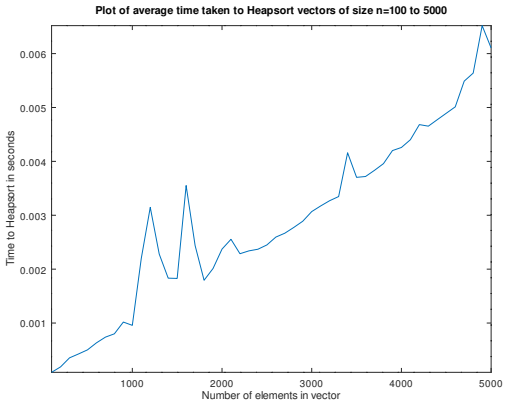

Figure 9: Plot of heapsort for varying vector sizes n

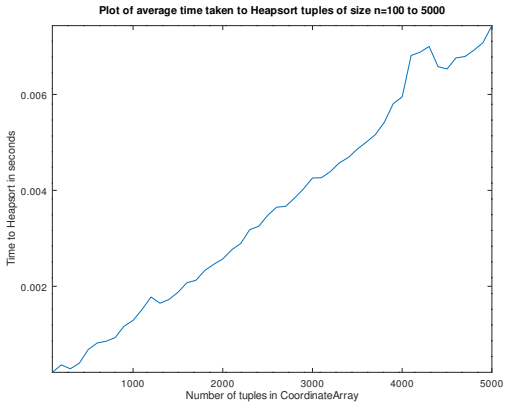

Figure 10: Plot of heapsort for varying tuple sizes n

As with the quicksort algorithm, we see the tuples case results in the same linear case as that of the one-dimensional case. Upon comparing Figures 9 and 10 we notice only a minor difference in timing (roughly two milliseconds) between the one and two dimensional cases.

When comparing which algorithm is preferable to implement and use for sorting coordinate arrays, it is obvious that the bubblesort algorithm is not suitable as this can take almost a second to sort a coordinate array containing 5000 tuples. This is poor in comparison to the quicksort and heapsort algorithms which can sort arrays of this size in the range of 6 to 8 milliseconds. There is a slight speed advantage using the quicksort algorithm compared to heapsort and thus, we shall continue using the quicksort algorithm in our future developments.

4 Game Of Life

The Game of Life (GoL) is an automated, recursive structure where each cell's life depends on the surrounding eight neighbours. There are a range of rules which can be implemented, but the standard set which we will use is:

- If a dead cells has exactly 3 living neighbours it becomes alive, else it remains dead
- If a living cell has 2 or 3 living neighbours it remains living, else it dies.

Initial implementations may create a two-dimensional array of booleans which store the locations of the living and dead cells. However, this is largely memory inefficient as configurations often branch out quickly. In addition, we would have to guess the size of the board needed to evolve the configuration prior to execution, which would often lead to wasted memory.

Our implementation of the GoL simply stores the locations of the living cells, with any cells not contained within this `CoordinateArray` assumed dead. This is far more memory efficient in general. For example, if we had a few living cells near the origin and another cluster a distance of n cells away, we would have to store an n by n array of booleans which would have to be updated at each GoL iteration. In our implementation, we would only store the location of the few living cells, a significant memory reduction.

Note that our implementation has coordinates stored in row-major order. This means that the origin is considered to be the top left entry, with the first entry of the coordinate being the row and the second being the column. This is the case as the game board can be considered a matrix, where mathematicians often denote the first ordinate as the row and the second the column. This row-major order structure is shown in Figure 11.

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Figure 11: Coordinates of Storage Array

4.1 Block

We can now evolve the system given a set of coordinates as initial conditions. We represent currently living cells with an X. Beginning with the first set of initial conditions $\{(1,1),(1,2),(2,1),(2,2)\}$ called a block, we evolve for one time step and get an identical set of coordinates as the output. This means that this configuration is a steady state, where the input cells is the same of the output after evolving one tick.

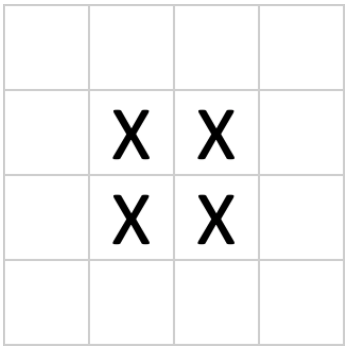

Figure 12: Block Configuration (a steady state)

4.2 Blinker

We can also consider another set of initial conditions which produces an alternate behaviour. Consider the set of initial coordinates $\{(5,4),(5,5),(5,6)\}$. We again see how this set of initial conditions evolve after one time step.

