CSE 12
Recursion

• What is recursion?
• Recursive definitions
• Recursion as a problem solving strategy
• Recursion and the runtime stack
• Relationship between recursion and iteration
• Time and space costs of recursion
What is recursion?

• A recursive function is a function that calls itself
• A recursive definition is a definition that defines a concept in terms of itself

• Writing recursive functions is a very useful way to program… sometimes
• Using recursive definitions of concepts is useful too… sometimes

• However, it can be tricky to do it right
Recursive definitions in mathematics

• Recursive definitions are common in mathematics. For example:

The Nth Fibonacci number is
  a) 1, if N is equal to 1 or 2; or,
  b) the sum of the N-1 and N-2 Fibonacci numbers, if N > 2

• a) is called the “base part” of the definition: it keeps the definition from being completely circular, because it makes no mention of the concept that is being defined

• b) is called the “recursive part” of the definition: it mentions the concept that is being defined (“Fibonacci number”)
“base part” and “recursive part”

• The base part of a recursive definition is important: it keeps the definition from being completely circular.

• But it is also important that the recursive part of the definition brings the definition closer to the base part, so that the base part will eventually apply.

• That is: the base part involves the smallest possible version of the problem, and the recursive part involves a smaller version of the problem than the original.

• In the definition of the $N^{th}$ Fibonacci number, the recursive part involves definitions of the $N-1^{st}$ and $N-2^{st}$ Fibonacci numbers, which are “smaller” versions of the same concept.
A recursive definition

• In Java or C or C++, you can define “integer literal constant” this way:

An integer literal constant is
  a) a digit, or
  b) a digit followed by an integer literal constant

A digit is
  one of 0,1,2,3,4,5,6,7,8, or 9

• Does 8 fit the definition? does 82? does 82X?
Levels in Programming Languages

• In defining a programming language, usually there are two levels of rules:

  – **lexical** rules: these define tokens, the most basic building blocks of the language (such as constants, identifiers, reserved words, punctuation)

  – **grammar** rules: these define the legal arrangements of tokens to form expressions, statements, etc.
Backus-Naur Form (BNF)

- BNF is a notation for specifying the lexical or grammar rules of formal languages (for example, computer programming languages)
- Invented by John Backus and Peter Naur in late 1950’s. Variants of it are now the standard way to define programming languages
- Modern compiler-writing tools are programs that take a BNF definition of a language as input, and automatically generate a syntax checker for the language
- BNF uses recursive definitions!
Backus-Naur Form notation

- BNF notation:
  - `:=` means “is defined as”
  - `|` means “or”

- *nonterminal* symbols, which are the names of concepts that are being defined, appear in brackets `<>`

- anything else is a *terminal* symbol, and is taken literally
“integer literal constant” defined in BNF

• Translating the recursive definition of integer literal constant into BNF:

\[
\text{<IntConst> := <digit> | <digit><IntConst>}
\]
\[
\text{<digit> := 0|1|2|3|4|5|6|7|8|9}
\]

• Try applying the definition of \text{<IntConst>} to these strings:

  8
  T
  82
  82R9
Defining “identifier” in BNF

- In Java and C, an identifier is: a sequence of letters, digits, or underscores that starts with a letter or underscore

- How to define `<identifier>` in BNF?

- First, write down in English a recursive definition of the concept.
  - Hint: you can introduce additional concepts in the definition to make it easier (though of course you also need to define them)

- Then translate that recursive definition into BNF
  - The concepts in the definition correspond to nonterminals in the BNF
A BNF definition of “identifier”

1. `<underscore> := _`
2. `<letter> := a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z`
3. `<digit> := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`
4. `<identifier> := <letter> | <underscore> | `<identifier><letter> | `<identifier><digit> | `<identifier><underscore>`

The definition of `<identifier>` is self-referential; the concept being defined appears in its own definition. Recursion!
Derivation of identifier \texttt{ir2\_D2}

- BNF definitions can be used to derive strings that satisfy the definitions
- Here we derive the identifier \texttt{ir2\_D2} by applying rules in the definition of \texttt{<identifier>}

The numbers above the arrows are rule numbers.
Parts of a Recursive Definition

Always keep in mind that a recursive definition contains two parts:

- a *base case*, which is *non*-recursive and, consequently, terminates the recursive application of the rule. (Rules 4 and 5 are the base cases for `<identifier>`)

- a *recursive case*, which re-applies a rule. In BNF, these are the rules for non-terminals whose definition is self-referential. (Rules 6, 7, and 8 are the recursive cases for `<identifier>`)

13
Defining data structures recursively: linked lists

- It is possible to define the concept “linked list”, using a recursive definition:
  
  A linked list is null, or a node followed by a linked list

- In BNF, you could write:

  1)  `<LinkedList> := null |`
  2)  `node <LinkedList>`
Recursion & Data Structures: Linked List

A singly linked list viewed from a recursive perspective

non-empty linked lists (rule 2)

empty linked list (rule 1)

non-null successor provides the recursive case
null successor provides the base case
Recursive functions

• A recursive function or method is a function that calls itself, either directly or indirectly.

