1. **Finite automata and regular languages:**

1a. Design a finite automaton for recognizing binary sequences that have exactly two zeros. Assume that the input strings contain only symbols 0 and 1. The easiest is to have 4 states: no-zeros, 1-zero, 2-zeros, and more-than-2-zeros, you just need to describe transitions between these states, and which states are final. Show, step-by-step, how your automaton will accept the string 01101.

1b. On the example of this automaton, show how the word 01101 can be represented as xyz in accordance with the pumping lemma.

1c. Use a general algorithm to describe a regular expression corresponding to the finite automaton from the Problem 1a.

1d-e. The resulting language can also be described by a regular expression $1^*01^*01^*$. Use a general algorithm to transform this regular expression into a finite automaton: first a non-deterministic one, then a deterministic one.
1. **Finite automata and regular languages:**

1a. Design a finite automaton for recognizing binary sequences that have exactly two zeros. Assume that the input strings contain only symbols 0 and 1. The easiest is to have 4 states: no-zeros, 1-zero, 2-zeros, and more-than-2-zeros, you just need to describe transitions between these states, and which states are final. Show, step-by-step, how your automaton will accept the string 01101.

1b. On the example of this automaton, show how the word 01101 can be represented as xyz in accordance with the pumping lemma.

1c. Use a general algorithm to describe a regular expression corresponding to the finite automaton from the Problem 1a.

1d-e. The resulting language can also be described by a regular expression $1^*01^*01^*$. Use a general algorithm to transform this regular expression into a finite automaton: first a non-deterministic one, then a deterministic one.

![Diagram of automaton](image-url)
\( R_{s1} = \emptyset U \lambda^*0 \)
\( = 1^*0 \)
\( R_{s2} = \emptyset U \lambda^*0 \)
\( = \emptyset \)
\( R_{s3} = \emptyset U \lambda^*0 \)
\( = \emptyset \)
\( R_{sf} = \emptyset U \lambda^*0 \)
\( = \emptyset \)
\( R_{s1} = 1 U \emptyset \)
\( = 1 \)
\( R_{s2} = 0 U \emptyset \)
\( = 0 \)
\( R_{sm} = \emptyset U (1^*0)^* \emptyset \)
\( = \emptyset \)
\( R_{s2} = 0 U (1^*0)^* \emptyset \)
\( = 1^*0^*1^*0 \)
\( R_{sm} = \emptyset U (1^*0)^* \emptyset \)
\( = \emptyset \)
\( R_{sf} = \emptyset U (1^*0)^* \emptyset \)
\( = 1^*0^*1^*0 \)
\( R_{mm} = (01^*) U \emptyset \)
\( = 01^* \)
\( R_{mf} = \emptyset U \emptyset \)
\( = \emptyset \)
\( R_{sf} = 1^*0^*1^*0 U 1^*0^*1^*0 (01^*) \emptyset \)
\( = 1^*0^*1^*0 \)
Non-Deterministic

Deterministic
2. Beyond finite automata: pushdown automata and context-free grammars:

2a. Prove that the language consisting of all expressions that contain twice as many a's as b's is not regular.

2b. Use a general algorithm to transform the finite automaton from the Problem 1a into a context-free grammar (CFG). Show, step-by-step, how this CFG will generate the word 01101.

2c. For the context-free grammar from the Problem 2b, show how the word 01101 can be represented as uvxz in accordance with the pumping lemma.

2d. Use a general algorithm to translate the CFG from 2b into Chomsky normal form.

2e. Use a general algorithm to translate the CFG from 2b into an appropriate push-down automaton. Explain, step-by-step, how this automaton will accept the word 01101.

2f. Use the general stack-based algorithms to show:

- how the compiler will transform the expression \((2 - 3) \times (1 + 2)\) into inverse Polish notation, and
- how it will compute the value of this expression.

2a) \[ L = \{ a^n b^n \mid n \geq 1 \} \]

Let's assume that \( L \) is regular then by pumping lemma there exists some \( p \) such that for every word \( s \in L \) with length \( \geq p \) can be represented as \( s = xyz \) where \( \text{len}(y) \geq 1 \), \( \text{len}(xy) \leq p \) and for every \( i \geq 0 \), \( xyz^i \in L \).

