CSCI-UA.0310-004/005 Basic Algorithms March 22, 2015

Solutions to Problem 1 of Homework 5 (6 points)

Name: Jason Yao *Due: Wednesday, March 4*

(a) Assume you are given an array of n integers with many duplications, so that you know that there are at most $\log n$ distinct elements in the array. Show how to sort this array in time $O(n \operatorname{loglog} n)$.

Solution:

Let $k = \log n$

Let each node contain:

- Data value

- Number of times it occurs

By implementing HEAP-SORT, it is possible to get $2 * n * logk$

```
fastDuplicateSort(A[])
n = A.length
for i = 1 to n
  INSERT(A[i])
endfor

for j = 1 to n
  EXTRACT-MAX()
endfor
END fastDuplicateSort
```

Runtime Analysis:

As each for loop is running in $T(n) = \log k + n \log k$, that gives a total runtime of $T(n) = 2(\log k + n \log k)$, or $T(n) = \Theta(\log k + n \log k)$

$T(n) = \Theta(n \log k)$

$T(n) = \Theta(n \log \log n)$ ☐

(b) Assume you are given an array of n integers in the range $\{1, \ldots, (\log n)^{\log n}\}$. Show how to sort this array in time $O(n \operatorname{loglog} n)$.

Solution: **************** INSERT YOUR SOLUTION HERE *************** ☐

CSCI-UA.0310-004/005 Basic Algorithms March 22, 2015

Solutions to Problem 2 of Homework 5 (9 points)

Name: Jason Yao *Due: Wednesday, March 4*

For each example choose one of the following sorting algorithms and carefully justify your choice: HeapSort, RadixSort, CountingSort. Give the expected runtime for your choice as precisely as possible. If you choose Radix Sort then give a concrete choice for the basis (i.e. the value of "r" in the book) and justify it. (**Hint**: We assume that the array itself is stored in memory, so before choosing the fastest algorithm, make sure you have the space to run it!)

(a) Sort the length 2^{16} array A of 128-bit integers on a device with 100MB of RAM.

Solution:

16 Bytes per integer, Memory already in use: $(2^{16}) * 16 = 1.048576MB$

98.951424MB remains, compared to 1.048576 MB that was used to load the integers to memory

Maximum represented value: $2^{128} - 1$, which is too large to implement Radix or Counting Sort, since the k value (maximum value) is too large, and would not be efficient.

Thus, HEAPSORT would be the recommended sorting algorithm, which is an in-place sorting algorithm, with a running time of $T(n) = n \log n$. □

(b) Sort the length 2^{24} array A of 256-bit integers on a device with 600MB of RAM.

Solution:

32 Bytes per integer, Memory already in use: $(2^{24}) * 32 = 536.870912MB$

63.129088MB is available for the sorting, meaning that an in-place, or near in place sort would be required due to memory constraints.

Maximum represented value: $2^{256} - 1$, which would be too big to implement Radix Sort or Counting Sort, since the k value (maximum value) is too large, and would not be efficient.

Thus, HEAPSORT would be the recommended sorting algorithm, which is an in-place sorting algorithm, with a running time of $T(n) = n \log n$. □

(c) Sort the length 2^{16} array A of 16-bit integers on a device with 1GB of RAM.

Solution:

2 Bytes per integer, Memory already in use: $(2^{16}) * 2 = 0.131072MB$

Over 999.869MB remains available for sorting, a huge amount of memory, especially considering the small amount of memory the integers take up.

Maximum represented value: $2^{16} - 1 = 65,535$, which is small enough that we can implement Radix Sort and Counting Sort.

In this case, as Counting Sort would be which would run in $T(n) = O(k + n)$, in which k is the number of digits each integer has.

In this case, $T(n) = O(16 + n)$, $T(n) = O(n)$ $\square$

CSCI-UA.0310-004/005 Basic Algorithms March 22, 2015

Solutions to Problem 3 of Homework 5 (16 points)

Name: Jason Yao *Due: Wednesday, March 4*

Recall that there exists a trivial algorithm for searching for a minimal element of an array of size n that needs exactly $(n-1)$ comparisons. The purpose of this problem is to prove that this is optimal solution.

