Shuyi Jin sj445
ECE 590: Theory and Practice of Algorithms
November 10, 2024

Homework 6

Question 1. 1. Indicate which of the following are True (provide a sentence explaining why your choices are correct)

1. Indicate which of the following are true and provide a reason:

(a) $2n^3 - 8n^2 + 32n + 9 \in O(n^3)$

Answer: This is **true** because, according to the definition of asymptotic notation, the highest-degree term n^3 dominates the growth rate. The lower-degree terms $-8n^2$, $32n$, and 9 become negligible as $n \to \infty$, so $2n^3 - 8n^2 + 32n + 9 = O(n^3)$.

(b) $n^p \in O(e^n)$, where $p \in \mathbb{R}$ and $p \geq 0$

Answer: This is **true**. The exponential function e^n grows significantly faster than any polynomial n^p (even for very large p). Thus, $n^p = O(e^n)$.

(c) $e^n \in O(n^p)$, where $p \in \mathbb{R}$ and $p \geq 0$

Answer: This is **false**. The exponential function e^n grows faster than any polynomial n^p, so there cannot exist constants C and n_0 such that $e^n \leq C \cdot n^p$ for all $n > n_0$. Therefore, $e^n \notin O(n^p)$.

(d) $\sqrt{n} \in O(1)$

Answer: This is **false**. $O(1)$ represents constant time complexity, but $\sqrt{n}$ grows as n increases and is not a constant function. Therefore, $\sqrt{n} \notin O(1)$.

Question 2. Show that for any real constants x and y, where y > 0, (n + x) with the power of y = (n with the power of y).

To show that for any real constants x and y, where $y > 0$, $(n + x)^y = \theta(n^y)$, we can use the definition of θ-notation. Specifically, we need to show that there exist positive constants c_1, c_2, and n_0 such that:

$$c_1 \cdot n^y \leq (n + x)^y \leq c_2 \cdot n^y \quad \text{for all } n \geq n_0.$$

Proof:

1. Upper Bound: Since $y > 0$, as $n \to \infty$, the term n dominates x in $(n + x)^y$. Thus, there exists a constant $c_2 > 0$ such that:

$$(n + x)^y \leq c_2 \cdot n^y.$$

This inequality holds because $n + x \approx n$ as n grows large.

2. Lower Bound: Similarly, for large n, n also dominates x in $(n + x)^y$, so there exists a constant $c_1 > 0$ such that:

$$(n + x)^y \geq c_1 \cdot n^y.$$

3. Conclusion: Together, these bounds imply that:

$$c_1 \cdot n^y \leq (n + x)^y \leq c_2 \cdot n^y$$

for all $n \geq n_0$. Therefore, we conclude that:

$$(n + x)^y = \theta(n^y).$$

Question 3. Give asymptotic upper and lower bounds for T(n) in each of the following recurrences. Assume that T(n) is constant for sufficiently small n. Make your bounds as tight as possible, and justify your answers[8 points].

Here are the solutions with asymptotic bounds for each recurrence, along with justifications:

(a) $T(n) = 16T\left(\frac{n}{4}\right) + n^2$

Using the Master Theorem, the recurrence is of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a = 16$, $b = 4$, and $f(n) = n^2$.

Calculate $n^{\log_b a}$:

$$n^{\log_4 16} = n^2$$

Now we compare $f(n) = n^2$ with $n^{\log_b a} = n^2$: Since $f(n) = n^2 = n^{\log_b a}$, this case falls under Case 2 of the Master Theorem, which states that if $f(n) = \theta(n^{\log_b a})$, then $T(n) = \theta(n^{\log_b a} \log n)$.

Therefore:

$$T(n) = \theta(n^2 \log n)$$

(b) $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$

Using the Master Theorem, this recurrence has $a = 2$, $b = 4$, and $f(n) = \sqrt{n} = n^{1/2}$.

