1. (6 pt.) Recall from lecture that **RadixSort** runs in \( O(d(n + r)) \) time, where we run the stable **CountingSort** algorithm (which takes \( O(n + r) \) time) a total of \( d \) times.

   (a) (3 pt.) Suppose we want to use **RadixSort** to sort a list of lowercase words \( W \) in alphabetical order. All words can be padded with ‘%’ characters (assume ‘%’ comes before ‘a’ alphabetically) to make them the same length. For the following \( W \), describe what the three variables \( d \), \( n \), and \( r \) refer to (your explanation should include some reference to this \( W \) or its elements) and what their particular values would be for this problem.

   \[ W = [\text{the, quick, brown, fox, jumps, over, the, lazy, dog}] \]

   (This isn’t the correct answer, but to give you an example of what we want, you might say “\( n \) is the length of the first word in the list, and \( n \) would have a value of 3 for this list \( W \).”)

   **[We are expecting: Descriptions and concrete values for each of \( d \), \( n \), and \( r \).]**

   (b) (3 pt.) You happen to have the date (day, month, year) that each word was first mentioned during a CS 161 lecture. You want to now sort all the words in \( W \) first by date, then use alphabetical ordering to break ties. You still want to use **RadixSort**, but you need to augment each of the original words in \( W \) with the date information (represented as numbers) by prepending, appending, or somehow inserting these date numbers somewhere in the string. Months are represented using 2 digits (01, 02, …, 11, 12), days are also 2 digits (01, 02, …, 30, 31), and years are 4 digits (e.g. 2014). Assume the digits 0-9 are alphabetically smaller than the ‘%’ character and a-z. Describe your augmentation procedure and write the string you would use to represent the word “fox”, which was first mentioned on October 24, 1989.

   **[We are expecting: A description of your augmentation procedure, and a string.]**

   **[HINT: Make sure you remember to appropriately incorporate the ‘%’ character (note that the maximum length of any word here is 5)!]**
2. (3 pt.) Suppose you have a valid Binary Search Tree $T$ containing $n$ distinct elements. Design an $O(n)$ time algorithm that takes as input the root of the tree $T$ and prints out the keys in the tree in sorted order. If your algorithm uses recursion, that is completely fine.

You may assume that for any node $x$, its key can be found via $x$.key, and you can access its left and right children nodes via $x$.left and $x$.right, respectively. If a node $x$ doesn’t have a left child, then $x$.left will evaluate to NIL. Similarly, if a node $x$ doesn’t have a right child, then $x$.right will evaluate to NIL.

[We are expecting: Pseudocode, and a brief justification of runtime.]

3. (3 pt.) For each of the following examples, if the nodes can be colored red or black to make a legitimate red-black tree, then give such a coloring. If not, then say that they cannot.

(For reference, instructions about how to color the nodes red or black are included at the end of this handout. You are also welcome to take a screenshot and then color the trees in MSPaint, or make a hand-drawn copy and take a photo, or...)

[We are expecting: For each tree, either an image of a colored-in red-black tree or a statement “No such red-black tree.” No justification is required.]

4. (5 pt.) Suppose that $h : \mathcal{U} \rightarrow \{0, \ldots, n-1\}$ is a uniformly random function. That is, for each $x \in \mathcal{U}$, $h(x)$ is distributed uniformly at random in the set $\{0, \ldots, n-1\}$, and the values $\{h(x) : x \in \mathcal{U}\}$ are independent. Prove that for any $x \neq y \in \mathcal{U}$,

$$\mathbb{P}_h\{h(x) = h(y)\} = \frac{1}{n}.$$  

Above, notice that $x$ and $y$ are fixed and the probability is over the choice of $h$.

[We are expecting: A short but rigorous proof.]

[HINT: You can interpret the statement “for each $x \in \mathcal{U}$, $h(x)$ is distributed uniformly at random in the set $\{0, \ldots, n-1\}$” as $\mathbb{P}_h\{h(x) = i\} = \frac{1}{n}$ for all $i \in \{0, \ldots, n-1\}$.]

