Homework 11: Divide and Conquer
Due: 10:59 PM, Nov 20, 2019

Contents

1 Binary Search (Practice) 
2 Binary Search Tree Analysis (24 Points)
3 Find the GCD (10 Points)
4 Discrete Root-finding (10 Points)

Objectives

By the end of this homework, you will know:

1. binary search (if you do the practice problem)
2. Euclid’s algorithm for computing greatest common divisors
3. an efficient root finding algorithm

How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. For this homework specifically, this entails answering Binary Search Tree Analysis, Find the GCD, and Discrete Root-finding questions.

In order to hand in your solutions to these problems, they must be stored in appropriately-named files. In particular, each should be named for the corresponding problem, as follows:

- CS17SetupDnC.re
- README.txt
- BstAnalysis.pdf
- Gcd.re
- FindRoot.re

For this assignment, you must hand in a ReasonML file, which must end with the extension .re, and a \LaTeX\.pdf.
You should have a README.txt file whose first line contains only your Banner ID, and optionally with a message to the person grading explaining anything peculiar about the handin. For example:

README.txt:
B01234567
There’s nothing to say except that I’m turning in these files plus this README the way the instructions say that I should.

To hand in your solutions to these problems, you must upload them to Gradescope. Do not zip or compress them. If you re-submit your homework, you must re-submit all files.

Set-Up

To start, navigate to the directory containing your homework assignments on your local computer and run the following command, replacing ’homework11’ with whatever you would like to call your homework directory for this homework. You only need to run this command once for each homework/project:

bsb -init homework11 -theme basic-reason

Copy CS17SetupDnC.re (provided in the email) into your homework11/src directory. DO NOT EDIT. The files that you write for this assignment should also reside in the homework11/src directory. For each file that requires functions from another file, open the required files at the top using:

open <FileName>;

To compile and run your code, from the homework11/src directory run

npm run build
node <file-name-to-run>.bs.js

Practice

1 Binary Search (Practice)

Suppose that you and a friend are playing a guessing game. You think of a number between 1 and 10, and your friend tries to guess it. At each attempt, you tell her if she guessed correctly or not. We can simulate this game with a procedure that consumes a number as well as upper and lower bounds, and produces the number of tries it takes to guess that number.

Task: Write a procedure, guess, which consumes a number, a lower bound, and an upper bound, and tries to guess the number by starting at lower bound and counting up to the upper bound. Your procedure should produce the number of guesses it makes.

You and your friend are bored with this game, so you decide to change the rules a bit. Now, when she guesses a number, you will tell her if her guess is too high, too low, or just right.

Task: Write another procedure, guess2, which consumes a number, a lower bound, and an upper bound, and tries to guess the number using the method of divide and conquer. As above, your
procedure should produce the number of guesses it makes.

**Hint:** Create a type `answer` to represent the result of a guess.

Think about how many tries it will take `guess` and `guess2` to guess a given number `n`. Note that the `guess2` procedure we’re looking for should be more efficient (i.e., it should make fewer guesses) than `guess`.

## Problems

### 2 Binary Search Tree Analysis (24 Points)

In class we briefly discussed binary search trees, which are binary trees storing, say, integer values, with the property that for every node, `N`, all of the values of nodes in the left subtree of `N` are less than or equal to the value `v` at `N`, and all the values of nodes in the right subtree of `N` are greater than `v`. Furthermore, the left and right subtrees each contain about half the elements of the tree; for the sake of this problem, you may assume that the number `n` of nodes in the tree is one less than a power of 2, so that there are \( \frac{n-1}{2} \) nodes in the left and right subtree, and in general, if a node has `k` descendants (including itself), then its left subtree contains \( \frac{k-1}{2} \) nodes, as does its right subtree.

Maintaining these two properties — “search tree order” and “balance” — is pretty tough, and many algorithms have been developed to do this (with “balance” often being replaced by “approximate balance”, which turns out to work pretty well). You’ll discuss these further in CS18.

But for this problem, consider the task of checking to see whether some number `k` is the value stored at any node of the tree. The code looks like this:

```ocaml
let rec bst_search: (int, tree) => bool = (v, t) =>
  switch (t) {
    | Leaf => false
    | Node(v1, left, right) when v == v1 => true
    | Node(v1, left, right) when v < v1 => bst_search(v, left)
    | Node(v1, left, right) when v > v1 => bst_search(v, right)
    | Node(_, _, _) => failwith("Node is invalid!")
  };
```

Let \( V(n) \) be the time to search for a value in a balanced search tree of `n` nodes. Suppose that \( V : \mathbb{N} \to \mathbb{R} \) satisfies the recurrence:

\[
V(0) = 2 \\
V(n) = 3 + V(\lfloor \frac{n}{2} \rfloor), \text{ for } n > 0.
\]

**Task:**

(a) Write out \( V(0), V(1), \ldots, V(6) \), i.e., do plug-n-chug.

