15-122: Principles of Imperative Computation, Spring 2011
Assignment 2: Searching and Strings

Karl Naden (kbn@cs)

Out: Thursday, January 27, 2011
Due: Thursday, February 3, 2011

(Written part: before lecture,
Programming part: 11:59 pm)

1 Written: (20 points)

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 begins. Please remember to *staple* your written homework before submission.

Exercise 1 (8 pts)**.** 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;
}
```

(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.

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

(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).

Exercise 2 (2 pts)**.** 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: (`&`, `|`, `^`, `~`, `<<`, `>>`).

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

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

Exercise 3 (4 pts)**.** 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 /*@assert(A[mid] >= x); @*/ 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.

Exercise 4 (6 pts)**.** 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 that we went over in recitation 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;
}
```

(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 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). (**Hint:** The first check (a check of the loop invariant just before the first check of the loop guard) is provided.

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

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

  CHECK x * y == k * n + res;
  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;
}
```

(b) When reasoning about programs using contracts, there are certain points in where we can reason assuming that a certain fact is true based on what we know from the contracts. In the copy of the function below, write `KNOW B` at any point in the program where you know a boolean expression `B` to be true (Note: not all blank lines below should be filled in). (**Hint:** As an example, one of the `KNOW` statement is provided. You know at the beginning of the function that the precondition holds)

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

  KNOW x >= 0 && y >= 0
  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;
}
```

2 Programming: String Processing (30 points)

For the programming portion of this week's homework, you'll write three C_0 files corresponding to three different string processing tasks: `duplicates.c0` (described in Section 2.2), `common-unsorted.c0`, and `common-sorted.c0` (described in Section 2.3).

You should submit your code electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

Starter code. Download the file `hw2-starter.zip` from the course website. When you unzip it, you will find two C_0 files, `stringsearch.c0` and `readfile.c0`. The first contains linear and binary search as developed in class, adapted to work over `string` arrays. The second contains functionality for reading a text file into an array of strings with its length, a type called `string_bundle`.

```
string_bundle read_words(string filename);
```

You need not understand anything about this type other than that you can extract its underlying `string` array and the length of that array:

```
string[] string_bundle_array(string_bundle b);
int string_bundle_length(string_bundle b);
```

You can assume that all the strings returned have been converted to lowercase. You will also see a `texts/` directory with some sample text files you may use to test your code.

For this homework, you are not provided any `main()` functions. Instead, you should write your own `main()` functions for testing your code. You should put this test code in separate files from the ones you will submit for the problems below. You should not submit your testing code.

You should not modify or submit the starter code.

Compiling and running. You will compile and run your code using the standard C_0 tools. For example, if you've completed the program `duplicates` that relies on functions defined in `stringsearch.c0` and you've implemented some test code in `duplicates-test.c0`, you might compile with a command like the following:

```
cc0 stringsearch.c0 duplicates.c0 duplicates-test.c0
```

Don't forget to include the `-d` switch if you'd like to enable dynamic annotation checking.

Submitting. Once you've completed some files, you can submit them by running the command

```
handin -a hw2 <file1>.c0 ... <fileN>.c0
```

The `handin` utility accepts a number of other switches you may find useful as well; try `handin -h` for more information. As with the first assignment, the handin script will ensures that your submissions compile and will run some basic tests on your code. Remember to write your own tests as we reserve the right to run other tests while grading.

You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. For this assignment, we have provided the pre- and postconditions for many of the functions that you will need to implement. However, you should provide loop invariants and any assertions that you use to check your reasoning. If you write any "helper" functions, include precise and appropriate pre- and postconditions.

You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations are weak or missing.**

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on the course bboard (`academic.cs.15-122`) if you're unsure of what constitutes good style.

2.1 String Processing Overview

The three short programming problems you have for this assignment deal with processing strings. In the C_0 language, a `string` is a sequence of characters. Unlike languages like C, a string is not the same as an array of characters. (See section 8 in the C_0 language reference and section 2.2 of the C_0 library reference for more information on strings). There is a library of functions you can use to process strings:

```
// Returns the length of the given string
int string_length(string s);

// Returns the character at the given index of the string.
// If the index is out of range, aborts.
char string_charat(string s, int idx)
  //@requires 0 <= idx && idx <= string_length(s);
```

```
  ;

// Returns a new string that is the result of concatenating b to a.
string string_join(string a, string b)
  //@ensures string_length(\result) == string_length(a) +
                                       string_length(b);
  ;

// Returns the substring composed of the characters of s beginning at
// index given by start and continuing up to but not including the
// index  given by end.  If end <= start, the empty string is returned
string string_sub(string a, int start, int end)
  //@requires 0 <= start && start <= string_length(a);
  //@requires start <= end && end <= string_length(a);
  //@ensures string_length(\result) == end - start;
  ;

bool string_equal(string a, string b);

int string_compare(string a, string b)
  //@ensures -1 <= \result && \result <= 1;
  ;
```

The `string_compare` function performs a *lexicographic* comparison of two strings, which is essentially the ordering used in a dictionary, but with character comparisons being based on the characters' ASCII codes, not just alphabetical. For this reason, the ordering used here is sometimes whimsically referred to as "ASCIIbetical" order. A table of all the ASCII codes is shown in Figure 1.

