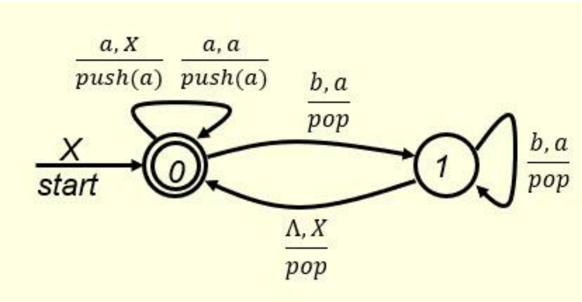
Solution to CS375 Sample Final Exam (100 points) (Spring 2024)

Closed book & closed notes April 20, 2024

Name________Sample____________

1. (5 points)
Final-state acceptance and empty-stack acceptance are equivalent only for NPDA's. They are not equivalent for DPDA's. For DPDA's the class of languages defined by final-state acceptance is bigger. Given the following final-state DPDA



and the following strings

Λ, ab, aa, aabb

which of these strings are accepted by the given final-state DPDA? Put your answer in the following blank.

$\boxed{\text{Λ, ab, aa, aabb}}$ (2 points)

If the given final-state DPDA is considered as an empty-stack DPDA (state 0 is no longer a final state), then which of the given strings are accepted by the empty-stack DPDA? Put your answer in the following blank.

$\boxed{\text{ab, aabb}}$ (1 point)

Now, consider the following two general questions. First, what is the language L_1 accepted by the given final-state DPDA? Put your answer in the following blank.

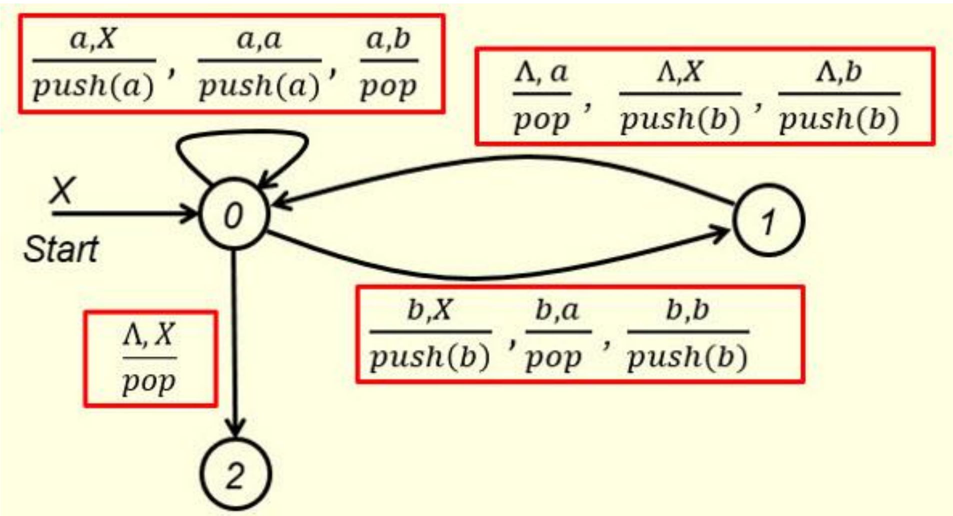
$$L_1 = \boxed{\{a^n, a^m b^m \mid n \in N; m \in N^+\}}$$ (1 points)

Second, what is the language L_2 accepted by this DPDA when viewed as an empty-stack DPDA? Put your answer in the following blank.

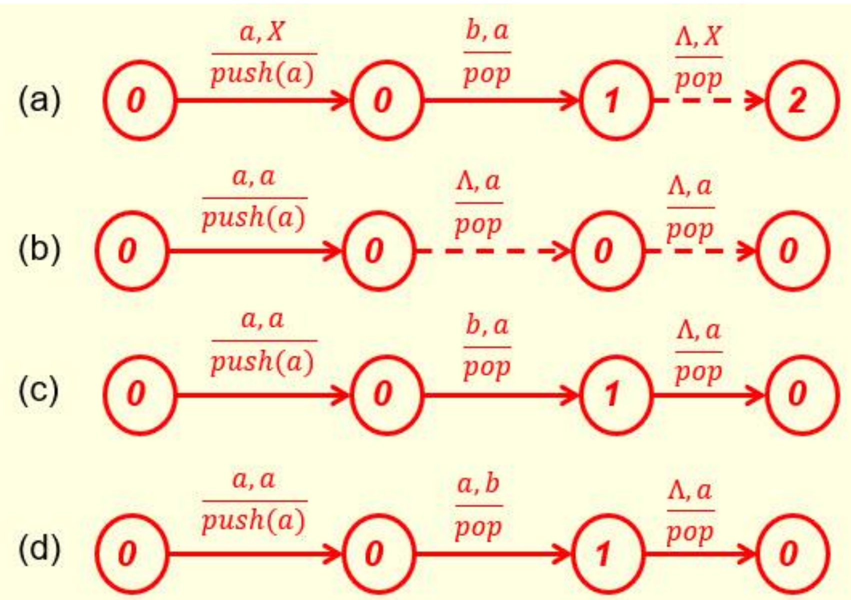
$$L_2 = \boxed{\{a^m b^m \mid m \in N^+\}}$$ (1 points)

L_1 obviously is bigger than L_2.

2. (4 points)
The following empty-stack PDA accepts the language L = { w ϵ {a, b}* | n_a(w) = 2n_b(w) } (assuming $\Lambda \in L$).



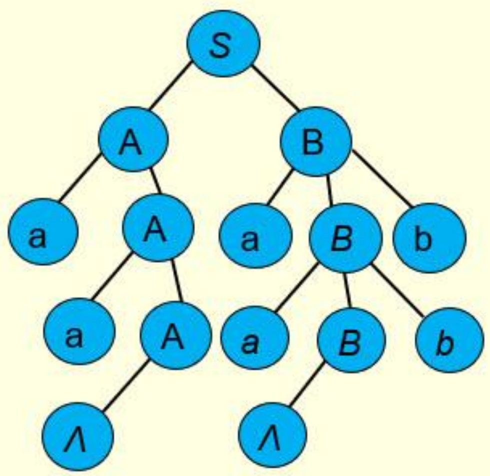
For this PDA, tell which one(s) of the following four possible type 3 instructions are legitimate type 3 for this empty-stack PDA?



