Algorithms and Data Structures

Spring 2019

Assignment 6

Date: March 23, 2019

Taiyr Begeyev

Problem 6.1 Sorting in Linear Time

(a) Implement the algorithm for Counting Sort and then use it to sort the sequence $< 9, 1, 6, 7, 6, 2, 1 >$

```
/*
----------              .__
\__    ___/ _____    /__/  ___.__. _______
  /    /    \__  \   /  / <   /  / \_  __ \
  /    /     / __ \_ /  /  \___  /  /  / \/
  /____/    (____  / /__/  / ____/  /__/
                \/        \/
*/
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

void CountingSort(int*, int);
void printArr(int*, int);

int main() {
    int sequence [] = {9, 1, 6, 7, 6, 2, 1};
    CountingSort(sequence, 7);
    printArr(sequence, 7);
    return 0;
}

void CountingSort(int *A, int n) {
    // create Auxiliary Array
    int max = *max_element(A, A + n);
    int min = *min_element(A, A + n);
    int range = max - min + 1;
    int *C = new int[range];
    // create the resultant array
    int *B = new int[n];

    // initialize the array with zeros
    for (int i = 0; i < range; i++) {
        C[i] = 0;
    }
    // iterate through array, calculate and the number
    // of occurences of A[i] to C[A[i]]
    for (int i = 0; i < n; i++) {
        C[A[i] - min]++;
    }

    // determine for each i = 0 .. k how many input elements
    // are less than or equal to i
    for (int i = 1; i < range; i++) {
        C[i] += C[i - 1];
    }

    // place each element from C into its correct sorted position
    for (int i = n - 1; i >= 0; i--) {
        B[C[A[i] - min] - 1] = A[i];
        C[A[i] - min]--;
    }
    for (int i = 0; i < n; i++) {
        A[i] = B[i];
```

```
    }

    // deallocate
    delete [] B;
    delete [] C;
}

void printArr(int *arr, int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}
```

(b) Implement the algorithm for Bucket Sort and then use it to sort the sequence $< 0.9, 0.1, 0.6, 0.7, 0.6, 0.3, 0.1 >$

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

void BucketSort(vector<float>&);
void printArr(vector<float>&);

int main() {
    vector<float> sequence = {0.9, 0.1, 0.6, 0.7, 0.6, 0.3, 0.1};
    BucketSort(sequence);
    printArr(sequence);
    return 0;
}

void BucketSort(vector<float> &arr) {
    int n = arr.size();
    // set up an array of initially empty "buckets"
    vector<float> bucket[n];

    // go over the original array, putting each object
    // in its bucket
    for (int i = 0; i < n; i++) {
        int bi = n * arr[i];
        bucket[bi].push_back(arr[i]);
    }

    // sort each non-empty bucket
    for (int i = 0; i < n; i++) {
        sort(bucket[i].begin(), bucket[i].end());
    }
    arr.clear();
    // gather all buckets
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < bucket[i].size(); j++)
            arr.push_back(bucket[i][j]);
    }
}

void printArr(vector<float>& arr) {
    for (int i = 0; i < arr.size(); i++)
        cout << arr[i] << " ";
    cout << endl;
```

```
}
```

(c) Given n integers in the range 0 to k, design and write down an algorithm (only pseudocode) with preprocessing time (n+k) for building auxiliary storage, which counts in O(1) how many of the integers fall into the interval [a,b].

Algorithm 1 Pseudocode