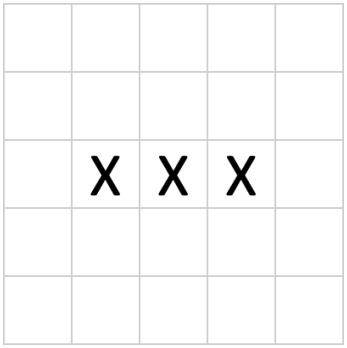

Figure 13: Initial condition of the Blinker configuration

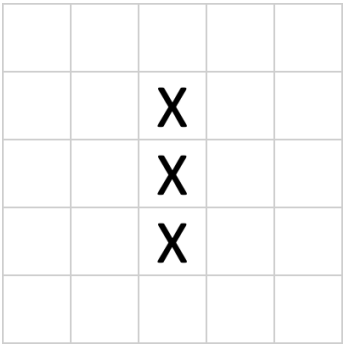

Figure 14: Blinker configuration after one iteration

The output after a single iteration is the coordinates $\{(4,5),(5,5),(6,5)\}$. This is a similar configuration to the initial condition, but with the horizontal initial line of 3 transformed into a vertical line. Performing the Game of Life with two iterations from the initial condition results in the same set of initial coordinates. This implies this set of initial condition produces a repeating pattern of three horizontal cells followed by three vertical cells and so on. This pattern is often called a blinker, with a periodicity of 2.

4.3 Acorn

Further sets of initial conditions can be implemented and run, producing interesting interactions and patterns. For example, one pattern of interest is called the acorn. This pattern was developed by Charles Corderman in around 1971 and was named by Robert Wainwright. He named it this after seeing the stable configuration which looked like an 'oak tree', mentioning the proverb "From little acorns mighty oaks do grow". [4]

The initial condition we shall use is

{(10001,10001), (10002, 10001), (10002, 10003), (10004, 10002), (10005, 10001),
(10006, 10001), (10007, 10001)}.

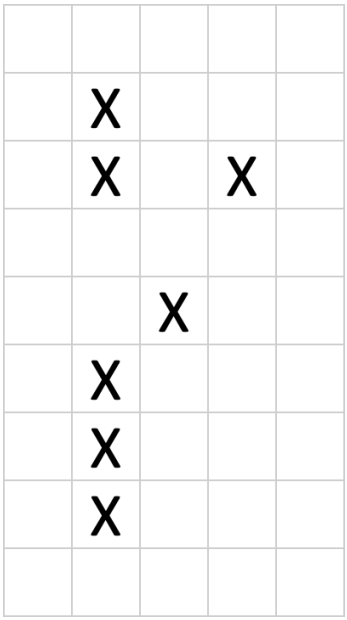

Figure 15: Acorn Configuration

This is a good test for our implementation of the algorithm. If we chose to use the array of booleans implementation, we would've required a board of size larger that 10000 by 10000, storing one hundred million entries which would have to be updated at each iteration. This would require huge amounts of memory and computation time. As our implementation only stores the coordinates of living cells, initially 7 entries, increasing to the maximal amount of living cells, in this case 1057 entries.

This set of initial conditions produces a stable configuration after 5206 iterations, with 633 living cells. This configuration is known as a 'Methuselah', a subset of patterns which takes a large number of generations (generally more than 100) to stabilise.

5 Conclusion

In this report, we implemented and discussed the efficiency and efficacy of three sorting algorithms and their use in a range of applications. We began by implementing the bubblesort, quicksort and heapsort, initially used to sort arrays in one dimension. We ran these sorting algorithms for varying sizes of randomly filled arrays and plotted the average time to sort them. From this analysis, we found that the bubblesort was not an appropriate algorithm for arrays of large size, with quicksort and heapsort both of a similar preferable efficacy.

We then abstracted our sorting algorithms to work in any number of dimensions. This allowed us to create an object which stores two dimensional coordinates, which can be sorted using a lexicographical ordering. We again create randomised the initial values of varying array lengths and calculating the average time to sort these. We plotted the results and found that the abstraction to two dimensions doubled the time to quicksort the arrays, whereas the difference in time between the one and two-dimensional case was marginal.

We again discounted the bubblesort as viable as arrays with 5000 tuples took over a second to fully sort. We found that the quicksort algorithm was slightly quicker than the heapsort, but for our purposes, both were just as effective as each other.

