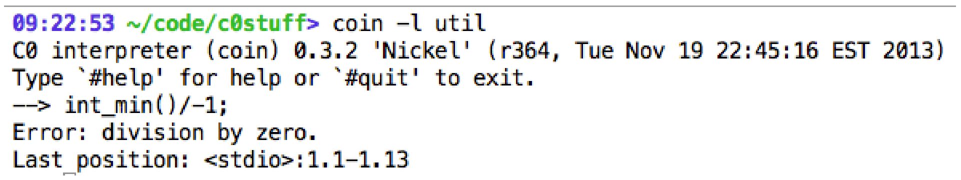
15-122: Principles of Imperative Computation

Recitation 6 Solutions — Josh Zimmerman, Nivedita Chopra

Checkpoint 0

Look at the following snippet of code.



```
09:22:53 ~/code/c0stuff> coin -l util
C0 interpreter (coin) 0.3.2 'Nickel' (r364, Tue Nov 19 22:45:16 EST 2013)
Type `#help' for help or `#quit' to exit.
--> int_min()/-1;
Error: division by zero.
Last position: <stdio>:1.1-1.13
```

Why do we get a “division by zero" error here?

Solution: In C0, if $x = a/b$ and $y = a\%b$, we always maintain the invariant: $a = b * x + y$, with $|y| < b$ and the sign of b being the same as the sign of y.
With these constraints, it is not possible to find a solution to the invariant when $a = int_min$ and $b = -1$.

Debugging tips and tricks

Some very useful tips on debugging, and addresses common pitfalls can be found at`http://c0.typesafety.net/tutorial/Debugging-C0-Programs.html`. We're summarizing some key points here, but you should read the entire page for more details.

- Compilation errors
  - *Lexical errors* occur when you have incorrect numbers or variable names, like `0a4`.
  - *Syntax errors* are generated by sequences of characters that don't make sense, like `f(, 4);` or `x + 3 = 4;`
  - *Type errors* arise when you write expressions that don't make sense based on the types of variables, like `true == 3` or `3 + ‘‘hello’’`.
  - *Unexpected EOF* errors are generally caused by an unmatched brace, parenthesis, or similar character. Most editors can be configured to highlight unmatched parens and braces.
  - *Undeclared variable errors* happen if you use a variable before declaring it.
- Runtime errors
  - *Floating point* or *division by zero* errors generally indicate that you divided by zero, or divided `int_min()` (`0x80000000`) by `-1`. They will also occur if you try to shift left or right by 32 or more or by a number less than 0.
  - *Segmentation faults* occur if you attempt to access memory that you can't access. Right now, the only thing we've covered that can cause this is out-of-bounds array access (accessing a negative index of an array or accessing something past the end of the array), but later we'll see that `NULL` pointer dereferences can also cause this.
  - *Contract errors* occur when a contract is violated and contract checking is turned on.

- Weird behavior with conditionals and loops: If some code that should be running in a conditional or loop isn't, make sure you have braces around the block. It's much harder to debug otherwise.

  ```
  while (some_condition)
      printint(i);
      print(``\n'');
  ```

  will only print the newline after `some_condition` is false. You should add braces before and after the loop to get correct behavior.

- Printing: C0 does not print anything until it sees a newline. This can cause things to get printed at unexpected times when you are debugging your program. You should ALWAYS print a newline after any string you print, using either `print("\n")` or `println("")`. `println` will work for any string: `println("Hello!")` is the same as `print("Hello!\n");`

Using contracts to debug is invaluable. If you can catch array out of bound errors or arithmetic before they happen, the extra information contract failures give you could save hours of debugging.

Print statements are also very useful to help investigate *why* your contracts are failing or your code is returning strange results. They let you examine the values of variables and see where things go wrong.

Another useful tactic is to use a small example, and see what your code does with it by evaluating your code *by hand*. When you evaluate by hand, you can see exactly where a mistake happens as soon as possible, allowing you to catch and fix it quickly.

Binary search

Binary search lets us search arrays *substantially* more quickly than linear search does.

