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%) What asymptotic solution, if any, does the master method give for each of the following recurrences?

(a) \( T(n) = 16T\left(\frac{n}{2}\right) + 3n^4\log^2 n \)
(b) \( T(n) = T\left(\frac{n}{3}\right) + 4\log n \)
(c) \( T(n) = 4T\left(\frac{n}{4}\right) + \frac{n}{\log\log n} \)
(d) \( T(n) = 9T\left(\frac{n}{4}\right) + 2n^{\log_4 9} \)
(e) \( T(n) = 11T\left(\frac{n}{3}\right) + n^2 \)
(f) \( T(n) = 999T\left(\frac{n}{10}\right) + 2n^3 \)
(g) \( T(n) = 7T\left(\frac{n}{2}\right) + n^3\log n\log\log n \)

2. (25%) Consider the general recurrence

\[
T(n) = \begin{cases} 
  c_0 & \text{if } n = 1 \\
  aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise}
\end{cases}
\]

In class, we showed that this recurrence has solution

\[
T(n) = c_0 n^{\log_b a} + \sum_{k=0}^{\log_b n-1} a^k f\left(\frac{n}{b^k}\right).
\]

Suppose now that we alter the recurrence so that it terminates not for \( n = 1 \) but for \( n = n_0 \), for some constant \( n_0 > 1 \). In other words, we replace the base case by “\( c_0 \) if \( n = n_0 \).” You may assume for simplicity that \( n \) and \( n_0 \) are both powers of \( b \).

(a) Thinking about the recursion tree for the revised recurrence, it isn’t surprising that its solution has the form

\[
T'(n) = c_0 a^n + \sum_{k=0}^{x-1} a^k f\left(\frac{n}{b^k}\right).
\]

What expression should replace the \( x \) above? Justify your answer. You might find it helpful to sketch the recursion tree for the revised recurrence.

(b) Show that \( c_0 a^n \) above is still a constant times \( n^{\log_b a} \).

(c) The summation in (a) above can be rewritten as

\[
\sum_{k=0}^{\log_b n-1} a^k f\left(\frac{n}{b^k}\right) - \sum_{k=\log_b n - y}^{\log_b n-1} a^k f\left(\frac{n}{b^k}\right)
\]

for some constant \( y \) independent of \( n \). What is this \( y \)?

(d) Show that the right-hand summation in (c) can be simplified to \( c'n^{\log_b a} \) where \( c' \) is an expression independent of \( n \).
We can conclude from (b) and (d) that $T'(n) = T(n) - c^* n^\log_b a$ for some constant $c^*$, which, as we mentioned in class, proves that it’s usually OK to leave the base case out when describing a recurrence if you only care about its asymptotic solution.

3. (25%) Suppose you have an empty hash table $T$ with $m = 8$ slots that stores integer records (i.e., the record and its key are one and the same). The table resolves collisions by open addressing with double hashing, and its hash functions are $h_1(k) = (5 \cdot k) \mod 8$ and $h_2(k) = (3 \cdot k) \mod 7 + 1$. (Disclaimer: these are not good hash functions, just examples to help you reinforce your understanding of open-addressed hashing.)

(a) Sketch diagrams showing the state of the hash table and give its load factor after each of the following series of calls:

- T.insert(293)
- T.insert(30181)
- T.insert(38388)
- T.insert(62)
- T.insert(3421)
- T.delete(293)

(b) After the hash table $T$ has performed the above operations, what is the sequence of slots in the table inspected by each of the following find operations before it returns?

- T.find(293)
- T.find(38924)
- T.find(62)

(c) Just to show that the above table and hash function parameters aren’t great, give an example of a key for which the insert operation could fail even if the table is not full. Explain why insertion could fail for this key.

4. (25%) This problem is an example of universal hashing, a strategy for picking hash functions for a hash table randomly so that no input always exhibits bad hashing behavior.

Let $p$ be a prime number. I want to hash pairs of numbers $(x, y)$, where $x$ and $y$ are always between 0 and $p - 1$ inclusive. I decide to use a chained hash table with hash function

$$h_{a,b}(x, y) = (ax + by) \mod p$$

where $a$ and $b$ also lie between 0 and $p - 1$.

