

Activity No. 8	
SORTING ALGORITHMS: SHELL, MERGE, AND QUICK SORT	
Course Code: CPE010	Program: Computer Engineering
Course Title: Data Structures and Algorithms	Date Performed: OCTOBER 21, 2024
Section: CPE21S1	Date Submitted:
Name(s): GASPAS, AARON ROWEN O.	Instructor: MS. MARIA RIZZETE SAYO

6. Output

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

main.cpp

Share

Run

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      const int size = 100;
6      int arr[size];
7
8      srand(time(0));
9
10     for (int i = 0; i < size; ++i) {
11         arr[i] = rand() % 100;
12     }
13     cout << "Unsorted array:" << endl;
14     for (int i = 0; i < size; ++i) {
15         cout << arr[i] << " ";
16     }
17     return 0;
18 }
19
```

Output

[Clear](#)

```
/tmp/SREObRCkpw.o
```

```
Unsorted array:
```

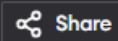
```
22 61 75 69 65 89 76 55 38 84 13 64 82 89 34 40 71 28 44 54 82 69 24 22 67 77 39 40 89 80 67 11 42
 42 32 59 84 60 66 74 44 80 38 79 69 73 19 40 53 15 46 35 84 71 10 51 48 1 43 89 82 11 53 24 5
 85 83 89 46 50 63 42 82 2 21 51 27 93 43 32 8 90 68 93 61 78 96 61 79 92 51 13 3 4 37 60 41 73
 50 39
```

```
=== Code Execution Successful ===
```

OBSERVATION: This code correctly initializes an array of 100 elements, fills it with random values, and prints the unsorted array. This will be used as a constant code in performing the following sorting techniques that is introduced in this module.

Table 8-2. Shell Sort Technique

main.cpp



Run

```
1  #include <iostream>
2  using namespace std;
3
4  template <typename T>
5  void shellSort(T arr[], int size) {
6      for (int interval = size / 2; interval > 0; interval /= 2) {
7          for (int i = interval; i < size; i++) {
8              T temp = arr[i];
9              int j;
10             for (j = i; j >= interval && arr[j - interval] > temp; j -= interval) {
11                 arr[j] = arr[j - interval];
12             }
13             arr[j] = temp;
14         }
15     }
16 }
17
18 int main() {
19     const int size = 100;
20     int arr[size];
21
22     srand(time(0));
23
24     for (int i = 0; i < size; ++i) {
25         arr[i] = rand() % 100;
26     }
27     cout << "Unsorted array:" << endl;
28     for (int i = 0; i < size; ++i) {
29         cout << arr[i] << " ";
30     }
31     cout << "\n\n";
32     shellSort(arr, size);
33     cout << "Sorted array:" << endl;
34     for (int i = 0; i < size; ++i) {
35         cout << arr[i] << " ";
36     }
37
38     return 0;
39 }
```

Output

Clear

/tmp/kEgq14AHXF.o

Unsorted array:

```
41 34 85 49 26 95 97 16 97 82 80 28 67 46 27 7 20 44 0 19 27 94 64 45 49 34 76 32 56 33 82 97 67 67 46 46 14 43
62 11 78 43 39 97 41 19 4 61 63 57 81 42 51 97 40 0 32 16 85 88 50 67 85 69 86 84 15 1 79 78 12 57 21 4 54
14 23 11 76 86 20 9 29 23 6 21 23 38 37 60 78 87 27 64 57 14 0 24 15 79
```





Sorted array:

```
0 0 0 1 4 4 6 7 9 11 11 12 14 14 14 15 15 16 16 19 19 20 20 21 21 23 23 23 24 26 27 27 27 28 29 32 32 33 34 34
37 38 39 40 41 41 42 43 43 44 45 46 46 46 49 49 50 51 54 56 57 57 57 60 61 62 63 64 64 67 67 67 67 69 76 76
78 78 78 79 79 80 81 82 82 84 85 85 85 86 86 87 88 94 95 97 97 97 97 97
```

=== Code Execution Successful ===

OBSERVATION: The code executes the shell sort. The sorting will be based on the gap between the indexes. The choice of gap sequence can be tailored for specific datasets, with some sequences yielding better performance than others. Sorting elements far apart first helps to reduce large scale disorder quickly, making the array easier to sort overall.

Table 8-3. Merge Sort Algorithm

