CSCE-629 Analysis of Algorithms

Spring 2019

Instructor: Dr. Jianer Chen
Office: HRBB 315C
Phone: 845-4259
Email: `chen@cse.tamu.edu`
Office Hours: T,Th 11:00 am–12:30 pm

Teaching Assistant: Qin Huang
Office: HRBB 315A
Phone: 402-6216
Email: `huangqin@email.tamu.edu`
Office Hours: MWF 3:00 pm–4:00 pm

Solutions to Assignment #1

(Prepared with TA Qin Huang)

1. Answer the following questions, and give a brief explanation for each of your answers.
a) True or False: Quicksort takes time $O(n \log n)$;
b) True or False: Quicksort takes time $O(n^2)$;
c) True or False: Mergesort takes time $O(n \log n)$;
d) True or False: Mergesort takes time $O(n^2)$;

Solutions.

a) False. As we studied in undergraduate algorithms, Quicksort may take time $\Omega(n^2)$ in the worst case, i.e., the running time for Quicksort can be at least $c \cdot n^2$, where c is a fixed constant, for some input of n elements (for all n's). Thus, there is no constant c' such that the running time of Quicksort is bounded by $c' \cdot n \log n$ for all n. That is, the running time of Quicksort cannot be $O(n \log n)$.

b) True. Again by undergraduate algorithms, the running time of Quicksort is bounded by $c \cdot n^2$ for a constant c. Thus, it runs in time $O(n^2)$.

c) True. By undergraduate algorithms, the running time of Mergesort is bounded by $c \cdot n \log n$ for a constant c. Thus, it runs in time $O(n \log n)$.

d) True. As explained about, the running time of Mergesort is bounded by $c \cdot n \log n$ for a constant c. Thus, it is also bounded by $c \cdot n^2$ because $n \log n \leq n^2$ for all $n \geq 1$. By the definition of O-notation, this means that Mergesort takes time $O(n^2)$.

2. Solve the following recurrence relations:
a) $T(1) = O(1)$, and $T(n) = 2T(n/2) + O(n^2)$;
b) $T(1) = O(1)$, and $T(n) = 2T(n-2) + O(n)$.

Solutions.

a) Write the relation as

$$T(1) \leq c_1, \qquad \text{and} \qquad T(n) \leq 2T(n/2) + c_2 n^2; \tag{1}$$

Replace n by $n/2$ in (1), we get an expression for $T(n/2)$:

$$T(n/2) \leq 2T(n/2^2) + c_2(n/2)^2 = 2T(n/2^2) + (c_2 n^2)/2^2. \tag{2}$$

Replace $T(n/2)$ in (1) by the last term in (2), we get

$$T(n) \leq 2^2 T(n/2^2) + (c_2 n^2)(1 + 1/2). \tag{3}$$

Now if we replace $T(n)$ by $T(n/2^2)$ in (1) then use the resulting expression to replace $T(n/2^2)$ in (3), we will get

$$T(n) \leq 2^3 T(n/2^3) + (c_2 n^2)(1 + 1/2 + 1/2^2). \tag{4}$$

This is (probably) sufficient for us to derive that

$$T(n) \leq 2^k T(n/2^k) + (c_2 n^2)(1 + 1/2 + \cdots + 1/2^{k-1}). \tag{5}$$

(You may want to apply another run of replacement to verify this if you are not convinced.)

Let $k = \log n$ in (5), we get (note that we use the condition $T(1) \leq c_1$)

$$\begin{aligned} T(n) &\leq 2^{\log n} T(n/2^{\log n}) + (c_2 n^2)(1 + 1/2 + \cdots + 1/2^{\log n - 1}) \\ &= nT(1) + (c_2 n^2)(2 - 1/2^{\log n - 1}) \\ &\leq c_1 n + 2c_2 n^2 \\ &= O(n^2). \end{aligned}$$

Thus, $T(n) = O(n^2)$.

b) Using the replacement techniques similar to those used in a), we can derive a general formula:

$$T(n) \leq 2^k T(n - 2k) + c_2 n(1 + 2 + \cdots + 2^{k-1}) - 2c_2(2^1 \cdot 1 + 2^2 \cdot 2 + \cdots + 2^{k-1}(k-1)). \tag{6}$$

We first compute $S = 2^1 \cdot 1 + 2^2 \cdot 2 + \cdots + 2^{k-1}(k-1)$. We have

$$\begin{aligned} 2S &= 2^2 \cdot 1 + 2^3 \cdot 2 + \cdots + 2^{k-1}(k-2) + 2^k(k-1) \\ &= (2^1 \cdot 1 + 2^2 \cdot 2 + 2^3 \cdot 3 + \cdots + 2^{k-1}(k-1)) - (2^1 + 2^2 + \cdots + 2^{k-1}) + 2^k(k-1) \\ &= S - (2^k - 2) + 2^k(k-1) \\ &= S + 2^k k - 2^{k+1} + 2. \end{aligned}$$

