CH08-320143: Assignment #7

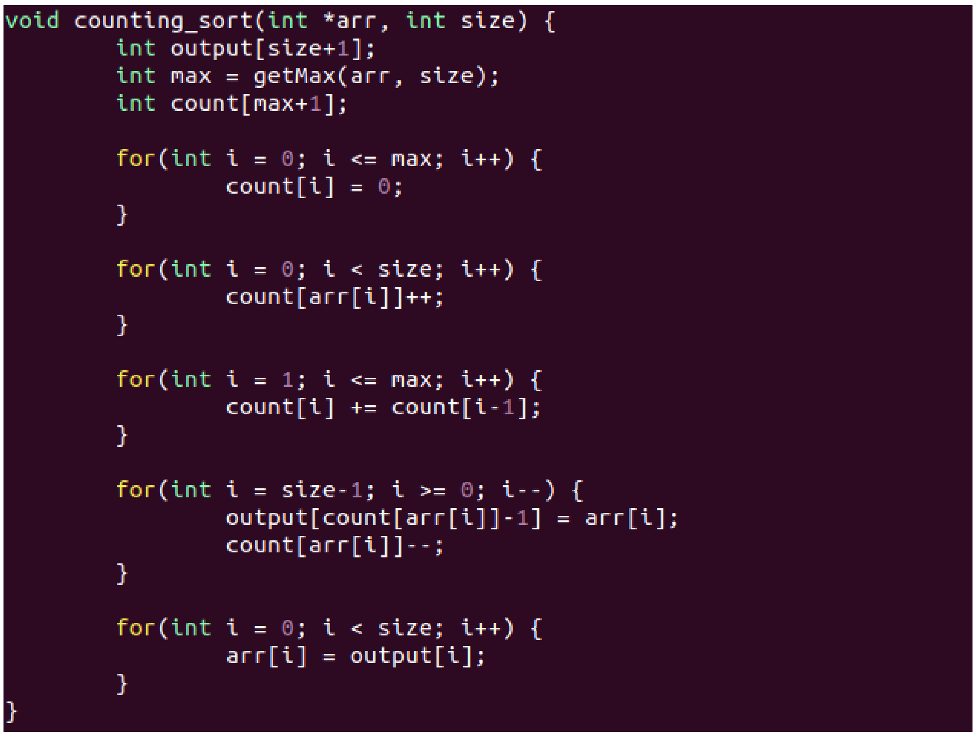
Harit Krishan
h.krishan@jacobs-university.de

Jacobs University Bremen — March 22, 2020

1 Sorting in Linear Time

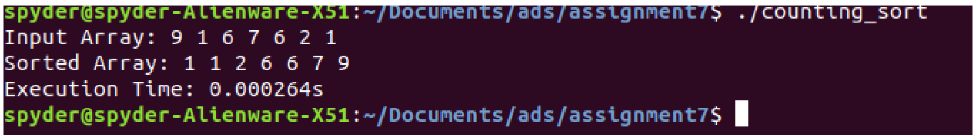
1.1 Counting Sort

In Figure 1, we can see the implementation of counting sort in C++. Figure 2 shows the driver window of the execution of the algorithm. The console output displays the input sequence, the sorted sequence and the time taken for the algorithm to sort the input sequence.



```
void counting_sort(int *arr, int size) {
        int output[size+1];
        int max = getMax(arr, size);
        int count[max+1];

        for(int i = 0; i <= max; i++) {
                count[i] = 0;
        }

        for(int i = 0; i < size; i++) {
                count[arr[i]]++;
        }

        for(int i = 1; i <= max; i++) {
                count[i] += count[i-1];
        }

        for(int i = size-1; i >= 0; i--) {
                output[count[arr[i]]-1] = arr[i];
                count[arr[i]]--;
        }

        for(int i = 0; i < size; i++) {
                arr[i] = output[i];
        }
}
```

