Ethan Wong
COM SCI 180
Discussion 1B
UID 305319001

Homework 3

1. This problem can be solved with a dynamic programming approach. The general idea for the algorithm is to have two pointers: “a” that starts at the beginning of the given string, and “b” which is at the end of the given string. These pointers will move through the string and process it. There will also be a table to keep track of memory.
   To begin with, there are two possibilities: the character that “a” points to is different from the character “b” points to, or the character that “a” points to is the same as the character “b” points to. The second case is good because it helps in forming the palindrome. In that scenario, “a” is simply incremented by one and “b” is decremented by one. In the first case, that spawns another two options: either insert a character at “b” so that it is the same as the character at “a” and increment “a” by one, or insert a character at “a” so that it is the same character at “b” and decrement “b” by one. This will count as an insertion that will be tracked. The minimum number of insertions required to transform the string into a palindrome will be returned from the table.

   The algorithm is as follows (it is partially real code, partially pseudocode)

```
def makePalindrome(str):
  str = a nonempty string (given as input to algorithm, I'm assuming str is nonempty)
  x = length of str
  if (x = 1 or str is already a palindrome):       # no need to do anything here
    return 0
  else:
    tab = [[0]*(x+1) for k in range(x+1)]      # create 2D table for memory (memoization)
    for a in range(x-1, -1, -1):
      for b in range(x):
        if (a < b):
          if (str[a] != str[b]):
            tab[a][b] = min(tab[a+1][b], tab[a][b-1]) + 1
          else:
            tab[a][b] = tab[a+1][b-1]
  return tab[0][x-1]
```

   The time complexity of this algorithm comes from the nested for loop. All other operations are less costly than the nested for loop, so the overall time complexity is $O(n^2)$.

2. This problem can be solved from an induction approach. We will start with a base case.

Base case

- n = 1 for the 3-SAT (the instance of the 3-SAT only contains a single clause)
- If we want an Approx-3-SAT with the same number of clauses as the 3-SAT, that would mean that n=2 for the Approx-3-SAT. To do this, we add a clause to Approx-3-SAT so that it has n=2 now.
- Now the Approx-3-SAT (n=2) matches the 3-SAT (n=1)
- Since they match, the answer for 3-SAT at n=1 can be found by using the solver of Approx-3-SAT at n=2

Inductive Hypothesis

- Based on the base case explained above, our inductive hypothesis is as follows: Any 3-SAT with n clauses can be solved by using the solver of approx-3-SAT with n+1 clauses
- Based on this, it can be proven that any 3-SAT with n+1 clauses can be solved by using the solver of approx-3-SAT with n+2 clauses

Proof

- Say that we have a conjunctive normal form (CNF) for both 3-SAT and Approx-3-SAT
- The CNF for 3-SAT is called “A”, and the CNF for Approx-3-SAT is called “B”
- $A(n+1) = C_1 \cap \ldots \cap C_n \cap C_{n+1} \Rightarrow B(n+1) = C_1 \cap \ldots \cap C_n$
- $A(n+2) = C_1 \cap \ldots \cap C_n \cap C_{n+1} \cap C_{n+2} \Rightarrow B(n+1) = C_1 \cap \ldots \cap C_n \cap C_{n+1}$
- $A(n+1) = C_1 \cap \ldots \cap C_n \cap C_{n+1} = B(n+2)$
- Based on this proof, we see that the CNF for Approx-3-SAT B(n+2) has the exact same number of clauses as the CNF for the 3-SAT A(n+1). Thus, we have performed a polynomial-time reduction of the 3-SAT into the Approx-3-SAT. Instances of Approx-3-SAT can be used to solve the 3-SAT problem. Basically, you just need to use a (n+1) clause version of Approx-3-SAT to solve a 3-SAT problem.

My initial intuition — this might be a bit clearer than the above explanation:

- Use the Approx-3-SAT solver on an instance (CNF) of 3-SAT - if the solver returns False, you know the instance of 3-SAT will also be false since at least two clauses were False
- Add one clause to the instance of 3-SAT that is guaranteed to be False
- Use the Approx-3-SAT solver on the modified instance - if it returns True, you know that the 3-SAT will also be true since only the newly added False clause evaluated to False => the rest of the original clauses must be True
- If the Approx-3-SAT solver returns False, you know that the 3-SAT will be False since the newly added False clause and one additional clause evaluated to False => one of the original clauses from the instance of 3-SAT must be False as well
- My explanation above uses induction to be a bit more formal, but this is initially what I was thinking