

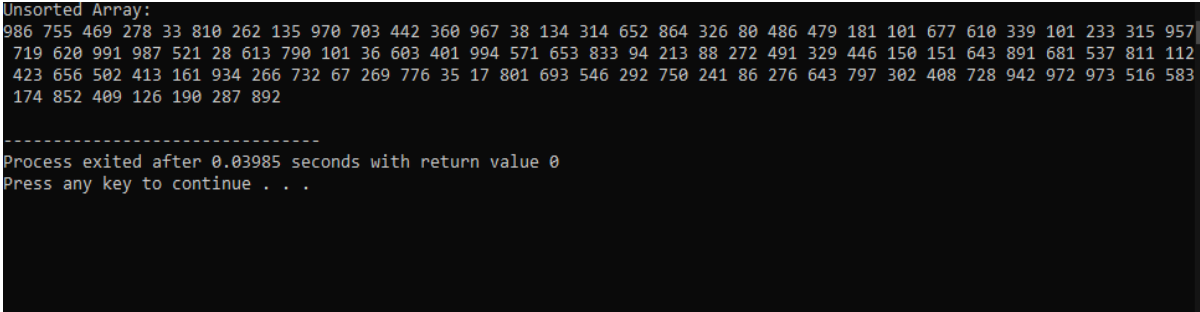
Activity No. 7.2	
Hands-on Activity 7.2 Sorting Algorithms	
Course Code: CPE010	Program: Computer Engineering
Course Title: Data Structures and Algorithms	Date Performed: 21/10/2024
Section: CPE21S4	Date Submitted: 21/10/2024
Name(s): Kenn Jie Valleser	Instructor: Engr. Ma. Rizette Sayo
6. Output	
Code + Console Screenshot	<pre> #include &lt;iostream&gt; #include &lt;ctime&gt;  using namespace std;  void CRandomValues(int arr[], int size) {     for (int i = 0; i &lt; size; ++i) {         arr[i] = rand() % 1000 + 1;     } }  int main() {     srand(static_cast&lt;unsigned int&gt;(time(0)));     const int size = 100;     int randomArray[size];      CRandomValues(randomArray, size);      cout &lt;&lt; "Unsorted Array:\n";     for (int i = 0; i &lt; size; ++i) {         cout &lt;&lt; randomArray[i] &lt;&lt; " ";     }     cout &lt;&lt; endl;      return 0; } </pre>  <pre> Unsorted Array: 986 755 469 278 33 810 262 135 970 703 442 360 967 38 134 314 652 864 326 80 486 479 181 101 677 610 339 101 233 315 957 719 620 991 987 521 28 613 790 101 36 603 401 994 571 653 833 94 213 88 272 491 329 446 150 151 643 891 681 537 811 112 423 656 502 413 161 934 266 732 67 269 776 35 17 801 693 546 292 750 241 86 276 643 797 302 408 728 942 972 973 516 583 174 852 409 126 190 287 892  ----- Process exited after 0.03985 seconds with return value 0 Press any key to continue . . . </pre>
Observation	It uses ctime to be able to create numbers from 1 to 1000 and then when it is executed it displays random values

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

Code +  
Console  
Screenshot

```
#include <iostream>
#include <ctime>
#include "Shell Sort.h"
using namespace std;

void CRandomValues(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % 1000 + 1;
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    const int size = 100;
    int randomArray[size];

    CRandomValues(randomArray, size);

    cout << "Unsorted Array:\n";
    for (int i = 0; i < size; ++i) {
        cout << randomArray[i] << " ";
    }
    cout << endl;

    shellSort(randomArray, size);

    cout << "Sorted Array:\n";
    for (int i = 0; i < size; ++i) {
        cout << randomArray[i] << " ";
    }
    cout << endl;
    return 0;
}

=====
#ifndef SHELL_SORT_H
#define SHELL_SORT_H

void shellSort(int array[], int size) {

    for (int interval = size / 2; interval > 0; interval /= 2) {
        for (int i = interval; i < size; i++) {
            int temp = array[i];
            int j = i;
            while (j >= interval && array[j - interval] > temp) {
                array[j] = array[j - interval];
                j -= interval;
            }
            array[j] = temp;
        }
    }
}
```

