Practical Session #11

Sorting in Linear Time

Counting-Sort

A sort algorithm that is not based on comparisons, and supports duplicate keys.

- A is an input array of length n.
- B is the output array of length n too.
- C is an auxiliary array of size k.
- Assumption: the input array **A** consists of elements with integer keys in the range [1..k].

Counting-Sort (A, B, k)

```
for i ← 1 to k
    C[i] ← = 0
for j ← 1 to n
    C[A[j]] ← C[A[j]] + 1
    // C[i] = the number of appearances of i in A.
for i ← 2 to k
    C[i] ← C[i] + C[i-1]
    // C[i] = the number of elements in A that are ≤ i
for j ← n downto 1
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]] - 1
return B
```

Example:

n=6, k=3
Index 1 2 3 4 5 6
A= 3 2 3 1 3 1
C 2 1 3
C 2 3 6
j=6, A[6]=1, C[1]=2, B[2]← 1
B 1
C 1 3 6
j=5, A[5]=3, C[3]=6, B[6]←3
B 1 3
C 1 3 5
j=4, A[4]=1, C[1]=1, B[1]←1
B 1 1 3
C 0 3 5
j=3, A[3]=3, C[3]=5, B[5]←3
B 1 1 3 3
C 0 3 4
j=2, A[2]=2, C[2]=3, B[3]←2
B 1 1 2 3 3
C 0 2 4
j=1, A[1]=3, C[3]=4, B[4]←3
B 1 1 2 3 3 3
C 0 2 3

For more examples you can use the following Counting-Sort animation.

Run-time Complexity: O(n+k)

This is an improvement on comparison-based sorts, which need at least n*logn time in

the worst-case.
Counting sort is ***stable***, two elements with the same key value will appear in the output in the same order as they appeared in the input. Stability is important when there is additional data besides the key.

Radix-Sort

A stable sort algorithm for sorting elements with **d** digits, where digits are in base **b**, i.e., in range [0,**b-1**]. The algorithm uses a stable sort algorithm to sort the keys by each digit, starting with the least significant digit (the rightmost one) marked in the algorithm as the 1^{th} digit.

Radix-Sort(A[1..n], d) מיון בסיס

```
for i ← 1 to d
    Use stable sort to sort A on digit i (like counting-sort)
```

Run-time Complexity:
Assuming the stable sort runs in O(n+b) (such as counting sort) the running time is O(d(n+b)) = O(dn+db).
If d is constant and b=O(n), the running time is O(n).

Example
7 numbers with 3 digits. n=7, d=3,b=10 (decimal)
32**9**, 45**7**, 65**7**, 83**9**, 43**6**, 72**0**, 35**5**
7**2**0, 3**5**5, 4**3**6, 4**5**7, 6**5**7, 3**2**9, 8**3**9 – sorted by digit 1
720, **3**29, **4**36, **8**39, **3**55, **4**57, **6**57 – sorted by digit 2 (and 1)
329, **3**55, **4**36, **4**57, **6**57, **7**20, **8**39 – sorted
For more examples you can use the following Radix-Sort animation.

Bucket-Sort

- Assumption: Input array elements are uniformly distributed over the interval [0,1).

The idea of bucket-sort is to divide the interval [0,1) into n equal-sized subintervals (*buckets)*, and then distribute the n input numbers into the buckets. To produce the output we simply sort the elements in each bucket and then creating a sorted list by going through the buckets in order.

Bucket-sort (A)

```
n ← length(A)
for i ← 1 to n do
    B[⌊nA[i]⌋] ← A[i]
for i ← 1 to n do
    sort B[i] with insertion sort
concatenate the lists B[0], B[1] ,…, B[n-1] together in order
```

Run-time Complexity:
Assuming the inputs are uniformly distributed over [0,1), we expect O(1) elements in each bucket (average case), thus sorting them takes O(1) expected time.
We insert n elements into n buckets in O(n) and we concatenate the
Lists in O(n) ⇒ Total expected run time: O(n).

