Homework 5

- Due: See homework page
- Directory name in your github repository for this homework (case sensitive): `hw5`

Skeleton Code

Some skeleton code has been provided for you in the `hw5` folder and has been pushed to the Github repository `resources`. If you already have this repository locally cloned, just perform a `git pull`. Otherwise you'll need to clone it.

Written Portion

Problem 1 - Counting (25%)

You must show work supporting your answer to receive credit.

Place your answers in a file **`counting.pdf`**.

1. Consider a 5-character string made up of lower-case, alpha characters, `a-z`. It is known that the second and third letters are `e` and `a`, respectively, (e.g. `-ea--`) and that the letters `s` and `t` appear somewhere in the remaining locations. How many possible such (unique) strings exist?

2. Alice, Bob, and Carlos enter an elevator on the first floor in a six-story building and start going up. Any one of them can exit on any floor from 2nd to 6th, except we know that Alice definitely won't exit the elevator on the 3rd floor. How many different events can happen? Two events are different if at least one of the three people ends up on a different floor, but if several people exit on the same floor, it does not matter in what order they do it.

3. Consider that one way to categorize Pokemon cards is into water type, fire type, and all other types.

- **3.1**. You have a total 21 Pokemons, out of which 7 are water type and 5 are fire type. How many combinations of 3 Pokemon cards (the order they are selected does not matter) exist such that none of the following occur: a selection has at least two fire types and a selection has at least two water types.
- **3.2**. Your friend sends you 10 random Pokemon cards. How many different combinations exist of their types: water, fire, and other?

4. In how many ways can 4 identical copiers be used to make 6 additional copies of a document (i.e. assuming we have an original for each copier). However, the most copies any single copier can make is **5**.

5. There is a Binary Search Tree with 13 nodes. Each node has a distinct value between 0 and 12. The root has value 8, and its left child has value 5. How many possible Binary Search Trees could this be? Tip: *Try to define how many ways there are to form a BST of 2 nodes. Then try to define how many ways there are to form a BST of 3 nodes (think about the possible insertion order based on rank: smallest, medium, largest)* ***in terms of 2 node trees*** *for certain cases. Continue to do this for 4 node trees (in terms of 3- and 2-node trees for various cases of insertion ordering based on rank) and 5 node trees.*

Problem 2 - Probability (25%)

You must show work supporting your answer to receive credit.

Place your answers in a file **`probability.pdf`**.

1 Suppose a college application asks students to choose, in order, 3 of the following words that best describe them: `leader`, `innovative`, `determined`, `logical`, `empathetic`, `dreamer`, `analytical`. A college admission officer is lazy and decides to admit any student that chooses `innovative` and `analytical` as two of the three as long as `innovative` is **before** `analytical`, **OR** if they choose `empathetic` and `innovative` (in any order).

- **1.1** Supposing that applicants are equally likely to choose any 3 of the 7 words, what is the probability that a student will be admitted by this admission officer?
- **1.2** What is the probability that in the first 10 applications the officer reads, exactly 4 students are admitted.

2 Consider 4 sequential flips of a fair coin.

- **2.1**. Let A be the event that 2 consecutive flips both yield **heads** and let B be the event that the first **OR** last flip yields **tails**. Prove or disprove that events A and B are independent.
- **2.2**. Let X be the random variable of how many pairs of consecutive flips (of the 4 total flips) both yield **heads**. What is the expected value of X?

3. You are playing a very limited variant of the famous game WORDLE (find out the rules of this game by clicking on the question mark button at this site

In your version of the game the secret answer can be only one of these words:

```
NASTY
HASTY
BOARD
HOARD
MATHS
```

The secret word is chosen randomly (uniformly) before you start playing. You can use only words from the list as guesses. The best starting word is the word with the smallest expected number of moves to finish the game under optimal play. You are choosing between NASTY and HASTY for a starting word. What is the expected number of moves for each of them if you play optimally?

4. Lewix and Zax 27.5: A man is accused of robbing a bank. Eyewitnesses testify that the robber was 6 feet tall and had red hair and green eyes; the suspect matches this descirption. Suppose that only 100 of the 100000 residents in the town are men who are 6 feet tall with red hair and green eyes, and assume that one of them robbed the bank.

