AlgoVisualizer

Reem Younis
Lecturer : Prof. Roi Poranne
Laboratory in Computer Graphics, University of Haifa
Email: reembyounis@gmail.com

August 1, 2021

Contents

Abstract. Algorithm visualization refers to visualization of information of software systems or algorithms. The objectives of software visualization are to support the understanding of software systems and algorithms(e.g., by animating the behavior of sorting algorithms) as well as their development and evolution. Algorithm visualizing is a very required tool since it helps in understanding how every algorithm works which leads to better implementations of such algorithms. In our work, we implemented most common algorithms in pathfinding and sorting, using Unity 3D.

1 Introduction

In this paper, we present our algorithm visualizer that supports three of common pathfinding algorithms in addition to four of common sorting algorithms.
Using UI buttons the user can pick features that he would like to see in one of those algorithms, which helps in better understanding of those algorithms.

2 Unity UI

Unity UI is a UI toolkit for developing user interfaces for games and applications. The user interface is the graphical layout of an application. It consists of the buttons users click on, the text they read, the images, sliders, text entry fields, and all the rest of the items the user interacts with. This includes screen layout, transitions, interface animations and every single micro-interaction. Any sort of visual element, interaction, or animation must all be designed. UI toolkit is a GameObject-based UI system that uses Components and the Game View to arrange, position, and style user interfaces.
In our project, we used Canvas which is the area that all UI elements are inside. It uses the EventSystem object to help the Messaging System.

2.1 Pathfinding UI

In the Pathfinding algorithms Scene we used **Sphere** as the start point and **Capsule** as the end point. In addition to **Canvas**, which contains : **Info**, is a panel that contains informations about the current pathfinding algorithm. **Visualize**, is click-on button to visualize the current algorithm. **PauseMenu** is a panel that is used to pause the current visualization. **Obstacles** is a click-on button to add obstacles to the plane. **Size** is a Dropdown button to pick Plane's size. **Info** is a click-on button to open Info panel. **Top** is the algorithm name's text. **Pause** is button to open the PauseMenu and pause the current visualization. **newPlane** is a click-on button to generate new plane.

Figure 1: UI used in pathfinding algorithm scene

2.2 Sorting UI

In the Sorting algorithms Scene we used **Canvas** which contains : **Info**, is a panel that contains informations about the current sorting algorithm. **Visualize**, is click-on button to visualize the current algorithm. **SortingSpeed** is a slider to pick the sorting speed visualizing. **CubesNum** is a dropdown to pick number of cubes to visualize. **NewArray** is a click-on button to generate new array. **Info** is a click-on button to open Info panel. **Top** is the algorithm name's text. **Pause** is button to open the PauseMenu and pause the current visualization. **PauseMenu** is a panel that is used to pause the current visualization.

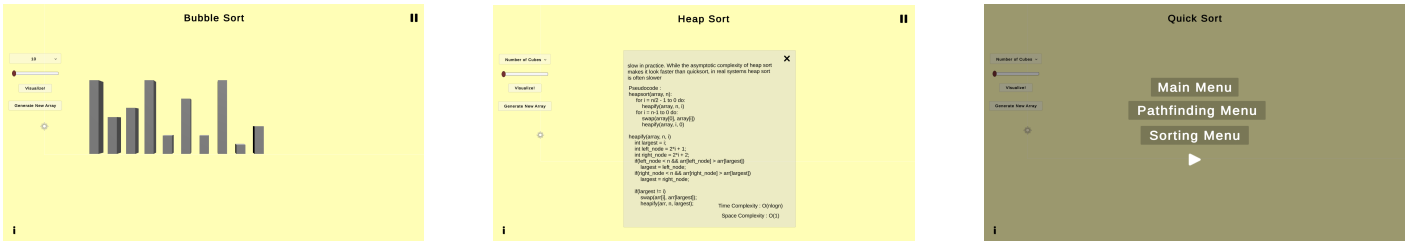

Figure 2: UI used in Sorting algorithm scene

3 Pathfinding Algorithms

Pathfinding is a plotting of the shortest path between two points. It is a more practical variant of solving maze. Pathfinding is closely related to the shortest path problem, within graph theory, which examines how to identify the path that best meets some criteria (shortest, cheapest, fastest, etc) between two points in a large network. A pathfinding method searches a graph by starting at one vertex and exploring adjacent nodes until the destination node is reached, generally with the intent of finding the cheapest route. Two primary problems of pathfinding are (1) to find a path between two nodes in a graph; and (2) the shortest path problem—to find the optimal shortest path.

3.1 A*

Arguably the best pathfinding algorithm; uses heuristics to guarantee the shortest path much faster than Dijkstra's Algorithm. A* assigns a weight to each open node equal to the weight of the edge to that node plus the approximate distance between that node and the finish. This approximate distance is found by the heuristic, and represents a minimum possible distance between that node and the end. This allows it to eliminate longer paths once an initial path is found.

