Merge sort as described in text books:

```
mergeSort(A):
    mergeSort(A, 0, A.length − 1)

mergeSort(A, p, r):     // Sort A[ p ··· r ]
    if p < r then
        q ← (p + r) / 2
        mergeSort(A, p, q)
        mergeSort(A, q + 1, r)
        merge(A, p, q, r)
```

```
merge(A, p, q, r):
    //Merge A[p ··· q] and A[q + 1 ··· r] into A[p ··· r], in sorted order
    //Pre: A[p ··· q], and A[q + 1 ··· r] are in sorted order
    Copy A[ p ··· q ] into L[1 ··· q − p + 1 ]
    Copy A[ q + 1 ··· r ] into R[1 ··· r − q ]
    L[ q − p + 2 ] ← R[ r − q + 1 ] ← ∞
    i ← 1,  j ← 1
    for k ← p to r do
        if L[ i ] ≤ R[ j ] then
            A[ k ] ← L[ i++ ]
        else
            A[ k ] ← R[ j++ ]
```

把A的两个部分分别放进两个数组L,R

Ideas for improvement:

0. Avoid using sentinels (infinity)
1. Pass temporary array to avoid creation of arrays L and R locally in merge.
2. When size is smaller than a threshold T, use another algorithm to sort, such as insertion sort.

```
mergeSort(A):
    mergeSort(A, 0, A.length − 1)

mergeSort(A, p, r):   // Sort A[ p ··· r ]
    if r − p < T then
        insertionSort(A, p, r)
    else
        q ← (p + r) / 2
        mergeSort(A, p, q)
        mergeSort(A, q + 1, r)
        merge(A, p, q, r)
```

```
merge(A, B, p, q, r):
    //Merge A[p ··· q] and A[q + 1 ··· r] into A[p ··· r], in sorted order
    //Pre: A[p ··· q], and A[q + 1 ··· r] are in sorted order

    Copy A[ p ··· r ] into B[ p ··· r ]
    i ← p,  j ← q + 1
    for k ← p to r do
        if j > r  ||  ( i ≤ q and B[ i ] ≤ B[ j ] ) then
            A[ k ] ← B[ i++ ]
        else
            A[ k ] ← B[ j++ ]
```

3. Avoid unnecessary copying of values between the arrays.

```
mergeSort(A):
    Create a new array B = copy of A
    mergeSort(A, B, 0, A.length − 1)

mergeSort(A, B, p, r):
//Sort A[p ··· r] or B[p ··· r] into A[p ··· r]
//Pre: A[ p ··· r ] and B[ p ··· r ] have
//       the same elements
    if r − p < T then
        insertionSort(A, p, r)
    else
        q ← (p + r) / 2
        mergeSort(B, A, p, q)
        mergeSort(B, A, q + 1, r)
        merge(B, A, p, q, r)
```

```
merge(A, B, p, q, r):
    //Merge A[p ··· q] and A[q + 1 ··· r] into B[p ··· r], in sorted order
    //Pre: A[p ··· q], and A[q + 1 ··· r] are in sorted order

    i ← p,  j ← q + 1
    for k ← p to r do
        if j > r  ||  (i ≤ q and A[ i ] ≤ A[ j ]) then
            B[ k ] ← A[ i++ ]
        else
            B[ k ] ← A[ j++ ]
```

i指的那个小或者右边部分已经走完了

Recursion: Binary Search (recursive / iterative), Merge Sort

Improving Merge Sort implementations:

0. Avoid sentinels 1. Do not create new arrays in each call to merge.

merge(A, B, p, q, r): A[p..q] is sorted, A[q+1..r] is sorted

// Merge into A[p..r], use B for temp work.

Copy A[p..r] into B[p..r]

for k ← p to r do B[k] ← A[k].

i ← p j ← q+1

for k ← p to r do

if [j > r] or ([i ≤ q] and B[i] ≤ B[j]) ← Left subarray is not finished

Right subarray is finished

A[k] ← B[i]

i++

else A[k] ← B[j]

j++

Fibonacci numbers

Fib(0) = 0 Fib(1) = 1 Fib(n) = Fib(n-1) + Fib(n-2) if n > 1.

```
Fib(n)
  if n = 0 or n = 1 then return n
  else return Fib(n-1) + Fib(n-2)
```

← RT = $\theta(c^n)$, $c = \frac{\sqrt{5}+1}{2}$

Dynamic Program.

