Midterm I Exam Review

15-122 Principles of Imperative Computation
Alex Cappiello

October 7, 2012

1 Modular Arithmetic

In C0, values of type `int` are defined to have 32 bits. In this problem we work with a version of C0 where values of type `int` are defined to have only 9 bits. In other respects it is the same as C0. All integer operations are still in two's complement arithmetic, but now modulo 2^9. All bitwise operations are still bitwise, except on only 9 bit words instead of 32 bit words.

Task 1. Fill in the missing quantities, in the specified notation.

a. The minimal negative integer, in decimal: -256

b. The maximal positive integer, in decimal: 255

c. -4, in hexadecimal: 0x 1FC

d. 44, in hexadecimal: 0x 2C

e. `0x49`, in decimal: 73

Task 2. Assume `int x` and `int y` have been declared and initialized to unknown values. For each of the following, indicate if the expression always evaluates to `true`, or if it could sometimes be `false`. In the latter case, indicate a counterexample in the C0 dialect described here by giving a value for `x` and `y` that falsifies the claim. You may use decimal or hexadecimal notation. You may use the functions `int_min()` and `int_max()` and assume they are correct for 9 bit ints.

a. `x >= x - 1` false, x = int_min()=-256

b. `(~x) ^ (x) == -1` true

c. `x+(y+1)-2*(x-1)-3 == -x+y` true

d. `(x!=-x || y!=-y) || x==y` false, x=0, y=int_min()=-256 or x=-256, y=0

e. `((x<<1)>>1) | (x & 0x100) == x` false, x = int_max()

f. `(((x>>1)<<1) | (x&1)) == x` true

g. `x <= (1<<(7-1))-1` false, `x=63,...,255`

h. `x+x == 2*x` true

2 Ternary Search.

Consider the following variation of binary search algorithm. Instead of checking the middle element of the sorted array $A[]$, check the element at position $n/3$. Then proceed in the same way as in binary search. If you are looking for x then

- if $x == A[n/3]$ you have found it.
- if $x < A[n/3]$ you search the first third of the array, namely at indexes $< n/3$.
- if $x > A[n/3]$ you search the rest two thirds of the array at indexes $> n/3$.

Consider the following implementation:

```
int tersearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, n);
/*@ensures (-1 == \result && !is_in(x, A, 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)/3;
      if (A[mid] < x)
        lower = mid+1;
      else if (A[mid] > x)
        upper = mid;
      else //@assert A[mid] == x;
        return mid;
    }
  //@assert lower == upper;
  return -1;
}
```

Task 1. Prove the correctness of this function, using the method presented in class.

Init: When the loop is first reached, we have $lower = 0$ and $upper = n$, so the first loop invariant follows from the precondition to the function. Furthermore, the first disjunct in loop invariants two (`lower == 0`) and three (`upper == n`) is satisfied.

Preservation: Assume the loop invariants are satisfied and we enter the loop:

$$\begin{array}{ll} 0 \leq lower \leq upper \leq n & \text{(Inv 1)} \\ (lower = 0 \text{ or } A[lower - 1] < x) & \text{(Inv 2)} \\ (upper = n \text{ or } A[upper] < x) & \text{(Inv 3)} \\ lower < upper & \text{(loop condition)} \end{array}$$

We compute $mid = lower + \lfloor (upper - lower)/3 \rfloor$. Now we distinguish three cases:

$A[mid] = x$: In that case we exit the function, so we don't need to show preservation. We do have to show the postcondition, but we'll come back to that.

$A[mid] < x$: Then

$$\begin{aligned} lower' &= mid + 1 \\ upper' &= upper \end{aligned}$$

The first loop invariant $0 \leq lower' \leq upper' \leq n$ follows from the formula for mid, our assumptions, and elementary arithmetic.

