Homework 7

Problem1

You are asked to design a schedule for a round-robin tennis tournament.
There are n $= 2^k$ (k ≥ 1) players. Each player must play every other player, and each player must paly one match per round for n$-$1 rounds.
Denote the players by $P_1, P_2, ..., P_n$. Output the schedule for each player.
(Hint: use divide and conquer in the following way. First, divide the players into two equal groups and let them play **within the groups** for the first $\frac{n}{2} - 1$ rounds. Then, design the games **between the groups** for the other $\frac{n}{2}$rounds.)

Problem1

```
1     Algorithm RoundRobin(First, Last, Round)
2     Input: First(First Player Num), Last(Last Player Num), Round(Total round to schedule)
3     Output: result(the result schedule, n*(n-1) matrix)
4
5     begin
6         if Last = First+1 then
7             result[First][Round] = Last;
8             result[Last][Round] = First;
9         else
10             Middle := (First+Last-1) / 2;
11             //Divide
12             RoundRobin(First, Middle, (Round+1)/ 2- 1);
13             RoundRobin(Middle+1, Last, (Round+1)/ 2- 1);
14
15             //Conquer(between the groups)
16             for round_num := (Round+1) / 2 to Round
17                 for offset := 0 to (Middle-First)
18                     Group_size = (Round+1) /2;
19                     play1 = First + offset;
20                     play2 = (Middle+1) + [(round_num + offset)% Group_size];
21                     result[play1][round_num] = play2;
22                     result[play2][round_num] = play1;
```

- 此處以 k=3 為例
  - n = 2^k = 8, round = n-1 = 7
  - Input: RoundRobin(1, 8, 7)

Problem1

```
1       Algorithm RoundRobin(First, Last, Round)
2       Input: First(First Player Num), Last(Last Player Num), Round(Total round to schedule)
3       Output: result(the result schedule, n*(n-1) matrix)
4
5       begin
6           if Last = First+1 then
7               result[First][Round] = Last;
8               result[Last][Round] = First;
9           else
10              Middle := (First+Last-1) / 2;
11              //Divide
12              RoundRobin(First, Middle, (Round+1)/ 2- 1);
13              RoundRobin(Middle+1, Last, (Round+1)/ 2- 1);
14
15              //Conquer(between the groups)
16              for round_num := (Round+1) / 2 to Round
17                  for offset := 0 to (Middle-First)
18                      Group_size = (Round+1) /2;
19                      play1 = First + offset;
20                      play2 = (Middle+1) + [(round_num + offset)% Group_size];
21                      result[play1][round_num] = play2;
22                      result[play2][round_num] = play1;
```

- 當只有 2 個 players 的時候

	Rd_1
P_1	2
P_2	1

Problem1

```
...
5       begin
6           if Last = First+1 then
7               result[First][Round] = Last;
8               result[Last][Round] = First;
9           else
10              Middle := (First+Last-1) / 2;
11              //Divide, (Round+1) 即為人數, 因為 divide 所以人數砍半, 人數再 -1 就是 divide 後的 Round 值
12              RoundRobin(First, Middle, (Round+1)/ 2- 1);
13              RoundRobin(Middle+1, Last, (Round+1)/ 2- 1);
14
15              //Conquer(between the groups)
16              for round_num := (Round+1) / 2 to Round
17                  for offset := 0 to (Middle-First)
18                      Group_size = (Round+1) /2;
19                      play1 = First + offset;
20                      play2 = (Middle+1) + [(round_num + offset)% Group_size];
21                      result[play1][round_num] = play2;
22                      result[play2][round_num] = play1;
```

- 超過 2 個 players 的時候，會進行 divide 的動作
  - Middle = (1+8-1) / 2 = 4
  - RoundRobin(First, Middle, (Round+1)/ 2- 1)
    is RoundRobin(1, 4, (7+1)/2-1)=RoundRobin(1, 4, 3)
  - RoundRobin(Middle+1, Last, (Round+1)/ 2- 1)
    is RoundRobin(5, 8, 3)

Problem1

```
...
9           else
10              Middle := (First+Last-1) / 2;
11              //Divide
12              RoundRobin(First, Middle, (Round+1)/ 2- 1);
13              RoundRobin(Middle+1, Last, (Round+1)/ 2- 1);
```

- RoundRobin(1, 4, 3)
- RoundRobin(5, 8, 3)

