Ch 5.4: Recursive Algorithms

ICS 141: Discrete Mathematics for Computer Science I

Kyle Berney
Department of ICS, University of Hawaii at Manoa

Recursive Algorithms

- Definition: An algorithm is called recursive if it solves a problem by reducing it to an instance of the same problem with smaller input
- Correspondence to mathematical induction
  - Base Case(s)
    - Recursive algorithms explicitly solves the problem for "small" values
  - Inductive Case
    - Recursive algorithms solves the problem by assuming the algorithm correctly executes for smaller values

Recursive Factorial

- Base Case: $n = 0$
  - $0! = 1$
- Inductive Case: $n > 0$
  - $n! = n \cdot (n-1)!$

```
FACTORIAL(n)
    if n == 0
        return 1
    return n · FACTORIAL(n − 1)
```

Recursive Exponential

- Base Case: $n = 0$
  - $a^n = 1$
- Inductive Case: $n > 0$
  - $a^n = a \cdot a^{n-1}$

```
EXPONENT(a, n)
    if n == 0
        return 1
    return a · EXPONENT(a, n − 1)
```

Recursive Linear Search

- Base Case: $n = 0$
  - x is not in the array containing 0 elements
- Inductive Case: $n > 0$
  - If the first element is x, then return the index
  - Otherwise, recurse on the remaining $n - 1$ elements

```
LinearSearch(x, A[left ... right])
    if left > right
        return Not Found
    if x == A[left]
        return left
    else
        return LinearSearch(x, A[left + 1 ... right])
```

Recursive Binary Search

- Base Case: $n = 0$
  - x is not in the array containing 0 elements
- Inductive Case: $n > 0$
  - If the median element is x, then return the index
  - If the median element is greater than x, recurse on all elements smaller than the median
  - If the median element is smaller than x, recurse on all elements larger than the median

Recursive Binary Search

$\textsc{BinarySearch}(A[\mathit{left} \ldots \mathit{right}], x)$

if $\mathit{left} > \mathit{right}$

  return $\textsc{Not Found}$

$\mathit{mid} = \lfloor (\mathit{left} + \mathit{right})/2 \rfloor$

if $x == A[\mathit{mid}]$

  return mid

else if $x < A[\mathit{mid}]$

  $\textsc{BinarySearch}(\mathit{left}, \mathit{mid} - 1)$

else

  $\textsc{BinarySearch}(\mathit{mid} + 1, \mathit{right})$

Tower of Hanoi

- Given three rods and n disks of various diameters initially stacked on one rod, in order of decreasing size. The objective is to move the entire stack of n disks onto one of the other rods, while obeying the following rules:

1. Only a single disk can be moved at a time
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or an empty rod
3. No disk may be placed on top of a disk that is smaller than it

Tower of Hanoi

- Designing a recursive algorithm:
  - If a given instance of a problem can be solved direcly, solve it
  - Otherwise, reduce the problem into one or more simpler instances of the same problem

- Do not be concerned with solving the smaller instances (i.e., recursive calls)
  - Similar to induction, we assume smaller instances of the same problem can be solved correctly
- Use the solution of the smaller subproblems to solve the problem

Tower of Hanoi

- Inductive Case: $n > 0$

1. Recursively move $(n-1)$ disks onto another rod (leaving the largest diameter disk on the original rod)
2. Move the largest diameter disk onto the empty destination rod
3. Recursively move $(n-1)$ disks on top of the largest diameter disk

Tower of Hanoi

- Inductive Case: $n > 0$

1. Recursively move $(n-1)$ disks onto another rod (leaving the largest diameter disk on the original rod)
2. Move the largest diameter disk onto the empty destination rod
3. Recursively move $(n-1)$ disks on top of the largest diameter disk

- Base Case: $n = 0$

1. The tower of hanoi problem is vacuously solved when there are no disks

Tower of Hanoi

$\textsc{TowerOfHanoi}(n, src, dest, temp)$
  if $n > 0$
    $\textsc{TowerOfHanoi}(n-1, src, temp, dest)$
    Move disk n from src to $dest$
    $\textsc{TowerOfHanoi}(n-1, temp, dest, src)$

Divide-and-Conquer

- Many recursive algorithms follow a divide-and-conquer approach
  - Divide: Break the problem into smaller subproblems
  - Conquer: Recursively solve the subproblems
  - Combine: Use the solutions of the subproblems to solve the original problem

Merge Sort

- Divide: Divide the array of n elements into two subarrays of size $n/2$
- Conquer: Sort each subarray recursively
- Combine: Merge the two sorted subarrays into a single sorted array of n elements

- Requires the use of an auxilliary MERGE procedure

Merge

- Given two sorted sequences L and R
  1. Starting with the first elements in L and R
  2. Choose the smaller of the two elements and place it into the sorted sequence
  3. Repeat until all elements from L and R have been placed into the sorted sequence

Merge

- For simplicity of our pseudocode, we will append ∞ to the end of each sorted sequence
  - $L = A[p \dots q]$
  - $R = A[q + 1 \dots r]$

Merge

```
MERGE(A, p, q, r)
    n1 = q − p + 1
    n2 = r − q
    Let L[1 . . . n1 + 1] and R[1 . . . n2 + 1] be new arrays
    for i = 1 to n1
        L[i] = A[p + i − 1]
    for i = 1 to n2
        R[i] = A[q + j]
    L[n1 + 1] = ∞
    R[n2 + 1] = ∞
    i = 1
    j = 1
    for k = p to r
        if L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[i]
            j = j + 1
```

Merge Sort

MERGESORT(A, p, r)
  if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGESORT(A, p, q)
    MERGESORT($A, q + 1, r$)
    MERGE(A, p, q, r)