For the second loop invariant, we calculate:

$$\begin{array}{rcll} A[lower' - 1] & = & A[(mid + 1) - 1] & \text{since } lower' = mid + 1 \\ & = & A[mid] & \text{by arithmetic} \\ & < & x & \text{this case } (A[mid] < x) \end{array}$$

The third loop invariant is preserved, since $upper' = upper$.

$A[mid] > x$: Then

$$\begin{aligned} lower' &= lower \\ upper' &= mid \end{aligned}$$

Again, by elementary arithmetic, $0 \leq lower' \leq upper' \leq n$.

The second loop invariant is preserved since $lower' = lower$.

For the third loop invariant, we calculate

$$\begin{array}{rcll} A[upper'] & = & A[mid] & \text{since } upper' = mid \\ & > & x & \text{since we are in the case } A[mid] > x \end{array}$$

Postcondition: If we return from inside the loop because $A[mid] = x$ we return mid, so `A[\result] == x` as required. We know we are still in bounds because we assumed the first invariant and none of those variables have been changed.

If we exit the loop because $lower < upper$ is false, we know $lower = upper$, by the first loop invariant. Now we have to distinguish some cases.

1. If $A[lower - 1] < x$ and $A[upper] > x$, then $A[lower] > x$ (since $lower = upper$). Because the array is sorted, x cannot be in it.
2. If $lower = 0$, then $upper = 0$. By the third loop invariant, then either $n = 0$ (and so the array has no elements and we must return -1), or $A[upper] = A[lower] = A[0] > x$. Because A is sorted, x cannot be in A if its first element is already strictly greater than x.

3. If $upper = n$, then $lower = n$. By the second loop invariant, then either $n = 0$ (and so we must return -1), or $A[n-1] = A[upper-1] = A[lower-1] < x$. Because A is sorted, x cannot be in A if its last element is already strictly less than x.

Termination: Does this function terminate? If the loop body executes, that is, $lower < upper$, then the interval from $lower$ to $upper$ is non-empty. Moreover, the intervals from $lower$ to mid and from $mid + 1$ to $upper$ are both strictly smaller than the original interval. Unless we find the element, the difference between $upper$ and $lower$ must eventually become 0 and we exit the loop.

Task 2. Find an expression for the worst-case runtime of ternary search. Compare this asymptotically to binary search.

Initially, it might be confusing to approach this question because we change the interval of the array we're looking at depending on what happens inside our conditionals. If $x < A[n/3]$, we have $1/3$ as many elements left to look at. On the other hand, if $x > A[n/3]$ then we have $2/3$ as many elements left to look at. However, we're doing a worst-case analysis, so we will choose the second case each time, since this clearly gives us a worse runtime. Thus the expression we get is $\log_{\frac{3}{2}} n$.

We know that binary search is $O(\log n)$. Intuitively, since we also have a logarithmic expression, the two should be the same. However, this does not suffice as a proof. To do so, we invoke the change of base formula.

$$\log_{\frac{3}{2}} n = \frac{\log n}{\log \frac{3}{2}} = \frac{1}{\log \frac{3}{2}} \log n$$

Since we have found an expression of the form $c \log n$, we can now argue that ternary search is $O(\log n)$.

3 Runtime Analysis

Determine the big O complexity of the following expression:

$$T(n) = 4n^5 + 10n^3 + \log n$$

Use the formal definition of big O to justify your answer.

Intuitively, this is $O(n^5)$. However, this must be formally argued. Recall the formal definition of big O:

$f(n) \in O(g(n))$ if and only if there exists an n_0 and $c > 0$ such that $f(n) \leq c * g(n)$ for all $n \geq n_0$.

Therefore, we must find n_0 and c. So,

$$4n^5 + 10n^3 + \log n \leq ? cn^5$$

Again, this is intuitively clear, but does not constitute a proof. We note that:

$$4n^5 + 10n^3 + \log n \leq 4n^5 + 10n^5 + n^5 = 15n^5$$

Therefore, we let $c = 15$. We see that this expression holds as long as $n \geq 1$, so we will say that $n_0 = 1$.