store Fib(i) in F[i]

Find Fib(i): i = 2 ... n

```
F[0] ← 0   F[1] ← 1
for i ← 2 to n do
    F[i] ← F[i-1] + F[i-2]
return F[n]
```

RT = $O(n)$

Better? Matrix form of Fib:

$$\begin{aligned} Fib(n) &= Fib(n-1) + Fib(n-2) \\ Fib(n-1) &= Fib(n-1) \end{aligned}$$

$$V_n = \begin{pmatrix} Fib(n) \\ Fib(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} Fib(n-1) \\ Fib(n-2) \end{pmatrix} = V_{n-}$$

$$V_n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} V_{n-1}$$

After i iterations,

$$V_n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^i V_{n-i}$$

Choose $i = n-1$:

$$V_n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} V_1$$

$$V_1 = \begin{pmatrix} Fib(1) \\ Fib(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$V_n = \begin{pmatrix} Fib(n) \\ Fib(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

V_n = Power$\left(\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, n-1\right)$ $* \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ ← matrix *

V_n can be computed in $O(\log n)$ time.

Euler Tours

Let $G = (V, E)$ be a graph (directed or undirected).

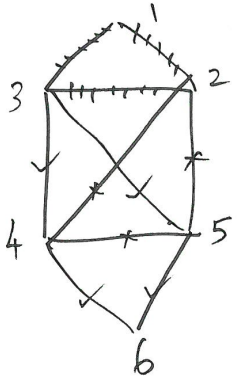
A tour is a walk along the edges of G that starts and ends in same place.
(tour - edges can be repeated, vertices can be visited any number of times)

An Euler (pronounced "Oiler") tour is a tour of G that uses every edge exactly once. If G has an Euler tour, it is Eulerian.

Theorems:

1. If an undirected graph G is connected, and every vertex has even degree, then it is Eulerian.
2. If a directed graph G is strongly connected, and every vertex has indegree = outdegree, then it is Eulerian.

Example:



~~1~~ — ~~2~~ — 3 — 1 - tour # 1

✓ 2 — 5 - 4 - 2 - tour # 2

✓ 3 - 4 - 6 - 5 - 3 - tour # 3

stitch tours together

tour #2 tour # 3

1 - [2 - 5 - 4 - 2] - [3 - 4 - 6 - 5 - 3] - 1

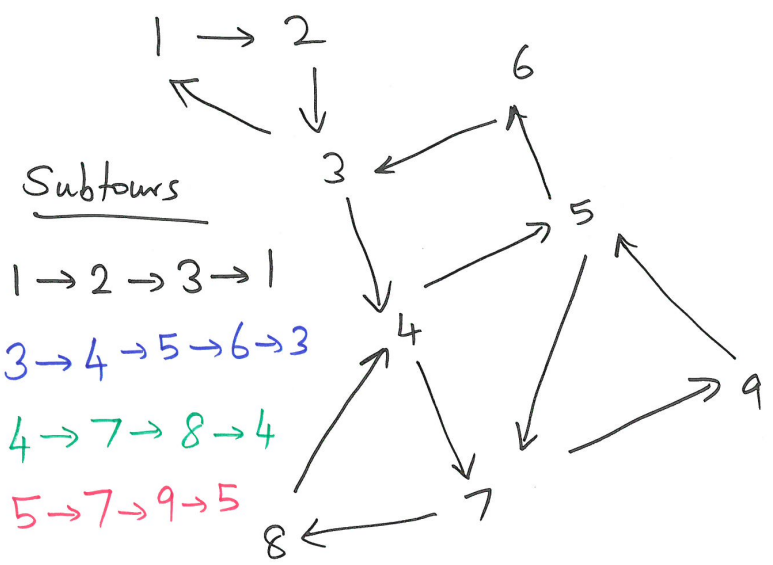
Project LP2 (easy) — upto 5 ec

Input: Directed graph $G=(V, E)$

Output: Euler tour of G

} starter code will be given

Directed graph Example: Euler Tour:



Subtours

1 → 2 → 3 → 1

3 → 4 → 5 → 6 → 3

4 → 7 → 8 → 4

5 → 7 → 9 → 5

Output:

1 → 2 → 3 → 4 → 7 → 8 → 4 → 5 → 7 → 9 → 5 → 6 → 3 → 1