Recursion and Induction

- For advanced algorithm development, recursion is an essential design technique
- Recursive Procedures
- What is a Proof?
- Induction Proofs
- Proving Correctness of Procedures
- Recurrence Equations
- Recursion Trees

TECH
Computer Science

Recurrence Equations vs. Recursive Procedures

- Recurrence Equations:
  - **defines a function over the natural numbers, say T(n), in terms of its own value at one or more integers smaller than n.**
  - **T(n) is defined inductively.**
  - **There are base cases to be defined separately.**
  - **Recurrence equation applies for n larger than the base cases**
- Recursive Procedures:
  - **a procedure calls a *unique* copy of itself**
  - **converging to a base case (stopping the recursion)**

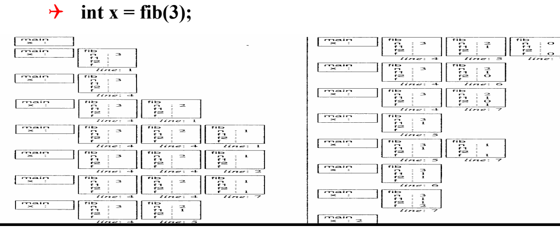
e.g. Fibonacci Function

- Recurrence Equation: e.g. Fibonacci Function
  - **fib(0) = 0; fib(1) = 1; // base cases**
  - **fib(n) = fib(n-1) + fib(n-2) // all n>2**
- Recursive Procedure:
- int fib(int n)

```
  int f, f1, f2;
1. if (n < 2)
2.    f = n;  // base cases
3. else
4.    f1 = fib( n-1 );
5.    f2 = fib( n-2 );
6.    f = f1 + f2;
7. return f;
```

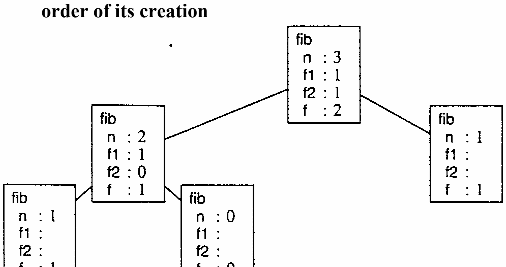
The working of recursive procedure

- a unique copy for each call to itself
  - **individual procedure invocation at run time**
  - **i.e. activation frame**
- e.g. The working of fib(n)
  - **main ()**
  - **int x = fib(3);**



Activation Tree

- **Each node corresponds to a different procedure invocation, just at the point when it is about to return.**
- **A preorder traversal visits each activation frame in order of its creation**



Analysis for algorithm without loops

- In a computation without loops, but possible with recursive procedure calls:
  - **The time that any particular activation frame is on the top of the frame stack is O(L),**
  - **where L is the number of lines in the procedure that contain either a simple statement or a procedure call.**
  - **The total computation time is θ(C),**
  - **where C is the total number of procedure calls that occur during the computation.**

Designing Recursive Procedures

- // Think Inductively
- converging to a base case (stopping the recursion)
  - **identify some unit of measure (running variable)**
  - **identify base cases**
- assume p solves all sizes 0 through 100
  - **assume p99 solve sub-problem all sizes 0 through 99**
  - **if p detect a case that is not base case it calls p99**
- p99 satisfies:
  - **1. The sub-problem size is less than p's problem size**
  - **2. The sub-problem size is not below the base case**
  - **3. The sub-problem satisfies all other preconditions of p99 (which are the same as the preconditions of p)**

Recursive Procedure design e.g.

- Problem:
  - **write a delete(L, x) procedure for a list L**
  - **which is supposed to delete the first occurrence of x.**
  - **Possibly x does not occur in L.**
- Strategy:
  - **Use 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 IntList

- IntList cons(int newElement, IntList oldList)
  - **Precondition: None.**
  - **Postconditions: If x = cons(newElement, oldList) then**
    1. x refers to a newly created object;
    2. x != nil;
    3. first(x) = newElement;
    4. rest(x) = oldList
- int first(IntList aList) // access function
  - **Precondition: aList != nil**
- IntList rest(IntList aList) // access function
  - **Precondition: aList != nil**
- IntList nil //constant denoting the empty list.

Recurrence Equation for delete(L, x) from list L

- **Think Inductively**

- delete(nil, x) = nil
- delete(L, x) = rest(L) /; x == first(L)
- delete(L, x) = cons(first(L), delete(rest(L), x))

Algorithm for Recursive delete(L, x) from list //

```
IntList delete(IntList L, int x)
 IntList newL, fixedL;
 if (L == nil)
  newL = L;
 else if (x == first(L))
  newL = rest(L);
 else
  fixedL = delete99(rest(L), x);
  newL = cons(first(L), fixedL);
return newL;
```

Algorithm for non-recursive delete(L, x)

```
IntList delete(IntList L, int x)
 IntList newL, tempL;
 tempL = L; newL = nil;
 while (tempL != nil && x != first(tempL)) //copy elements
     newL = cons(first(tempL), newL);
     tempL = rest(tempL)
 If (tempL != nil) // x == first(tempL)
   tempL = rest(tempL) // remove x
 while (tempL != nil)  // copy the rest elements
     newL = cons(first(tempL), newL);
     tempL = rest(tempL)
 return newL;
```

Convert a non-recursive procedure to a recursive procedure

- Convert procedure with loop
  - **to recursive procedure without loop**
- Recursive Procedure acting like WHILE loop
  - **While(Not Base Case)**
  - **Setting up Sub-problem**
  - **Recursive call to continue**
- The recursive function may need an additional parameter
  - **which replaces an *index* in a FOR loop of the non-recursive procedure.**

Transforming loop into a recursive procedure

- Local variable with the loop body
  - **give the variable only one value in any one pass**
  - **for variable 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 while condition is false**
    - a break also corresponds to a procedure return
  - **Continue by updating variable and make recursive call**

Removing While loop, e.g.

```
int factLoop(int n)
  int k=1; int f = 1

  while (k <= n)
    int fnew = f*k;
    int knew = k+1
    k = knew; f = fnew;
return f;
```

```
int factLoop(int n)
  return factRec(n, 1, 1);
int factRec(int n, int k, int f)
  if (k <= n)
    int fnew = f*k;
    int knew = k+1
    return factRec(n, knew, fnew)
return f;
```

Removing For loop, e.g.

