Algorithms and Data Structure 6

Yiping Deng

March 19, 2018

Problem 1

a)

We can see the python code here and explain it.

```
def counting_sort(arr, maximum):
    """Implementing the counting sort algorithm
    """
    countings = [0] * (maximum + 1)
    # initiate a list with zeros
    res = [-1] * len(arr)
    # result
    for entry in arr:
        print("counting of {} is incremented".format(entry))
        countings[entry] = countings[entry] + 1
        # self increment
    print("countings: {}".format(countings))
    print("now we aggregate the countings")
    for idx in range(len(countings) - 1):
        countings[idx + 1] = countings[idx + 1] + countings[idx]
        # aggregate the countings
    print("countings: {}".format(countings))
    print("restore the result")
    for entry in arr:
        # build the result
        # print("entry: {}, count: {}".format(entry, countings[entry]))
        res[countings[entry] - 1] = entry
        print("put {} at index {}".format(entry, countings[entry] - 1))
        countings[entry] = countings[entry] - 1

    return res

if __name__ == "__main__":
    arr = arr = [9, 1, 6, 7, 6, 2, 1]
    print(counting_sort(arr, 9))
```

Basically, in counting sort, we will count the appearance of each entries in the array, aggregate the counts in a ascending order, and then using such information to put every entries in the right position. We starts by running the script, we have

```
counting of 9 is incremented
counting of 1 is incremented
counting of 6 is incremented
counting of 7 is incremented
counting of 6 is incremented
counting of 2 is incremented
counting of 1 is incremented
```

```
countings: [0, 2, 1, 0, 0, 0, 2, 1, 0, 1]
now we aggregate the countings
countings: [0, 2, 3, 3, 3, 3, 5, 6, 6, 7]
restore the result
put 9 at index 6
put 1 at index 1
put 6 at index 4
put 7 at index 5
put 6 at index 3
put 2 at index 2
put 1 at index 0
[1, 1, 2, 6, 6, 7, 9]
```

b)

Similarly, we have a python script that executes the bucket sort and print the intermediate steps. It first put elements in the array into n equidistance buckets, or arrays, and then do insertion sort on every buckets.

```python
def insertion_sort(lst):
    """ Implementation of insertion sort in python
    >>> insertion_sort([4, 3, 2 ,1])
    [1, 2, 3, 4]
    """
    print("calling the insertion sort on {}".format(lst))
    for idx in range(1, len(lst)):
        val = lst[idx] # current value
        pos = idx # current position

        # move the element
        while pos > 0 and lst[pos -1] > val:
            lst[pos] = lst[pos - 1]
            pos = pos - 1

        lst[pos] = val

    return lst

def bucket_sort(arr):
    maximum = 1.0
    minimum = 0.0
    num_bucket = len(arr)
    step = (maximum - minimum) / float(num_bucket)
    print("number of bucket: {}".format(num_bucket))
    buckets = [[] for _ in range(num_bucket)]
    print(buckets)
    print("build empty buckets")

    def find_bucket(i):
        print("find bucket for {}".format(i))
        return int(i / step)

    res = []

    print("start to put in buckets")
    for entry in arr:
        b = find_bucket(entry)
        print("put in bucket {}".format(b))
        buckets[b].append(entry)
```

```python
        print("now in buckets we have {}".format(buckets))

    for bucket in buckets:
        n_bkt = insertion_sort(bucket)
        res = res + n_bkt
        print("building results by adding {}".format(n_bkt))

    return res

if __name__ == "__main__":
    arr = [0.9, 0.1, 0.6, 0.7, 0.6, 0.2, 0.1]
    print(bucket_sort(arr))
```

Let's run this script and see this algorithm is executed.

```
number of bucket: 7
[[], [], [], [], [], [], []]
build empty buckets
start to put in buckets
find bucket for 0.9
put in bucket 6
now in buckets we have [[], [], [], [], [], [], [0.9]]
find bucket for 0.1
put in bucket 0
now in buckets we have [[0.1], [], [], [], [], [], [0.9]]
find bucket for 0.6
put in bucket 4
now in buckets we have [[0.1], [], [], [], [0.6], [], [0.9]]
find bucket for 0.7
put in bucket 4
now in buckets we have [[0.1], [], [], [], [0.6, 0.7], [], [0.9]]
find bucket for 0.6
put in bucket 4
now in buckets we have [[0.1], [], [], [], [0.6, 0.7, 0.6], [], [0.9]]
find bucket for 0.2
put in bucket 1
now in buckets we have [[0.1], [0.2], [], [], [0.6, 0.7, 0.6], [], [0.9]]
find bucket for 0.1
put in bucket 0
now in buckets we have [[0.1, 0.1], [0.2], [], [], [0.6, 0.7, 0.6], [], [0.9]]
calling the insertion sort on [0.1, 0.1]
building results by adding [0.1, 0.1]
calling the insertion sort on [0.2]
building results by adding [0.2]
calling the insertion sort on []
building results by adding []
calling the insertion sort on []
building results by adding []
calling the insertion sort on [0.6, 0.7, 0.6]
building results by adding [0.6, 0.6, 0.7]
calling the insertion sort on []
building results by adding []
calling the insertion sort on [0.9]
building results by adding [0.9]
[0.1, 0.1, 0.2, 0.6, 0.6, 0.7, 0.9]
```