• Direct recursion: A is “directly recursive” if A contains a call to A.

• Indirect recursion: A is “indirectly recursive” if, for example, A calls function B, B calls function C, and C calls A.

• Often, recursive functions can be based on existing recursive definitions of what is to be computed…
A function that computes the Nth Fibonacci number can follow the recursive definition:

```c
/** Compute the value of the Nth Fibonacci number.
 * PRECONDITION: n>0
 * @param n which Fibonacci number to compute
 * @return the value of the nth Fibonacci number
 */

int fibonacci(int n) {
    // is n equal to 1 or 2? Then return 1

    // Otherwise, return the sum of the n-1 and n-2 Fibonacci nrs
}
```
A function for checking integer literals

- A function that tells whether a string is an integer literal constant can follow the recursive BNF definition (rewrite BNF in pseudocode, then use comment translation)

```java
/** Return true if a String represents an integer literal constant, false otherwise. */
boolean isIntConstant(String s) {
    // is it a single digit? then return true
    if(s.length()==1 && Character.isDigit(s.charAt(0)) )
        return true;
    // else is it a digit followed by an integer literal constant?
    // then return true. Note the recursive call!
    else if(s.length()>1 && Character.isDigit(s.charAt(0))
        && isIntConstant(s.substring(1)))
        return true;
    // else return false.. it doesn't fit the definition
    else return false;
}
```
A function for computing factorial

<table>
<thead>
<tr>
<th>Recursive Definition</th>
<th>Recursive Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n! )</td>
<td>\textbf{Function}</td>
</tr>
<tr>
<td>if ( n ) is 1, then ( n = 1 )</td>
<td>base case</td>
</tr>
<tr>
<td>if ( n &gt; 1 ), then ( n = n \times (n - 1)! )</td>
<td>recursive case</td>
</tr>
<tr>
<td></td>
<td>1 \ long factorial ( \ int n ) \ {</td>
</tr>
<tr>
<td></td>
<td>2 \ \text{if} \ (n == 1) \ \text{return} \ 1;</td>
</tr>
<tr>
<td></td>
<td>3 \ \text{return} \ n \times \text{factorial}(n - 1);</td>
</tr>
<tr>
<td></td>
<td>4 \ }</td>
</tr>
</tbody>
</table>
Recursion as a problem solving technique

• Recursion is a very useful problem solving technique
• If your problem P has these properties:
  – There is a *smallest size* instance of P that you can easily identify and solve directly (this is the base case)
  – For instances of P larger than the smallest one, you can break the solution into two parts: solving a *somewhat smaller* instance of the *same kind of problem* P, together with a little additional work to complete the solution (this is the recursive case)
• … then solve it recursively!
• This is a kind of ‘divide and conquer’ problem solving strategy
Example: linear search in an array

- Consider the problem of searching an array for a target

- Base case (smallest version of the problem):
  - The array has length 0. The target is not in the array. Done!

- Recursive case (larger than the smallest version):
  - Is the first element of the array equal to the target? If so, the target is in the array. Done!
  - Otherwise, solve a somewhat smaller instance of linear search: search for the target in the rest of the array
Example: linear search in an array

• Just translating that recursive, divide-and-conquer solution into code might look something like this:

```java
boolean search(Object[] a, Object target) {
    if (a.length = 0) return false;
    if  (target.equals(a[0])) return true;
    return  search ( restof(a), target );
}
```

• But Java does not provide a good, efficient way to implement that `restof()` function, that returns an array with the first element taken off!

• So we will have to implement the idea slightly differently…
Example: linear search in an array

- We will define a recursive ‘helper’ method that takes an additional argument, the index where to begin the search in the array:

```java
boolean search(Object[] a, Object target) {
    return search_recursive(a, 0, target);
}

boolean search_recursive(Object[] a, int start, Object t) {
    if (start >= a.length) return false;
    if (t.equals(a[start])) return true;
    return search_recursive(a, start + 1, t);
}
```
Time cost of linear search in an array

• How many times is search_recursive called when searching an array of length N?
  • In the worst case: N+1 times

• What is the time cost of each call?
  • It is constant: at most 3 statements are executed each time

• Therefore the total number of statements executed is no more than 3N + 3, and so the overall time cost is O(N)
Example: binary search in a sorted array

• Consider the problem of binary searching a sorted array for a target

• Base case (smallest version of the problem):
  • The array has length 0. The target is not in the array. Done!