Let's take \( s = a^p b^p \), hence \( \text{len}(s) = 2p \), so by pumping lemma there exists some corresponding \( xyz \) since \( s = xyz \) and \( \text{len}(xy) \leq p \) this means that \( x \) and \( y \) are in \( a^p \) so when we take \( xyz \), we add \( a^p \) but the number of b's does not change so the number of a's in \( xyz \) is no longer twice the number of b's so \( xy^2 \notin L \), but by pumping lemma \( xy^2 z \in L \) a contradiction. This contradiction shows that our assumption is wrong so \( L \) is not regular.
Adding more $a$'s to the string by pumping lemma shows that now the new $s$ will not be in the language and so we get a contradiction.

Therefore the language $L$ is not regular.

2.6) 

$Q_1 \rightarrow 1Q_1$
$Q_1 \rightarrow 0Q_2$
$Q_2 \rightarrow 1Q_2$
$Q_2 \rightarrow 0Q_3$
$Q_3 \rightarrow 1Q_3$
$Q_3 \rightarrow 0Q_4$
$Q_3 \rightarrow \epsilon$

$Q_1 \rightarrow 0Q_2 \rightarrow 01Q_2 \rightarrow 011Q_2 \rightarrow 0110Q_3 \rightarrow 01101Q_3 \rightarrow 01101$

2.2) 

$UVXVY = 01101$

accepted

file:///Q:/cs3350.17/final.html
2d) \[ s_0 \rightarrow q_1 \]
\[ q_1 \rightarrow 0 q_2 \]
\[ q_2 \rightarrow 1 q_2 \]
\[ q_3 \rightarrow 0 q_3 \]
\[ q_3 \rightarrow 0 q_4 \]
\[ q_4 \rightarrow \in \]

**Step 2**

\[ q_2 \rightarrow 0 \]
\[ q_3 \rightarrow 1 \]

**Step 3**

\[ s_0 \rightarrow 1 q_1 \]
\[ s_0 \rightarrow 0 q_2 \]

**Step 4**

\[ u_0 \rightarrow 0 \]
\[ u_1 \rightarrow 1 \]
\[ q_1 \rightarrow u_1 q_1 \]
\[ q_1 \rightarrow u_0 q_2 \]
\[ q_2 \rightarrow u_1 q_2 \]
\[ q_2 \rightarrow u_0 q_3 \]
\[ q_3 \rightarrow u_1 q_3 \]
\[ q_3 \rightarrow u_0 q_4 \]
\[ s_0 \rightarrow u_1 q_1 \]
\[ s_0 \rightarrow u_0 q_2 \]
26) \[ \begin{align*}
Q_0 & \rightarrow 1Q_0 \\
Q_0 & \rightarrow 0Q_1 \\
Q_1 & \rightarrow 1Q_1 \\
Q_1 & \rightarrow 0Q_2 \\
Q_2 & \rightarrow 1Q_2 \\
Q_2 & \rightarrow 0Q_m \\
Q_m & \rightarrow 0Q_m \\
Q_m & \rightarrow 1Q_m \\
Q_2 & \rightarrow \varepsilon
\end{align*} \]

2c) 

\[ \begin{align*}
Q_0 & \rightarrow 1Q_0 \\
Q_0 & \rightarrow 0Q_1 \\
Q_1 & \rightarrow 1Q_1 \\
Q_1 & \rightarrow 1Q_1 \\
Q_2 & \rightarrow 0Q_2 \\
Q_2 & \rightarrow 0Q_2 \\
Q_0 & \rightarrow 0Q_0 \\
Q_0 & \rightarrow 0Q_0 \\
Q_1 & \rightarrow 0Q_1 \\
Q_1 & \rightarrow 0Q_1 \\
Q_2 & \rightarrow V_1Q_2 \\
Q_2 & \rightarrow V_1Q_2 \\
Q_m & \rightarrow V_1Q_m \\
Q_m & \rightarrow V_0Q_m \\
Q_0 & \rightarrow V_0Q_0 \\
Q_0 & \rightarrow V_0Q_1 \\
Q_1 & \rightarrow V_0Q_1 \\
Q_1 & \rightarrow V_0Q_1 \\
Q_2 & \rightarrow V_1Q_2 \\
Q_2 & \rightarrow V_1Q_2 \\
Q_2 & \rightarrow V_1Q_2 \\
Q_2 & \rightarrow V_0Q_2 \\
Q_2 & \rightarrow V_0Q_2 \\
Q_2 & \rightarrow V_0Q_m
\end{align*} \]
2e) \[ Q_1 \rightarrow 1Q_1 \]
\[ Q_1 \rightarrow 0Q_2 \]
\[ Q_2 \rightarrow 1Q_2 \]
\[ Q_2 \rightarrow 0Q_3 \]
\[ Q_3 \rightarrow 1Q_3 \]
\[ Q_3 \rightarrow 0Q_4 \]
\[ Q_3 \rightarrow \epsilon \]

