Bautista,Mariela S.
CPE21S4

Assignment 8.1 Using Sorting Algorithms 2

Objectives: To create a program in C++ using algorithms.

1. To be familiarized with the sorting algorithms.
2. To be able to differentiate the types of sorting algorithms.
3. To be able to create a program with sorting algorithms.

Answer the following questions:

1. Explain the quick sort, shell sort, and merge sort types of sorting algorithms.

   Quick Sort - This is a divide-and-conquer algorithm. For me, this feels like cutting the problem into smaller parts until everything is easy to handle. I pick a pivot element, then divide the array into numbers smaller than the pivot and numbers bigger than it. After that, I'll sort each side again by applying the same process. It is fast because it reduces the big problem into many smaller ones instead of sorting everything at once. Hence the word 'quick' sort, because this is much faster than the other sorting algorithms.

   Shell Sort - This is basically the improved version of insertion sort. Instead of only comparing adjacent element, it starts by comparing numbers that are far apart using a gap sequence. This usually begin with a big gap, like half the array size, and sort elements that are that distance away. As the gap decreases, the array becomes more and more organized and sorted until the gap is just 1. At that point, it turns into a simple insertion sort, but the list is already almost sorted.

   Merge Sort - is also a divide-and-conquer algorithm taht works by splitting the array into halves until single elements remain. Then, it carefully merges those smaller parts back together in the correct manner or order. This process repeats until the entire array is fully sorted. What makes it reliable is that it always runs in O(n log n) time, even in the worst case. The only drawback is that it needs extra space for merging, unlike the quick sort which sorts in place.

Bautista,Mariela S.
CPE21S4

2. Give simple sample programs in C++ that uses the above sorting algorithms. Use a user input of 10 integer values in your example elements in an array to be sorted. Explain how the program works.

Quick Sort

Code:

CMakeLists.txt × main.cpp ×

```cpp
// C++ Program to demonstrate how to implement the quick
// sort algorithm
#include <bits/stdc++.h>
using namespace std;

int partition(vector<int> &vec, int low, int high) {

    // Selecting last element as the pivot
    int pivot = vec[high];

    // Index of elemment just before the last element
    // It is used for swapping
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {

        // If current element is smaller than or
        // equal to pivot
        if (vec[j] <= pivot) {
            i++;
            swap( a: [&] vec[i], b: [&] vec[j]);
        }
    }

    // Put pivot to its position
    swap( a: [&] vec[i + 1], b: [&] vec[high]);

    // Return the point of partition
    return (i + 1);
}
```

Bautista,Mariela S.
CPE21S4

```cpp
void quickSort(vector<int> &vec, int low, int high) {

    // Base case: This part will be executed till the starting
    // index low is lesser than the ending index high
    if (low < high) {

        // pi is Partitioning Index, arr[p] is now at
        // right place
        int pi = partition( [&] vec, low, high);

        // Separately sort elements before and after the
        // Partition Index pi
        quickSort( [&] vec, low,  high: pi - 1);
        quickSort( [&] vec,  low: pi + 1, high);
    }
}

int main() {
    vector<int> vec = {10, 7, 8, 9, 1, 5};
    int n = vec.size();

    // Calling quicksort for the vector vec
    quickSort( [&] vec,  low: 0,  high: n - 1);

    for (auto i :int  : vec) {
        cout << i << " ";
    }
    return 0;
}
```

Output:

```
Run    quick_sortttt
"C:\Users\Mariela\CLionProjects\quick sortttt\cmake-build-debug\quick_sortttt.exe"
1 5 7 8 9 10
Process finished with exit code 0
```

Bautista,Mariela S.
CPE21S4

Analysis:

What ive observed in the program is that i noticed that the partition function is really the core of the algorithm. It picks the last element of the array as the pivot and then rearranges the other elements so that anything smaller or equal goes to the left, and the bigger ones stay on the right side. The swapping process makes sure the pivot ends up in its sorted position. After that, the quickSort() function takes over by breaking the array into smaller subarrays around the pivot. I realize that it keeps calling itself repeatedly, first we operate the left side, then the right side, until all parts are sorted. This divide-and-conquer approach makes the sorting efficient because it sorts sections individually. When i ran the program, i see the array {10,7,8,9,1,5} get broken down step by step, and after quick sort finishes, it outputs the sorted sequence: 1 5 7 8 9 10 . Observing the flow, its clear taht the efficiency comes from how the pivot divides the work and how recursion handles the sorting in smaller pieces.

