ECS 122B: Programming Assignment #3

Instructor: Aaron Kaloti

Summer Session #2 2020

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: To part #1, added another example and clarified that `next_permutation()` from the `<algorithm>` header is prohibited.
- v.3: Emphasized that you must use the Z-algorithm for part #2.
- v.4: Added part #3.
- v.5: Fixed the example added to part #1 in v.2 so that it obeys the constraints set at the beginning of the problem.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Thursday, September 10. Gradescope will say 12:30 AM on Friday, September 11, due to the "grace period" (as described in the syllabus). *Rely on the grace period for extra time at your own risk.*

3 Grading Breakdown

Here is an approximate breakdown of the weight of this assignment. There will be no manual review of your code, but you will submit a small "report", as mentioned later. The plan is for this assignment to be worth 17% of your final grade, so I will make the assignment out of 170 points.

- Part #1: 30 points.
- Part #2: 60 points.
- Part #3: 80 points.

4 Gradescope Autograder Details

Once the autograder is released, I will update this document to include important details here. Note that you should be able to verify the correctness of your functions without depending on the autograder.

The autograder will use `g++` on a Linux command line to compile your code. On your end, you can likely get away with not using `g++` or a Linux command line. However, just make sure that whatever you use is pretty good about telling you compiler error messages *and* warnings. You should make sure to avoid sources of undefined behavior, such as uninitialized variables.

Below is a list of files that you will submit to Gradescope.

- `prog3.cpp`
  - This file should not contain a `main()` function when you submit it to Gradescope.
- `prog3.hpp`
- `report.md`

4.1 Autograder's Compilation Commands

The autograder will attempt to compile your code with the following commands in an Ubuntu 18.04 environment. (The `autograder_prog3.cpp` is not a file that you provide. It is a file that contains a `main()` implementation used by the autograder to test your code.)

- `g++ -std=c++11 -c prog3.cpp -o prog3.o`
- `g++ -std=c++11 autograder_prog3.cpp prog3.o -o autograder_prog3`

Make sure to avoid sources of undefined behavior in C++, e.g. uninitialized variables. Undefined behavior can lead to situations where your compiled code works on your machine but not on the autograder, and that would be a highly inconvenient thing for you to have to fix.

4.2 Changing Gradescope Active Submission

Once the deadline occurs, whatever submission is your **active submission** will be the one that dictates your final score. By default, your active submission will be your latest submission. However, you can change which submission is your active submission, which I talk about a small video that you can find on Canvas here. This video is from my ECS 32A course last summer session, so the autograder referenced therein is for a different assignment. Note that due to the hidden test cases, your score will show as a dash for all of the submissions, but you can still identify how many of the visible test cases you got correct for each submission.

5 Tasks

On Canvas, you are provided an initial version of `prog3.hpp`. You must create `prog3.cpp`.

You should be using C++11. If you are compiling on a Linux command line with `g++`, then make sure to use the `-std=c++11` flag.

5.1 Small "Report"

You must submit a small report containing certain details about part #3, as specified in blue in the directions for that part. This report is not worth points (at least, not directly). Rather, it is to help us identify certain parts of your code that show that you obeyed certain constraints of the problems, where applicable. If you do not submit this report, we will still look to see if you obeyed said constraints, but we will decrease the score that the autograder gave you. Although the report is not meant to be a true, detailed, exhaustive report, you should still make sure that the report's content can be understood; if it cannot be, we will ignore its content and penalize you as if you did not submit the report.

The name of the report should be `report.md`, meaning that it should be a Markdown file and not some Word document or MP3 file renamed with the wrong file extension.

Either in comments in your code or in your report, you should cite any resources used in the creation of your code. Potentially, C++ code for all of the problems in this assignment can be looked up, due to how common the algorithms are; **do not look up code for these algorithms and copy it. If you copy code from a source and make small tweaks, you may still be given a zero even if you cite that source, because you did not demonstrate any significant effort on your end. The safest bet is to try to implement these algorithms based on the steps of these algorithms.**

5.2 Part #1: Next Lexicographic Permutation

Function: `nextPermutation()`.

In `prog3.cpp`, implement the function `nextPermutation()`. This function is declared in `prog3.hpp`. It assumes that the given vector `vals` only contains integers in the range $[0, n)$, where n is `vals.size()`, and that `vals` contains no duplicates. The function should modify `vals` so that `vals` contains the next permutation according to lexicographic order. In other words, you should implement the algorithm that is on slide #11 of the brute force lecture slides. The function should return `true` if the next permutation could be generated; it should return `false` if `vals` already represents the last permutation according to lexicographic order. For example, calling `nextPermutation()` on a vector containing the sequence $3, 1, 2, 0$ should result in the vector being modified to contain the sequence $3, 2, 0, 1$ and `true` being returned. As another example, calling `nextPermutation()` on a vector containing the sequence $0, 5, 4, 6, 1, 2, 3$ should result in the vector being modified to contain the sequence $0, 5, 4, 6, 1, 3, 2$ and `true` being returned.

Needless to say, you are not allowed to use functions like `next_permutation()` from the `<algorithm>` header that would make this part of the assignment trivial and not even worth assigning.

5.3 Part #2: Z-Algorithm

Function: `findZBoxesAndMatches()`.