- **4.1**. What is the probability that the suspect is innocent, given that he matches the description?
- **4.2**. What is the probability that the suspect would match the description, given that he is innocent?
- **4.3**. Suppose the police reviewed images of 1000 potential suspects when investigating the case, and everyone of the 100000 residents were equally likely to appear in this review. What is the probabilty that at least one of the 1000 would match the description?

5. You are an astronomer studying a planetary system of a red dwarf star. Over years of research you've discovered that for this type of star the following is true:

- 30% of stars have 2 planets
- 70% of stars have 3 planets
- No star of this type has less than 2 or more than 3 planets.

There are two general types of planets: gas giants and rocky Earth-like planets that are equally abundant, i.e. if any particular planet orbiting around a star has an equal chance of being giant or rocky.

Some exoplanets are very difficult to observe, specifically, modern telescopes have a 100% chance of detecting a giant Jupiter-like planet if it orbits around a star, but only 20% chance of detecting a rocky Earth-like planet.

After careful observation you found two giant exoplanets in the star system. How likely is it that this star has a rocky Earth-sized planet orbiting it?

Programming Portion

Problem 3 - Recursion and Combinations (25%)

Consider the popular game (at least at the time of this writing) of Wordle. A 5-letter English word is chosen at random (though we will extend this to support `n`-letter words) and the user has some number of chances to guess the word. As the user guesses words, letters in their guess that match the selected word in the **correct location** are highlighted in green, while letters (which may include duplicates) that are part of the word but **not in the correct location** are highlighted in yellow. So, if the selected word was `total` and the user guessed `treat`, the first `t` and `a` in `treat` would be highlighted in *green* to indicate they are in the correct location and the last `t` would be marked in *yellow* to indicate that there is another `t` in the select word but not at the correct location, and all other letters (such as `r`) would appear in gray to indicate they are not found in the word.

In this program, you will write a tool to aide players of a **Wordle-like** game (it does not have the exact semantics of Wordle, but is similar). Suppose the user would like to know what English-language words exist that meet certain criteria (so they can think of potential guesses). The user will provide you an input of how many letters the word is, what letters have already been guessed in their correct location (akin to the green letters), and a list of letters that they know MUST appear in the word (akin to the yellow letters, but without indicating where those letters CANNOT be). They will NOT provide the list of gray letters, meaning you will need to try all possible letters when trying to fill in a blank location in the word.

So your task will be to write a function (and any desired helper functions), to compute all possible *English-language words* that exist given a string containing the already-guessed letters that are in the **correct location** (we will refer to these as **fixed** letters since they may not change position and must stay in that location) and a string of the already-guessed letters that are **MUST** appear in the word somewhere (we will refer to these as **floating** letters since their position may be in any of the remaining locations). Again, we are **not actually writing code to play the game but code that can help players consider potential future guesses** given *some* of the knowledge they've learned from previous guesses. Your approach should be to generate all possible strings of lower-case, alpha characters that contain the fixed haracters (in their given location), floating characters (in any locations), and fill in any remaining letters with all possible options and then check if any of those strings are true *English-language* words. We will provide a `std::set<std::string>` of all possible English-language words (i.e. essentially a dictionary) that you may use for this task (which we read from a provided file: `dict-eng.txt`).

The signature of the function you are to write must be (and cannot be changed):

```
std::set<std::string> wordle(
    const std::string& in,
    const std::string& floating,
    const std::set<std::string>& dict);
```

Note: We realize that this may be an inefficient approach to finding all the words that match the given constraints. If we were not forcing you to use this approach, it may be easier to find the words that start with a given letter and iterate through them in the sorted order that the set provides checking to see if the word matches the given constraints. However, one can see that if the first letter or two are unknown, then this may require iterating through the whole dictionary, and so, in some cases, may be faster to generate all the possible strings, as we do here.

Input format

To indicate how many letters the selected word has **AND** the **correct, fixed location** letters already guessed, we will use a string of length `n` (for an `n`-letter word), using the `-` character to indicate a blank location and characters filling in the **correct, fixed locations**. So the input string `-i-` means we want to find all **3 letter** words that have an `i` as the middle character.

We will then pass in a second string of the **floating** (yellow) characters that contains the characters (in any order) that we know must be part of the selected word but whose location is unknown. So if you were given a first input string of `-i--` and second input string `dn`, you should output all words that have `i` as the 2nd letter and contain `d` and `n`. A sample output is below. Since the results are stored in a set, they can be printed in any order. We have provided a "driver"/test program (`wordle-driver.cpp`) that will take in these two strings from the command line, call your function, and then print out the results from the set of strings returned.