The basic idea behind binary search is that if we're searching for `x`, we look in the middle of a sorted array and compare that element to `x`. If that element is smaller than `x`, we look in the top half of the array and if that element is bigger than `x`, we look in the bottom half of the array. (If that element is equal to `x`, then we're done.)

We're going to work through a few examples on the board to illustrate the number of steps binary search takes on arrays in practice, but the theoretical view is as follows: On every iteration of the loop, we roughly cut in half the amount of the array that we still have to look at—at every step, we throw out half what's left of the array.

So, we look at half of the array, and we then look at half of that, and so on.

Checkpoint 1

How many halvings will it take until we're looking at 1 element?

Solution: We're looking for i such that $\frac{n}{2^i} = 1$, or $n = 2^i$. The solution to this, of course, is $\log_2(n) = i$. This gives a rough approximation of how the algorithm's performance changes as the input array size grows. We'll talk more formally about this next week.

Checkpoint 2

Here's the code for binary search.

```
int binsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (-1 == \result && !is_in(x, A, 0, n))
      || ((0 <= \result && \result < n) && A[\result] == x);
 @*/
{
  int lower = 0;
  int upper = n;
  while (lower < upper)
  //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
  //@loop_invariant lower == 0 || A[lower-1] < x;
  //@loop_invariant upper == n || A[upper] > x;
  {
    int mid = lower + (upper-lower)/2;
    if (A[mid] < x) {
      // We can ignore the bottom half of the array now, since we
      // know that every thing in that half must be less than x
    } else if (A[mid] > x) {
      // We can ignore the upper half of the array, since we know
      // that everything in that half must be greater than x
      upper = mid;
    } else {
      //@assert A[mid] == x;
      return mid;
    }
  }
  //@assert lower == upper;
  return -1;
}
```

Give a proof for termination of binary search.

Solution: The loop must terminate since the interval between `upper - lower` is strictly decreasing with every iteration of the loop and we know by the loop guard that it has a lower bound of 0.

Binary search for integer square root

Recall the linear search for integer square root function we discussed last recitation. Here's a function that binary searches instead. (Note: this function has a bug related to integer overflow with sufficiently large inputs.)

```
#use <util>
int bin_search_sqrt (int n)
//@requires n > 0;
//@ensures \result * \result <= n;
//@ensures n < (\result+1) * (\result+1) || (\result+1)*(\result+1) < 0;
//@ensures \result == isqrt(n);
{
  int lower = 1;
  int upper = n;
  int mid = upper/2;
  int mid_plus_one_square = (mid + 1) * (mid + 1);
  while (!(mid * mid <= n
        && ((mid_plus_one_square > n) || mid_plus_one_square < 0)))
  // Note that the <= is necessary here because isqrt rounds down.
  //@loop_invariant lower <= isqrt(n);
  //@loop_invariant upper >= isqrt(n);
  {
    mid = lower + (upper - lower)/2;
    int square = mid * mid; // Only compute once for efficiency
    if ((mid != 0 && mid >= int_max() / mid) || square > n) {
      upper = mid;
    }
    else if (square < n) {
      lower = mid;
    }
    else {
      //@assert mid * mid == n;
      return mid;
    }
    mid_plus_one_square = (mid + 1) * (mid + 1);
  }
  return mid;
}
```

Note the similarities between this and binary search on an array. If you consider an array `A` of all nonnegative integers, where the ith entry of the array is i^2, we're simply searching for the i such that `A[i] <= n && A[i + 1] > n`. We could implement the function this way, but it would be a large waste of memory.

If we take the square roots of n 5,000,000 times, the UNIX utility `time` reports the following:

n	Binary search time (s)	Linear search time (s)
10,000	1.091	1.413
20,000	1.300	1.934
30,000	1.322	2.374
40,000	1.161	2.863
50,000	1.217	3.187
60,000	1.443	3.447
70,000	1.446	3.777
80,000	1.450	4.032

As you can see, in practice linear search takes much longer than binary search and the amount of time linear search takes increases far more quickly than binary search (as number of elements we're taking the square root of goes up). We'll formalize the notion of linear search being slower than binary search next lecture when we talk about big-O notation.