CSCI-GA.1170-001/002 Fundamental Algorithms October 14, 2015

Solutions to Problem 1 of Homework 5 (8 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, October 13*

Consider a sorted array A of n elements and two integers x and y not in the array, with $x \leq y$. A comparison based algorithm computes how many elements in A are less than both x and y, how many elements are between x and y and how many are bigger than both x and y. What is the best lower bound (precise answer, not asymptotic) you can prove (using the decision tree technique) for the time complexity of the algorithm?

Solution:

Each leaf node in the decision tree can be viewed as a sequence of elements of A that are divided into 3 parts.

- The first part corresponds to all those elements of A that are lesser than both x and y (could be empty)
- The second part corresponds to all those elements of A that are in between x and y (could be empty)
- The third part corresponds to all those elements of A that are greater than both x and y (could be empty)

Therefore the number of leaf nodes in the decision tree will be equal to the number of ways in which we can place two partitions among the n sorted elements of A which can be viewed as selecting two elements from $n+2$ elements $= \binom{n+2}{2} = \frac{(n+2)(n+1)}{2}$. Let it be l

The lower bound on the height of the decision trees in which each permutation appears as a reachable leaf is a lower bound on the running time of any comparison sort algorithm. So let h be the height of the binary tree and since a binary tree of height h can have no more than 2^h leaves, we have

$$l \leq 2^h \implies \frac{(n+2)(n+1)}{2} \leq 2^h \implies h \geq \log_2 \frac{(n+2)(n+1)}{2} = \log_2(n+2)+\log_2(n+1)-1 \implies h = \Omega(\log n)$$

Therefore, the precise lower bound is $\log_2(n+2) + \log_2(n+1) - 1$

$\square$

CSCI-GA.1170-001/002 Fundamental Algorithms October 14, 2015

Solutions to Problem 2 of Homework 5 (15 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, October 13*

Assume that we are given n bolts and n nuts of different sizes, where each bolt exactly matches one nut. Our goal is to find the matching nut for each bolt. The nuts and bolts are too similar to compare directly; however, we can test whether any nut is too big, too small, or the same size as any bolt.

(a) (4 points) Prove that in the worst case, $\Omega(n \log n)$ nut-bolt tests are required to correctly match up the nuts and bolts.

Solution:

We can model the algorithm for the matching nuts and bolts problem using a decision tree. The tree will be a ternary tree as every comparison can lead to three possible outcomes, less than, greater than or equal to. Therefore, if the height of the tree is h, there will be atmost 3^h number of leaf nodes. We know that the height of the tree corresponds to the worst case number of comparisons made by the algorithm, which is the lower bound of the running time.

Now consider the input of every permutation of the bolts. Now each of this permutation has to map to a distinct leaf node. Let's assume that any two different inputs from this permutation map to the same leaf node, it means that the algorithm applied to both of these inputs the same permutation with respect to the nuts and the algorithm didn't match them correctly to the corresponding permutation of nuts. Therefore, out assumption is wrong. Therefore, every permutation of bolts map to a unique leaf node. This means there are at least $n!$ leaf nodes. Let the number of leaves be l

$$l \geq n!,\ 3^h \geq l \implies 3^h \geq n! \implies h \geq \log_3 n! \implies h = \Omega(n \log n)$$

$\square$

(b) (6 points) Prove that in the worst case, $\Omega(n + k \log n)$ nut-bolt tests are required to find k arbitrary matching pairs. (Hint: prove two separate lower bounds: $\Omega(n)$ and $\Omega(k \log n)$.)

Solution:

We can model the algorithm for matching k arbitrary matching pairs using a decision tree.The tree will be a ternary tree as every comparison can lead to three possible outcomes, less than, greater than or equal to. Therefore, if the height of the tree is h, there will be atmost 3^h number of leaf nodes. We know that the height of the tree corresponds to the worst case number of comparisons made by the algorithm, which is the lower bound of the running time.

