15-122: Principles of Imperative Computation, Spring 2011
Assignment 3: Sorting, Stacks, Queues and Word Ladder

Ananda Gunawardena(guna@cs.cmu.edu)

Out: Tuesday, February 8, 2011
Due (Written): Tuesday, February 15, 2011 (before lecture)
Due (Programming): Tuesday, February 15, 2011 (11:59 pm)

1 Written: (25 points)

The written portion of this week's homework will give you some practice working with sorting algorithms, stacks and queue data structures and performing some asymptotic analysis tasks. 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 Run-time Complexity

Exercise 1 (9 pts)**.** Consider the following function that performs some operations on an array of integers. (You may assume that the code operates as intended, and sorts the array)

```
void sort(int[] A, int n)

{ int i=0;
  while (i < n)
  { int j = findmin(A, i, n);
    // findmin(A, i, n) is a function that returns the index j
    // of the min element in the array segment A[i..n).
    // for example if A={1,2,8,4,5} then findmin(A,2,5) returns 3
    swap(A, i, j);   // a function that swaps A[i] with A[j]
    i = i + 1;
  }
}
```

(a) Improve the code (if possible) and write all annotations (pre-conditions, assertions, loop invariants and post-conditions) required for this code to run with the -d flag. You may use functions such as `is_sorted`, `lt`, `gt`, `leq`, `geq` and other functions in `qsort`. You are allowed to use any function from lecture to write your annotations.

(b) Consider the original code given (not your updated code in part (a)). Assume that findmin requires $i - 1$ "operations" to find the minimum of an unsorted array segment of size i. Let $T(n)$ be the total "operations" in calling `findmin` on an array A of size n in the function `sort`. Express $T(n)$ as a function of n in closed form (that is, do not include "..." in your final answer). Show all work.

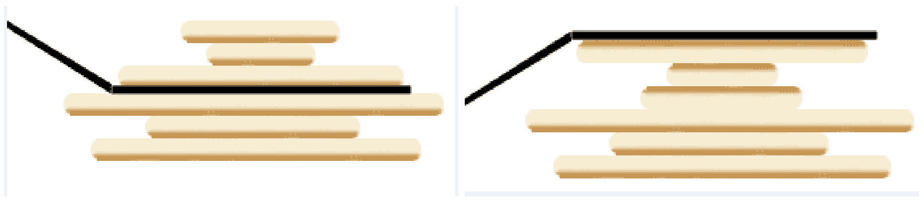
(c) We say that $T(n)$ is $O(f(n))$ if there exist a constant $c > 0$ and $n_0 \geq 0$ such that $T(n) \leq cf(n)$ for all $n \geq n_0$. Using the results obtained in previous part, show that $T(n)$ is $O(n^2)$: find some values of c and n_0 that shows that your $T(n)$ is $O(n^2)$.

Exercise 2 (4 pts)**.** Following run times were obtained when using two different algorithms on a data set of size n. You are asked to determine asymptotic complexity of the algorithms based on this time data. Determine the asymptotic complexity of each algorithm as a function of n. Use big-O notation in its simplest form and briefly explain how you reached the conclusion. Note that you may use of any of the standard big-O complexities we discussed in class.

(a)
```
n           Execution Time (in seconds)
1000        0.743
2000        3.021
4000        12.184
8000        50.320
```

(b)
```
n           Execution Time (in microseconds or millionths of a second)
1000        0.01
1000000     20
1000000000  30000
```

Exercise 3 (6 pts)**.** Pancake sort is a sorting algorithm that works as follows. Suppose you have a stack of pancakes that you would like to sort from smallest to largest such that largest pancake is at the bottom of the stack after sorting. However, the only operation allowed on this data structure is flipping (or making a sub stack up side down). Only the sub stacks that *contain the top element* can be flipped and we assume flip is an $O(1)$ operation. As an example, here is what happens when the top 3 pancakes are flipped upside down. The first image shows the original pancake stack before flipping the top 3 pancakes. The second image shows the pancake stack after flipping the top 3 pancakes.



source: wikipedia.com

(a) Here is a possible algorithm for sorting the pancakes. Assume that pancake is a struct defined as follows.

```
struct pancake {
    int size;
    string flavor;
};
typedef struct pancake* pancake;

void pancakesort(pancake[] A, int n){
  int i=n;
  while (i>1) {
    int max = findmax(A,0,i);
   //findmax(A,0,i) finds the index of the max size pancake in A[0..i)
   flip(A,0,max); // flip will reverse the array A[0..max+1)
   flip(A,0,i-1); // flip will reverse the array A[0..i)
   i=i-1;
  }
}
```

Write all annotations (pre-conditions, loop invariants, assertions and post-conditions) and prove that the above code correctly sorts the pancakes from smallest to largest. Again you can use functions used in class lectures to write your annotations.

