Algorithms and Data Structures: Homework #7

Due on March 23, 2020 at 23:00

Henri Sota

Problem 7.1

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

Algorithm 1 Counting Sort[A, B, k]

Declare C: Array with k elements
for $i = 0$ **to** k **do**
  $C[i] = 0$
end for
for $j = 1$ **to** $A.length$ **do**
  $C[A[j]] = C[A[j]] + 1$
end for
for $j = 1$ **to** $A.length$ **do**
  $C[i] = C[i] + C[i - 1]$
end for
for $j = A.length$ **to** 1 **do**
  $B[C[A[j]]] = A[j]$
  $C[A[j]] = C[A[j]] - 1$
end for

```python
def countingSort(A, B, k):
    # Initialize array C with values 0
    C = [0 for _ in range(k)]

    # Populate C with frequencies of indices in A
    for j in range(len(A)):
        C[A[j]] = C[A[j]] + 1

    # Fill C with number of elements smaller or equal to index
    for i in range(1, k):
        C[i] = C[i] + C[i - 1]

    # Place each element in its stable position by traversing the array
    # backwards and inserting at index of elements smaller or equal to it
    for j in range(len(A) - 1, -1, -1):
        B[C[A[j]] - 1] = A[j]
        # Lower the frequency count of element and prepare to place the next
        # element with that value in the index before it to maintain stability
        C[A[j]] -= 1

    return B

if (__name__ == '__main__'):
    A = [9, 1, 6, 7, 6, 2, 1]

    # Preprocess storage and size of frequency table
    B = [None for _ in range(len(A))]
    k = max(A) + 1

    print('Counting Sort:')
    print(countingSort(A, B, k))
```

Listing 1: Counting Sort in Python

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

Algorithm 2 Bucket Sort[A]

Declare B: Array with n elements
$n = A.length$
for $i = 1$ **to** n **do**
  Declare $B[i]$: **List**
end for
for $i = 1$ **to** n **do**
  Insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
end for
for $i = 1$ **to** n **do**
  Insertion Sort($B[i]$)
end for
Concatenate the lists $B[1], B[2], \ldots, B[n]$ together in order

```
def insertionSort(Bucket):
    for j in range(1, len(Bucket)):
        key = Bucket[j]
        i = j - 1
        # Search for the position where to place key value in the sorted
        # subarray Bucket[0..i]
        while (i > -1 and Bucket[i] > key):
            Bucket[i + 1] = Bucket[i]
            i -= 1
        Bucket[i + 1] = key
    return Bucket

def bucketSort(A):
    sorted = []
    n = len(A)

    # Declare each element of B as a list
    B = [[] for _ in range(n)]

    # Insert each element into its corresponding bucket
    for i in range(n):
        B[int(math.floor(n * A[i]))].append(A[i])

    # Sort elements inside the bucket i with Insertion Sort
    # Note: Other sorting algorithms with better time complexity can be used
    for i in range(n):
        B[i] = insertionSort(B[i])

    # Concatenate lists together to yield sorted array
    for i in range(n):
        sorted += B[i]

    return sorted

if (__name__ == '__main__'):
    A = [0.9, 0.1, 0.6, 0.7, 0.6, 0.3, 0.1]

    print('Bucket Sort:')
```

```
    print(bucketSort(A))
```

Listing 2: Bucket Sort in Python

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

In order to have a time complexity of $O(1)$, preprocessing has to do all of the work of finding the number of elements smaller or equal to the index in an auxiliary array C. In order to do so, the algorithm can utilize some part of the Counting Sort algorithm as its preprocessing part. The time complexity of the preprocessing part will be $\Theta(n+k)$ as the 2 loops in it will depend on the value of the largest element of the loop and on the size of the array. When called with the inclusive interval bounds, a and b, it will calculate the difference between the number of elements smaller or equal to b and the number of elements smaller or equal to $a-1$, because the goal is to calculate the number of elements in between a and b inclusively. Performing subtraction only needs constant time. Therefore, time complexity will be $O(1)$.

Algorithm 3 Interval Count[A, a, b]

$k = \max(A) + 1$
Declare C: Array with k elements
for $i = 0$ **to** k **do**
  $C[i] = 0$
end for
for $j = 1$ **to** $A.length$ **do**
  $C[A[j]] = C[A[j]] + 1$
end for
for $j = 1$ **to** $A.length$ **do**
  $C[i] = C[i] + C[i-1]$
end for
return $C[b] - C[a-1]$

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"* $>$.