Figure 1: Implementation of counting sort in C++



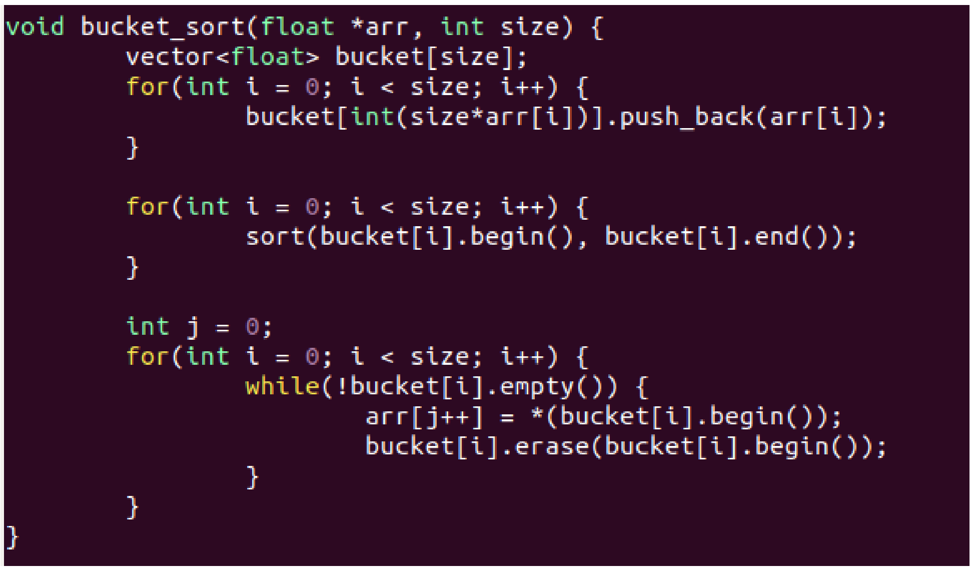
```
spyder@spyder-Alienware-X51:~/Documents/ads/assignment7$ ./counting_sort
Input Array: 9 1 6 7 6 2 1
Sorted Array: 1 1 2 6 6 7 9
Execution Time: 0.000264s
spyder@spyder-Alienware-X51:~/Documents/ads/assignment7$
```

Figure 2: Counting Sort Algorithm Execution

The algorithm functions using three different arrays in theory. We have the input array, an auxiliary storage array and the array holding the elements in sorted order. These arrays are called arr, $count$ and $output$ respectively in reference to Figure 1.

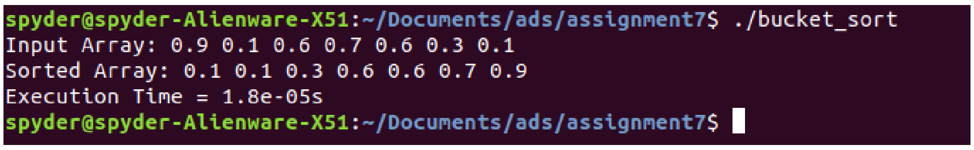
1.2 Bucket Sort

In Figure 3, we can see the implementation of bucket sort in C++. Figure 4 shows the driver window of the execution of the algorithm. The console output displays the input sequence, the sorted sequence and the time taken for the algorithm to sort the input sequence.



```
void bucket_sort(float *arr, int size) {
        vector<float> bucket[size];
        for(int i = 0; i < size; i++) {
                bucket[int(size*arr[i])].push_back(arr[i]);
        }

        for(int i = 0; i < size; i++) {
                sort(bucket[i].begin(), bucket[i].end());
        }

        int j = 0;
        for(int i = 0; i < size; i++) {
                while(!bucket[i].empty()) {
                        arr[j++] = *(bucket[i].begin());
                        bucket[i].erase(bucket[i].begin());
                }
        }
}
```

Figure 3: Implementation of bucket sort in C++



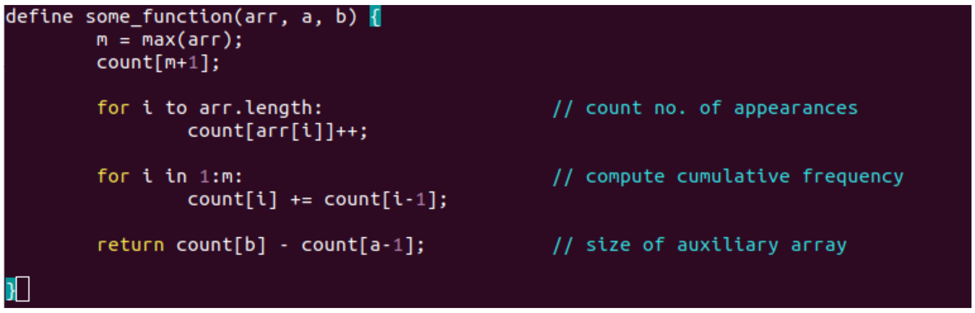
```
spyder@spyder-Alienware-X51:~/Documents/ads/assignment7$ ./bucket_sort
Input Array: 0.9 0.1 0.6 0.7 0.6 0.3 0.1
Sorted Array: 0.1 0.1 0.3 0.6 0.6 0.7 0.9
Execution Time = 1.8e-05s
spyder@spyder-Alienware-X51:~/Documents/ads/assignment7$
```