Sort Algorithms Review

	Keys Type	Average run-time	Worst case run-time	Extra Space	In Place	Stable
Insertion Sort	any	$O(n^2)$	$O(n^2)$	O(1)	√	√
Merge Sort	any	O(nlogn)	O(nlogn)	O(n)	X	√
Heap Sort	any	O(nlogn)	O(nlogn)	O(1)	√	X
Quick Sort	any	O(nlogn)	$O(n^2)$	O(1)	√	X
Counting Sort	integers [1..k]	O(n+k)	O(n+k)	O(n+k)	X	√
Radix Sort	d digits in base b	O(d(b+n))	O(d(b+n))	Depends on the stable sort used	Depends on the stable sort used	√
Bucket sort	[0,1)	O(n)	$O(n^2)$	O(n)	X	√

Question 1: Size of ranges using Counting-Sort

There are n integers in range [1...k]. Suggest an algorithm that preprocesses the input in O(n+k) time, and then can return how many numbers there are in the range [a...b] in O(1) time for any given a>=1 and b<=k.

Solution

The algorithm is based on counting-sort:

```
Preprocess(A, k)
   for i ← 1 to k
       C[i] ←  0
   for j ← 1 to n
       C[A[j]] ← C[A[j]] + 1  // C[i] = the number of appearances of i in A.
   for i ← 2 to k
       C[i] ← C[i] + C[i-1]   // C[i] = the number of elements in A that are ≤ i
   return(C)
end

Range(C,a,b)
   return(C[b]-C[a-1])
end
```

Question 2: An improved Bucket-Sort algorithm

Design an algorithm for sorting n data items with keys in the range [x,x+d) that runs in **expected** O(n) time if the items are uniformly distributed over [x,x+d], and runs in O(nlogn) in the worst distribution case.

Solution 1:

Use bucket sort over the range [x,x+d) with the following changes:

1. The elements in each bucket are stored in a AVL tree (instead of a linked list)
2. In the last stage, concatenate all the *inorder* visits of all the buckets one after another.

Note: bucket distribution function will be $B[\lfloor n((A[i]-x)/d)\rfloor] \leftarrow A[i]$

Time Complexity:

Let n_i be the number of elements in the tree in bucket i.

1. Inserting the n elements into the buckets takes $O(n_1\log n_1 + n_2\log n_2 + ... + n_n\log n_n)$
   - ♦ When the keys are uniformly distributed $n_i = O(1)$ for every i, hence $O(n_1\log n_1 + n_2\log n_2 + ... + n_n\log n_n) \leq c(n_1 + n_2 + ... + n_n) = cn$, where c is a constant.
   - ♦ In the worst distribution cases: $O(n_1\log n_1 + n_2\log n_2 + ... + n_n\log n_n) \leq O(n_1\log n + n_2\log n + ... + n_n\log n) = O((n_1 + n_2 + ... + n_n)(\log n)) = O(n\log n)$
2. *Inorder* traversals of all buckets takes $O(n_1 + n_2 + ... + n_n) = O(n)$
3. Concatenation of all *inorder* traversal lists takes O(n)

The algorithm runs in O(n) time for uniformly distributed keys and runs in O(nlogn) in the worst distribution case.

Solution 2:

Execute in parallel the following two algorithms:

1. Original bucket sort
2. Any sort algorithm that takes O(nlogn)

Stop when one of the algorithms has stopped and return the sorted elements.

Question 3: Choosing sorting algorithm efficiently

Given a set of n integers in the range $[1,n^3]$, suggest an efficient sorting algorithm .

Solution:

- Comparison-based algorithm takes O(nlogn) .
- Counting-sort: $k= n^3 \rightarrow O(n+ n^3)=O(n^3)$
- Radix-sort: b=n, d =4, $\rightarrow$ d(b+n)=O(4 (n+n))=O(n)
  Why is that ?
  Use radix-sort after preprocessing:
  1. Convert all numbers to base n in O(n) total time using mod and div operations.
     $x = [x_3,x_2,x_1,x_0]$ ($x_0= x \bmod n$, $x_i = \lfloor x/n^i \rfloor \bmod n$)
  2. Call radix-sort on the transformed numbers. O(n)

  All the numbers are in range 1 to n^3, therefore, we have at most 4 digits for each number. The running time for the suggest algorithm: d=4, b=n $\rightarrow$ O(4(n+n))=O(n).