Algorithm 4 Sort Alphabetically[A,i]

Declare C: Array with 27 elements
Declare Sorted: Array with $A.length$ elements
for $j = 1$ **to** $A.length$ **do**
  if $A[j].length > i$ **then**
    ch = Position in Alphabet$(A[j][i]) + 1$
  else
    ch $= 1$
  end if
  $C[ch] = C[ch] + 1$
end for
for $j = 2$ **to** 27 **do**
  $C[i] = C[i] + C[i-1]$
end for
for $j = A.length$ **to** 1 **do**
  if $A[j].length > i$ **then**
    ch = Position in Alphabet$(A[j][i]) + 1$
  else
    ch $= 1$
  end if
  Sorted$[C[ch] - 1] = A[j]$
  $C[ch] = C[ch] - 1$
end for

Algorithm 5 Sort Strings Linear[A]

$d =$ maximum size of a word in $A - 1$
for $i = d$ **to** 1 **do**
  $A =$ Sort Alphabetically(A, i)
end for

```
def sortAlphabetically(A, i):
    # Create empty list for each character of the alphabet and for the result
    # with one extra element to hold empty positions of smaller sized words
    C = [0 for _ in range(27)]
    sorted = ['' for _ in range(len(A))]

    # Populate C with the frequencies of the characters of the alphabet
    for j in range(len(A)):
        if (len(A[j]) > i):
            ch = ord(A[j][i]) - 96
        else: ch = 0
        C[ch] += 1

    # Fill C with the number of characters "smaller" than or equal to the letter
    # at j-th position of the alphabet from the existing frequency list
```

```
    for j in range(1, 27):
        C[j] += C[j - 1]

    # Place each word in its stable position by traversing the array
    # backwards and inserting at index of characters smaller or equal to it
    for j in range(len(A)-1, -1, -1):
        if (len(A[j]) > i):
            ch = ord(A[j][i]) - 96
        else: ch = 0
        sorted[C[ch] - 1] = A[j]
        # Lower the frequency count of letter and prepare to place the next
        # word with that character in the index before it to maintain stability
        C[ch] -= 1

    return sorted

def sortStringsLinear(A):
    d = max([len(i) for i in A]) - 1
    for i in range(d, -1 , -1):
        A = sortAlphabetically(A, i)
    return A

if (__name__ == '__main__'):
    A = ['word', 'category', 'cat', 'new', 'news', 'world', 'bear', 'at',
         'work', 'time']
    print('Alphabetical Sorting of Strings:')
    sorted = sortStringsLinear(A)
    for i, j in enumerate(sorted):
        if (i == 0):
            print(j, end='')
        else: print(', ' + j, end='')
```

Listing 3: Sort Words Linear in Python

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.
The worst-case time complexity for Bucket Sort happens when all the elements to be sorted fall into only one bucket. The sorting procedure would therefore be the main contributor to the time complexity. The worst-case time complexity would be $O(n^2)$ if Insertion Sort is the sorting algorithm. Taking an example case when array is $[0.24, 0.27, 0.23, 0.21, 0.28]$.

$$
\begin{array}{ll}
1 \ .24 & 0 \ / \\
2 \ .27 & 1 \ / \\
3 \ .23 & 2 \ \rightarrow .21 \rightarrow .23 \rightarrow .24 \rightarrow .27 \rightarrow .28 \\
4 \ .21 & 3 \ / \\
5 \ .28 & 4 \ /
\end{array}
$$

The other parts of Bucket Sort, the loop for creating an array of lists and the loop for inserting the elements into the buckets each take $O(n)$. The loop which sorts each of the buckets won't take $O(n^3)$ as sort will only work on only one of the buckets. Therefore worst-case time complexity will be:

$$
\begin{aligned}
T(n) &= O(n) + O(n) + O(n^2) \\
&= 2O(n) + O(n^2) \\
&= O(n^2)
\end{aligned}
$$

