• An object is said to be recursive if it partially consists or is defined in terms of itself.

• Recursion is a powerful means of mathematical definition.

Example: Recursive definition of a set $E$
  1. $2$ is in $E$ \textbf{(Base Case)}
  2. If $x$ is in $E$ then $x+2$ is also in $E$ \textbf{(Recursive Case)}

$E = \{2, 4, 6, 8, \ldots\}$ even positive numbers.

• A recursive definition has 2 parts
  1. Base Case: A statement that can be resolved directly
  2. Recursive case: in which the object is defined in terms of itself

Example: The set of all strings of balanced parentheses.

\texttt{''} \texttt{''} \texttt{''(())''} \texttt{ok} \texttt{''(())''} \texttt{no}

1. \texttt{''} is in the set:

2. If $s_1$ is in the set, then so is \texttt{''(s_1)''}

3. If $s_1$ and $s_2$ are in the set, then so is \texttt{''s_1 s_2''}

\texttt{''(())(()''}

(1) by 1) then so \texttt{''(())''} by 2) then so \texttt{''(())''} by 3)
Then so \[ \text{by 2) \} \]

* The power of recursion is that it allows the definition of infinite objects by finite means.

**Recursive Algorithms:**

- **Base Case:** an instance of the problem that can be solved directly.
- **Recursive Case:** decomposing the problem into simpler instances. Constructing a solution from the solutions of the simpler instances.

**Example:** "the triomino problem"  
```
  1
  \_\_\_\_
  \_/\_/\_/
```

Problem: Given a board of size \( 2^n \) where there is one hole of size 1x1, cover the board with triominos.

```
\_\_\_\_
\_/\_/\_/
```

Recursive approach: Base case.
Recursive approach: Base case.

\[ n=1 \]

Recursive approach: Recursive case

- Split in 4
- Place a trinion across the board without a hole.

\[ n=3 \]

Recursive Programs:

\[
\text{foo}(x) \\
\{ \\
\text{if } x \text{ is the base case} \\
\quad \text{return direct solution} \\
\text{else} \\
\quad \text{decompose } x \text{ into sub-problems } x' \\
\quad \text{foo}(x') \\
\quad \text{return solution constructed from solution to } x' \\
\}
\]

Note: Recursion has an (undeserved) bad rep.
1) “Every recursion can be re-written as iteration”
“Every iteration can be re-written as recursion”

2) Bag Examples:

\[
\text{fibonacci}
\]

\[
\begin{align*}
\text{fib}(1) &= 1 \\
\text{fib}(2) &= 1 \\
\text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \quad n>2
\end{align*}
\]

\[
\text{fib}(n)
\]

\{
\begin{align*}
\text{if } ( n==1 \text{ || } n==2 ) \\
& \quad \text{return } 1; \\
\text{return } \text{fib}(n-1) + \text{fib}(n-2)
\end{align*}
\}

**Good Example:** Power \((x,y) = x^y\)

\[
\text{pow}(x, y)
\]

\{
\begin{align*}
r &= 1; \\
\text{for } (i = 0; i < y; i++) \\
& \quad r = r \times x; \\
\text{return } r;
\end{align*}
\}

\[
y = \frac{y}{2} \times \frac{y}{2} \times \frac{y}{2} \ldots \\
y \text{ is } \mathcal{O}(n)
\]

\[
\text{pow}(x, y)
\]

\{
\begin{align*}
& \text{if (y == 0 ) return 1; } \\
& \quad r = \text{pow}(x, y/2); \\
& \text{if ( y is even ) } \\
& \quad \text{return } r \times r;
\end{align*}
\}

\[
\text{pow}(2, 4) = 16
\]

\[
\text{pow}(2, 2) = 4
\]
\[
r = \text{pow} (x, y/2);
\]
if \(y\) is even
\[
\text{return } r \times r;
\]
else
\[
\text{return } r \times r \times x;
\]
}

**Multiplications:** \(2 \times \log_2 y\) is \(O(\log n)\)

---

**Recursive Backtracking**

- Many problems do not have a “fixed rule” solution
- Strategy = Decompose problem into a sequence of trial & error tasks.

**Example:** The n-queens problem.

- Given an \(n \times n\) chessboard, place \(n\) queens on the board such that the queens do not attack each other.

\[n=4\]

1850's Gauss: no general rule

\[
16 \times 15 \times 14 \times 13 = 43,680
\]

- trick one queen per row:
  \[
  4 \times 4 \times 4 \times 4 = 4^4 = 256
  \]
  \[
  6^6 = 46656
  \]
  \[
  8^8 = 16,777,216
  \]

- trick one queen per row:
- place one queen at a time and stop when conflicts arise.
On Object Oriented Programming

- class Board
- class Queen, derived from class piece.

Then ???

Algorithm:

```cpp
try_queen(i)
repeat
  place i'th queen
  if no more queens to place.
    return success!
  else
    try = try_queen(i+1)
    if try is successful
      return success!
    else
      retract queen;
  until out of places for i'th queen
return fail!
```

Refining algorithm:

```cpp
bool try_queen(int row, Board board, int n)

  bool try:
  for (int col = 0; col < n; col++)
    if valid(row, col, board)
      record(board, row, col)
      if (col == n-1)
        return true;
  return false;
```
else
    try = try-queen(row+1, board, n)
    if (try)
        return true;
    else
        retract(board, row, col)

3
    return false.

the board

1) Idea 2d Array:
   2) 1-d Array; store column

General form:

Try
initialize choices
do
    select choice
    if choice is acceptable
        record choice
        if solution complete
            return success!
        else
            try next step
            if next step succeeds
                return success!
            else
                retract choice.
    while more choices available.
while move choices available.
retract choice.
return fail.