Course Title: CS350 Student Name: Sean Poston SEMO ID#: S01882786
Student Email: sposton1s@semo.edu Date: 11/4/2021

Codes

Naturally, I had to look up the algorithms for each of these. You will find the citation for each algorithm within the main function of each sort.

Radix

```
#include <stdio.h>

void radix_sort(int arr[], int n) {
  // Get max elem of array
  int max = arr[0];

  for(int i = 1;i < n; i++) {
    if(max < arr[i]) {
      max = arr[i];
    }
  }

  // Calculate how many digits there are
  int digits = 0;

  while(max > 0) {
    max /= 10;
    digits++;
  }

  // Digits place
  int power = 1;

  // Arrange numbers based on digit places.
  for(int i = 0; i < digits; i++) {
    int new_array[n];

    int count[10];

    for(int j = 0; j < 10; j++) {
      count[j] = 0;
    }

    // Calculating frequency of digits
    for(int j = 0; j < n; j++) {
      int num = (arr[j] / power) % 10;
      count[num]++;
```

```
    }

    // Cumulative frequency of count array
    for(int j = 1; j < 10; j++) {
      count[j] += count[j-1];
    }

    // Designating new positions in the updated array
    for(int j = n - 1; j >= 0; j--) {
        int num = (arr[j] / power) % 10;
        new_array[count[num] - 1] = arr[j];
        count[num]--;
    }

    // Updating the original array using New Array
    for(int j = 0; j < n; j++) {
      arr[j] = new_array[j];
    }

    // Updating the digit to be considered next iteration
    power *= 10;
  }
}

int main() {
  // Radix Sort Citation: https://www.journaldev.com/42955/radix-sort-algorithm
  int arr[] = {24, 567, 23, 99, 01, 2, 50};
  int arr_len = sizeof(arr) / sizeof(arr[0]);
  radix_sort(arr, arr_len);

  for(int i = 0; i < arr_len; i++) {
    printf("%d ", arr[i]);
  }
}
```

Merge

```
#include <stdio.h>
#include <stdlib.h>

void merge(int arr[], int l, int m, int r) {
  // Merge two arrays in sorted order
  int i, j, k;

  // Length of temp arrays
  int n1 = m - l + 1;
  int n2 = r - m;

  // Initalize temp arrays and copy data from subarrays over to them
  int tempL[n1];
  int tempR[n2];
  for (i = 0; i < n1; i++) {
    tempL[i] = arr[l + i];
  }
  for (j = 0; j < n2; j++) {
    tempR[j] = arr[m + 1 + j];
  }

  // Initial indices of first subarr, second subarr, and merged subarr
  i = 0;
  j = 0;
  k = l;

  // Merge tempL and tempR back into arr
  while (i < n1 && j < n2) {
    if (tempL[i] <= tempR[j]) {
      arr[k] = tempL[i];
      i++;
    }
    else {
      arr[k] = tempR[j];
      j++;
    }
    k++;
  }

  // Copy remaining elems of tempL
  while (i < n1) {
    arr[k] = tempL[i];
```

```
    i++;
    k++;
  }

  // Copy remaining elems of tempR
  while (j < n2) {
    arr[k] = tempR[j];
    j++;
    k++;
  }
}

void merge_sort(int arr[], int l, int r) {
  if (l < r) {
    // Get midsection of array, split into halves, and merge back together
sorted.
    int m = l + (r - l) / 2;
    merge_sort(arr, l, m);
    merge_sort(arr, m + 1, r);
    merge(arr, l, m, r);
  }
}

int main() {
  // Merge Sort Citation: https://www.geeksforgeeks.org/merge-sort/
  int arr[] = {24, 567, 23, 99, 1, 2, 50};
  int arr_len = sizeof(arr) / sizeof(arr[0]);
  merge_sort(arr, 0, arr_len - 1);

  for (int i = 0; i < arr_len; i++) {
    printf("%d ", arr[i]);
  }
}
```

merge_sort (arr, 0, 6)
0 < 6
m = 0 + (6 - 1) / 2 = 5/2 = 2
merge_sort (arr, 0, 2)
merge_sort (arr, 3, 6)
0 < 2
m = 0 + (2 - 1) / 2 = 1/2 = 0
Found the error!

