```
// Helper method
findTour( u, tour)
// Find a subtour starting at u
// tour is a initially empty list
// Output = tour
    while u has another unexplored
                    outgoing edge do
        Let e be next outgoing edge
         e=(u,v)          from u
        tour.add(e)
        u ← v
    ~~return tour~~.
```

1: ~~2~~
2: ~~3~~
3: ~~4~~, ~~1~~
4: ~~5~~, ~~7~~
5: ~~7~~, ~~6~~
6: ~~3~~
7: ~~8~~, ~~9~~
8: ~~4~~
9: ~~5~~

findTour(1, tour):
tour = (1,2)(2,3)(3,4)(4,5)(5,7)(7,8)(8,4)(4,7)(7,9)(9,5)(5,6)(6,3)(3,1)
u = ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~7~~ ~~8~~ ~~4~~ 7 ~~9~~ ~~5~~
~~6~~ ~~3~~ 1

~~findTour(5, tour)~~
~~tour: (5,6)~~
~~u: 5~~ 6

```
findTour( ):
// Break g into a set/list
        of subtours

    findTour (start, l1)
    while there exists a node u with unexplored outgoing edges
                                                    do
        findTour ( u, l2 )
```

No hash tables or list of lists
or indexing is needed.

```
stitchTours ( )         mapping:  u → tour that
                                     starts at u
    explore (start)
    T = initially empty list

explore (u)      // append tour starting at u to T

    tmp ← u
    for ( Edge e : u's tour) do
        ~~T~~
        T.add(e)
        tmp ← e.otherEnd (tmp)
        if there is an unexplored tour
           starting at tmp
           explore (tmp)

    Output is in T          (T = class object or
                                 passed as a
                                   parameter)
```

Every edge is visited once
$RT = O(|E|)$

Applications of Euler tours

City management — scheduling of trash pickup,
mail delivery, street cleaning

Quick Sort : another sorting algorithm.

Helper function: partition

Partition(A, p, r) // A[p..r]

// Rearrange A[p..r] such that

// $A[p..q-1] \le x$, $A[q] = x$, $A[q+1..r] > x$

$\le x$	x	$> x$

x = pivot

Q: What is x? How is it chosen?

- Deterministic choice of x — bad

Good choice: choose ~~x~~ uniformly at random from $A[p..r]$.

---

i = index of x

Exchange $x = A[i] \leftrightarrow A[r]$ $\quad A[r] = x$

$i \leftarrow p-1$ $\quad$ LI: $A[p..i] \le x$

$A[i+1..j-1] > x$

for $j \leftarrow p$ to $r-1$ do $\quad A[j..r-1]$ - unprocessed

if $A[j] \le x$ then

i++

Exchange $A[i] \leftrightarrow A[j]$

Exchange $A[i+1] \leftrightarrow A[r]$ // bring pivot back in

Return $i+1$

How to get good RT when A has duplicates:

1. Modify partition to split A into 3 parts:

$< x$	$= x$	$> x$

$\underbrace{= x}$ no sorting needed

2. Treat $= x$ elements alternately as $< x$ and $> x$

or randomly decide between $< x$, $> x$

3. Dual-pivot partition:

x_1 [ ] x_2 $\quad x_1 \le x_2$

$\downarrow$

$\boxed{< x_1}$ x_1 $\boxed{x_1..x_2}$ x_2 $\boxed{> x_2}$

not sorted recursively if $x_1 = x_2$

Quick sort algorithm

partition(A, p, r):
```
Select i uniformly at random in [ p···r ]
Exchange A[ i ] ↔ A[ r ]
x ← A[ r ]      // Pivot element
i ← p − 1
// LI: A[ p···i ] ≤ x,  A[ i+1···j−1 ] > x,
//     A[ j···r−1 ] is unprocessed, A[ r ] = x.
for j ← p to r − 1 do
    if A[ j ]  ≤  x  then
        i ← i+1
        Exchange A[ i ] ↔ A[ j ]
// Bring pivot back to the middle
Exchange A[ i+1 ] ↔ A[ r ]
// A[ p···i ] ≤ x, A[ i+1 ]  = x, A[ i+2···r ] > x
return i+1
```

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

quickSort(A, p, r): // Sort A[p···r]
```
if p < r then
    q ← partition(A, p, r)
    quickSort(A, p, q − 1)
    quickSort(A, q + 1, r)
```

经过partition()使得q前的比q小，后的比q大

There is another partition algorithm given by Hoare:

partition2(A, p, r):
```
Choose x uniformly at random from A[ p···r ]
i ← p − 1,  j ← r + 1
// LI: A[ p···i ] ≤ x,  A[ j···r ] ≥ x
while true do
    do { i++ } while A[ i ] < x
    do { j−− } while A[ j ] > x
    if i ≥ j then
        return j
    Exchange A[ i ] ↔ A[ j ]
    ~~i++ ,  j−−~~
```

If quickSort calls this version of partition, the second recursive call should be changed to quickSort(A, q, r).

Dual-pivot partition (Yaroslavskiy)

Choose 2 elements of A[p···r] uniformly at random, and exchange them as: A[p] = x_1, A [r] = x_2, $x_1 \le x_2$.

Initially, k = i = p + 1, j = r − 1. S_1= A[p+1···k − 1]. S_2= A[k···i − 1]. S_3= A[j+1···r − 1].

Loop Invariant:

	S_1	S_2		S_3	
x_1	$< x_1$	$x_1 - x_2$	unprocessed	$> x_2$	x_2
p		k	i	j	r

Unprocessed elements are processed from both ends:

Case 1: $x_1 \le$ A[i] $\le x_2$. S_2 grows by 1. i++

Case 2: A[i] < x_1. S_1 grows by 1. Exchange A[i] with A[k], the left-most element of S_2. i++

Case 3: A[j] > x_2. S_3 grows by 1. j−−

swap(A[k],A[j]); swap(A[i],A[j]);

Case 4: A[i] > x_2, A[j] < x_1. Circular swap A[k] → A[i] → A[j] → A[k]. k++ i++ j−−

Case 5: A[i] > x_2, $x_1 \le$ A[j] $\le x_2$. S_2 and S_3 grow by 1 each. Exchange A[i] ↔ A[j]. i++ j−−

At the end of the algorithm, exchange A[p] ↔ A[k − 1], and, A[j+1] ↔ A[r].

dPQuickSort:

dualPivotPartition

dPQuickSort S_1, dPQuickSort S_3

if $x_1 \neq x_2$ then dPQuickSort S_2

Improvements: handle sizes below some threshold with another algorithm. One of the best implementations of Quick sort uses dual-pivot partition, with hand-coded, loopless sorting algorithm for n < 8.