Overview

- what is recursion?
- recursion vs. iteration
- analyzing the running time of recursive algorithms
Recursion

- The term *recursion* refers to a method that calls itself.
- Recursion is a powerful programming technique that results in efficient algorithms with concise descriptions.
Recursion example

- Suppose we replace every line segment by eight shorter line segments according to the following geometric rule:

- By applying this rule recursively we obtain the following “fractal”: 

![Recursive Fractal Diagram]
Fractals

- Fractals are beautiful “creatures” often built on the recursion principle:
Iteration versus Recursion

- An iterative algorithm for solving a problem $P$ makes use of a loop to compute a sequence of analogous steps that solve $P$.

$$n! = n(n-1)(n-2)\ldots3\cdot2\cdot1$$

- A recursive algorithm for solving a problem $P$ computes one step and calls itself to solve the remaining subproblem.

$$n! = \begin{cases} 
1 & n \leq 1 \\
n(n-1)! & n > 1 
\end{cases}$$
Iteration versus Recursion

- An iterative algorithm for computing $n!$:
  ```c
  int factorialIterative(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;
    return result;
  }
  ```

- A recursive algorithm for computing $n!$:
  ```c
  int factorialRecursive(int n) {
    int result = 1;
    if (n > 1) result = n * factorialRecursive(n - 1);
    return result;
  }
  ```
Tracing a Recursive Function Call

\[
\text{factorialRecursive}(5) = 5 \times \text{factorialRecursive}(4);
\]

\[
\text{factorialRecursive}(4) = 4 \times \text{factorialRecursive}(3);
\]

\[
\text{factorialRecursive}(3) = 3 \times \text{factorialRecursive}(2);
\]

\[
\text{factorialRecursive}(2) = 2 \times \text{factorialRecursive}(1);
\]

\[
\text{factorialRecursive}(1) = 1;
\]

- For the factorial problem, both iterative and recursive functions run in time linear to \( n \).
Fibonacci numbers

- The Fibonacci numbers are the sequence

\[ 0, 1, 1, 2, 3, 8, 13, 21, 34, 55, \ldots \]

- The first two numbers in the sequence are 0 and 1.
- Each succeeding number (third, fourth, ...) is defined as the sum of the two numbers that precede it in the sequence.
- That is, Fibonacci numbers are defined recursively:

\[
\text{fib}(n) = \begin{cases} 
0 & n = 0 \\
1 & n = 1 \\
\text{fib}(n) + \text{fib}(n - 1) & n > 1 
\end{cases}
\]
A Recursive Solution

```c
int fib(int n) {
    int result = 0;

    if (n == 1)
        result = 1;
    else if (n > 1)
        result = fib(n-1) + fib(n-2);

    return result;
}
```

Unlike our solution for computing factorial numbers, every call to fib generates two recursive calls.
Recursion Inefficiency

- This implementation of Fibonacci numbers is inefficient because it recomputes values that have already been found.

- A more efficient solution can be obtained by storing Fibonacci numbers that have already been computed in an array.
Efficient Fibonacci Computation

```java
int fib2(int n) {
    int [] F = new int [n+1];
    for (int i = 0 ; i < n + 1 ; i++) F[i] = -1;
    return fibMemo(n, F);
}

int fibMemo(int n, int [] F) {
    if (n <= 1) F[n] = n;
    else if (n > 1 && F[n] == -1)
        F[n] = fibMemo(n-1, F) + fibMemo(n-2, F);
    return F[n];
}
```

- fib2 is a public function:
  - It calls its private fibMemo function.
  - fib2 just initializes all values to -1
  - A value of -1 for entry $F[i]$ means the Fibonacci number of $i$ is not computed yet.
Recursion Tree

- Blue numbers indicate the order at which the functions are called.
- When the value of $F[i]$ is not -1 (is already computed), the recursion functions are not called (the calculated number is returned).
Recursion Tree for Efficient Recursion Trees

- For large values of $n$, the tree looks like this:
  - Nodes with the same color are calling the same function (with the same parameter)
Tail Call

• A tail call is a method \( B \) called inside a method \( A \), whose return value is immediately returned by the calling method \( A \).

```java
int methodA(int x) {
    int y = x + 2;
    return methodB(y);
}
```

• No code is executed in the calling method \( A \) after returning from the method call to \( B \).

• advantage: no need to store local variables on the call stack \( A \) tail recursive algorithm uses tail calls.
Tail Recursion

- A tail call to the same function!
  - Tail recursion often requires an additional parameter that accumulates a partial solution.
- E.g.: summing integer values in a linked-list using tail recursion:
  - Here partialSum accumulates partial solution!

```java
int listSumTail(Node node) {
    return listSumTailHelper(0, node);
}

int listSumTailHelper(int partialSum, Node node) {
    int result = partialSum;

    if (node != null)
        result = listSumTailHelper(partialSum + node.getValue(), node.getNext());

    return result;
}
```

```java
int listSumTail(Node node) {
    return listSumTailHelper(0, node);
}
```
Tail Recursion

Summing integer values in a linked-list without using tail recursion:

```java
int listSum(Node node) {
    int result = 0;

    if (node != null)
        result = node.getValue() + listSum(node.getNext());

    return result;
}
```
Tail Recursion

- summing integer values in a linked-list using tail recursion:

```
head → 16 → 30 → 42 → NULL

0 + 16 = 16
16 + 30 = 46
46 + 42 = 88
```

- summing integer values in a linked-list without using tail recursion:

```
head → 16 → 30 → 42 → NULL

16 + 72 = 88
30 + 42 = 72
42 + 0 = 42
0
```
Tail Recursion Summary

- When writing a recursive code, consider applying the tail recursion.
- Tail recursion can significantly improve the memory usage of your algorithm (in the stack memory).
- The time complexity does not change (asymptotically).
- You often need to pass partial solutions in the recursive calls.
Recursion vs. Iteration

- Every recursive algorithm can be written iteratively.
- A recursive algorithm (what does it do?)
  ```java
  void recursiveFn(int n) {
    System.out.println("n = " + n);
    if (n > 1)
      recursiveFn(n/2);
  }
  ```

- Iterative version of the same algorithm
  ```java
  void iterativeFn(int n) {
    for (int i = n ; i >= 1 ; i /= 2)
      System.out.println("n = " + i);
  }
  ```

- These two algorithms run in the same time!
Recursion vs. Iteration

- What is the running time of iterativeFn?
- How many times does the for loop iterate?
  - On the first iteration \( i = n \)
  - On the second iteration \( i = n/2 \)
  - On the third iteration \( i = n/4 \)
  - On the fourth iteration \( i = n/8 \)
  - \( \ldots \)
  - On the \( k \)th iteration \( i = \frac{n}{2^{k-1}} \)
  - \( \ldots \)
  - On the last iteration \( i = 1 \)

- The loop stops when \( i < 1 \)
  - \( \frac{n}{2^{k-1}} < 1 \rightarrow n < 2^{k-1} \rightarrow \log n < k - 1 \rightarrow k > \log n + 1 \)
- Therefore, the loop iterates \( \log n + 1 \) times!
  - Remember! \( \log n \) always mean \( \log_2(n) \)
Determining the Running Time of Recursive Algorithms

- We don't need to rewrite a recursive algorithm as an iterative algorithm to find its running time.
- A recurrence relation can be used to express the running time of a recursive algorithm.
- Solving recurrence relations requires using a proof by induction.
- Mathematical induction and recurrence relations are covered in COMP 2080 and COMP 2130.
Next Topic

- Next topic is sorting.
- What is a **Comparison Based Sorting**?