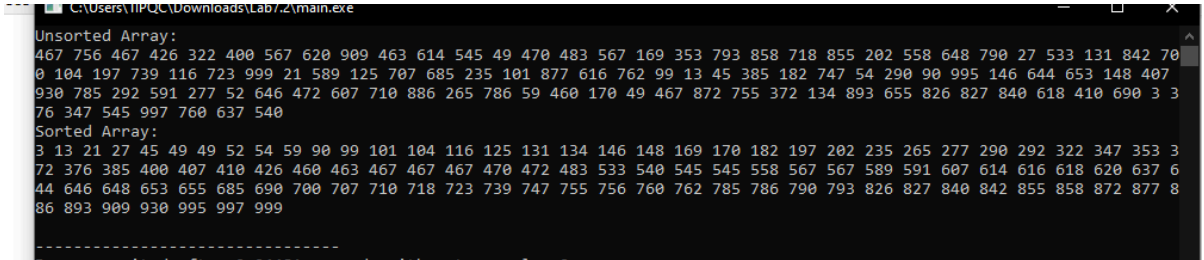
	<pre>         }         array[j] = temp;     } } }  #endif </pre>  <p>Unsorted Array:  467 756 467 426 322 400 567 620 909 463 614 545 49 470 483 567 169 353 793 858 718 855 202 558 648 790 27 533 131 842 70  0 104 197 739 116 723 999 21 589 125 707 685 235 101 877 616 762 99 13 45 385 182 747 54 290 90 995 146 644 653 148 407  930 785 292 591 277 52 646 472 607 710 886 265 786 59 460 170 49 467 872 755 372 134 893 655 826 827 840 618 410 690 3 3  76 347 545 997 760 637 540  Sorted Array:  3 13 21 27 45 49 49 52 54 59 90 99 101 104 116 125 131 134 146 148 169 170 182 197 202 235 265 277 290 292 322 347 353 3  72 376 385 400 407 410 426 460 463 467 467 467 470 472 483 533 540 545 545 558 567 567 589 591 607 614 616 618 620 637 6  44 646 648 653 655 685 690 700 707 710 718 723 739 747 755 756 760 762 785 786 790 793 826 827 840 842 855 858 872 877 8  86 893 909 930 995 997 999</p>
Observation	After applying Shell Sort, the output revealed a sorted array, meaning the algorithm's effectiveness in organizing the data efficiently.

Table 8-2. Shell Sort Technique

Code + Console Screenshot	<pre> #include &lt;iostream&gt; #include &lt;ctime&gt; #include "Merge Sort.h" using namespace std;  void CRandomValues(int arr[], int size) {     for (int i = 0; i &lt; size; ++i) {         arr[i] = rand() % 1000 + 1;     } }  int main() {     srand(static_cast&lt;unsigned int&gt;(time(0)));     const int size = 100;     int randomArray[size];      CRandomValues(randomArray, size);      cout &lt;&lt; "Unsorted Array:\n";     for (int i = 0; i &lt; size; ++i) {         cout &lt;&lt; randomArray[i] &lt;&lt; " ";     }     cout &lt;&lt; endl;      mergeSort(randomArray, 0, size - 1);      cout &lt;&lt; "Sorted Array:\n"; </pre>
---------------------------------	---

```

    for (int i = 0; i < size; ++i) {
        cout << randomArray[i] << " "; // Display all elements in a single line
    }
    cout << endl;

    return 0;
}

```

```

=====
#ifndef MERGE_SORT_H
#define MERGE_SORT_H

```

```

void merge(int array[], int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

```

```

    int* L = new int[n1];
    int* R = new int[n2];

```

```

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

```

```

    int i = 0;
    int j = 0;
    int k = left;

```

```

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

```

```

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

```

```

    while (j < n2) {
        array[k] = R[j];

```

```

7 Unsorted Array:
6 695 481 7 690 446 128 62 318 840 83 261 629 674 833 838 817 653 166 872 185 371 176 664 59 48 241 297 183 511 773 159 11
7 7 330 836 38 318 830 434 440 425 190 325 614 647 495 318 454 497 660 84 479 59 554 173 401 488 999 135 24 560 351 750 71
8 5 31 375 407 847 568 6 860 522 458 944 192 181 385 629 822 281 592 185 373 957 990 318 175 358 935 711 602 521 469 629 7
9 182 933 646 256 648 493 159
10 Sorted Array:
11 3 6 7 24 31 38 48 59 62 83 84 117 128 135 159 159 166 173 175 176 181 183 185 185 190 192 241 256 261 281 297 318 318 3
12 4 18 318 325 330 351 358 371 373 375 385 401 407 425 434 440 446 454 458 469 479 481 488 493 495 497 511 521 522 554 560 5
13 6 68 592 602 614 629 629 629 646 647 648 653 660 664 674 690 695 711 715 750 773 782 817 822 830 833 836 838 840 847 860 8
14 7 72 933 935 944 957 990 999
15 -----
16 Process exited after 0.03984 seconds with return value 0
17 Press any key to continue . . .
18
19
20
21

```

Observation	The sorted array after using Merge Sort confirmed the algorithm's ability to consistently arrange elements in ascending order.
-------------	--

Table 8-3. Merge Sort Algorithm

Code + Console Screenshot	<pre>#include &lt;iostream&gt; #include &lt;ctime&gt; #include "Quick Sort.h" using namespace std;  void CRandomValues(int arr[], int size) {     for (int i = 0; i &lt; size; ++i) {         arr[i] = rand() % 1000 + 1;     } }</pre>
---------------------------------	---

```

}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    const int size = 100;
    int randomArray[size];

    CRandomValues(randomArray, size);

    cout << "Unsorted Array:\n";
    for (int i = 0; i < size; ++i) {
        cout << randomArray[i] << " ";
    }
    cout << endl;

    quickSort(randomArray, 0, size - 1);

    cout << "Sorted Array:\n";
    for (int i = 0; i < size; ++i) {
        cout << randomArray[i] << " ";
    }
    cout << endl;

    return 0;
}
=====
#ifndef QUICK_SORT_H
#define QUICK_SORT_H

void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int partition(int array[], int low, int high) {
    int pivot = array[high];
    int i = low - 1;

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