(b) What is the worst case asymptotic complexity of your algorithm using big O notation in terms of n where n is the number of pancakes? Justify your answer.

1.2 Stacks and Queues

A `queue` and a `stack` that contains elements of of type `elem` are implemented with the following interfaces:

```
queue queue_new();
bool queue_empty(queue Q);
void enq(queue Q, elem e);
elem deq(queue Q);

stack stack_new();
bool stack_empty(stack S);
void push(stack S, elem e);
elem pop(stack S);
```

We do not know how these data structures are implemented. That is, we don't know if the programmer used arrays or linked lists, or something else—just that they somehow implemented the functions shown above, and that they have the same observable behavior as the ones we implemented in class.

Exercise 4 (6 pts)**.** Consider the interfaces for `queue` and `stack` as given above.

(a) Write a function `reverse` for reversing the order of a queue using a stack. Include proper annotations in your solution. What is the worst case run-time complexity of this function if there are n elements in the queue? Explain.

(b) Write a function `isdiff` that takes two stacks (of the same size with elem types) as its arguments and returns true if the stack elem order differs by at least one elem. That is, the stacks are not identical. You may assume that the function bool `isequal(elem e1, elem e2)` returns true if e1 is "equal" to e2. After the operations, both stacks must remain in the original order and you are allowed to use additional stacks or queues (but no other data structures) as needed. What would be the worst case asymptotic complexity of your code if each stack contains n elements each? Explain.

Helpful suggestion: Try to write these functions by hand without using the computer to test anything until you are very certain that your functions are correct. On your written exams, you will be asked to write some small amounts of code, and you won't have a compiler to help you check for syntax errors or correctness of your algorithm, so you should work on this exercise as we do in class, reasoning about your algorithm and code and writing down annotations to test your reasoning.

2 Programming: Word Ladders (25 points)

2.1 Overview

Word ladders were invented by Lewis Carroll in 1878, the author of *Alice in Wonderland*. A ladder is a sequence of words that starts at a starting word, and ends at an ending word (also called the "target"), and contains a sequence of words in between where each word differes by one letter from the word before it. Put another way, in a word ladder you have to change one word into another by altering a single letter at each step. Each word in the ladder must be a valid English word, and must have the same length. For example, to turn `stone` into `money`, one possible ladder is given below. Note that the letter that was changed is shown in the color red.

stone
atone
alone
clone
clons
coons
conns
cones
coney
money

Many word ladders have more than one possible solution, and there could be more than one shortest solution. For example another path from `stone` to `money` is:
`stone store shore chore choke choky cooky cooey coney money`
Your program must determine a *shortest* word ladder for a set of starting and ending words.

2.2 Instructions

Your program will accept a file that contains starting and ending words from the input file called "testwords.txt". The file testwords.txt is formatted as follows. Each line contains two words where the first word in each line is the starter word and the second word is the target word. You may assume that the file is valid. That is, each line contains two words of the same length

```
test case
graph paper
compute program
bug cat
computer programs
```

You may use readfile.c0 to process this file and break start and target words appropriately.