Figure 4: Bucket Sort Algorithm Execution

Bucket sort divides the interval [0,1] into n equal-sized sub-intervals and then distributes the n input numbers into the sub-intervals (buckets). To produce the output, we simply sort the numbers in each bucket then go through the buckets in order. The algorithm assumes that the input is an $n - element$ array. It requires an auxiliary array of linked-lists and assumes that there is a mechanism for maintaining such lists.

1.3 Building Auxiliary Storage

Given n integers in the range 0 to k, we can design and write an algorithm with pre-processing time $\Theta(n + k)$ for building auxiliary storage. Figure 5 shows the rough pseudocode for the possible solution.



```
define some_function(arr, a, b) {
        m = max(arr);
        count[m+1];

        for i to arr.length:                    // count no. of appearances
                count[arr[i]]++;

        for i in 1:m:                           // compute cumulative frequency
                count[i] += count[i-1];

        return count[b] - count[a-1];           // size of auxiliary array

}
```

Figure 5: Auxiliary Array Creation Pseudocode

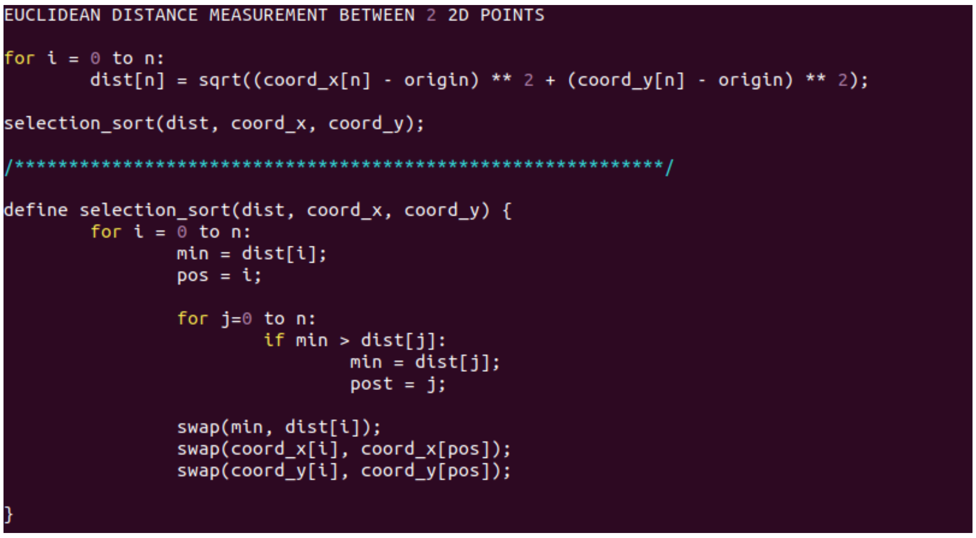
1.4 Alphabetization

1.5 Bucket Sort Worst Case Analysis

Assume out algorithm puts all input elements in the same bucket and then sorts that bucket using insertion sort. Assume all these numbers are in decreasing order. This set of assumptions defines the worst case for bucket sort. We know from earlier solutions that the worst case time complexity for insertion sort is $O(n^2), \quad \therefore \quad \Theta(n^2)$.

1.6 BONUS: Euclidean Distance Measure

Given n 2D points that are uniformly randomly distributed within the unit circle, we can design and write an algorithm that sorts the points in linear time by increasing Euclidean distance between two 2D points. Figure 6 shows a snapshot of the pseudocode which would define an algorithm for computing the Euclidean distance between two 2D points.



