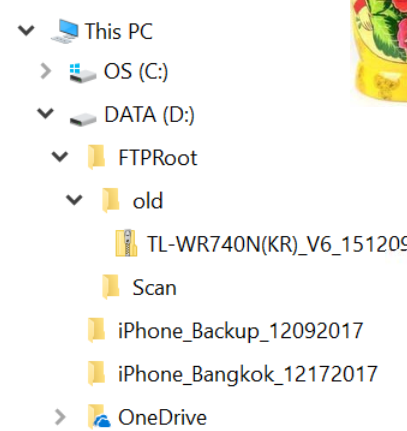
CSE214 Data Structures
Recursion

YoungMin Kwon

Repetition

- Loops
  - for-loop, while-loop

- Recursion
  - Russian Matryoshka dolls



  - File system

  - Fractal patterns
    - https://www.youtube.com/watch?v=foxD6ZQlnlU

SUNY Korea

Factorial Function

- Definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

- Recursive definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases}$$

  - Base case
  - Recursive case

Factorial Function

```
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException( );      // argument must be nonnegative
  else if (n == 0)
    return 1;                                   // base case
  else
    return n * factorial(n−1);                  // recursive case
}
```

Drawing an English Ruler

```
---- 0
-
--
-
---
-
--
-
---- 1
-
--
-
---
-
--
-
---- 2
```

(a)

```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
```

(b)

```
--- 0
-
--
-
--- 1
-
--
-
--- 2
-
--
-
--- 3
```

(c)

(a) a 2-inch ruler with major tick length 4;
(b) a 1-inch ruler with major tick length 5;
(c) a 3-inch ruler with major tick length 3.

English Ruler

- English ruler is a simple fractal
  - Self-recursive structure at various levels of magnification
- An interval with a central tick length $L \geq 1$ has
  - An interval with a central tick length L - 1
  - A single tick of length L
  - An interval with a central tick length L - 1

```
/** Draws an English ruler for the given number of inches and major tick length. */
public static void drawRuler(int nInches, int majorLength) {
  drawLine(majorLength, 0);                    // draw inch 0 line and label
  for (int j = 1; j <= nInches; j++) {
    drawInterval(majorLength − 1);             // draw interior ticks for inch
    drawLine(majorLength, j);                  // draw inch j line and label
  }
}
private static void drawInterval(int centralLength) {
  if (centralLength >= 1) {                    // otherwise, do nothing
    drawInterval(centralLength − 1);           // recursively draw top interval
    drawLine(centralLength);                   // draw center tick line (without label)
    drawInterval(centralLength − 1);           // recursively draw bottom interval
  }
}
```

```
private static void drawLine(int tickLength, int tickLabel) {
  for (int j = 0; j < tickLength; j++)
    System.out.print("-");
  if (tickLabel >= 0)
    System.out.print(" " + tickLabel);
  System.out.print("\n");
}
/** Draws a line with the given tick length (but no label). */
private static void drawLine(int tickLength) {
  drawLine(tickLength, -1);
}
```

```
---- 0
-
--
-
---
-
--
-
---- 1
-
--
-
---
-
--
-
---- 2
```

Binary Search

- Efficient way of locating a target value within a sorted sequence of *n* elements

