4190.101

Discrete Mathematics

Chapter 5 Induction and recursion

Gunhee Kim

[Based on S. Srinivasa Rao's Lecture Notes]

Recursive Algorithms

Section 5.4

Section Summary

- Recursive Algorithms
- Proving Recursive Algorithms Correct
- Recursion and Iteration (*not yet included in overheads*)
- Merge Sort

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.
- For the algorithm to terminate, the instance of the problem must eventually be reduced to some initial case for which the solution is known.

Recursive Factorial Algorithm

- **Example**: Give a recursive algorithm for computing $n!$, where n is a nonnegative integer.
- **Solution**: Use the recursive definition of the factorial function.

procedure *factorial* (n: nonnegative integer)
if $n = 0$ **then return** 1
else return $n \cdot$ factorial ($n - 1$)
{output is $n!$}

Recursive Power Algorithm

- **Example**: Give a recursive algorithm for computing a^n, where a is a nonzero real number and n is a nonnegative integer.
- **Solution**: Use the recursive definition of a^n.

procedure *power* (a: nonzero real number,
    n: nonnegative integer)

if $n = 0$ **then return** 1

else return $a \cdot$ *power* $(a, n - 1)$

{output is a^n}

Recursive GCD Algorithm

- **Example**: Give a recursive algorithm for computing the greatest common divisor of two nonnegative integers a and b with $a < b$.
- **Solution**: Use the reduction

$$\gcd(a,b) = \gcd(b \textbf{ mod } a, a)$$

and the condition $\gcd(0,b) = b$ when $b > 0$.

procedure *gcd* (a,b: nonnegative integers with $a < b$)

if $a = 0$ **then return** b

else return *gcd* (b **mod** a, a)

{output is *gcd*(a, b)}

Recursive Modular Exponentiation Algorithm

- **Example**: Devise a a recursive algorithm for computing b^n **mod** m, where b, n, *and* m are integers with $m \geq 2$, $n \geq 0$, and $1 \leq b \leq m$.
- **Solution**: Use the fact that $b^n = b \cdot b^{n-1}$ and that
  $x \cdot y$ **mod** $m = x \cdot (y$ **mod** $m)$ **mod** m
- Note that this algorithm takes O(n) step

procedure *mpower* (b,m,n: integers with $b \geq 1$, $m \geq 2$, and $n \geq 0$)
if $n = 0$ **then**
        return 1
else
        return ($b \cdot$ *mpower* ($b, m, n-1$)) **mod** m
{output is b^n **mod** m}

Recursive Modular Exponentiation Algorithm

- **Example**: Devise a recursive algorithm for computing b^n **mod** m, where b, n, *and* m are integers with $m \geq 2$, $n \geq 0$, and $1 \leq b \leq m$.
- **Solution**: Use the fact that $b^{2k} = b^{k \cdot 2} = (b^k)^2$.
- What is the complexity? Takes O(log n) step
  - The time complexity of a recursive algorithm depends on the number of recursive calls it makes.

procedure *mpower* (b,m,n: integers with $b > 0$ and $m \geq 2$, $n \geq 0$)
if $n = 0$ **then**
  return 1
else if *n is even* **then**
  return $mpower(b,n/2,m)^2$ **mod** m
else
  return $(mpower(b,\lfloor n/2 \rfloor,m)^2$ **mod** $m \cdot b$ **mod** $m)$ **mod** m
{output is b^n **mod** m}

Recursive Binary Search Algorithm

- **Example**: Construct a recursive version of a binary search algorithm.
- **Solution**: Assume we have $a_1, a_2, \ldots, a_n$, an increasing sequence of integers. Initially i is 1 and j is n. We are searching for x.

procedure *binary search*(i, j, x : integers, $1 \leq i \leq j \leq n$)
$m := \lfloor (i + j)/2 \rfloor$
if $x = a_m$ **then**
    return m
else if ($x < a_m$ and $i < m$) **then**
    return *binary search*($i, m-1, x$)
else if ($x > a_m$ and $j > m$) **then**
    return *binary search*($m+1, j, x$)
else return 0
{output is location of x in $a_1, a_2, \ldots, a_n$ if it appears, otherwise 0}

Proving Recursive Algorithms Correct

- Both mathematical and strong induction are useful techniques to show that recursive algorithms always produce the correct output.
- **Example**: Prove that the algorithm for computing the powers of real numbers is correct.
- **Solution**: Use mathematical induction on the exponent n.
  - BASIS STEP: $a^0 = 1$ for every nonzero real number a, and $power(a,0) = 1$.
  - INDUCTIVE STEP: The inductive hypothesis is that $power(a,k) = a^k$, for all $a \neq 0$. Assuming the inductive hypothesis, the algorithm correctly computes a^{k+1}, since

$$power(a,k + 1) = a \cdot power(a, k) = a \cdot a^k = a^{k+1}.$$

procedure *power*(a: nonzero real number, n: nonnegative integer)
if $n = 0$ **then return** 1
else return $a \cdot power(a, n - 1)$
{output is a^n}