2f) \[(2-3)*(1+2)\]

\[ 2 \ 3 \ - \ 1 \ 2 \ + \ * \]

answer: \[-3\]
2f) \[
\begin{array}{c}
(2-3) \times (1+2) \\
2 \ 3 - 1 2 \\
4 \ 5 \ 6
\end{array}
\]

```
2 3 -1 1 2 + 3 * -3
```
3. Beyond pushdown automata: Turing machines

3a. Prove that there exists a language that is not context-free and therefore, cannot be recognized by a pushdown automaton. You can use the same language we had in class or -- for extra credit -- some other language.

3b-c. Use a general algorithm to design a Turing machine that accepts exactly all sequences accepted by a finite automaton from Problem 1a. Show, step-by-step, how this Turing machine will accept the word 01101. Describe, for each step, how the state of the tape can be represented in terms of states of two stacks.

3d. Design a Turing machine for computing \( n + 2 \) in binary code. Trace it for the binary number \( n = 10_2 \) (which is 2 decimal); the result of the computation should be \( 2 + 2 = 4_{10} \), i.e., \( 100_2 \).

\[
\frac{52}{90} = \frac{13}{20}
\]
3. Beyond pushdown automata: Turing machines

3a. Prove that there exists a language that is not context-free and therefore, cannot be recognized by a pushdown automaton. You can use the same language we had in class or -- for extra credit -- some other language.

3b-c. Use a general algorithm to design a Turing machine that accepts exactly all sequences accepted by a finite automaton from Problem 1a. Show, step-by-step, how this Turing machine will accept the word 01101. Describe, for each step, how the state of the tape can be represented in terms of states of two stacks.

3d. Design a Turing machine for computing $n + 2$ in binary code. Trace it for the binary number $n = 10_2$ (which is 2 decimal); the result of the computation should be $2 + 2 = 4_{10}$, i.e., 100_2.

3a) Proof by contradiction

Let’s assume that $L$ can be recognized by $CFG$. This means there exists a $p$ in all words of the language $L$ with $\text{len}(w) \geq p$ and $\exists u,v,x,y,z$ with $w = uv^xy^iz$ and $\text{len}(xy) > 0$ and $\text{len}(vx) \leq p$ and $\forall i$ in $uv^ix^iz$ is still in the language.

Take $w = a^{3p}b^{2p}c^p$.

We get a contradiction if the repetition $vy$ is in the single letters a's, b's, or c's because the word will not be in the language if we pump more of that single letter, because we will have too much of that single letter, and it won’t be a part of the language.

We get a contradiction if the repetition is in between the a's and b's because if we pump a's and b's, there will be too little c's so the new word won’t be in the language.

We get a contradiction if the repetition $vy$ is in between the b's and the c's because pumping b's and c's will make it not part of the language since it will have too many b's and c's in comparison to a's.
We get a contradiction because by pumping lemma, we will get a word that is not part of the language; therefore \( L \) cannot be recognized by a pushdown automata.

\( 3b - c \)
3b) $s_{start} \rightarrow R_1 Q_0$

$Q_0 \rightarrow \text{reject}$

$Q_0 1 \rightarrow R$

$Q_0 0 \rightarrow R_1 Q_4$

$Q_1 \rightarrow \text{reject}$

$Q_1 1 \rightarrow R$

$Q_1 0 \rightarrow R_1 Q_2$

$Q_2 \rightarrow \text{accept}$

$Q_2 1 \rightarrow R$

$Q_2 0 \rightarrow R_1 Q_m$

$Q_m \rightarrow \text{reject}$

$Q_m 1 \rightarrow R$

$Q_m 0 \rightarrow R$

$Q_{accept} \rightarrow \text{accept}$
3d) \[ n + 2 \]