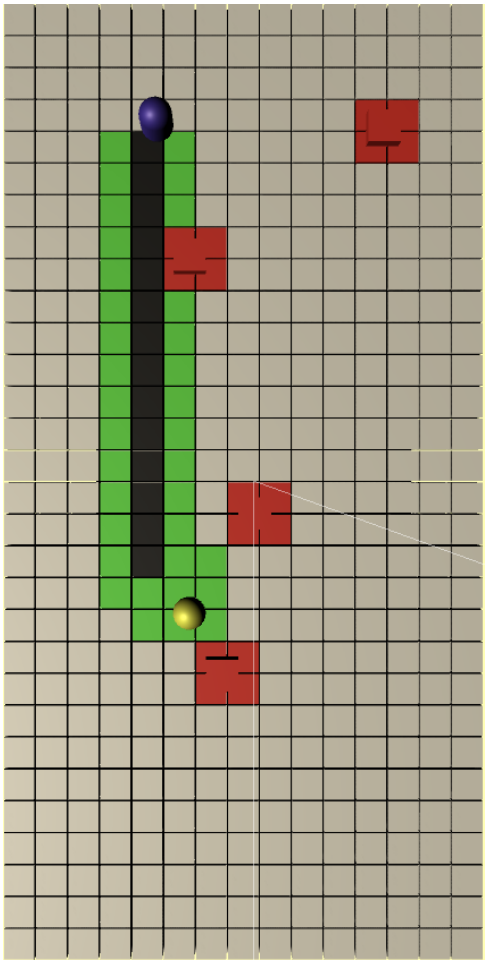

Figure 3: A* algorithm visualization

In figure 2, we can see that the obstacles are colored **red**, while the opened vertices (grids that has been visited) are colored **green**, and the final path is **black**.

3.1.1 Pseudocode

```
OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
        current = node in OPEN with the lowest f_cost
        remove current from OPEN
        add current to CLOSED

        if current is the target node //path has been found
                return

        foreach neighbour of the current node
                if neighbour is not traversable or neighbour is in CLOSED
                        skip to the next neighbour

                if new path to neighbour is shorter OR neighbour is not in OPEN
                        set f_cost of neighbour
                        set parent of neighbour to current
                        if neighbour is not in OPEN
                                add neighbour to OPEN
```

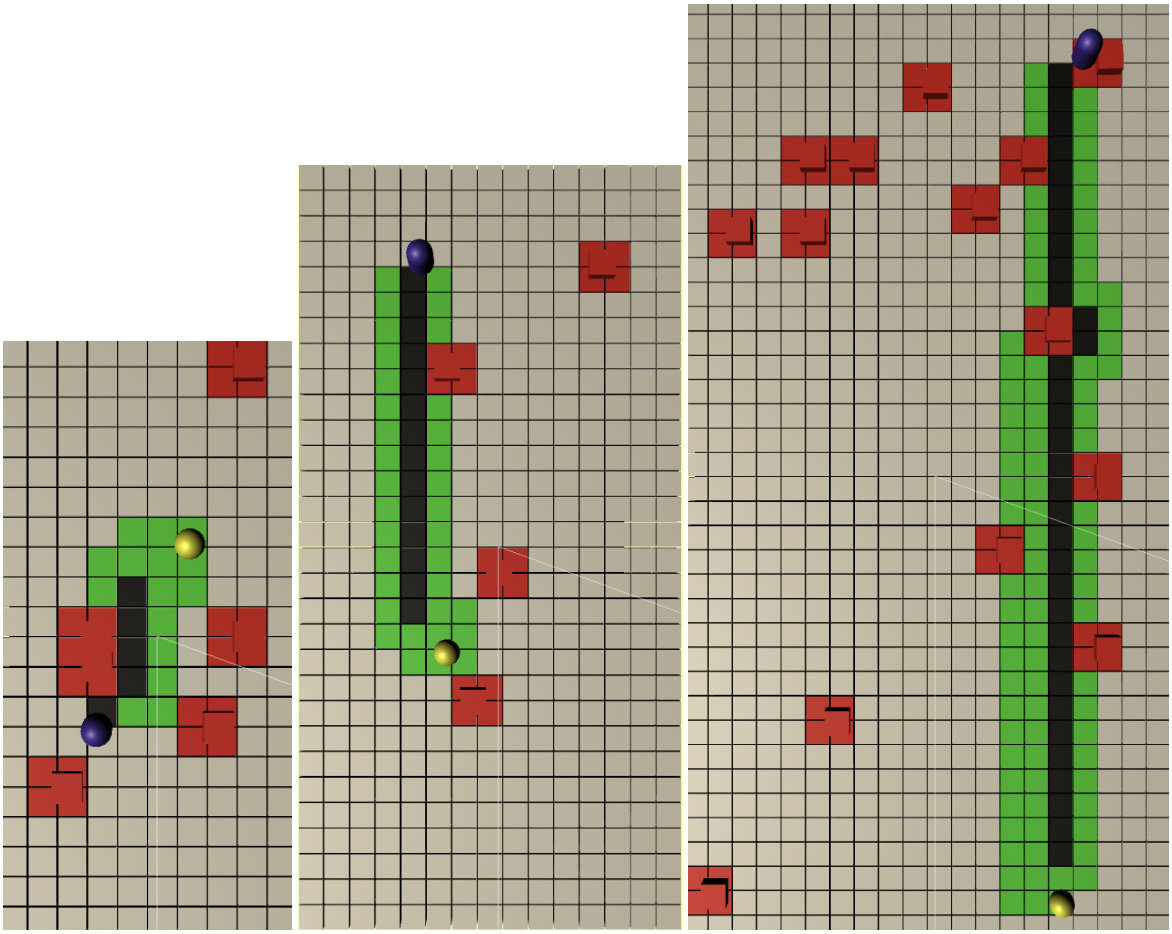

Figure 4: A* algorithm visualization of 3 different plane sizes

3.2 BFS

Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. It uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.
The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

