School of Computing and Information Systems

comp10002 Foundations of Algorithms
Semester 2, 2018
Assignment 1

Learning Outcomes

In this project you will demonstrate your understanding of arrays, strings, and functions. You may also use `typedefs` and `structs` if you wish (see Chapter 8) – and will probably find the program easier to assemble if you do – but you are not required to use them in order to obtain full marks. You should not make any use of `malloc()` (Chapter 10) or file operations (Chapter 11) in this project.

Superstring Assembly

Suppose that multiple copies of a long string are cut up into much smaller pieces. If you are given the available fragments, is it possible to rebuild the larger string? For example, if the original string was `"algorithmsarefunfunfun"`, and you got given a list of fragments like `"refun"` and `"unfun"` `"rith"` and `"lgo"` and so on, plus lots more small fragments, could you rebuild the original string? Probably you could, right? But what about if the original string was millions of characters long, like *War and Peace*? Or billions of characters long, like your genetic sequence?

Let's ask a slightly different question now: if you got given a set of small string fragments each from a larger string that you didn't know anything about, could you at least *find the shortest string in which every supplied fragment occurs at least once*? This problem is known as the *shortest superstring problem*, and can be thought of as having parameters n, the number of string fragments provided, and m, the total length of those string fragments. The shortest superstring problem is very important in bioinformatics, where genome reassembly from short fragments called *reads* helps contribute to our understanding of the behavior of organisms.

Unfortunately, finding optimal solutions to the shortest superstring problem has been shown to be very challenging, and falls into a family of intractable problems that we'll talk about later in the semester. In this project you will implement some approximation techniques that generate non-optimal superstrings from a collection of string fragments.

Input Data

Input to your program will (always) come from `stdin`, and will (always) consist of strictly lower-case alphabetic strings, one per input line, each line terminated by a single newline `'\n'` character. *Be sure to read the message on the FAQ page about newlines.* For example, the following file `test0.txt` containing six fragments is a valid input:

```
accgtcgatg
gcctag
gtacctacta
cgatgcc
tcgatgccgca
atgagaccgtc
```

As you develop your program according to the stages listed below, the output will evolve. Full output examples for this and two other test files are linked from the FAQ page. You should also check your program against other inputs too; and can easily generate your own tests using a text editor. Testing and debugging is *your* responsibility.

In terms of developing your program for this project, you may assume that no string fragment will be longer than 20 characters, and that there will be at most 1,000 such fragments supplied. You may also use brute-force programming rather than try and use more elegant data structures and algorithms – this activity is primarily about C programming rather than algorithm design.

Stage 0 – Reading Strings (0/15 marks)

Before tackling the more interesting parts of the project, your very first program should read the set of input fragments into an array of strings, and then print them out again so that you can be sure you have read them correctly. You can do that as part of your `DEBUG` mode code, see the suggestion below. Each of the following stages will then require a function that carries out the required processing, in each case working through the strings that were read, and then building (and selectively writing) the required superstring.

Stage 1 – Overlapping Suffixes (8/15 marks)

In this first stage you are to build a superstring according to the following process: start with the first of the input fragments and use it to initialize a partial superstring; then process every other fragment in input order, checking first to see if it is already wholly present in the partially built superstring, and if it is not, checking to see if there is any overlap between the head of the new fragment and the tail of the superstring. If the fragment appears already within the superstring, no further action is needed. Or, if there are overlapping characters, the required part of the fragment is appended to the end of the partial superstring. If there is no tail overlap, the whole of the fragment is appended to the partial superstring. For example, on the `test0.txt` file, the six input fragments result in this sequence of superstrings being formed:

```
Stage 0 Output
--------------
6 fragments read, 55 characters in total

Stage 1 Output
--------------
 0: frg= 0, slen= 10  Accgtcgatg
 1: frg= 1, slen= 15  AccgtcgatGcctag
 2: frg= 2, slen= 24  AccgtcgatGcctaGtacctacta
 3: frg= 3, slen= 24  AccgtCgatGcctaGtacctacta
 4: frg= 4, slen= 35  AccgtCgatGcctaGtacctactaTcgatgccgca
 5: frg= 5, slen= 45  AccgtCgatGcctaGtacctactaTcgatgccgcAtgagaccgtc
```

Pointer to fragments read and pointer to characters

where the three numbers shown are the operation number, the number of the fragment that is being added, and the new length of that partial superstring. Note how the letters that are at the start of each fragment have been capitalized in the output so that they can be identified in the superstring, and how one of the fragments was found within the partial superstring. At the end of the process there are 45 characters in the superstring, a saving of 10 compared to the original input size of $m = 55$ characters.

There is detailed example output provided on the FAQ page showing the required output for longer inputs that your program must match.

Stage 2 – Choosing Extension Fragments (13/15 marks)

The approach used in Stage 1 is simple, but a bit limited, since it only saves space if there is overlap between the first characters of one fragment and the last characters of the previous one. A better approach is to start by initializing the superstring to the first fragment (and marking that one as being processed), and then repeating these steps until every other fragment has also been processed:

- Check every unprocessed fragment; if it appears anywhere within the superstring, mark it as processed.
- Then, check every remaining unprocessed fragment, and calculate the length of the overlap between the beginning of the fragment and the end of the superstring;
- Choose the unprocessed fragment with the longest overlap, append it to the superstring, and then mark it as being processed.
- If two fragments have the same maximum overlap length (including the case when the overlap length is zero), select the one that appeared earliest in the input.

