1 Esercizio 1

Abbiamo una funzione f per la quale esiste una valore i tale che $f(i) = 0$. Gli algoritmi che seguono cercano una tale valore i, chiamato *punto zero*, dividendo lo spazio di ricerca in due parte: i punti positivi e i punti negativi.

Un algoritmo è corretto se, per **tutte** le esecuzione, i due processi finiscono dopo che uno dei due ha trovato un punto zero. Per ciascun algoritmo A, B, C e D determinate se è corretto o no, giustificando la vostra risposta.

Per gli algoritmi C e D usiamo una istruzione speciale: `await C then R` che è eseguita in un modo atomico e ha il comportamento seguente: se la condizione `C` è vera, allora il processo esegue `R` e può procedere la sua esecuzione, se invece la condizione è falsa, il processo verrà interrotto e potrà rifare più tardi l'istruzione.

```
bool found=false;
```

```
Process P
int i=0;
while(!found){
  i=i+1;
  found=(f(i)==0);
}
```

```
Process Q
int j=1;
while(!found){
  j=j-1;
  found=(f(j)==0);
}
```

Figura 1: Algoritmo A

```
bool found=false;
```

```
Process P
int i=0;
while(!found){
  i=i+1;
  if(f(i)==0){
    found=true;
  }
}
```

```
Process Q
int j=1;
while(!found){
  j=j-1;
  if(f(j)==0){
    found=true;
  }
}
```

Figura 2: Algoritmo B

```
bool found=false;
int turn=1;
```

```
Process P
int i=0;
while(!found){
  await turn==1 then turn=2;
  i=i+1;
  if(f(i)==0){
    found=true;
  }
}
```

```
Process Q
int j=1;
while(!found){
  await turn==2 then turn=1;
  j=j-1;
  if(f(j)==0){
    found=true;
  }
}
```

Figura 3: Algoritmo C

```
bool found=false;
int turn=1;
```

```
Process P
int i=0;
while(!found){
  await turn==1 then turn=2;
  i=i+1;
  if(f(i)==0){
    found=true;
  }
}
turn=2;
```

```
Process Q
int j=1;
while(!found){
  await turn==2 then turn=1;
  j=j-1;
  if(f(j)==0){
    found=true;
  }
}
turn=1;
```

Figura 4: Algoritmo D

2 Esercizio 2

Consideriamo la soluzione seguente per il problema della sezione critica proposto dai ricercatori Manna e Pnueli.

```
int wantp=0;
int wantq=0;
```

```
Process P
while(true){
  SNC
  if(wantq==-1){
    wantp=-1;
  }else{
    wantp=1;
  }
  while(wantp==wantq){}
  SC
  wantp=0;
}
```

```
Process Q
while(true){
  SNC
  if(wantp==-1){
    wantq=1;
  }else{
    wantq=-1;
  }
  while(wantq==-wantp){}
  SC
  wantq=0;
}
```

Figura 5: Algoritmo di Manna-Pnueli

1. Costruire il diagramma degli stati per questo algoritmo.
2. Dedurre Quale sono le proprietà verificate da questo algoritmo (mutua esclusione, assenza di deadlock, assenza di starvation con fairness fra i processi)?

3 Esercizio 3

Supponiamo che abbiamo a disposizione l'istruzione atomica che segue:

```
int flip(int x){
  atomic{
    x=(x+1)%2; //inversione della variabile
    return x;  //ritorna la nuova valore
  }
}
```

Consideriamo la soluzione seguente per risolvere il problema della sezione critica.

1. Dimostrate perché questo algoritmo non rispetta la proprietà di mutua esclusione.

```
int x=0;
```

```
Process P
while(true){
  SNC
  while(flip(x)!=1){
    while(x!=0){}
  }
  SC
  x=0;
}
```

```
Process Q
while(true){
  SNC
  while(flip(x)!=1){
    while(x!=0){}
  }
  SC
  x=0;
}
```

Figura 6: Algoritmo di mutua esclusione usando `flip`

2. Cosa cambia se nella funzione `flip`, il `%2` è sostituito da `%3`?
3. Per questa ultima soluzione, giustificare se l'algoritmo rispetta le proprietà di assenza di deadlock e di starvation (con ipotesi di fairness fra i processi).

4 Esercizio 4

Consideriamo la soluzione seguente per risolvere il problema della sezione critica per N processi dove abbiamo un processo S in più che aiuta la gestione.

```
int req=0;
int aut=0;
bool end=true;
```

```
Process P[id] //id in {1,...,N}
while(true){
  SNC
  while(aut!=id){
   req=id;
  }
  SC
  end=true;
  req=0;
}
```

```
Process S
while(true){
  while(req==0){}
  end=false;
  aut=req;
  while(!end){}
  aut=0;
}
```

Figura 7: Algoritmo di mutua esclusione per N processi

1. Dimostrare perché l'algoritmo della figura 7 non rispetta la mutua esclusione.
2. Modifichiamo l'algoritmo per ottenere quello della figura 8. Questo algoritmo verifica la proprietà di mutua esclusione? Qual è il problema di questo ultimo algoritmo?

```
int req=0;
int aut=0;
bool end=true;
```

```
Process P[id] //id in {1,...,N}
while(true){
  SNC
  while(aut!=id){
   req=id;
  }
  SC
  end=true;
  req=0;
  while(aut!=0){}
}
```

```
Process S
while(true){
  while(req==0){}
  end=false;
  aut=req;
  while(!end){}
  aut=0;
}
```

Figura 8: Algoritmo di mutua esclusione per N processi