Mid Semester Test, 16 September 2015
ESO207: Data Structures and Algorithms

Time 120 minutes Max Marks 50

Please clearly mark the question number and put a horizontal line after each answer. You may use any result proven in the class without proving again. But make sure that you state the result first.

Question 1. [Marks 2+2+2+2+2+2+2+2+2+2].

(A) For any integer $k > 1$, show that $\log_k n = \Omega(\log_2 n)$.

(B) Is $n^n = \Theta(n!)$? Justify your answer. You may use Sterling's approximation for $n!$ which is $n! \approx \sqrt{2\pi n}.(n/e)^n.(1+O(1/n))$.

(C) Given a sequence of integers $a_1, a_2, \ldots, a_n$ where each $a_i \in \{1,2,\ldots,k\}$. Assuming that k is much smaller that $\log_2 n$, describe an efficient (in terms of time) algorithm to sort the sequence. Also give its time and space complexity in terms of n and k.

(D) A person, standing on a north-south national highway at a place P, wants to walk up to the nearest bus stop. But he does not know in which of the two directions is the nearest bus stop nor does he know how far is it. He plans the following strategy: walk 100 meters to north and return to P if no bus stop is found. Then do the same in the southern direction. Then go 200 meters in north and return and go 200 meters to south and return. Keep doubling the distance each time till the bus stop is located. What is the maximum distance will he walk if the bus stop is at a distance D from point P?

(E) Solve the recurrence relation $T(n) = 4.T(n/3) + n^{4/3}$ using Master's theorem. Note $\log_3 4 = 1.2618595$.

(F) Consider the problem of deletion of the maximum element of a max-heap. Suppose it takes c_1 time to read the root element, output and to write the last element into the root node. Also suppose it takes c_2 time to compare the contents of a parent and its child nodes (two comparisons) and perform one swap. Which of the following expressions gives a tight upperbound for the time to delete the max element and re-fix the heap which originally had $n > 1$ elements?
(i) $c_1 + \lfloor \log_2 n \rfloor$ (ii) $c_1 + \lfloor \log_2 (n-1) \rfloor$ (iii) $c_1 + \lceil \log_2 n \rceil$ (iv) $c_1 + \lceil \log_2 (n-1) \rceil$

(G) Which of the following gives the tightest time complexity of heapsort? Justify your answer.
(i) $O(n \log n)$, (ii) $O(\log n!)$, (iii) $O(n(\log n)^2)$, (iv) $O(n^2)$.

(H) In a hashing scheme with chaining, the hash function is $h(k) = k(mod 9)$. Starting from the empty table following numbers are entered in order: $59, 79, 34, 41, 58, 12, 23, 16, 88, 25$. Draw a diagram showing the linked lists for each slot starting from the slot pointed by $T[i]$ for each i.

(I) Following code is for Insertion Sort in increasing order. It sorts the content of the array A in the range $1 : n$.

```
for j := 2 to n do
    key := A[j];
    i := j − 1;
    while i > 0 and A[i] > key do
        A[i + 1] := A[i];
        i := i − 1;
    end
    A[i + 1] := key;
end
```

Assume that array contains integers $\{1,2,\ldots,n\}$. For which initial permutation the algorithm will make the maximum number of comparisons? Justify.

(J) If all $n!$ permutation are equally probable as input to the above Insertion Sorting algorithm, then what will be the expected (average) number of comparisons? Explain your answer.

Solution

(A) For $\log_k n$ to be $\Omega(\log_2 n)$, we need to show for all value of k there exists a constant c such that $\log_2 n \leq c * \log_k n$ asymptotically.
1 mark for writing the condition.
1 mark for showing that such c exists.

(B) n^n is not $\theta(n!)$.
To show that we can use Sterling's approximation and prove by contradiction.
Let us suppose that n^n is $\theta(n!)$. Then there must exist a **constant** c such that $n^n \leq c*(n!)$ asymptotically. Using Sterling's approximation we get(ignoring other constants) $\frac{e^n}{\sqrt{(n)}} \leq c$, but $\frac{e^n}{\sqrt{(n)}}$ tends to infinity as n tends to infinity. Hence it cannot be bounded a **constant**.
No partial marks have been awarded for just stating the answer(Yes/No).
1 marks have been deducted for incorrectly proven proofs.

(C) The solution is to use count sort or radix sort.
1 mark for briefly describing the algorithm.
1 mark for correctly writing time and space complexities .

(D) The answer is $O(D)$,it can be very easily proven that answer will never be greater than $8D$.
Hint : $D + \frac{D}{2} + \frac{D}{2} + \frac{D}{4} \ldots\ldots \leq 2D$
Full marks have been given if given answer was **clearly** $O(D)$.
1 marks have been deducted otherwise(for unsolved expressions).