Program execution and command line arguments.

```
./wordle-driver -i-- dn
```

Printed results:

```
bind
dine
ding
dins
dint
find
hind
kind
mind
rind
wind
```

Use `wordle-driver` to do some sanity tests of your code before moving on to any of the tests from our grading suite. Note: To discourage any attempt to hard-code or game the system, the instructor may run additional tests after submission that were not provided, though they will be similar in format and content.

Requirements and Assumptions

- As always you may not change the signature of the primary function provided.
- You MAY define helper functions in `wordle.cpp`.
- You **MUST** use a recursive approach to find all combinations of letters to form the length-`n` word. **Failure to do so will lead to a 0 on this part of the assignment.**
    - Really you should only have 1 or 2 loops to help set the characters in any given location, and maybe 1 to 2 other loops to help with various constraint checks, etc. But to ensure you do not somehow brute-force the solutions, you may use at most 5 loops in your entire implementation in `wordle.cpp`.
- You may NOT use any functions from the `algorithm` library (nor should you really need to).
- You do NOT need to implement any kind of *backtracking* approach where you attempt to determine if a string can possibly be an valid English-language word as you are forming it. Instead, just generate all possible strings and check if each word is in the dictionary once it has the full `n` letters.

Hints and Approach

Recall that when generating all combinations you use recursion to build up a combination 1 "place" at a time, with each recursion responsible for 1 "place/location" of the combination. For that place, each recursion should try all the viable "options", recursing after each one, and, upon return, undo and try the next option if a solution has not been found (or if multiple solutions are desired). Think carefully about what "options" should be tried at each location. Can any letter be used at any open place? What limitaiton do the **floating** letters provide?

By generating all combinations of `n`-length words *that meet the restrictions given by the **fixed** and **floating** characters*, it becomes simple to check whether the combination is a valid *English-language* word once the combination has its full `n` letters.

Problem 4 - Recursion and Backtracking and Scheduling (25%)

Suppose a company needs to schedule `d` (which stands for `needed`) out of `k` (possible) workers (e.g. nurses at a hospital) per day given their availability over an `n` day period. The company requires exactly `d` workers (nurses) per day but does not allow any individual to work more than `m` (`maxShifts`) shifts over the `n` day period. Given `d`, `m`, and the `k` workers' availability for each of the `n` days, write a backtracking search function to schedule the nurses, such that exactly `d` work on each of the `n` days and no one works more than `m` shifts.

The prototype for the function you will write is given below, along with some `typedef`s for the input and output data structures.

```
// type for the ID of a worker
typedef unsigned int Worker_T;
// n-by-k Matrix of each of the k workers' availability over an n-day period
typedef std::vector<std::vector<bool>> AvailabilityMatrix;
// n-by-d matrix with the d worker IDs who are scheduled
// to work on each of the n days
typedef std::vector<std::vector<Worker_T> > DailySchedule;

/**
 * @brief Produces a work schedule given worker availability,
 *        the number of needed workers per day, and the maximum
 *        shifts any single worker is allowed. Returns true if
 *        and the valid schedule if a solution exists, and false
 *        otherwise.
 *
 * @param [in]  avail n x k vector of vectors (i.e. matrix) of the availability
 *                    of the k workers for each of the n days
 * @param [in]  dailyNeed Number of workers needed per day (aka d)
 * @param [in]  maxShifts Maximum shifts any worker is allowed over
 *                        the n day period (aka m)
 * @param [out] sched n x d vector of vectors indicating the d workers
 *                    who are scheduled to work on each of the n days
 * @return true if a solution exists; sched contains the solution
 * @return false if no solution exists; sched is undefined (can be anything)
 */
bool schedule(
    const AvailabilityMatrix& avail,
    const size_t dailyNeed,
    const size_t maxShifts,
    DailySchedule& sched
);
```

Explanation

The `avail` matrix is an `n` row by `k` column matrix of booleans. Each row represents one day of the schedule and each column is one worker's availability to work on each day.

```
           W  W  W  W
           o  o  o  o
           r  r  r  r
           k  k  k  k
           e  e  e  e
           r  r  r  r
           0  1  2  3

           |  |  |  |
           |  |  |  |
           V  V  V  V

Day 0 -->  1  1  1  1
Day 1 -->  1  0  1  0
Day 2 -->  1  1  0  1
Day 3 -->  1  0  0  1
```

So we see in the `avail` matrix above that every worker is available to work on day 0 while only worker 0 and 2 are available on day 1, and so on.