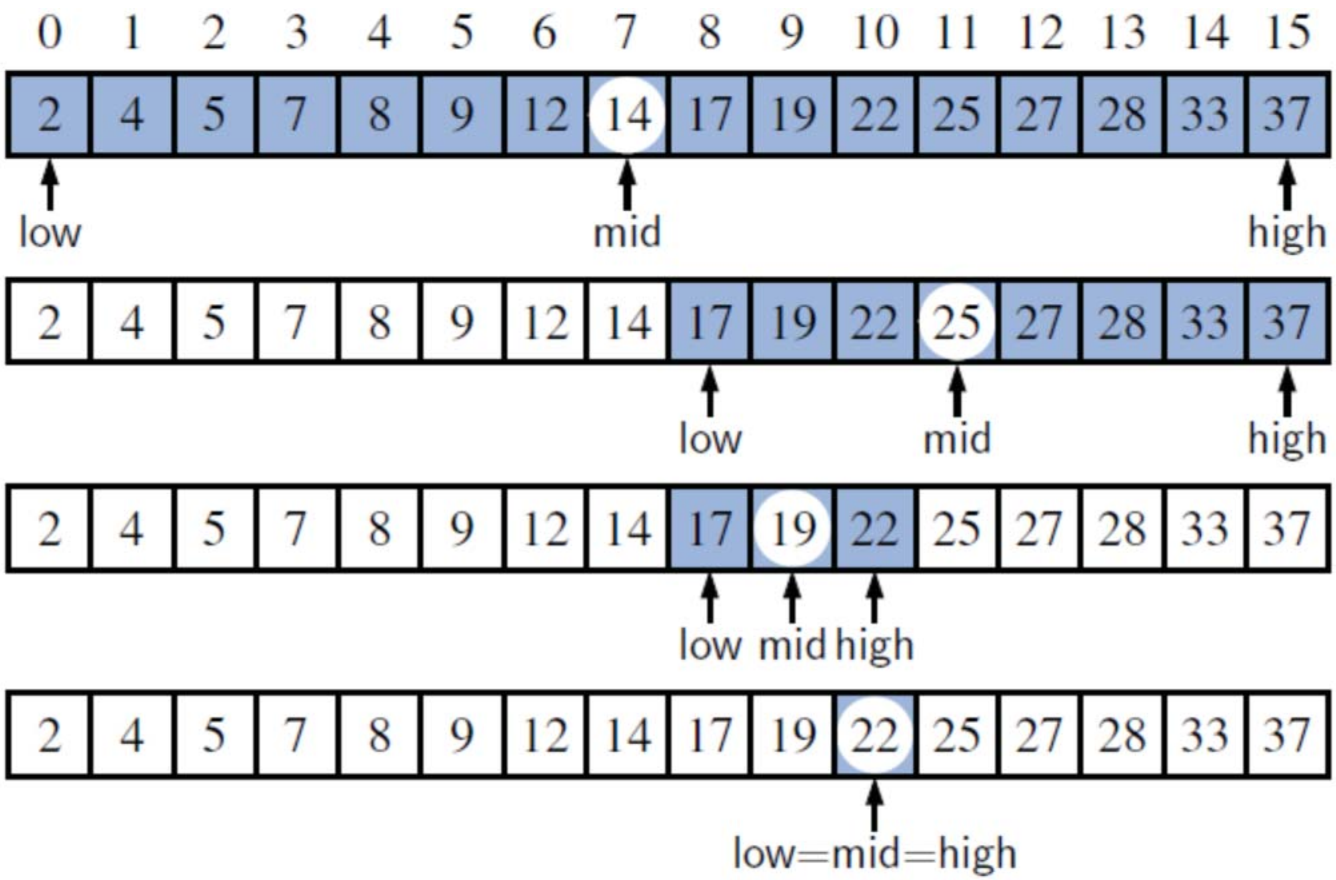
$$\text{Initially, low} = 0 \text{ and high} = n - 1$$

$$\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor \,.$$

- Three cases
  - target == data[mid]: found!
  - target < data[mid]: search within low … mid – 1
  - target > data[mid]: search within mid + 1 … high

Binary Search

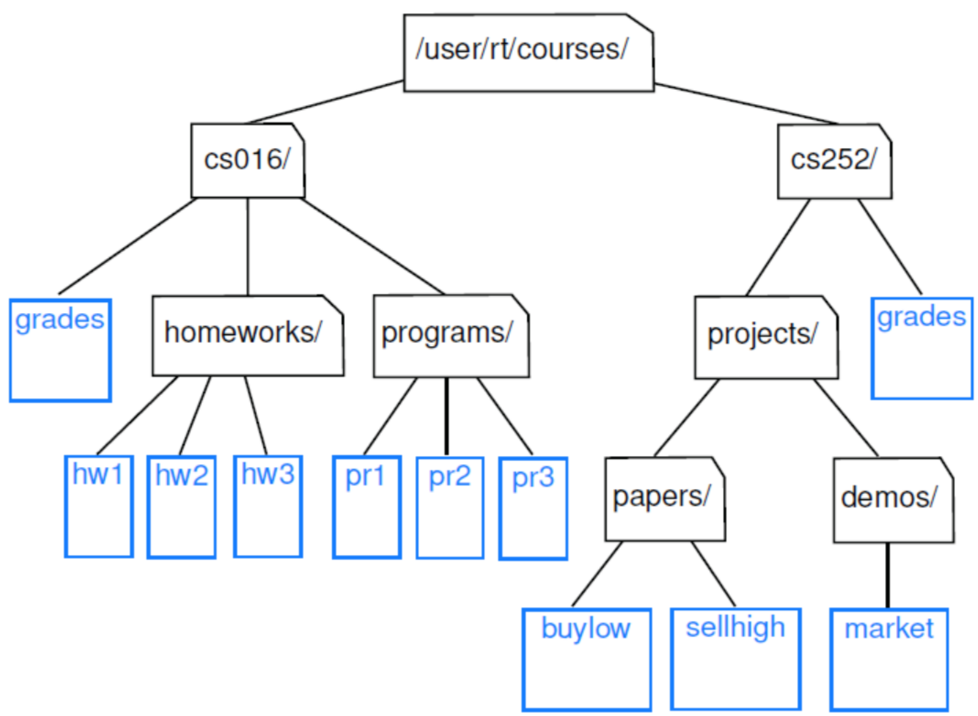
- A binary search for 22



```
/**
 * Returns true if the target value is found in the indicated portion of the data array.
 * This search only considers the array portion from data[low] to data[high] inclusive.
 */
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
  if (low > high)
    return false;                                       // interval empty; no match
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true;                                      // found a match
    else if (target < data[mid])
      return binarySearch(data, target, low, mid − 1);  // recur left of the middle
    else
      return binarySearch(data, target, mid + 1, high); // recur right of the middle
  }
}
```