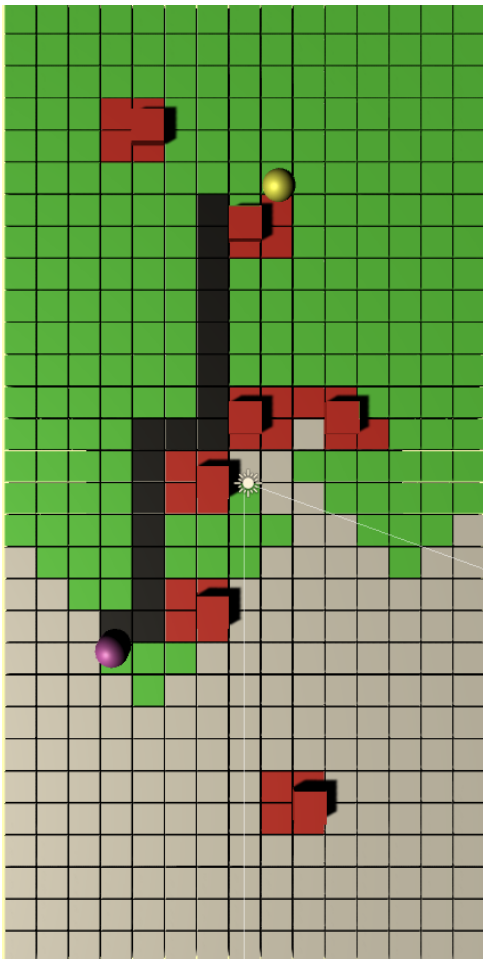

Figure 5: BFS algorithm visualization

In figure 4, we can see that the obstacles are colored **red**, while the opened vertices (grids that has been visited) are colored **green**, and the final path is **black**.

3.2.1 Pseudocode

```
BFS (G, s)     //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

      mark s as visited.
      while ( Q is not empty)
           //Removing that vertex from queue,whose neighbour will be visited now
           v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
```

```
Q.enqueue( w )  //Stores w in Q to further visit its neighbour
mark w as visited.
```

