Activity No. 8

Sorting Algorithms

Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/ 21 / 24**
Section: CPE21S4	**Date Submitted: 10/ 21 / 24**
Name(s): Leoj Jeam B. Tandayu	**Instructor: Engr. Ma. Rizette Sayo**

6. Output

Code + Console Screenshot

```
#include <iostream>
#include <cstdlib>
#include <ctime>

const int size = 30;

void generateRandomArray(int arr[], int size) {
std::srand(std::time(0));
for (int i = 0; i < size; i++) {
arr[i] = std::rand() % 1000;
}
}

void printArray(int arr[], int size) {
for (int i = 0; i < size; i++) {
std::cout << arr[i] << " ";
}
std::cout << std::endl;
}

int main() {
int arr[size];

generateRandomArray(arr, size);

std::cout << "Original Array: ";
printArray(arr, size);
return 0;
}
```

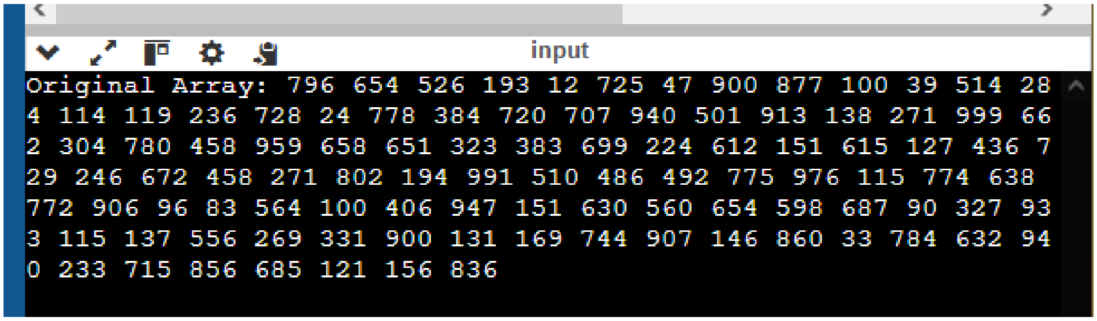
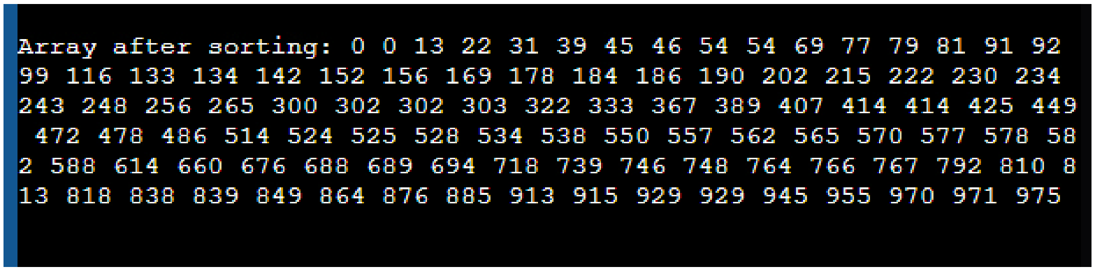
	 input Original Array: 796 654 526 193 12 725 47 900 877 100 39 514 28 4 114 119 236 728 24 778 384 720 707 940 501 913 138 271 999 66 2 304 780 458 959 658 651 323 383 699 224 612 151 615 127 436 7 29 246 672 458 271 802 194 991 510 486 492 775 976 115 774 638 772 906 96 83 564 100 406 947 151 630 560 654 598 687 90 327 93 3 115 137 556 269 331 900 131 169 744 907 146 860 33 784 632 94 0 233 715 856 685 121 156 836
Observation	Just like in the previous lab activity, I used the same code to generate a random 100 elements size array.

Table 8-1. Array of Values for Sort Algorithm Testing

Code + Console Screenshot

```
#include <iostream>
#include <cstdlib>
#include <ctime>

const int size = 100;

void generateRandomArray(int arr[], int size) {
std::srand(std::time(0));
for (int i = 0; i < size; i++) {
arr[i] = std::rand() % 1000;
}
}

void printArray(int arr[], int size) {
for (int i = 0; i < size; i++) {
std::cout << arr[i] << " ";
}
std::cout << std::endl;
}

void shellSort(int arr[], int size) {
for (int gap = size / 2; gap > 0; gap /= 2) {
for (int i = gap; i < size; i++) {
int temp = arr[i];
int j;
for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
arr[j] = arr[j - gap];
}
arr[j] = temp;
}
}
}

int main() {
int arr[size];
```