Merge Sort

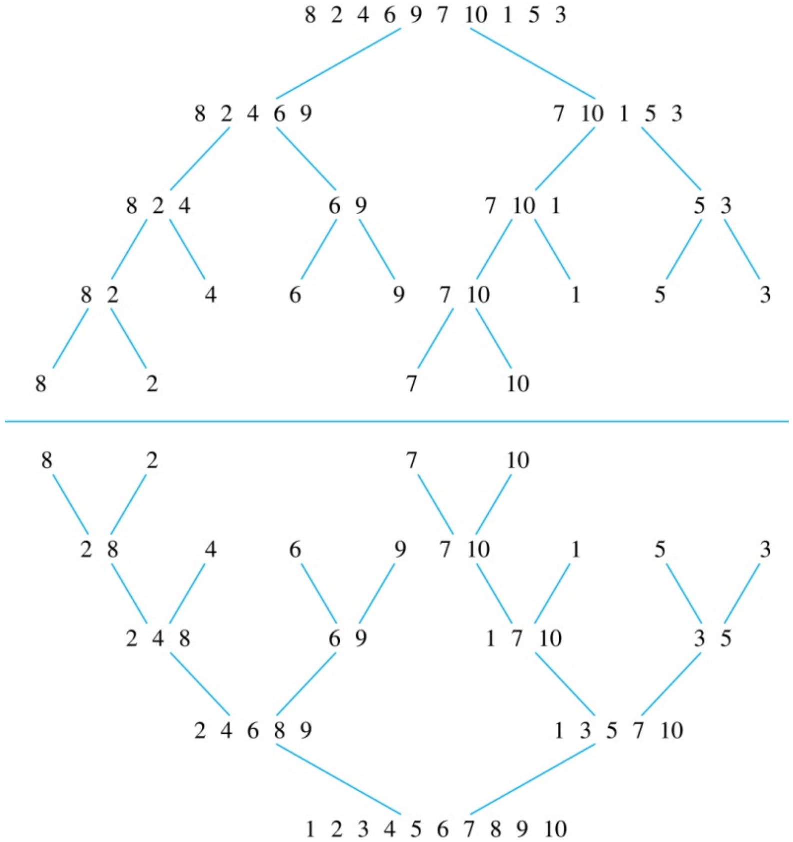
- *Merge Sort* works by iteratively splitting a list (with an even number of elements) into two sublists of equal length until each sublist has one element.
- Each sublist is represented by a balanced binary tree.
- At each step a pair of sublists is successively merged into a list with the elements in increasing order. The process ends when all the sublists have been merged.
- The succession of merged lists is represented by a binary tree.

Merge Sort

- **Example**: Use merge sort to put the list

  8,2,4,6,9,7,10, 1, 5, 3

  into increasing order.
- **Solution**:



Recursive Merge Sort

- **Example**: Construct a recursive merge sort algorithm.
- **Solution**: Begin with the list of n elements L.

procedure *mergesort* ($L = a_1, a_2,\ldots,a_n$)
if $n > 1$ **then**
    $m := \lfloor n/2 \rfloor$
    $L_1 := a_1, a_2,\ldots,a_m$
    $L_2 := a_{m+1}, a_{m+2},\ldots,a_n$
    $L := merge(mergesort(L_1), mergesort(L_2))$
{L is now sorted into elements in increasing order}

continued →

Recursive Merge Sort

- Subroutine *merge*, which merges two sorted lists.

procedure *merge*(L_1, L_2 :sorted lists)
L := empty list
while L_1 and L_2 are both nonempty
    remove smaller of first elements of L_1 and L_2 from its list;
        put at the right end of L
    if this removal makes one list empty
        then remove all elements from the other list and append them to L
return L {L is the merged list with the elements in increasing order}

- **Complexity of Merge**: Two sorted lists with m elements and n elements can be merged into a sorted list using no more than $m + n - 1$ comparisons.

Merging Two Lists

- **Example**: Merge the two lists 2,3,5,6 and 1,4.
- **Solution**:

TABLE 1 Merging the Two Sorted Lists 2, 3, 5, 6 and 1, 4.

First List	*Second List*	*Merged List*	*Comparison*
2 3 5 6	1 4		$1 < 2$
2 3 5 6	4	1	$2 < 4$
3 5 6	4	1 2	$3 < 4$
5 6	4	1 2 3	$4 < 5$
5 6		1 2 3 4	
		1 2 3 4 5 6	

Complexity of Merge Sort

- **Complexity of Merge Sort**: The number of comparisons needed to merge a list with n elements is $O(n \log n)$.
- For simplicity, assume that n is a power of 2, say 2^m.
- At the end of the splitting process, we have a binary tree with m levels, and 2^m lists with one element at level m.
- The merging process begins at level m with the pairs of 2^m lists with one element combined into 2^{m-1} lists of two elements. Each merger takes one comparison.
- The procedure continues, at each level ($k = m, m-1, \ldots,3,2,1$) 2^k lists with 2^{m-k} elements are merged into 2^{k-1} lists, with 2^{m-k+1} elements at level $k-1$.
  - We know (by the complexity of the merge subroutine) that each merger takes at most $2^{m-k} + 2^{m-k} - 1 = 2^{m-k+1} - 1$ comparisons.

continued →

Complexity of Merge Sort

- Summing over the number of comparisons at each level, shows that

$$\sum_{k=1}^{m} 2^{k-1}(2^{m-k+1}-1) = \sum_{k=1}^{m} 2^m - \sum_{k=1}^{m} 2^{k-1} = m2^m - (2^m - 1) = n \log n - n + 1,$$

because $m = \log n$ and $n = 2^m$.

- (The expression $\sum_{k=1}^{m} 2^{k-1}$ in the formula above is evaluated as $2^m - 1$ using the formula for the sum of the terms of a geometric progression, from Section 2.4.)
- In Chapter 11, we'll see that the fastest comparison-based sorting algorithms have $O(n \log n)$ time complexity. So, merge sort achieves the best possible big-O estimate of time complexity.