	1	2	3	4	5	6	7
P_1	2	3	4	5	6	7	8
P_2	1	4	3	6	7	8	5
P_3	4	1	2	7	8	5	6
P_4	3	2	1	8	5	6	7
P_5	6	7	8	1	4	3	2
P_6	5	8	7	2	1	4	3
P_7	8	5	6	3	2	1	4
P_8	7	6	5	4	3	2	1

Problem1

```
. . .
9          else
10             Middle := (First+Last-1) / 2;
11             //Divide
12             RoundRobin(First, Middle, (Round+1)/ 2- 1);
13             RoundRobin(Middle+1, Last, (Round+1)/ 2- 1);
14
15             //Conquer(between the groups)
16             for round_num := (Round+1) / 2 to Round
17                 for offset := 0 to (Middle-First)
18                     Group_size = (Round+1) /2;
19                     play1 = First + offset;
20                     play2 = (Middle+1) + [(round_num + offset)% Group_size];
21                     result[play1][round_num] = play2;
22                     result[play2][round_num] = play1;
```

- 假設上述 divide 遞迴處理完後，回到最上層 conquer 的部分
  - **round_num := [(Round+1) / 2] to Round**
    is round_num = [(7+1)/2] to 7 = 4 to 7
  - **offset := 0 to (Middle-First)**
    is offset := 0 to (4-1) = 0 to 3
  - Group_size = (Round+1) /2
    is Group_size = (7+1) /2 = 4

Problem1

```
16          for round_num := (Round+1) / 2 to Round
17              for offset := 0 to (Middle-First)
18                  Group_size = (Round+1) /2;
19                  play1 = First + offset;
20                  play2 = (Middle+1) + [(round_num + offset)% Group_size];
21                  result[play1][round_num] = play2;
22                  result[play2][round_num] = play1;
```

- round_num = 4 to 7、offset= 0 to 3、Group_size = 4

	1	2	3	4	5	6	7
P_1	2	3	4	5	6	7	8
P_2	1	4	3	6	7	8	5
P_3	4	1	2	7	8	5	6
P_4	3	2	1	8	5	6	7
P_5	6	7	8	1	4	3	2
P_6	5	8	7	2	1	4	3
P_7	8	5	6	3	2	1	4
P_8	7	6	5	4	3	2	1

Problem1

```
19                    play1 = First + offset;
20                    play2 = (Middle+1) + [(round_num + offset)% Group_size];
21                    result[play1][round_num] = play2;
22                    result[play2][round_num] = play1;
```

- round_num = 4 to 7、offset = 0 to 3、Group_size = 4

r_num=4 offset=0 g_size=4	r_num=4 offset=1 g_size=4
play1=1+0=1 play2=(4+1)+ (4+0)%4=5	play1=1+1=2 play2=(4+1)+ (4+1)%4=6
r[p1][r_num] =r[1][4]=5 r[p2][r_num] =r[5][4]=1 /* $P_1 P_5$ 對打 */	r[p1][r_num] =r[2][4]=6 r[p2][r_num] =r[6][4]=2 /* $P_2 P_6$ 對打 */

4	5	6	7
5	6	7	8
6	7	8	5
7	8	5	6
8	5	6	7
1	4	3	2
2	1	4	3
3	2	1	4
4	3	2	1

Problem1

Final Output: n*(n-1) matrix

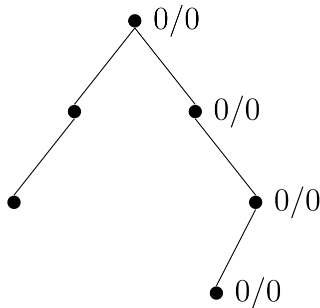
	1	2	3	4	5	6	7
P_1	2	3	4	5	6	7	8
P_2	1	4	3	6	7	8	5
P_3	4	1	2	7	8	5	6
P_4	3	2	1	8	5	6	7
P_5	6	7	8	1	4	3	2
P_6	5	8	7	2	1	4	3
P_7	8	5	6	3	2	1	4
P_8	7	6	5	4	3	2	1

Problem2

Consider the problem of finding balance factors in binary trees discussed in class (see slides for "Design by Induction").
Solve this problem using DFS. You need only to define preWORK and postWORK.

