COMP2012H Final Project Documentation

Lee Young Kyu 20219190

**Implementation**

**Sorting**

텍스트이(가) 표시된 사진

자동 생성된 설명

Sorting class is the class where the actual sorting procedures take place. It creates, shuffles, and sorts integer array of specific size. It is a member variable of MainWindow class.

**Data members explanation**

* QStringList algorithms: List of available sorting algorithms in this project.
* QString algorithm: The algorithm which is chosen currently.
* QStringList shuffles: List of available options for the shuffling of the array.
* QString shuffleType: The shuffling option which is chosen currently.
* QStringList sizelist: List of available size of the array.
* int size: Size of arr.
* int\* arr: The actual array which sorting and shuffling procedures take place.
* const int MAX\_NUMBER: An upper bound to the elements in the array.
* int animDelay: Delay between each change in the visualization
* unsigned int num\_comparisons: Number of comparisons made in the sorting procedure. Note that since Radix sort and Counting sorts are not comparison-based sorting algorithm, they do not increase num\_comparisons.
* unsigned int num\_changes: Number of times where the elements of the array are changed.
* int\* color\_index: An array of indices to be colored in the visualization.
* int color\_size: Size of color\_index.

**Member functions explanation**

* void swap(int\* arr, int size, int i, int j): Swap arr[i] and arr[j] and increase num\_changes by 2. Used in most in-place sorting algorithm.
* void insert(int\* arr, int size, int i): Search for correct relative position for arr[i] and insert arr[i] into that position, and increase the num\_changes and num\_comparisons accordingly. In the searching of the position, linear search is used.
* void merge(int\* arr, int size, int start, int end): For subarray of arr (from index start to end), split it into two smaller array of similar size, and merge them with typical merging method.
* void heapify(int\* arr, int size,int heapsize, int i): Heapify a subtree rooted with node i, with size heapsize. Max heap is used in this function.
* Sorting(QObject\* parent): A constructor for Sorting class.
* QStringList getAlgorithms(): Returns algorithms, which is used in comboBoxAlgorithm in MainWindow class.
* QStringList getShuffles(): Returns shuffles, which is used in comboBoxShuffle in MainWindow class.
* QStringList getSizeList(): Returns sizelist, which is used in comboBoxArraySize in MainWindow class.
* void run(): Starts sorting arr with chosen algorithm.
* void shuffle(): Shuffles arr with chosen shuffling option.
* void setSize(int size): Changes size of arr. For size 1024, there will be some repeated elements, where for other sizes, every element is unique.
* void setAnimSpeed(int speed): Changes speed of visualization.
* void setAlgorithm(QString option): Changes sorting algorithm to be applied on arr.
* Void setShuffle(QString option): Changes shuffling option. “Random” and “Reverse” are self explanatory, where “Almost sorted” is different from typical definition, it swap two random elements in sorted array. (If we use typical definition, which every element is at most 1 unit away from it’s correct position, it is almost impossible to see the difference of almost sorted array and sorted array when the size of array is large, thus this alternative option is used.)
* void createArray(): Creates an array to be shuffled and sorted.
* int getDefaultAnimSpeed(): Returns DEFAULT\_ANIM\_SPEED. It is used for spinBoxAnimSpeed and SliderSpeed in MainWindow class.
* int getMaxAnimSpeed(): Returns MAX\_ANIM\_SPEED. It is used for spinBoxAnimSpeed and SliderSpeed in MainWindow class.
* unsigned int get\_num\_comparisons(): Returns num\_comparisons. It is used in labelComparisons in MainWindow class.
* unsigned int get\_num\_changes(): Returns num\_changes. It is used in labelChanges in MainWindow class.
* void sort\_bubble(int\* arr, int size): Bubble sort. Repeatedly steps through the list and compares adjacent elements, and swaps them if they are in wrong order. Repeat the procedure until arr is sorted. Note that after iteration, arr[size-n] is in correct position, so we. don't need to consider them in remaining procedure. This optimization is applied in the algorithm.
* void sort\_insertion(int\* arr, int size): Insertion sort. Calls insert(int\* arr, int size, int i) function for each i in increasing order.
* void sort\_selection(int\* arr, int size): Selection sort. Look for the minimum value from the element to the last element, and swap the minimum value with value, for each i in increasing order.
* void sort\_merge(int\* arr,int size, int start, int end): Merge sort. It is a recursive algorithm. Divide arr into 2 subarrays of smaller size, and use Merge sort to sort those 2 subarrays, then merge them using merge(int\* arr, int size, int start, int end) function.
* void sort\_heap(int\* arr, int size): Divide arr into sorted part and unsorted part, finds the largest element in unsorted part using heap structure, then move the largest element in unsorted part to sorted part, and repeat the procedure.
* void sort\_quick(int\* arr, int size, int start, int end): Quicksort. Choose a pivot element, and place all elements smaller than pivot to the left of pivot, and all the elements larger than pivot to right of pivot. Then use quicksort again for the smaller part and larger part. In our project, rightmost element is always chosen to be pivot.
* void sort\_radix(int\* arr, int size, int base): Radix sort. Use stable sorting algorithm to sort arr, not based on their exact value, but based in their value at certain digit when they are represented in number of base base. Repeat the procedure until all the digits are considered. In our project, the sorting order is set to be from least significant digit to most significant digit, and counting sort is used for the stable sorting algorithm. It is not a comparison-based sorting algorithm, thus it does not increase the num\_comparisons. In our project, base 2 and base 10 are used.
* void sort\_counting(int\* arr, int size, int base, int exp): Counting sort. Use another array to store the frequency of each element in arr, and use frequency array to find the correct place for each element and put them into arr. It is not a comparison-based sorting algorithm, thus it does not increase the num\_comparisons.
* void sort\_gnome(int\* arr, int. size): Gnome sort. Similar to insertion sort. But instead of using linear search to find the correct relative position, it repeatedly swaps the element with element before that until the element is in correct relative position.
* void sort\_cocktail(int\* arr, int size): Cocktail sort (also known as Cocktail shaker sort). Similar to bubble sort, but after each iteration, it iterates in reverse direction too (hence it canbe considered as bi-directional bubble sort). Note that after iteration, arr[n-1] and arr[size-n] are in correct position, thus we do not need to consider them in the sorting procedure. This optimization us applied in this project.
* void sort\_comb(int\* arr, int size): Comb sort. Similar to bubble sort, but instead of comparing adjacent elements, it compares elements with certain gap between their position and swap them if they are in wrong relative position, then shrink the gap (unless the gap is 1) by certain factor and repeat the procedure. Initial gap of and shrinking factor of 1.3 is used in this project.
* void sort\_oddeven(int\* arr, int size): Odd Even sort. Similar to bubble sort, but it first iterate for all (odd,even) indices, then iterate for all (even,odd) indices, and repeat the procedure.
* void visualize(int\* arr, int size\_arr, int\* color\_index, int color\_size): Update the current state of arr to the UI, and colors indices in color\_index with different colors, and emit changed(int\* arr, int size\_arr, int\* index, int size\_color) signal.
* void changed(int\* arr, int size\_arr, int\* index, int size\_color): A signal to indicate that arr is changed. Used in onNumbersChanged(int\*, int, int\*, int) slot in MainWindow class.
* void done(): A signal to indicate that the sorting is done. Used in onSortingFinished() slot in MainWindow class.

