Complexity and Advanced Algorithms – Assignment 3

Siddharth Bhat (20161105)

August 20, 2019

1 Give examples of sparse, non-sparse languages, context-free, non-context-free languages

1.1 Sparse language

$L = \{x \mid x \in \{0,1\}^*, x \text{ has exactly 2 bits equal to 1 in its binary representation}\}$

This is a sparse language, because for a fixed n, there are ${}^nC_2 = n(n-1)/2 = O(n^2)$ strings with exactly 2 bits as 1. Hence, the number of strings for any n is polynomial in n.

1.2 Non-Sparse language

$L = \{x \mid x \in 0, 1^*, x \text{ is even}\}$

for a given n, there are 2^n binary strings, of which $2^n/2 = 2^{(n-1)} = O(2^n)$ are even. Here, for a fixed n, the number of strings is exponential in n, which makes this language non-sparse

1.3 Context-free language

$L = \{a^n b^n \mid n \in \mathbb{N}\}$

We know that context free-languages are recognized by adeterministic push-down automata. So, we can construct an PDA which pushes every time it sees as a. When it sees the first b, it switches to a state that pops as. The PDA accepts if it empties the stack, when the string has been exhausted.

1.4 Non context-free language

$L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$

This language is not context free (a rigorous proof can be arrived at by pumping). Since context free languages can be recognized by push-down automata, the intuition is that the push-down automata is forced to use the stack to match $a^n b^n$, after which the stack is empty, and hence cannot match c^n.

2 Is every regular, context-free language decidable in `LOGSPACE`

2.1 Regular languages

Every regular language can be encoded as a deterministic finite automata. The size of a DFA does not change with problem size. Hence, a DFA can be simulated in $O(1)$ space, and is therefore definitely in `LOGSPACE`

2.2 Context-free languages

Yes, this can be done in `LOGSPACE` , by using the CYK algorithm.

We use the fact that any context-free grammar has an equivalent chomsky-normal form. Hence, given a grammar in the chomsky-normal form, we can employ the CYK algorithm to decide if the grammar is in `LOGSPACE` .

The intuition of the algorithm is that, given a grammar G, with nonterminals $R_1, R_2, \ldots R_r$, and an input string $I = i_1 i_2 \ldots i_n$, we bottom-up construct an array $P[n, n, r]$, where

$$P[start, length, r] \equiv \text{Can non-terminal } R_r \text{ produce the substring } a_{start} \ldots a_{start+length-1}?$$

We bottom-up construct this array P, which we will finally query with $P[1, n, S]$ where S is the start symbol of the grammar. This will tell us if the "substring $a_1 \ldots a_n \equiv a$ can be produced starting with the non-terminal S (which is the start symbol), thereby answering the membership query.

Note that $|P| = n^2 r$, so we cannot directly "store" P. However, since P is recursively defined, we can use recursion to find values on-the-fly.

2.2.1 deciding $CFG \in$ `NLOGSPACE`

```
# a := input string is a global

def is_substring(start, len, r):
    """Returns whether the substring
       "a_start a_{start+1} ... a_{start + len - 1}"
       can be produced from the nonterminal R_r

       r := O(1)
       start, len := O(log(n))
    """
    if (len == 1):
        # Return if R_r is of the correct shape to produce a_start
        return (R_r -> a[start])
    else:
        # Let there be a valid production := R_r -> R_p R_q
        # which can consume the string "a".
```

```
        p = <guess with non-determinism>
        plen = <guess with non-determinism>
        q = <guess with non-determinism>
        qlen = len - plen

        assert (plen >= 0 && qlen >= 0 && plen + qlen = len)

        # Note that the maximum value of (plen, qlen) = (n/2, n/2).
        # So, when we recurse, we will take at max log(n) steps
        # of the recursive call!
        if plen < qlen:
            if !is_substring(start, plen, p): return false
        else:
            if !is_substring(start + plen, qlen, q): return false


        # Now, this final recursive call is a tail-call, which
        # can be converted into a loop with no use of stack
        # Now that we have checked that the smaller string is legal,
        # we now check the larger substring can be legally produced
        # as a tail call, so we use no extra space
        if plen < qlen:
            return is_substring(start + plen, qlen, q)
        else:
            return is_substring(start, plen, p)

def membership():
    # The start symbol is the first non-terminal by convention
    return p(1, length(a), 0)
```