Put your answer in the following blank.

(a), (c)

3. (6 points)
Given the following parse tree where S, A, B are non-terminals, a and b are terminals and Λ is the empty string, show the corresponding left-most derivation of the yield in the blanks on the right side. (6 points)



S ⇒ AB
⇒ aAB
⇒ aaAB
⇒ aaΛB
⇒ aaΛaBb
⇒ aaΛaaBbb
⇒ aaΛaaΛbb
⇒ aaaabb (4 pts)

Does the derivation show the grammar is an LL(1) grammar?

[ ] Yes [X] No (1 point)

Does the derivation show the grammar is an LL(2) grammar?

[ ] Yes [X] No (1 point)

4. (6 points)
The following grammar is a C-F grammar for $\{a^{m+n}b^m c^n \mid m,n \in N\}$

S → aSc | B B → aBb | Λ

If Λ, ac, ab, aabc, aaabbc and aaabcc are considered, which one(s) do not satisfy the LL(1) requirement? Put the one(s) which do not satisfy the LL(1) requirement in the following box. (6 points)

$\boxed{\text{ab, aabc, aaabbc, aaabcc}}$

5. (4 points)
   The following given grammar is a left recursive grammar

   S → Scb | c

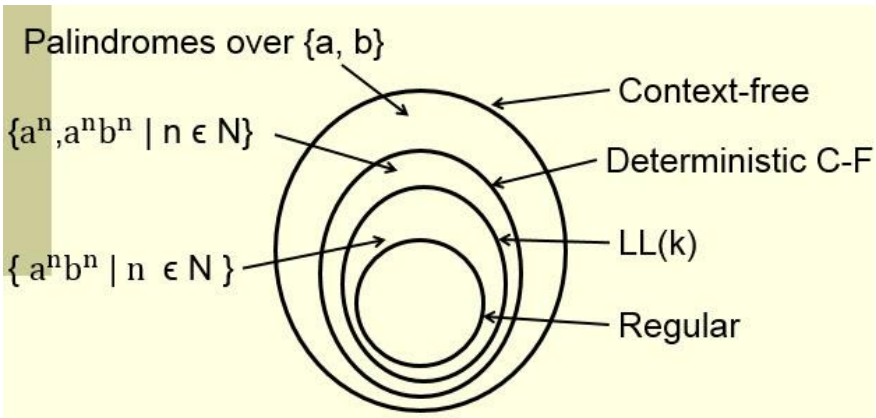
   This left recursive grammar can be transformed to a right recursive grammar as follows:

   S → $\boxed{\text{cB}}$

   B → $\boxed{\text{cbB}}$ $\boxed{\Lambda}$

   This right recursive grammar is an LL($\boxed{1}$) grammar.

6. (7 points)
   In slide 43 of the notes "Context-free Languages and Pushdown Automata IV", it is shown that the set of LL(k) languages is a proper subset of the set of deterministic C-F languages (see the following figure). In particular, it points out that the language $\{a^n, a^n b^n \mid n \in N\}$ is a deterministic C-F, but not LL(k) for any k.



To show the language is not LL(k) for any k, note that the grammar for this language is

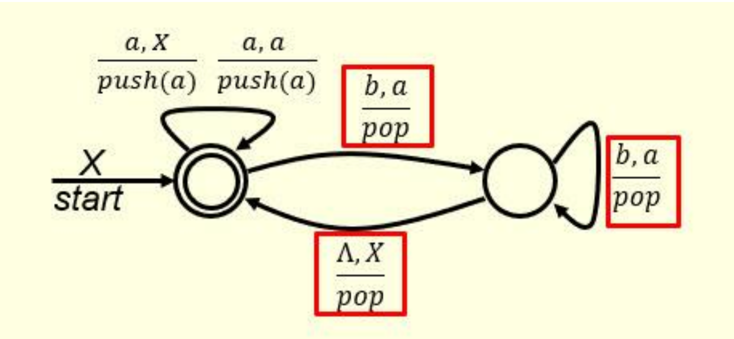
$S \to A \mid B \quad A \to \boxed{aA} \mid \boxed{\Lambda} \quad B \to \boxed{aBb} \mid \boxed{\Lambda}$

or

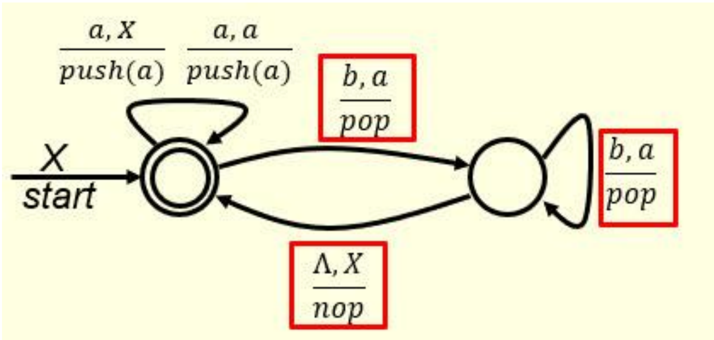
$S \to A \mid B \quad A \to \boxed{aAb} \mid \boxed{\Lambda} \quad B \to \boxed{aB} \mid \boxed{\Lambda}$ (4 points)

(you only need to answer one case here, either one). The language contains Λ as an element. Now consider the case k = 1 and consider the input string ab. When the first symbol is scanned, we get an ‘a’. This information alone is not enough for us to make a proper choice. So we don’t even know what to do with the first step in the parsing process. For k = 2, if we consider the input string aabb, we face the same problem. For any k > 2, the input string $a^k b^k$ would cause exactly the same problem. So this grammar is not LL(k) for any k.

On the other hand, by putting proper instructions into the blanks in the following figure, we get a deterministic final-state PDA that accepts the language $\{a^n, a^n b^n \mid n \in N\}$.



or



(3 points)

(again, you only need to answer one case here, either one). Hence, this language is indeed deterministic C-F, but not LL(k) for any k.