Calculate $n^{\log_b a}$:

$$n^{\log_4 2} = n^{1/2}$$

Now we compare $f(n) = n^{1/2}$ with $n^{\log_b a} = n^{1/2}$: Since $f(n) = n^{1/2} = n^{\log_b a}$, this is again Case 2 of the Master Theorem.

Thus:

$$T(n) = \theta(n^{1/2} \log n) = \theta(\sqrt{n} \log n)$$

(c) $T(n) = T(n-2) + n^2$

This recurrence does not fit the Master Theorem, as it is not in the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. However, we can analyze it by expanding terms:

$$T(n) = T(n-2) + n^2 = T(n-4) + (n-2)^2 + n^2 = \cdots = T(0) + \sum_{k=1}^{n/2} (n-2k)^2$$

The sum $\sum_{k=1}^{n/2} (n-2k)^2$ approximates $\theta(n^3)$, as the dominant term in this expansion is cubic.

Therefore:

$$T(n) = \theta(n^3)$$

(d) $T(n) = 3T\left(\frac{n}{3}\right) + \frac{n}{\log n}$

Using the Master Theorem, this recurrence has $a = 3$, $b = 3$, and $f(n) = \frac{n}{\log n}$.

Calculate $n^{\log_b a}$:

$$n^{\log_3 3} = n$$

Now we compare $f(n) = \frac{n}{\log n}$ with $n^{\log_b a} = n$: Since $f(n) = o(n)$ (i.e., it grows slower than n), this falls under Case 3 of the Master Theorem, which states that if $f(n) = o(n^{\log_b a})$, then $T(n) = \theta(n^{\log_b a})$.

Thus:

$$T(n) = \theta(n)$$

Question 4. The first problem you should implement is in matrixmult.py. Here, you are going to write matrix multiplication. Note that there are many fancy matrix multiplication algorithms with improvements to the runtime over the "standard" approach (that you would do by hand). You are welcome to do a fancy algorithm if you want to learn such—however, please note that what you write should reflect your understanding of that algorithm and not copying code. If you do learn a fancy algorithm, please cite any relevant sources.

psuedocode1

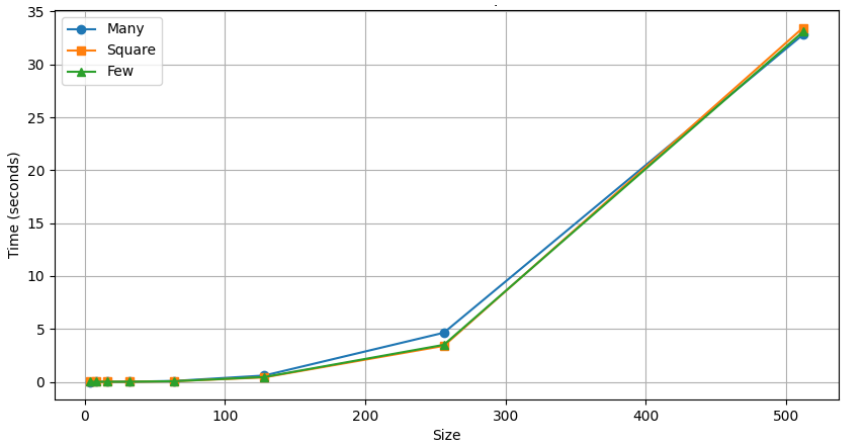
```
psuedocode1:
Function matrix_mul(a, b):
    Assert: Number of columns in a is equal to the number of rows in b

    # Initialize result matrix res with dimensions (rows of a) x (columns of b), filled with zeros
    Create matrix res with dimensions len(a) x len(b[0]), initialized to 0

    # Iterate over each row i of matrix a
    For each i from 0 to len(a) - 1:
        # Iterate over each column j of matrix b
        For each j from 0 to len(b[0]) - 1:
            # Perform the dot product for res[i][j]
            For each k from 0 to len(a[0]) - 1:
                res[i][j] = res[i][j] + (a[i][k] * b[k][j])

    Return res
End Function
```

list:(unit: size, runtime seconds)

