Algorithms and Data Structures (ECE 345)

Issued on: Jan 31st, 2022
Due on: Feb 17th, 2022

ASSIGNMENT 2

1. Assignment must be submitted by 5:00 PM EST on the due date through the Quercus submission system as a single PDF file.
2. Assignment must be completed **individually except the programming exercise** for which you can work in group of up to **two** students. Report for the programming exercise must be submitted separately at the same time. Only one report is needed for each group.
3. All pages must be numbered and no more than a single answer for any question.
4. Use LaTeX or Microsoft Word for your assignment writeup. You can find a LaTeX and a Word template for writing your assignment on Quercus.
5. Any problem encountered with submission must be reported to the head TA as soon as possible.
6. Unless otherwise stated, you need to justify the correctness and complexity of algorithms you designed for the problems.
7. Partial marks will be given if your algorithm requires more time or space complexity than specified.

EXERCISE 1 Algorithm Analysis, 15 points

Consider the **sort** algorithm below, which takes as input an unsorted array A of n integers with no duplicates.

(a) Determine the runtime of **sort**$(A, 0, n-1)$.

(b) Prove the algorithm is correct by induction. For simplicity, you may assume n is a multiple of 3 in your inductive step. Alternatively, prove the algorithm is incorrect by providing some input that breaks the algorithm.

Algorithm 1 **sort**(A, i, j)

```
if A[i] > A[j] then                 // Swap A[i] and A[j]
    tmp ← A[j]
    A[j] ← A[i]
    A[i] ← tmp
end if
if j − i ≥ 2 then
    t ← ⌊(j − i + 1)/3⌋
    sort(A, i, j − t)               // Sort the first 2/3
    sort(A, i + t, j)               // Sort the last 2/3
    sort(A, i, j − t)               // Sort the first 2/3 again
end if
```

EXERCISE 2 Linear Time Sorting, 10 points

You are given an unsorted array A of n elements, each of which is either of the form i^2 or $-i^2$, where i is an integer in the range $[0,n]$. Devise an algorithm to sort A in $O(n)$ time using only $O(n)$ memory. Partial credit will be awarded if your algorithm requires more time or memory than what is specified above.

EXERCISE 3 AVL Trees, 15 points

Let T_1 and T_2 be two AVL trees of size n_1 and n_2 respectively.

(a) Devise an $O(n_1+n_2)$ algorithm to merge T_1 and T_2 such that the resulting tree is still an AVL tree.

(b) Suppose each node in T_1 is strictly less than each node in T_2. Devise an $O\Big(\log\big(\max(n_1,n_2)\big)\Big)$ algorithm to join T_1 and T_2 such that the resulting tree is still an AVL tree.

EXERCISE 4 Hashing, 10 points

Demonstrate the insertion of keys 24, 19, 51, 30, 10, 32, 41, 8, 63 into a doubly-hashed table where collisions are resolved with open addressing. Let the table have 7 slots, let the *primary* hash function be $h_p(key) = key \mod 7$ and let the *secondary* hash function be $h_s(key) = (3 \cdot key) \mod 4$. For any keys that can not be placed, list the key and give a reason.

EXERCISE 5 Hashing in Depth, 20 points

Suppose we have a hash table with m slots, and we have n values to hash into the hash table. Suppose $m \gg n$ and the hashing function we use satisfies the *simple uniform hashing* assumption.

(a) Give a lower bound for the probability that *a specific slot* is empty after hashing n values. Your answer should be given in the form of $c_0 + c_1\frac{n}{m}$, where c_0 and c_1 are constants. (Hint: You may want to use Binomial expansion).

(b) Give a lower bound for the probability that a specific slot stores only one value after hashing n values. Your answer should be given in the form of $\frac{f_1(n)}{m} + \frac{f_2(n)}{m^2}$, where f_1 and f_2 are polynomial functions of n.