Through our analysis of the three different sorting algorithms, we found that the quicksort algorithm was the most effective and efficient in both the single and multi-dimensional cases when compared to the heapsort and bubblesort. The simplicity of the algorithm is a huge advantage as it can be explained quickly and implemented fairly simply, making it the ideal balance of ease of understanding and efficiency. Thus, I would advise using this in most cases where an array of any dimension is to be sorted.

We then implemented a Game of Life simulation which allowed us to run iterations using the Game of Life rules as seen in section 4. This allows the evolutions of different initial conditions to be found, given a set of rules which each living/dead cell must follow. Note in our implementation of the algorithm, we chose the quicksort algorithm to help condense of the array of coordinates which live in the next generation of the program. This is appropriate as dictated in our analysis in section 3 when comparing the efficiency of sorting coordinate arrays.

There are a number of applications which can be modelled using the Game of Life, with areas of interest often in physics and small scale biological problems due to the simplicity of the instructions used to evolve. The theory can also be extended to areas such as traffic problems and computer science issues such as computer viruses. [2]

Appendix A Source code listings

A.1 Bubblesort.cpp

```
1  //BUBBLESORT FUNCTIONS
2  //Sorts a vector using the bubble sort algorithm. Compares pairs of
       consecutive elements and swaps them if the ith element is larger than
       the (i+1)th.
3  void bubble(SortableContainer &v){
4
5      int VectorLength = v.size();
6
7      for (int j=0; j<VectorLength-1; j++){
8          for (int i=0; i<VectorLength-j-1; i++){
9              if (v.cmp(i+1,i)){
10                 v.swap(i,i+1);
11             }
12         }
13     }
14 }
```

A.2 Quicksort.cpp

```
1  //QUICKSORT FUNCTIONS
2  //Choose random pivot, move to end of vector, move all elements smaller
       than pivot to left and all larger to right.
3  int SetPartition(SortableContainer &v, int start, int end){
4      int pivotIndex = start + rand() % (end - start);
5      v.swap(end, pivotIndex);
6
7      int i = start - 1;
8
9      for(int j = start; j <= end-1; j++){
10         if(v.cmp(j,end)){
11             i++;
12             v.swap(i,j);
13         }
14     }
15
16     v.swap(i+1,end);
17
18     return(i+1);
19 }
20
21
22 //While the sublist is larger than one, quicksort the remaining sublists.
23 void quick_recursive(SortableContainer &v, int start, int end){
24     if(start < end){
25         int PartIndex = SetPartition(v, start, end);
26         quick_recursive(v, start, PartIndex - 1);
27         quick_recursive(v, PartIndex + 1, end);
28     }
29 }
30
31 //Wrapper function
32 void quick(SortableContainer &v) {quick_recursive(v,0,v.size()-1);}
```

A.3 Heapsort.cpp

```
1  //HEAPSORT FUNCTIONS
```

```
2  //Sorts a vector using the heapsort algorithm. Creates a heap from the
       initial array, create a max heap, then swap root and last item of heap
       and reduce heap size by 1. Repeat until only one element remains in
       heap.
3  void heap_from_root(SortableContainer &v, int i, int n){
4
5      //Declares
6      int biggest = i;
7      int left = 2*i+1;
8      int right = 2*i+2;
9
10     //Checks if left child is larger than root node
11     if(left < n && v.cmp(biggest,left)){
12         biggest = left;
13     }
14
15     //Checks if right child is larger than current largest
16     if(right < n && v.cmp(biggest,right)){
17         biggest = right;
18     }
19
20     //Checks if largest is not the root(i.e. if only one element in heap)
       and heapsorts remaining heap.
21     if(biggest != i){
22         v.swap(i, biggest);
23         heap_from_root(v, biggest, n);
24     }
25 }
26
27 void heap(SortableContainer &v){
28     //Declares
29     int n = v.size();
30
31     //Build the heap
32     for(int i = n/2 - 1; i>=0; i--){
33         heap_from_root(v, i, n);
34     }
35
36     //Take elements from heap one at a time, move root to end and heapsort
       the reduced heap
37     for(int i = n - 1; i>=0; i--){
38         v.swap(0, i);
39         heap_from_root(v, 0, i);
40     }
41 }
```