Size(n)	few(s)	square(s)	many(s)
4	0.00004	0.00400	0.00417
8	0.00013	0.00012	0.00012
16	0.00101	0.00400	0.00085
32	0.00988	0.00640	0.00774
64	0.08039	0.05330	0.05454
128	0.59022	0.41712	0.44878
256	4.63811	3.39450	3.49501
512	32.81978	33.46625	33.08475

TABLE 1. Data Table

graph :



Analysis: For standard matrix multiplication algorithm, the big O is O(m*x*p) n is the number of rows in matrix m is the number of columns in matrix p is the number of columns in matrix

Therefore it is $O(n^3)$.

Just now look at the graph, as N increases from 4 to 512, the computation time of matrix multiplication will increase cube wise, which is consistent with The time complexity of O (n^3).

The impact of matrix shape: Different matrix shapes (such as many rows, few columns, and square) can affect performance, depending on the operands and memory layout. But the overall complexity remains high $O(n^3)$

However, actual matrix multiplication calculations may result in different constant factors due to differences in data storage and access modes.

For another aspect, for the same size, the different time of many ,square and few are almost the same.

Question 5. For this problem, you are going to work in eqsubstr.py, and write The matchinglengthsubstrs function. This function takes a string (s) and two characters (c1 and c2). The goal of this function is to find consecutive sequences of c1 and consecutive sequences of c2 that are the same length. It should return a set of three-tuples where each element of the tuple is (c1-index, c2-index, length). For example, an element in the answer with value (5, 12, 3) means "there are 3 c1s at index 5 and also 3 c2s at index 12". In this problem, we only consider the total sequence length (if you see 4 "a"s in a row, you only call it a sequence of length 4, not also a sequence of length 3), and we only consider lengths > 0.

Analysis:

1. Best Case: Matching consecutive 'c1' characters:

$$c1^+$$

This regular expression matches one or more consecutive occurrences of the character 'c1'.

Matching consecutive 'c2' characters:

$$c2^+$$

This regular expression matches one or more consecutive occurrences of the character 'c2'.

2. Worst Case: Matching alternating 'ab' substrings:

$$(ab)^+$$

This regular expression matches one or more alternating substrings of 'a' and 'b'.

Matching alternating 'ba' substrings:

$$(ba)^+$$

This regular expression matches one or more alternating substrings of 'b' and 'a'.

These regular expressions describe the patterns of the characters 'c1' and 'c2' in the best and worst case scenarios.

Test Outputs:

Best Case: As the data size increases, the runtime of the algorithm remains relatively stable because characters a and b are consecutive. The process of updating the dictionary and matching sequences is not very complex.

Worst Case: As the data size increases, the runtime may grow significantly, especially in the case where each pair of characters is different (e.g., "ababab..."), resulting in frequent dictionary updates.

Random Case: For random inputs, the algorithm's runtime is affected by the distribution of characters. As the data size increases, the runtime gradually increases but remains within a reasonable range.

Conclusion:

Overall Performance: The algorithm exhibits a linear time complexity of $O(n)$ across different input sizes, which is effective for most use cases.

Optimization Potential: While the algorithm's time complexity is already $O(n)$, the operations on the dictionary in the worst case may introduce higher constant factors. If the input contains highly complex character distributions (such as alternating characters), the efficiency might be slightly impacted. In such cases, optimizing the use of the dictionary or merging sequences more efficiently could reduce the number of dictionary insertions.

