Recursive Procedures

- You were supposed to be introduced to recursive procedures in CSCE 156.
- What recursive procedures have you seen?
- Many loops can be replaced with recursive procedures.
  - In some algorithms, it is easier to use recursion than loops!
- Divide and Conquer algorithms frequently use recursive procedures to divide the data set into one or more parts and then recursively apply the algorithm to the smaller parts.

Example: Binary Search

```c
int binarySearch(int[] entry, int first, int last, int key)
1.  if (last < first)
2.       index = -1
3.  else
4.      int middle = (first + last)/2
5.      if (key == entry[middle])
6.         index = middle
7.      else if (key < entry[middle])
8.         index = binarySearch(entry, first, middle -1, key)
9.      else
10.       index = binarySearch(entry, middle +1, last, key)
11. return index
```

Designing Recursive Procedures

- Think Inductively
- Converge to a base case (stopping the recursion)
  - identify some unit of measure (running variable)
  - identify the easy cases, called base cases
- Assume algorithm p must solve the problem with input sizes ranging from 0 through 100
  - assume \( p_{99} \) solved a subproblem for all sizes 0 through 99
  - if \( p \) detects a case that is not the base case, it calls \( p_{99} \) with a proper subset of the input data
- \( p_{99} \) satisfies:
  1. The subproblem size is less than \( p \)'s problem size
  2. The subproblem size is not below the minimum
  3. The subproblem satisfies all other preconditions of \( p_{99} \) (which are the same as the preconditions of \( p \))

Recursive Procedure Design Example

- Problem:
  - Write a delete(L, x) procedure for a list L, which is supposed to delete the first occurrence of x
  - However, it is possible x does not occur in L
- Strategy:
  - Use a recursive Procedure
  - The size of the problem is the number of elements in list L
  - Use IntList ADT
  - Base cases: ??
  - Running variable (converging number): ??
ADT for Lists

IntList nil //constant denoting the empty list.
IntList constructList(int newElement, IntList oldList)
Precondition: None.
Postconditions: If newList = constructList(newElement, oldList) then
1. newList refers to a newly created list object;
2. newList ≠ nil;
3. first(newList) = newElement;
4. rest(newList) = oldList

Algorithm for Recursive delete(L, x) from list

IntList delete(IntList initialList, int anElement)
1. IntList resultList, subproblemList;
2. if (initialList == nil)
3. resultList = initialList;
4. else if (anElement == first(initialList))
5. resultList = rest(initialList);
6. else
7. subproblemList = delete(rest(initialList), anElement);
8. resultList = constructList(first(initialList), subproblemList);
9. return resultList;

Now remove "99" from the called subroutine. That is, change delete99() to delete().

Algorithm for non-recursive delete(L, x)

IntList delete(IntList L, int x)
1. IntList newL, tempL;
2. tempL = L; newL = nil;
3. // search for x, copying elements to newL until x is found or tempL is empty
4. while (tempL != nil && x != first(tempL))
5. newL = constructList(first(tempL), newL); //copy element
6. tempL = rest(tempL); // skip copied element
7. if (tempL != nil) // ⇒ x == first(tempL)
8. tempL = rest(tempL); // remove x
9. while (tempL != nil) // copy remaining elements
10. newL = cons(first(tempL), newL);
11. tempL = rest(tempL);
12. return newL; // x is not in newL.

Convert a non-recursive procedure to a recursive procedure

◆ Change the procedure with a loop to call a recursive procedure without a loop
◆ Recursive Procedure begins by acting like a WHILE loop
  » While(Not Base Case)
  » Set Up Sub-problem
  » Recursive call to continue
◆ The recursive function may need an additional parameter
  » E.g., to replace an index in a FOR loop of the non-recursive procedure.