• Recursive case (larger than the smallest version). Compare the target to the middle element of the array...
  • Is the target equal to that element? Target is in array. Done!
  • Is the target less than that element? Then binary search for the target in the ‘left half’ of the array
  – Else, binary search for the target in the ‘right half’ of the array
Example: binary search in a sorted array

• Just translating that recursive, divide-and-conquer solution into code might look something like this:

```java
boolean binary_search(Object[] a, Object target) {
    if (a.length == 0) return false;
    int mid = a.length/2;
    if (target.equals(a[mid])) return true;
    if (target.lessThan(a[mid])) return binary_search(lefthalf(a), target);
    else return binary_search(righthalf(a), target);
}
```

• Again, Java does not provide a good way to implement the firsthalf() and secondhalf() methods… so consider using a recursive helper method
Example: binary search in a sorted array

boolean binary_search(Object[] a, Object target) {
    return search_recursive(a, 0, a.length, target);
}

boolean search_recursive(Object[] a, int start, int end, Object t) {
    if (start >= end) return false;
    int mid = (end - start) / 2;
    if (t.equals(a[mid])) return true;
    if (t.lessThan(a[mid]))
        return search_recursive(a, start, mid, t);
    else return search_recursive(a, mid+1, end, t);
}

• Wait… what is that method lessThan()? This is an issue we will need to get back to later…
Time cost of binary search

• How many times is search_recursive called when binary searching an array of length N?
  • In the worst case: \( \log_2(N) + 1 \) times

• What is the time cost of each call?
  • It is constant: at most 5 statements are executed each time

• Therefore the total number of statements executed is no more than 5 \( \log_2(N) + 5 \), and so the overall time cost is \( O(\log N) \)
To understand how recursive methods work, it helps to understand activation records and the runtime stack.

A thread is a particular sequence of statements executed in a program.

The operating system gives each thread a runtime stack to store data and other information.

When a method is called in a thread, an activation record is created and pushed onto that thread's runtime stack.

When that call to the method returns, its activation record is popped from the top of the stack.

Note: the need to create activation records contributes to the time and space cost of recursive methods.
Recursion, Activation Records, Runtime Stack

• An activation record stores:
  – variables local to the invoked method
  – the parameters passed to the method
  – the method’s return value
  – administrative information necessary to restore the runtime environment of the calling method when the called method returns

• Variables in the activation record are said to be automatic variables
  – the memory is allocated on the stack automatically at run time when it is needed, and deallocated automatically when no longer needed
Activation Records & the Runtime Stack

Contents of an activation record

<table>
<thead>
<tr>
<th>Local Variables</th>
<th>Parameters</th>
<th>Administrative Information</th>
</tr>
</thead>
</table>

- **n = 4**
  - return value = ?
  - original call to factorial()

- **n = 3**
  - return value = ?
  - 1st recursive call

- **n = 2**
  - return value = ?
  - 2nd recursive call

- **n = 1**
  - return value = ?
  - 3rd recursive call

- **n = 4**
  - return value = 24
  - = 4 \* factorial(3)

- **n = 3**
  - return value = 6
  - = 4 \* 6
  - = 4 \* factorial(2)

- **n = 3**
  - = n \* factorial(n - 1)
  - = 3 \* factorial(2)

- **n = 2**
  - return value = 2
  - = 2 \* 1

- **n = 2**
  - return value = 2
  - = 2 \* factorial(1)

- **n = 3**
  - return value = ?

- **n = 4**
  - return value = ?

---

≤ method returns (recursion unwinding)

State of the process stack and activation records for an initial call to factorial(4)
Recursion and iteration

• In computer programming, you don’t need recursion, if you have iteration; and vice versa.

• They are computationally equivalent:
  – Any while-loop can be recoded as a recursive function
  – Any recursive function can be recoded as a while-loop (plus perhaps some dynamically allocated data structures... like a stack!)

• So how do you decide which technique to use?
Recursion and iteration

• Sometimes recursion is the way to go:
  – For a given problem, a recursive solution can be much faster to write and easier to understand and prove correct
  – Examples: BNF syntax checking, Fibonacci, towers of Hanoi, region pixel counting...

• But there are tradeoffs…
Recursion and iteration tradeoffs

• For a given problem, an iterative solution will usually run faster and use less memory than a recursive solution (on typical machine architectures), because of the added time and space required to create, push, and pop activation records.

• ... but the iterative solution may be take longer to develop and write in the first place.