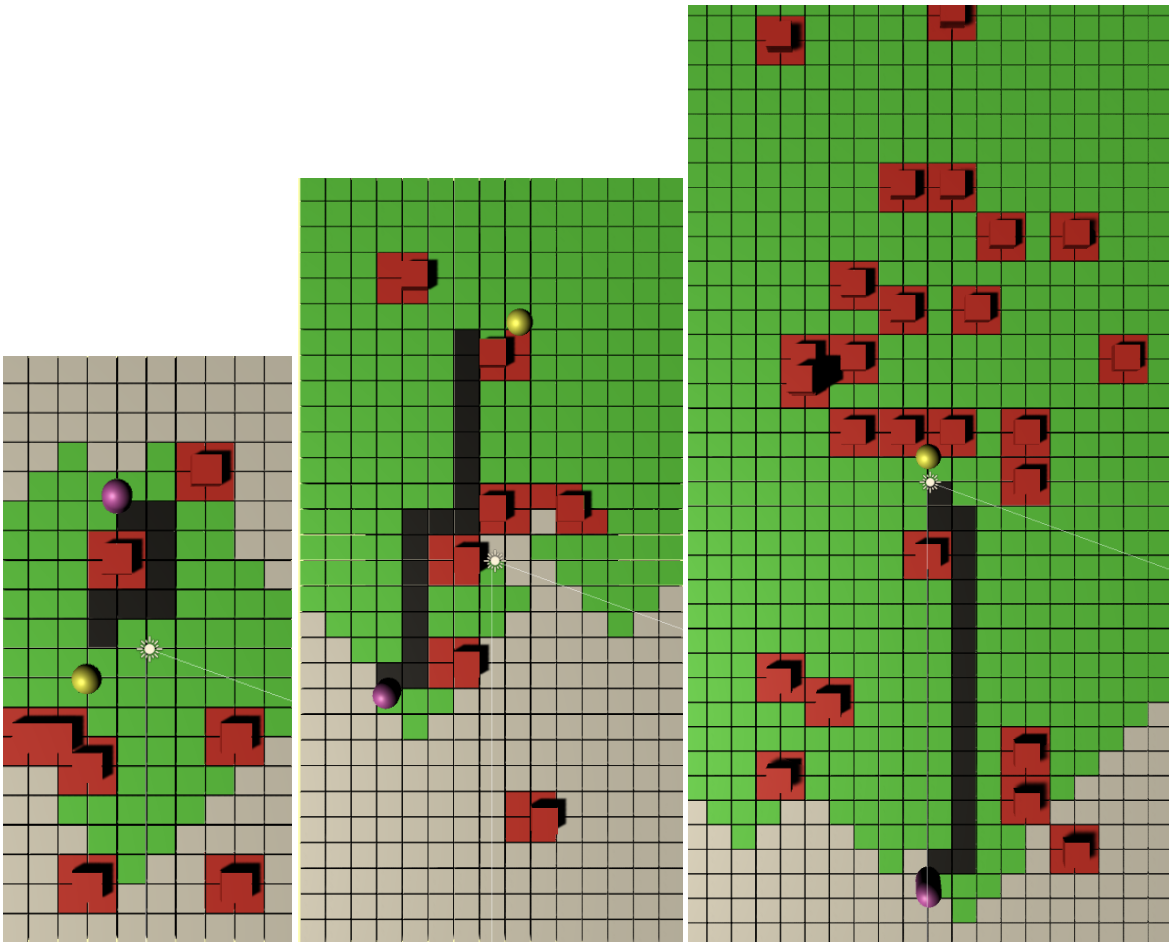

Figure 6: BFS algorithm visualization of 3 different plane sizes

3.3 DFS

Depth-first search (DFS) is an algorithm for searching a graph or tree data structure. The main strategy of depth-first search is to explore deeper into the graph whenever possible. It explores edges that come out of the most recently discovered vertex, ss. Only edges going to unexplored vertices are explored. When all of ss's edges have been explored, the search backtracks until it reaches an unexplored neighbor. This process continues until all of the vertices that are reachable from the original source vertex are discovered. If there are any unvisited vertices, depth-first search selects one of them as a new source and repeats the search from that vertex. The algorithm repeats this entire process until it has discovered every vertex. This algorithm is careful not to repeat vertices, so each vertex is explored once. DFS uses a stack data structure to keep track of vertices.

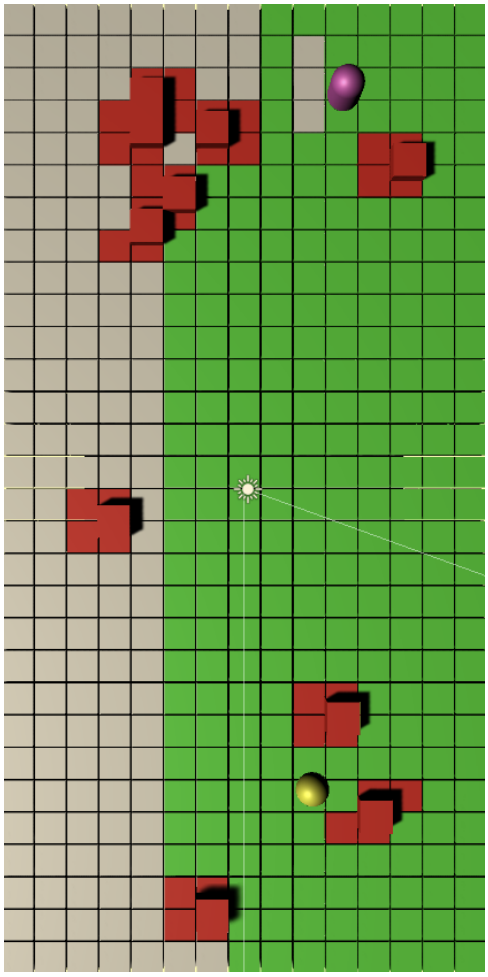

Figure 7: DFS algorithm visualization

In figure 6, we can see that the obstacles are colored **red**, while the opened vertices (grids that has been visited) are colored **green**.

3.3.1 Pseudocode

```
DFS-iterative (G, s):        //Where G is graph and s is source vertex
      let S be stack
      S.push( s )             //Inserting s in stack
      mark s as visited.
      while ( S is not empty):
          //Pop a vertex from stack to visit next
          v = S.top( )
         S.pop( )
         //Push all the neighbours of v in stack that are not visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                    S.push( w )
                   mark w as visited
```