It took me a long time to find my error on merge_sort. I started tracing the program until it popped up. That second midpoint should have been 1, but I was subtracting 1 from the right value instead of subtracting L.

Selection

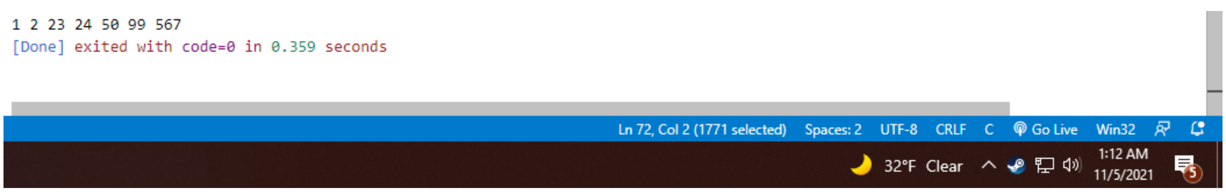
```c
#include <stdio.h>

void swap(int arr[], int i, int j) {
  // Takes the two indices of the array to swap.
  int temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}

void selection_sort(int arr[], int n) {
  int minimum;
  int min_ind;

  // Use the nested loop to loop through the array, finding the minimum value
each time.
  // Swap this minimum value with the pivot point of i, and the array will be
sorted by the end.
  for (int i = 0; i < n - 1; i++) {
    minimum = arr[i];
    min_ind = i;

    for (int j = i; j < n; j++) {
      if (arr[j] < minimum) {
        minimum = arr[j];
        min_ind = j;
      }
    }

    swap(arr, i, min_ind);
  }
}

int main() {
  // Reference Material: https://www.geeksforgeeks.org/selection-sort/

  int arr[] = {24, 567, 23, 99, 1, 2, 50};
  int arr_len = sizeof(arr) / sizeof(arr[0]);

  selection_sort(arr, arr_len);
```

```
  for (int i = 0; i < arr_len; i++) {
    printf("%d ", arr[i]);
  }
}
```

Heap

```
#include <stdio.h>

int max(int i, int j, int k) {
  // Return max element of three given
  if (i >= j && i >= k) {
    return i;
  }
  else if (j >= i && j >= k) {
    return j;
  }
  else if (k >= i && k >= j) {
    return k;
  }
  else {
    return i;
  }
}

void swap(int arr[], int i, int j) {
  // Takes the two indices of the array to swap.
  int temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}

void heapify(int arr[], int i, int n) {
  // Set indices using heap child algorithm
  int left_child = 2 * i + 1;
  int right_child = 2 * i + 2;
  int max_index = i;

  // Set max based on which is largest out of child and root
  if (left_child < n && arr[left_child] > arr[max_index]) {
    max_index = left_child;
  }
  if (right_child < n && arr[right_child] > arr[max_index]) {
    max_index = right_child;
  }

  // If root isn't the largest, swap it with the child and run heapify on the
child.
  if (i != max_index) {
```

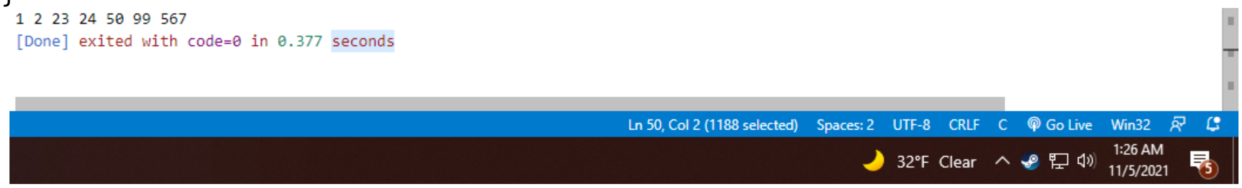
```
    swap(arr, i, max_index);
    heapify(arr, max_index, n);
  }
}

void heap_sort(int arr[], int n) {
  // Build the heap to be able to heap sort
  for (int i = n / 2 - 1; i >= 0; i--) {
    heapify(arr, i, n);
  }

  // Remove one element from the heap by moving the root to the end, then
  // running heapify on the newly shrunk heap.
  for (int i = n - 1; i > 0; i--) {
    swap(arr, 0, i);
    heapify(arr, 0, i);
  }
}

int main() {
  // Heap Sort Citation: https://www.geeksforgeeks.org/heap-sort/
  //                     https://www.mygreatlearning.com/blog/heap-sort/
  int arr[] = {24, 567, 23, 99, 1, 2, 50};
  int arr_len = sizeof(arr) / sizeof(arr[0]);

  heap_sort(arr, arr_len);

  for (int i = 0; i < arr_len; i++) {
    printf("%d ", arr[i]);
  }
}
```