```
main.cpp    Share  Run
```

```
1  #include <iostream>
2  using namespace std;
3
4  void merge(int arr[], int l, int m, int r) {
5      int n1 = m - l + 1;
6      int n2 = r - m;
7
8      int* leftArr = new int[n1];
9      int* rightArr = new int[n2];
10
11     for (int i = 0; i < n1; ++i)
12         leftArr[i] = arr[l + i];
13     for (int j = 0; j < n2; ++j)
14         rightArr[j] = arr[m + 1 + j];
15
16     int i = 0, j = 0, k = l;
17     while (i < n1 && j < n2) {
18         if (leftArr[i] <= rightArr[j]) {
19             arr[k] = leftArr[i];
20             i++;
21         } else {
22             arr[k] = rightArr[j];
23             j++;
24         }
25         k++;
26     }
27
28     while (i < n1) {
29         arr[k] = leftArr[i];
30         i++;
31         k++;
32     }
33
34     while (j < n2) {
35         arr[k] = rightArr[j];
36         j++;
37         k++;
38     }
39
40     delete[] leftArr;
41     delete[] rightArr;
42 }
```

```

43
44 void mergeSort(int arr[], int l, int r) {
45     if (l >= r)
46         return;
47
48     int m = l + (r - l) / 2;
49
50     mergeSort(arr, l, m);
51     mergeSort(arr, m + 1, r);
52
53     merge(arr, l, m, r);
54 }
55
56 int main() {
57     const int size = 100;
58     int arr[size];
59
60     srand(time(0));
61
62     for (int i = 0; i < size; ++i) {
63         arr[i] = rand() % 100;
64     }
65     cout << "Unsorted array:" << endl;
66     for (int i = 0; i < size; ++i) {
67         cout << arr[i] << " ";
68     }
69     cout << "\n\n";
70     mergeSort(arr, 0, size - 1);
71
72
73     cout << "Sorted array:" << endl;
74     for (int i = 0; i < size; ++i) {
75         cout << arr[i] << " ";
76     }
77
78     return 0;
79 }

```

Output

Clear

/tmp/lCngnoLaLT.o

Unsorted array:

```

67 37 5 36 99 81 87 2 97 84 4 32 71 44 88 58 75 62 86 87 60 78 0 27 63 46 33 34 91 3 65 58 92 23
47 44 4 34 46 53 70 2 37 94 46 25 4 73 87 91 12 99 21 12 26 36 11 59 70 2 14 87 13 7 10 60 51
66 46 49 19 16 3 9 62 1 34 67 74 74 10 38 73 31 3 0 67 66 11 37 20 78 76 33 85 87 93 88 5 39

```

Sorted array:

```

0 0 1 2 2 2 3 3 3 4 4 4 5 5 7 9 10 10 11 11 12 12 13 14 16 19 20 21 23 25 26 27 31 32 33 33 34 34
34 36 36 37 37 37 38 39 44 44 46 46 46 46 47 49 51 53 58 58 59 60 60 62 62 63 65 66 66 67 67
67 70 70 71 73 73 74 74 75 76 78 78 81 84 85 86 87 87 87 87 87 88 88 91 91 92 93 94 97 99 99

```

=== Code Execution Successful ===

OBSERVATION: First, it splits the array into two halves until each sub-array has only one element. Next, it will recursively sort the two halves. Initially, since each sub-array has only one element, they are already sorted. As the recursion unwinds, it starts merging these sorted arrays back together. Last, It will merge the two sorted halves to produce a sorted array.

Table 8-4. Quick Sort Algorithm

main.cpp



Share

Run

```
1  #include <iostream>
2  using namespace std;
3
4  int partition(int arr[], int low, int high) {
5      int pivot = arr[high];
6      int i = low - 1;
7
8      for (int j = low; j < high; ++j) {
9          if (arr[j] <= pivot) {
10             i++;
11             swap(arr[i], arr[j]);
12         }
13     }
14     swap(arr[i + 1], arr[high]);
15     return i + 1;
16 }
17
18 void quickSort(int arr[], int low, int high) {
19     if (low < high) {
20         int pivot = partition(arr, low, high);
21         quickSort(arr, low, pivot - 1);
22         quickSort(arr, pivot + 1, high);
23     }
24 }
```

