Homework 7

Discrete Structures
CS 173 [B] : Fall 2015

Released: Fri Apr 24
Due: Fri May 1, 5:00 PM

Submit on Moodle.

PART 1 (Machine-Graded Problems) on Moodle. [40 points]

PART 2 [60 points]

1. **Algorithm analysis** [20 points]

   Consider the following mystery function, which takes as input a list of real numbers and outputs a non-negative real number.

```
1: function Foo(A_1, A_2, ... A_n: real numbers)
2:     d := |A_1 − A_2|
3:     for i := 1 to n do
4:         for j := 1 to n do
5:             q := |A_i − A_j|
6:             if i ≠ j and q < d then
7:                 d := q
8:     return d
```

   (a) Give a brief English description of what the function foo computes.
   Solution: Given a list of numbers, Foo computes the smallest difference possible between two distinct elements of the list.

   (b) How many times are lines 5 and 6 executed, as a function of n?
   Solution: n^2

   (c) Express the running time of the algorithm expressed as $\Theta(f(n))$ for a simple function f? Briefly justify your answer.
   Solution: $\Theta(n^2)$.
   Here is a detailed analysis (you don't have to *write down* your analysis at this level of detail). Line 2 is executed once which takes constant amount of time. Line 3 is executed n times and each execution takes constant amount of time. Line 4 is executed n^2 times and each execution of line 4 takes constant time. Line 5 and 6 are executed n^2 time and each execution takes constant time. Line 7 is executed less than n^2 times and each execution takes constant time for execution. Line 8 is executed once which takes constant time.
   So, Running time is atmost c_1 (for line 2) + $c_2.n$ (Line 3) + $c_3.n^2$ (Line 4) + $c_4.n^2$ (Line 5) + $c_5.n^2$ (Line 6) + $c_6 n^2$ (Line 7) + c_8. Hence, running time is $O(n^2)$.
   Running time is atleast c_1 (for line 2) + $c_2.n$ (Line 3) + $c_3.n^2$ (Line 4) + $c_4.n^2$ (Line 5) + $c_5.n^2$ (Line 6) + $c_6.0$ (Line 7) + c_8 (If Line 7 is not executed at all). Hence, running time is at least $c \cdot n^2$ for some constant c (in this case, we say that the running time is $\Omega(n^2)$).
   From the above two statements, we can conclude that running time is $\Theta(n^2)$.

(d) This is a really bad algorithm for this task. Write pseudocode for a function with a better big-O running time.

[Hint: First sort the input.]

Solution:

```
1: function FOO-IMPROVED(A_1, A_2, ... A_n: real numbers)
2:     B_1, B_2, ..., B_n = sort(A_1, A_2, ... A_n)
3:     d := |B_1 − B_2|
4:     for i := 1 to n − 1 do
5:         q := |B_{i+1} − B_i|
6:         if i ≠ j and q < d then
7:             d := q
8:     return d
```

(e) Express the running time of your new algorithm from part d as a $\Theta(\cdot)$ expression. (You can use the fact that sorting can be done in $\Theta(n \log n)$ time. How much additional time do you take? What is the overall time?)

Solution: Running time is $\Theta(n \log n)$. Sorting (Line 2) takes $\Theta(n \log n)$ time. Line 3 takes constant time. Line 4 is executed n times and each execution takes constant time. Similar for Line 5 and 6. Line 7 is executed less than n times and each execution takes constant amount of time. Line 8 is executed once and it takes constant time. By doing the similar analysis as part (c), we get $\Theta(n \log n)$.

2. **Solving Search Problems using Decision Oracles** [20 points]

In this problem, you need to devise a simple algorithm for efficiently solving a sudoku puzzle, but with the help of a "decision" oracle. In a sudoku puzzle you are given an $n \times n$ grid with some cells already filled in with numbers in the range 1 to n. Your goal is to fill in the rest of the cells with numbers in the range 1 to n, so that the grid satisfies some rule (for our purposes, the specific rule doesn't matter), or to discover that this is not possible (which is, in real-life puzzles, promised not to be the case).

You can query the decision oracle with a sudoku puzzle and it will only answer yes/no to indicate if there exists a solution for the sudoku puzzle or not.

Devise a strategy to solve any sudoku puzzle by making $O(n^3)$ queries to the decision oracle.

[Hint: Note that the oracle works no matter how many cells are already filled in. You can query it with partially filled grids to check if you are on the right track. Your algorithm will not need to rely on the exact rules of sudoku.]

Note: This is an instance of efficiently solving a search problem ("find a solution") by making use of a sub-routine that solves only the decision problem ("does there exist a solution"). When this is possible we say that the search problem can be reduced *to the decision problem. If the reduction can be carried out efficiently, the search problem "is not much more complex than" the decision problem, or alternately, the decision problem is "almost as complex as" the search problem.*

Solution: Firstly, we can call the oracle with the puzzle as given, to find out if it has a solution or not. If there is a solution, we can find the solution using a simple strategy, as follows.