For any input of bolts we need to match k arbitrary pairs of nuts with bolts. Therefore, first select any k nuts $= \binom{n}{k}$ and these n can be permuted. Thus, total number of leaf nodes will be at least $n! \times \binom{n}{k} = \frac{n!}{(n-k)!}$. Let l be the number of leaf nodes

$$l \geq n!,\ 3^h \geq l \implies 3^h \geq \frac{n!}{(n-k)!} \implies h \geq \log_3 \frac{n!}{(n-k)!} \implies h = \Omega(n + k\log n)$$

□

(c) (5 points) Give a randomized algorithm that runs in expected time $O(n)$ and finds the k-th largest nut given any integer k. You may assume that it is possible to efficiently sample a random nut/bolt.

Solution:

Algorithm

Let N be nuts and B be bolts. The idea is similar to Quickselect. So the expected running time will be $O(n)$

- Take a random bolt from the set of bolts
- Using this bolt, partition the nuts into two parts, which is smaller than this bolt and larger than this bolt i.e $N[1 \dots p-1]$ and $N[p+1 \dots n]$
- If the matched nut with the above bolt is the k^{th} largest nut then return it
- Else, Using this nut to partition the bolts into two parts, which is smaller than this nut and larger than this nut i.e $B[1 \dots p-1]$ and $B[p+1 \dots n]$
- If $N[p]$ is smaller than the k^{th} largest nut, then recursively try to find k^{th} largest nut in the $N[p+1 \dots n]$ partition
- If $N[p]$ is larger than the k^{th} largest nut, then recursively try to find k^{th} largest nut in the $N[1 \dots p-1]$ partition

Psuedocode

In the below psuedocode, Partition procedure partitions the nuts and bolts into two parts, the first part contains all the nuts/bolts smaller than the pivot and the second part contains all the nuts/bolts larger than the pivot. The pivot is provided as an argument. When partitioning the nuts, a the last bolt is taken as pivot and when partitioning bolts, the last nut is taken as pivot

```
Algorithm: FIND-NUT($N, B, low, high, k$)
pivot = PARTITION($N, B, low, high, B[high]$)
if $pivot$ is $k$ then
    Return $N[pivot]$
end
PARTITION($B, N, low, high, N[high]$)
if $pivot < k$ then
    FIND-NUT($N, B, pivot + 1, high, k$)
end
else if $pivot > k$ then
    FIND-NUT($N, B, low, pivot - 1, k$)
end
```

Algorithm 1: Algorithm to find k^{th} largest nut in $O(n)$ time

The algorithm is similar to Quickselect, taught in class. Therefore the expected running time is $O(n)$ $\square$

CSCI-GA.1170-001/002 Fundamental Algorithms October 14, 2015

Solutions to Problem 3 of Homework 5 (12 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, October 13*

We will consider a variant of the counting sort algorithm that sorts an n element array A whose elements are integers between 1 and k.

(a) (4 pts) Recall the code of standard counting sort.

```
COUNTINGSORT($A, B, k,$)
1 let $C$ be a new array
2     For $i = 1$ to $k$
3         $C[i] = 0$
4     For $j = 1$ to $n$
5         $C[A[j]] = C[A[j]] + 1$
6     For $i = 1$ to $k$
7         $C[i] = C[i] + C[i-1]$
8     For $j = n$ to 1
9         $B[C[A[j]]] = A[j]$
10        $C[A[j]] = C[A[j]] - 1$
11 Return $B$
```

After the run of the counting sort, somebody accidentally erased both array A and B, but only left array C. Design an $O(n)$ procedure RECOVERBFROMC that still reconstructs the sorted array B from C. Why does it only work when sorting integers?

