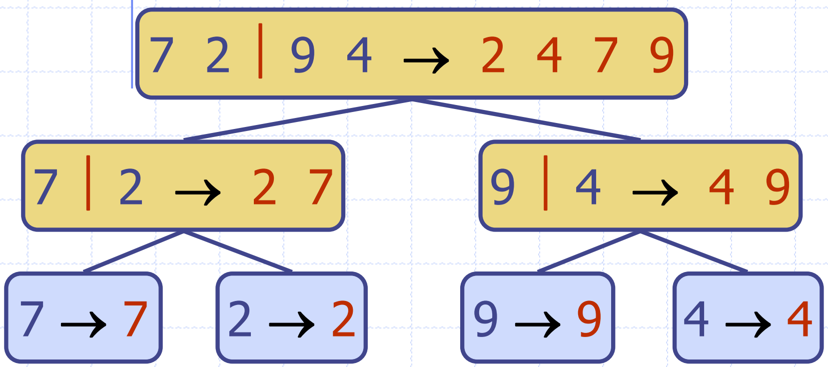
Lesson 4
Recursion: Collapsing Infinity To a Point



Wholeness of the Lesson

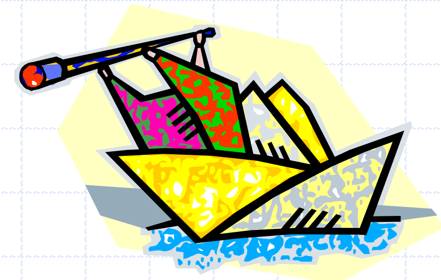
Recursive algorithms keep reducing the size of the input instances until a base case is reached, then the solution is computed from the base case up to the solution for the whole problem. Maharishi describes the process of creation as a self-referral process within pure consciousness that unfolds sequentially.

Algorithm Design Strategy

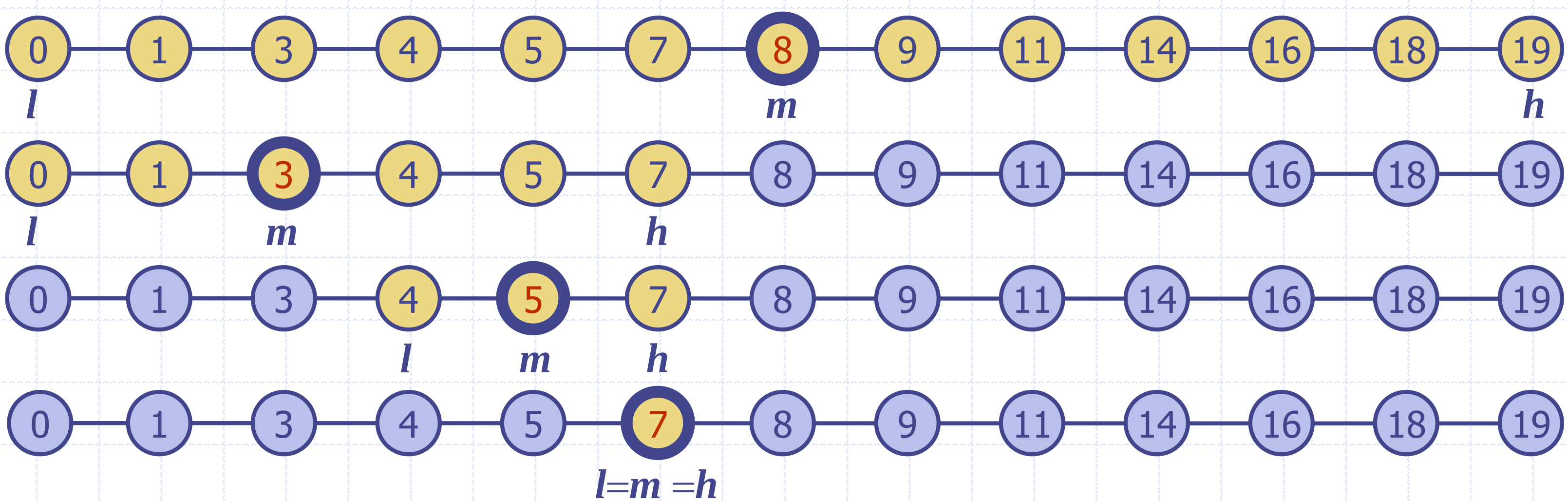
- Prune and Search
  - AKA Decrease and Conquer
  - Examples:
    - **binary search**
    - quick select (randomized prune and search)

- Randomized Algorithms (later)
  - Quick Sort and Quick Select

Binary Search (§ 3.1.1)



- Binary search performs operation findElement(k) on a dictionary implemented by means of an array-based sequence, sorted by key
  - similar to the high-low game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: findElement(7)



Binary Search Algorithm (iterative)

Algorithm BinarySearch(*S, k*):

Input: *A sorted array S storing n items*

Output: *An element of S with value k.*

```
low ← 0
high ← S.length - 1
while low ≤ high do
    mid ← (low + high)/2
    if k = S[mid] then
        return S[mid]
    else if k < S[mid] then
        high ← mid - 1
    else
        low ← mid + 1
return NO_SUCH_KEY
```

Binary Search Algorithm (one key comparison per iteration)

Algorithm BinarySearch(*S, k*):

Input: *A sorted array S storing n items*

Output: *An element of S with value k.*

```
low ← 0
high ← S.size() - 1
mid ← 0
while low < high do
     mid ← (low + high)/2
     if  k < S[mid]   then
          high ← mid-1
      else
          low ← mid
if S.size() > 0  /\  k = S[mid]   then
    return S[mid]
else
    return NO_SUCH_KEY
```

