Control Flow Paradigms

Iterative algorithms proceed forward towards the solution one step at a time.

- repetition through loops

Recursive algorithms have friends solve smaller instances of the same problem (subproblems).

- repetition through recursion

Recursion

Three parts –

- simple *base case(s)* that can be solved directly, without recursion
- *recursive case(s)* which reduce the problem to one or more smaller instances of the same problem (subproblems)
- a *combine step* which uses the results of the recursive calls to construct the answer

```
solve(problem P):
    if P is simple (base case)
        return direct answer
    else
        break P into smaller problem(s)
        recursively solve smaller problem(s)
        combine the results
```


How to Design Recursive Algorithms

- establish the problem
- identify avenues of attack
- define the algorithm
- show termination and correctness
- determine efficiency

Common Recursive Patterns

Identify avenues of attack.

- *Paradigms and patterns.*

Recursive algorithms can be characterized by the number and size of the subproblems.

- 1-friend solutions
- 2+-friend solutions

Recursive algorithms can also be categorized by the nature of the subproblems.

- solve a smaller problem
- solve the rest of the problem

Recursive Patterns

Characterized by the number and size of subproblems –

- 1 friend – can often easily be written as iterative instead
  - *constant amount* – subproblem is smaller by a fixed number of elements (typically 1)
    - e.g. $a^n = a\ a^{n-1}$ or n! = n (n-1)!
  - *constant factor* – subproblem is a fixed fraction of the size (typically ½) – "decrease and conquer"
    - e.g. binary search
    - e.g. $a^n = (a^{n/2})^2$ if n is even, a $(a^{(n-1)/2})^2$ if n is odd
  - *variable factor* – subproblem is smaller, but the size of the reduction varies
    - e.g. gcd(m,n) = gcd(n,m mod n)

Recursive Patterns

Characterized by the number and size of subproblems –

- 2+ friends
  - *divide-and-conquer* – split into $b \geq 2$ subproblems of size n/b (b is typically 2)
  - *case analysis* – each friend gets a subproblem resulting from a different alternative

Recursive Patterns

Characterized by the nature of the subproblems –

- *solve a smaller problem*
  - subproblem is treated as a standalone problem to be solved
  - typical of 1-friend and divide-and-conquer patterns
- *solve the rest of the problem*
  - solving the subproblem completes a partial solution already begun
  - typical of case analysis patterns

How to Design Recursive Algorithms

Define the algorithm. The core of a recursive algorithm is defining the subproblems.

- *Size.*
  What is the size of the problem, and what constitutes a smaller problem? What is the simplest/smallest instance of the problem?
- *Generalize / define subproblems.*
  Define the subproblem — the task, the input, and the output. Write this as a function definition where parameters and return values are explicitly identified.
- *Base case(s).*
  Address how to solve the smallest problem(s). This is often trivial, or is solved via brute force.

- *Main case.*
  Address how to solve a typical large problem instance (one that is not a base case). Specify how many friends are needed and what subproblem is handed to each friend, as well as how the results the friends hand back are combined to solve the problem.
- *Top level.*
  The top level puts the context around the recursion.
  - *Initial subproblem.*
    Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem.
  - *Setup.*
    Whatever must happen before the initial subproblem is solved.
  - *Wrapup.*
    Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

How to Design Recursive Algorithms

Define the algorithm. The core of a recursive algorithm is defining the subproblems.

- *Special cases.*
  Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).
- *Algorithm.*
  Assemble the algorithm from the previous steps and state it.
  There shouldn't be new elements here, instead bring together the base case(s), main case, and top level along with any handling needed for special cases and state the whole algorithm.

How to Design Recursive Algorithms

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.* Show that the recursion — and thus the algorithm — always terminates.
  - *Making progress.*
    Explain why what each of your friends get is a smaller instance of the problem.
  - *The end is reached.*
    Explain why a base case is always reached.

- key things to address
  - making progress – show that your friends are guaranteed subproblems at least one smaller than you got
    - check for special cases or bugs that mean subproblems aren't really smaller
  - reaching the end – show that base cases can't be skipped
    - e.g. fib(n) = fib(n-1)+fib(n-2) with base case fib(1) = 1
      - fib(2) = fib(1)+fib(0)

How to Design Recursive Algorithms

- *Correctness.* Show that the algorithm is correct.
  - *Establish the base case(s).*
    Explain why the solution is correct for each base case.
  - *Show the main case.*
    Assume that the friends return the correct results for their subproblem, and explain why the correct answer is then produced from those results.
  - *Final answer.*
    Explain why the top level — the setup plus a correct solution to the initial subproblem followed by the wrapup — means that the final result is a correct answer to the problem.

- proof by induction
  - explain why the base case(s) correctly solve subproblems of that size
  - for the recursive case(s), assume that the friends return correct solutions for their subproblems – address
    - why you correctly split the problem into subproblems
    - why you correctly combine the friends' solutions into your solution

How to Design Recursive Algorithms

Determine efficiency. Evaluate the running time and space requirements of the algorithm.

- *Implementation.*
  Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.
- *Time and space.*
  Assess the running time and space requirements of the algorithm given the implementation identified.

Recursive algorithms tend to lead to recurrence relations in one of two forms:

split off *b* elements
$T(n) = a\,T(n-b) + f(n)$ where $f(n) = 0$ or $\Theta(n^c \log^d n)$

divide into subproblems of size *n/b*
$T(n) = a\,T(n/b) + f(n)$ where $\Theta(n^c \log^d n)$

- *Room for improvement.*
  Are the targets met? Is it necessary to do better? If improvements in running time and/or space are needed, identify possible avenues for improvement.