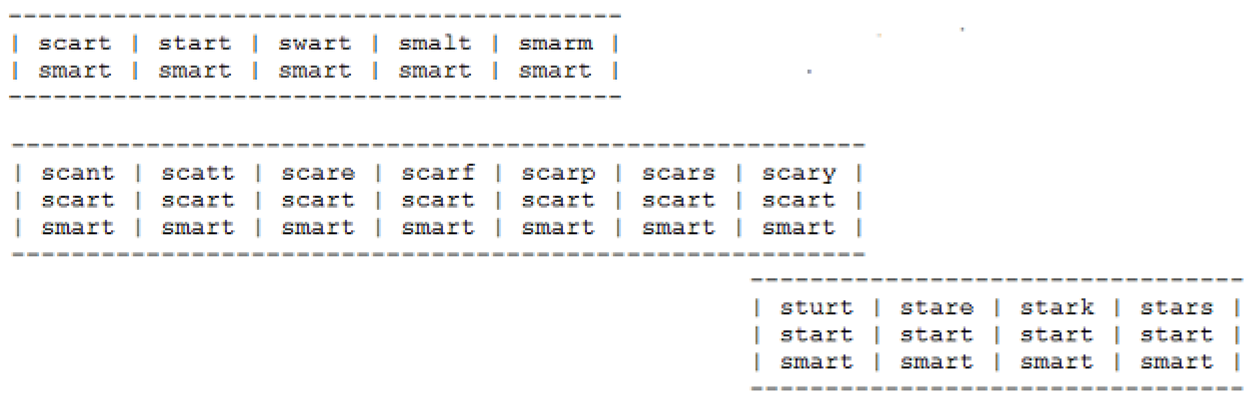
2.3 Suggested Algorithm.

There are several ways to solve this problem. One method involves using a queue of stacks to find word ladders. The algorithm works as follows.

Get the starter word and search through the dictionary to find all words that differ by one letter from the starter word. Let's call this set N. If any of these words in set N is the target word, then you are done. Otherwise create a new stack for each of the words in N. For each new stack you create, push the starter word first and then a word from the set N. (You should end up with N stacks.) Enqueue these stacks into the queue. Now deque the first stack from the queue, pop the top word out of it (this should be one of the words that belongs to set N), and find words that are one letter apart to that. Lets call this set M. If any one of the words in set M is the target word, then you are done: your stack words and the target word is the word ladder that takes you from start to target word. Otherwise for each word in M, create a new copy of the old stack and push the word from M on to the new stack. Enqueue all these stacks to the queue. You continue this process until a word path is found or the queue is empty. A graphical representation of this process is given in the figure below. You are allowed to use slight variations of this algorithm, but be sure to comply with the function prototypes provided.

Caution you have to keep track of the used words! Otherwise an infinite loop can occur. With a minor tweak to the code that processes the dictionary of equal length words, you can make this happen.

Example 1. Here is an example of how the queue of stacks solves this problem. The start word in this case is ”smart” and the target word is ”count”. and we find 5 words that are one letter apart from that. Since none of them is the target word, we create 5 new stacks and enque them into a queue. (See the figure on the next page.) Now we deque the first stack in the queue. That contains smart and scart. Find all words that are one letter apart from ”scart”. We find seven such words. Since none of them is the target word, we enqueue all of the new stacks into the queue. Then we take the second stack of size 2 (that contains start and smart and find all words that are one letter apart from start. We find 4 such words and since none of them is the target word, we enqueue these new stacks of size 3 into the queue. The following diagram shows 3 segments of the same queue.



```
-----------------------------------------
| scart | start | swart | smalt | smarm |
| smart | smart | smart | smart | smart |
-----------------------------------------

---------------------------------------------------------
| scant | scatt | scare | scarf | scarp | scars | scary |
| scart | scart | scart | scart | scart | scart | scart |
| smart | smart | smart | smart | smart | smart | smart |
---------------------------------------------------------
                                                  ---------------------------------
                                                  | sturt | stare | stark | stars |
                                                  | start | start | start | start |
                                                  | smart | smart | smart | smart |
                                                  ---------------------------------
```

Figure 2: Example of a queue of 16 stacks, with the front at the top left and the back at the bottom right.

We note that the queue is growing in size, and it contains all stacks of size 2, then size 3 and so on. This process will continue until the target word is found or we find an empty queue. As you can imagine there is quite a bit of memory copying here. Still our test files with this algorithm must terminate in a reasonable time.

Starter code. Download the file `hw3-starter.zip` from the course website. Inside, you'll find the following files. The one's in red are the files that are complete. Do not change them. The files in blue are the files that you need to complete.

`path-stacks.c0`	Stacks containing `paths` as required by the assignment
`elem.c0`	A type alias defining `elem` to be `path-stack`
`lists.c0`	Linked list containing `elems` as developed in lecture
`queue.c0`	Queue containing `elems` that are path-stacks
`readfile.c0`	Code for reading words from a file
`path-util.c0`	path utility functions for accessing path information
`path-search.c0`	search functions for finding path information
`path-main.c0`	the driver program for finding the word ladder

In this homework, we are asking you to write code to complete function prototypes provided in three files, `path-main`, `path-search` and `path-util`.