(b) \( V \) has some rate of growth that’s related to \( \log_2 \) in some way. Make a conjecture of the form, “For all `n` greater than fill in something, \( V(n) \) is no greater than fill in something,” where the first item to fill in should be a constant (an actual number, like 0, or 3) and the second should
be a function of \( n \), which I’ll call \( n \mapsto f(n) \). You might pick \( f(n) = n^2 \), for instance, to show that \( V \) grows at most quadratically. But that would be a bad idea, because of part c, which you should read before trying this part.

Then prove your conjecture. You may, in your proof, replace \( V(\lfloor \frac{n}{2} \rfloor) \) by \( V(n/2) \) wherever you like, i.e., you may assume that all divisions by 2 result in integers because (as with the analysis of mergesort), this turns out to have no important consequences, but makes the algebra a lot prettier.

(c) Make another conjecture, of the form “For all \( n \) greater than \( \text{fill in something} \), \( V(n) \) is at least \( c \ f(n) \),” where \( f \) is the function from above, and \( c > 0 \) is some constant (like 0.3 or 11). Prove this conjecture as well. (Again, you may use the “division always yields integers” assumption to simplify things.)

When you’re done, you’ll have shown that \( V \in \Theta(f) \).

(d) Suppose that \( W \) satisfies a recurrence like

\[
W(0) = 2
\]
\[
W(n) = n + W(\lfloor \frac{n}{2} \rfloor), \text{ for } n > 0.
\]

Prove that \( W \in O(n) \). (Again, use the simplifying assumption.)

(e) Suppose that \( Z \) satisfies a recurrence like

\[
Z(0) = 1
\]
\[
Z(n) = 1 + 2 \cdot Z(\lfloor \frac{n}{2} \rfloor), \text{ for } n > 0.
\]

Make a conjecture about the big-O order of growth of \( Z \)—you need not prove it.

3 Find the GCD (10 Points)

The greatest common divisor (gcd) of two (let’s say, positive) integers is the greatest integer that divides them both evenly. For example, the gcd of 100 and 15 is 5. Note: Both procedures, gcdStruct and gcd, that you will be writing in this section will go in a file named Gcd.re.

Task: Using structural recursion, write a procedure gcdStruct that consumes two natural numbers, say \( n \) and \( m \), and produces their gcd. Recall that structural recursion relies on recurring systematically through some organized structure (like traversing a list element by element, or traveling down a binary search tree node by node).

Hint: Starting with some reasonable, yet high enough, number, say \( k \), your procedure should check whether \( k \) is the gcd of \( n \) and \( m \), and if it is not, it should try \( k - 1 \), and so on. In this case, the structure being traversed is the number \( k \).

Note: An integer \( a \) divides an integer \( b \) if there exists an integer \( x \) such that \( ax = b \). (Note that every integer divides 0.) Hence, the gcd of any integer \( n \) (including 0) and 0 is that integer, because \( n \) divides itself, and every integer divides 0.

We can do better than this slow algorithm for finding the gcd by using non-structural recursion! Indeed Euclid invented such a method for searching for the gcd over 2,000 years ago! His key
observation was: \( \gcd(n, m) = \gcd(n - m, m) \), if \( n > m \). (Likewise, \( \gcd(n, m) = \gcd(n, m - n) \), if \( m > n \).) For example, \( \gcd(8, 5) = \gcd(3, 5) = \gcd(3, 2) = \gcd(1, 2) = \gcd(1, 1) = 1 \).

**Task:** Using Euclid’s observation, write a procedure \( \text{gcd} \) that consumes two natural numbers and produces their gcd using non-structural recursion. Instead of checking every integer between \( k \) and 0, you can use Euclid’s observation to define the \( m \) and \( n \) values in your recursive step, thus making your gcd algorithm more efficient.

**Hint:** On examples like \( \gcd(100, 15) \), repeated subtractions, like we did in our calculations above, is still too slow. Can you come up with a generalization of repeated subtractions which does the necessary computations faster? If so, implement it. If not, you can implement the straight-up subtraction algorithm described above for half credit.

## 4 Discrete Root-finding (10 Points)

For this problem, let’s assume that the outdoor temperature varies continuously, i.e., the temp cannot be 5 degrees at one instant and 2 degrees at the next instant without having passed through every possible temperature between 5 and 2. If you know the temperature was above 0 (Celsius) at 1 PM, but below 0 at 2 PM, you can say that at least once between one and two PM, the temperature was exactly zero.

