1 Tromino Puzzle

a) The algorithm, L-tile, below simulates and solves the tromino puzzle represented by a 2 dimensional array. For \( m > 0 \), a \( 2^m \times 2^m \) sized array, \( A \), is given to L-tile along with the index of the missing tile, \((a, b)\), the row number, \(k\), and the column number \(j\) of the upper left corner of the array, and the number of columns of the array, \( n = 2^m \). Initially all indexes of \( A \) will be set to -1, representing unfilled spaces on the chessboard, except \( A[a, b] \) is set to zero, representing the missing tile. The idea of the algorithm is for \( m > 1 \), to break up the array into 4 square subarrays. For the three subarrays which do not contain the missing tile, the problem is solved for each with a ”missing tile” in the corner meeting the other subarrays in the middle. Since these corners connect in the required \( L \) shape, once the problem is solved for the four subarrays a tile can be placed there and all squares are covered with ”L’s” except for the square with the missing tile. Assume also that the original array handed to L-tile has indexes beginning with 1. The algorithm is hard to read but you can get the basic idea if you replace all j’s and k’s with 1’s for the first division of the problem. Also, there is a graphic on page 4 which indicates how L-tile places trominoes.
1. L-tile( A, (a, b), j, k, n)

2. \( L_1 \leftarrow (0, 0), L_2 \leftarrow (0, 0), L_3 \leftarrow (0, 0) \)

3. if \( n = 2 \)

4. for row \( \leftarrow j \) to \( j + 1 \).
5. for col \( \leftarrow k \) to \( k + 1 \)
6. if row \( \neq a \) or col \( \neq b \)
7. \( A[row, col] \leftarrow 1 \) // places an L over squares without missing tile

//Below the problem is divided into 4 subproblems

8. else
9. if not(\( j \leq a \leq \frac{n}{2} + (j - 1) \)) or not(\( k \leq b \leq (k - 1) \))
10. L-tile(A, (\( \frac{n}{2} + (j - 1), \frac{n}{2} + (k - 1) \)), j, k, \( \frac{n}{2} \))
11. \( L_1 \leftarrow (\frac{n}{2} + (j - 1), \frac{n}{2} + (k - 1)) \)
12. else
13. L-tile(A, (a, b), j, k, \( \frac{n}{2} \))

14. if not(\( \frac{n}{2} + j \leq a \leq n \)) or not(\( k \leq b \leq \frac{n}{2} + (k - 1) \))
15. L-tile(A, (\( \frac{n}{2} + j, \frac{n}{2} + (k - 1) \)), \( \frac{n}{2} + j \), k, \( \frac{n}{2} \))
16. if \( L_1 \neq (0, 0) \)
17. \( L_2 \leftarrow (\frac{n}{2} + (j - 1), \frac{n}{2} + (k - 1)) \)
18. else
19. \( L_1 \leftarrow (\frac{n}{2} + (j - 1), \frac{n}{2} + (k - 1)) \)
20. else
21. L-tile(A, (a, b), \( \frac{n}{2} + j \), k, \( \frac{n}{2} \)).
22. if not($j \leq a \leq \frac{n}{2} + (j - 1)$) or not($\frac{n}{2} + k \leq b \leq n$)

23. L-tile($A, (\frac{n}{2} + (j - 1), \frac{n}{2} + k), \frac{n}{2} + j, \frac{n}{2} + (k - 1), \frac{n}{2}$)

24. if $L_1 \neq (0, 0)$ and $L_2 \neq (0, 0)$

25. $L_3 \leftarrow (\frac{n}{2} + (j - 1), \frac{n}{2} + k)$

26. else

27. $L_2 \leftarrow (\frac{n}{2} + (j - 1), \frac{n}{2} + k)$

28. else

29. L-tile($A, (a, b), \frac{n}{2} + j, k, \frac{n}{2}$)

30. if not($\frac{n}{2} + j \leq a \leq n$) or not($\frac{n}{2} + k \leq b \leq n$)

31. L-tile($A, (\frac{n}{2} + j, \frac{n}{2} + k), \frac{n}{2} + j, \frac{n}{2} + k, \frac{n}{2}$)

32. $L_3 \leftarrow (\frac{n}{2} + j, \frac{n}{2} + k)$

33. else

34. L-tile($A, (a, b), \frac{n}{2} + j, \frac{n}{2} + k, \frac{n}{2}$)

//Now to cover the squares where three subarrays meet with an L shape.

35. $A[L_1] \leftarrow 1; A[L_2] \leftarrow 1; A[L_3] \leftarrow 1;

b-i) The divide and combine steps of the algorithm take constant time. The conquer step splits the $n$-sized problem (an $n \times n$ array) into four $\frac{n}{2}$ sized (four $\frac{n}{2} \times \frac{n}{2}$ arrays) subproblems.