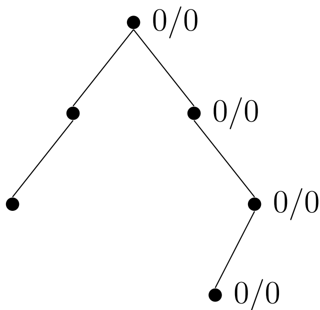
Problem2

```
1        preWork:
2              v.height := 0;
3              v.balance_factor := 0;
4              v.left_height := 0;
5              v.right_height := 0;
6        postWork I:
7              v.height := MAX(v.height, w.height+1);
8              if w = v.leftchild then
9                    v.left_height := w.height + 1;
10              else if w:= v.rightchild then
11                    v.right_height := w.height + 1;
12        postWork II:/*After left and right child are traversed*/
13              v.balance_factor := v.left_height - v.right_height;
```



Problem2

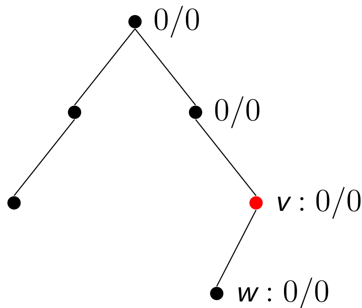
```
1       preWork:
2           v.height := 0;
3           v.balance_factor := 0;
4           v.left_height := 0;
5           v.right_height := 0;
...
```



- 進行 DFS traverse, 0/0= height / balance factor
- preWORK: 一個 node v 被 traverse 到並要標記前的前置動作

Problem2

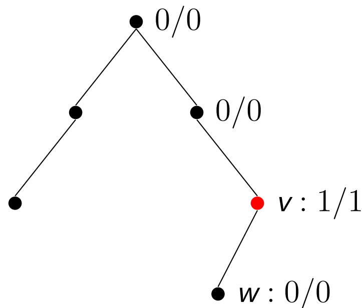
```
. . .
6        postWork I:
7              v.height := MAX(v.height, w.height+1);
8              if w = v.leftchild then
9                    v.left_height := w.height + 1;
10              else if w:= v.rightchild then
11                    v.right_height := w.height + 1;
12         postWork II:/*After left and right child are traversed*/
13              v.balance_factor := v.left_height - v.right_height;
```



- postWORK I: DFS 到底沒有路後，從 node w 回溯到 node v 時的動作
- v.h=MAX(v.h, w.h+1)=(0,0+1)=(0,1)=1
- v.bf = v.left_height - v.right_height=1-0=1

Problem2

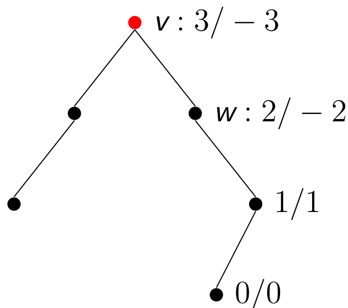
```
...
6       postWork I:
7           v.height := MAX(v.height, w.height+1);
8           if w = v.leftchild then
9               v.left_height := w.height + 1;
10           else if w:= v.rightchild then
11               v.right_height := w.height + 1;
12        postWork II:/*After left and right child are traversed*/
13           v.balance_factor := v.left_height - v.right_height;
```



- postWORK I: DFS 到底沒有路後，從 node w 回溯到 node v 時的動作
- v.h=MAX(v.h, w.h+1)=(0,0+1)=(0,1)=1
- v.bf = v.left_height - v.right_height=1-0=1

Problem2

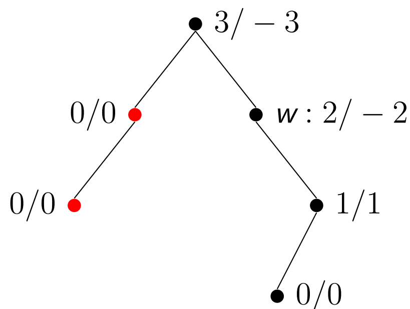
```
. . .
6       postWork I:
7           v.height := MAX(v.height, w.height+1);
8           if w = v.leftchild then
9               v.left_height := w.height + 1;
10           else if w:= v.rightchild then
11               v.right_height := w.height + 1;
12       postWork II:/*After left and right child are traversed*/
13           v.balance_factor := v.left_height - v.right_height;
```



- v.h=MAX(v.h, w.h+1)=(0,2+1)=(0,3)=3
- v.bf = v.left_height - v.right_height = 0 - 3 = -3

Problem2

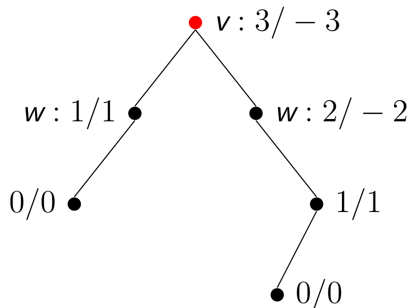
```
1       preWork:
2           v.height := 0;
3           v.balance_factor := 0;
4           v.left_height := 0;
5           v.right_height := 0;
...
```



