15-122 : Principles of Imperative Computation

Midterm Exam - I

Fall 2011

SOLUTIONS

1. **Modular Arithmetic.**

   (a) For each of the following statements, indicate whether it is true or false. If the statement is true, explain why. If it is false, provide a counterexample.

(4) i. If x and y are ints in C_0, and $y >= x$, then $x + (y - x)/2 = (x + y)/2$.

> **Solution:** False, $x + y$ could lead to overflow.

(4) ii. If x is an int in C_0, then $(x << 2) >> 2 = x$.

> **Solution:** False, if `x = 0x80000000` then `(x<<2)>>2 = 0`

(4) iii. If x, y and z are ints in C_0, then $(x + y) + z = x + (y + z)$.

> **Solution:** True, modulo computation

   (b) Answer the following questions.

(4) i. If your machine implements 16-bit signed `ints`, what would be the `max_int` in hex?

> **Solution:** `x = 0x7FFF`

(4) ii. Find the 16-bit two's complement of `max_int` from part (iv). Is it equal to `min_int`? If not, what is the `min_int` in hex ?

> **Solution:** 1's complement of `0x7FFF` is `0x8000`
> + 1 = `0x8001`
> which is not `min_int`.

(5) iii. Write a C_0 function `bit()` which returns a bit at index `i` in the two's complement representation of `n`. Also supply a postcondition capturing the value range of the output.

> **Solution:**
> ```
> int bit(int n, int i)
> //@requires 0 <= i && i < 32;
> //@ensures \result == 0 || \result = 1;
> {
>    return (n >> i) & 1;
> }
> ```

(5) (c) Implement a function `iushr(n, k)` which is like n `>>` k except that it fills the highest bits with zeros instead of copying the sign bit. `iushr` stands for integer unsigned shift right.

Solution:

```
int iushr(int n, int i)
{
   return (n >> k) & (0x7FFFFFFF >> k);
}
```

2. **Searching.** Shown here is a binary search program that finds the largest index n of the target `key` in the array. For example, if `A = [1 2 3 3 4 4 4]`, then `search_largest(4, A, 7)` will return 6, and `search_largest(1, A, 7)` returns 0.

```
int search_largest(int key, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, n);
{
   int lower = 0;
   int upper = n;
   while (lower < upper)
   {
      int mid = lower + (upper-lower)/2;
      if (key >= A[mid]) lower = mid+1;
      else upper = mid;
   }
   //@assert lower == upper;
   if ( ????? )
      return lower - 1;
   else
      return -1;
 }
```

(4) (a) Write in the box below the missing if-condition at the end of the function so that the proper value is returned.

Solution:

```
lower != 0 && A[lower-1] == key
```

(5) (b) Write in the box below postconditions for the successful search, namely when `search_largest` returns the valid index.

Solution:

```
//@ensures A[\result] == key &&
           (\result == n-1 || A[\result+1] > key);
```

(5) (c) Write in the box below postconditions for the unsuccessful search, namely when `search_largest` returns -1. You may use function `is_sorted()` and `is_in()`.

Solution:

```
//@ensures \result == -1 && !is_in(key, A, n);
```

(5) (d) Write in the box below loop invariants for the successful search.

Solution:

```
//@loop_invariant 0 <= lower && lower <= upper && upper <= n;
```

(5) (e) Write in the box below loop invariants for the unsuccessful search.

Solution:

```
//@loop_invariant (lower == 0 || A[lower-1] <= key) &&
                  (upper == n || A[upper] > key);
```

(4) (f) There are the two most common methods of searching arrays for an element: linear and binary search. You know that the latter is asymptotically faster than the former. However, linear search has some advantages compare to binary search. When will you be using linear search instead of binary search? Circle the T or the F on *each line* below to indicate true or false.

(T F) The input data is sorted.

(T F) The input data is random.

(T F) The input data is sorted but of a relatively small size.

(T F) The input data is sorted though it has too many duplicates.