If I gave you a table of values, like

<table>
<thead>
<tr>
<th>time</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>-4</td>
</tr>
</tbody>
</table>

of time versus temperature, you could plot these points on a graph, and perhaps even connect the dots. You’d see that the connect-the-dots graph crosses the \( x \)-axis between 1 and 2, between 2 and 3, and touches the \( x \)-axis at 5, so that you could reasonably say it was zero somewhere between 4 and 5 (namely, right at 5) and somewhere between 5 and 6 (again, right at 5).

A place where a graph crosses the \( x \)-axis is called a “root”, and finding roots is in general quite difficult, partly because numbers on a computer aren’t like real numbers. Just as a for-instance, if \( a \) and \( b \) are unequal real numbers, then their average, \( (a + b)/2 \), is different from both of them. But if in sketch.sh, you let \( a = 1.0 \) and \( b = 0.9999999999999889 \), and compute \( c = (a + . b) / . 2.0 \), you’ll find that it’s exactly equal to \( a \).

But for integer-valued functions recorded at integer points, like our temperature data, and then defined at other points by 'connect the dots’, root finding is easier: if we have a list of data like

\[
[5, -1, 4, 2, 0, -4]
\]

we can just walk through it, and find a pair of adjacent numbers whose signs differ (“one positive, one negative”) or where one or both are zero (which certainly gives us a root.) We could call the
indexes of such a pair of adjacent numbers a “root-containing interval.” In the case of the list above, we could identify the interval \([0, 1]\) as a root-containing interval, because the 0th and 1st numbers, 5 and \(-1\), have opposite signs. The other root-containing intervals are \([1, 2], [3, 4],\) and \([4, 5]\). (You should check that this makes sense to you.)

If the only root in some set of data is in the very last interval, this walk-through-the-list interval-finding process takes time linear in the length of the list of data, but it’s pretty much the only way to do the job.

Suppose, though, that I said I had a function \(T\) that would consume an integer (the hour of the day, for instance) and produce as output the temperature, and could do so in constant time. And suppose we have temperature observations from hour 0 to hour 20, and that \(T(0) > 0\) and \(T(20) < 0\). We could build a list

\[
[T(0), T(1), \ldots, T(20)]
\]

and apply the previous approach to find a root-containing interval.

The idea from the first paragraph is useful though: as a first step, we could check that \(T(0)\) and \(T(20)\) have opposite signs (or that one of them is zero). If so, then there’s guaranteed to be a root-containing interval somewhere in that list, but we don’t need to compute the whole list to find it!

Let’s start with the case where \(T(0)\) is positive and \(T(20)\) is negative. Suppose we compute \(T(10)\) and find it’s positive. Then there must be a root between 10 and 20, i.e. a number \(k\) between 10 and 20 where \(T(k)\) and \(T(k+1)\) have opposite signs, or where one of them is zero. So now we only have to examine half of the list (assuming our goal is to find some root-containing interval, not all of them). Recurring, we can find the root-containing interval very fast.

This brings us to the actual problem.

**Task:** Given

- a pair of integers \(a\) and \(b\) with \(a < b\), and
- an \(\text{int} \to \text{int}\) function \(T\) whose domain contains all the numbers from \(a\) to \(b\), and
- either \(T(a)\) and \(T(b)\) have opposite signs, or at least one of them is zero,

find a root-containing interval for the connect-the-dots function defined by \(T\) on the interval \([a, b]\).

In particular, write a procedure

\[
\text{let } \text{iroot: (int, int, int }\to\text{ int) }\to\text{ int }= \ldots
\]

that consumes the values \(a\) and \(b\), and a function \(T\), and returns a number \(k\) between \(a\) and \(b - 1\) with the property that \([k, k + 1]\) is a root-containing interval for the function.

For instance,

\[
\text{let } T:\text{int }\to\text{ int }= x \to (2x - 5);
\text{let } a = 0;
\text{let } b = 10;
\text{print_int(iroot(a, b, T));}
\]
should print the value "2", because between $x = 2$ and $x = 3$, the function $T$ changes sign; it goes from $T(2) = -1$ to $T(3) = +1$.

Note: when the interval $[a, b]$ is small — like 0, 1, or 2 items — there may be some tricky edge-cases to handle.

Your procedure should run in time that’s in $O(n \mapsto \log n)$, where $n = b - a + 1$ is the number of integers between $a$ and $b$, inclusive. (You may assume, as mentioned above, that calling the function $T$ on a single argument is a unit-cost operation.)