We will go through every empty cell in the puzzle one-by-one, and fill in a value there (thereby obtaining a new puzzle, with one less empty cell). We shall maintain the *invariant* that the new puzzles we get after filling in i cells has a solution. The invariant is indeed true for $i = 0$ (before we start filling in any cells). If we maintain the invariant after filling in all the empty cells, that means we have a valid solution.

So it only remains to show that we can maintain the invariant each time we fill in a cell. For this we shall use the oracle (again). We tentatively fill the cell with each possible value $k = 1, \ldots, n$ to create

n tentative puzzles, and query the oracle to find out which ones have valid solutions. For at least one value of k (but possibly for more), it must be the case that the tentative puzzle has a valid solution (why?). We pick such a k to be the value of this cell.

The following pseudo-code implements this strategy. The number of calls to the oracle is $O(n^3)$, as it makes at most n for each of the n^2 cells, plus an additional query at the beginning.

```
1:  function SUDOKU-SOLVER(A: n × n matrix of numbers in {0, ..., n}) ▷ 0 indicates a blank entry
2:      answer := Oracle(A)
3:      if answer = No then
4:          return "not possible"
5:      for i := 1 to n do
6:          for j := 1 to n do                                          ▷ Go over all cells
7:              if A_ij = 0 then filled := False
8:              else filled := True
9:              k := 1                          ▷ We'll check if A_ij can be set to k for k := 1 to n
10:             while (filled = False) do
11:                 A_ij := k
12:                 answer := Oracle(A)
13:                 if answer = Yes then filled := True
14:                 else
15:                     if k = n then return "Oracle is not truthful!"
16:                     k := k + 1
17:     return A
```

3. **Finite State Machines** [20 points]

(a) Consider a machine which has n bits of memory (initialized to all 0s). Its state is determined by the contents of its memory. This machine takes an input from the set $\Sigma = \{1, \ldots, n\}$: on input i, it toggles its i^{th} bit. (Toggling a bit changes its value from 0 to 1 or from 1 to 0.)

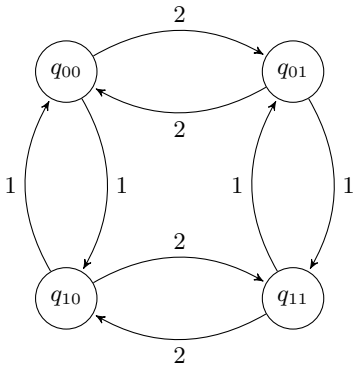
i. Write down the transition function for this machine for $n = 2$, as table, with columns "current state," "input" and "next state." (Note that each state is labeled with 2 bits.)

ii. What graph does the state-diagram for this machine (for a general value of n) resemble? You can treat pairs of directed edges between same two states, but pointing in opposite directions as a single undirected edge. Justify your answer.
[Hint: For this part, it may be helpful to draw a diagram for the case $n = 2$ from the previous part. You don't need to submit your diagram.]

Solution:

i. The state machine has 4 states which we denote as q_{ab}, where a, b are the two bits in the memory. That is, let $S = \{q_{00}, q_{01}, q_{10}, q_{11}\}$ be the state-space. The input alphabet is $\Sigma = \{1, 2\}$. The transition function $\delta : S \times \Sigma \to S$ is given by the following table:

q	x	$\delta(q, x)$
q_{00}	1	q_{10}
q_{10}	1	q_{00}
q_{01}	1	q_{11}
q_{11}	1	q_{01}
q_{00}	2	q_{01}
q_{10}	2	q_{11}
q_{01}	2	q_{00}
q_{11}	2	q_{10}



ii. The state-diagram resembles an n-dimensional hypercube, Q_n. For instance, the above state diagram resembles a square (Q_2).
This is because, there will be 2^n states (nodes), each labeled with a different n-bit string, and there is an edge from state x to state y labeled with input i iff y is obtained by toggling the i^{th} bit of x (in which case there is also an edge from y to x). That is, there is a pair of edges (in opposite directions) between the states labeled x and y if they differ in exactly one position. Treating this pair of edges as a single undirected edge, this matches the definition of Q_n.

(b) Construct a finite state acceptor that accepts exactly those strings which consist only of a's and b's such that the number of a's is even and the number of b's is odd. Represent this state machine by its transition function (in the form of a table). Indicate the start and final states.

[Hint: How many states must this machine have?]

Solution: The machine has 4 states of the form q_{ab} where a is the number of a's modulo 2, and b is the number of b's modulo 2 that have been seen so far. The start state is q_{00} and the final state is q_{01}. The transition function is given below.

q	x	$\delta(q,x)$
q_{00}	a	q_{10}
q_{10}	a	q_{00}
q_{01}	a	q_{11}
q_{11}	a	q_{01}
q_{00}	b	q_{01}
q_{10}	b	q_{11}
q_{01}	b	q_{00}
q_{11}	b	q_{10}