2.2.2 Deciding $CFG \in$ LOGSPACE

```
# a := input string is a global

def is_substring(start, len, r):
    """Returns whether the substring
       "a_start a_{start+1} ... a_{start + len - 1}"
       can be produced from the nonterminal R_r

       r := O(1)
       start, len := O(log(n))
    """
    if (len == 1):
        # Return if R_r is of the correct shape to produce a_start
        return (R_r -> a[start])
    else:
```

```
        # Let there be a valid production := R_r -> R_p R_q
        # This is O(1), and needs O(1) space to index [1..r]
        for every production R_r -> R_p R_q:
            # Consider all partitions of (plen, qlen) for the strings (R_p, R_q)
            for plen in range(0, len + 1):
                qlen = len - plen

                # Note that the maximum value of (plen, qlen) = (n/2, n/2).
                # So, when we recurse, we will take at max log(n) steps
                # of the recursive call!
                if plen < qlen:
                    if is_substring(start, plen, p):
                        # this is a tail call and takes no extra space.
                        return is_substring(start + plen, qlen, q)

                    return false
                else:
                    if is_substring(start + plen, qlen, q):
                        return is_substring(start, plen, p)
                    return false

def membership():
    # The start symbol is the first non-terminal by convention
    return p(1, length(a), 0)
```

3 Give examples of functions that are not space constructible or time constructible

3.1 non time & space constructible

Trivially, any non-computable is not time-constructible (indeed, it is not even constructible!). For example, pick the function $f(x) = \langle x \rangle (x) halts$ where $\langle x \rangle$ is x interpreted as a program. This function is not constructible by a diagonalization proof.

4 k-ary search on an array

It can be done in log-space. The intuition is that we require a logarithmic number of recursive calls, and for each step of the recursion, we need a constant (k) number of pointers into the memory. Hence, the full algorithm operates in log-space.

```
# arr is global scope
# arr := list(int)
```

```
# needle is global scope
# needle := int

def kary_search():
    kary_search_recur(arr, 0, length(arr));

# [begin, end) indexing.
def kary_search_recur(begin, end):
    assert (end > begin)

    # base case
    # -------------
    if (end == begin + 1):
        return arr[begin] == ix

    # recursive case
    # -------------
    # O(log(|arr|))
    max_ix_smaller_than_x = begin - 1;

    # O(log(|arr|))
    min_ix_larger_than_x = end + 1;

    #k := O(1), since K is independent of the problem size
    for k in range(1, K + 1):
        # ix = O(log(|arr|))
        ix = (end - begin + 1) / K

        # Indexed arr[ix], arr[max_ix_smaller_than_x] = constant space
        if (ix > max_ix_smaller_than_x and  arr[ix] < needle)
            max_ix_smaller_than_x = ix

        if (min_ix_larger_than_x < ix && arr[ix] > needle):
            min_ix_larger_than_x = ix


    assert (max_ix_smaller_than_x < min_ix_larger_than_x)
    # The recursion won't take more stack space because it's a tail
    # recursive call. One could imagine the whole function surrounded
    # by a while(1) {..} loop, which simply re-assigns the
    # values of begin, end!
    #
    # That is, one can transform a function call of then form:
    # def f(x, y):
    # ...
    # f(x', y')
```

```
# into:
# def f_no_stack(x, y):
#     xcur = x, ycur = y
#     while(1):
#         <body>
#         xcur = x', ycur = y'
# note that f_no_stack consumes no extra stack!
# https://stackoverflow.com/questions/33923/what-is-tail-recursion
kary_search_recur(max_ix_smaller_than_x, min_ix_larger_than_x)
```