\[
\begin{align*}
\text{start, #} & \rightarrow \text{find } 2, R \\
\text{find } 2, 1 & \rightarrow \text{add } 2, R \\
\text{find } 2, 0 & \rightarrow \text{add } 2, R \\
\text{find } 2, \# & \rightarrow 0, \text{add } 2, R \\
\text{add } 2, 0 & \rightarrow \text{rewind, } 1, L \\
\text{add } 2, \# & \rightarrow \text{rewind, } 1, L \\
\text{add } 2, 1 & \rightarrow \text{add } 2, 0, R \\
\text{rewind, } 0 & \rightarrow \text{rewind} \\
\text{rewind, } 1 & \rightarrow \text{rewind, } L \\
\text{rewind, } \# & \rightarrow \text{halt}
\end{align*}
\]
3D D

start \rightarrow R \text{ first}
first \rightarrow 0, R, \text{ add}
first \rightarrow R, \text{ add}
first \rightarrow R, \text{ add}
add 0 \rightarrow 1, R, \text{ left}
add 1 \rightarrow O, R, \text{ to 01}
too 0 \rightarrow 1, L, \text{ done}
too 1 \rightarrow O, R
too 1 \rightarrow 1, L, \text{ done}
add \rightarrow 1, L, \text{ done}
done 0 \rightarrow .L
done 1 \rightarrow .L
done \rightarrow \text{ halt}
4. Beyond Turing machines: computability

4a. Formulate Church-Turing thesis. Is it a mathematical theorem? Is it a statement about the physical world?

4b. Prove that the halting problem is not algorithmically solvable.

4c. In the proof that the halting problem is not algorithmically solvable, we use a diagonal function $f(n)$ which was defined as follows:

- $f(n) = j_n(n) + 1$ if $n$ is a valid code of a Java program and $j_n$ halts on $n$;
- $f(n) = 0$ otherwise.

Write down the first 3 values $f(0)$, $f(1)$, and $f(2)$ of the diagonal function $f(n)$ for the following case:

- $j_0(n) = n + 2$;
- 1 is not a valid numerical code of a program; and
- $j_2(n) = 4 \times n$.

4d. Not all algorithms are feasible, but, unfortunately, we do not have a perfect definition is feasibility. Give two examples:

- an example of an algorithm which is feasible according to the current definition but not practically feasible, and
- an example of an algorithm which is practically feasible but not feasible according to the current definition.

4e. Briefly describe what is P, what is NP, and what is NP-hard.

4a) Church-Turing Thesis says that anything that can be computed on any physical device can also be computed on a turing machine (or Java program).

Church-turing thesis is a statement about the physical world; not a mathematical theorem.
4. Beyond Turing machines: computability

4a. Formulate Church-Turing thesis. Is it a mathematical theorem? Is it a statement about the physical world?

4b. Prove that the halting problem is not algorithmically solvable.

4c. In the proof that the halting problem is not algorithmically solvable, we use a diagonal function $f(n)$ which was defined as follows:

- $f(n) = j_\alpha(n) + 1$ if $n$ is a valid code of a Java program and $j_\alpha$ halts on $n$;
- $f(n) = 0$ otherwise.

Write down the first 3 values $f(0)$, $f(1)$, and $f(2)$ of the diagonal function $f(n)$ for the following case:

- $j_0(n) = n + 2$;
- 1 is not a valid numerical code of a program; and
- $j_2(n) = 4 \ast n$.

4d. Not all algorithms are feasible, but, unfortunately, we do not have a perfect definition is feasibility. Give two examples:

- an example of an algorithm which is feasible according to the current definition but not practically feasible, and
- an example of an algorithm which is practically feasible but not feasible according to the current definition.

4e. Briefly describe what is P, what is NP, and what is NP-hard.
Halt checker: No algorithm is possible that given a program $p$ & data $d$ checks whether $p$ halts on $d$

We want to prove that halt checker is impossible

Let $p \rightarrow \square

\text{true if } p \text{ halts on } d \rightarrow \square

\text{false if } p \text{ doesn't halt on } d

Problem: $001_2 = 12$

Different binary strings may correspond to same numbers.

To avoid this we append 1 to the front only then interpret string as binary.