- Convert the following seqSearch
  - **to recursive procedure without loop**

```
int seqSearch(int[] E, int num, int K)
1. int ans, index;
2. ans = -1; // Assume failure.
3. for (index = 0; index < num; index++)
4.     if (K == E[index])
5.         ans = index; // Success!
6.         break; // Done!
7. return ans;
```

Recursive Procedure without loops e.g.

- Call with: seqSearchRec(E, 0, num, K)
- seqSearchRec(E, index, num, K)

```
1: if (index >= num)
2:   ans = -1;
3: else if (E[index] == K)  // index < num
4:   ans = index;
5: else
6:   ans = seqSearchRec(E, index+1, num, K);
7: return ans;
```

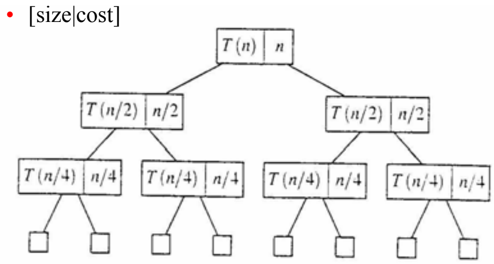
- Compare to: for (index = 0; index < num; index++)

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**
  - **add the maximum cost for an alternation of blocks**
  - **add the cost of subroutine call, S(f(n))**
  - **add the cost of recursive procedure call, T(g(n))**
- e.g. seqSearchRec,
  - **Basic operation is comparison of array element, cost 1**
  - **statement: 1: + max(2:, (3: + max(4:, (5: + 6:))) + (7:)**
  - **Cost: 0 + max(0, (1 + max(0, (0+T(n-1))) + 0**
- $T(n) = T(n-1) + 1;\ T(0) = 0$
- $\Rightarrow T(n) = n;\quad T(n) \in \theta(n)$

Evaluate recursive equation using Recursion Tree

- Evaluate: $T(n) = T(n/2) + T(n/2) + n$
  - **Work 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 converging to base case, i.e. case 1**
  - **such that, $n/(2^d) = 1$,**
  - **$d = \lg(n)$**
  - **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*:
  - ***axiom*: 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:
- $\forall x \in W \, (A(x) \Rightarrow C(x))$
- 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**
  - **=> is read as "implies"**
- Proof sketches provides 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,…}
  - **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

- 0: Prove: $\forall x \in W \, (A(x) \Rightarrow C(x))$
- Proof:
  - **1: The Proof is by induction on x, <description of x>**
  - **2: The base case is, cases are, <base-case>**
  - **3: <Proof of goal statement with base-case substituted into it, that is, C(base-case)>**
  - **4: For <x> greater than <base-case>, assume that A(y) => C(y) holds for all $y \in W$ such that $y < x$.**
  - **5: <Proof of the goal statement, C(x), exactly as it appears in the proposition>.**

Induction Proof e.g.

- Prove:

  For all n >= 0,

  $\sum_{i=0}^{n} i(i+1)/2 = n(n+1)(n+2)/6$
- Proof: …

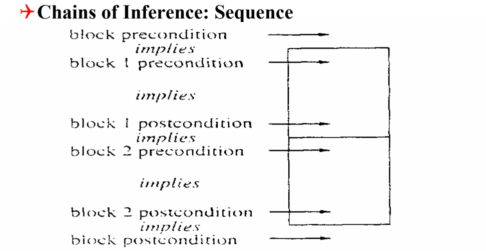
Proving Correctness of Procedures

- *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 is 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 Binary Search, e.g.

- int binarySearch(int[] E, int first, int last, int K)
- 1. if (last < first)
- 2. index = -1;
- 3. else
- 4. int mid = (first + last)/2
- 5. if (K == E[mid])
- 6. index = mid;
- 7. else if (K < E[mid])
- 8. index = binarySearch(E, first, mid-1, K)
- 9. else
- 10. index = binarySearch(E, mid+1, last, K);
- 11. return index

Proving Correctness of Binary Search

- Lemma (*preconditions => postconditions)*
  - ***if* binarySearch(E, first, last, K) is called, and the problem size is n = (last – first + 1), for all n >= 0, and E[first], … E[last] are in nondecreasing order,**
  - ***then* it returns –1 if K does not occur in E within the range first, …, last, and it returns index such that K=E[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(E, first, last, K) satisfies the lemma on problems of size k, such that 0 <= k < n, and first and last are any indexes such that k = last – first + 1
  - **For n > 0, line 1 is false, … mid is within the search range (first <= mid <= last). If line 5 is true, the procedure *terminates* with index = mid. *(the postcondition is true)***
  - **If line 5 is false, from (first <= mid <= last) and def. of n, (mid – 1) – first + 1 <= (n – 1) last – (mid + 1) + 1 <= (n – 1)**
  - **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 return positive index, done.**
  - **If line 8 returns –1, this implies that K is not in E in the first … mid-1, also since line 7 is true, K is not in E in range min… last, so returning – 1 is correct (done).**
  - **If line 7 is false, … similar the postconditions are true. (done!)**