Question 4: Sorting a collection of sets efficiently

There are m sets $S_1, S_2,...,S_m$.
Each set contains integers in the range [1..n] and m =O(n).
$n_i = |S_i|$ = number of elements in S_i. $|S_1| + |S_2| + ... + |S_m| = O(n)$

Suggest an algorithm for sorting all sets $S_1, S_2,...,S_m$ in O(n) time complexity and O(n) space (memory) complexity.
Note: The output is m sorted sets and not one merged sorted set.

Solution:

- If we sort each set using O(nlogn) run time algorithm we get :
  $T(n) = O(n_1\log n_1 + n_2\log n_2 + ... + n_m\log n_m) \leq \log n * O(n_1 + n_2 + ... + n_m) =$ nlogn
- If we sort each set with counting sort we get:
  $T(n) = O((n_1+n) + (n_2+n) + ... + (n_m+n)) = O(n+mn) = O(n^2)$.
  The space complexity is O(n).

The following algorithm runs in O(n) time and space complexity:

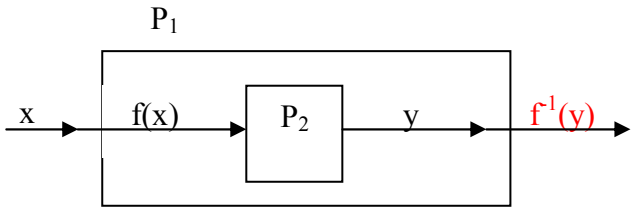
1. Add a new variable ***set-num*** to each element – O(n)
2. Build an array A of all the elements in all the sets - O(n)
3. Sort A using counting-sort on the keys - O(n+n)=O(n)
4. Split A into the original k sets according to the set field – O(n)
   (Counting sort by ***set-num***)

Question 5: Sorting lexicographically

Given n words in English (assume they are all in lower case). The words are not of the same length. Suggest an algorithm for sorting the words in a lexicographic order in O(n) time.

Solution:

Reduction:
Assume you have two problems P_1 and P_2 and the algorithm for solving P_2 is known.
A ***reduction*** from P_1 to P_2 is a way to solve problem P_1 by using the algorithm for solving P_2.



In our question the reduction is from lexicographical-sort to radix-sort:

1. **Translate Input of P1 to input for P2:**
   a. Let ***m*** be the maximal length of word in the input, ***m*** is constant.
   b. Represent each letter as a digit in base 27, i.e., in range [0..26], where a=1, b=2,…, z=26 - O(nm)=O(n)
   c. Words with k < m letters will have succeeding zeros added - O(nm)=O(n)
2. **Solve P2 on the transformed input:**
   a. Execute radix sort where d=*m* and b=27 - O(d(n+b))=O(n)
3. **Translate output of P2 to output of P1:**
   a. Change the numbers sequences back to words - O(nm)=O(n)

Total run time is O(n).

Example:
Lexicographical-sort input: {blue, red, green}
Radix-sort input: { (2,12, 21,5,0), (18,5,4,0,0), (7, 18,5,5,14) }
Radix-sort output: { (2,12, 21,5,0), (7, 18,5,5,14) , (18,5,4,0,0) }
Lexicographical-sort output: {blue, green, red}

Question 6: Sorting input with anomalies

Given an array A of n positive integer numbers. All the numbers **except for ten** are in the range [10, 10n]. Design an algorithm to sort an array A in O(n) time in the worst case.

Solution:

1. Traverse the array and remove 10 numbers that are not in the range into a help array B. *//O(n) time*
2. Counting sort on the array A. *//O(n-10 +10n)=O(n) time*
3. Some sort on the array B. *//O(1) time*
4. Merge A and B. *//O(n) time*