# 15-122: Principles of Imperative Computation, Fall 2010
# Assignment 2: Searching, Sorting, and Strings

William Lovas (wlovas@cs) Tom Cortina (tcortina@cs)

Out: Wednesday, September 8, 2010
Due: Tuesday, September 14, 2010

(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 the searching and sorting, reasoning with invariants and working with big-*O* notation. 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.

### 1.1 Warmup

**Exercise 1** (6 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 loop invariants hold for this loop. (**Hint**: Recall the linear search example from Lecture 3.)

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

(c) Suppose you are looking for the integer 1,000,000 (1 million) in the array. Does it pay to look for it by scanning from the end of the array as we do above rather than from the beginning of the array as we did in lecture? Why or why not?

**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))
        || (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.

## 1.2 Run-time Complexity

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

```
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;
  }
}
```

(a) Let $T(n)$ be the number of comparisons between array elements as a function of $n$. What is $T(n)$ in the worst cast? Justify your answer.

(b) Using big-$O$ notation, we can say that this function has a worst-case running time complexity of $O(f(n))$. What is $f(n)$ in this case? Use the simplest form of $f(n)$ that is possible to describe the run-time complexity of the function.

(c) Using your answers from the previous two parts, show that $T(n) \in O(f(n))$ using the formal definition of big $O(-)$. (That is, find a $c > 0$ and $n_0 \geq 0$ such that for every $n \geq n_0$, $T(n) \leq cf(n)$.)

**Exercise 5** (4 pts)**.** For each of the following problems, a programmer has a function that processes an array with $n$ integers in it. The programmer wrote down the value of $n$ along with the execution time for each algorithm. Using this information, determine the running time complexity of the algorithm as a function of $n$ using big-$O$ notation in its simplest form and briefly explain how you found each answer.

(a)
```
n     Execution Time (in seconds)
1000  0.743
2000  2.915
4000  11.896
8000  48.108
```

(b)

| n | Execution Time (in seconds) |
|---|---|
| 1000 | 0.584 |
| 2000 | 1.190 |
| 4000 | 2.367 |
| 8000 | 4.801 |

(c)

| n | Execution Time (in seconds) |
|---|---|
| 1000 | 0.934 |
| 2000 | 0.949 |
| 4000 | 0.963 |
| 8000 | 0.979 |

(d)

| n | Execution Time (in seconds) |
|---|---|
| 1000 | 0.996 |
| 2000 | 2.193 |
| 4000 | 4.786 |
| 8000 | 10.372 |

## 2 Programming: String Processing (30 points)

For the programming portion of this week's homework, you'll write two $C_0$ files corresponding to two different string processing tasks: `duplicates.c0` (described in Section 2.2) and `common.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, but you may additionally submit your test code for extra credit—use any file names you like, as long as they are distinct from the ones we require you to submit.

You should not modify or submit the starter code.

**Compiling and running.** For this homework, we are not providing you with a script to compile your code. Instead, you will use the standard `cc0` command with command line options. For this assignment, you will need to use libraries for file I/O (`file`), console I/O (`conio`), and strings (`string`). To compile one of your files against one of our `main()` functions, just specify both files on the command line along with library switches `-l<lib>` (dash-lowercase-L) for each library `<lib>`. For example, if you've completed the program `common` and you've implemented some test code in `common-test.c0`, you might compile with a command like the following:

```
cc0 common.c0 common-test.c0 -lfile -lconio -lstring
```

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.

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. 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.) 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 a, int idx);

// Returns a new string that is the result of concatenating b to a.
string string_join(string a, string b);
```

```
// Returns the substring composed of the characters of s beginning
// at index given by start and up to but not including the index
// given by end>
// If end <= start, the empty string is returned
// If end < 0 or end > the length of the string, it is treated as
//    though it were equal to the length of the string.
// If start < 0 the empty string is returned.
string string_sub(string a, int start, int end);

// Returns true if a and b are exactly the same, character for
// character, Returns false otherwise
bool string_equal(string a, string b);

// Returns a negative integer if a is lexicographically "less" than
// b, a positive integer if a is lexicographically "greater" than b,
// or 0 if a and b are equal.
int string_compare(string a, string b);
```

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:

```
bool is_unique(string[] A, int n)
//@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.

| | 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)

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

```
int count_unique(string[] A, int n)
//@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 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. Place your code for this section in a file called `common.c0`.

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

**Task 4** (6 pts)**.** Write a function `common1` matching the following prototype:

```
int common1(string[] wordlist, int w, string[] book, int b)
//@requires w <= \length(wordlist) && b <= \length(book);
//@requires is_sorted(wordlist, 0, w) && is_unique(wordlist, w);
;
```

The function should return the number of words in the array `book` that also appear in the array `wordlist`. (If a word appears multiple times in the `book`, 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 `common1` is that the `wordlist` must be sorted, a fact you should exploit.

**Task 5** (6 pts)**.** Write a function `common2` matching the following prototype:

```
int common2(string[] wordlist, int w, string[] book, int b)
//@requires w <= \length(wordlist) && b <= \length(book);
//@requires is_sorted(wordlist, 0, w) && is_unique(wordlist, w);
//@requires is_sorted(book, 0, b);
;
```

As above, the function should return the number of words in the array `book` that also appear in the array `wordlist`, 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 `wordlist` be sorted, but also that the contents of the `book` be sorted.

**Task 6** (4 pts)**.** Write a function `int common_test()` that uses the functionality from `readfile.c0` to read in the *Complete Works of Shakespeare* (`texts/shaks12.txt`) and the Scrabble word list (`texts/scrabble.txt`) and answer the question of how many of the Bard's words are fair game in Scrabble.

**Extra Credit Task 1.** Implement a subquadratic sort—possibly the one seen in lecture—and use it to test that your `common1` and `common2` functions compute the same answer for the Shakespeare/Scrabble task above. Submit the test code in a file `common_compare.c0`.

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 Required: Brief Course Survey

**Task 7** (2 pts)**.** Create a file `README` answering the following questions:

(a) How long did it take you to complete the written portion of this homework?

(b) How long did it take you to complete the programming portion of this homework?

Submit this file along with the rest of your code. If you wish, you may also include any additional explanations of your testing code in the `README` file.

### 2.5 Optional: Judges' Prize

As a judges' prize for this assignment, we will award the student whose code contains the most elegant and comprehensize annotations.