7. (6 points)
Use left-factoring to find an equivalent LL(k) grammar for the following grammar with k being as small as possible. In this case, left-factoring will create a production that is always used as the first step in the left-most derivation and the common factor contained on the right side of the production will provide an automatic match on the first part of the input string. (6 points; 1 point for each of the six blanks)

S → abA | abcS   A → aA | Λ

The language generated by the given grammar is:

$$\boxed{\{(abc)^m aba^n \mid m, n \in N\}}$$

The given grammar is LL(3).

By factoring ab out from S → abA | abcS, the given grammar can be converted to

$\boxed{S \rightarrow abT}$ $\boxed{T \rightarrow A \mid cS}$ $\boxed{A \rightarrow aA \mid \Lambda}$ (1)

This grammar can also be written as

$\boxed{S \rightarrow abT}$ $\boxed{T \rightarrow aA \mid cS \mid \Lambda}$ $\boxed{A \rightarrow aA \mid \Lambda}$ (2)

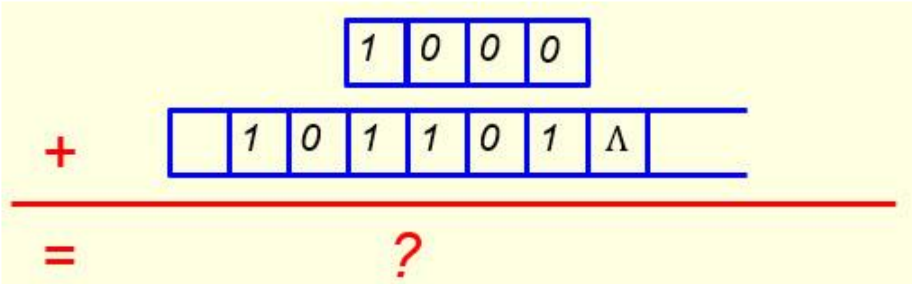
Grammars (1) and (2) are both LL(1).

8. (8 points)
Fill out the following blanks for the instructions of a Turing machine that moves an input string over {a, b} to the right one cell position. The tape head initially is at the left end of the input string. The machine will move the entire string to the right one cell position and leave all remaining tape cells blank. The tape head ends at the right end of the output string. (8 points)

(0, a, Λ, R, 1)	found an a
(0, b, Λ, R, 2)	found a b
(0, Λ, Λ, S, halt)	Done
(1, a, a, R, 1)	found an a & to write an a
(1, b, a, R, 2)	found a b & to write an a
(1, Λ, a, S, halt)	Done
(2, a, b, R, 1)	found an a & to write a b
(2, b, b, R, 2)	found a b & to write a b
(2, Λ, b, S, halt)	Done

9. (9 points)
Given an integer say 45, to find the sum of 45 with 8 in binary form (see the figure below), we can use the TM designed in slides 79-90 of the notes “Turing Machines and Equivalent Models I” twice to find the result. First, we use that TM to find the sum of the given number

with 4, and then use that TM again to find the sum of the first sum with 4 again. A more effective way is to design a TM to do the addition with 8 directly.



Such a TM can be designed by extending the TM designed in slides 79-90 of the notes “Turing Machines and Equivalent Models I”. The TM has 14 instructions and 6 states: 0, 1, 2, 3, 4, and halt. Five instructions of such a TM have been given in the first table below. Fill out the remaining blanks of the first table and also blanks in the second and third tables to show the remaining instructions of the TM

Move three cells left
(0, 0, 0, L, 1)
(0, 1, 1, L, 1)
(1, 0, 0, L, 2)
(1, 1, 1, L, 2)
(1, Λ, 0, L, 2)
(2, 0, 0, L, 3)
(2, 1, 1, L, 3)
(2, Λ, 0, L, 3)

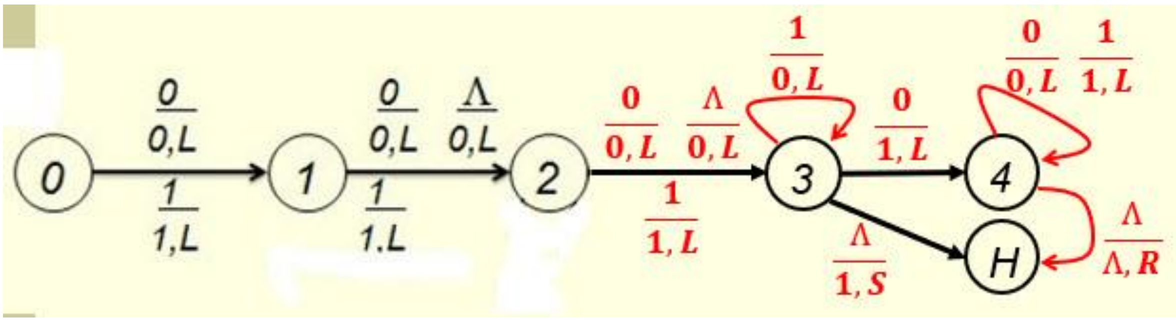
Add 1:	
(3, 0, 1, L, 4)	Move left
(3, 1, 0, L, 3)	Carry
(3, Λ, 1, S, halt)	Done

Find left end of the string:	
(4, 0, 0, L, 4)	
(4, 1, 1, L, 4)	
(4, Λ, Λ, R, halt)	Done

+

and then fill out the blanks in the following chart to make it a complete state transition diagram for this TM. In the diagram, state H means the halt state.

Note that there are several ways to define the remaining instructions of the TM, but make sure the instructions you choose fit into the following diagram naturally and logically.



10. (12 points)

Given a non-empty string, we can move the string two units to the left using three different approaches. The first approach is to use the TM introduced in slides 44-51 of the notes "Turing Machines and Equivalent Models-I" twice; the second approach is to move each letter of the string two units to the left directly; the third approach is to use a stack to assist the moving process.