(E) We can simply use Master's theorem.
Since $n^{\frac{4}{3}} = \Omega(n^{\log_3 4})$, hence its case 3 or Master's theorem.
Therefore $T(n) = O(n^{\frac{4}{3}})$.
Full marks have been given only when $T(n)$ is computed correctly.

(F) Option 2
When the root node is deleted,the height of the tree either reduces or remins the same. The new replaced node has to be compared with the number of elements equal to the height of the tree. When the height reduces, the new height becomes $\lfloor \log_2 (n-1) \rfloor$. In the other case the height remains the same which is $\lfloor \log_2 (n) \rfloor$.

Observe that when one leaf node is removed and the height doesn't change, it means that $\lfloor \log_2 (n-1) \rfloor = \lfloor \log_2 (n) \rfloor$. Hence, the best upperbound is Option 2.

(G) Option 2
Each deletion takes $c_1 + c_2 \lfloor \log_2 (n-1) \rfloor$ time as mentioned in the above solution. So, during sort, we remove each element one by one.
Total time taken would be $\sum_{i=n-1}^{1} (c_1 + c_2 \lfloor \log_2 i \rfloor)$ which becomes equal to Option 2 because of log summation property. This is the tightest bound.
1 mark of correct option.

1 mark for explanation.

(H) Leftmost term index i indicates T[i] .

0 →
1 →
2 →
3 → 12
4 → 58
5 → 59→41→23
6 →
7 →79→34→16→88→25
8 →

(I) When the array is sorted in reverse order. Each element has to be checked with all the previous elements. Hence it is $\sum_{i=1}^{n}(i-1)$, which is $\frac{n(n-1)}{2}$.

1 mark for order

1 mark for explanation.

(J) The number of comparisons done in a given array is equal to number of inversions. So, we have to find the expected number of inversions in a random permutation.

Every permutation π with N inversions can be read backwards to give a permutation with ${}^{N}C_2 - N$ inversions. This gives a 1-1 map between permutations with N inversions and ${}^{N}C_2 - N$ inversions. Then by symmetry it's clear that the expected value is $\frac{{}^{N}C_2}{2}$.

There is another method to do it with indicator function.

Question 2. [Marks 10].

(a) How to locate and delete the maximum element of the set stored in a Red-Black tree (R-B tree)? Describe *without* writing a pseudocode.

(b) What is the time complexity of part (a)?

(c) What is/are the advantage(s) of an R-B tree over a Heap?

(d) What is/are the disadvantage(s) of an R-B tree over a Heap?

Solution

(a)

if $root = nil$ **then**
  return empty-set
else
  if $root.right = nil$ **then**
    $x := root.val$;
    $root := root.left$;
    return x;
  else
    ExtractSubTreeMax($root - right$)
  end
end

Algorithm 1: ExtractMax($root$): extract the maximum element from the tree rooted at the node pointed by $root$

Justification: If the right subtree of the root is empty, then the root must contain the maximum element, otherwise the maximum element of the right subtree is also the maximum element of the main tree.

if $p.right = nil$ **then**
  $p.parent.right := p.left$;
  if $p.left \neq nil$ **then**
    $p.left.parent := p.parent$
  end
  ;
  if $p.color = black$ **then**
    $fixBlack(root, p.left)$
  end
  ;
  return $p.val$
else
  $ExtractSubTreeMax(root, p.right)$
end

Algorithm 2: ExtractSubTreeMax($root, p$): extract the maximum element from the subtree rooted at the node pointed by p which is not the root

Justification: Again, the same justification as above. Only difference is that if the right subtree of the node pointed by p is empty, then deletion of that node can cause black-height deficiency. Hence we need to

fix it. Here $fixBlack(r,s)$ stands for fixing the black-height in the tree rooted at the node pointed by r and the problem is at the node pointed by p.

(b) The node containing the maximum element, y, must have right child nil. Hence its black-depth must be $b-1$ where b is the black depth of the main tree. Thus the distance of that node from the root is between $b-1$ and $2(b-1)$. So its depth is at most $d-1$ and at least $d/2$, where d is the depth of the main tree. The cost of finding the maximum element is $\Theta(depth(y)) = \Theta(d) = \Theta(\log_2 n)$ where n is the total number of elements in the tree.

The cost of $fixBlack$ is $O(d) = O(\log_2 n)$. SO the total cost is $\Theta(\log_2 n)$.

(c) The advantages of RB-tree over a max-heap (respectively min-heap).

i) The worst case cost of finding any element in a heap is $O(n)$ but that in RBT is $O(\log n)$.

ii) It is also possible to find i-th element in an RBT (if extra information is stored in the nodes) in $O(\log n)$ time. In a heap it will take as much time as in any unstructured set. In particular, we can get min and max elements in $O(\log n)$ time.

iii) We can traverse the RBT and output the entire set in sorted order (increasing or decreasing) in $O(n)$ time. But heap will require $O(n.\log n)$ time.

d) i) A heap can be cobnstructed in $O(n)$ time, while RBT takes $O(n \log n)$ time.

ii) It can read the value of the maximum element in $O(1)$ time while RBT takes $O(\log n)$ time.

iii) The maintanence of RBT in insert and delete is more complex than heap and the former take more memory compared to a heap.