Shell Sort

Code:

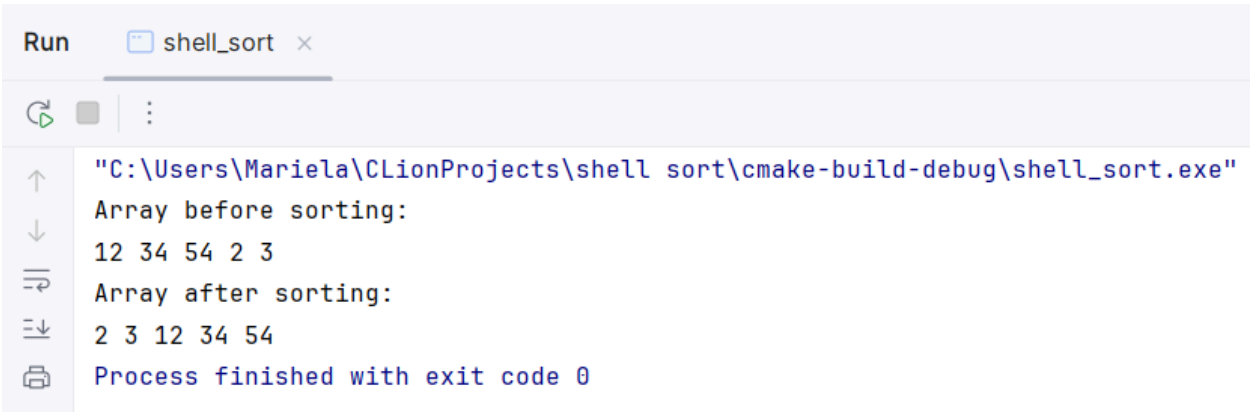
CMakeLists.txt main.cpp

```cpp
// C++ implementation of Shell Sort
#include  <iostream>
using namespace std;

/* function to sort arr using shellSort */
int shellSort(int arr[], int n)
{
    // Start with a big gap, then reduce the gap
    for (int gap = n/2; gap > 0; gap /= 2)
    {
        // Do a gapped insertion sort for this gap size.
        // The first gap elements a[0..gap-1] are already in gapped order
        // keep adding one more element until the entire array is
        // gap sorted
        for (int i = gap; i < n; i += 1)
        {
            // add a[i] to the elements that have been gap sorted
            // save a[i] in temp and make a hole at position i
            int temp = arr[i];

            // shift earlier gap-sorted elements up until the correct
            // location for a[i] is found
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];

            //  put temp (the original a[i]) in its correct location
            arr[j] = temp;
        }
    }
    return 0;
```

Bautista,Mariela S.
CPE21S4

```
32      }
33
34      void printArray(int arr[], int n)
35      {
36          for (int i=0; i<n; i++)
37              cout << arr[i] << " ";
38      }
39
40      int main()
41      {
42          int arr[] = {12, 34, 54, 2, 3}, i;
43          int n = sizeof(arr)/sizeof(arr[0]);
44
45          cout << "Array before sorting: \n";
46          printArray(arr, n);
47
48          shellSort(arr, n);
49
50          cout << "\nArray after sorting: \n";
51          printArray(arr, n);
52
53          return 0;
54      }
```

Output:



Bautista,Mariela S.
CPE21S4

Analysis:

What ive observed in this Shell Sort codew, it works by starting with a big gap between compared elements, then keeps cutting the gap in half until it becomes 1. In each pass, it takes the current element which can also be called as “temp” and shifts bigger elements forward by the gap until “temp” can be placed in the right spot. The printArray() function is just there to show what the array looks like before and after sorting, while the tests the algorithm using {12, 34, 54, 2, 3}. I noticed that the early passes with big gaps quickly make the array more organized, and the final pass with a gap of 1 finishes the sorting easily. For me, the cool part is how a small change adding a gap makes the insertion sort much faster. It shows me that even simple tweaks in an algorithm can really improve efficiency and performance.