More problems on next page...
5. (6 pt.) [Dream Interview!] You’re interviewing for your dream job at the most ecological and ethical algorithms company with very healthy free snacks. After 35 stages of interviews, your final interviewer asks you to design a new kind of binary search tree, called amazingTree, even better than red-black trees!

More precisely, your interviewer wants you to design a data structure that stores data in a binary search tree. You may also store other auxiliary information and be as creative as you’d like with your implementation, as long as your amazingTree supports the following operations:

- **amazingInsert(k)** inserts an item with key \(k\) into the amazingTree, maintaining the BST property. It does not return anything. It needs to run in time \(O(1)\).
- **amazingSearch(k)** finds and returns a pointer to node with key \(k\), if it exists in the tree. It needs to run in time \(O(\log n)\).
- **amazingDelete(k)** removes and returns a pointer to an item with key \(k\), if it exists in the tree, maintaining the BST property. It needs to run in time \(O(\log n)\).
- **getAmazingBST()** returns a pointer to the root of the BST that the amazingTree maintains. It needs to run in time \(O(1)\).

Above, \(n\) is the number of items stored in the amazingTree. Your interviewer says that all these operations need to be deterministic, and that amazingTree should handle arbitrary comparable objects (that is, it only interacts with the objects by comparing them).

As much as you want the job, you realize that you can’t give your final interviewer what they want. Prove that no such amazingTree exists that can support those operations with the specified runtimes.

[We are expecting: A proof that no such amazingTree could possibly exist. You may invoke results or algorithms that we have seen in class.]

[HINT: Consider a proof by contradiction! Also, solving Exercise 2 first may help you get used to BSTs.]
6. (16 pt.) [20-questions.]

Suppose you want to sort an array $A$ of $n$ numbers (not necessarily distinct), and you are guaranteed that all the numbers in the array are in the set \{1, 2, \ldots, k\} (assume $k \leq n$). A **20-question sorting algorithm** is any deterministic algorithm that asks a series of YES/NO questions (not necessarily 20 of them, that’s just a name) about $A$, and then writes down the elements of $A$ in sorted order. Specifically, the algorithm cannot directly access the array or its elements, and the algorithm does not need to rearrange the elements of $A$ — it just needs to write down the sorted numbers in a separate location.

Don’t worry about where you’d get the answers to these YES/NO questions (if it helps, assume your algorithm asks some All-Knowing Genie who always answers YES/NO questions correctly). Note that there are many YES/NO questions beyond just comparison-questions. For example, the following are also valid YES/NO questions: “If I remove whatever’s at $A[3]$ and swap $A[8]$ with $A[1]$, would the array be sorted?” and “Is 5 a prime number?”

(a) (8 pt.) Describe a 20-question sorting algorithm that, for every input, asks only $O(k \log n)$ questions. Assume that the algorithm accepts the values of $n$ and $k$ as input.

**We are expecting:**
- Pseudocode **AND** a clear English explanation of what it is doing.
- An explanation of why the algorithm asks $O(k \log n)$ questions.

You do not need to prove that your algorithm is correct.

(b) (4 pt.) Suppose your algorithm does not receive the values of $n$ or $k$ as input. Describe a 20-question sorting algorithm that determines both these values and writes down the sorted array while still asking only $O(k \log n)$ questions.

**We are expecting:**
- A clear English explanation of the algorithm (you should clearly describe how you would recover the values of $n$ and $k$, and then you may cite your algorithm from (a) without re-describing it).
- An explanation of why the algorithm asks $O(k \log n)$ questions.

You do not need to write pseudocode or prove that your algorithm is correct.

(c) (4 pt.) Prove that any procedure to solve this problem must ask $\Omega(k \log \frac{n}{k})$ questions. Again, assume $k \leq n$, and you may also assume that $k > 1$.

