CS39N The Beauty and Joy of Computing

Lecture #11 Recursion II

UC Berkeley
Computer Science
Lecturer SOE
Dan Garcia

2009-11-02

MUSCLE-BOUND COMPUTER INTERFACE

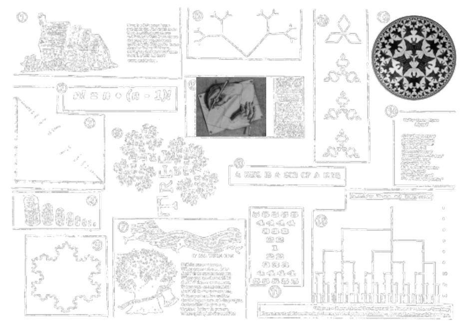
Researchers at Microsoft, UW and U Toronto have come up with a technique to interact with a computer by flexing muscles (sensor electrodes on forearm)

www.technologyreview.com/computing/23813/

---

doi.acm.org/10.1145/330908.331877

You already know it!



---

www.catb.org/~esr/jargon/html/R/recursion.html
www.nist.gov/dads/HTML/recursion.html

Definition

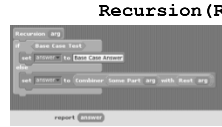
- Recursion: (noun) See recursion. ☺
- *An algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task*
- Recursive solutions involve two major parts:
  - Base case(s), the problem is simple enough to be solved directly
  - Recursive case(s). A recursive case has three components:
    - Divide the problem into one or more simpler or smaller parts
    - Invoke the function (recursively) on each part, and
    - Combine the solutions of the parts into a solution for the problem.
- Depending on the problem, any of these may be trivial or complex.

Linear Functional Pattern

- Functional programming
  - It's all about the reporter return value
  - There are no side-effects
- Recursion(*arg*) {

```
Recursion(arg) {
  if(base_case_test) {
    return(base_case_answer)
  } else {
    return(Combiner(SomePart(arg),
                    Recursion(Rest(arg))))
  }
}
```

- Base case(s)
- Recursive case(s)
  - Divide
  - Invoke
  - Combine



---

en.wikipedia.org/wiki/Factorial

Linear Functional Example: `n!`

n	n!
0	1
1	1
2	2
3	6
4	24
5	120

$$n! = \prod_{k=1}^{n} k \qquad \text{for all } n \in \mathbb{N} \geq 0.$$

- **Factorial(n) = n! Inductive definition:**
  - n! = 1 , n = 0
  - n! = n * (n-1)!, n > 0
- **What are…**
  - `base_case_test`
    - `n == 0`
  - `base_case_answer`
    - `1`
  - `SomePart(arg)`
    - `n`
  - `Rest(arg)`
    - `n-1`
  - `Combiner: *`

```
Recursion(arg) {
  if(base_case_test) {
    return(base_case_answer)
  } else {
    return(Combiner(SomePart(arg),
                    Recursion(Rest(arg))))
  }
}
```

```
Factorial(n) {
  if(n == 0) {
    return(1)
  } else {
    return(n * Factorial(n-1))
  }
}
```

Let's now trace…

---

en.wikipedia.org/wiki/Fibonacci_number
www.ics.uci.edu/~eppstein/161/960109.html

Non-linear Functional Example: `Fib`

n	F(n)
0	0
1	1
2	1
3	2
4	3
5	5

Leonardo de Pisa aka, Fibonacci

- **Inductive definition:**

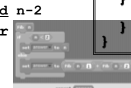
$$F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

- **What are…**
  - `base_case_test`
    - `n <= 1`
  - `base_case_answer`
    - `n`
  - `SomePart(arg)`
    - `0`
  - `Rest(arg)`
    - `n-1` and `n-2`
  - `Combiner`
    - `+`

```
Recursion(arg) {
  if(base_case_test) {
    return(base_case_answer)
  } else {
    return(Combiner(SomePart(arg),
                    Recursion(Rest(arg))))
  } }
```

```
Recursion(arg) {
  if(base_case_test) {
    return(base_case_answer);
  } else {
    return(Combiner(SomePart(arg),
                    Recursion(Rest1(arg)),
                    Recursion(Rest2(arg)),
                    …
                    Recursion(Restn(arg)) )) } }
```

```
Fib(n) {
  if(n <= 1) {
    return(n)
  } else {
    return(Fib(n-1)+Fib(n-2))
  }
}
```



Let's now: trace… (gif from Ybungalobill@wikimedia)

Authoring: Trust the Recursion!