You should produce a schedule solution that is an `n` by `d` matrix, where each row represents a day in the schedule and stores the `d` **IDs of the workers who have been scheduled to work that day** (each entry must thus be a value in the range `0` to `k-1`). So given the availability matrix above and inputs of `m=2` and `d=2`, a valid schedule results would be:

```
1 2
0 2
1 3
0 3
```

The above indicates that on day 0 (top row), worker 1 and 2 will be scheduled. Then on day 1 (next row), worker 0 and 2 will be scheduled and so on. You can verify with the avaialbility matrix that all workers are available on their scheduled days and no worker works more than `m=2` shifts.

Testing

We have provided a “driver”/test program (`schedwork-driver.cpp`) where you can alter an availability matrix and values for `d` (`dailyNeed`) and `m` (`maxShifts`) and then call your algorithm and print the results.

Use `schedwork-driver` to do some sanity tests of your code before moving on to any of the tests from our grading suite. Note: To discourage any attempt to hard-code or game the system, the instructor may run additional tests after submission that were not provided, though they will be similar in format and content.

Requirements and Assumptions

- As always you may not change the signature of the primary function provided.
- You **MAY** define helper functions in `schedworker.cpp`
- You **MUST** use a recursive approach that follows the general backtracking structure presentedin class. **Failure to use such a recursive approach will lead to a 0 on this part of the assignment.**
- You MAY use functions from the `algorithm` library such as `std::find`, if you desire.
- The order in which you list the worker IDs in each row of the final schedule doesn’t matter (i.e. if Worker 1, 2, 3 is scheduled to work on a given day, then 3, 2, 1 is also acceptable).

Hints and Approach

Recall that a backtracking search algorithm is a recursive algorithm that is similar to generating all combinations, but skipping the recursion and moving on to another option if the current option violates any of the constraints. It is likely easiest to recurse over each place in the schedule (i.e. the 2D `sched` matrix). Each recursive call would be responsible for filling in one of the `n*d` schedule openings, ensuring the constraints of availability and the maximum number of shifts allowed for each worker is met. If you have already done a lab regarding backtracking search, it would likely be beneficial to look it over.

Checkpoint

For checkpoint credit, submit your working code for the **Wordle** problem. Ensure you add/commit/push your `hw-username` repo with a `hw5` subfolder that contains:

- `wordle.h`, `wordle.cpp` (it’s fine to include your other **source** files like `wordle-driver.cpp`, `Makefile`, and `sched` files)
- **THEN** you must submit your SHA on our Submit page linked from the Homework Page.

We will use `hw5_tests/wordle-tests/wordle-tests` for the checkpoint. They must compile, run, and pass all tests with no `valgrind` or other memory errors. Failure to pass even one test or having 1 valgrind error will result in 0 credit for the checkpoint. It is fine to push other source files or input test files if you like, though we will not grade them.

Submission Files

Ensure you add/commit/push all your source code files, `Makefile`, and written problem files. Do **NOT** commit/push any test suite folder/files that we provide from any folder other than the `resources/hw5` repo. Then submit your SHA on our submission site.

Please note we may run additional instructor tests for the coding problems that will affect your grade. They will be similar in style to the provided test suite, but we do so to ensure you do not hard-code solutions to specific tests.

WAIT You aren't done yet. Complete the last section below to ensure you've committed all your code.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your hw5 directory to your `hw-username` repository. Now double-check what you've committed, by following the directions below (failure to do so may result in point deductions):

1. In your terminal, `cd` to the folder that has your `resources` and `hw-username`
2. Create a `verify-hw5` directory: `$ mkdir verify-hw5`
3. Go into that directory: `$ cd verify-hw5`
4. Clone your hw_username repo: `$ git clone git@github.com:usc-csci104-spring2022/hw-username.git`
5. Go into your hw5 folder `$ cd hw-username/hw5`
6. Switch over to a docker shell, navigate to the same `verify-hw5/hw-username/hw5` folder.
7. Recompile and rerun your programs and tests to ensure that what you submitted works. You may need to copy over a test-suite folder from the `resources` repo, if one was provided.