THE UNIVERSITY OF MELBOURNE
SCHOOL OF COMPUTING AND INFORMATION SYSTEMS
COMP90038 ALGORITHMS AND COMPLEXITY

Assignment 1, Semester 2 2021

Released: Friday August 20 2020. Deadline: Sunday September 5 2020 23:59

This assignment is marked out of 30 and is worth 15% of your grade for COMP90038.

Objectives

To improve your understanding of the time complexity of algorithms and recurrence relations. To develop problem-solving and design skills. To improve written communication skills; in particular the ability to present algorithms clearly, precisely and unambiguously.

Scenario

In this assignment we will consider the following scenario. Suppose you are writing algorithms that will form part of the implementation for a cryptocurrency client for the fictitious *MugsCoin* cryptocurrency system.

Each user who participates in the MugsCoin system has a MugsCoin *account* that stores their funds. The MugsCoin system maintains an *accounts database* that stores, for each user account in the system, the *balance* of that account, i.e. the amount of MugsCoin that it holds.

Each user who participates in the MugsCoin system runs a program on their computer called the *MugsCoin client*. This client allows the user to perform transactions, including moving MugsCoins to and from their account, and transferring MugsCoins between their account and the accounts of other users. As each transaction is performed the MugsCoin accounts database is updated.

Updating the accounts database requires solving a *proof of work* problem. This problem involves a so-called *hash* function f, and requires finding an input x such that $f(x) = y$ for a value y that is a function of the information to be added to the database (see below).

In this assignment you will implement and analyse some alternative algorithms and designs for parts of this system.

Proof-of-Work

As mentioned to modify the accounts database clients must solve a so-called *proof-of-work* problem. MugsCoin's proof-of-work uses a so-called *hash function* f as part of its proof-of-work scheme. To add a new record r to the database, the client computes a *target* value $y = g(r \cdot h)$ by applying a separate deterministic function to the record r and the database history h. The client must then find an input value x for the hash function such that $f(x) = y$. The inputs x to the hash function f are just bitstrings (arrays of bits) of arbitrary length. Its output is a bitstring of some fixed length (the actual output length is not important).

Suppose we are trying to write an algorithm to solve the MugsCoin proof-of-work problem for inputs *up to* length n-bits. The desired algorithm takes as its input the fixed output value y and searches through all inputs up to length n-bits, trying to find an input x such that $f(x) = y$. It returns a boolean (true/false) indicating whether a solution can be found or not.

We might write this algorithm as follows. It makes use of an auxiliary function SEARCHPOWFIXED(x, k, y) that takes a bitstring x whose length is at least k and is assumed to be filled with zeros, plus the target value y. The job of SearchPOWFixed is to see whether there exists any bitstring x' of length k such that $f(x') = y$. If this is the case it returns true and otherwise returns false.

```
function SearchPOW($n, y$)
    //$n$ is the (positive) maximum bitstring length to search up to
    //$y$ is the fixed output value
    $x \leftarrow$ allocate a bitstring of length $n$
    //note: $x$ can be considered a bitstring of length $k$ for all $0 < k \leq n$
    $k \leftarrow 1$
    while $k \leq n$ do
        fill $x$ with all zeros
        if SearchPOWFixed(x,k,y) then
            return true
        $k \leftarrow k + 1$
    return false
```

1. [2 marks] Brogrammer Barry has written the following implementation of SearchPOWFixed.

```
function SearchPOWFixed'($x[\cdot], i, k, y$)
    //$x$ is a bitstring of length $k$
    //$i$ is an integer in the range [0,k] (i.e. 0 to $k$ inclusive)
    //$y$ is the fixed output value
    //returns true if a solution is found; false otherwise
    if $i = k$ then
        return $(f(x) = y)$
    for $j \leftarrow i$ to $k - 1$ do
        $x[j] \leftarrow 1$
        if SearchPOWFixed'($x, i + 1, k, y$) then return true
        else
            $x[j] \leftarrow 0$
            if SearchPOWFixed'($x, i + 1, k, y$) then return true
    return false
function SearchPOWFixed($x[\cdot], k, y$)
    return SearchPOWFixed'($x, 0, k, y$)
```

Here the basic operation is executing the hash function f. We can define the size of the input to the auxiliary function SearchPOWFixed'(x, i, k, y) as $k - i$. Write down a recurrence relation $H(n)$ for the number of calls to the hash function f that SearchPOWFixed' performs for an input of size n. Use telescoping (aka back-substitution) to derive a closed form for $H(n)$. From this state a precise formula for the number of basic operations (calls to f) that SearchPOWFixed(x, k, y) performs.