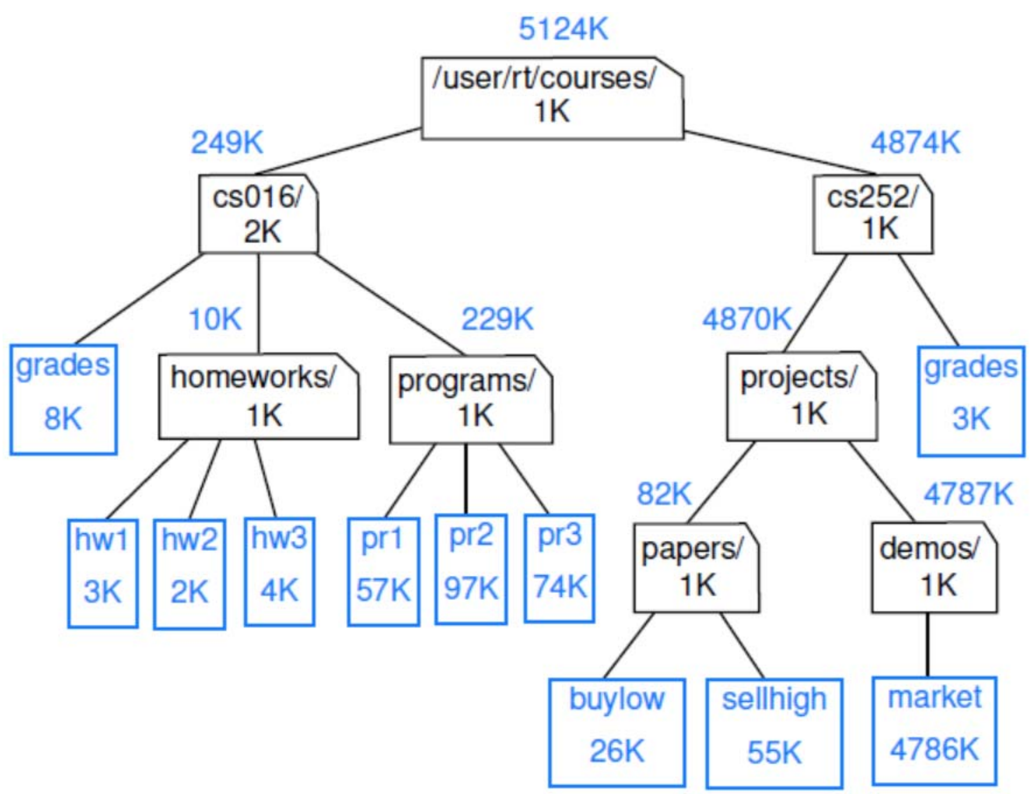
Disk Usage in File Systems

- Modern operating systems define file system directories in a recursive way
  - Directories contain files and other directories...