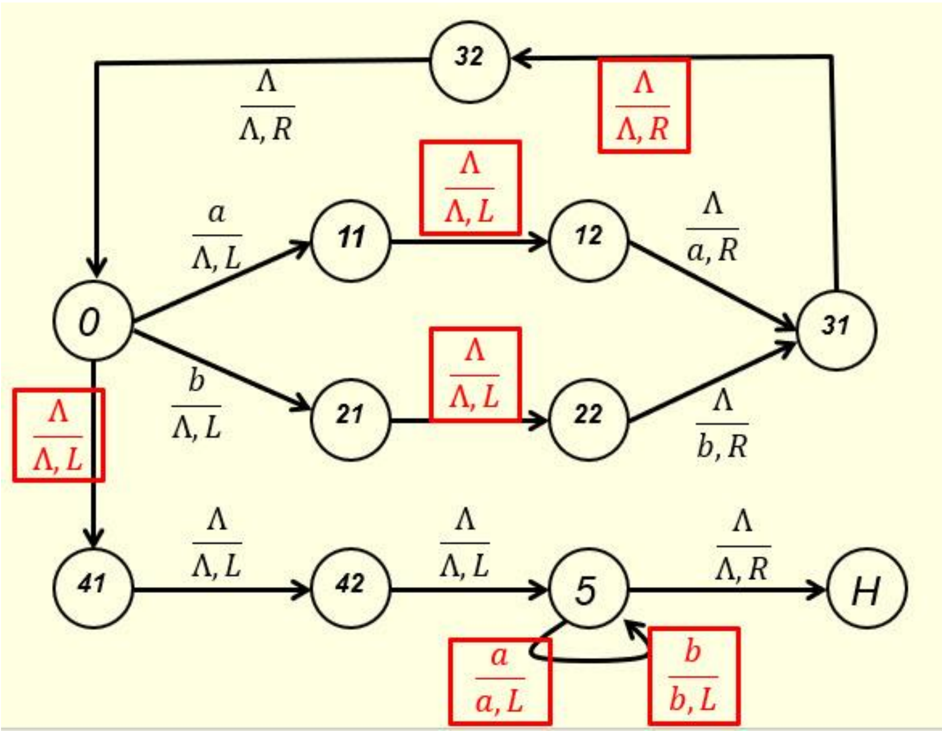
A TM that can move the string two units to the left directly requires 14 instructions and 11 states. In the following, eight instructions for such a TM are given in the tables. Fill out the remaining blanks to make the resulting instruction set the instruction set for such a TM and then fill out the blanks in the next chart to make it a complete state transition diagram for this TM. In the diagram, state H means the halt state.

Find a or b to move:	
(0, a, Λ, L, 11)	found a
(0, b, Λ, L, 21)	found b
(0, Λ, Λ, L, 41)	no more a's or b's

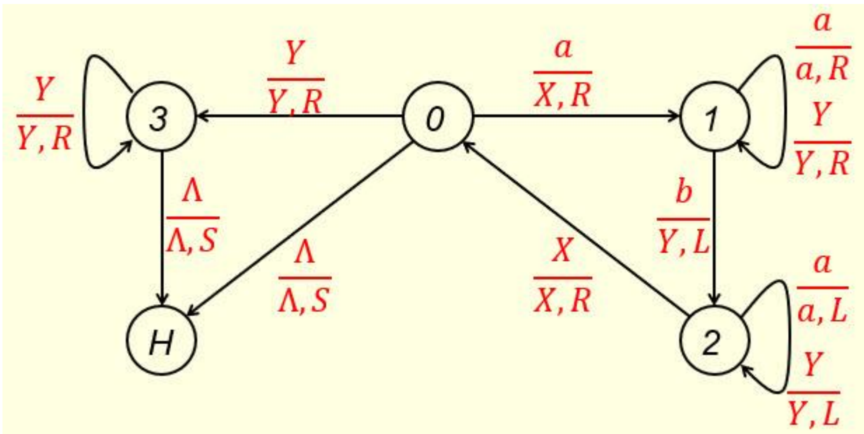
Move to left end of output:	
(41, Λ, Λ, L, 42)	skip Λ
(42, Λ, Λ, L, 5)	skip Λ
(5, a, a, L, 5)	skip a
(5, b, b, L, 5)	skip b
(5, Λ, Λ, R, halt)	Done

Write a or b:	
(11, Λ, Λ, L, 12)	skip Λ
(12, Λ, a, R, 31)	write a
(21, Λ, Λ, L, 22)	skip Λ
(22, Λ, b, R, 31)	write b
(31, Λ, Λ, R, 32)	skip Λ
(32, Λ, Λ, R, 0)	skip Λ

Note that there are several ways to define the remaining instructions of the TM, but make sure the instructions you choose fit into the following diagram naturally and logically.

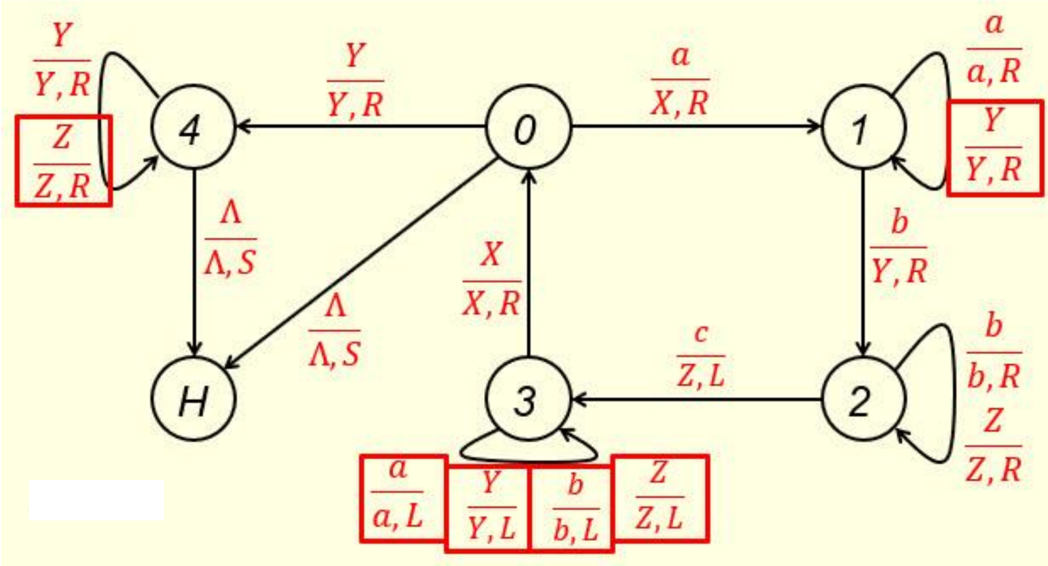


11. (6 points)
We know how to design a TM to accept the language $\{a^n b^n \mid n \in N\}$. The state transition diagram of this TM is shown below. This TM has 8 instructions and 5 states: 0, 1, 2, 3 and halt. It uses an implicit stack to match the number of a's in the string with the number of b's in the string.