2. [3 marks] A brute-force implementation of SearchPOWFixed must, in the worst case, try 2^k inputs x to see if any hash to the target value y. Write a brute-force implementation of SearchPOWFixed. Your SearchPOWFixed(x, k, y) implementation should have worst-case complexity $\Theta(2^k)$, where the basic operation is executing the hash function f.

3. [2 marks] Calculate the worst-case complexity of the algorithm SearchPOW above, assuming it uses the brute-force implementation of SearchPOWFixed (rather than Barry's implementation). The complexity of SearchPOW(n, y) should be expressed in terms of its input n, i.e. n measures

the size of the input. The basic operation is executing the hash function f. Show your working. Your final answer should be expressed in Big-Theta notation and be as simple as possible, i.e. it should be of the form $\Theta(g(n))$ for some function $g(n)$ that is as simple as possible.

4. [7 marks] Unfortunately, the MugsCoin hash function f has the following *monotonicity* property. For two inputs x_1 and x_2 of the same length, if $x_1 < x_2$ then $f(x_1) \leq f(x_2)$. When we write $x_1 < x_2$ and $f(x_1) \leq f(x_2)$ we are comparing the (decimal) non-negative (unsigned) integer values represented by each bitstring.

   This means we can do much better than brute-force exhaustive search to solve the MugsCoin proof-of-work scheme.[1]

   Design and implement a more efficient version of SEARCHPOWFIXED. When your more efficient SEARCHPOWFIXED is used by SEARCHPOW to solve the MugsCoin proof-of-work for inputs up to length n-bits, SEARCHPOW's worst-case complexity should be in $O(n^2)$ (where the basic operation is calling the hash function f).

   State precisely in terms of k the number of basic operations (calls to the hash function f) that your SEARCHPOWFIXED performs. Justify why the resulting complexity of SEARCHPOW is in $O(n^2)$.

5. [2 marks] Analysing the complexity of these algorithms by counting the number of times the hash function f is called is not very precise. In particular, doing so assumes that the hash function always executes in constant time ($O(1)$) regardless of the size of its input. In reality, the running time of the hash function $f(x)$ will depend on the length k of its input x.

   Let us assume that f's running time is linear in the size of its input, i.e. executing f on an input of length k-bits requires k basic operations, how does this change your answer to Question 3 above (where you had to calculate the complexity of the brute-force SEARCHPOW function)? In particular, is SEARCHPOW's complexity now asymptotically different to your answer from Question 3 above? You *must* justify your answer. Use the following formula from lectures:

   Recall that $f(n)$ grows asymptotically faster than $g(n)$ when

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

   Hint: express the complexity again in Big-Theta notation $\Theta(g'(n))$. Then compare whether $g'(n)$ grows asymptotically faster than your answer from Question 3.

Accounts Database

Updates to the accounts database (i.e. transactions) are batched together in groups called *blocks*. Each transaction has a unique id associated with it. Each block represents 10 seconds of time: the transactions that arrive at the database within the first 10 seconds form the first block; those that arrive in the following 10 seconds form the second block, and so on. Each of these 10 second intervals is called a *block interval.*

Each client that submits transactions to the accounts database is of course aware of the transactions it is performing. It might also be aware of some of the transactions performed by other clients (e.g. those that transfer money to that client). In general, no client is aware of all of the transactions being performed in a particular block. Therefore, when a client submits its transactions to the accounts database, it also

[1]Being able to do better than exhaustive search means that the hypothetical MugsCoin proof-of-work scheme is not secure. But that is beyond the scope of this subject.

includes information about the order in which those transactions should be performed, and whether they depend on (i.e. should be performed after) other transactions (e.g. performed by other clients in the same block). Clients submit this information in the form of Transaction Dependency Graphs (DPGs). A DPG is a graph whose nodes are individual transaction ids. There is an edge (u, v) when transaction v cannot execute before transaction u.