Removing While Loop Example

Transforming loop into a recursive procedure

◆ Local variables within the loop body
  » Give the variable only one value in any one pass
  » For variables that must be updated, do all the updates at the end of the loop body
◆ Re-expressing a while loop with recursion
  » Additional parameters
    » Variables updated in the loop become procedure input parameters. Their initial values at loop entry correspond to the actual parameters in the top-level call of the recursive procedure.
    » Variables referenced in the loop but not updated may also become parameters
  » The recursive procedure begins by mimicking the while condition and returns if the while condition is false
    » A break also corresponds to a procedure return
  » Continue by updating variables and make the recursive call

1. int factLoop(int n)
2. int k=1; int f = 1
3. while (k ≤ n)
   4. int fnew = f*k;
   5. int knew = k+1;
   6. k = knew; f = fnew;
7. return f;
1. int factLoop(int n)
2. return factRec(n, 1, 1);
3. int factRec(int n, int k, int f)
4. if (k ≤ n)
   5. int fnew = f*k;
   6. int knew = k+1
7. return f;
8. return f;
Removing For Loop Example

Convert the following sequentialSearch() procedure to a recursive procedure without a loop

```c
int sequentialSearch(int[] entry, int nEntries, int key)
1. int answer, index;
2. answer = -1; // Assume failure.
3. for (index = 0; index < nEntries; index++)
4.     if (key == entry[index])
5.         answer = index; // Success!
6.         break; // Done!
7. return answer;
```

Recursive Procedure without loops e.g.

```c
Call with: sequentialSearchRecursive(entry, 0, nEntries, key)

seqSearchRecursive(int[] entry, int index, int nEntries, int key)
0. int answer;
1. if (index ≥ nEntries)
2.   answer = -1;
3. else if (entry[index] == key)  // index < nEntries
4.   answer = index;
5. else
6.   answer = sequentialSearchRecursive(entry, index+1, nEntries, key);
7. return answer;
```

Analyzing Recursive Procedure using Recurrence Equations

- Let n be the size of the problem
- Worst-Case Analysis (for procedure with no loops)
  » T(n) = the individual cost for a sequence of blocks
  » the maximum cost for an alternation of blocks
  » the cost of subroutine call, T( f(n) )
  » the cost of recursive procedure call, T( g(n) )
- e.g. sequentialSearchRecursive()
  » Basic operation is comparison of array element, cost 1
  » f(n) = max(2., (3. + max(4., (5. + 6.))) + 7.)
  » g(n) = 0
  » T(n) = (n-1) + 1; T(0) = 0
  » T(n) = n; T(n) ∈ θ(n)

Consider binarySearch()

```c
int binarySearch(int[] entry, int first, int last, int key)
1. if (last < first)
2.   index = -1
3. else
4.   int middle = (first + last)/2
5.   if (key == entry[middle])
6.     index = middle
7.   else if (key < entry[middle])
8.     index = binarySearch(entry, first, middle -1, key)
9.   else
10.    index = binarySearch(entry, middle +1, last, key)
11. return index
```

Evaluate recursive equation using Recursion Tree

- Evaluate: T(n) = T(n/2) + T(n/2) + n
  » Working copy: T(k) = T(k/2) + T(k/2) + k
  » For k=n/2, T(n/2) = T(n/4) + T(n/4) + (n/2)
- [size|cost]

Recursion Tree e.g.

- To evaluate the total cost of the recursion tree
  » sum all the non-recursive costs of all nodes
  » = Sum (rowSum(cost of all nodes at the same depth))
- Determine the maximum depth of the recursion tree:
  » For our example, at tree depth d, the size parameter is n/2^d
  » the size parameter converges to the base case, i.e. case 1 where n/2^d = 1 ⇒ d = lg(n)
- The rowSum for each row is n
- Therefore, the total cost, T(n) = n lg(n)
Proving Correctness of Procedures: Proof

- What is a Proof?
  - A Proof is a sequence of statements that form a logical argument.
  - Each statement is a complete sentence in the normal grammatical sense.
- Each statement should draw a new conclusion from:
  - axioms: well known facts
  - assumptions: premises of the theorem you are proving or inductive hypothesis
  - intermediate conclusions: statements established earlier
- To arrive at the last statement of a proof that must be the conclusion of the proposition being proven

Format of Theorem, Proof Format

- A proposition (theorem, lemma, and corollary) is represented as:
  \[ \forall x \in W ( A(x) \implies C(x) ) \]
  for all \( x \) in \( W \), if \( A(x) \) then \( C(x) \)
  - the set \( W \) is called world,
  - \( A(x) \) represents the assumptions
  - \( C(x) \) represents the conclusion, the goal statement
  - \( \implies \) is read as “implies”
- Proof sketches provide an outline of a proof
  - the strategy, the road map, or the plan.
- Two-Column Proof Format
  - Statement : Justification (supporting facts)

