1. Show why the variable Choosing[i] is needed in the bakery algorithm. Specifically, consider a version of the algorithm in which this variable is omitted, and present in detail an execution in which mutual exclusion is violated.

2. The next page presents Peterson’s mutual exclusion algorithm which ensures no starvation. Consider Algorithm 2 and Algorithm 3, presented on the last page, and prove or refute the following claims for each of them:
   (a) The algorithm provides mutual exclusion.
   (b) The algorithm provides no-deadlock.
   (c) The algorithm provides no-starvation.

3. The last page presents Burns’ mutual exclusion algorithm (Algorithm 4). Prove or refute the following claims for Burns’ algorithm:
   (a) The algorithm provides mutual exclusion.
   (b) The algorithm provides no-deadlock.
   (c) The algorithm provides no-starvation.

4. Consider the Tournament Tree algorithm for n-process mutual exclusion algorithm, presented in class and tutorial.
   (a) Prove that there is an integer k such that the algorithm ensures k bounded-waiting.
      Assume a synchronized schedule in which for every two processors pi, pj, between any two consecutive steps of pi, there is one step of pj.
   (b) Compute the value of k.
5. One of the key reasons for mutual exclusion is to control the access to shared resources. There are cases where the amount of shared resource might differ from request to request (e.g. the bandwidth of communication lines), therefore the \textit{h-out of-k mutual exclusion} problem was proposed.

In this problem, each process specifies which is the number \( h \) of units of shared resource it needs. In total, the shared resource has \( k \) units that can be used simultaneously. A process can use the shared resource only if there are at least \( h \) free units of the resource. In addition, the resource should be used efficiently: a process which needs \( h \) units, does not wait when there are \( h \) free units.

(a) Give an algorithm using RMW operations that solves \( h \)-out of-\( k \) mutual exclusion and exploits the resource efficiently.

(b) Does your algorithm provide no starvation? Prove your answer.

\begin{algorithm}
\textbf{Algorithm 1} Peterson’s algorithm
Initially flag[0] and flag[1] are false and victim is 0
\begin{align*}
\text{code for } p_0 & \\
\langle \text{Entry} \rangle: & \\
1: & \text{flag}[0]:=\text{true} \\
2: & \text{victim}:=0 \\
3: & \textbf{while} \text{ flag}[1] \text{ and victim==0 do} \text{ wait} \\
\langle \text{Critical Section} \rangle: & \\
\langle \text{Exit} \rangle: & \\
4: & \text{flag}[0]:=\text{false} \\
\langle \text{Remainder} \rangle: & \\
\end{align*}
\begin{align*}
\text{code for } p_1 & \\
\langle \text{Entry} \rangle: & \\
1: & \text{flag}[1]:=\text{true} \\
2: & \text{victim}:=1 \\
3: & \textbf{while} \text{ flag}[0] \text{ and victim==1 do} \text{ wait} \\
\langle \text{Critical Section} \rangle: & \\
\langle \text{Exit} \rangle: & \\
4: & \text{flag}[1]:=\text{false} \\
\langle \text{Remainder} \rangle: & \\
\end{align*}
\end{algorithm}
Algorithm 2
Initially flag[0] and flag[1] are false

code for \( p_0 \)

\[ \langle \text{Entry} \rangle : \]

1: while flag[1] do wait
2: flag[0]:=true
3: flag[0]:=false

\[ \langle \text{Exit} \rangle : \]

\[ \langle \text{Critical Section} \rangle : \]

\[ \langle \text{Remainder} \rangle : \]

code for \( p_1 \)

\[ \langle \text{Entry} \rangle : \]

1: while flag[0] do wait
2: flag[1]:=true
3: flag[1]:=false

\[ \langle \text{Exit} \rangle : \]

\[ \langle \text{Critical Section} \rangle : \]

\[ \langle \text{Remainder} \rangle : \]

Algorithm 3
Initially turn is 0

code for \( p_0 \)

\[ \langle \text{Entry} \rangle : \]

1: while turn=1 do wait
2: turn:=1
3: turn:=0

\[ \langle \text{Exit} \rangle : \]

\[ \langle \text{Critical Section} \rangle : \]

\[ \langle \text{Remainder} \rangle : \]

code for \( p_1 \)

\[ \langle \text{Entry} \rangle : \]

1: while turn=0 do wait
2: turn:=0

\[ \langle \text{Exit} \rangle : \]

\[ \langle \text{Critical Section} \rangle : \]

\[ \langle \text{Remainder} \rangle : \]

Algorithm 4 Burns’ algorithm
Flag[N] is array of boolean initially 0.

code for \( p_i \)

\[ \langle \text{Entry} \rangle : \]

1: Flag[i]:= 0
2: for \( (j:=0; j\leq i-1 ; j++) \) do
3: if Flag[j]=1 then
4: goto line 1
5: Flag[i]:= 1
6: for \( (j:=0; j\leq i-1 ; j++) \) do
7: if Flag[j]=1 then
8: goto line 1
9: for \( (j:=i+1; j\leq n-1 ; j++) \) do
10: if Flag[j]=1 then
11: goto line 9
12: Flag[i]:= 0

\[ \langle \text{Critical Section} \rangle : \]

\[ \langle \text{Exit} \rangle : \]

\[ \langle \text{Remainder} \rangle : \]