In `prog3.cpp`, implement the function `findZBoxesAndMatches()`. This function is declared in `prog3.hpp`. The return type is `std::tuple<std::vector<std::array<unsigned, 2>>, std::vector<unsigned>>`. This function takes as argument two strings, T and P, and the goal of the function, broadly speaking, is to find all occurrences of P in T. As is done in the updated lecture slides on string matching, the Z algorithm is performed on the string $P\$T$ (P concatenated with a dollar sign concatenated with T), resulting in the discovery of Z-boxes. Each of these Z-boxes has a start index and an end index. The first element of the `std::tuple` instance returned by your function will be a `vector` of all of the Z-boxes in $P\$T$ (in order from lowest start index to greatest). Each of these Z-boxes is represented as an instance of `std::array<unsigned, 2>`. The second element of the `std::tuple` instance will be a `vector` containing all of the indices of T at which a match with P begins.

This method must run in linear time, i.e. $\Theta(|T| + |P|)$ time, in the worst case, necessitating that you use the Z-algorithm. We will manually check for this.

As per the assumption that \$ is not part of the alphabet over which T and P are defined, you may assume that neither T nor P will contain a dollar sign.

You should be careful about the fact that the slides use one-based indexing while your implementation is expected to use zero-based indexing.

You may want to consider using `typedef` to avoid the long type names. (I did not try it because I am not a fan of `typedef`.) Certainly, you can use `auto` where appropriate as well.

Below are two Catch unit tests that demonstrate how your function should behave. The first unit test is based on example #1 on slide #9 of the string matching slides, and the second unit test is based on the example on slide #17.

```
TEST_CASE("findZBoxesAndMatches(): Case #1")
{
    std::string T = "banana";
    std::string P = "ana";
    auto retval = findZBoxesAndMatches(T, P);
    auto zBoxes = std::get<0>(retval);
    std::vector<std::array<unsigned, 2>> expectedZBoxes;
    expectedZBoxes.push_back(std::array<unsigned, 2>{2,2});
    expectedZBoxes.push_back(std::array<unsigned, 2>{5,7});
    expectedZBoxes.push_back(std::array<unsigned, 2>{7,9});
    expectedZBoxes.push_back(std::array<unsigned, 2>{9,9});
    REQUIRE(zBoxes == expectedZBoxes);
    auto matches = std::get<1>(retval);
    std::vector<unsigned> expectedMatches{1,3};
    REQUIRE(matches == expectedMatches);
}

TEST_CASE("findZBoxesAndMatches(): Case #2")
{
    std::string T = "dadabacabacd";
    std::string P = "abaca";
    auto retval = findZBoxesAndMatches(T, P);
    auto zBoxes = std::get<0>(retval);
    std::vector<std::array<unsigned, 2>> expectedZBoxes;
    expectedZBoxes.push_back(std::array<unsigned, 2>{2,2});
    expectedZBoxes.push_back(std::array<unsigned, 2>{4,4});
    expectedZBoxes.push_back(std::array<unsigned, 2>{7,7});
    expectedZBoxes.push_back(std::array<unsigned, 2>{9,13});
    expectedZBoxes.push_back(std::array<unsigned, 2>{11,11});
```

```
    expectedZBoxes.push_back(std::array<unsigned, 2>{13,16});
    expectedZBoxes.push_back(std::array<unsigned, 2>{15,15});
    REQUIRE(zBoxes == expectedZBoxes);
    auto matches = std::get<1>(retval);
    std::vector<unsigned> expectedMatches{3};
    REQUIRE(matches == expectedMatches);
}
```

5.4 Part #3: Decoding an HMM with Dynamic Programming

Function: `findMostProbablePath()`.

This function, given an instance of `struct HMM` and a `vector` of T `observations`, should use the Viterbi algorithm to return a tuple containing the following:

- A `vector` containing the most likely sequence of states corresponding to the given observations.
- The probability that this sequence of states was indeed the sequence that actually occurred.

The definition of `struct HMM` is provided in `prog3.hpp`. You may add methods to it, but you cannot change the order or number of members, since the autograder will use an initializer list to create an instance of `struct HMM`. One thing that I probably did not sufficiently stress in the lecture slides (since it is arguably more of an implementation detail) is that with an HMM, there is a distinction between the set of possible observations and the sequence of observations that we seek to decode. The latter can include multiple of the same observations and can include zero of certain observations; for example, we could have the sequence of observations *clean*, *clean*, *shop*, *clean*, *shop* in the weather example. Although you may feel that this suggests that the `states` and `possibleObs` fields of `struct HMM` should be instances of `std::unordered_set<std::string>`, giving each state and possible observation a unique index (with a `vector`) is necessary in implementing the Viterbi algorithm.

For the report: You should briefly mention which variable corresponds to the DP table. We will verify that you implemented the Viterbi algorithm instead of using some other approach.

Below is a Catch unit test that demonstrates how your function should behave. The unit test is based on the weather example that we did in the lecture slides.

```
TEST_CASE("findMostProbablePath(): Case #1")
{
    HMM hmm{{"rainy", "sunny"},
        {0.6, 0.4},
        {{0.7, 0.3}, {0.4, 0.6}},
        {"walk", "shop", "clean"},
        {{0.1, 0.4, 0.5}, {0.6, 0.3, 0.1}}};
    auto best = findMostProbablePath(hmm, {"clean", "shop", "walk"});
    auto bestPath = std::get<0>(best);
    std::vector<std::string> expectedPath{"rainy","rainy","sunny"};
    float bestPathProb = std::get<1>(best);
    float expectedProb = 0.01512;
    REQUIRE(bestPath == expectedPath);
    REQUIRE(bestPathProb == Approx(expectedProb));
}
```

UCDAVIS
COMPUTER SCIENCE