1 **Recursion**

CST141

2 **Recursion**

- Earlier programs in CST141 have been structured as methods that call one another in a disciplined, hierarchical manner
- Recursive methods *call themselves*
  - Called directly or indirectly through another method
  - Useful and often can be a more intuitive *alternative to iteration/repetition*

3 **Recursion Concepts (Page 1)**

- There are two parts to recursion:
  - Base case—the recursive method is capable of directly solving only this simplest case
    - If the problem is easy, solve it immediately, e.g. "Are we done yet?"
    - When the method is called that contains the base case (e.g. the simple problem), the method returns the result (the piece it knows how to do)

4 **Recursion Concepts (Page 2)**

- There are two parts to recursion: (*con.*)
  - Recursive call/recursion step—resembles the original problem, but is a slightly more complex or larger version of it
    - Normally within a return statement, the method calls a fresh copy of itself to work on the "slighty" smaller problem

5 **Recursion Concepts (Page 3)**

- With each new recursive call to itself, the problem gets smaller and smaller until it "converges" on the base problem

6 **Recursion Example: Breaking Rock into Dust**

- Using the two parts of recursion:
  - Recursive call/recursion step—if the problem cannot be solved immediately, divide it into smaller problems; then solve the smaller problem
    - To destroy rock, hit it with the hammer so that it shatters into smaller pieces
    - Apply the same procedure to the pieces
  - Base problem—if the problem is easy, solve it immediately
    - When a piece is small enough, stop hitting it

7 **Recursion Example: Factorials**

- Factorial of n, or $n!$ is the product of:
  - $n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$
  - For example: $4! = 4 \times 3 \times 2 \times 1 = 24$
  - By definition $1! = 1$ and $0! = 1$
- The recursive solution uses the *algorithmic* relationship:

  $\underline{n! = n \cdot (n - 1)!}$

  $2! = 2 \cdot (2 - 1)! = 2$

$3! = 3 \cdot (3 - 1)! = 6$

$4! = 4 \cdot (4 - 1)! = 24$

$5! = 5 \cdot (5 - 1)! =$

10 **Recursion vs. Iteration (Page 1)**

- Any problem that can be solved recursively can be solved iteratively
- Both iteration and recursion use a *control statement*
  - Iteration uses a repetition statement (for or while)
  - Recursion uses a selection statement (if…else)

11 **Recursion vs. Iteration (Page 2)**

- Iteration and recursion both involve the use of a *termination test*
  - Iteration terminates when the loop-continuation condition fails
  - Recursion terminates when a *base case is reached*
- Recursion may be more "expensive" in terms of processor time and memory space, but usually provides a more intuitive solution

19 **Recursion Advantages**

- Main advantage is that recursion can be used to create *clearer*, *simpler* versions of several algorithms
  - As opposed to alternative algorithms using an iterative (looping) solution
- A recursive approach may be implemented with fewer lines of code
- Select recursive approach when iterative one might not be apparent

20 **Recursion Disadvantages (Page 1)**

- Recursive applications may execute a bit more slowly than their iterative equivalent
  - Added overhead of the additional method calls (may use more memory and CPU time)
  - Avoid using recursion in situations requiring high performance

21 **Recursion Disadvantages (Page 2)**

- Many recursive calls to a method could cause a stack overrun
  - Due to extra storage for parameters and local variables (stack could become exhausted)
  - If this occurs, the Java run-time system will throw an exception
  - Not usually a concern for standard problems

31 **Recursive Methods and the Stack (Page 1)**

- The method call stack is used to keep track of method calls as well as local variables (including parameters) within a method call
- When a method calls itself, new local variables are allocated storage on the stack (plus a pointer to the address of the method call)
  - The recursive call does not make a new copy of the method; only the local variables are new

32 **Recursive Methods and the Stack (Page 2)**

- As the recursive method executes return, its local variables are removed from the stack
  - Previous recursive calls resume executing at the point of the call inside the method and retrieve the copy of its local variables
- Variables of current method executing are always at “top of stack”
- Recursive method calls often said to be “telescoping” out and back

34 **Fibonacci Numbers (Page 1)**

- In mathematics the Fibonacci numbers are a series numbers in the following integer sequence called the Fibonacci sequence
- They are characterized by the fact that every number after the first two is the sum of the two preceding ones, e.g.
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

35 **Fibonacci Numbers (Page 2)**

- Often, especially in modern useage, the sequence is extended by one more initial turn, e.g.
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
- Depending on the chosen starting point, by definition the first two numbers are either:
  - 1 and 1
  - 0 and 1
- Each subsequent number is the sum of the previous two

36 **Fibonacci Numbers (Page 3)**

- Sequence F_n of Fibonacci numbers is defined by recurrence relation:
  - $F_n = F_{n-1} + F_{n-2}$

38 **Recursive Backtracking**

- Recursive backtracking is the process of returning to an earlier decision point using recursion
- If one set of recursive calls does not result in a solution, program backs up to previous decision point and makes different decision
  - Often results in another set of recursive calls

39 **The Tower of Hanoi**

- The “Tower of Hanoi” puzzle can be solved with recursive backtracking
- https://www.youtube.com/watch?v=buWXDMbY3Ww
  - (Start at 5 minutes)
-

40 **The Tower of Hanoi Algorithm**

- The method call:

  moveDisks(n – 1, fromTower, auxTower, toTower)
- The algorithm for the method:

  if (n == 1)

    Move disk 'n' from the fromTower to the toTower

```
else
{
    moveDisks(n – 1, fromTower, toTower, auxTower)
    Move disk 'n' from the fromTower to the toTower
    moveDisks(n – 1, auxTower, fromTower, toTower)
}
```

43 **Merge Sort** **(Page 1)**

- The process of *sorting an array* involves putting it into sequence, either ascending or descending
- The merge sort is an efficient sorting algorithm that conceptually is more complex than selection sort and insertion sorts
- Sorts an array by splitting it into two equal-sized sub-arrays, sorting each sub-array, then merging them into one larger array

44 **Merge Sort** **(Page 2)**

- The implementation of the merge sort in this example is recursive
  - The base case is an array with one element (which of course is sorted already), so the merge sort immediately returns in this case
  - Recursion step splits array into two approximately equal pieces, recursively sorts them, then merges the two sorted arrays into one larger, sorted array