Recursive Programming

Basic Concepts

- Recognizing Recursion
  - When smaller or simpler instances form sub-constituents of the overall solution
    - E.g., when a function calls itself on smaller subproblem instances to solve the larger, global problem
- Theoretically, any problem that can be solved using iteration (while and for loops) can be solved using recursion (functional style is supported by functional languages)

Types of Recursion

- Linear recursion
- Tail recursion
- Multiple recursion
- Mutual recursion
- Nested recursion

Types of Recursion

- Linear recursion
  - When a method calls itself only once in the body of the function

```
Algorithm sumFirst(n)
    if n < 0 then Throw InvalidInputException
    if n = 0 then
        return 0
    else
        return n + sumFirst(n-1)
```

Types of Recursion

- Tail recursion
  - A special case of linear recursion in which a method calls itself only once but the call occurs as the last operation executed in the body of the method
  - Functional languages optimize tail recursive functions since there is no need to create a new stack frame (activation record)

```
Algorithm sumFirst(n)
    if n < 0 then Throw InvalidInputException
    return sumFirstHelper(n, 0)

Algorithm sumFirstHelper(n, s)
    if n = 0 then
            return s
    else
            return sumFirstHelper(n-1, n+s)
```

Types of Recursion

- Multiple recursion
  - When a function calls itself two or more times
- Example is MergeSort and QuickSort (next week)
- Functions that traverse a binary tree (later today)

- Must be careful because multiple recursion algorithms can quickly explode to $O(2^n)$

```
Algorithm Fib(n)
    if n = 0 then
        return 0
    else if n = 1 then
        return 1
    else
        return Fib(n-2) + Fib(n-1)
```

Types of Recursion

- Mutual recursion
  - When a group of methods repeatedly call each other until a base case is reached

```
Algorithm isEven(n)
    if n = 0 then
            return true
    else
            return isOdd(n-1)

Algorithm isOdd(n)
   if n = 0 then
            return false
    else
            return isEven(n-1)
```

Types of Recursion

- Nested recursion
  - When the argument to a recursive call is calculated via another recursive call
  - Sometimes called Double Recursion

```
Algorithm A(n, s) {Ackerman function}
    if n = 0 then
        return s + 1
    else if s = 0 then
        return A(n-1, 1)
    else {n > 0 and s > 0}
        return A(n-1, A(n, s-1))
```

Recursive Thinking

Think declaratively

1. Define the base cases
   - Instance(s) that can be calculated without using recursive calls
2. Decompose the problem into simpler or smaller instances of the original problem
   - A smaller/simpler instance must be moving toward one of the base cases (so the function terminates)
3. Create an induction diagram to determine what to do in addition to the recursive calls

Recursive Thinking (AKA Subgoal Induction)

Input Parameters → Final Solution

↓ ↑

Smaller Input Parms. → Smaller Solutions

Exercises

1. Write a pseudo code function, *isEven*(n) to <u>recursively</u> determine whether a natural number, n, is an even number.
2. Write a pseudo code function, *sum*(n), to <u>recursively</u> calculate the sum of the first n natural numbers.
3. Write a pseudo code function, *sum2*(n), to recursively sum the first n natural numbers but divide the problem in half and make two recursive calls.
4. Write a pseudo code function, power(x, k), that computes x^k. Can you do this in log k time?

Exercise on List

- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
- Accessor methods:
  - position first()
  - position last()
  - position before(p)
  - position after(p)
- Query methods:
  - boolean isFirst(p)
  - boolean isLast(p)
- Update methods:
  - swapElements(p, q)
  - replaceElement(p, o)
  - insertFirst(o)
  - insertLast(o)
  - insertBefore(p, o)
  - insertAfter(p , o)
  - remove(p)

Exercise:

- Write a recursive method to calculate the sum of the integers in a list of integers

Algorithm sum(L)

Hint: you also need a helper function with argument Position p

Algorithm sumHelper(L, p)

Binary Search Algorithm (recursive)

Algorithm BinarySearch(*S, k, low, high*):
Input: *A sorted array S storing n items*
Output: *An element of S with value k between indices low & high.*

```
if low > high then
    return NO_SUCH_KEY
else
    mid ← (low + high)/2
    if k = S[mid] then
        return S[mid]
    else if k < S[mid] then
        return BinarySearch(S, k, low, mid-1)
    else
        return BinarySearch(S, k, mid + 1, high)
```

Main Point

1. Any iterative algorithm can be computed using recursion, i.e., a function calling itself. In fact, the meaning of while- and for-loops are defined using recursive functions in programming language semantics (Denotational Semantics). Recursive algorithms keep reducing the size of the inputs instances until a base case is reached, then the solution is computed from the base case up to the solution for the whole problem.

   Science of Consciousness: Maharishi describes the process of creation as a self-referral process that unfolds sequentially. The dynamism of the unified field seems chaotic when studied at the macroscopic level, yet it is a field of perfect order, responsible for the order and balance in creation.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Recursion

1. A recursive algorithm calls itself repeatedly until it reaches the base case, then the results of these calls are combined to compute the final result.

2. There are different categories of recursive algorithms depending on how many and where the recursive calls occur, i.e., linear, tail, multiple, mutual, and nested (double) recursion.

3. *Transcendental Consciousness* is the self-referral field of *infinite correlation*, from which all of creation emerges, where “an impulse anywhere is an impulse everywhere.”

4. *Impulses within the Transcendental field*. The dynamic natural laws reside in this field, where the self-referral laws of nature govern all of creation.

5. *Wholeness moving within itself.* In Unity Consciousness, the field of action effortlessly unfolds as the play of one’s own Self, one’s own pure consciousness.