RECITATION 8

FADI ATIEH

1. Complexity Analysis with Recursion

When using a recursive algorithm, the trickiest part to understanding it is, sometimes, the time/space complexity. Thus, it may be worth spending some time on the concept of recursion.

When thinking about recursive algorithms, it's often useful to think about the recursion tree. Recall that a recursive algorithm is a procedure that calls itself. For every such call, we designate a node in the recursion tree. We start at the root. The number of children for every node is equal to the number of times the corresponding call calls itself. Finally, the base cases correspond to the leaves.

When analyzing the *time* complexity for recursive algorithms, one usually needs to account for *all* the nodes in the tree and the time spent at each of them. This is because, in the worst case scenario, the algorithm may need to execute every single call (node) in the tree. This can get even more costly when one is not careful and executes some nodes more than once - that's what top-down dynamic programming tries to prevent. All of this complexity adds up as one *cannot reuse time*. This brings us to one crucial difference with space complexity analysis.

Space can be reused. Thus, for certain algorithms, we may not need to account for all the nodes in the recursion tree when analyzing space complexity. It may be possible to get away with only "keeping track of our progress". In other words, we may only need to store the information relevant to how we got to our current node starting from the root at any one time. Then, if we are to explore a different branch/path, we can simply "erase our progress" up to the first common ancestor node and continue from there. In such a case, space complexity analysis reduces to figuring out the most costly *path* from root to leaf instead of the whole tree. This is gives us a rough formula for space complexity analysis:

$$\text{Space Complexity} \;=\; \text{Depth} \;\times \text{Space-per-node}$$

Date: September 2019.

2. Example for space complexity analysis with recursion

Definition 1. A ladder is a sequence of words over some alphabet such that consecutive words differ by exactly one character.

Define

$$LADDER_{DFA} = \{\langle B, u, v\rangle \mid B \text{ is a DFA and } L(B) \text{ contains a ladder } y_1, y_2, \ldots, y_k \text{ where } y_1 = u \text{ and } y_k = v\}.$$

Problem 1. Show that $LADDER_{DFA} \in \text{SPACE}(n^2)$.

Proof. We define a more constrained version of the problem

$$BOUNDED - LADDER_{DFA} = \{\langle B, u, v, b\rangle \mid \langle B, u, v\rangle \in LADDER_{DFA} \text{ and the length of the ladder is } \leq b\}.$$

We give the following algorithm B-L for $BOUNDED - LADDER_{DFA}$

B-L = " On input $\langle B, u, v, b\rangle$,

1. For $b = 1$, accept if $u, v \in L(B)$ and differ in ≤ 1 place, else reject.
2. For $b > 1$, repeat for every w of length $|u|$,
3. Recursively test $u \xrightarrow{b/2} w, w \xrightarrow{b/2} v$.
4. Accept if both accept.
5. Reject".

For an input $\langle B, u, v\rangle$ for $LADDER_{DFA}$, we call B-L on $\langle B, u, v, |\Sigma|^m\rangle$ and output whatever B-L outputs.

To analyze the space complexity of this algorithm, we apply the formula in the previous section. At every recursive step we're dividing the length of the ladder by half. Thus, the depth of the tree is $\log(|\Sigma|^m) = \mathcal{O}(m) = \mathcal{O}(n)$. At every tree node, we're storing w requiring space $|w| = \mathcal{O}(m) = \mathcal{O}(n)$. Thus, the overall space complexity is Depth $\times$ Space-per-cell $= \mathcal{O}(n)\mathcal{O}(n) = \mathcal{O}(n^2)$. We are done. □

3. Another NP-complete problem

Problem 2. Let ϕ be a 3cnf-formula. An $\neq$assignment to the variables of ϕ is one where each clause contains two literals with unequal truth values. In other words, an $\neq$assignment satisfies ϕ without assigning three true literals in any clause. Let $\neq$SAT be the collection of 3cnf formulas that have an $\neq$assignment. Show that $\neq$SAT is NP-complete.

Proof. It is easy to see that $\neq$SAT is in NP. To show that the language is NP-complete, we make a reduction from 3SAT.

For a given Boolean formula ϕ, replace every clause c_i,

$$(y_1 \vee y_2 \vee y_3)$$

with the two clauses

$$(y_1 \vee y_2 \vee z_i) \wedge (\bar{z}_i \vee y_3 \vee b)$$

where z_i is a new variable for each clause and b is a single additional new variable. This reduction can obviously be done in polynomial time. Now, we prove its correctness.

Suppose ϕ is satisfiable. Then, at least one of y_1, y_2, y_3 is true. If $y_1 \vee y_2$ is true, assign z_i to False. Otherwise, assign z_i to True. Always assign b to False. This always gives a $\neq$assignment and proves the first direction of the reduction.

To prove the other direction, first notice that for a given $\neq$ assignment, the negation of such an assignment is also a valid $\neq$assignment. Thus, we can always assume without loss of generality that b is False. Then, for every pair of clauses

$$(y_1 \vee y_2 \vee z_i) \wedge (\bar{z}_i \vee y_3 \vee b),$$

if z_i is False, then $y_1 \vee y_2$ is True. Otherwise, y_3 must be true since both $\bar{z}_i, b$ are False. So ϕ can be satisfied and the reduction is correct. This ends the proof. $\square$