Harvard University Extension School
Computer Science E-121

Problem Set 8

Due November 18, 2016 11:59pm

Problem set by Walter Thornton

Collaboration Statement: I worked alone and only with course materials

When writing proofs, avoid the inclusion of redundant or irrelevant information. Sometimes these details can obscure the argument and we will take off half points when these issues arise.

PROBLEM 1 (2+2+2 points, suggested length of 1/3 page)

Let $f(n)$ and $g(n)$ be functions over: $\mathbb{N}^+ \to \mathbb{R}^+$. Note that $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. Prove or give a counterexample for the following conjectures.

(A) If $f(n) = \Theta(g(n))$ then $g(n) = O(f(n))$.
(B) $f(n) = O((f(n))^2)$.
(C) If $f(n) = \Theta(g(n))$ then $2^{f(n)} = \Theta(2^{g(n)})$.

Solution.

(A) True
By definition given in complexity.pdf slide 19

(B) False
If $f(n) = 1/n$ then $1/n \neq O((1/n)^2)$

(C) False
If $f(n) =$lg n and $g(n) = 2$lg n, then $2^{f(n)} = n$ and $2^{g(n)} = n^2$
Therefore $n \neq \Theta(n^2)$

PROBLEM 2 (6 points, suggested length of 1/2 page)

Show that every infinite regular language has a subset that is Turing-recognizable but undecidable.
Hint: Consider ordering the set of all Turing Machines.

Solution.
Consider the language $A_{TM} = \{\langle M \rangle \mid M$is TM that accepts w$\}$
The set of all TMs can be ordered, additionally, our infinite regular language R can be ordered lexicographically and is decideable.
We must show that some subset or R, R' is Turing-recognizable but undecidable.
Define R' as $R' = \{r_i \in R : \langle M \rangle \in A_{TM}\}$
R' is r.e. because $TMM_{R'}$ will recognize it

We can show that R' is not r by contradiction.

Assume that R' is decided by $TMM_{R'}$
Construct a decider TMM_{ATM} = "On input $\langle M \rangle$, output r_i.
Input r_i to $TMM_{R'}$
$TMM_{R'}$ will be forced to accept or reject, deciding A_{TM}, a known undecideable language.
This shows that R' is undecideable

Therefore R', a subset of our infinite regular language is undecideable but still Turing recognizable.

PROBLEM 3

(5+3+1 points, suggested length of 3/4 page, 1/4 page and 1/4 page for each of the subparts respectively)

Suppose a company with n total employees is represented by a tree T (which is just a special case of a graph–think of it as being directed for this question). Each node represents an employee, and an arrow from node a to b represents the relation "a is a supervisor of b". You are acting as interim director of fun for the company and have been given a "fun index" for everybody who works there.

You want to throw the company a fun party, but a party is no fun if an employee might meet his/her supervisor in the party. So, you have to design an invite list that includes the most fun people (that is, you want to maximize the sum of the fun indices of people at the party), but still avoids anyone having to meet his/her immediate supervisor.

You are busy with CS121 homework, and don't have enough time to go through all possible party lists to find the most fun one, so clearly something else has to be done.

(A) Design a polynomial-time algorithm to find the list of people that will make the party most fun. (If there is more than one possible list, just find one of them).

Your algorithm, GREATEST-PARTY-EVARR, should take as input $\langle T, (e_0, f_0), (e_1, f_1), \ldots (e_n, f_n) \rangle$ where f_i is the fun index of employee e_i (each also a node of the tree). Your algorithm should output a list of employees. You may assume that the CEO of the company (the root of the tree) will graciously accept either outcome of being invited or not invited to the party.

Hint: Recall the algorithm for checking to see whether a string can be generated by a grammar in Chomsky Normal Form and how we built up a solution using solutions to subproblems. In this case start with the smallest subproblems involving the lowliest employees (ones that have no subordinates) and build up your solution from there using larger and larger subtrees.

(B) Explain why your algorithm is correct.

(C) Explain why your algorithm runs in polynomial-time (no need to give a numeric bound on runtime though).

Solution.

(A)

GREATEST-PARTY-EVARR = "on input $\langle T, (e_0, f_0), (e_1, f_1), \ldots (e_n, f_n)\rangle$
1. Let F[] be an array that tracks the fun index
2. Let E[] be an array that tracks names of invited guests
3. For i = node n to 1
4. F[i]=MAX((f_n+SUM F[i.grandchild]), SUM F(i.child))
5. If (f_n+SUM F[i.grandchild])≥SUM F(i.child))
6. E[i]=E[i.grandchild]+e_n
7. Else E[i]=E[i.child]
8. Return E[i]"

(B)
The algorithm uses dynamic programming to break the problem into smaller subproblems that are stored, then reused. Rather than a complete traversal of every path, the algorithm visits each node starting at the bottom.
Because no supervisor/ subordinate pair can be in attendance, at each node the algorithm makes two calculations and compares the two.
For each node, staring with n, the fun index of the present node is added to the fun index of the node's grandchildren. This value is compared to the fun index of the node's children. Whichever value is greater is then assigned to that node. This is repeated for each node. As the computation moves up the tree, the algorithm computes the fun index value using the already computed values from the nodes below.
Additionally, whenever the fun index is stored, determined by the greater of the preceding paths, the name associated with that node is added to the employee array for that path, thus keeping a running tally of those invited along a particular path. When the algorithm reaches node 1, the CEO, the algorithm has cumulative fun indexes from which it chooses the greater and outputs the list of names of those invited.

(C) Each node is visited only once in the algorithm with a constant number of operations at each node. Therefore it is in O(n) and is a member of P.

PROBLEM 4 (3 points, suggested length of 1/4 page)

Given a DFA M with start state q_0 show that determining if every state is reachable from q_0 is in P.

Solution.

We are given a DFA which can be represented as a directed graph. A brute force search of each path would end up being an NP problem. Alternatively we can use Sipser's path algorithm to mark each state that is reached.
Algorithm M = "On input $\langle G, q, t\rangle$ where G is a DFA, q are the states and t are the transitions,
1. mark state q
2. Repeat until there are no unmarked states
3. Scan all t of G. If a transition (a,b) is found from a marked state a to unmarked state b, mark state b
4. Check to see if each t of G is marked, if each t is marked, accept otherwise reject"

This algorithm is in P
(Sipser 288)

PROBLEM 5 (3 points, suggested length of 1/4 page)

For $n \geq 1$, define the following function:

$$f(n) = |\{\omega \in \{a,b\}^n : \text{ aa not in } \omega \}|$$

Provide an asymptotic upper bound on f as n increases and justify your answer.

Solution.

$f(n) = |\{\omega \in \{a,b\}^n : \text{ aa not in } \omega \}|$

When $n = 1$ then $f(n) = 2$, (a or b)
When $n = 2$ then $f(n) = 3$, (bb, ab, ba)
When $n = 3$ then $f(n) = 5$, (aba, bbb, bba, bab, abb)
Similarly
When $n = 4$ then $f(n) = 8$
When $n = 5$ then $f(n) = 13$
When $n = 6$ then $f(n) = 21$
This is $f(n) = fib(n + 2)$
The fibonacci sequence, defined as $F_n = F_{n-1} + F_{n-2}$ can be thought of as a recursive tree with height n. Therefore it is clear that it has an upper bound of $O(n^2)$