```

25
26 int main() {
27     const int size = 100;
28     int arr[size];
29
30     srand(time(0));
31
32     for (int i = 0; i < size; ++i) {
33         arr[i] = rand() % 100;
34     }
35     cout << "Unsorted array:" << endl;
36     for (int i = 0; i < size; ++i) {
37         cout << arr[i] << " ";
38     }
39     cout << "\n\n";
40     quickSort(arr, 0, size - 1);
41
42
43     cout << "Sorted array:" << endl;
44     for (int i = 0; i < size; ++i) {
45         cout << arr[i] << " ";
46     }
47
48     return 0;
49 }

```

Output

Clear

/tmp/WGSEmtbNet.o

Unsorted array:

```

67 58 18 40 0 49 60 28 14 43 27 35 57 64 54 45 50 26 47 79 23 78 36 41 37 22 78 99 45 36 91 65 46
61 57 47 10 17 27 25 12 55 12 70 71 66 15 21 92 62 0 68 92 89 9 81 63 88 80 9 76 72 26 23 33
83 70 96 0 97 73 13 52 85 35 76 51 2 97 96 17 50 64 9 39 73 91 2 13 71 63 42 95 41 65 81 24 35
29 77

```

Sorted array:

```

0 0 0 2 2 9 9 9 10 12 12 13 13 14 15 17 17 18 21 22 23 23 24 25 26 26 27 27 28 29 33 35 35 35 36
36 37 39 40 41 41 42 43 45 45 46 47 47 49 50 50 51 52 54 55 57 57 58 60 61 62 63 63 64 64 65
65 66 67 68 70 70 71 71 72 73 73 76 76 77 78 78 79 80 81 81 83 85 88 89 91 91 92 92 95 96 96
97 97 99

```

=== Code Execution Successful ===

OBSERVATION: The partition function correctly selects a pivot (last element) and rearranges the array elements based on the pivot. The quicksort function effectively divides the array into subarrays around the pivot and recursively sorts them. The main function generates a random array and initializes the seed using `srand(time(0))` for varied results on each run. Both unsorted and sorted arrays are printed.

7. Supplementary Activity

main.cpp



Share

Run

```
1  #include <iostream>
2  using namespace std;
3
4  int partition(int arr[], int low, int high) {
5      int pivot = arr[high];
6      int i = low - 1;
7
8      for (int j = low; j < high; ++j) {
9          if (arr[j] <= pivot) {
10             i++;
11             swap(arr[i], arr[j]);
12         }
13     }
14     swap(arr[i + 1], arr[high]);
15     return i + 1;
16 }
17
18 void quickSort(int arr[], int low, int high) {
19     if (low < high) {
20         int pivot = partition(arr, low, high);
21         quickSort(arr, low, pivot - 1);
22         quickSort(arr, pivot + 1, high);
23     }
24 }
25
26 void printArray(int arr[], int size) {
27     for (int i = 0; i < size; ++i) {
28         cout << arr[i] << " ";
29     }
30     cout << endl;
31 }
32
33 int main() {
34     int arr[] = {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19, 74};
35     int size = sizeof(arr) / sizeof(arr[0]);
36
37     cout << "Original array:" << endl;
38     printArray(arr, size);
39
40     quickSort(arr, 0, size - 1);
41
42
43     cout << "Sorted array:" << endl;
44     printArray(arr, size);
45
46     return 0;
47 }
```

Output

Clear

/tmp/SLAFIFtBys.o

Original array:

4 34 29 48 53 87 12 30 44 25 93 67 43 19 74

Sorted array:

4 12 19 25 29 30 34 43 44 48 53 67 74 87 93

=== Code Execution Successful ===

8. Conclusion

Learning sorting techniques like Bubble Sort, Selection Sort, Insertion Sort, Shell Sort, and Merge Sort brings fundamental insights into the world of algorithms and data manipulation. Each of these sorting methods equips you with unique strategies to order data efficiently, catering to different scenarios and data sets. By mastering these techniques, you not only enhance your problem-solving skills but also gain a robust toolkit to tackle various data ordering challenges in software development. The journey through sorting algorithms opens up a broader understanding of efficiency, optimization, and algorithmic thinking—a cornerstone of effective programming.

9. Assessment Rubric