(a) Suppose that my $a$ and $b$ are fixed, and that you’ve discovered what they are (perhaps by hacking into my computer). Describe how to generate $p$ distinct input pairs $(x_i, y_i)$ for which $h_{a,b}(x_i, y_i)$ yields the same value. That is, all the inputs $(x_i, y_i)$ will hash to the same slot of my table.

(Hint: for any $c$, $1 \leq c < p$, and every $i$, $0 \leq i < p$, there exists exactly one $j$, $0 \leq j < p$ for which $cj \equiv i \pmod{p}$.)

(b) To defend against your malicious hackery, I have decided not to fix $a$ and $b$ once and for all, but rather to choose them randomly every time I instantiate my hash table class. Each value will be chosen uniformly at random (with replacement) from the range $0 \ldots p - 1$. 
Fix two non-identical inputs \((x, y)\) and \((x', y')\). (They may have the same \(x\) or \(y\) values, but not both.) For how many distinct pairs \((a, b)\) will these two inputs hash to the same slot? 
(Use the hint from part (a)).

(c) If I choose each of \(a\) and \(b\) uniformly at random from \(0 \ldots p - 1\), what is the probability that \((x, y)\) and \((x', y')\) will hash to the same value? 
\((\text{Hint: this probability is just the number of ways to choose } a \text{ and } b \text{ to make the two inputs hash to the same value, divided by the total number of ways to choose } a \text{ and } b.\)\)

(d) Given an arbitrary set of \(n\) distinct inputs \((x_i, y_i)\), what is the expected number (over my random choices of \(a\) and \(b\)) of pairs \(i, j, \ i < j\), for which \(h_{a,b}(x_i, y_i) = h_{a,b}(x_j, y_j)\)? 
\((\text{Hint: use linearity of expectation!})\)

The moral of this story is that, by choosing my \(a\) and \(b\) uniformly at random for each instantiation of my table, I can avoid exhibiting worst-case behavior all the time for any single input. Moreover, every possible input exhibits a small number of collisions \textit{on average}, so most inputs exhibit good behavior most of the time.

5. (25%) In this problem we will explore an efficient strategy for maintaining extensible array-based data structures. The arrays used by these structures have no \textit{a priori} upper limit on their size; rather, they grow as the number of elements in them increases. Java’s \texttt{ArrayList} type implements an extensible array.

An extensible array \(A\) starts out empty with some fixed number of slots and grows as elements are inserted into it according to the following \textbf{doubling rule}:

\textit{If we are about to insert an element into } \(A\) \textit{and find that it is full, double the number of slots in } \(A\) \textit{before inserting the new element.}

Doubling the array size typically cannot be done in-place; it requires that we allocate a new, larger array, then copy all the elements of the old array into the new array.

Suppose now that we create an empty extensible array with one slot, then insert \(2^k\) elements into it. Justify your answers to each of the following questions.

(a) What is the final number of slots in the array after all \(2^k\) elements have been inserted? (\textit{Hint: work out the answer for a few small values of } \(k\) \textit{to see the pattern, then check your guess with an induction proof.})

(b) How many times did we have to double the array size to insert all \(2^k\) elements?

(c) How many times total did we have to copy an element of the array over all these doubling operations? (\textit{Hint: figure out the number of elements copied during each doubling operation, then sum over all doublings.})

(d) What is the average number of elements copied per insertion over all \(2^k\) insertions? In other words, what is the ratio of the number of copies to the number of new insertions? Assuming each copy takes constant time, does extensibility affect the average asymptotic complexity of the \texttt{insert} operation?

(e) How can you use the idea of extensible arrays to implement an \textit{extensible hash table}? Such a table grows dynamically but always maintains its load factor at or below some bound \(\alpha < 1\). Assume that you can hash a record and place it into the table in constant time.
(Fun fact: this problem is an example of amortized analysis. We periodically do a big chunk of work (the doubling operation), but we can show that the work per insert operation is still small. In other words, the cost of each doubling operation is amortized over a large number of insertions.)