**Data structures used**

There are many types of data structures which can represent a list of numbers (e.g. linked list and array). In this project, we use an array (arr in Sorting class and numbers in Paint class) to represent the list of numbers. to be sorted, instead of using a linked list. We chose to use an array, instead of a linked list, for this project for numerous reasons:

* Faster & easier access to the numbers. We can directly access to the number at specific index if we use an array. If we use a linked list, we must access the numbers sequentially from the beginning of the list. Since we access element of the list many times throughout the sorting procedure, this greatly increase the performance of the sorting.
* Memory-saving. Using an array to represent a list of numbers uses significantly less memory than using a linked list to represent the same list of numbers because linked list requires extra memory to store pointers.
* Size of the list does not change in the sorting procedure. Although the size of the list is changeable, all the changes are made before the sorting begins, thus using a linked list in unnecessary.
* Simplicity in term of coding.

Also, dynamic array is used instead of static array to allow users to choose different sizes of the array for the visualization. Although the size of the array is fixed during the sorting procedure, users can change the size of the array before the sorting procedure begins. For similar reasons, dynamic array (color\_index in Sorting class and colorIndices in Paint class) is used to store the indices of the numbers to be represented using different color in the visualization.

**Conclusion**

To conclude, this project visualizes different sorting algorithms to show users how different sorting algorithms works on different initial conditions of the array. It also shows the number of comparisons and changes made, which sum of them is a measure of time complexity of the algorithm. This is important because many important algorithms require sorting of the data to be done before the algorithms are actually applied. For example, binary searching algorithm requires a sorted list of data before the actual searching begins, greedy algorithm of interval scheduling also requires jobs to be sorted based on their finish time, Kruskal's algorithm on building minimum spanning tree of a weighted graph also requires edges to be sorted based on their weights. To improve the performances of these algorithms, choice of suitable sorting algorithm is necessary.