Gina Ferguson || 12/3/2025 || CS350 || HW 6: Dynamic Programming

Problem 1

Part A:

We will work on a classic coding interview problem: Find the length of the longest palindromic substring* for a given input string.

(a) [7 points] Problem 6 (MysteryAlgorithm) in the bonus midterm solved this problem. Provide a one sentence justification for why you think that implementation does or does not belong to the following design strategies: [iterative algorithm design, divide and conquer, brute force, decrease and conquer, string processing, recursive algorithm design, decrease-by-variable-size and conquer]. Remember that an algorithm can belong to more than one design category!

(* A substring should not be confused with a subsequence. A substring is a set of **contiguous** characters located within an input string. A subsequence is any subset of characters from a string that appear in the same sequence in the string. For example, `abc` is a substring of `yrabcbiiu` but `yciu` is a not a substring because these characters do not occur contiguously. However `yciu` is a subsequence of `yrabcbiiu` because these characters appear in that sequence in the longer string.)

- Iterative Algorithm Design: No because there are not any loops, just conditional statements using if else statements and recursive calls.
- Divide and Conquer: Yes the string is an array that is being divided into sections using conditional if else statements.
- Brute Force: Yes, because it tries each possibility using if else statements.
- Decrease and Conquer: Yes, because the recursive calls move the testing section around to smaller arrays to test.
- String Processing: Yes, because a string is literally being processed using arrays with indices character by character and searches for a substring.
- Recursive Algorithm Design: Yes, because the algorithm calls itself.
- Decrease-by-variable-size and Conquer: Yes, the substring size is decreased depending on the recursive call

---

Part B: (below)

(b) [5 points] Using your favorite programming language, implement a brute-force iterative algorithm that returns the length of the longest palindromic substring of a given input string. Make sure to clearly document your code and include instructions on how to run it. Show a screenshot for outputs for five example strings that each include palindromic substrings of different lengths.

- See part-b.py code.
- Run the code by opening the parent folder of the code file in the terminal.
- In the same terminal run command: "python3 part-b.py" without the " " marks.
- Test away :)

Part C:

(c) [2 points] What is the asymptotic runtime of the iterative approach? What is the space complexity of the iterative approach? Your answers should be in terms of big-Theta or big-O.

- **Time Complexity:**
  - **Operation: All possible substrings (comparison)**
    - **--> Outer + inner loops = $\Theta(n^2)$**
  - **Operation: Checking if Palindrome (comparison)**
  - **--> function is_palindrome worst case: $\Theta(n)$**

- **Final Time Complexity:**
  - **$\Theta(n^3)$**

- **Space Complexity: $\Theta(n)$**
  - **2 indices, length of string variable, etc**
  - **1 temporary substring candidate, only one at a time**
    - **Worse Case: $\Theta(n)$**
      - **This doesn't change because we are only storing 1 at a time on the side.**
- **Recurrence Relation:**
  - **Looping over i and j≥i**
    - **k = j - i + 1 (cost $\Theta(k)$)**
    - **T(n) ={ ∑[(i=0) to (n-1)] } * { [∑[(j=i) to (n-1)] }**
    - **Let k = j - i + 1**
    - **T(n) = ∑[(k=1) to m] = [m(m+1)]/2**
    - **n(n+1)(n+1)/6**
    - **= Θ(n3)**

    - **Also: pal = palindrome**
      - **T(n) = n^2 * T_pal(n) = Θ(n3)**

Part D:

(d) [5 points] Implement a dynamic program for the longest palindromic substring problem in your favorite programming language.

- See part-d.py code.
- Run the code by opening the parent folder of the code file in the terminal.
- In the same terminal run the command: "python3 part-d.py" without the " " marks.
- Test away :)

Part E:

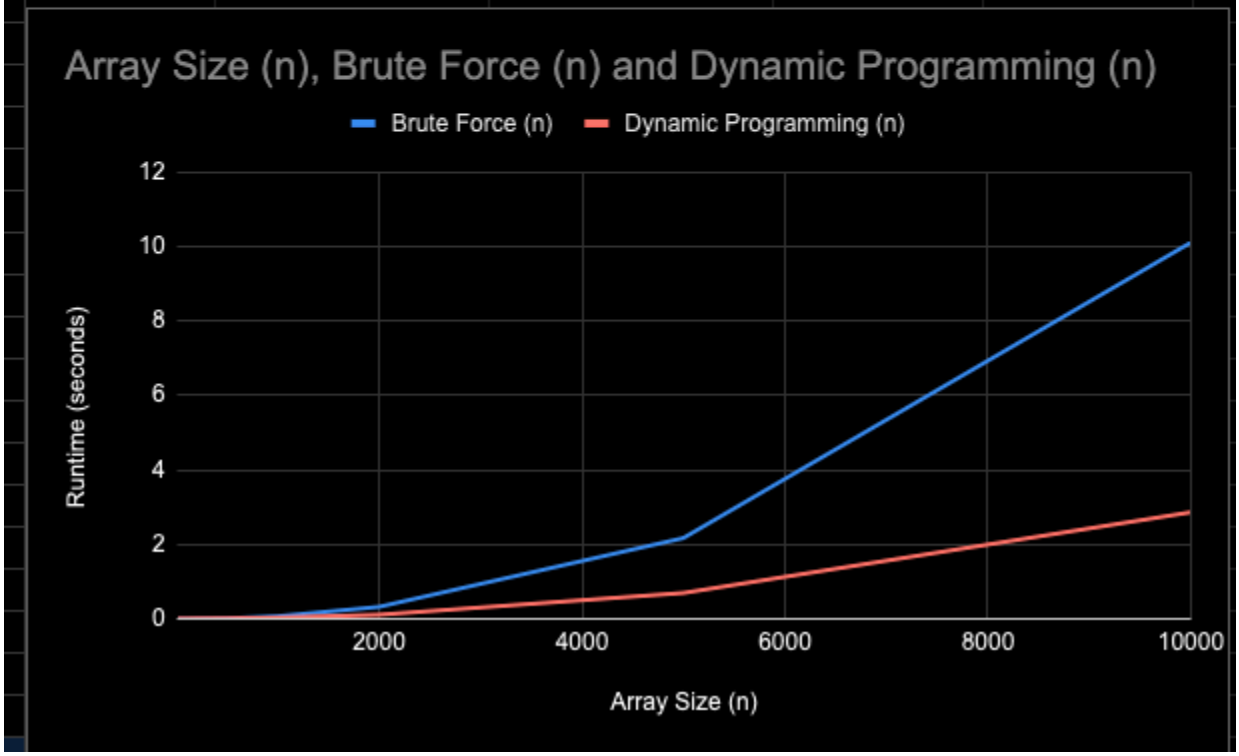
(e) [2 points] What is the asymptotic runtime of your dynamic programming approach? What is the space complexity of the dynamic programming approach? Your answers should be in terms of big-Theta or big-O.

- Time Complexity:
  - Operation: fill memoTable (like a truth table)
    - --> 2 nested loops over all substring
    - --> outerloop (length) + inner loop (start index) = Θ(n^2) quadratic time
  - Operation: Dynamic Prog. recurrence
    - --> each cell memoTable[i][j] computed in Θ(1) Constant time
    - --> no substring scanned like brute force
- Final Time Complexity:
  - Θ(n^2)
- Space Complexity:
  - 2D list of booleans --> memoTable[i][j]
  - no substring allocations, no recursion
  - Worse Case:
    - --> n^2 entries
    - --> Θ(n^2)
      - Θ(n^2) because store results as you go for every substring pair
- Summation:
  - like the:
    - T(n) = ∑(k = 1 to n) k
      = n(n+1)/2 {the consecutive sum of integers recurrence relation :) }
- = Θ(n^2)

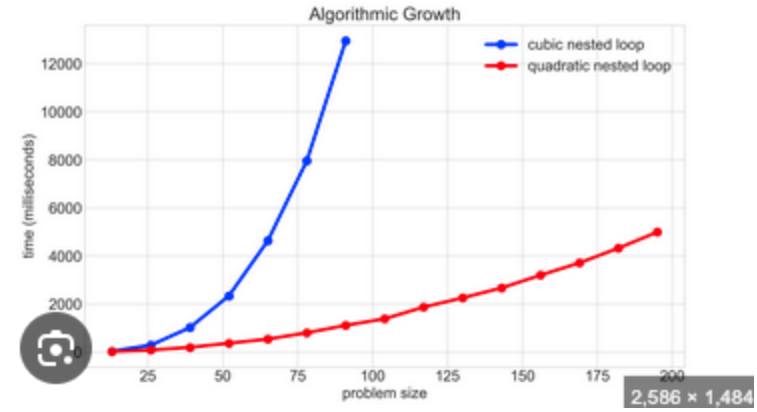
Part F:

(f) [3 points] Generate plots of the runtime of your iterative implementation (part b) and your dynamic programming implementation (part d) for increasing lengths of input strings (e.g. 10, 100, 1000, 10000, etc.). Show both plots on the same axis and comment on how they compare to what you expect from parts (c) and (e).

Array Size (n)	Brute Force (n)	Dynamic Programming (n)
10	0.00001287	0.00001311
50	0.00017118	0.00005889
100	0.00059199	0.00023484
200	0.00235486	0.00088191
500	0.01650214	0.00591278
1000	0.073807	0.02490425
2000	0.32416797	0.10386896
5000	2.17450595	0.68790102
10000	10.11177707	2.85548162



Cubic Runtime grows a little faster unless this just shows the beginning of where the 2 separate. It doesn't match up fairly well when I compare my graph above to the graph below that I found online. And the red lines looks like quadratic growth as well.



Part G (BONUS):

(g) **[Bonus question]** Read about Manacher's algorithm (on Wikipedia or elsewhere). Does it solve the same problem? What is its asymptotic complexity?

I believe that it solves the same problem using radii (comparing matching symbols from the center outward. It seems to use partition characters in between letters called the pre-processing stage? I feel like I might be missing an edge case where it wouldn't work, but it supposedly uses linear O(n) time and only needs extra space if it isnt able to compare properly.

SOURCES:

Link:

https://www.geeksforgeeks.org/dsa/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/