Now $1001_2 \neq 11_2$

There is an algorithm that given a natural number $n$ checks whether $n$ corresponds to a Java program, by stripping the first 1 then applying compiler & creating an executable file denoted as $J_n$

Proof by contradiction

Let's assume that halt checker exists then we define the following function

$$f(n) = \begin{cases} sdn(n) + 1 & \text{if } n \text{ is a number corresponding to a Java program } \downarrow \text{ in halts on } n \\ 0 & \text{otherwise} \end{cases}$$

$f(n)$ always halts. Let $n$ denote the number corresponding to Java program that computes $f(n)$, this means that for every input corresponding to a Java program $J_n$ is $\downarrow$, it halts so it halts on $n$ as well.

$$f(n_0) = J_{n_0}(n_0) + 1$$

file:///Q:/cs3350.17/final.html

4/17/2017
No algorithm is possible that given a problem \( p \) and data \( d \) checks if \( p \) halts on \( d \).

Prove that a halt checker is impossible.

\[
\begin{align*}
P &= \{ \text{true if } p \text{ halts on } d \} \\
D &= \{ \text{false otherwise} \}
\end{align*}
\]

A program is a series of ASCII symbols which are used to turn every symbol into 0's and 1's, that when put together form a long sequence of 0's and 1's. This sequence is what is stored in the loc when typing a program. Different binary strings may lead to the same number. To avoid that, we append a 1 to the beginning of the string and interpret it as a binary number.

\[
\begin{align*}
001_2 & = 1_2 \\
1110_2 & = 10_2 \\
11_2 & = 3
\end{align*}
\]

There is an algorithm that given a number \( n \) checks if \( n \) corresponds to a program. If \( n \) is valid, a compiler generates an executable file denoted by \( J_n \).

By contradiction, assume a halt checker exists.

Define function \( f(n) \) as:

\[
\begin{align*}
f(n) &= \begin{cases}
J_n(n) + 1 & \text{if } n \text{ corresponds to a program and halts on } n \\
0 & \text{otherwise}
\end{cases}
\end{align*}
\]

Diagram:

- If \( n \) corresponds to a program, return 0.
- If \( J_n \) halts on \( n \), apply \( J_n \) and return 1 and return \( J_n(n+1) \).
Let \( n_0 \) denote the number corresponding to the program that computes \( f(n) \).

For every \( n \), \( J_{n_0}(n_0) = f(n) \)

for \( n = n_0 \)

\[ J_{n_0}(n) = F(n_0) \]

By definition, \( n_0 \) is a number corresponding to a program. 

\( J_0 \) is \( F \), it always halts, so it halts on \( n_0 \) too.

Therefore, \( J_0(n_0) = J_{n_0}(n_0) + 1 \)

\[ 0 \neq 1 \]

This is a contradiction, thus a halting checker does not exist.

4c) \[ J_0(n) = n + 2 \]

1 not valid

\[ J_2(n) = 4 \times n \]

\[ f(n) \]

\[ \begin{array}{c|ccc}
0 & 2 & 3 & 4 \\
1 & - & - & - \\
J_2 & 0 & 4 & 8 \\
f(n) & 3 & 0 & 9 \\
\end{array} \]
4c) 

<table>
<thead>
<tr>
<th>m</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>O</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>f(3)</td>
<td>3</td>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>

4d) 

\[ f_A(n) = 10^{3n} \] feasible by definition but not by practice

\[ f_A(n) = e^{-n^{10}} \] feasibly by practice but not by definition

4e) 

P: problems that can be solved in feasible time

NP: Once we have a candidate for a solution, we can check in feasible time whether it is indeed a solution

NP-hard: every problem from NP can be reduced to this problem
4d) 
\[ F_A(n) = 10^{301} n \] feasible by definition, 
not feasible in practice.

\[ F_A(n) = e^{10^{-10}} n \] not feasible by definition, 
feasible in practice.

4e) \( P \)- are problems that can be solved with 
theoretical algorithms that run in polynomial time.

\( NP \)- Once there is a candidate for a solution, 
we can check in feasible time, whether it 
is indeed a solution.

\( NP \)-Hard - A problem is said to be \( NP \)-hard 
if an algorithm for solving it can be 
translated into one for solving any other 
\( NP \)-problem, meaning every problem from 
the class \( NP \) can be reduced to this 
program.