One can extends the concept of this TM to design a new TM to accept the language $\{a^n b^n c^n \mid n \in N\}$. The new TM has 14 distinct instructions and 6 states: 0, 1, 2, 3, 4, and halt. It uses two implicit stacks to match the number of a's in the string with the number of b's and the number of c's in the string.

Fill out the blanks in the following diagram to make it a complete state transition diagram for the new TM.



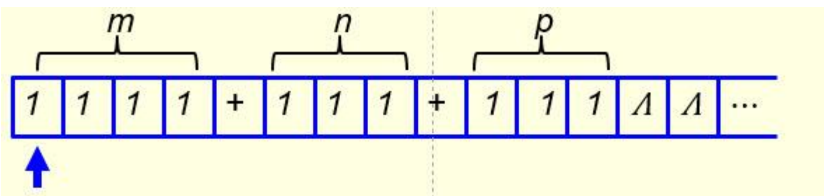
12. (2 points)

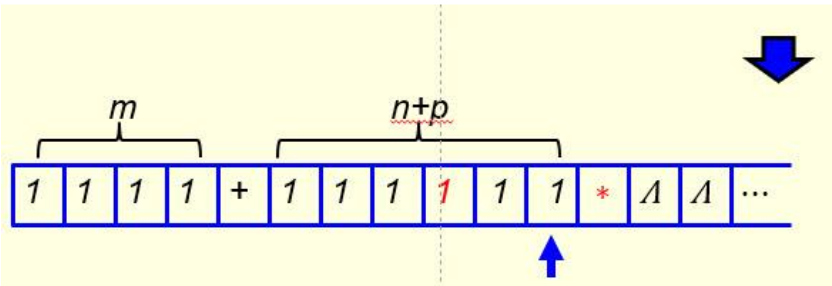
The 'P versus NP' problem is a major unsolved problem in computer science. It is an important problem because if we can prove that P = NP (i.e., all problems can be solved in polynomial time) then

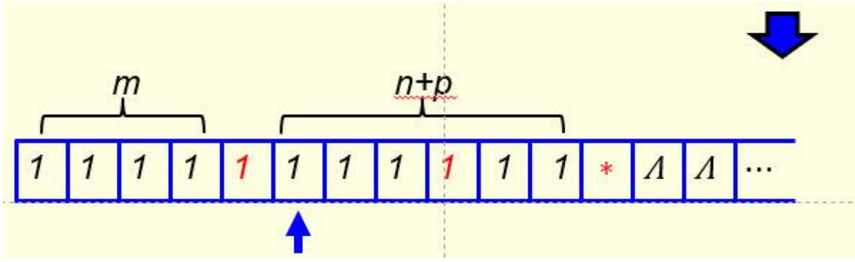
there will be no need to build quantum computers because standard computers would be fast enough (for all questions), and easier to build.

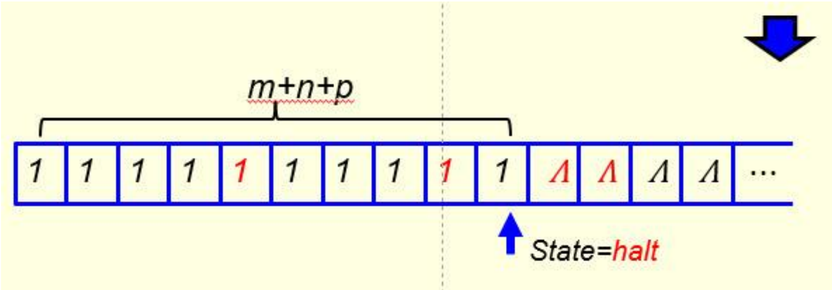
13. (3 points)

To build a TM to perform addition on three non-zero positive numbers *m, n,* and *p* in unary form (see the first figure below for the case *m=4*, *n=3* and *p=3*), a better approach is to perform the addition *(n+p)* first, and then perform *m+(n+p)*. To perform the addition *(n+p)* first, on its way moving right, the TM will ignore the first '+' sign (not change it to '1'), only change the second '+' to '1', then look for a 'Λ'. When a 'Λ' is reached, the TM keeps that 'Λ', turns left and changes the '1' in the next cell to '∗' (instead of '1') and then turns left (see the second figure below). To perform *m+(n+p)*, the TM then moves left to find the first '+', changes it to '1' and turns right (see the third figure below). It then moves right to find '∗'. Once '∗' is reached, the TM converts '∗' to 'Λ', turns left, changes the '1' in the next cell to 'Λ', moves one unit to the left and stop.