- **When authoring recursive code:**
  - The base is usually easy: "when to stop?"
  - In the recursive step
    - How can we break the problem down into two:
      - A piece I can handle right now
      - The answer from a smaller piece of the problem
    - Assume your self-call does the right thing on a smaller piece of the problem
    - How to combine parts to get the overall answer?
- **Practice will make it easier to see idea**

http://doi.acm.org/10.1145/1473195.1473219
mathworld.wolfram.com/RenyisParkingConstants.html

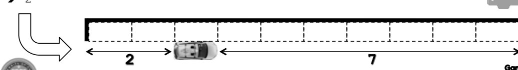
Now you try one…

- **Want to park unit-length cars on a block 10 units wide.**

- **In the ideal case, you can get 10 cars in.**
  - Assume no wiggling needed, they just drop in
- **Assuming cars arrive & park randomly on the open spaces, how many cars can park on avg?**
- **With a partner, write `park(room)` → # of cars**

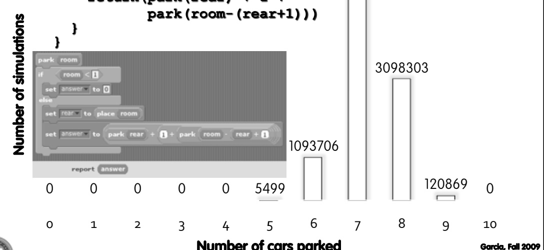
  - Answer will be `park(10)`
  - Given: `place(room)` randomly places rear bumper on a place from `0` to `room-1` and returns location. Assumes `room` > `1`!

E.g., place(10) → 2



10^7 trials; avg = 7.2235337

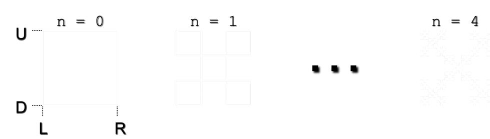
```
park(room) {
  if(room < 1) {
    return(0)
  } else {
    rear = place(room)
    return(park(rear) + 1 +
           park(room-(rear+1)))
  }
}
```



en.wikipedia.org/wiki/Fractal

Fractal Beauty in Nature

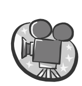
- **Fractals are self-similar objects**
  - They appear in nature: rivers, coastlines, mountains, snow
- **They are perfect for describing with recursion**
  - Same ideas apply: base case + recursive case
  - Tip: look at n=0 and n=1 case; n=∞ is often hard to decrypt
- **How to write (pseudo)code for the Sierpinski Square:**
  - `SierpinskiSquare(L,R,U,D,n)` given `DrawRectangle (L,R,U,D)` and `OneThird(from,to)`



Sierpinski Square

```
SierpinskiSquare(L,R,U,D,n) {
  if(n == 0) {
    DrawRectangle(L,R,U,D)
  } else { // We shorten OneThird to OT here…
    SierpinskiSquare(L,OT(L,R),U,OT(U,D),n-1) // NW
    SierpinskiSquare(OT(R,L),R,U,OT(U,D),n-1) // NE
    SierpinskiSquare(OT(L,R),OT(R,L),OT(U,D),OT(D,U),n-1)
    SierpinskiSquare(L,OT(L,R),OT(D,U),D,n-1) // SW
    SierpinskiSquare(OT(R,L),R,OT(D,U),D,n-1) // SE
  }
}
```

- **This is procedural recursion -- purpose is side-effect, not return value**
  - **Sometimes we want a side-effect AND a return value…**



Conclusion

- **Many flavors, patterns**
  - Functional
  - Procedural
- **Inductive definitions lead naturally to recursion**
- **Recursion simpler code**
  - Fractals scream for it
- **Thinking recursively**
  - Breaking problem down by trusting recursion, building off of that