Solution:

Psuedocode

```
Algorithm: RecoverBfromC(C, k)
B ← NewArray($n$)
for $j$ ← $k$ to 0 do
    if $j$ is 0 then
        while C[j] $\neq$ 0 do
            $B[C[j]-1] = j$
            $C[j] = C[j]-1$
        end
    end
    else
        while C[j] $\neq$ C[j-1] do
            $B[C[j]-1] = j$
            $C[j] = C[j]-1$
        end
    end
end
Return B
```

Algorithm 2: Algorithm to recover B from C in $O(n)$ time

Counting Sort can be used only on integers because the values of A are used as the indices of C and as the indices can only be integers (and not floating point numbers).However, it is possible to use it on characters but they have to be mapped to integers so that they can be used as indices. □

(b) (4 pts) Now, consider the following variant of the Counting Sort algorithm that sorts the n-element array A whose elements are integers between 1 and k.

```
1 CountingSortFast($A, k, n$)
2     For $i = 1$ to $k$
3         $C[i] = 0$
4     For $j = 1$ to $n$
5         $C[A[j]] = C[A[j]] + 1$
6     $b = 1$
7     For $i = 1$ to $k$
8         For $j = 1$ to $C[i]$
9             $A[b] = i$
10            $b = b + 1$
11    Return $A$
```

Formally argue the correctness of CountingSortFast.

Solution:

Proof of Correctness

Base Case

$n = 1$, Let $A[1] = k$

The first two *for* loop sets $C[1 \dots k-1] = 0$ and $C[k] = 1$ The third *for* loop sets $A[1] = k$ and then returns A.

Therefore, A is sorted. Hence the base case true.

Induction Hypothesis

At the end of i^{th} iteration of the *for* loop, A is sorted until $\sum_{j=1}^{i} C[j]$ terms, i.e $A[0 \dots b_i]$ is sorted where b_i is the value of b at the end of i^{th} iteration

Induction Step

At the end of the i^{th} iteration, b is increased by one value. Now in the $(i+1)^{th}$ iteration, the value of b is one more than that of the previous iteration i.e $b_{i+1} = b_i + 1$

At the end of i^{th} iteration, $A[b_i] = i$. Therefore, in the beginning of $(i+1)^{th}$ iteration, $A[b_{i+1}] = i + 1 \implies A[b_i + 1] > A[b_i]$

By the induction hypothesis we already know that $A[0 \dots b_i]$ is sorted. Therefore, $A[0 \dots b_{i+1}]$ is sorted ($\because A[b_i + 1] > A[b_i]$)

Hence by Induction, we can conclude that the given algorithm is correct

$\square$

(c) (4 pts) Notice that while CountingSort makes in time $3n + 2k$ array assignments, but CountingSortFast makes only in time $2n + k$ array assignments. Further CountingSortFast does not use an extra array B used by sc CountingSort. Explain what the problem with CountingSortFast. In which realistic use cases would one prefer slightly slower CountingSort? Justify your answer.

Solution:

The CountingSort procedure has the property of being stable. i.e It always preserves the relative order of equal elements even after the sorting. If two elements $A[i]$ and $A[j]$ such that $A[i] = A[j]$ and $i < j$ then after the elements are sorted, $A[i]$ will appear before $A[j]$ in B. This is because we iterated through A backwards and we decrement $C[i]$ everytime we see i, the relative ordering of duplicate elements is preserved whereas CountingSortFast is not stable, the relative ordering of the same elements may or may not be preserved

For radix sort to work, the sorting aglorithm performed each time from the least significant digit to the most significant digit has to be stable. So if CountingSortFast is used, radix sort might not work

For example, consider that we have a list of first names and last names. Now our task is to sort these names by last name and then by the first. We could use a stable sorting algorithm on the first name. Now all the names are sorted by their first names. Next, we stable sort by the last names. After these two sorts, the names are primarily sorted by their last names. In case, the last names are same, they get sorted by their first names as the relative ordering is preserved. If we use CountingSortFast in this case, the names might not get sorted because it doesn't guarantee the relative ordering in the case of same last names

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 14, 2015

Solutions to Problem 4 of Homework 5 (9 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, October 13*

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:

Given, $n = 2^{16}$

Memory needed to store array of length 2^{16} of 128-bit integers $= 2^{23}$
RAM capacity is 100 MB $= 100 \times 2^{10}$ KB $= 100 \times 2^{20}$ B $= 100 \times 2^{23}$ bits

Time taken to sort the given array using heap sort $= O(n \log n) = O(2^{16}.16) = O(2^{20})$
Extra space required using heap sort $= O(1)$

$b = 128 > 16 = \log n \therefore$ choose $r = \log_n = 16$
Time taken to sort the given array using radix sort $= O(2(bn)/(\log n)) = O(256 \times 2^{16}/16) = O(2^{20})$
Extra space required using radix sort $= O(n + 2^r) = O(2^{16} + 2^{16}) = O(2^{17}))$

Time taken to sort the given array using counting sort $= O(n + k) = O(2^{16} + 2^{128})$
Extra space required using counting sort $= O(n + k) = O(2^{16} + 2^{128})$

The running time of radix sort and heap sort is the lowest of the three and also there is sufficient space in RAM to allocate the extra space used by radix sort and heap sort. Hence it would be ideal to use heap sort or radix in this case

□

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

Solution:

Given, $n = 2^{16}$

Memory needed to store array of length 2^{24} of 256-bit integers $= 2^{32}$ bits $= 512 \times 2^{23}$ bits
RAM capacity is 600 MB $= 600 \times 2^{10}$ KB $= 600 \times 2^{20}$ B $= 600 \times 2^{23}$ bits

Time taken to sort the given array using heap sort $= O(n \log n) = O(2^{24}.24) = O(24 \times 2^{24})$
Extra space required using heap sort $= O(1)$

$b = 256 > 24 = \log n \therefore$ choose $r = \log_n = 24$
Time taken to sort the given array using radix sort $= O(2(bn)/(\log n)) = O(512 \times 2^{24}/24) =$

$O(64 \times 2^{24}/3)$
Extra space required using radix sort $= O(n + 2^r) = O(2^{24} + 2^{24}) = O(2^{25}))$

Time taken to sort the given array using counting sort $= O(n + k) = O(2^{24} + 2^{256})$
Extra space required using counting sort $= O(n + k) = O(2^{24} + 2^{256})$

It is quite clear that there is no sufficient memory in RAM to allocate the extra space required for radix sort and counting sort. So the only option here is to use heap sort

□

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

Solution:

Given, $n = 2^{16}$

Memory needed to store array of length 2^{16} of 16-bit integers $= 2^{20}$ bits
RAM capacity is 1 GB $= 2^{10}$ MB $= 2^{20}$ KB $= 2^{30}$ B $= 2^{33}$ bits

Time taken to sort the given array using heap sort $= O(n \log n) = O(2^{16}.2^4) = O(2^{20})$
Extra space required using heap sort $= O(1)$

$b = 16 = 16 = \log n \therefore$ choose $r = \log_n = 16$ Time taken to sort the given array using radix sort $= O(2(bn)/(\log n) = O(32 \times 2^{16}/16) = O(2^{17})$
Extra space required using radix sort $= O(n + 2^r) = O(2^{16} + 2^{16}) = O(2^{17}))$

Time taken to sort the given array using counting sort $= O(n + k) = O(2^{16} + 2^{16}) = O(2^{17})$
Extra space required using counting sort $= O(n + k) = O(2^{16} + 2^{16}) = O(2^{17})$

counting sort and radix sort takes the least time to sort the given array and also there is sufficient space in the RAM to allocate the extra space that counting sort and radix sort takes. Hence it would be ideal to use counting sort or radix sort in this case(In this case, radix sort = counting sort)

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 14, 2015

Solutions to Problem 5 of Homework 5 (15 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, October 13*

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 pts) Give $O(n)$-time algorithm to find all 2-major elements of A. How many could there be?

Solution:

The key idea here is that the 2-majority element remains preserved when a pair of distinct elements are canceled out from the array.