Question 3. [Marks 10].

Consider the following algorithm to compute the k smallest numbers from a set of n numbers. For convenience assume that n is a power of k.

1. Partition the n elements into k subsets of n/k numbers each.
2. Recursively use this subroutine to compute k smallest numbers from each subset.
3. Collect all the k^2 numbers, sort them, and return the k smallest numbers from this sorted set.

(a) Will this algorithm give correct solution. Justify.

(b) Write the recurrence relation for the time complexity. Explain.

(c) Determine the time complexity.

Solution

(a) Yes, the algorithm gives correct solution (1 mark)

Justification:

Let $n = k^a, a \in N$

Suppose the base case is when there the array is of size k. The algorithm then returns the whole array.

Assume that the algorithm works for all arrays of size $\leq$ powers of k upto k^{m-1}

We have to show that it will also work for k^m

We must show that the smallest k numbers in the array are chosen in the set of k^2 numbers. Then the second part of the algorithm will sort these k^2 elements and give us the smallest k.

Let the $S1, S2, \ldots S_{k^{m-1}}$ be the subsets of size k.

Assume that one of the k smallest numbers is not chosen and is in the subset S_i

$\implies$ There are k numbers smaller than it (which are chosen from this set)

This is not possible since it is one of the k smallest overall.

Thus, all of the k smallest numbers are picked in the k^2 picked numbers.

(2 marks)

(b)

$$T(n) = kT(n/k) + c_1k^2log(k^2) + c_2k$$

The term $kT(n/k)$ corresponds to k recursive calls for each of the subsets of size n/k

The term $c_1k^2log(k^2)$ is the time taken to sort k^2 numbers

The term c_2k is the time taken to choose the smallest k from this sorted array

Marking scheme:

For correct recurrence relation: 1

If recurrence is correct, 2 marks for explanation, 1 for each term explained correctly.

(c)

$$\begin{aligned}
&T(n)\\
&= kT(n/k) + c_1k^2log(k^2) + c_2k\\
&= k(kT(n/k^2) + c_1k^2log(k^2) + c_2k) + c_1k^2log(k^2) + c_2k\\
&= k(kT(n/k^3) + c_1k^2log(k^2) + c_2k) + c_1k^2log(k^2)[1+k] + c_2k[1+k]\\
&\ldots\\
&= kT(k) + c_1k^2log(k^2)[1+k+k^2+\ldots+k^{a-1}] + c_2k[1+k+k^2+\ldots+k^{a-1}]
\end{aligned}$$

We take base case as T(k) = O(k) (1 mark)
(taking base case as $T(k^2)$ is also fine, with corresponding derivation)

$$= k \times O(k) + c_1 k^2 log(k^2) \times \frac{k^a - 1}{k - 1} + c_2 k \times \frac{k^a - 1}{k - 1}$$

(2 marks)

substituting $k^a = n$

$$T(n) = c' nklog(k)$$

(1 mark)

Question 4. [Marks 10].

A student wrote a sorting program to sort an array $A[1:n]$ which has three steps: (1) sort $A[1:q]$, (2) sort $A[p,n]$, (3) again sort $A[1:q]$. Here $1 < p \leq q < n$. Assume that each step is executed using some correct algorithm.

For a specified p, what is the least value of q for which the above algorithm will correctly sort $A[1:n]$? Prove your claim.

Solution

- Step1: Sort array from index 1-q
- Step2: Sort array from index p-n
- Step3: Sort array from index 1-q

Given that after step 3 array is sorted.

Let the set of largest n-q elements be S.
The array is sorted after step3
$\Leftrightarrow$ after step2 elements at indices q+1 to n are sorted and contains S
$\Leftrightarrow$ after step1 elements at indices p to n contains S
$\Leftrightarrow$ after step1 elements at indices 1 to p-1 $\bigcap$ S = Φ
Let S_1= Initial elements at indices 1 to q $\bigcap$ S
After step1 $S_1 \subseteq$ elements at indices p to q
As $|S_1| \leq min(|S|, q)$
But $|S|$ = n-q
$\rightarrow |S_1| \leq min(n-q, q)$
So if q $\leq$ n/2 $\rightarrow$ No of values in range between indices p to q $\leq$ q $\rightarrow$ p=1 so we discard this case as we cant fix p
else q $\geq$ n/2 $\rightarrow$ No of values in range between indices p to q $\leq$ n-q $\rightarrow q = \frac{n+p-1}{2}$

The other solution is to show a case where we need to take q be to atleast $\geq \frac{n+p-1}{2}$ and then prove that it is necessary and sufficient.
Getting to the correct value of q fetch 6 marks and proving correctness is 4 marks