A.4 CoordinateArrayClass.cpp

```
1  //Integer Coordinates
2  class IntegerCoordinate{
3  public:
4      unsigned X, Y;
5  };
6
7
8  //CoordinateArray class, deriving from SortableContainer
9  class CoordinateArray : public SortableContainer{
10     std::vector<IntegerCoordinate> v;
11 public:
12     CoordinateArray() {}
13     CoordinateArray(int n):v(n){}
```

```
14
15      //Random number boundaries
16      int MinRand = 0;
17      int MaxRand = 100;
18
19      //Set X-coord of vector to chosen integer
20      void setX(int i, int x){
21          if (i+1 > v.size() || i < 0){
22              std::cout << "Error: Index outside range" << std::endl;
23              exit(-1);
24          }
25          else{
26              v[i].X = x;
27          }
28      }
29
30      //Set X-coord of vector to random value between MinRand and MaxRand
31      void setRandX(int i){
32          if (i + 1 > v.size() || i < 0){
33              std::cout << "Error: Index outside range" << std::endl;
34              exit(-1);
35          }
36          else{
37              v[i].X = rand() % (MaxRand - MinRand + 1) + MinRand;
38          }
39      }
40
41      //Set Y-coord of vector to chosen integer
42      void setY(int i, int y){
43          if (i + 1 > v.size() || i < 0){
44              std::cout << "Error: Index outside range" << std::endl;
45              exit(-1);
46          }
47          else{
48              v[i].Y = y;
49          }
50      }
51
52      //Set Y-coord of vector to random value between MinRand and MaxRand
53      void setRandY(int i){
54          if (i + 1 > v.size() || i < 0){
55              std::cout << "Error: Index outside range" << std::endl;
56              exit(-1);
57          }
58          else{
59              v[i].Y = rand() % (MaxRand - MinRand + 1) + MinRand;
60          }
61      }
62
63      //Returns X value of selected vector
64      double getX(int i){
65          if (i + 1 > v.size() || i < 0){
66              std::cout << "Error: Index outside range" << std::endl;
67              exit(-1);
68          }
69          else{
70              return v[i].X;
71          }
72      }
73
74      //Returns Y value of selected vector
```

```
75      double getY(int i){
76          if (i + 1 > v.size() || i < 0){
77              std::cout << "Error: Index outside range" << std::endl;
78              exit(-1);
79          }
80          else{
81              return v[i].Y;
82          }
83      }
84
85      //Comparison operator
86      bool cmp(int i, int j){
87          if((getX(i) < getX(j)) || ((getX(i) == getX(j)) && (getY(i) < getY(
        j)))){
88              return true;
89          } else {
90              return false;
91          }
92      }
93
94      //Swap two values in a vector
95      void swap(int i, int j){
96          double TempX = 0;
97          double TempY = 0;
98
99          TempX = getX(i);
100         TempY = getY(i);
101         setX(i, getX(j));
102         setY(i, getY(j));
103         setX(j, TempX);
104         setY(j, TempY);
105     }
106
107     //Resizing chosen vector
108     void resize(int n){
109         v.resize(n);
110     }
111
112     //Returns size of vector
113     int size() const { return v.size(); }
114 };
```