Problem 7.2

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

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

Algorithm 6 Radix Sort MSD[A, i, longest]

if $A.length < 2$ **or** $i \geq longest$ **then**
  return A
end if
Declare B: Array with 10 lists
for $j = 1$ **to** $A.length$ **do**
  digit $= \text{int}(A[j]/(\text{pow}(10, \text{longest} - i - 1)))\%10$
  B[digit].append($A[j]$)
end for
for $j = 1$ **to** $B.length$ **do**
  $B[j] = \text{Radix Sort MSD}(B[j], i + 1, \text{longest})$
end for
Concatenate the lists $B[1], B[2], \ldots, B[n]$ together in order

```
def findHighestDigitCount(A):
    # Get biggest element in array
    maxElement = max(A)
    exp = 1

    # Iteratively raise the exponent until biggest number over 10 to exp is < 1
    while (maxElement / exp > 1.0):
        exp *= 10

    return int(math.log(exp, 10))

def radixSortMSD(A, i, longest):
    # Check if array contains one or less elements or if digit being compared
    # is bigger than the number of digits of the biggest element
    if ((len(A) < 2) or (i >= longest)):
        return A

    sorted = []

    # Create buckets corresponding to each digit of the base
    B = [[] for x in range(10)]

    # Traverse each number in the array
    for number in A:
        # Retrieve the digit by dividing by power of 10 to (longest - 1 - i)
        # and calculating the remainder of division by 10
        digit = (number // (10**(longest - i - 1))) % 10
        # Place number in bucket which corresponds to the digit being compared
        B[digit].append(number)

    # Recursively sort buckets
    for j in range(len(B)):
        B[j] = radixSortMSD(B[j], i + 1, longest)

    # Concatenate lists together to yield sorted array
```

```
    for j in B:
        for number in j:
            sorted.append(number)

    return sorted
```

Listing 4: Radix Sort MSD in Python

Implementation was derived from the slides found in the lectures of Princeton here and from Wikipedia article here, based off of Hollerith's work on tabulating machines.

b) Determine and prove the asymptotic time complexity and the asymptotic storage space required for your implementation.

Time Complexity

- Best Case
  Radix Sort MSD's best case happens when elements of the array get evenly distributed between the buckets in such a way to minimize the time taken for the recursive step. In cases when the number of elements matches the base or the number of buckets, which is rare, one element would occupy one bucket and the bucket would therefore be sorted as it has only one element. It would only require one pass through the array. Time complexity of Radix Sort MSD in the best case would be $\Omega(n)$.
- Average Case
  Radix Sort MSD's average case happens when elements of the array are distributed into the buckets in such a way that half of them would have multiple elements and the other half would have 0 or 1 element. The recursion would run for half of the buckets and it would stop for the other half of the buckets. A higher number of buckets and less items per bucket would have to be sorted compared to the worst case, but the time complexity would not change. Time complexity of Radix Sort MSD in the average case would be $\Theta(n) + \Theta(d * \frac{n}{2}) = \Theta(d * \frac{n}{2}) = \Theta(dn)$, where $d = \log_{base}(\textit{biggest element})$.
- Worst Case Radix Sort MSD's worst case happens when elements of the array are all distributed into one bucket and all of them are sufficiently close to each other. The time complexity of Radix Sort would depend on the recursive step of sorting that bucket. Time complexity of Radix Sort MSD in the worst case would be $O(d * n)$, where $d = \log_{base}(\textit{biggest element})$.

Space Complexity

- Best Case
  Radix Sort MSD's best case happens when elements of the array get evenly distributed between the buckets in such a way to minimize the number of recursive steps. In cases when the number of elements matches the base or the number of buckets, which is rare, one element would occupy one bucket and the bucket would therefore be sorted as it has only one element. It would require to only use n buckets. Space complexity of Radix Sort MSD in the best case would be $\Omega(n)$.
- Average Case
  Radix Sort MSD's average case happens when the number of digits of the elements would be higher than 1. The algorithm would need to create the same number of buckets most of the time, but it wouldn't be the same as the worst case. On the next recursive step, the number of buckets to sort would grow smaller sometimes, but yet the space complexity would be affected by the number of digits of the biggest element mostly. Space complexity of Radix Sort MSD in the best case would be $\Theta(d * n)$, where $d = \log_{base}(\textit{biggest element})$.
- Worst Case
  Radix Sort MSD's worst case happens when elements of the array would all have the same number

of digits. Every recursive step would require to sort and save space for the same number of buckets. The space complexity of Radix Sort would depend on the recursive step of sorting that bucket. Space complexity of Radix Sort MSD in the worst case would be $O(d * n)$, where $d = \log_{base}(biggest\ element)$.