The database takes the DPGs from each client received during a single block interval and combines them together to form a *Block Transaction Dependency Graph* (BDPG). The BDPG is a graph whose nodes are all of the individual transactions in a block. An edge (u, v) means that transaction v cannot happen before transaction u. Given m client DPGs $\langle V_1, E_1\rangle, \langle V_2, E_2\rangle \dots \langle V_m, E_m\rangle$, the BDPG $\langle V, E\rangle$ is formed by taking $V = \bigcup_{i=1}^{m} V_i$ and $E = \bigcup_{i=1}^{m} E_i$.

1. [6 marks] Suppose clients 1 and 2 each submit the DPG encoded in the following adjacency matrices within the same block interval. Node 0 in Client 1's DPG represents the same transaction as node 0 in Client 2's DPG. The same is true for the other nodes.

Client 1 DPG

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Client 2 DPG

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

   (a) Write down the adjacency matrix corresponding to the BDPG that is the combination of these two DPGs.

   (b) What is the complexity of the algorithm that performs this combining, assuming DPGs are represented for now as $|V| \times |V|$ adjacency matrices?

   (c) Explain why the transactions in this BDPG cannot be executed by the database while respecting all of the ordering constraints encoded in the graphs above.

2. [8 marks] Suppose you have to implement part of the accounts database that executes the transactions encoded in a BDPG. Design and implement an algorithm RunBlock that takes as its input a BDPG $\langle V, E\rangle$. Your algorithm should execute the transactions in that BDPG while ensuring that all of the ordering constraints are respected.

   State the complexity of your algorithm in terms of $|V|$ and $|E|$ using Big-Theta notation.

   - To execute an individual transaction $v \in V$, your algorithm should call RunTransaction(v)
   - You can assume that all the transactions in V can be executed while respecting the ordering constraints encoded in E.
   - Your algorithm should use a fixed amount of call-stack space, i.e. the number of recursive calls that it makes should not depend on unbounded quantities like $|V|$ and $|E|$ etc.
   - Your algorithm should have complexity that is a linear function of $|V|$ and $|E|$, i.e. its complexity should be $\Theta(c_1|V| + c_2|E|)$ for some constants c_1 and c_2.
   - You can assume the existence of standard abstract data types including lists, stacks, queues, etc. as introduced in lectures.
   - Use notation consistent with that used in lectures, both for standard data types and for e.g. iterating over the outgoing edges of a node $v \in V$.

- When calculating the complexity of your algorithm you can assume the graph is represented as an adjacency list, and so traversing the graph requires $\Theta(|V| + |E|)$ operations. However your algorithm does not (and *should not*) operate on the adjacency list representation; instead it should operate on the mathematical representation $\langle V, E \rangle$ as shown in lectures.

Submission and Evaluation

- You must submit a PDF document via the LMS. Note: handwritten, scanned images, are acceptable *only if* they are clearly legible. Write very neatly, and if you photograph your submission be sure to use an app that auto crops and rotates images, such as OfficeLens, to ensure the resulting submission is easy to mark. Convert images to PDF before submission. Do not submit Microsoft Word documents — if you use Word, create a PDF version for submission.

- Marks are primarily allocated for correctness, but elegance of algorithms and how clearly you communicate your thinking will also be taken into account. Where indicated, the complexity of algorithms also matters.

- We expect your work to be neat – parts of your submission that are difficult to read or decipher will be deemed incorrect. Make sure that you have enough time towards the end of the assignment to present your solutions carefully. Time you put in early will usually turn out to be more productive than a last-minute effort.

- You are reminded that your submission for this assignment is to be your own individual work. For many students, discussions with friends will form a natural part of the undertaking of the assignment work. However, it is still an individual task. You should not share your answers (even draft solutions) with other students. Do not post solutions (or even partial solutions) on social media or the discussion board. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student concerned.

  Please see `https://academicintegrity.unimelb.edu.au`

If you have any questions, you are welcome to post them on the LMS discussion board *so long as you do not reveal details about your own solutions.* You can also email Toby Murray (toby.murray@unimelb.edu.au). In your message, make sure you include COMP90038 in the subject line. In the body of your message, include a precise description of the problem.

Late Submission and Extension

Late submission will be possible, but a late submission penalty will apply of 3 marks (10% of the assignment) per day.

Extensions will only be awarded in extreme/emergency cases, assuming appropriate documentation is provided simply submitting a medical certificate on the due date will not result in an extension.