

Programming contest: Detecting repetitive redundancies

Due Apr 17, 11:59pm

Problem statement. Design and implement an algorithm to solve the following problem:

Given a string S , find the longest substring of S that appears two or more times in S .

More precisely, you will be given as input an n -byte string $S[1, \dots, n]$. You must find indices i, j, k so that $S[i, \dots, i+k] = S[j, \dots, j+k]$, so that $k \geq 0$ is as large as possible, and $i \neq j$; you should output $S[i, \dots, i+k]$.

If there is a tie for the longest repeated substring, you should output just one such substring (any one is fine). Note that the substrings are allowed to overlap. For instance, if the input is the string `MISSISSIPPI`, the output should be `ISSI`.

Collaboration forbidden. You must work on your own for this assignment. You may not discuss the problem with anyone else. As always, what you turn in must be your own work. You may not look on the web or in other sources for algorithms to solve this problem.

Submissions. You may write your program in any language you like. Submission instructions will appear on the CS170 web site when they are ready. A reference implementation can be downloaded from the CS170 web site.

All submissions will be made using the `test_and_submit` ruby script. Your submission must include at least **THREE** files, `INFO`, `README`, and a `Makefile`, as follows:

- The `INFO` should contain personal information, including your username, your full name, section number, and the files you will be uploading.

```
username: jschmoe
fullname: Joe Schmoe
section: 104
files:
- INFO
- README
- Makefile
- sourcefile1
- sourcefile2
```

If you have multiple accounts, use the account that you used to register with the online grading database. Make sure to write your section number (101–108), not the name of your TA or the time of

your section. This file will be read by `test_and_submit` as a YAML file and used for submitting and grading, so it MUST have the above format. `test_and_submit` will throw an error if the file format is off or fields are missing.

- The `README` file will be read by a grader and should contain the following. First, state the asymptotic running time of your algorithm (e.g., $O(n^3 \lg n)$ worst-case time, or $O(n^{3.5})$ expected running time on uniformly distributed strings, or whatever). Second, you should give high-level pseudo-code for your algorithm. Finally, you should give a brief English description of the main idea(s) behind your algorithm (this is just to help us understand your algorithm; it is sometimes easier to read these explanations than to read the pseudo-code). Your `README` file should be in plain ASCII text.
- Your submission must also include a `Makefile` file, with two targets defined: `compile` and `run`. Our grading scripts will execute the command `'make compile'` to compile your program. Then, our grading scripts will execute some command like `'make run < input > output'` to run your program. Your submission must be set up so that this will work on the instructional machines. Make sure that `'make run'` does not produce any extraneous output on `stdout`; otherwise, your program will fail our autograding tests. See the reference code for an example.

Your program must respect the following. First, it must accept a sequence of bytes on `stdin`, which should be interpreted as the string S (including all newlines, `'\0'` characters, and the like; the string is terminated only by end-of-file). Your program should output the selected substring on `stdout`. Make sure that your program does not output any extra newlines, debugging information, or other extraneous information on `stdout`; otherwise, your program will fail our autograding tests and you will lose points on the assignment. (You may send debugging information to `stderr` if you like; whatever you write to `stderr` will be ignored.)

Finally, your submission should include all the source code needed to make your program. You should *not* submit any executables, object files, or other compiler output; instead, `'make compile'` should automatically produce them from the source you submitted.

Once you have written the code, entered all information into `INFO` and `README`, and organized a correct `Makefile`, you will do the following to test and submit your code. Log into `cory.eecs.berkeley.edu` (the submission process will only work on `cory.eecs.berkeley.edu`, and your files must be stored in a single directory on that machine). Use `cd` to change into the directory containing all your relevant files. Then, run the command

```
/home/ff/cs170/bin/test_and_upload
```

First you will be asked to verify everything in `INFO`. Then this program will create a temp directory called `temp_test_dir/` and will copy the relevant files there. It will then test `make compile` and `make run < input > output` and, if all is successful, it will ask if you would like to submit your code. If yes, then it will copy the code to a private grading directory. Multiple submissions from the same username are OK, but we will only use the last for grading.

Grading. I plan to grade your solution as follows: 35 points for a correct implementation that passes all our test cases; 20 points for correctly stating the running time of your algorithm; and 45 points for creativity and clarity in the algorithm description found in your `README` file. (Note that the speed of your implementation is primarily for fun, not the primary factor in determining your grade.)

In addition to grading your solution, we will also run a speed trial across all the programs. The winner(s) of the programming contest will be those program(s) that run the fastest on our test cases. I will award some

prize (to be determined) to the winner(s). If your program takes more than a minute of CPU time, it will be terminated. You should be able to process 1000-byte strings in well under a minute. Can you handle a million-byte string in under a minute?

Exhortations and advice. I encourage you to focus your efforts on algorithm design and correctness. While it is true that which language you pick to write your code in will have some effect on its running time, it would be a mistake to pick assembly language and spend all your effort trying to hand-tune the instruction scheduling. You are much more likely to be successful by focusing mainly on improving the algorithm. Therefore, you are welcomed and encouraged to program in whatever language you feel most comfortable in.

Don't forget to place a high priority on correctness. Anyone can write a program that gives incorrect answers very quickly, but that's unlikely to be very useful. I encourage you to use all available methods to verify your implementation, such as writing a number of test cases and checking them against the reference code.

Check the CS170 web site and newsgroup before submitting to see if there have been any updates on this assignment.