```
1: procedure F(A, n, a, b)
2:     B[k] ← array with all 0s
3:     for i = 0 up to n do
4:         B[A[i]] ← B[A[i]] + 1
5:     end for
6:     for i = 0 up to k do
7:         B[i] ← B[i] + B[i − 1]
8:     end for
9:     return B[b] − B[a − 1]
```

(d) Given a sequence of n English words of length k, implement an algorithm that sorts them alphabetically in $\Theta(n)$. Let k and n be flexible, i.e., automatically determined when reading the input sequence. You can consider k to behave like a constant in comparison with n. Example sequence of words to sort: < "word", "category", "cat", "new", "news", "world", "bear", "at", "work", "time" >.

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

void SortWords(vector<string>&);
void print(vector<string>&);

int main() {
    vector<string> words =
        {
            "word", "category", "cat", "new",
            "news", "world", "bear", "at",
            "work", "time"
        };

    cout << "Initial sequence: " << endl;
    print(words);
    SortWords(words);
    cout << "Sorted sequence: " << endl;
    print(words);

    return 0;
}

/**
 * @brief Sort words alphabetically
 *
 * This algorithm doesn't rely on data comparison methods
 * For characters' of each word two operations are performed:
 * division by a factor to disregard digit following digit i
```

```
 * being processed in the current pass and division modulo radix
 * to disregard all digits preceeding i for a total of 2n digits
 * = O(n) operations.
 *
 * In each pass, all characters are moved to piles (queues) and then
 * back to words for a total of 2n digits = O(n) moves. The algorithm
 * requires additional space for piles, which is implemented using STL
 * feature: Queue. My implementation uses only for loops with counters;
 * therefore, it requires the same amount of passes for each case: best,
 * average, worst. The body of the while loop is always executed n times
 * to deque all integers from all queues
*/

void SortWords(vector<string>& words) {
    // sequence of n English words
    int n = words.size();
    // we can consider k to behave like
    // a constant in comparison with n
    const int radix = 128; // ASCII table from 0..127
    const int k = 10; // the maximum number of digits
    // create radix numbers of queues. In other words, create radix
    // numbers of piles. We will use them to put each character in its pile
    queue <string> characters[radix];

    int i, j, factor;
    // walk through all digits left -> right
    for (i = 0, factor = 1; i < k; factor *= radix, i++) {
        // walk through all words and put a character on a i position
        // into one of the piles (queue)
        for (j = 0; j < n; j++) {
            int dummy = ((int)(words[j][i]) / factor) % radix;
            characters[dummy].push(words[j]);
        }
        for (i = j = 0; i < radix; i++) {
            while(!characters[i].empty()) {
                words[j++] = characters[i].front();
                characters[i].pop();
            }
        }
    }
}

/**
 * @brief print the passed Array
 * @param array - to be printed
 * @param n - number of elements
*/

void print(vector<string>& words) {
    for (int i = 0; i < words.size(); i++) {
        cout << words[i] << " ";
    }
    cout << endl;
}
```

(e) Given any input sequence of length n, determine the worst-case time complexity for Bucket Sort. Give an example of a worst-case scenario and the prove corresponding the complexity.

Bucket Sort has a time complexity of $O(n^2)$ at the worst case, when all elements are placed in a single bucket. The overall performance would then be dominated by the algorithm used to sort each bucket, which is typically insertion sort. Now we need to ask ourselves, what is the worst-case time complexity for Insertion Sort. Of course, it occurs when elements are in descending order. Therefore, the time complexity for the Insertion Sort is $O(n^2)$. It proves that Bucket Sort has a time complexity of $O(n^2)$ at the worst case.

(f) Given n 2D points that are uniformly randomly distributed within the unit circle, design and write down an algorithm (only pseudocode) that sorts the points by increasing Euclidean distance to the circle's origin. Write also a pseudocode function for the computation of the Euclidean distance between two 2D points.

Two points of type structure are passed. Use the Euclidean distance formula in order to find the distance and then return it.

Algorithm 2 Distance between two 2D points.

```
1: procedure DISTANCE(p, q)
2:     return sqrt((q.x − p.x)^2 + (q.y − p.y)^2)
```

Pseudocode assumes that a list of structure is passed. Every point besides x and y components also has a d component, which stands for the distance between two points, that we need to find.