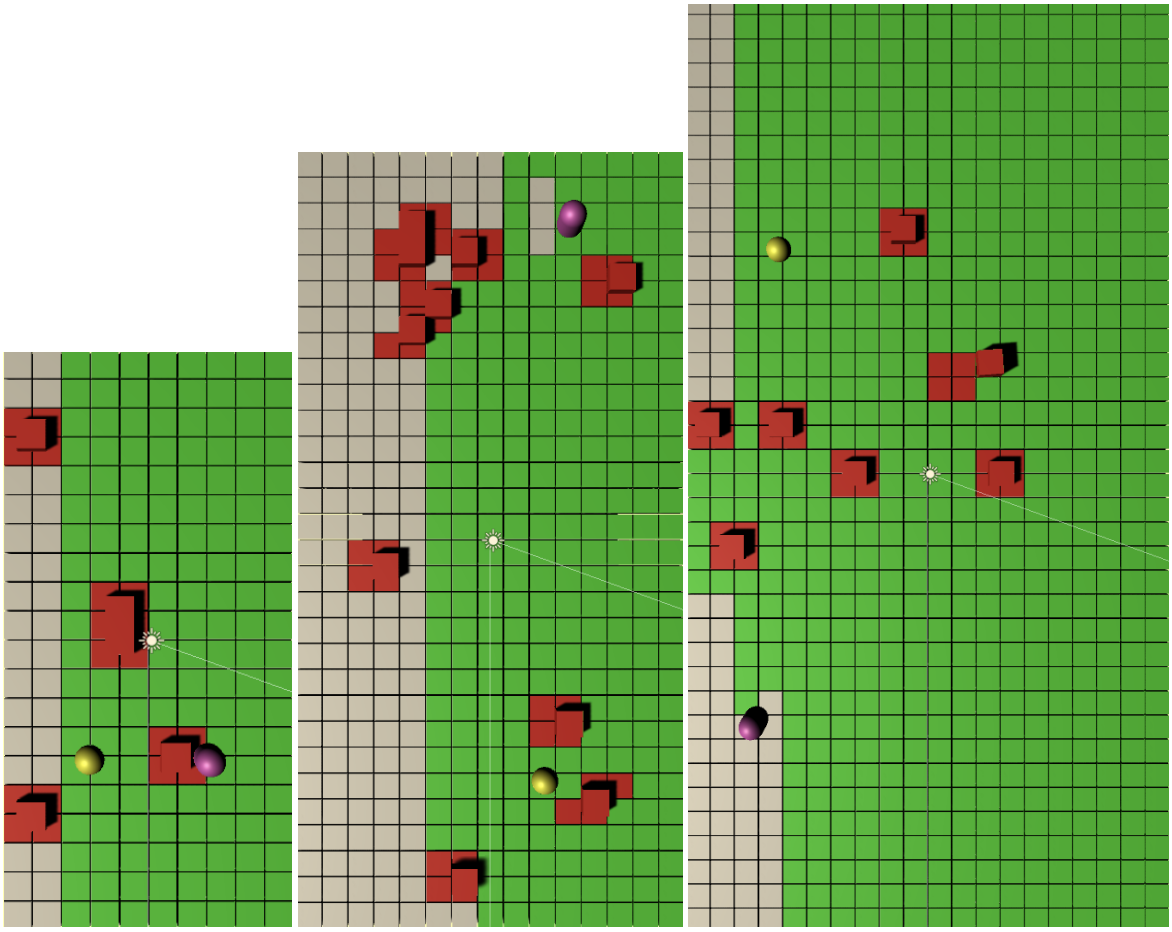

Figure 8: DFS algorithm visualization of 3 different plane sizes

4 Sorting Algorithms

Sorting algorithms are ways to organize an array of items from smallest to largest. These algorithms can be used to organize messy data and make it easier to use. Furthermore, having an understanding of these algorithms and how they work is fundamental for a strong understanding of Computer Science which is becoming more and more critical in a world of premade packages.
It is essential to explain the methods that professionals use to analyze and assess algorithm complexity and performance. The current standard is called “Big O notation” named according to its notation which is an “O” followed by a function such as “O(n).” In computer science, big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows.

4.1 Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. This algorithm performs poorly in real world use and is used primarily as an educational tool. Bubble sort loops through an array and sees if the number at one position is greater than the number in the following position which would result in the number moving up. This cycle repeats until the algorithm has gone through the array without having to change the order.

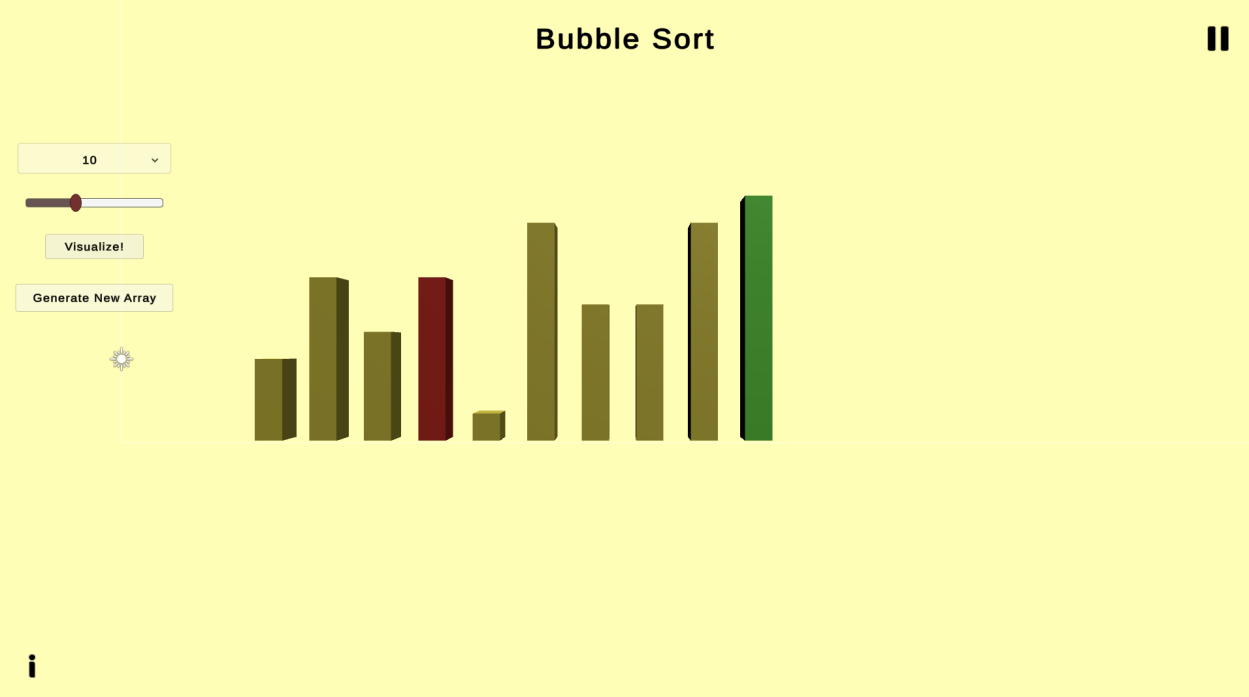

Figure 9: Bubble Sort algorithm visualization

In figure 9, we can see that the cube that is in the right position is colored **green**, while the current cube in hand that isn't in the right position is colored **red**.