The ASCII value for `'0'` is 0x30 (48 in decimal), the ASCII code for `'A'` is 0x41 (65 in decimal) and the ASCII code for `'a'` is 0x61 (97 in decimal). Note that ASCII codes are set up so the character `'A'` is "less than" the character `'B'` which is less than the character `'C'` and so on, so the "ASCIIbetical" order coincides roughly with ordinary alphabetical order.

2.2 Required: Removing Duplicates

In this programming exercise, you will take a sorted array of strings and return a new sorted array that contains the same strings without duplicates. The length of the new array should be just big enough to hold the unique strings. Place your code for this section in a file called `duplicates.c0`.

Task 1 (3 pts)**.** Implement a function matching the following prototype:

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Figure 1: The ASCII table (from http://ascii-table.com/img/table.gif)

```
bool is_unique(string[] A, int n)
    //@requires 0 <= n && n <= \length(A);
    //@requires is_sorted(A, 0, n);
    ;
```

where `n` represents the number of strings in the array `A`. This function should return `true` if the given string array contains no repeated strings and `false` otherwise.

Task 2 (3 pts)**.** Implement a function matching the following prototype:

```
int count_unique(string[] A, int n)
    //@requires 0 <= n && n <= \length(A);
    //@requires is_sorted(A, 0, n);
    ;
```

where `n` represents the number of strings in the array `A`. This function should return the number of unique strings in the array, and your implementation should have an appropriate asymptotic running time given the precondition.

Task 3 (6 pts)**.** Implement a function matching the following prototype:

```
string[] remove_duplicates(string[] A, int n)
    //@requires 0 <= n && n <= \length(A);
    //@requires is_sorted(A, 0, n);
    //@ensures \length(\result) == count_unique(A, n);
    //@ensures is_sorted(\result, 0, \length(\result));
    //@ensures is_unique(\result, \length(\result));
    ;
```

where `n` represents the number of strings in the array `A`. The strings in the array should be sorted before the array is passed to your function. This function should return a new array that contains only one copy of each distinct string in the array `A`. Your new array should be sorted as well. Your implementation should have an appropriate asymptotic running time given the preconditions.

You must include annotations for the precondition(s), postcondition(s) and loop invariant(s) for each function. You may include additional annotations for assertions as necessary. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function.

2.3 Required: Counting Common Words

In this exercise, you will write two functions for counting the number of words from a text that appear in a word list. A practical application of such a function would be determining how many words in the *Complete Works of Shakespeare* are valid in the game Scrabble.

For the following tasks, you may find the functions in `stringsearch.c0` to be useful!

Task 4 (6 pts)**.** Create a file `common-unsorted.c0` containing a function `common_unsorted` that matches the following signature:

```
int common_unsorted(string[] dictionary, int d, string[] wordlist, int w)
    //@requires 0 <= d && d <= \length(dictionary);
    //@requires 0 <= w && w <= \length(wordlist);
    //@requires is_sorted(dictionary, 0, d) && is_unique(dictionary, d);
    ;
```

The function should return the number of words in the array `wordlist` that also appear in the array `dictionary`. (If a word appears multiple times in the `wordlist`, you should count each occurrence separately.) Your function should be asymptotically efficient given the preconditions; analyze its running time using big-O notation in a comment in your source code. Note that a precondition of `common-unsorted` is that the `dictionary` must be sorted, a fact you should exploit.

Task 5 (6 pts)**.** Create a file `common-sorted.c0` with a function `common_sorted` that matches the following signature:

```
int common_sorted(string[] dictionary, int d, string[] wordlist, int w)
    //@requires 0 <= d && d <= \length(dictionary);
    //@requires 0 <= w && w <= \length(wordlist);
    //@requires is_sorted(dictionary, 0, d) && is_unique(dictionary, d);
    //@requires is_sorted(wordlist, 0, w);
    ;
```

As above, the function should return the number of words in the array `wordlist` that also appear in the array `dictionary`, and your function should be asymptotically efficient; analyze its running time using big-*O* notation in a comment in your source code. Note the additional precondition in the specification above: this function requires not only that the `dictionary` be sorted, but also that the contents of the `wordlist` be sorted.

Task 6 (4 pts)**.** Create a file `common-test.c0` which contains a function `int common_test()` that uses the functionality from `readfile.c0` to read in the *Complete Works of Shakespeare* (`texts/shakespeare.txt`) and the Scrabble word list (`texts/scrabble.txt`) and answer the question of how many words from Shakespeare's writing are in the Scrabble dictionary.

You must include annotations for the precondition(s), postcondition(s) and loop invariant(s) for each function. You may include additional annotations for assertions as necessary. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function.

2.4 Optional: Judges' Prize

Judges' prizes for this assignment will be awarded to the first 6 students to submit a wordle (www.wordle.net) of the 20 most prevalent Scrabble words from Shakespeare's collected works (**Note:** you may chose another reasonably sized body of work that you have access to the text version of if you choose. If you would like to go down this route, Project Gutenberg (www.gutenberg.org) is a good place to look for texts). You will need to use a sorted version of *Complete Works of Shakespeare* (`texts/shakespeare_sorted.txt`) to find which words are frequently used. You will then check that they are in the Scrabble dictionary and then add them to your wordle with their relative weights. Note that to find "interesting" words, you may want to restrict the length of the words you choose as well. Paste your weighted list of words (more copies of a single word means heavier weight) into the wordle creation interface found on the wordle website and submit it by emailing a pdf to kbn@cs.