Disk Usage in File Systems

- Disk usage
  - Immediate disk space used by each entry
  - Cumulative disk space used by that entry and all nested features



Disk Usage in File Systems

- `java.io.File` class
  - `new File(pathString)` or `new File(parentFile, childString)`: create a File instance
  - `file.length()`: disk usage of the file
  - `file.isDirectory()`: true iff file is a directory
  - `file.list()`: name of all entries within the directory

Disk Usage in File Systems

```
/**
 * Calculates the total disk usage (in bytes) of the portion of the file system rooted
 * at the given path, while printing a summary akin to the standard 'du' Unix tool.
 */
public static long diskUsage(File root) {
  long total = root.length();                      // start with direct disk usage
  if (root.isDirectory()) {                        // and if this is a directory,
    for (String childname : root.list()) {         // then for each child
      File child = new File(root, childname);      // compose full path to child
      total += diskUsage(child);                   // add child's usage to total
    }
  }
  System.out.println(total + "\t" + root);         // descriptive output
  return total;                                    // return the grand total
}
```

Disk Usage in File Systems

8	/user/rt/courses/cs016/grades
3	/user/rt/courses/cs016/homeworks/hw1
2	/user/rt/courses/cs016/homeworks/hw2
4	/user/rt/courses/cs016/homeworks/hw3
10	/user/rt/courses/cs016/homeworks
57	/user/rt/courses/cs016/programs/pr1
97	/user/rt/courses/cs016/programs/pr2
74	/user/rt/courses/cs016/programs/pr3
229	/user/rt/courses/cs016/programs
249	/user/rt/courses/cs016
26	/user/rt/courses/cs252/projects/papers/buylow
55	/user/rt/courses/cs252/projects/papers/sellhigh
82	/user/rt/courses/cs252/projects/papers
4786	/user/rt/courses/cs252/projects/demos/market
4787	/user/rt/courses/cs252/projects/demos
4870	/user/rt/courses/cs252/projects
3	/user/rt/courses/cs252/grades
4874	/user/rt/courses/cs252
5124	/user/rt/courses/

Analysis: Computing Factorials

- Total $n + 1$ activations: the parameter decreases from n to 0
  - Each individual activation executes a constant number of operations
  - factorial(n) is $O(n)$

```
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException();      // argument must be nonnegative
  else if (n == 0)
    return 1;                                   // base case
  else
    return n * factorial(n−1);                  // recursive case
}
```

Analysis: Drawing an English Ruler

```
private static void drawInterval(
                        int centralLength) {
  if (centralLength >= 1) {
    drawInterval(centralLength − 1);
    drawLine(centralLength);
    drawInterval(centralLength − 1);
  }
}
```

- Drawing an English Ruler
  - How many lines are generated by drawInterval(c), where c denotes the center length
  - Each drawInterval(c) for c > 0, spawns two calls to drawInterval(c-1) and one call to drawLine
- drawInterval(n) is $O(2^n)$

Analysis: Drawing an English Ruler

- Proposition
  - For $c \geq 0$, a call to *drawInteval(c)* results in $2^c - 1$ *lines* of output

- Justification (by induction)
  - Base step: drawInterval(0) generates no output. That is, $2^0 - 1 = 0$.
  - Induction step: if drawInterval(c - 1) prints $2^{c-1} - 1$ lines, drawInterval(c) draws $1 + 2 \cdot (2^{c-1} - 1) = 2^c - 1$ lines.

Analysis: Binary Search

```
public static boolean binarySearch(int[ ] data, int target,
                                   int low, int high) {
  if (low > high)
    return false;
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true;
    else if (target < data[mid])
      return binarySearch(data, target, low, mid − 1);
    else
      return binarySearch(data, target, mid + 1, high);
  }
}
```

- Binary Search
  - During each recursive call, a constant number of operations are executed
  - Running time is proportional to the number of recursive calls performed
- Proposition
  - binarySearch runs in $O(\log n)$ time for a sorted array with n elements

SUNY Korea

Analysis: Binary Search

- Justification
  - The number of elements to search is *high – low + 1*
  - The number of remaining candidates after a recursive call is either

$$(\mathrm{mid}-1)-\mathrm{low}+1 = \left\lfloor \frac{\mathrm{low}+\mathrm{high}}{2} \right\rfloor - \mathrm{low} \leq \frac{\mathrm{high}-\mathrm{low}+1}{2}$$

or

$$\mathrm{high}-(\mathrm{mid}+1)+1 = \mathrm{high} - \left\lfloor \frac{\mathrm{low}+\mathrm{high}}{2} \right\rfloor \leq \frac{\mathrm{high}-\mathrm{low}+1}{2}$$