A.5 GameOfLife.cpp

```
1  //Function which takes the cells to be tested TestCells, a cell index j and
       the current living cells LiveCells and returns whether the selected
      test cell is in the current living cells
2  bool inLiveCells(CoordinateArray &TestCells, CoordinateArray &LiveCells,
       int j){
3      bool inLiveCells = false;
4      for(int i = 0; i < LiveCells.size(); i++){
5          if(LiveCells.getX(i) == TestCells.getX(j) && LiveCells.getY(i) ==
       TestCells.getY(j)){
6              inLiveCells = true; break;
7          }
8      }
9      return inLiveCells;
10 }
11
12
13 //Function which takes all possible next gen cells NeighbouringCells, a
```

```
    cell index j and the cells to be tested and returns how many occurences
     in NeighbouringCells the current test cell has
int HowManyNeighbours(CoordinateArray &TestCells, CoordinateArray &
    NeighbouringCells, int j){
    int count = 0;
    for(int i = 0; i < NeighbouringCells.size(); i++){
        if(NeighbouringCells.getX(i) == TestCells.getX(j) &&
    NeighbouringCells.getY(i) == TestCells.getY(j)){
            count += 1;
        }
    }
    return count;
}

//Function which removes all duplicate tuples from the CoordinateArray
    inputted
CoordinateArray RemoveDuplicates(CoordinateArray &CoordArr){
    CoordinateArray Temp(0);
    for(int i = 0; i < CoordArr.size()-1; i++){
        if((CoordArr.getX(i) != CoordArr.getX(i+1)) || (CoordArr.getY(i) !=
     CoordArr.getY(i+1))){
            Temp.resize(Temp.size()+1);
            Temp.setX(Temp.size()-1, CoordArr.getX(i));
            Temp.setY(Temp.size()-1, CoordArr.getY(i));
        }
    }

    Temp.resize(Temp.size()+1);
    Temp.setX(Temp.size()-1, CoordArr.getX(CoordArr.size()-1));
    Temp.setY(Temp.size()-1, CoordArr.getY(CoordArr.size()-1));
    return Temp;
}

//Working Game of Life code
CoordinateArray PlayGOL(CoordinateArray &LiveCells, int MaxIterations){
    for(int i = 0; i < MaxIterations+1; i++){

        //Outputs current generation number if MaxIterations not reached
        //Outputs final generation number, number of current living cells
    and the coordinates of said living cells

        if(i != MaxIterations){
            std::cout << "Generation " << i << "\n";
        }else{
            std::cout << "Generation " << i << ": Amount living " <<
    LiveCells.size() << ", Living Cells: " << LiveCells << "\n";
        }

        //Reset storage CoordinateArrays
        CoordinateArray NextGenLivingCells(0);
        CoordinateArray NeighbouringCells(0);

        //Generates CoordinateArray of NeighbouringCells for all currently
    living cells
        for(int counter = 0; counter < LiveCells.size(); counter++){
            for (int dx = -1; dx <= 1; dx++) {
                for (int dy = -1; dy <= 1; dy++) {
                    if (dx != 0 || dy != 0) {
                        NeighbouringCells.resize(NeighbouringCells.size()
    +1);
                        NeighbouringCells.setX(NeighbouringCells.size() -
```

```
    1, LiveCells.getX(counter) + dx);
65                      NeighbouringCells.setY(NeighbouringCells.size() -
    1, LiveCells.getY(counter) + dy);
66                  }
67              }
68          }
69      }
70
71          //Go through each NeighbouringCells entry, test whether it is in
    live cells, count how many times in NeighbouringCells and add to
    NextGenLiving according to the rules of GOL
72          for(int i = 0; i < NeighbouringCells.size(); i++){
73              int NeighbourCount = HowManyNeighbours(NeighbouringCells,
    NeighbouringCells, i);
74              bool LiveCellBoolean = inLiveCells(NeighbouringCells, LiveCells
    , i);
75              if(LiveCellBoolean == true && (NeighbourCount == 2 ||
    NeighbourCount == 3)){
76
77                  NextGenLivingCells.resize(NextGenLivingCells.size()+1);
78                  NextGenLivingCells.setX(NextGenLivingCells.size() - 1,
    NeighbouringCells.getX(i));
79                  NextGenLivingCells.setY(NextGenLivingCells.size() - 1,
    NeighbouringCells.getY(i));
80
81              } else if (LiveCellBoolean == false && (NeighbourCount == 3)){
82
83                  NextGenLivingCells.resize(NextGenLivingCells.size()+1);
84                  NextGenLivingCells.setX(NextGenLivingCells.size() - 1,
    NeighbouringCells.getX(i));
85                  NextGenLivingCells.setY(NextGenLivingCells.size() - 1,
    NeighbouringCells.getY(i));
86
87              }
88          }
89
90          //Check to see if all living cells have died, end game if so
91          if(NextGenLivingCells.size() == 0){
92              std::cout << "All cells have died: GAME OVER \n";
93              break;
94          }
95
96          //Run heapsort and removes duplicate cells found in NextGenLiving
    to save memory
97          quick(NextGenLivingCells);
98          NextGenLivingCells = RemoveDuplicates(NextGenLivingCells);
99
100         //Set NextGenLivingCells as LiveCells
101         LiveCells.resize(NextGenLivingCells.size());
102         for(int i = 0; i < NextGenLivingCells.size(); i++){
103             LiveCells.setX(i, NextGenLivingCells.getX(i));
104             LiveCells.setY(i, NextGenLivingCells.getY(i));
105         }
106     }
107
108     return LiveCells;
109 }
```

References

[1] Shaffer, C. (2013). Data Structures and Algorithm Analysis. 3.2nd edn. VA, United States

[2] Callahan, P. (2005). What is the Game of Life? Available at: http://www.math.com/students/wonders/life/life.html (Accessed on 11/11/19)

[3] Cormen, T., Leiserson, C., Rivest, R. and Stein, C. (2009), Introduction to Algorithms 3rd edn. Cambridge, MA, USA: MIT Press

[4] 'Acorn' (2020). LifeWiki. Available at https://www.conwaylife.com/wiki/Acorn. (Accessed 7th May 2021)