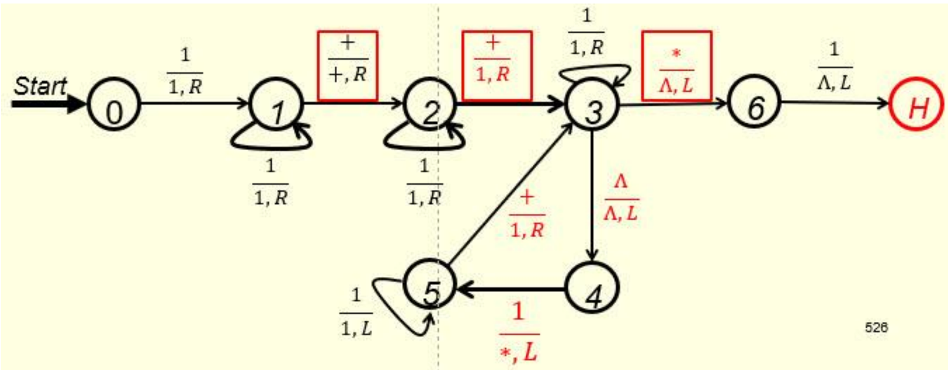








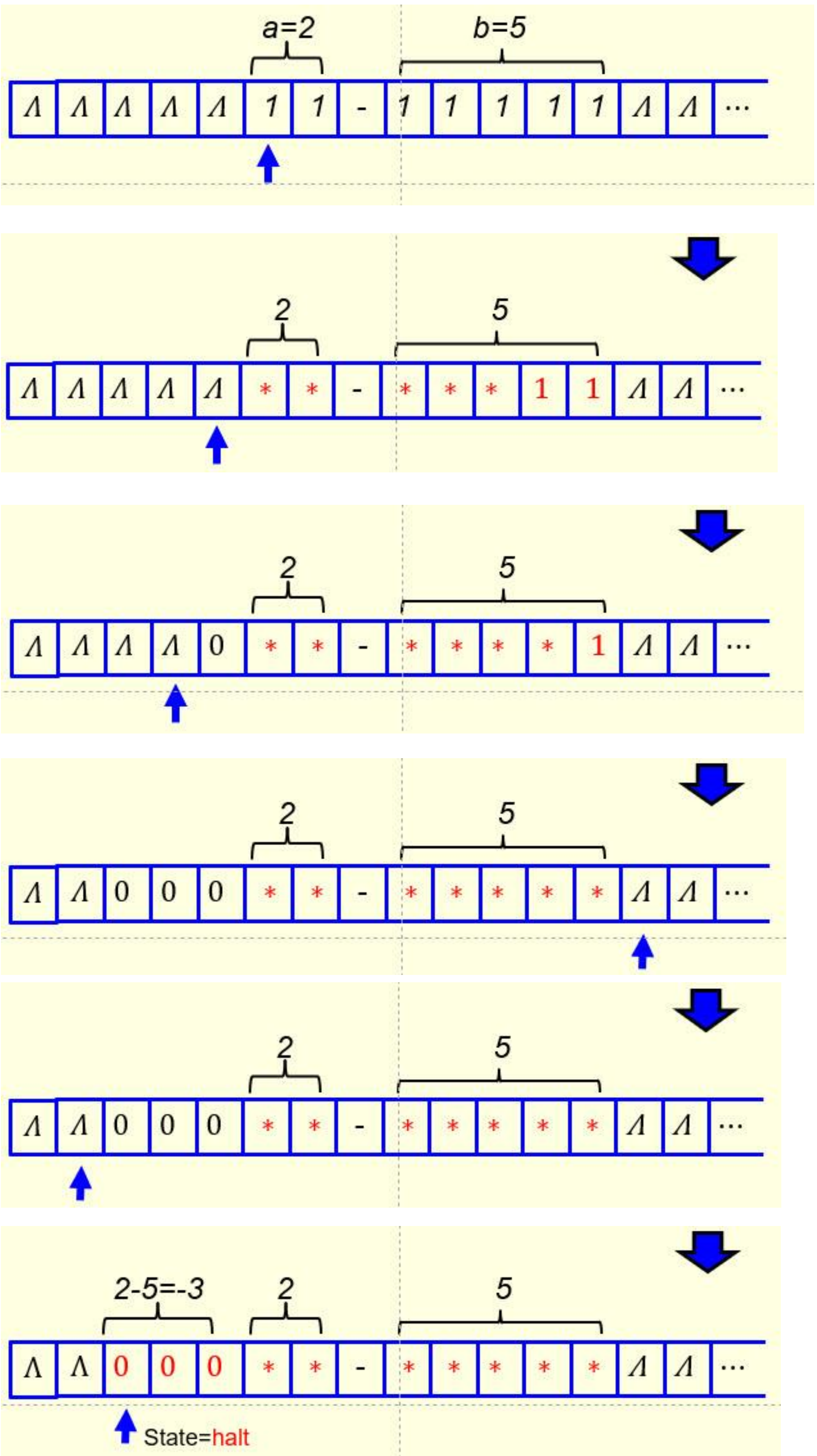
Your task here is to fill out the three blanks in the following figure to make it a TM that can perform addition on three given non-zero positive numbers in unary form directly.



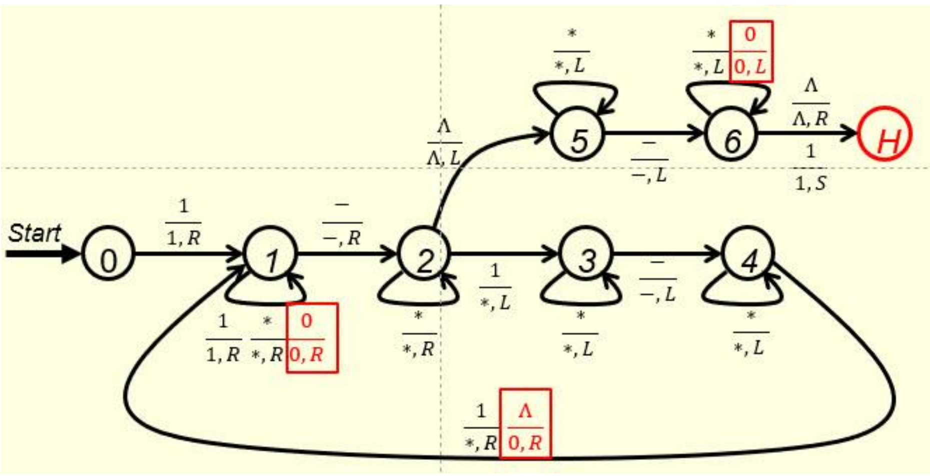
14. (3 points)