Induction Proofs

- Induction proofs are a mechanism, often the only mechanism, for proving a statement about an infinite set of objects.
- Inferring a property of a set based on the property of its objects
- Induction is often done over the set of natural numbers \( \{0,1,2,\ldots\} \)
  - starting from 0, then 1, then 2, and so on
- Induction is valid over a set, provided that:
  - The set is partially ordered;
  - i.e. an order relationship is defined between some pairs of elements, but perhaps not between all pairs.
  - There is no infinite chain of decreasing elements in the set. (e.g. cannot be set of all integers)

Induction Proof Schema

Prove: \[ \forall x \in W ( A(x) \implies C(x) ) \]
- Proof:
  1. The Proof is by induction on \( x \), \langle description of \( x \rangle
  2. The base case is, cases are, \langle base-case\rangle
  3. \langle Proof of goal statement with base-case substituted into it, that is, \( C(\text{base-case}) \rangle
  4. For \( x \) greater than \langle base-case\rangle, assume that \( A(y) \implies C(y) \)
      holds for all \( y \) \( \in W \) such that \( y < x \).
  5. \langle Proof of the goal statement, \( C(x) \), exactly as it appears in the proposition\rangle.

Induction Proof Example

Prove:

\[ \sum_{i=1}^{n} i(i+1)/2 = n(n+1)(n+2)/6 \]
- Proof:
  - Left as an exercise for the student ☺

Proving Correctness of Procedures

- Things should be made as simple as possible – but not simpler
  - Albert Einstein
- Proving Correctness of procedures is a difficult task in general; the trick is to make it as simple as possible.
  - No loops are allowed in the procedure!
  - Variable is assigned a value only once!
- Loops are converted into Recursive procedures.
- Additional variables are used to make single-assignment (write-once read many) possible.
  - \( x = y+1 \) does imply the equation \( x = y+1 \) for entire time
General Correctness Lemma

- If all preconditions hold when the block is entered,
- then all postconditions hold when the block exits
- And, the procedure will terminate!

Chains of Inference: Sequence

Proving Correctness of binarySearch()

```
int binarySearch(int[] entry, int first, int last, int key)
1. if (last < first)
2.     index = -1
3. else
4.     int middle = (first + last)/2
5.     if (key == entry[middle])
6.         index = middle
7.     else if (key < entry[middle])
8.         index = binarySearch(entry, first, middle -1, key)
9.     else
10.    index = binarySearch(entry, middle +1, last, key)
11. return index
```

Proving Correctness of Binary Search

- Lemma (preconditions => postconditions)
  - if binarySearch(entry, first, last, key) is called, and
    the problem size is \( n = (\text{last} - \text{first} + 1) \), for all \( n \geq 0 \), and
    entry[first], ..., entry[last] are in nondecreasing order,
  - then it returns –1 if key does not occur in entry within the range
    first, ..., last, and
  - it returns index such that key = entry[index] otherwise

- Proof
  - The proof is by induction on \( n \), the problem size.
  - The base case in \( n = 0 \).
  - In this case, line 1 is true, line 2 is reached, and –1 is returned. (the postcondition is true)

Inductive Proof, continue

- For \( n > 0 \), assume that binarySearch(entry, f, l, key) satisfies the lemma on problems of size \( k \), such that
  \( 0 \leq k < n \), and \( f \) and \( l \) are any indices such that \( k = \text{last} - \text{first} + 1 \).
  - For \( n > 0 \), line 1 is false, \( f \) and \( l \) are within the search range (first \( \leq \) middle \( \leq \) last).
  - If line 5 is true, the procedure terminates with index = middle. (the postcondition is true)
  - If line 5 is false, from (first \( \leq \) middle \( \leq \) last) and def. of \( n \),
    \( \text{middle} - 1 \leq \text{first} \leq \text{last} \) \( \Rightarrow \) \( \text{middle} - 1 \leq (n - 1) \leq \text{last} \).
  - So the inductive hypothesis applies for both recursive calls.
    - If line 7 is true, ... the preconditions of binarySearch are satisfied, we can assume that the call accomplishes the objective.
      - If line 8 returns a positive index, done.
      - If line 8 returns –1, this implies that key is not in entry in the first ... middle-1, also since line 7 is true, key is not in entry in range min ... last, so returning –1 is correct (done).
    - If line 7 is false, ... similarly the postconditions are true. (done!)