Algorithm 3 Sort the points by increasing Euclidean distance to the circle's origin

```
1: procedure SORT2D(A, n)
2:     let B[0..n − 1] be a new array
3:     for i = 0 to n − 1
4:         make B[i] an empty list
5:     for i = 1 to n
6:         A[i].d ← Distance(A[i].d, origin)
7:         Insert A[i] into list B[n ∗ A[i].d]
8:     for i = 0 to n − 1
9:         sort list B[i] with insertion sort
10:    concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

Problem 6.2 Radix Sort

Consider Hollerith's original version of the Radix Sort, i.e., a Radix Sort that starts with the most significant bit and propagates iteratively to the least significant bit (instead of vice versa).

(a) Implement Hollerith's original version of the Radix Sort.

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

void BucketSort(vector<int>&, int);
void RadixSort(vector<int>&);
void print(vector<int>&);

int main() {
    cout << "Enter the number of elements: ";
    int n;
    cin >> n;
```

```
    vector<int> arr(n);
    for(int i = 0; i < n; i++){
        cin >> arr[i];
    }

    RadixSort(arr);
    print(arr);

    return 0;
}

/**
 * @brief Bucket Sort implementation algorithm
 *
 * Divides the interval into n equal-sized buckets
 * and then distributes the n input numbers into the buckets
 *
 * @param arr - array to be sorted
 * @param n - number of elements in the array
*/

void BucketSort(vector<int>& arr, int exp) {
    // to make sure that we don't divide by 0
    if (exp == 0) {
        return;
    }

    // set up an array of initially empty "buckets"
    vector<int> b[10];

    // go over the original array, putting each object
    // in its bucket
    for(int i = 0; i < arr.size(); i++){
        int bi = (arr[i] / exp) % 10;
        b[bi].push_back(arr[i]);
    }

    // sort each non-empty bucket
    for(int i = 0; i < 10; i++){
        if(b[i].size() > 1){
            BucketSort(b[i], exp / 10);
        }
    }
    arr.clear();
    // gather all buckets
    int idx = 0;
    for(int i = 0; i < 10; i++)
        for(int j = 0; j < b[i].size(); j++)
            arr.push_back(b[i][j]);
}

void RadixSort(vector<int>& arr) {
    int maxEl = 0;
    // find the largest number (number with most digits)
    for(int i = 0; i < arr.size(); i++){
        maxEl = max(maxEl, arr[i]);
    }

    // find exponent
```

```
    int exp = 1;
    while(maxEl /exp > 0)
        exp *= 10;

    exp /= 10;
    BucketSort(arr, exp);
}

/**
 * @brief print the passed vector container
 * @param arr - to be printed
 * @param n - number of elements
*/

void print(vector<int>& arr) {
    for(int i = 0; i < arr.size(); i++){
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

(b) Determine and prove the asymptotic time complexity and the asymptotic storage space required for your implementation.
Hollerith's version of Radix Sort uses the bucket sort subroutine and starts from the most significant bit. If the difference between the numbers is big, it might not be necessary to spread through the digits. The reason is that all numbers will fall into different buckets, and a bucket of size 1 is an already sorted array.
So the time complexity for the best case is $\Theta(n)$, same as bucket sort.
Let's consider the worst case, when all numbers in the array the same. In this case, all the numbers will fall into the same bucket every time bucket sort is called. So the time complexity for the worst case is $\Theta(dn)$, where $d = \log_b k$ and k is the maximum element in the array and b is the base.
In the average case, half of the buckets will have either 0 or 1, and the another half will have more than 1 element in them. Therefore, the total running time will be $\Theta(n + \frac{d}{2}n) \Rightarrow \Theta(\frac{d}{2}n) \Rightarrow \Theta(dn)$

The space complexity for the best case is $\Theta(n)$. For average cases and worst cases, n buckets are declared. For every recursive call of Bucket Sort, new buckets are created. Therefore, dn buckets are created. That means that the space compelxity is $\Theta(dn)$.

(c) Write down the pseudocode for an algorithm which sorts n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

First run through the list of integers and convert each one to base n, then radix sort them. Each number will have at most $log_n n^3 = 3$ digits so there will only need to be 3 passes. For each pass, there are n possible values which can be taken on, so we can use counting sort to sort each digit in $O(n)$ time.

Reference:

1. Wikipedia contributors. "Counting sort." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 22 Dec. 2018. Web. 24 Mar. 2019.
`https://en.wikipedia.org/wiki/Counting_sort`
2. "Bucket Sort" Growing with the Web
`https://www.growingwiththeweb.com/2015/06/bucket-sort.html`
3. Drozdek, Adam. "Data Structures and Algorithms in C++", Second Edition, 2001. 482-483
4. Cormen, T. H., Cormen, T. H. (2001). Introduction to algorithms. Cambridge, Mass: MIT Press.
5. Wikipedia contributors. "Radix sort." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 20 Mar. 2019. Web. 25 Mar. 2019.
`https://en.wikipedia.org/wiki/Radix_sort#Stable_MSD_radix_sort_implementations`