(c) Give a lower bound for the probability that *all slots* store at most one value in each after hashing n values. Given $\varepsilon > 0$, determine m in the form of $n^{g(\varepsilon)}$ such that the probability for no chaining is at least $1 - \frac{1}{n^\varepsilon}$. (Hint: Use part (a) and (b)).

EXERCISE 6 Programming Exercise, 30 points

(In)secure Password Storage/Checking: In this programming assignment, you will experiment with simple and (in)secure password storage/checking, and you will measure some performance statistics.

1. **Outline:**

   - Assume that you will be given a file containing a list of existing passwords. The name of the file will be `passwords.txt`, and the format will be one password per line without comma separators. Assume the file will reside in the same path as your executables. A sample file will be provided to you on Blackboard. Your program will need to parse the file and hash the passwords into a hash table. You may assume that your program will be tested on files with $100 - 10000$ passwords.

- Each password in `passwords.txt` will be alphanumeric **without** any complex characters such as `!, ?, #, &,` etc. It will only consist of a mix of lower and/or upper case English alphabet letters with digits $\in \{0,\ldots,9\}$. Each password will be 6 to 12 characters long. For example, `ece345LoVe` is a valid password, whereas `HATEece345HATE` is invalid due to length violation (and probably other reasons...!), and `Pa33word!!??` is invalid due to containing complex characters.
- You are free to decide how to implement your hashing. Use only methods described in lecture. You are **not** allowed to use cryptographic libraries such as `OpenSSL`. Security is not our concern in this exercise.
- Your executable should be named as `checkpass`, and it's behavior should be as follows:
  - It should be callable from command line as `checkpass passwords.txt password`, where `passwords.txt` is a file containing passwords and `password` is user provided. For example: `checkpass passwords.txt 23CdB9a13`
    For C/C++ please include a `Makefile` to build your sources into the executable. For any other compiled language, please follow the same format as for C/C++.
    For Java please create a `Makefile` that compiles your sources and a `checkpass.sh` script that runs your code for the given input, with the correct classpath.
    For Python please name your script `checkpass.py`, have `#!/usr/bin/env pythonX` (where `X` is the version of python you are using) as the first line of your file. Additionally please do `chmod +x checkpass.py` so that you can run your file as `./checkpass.py passwords.txt password`. For any other scripting language (supported by the UG machines), please follow the same format as for python, except name your file as per that language (ie for perl use `checkpass.pl`)
  - Your program then should print in standard output `VALID` if and only if the password is valid according to the constraints described above AND the password does not already exist in `passwords.txt` AND the reverse of the password does not exist in `passwords.txt`. Think how to make the check for reverse passwords efficient. In any other case your program will print `INVALID`.
  - If the password entered is valid then your program should hash it and store the password (not its hash) into `passwords.txt`, and then it should terminate. If the password is not valid then the program should simply terminate after outputting `INVALID`.

2. **Deliverables:**

- Your full source code. Any code that does not build or does not run will receive a mark of 0. All code will be marked on the UG machines. This should be handed in electronically.
- A written report with readable graphs (with proper lables, grid ticks, and legends as needed). An electronic submission of the report is not necessary, a physical submission is **required**. The report must address the following points:
  - Implementation details regarding your hashing approach. What is the size of your hash table and which hash function are you using? Are you applying chaining or open addressing? If you are using open addressing state what probing sequence you are implementing.
  - A brief justification of your implementation decisions above.
  - For a fixed number of entries $n = 1000$, vary the size of your table to get various load factors. For each load factor, run your code on **five different** `password.txt` files which you will create. Each file should include 1000 password entries. Plot the **average number of collisions vs. load factor**. You will need several values of load factor to see a trend on your plot. Briefly explain what you observe in the plot and why.
    *NOTE: To set up this experiment, you can easily create another version of your program where there is no password input from the command line. You only need to parse the `passwords.txt` files that you will create and insert the entries in your hashtable. However, **DO NOT** submit this version of the code.*
- A version of submission.toml file that contains your group information and details about your source files. You can reuse the one from previous assignments. This should be handed in electronically.