This gives $S = 2^k k - 2^{k+1} + 2$. Therefore, from (6),

$$T(n) \leq 2^k T(n - 2k) + c_2 n(1 + 2 + \cdots + 2^{k-1}) - 2c_2(2^k k - 2^{k+1} + 2). \tag{7}$$

Let $k = (n-1)/2$ in (7), we get (here we use the condition $T(1) \leq c_1$):

$$\begin{aligned} T(n) &\leq 2^{(n-1)/2} T(1) + c_2 n(1 + 2 + \cdots + 2^{(n-3)/2}) - 2c_2(2^{(n-1)/2}(n-1)/2 - 2^{(n+1)/2} + 2) \\ &\leq c_1 2^{(n-1)/2} + c_2[2^{(n-1)/2} n - n - 2^{(n-1)/2}(n-1) + 4 \cdot 2^{(n-1)/2} - 4] \\ &\leq c_1 2^{(n-1)/2} + c_2[5 \cdot 2^{(n-1)/2} - n - 4] \\ &= O(2^{(n-1)/2}) = O(2^{n/2}). \end{aligned}$$

Thus, $T(n) = O(2^{n/2})$.

3. Consider the following operation on a set S:

Neighbors(S, x): find the two elements y_1 and y_2 in the set S, where y_1 is the largest element in S that is strictly smaller than x, while y_2 is the smallest element in S that is strictly larger than x.

Develop an $O(\log n)$-time algorithm for this operation, assuming that the set S is stored in a 2-3 tree. *Hint*: the element x can be either in or not in the set S.

Solutions. We used the following facts mentioned in the notes:

(1) $l(v)$: the largest element stored in the subtree rooted at $child1(v)$.
(2) $m(v)$: the largest element stored in the subtree rooted at $child2(v)$
(3) $h(v)$: the largest element stored in the subtree rooted at $child3(v)$ (if $child3(v)$ exists).

The above facts imply $l(v), m(v)$, and $h(v)$ appear in the leaves of the 2-3 tree.

We use the following two algorithms to solve the problem. Algorithm 1 is to find the element y_1, and Algorithm 2 is to find the element y_2.

Algorithm 1 Algorithm SearchS(r, x)

Input: A 2-3 tree with root r and x

Output: y_1, the largest element that is strictly smaller than x, or "not exist"

```
if r is empty then
    return "not exist";
end if
if r is a leaf node then
    if value(r) ≥ x then
        return "not exist";
    else
        return value(r);
    end if
end if
if l(r) ≥ x then
    return SearchS(child1(r), x);
else if m(r) ≥ x or r doesn't have the third child then
    let y = SearchS(child2(r), x);
    if y == "not exist" then
        return l(r);
    else
        return y;
    end if
else
    // r has a third child
    let y = SearchS(child3(r), x);
    if y == "not exist" then
        return m(r);
    else
        return y;
    end if
end if
```

Algorithm 2 Algorithm SearchL(r, x)

Input: A 2-3 tree with root r and x

Output: y_2, the smallest element that is strictly larger than x, or "not exist"

```
if r is empty then
    return "not exist";
end if
if r is a leaf node then
    if value(r) ≤ x then
        return "not exist";
    else
        return value(r);
    end if
end if
if l(r) > x then
    return SearchL(child1(r), x);
else if m(r) > x then
    return SearchL(child2(r), x);
else if r has a third child and h(r) > x then
    return SearchL(child3(r), x);
else
    return "not exist";
end if
```

Thus, the combination of the algorithms gives a solution to the problem. Since each algorithm basically traverses a path from the root to a leaf in the 2-3 tree to a leaf, and spends constant time at each node in the tree, its running time is bounded by $O(h)$, where h is the height of the 2-3 tree. Since the 2-3 tree is for the set S of n elements, the height of the 2-3 tree is bounded by $\log n$. In conclusion, each of the algorithms runs in time $O(\log n)$, and the combination of the algorithms that solves the given problem thus also runs in time $O(\log n)$.

4. Consider the following problem: given a 2-3 tree T of n leaves, and an integer k such that $\log n \leq k \leq n$, find the k smallest elements in the tree T. Develop an $O(k)$-time algorithm for the problem. Give a detailed analysis to explain why your algorithm runs in time $O(k)$.

