CSCI-GA.1170-001/002 Fundamental Algorithms October 12, 2019

Solutions to Problem 1 of Homework 5 (7 points)

Name: jingshuai jiang (jj2903) *Due: 5 pm Thursday, October 10*

Collaborators: NetID1, NetID2

We will call an array $B[1 \dots n]$ *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. Briefly argue correctness and the run time of your algorithm. (**Hint**: A median may be useful here.)

Solution:

```
1 ROLLER-COASTER($B$)
2     $for\ i = 1\ to\ n-1$
3         if i is an odd
4             if B[i] > B[i+1]
5                 swap(B[i],B[i+1])
6         if i is an even
7             if B[i] < B[i+1]
8                 swap(B[i],B[i+1])
9     return B
```

Since we have to satisfy the condition of $B[2i] > B[2i+1]$ and $B[2i-1] < B[2i]$ for all i we just scan the whole array from the beginning and check it with the following element. And if they do not obey the rule, just swap them. Because we have make sure that $B[2i-1] < B[2i]$ if $B[2i] < B[2i+1]$ when we swap $2i$ and $2i+1$. Then the $B[2i]$ after the swap will also be bigger than $B[2i-1]$. And it is the same about an odd number with its precursor and its successor.

The running time will be $O(n)$ □

CSCI-GA.1170-001/002 Fundamental Algorithms October 12, 2019

Solutions to Problem 2 of Homework 5 (8 points)

Name: jingshuai jiang (jj2903) *Due: 5 pm Thursday, October 10*

Collaborators: NetID1, NetID2

You receive a sales call from a new start-up called *MYPD* (which stands for "Manage Your Priorities... Differently"). The MYPD agent tells you that they just developed a ground-breaking *comparison-based* priority queue. This queue implements *Insert* in time $\log_2(\sqrt{n})$ and *Extract_max* in time $\sqrt{\log_2 n}$. Explain to the agent that the company can soon be sued by its competitors because either (1) the queue is not comparison-based; or (2) the queue implementation is not correct; or (3) the running time they claim cannot be so good. To put differently, no such comparison-based priority queue can exist.
(**Hint**: Be very precise with the constant c s.t. $\log_2(n!) \approx cn \log n$.)

Solution:

Conclusion: The Insersion time for comparsion-based priority-queue should be $O(h) = O(logn)$ We can view comparison-based insertion abstractly in terms of decision tree. In a decision tree, we annotate each inernal node by (insertion number : i) for some i in the range $1 \leq i \leq n$ where n is the number of the array length before the insertion .Then the leaves will be a permutation that present the array after the insertion. actually insertion a number into n-size array will have n+1 kinds of insertions. Consider the tree with height h, have l leaves,because each possible insertion is a leaf then we get

$$n + 1 \leq l \leq 2^h$$

and

$$h \geq log(n + 1) = O(logn)$$

Which means the running time of insertion should at least be $O(logn)$ not $O(log\sqrt{n})$ □

CSCI-GA.1170-001/002 Fundamental Algorithms October 12, 2019

Solutions to Problem 3 of Homework 5 (11 points)

Name: jingshuai jiang (jj2903) *Due: 5 pm Thursday, October 10*

Collaborators: NetID1, NetID2

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

Solution:

The comparison-based decision tree to get the minimum value of an array will have l leaves, and the height of h

and the binary tree of height of h can have at most 2^h leaves and the $l$leaves must cover all the possible situations that $minimum = A[i]$ and we get

$$n \leq l \leq 2^h$$

then

$$h > logn$$

which means that the lower bound is $\Omega(logn)$ □

(b) (3 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:

$S(root) = all\ the\ index\ from\ 1\ to\ n$

$S(root.left) = all\ the\ index\ from\ 1\ to\ n\ except\ for\ j$

$S(root.right) = all\ the\ index\ from\ 1\ to\ n\ except\ for\ i$

□

(c) (3 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:

$S(v.\text{left}) = S(v) \setminus \{v.j\}$

$S(v.\text{right}) = S(v) \setminus \{v.i\}$ □

(d) (3 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: according to problem 5-3-(c) we can see that by using comparison-based min-element algorithm,we can only exclude one element from the whole index from 1 to n for one comparison. In order for the $S(v)$ to have only one element. we have to perform at least $n-1$ comparisons. Leaf nodes are those nodes with $|S(v)| = 1$. Any in order for leaf nodes to have only one element in $s(v)$ we should experience at least $n-1$ comparsions, which is a path all the way down from the root to the leaf. And we have n-1 nodes to wipe out an element on this way. so it must have depth at least $n-1$ □