- preWORK: 一個 node v 被 traverse 到並要標記前的前置動作

Problem2

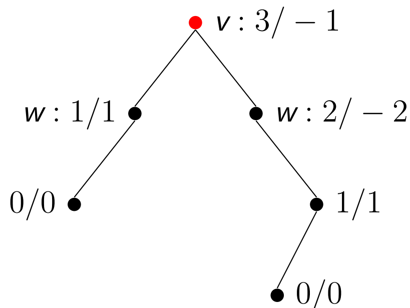
```
...
6       postWork I:
7           v.height := MAX(v.height, w.height+1);
8           if w = v.leftchild then
9               v.left_height := w.height + 1;
10           else if w:= v.rightchild then
11               v.right_height := w.height + 1;
12       postWork II:/*After left and right child are traversed*/
13           v.balance_factor := v.left_height - v.right_height;
```



- v.h=MAX(v.h, w.h+1)=(3,1+1)=(3,2)=3
- v.bf = v.left_height - v.right_height = 2 - 3 = -1

Problem2

```
. . .
6       postWork I:
7            v.height := MAX(v.height, w.height+1);
8            if w = v.leftchild then
9                 v.left_height := w.height + 1;
10            else if w:= v.rightchild then
11                 v.right_height := w.height + 1;
12       postWork II:/*After left and right child are traversed*/
13            v.balance_factor := v.left_height - v.right_height;
```



- v.h=MAX(v.h, w.h+1)=(3,1+1)=(3,2)=3
- v.bf = v.left_height - v.right_height = 2 - 3 = -1

Question3

給定一個 connected 無向圖 G = (V, E)，
一個 G 的 spanning tree T = (V', E')，
以及 G 上的一點 v，
寫出一個函數判斷 T 是否為 G 以 v 為起點的 spanning tree。
複雜度要是 O(|V|+|E|)

Question3

檢查 T 是否為 G 的 subgraph (optional)
假設 vertex 資訊是單純的整數 (總共幾個點)，可以在 O(1) 檢查 $V' \subseteq V$
假設有 T 的 adjacency list 與 G 的 adjacency matrix，可以在 O(|E|) 檢查 $E' \subseteq E$
其實題目講了 T 是 G 的 spanning tree，可以當做 pre-condition

Question3

檢查 T 有沒有 cycle (optional)
若把 T 是 spanning tree 當成 pre-condition，那麼可以不用檢查

Question3

T 圖以 v 點為樹根，向外延伸出若干子樹
若是 $(w_1, w_2) \in E$，那麼這兩點應該要在同一個子樹

雖然我們無法事先預測當初 DFS 的執行順序，但仍然能夠觀察
T 的子樹判斷合理性

Question3

對 T 做 DFS
沿途標記經過的各點
對 T 上的點 w_1 而言，若是 DFS 做完了，但卻有在 G 上的相鄰點 w_2 沒有被標記？
如果當初是 w_1 先被經過，那麼 w_2 必然會在 w_1 的 DFS 過程中被標記
如果當初是 w_2 先被經過，那麼 w_1 會是 w_2 的 descent，對 T 做 DFS 時不可能會先看到 w_1

發生這種狀況有兩個可能：
w_2 沒有被 T 經過
(w_1, w_2) 為 cross edge

Question3

```
function isDFSFrom(G = (V, E), T = (V, E'), v)
    /* (optional) isSubgraph(T, G); */
    mark v;
    for (v, w) ∈ E' do
        if v.parent = w then
            continue;
        if w.mark /*cycle*/ then
            result := false;
        w.parent := v;
        isDFSfrom(G, T, w)
    for (v, w) ∈ E do
        if not w.mark /*cross edge*/ then
            result := false;
```

Question3

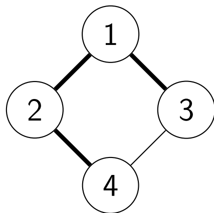
```
from networkx import *

def isDFSfrom(G: Graph, T: Graph, v):
    #if not isSubgraph(T, G): return False
    T.nodes[v]['mark'] = True
    for w in T.neighbors(v):
        if T.nodes[v].get('parent') == w:
            continue
        if T.nodes[w].get('mark'):
            return False #cycle
        T.nodes[w]['parent'] = v
        if not isDFSfrom(G, T, w):
            return False

    for w in G.neighbors(v):
        if not T.nodes[w].get('mark'):
            return False

    return True
```

