15-122 : Principles of Imperative Computation

Fall 2012

Assignment 2

(Theory Part)

Due: Tuesday, September 25, 2012 at start of lecture

IMPORTANT: See last page for a change log of post-release notes.

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 at the beginning of lecture. Please remember to staple your written homework before submission.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>
1. **Reasoning with Invariants.** Consider the following implementation of the linear search algorithm that finds the first occurrence of \( x \) in array \( A \):

```cpp
int find(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
//@answer to 1.b goes here
{
    int i = 0;
    while (i < n && A[i] <= x)
        //@answer to 1.a goes here
        {
            if (A[i] == x) return i;
            i = i + 1;
        }
    return -1;
}
```

The function \( \text{is\_sorted} \) has the following signature:

```cpp
bool is\_sorted(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= \length(A) - 1;
//@requires 0 <= upper && upper <= \length(A);
//@requires lower <= upper;
;
```

and returns true if the array \( A \) is sorted in increasing order from \([lower, upper)\).

You may also use the function \( \text{is\_in} \) in any of the following questions. Its signature is:

```cpp
bool is\_in(int[] A, int x, int n)
//@requires 0 <= n && n <= \length(A);
;
```

and it returns true if \( A[i] == x \) for some \( i \) in \([0, n)\).
(2) (a) Add loop invariants to the while loop in the code and show that 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:
(5) (c) Show that the loop invariant is *strong enough* by using the loop invariant to prove that the postconditions hold when the function returns. You do not need to prove the loop invariant’s correctness (you already did that in 1.a); that is, you assume it is correct for this problem. Make sure to deal with the fact that `find` can return in two different ways.

**Solution:**
2. Binary Search.

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

```c
int binsearch_largest(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
/*@ensures ((\result == -1 && !is_in(x, A, n))
   || (0 <= \result && \result < n && A[\result] == x
    && (\result == n || A[\result+1] > x))); @*/
{
    int lower = -1;
    int upper = n-1;
    while (lower < upper)
    //@loop_invariant -1 <= lower && lower <= upper && upper <= n-1;
    //@loop_invariant lower == -1 || A[lower] <= x;
    //@loop_invariant upper == n-1 || A[upper+1] > x;
    {
        int mid = lower + (upper-lower)/2;
        if (A[upper] == x) return upper;
        if (A[mid] > x) upper = mid-1;
        else lower = mid;
    }
    //@assert lower == upper;
    return -1;
}
```

There are two bugs in this implementation. One bug is exclusively annotation-related, and the other is related directly to the code. That is, compiling without `-d` will cause the first bug to not appear.

(2) (a) Describe the annotation-related bug in `binsearch_largest`, and propose how to fix it.

Solution:
(5) (b) Describe the other bug in `binsearch_largest`, and propose how to fix it.

Solution:
(5) 3. Pentary Search.

Consider the following variation of binary search algorithm called pentary search. Instead of checking the middle element of the sorted array \(A\) (that is \(A[n/2]\)), we check the four elements at \(A[n/5], A[(2n)/5], A[(3n)/5], A[(4n)/5]\), and continue searching in the appropriate 1/5 of the array.

- if \(x\) is any of the four indexes we checked, we’re done (just return that index)
- if \(x < A[n/5]\) you search the first fifth of the array, namely at indexes \(< n/5\).
- if \(A[n/5] < x < A[(2n)/5]\) you search the second fifth of the array, namely at indexes \(>= (n/5)\) and \(< (2n)/5\).
- if \(A[(2n)/5] < x < A[(3n)/5]\) you search the third fifth of the array, namely at indexes \(>= ((2n)/5)\) and \(< (3n)/5\).
- if \(A[(3n)/5] < x < A[(4n)/5]\) you search the fourth fifth of the array, namely at indexes \(>= ((3n)/5)\) and \(< (4n)/5\).
- if \(A[(4n)/5] < x\) you search the last fifth of the array, namely at indexes \(>= ((4n)/5)\).

Compare the asymptotic complexity classes of pentary search and binary search. That is, is pentary search asymptotically faster, slower or the same as binary search? 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.)

```c
void 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;
    }
}
```

(a) Let \( T(n) \) be the worst-case number of comparisons made when \( \text{sort}(A, n) \) is called. Using big-\(O\) notation, what is asymptotic complexity of \( T(n) \)? This is the worst-case runtime complexity of \( \text{sort} \).

**Solution:**

\[ T(n) = O(\]
(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:
5. **More on Contracts.** This question is designed to test your knowledge of contracts, when they are checked, and how they can be used to reason about the correctness of your program. Your job is to identify the locations in a C₀ function and a main function that calls it where contracts are checked. Consider the `mult` function and a `main` function that calls it:

```c
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 C₀ 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. For example, if the contract `//@requires 0 <= n;` is checked, you would write `CHECK 0 <= n`. Note: not every blank line below should be filled in.

```c
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;
}
```

```c
int main() {
    int a;

    a = mult(3,4);

    return a;
}
```
Post-Release Notes

Below is a list of updates to the homework. In general, these are cosmetic changes, with the exception of an omitted precondition on is_sorted and an incorrect suggestion that you do not need to turn in your test cases.

Theory: Revision 2

- Some function names contain \ – you should ignore these
- In Section 1, (Reasoning with Invariants), the return condition for the function is_in should read: “and it returns true if and only if A[i] == x for some i in [0..n].”
- In Section 1 (Reasoning with Invariants), the preconditions for the function is_sorted were written loosely as:
  ```
  //@requires 0 <= lower && lower <= \length(A) - 1;
  //@requires 0 <= upper && upper <= \length(A);
  //@requires lower <= upper;
  ```
  and could be better written as:
  ```
  //@requires 0 <= lower && lower <= upper && upper <= \length(A);
  ```
- In Section 2 (Binary Search), the function binsearch_largest was missing a precondition for is_sorted. The function should also include the following precondition:
  ```
  //@requires is_sorted(A, 0, n);
  ```

Programming: Revision 2

- The preamble under Starter Code incorrectly stated that you should not turn in your testing code. You should turn in your testing code, as stated in other locations in the assignment.