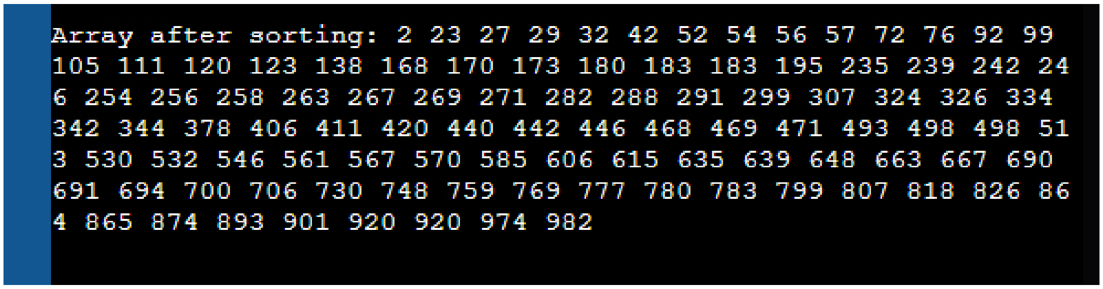
	generateRandomArray(arr, size);
	
	std::cout << "Original Array: ";
	printArray(arr, size);
	
	std::cout<<std::endl;
	std::cout<<std::endl;
	std::cout<<std::endl;
	
	shellSort(arr, size);
	std::cout << "Array after sorting: ";
	printArray(arr, size);
	
	return 0;
	}
	
	
	Array after sorting: 0 0 13 22 31 39 45 46 54 54 69 77 79 81 91 92
	99 116 133 134 142 152 156 169 178 184 186 190 202 215 222 230 234
	243 248 256 265 300 302 302 303 322 333 367 389 407 414 414 425 449
	 472 478 486 514 524 525 528 534 538 550 557 562 565 570 577 578 58
	2 588 614 660 676 688 689 694 718 739 746 748 764 766 767 792 810 8
	13 818 838 839 849 864 876 885 913 915 929 929 945 955 970 971 975

Observation	The code sorts the 100 element array from least to greatest using shell sorting.

Table 8-2. Shell Sort Technique

Code + Console Screenshot	#include <iostream>
	#include <cstdlib>
	#include <ctime>
	
	const int size = 100;
	
	void generateRandomArray(int arr[], int size) {
	std::srand(std::time(0));
	for (int i = 0; i < size; i++) {
	arr[i] = std::rand() % 1000;
	}
	}
	
	void printArray(int arr[], int size) {
	for (int i = 0; i < size; i++) {
	std::cout << arr[i] << " ";
	}
	std::cout << std::endl;
	}

```
void merge(int arr[], int left, int mid, int right) {
int n1 = mid - left + 1;
int n2 = right - mid;
int* L = new int[n1];
int* R = new int[n2];

for (int i = 0; i < n1; i++)
L[i] = arr[left + i];
for (int j = 0; j < n2; j++)
R[j] = arr[mid + 1 + j];

int i = 0;
int j = 0;
int k = left;
while (i < n1 && j < n2) {
if (L[i] <= R[j]) {
arr[k] = L[i];
i++;
} else {
arr[k] = R[j];
j++;
}
k++;
}

while (i < n1) {
arr[k] = L[i];
i++;
k++;
}

while (j < n2) {
arr[k] = R[j];
j++;
k++;
}

delete[] L;
delete[] R;
}

void mergeSort(int arr[], int left, int right) {
if (left < right) {
int mid = left + (right - left) / 2;
mergeSort(arr, left, mid);
mergeSort(arr, mid + 1, right);
merge(arr, left, mid, right);
}
}

int main() {
```

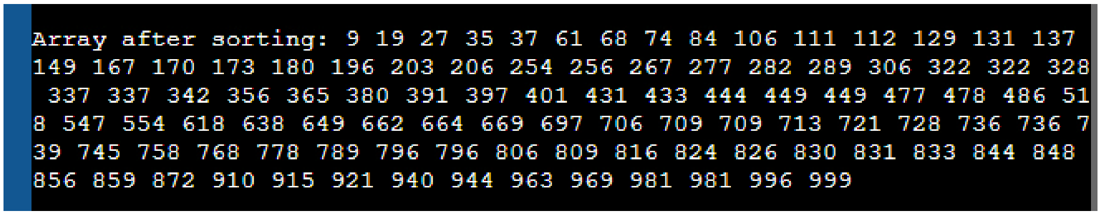
	int arr[size];
	generateRandomArray(arr, size);
	std::cout << "Original Array: ";
	printArray(arr, size);
	std::cout << std::endl;
	mergeSort(arr, 0, size - 1);
	std::cout << "Array after sorting: ";
	printArray(arr, size);
	return 0;
	}
	