```

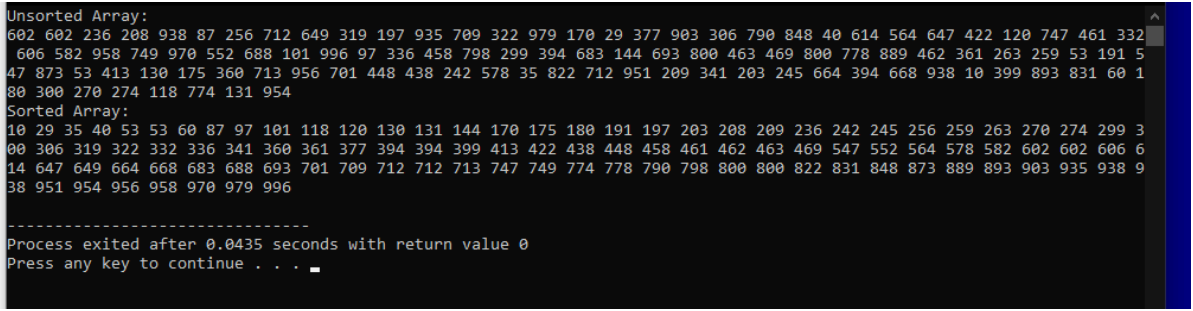
	<pre>void quickSort(int array[], int low, int high) {     if (low &lt; high) {         int pivot = partition(array, low, high);         quickSort(array, low, pivot - 1);         quickSort(array, pivot + 1, high);     } }  #endif</pre>  <pre>Unsorted Array: 602 602 236 208 938 87 256 712 649 319 197 935 709 322 979 170 29 377 903 306 790 848 40 614 564 647 422 120 747 461 332 606 582 958 749 970 552 688 101 996 97 336 458 798 299 394 683 144 693 800 463 469 800 778 889 462 361 263 259 53 191 5 47 873 53 413 130 175 360 713 956 701 448 438 242 578 35 822 712 951 209 341 203 245 664 394 668 938 10 399 893 831 60 1 80 300 270 274 118 774 131 954 Sorted Array: 10 29 35 40 53 53 60 87 97 101 118 120 130 131 144 170 175 180 191 197 203 208 209 236 242 245 256 259 263 270 274 299 3 00 306 319 322 332 336 341 360 361 377 394 394 399 413 422 438 448 458 461 462 463 469 547 552 564 578 582 602 602 606 6 14 647 649 664 668 683 688 693 701 709 712 712 713 747 749 774 778 790 798 800 800 822 831 848 873 889 893 903 935 938 9 38 951 954 956 958 970 979 996 ----- Process exited after 0.0435 seconds with return value 0 Press any key to continue . . .</pre>
Observation	Sorting with Quick Sort, the array was successfully ordered, highlighting the algorithm's performance in handling diverse datasets efficiently.

Table 8-4. Quick Sort Algorithm

<b>7. Supplementary Activity</b>	
Problem 1:	
<p>Yes, we can sort the left and right sublists from the partition method in Quick Sort using other sorting algorithms. For example, if we have the array [3, 6, 8, 10, 1, 2, 1] and use 1 as the pivot, the array might be partitioned into [1] as the left sublist and [3, 2, 6, 8, 10] as the right sublist. We could then sort the right sublist using Merge Sort, which involves dividing it into smaller parts, sorting those, and then merging them back together. This way, even though Quick Sort is used initially, we can still utilize another sorting algorithm to organize the sublists effectively.</p>	
Problem 2:	
<p>For the array {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19, 74}, both Merge Sort and Quick Sort would give the best time performance, each with a complexity of <math>O(N \cdot \log N)</math>. Merge Sort works by dividing the array into smaller parts and then merging them back in a sorted order, while Quick Sort selects a pivot to partition the array and sorts the sublists. Both methods use a divide-and-conquer approach, which allows them to be efficient even for larger datasets.</p>	
<b>8. Conclusion</b>	
<p>In this activity, I learned about Shell Sort, Merge Sort, and Quick Sort, and how each algorithm has unique strengths and use cases. Analyzing the procedures helped me understand that Shell Sort is efficient for smaller or nearly sorted datasets, while Merge Sort consistently performs well on larger arrays due to its stable <math>O(N \cdot \log N)</math> complexity. Quick Sort, although faster in practice, can vary in efficiency based on the choice of pivot. Overall, I feel I did well in grasping these sorting algorithms, but I aim to improve my ability to compare their efficiencies in different scenarios and practice their implementations more frequently.</p>	
<b>9. Assessment Rubric</b>	