Question3



v = 1 或 2 時結果為 false
v = 3 或 4 時結果為 true

Question3

上述解答用的演算法只適用在兩點之間最多一邊的情境
某版本的舊解答（w.parent != v）以及不使用 parent 的版本可以處理多邊
但成立的前提是一個 edge 不會被挑第二遍
也就是兒子跑遞迴時不會看到自己老爸
不然會爆炸。

所以重點是標記 edge
而標記了 edge，也不需要記錄 parent，碰到 mark 就死
改良版 algorithm 如下

Question3

```
function isDFSFrom(G = (V, E), T = (V, E'), v)
    /* (optional) isSubgraph(T, G); */
    mark v;
    for e = (v, w) ∈ E' do
        if e.pass then
            continue;
        e.pass = true;
        if w.mark /*cycle*/ then
            result := false;
        isDFSfrom(G, T, w)
    for (v, w) ∈ E do
        if not w.mark /*cross edge*/ then
            result := false;
```

Question3

```
def isDFSfrom(G, T, v):
    #if not isSubgraph(T, G): return False
    T.nodes[v]['mark'] = True
    for w in T.neighbors(v):
        if isinstance(T, MultiGraph):
            es = T.adj[v][w]
            e = None
            for i in es:
                if not es[i].get('pass'):
                    e = es[i]
                    break
            if e is None: continue
        else:
            e = T.adj[v][w]
            if e.get('pass'): continue
        e['pass'] = True
        if T.nodes[w].get('mark'):
            return False #cycle
        if not isDFSfrom(G, T, w):
            return False
    for w in G.neighbors(v):
        if not T.nodes[w].get('mark'):
            return False
    return True
```

Question4

擁有 Eulerian circuit 的條件是每個點連出去的邊都有正偶數個
若用 adjacency list，花 O(|E|) 時間檢查

題目提示：將 Eulerian circuit 看成 cycles 的組合
先走出一個 cycle，再從 cycle 上每一點走出其他 cycles
並將其他 cycles 的路過順序插入主 cycle

Question4

```
function FINDEULERIANCIRCUIT(G = (V, E))
    if any degree of vertices is odd or zero then
        return;
    initialize Path and a list L
    pick one vertex v
    TraceCycle(v, Path, G, L);
    while L is not empty do
        pick w from L
        if w.degree > 0 then
            initialize subPath
            TraceCycle(w, subPath, G, L)
            insert subPath into Path
        else
            pop w
    return Path;
```

Question4

```
function TraceCycle(v, Path, G, L)
    nowPosition := v;
    repeat
        pick an edge (nowPosition, w)
        L.append(w);
        put (nowPosition, w) in Path
        remove (nowPosition, w) from G
        nowPosition := w;
    until nowPosition == v
```

Question4

Trace 的總運行時間複雜度是 O(|E|)
insert subPath into Path 可以在 O(1) 完成，實作上需要一些技巧

Question4

另外一種解法是找到 edge (v, w) 就刪除 edge 並走出去到 w
直到所有 edges 都消失

在走完一個分支後插入 edge (v, w) 到 path 前端
代表我從現在的節點 v 走到 w，再接著從 w 走一些路徑（遞迴）
或是走完所有分支後（回到上個節點前）插入現在的節點 v。概念上等價於前者

分支不見得是一個 circuit
但必然使分支下去的所有節點 degree 變成 0
又由於是照著相鄰關係把 edge(vertex) 放入 path 當中，所以這個 path 就是一個 Eulerian circuit

Question4

```
function FindEulerianCircuit(G = (V, E))
    if any degree of vertices is odd or zero then
        return;
    initialize vertex/edge path list
    pick some v in G
    Euler(G, v)
function Euler(G, v)
    for all neighbors w of v do
        remove (v, w) from G
        Euler(G, w)
        append (v, w) into the front of the edge path list
    append v into the front of the vertex path list
```

Question4

```
from networkx import *
from networkx.utils import arbitrary_element

def EulerCircuit(G: MultiGraph):
    if ...:
        return None
    vertex_path, edge_path = [], []
    v = arbitrary_element(G.nodes)
    Euler(G.copy(), v, vertex_path, edge_path)
    return vertex_path[::-1], edge_path[::-1] #reverse

def Euler(G: MultiGraph, v, vertex_path, edge_path):
    while len(G.adj[v]) > 0:
        w = arbitrary_element(G.adj[v]) #pick one neighbor
        G.remove_edge(v, w)
        Euler(G, w, vertex_path, edge_path)
        edge_path.append((v, w))
    vertex_path.append(v)
```