Solution. Algorithm 3 is used to find the k smallest elements. In this algorithm, k is a global variable. If the 2-3 tree is empty or contains a single leaf, then the algorithm returns in step 2 or step 5, respectively, which is obviously correct. Inductively, assume that the algorithm Topk(r_h) correctly outputs the assumed number of elements and decreases the global variable k on 2-3 trees of $h < n$ leaves. Then on a 2-3 tree that has n leaves and is rooted at r_n, steps 7-8 of the algorithm Topk(r_n) will correctly work on the first child $child1(r_n)$ of the root r_n (note that $child1(r_n)$ has fewer leaves than r_n). Thus, if $child1(r_n)$ has at least k leaves, then the recursive call Topk$(child1(r_n))$ in step 8 will output the k smallest elements in $child1(r_n)$ and set the global variable $k = 0$, so steps 10-15 of the algorithm will not be executed and the algorithm Topk(r_n) returns correctly. On the other hand, if $child1(r_n)$ has fewer than k leaves, then the recursive call Topk$(child1(r_n))$ in step 8 will output all elements in $child1(r_n)$ and decrease the global variable k. Since $child1(r_n)$ has fewer than k leaves, k remains larger than 0, so step 11 of the algorithm will continue finding the rest of the elements in $child2(r_n)$, and so on. This shows that correctness of the algorithm.

Algorithm 3 Algorithm Topk(r)

Input: A 2-3 tree with root r and k
Output: the k smallest elements

1: **if** r is empty and $k > 0$ **then**
2: return "no enough elements";
3: **end if**
4: **if** r is a leaf node and $k > 0$ **then**
5: let $k = k - 1$; output value(r); return;
6: **end if**
7: **if** $k > 0$ **then**
8: Topk($child1(r)$);
9: **end if**
10: **if** $k > 0$ **then**
11: Topk($child2(r)$);
12: **end if**
13: **if** $k > 0$ and r has a third child **then**
14: Topk($child3(r)$);
15: **end if**
16: return;

To see the time complexity of the algorithm, let us say that a node v in the 2-3 tree is *visited* if a recursive call Topk(v) is made on the node v during the execution of the algorithm. Let r be a node in the 2-3 tree of height h_r, and assume that Topk(r) is called on r with the global variable k having value k_0. We claim that the total number of visited nodes in the subtree rooted at r is bounded by $h_r + 2k_0$. This is obviously correct when r is a leaf. Now consider the case where r is not a leaf.

If the first child $child1(r)$ of r has at least k_0 leaves, then by induction, the total number of visited nodes in the subtree rooted at $child1(r)$ is bounded by $(h_r - 1) + 2k_0$ since the the subtree rooted at $child1(r)$ has height $h_r - 1$. Since in this case, k will become 0 at step 9, no recursive calls will be made on the other children of r. Thus, the total number of visited nodes in the tree rooted at r is $((h_r - 1) + 2k_0) + 1 = h_r + 2k_0$ (including the node r and all visited nodes in the subtree rooted at $child1(r)$).

If the first child $child1(r)$ of r has k_1 nodes such that $k_1 < k_0$ leaves, then all nodes in the subtree rooted at $child1(r)$ are visited, and the recursive call on $child2(r)$ in step 11 will be made (with the global variable $k = k_0 - k_1$). Note that the subtree rooted at $child1(r)$ has less than $2k_1$ nodes.

Suppose that $child2(r)$ has k_2 leaves, and $k_2 \geq k_0 - k_1$, then by the induction, at most $(h_r - 1) + 2(k_0 - k_1)$ nodes in the subtree rooted at $child2(r)$ are visited, and no recursive call will be made on the third child $child3(r)$ of r. Therefore, in this case, the total number of visited nodes in the tree rooted at r is bounded by

$$2k_1 + [(h_r - 1) + 2(k_0 - k_1)] + 1 = h_r + 2k_0,$$

where the subtree rooted at $child1(r)$ has no more than $2k_1$ visited nodes, the subtree rooted at

$child2(r)$ has no more than $(h_r - 1) + 2(k_0 - k_1)$ visited nodes, and the root r is also a visited node. Again the inductive proof goes through.

Finally, if $k_2 < k_0 - k_1$, then the number of visited nodes in the subtree rooted at $child1(r)$ is bounded by $2k_1$, the number of visited nodes in the subtree rooted at $child2(r)$ is bounded by $2k_2$, the global variable k will have value $k_0 - (k_1 + k_2)$ at step 12 of the algorithm, and a recursive call will be made on the third child $child3(r)$ of r, which will make at most $(h_r - 1) + 2(k_0 - (k_1 + k_2))$ visited nodes in the subtree rooted at $child3(r)$. Adding all the visited nodes in this case, we again derive that the number of visited nodes in the tree rooted at r is bounded by $h_r + 2k_0$. This completes the proof for our claim.

In particular, the number of visited nodes in the given input tree to the algorithm is bounded by $\log n + 2k$. Since we spend only constant time on each visited node in the tree and since $k \geq \log n$, we conclude that the algorithm runs in time $O(\log n + k) = O(k)$.