Analyzing Complexity of Lists

Operation	Sorted Array	Sorted Linked List	Unsorted Array	Unsorted Linked List
Search(L, x)	O(logn)	O(n)	O(n)	O(n)
Insert(L, x)	O(logn) + O(n)	O(n) + O(1)	O(1)	O(1)
Delete(L, x)	O(logn) + O(n)	O(n) + O(1)	O(n) + O(n)	O(n) + O(1)

CISC 235 Topic 2

Design and Complexity Analysis
of Recursive Algorithms

Outline

- Design of Recursive Algorithms
  - Recursive algorithms for Lists
- Analysis of Recursive Algorithms
  - Modeling with recurrence relations
  - Solving recurrence relations

Thinking Recursively

1. What is the measure of the size of input?
2. What is the base case?
3. What is the recursive case?
4. In what ways could the input be reduced in size (and how easy is it to do so)?
5. If we assume that we have the solution to the same problem for a smaller size input, how can we solve the whole problem?

Example: Find Largest in List

1. Measure of input size: length of list
2. Base Case: list of length 1
3. Recursive Case: length > 1
4. Ways to reduce in size
   a. All except first item in list
   b. All except last item in list
   c. Divide list into two halves
5. Assume we have solution to smaller size list(s).
   a. Take max of first item and max of rest of list
   b. Take max of last item and max of rest of list
   c. Take max of the max of the two halves

Example: Find Largest in Array

```
// Assumes list is not empty
static int largest ( int[ ] A, int first, int last )
{     int n = last - first + 1;
      if ( n == 1)
            return ( A[first] );
      else
            return ( Math.max( A[first],
                          largest( A, first + 1, last ) ) );
}
```

Version of largest method that divides list in two halves

```
static int largest ( int[ ] A, int first, int last )
{
    int n = last - first + 1;
    if ( n == 1)
        return ( A[first] );
    else
    {   int mid = ( first + last ) / 2;
        return ( Math.max(largest( A, first, mid ), largest( A, mid  + 1, last ) ) );
    }
}
```

Is there any advantage to dividing the list this way?`

Design Rules for Recursive Functions

1. Make sure there's a base case
2. Make progress towards the base case

   Reduce size of input on each recursive call

3. Assume the recursive calls are correct

   Write the method so that it's correct if the recursive calls are correct

4. Compound Interest Rule

   Don't duplicate work by solving the same problem instance in different calls

Incorrect Recursive Functions

Which design rules do these violate?

```
static int factorial ( int n )
{     if ( n == 0 )
            return (1);
      else
            return ( factorial( n ) * n-1 );
}

static int factorial ( int n )
{
            return ( n * factorial( n-1 ) );
}
```

Inefficient Recursive Functions

Which design rule does this violate?

```
static int fibonacci ( int n )
{     if ( n <= 1)
            return (n);
      else
            return ( fibonacci ( n – 1 )
                       + fibonacci ( n – 2 ) );
}
```

Divide and Conquer Algorithms

- Divide the problem into a number of subproblems of size ~n/2 or n/3 or …
- Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- Combine the solutions to the subproblems into the solution for the original problem.

Example: Merge Sort

- Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
- Conquer: Sort the two subsequences recursively using merge sort.
- Combine: Merge the two sorted subsequences to produce the sorted answer.

Merge Sort Algorithm

// if $p \geq r$, subarray A[p..r] has at most one element,

// so is already in sorted order

```
MERGE-SORT ( A, p, r )
   if p < r
        then q ← ⌊ (p+r) / 2 ⌋
               MERGE-SORT( A, p, q )
               MERGE-SORT( A, q+1, r )
               MERGE( A, p, q, r )
```

Merge Algorithm

// preconditions: $p \le q < r$

// A[p..q] and A[q+1..r] are in sorted order

MERGE(A, p, q, r)

  $n_1 \leftarrow q - p + 1$

  $n_2 \leftarrow r - q$

  create arrays $L[1..\ n_1 + 1]$ and $R[1..\ n_2 + 1]$

  for $i \leftarrow 1$ to n_1

    do $L[\ i\ ] \leftarrow A[\ p + i - 1\ ]$

  for $j \leftarrow 1$ to n_2

    do $R[\ j\ ] \leftarrow A[\ q + j\ ]$

  $L[n_1 + 1\ ] \leftarrow \infty$

  $R[n_2 + 1\ ] \leftarrow \infty$

Merge Algorithm, con.

```
// Merge arrays L and R and place back in array A
i ← 1
j ← 1
for k ← p to r
      do if L[ i ] ≤ R[ j ]
            then A[ k ] ← L[ i ]
                  i ← i + 1
            else A[ k ] = R[ j ]
                  j ← j + 1
```

Recursive Algorithms for Linked Lists

1. Measure of input size: length of list
2. Base Case: list of length 0 or 1
3. Recursive Case: length > 1
4. Ways to reduce in size on each recursive call?