• So the rule of thumb is:
  – If the recursive solution is easy to write, use it.
  – If runtime efficiency is really important, then take the extra time to recode it as an iterative solution.
# Recursion and Iteration

## Components of a loop and loop terminology

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>loop entry condition</strong></td>
<td>Also called the <em>continuation condition</em>; this condition determines if the loop body is to be executed.</td>
</tr>
<tr>
<td><strong>loop exit condition</strong></td>
<td>The condition under which the loop exits; the negation of the entry condition.</td>
</tr>
<tr>
<td><strong>loop control variable (LCV)</strong></td>
<td>The variable in the loop entry condition that determines if the loop terminates or executes again; the LCV must be updated so that eventually the exit condition is met.</td>
</tr>
<tr>
<td><strong>loop body</strong></td>
<td>The block of code that executes each time the loop entry condition is met.</td>
</tr>
</tbody>
</table>
Recursion as Iteration

Comparison of elements of a loop and a recursive function

<table>
<thead>
<tr>
<th>Loop</th>
<th>Recursive Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop control variable</td>
<td>method input</td>
</tr>
<tr>
<td>loop exit condition</td>
<td>base case</td>
</tr>
<tr>
<td>loop entry condition</td>
<td>recursive case</td>
</tr>
<tr>
<td>loop body</td>
<td>method body</td>
</tr>
</tbody>
</table>
Iterative and Recursive Solutions

long factorial ( int n ) {
    long f = 1;
    while ( n > 1 ) {
        // entry condition met;
        // iterate again
        f = f * n;
        n = n - 1; // update LCV to get
                    // closer to exit
                    // condition and
                    // iterate again
    }
    // exit condition: iteration ends
    // return result f is n! for n >= 0
    return f;
}

long factorial ( int n ) {
    if ( n == 1 ) // base case; iteration ends
        return 1; // return a result
    else
        return n * factorial( n - 1 );
    }

Tail Recursion

• A recursive method call is **tail recursive** if the recursive call is *the last statement to be executed* in the method body before the method returns.

• Tail recursive methods are of interest because they can easily be converted to use a loop to do the iteration.

• Why bother to do that? Because the iterative version will usually be more time and space efficient...
An Example

void printArray( int[] values, int n ) {
    if ( n == values.length ) // base case
        return;
    System.out.println( values[n] );
    n++;
    printArray( values, n ); // recursive case
}

tail recursive – data in this activation record will never be used again
Convert Tail Recursion to a Loop

<table>
<thead>
<tr>
<th>Recursive Version</th>
<th>Iterative Version <em>(Tail Recursion Removed)</em></th>
</tr>
</thead>
</table>
| void printArray(int[] values, int n){  
  if ( n == values.length ) // base case
    return;
  System.out.println(values[n]);
  n++; 
  printArray(values, n); // recursive // case | void printArray(int[] values, int n){  
  while ( n != values.length ){
    System.out.println(values[n]);
    n++; 
  }
} |
Recursion space costs

How much **space** activation records require depends on two factors:

- the size of the activation record, which depends on how many parameters and local variables there are
- the **depth** of the recursion; that is, how many calls will be made before the base case is met, at which point no more recursive calls are made

The *relative* time cost of the method invocation goes up as the execution time of the body goes down
Dynamic Programming

• Occasionally, a recursive definition provides a nice, clear decomposition of a problem, but a straightforward implementation results in duplication of effort.

• **Space-Time Tradeoff:** it can be much faster to compute a result once and save the value in memory for reuse later, rather than re-compute it each time it is needed.

• Dynamic programming is a technique that can dramatically speed up recursively-defined algorithms, but with somewhat increased memory cost.
Example: Computing Fibonacci Numbers

fib\_n = \begin{cases} 
0, & \text{if } n == 0 \\
1, & \text{if } n == 1 \\
\text{fib}_{n-1} + \text{fib}_{n-2}, & \text{if } n > 1 
\end{cases}

Call tree for computing the sixth Fibonacci number.

<table>
<thead>
<tr>
<th>n</th>
<th>Number of Times fib(n) Is Computed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Count of number of times a Fibonacci number is computed while computing fib(6)
Recursive & Dynamic Programming Versions

```java
/**
 * Dynamic programming version of the Fibonacci function.
 */
public class DynamicFibonacci {
    private long[] fibs; // store the fibs here
    private int lastComputedFib;
    private int capacity;

    public long getFib( int n ) { // compute and store for reuse
        int i;
        if ( n > capacity ) return -1;
        lastComputedFib++;
        for ( ; lastComputedFib <= n; lastComputedFib++ )
            fibs[lastComputedFib] = fibs[lastComputedFib - 1] +
                                   fibs[lastComputedFib - 2];
        // undo the last ++ from the last iteration of the for loop
        lastComputedFib--;
        return fibs[lastComputedFib];
    }

    // recursive version
    long fibonacci( int n ) {
        if ( ( n == 0 ) || ( n == 1 ) )
            return n;
        return fibonacci( n - 1 ) +
               fibonacci( n - 2 );
    }
}
```
Next time

- The PriorityQueue ADT
- PriorityQueue implementations
- Trees and tree terminology
- Binary trees
- Heaps and heap implementations
- Heapsort

Reading: Gray, Ch 10