**Overview**

What is recursion?
- When one function calls ITSELF directly or indirectly.

Why learn recursion?
- New mode of thinking.
- Powerful programming tool to solve a problem by breaking it up into one (or more) smaller problems of similar structure.
  - “Divide et impera”
  - “Veni, vidi, vici”

---

**Quicksort**

Quicksort.
- Partition array so that:
  - some partitioning element \( a[m] \) is in its final position
  - no larger element to the left of \( m \)
  - no smaller element to the right of \( m \)

- Sort each “half” recursively.
Quicksort

Partition array so that:
- some partitioning element \(a[m]\) is in its final position
- no larger element to the left of \(m\)
- no smaller element to the right of \(m\)

Sort each “half” recursively.

```c
void quicksort(char a[], int left, int right) {
    int m; if (right > left) {
        m = partition(a, left, right);
        quicksort(a, left, m - 1);
        quicksort(a, m + 1, right);
    }
}
```

Quicksort : Implementing Partition

```c
int partition(char a[], int left, int right) {
    int i = left - 1;    /* left to right pointer */
    int j = right;     /* right to left pointer */
    while(1) {
        while (a[++i] < a[right])
            ;
        while (a[right] < a[--j])
            if (j == left)
                break;
        if (i >= j)
            break;
        swap(a, i, j);
    }
    swap(a, i, right);
    return i;
}
```

main()
```c
#include <stdio.h>
#define N 14

int main(void) {
    char a[] = "pseudomythical";
    printf("Before: %s\n", a);
    quicksort(a, 0, N-1);
    printf("After:  %s\n", a);
    return 0;
}
```

swap()
```c
void swap(char a[], int i, int j) {
    char t;
    t = a[i]; a[i] = a[j]; a[j] = t;
}
```
Quicksort: Performance

**Quicksort vs. Insertion sort.**

<table>
<thead>
<tr>
<th>Insertion Sort</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>computer</strong></td>
<td>thousand</td>
</tr>
<tr>
<td>home pc</td>
<td>instant</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
</tr>
<tr>
<td><strong>QuickSort</strong></td>
<td>thousand</td>
</tr>
<tr>
<td>instant</td>
<td>0.3 sec</td>
</tr>
<tr>
<td>instant</td>
<td>instant</td>
</tr>
</tbody>
</table>

Stay tuned: Lecture T5.

---

**Drawing a Dragon Curve**

*Use simple “turtle graphics.”*

- F: move turtle forward one step (pen down).
- L: turn left 90°.
- R: turn right 90°.

**Example.**

- F L F L F

---

**Dragon (Jurassic Park) Curve**

Fold a wire in half n times. Unfold to right angles.

- n = 0
- n = 1
- n = 2
- n = 3
- n = 4

---

Drawing a Dragon Curve

*Use simple “turtle graphics.”*

- F: move turtle forward one step (pen down).
- L: turn left 90°.
- R: turn right 90°.

**Example.**

- dragon(0): F
- dragon(1): F L F
- dragon(2): F L F L F R F
- dragon(3): F L F L F R F L F R F R F

**dragon(3)**

"backwards" dragon(3): reverse string, switch L and R
Recursive Dragon Curve Program

A dragon curve of order n is:
- Dragon curve of order n-1.
- Move left.
- Dragon curve of order n-1 backwards (switch L and R).

```
void dragon(int n) {
    if (n == 0)
        F();
    else {
        dragon(n-1);
        L();
        nogard(n-1);
    }
}
```

Drawing a Dragon Curve

To get nogard(n):
- dragon(2): F L F L F R F
- nogard(2): F L R F R F
- dragon(3): F L F L F R F L F L F R F F
- nogard(3): F L R F R F L R F R F L F R F R F

```
void nogard(int n) {
    if (n == 0)
        F();
    else {
        dragon(n-1);
        R();
        nogard(n-1);
    }
}
```

Unwinding Tail Recursion

Replace nogard() with its results.

```
void nogard(int n) {
    if (n == 0)
        F();
    else {
        dragon(n-1);
        R();
        dragon(n-2); R();
        nogard(n-2);
        ...
        dragon(0);
        R();
        F();
    }
}
```

```
void nogard(int n) {
    if (n == 0)
        F();
    else {
        for (k = n-1; k >= 0; k--)
            R();
        dragon(k);
        F();
    }
}
```

Alternate Dragon

Replace call to nogard() by non-recursive version.

```
void dragon(int n) {
    int k;
    for (k = n-1; k >= 0; k--)
        R();
    dragon(k);
    F();
}
```

```
void dragon(int n) {
    int k;
    if (n == 0)
        F();
    else {
        dragon(n-1);
        L();
        for (k = n-2; k >=0; k--)
            dragon(k);
        R();
        F();
    }
}
```
Enumerating All Permutations

Enumerate all permutations of a set of elements.

- **N elements** ⇒ **N! possibilities**
- If elements named a, b, c, then 6 possible permutations are:
  - abc, acb, bac, bca, cab, cba.

**Key idea:** permutations of abcde are one of the following:
- End with a preceded by one of 4! permutations of bcde.
- End with b preceded by one of 4! permutations of acde.
- End with c preceded by one of 4! permutations of abde.
- End with d preceded by one of 4! permutations of abce.
- End with e preceded by one of 4! permutations of abcd.

Reduces enumerating permutations of N elements to enumerating permutations of N-1 elements.

**Inelegant Solution (for N = 3)**

```c
#include <stdio.h>
#define N 3

int main(void) {
    char a[] = "abc";
    int i, j, k;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                if (i != j && i != k && j != k)
                    printf("%c%c%c
", a[i], a[j], a[k]);
    return 0;
}
```

**Recursive solution for trying all permutations:**
- Array a[] stores current permutation.
- Initially a[] = "abcde"

```c
#include <stdio.h>

void swap(char a[], int i, int j) {
    char t;
    t = a[i]; a[i] = a[j]; a[j] = t;
}

void enumerate(char a[], int n) {
    int i;
    if (0 == n)
        printf("%s
", a);
    else
        for (i = 0; i < n; i++)
            swap(a, i, n-1);
        enumerate(a, n-1);
}

int main(void) {
    char a[] = "abcde";
    enumerate(a, 5);
    return 0;
}
```

```bash
% a.out
bcacbacabacb
bac
abc
```
Application: Traveling Salesperson Problem

Given N points, find shortest tour connecting them.
- Brute force: try all N! possible permutations.

Recursive solution for finding best TSP tour.
- Store coordinates of points in a[].
- Replace printf() with checklength().
- Takes N! steps.
- No computer can run this for N ≥ 100.
  - 100! > $10^{150}$.

Is there an efficient way to do this computation?