Algorithms and Data Structures

Drishti Maharjan

March 25, 2019

Assignment 6

Problem 6.1

a) Code in countsort.c

b) Code in bucketsort.c

c) Using countsort for this question. Using parts of 6.1 a to implement it, so please find the references in the comments. The initial call to this algorithm is same as 6.1 a i.e. Countsort(array, sizeArray)

Algorithm 1 Counting ints in range [a,b] in CountSort

```
1: Countsort(array,n)
   //Function Countsort takes parameters array and $n$ = size of array
2: ..insert lines 21-34 of 6.1 a..
3: $result \leftarrow$ RangeCount(C,a,b)
   // End of Countsort function
   //result holds the number of ints in range [a,b]

1: RangeCount(C,a,b)
   // RangeCount takes array C from CountSort, a and b (lower and upper
   bounds) of the range required
2: $Count \leftarrow 0$
3: for $i = a$ to $b$ do
4:     $Count \leftarrow Count + C[i]$
5: return $Count$
```

d) Code in wordsort.cpp (c++ so that easy to use string arrays)

e) **Worst case for Bucket Sort**
For a uniformly distributed array, let's assume that the difference between the minimum element and maximum element of an array is very small. This would mean the range of the array would be very small. According to my Bucket sort implementation, all the elements will go in a single bucket and, insertion sort will be performed in the entire array. If the array is reverse sorted(the worst case of insertion sort), this would make the time complexity of insertion sort $\theta(n^2)$. The other operations of the bucket sort algorithm has time complexity $\theta(n)$. Hence, the total time complexity will be $\theta(n^2 + n) = \theta(n^2)$

An example of an array for worst case could be something like [0.99,0.98,0.97,0.96]. For my code, I tried computing time for the aforementioned array(*case 1*) and another array [0.96,0.17,0.98,0.69] (*case 2*) of the same length. The time computation for case 1 was 0.000051 and for case 2 was 0.000038. Even for smaller array size, we can see the time difference clearly, so the time complexity of worst case for bucket sort is $\theta(n^2)$.

f) Let's say we have a list A with 2D points represented as:

$A = [(x1, y1)..(xn, yn)]$.

And, we have a 1-D array:
$A[i].d = d1..dn$ where di = Euclidean distance of point (xi,yi) from origin of unit circle, and so on.

As the 2d points are uniformly distributed and we take Euclidian distance of unit circle, the distance range will be between 0 and 1. Due to these reasons, bucketsort seems to be the most suitable algorithm for this problem. But, bucket sort algorithm is slighlty modified to map the indexes of distance with its respective points.

I have 2 functions for this problem: Distance and BucketSort. Distance computes distance of respective points. BucketSort applies regular bucket sort over the distance array, and in order to be time efficient(rather than space efficient), n * distance of respective point is used as the index to store points, so buckets are divided according to order of distance. This mapping is later used to sort the points as we have sorted buckets.

Pseudocode for bucketsort according to Euclidian distance:

Algorithm 2 Pseudocode

```
1: **Distance(A,i)** // Distance takes struct A, and index of points i
2: $d \leftarrow \sqrt{(A[i].x)^2 + (A[i].y)^2}$
3: **return** $d$ //Returns distance of the given point

4: **BucketSort(A,n)** //BucketSort takes struct A, and n(no of points)
5: Bucket[n] ← [0] //initialization
6: **for** i = 1 to n **do**
7:     A[i].d ← **Distance(A,i)** //filling distance array
8:     $dig \leftarrow$ n $*A[i].d$ //index of bucket
9:     Bucket[dig] ← A[i] //A[i] is filled in buckets in order of distance
10: **end for**
11: **for** i = 1 to n **do**
12:     Sort Bucket[i] using any fast algorithm (in 6.1 b, insertion sort)
13: **end for**
14: index ← 1
15: **for** i = 1 to n **do** //Concatenating buckets
16:     **while** Bucket[i] is not null **do**
17:         A[index] ← element of Bucket[i]
18:         A[index] ← A[index] + 1
19:     **end while**
20: **end for**
21: **return** A //return sorted points (struct A)
```

Problem 6.2

a) Code in Hollerith.cpp

b) **Hollerith's Radix Sort**
This algorithm uses bucket sort and starts from the most significant bit.

Time Complexity

Best Case:
If the difference between the numbers in the array is significant, the implementation of bucket sort in the algorithm places the elements into different buckets. There is no need of traversing through every digit for these numbers as every bucket has one element. This means every bucket is sorted in its own as it has only one element, making it the best case. Then, the buckets are concatenated to give a list. So, the time complexity remains the same as of bucket sort: $\theta(n)$

Worst Case:
The difference between the elements is very less, or all the elements in the array are the same is the worst case. This would mean all the elements fall in the same bucket everytime bucket sort is performed. If we have d digits, then for every digit, bucket sort will be performed once. Therefore, running time for the worst case would be $\theta(dn)$ where $d = log_b k$ where b = base and k = no of digits in the maximum element of the array.

Average Case:
For average case, let's assume half of the buckets are empty or has single element(close to best case), and the other half buckets have more than 1 element (between best and worst case). As mentioned in best case, time complexity for n/2 buckets of size 1 or 0 is $\theta(n/2)$. For the other half, bucket sort is performed 'd' times for n/2 buckets, so time complexity is $\theta(dn/2)$. Therefore, running time of average case is $\theta(n/2 + dn/2) = \theta(dn)$ where d = no of digits

Space Complexity

Best Case:
A single element is placed in a bucket, and no further buckets are required. The storage required is only for n buckets of size 1 each. Hence, the space complexity is $\theta(n)$.

Worst Case and Average Case:
n buckets will be initialized at first. Bucket sort will be performed for maximum d passes, where d = number of digits of max element in the array. This means, at most, dn buckets can be created. Therefore, space complexity for worst case is $\theta(dn)$.

c) Given the big range of array $n^3 - 1$, radix sort traversing from least significant digit to most significant digit would be the most efficient algorithm as its time complexity is $\theta(dn)$ where d is linear. Count Sort works for integers with smaller range, and bucket sort works only for uniformly distrivuted arrays. So, radix sort with a subroutine of count sort for each pass is a better choice. In this case, $d = log_b(n^3 - 1)$ if we have range of $(n^3 - 1)$ values with max value having d digits.

Pseudocode for radix sort

Algorithm 3 Radix Sort

```
 1: Radix-Sort(A, d) //A = array of no to sort
    //Each key in A[1..n] is a d-digit integer. Digits are numbered from 1 to d from least
    to most significant digit
 2: for j = 1 to d do
    //CountSort on digit j of all elements
 3:     $count[10] \leftarrow [0]$ //initialize count array as 0
 4:     for i = 1 to n do
 5:         count[keyof(A[i]) in pass j] ← count[keyof(A[i]) in pass j] + 1
 6:     end for
 7:     for k = 1 to 10 do
 8:         $count[k] \leftarrow count[k] + count[k-1]$
 9:     end for
    //Build resulting array by checking correct position of A[i] with the help of count[k]
10:     for i = n downto 1 do
11:         $result[count[keyof(A[i])]] \leftarrow A[j]$
12:         count[keyof(A[i])] ← count[keyof(A[i])] - 1
13:     end for
    //According to current digit j, array A now contains the sorted numbers

14:     for i = 1 to n do
15:         $A[i] \leftarrow Result[i]$
16:     end for
17: end for
```