4.1.1 Pseudocode

```
int i, j;
for (i = 0; i < n-1; i++)
        for (j = 0; j < n-i-1; j++)
                if (arr[j] > arr[j+1])
                        swap(arr[j], arr[j+1])
```

4.1.2 Complexity

In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = n(n-1)/2 = $O(n^2)$. Hence the **worst case time complexity** of Bubble Sort is **$O(n^2)$**. Also, the **best case time complexity** will be **O(n)**, it is when the list is already sorted.
The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for temp variable.

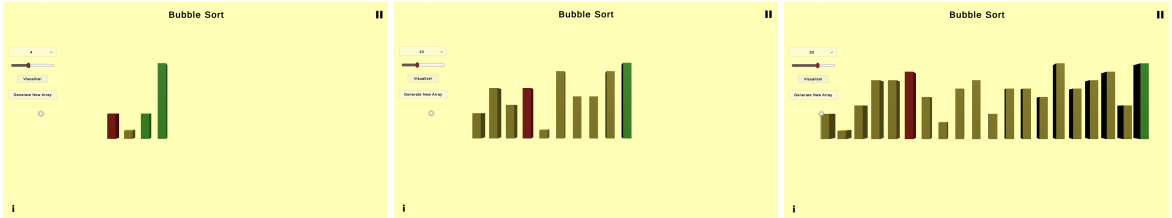

Figure 10: Bubble Sort algorithm for different number of cubes

4.2 Insertion Sort

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
Insertion sort works similarly as we sort cards in our hand in a card game. We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

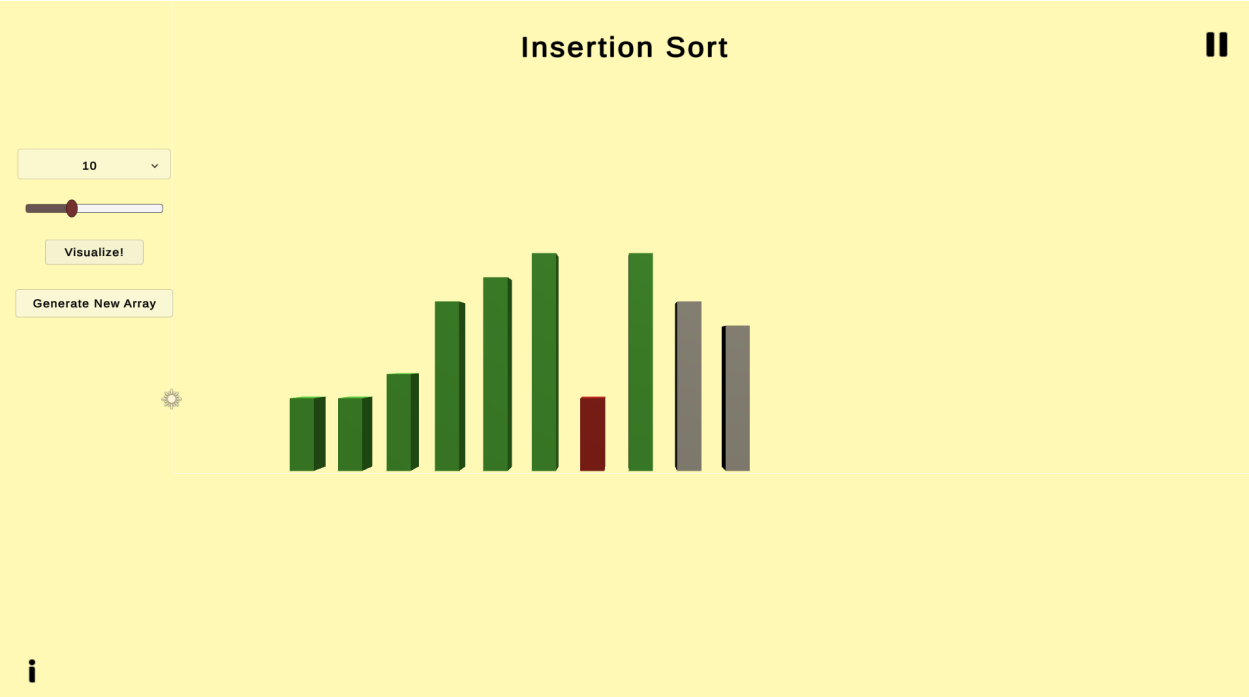

Figure 11: Bubble Sort algorithm visualization

In figure 11, we can see that the cube that is in the right position is colored **green**, while the current cube in hand that isn't in the right position is colored **red**.

4.2.1 Pseudocode

```
i = 1
while i < length(A)
    j = i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j = j - 1
    end while
    i = i + 1
end while
```

4.2.2 Complexity

The **worst case time complexity** for insertion sort will occur when the input list is in decreasing order. To insert the last element, we need at most n-1 comparisons and at most n-1 swaps. To insert the second to last element, we need at most n-2 comparisons and at most n-2 swaps, and so on. The number of operations needed to perform insertion sort is therefore: $2 \times (1 + 2 + \ldots + \text{n-2} + \text{n-1})$, which is equal to $n \times (n-1)$. It follows that the time complexity is $\mathbf{O(n^2)}$.
Insertion sort is a stable sort with a **space complexity** of **O(1)**.