The TM that can perform the subtraction function $f(a-b)=c$ on two unary numbers *a* and *b* when *a* is bigger than or equal to *b* in notes "Turing Machines and Equivalent Models-II" can be modified to cover the case when *b* is bigger than *a* as well. Consider the following input string with a=2 and b=5. After two 1's have been converted to '$*$' in both *a* and *b*, when the third 1 in *b* is converted to '$*$', we don't have a 1 in *a* to convert, instead, we find a 'Λ' (see the second figure below). We change that 'Λ' to '0' and turn right to find another 1 in *b* to convert to '$*$'. Again, there is no 1 in *a* to match this '$*$, but a '0'. We skip this '0' to reach a 'Λ' on the left-hand side (see the third figure below). We convert this 'Λ' to 0 and turn right to find another 1 in *b* to convert to '$*$'. We repeat the same process again, get one more 0 on the *a* side (we have three 0's now) and turn right to find another

1 in *b* to convert. We don't find any 1, but a 'Λ' (see the fourth figure below). That is, no more 1's in *b* to convert. So we turn left to find the left end of the 0 string to stop. This is done by moving left to find a 'Λ' (see figure five below) and then turn right, move one cell to the right and stop (see figure six below).

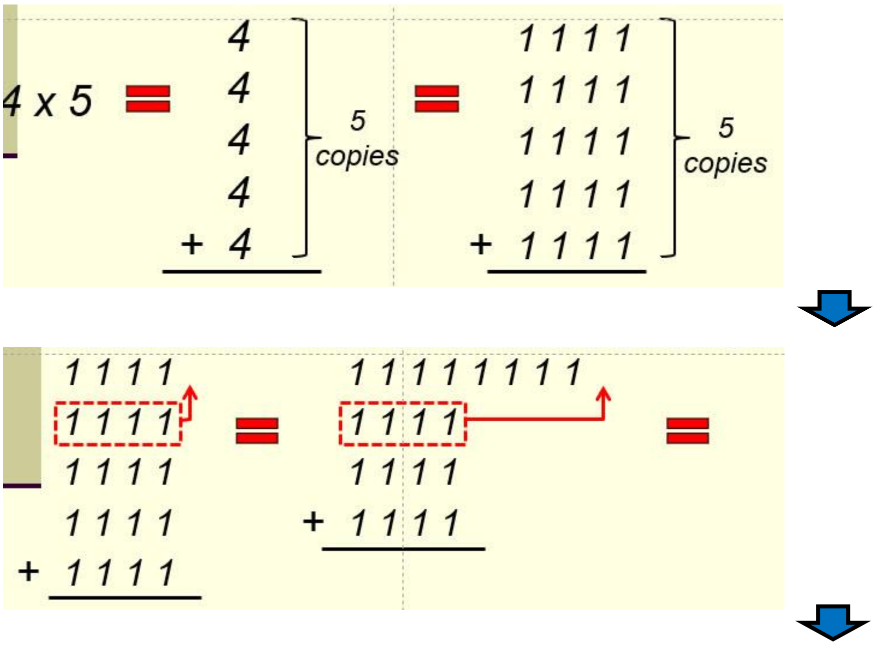


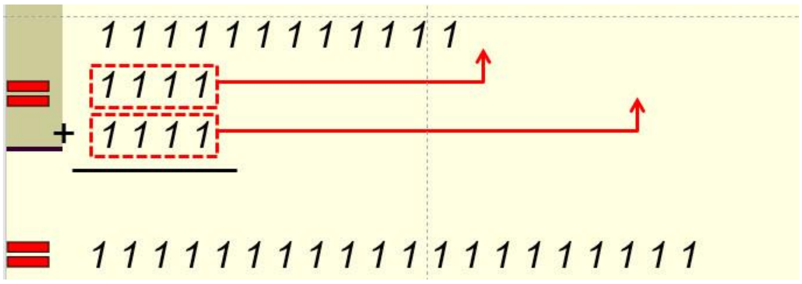
Your task here is to fill out the three blanks in the following figure to make it a TM that can subtract a bigger number from a smaller number as well.



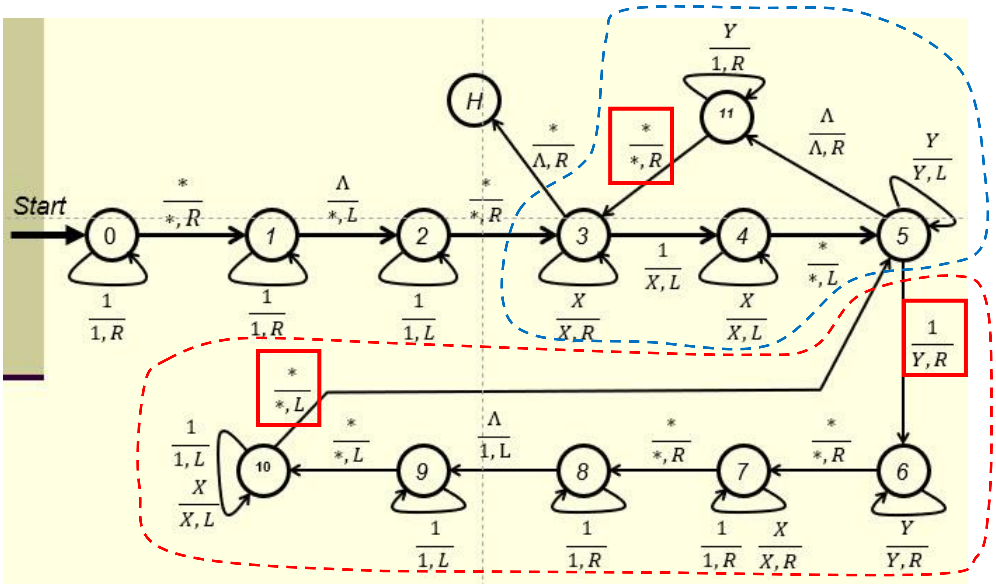
15. (6 points)

A TM that can perform *multiplication* on two positive unary numbers is developed based on the concept that "multiplication is extended addition". For instance, 4×5 can be viewed as the addition of five 4's in unary form (see the first figure below). The process is to repeatedly perform addition on these five 4's two at a time (in unary form; see the second and the third figures below) until four additions have been performed.