	Array after sorting: 2 23 27 29 32 42 52 54 56 57 72 76 92 99
	105 111 120 123 138 168 170 173 180 183 183 195 235 239 242 24
	6 254 256 258 263 267 269 271 282 288 291 299 307 324 326 334
	342 344 378 406 411 420 440 442 446 468 469 471 493 498 498 51
	3 530 532 546 561 567 570 585 606 615 635 639 648 663 667 690
	691 694 700 706 730 748 759 769 777 780 783 799 807 818 826 86
	4 865 874 893 901 920 920 974 982

Observation	The code sorts the 100 element array from least to greatest using merge sorting.

Table 8-3. Merge Sort Algorithm

Code + Console Screenshot	#include <iostream>
	#include <cstdlib>
	#include <ctime>
	
	const int size = 100;
	
	void generateRandomArray(int arr[], int size) {
	std::srand(std::time(0));
	for (int i = 0; i < size; i++) {
	arr[i] = std::rand() % 1000;
	}
	}
	
	void printArray(int arr[], int size) {
	for (int i = 0; i < size; i++) {
	std::cout << arr[i] << " ";
	}
	std::cout << std::endl;
	}
	
	void swap(int& a, int& b) {
	int temp = a;
	a = b;
	b = temp;
	}

```
int partition(int arr[], int low, int high) {
int pivot = arr[high];
int i = low - 1;
for (int j = low; j < high; j++) {
if (arr[j] < pivot) {
i++;
swap(arr[i], arr[j]);
}
}
swap(arr[i + 1], arr[high]);
return i + 1;
}

void quickSort(int arr[], int low, int high) {
if (low < high) {
int pi = partition(arr, low, high);
quickSort(arr, low, pi - 1);
quickSort(arr, pi + 1, high);
}
}

int main() {
int arr[size];
generateRandomArray(arr, size);
std::cout << "Original Array: ";
printArray(arr, size);
std::cout << std::endl;
quickSort(arr, 0, size - 1);
std::cout << "Array after sorting: ";
printArray(arr, size);
return 0;
}
```



```
Array after sorting: 9 19 27 35 37 61 68 74 84 106 111 112 129 131 137
149 167 170 173 180 196 203 206 254 256 267 277 282 289 306 322 322 328
 337 337 342 356 365 380 391 397 401 431 433 444 449 449 477 478 486 51
8 547 554 618 638 649 662 664 669 697 706 709 709 713 721 728 736 736 7
39 745 758 768 778 789 796 796 806 809 816 824 826 830 831 833 844 848
856 859 872 910 915 921 940 944 963 969 981 981 996 999
```

Observation	The code sorts the 100 element array from least to greatest using quick sorting.

Table 8-4. Quick Sort Algorithm

7. Supplementary Activity

Problem 1

- Yes, we can sort the left and right sublists from quick sort using different sorting methods. For example, if we take the list [8, 3, 1, 7, 0, 10, 14] and choose 7 as the pivot, we end up with the left sublist [3, 1, 0] and the right sublist [10, 14]. We could sort the left sublist with insertion sort to get [0, 1, 3] and sort the right sublist with merge sort to get [10, 14]. When we put these sorted lists together with the pivot, we get the final sorted list: [0, 1, 3, 7, 10, 14].

Problem 2

- For the given set of array, quick sort and merge sort are the best for larger set of elements in an array as both have a time complexity of O(N • log N). Quick sort selects a pivot partition in the array and sorts the sublist while merge sort divides the array into smaller parts and then combines them back into a sorted array.

8. Conclusion

This activity showed me another set of sorting techniques which are shell sort, merge sort, and quick sort. Shell sorting is used best for a smaller set of data. Merge sort performs well on larger set of data. Merge sort divides the data into small parts which then is combined after into a sorted order. Lastly, quick sort uses the pivot of the elements and arranging them into 2 sublist which contains the smaller elements an the larger elements and then combined to create the arranged set of data.

9. Assessment Rubric