# Sorting Algorithm Visualizer Using py-game

**Summary of Features:**

* **Graphical Visualization**: Visual representation of sorting algorithms.
* **User Input via GUI**: Interactive input for the number of elements and their value range.
* **Multiple Algorithms**: Visualization for various sorting algorithms.
* **Algorithm Selection**: Easy switching between different algorithms using keyboard shortcuts.
* **Sorting Order Control**: Option to sort in ascending or descending order.
* **Control Over Sorting**: Start, pause, and reset the sorting process.
* **Real-time Updates**: Visual feedback on the sorting process, with color-coded comparisons and swaps.
* **Responsive Layout**: Adaptation to different window sizes.
* **User Instructions**: Clear instructions and feedback provided within the GUI.  
    
  ---------------------------------------------------------------------------------------------------------  
  Algorithms Explanation:

1. Bubble Sort:

def bubble\_sort(draw\_info, ascending=True):  
 lst = draw\_info.lst  
  
 for i in range(len(lst) - 1):  
 for j in range(len(lst) - 1 - i):  
 num1 = lst[j]  
 num2 = lst[j + 1]  
  
 if (num1 > num2 and ascending) or (num1 < num2 and not ascending):  
 lst[j], lst[j + 1] = lst[j + 1], lst[j]  
 draw\_list(draw\_info, {j: draw\_info.GREEN, j + 1: draw\_info.RED}, True)  
 yield True  
  
 return lst

* **lst = draw\_info.lst**: Get the list to be sorted from the **DrawInformation** instance.
* The outer loop iterates over the list to perform multiple passes.
* The inner loop compares adjacent elements and swaps them if they are in the wrong order.
* The **if** statement checks the order condition based on the **ascending** parameter.
* **draw\_list** is called to update the visualization, highlighting the elements being compared.
* **yield True** is used to pause the sorting to update the display.

1. Insertion Sort:

def insertion\_sort(draw\_info, ascending=True):  
 lst = draw\_info.lst  
  
 for i in range(1, len(lst)):  
 current = lst[i]  
  
 while True:  
 ascending\_sort = i > 0 and lst[i - 1] > current and ascending  
 descending\_sort = i > 0 and lst[i - 1] < current and not ascending  
  
 if not ascending\_sort and not descending\_sort:  
 break  
  
 lst[i] = lst[i - 1]  
 i = i - 1  
 lst[i] = current  
 draw\_list(draw\_info, {i - 1: draw\_info.GREEN, i: draw\_info.RED}, True)  
 yield True  
  
 return lst

* **lst = draw\_info.lst**: Get the list to be sorted from the **DrawInformation** instance.
* The outer loop iterates over the list starting from the second element.
* The **while** loop shifts elements in the sorted portion to make space for the current element.
* **ascending\_sort** and **descending\_sort** determine if the current element should be shifted based on the **ascending** parameter.
* **draw\_list** updates the visualization, highlighting the elements being compared and moved.
* **yield True** pauses the sorting to update the display.

1. Quick Sort:

def quick\_sort(draw\_info, ascending=True):  
 lst = draw\_info.lst  
 stack = [(0, len(lst) - 1)]  
  
 while stack:  
 start, end = stack.pop()  
 if start >= end:  
 continue  
  
 pivot\_index = start  
 pivot = lst[end]  
 for i in range(start, end):  
 if (lst[i] < pivot and ascending) or (lst[i] > pivot and not ascending):  
 lst[i], lst[pivot\_index] = lst[pivot\_index], lst[i]  
 pivot\_index += 1  
 lst[pivot\_index], lst[end] = lst[end], lst[pivot\_index]  
 draw\_list(draw\_info, {pivot\_index: draw\_info.GREEN}, True)  
 yield True  
  
 stack.append((start, pivot\_index - 1))  
 stack.append((pivot\_index + 1, end))  
  
 return lst

* **lst = draw\_info.lst**: Get the list to be sorted from the **DrawInformation** instance.
* A stack is used to simulate the recursive calls.
* The **while** loop processes the stack until all segments are sorted.
* **pivot\_index** is initialized to **start**, and **pivot** is the element at **end**.
* The **for** loop partitions the segment around the pivot.
* **draw\_list** updates the visualization, highlighting the pivot element.
* **yield True** pauses the sorting to update the display.
* The stack is updated with the new segments to be sorted.