Let α is the 2-majority element in the array. Now if two distinct elements β and γ are discarded, the array length now becomes $n-2$. Therefore α is a 2-majority element. If two distinct elements α and β are discarded the array length becomes $n-2$ and there are $> n/2-1$ occurrences of α. Therefore α is a 2-majority element.

Let there be x number of 2 majority elements in A. Therefore, there are at least $xn/2$ elements in A

$\therefore xn/2 < n \implies x < 2 \implies$ There can be only one 2-majority element in A

Psuedocode

```
Algorithm: Find-2-Majority(A)
count ← 0
for i ← 0 to n-1 do
    if count is 0 then
        x ← A[i]
        count ← 1
    else if x ≠ A[i] then
        count ← count − 1
    Else count ← count + 1
end
num ← Count-Occurrences(x, A)
if num > n/2 then
    Return x
else
    Return NotFound
```

Algorithm 3: Algorithm to find 2-majority element of A in $O(n)$ time

In the above psuedocode, during i^{th} iteration, we compare $A[i-1]$ with x and cancel both if they are different, and increment count otherwise.

So if $count = 0$, then all elements upto $A[i-1]$ would have been eliminated through distinct-elements pair formations. If $count > 0$, then $\{x, \ldots \text{ count times } \ldots, x, A[i], \ldots A[n-1]\}$ elements would have still survived at the end of the i^{th} iteration. Therefore we are effectively canceling out the distinct elements. Thus at the end of the *for* loop, if there is a 2-majority element it survives. We are doing a linear scan of the array in the for loop and in the last step we find the number of occurrences of x in A which also takes linear time. Therefore running time of the algorithm is $O(n)$ □

(b) (9 pts) Give $O(cn)$-time algorithm to find all c-major elements of A. How many could there be?

Solution: Following the same idea, that if c distinct elements of the array are canceled out the c majority element still remains preserved in the array.

In the below psuedocode, we maintain two arrays *temp* and *count* of length k. We keep iterating over the n elements of the given array and if it matches with any of the k elements in *temp*, we increase the *count* of that element. If none of the elements of *temp* matches, we decrease the count of every element (i.e we are canceling out the distinct elements). If there is an empty slot in *temp* (i.e *count* of that element is 0) then we place the element at that position and set it's *count* to 1. At then end, we individually check for each element of *temp*, if it is a c-majority element.

Let there be x number of c majority elements in A. Therefore, there are at least xn/c elements in A

$\therefore xn/c < n \implies x < c \implies$ There can be at most $c-1$, c-majority elements in A

Psuedocode

```
Algorithm: Find-c-Majority(A)
$temp, count \leftarrow$ NewArray($c$)
Initialize $count$ to 0
for $i \leftarrow 0$ to $n-1$ do
    for $j \leftarrow 0$ to $c-1$ do
        if $temp[j]$ is $A[i]$ then
            $count[j] \leftarrow count[j] + 1$
            break
        end
        if $j$ is $c-1$ then
            for $p \leftarrow 0$ to $c-1$ do
                if $count[p]$ is $0$ then
                    $temp[p] \leftarrow A[p]$
                    $count[p] \leftarrow 1$
                    break
                end
            end
            if $p$ is $c-1$ then
                for $p \leftarrow 0$ to $c-1$ do
                    $count[p] \leftarrow count[p] - 1$
                end
            end
        end
    end
end
for $i \leftarrow 0$ to $c-1$ do
    $num \leftarrow$ NumberOfOccurrences($temp[i], A$)
    if $num > n/c$ then
        $temp[i]$ is a c-major element
    end
end
```

Algorithm 4: Algorithm to find c-majority element of A in $O(nk)$ time

Time Complexity

The first outer *for* loop runs for n times and all the nested *for* loops run for c times. Therefore, the first phase takes $O(nc)$

The second *for* loop runs for c times and in each iteration we find the number of occurrences of the element of *temp* in A. It takes $O(n)$ time. Therefore, the second phase takes $O(nc)$ time. Hence, the total running time of the algorithm is $O(nc)$ □