When $n = 2$, for a $2 \times 2$ array, the algorithm takes constant time. So, we can write the recurrence as:

$$T(n) = \Theta(1) \text{ when } n = 2$$

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(1) \text{ when } n > 2.$$
Illustration of how L-tile would fill an 8 \times 8 chessboard with missing tile at index (2, 6).
b-ii) \( T(n) = 4T\left(\frac{n}{2}\right) + \Theta(1) = \Theta(1) \)

\[
\begin{array}{c}
\Theta(1) \\
T\left(\frac{n}{2}\right) \quad T\left(\frac{n}{2}\right) \\
\Theta(1) \quad \Theta(1) \quad \Theta(1) \quad \Theta(1) \\
T\left(\frac{n}{4}\right) \quad T\left(\frac{n}{4}\right) \quad T\left(\frac{n}{4}\right) \quad T\left(\frac{n}{4}\right) \\
\Theta(1) \quad \Theta(1) \quad \Theta(1) \quad \Theta(1) \\
\vdots \\
T\left(\frac{n}{2^k}\right) \quad \ldots \quad \ldots \\
\end{array}
\]

\( = \Theta(1) \quad 4^0 \)

\( = \Theta(1) \quad 4^1 \)

\( = \Theta(1) \quad 4^2 \)

\( = \Theta(1) \quad 4^k \)

Solving for \( k \) gives:

\[
\frac{n}{2^k} = 2 \Rightarrow 2^{k+1} = n \Rightarrow \log_2 2^{k+1} = \log_2 n \Rightarrow k = \log_2 n - 1
\]

Counting the constants in the recursion tree gives:

\[
1 + 4 + 4^2 + \ldots + 4^{\log_2 n - 1} = \frac{4^{\log_2 n - 1 + 1}}{3} = \frac{2^{\log_2 n} 2^{\log_2 n}}{3} = \frac{n^2}{3}
\]

So the running time of L-tile looks to be \( \Theta(n^2) \).
b-iii) L-tile takes as input arrays of size $2^k \times 2^k$ for $k \geq 1$. According to our running time recurrence, For a $2 \times 2$ array L-tile has a $\Theta(1)$ running time. Since $\Theta(1) = \Theta(4) = \Theta(2^2)$, the base case of the algorithm, when $n = 2$ takes $\Theta(n^2)$ time.

Now assume that for some $n$ where $n = 2^k$, $T(2^k) = \Theta(2^k)$. According to the recurrence for L-tile, $\Theta(2^{k+1}) = 4T(\frac{2^{k+1}}{2}) + \Theta(1) = 4T(2^k) + \Theta(1)$, which according to our assumption equals:

$$4 \times (2^k)^2 + \Theta(1) = 4 \times (2^k) + \Theta(1) = 2^{2k+2} + \Theta(1) = (2^{k+1})^2 + \Theta(1).$$

Since, $(2^{k+1})^2 < (2^{k+1})^2 + \Theta(1)$, $T(2^{k+1}) = \Omega((2^{k+1})^2)$.

Now we wish to find $m$ such that (i) $(2^{k+1})^2 + \Theta(1) \leq m \times (2^{k+1})^2$. If we assume the constant hidden in the $\Theta$ notation equals $c$ we have:

$$(2^{k+1})^2 + c \leq m \times (2^{k+1})^2 \iff \frac{(2^{k+1})^2 + c}{(2^{k+1})^2} \leq m \iff m \geq 1 + \frac{c}{(2^{k+1})^2}$$

Since $k \geq 1$, $1 + c > 1 + \frac{c}{(2^{k+1})^2}$. So, if $m = 1 + c$, (i) is satisfied. Therefore, $T(2^{k+1}) = O((2^{k+1})^2)$. So, $T(2^{k+1}) = \Theta((2^{k+1})^2)$.

As the base case of the algorithm has $\Theta(n^2)$ running time, all cases are of size $2^k$ for some $k$, and if $T(2^k) = \Theta((2^k)^2)$ then $T(2^{k+1}) = \Theta((2^{k+1})^2)$, the running time of L-tile is $\Theta(n^2)$. 

6
2 Naive String Match Algorithm

a) Loop Invariant:
At the the start of the each iteration of the while loop in lines 3-4, the string $T[i...i+j-1]$ matches the string $P[0...j-1]$.

b) Proof by loop invariant.

Initialization:
At the start of the first iteration $j = 0$, $T[i...i-1]$ is empty and $P[0...-1]$ is empty. So, $T[i...i-1]$ matches $P[0...-1]$.

Maintenance:
Now, assume that the prior to the iteration where $j = n$ the string $T[i...i+n-1]$ matches the string $P[0...n-1]$. The loop will only progress when $P[n] = T[i+n]$. So, prior to the iteration where $j = n+1$ we have, $T[i...i+n]$ matches $P[0...n]$. Since $T[i...i+n] = T[i...i+(n+1)-1]$ and $P[0...n] = P[0...(n+1)-1]$ we have that: prior to the iteration where $j = n+1$, $P[0...i+(n+1)-1]$ matches $P[0...(n+1)-1]$. Since the loop invariant holds prior to the first iteration and when the loop invariant holds prior to an iteration it holds prior to the successive iteration, the loop invariant holds prior to every iteration.

Termination:
The loop terminates when either $j = m$ or $P[j] \neq T[i+j]$ for some $j$ less than $m$. For the case where $j = m$ by the loop invariant, we know that $P[0...m-1] = T[i...i+m-1]$. In this case the condition of the if statement on line 5 is satisfied and the index $i$, the beginning index of a matching subarray, is returned and the program works as advertised. In the case where $P[j] \neq T[i+j]$ for some $j$ less than $m$, it is clear that $P[0...m-1] \neq T[i...i+m-1]$. In this case $j \neq m$, and so the program attempts to make another iteration of the for loop which increments $i$ and checks for a matching subarray for array $P$ from one index further into array $T$.  
