Complexity and Advanced Algorithms – Assignment 2

Siddharth Bhat (20161105)

August 22, 2018

1 Write appropriate quantified formulae

1.1 infinitely many primes

We create a predicate called $Prime : \mathbb{N} \rightarrow \mathbb{P}$ which is true when a number is prime. We use this to define the infinitude of primes.

$$Prime : \mathbb{N} \rightarrow \mathbb{P}$$
$$Prime(n) \equiv \forall k \in \mathbb{N}, 2 \leq k < n \implies n\%k \neq 0$$

$$Infitude \equiv \forall n \in \mathbb{N}, Prime(n) \implies \exists m \in \mathbb{N}, m > n \land Prime(m)$$

Note that using the function $Prime$ is nothing special — it is simply shorthand for substituting the expression of $Prime$ into the expression for $Infitude$. I will continue to do this in the next question as well. This does not affect the correctness of the answer, since it can be converted into one large logical formula. Writing it this way simply makes it easier to reason about.

1.2 Every pair of positive integers have a unique GCD

We first define a predicate $CD : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{P}$, where $CD(n, m, d)$ means that d is a common divisor of n, m.

This is used to define $GCD : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{P}$, where $GCD(n, m, d)$ means that d is the GCD of n and m.

Finally, the definition of GCD is used to define the uniqueness of GCD.

$$CD : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{P}$$
$$CD(n, m, d) \equiv n\%d = 0 \wedge m\%d = 0$$

$$GCD : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{P}$$
$$GCD(n, m, d) \equiv CD(n, m, d) \wedge \forall d' \in \mathbb{N}, d' > d \implies \neg CD(n, m, d')$$

$$UniqueGCD \equiv \forall n, m, d, d' \in \mathbb{N}, GCD(n, m, d) \wedge GCD(n, m, d') \implies d = d'$$

2 Sorting $\in$ `LOGSPACE`

Assume we wish to sort in ascending order (The solution is symmetric in the descending order case).

Let us index the output list as $\langle o_1, o_2, o_3, \ldots, o_N$

In the output list, the numbers indexed by the set $[1, k_1]$ numbers are smaller than all the others: that is, they are greater than 0 of the other numbers in the list.

The numbers in the range of $[k_1 + 1, k_0]$ numbers are greater than 1 of the other numbers.

If we generalize, the numbers $[k_i + 1, k_{i+1}]$ are greater than $i - 1$.

we can iterate over the count $i - 1$ with a variable `nums_greater_required`, which represents the count of numbers the output must be greater than.

Next, for each `nums_greater_required`, we check each number in the list (indexed by `cur_num_position`) the count of numbers it is greater than (by walking the list using `compare_num_position`, and storing the count of numbers the current number is greater than in `cur_nums_greater`).

If this count is equal to `nums_greater_required`, we have the correct number.

```
def logsort_unique(input):
    # nums_greater_required := space O(log(|input|))
    for nums_greater_required in range(len(input)):

        # cur_num_position := space(O(log(|input|)))
        for cur_num_position in range(len(input)):

            # 0 <= cur_nums_greater <= nums_greater_required =>
            # cur_nums_greater := space O(log(|input|))
            cur_nums_greater = 0


            # compare_num_position := space(O(log(|input|)))
            for compare_num_position in range(len(input)):
```

```
        # input[cur_num_position] := space O(1)
        # input[compare_num_position] := space O(1)
        if input[cur_num_position] > input[compare_num_position]:
            cur_nums_greater += 1


    if cur_nums_greater == nums_greater_required:
        print (input[cur_num_position])
```

The time complexity of this is $O(|input| * |input| * |input|) = O(|input|^3)$

3 PSPACE is closed under union and intersection

Let L_1 and L_2 be languages in PSPACE . Let M_1 and M_2 be the turing machines which decide L_1 and L_2 respectively.

3.1 $L_1 \cup L_2 \in$ PSPACE

We create a new machine M which accepts the language $L_1 \cup L_2$.

This machine first runs M_1, and stores the output of M_1 in bit b_1. Since $M_1 \in$ PSPACE , M will not run out of space. Since M_1 is a decision procedure, we are guaranteed it will halt, so M waiting for M_1 to halti is legal.

Similarly, M then runs M_2 and stores b_2. finally, M outputs $b_1 \vee b_2$.

M accepts only when either M_1 or M_2 accept, that is, M accepts if a string belongs to either language.

3.2 $L_1 \cap L_2 \in$ PSPACE

Repeat the exact same construction, except change the output bit from $b_1 \vee b_2$ to $b_1 \wedge b_2$.

M accepts only when both M_1 and M_2 accept, that is, M accepts only if a string belongs to both languages.

4 Equivalence between NP as nondeterministic solver v/s deterministic verifier

4.1 nondeterministic solver $\implies$ deterministic verifier

Consider a language L, which has a non-deterministic solver $S \in$ NP . We construct a verifier V, and a function $witness : L \to \{0,1\}^*$ which constructs the witness string $y = witness(w)$ for a given string $w \in L$.

Consider a successful run of the non-deterministic solver S for a given $w \in L$. We create the witness string y by *recording* all of the values that the non-deterministic solver S *guessed for a successful run.*

So, $y =$ collection of all guesses made by S

We create the verifier V by running S deterministically, and at every point S tries to use non-determinism, we use the *correct value* that is stored in y.

Thus, if a successful run exists for S (that is, $w \in L$), then the verifier V will *have access to the right guesses.* On the other hand, if $w \notin L$, the there *are no right guesses*, so V will reject.

4.2 deterministic verifier $\implies$ nondeterministic solver

Consider a language L, with a deterministic verifier $V \in \mathtt{P}$.

We design a nondeterministic solver $S \in \mathtt{NP}$, which on input string w, guesses the witnesss string y using nondeterminism and then querifies the verifier with $V(w, y)$. That is,

$S(w) = V(w, \overline{y})$ where $\overline{y}$ is guessed using nondeterminism. This is in $\mathtt{NP}$, since the verifier V runs in polynomial time, and the non-determinism is used to guess $\overline{y}$.