Start of a Linked List Class

```
public class List
    private class Node {
            private Node next;
            private int data;
            Node( int data ) {
            this.data = data;
            this.next = null; }   }  // end Node class
    private Node head;
    public List() {
        head = null; }    … } // end List class
```

Add Methods to Class

```
public void append( int x )
```

```
public void insert( int x )
```

Functions: Complexity Analysis

```
static int bar ( int x, int n )
{    for ( int i=1; i<=n; i++ )
          x += i;
     return x;
}  // end bar
```

```
static int foo (int x, int n )
{    for ( int i=1; i<=n; i++ )
          x = x + bar( i, n );
     return x;
}  // end foo
```

```
static int m( int y )
{
   int a = 0;
   System.out.print(foo( a, y ));
   System.out.print(bar( a, y ));
}  // end m
```

What is the measure of the size of input of these methods?

```
// Calculates x^i
static double pow ( double x, int i )

// Counts the number of occurrences of each
// different character in a file (256 possible different chars)
static int countChars ( String inFile )

// Determines whether vertex v is adjacent to
// vertex w in graph g
static boolean isAdjacent( Graph g, Vertex v, Vertex w )
```

Analysis: Recursive Methods

```
static int factorial ( int n )
{     if ( n <= 1 )
            return (1);
      else
            return ( n * factorial( n – 1 ) );  }
```

Recurrence Relation:

$$T(n) = O(1), \quad \text{if } n = 0 \text{ or } 1$$
$$T(n) = T(n-1) + O(1), \quad \text{if } n > 1$$

Or:

$$T(n) = c, \quad \text{if } n = 0 \text{ or } 1$$
$$T(n) = T(n-1) + c, \quad \text{if } n > 1$$

Recurrence Relations

What if there was an O(n) loop in the base case of the factorial function? What would its recurrence relation be?

What if there was an O(n) loop in the recursive case of the factorial function? What would its recurrence relation be?

Recurrence Relations

What is the recurrence relation for the first version of the `largest` method?

What is the recurrence relation for the version of `largest` that divides the list into two halves?

What is the recurrence relation for the `fibonacci` method?

Binary Search Function

```
// Search for x in A[low] through A[high] inclusive
// Return index of x if found; return -1 if not found
int binarySearch( int[ ] A, int x, int low, int high )
{       if( low > high )
                return -1;
        int mid = ( low + high ) / 2;
        if( A[mid] < x )
                return binarySearch( A, x, mid+1, high);
        else if ( x < A[mid] )
                return binarySearch( A, x, low, mid-1 );
        else
                return mid;
}
```

Analysis: Binary Search

Measure of Size of Input:

Recurrence Relation:

Analysis: Merge Sort

Measure of Size of Input:

Recurrence Relation:

Solving Recurrences

Substitution Method:

1. Guess Solution
2. Prove it's correct with proof by induction

How to guess solution? Several ways:

- Calculate first few values of recurrence
- Substitute recurrence into itself
- Construct a Recursion Tree for the recurrence

Calculate First Few Values

$T(0) = c$

$T(1) = c$

$T(2) = T(1) + c = 2c$

$T(3) = T(2) + c = 3c$

$T(4) = T(3) + c = 4c$

. . .

Guess solution:

$T(n) = nc$, for all $n \geq 1$

Substitute recurrence into itself

$$T(n) = T(n-1) + c$$

$$T(n) = (T(n-2) + c) + c \quad = T(n-2) + 2c$$

$$T(n) = (T(n-3) + c) + 2c = T(n-3) + 3c$$

$$T(n) = (T(n-4) + c) + 3c = T(n-4) + 4c$$

. . .

Guess Solution:

$$\begin{aligned} T(n) &= T(n-(n-1)) + (n-1)c \\ &= T(1) + (n-1)c \\ &= c + (n-1)c \\ &= nc \end{aligned}$$

Prove Solution is Correct: $T(n) = nc$, for all $n \geq 1$

Base Case: $n = 1$, formula gives $T(1) = c$?

$T(1) = 1c = c$

Inductive Assumption: $T(k) = kc$

Show Theorem is true for $T(k+1)$,

i.e., $T(k+1) = (k+1)c$:

By the recurrence relation, we have:

$$\begin{aligned} T(k+1) &= T(k) + c \\ &= kc + c \quad \text{by inductive assump.} \\ &= (k+1)c \end{aligned}$$