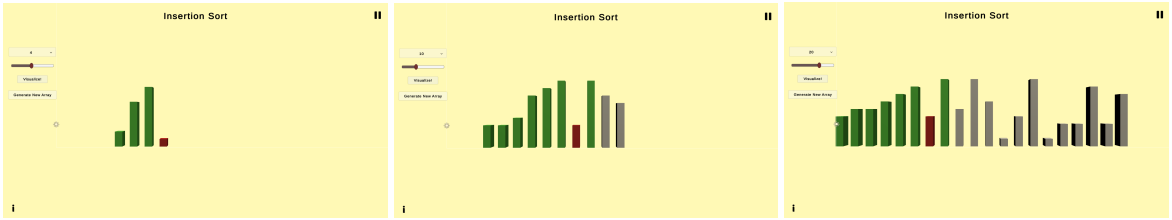

Figure 12: Insertion Sort algorithm for different number of cubes

4.3 Quick Sort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

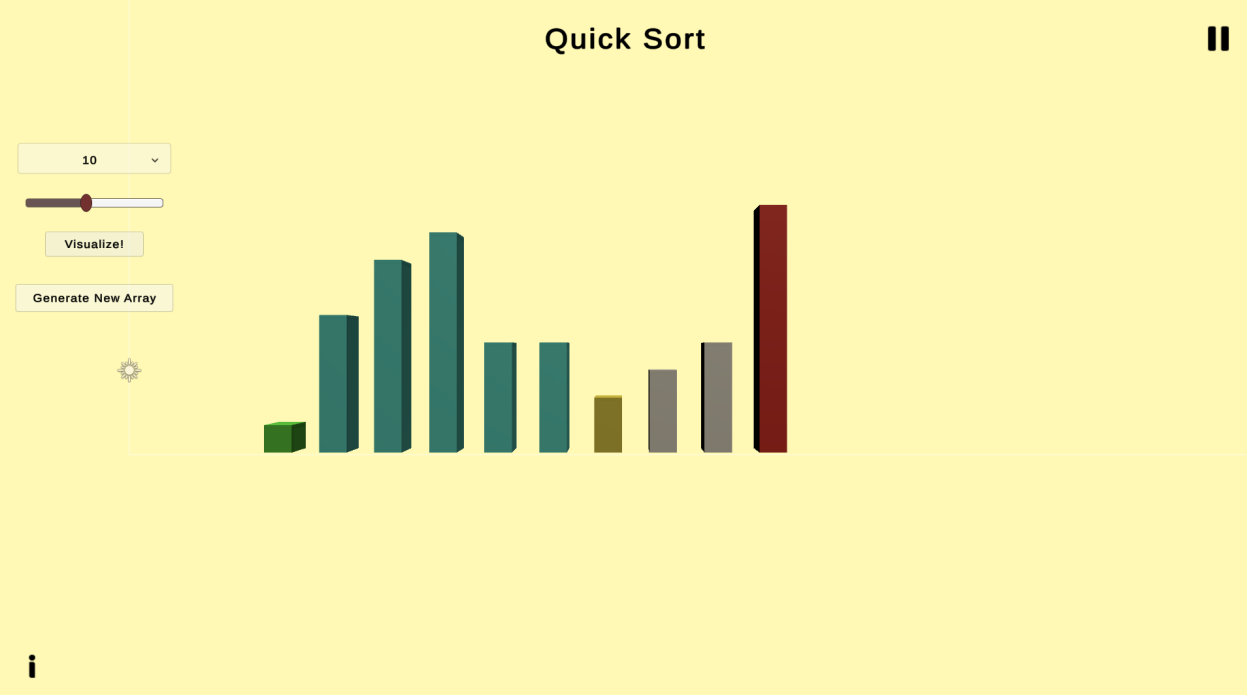

Figure 13: Quick Sort algorithm visualization

In figure 13, we can see that the cube that is in the right position is colored **green**, while the current cube in hand that isn't in the right position which is the pivot is colored **red**, on the other hand all the cubes that are smaller than the pivot are colored **cyan**.

4.3.1 Pseudocode

```
quickSort(arr[ ], low, high)
    if (low < high) :
        pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);

partition (arr[], low, high)
    pivot = arr[high];
    i = (low - 1)
    for (j = low; j <= high- 1; j++):
        if (arr[j] < pivot):
            i++;
            swap arr[i] and arr[j]

    swap arr[i + 1] and arr[high])
    return (i + 1)
```

4.3.2 Complexity

For an array, in which partitioning leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the pivot, hence on the right side. And if keep on getting unbalanced subarrays, then the running **worst time complexity** is **O(n^2)**.
Where as if partitioning leads to almost equal subarrays, then the **best time complexity** is **O(nlogn)**.

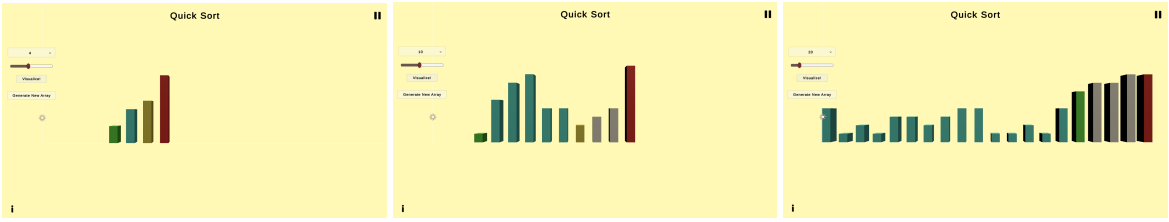

Figure 14: Quick Sort algorithm for different number of cubes

4.4 Heap Sort

Heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Heapsort maintains the unsorted region in a heap data structure to more quickly find the largest element in each step.
The Heapsort algorithm involves preparing the list by first turning it into a max heap. The algorithm then repeatedly swaps the first value of the list with the last value, decreasing the range of values considered in the heap operation by one, and sifting the new first value into its position in the heap. This repeats until the range of considered values is one value in length.

