1. Define $T$ as the smallest such such that:
   (a) Symbol $*$ $\in T$.
   (b) If $t_1, t_2 \in T$, then $(t_1t_2) \in T$

   Some examples of elements of $T$ are $*$, $(**)$, and $((**))$.

   (a) How many elements of $T$ have (a) 0 left parentheses (b) 1 left parentheses (c) 2 left parentheses, (d) 3 left parentheses, and (d) 4 left parentheses?
   (b) Devise a recurrence $c(n)$ such that $c(n)$ is the number of different elements of $T$ with $n$ left parentheses. Explain why your $c(n)$ is correct.

2. Devise a function $p(n)$ such that $p(n)$ is the number of different ways to create postage of $n$ cents using 3-, 4-, and 5-cent stamps.
   (a) Carefully explain why your $p(n)$ is correct.
   (b) Prove that $p(n)$ is monotonic nondecreasing on $\mathbb{N}^+$

3. Consider the recurrence $T$ that we derived for the worst-case time complexity of recBinSearch:

   $$T(n) = \begin{cases} 
   c & \text{if } n = 1 \\
   1 + T([n/2]) & \text{if } n > 1 
   \end{cases}$$

   (a) Emulate Lemma 3.6 from the course notes to prove that $T$ is nondecreasing.
   (b) Use simple induction on $k$ to prove that $\forall k, n \in \mathbb{N}, n = 2^k \Rightarrow T(n) = \lg(n) + c$.
   (c) Combine the previous two steps to prove that $T \in \Theta(\lg)$. Do not use induction.

4. Suppose you take the trouble to legally change your name to a string from the alphabet $\{A, C, T, G\}$, for example $TAGAC$ might make a fine name. Then you (discreetly) collect DNA samples from your friends, e.g. nail clippings, hair, used coffee cups. Your idea is to count the number of times your name occurs as a subsequence of their DNA (you may need some help sequencing that), and then invoice
them for that many licenses of your intellectual property. If your friend’s DNA contained the string
ATAGGACCA they’d owe you for at least four licenses!

Clearly you’ll need some computational help counting sequences. Read over the code below and either
(a) prove that its precondition plus execution implies its postcondition, or (b) provide a counterexample
that shows it is incorrect.

def count_subsequences(s1: str, s2: str, i: int, j: int) -> int:
    """ Return the number of times s1[: i] occurs as a
    subsequence of s2[: j].
    Precondition: 0 <= i <= len(s1), 0 <= j <= len(s2)
    >>> count_subsequences("", "Danny", 0, 5)
    1
    >>> count_subsequences("Danny", ",", 5, 0)
    0
    >>> count_subsequences("AAA", "AAAAA", 3, 5)
    10
    Postcondition: returns number of times s1[: i] occurs as a
    subsequence of s2[: j]
    """
    if i == 0:
        return 1
    elif i > j:
        return 0
    elif s1[i-1] != s2[j-1]:
        return count_subsequences(s1, s2, i, j-1)
    else:
        return (count_subsequences(s1, s2, i, j-1)
        + count_subsequences(s1, s2, i-1, j-1))

You may also find the above code, plus an efficient memoized version, in sequences.py
5. Read over unimplemented function `shade_sort` below:

```python
def shade_sort(colour_list: List[str]) -> None:
    """Put colour_list in order "b" < "g" < "r".
    precondition: colour_list is a List[str] from {"b", "g", "r"}
    >>> list_ = ["r", "b", "g"]
    >>> shade_sort(list_)
    >>> list_ == ["b", "g", "r"]
    True
    postcondition: colour_list has same strings as before, ordered "b" < "g" < "r"
    ""
    # TODO: initialize blue, green, red to be consistent with loop invariants.
    # Hint: blue, green may increase while red decreases.
    #
    # loop invariants:
    #
    # 0 <= blue <= green <= red <= len(colour_list)
    # colour_list[0 : green] + colour_list[red :] same colours as before
    # and all([c == "b" for c in colour_list[0 : blue]])
    # and all([c == "g" for c in colour_list[blue : green]])
    # and all([c == "r" for c in colour_list[red :]])
    #
    # TODO: implement loop using invariants above!
```

**implement**: Use the invariants as a guide to implement function `shade_sort`.

**prove**: Use those same invariants to prove that your function `shade_sort` is correct: first partial correctness, then termination.

**extend**: Modify the invariants of `shade_sort` so they specify function `four_shade_sort` which adds a fourth string "y" (meaning yellow) that must end up following the other three when `colour_list` is sorted. Then implement function `four_shade_sort`.

Submit both implementations (including invariants) as `sorting.py`, along with `a2.pdf`. You will find the above code in `sorting.py`