You can assume that elements of the input array are distinct. As usual, you can represent any comparison-based algorithm for Min-Element problem as a binary decision tree, whose internal nodes are labeled by $(i : j)$ (meaning "compare $A[i]$ and $A[j]$"), and a left child is chosen if and only iff $A[i] < A[j]$ (recall, we assume distinct elements). And the leaves are labeled by the index i meaning that the smallest element in the arrange is $A[i]$.

(a) (4 points warm-up) Show that the naive "counting leaves" application of decisional tree method (ala sorting lower bound) will only give a very suboptimal lower bound $\Omega(\log n)$ for the Min-Element problem.

Solution:

As the height of a binary decision tree is given as $\log n$, and getting to the leaves (finding the lower bound) would take that amount of node traversals, the counting leaves of decisonal trees would end up giving $\Omega(\log n)$ □

(b) (4 points) Here we will try to enrich the decisional tree by adding some additional info $S(v)$ to every vertex v. Precisely, $S(v)$ is the set of all indices i which are "consistent" with all the comparisons made from the *root* to v (excluding v itself), where "consistent" means that there exists an array A whose minimum is $A[i]$ and the node v is reached on input i. For example, if v is any (reachable) leaf labeled by i, then $S(v) = \{i\}$ (as otherwise the algorithm would not be correct). Assuming the root node *root* is labeled $(i : j)$, describe $S(\text{root})$, $S(\text{root.left})$ and $S(\text{root.right})$.

(**Hint**: What element is impossible to be minimal if you know that $A[i] < A[j]$?)

Solution:

The element that is to the right of the root S(root) would contain the set of all possible combinations to find the minimal value

S(Root.LEFT) contains the set of (n-1) comparisons, as there is now one less element that needs to be compared.

S(Root.RIGHT) contains the set of just 1 element, which was the element j such that it is greater than i. □

(c) (4 points) Generalize part (b): show that for every vertex v we have: $S(v.\text{left}) = S(v) \setminus \{\ldots\}$ and $S(v.\text{right}) = S(v) \setminus \{\ldots\}$ (you need to fill dots on your own).

Solution:

For every vertex v, by definition, has another two vertices v.left and v.right, such that S(v.left) = {the previous i}, and S(v.right) = {the previous j} $\square$

(d) (4 points) Use (c) to prove that in any valid decision tree for Min-Element, *any* leaf (which is stronger than *some* leaf) must have depth at least $(n - 1)$. (**Hint**: Think about $|S(v)|$ as v goes from root to this leaf.)

Solution:

As you have to do n - 1 comparisons due to having to compare against all values in S, the depth of all leaves in the decision tree must be at least n - 1, as $|S(v)|$ decreases from all possible combinations to 1 by the time it reaches a leaf.

$\square$

CSCI-UA.0310-004/005 Basic Algorithms March 22, 2015

Solutions to Problem 4 of Homework 5 (10 points)

Name: Jason Yao *Due: Wednesday, March 4*

We will call an array $B[1 \ldots n]$ *a roller coaster* if $B[1] < B[2]$, $B[2] > B[3]$, $B[3] < B[4]$ and so on. More formally: $B[2i] > B[2i+1]$ and $B[2i-1] < B[2i]$ for all i.

Give a linear algorithm that any given array A with n distinct elements transforms into a roller coaster array B. Namely, B must contain exactly the same n distinct elements as A, but must also be a roller coaster. (**Hint**: A median may be useful here.)

Solution:

```
makeRollerCoaster(A[])

n = A.length
// Sorts the array A
A = mergeSort(A)

// Takes an element from the end and the front, and builds a new array
   that alternates mins and maxes

for i = 1 to n/2, i += 2
  B[i] = A[i]
  B[i + 1] = A[n - i]
endfor

END makeRollerCoaster
```

□