This homework must be completed and submitted electronically. Formatting standards, submission procedures, and (optional) document templates for homeworks may be found at

http://classes.engineering.wustl.edu/cse241/ehomework/ehomework-guide.html

Advice on how to compose homeworks electronically, with links to relevant documentation for several different composition tools, may be found at

http://classes.engineering.wustl.edu/cse241/ehomework/composing-tips.html

Please remember to

• typeset (do not hand-write) your homework’s text;
• create a separate PDF file for each problem;
• include a header with your name, WUSTL key, and the homework number at the top of each page of each solution;
• include any figures (typewritten or hand-drawn) inline or as floats;
• upload and submit your PDFs to Blackboard before class time on the due date.

Always show your work.
The approximate weights of each problem (out of 100%) are given in parentheses, but things that come up in the course of grading may cause these weights to be adjusted. Note that you need only do any four problems out of five to get full credit for this homework, though you may do all five for extra credit.

Unless otherwise noted, all occurrences of “log” refer to the base-2 logarithm.

1. (25%)

(a) Illustrate the operation of the Partition algorithm on the following array: \{3, 9, 2, 7, 1, 6, 4, 5\}. Assume that we partition around the last element. Indicate where the swaps occur during the algorithm’s execution and show the state of the array after each swap.

(b) If Partition partitions around the largest element of an array, is the swap operation ever triggered? Do any of the elements move? Why or why not?

(c) Suppose we modify the Quicksort procedure to consistently pick the jth element of the array as the partition element. The modified code looks like this:

```plaintext
QUICKSORT(A, p, r)
if (p < r) {
    x ← MIN(p + j, r)
    swap(A[x], A[r])
    z ← Partition(A, p, r)
    QUICKSORT(A, p, z − 1)
    QUICKSORT(A, z + 1, r)
}
```

(We need the MIN operation so that we do something legal even if the array has size less than \(j + 1\).)

Argue that it is possible to make this modified Quicksort procedure take quadratic time, no matter which value of \(j\) we pick. In particular, show that, for every fixed \(j\) and every large enough \(n\), there exists an array \(A^n_j\) of size \(n\) that causes worst-case partitioning behavior at every level of Quicksort.

(Hint: \(A^n_j\) is a simple modification of the sorted array we considered in class.)

2. (25%)

(a) In class, we showed that any sort must take time \(Ω(n \log n)\), provided we are only allowed to compare elements using the \(>\) operator. Suppose we extend our model of computation to allow the \(<\) and \(=\) operators as well. Does the asymptotic lower bound change? Justify your answer.

(b) Now suppose we extend the model of computation with the trinary operator \(\text{Min-Of-Three}(x, y, z)\), which returns the minimum of its three arguments. Assume that this new operator runs in constant time. Does the asymptotic lower bound change? Justify your answer.

(c) Many modern CPUs include an instruction \(\text{PopCount}(x)\), which returns the number of ‘1’ bits in the binary representation of a number \(x\). \(\text{PopCount}\) is sometimes called the “NSA instruction,” because certain shadowy government agencies require it in the processors they purchase, presumably to speed up secret algorithms for breaking encryption.

Suppose we extend the comparison model to allow us to run \(\text{PopCount}\) on numbers of up to \(\log n\) bits in constant time. What is the new asymptotic lower bound for sorting in this model?
(a) Consider the following list of strings of length four:
  acta
gaac
cetag
aagc
tcat
gcgt
tct
Suppose we sort this list into lexicographic (i.e. alphabetical) order with a radix sort that operates on one string position at a time.
Show the state of the list after each of the four sorting passes.

(b) In Lab 2, we looked at a hashing strategy for detecting $k$-mer strings that appear in both a pattern string and a text corpus. This strategy performs well if the hash functions are good and the string is not pathological, but it can do quite badly if either of these assumptions fail.
Let $k$ be a small constant. Given a pattern string of length $n$ and a corpus string of length $m$, show that we can determine in worst-case total time $\Theta(m + n \log m)$ whether each $k$-mer in the pattern appears in the corpus. (*Hint:* start by making a list of all the $k$-mers in one string.)

(c) Show that the problem in (b) can in fact be solved more efficiently, in worst-case time $\Theta(m + n)$.

4. (25%) The following figure gives a binary search tree $T$ on the collection $\{3, 5, 8, 10, 14, 16, 19\}$.

(a) Give the sequence of nodes traversed by the operation $\text{succ}(8)$. How about $\text{succ}(10)$? $\text{succ}(19)$?
For each of these cases, assume that traversal *starts* from the node with the indicated key.

(b) Show the state of the tree after each of the following series of operations: $\text{delete}(10)$; $\text{delete}(5)$; $\text{delete}(14)$. Each operation should be done on the tree resulting from the previous one.

(c) Give a binary search tree of the least depth you can that contains the same nodes as the given tree but puts key 5 at the root. Do the same for key 16.

5. (25%) An *interval tree* is a special type of binary tree that stores an ordered collection of (closed) intervals on the real line. An interval $I$, such as $[3, 7]$, is defined by its left and right endpoints $I.\ell$ and $I.r$. We can sort intervals according to the following relation: $I_1 < I_2$ iff either $I_1.\ell < I_2.\ell$ or $I_1.\ell = I_2.\ell$ and $I_1.r < I_2.r$.

(a) Write pseudocode that takes an interval tree $T$ and a new node $x$ whose key is interval $I$, and inserts $x$ into $T$. Your code should take time proportional to the tree’s height. (*Hint:* start by
writing a helper function \textnormal{LEQ}(I_1, I_2) that implements the \leq relation on pairs of intervals. Now implement insert in terms of the helper.)

(b) A common query on interval trees is the \textit{containment test}: given a point \( p \), does some interval in the tree contain \( p \)? (We need not actually return this interval, merely determine whether it exists.)

Suppose the root of an interval tree contains the interval \( I \). We determine that \( p \) is not contained in \( I \). If \( p < I.\ell \), which subtrees of the root might hold an interval containing \( p \)? What about if \( p > I.r \)? Justify your answer.

\textit{(Hint: your answer should imply that performing an containment test on a tree of size \( n \) requires \( \Theta(n) \) time in the worst case, even if the tree is balanced.)}

(c) Suppose we augment our interval tree as follows. Each node \( x \) of the tree now contains not only an interval \( I \) but also a value \( maxR \), which is the largest right endpoint of any interval in the left subtree of \( x \) (or \(-\infty \) if the left subtree is null).

Argue that with this extra information, we can uniquely determine which subtree (if any) to search if the root’s interval \( I \) does not contain \( p \). It will help if you separately consider each of four cases:

i. \( p < I.\ell \) and \( p > maxR \);
ii. \( p < I.\ell \) and \( p \leq maxR \);
iii. \( p > I.r \) and \( p \leq maxR \);
iv. \( p > I.r \) and \( p > maxR \).

Hence, having \( maxR \) lets us perform containment tests in time proportional to the tree’s height, not its size!

(d) Finally, modify your pseudocode from Part (a) to update any \( maxR \) fields that change when a new interval is inserted into the tree. The revised insert procedure should still take time proportional to the tree height.