Merge Sort

Code:

CMakeLists.txt main.cpp

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Merges two subarrays of arr[].
// First subarray is arr[left..mid]
// Second subarray is arr[mid+1..right]
void merge(vector<int>& arr, int left,
                     int mid, int right){

    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temp vectors
    vector<int> L(n1), R(n2);

    // Copy data to temp vectors L[] and R[]
    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, j = 0;
    int k = left;

    // Merge the temp vectors back
    // into arr[left..right]
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
```

Bautista,Mariela S.
CPE21S4

```
            }
            else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        // Copy the remaining elements of L[],
        // if there are any
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        // Copy the remaining elements of R[],
        // if there are any
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
```

Bautista,Mariela S.
CPE21S4

```
56
57      // begin is for left index and end is right index
58      // of the sub-array of arr to be sorted
59      void mergeSort(vector<int>& arr, int left, int right){
60
61          if (left >= right)
62              return;
63
64          int mid = left + (right - left) / 2;
65          mergeSort( [&] arr, left,  right: mid);
66          mergeSort( [&] arr,  left: mid + 1, right);
67          merge( [&] arr, left, mid, right);
68      }
69
70      // Driver code
71      int main(){
72
73          vector<int> arr = {38, 27, 43, 10};
74          int n = arr.size();
75
76          mergeSort( [&] arr,  left: 0,  right: n - 1);
77          for (int i = 0; i < arr.size(); i++)
78              cout << arr[i] << " ";
79          cout << endl;
80
81          return 0;
82      }
```

Output:

```
Run        mergee_sort  ×
"C:\Users\Mariela\CLionProjects\mergee sort\cmake-build-debug\mergee_sort.exe"
10 27 38 43

Process finished with exit code 0
```

Bautista,Mariela S.
CPE21S4

Analysis:

Looking at both merge sort and quick sort, I cant help but to notice how different their personalities are even though they share the same goal of putting an array into order. Merge sort approaches the tasks thorougly, breaking the array into smaller halves through the mergeSort() function until each piece is just a single element, which doesnt need sorting, and then carefully putting them back together with the merge() function. I also like how it creates a temporary vectors for the left and right halves, making the comparisons straightforward, and the leftover loops at the end ensure that nothing is skipped. Tracing the example {38, 27, 43, 10}, I saw how it split into [38, 27] and [43, 10], broke themm into singles, then rebuilt them step by step until the array became [10, 27, 38, 43]. Quick sort, on the other hand, feels sharper and more decisive: it picks a pivot (the last element here) and rearranges the array so that smaller elements go to the left and larger ones to the right, repeating the process until everything is in place. Using the same array, the pivot 10 immediately engraved the array into [10, 27, 43, 38], and the recursion carried the rest. What makes this stand out is that merge sort is already and consistent with its O(n log n) runtime but spends extra memory, while quick sort is space-efficient and oftehn faster in practice but can slow down to O(n^2) if the pivots are unfortunate. To me, merge sort feels like carefully dissembling Lego blocks and rebuilding them in order, while quick sort feels like slicing clean cuts until the pieces naturally fall into the line, two different approaches, both dependable in their own way.

REFERENCES OF THE CODES:

GeeksforGeeks. (2025, July 23). *Shell Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/shell-sort/ GeeksforGeeks

GeeksforGeeks. (2025, September 23). *Merge Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/merge-sort/ GeeksforGeeks

GeeksforGeeks. (2025, July 23). *C++ Program for Quick Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/cpp/cpp-program-for-quicksort/ GeeksforGeeks