CSCI-GA.1170-001/002 Fundamental Algorithms October 12, 2019

Solutions to Problem 4 of Homework 5 (9 points)

Name: jingshuai jiang (jj2903) *Due: 5 pm Thursday, October 10*

Collaborators: NetID1, NetID2

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

Solution:

The length 2^{16} array A of 128-bit integers actually takes about $2^{23} bits = 2^{20} Bytes = 1MB$ RAM

If it is counting sort then we should also allocate the memory for the counter which is an array whose length is 2^{128} and each cell of this array should have take 16-bits to represent the numbers and there is no space for this giant array.

A is radixsort. $r = logn = 16$

The running time of A is $\Theta(\frac{b}{r} \cdot (n + 2^r)) = \Theta(2^{20})$

□

(b) (3 points) Sort the length 2^{24} array A of 256-bit integers on a device with 600MB of RAM.

Solution:

The length 2^{24} array A of 256-bit integers actually takes about $2^{32} bits = 2^{29} Bytes = 512MB$ RAM

If it is counting sort then we should also allocate the memory for the counter which is an array whose length is 2^{256} and each cell of this array should have take 24-bits to represent the numbers and there is no space for this giant array. And if we choose radixsort and using CountingSort as the stable sort of radixsort it also does not have enough space.

Then B is HeapSort.

The running time of A is $\Theta(nlogn) = \Theta(24 \cdot 2^{24})$

□

(c) (3 points) Sort the length 2^{16} array A of 16-bit integers on a device with 1GB of RAM.

Solution:

The length 2^{16} array A of 16-bit integers actually takes about $2^{20} bits = 2^{17} Bytes = 128KB$ RAM

If it is counting sort then we should also allocate the memory for the counter which is an array whose length is 2^{16} and each cell of this array should have take 16-bits to represent the numbers and there is enough space for this array.

Then C is CountingSort.

The running time of A is $\Theta(n + k) = \Theta(2^{16} + 2^{16}) = \Theta(2^{17})$

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 12, 2019

Solutions to Problem 5 of Homework 5 (6 (+9) points)

Name: jingshuai jiang (jj2903) *Due: 5 pm Thursday, October 10*

Collaborators: NetID1, NetID2

Let us say that a number x is *c-major* for an n-element array A, if more than n/c elements of A are equal to x.

(a) (6 points) Give $O(n)$-time algorithm to find all 2-major elements of A. How many could there be? Briefly argue correctness of your algorithm. (**Hint**: You may make calls to the SELECT algorithm discussed in class and textbook.)

Solution:

There could only be one such element. Because more than $\frac{n}{2}$ elements of A should be equal to x.

If there exist such element. Since there are more than $\frac{n}{2}$ elements of A equal to x, then the median should be this element.

Then we scan the whole array to see if there exist such elements. If over half of this array is this elements. Then return this element else return null.

```
2-MAJOR ELEMENTS($A$)
    a = Select(A, n/2)
    count = 0
    $for$ $j = 1$ $to$ $n$
        if $A[j] == a$
            count++
    $if$ $count \geq \frac{n}{2}$
        return a
    return null
```

□

(b) (9 points) [**Extra Credit**] Give $O(cn)$-time algorithm to find all c-major elements of A. How many could there be? Briefly argue correctness of your algorithm. (**Hint**: You may make calls to the SELECT algorithm discussed in class and textbook.)

Solution:

There could be at most c-1 such elements

```
C-MAJOR ELEMENTS($A$)
    $for\ i = 1\ to\ c$
        a[i] = Select(A,$\frac{i \cdot n}{c}$)
        count[i] =0
    $for\ j = 1\ to\ n$
        $for\ i = 0\ to\ c$
            if $A[j] == a[i]$
                count[i]++
    $for\ i = 1\ to\ c$
        $if\ count[i] \geq \frac{n}{c}$
            list.append(a[i])
    return list
```

If the element is a c-major element, it should cover a length at least for $\frac{n}{c}$. These elements should remain the same in this length. And this length will cover at least one $\frac{n}{c} \cdot i\ th$ point of this array. Then we just need to get these and check whether these $\frac{n}{c} \cdot i\ th$ points are actually c-majored. $\square$