Problem5

A binary de Bruijn sequence is a (cyclic) sequence of 2^n bits $a_1 a_2 \cdots a_{2^n}$ such that each binary string s of size n is represented somewhere in the sequence; that is, there exists a unique index i such that $s = a_i a_{i+1} \cdots a_{i+n-1}$ (where the indices are taken modulo $2n$). For example, the sequence 11010001 is a binary de Bruijn sequence for $n = 3$. Let $G_n = (V, E)$ be a directed graph defined as follows. The vertex set V corresponds to the set of all binary strings of size $n - 1$ ($|V| = 2n - 1$). A vertex corresponding to the string $a_1 a_2 \cdots a_{n-1}$ has an edge leading to a vertex corresponding to the string $b_1 b_2 \cdots b_{n-1}$ if and only if $a_2 a_3 \cdots a_{n-1} = b_1 b_2 \cdots b_{n-2}$. Prove that G_n is a directed Eulerian graph, and discuss the implications for de Bruijn sequences.

Problem5(cont'd)

長度為 2^n bits 的 binary de Bruijn sequence 中，所有長度為 n 的連續數列恰好是 n bits 所有可能的組合。(因為是環狀數列，長度為 n 的連續數列共有 2^n 個)
Example(n $=$ 3): the sequence 11010001 contains

$$\{110, 101, 010, 100, 000, 001, 011, 111\}$$

Problem5(cont'd)

De Bruijn graph $G_n = (V, E)$ 為一有向圖，
$V = \{$all binary strings of size $n-1\}$，E 的定義如下:
若點 v 對應一個 binary string $a_1a_2\cdots a_{n-1}$，則它有條有向邊通往點 u(對應 $b_1b_2\cdots b_{n-1}$) iff $a_2a_3\cdots a_{n-1} = b_1b_2\cdots b_{n-2}$。($v$ 的後 $n-2$ 個 bits 和 u 的前 $n-2$ 個 bits 相同，則 v 有條邊通向 u)
Edge Labeling: 每條邊上的 label 為指向的點的最後一個 bit。

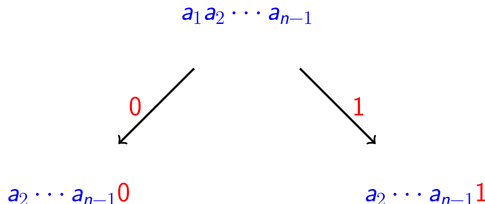


Figure: Edge construction of de Bruijn graph.(there may be self loops)

Problem5(cont'd)

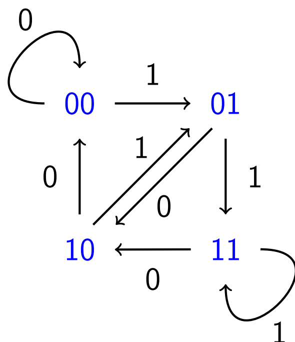


Figure: A de Bruijn graph for $n = 3$

Problem5(cont'd)

A directed Eulerian graph is a directed graph with an Eulerian cycle. To prove that G_n is a directed Eulerian graph, let's see the property of a directed graph.

A directed graph has an Eulerian cycle if and only if every vertex has equal in degree and out degree.

1. in degree：對於每個頂點，若該頂點的前 $n-2$ 個 bits 為 $a_1 a_2 \cdots a_{n-2}$, 則它恰好有兩條有向邊來自 $0 a_1 a_2 \cdots a_{n-2}$ 和 $1 a_1 a_2 \cdots a_{n-2}$ 對應的頂點，每個頂點的 in degree 均為 2。
2. out degree：對於每個頂點，若該頂點的後 $n-2$ 個 bits 為 $a_2 a_3 \cdots a_{n-1}$, 則它恰好有兩條有向邊指向 $a_2 a_3 \cdots a_{n-1} 0$ 和 $a_2 a_3 \cdots a_{n-1} 1$ 對應的頂點，每個頂點的 out degree 均為 2。

Hence, every vertex has equal in degree and out degree, G_n has an Eulerian cycle, that is, G_n is a directed Eulerian graph.
從 de Bruijn graph 的任意一個頂點走尤拉路徑 (同時也是尤拉環)，會得到一個可能的 de Bruijn sequence。