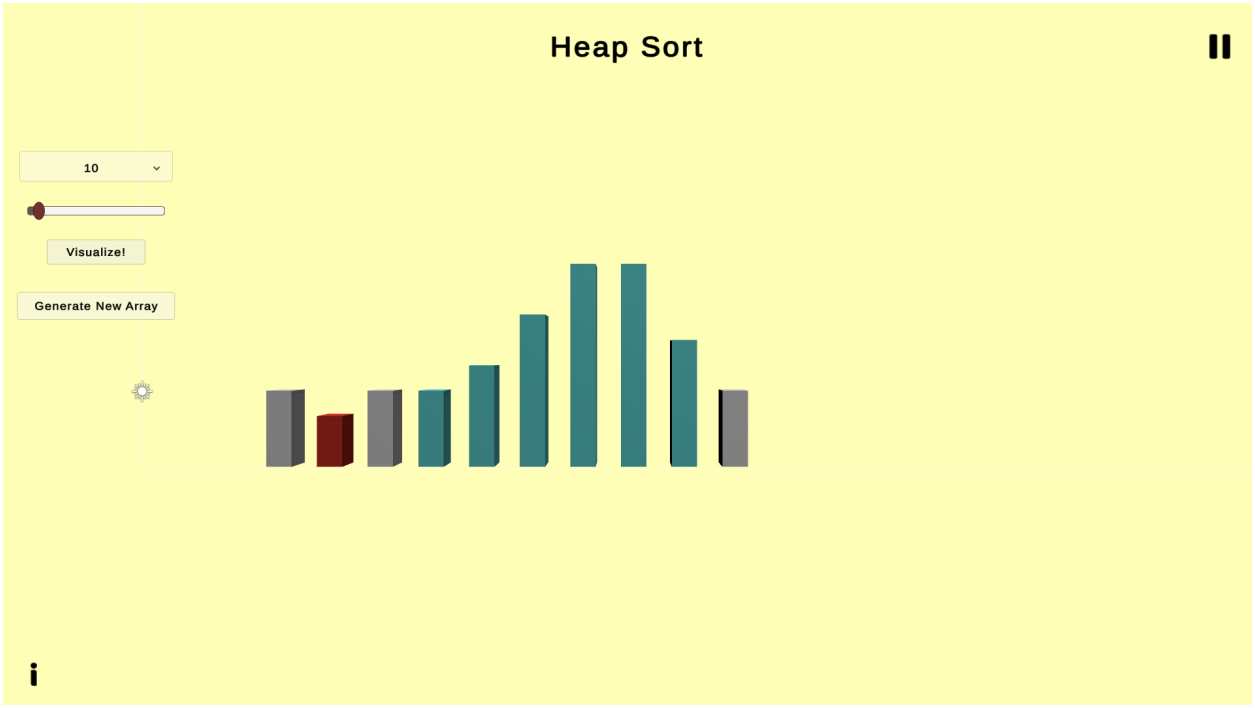

Figure 15: Heap Sort algorithm visualization

In figure 15, we can see that the cube that is in the right position is colored **green**, while the current cube in hand that isn't in the right position which is the pivot is colored **red**, on the other hand all the cubes that are smaller than the pivot are colored **cyan**.

4.4.1 Pseudocode

```
heapsort(array, n):
    for i = n/2 - 1 to 0 do:
        heapify(array, n, i)
```

```
    for i = n-1 to 0 do:
        swap(array[0], array[i])
        heapify(array, i, 0)

heapify(array, n, i)
    int largest = i;
    int left_node = 2*i + 1;
    int right_node = 2*i + 2;
    if(left_node < n && arr[left_node] > arr[largest])
        largest = left_node;
    if(right_node < n && arr[right_node] > arr[largest])
        largest = right_node;

    if(largest != i)
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
```

4.4.2 Complexity

Heapsort is a comparison-based sorting algorithm that uses a binary heap data structure. Like mergesort, heapsort has a **worst time complexity** of **O(nlogn)**, and like insertion sort, heapsort sorts in-place, so no extra space is needed during the sort.

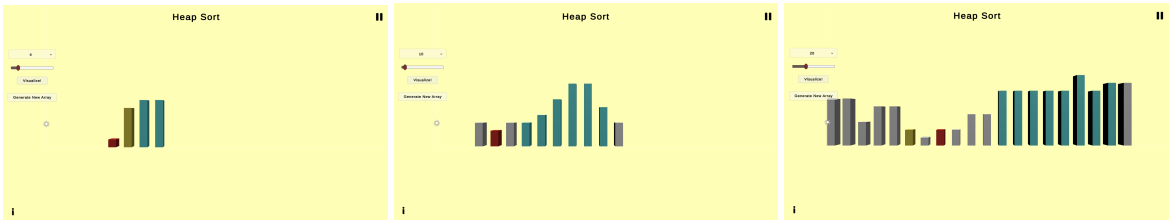

Figure 16: Heap Sort algorithm for different number of cubes