# CSCE 629-601 Analysis of Algorithms

## Fall 2022

**Instructor:** Dr. Jianer Chen
**Office:** PETR 428
**Phone:** (979) 845-4259
**Email:** `chen@cse.tamu.edu`
**Office Hours:** MWF 3:50 pm - 5:00 pm

**Teaching Assistant:** Vaibhav Bajaj
**Office:** EABC 107B
**Phone:** (979) 739-2707
**Email:** `vaibhavbajaj@tamu.edu`
**Office Hours:** T,TR 2:00 pm - 3:00 pm

## Solutions to Assignment #1

**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.** Prove: any comparison-based searching algorithm on a set of $n$ elements takes time $\Omega(\log n)$ in the worst case. (Hint: you may want to read Section 8.1 of the textbook for related terminologies and techniques.)

*Proof.* A searching algorithm solves the following search problem

$$\text{Search}(S, x): \ \text{Determine if the element } x \text{ is contained in the set } S.$$

As descried in the textbook (pages 191-193), here we assume that there is an order defined on the elements. Thus, any two elements $x$ and $y$ can be compared using one of the relations $x < y$, $y < x$, $x \leq y$, $y \leq x$, and $x = y$. A searching algorithm based on comparisons uses only comparisons to gain order information of the input. Therefore, no matter how the set $S$ is organized, a searching algorithm $A$ based on comparisons can always be depicted as a *decision-tree* $\mathcal{T}$. The decision-tree $\mathcal{T}$ is a binary tree in which each internal node corresponds to a comparison and its two children correspond to the two outcomes of the comparison, and each leaf corresponds to a conclusion of the search result (i.e., "Yes, $x$ is in $S$", or "No, $x$ is not in $S$"). In this model, for each given input $(S, x)$, a comparison (i.e., an internal node in $\mathcal{T}$) will have a unique outcome, so the algorithm on this input will follow a particular path from the root to a leaf in $\mathcal{T}$, in which the leaf gives the conclusion of the algorithm on the input.

Now fix a set $S$ of $n$ elements, $S = \{1, 2, \ldots, n\}$, and consider any searching algorithm $A$ on the input $(S, x)$, where $x$ may vary. Thus, if $x$ is an integer $i$ between 1 and $n$, then the algorithm $A$ will return Yes. Otherwise, the algorithm returns No.

We first show that the decision-tree $\mathcal{T}$ corresponding to the algorithm $A$ contains at least $n$ Yes-leaves. For this, it suffices to show that for two integers $i$ and $j$ between 1 and $n$, $i < j$, the two root-leaf paths in $\mathcal{T}$ corresponding to the inputs $(S, i)$ and $(S, j)$, respectively, are different. Assume the contrary that the inputs $(S, i)$ and $(S, j)$ correspond to the same root-leaf path $\mathcal{P}$ in $\mathcal{T}$. Then consider the input $(S, i + 0.5)$. We have the following facts:

(1) the inputs $(S, i)$ and $(S, j)$ follow the same root-leaf path $\mathcal{P}$ in $T$;

(2) the outcome of each branch in $T$ is determined by an element comparison, and on inputs $(S, i)$ and $(S, j)$, each comparison has the same outcome; and

(3) $i < i + 0.5 < j$.

These facts derive that the input $(S, i + 0.5)$ must also follow the same root-leaf path $\mathcal{P}$ and ends at the Yes-leaf in $\mathcal{P}$. But this is a contradiction because the algorithm $A$ should have concluded No on the input $(S, i + 0.5)$.

This shows that the decision-tree $\mathcal{T}$ has at least $n$ leaves (in fact, it has at least $n$ Yes-leaves, so it should have at least $n + 1$ leaves since it must have at least one No-leaf). It is well-known that a binary tree with at least $n$ leaves has a root-leaf path of length $h \geq \lfloor \log n \rfloor$ (**CSCE-629 Students**: you should verify this). This means that some input $(S, x)$ will follow this path and go through $h - 1$ internal nodes in $\mathcal{T}$. These $h - 1$ internal nodes in $\mathcal{T}$ correspond to $h - 1$ comparisons in the algorithm $A$. Thus, on the input $(S, x)$, the algorithm $A$ makes $h - 1$ comparisons so it runs at least $h - 1$ steps. Because our analysis is based on the worst-case performance, and because $h \geq \lfloor \log n \rfloor$, we conclude that the searching algorithm $A$ takes time $\Omega(h) = \Omega(\log n)$ (in the worst case). $\square$

**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).

We use the algorithm in Figure 1 to solve the problem. Two simple functions **Min** and **Max** are given, which return the smallest and the largest element in a 2-3 tree, respectively. In the main algorithm **Neigobors**, we have two pointers $L$ and $R$, which record the "left sibling" and "right sibling" of the current node $r$ in our search, respectively, so that if $x$ is the smallest in the subtree rooted at $r$, then $\mathbf{Max}(L)$ will be the element $y_1$, while if $x$ is the largest in the subtree rooted at $r$, then $\mathbf{Min}(R)$ will be the element $y_2$.

**function. Min**$(r)$ \\ return the smallest in $r$
1. **if** $(r = \text{Nil})$ return(Nil);
2. **while** ($r$ is not a leaf) $r = child1(r)$;
3. return(value($r$)).

**function. Max**$(r)$ \\ return the largest in $r$
1. **if** $(r = \text{Nil})$ return(Nil);
2. **while** ($r$ is not a leaf)
   **if** $(child3(r) = \text{Nil})$ $r = child2(r)$; **else** $r = child3(r)$
3. return(value($r$)).

**Algorithm. Neighbors**$(r, x)$
Input: the root $r$ of a 2-3 tree $T$ and element $x$
Output: the largest element $y_1$ in $T$ that is smaller than $x$, and
the smallest element $y_2$ in $T$ that is larger than $x$.

1. $L = \text{Nil}$; $R = \text{Nil}$;
2. **while** ($r$ is not a leaf)
   **if** $(l(r) \geq x)$ $r = child1(r)$; $R = child2(r)$;
   **else if** $(m(r) \geq x)$ or $(child3(r) == \text{Nil})$
      $r = child2(r)$; $L = child1(r)$;
      **if** $(child3(r) \neq \text{Nil})$ $R = child3(r)$;
   **else** $r = child3(r)$; $L = child2(r)$;

\\ $r$ is a leaf now

3. **if** (value$(r) == x$) $y_1 = \mathbf{Min}(L)$; $y_2 = \mathbf{Max}(R)$;
   **else if** (value$(r) < x$) $y_1 = r$; $y_2 = \mathbf{Max}(R)$;
   **else** $y_1 = \mathbf{Min}(R)$; $y_2 = r$;
4. return$(y_1, y_2)$.

Figure 1: Finding neighbors

Since the algorithm basically traverses a path from the root to a leaf in the 2-3 tree, 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, the algorithm runs in time $O(\log n)$ and solves the given problem.

**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 1 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 1** 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 $O(\log n + k) = O(k)$.