2.4 Required: Completing path-util

The purpose of this section is to develop the utility functions that you may need in the other parts of the program as well as to use in the annotations. You must maintain the following function prototypes. You can add more functions to be used in annotations.

Task 1 (2 pts)**.** Complete the function `word_distance` as specified below:
`int word_distance(string s1, string s2)`
The purpose of this function is to find the word distance between two given words of equal length. The function will find the number of places where two words differ. For example, "cat" and "bat" will return 1 and "frank" and "pranc" will return 2.

Task 2 (2 pts)**.** Complete the function `count_words` as specified below:
`int count_words(string[] dictionary,int dict_length,int n)`
The purpose of this function is to count words in the dictionary that are of length n. This function can be used by others. For example the function `find_words_of_length_n` below can use this function.

Task 3 (2 pts)**.** Complete the function `check_dictionary` as specified below:
`bool check_dictionary(string[] dictionary,int dict_length,int n)`
The purpose of this function is confirm (and return true) that all words in the dictionary are of length n. Return false otherwise.

Task 4 (2 pts)**.** Complete the function `words_rec find_words_of_length_n` as specified below:
`words_rec find_words_of_length_n(string[] dictionary, int dict_length, int n)`
The purpose of this function is to read a dictionary and returns a pointer to a struct containing an array of words and a count. For example,
`find_words_of_length_n(dictionary, 172000, 8)`
will return a pointer to a struct that contains all words in the dictionary (of size 172000) that are of length 8 and a count of the number of matching words. The struct `words_record`, and `words_rec` (pointer to a words_record) are defined as follows.

```
struct words_record {
   int count;
   string[] wordlist;
};
typedef struct words_record* words_rec;
```

The definition of a struct allows more information about the dictionary to be returned from the function. In this case, we return both the count and an array of strings in one struct. Be sure to allocate memory for `words_record` and `wordlist` before copying the words.

Task 5 (2 pts)**.** Complete the function `words_rec n_letter_apart` as specified below:
`words_rec n_letter_apart(string target, string[] dictionary, int dict_length, int n)`
The purpose of this function is to find all words from a dictionary that are n letters apart from the target word and return a struct `words_rec` (struct as defined above) that contains the total number of such words and the list of words that are n letters apart from the target. Be sure to allocate memory for `words_record` and `wordlist` before returning from the function.

2.5 Required: Completing path-search

This file should contain the functions required to successfully find (or not) a path from start to target word. We suggest that you complete the following function prototype. You may add other helper functions. But be sure to add annotations for all of your code.

Task 6 (10 pts)**.** Complete the function prototype:
`path_stack find_path(string start, string target, string[] reduced_dict, int dict_length);`
The function will take a dictionary that contains the words of equal length as start and target and returns a path_stack that contains the path found. If a path is not found return NULL. A `path_stack` is a special stack that contains paths stored in a stack. `path_stack` follows all standard stack operations such as `pop`, `push`, etc. Please refer to the `path_stack.c0` file for more information.

2.6 Required: Completing path-main

Task 7 (5 pts)**.** Complete `path_main.c0`. The purpose of `path_main` is to read a dictionary of words and a search word file (that contains pairs of start and target words) and find the word ladders for all pairs of words given in the file testwords.txt. If a path is not found between any pairs of words, print a message "path not found". A sample screen output file is given in the file sample_output.txt .

2.7 Compiling, running and submitting your program

Compiling and running. For this homework, use the `cc0` command as usual to compile your code. When compiling multiple files, be sure to list them in dependency order—the starter code is shown in dependency order above. Don't forget to test your annotations by compiling with the `-d` switch to enable dynamic checking.

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

```
    handin -a hw3 <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.

2.8 Optional: King's Prize

For the kings prize you are expected to find islands that no other word can get to. By an island we mean a word that has no path to any other word in the dictionary. You must find islands for all word lengths in the dictionary. The winners of the King's Prize are the first 3 students who find the complete island word list from the current dictionary and email the list and code that generated the list to `guna@cs.cmu.edu`. Good luck.