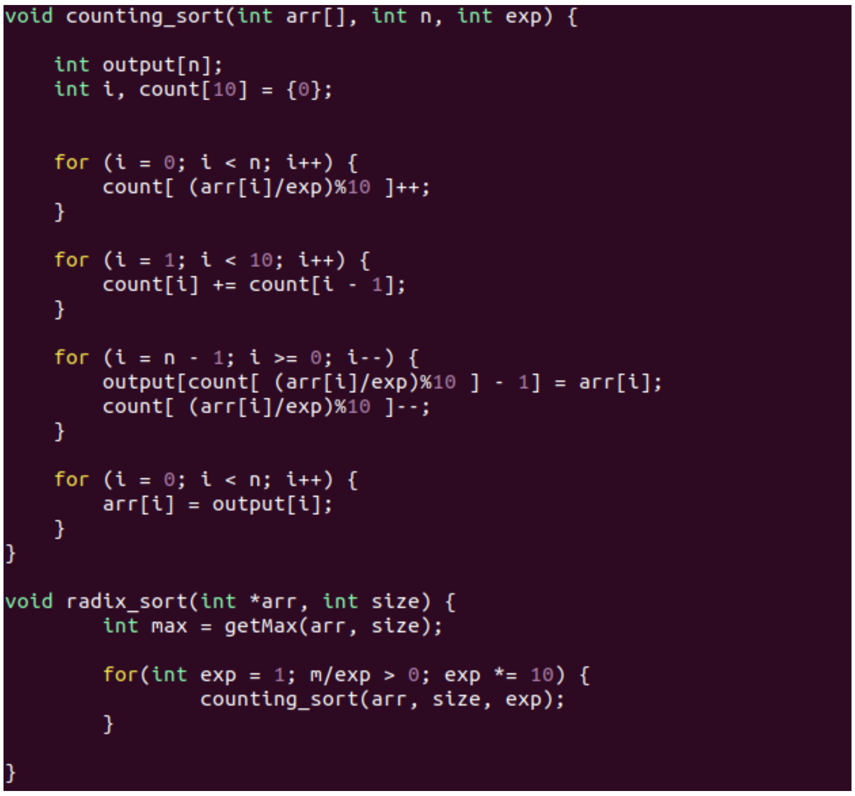
```
EUCLIDEAN DISTANCE MEASUREMENT BETWEEN 2 2D POINTS

for i = 0 to n:
        dist[n] = sqrt((coord_x[n] - origin) ** 2 + (coord_y[n] - origin) ** 2);

selection_sort(dist, coord_x, coord_y);

/*************************************************************/

define selection_sort(dist, coord_x, coord_y) {
        for i = 0 to n:
                min = dist[i];
                pos = i;

                for j=0 to n:
                        if min > dist[j]:
                                min = dist[j];
                                post = j;

                swap(min, dist[i]);
                swap(coord_x[i], coord_x[pos]);
                swap(coord_y[i], coord_y[pos]);

}
```

Figure 6: Pseudocode for computing Euclidean distance between two 2D points

In the algorithm, we make use of selection sort since we don't have any time complexity constraints.

2 Radix Sort

The implementation of Radix sort can be seen in Figure 7. We must use a variant of counting sort which can also be seen in Figure 7.



```
void counting_sort(int arr[], int n, int exp) {

    int output[n];
    int i, count[10] = {0};


    for (i = 0; i < n; i++) {
        count[ (arr[i]/exp)%10 ]++;
    }

    for (i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    for (i = n - 1; i >= 0; i--) {
        output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
        count[ (arr[i]/exp)%10 ]--;
    }

    for (i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

void radix_sort(int *arr, int size) {
        int max = getMax(arr, size);

        for(int exp = 1; m/exp > 0; exp *= 10) {
                counting_sort(arr, size, exp);
        }

}
```

Figure 7: Hollerith's Original Radix Algorithm Implementation in C++

2.1 Time Complexity

The storage of the implementation is $\Theta(n)$ for the original array. We also have $\Theta(10)$ for array C and $\Theta(n)$ for the output array. This leads to the conclusion that the asymptotic storage space is:

$$S(n) = 2n + 10 \rightarrow \Theta(n)$$
$$\therefore T(n) = \Theta(nlogn)$$

The best case for this algorithm would be the same as that for Bucket sort. This implies there would be one element in every 'bucket' yielding a time complexity of $\Theta(n)$. The worst case for this algorithm would be if every element was equal in value implying all elements fall in the same bucket for every iteration of bit-division. This process yields a time complexity of $\Theta(nlogk)$, where k is the element which is repeated.

2.2 BONUS

Given that we have to sort n integers in the range $[0, n^3 - 1]$, it would be ideal to use Radix sort. Most other algorithms would have either poor space or time complexity in a range of this magnitude.