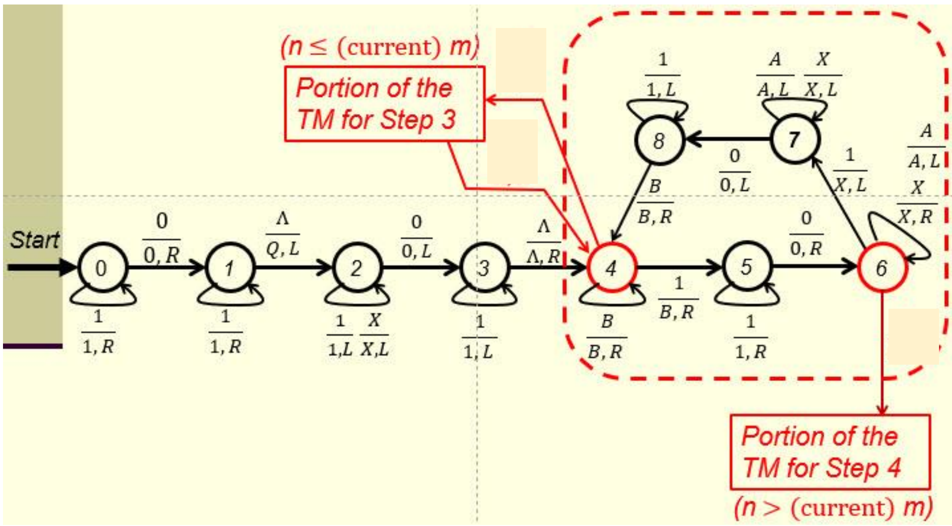


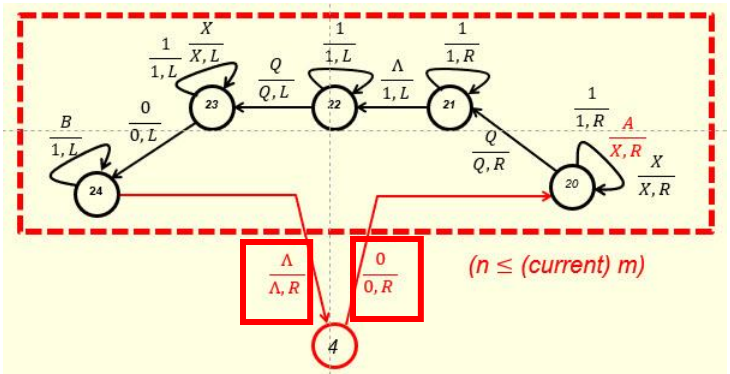
Your task here is to fill out the three blanks in the following TM that does multiplication of two given numbers *m* and *n* in unary form. Note that the portion circled by the red dotted curve is to perform the addition job (putting a copy of *m* 1's at the end of n) and the portion circled by the blue dotted curve is the portion that does the counting (making sure n copies of m in unary form are put at the end of *n*).

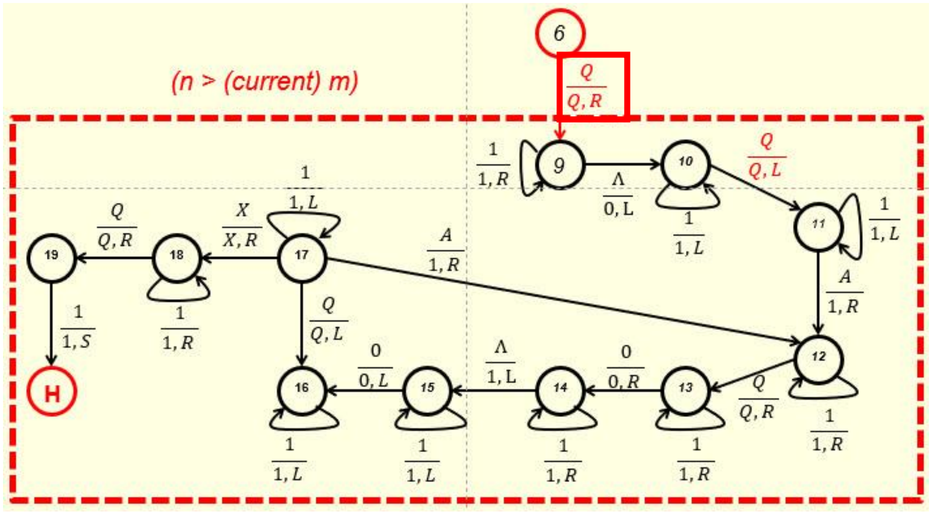


16. (6 points)

A TM that can perform *division* on two positive unary numbers is developed based on the concept that "division is extended subtraction". That concept and the implementation steps have been clearly described in the notes "Turing Machines and Equivalent Models-II''. The main body of this TM is shown in the first figure below with the portions that perform Step 3 and Step 4 are shown in the second and the third figures separately. Your tasks here is to fill out the blanks in the second and the third figures so that these two portions can be connected to the proper nodes of the main body of the TM correctly.







17. (4 points)

The Church-Turing Thesis has two versions. The following is the second version:

Anything that is intuitively computable can be computed by a Turing machine.

The first version is shown below. Fill out the blue blank in the following box to make it a complete statement.

A problem can be solved by an Algorithm if and only if it can be solved by a Turing machine. (1 point)

The first version is an if and only if statement and the second version is not. Does this mean the other direction of the second version ('Anything that can be computed by a Turing machine is intuitively computable') is not true?

YES ☐ NO X (1 point)

Justify your answer in the following text box.

The other direction is obviously true because a Turing machine is a step-by-step process, i.e, an algorithm. Therefore, anything that is computable by a Turing machine is intuitively computable (algorithmically solvable).

(2 points)

18. (3 points)

Church-Turing Thesis is not a theorem, but a thesis. Why? Put your answer in the following test box.

Even though most of us believe the thesis is true, but we cannot prove it. Nor can we disprove it.