15-122 : Principles of Imperative Computation

Summer 1 2012

Assignment 2

(Theory Part)

Due: Thursday, May 31, 2012 in class

Name: _____

Andrew ID: _____

Recitation: _____

The written portion of this week's homework will give you some practice working with searching algorithms and test your understanding of contracts. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture or recitation begins. Please remember to *staple* your written homework before submission.

Question	Points	Score
1	4	
2	8	
3	6	
4	4	
5	3	
Total:	25	

1. **C_0 Operators.** Let x be an `int` in the C_0 language. Express the following operations in C_0 using only one statement each. (Do not use an `if` statement here.) You should think about using some of the bitwise operators: (`&`, `|`, `^`, `~`, `<<`, `>>`).

(2) (a) Rotate x left one bit. (The leftmost bit reenters x in the rightmost position.) Store the result back in x.

Solution:

(2) (b) Rotate x right one bit. (The rightmost bit reenters x in the leftmost position.) Store the result back in x.

Solution:

2. **Reasoning with Invariants.** Consider the following implementation of the linear search algorithm that finds the last occurrence of `x` in array `A`:

```
int find(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
{
  int i = n-1;
  while (i >= 0 && A[i] >= x)
  {
    if (A[i] == x) return i;
    i = i - 1;
  }
  return -1;
}
```

Note that the function `is_sorted` is slightly different from the one we have been using in lecture—it takes a lower bound as well as an upper bound for the range of the array that we care about.

(2) (a) Add loop invariants to the code and show that the they hold for this loop. Be sure that the loop invariants precisely describe the computation in the loop.

Solution:

(2) (b) Add one or more `ensures` clause(s) to describe the intended postcondition in a precise manner.

Solution:

(4) (c) Show that the loop invariant is *strong enough* by using the loop invariant to prove that the postconditions hold at the end of the function (both if it ends by the return statement in the loop or the return statement after the loop exits).

Solution:

3. **Binary Search.**

(3) (a) An array can have duplicate values. A programmer wrote the following variant of binary search to find the first occurrence of x in a sorted array `A` of n integers so that the asymptotic complexity is still $O(\log n)$:

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

There is a bug in this implementation. Describe the bug and fix the code (and the annotations if necessary) so that it works correctly.

Solution:

(3) (b) 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 $> n/3$.

Is this algorithm is asymptotically faster or slower than binary search given in class? Explain your answer.

Solution:

4. **Runtime Complexity.** Consider the following function that sorts the integers in an array. (You may assume the code is correct so most annotations are not shown.)

```
int sort(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
{
  int i = 1;
  while (i < n)
  {
    int j = i;
    while (j != 0 && A[j-1] > A[j])
    {
      swap(A, j-1, j);    // function that swaps A[j-1] with A[j]
      j = j - 1;
    }
    i = i + 1;
  }
}
```

(2) (a) Let $T(n)$ be the number of comparisons needed to sort an array n elements. Using big-O notation describe the worst-case runtime complexity of `sort()`.

Solution:

$T(n) = O($

(2) (b) Using your answer from the previous part, prove that $T(n) = O(f(n))$ using the formal definition of big O. That is, find $c > 0$ and $n_0 \geq 0$ such that for every $n \geq n_0$, $T(n) \leq cf(n)$.

Solution:

(3) 5. **More on Contracts.** This question is designed to test your knowledge of contracts, how they are checked dynamically, and how they can be used to reason about the correctness of your program. Your job is to identify the locations in a C_0 function and a main function that calls it where contracts are checked and where you can assume that contracts must be true. Consider the `mult` function and a `main` function that calls it:

```
int mult(int x, int y)
//@requires x >= 0 && y >= 0;
//@ensures \result == x*y;
{
  int k = x;
  int n = y;
  int res = 0;
  while (n != 0)
  //@loop_invariant x * y == k * n + res;
  {
     if ((k & 1) == 1) res = res + n;
     k = k >> 1;
     n = n << 1;
  }
  return res;
}

int main()
{
  int a;
  a = mult(3,4);
  return a;
}
```

When you compile your C_0 program with the `-d` flag, it adds runtime tests to your program which are checked when it is executed. Based on the contracts for the `mult` function above, write `CHECK B` at any point in the copy of the function below where a boolean expression `B` is checked by a contract in the `main` and `mult` functions given that they are compiled with the `-d` flag. Note: not all blank lines below should be filled in.

```
int mult(int x, int y) {

  -------------------------
  int k = x; int n = y;
  int res = 0;

  ------------------------
  while (n != 0) {

      -------------------------
      if ((k & 1) == 1) res = res + n;
      k = k >> 1;
      n = n << 1;

      -------------------------
   }

  -------------------------
  return res;
}

int main() {
  int a;

  ------------
  a = mult(3,4);

  ------------
  return a;
}
```