c)

We basically implement the first half of bucket sort. We build k buckets and put them in accordingly. Without aggregating, to calculate the number of integer in inteval $[a, b]$, we just add up numbers from bucket a to bucket b, and it will give you the result.

d)

the concrete implementation is here

```
from collections import deque

def find_max_length(words):
    lens = map(len, words)
    return max(lens)

def find_bucket(word, idx):
    if(idx > len(word) - 1):
        # it is out of bound, count it as a very small number
        return 0
    else:
        # return the ascii code
        return ord(word[idx])

def pass_bucket(words, buckets, idx):
    # insert into buckets and pop them out again
    for word in words:
        # First in first out
        buckets[find_bucket(word, idx)].appendleft(word)
    res = []
    # restore the array
    for bucket in buckets:
        while len(bucket):
            res.append(bucket.pop())
    return res


def radix(words):
    # make the buckets
    buckets = [deque() for _ in range(256)]
    max_len = find_max_length(words)
    for idx in reversed(range(max_len)):
        words = pass_bucket(words, buckets, idx)
    return words

if __name__ == "__main__":
    # test case
    words = ["abp", "adp", "awewwss", "aaasdfwe"]
    print("before: {}".format(words))
    print("after: {}".format(radix(words)))
```

e)

Let's implement a variant of bubblesort

Algorithm 1 Distance

```
procedure DISTANCE(p)                         ▷ Input a point p
    d_p ← sqrt(p_1^2 + p_2^2)
return d_p
end procedure
```

1: **procedure** DISTANCE(p) ▷ Input a point p
2: $d_p \leftarrow \sqrt{p_1^2 + p_2^2}$
3: **return** d_p
4: **end procedure**

Algorithm 2 GreaterThan

```
procedure TESSTHAN(p, q)                    ▷ Compare two points p, q
    cmp ← Distance(p) > Distance(q)
return cmp
end procedure
```

Algorithm 3 Bubble Sort

```
procedure BUBBLESORT(A)                     ▷ Input a array A
    swapped ← true                          ▷ Make sure it will execute at least once
    while swapped do                        ▷ Keep executing until it is sorted
        swapped ← false
        for i from 1 to A.size - 1 do
            if GreaterThan(A[i], A[i + 1]) then
                swapped ← true
                swap(A[i], A[i + 1])        ▷ Swap two misplaced elements in the array
            end if
        end for
    end while
end procedure
```

Problem 2

a)

This is the variant of radix sort here.

```python
from collections import deque
import math
def radix(nums, base = 10, leftdigit = 0, total_digit = 10):
    # make the buckets
    rightdigit = 10 - leftdigit - 1
    def num_digits(num):
        return int(math.log(num, base)) + 1

    def cal_digit(num, digit):
        return (num // (base ** digit)) % base

    if(len(nums) <= 1):
        return nums
    done_bucket = []
    buckets = [[] for _ in range(base)]
    for num in nums:
        if(leftdigit >= total_digit):
            print(num_digits(num))
            done_bucket.append(num)
        else:
            buckets[cal_digit(num, rightdigit)].append(num)

    #divide and conquer
    buckets = [radix(b, base, leftdigit + 1, total_digit) for b in buckets]
    # extend the list
    return done_bucket + [b for blist in buckets for b in blist]
```

b)

Time complexity is straight forward. Assuming a uniform distribution on the input, every step will divide the input into *base* cases, and every depth you have n elements to put in and out of buckets in total, with a tree depth of number of digits, denoted as k. Thus, the time complexity is $O(k * n) = O(n)$ Space complexity will be also $O(k * n) = O(n)$. Every recursive call you will put n elements in and out of the buckets, thus it will have a space complexity of $O(k * n) = O(n)$

c)

You simply take $base = n$ as basis for Radix sort. Thus, it will be at most 3 as the depth of recursive call(at most 3 digit), and every recursive call will take n running time, leading to total $O(3n) = O(n)$