Solution: F, T, T, F

(10) 3. **Arrays.** Given a sorted in ascending order list of `ints`, write an insert function, that inserts a new element `key` into the array in order. The function should return a new array (of a **bigger** size-!) with the `key` inserted. Your loop invariants (together with the function preconditions) should be strong enough to guarantee the postconditions. You may use the helper function `bool is_sorted(int[] A, int lower, int upper)` that returns true if the array is sorted in ascending order

Solution:

```
int[] insert_inorder(int key, int[] A, int len)
//@requires 0 <= len && len <= \length(A);
//@requires is_sorted(A, 0, len);
//@ensures 0 < \length(\result) && len+1 <= \length(\result);
//@ensures is_sorted(\result, 0, len+1);
{
int[] ar = alloc_array(int, len + 10); //a bigger size
int k = 0;
while(k < len && A[k] <= key)
//@loop_invariant 0 <= k && k <= len;
//@loop_invariant k == 0 || A[k-1] <= key;
{
   ar[k] = A[k];
   k++;
}
//@assert(k == len || A[k] > key);
ar[k] = key;
for(; k < len; k++) ar[k+1] = A[k];
return ar;
}
```

4. **Sorting.**

(4) (a) External sorting is required when the data being sorted do not fit into the main memory of a computer. An external sort makes the use of external memory such as hard disks Which of the following sorting algorithms is the most suitable for external sorting.

A. quicksort

B. mergesort

C. bubble sort

D. insertion sort

Solution: B - sort data in pieces, write them back to disk, merge by reading two files and compare items.

(4) (b) There is an array of 100 million records of people each of which consists of different criteria such as name, birth year, zodiac sign and so on. You want to study the distribution of names according to other criteria. So what you need is to sort the array with respect to one criteria and then do further sorting with respect to another. However, you have to make sure that the next sort will preserve the order of records obtained by the previous sort. Which sort shall you use?

A. quicksort

B. mergesort

C. bubble sort

D. insertion sort

Solution: B - we need a stable sort

(4) (c) The merge function employed by mergesort as discussed in lecture allocates some auxiliary space each time it is called. If we call mergesort with an array of size n, how much extra space does mergesort allocate, overall? Choose the tightest bound from the list.

A. $O(1)$

B. $O(n \log n)$

C. $O(\log n)$

D. $O(n)$

Solution: B
During the merging step we first merge two arrays of size 1 (there are $n/2$ of them), than - of size 2 (there are $n/4$ of them) and so on until $n/2$:

$$2 * n/2 + 4 * n/4 + \ldots + n/2 * 2 = \Theta(n \log n)$$

5. **Asymptotic Complexity.**

(5) (a) What is the worst-case runtime complexity (in terms of n) of the following code using big-O notation?

```
for (int i = 0; i < n; i++) {
   for (int j = 1; j < i; j++) {
      for (int k = 1; k < j; k++) {
         printint(i + j + k);
      }
   }
}
```

Solution: $O(n^3)$

(5) (b) What is the worst-case runtime complexity (in terms of n) of the following code using big-O notation?

```
//@assert n >= 0;
int k = 1;
while ( k <= n) {
   k = 2 * k;
}
```

Solution: $O(\log n)$

(5) (c) What is the best-case runtime complexity (in terms of n) of the following code using big-O notation?

```
for (int k = 1; k < n; k++) {
   int value = a[k];
   int j = k;
   while (j > 0 && a[j-1] > value) {
      a[j] = a[j-1];
      j--;
   }
   a[j] = value;
}
```

Solution: $O(n)$

(5) (d) Prove that $3n^2 + 2n + 10 = O(n^2)$ using the formal definition of big-O. That is, find $c > 0$ and $n_0 \geq 0$ such that for every $n \geq n_0$, $3n^2 + 2n + 10 \leq c\,n^2$.

Solution:

Since

$$3n^2 + 2n + 10 \leq 3n^2 + 2n^2 + 10n^2 = 15n^2$$

for $n \geq 1$, we choose $c = 15$ and $n_0 = 1$.