The Stage 2 output for `test0.txt` is, with (noted in the "`frg=`" column) the order $0, 4, 3, 5, 1, 2$, is:

```
Stage 2 Output
--------------
 0: frg= 0, slen= 10  Accgtcgatg
 1: frg= 4, slen= 15  AccgTcgatgccgca
 2: frg= 3, slen= 15  AccgTCgatgccgca
 3: frg= 5, slen= 25  AccgTCgatgccgcAtgagaccgtc
 4: frg= 1, slen= 31  AccgTCgatgccgcAtgagaccgtcGcctag
 5: frg= 2, slen= 40  AccgTCgatgccgcAtgagaccgtcGcctaGtacctacta
```

This superstring requires 40 characters. Again, see the FAQ page for full output requirements.

Stage 3 – Double-Ended Strings (15/15 marks)

Now for a challenge (if you really want the last two marks). Instead of regarding the partial superstring as being "append only", observe that it has two ends, and hence two points where it might have fragments joined to it. That is, instead of searching at each iteration for the unprocessed fragment that has the most overlap with the end of the partial superstring, you should also check for overlaps between the *head* of the partial superstring and the *tail* of each fragment.

Then, out of all the possibilities, choose the one that gives the most overlap. To break ties, choose the fragment that appeared earliest in the input, and if the same fragment number has the same overlap (including zero overlap) at head and at tail of the superstring, place the fragment at the end of the superstring (rather than prepending at the start of it).

You might need to make some quite wide-ranging changes in your program to accomplish this stage, so be sure to: (a) submit your Stage 2 program (see the instructions on the FAQ page); and then (b) save a copy of your Stage 2 program before you start restructuring it. Example Stage 3 output is provided at the FAQ page, for `test0.txt` the superstring takes just 34 characters.

Beyond the Scope of the Project

Later in the semester, when you have been taught about `malloc()` and dynamic memory, if you feel like some fun, then try this approach: check every pair of strings against each other at both ends, to find the two that have the longest head-to-tail overlap. Join those two strings together to exploit that overlap. Doing so reduces the set of strings by one, and hence after a total of $n - 1$ such steps, a single string will remain. It is guaranteed to be within a factor of three of the length of the optimal superstring. (But please don't submit any such programs for assessment, even if you get them working prior to the submission deadline.)

General tips...

You will probably find it helpful to include a `DEBUG` mode in your program that prints out intermediate data and variable values. Use `#if (DEBUG)` and `#endif` around such blocks of code, and then

`#define DEBUG 1` or `#define DEBUG 0` at the top. Turn off the debug mode when making your final submission, but leave the debug code in place. The FAQ page has more information about this.

The sequence of stages described in this handout is deliberate – it represents a sensible path though to the final program. You can, of course, ignore the advice and try and write final program in a single effort, without developing it incrementally and testing it in phases. You might even get away with it, this time and at this somewhat limited scale, and develop a program that works. But in general, one of the key things that makes some people better at programming than others is the ability to see a design path through simple programs, to more comprehensive programs, to final programs, that keeps the complexity under control at all times. That is one of the skills this is subject is intended to teach you.

The boring stuff...

This project is worth 15% of your final mark. A rubric explaining the marking expectations is provided on the FAQ page. You need to submit your program for assessment; detailed instructions on how to do that will be posted on the FAQ page once submissions are opened. Submission will *not* be done via the LMS; instead you will need to log in to a Unix server and submit your files to a software system known as `submit`. You can (and should) use `submit` **both early and often** – to get used to the way it works, and also to check that your program compiles correctly on our test system, which has some different characteristics to the lab machines. *Failure to follow this simple advice is highly likely to result in tears.* Only the last submission that you make before the deadline will be marked. Marks and a sample solution will be available on the LMS before Tuesday 2 October.

Academic Honesty: You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** "lend" your "Uni backup" memory stick to others for any reason at all; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "**no**" when they ask for a copy of, or to see, your program, pointing out that your "**no**", and their acceptance of that decision, is the only thing that will preserve your friendship. *A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode. Students whose programs are so identified will either lose marks through the marking rubric, or will be referred to the Student Center for possible disciplinary action without further warning. This message **is** the warning.* See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to online forums or marketplaces, whether or not there is payment involved, is also Academic Misconduct. In the past students have had their enrollment terminated for such behavior. *The FAQ page contains wording for an Authorship Declaration that you must include as a comment at the top of your submitted program. Marks will be deducted if you do not do so.*

Deadline: Programs not submitted by **10:00am on Monday 17 September** will incur penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email `ammoffat@unimelb.edu.au` as soon as possible after those circumstances arise. If you attend a GP or other health care service as a result of illness, be sure to take a Health Professional Report (HPR) form with you (get it from the Special Consideration section of the Student Portal), you will need this form to be filled out if your illness develops in to something that later requires a Special Consideration application to be lodged. You should scan the HPR form and send it with any non-Special Consideration assignment extension requests.

And remember, algorithms are fun!