1. Merge Sort:

def merge\_sort(draw\_info, ascending=True):  
 lst = draw\_info.lst  
 current\_size = 1  
  
 while current\_size < len(lst):  
 left = 0  
 while left < len(lst) - current\_size:  
 mid = left + current\_size - 1  
 right = min((left + 2 \* current\_size - 1), (len(lst) - 1))  
  
 temp = []  
 left\_subarray = lst[left:mid + 1]  
 right\_subarray = lst[mid + 1:right + 1]  
 while left\_subarray and right\_subarray:  
 if (left\_subarray[0] < right\_subarray[0] and ascending) or (  
 left\_subarray[0] > right\_subarray[0] and not ascending):  
 temp.append(left\_subarray.pop(0))  
 else:  
 temp.append(right\_subarray.pop(0))  
 temp.extend(left\_subarray)  
 temp.extend(right\_subarray)  
  
 for i in range(left, right + 1):  
 lst[i] = temp[i - left]  
 draw\_list(draw\_info, {i: draw\_info.BLUE}, True)  
 yield True  
  
 left += current\_size \* 2  
  
 current\_size \*= 2  
  
 return lst

* **lst = draw\_info.lst**: Get the list to be sorted from the **DrawInformation** instance.
* **current\_size** starts at 1 and doubles each pass.
* The **while** loop processes segments of increasing size.
* The nested **while** loop iterates over the list, merging segments.
* **left\_subarray** and **right\_subarray** are created for merging.
* The **while** loop merges the two subarrays.
* **draw\_list** updates the visualization, highlighting the merging process.
* **yield True** pauses the sorting to update the display.

1. Selection Sort:

def selection\_sort(draw\_info, ascending=True):  
 lst = draw\_info.lst  
  
 for i in range(len(lst)):  
 min\_index = i  
  
 for j in range(i + 1, len(lst)):  
 if (lst[j] < lst[min\_index] and ascending) or (lst[j] > lst[min\_index] and not ascending):  
 min\_index = j  
  
 lst[i], lst[min\_index] = lst[min\_index], lst[i]  
 draw\_list(draw\_info, {i: draw\_info.GREEN, min\_index: draw\_info.RED}, True)  
 yield True  
  
 return lst

* **lst = draw\_info.lst**: Get the list to be sorted from the **DrawInformation** instance.
* The outer loop iterates over the list.
* The inner loop finds the minimum element in the unsorted portion.
* **draw\_list** updates the visualization, highlighting the elements being swapped.
* **yield True** pauses the sorting to update the display.

1. Heap Sort:

def heap\_sort(draw\_info, ascending=True):  
 lst = draw\_info.lst  
 n = len(lst)  
  
 for i in range(n // 2 - 1, -1, -1):  
 yield from heapify(lst, n, i, ascending, draw\_info)  
  
 for i in range(n - 1, 0, -1):  
 lst[i], lst[0] = lst[0], lst[i]  
 yield from heapify(lst, i, 0, ascending, draw\_info)  
 yield True

* **lst = draw\_info.lst**: Get the list to be sorted from the **DrawInformation** instance.
* Build the max heap from the bottom up.
* The **heapify** function ensures the heap property is maintained.
* Swap the root with the last element and reduce the heap size.
* **draw\_list** updates the visualization, highlighting the heapify process.
* **yield True** pauses the sorting to update the display.

1. Shell Sort

def shell\_sort(draw\_info, ascending=True):  
 lst = draw\_info.lst  
 n = len(lst)  
 gap = n // 2  
  
 while gap > 0:  
 for i in range(gap, n):  
 temp = lst[i]  
 j = i  
 while j >= gap and ((lst[j - gap] > temp and ascending) or (lst[j - gap] < temp and not ascending)):  
 lst[j] = lst[j - gap]  
 j -= gap  
 lst[j] = temp  
 draw\_list(draw\_info, {j: draw\_info.GREEN, j + gap: draw\_info.RED}, True)  
 yield True  
 gap //= 2  
  
 return lst

* **lst = draw\_info.lst**: Get the list to be sorted from the **DrawInformation** instance.
* The **gap** starts at half the length of the list and reduces by half each pass.
* The outer loop iterates over the list with the current gap.
* The inner **while** loop performs insertion sort within each gap.
* **draw\_list** updates the visualization, highlighting the elements being moved.
* **yield True** pauses the sorting to update the display.

1. Comb Sort:

def comb\_sort(draw\_info, ascending=True):  
 lst = draw\_info.lst  
 n = len(lst)  
 gap = n  
 shrink = 1.3  
 sorted = False  
  
 while not sorted:  
 gap = int(gap / shrink)  
 if gap <= 1:  
 gap = 1  
 sorted = True  
  
 for i in range(n - gap):  
 j = i + gap  
 if (lst[i] > lst[j] and ascending) or (lst[i] < lst[j] and not ascending):  
 lst[i], lst[j] = lst[j], lst[i]  
 sorted = False  
 draw\_list(draw\_info, {i: draw\_info.GREEN, j: draw\_info.RED}, True)  
 yield True  
  
 return lst

* **lst = draw\_info.lst**: Get the list to be sorted from the **DrawInformation** instance.
* **gap** starts as the length of the list and shrinks by a factor of 1.3 each pass.
* The **while** loop continues until the list is sorted.
* The inner loop compares elements that are **gap** positions apart and swaps them if they are in the wrong order.
* **draw\_list** updates the visualization, highlighting the elements being compared and swapped.
* **yield True** pauses the sorting to update the display.