Quick

```
#include <stdio.h>

void swap(int arr[], int i, int j) {
  // Takes the two indices of the array to swap.
  int temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}

int partition(int arr[], int low, int high) {
  // Set pivot to highest elem in subarray
  // Pivot will be placed at right position
  int pivot = arr[high];

  // Will indicate right position of pivot found
  int r = (low - 1);

  // If the current element in the loop is smaller than the pivot, increment r to
show
  for (int i = low; i <= high - 1; i++) {
    if (arr[i] < pivot) {
      r++;
      swap(arr, i, r);
    }
  }

  swap(arr, r + 1, high);
  return (r + 1);
}

void quick_sort(int arr[], int low, int high) {
  if (low < high) {
    // Get index to split array
    int pi = partition(arr, low, high);

    quick_sort(arr, low, pi - 1);
    quick_sort(arr, pi + 1, high);
  }
}

int main() {
  // Quick Sort Reference: https://www.geeksforgeeks.org/quick-sort/
```

```
  int arr[] = {24, 567, 23, 99, 1, 2, 50};
  int arr_len = sizeof(arr) / sizeof(arr[0]);

  quick_sort(arr, 0, arr_len - 1);

  for (int i = 0; i < arr_len; i++) {
    printf("%d ", arr[i]);
  }
}
```



Quick Sort Pseudo Code

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element and indicates the
                   // right position of pivot found so far

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Figure 1: https://www.geeksforgeeks.org/quick-sort/

Analysis

Radix

Radix sort will always run in the same complexity given any input. This is because for each integer input, it will run through the loop (*n*) and through each of the digits for each element (*k*).

$\theta(nk)$

$\Omega(nk)$

$O(nk)$

Merge

Merge sort also runs the same for any input. Since the initial input is subdivided multiple times, it ends up being a logarithmic time. However, since all these singular subarrays must be reconstituted into the final sorted array, it ends up taking linear time to do that operation. Thus, breaking the array apart and putting it back together is always the same time complexity.

$\theta(n\, log(n))$

$\Omega(n \log{(n)})$

$O(n\, log(n))$

Selection

This algorithm will *always* run through the array twice the first time. When you amortize the time of each subsequent run (n, n – 1, n – 2, n – 3…), it averages to n / 2 times, which is still counted as n. This means that the time is n^2 for every case.

$\theta(n^2)$

$\Omega(n^2)$

$O(n^2)$

Heap

Since the heapify() method will run down one sub tree while the largest node is not root, it is considered *log(n)* time (Heap Sort). In the heap_sort() method, it will loop through the entire array one by one, making it *n* time. Multiplying the two gives *n log(n)*.

$\theta(n\ log(n))$

$\Omega(n\ log(n))$

$O(n\ log(n))$

Quick

Quicksort will, like merge sort, subdivide the array in *log(n)* time and then put it back together in *n* time. However, the worst case of this will occur when (in my case, I chose the pivot to the right) the array is already sorted in descending order (It would be opposite if the pivot were to the left) (Datta). This will cause the subarray from the chosen pivot to only contain one element, which will require *n* operations to subdivide the array, followed by another *n* operations to re-stitch it together.

$\theta(n\ log(n))$

$\Omega(n\ log(n))$

$O(n^2)$

Big O Cheat Sheet

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Figure 2: https://www.bigocheatsheet.com/

This is a very good site to check out for almost anything Big O.

References

Heap Sort. https://www.geeksforgeeks.org/heap-sort/

Datta, Subham. https://www.baeldung.com/cs/quicksort-time-complexity-worst-case

Code references are in the main functions of each code.