psuedocode2

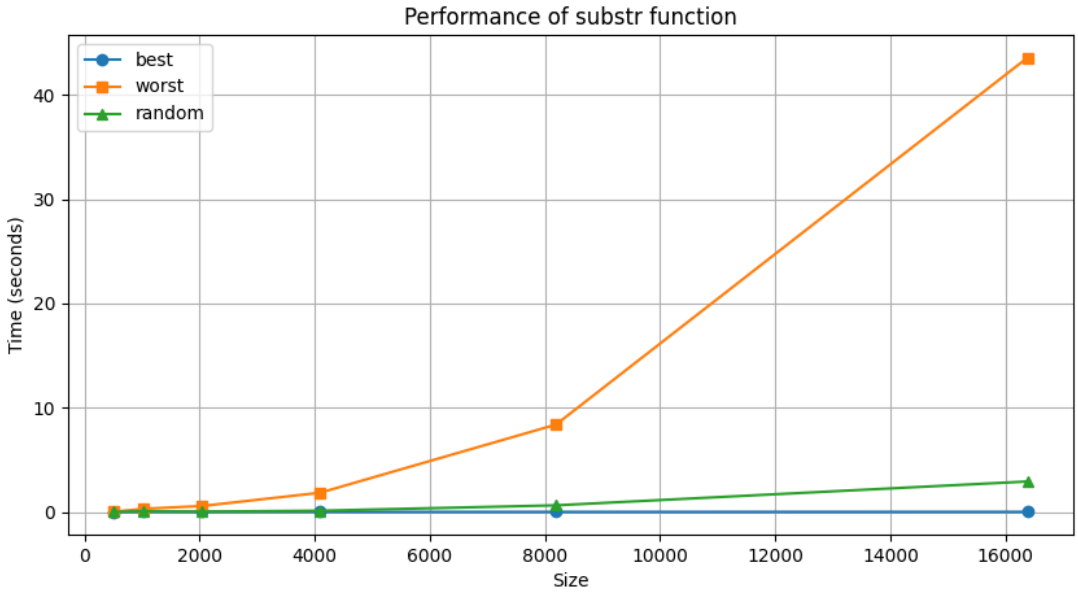
```
Function matching_length_sub_strs(s, c1, c2):
    Initialize empty dictionaries dic_c1 and dic_c2 to store substrings' lengths and starting indices for c1 and c2
    Initialize counters cnt1 and cnt2 to 0 for tracking consecutive occurrences of c1 and c2

    Define helper function update_dict(dic, key, value):
        If key exists in dic:
            Append value to dic[key]
        Else:
            Set dic[key] to a list containing value

    For each character in string s at index i:
        If character is equal to c1:
            Increment cnt1
            If cnt2 is not 0:
                Call update_dict(dic_c2, cnt2, i - cnt2) to store count and index in dic_c2
                Reset cnt2 to 0
        Else if character is equal to c2:
            Increment cnt2
            If cnt1 is not 0:
                Call update_dict(dic_c1, cnt1, i - cnt1) to store count and index in dic_c1
                Reset cnt1 to 0
        Else (character is neither c1 nor c2):
            If cnt1 is not 0:
                Call update_dict(dic_c1, cnt1, i - cnt1)
                Reset cnt1 to 0
            If cnt2 is not 0:
                Call update_dict(dic_c2, cnt2, i - cnt2)
                Reset cnt2 to 0

    After loop, check for remaining counts:
        If cnt1 is not 0:
            Call update_dict(dic_c1, cnt1, len(s) - cnt1)
        If cnt2 is not 0:
            Call update_dict(dic_c2, cnt2, len(s) - cnt2)

    Initialize an empty set ans to store matching length substrings

    For each key (length) in dic_c1:
        If key is also in dic_c2:
            For each start index j in dic_c1[key]:
                For each start index k in dic_c2[key]:
                    Add tuple (j, k, key) to ans

    Return ans
```

list:(unit: size n, runtime second)

Size (n)	best(n)	worst(n)	random(n)
512	0.00010	0.01716	0.00255
1024	0.00013	0.29560	0.06226
2048	0.00036	0.56760	0.02407
4096	0.00073	1.84010	0.11372
8192	0.00105	8.37546	0.63015
16384	0.00290	43.57670	2.91763

TABLE 2. Data Table

graph :


Performance of substr function
best
worst
random
Time (seconds)
Size
0
10
20
30
40
0
2000
4000
6000
8000
10000
12000
14000
16000