**We are expecting:** A convincing and mathematically rigorous argument (which does NOT necessarily mean something long and tedious). A similar level of rigor that we saw for sorting lower bounds in class would be appropriate.

[**HINT:** Use a similar decision tree argument from Lecture 6! Make sure you justify why we can reason about decision trees here in the first place. Then, let $N$ be the number of distinct sorted arrays of length $n$ containing the elements \{1, \ldots, k\}. Start off by showing that $N \geq \left(\frac{n}{k}\right)^{k-1}$, and you’ll be on your way.]
Feedback

There’s no “correct” answer here, and it is completely anonymous.

7. (1 pt.) Please fill out the following poll, which asks about how you’re finding these problem sets:

   https://forms.gle/tgFTw5UPu7Vny1Kk8

Did you fill out the poll?

[We are expecting: The answer “yes.”]
Helpful \LaTeX{}code!

Here is some code to generate the red-black trees; we’ve shown how to color one of the nodes green, you can use this example to color them red or black. You are also welcome to take a screenshot and color in your favorite paint program, or include a picture of a hand-drawn image, or any other way you can think of getting a picture of a colored red-black tree into your PSET.

If you use this code, make sure you include \usepackage{tikz} before the \begin{document} command.

\begin{center}
\begin{tikzpicture}[xscale=0.7]
  \begin{scope}
    \node[draw,circle,fill=green](b) at (0,0) {};
    \node[draw,circle](b0) at (-2,-1) {};
    \node[draw,circle](b1) at (2,-1) {};
    \node[draw,circle](b00) at (-3,-2) {};
    \node[draw,circle](b01) at (-1,-2) {};
    \node[draw,circle](b10) at (1,-2) {};
    \node[draw,circle](b000) at (-4,-3) {};
    \node[draw,circle](b001) at (-2.5,-3) {};
    \node[draw,circle](b010) at (-1.5,-3) {};
    \node[draw,circle](b0000) at (-5,-4) {};
    \node[draw,circle](b0001) at (-3,-4) {};
    \draw (b) -- (b0);
    \draw (b) -- (b1);
    \draw (b0) -- (b00);
    \draw (b0) -- (b01);
    \draw (b1) -- (b10);
    \draw (b00) -- (b000);
    \draw (b00) -- (b001);
    \draw (b01) -- (b010);
    \draw (b000) -- (b0000);
    \draw (b000) -- (b0001);
    \draw (b010) -- (b011);
    \draw (b0000) -- (b00000);
    \draw (b0000) -- (b00001);
  \end{scope}
  \begin{scope}[xshift=10cm]
    \node[draw,circle,fill=green](b) at (0,0) {};
    \node[draw,circle](b0) at (-2,-1) {};
    \node[draw,circle](b1) at (2,-1) {};
    \node[draw,circle](b00) at (-3,-2) {};
    \node[draw,circle](b01) at (-1,-2) {};
    \node[draw,circle](b10) at (1,-2) {};
    \node[draw,circle](b11) at (3,-2) {};
    \node[draw,circle](b000) at (-4,-3) {};
    \node[draw,circle](b001) at (-2.5,-3) {};
    \node[draw,circle](b010) at (-1.5,-3) {};
    \node[draw,circle](b0000) at (-5,-4) {};
    \node[draw,circle](b0001) at (-3,-4) {};
    \draw (b) -- (b0);
    \draw (b) -- (b1);
    \draw (b0) -- (b00);
    \draw (b0) -- (b01);
    \draw (b1) -- (b10);
    \draw (b1) -- (b11);
    \draw (b00) -- (b000);
    \draw (b00) -- (b001);
    \draw (b01) -- (b010);
    \draw (b000) -- (b0000);
    \draw (b000) -- (b0001);
    \draw (b010) -- (b011);
    \draw (b0000) -- (b00000);
    \draw (b0000) -- (b00001);
  \end{scope}
\end{tikzpicture}
\end{center}