  - After the j^{th} call, the number of remaining candidates is at most $n/2^j$

Analysis: Binary Search

- Justification (cont'd)
  - The maximum number of recursive calls is the *smallest integer r* such that

$$\frac{n}{2^r} < 1$$

  - r is the smallest integer such that r > log n. That is

$$r = \lfloor \log n \rfloor + 1$$

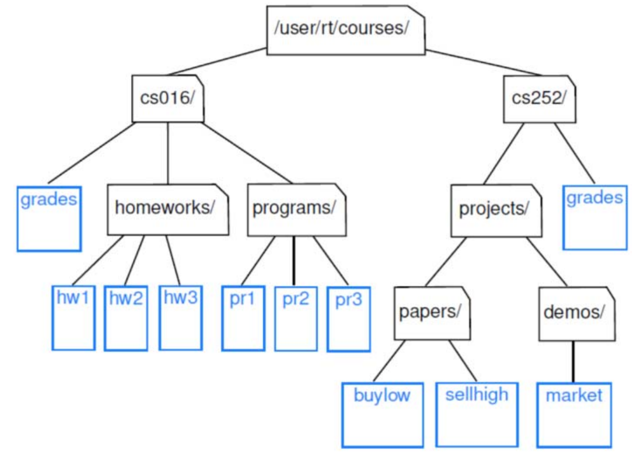
  - That is, binarySearch is *O(log n)*

Analysis: DiskUsage

```
public static long diskUsage(File root) {
  long total = root.length();
  if (root.isDirectory()) {
    for (String childname : root.list()) {
      File child = new File(root, childname);
      total += diskUsage(child);
    }
  }
  System.out.println(total + "\t" + root);
  return total;
}
```

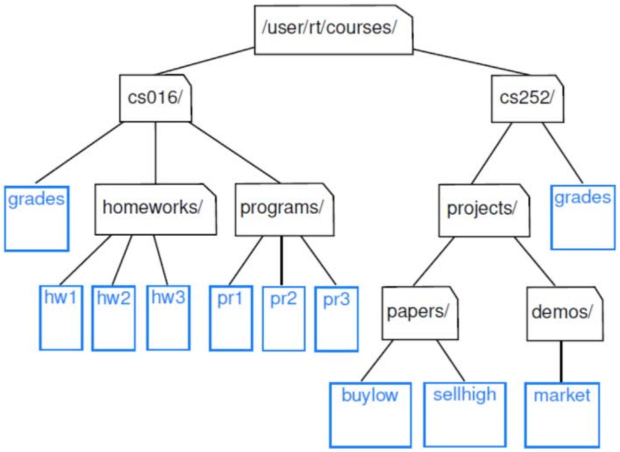
- Analysis
  - Let *n* be the number of file system entries
  - Find the *number of recursive invocations*
  - Find the *number of operations executed in each invocation*

Analysis: DiskUsage



- *The number of recursive invocations is n*
  - Nesting level
    - Entries at the initial level has a nesting level of 0
    - Entries stored within it has a nesting level of 1
    - Entries stored within those has a nesting level of 2, …

Analysis: DiskUsage



- *The number of recursive invocations is n*
  - *There is only one invocation for each entry*
    - Base step: when the nesting *level k is 0*, the only recursive invocation is the initial one
    - Induction step: if there is exactly one invocation for each entry in nesting *level k*, there is exactly one invocation of each entry in nesting *level k + 1*.

Analysis: DiskUsage

```
public static long diskUsage(File root) {
  long total = root.length();
  if (root.isDirectory()) {
    for (String childname : root.list()) {
      File child = new File(root, childname);
      total += diskUsage(child);
    }
  }
  System.out.println(total + "\t" + root);
  return total;
}
```

- The number of operations in each invocation
  - diskUsage has a *for loop* in it.
    - Worst case: the initial root contains n-1 others
    - Is diskUsage $O(n^2)$: n invocation of diskUsage and each invocation takes $O(n)$?
  - Overall number of iterations
    - There are precisely n - 1 iterations
    - Because each iteration makes 1 diskUsage call and there are total n invocations (including the initial one)

Analysis: DiskUsage

```
public static long diskUsage(File root) {
  long total = root.length();
  if (root.isDirectory()) {
    for (String childname : root.list()) {
      File child = new File(root, childname);
      total += diskUsage(child);
    }
  }
  System.out.println(total + "\t" + root);
  return total;
}
```

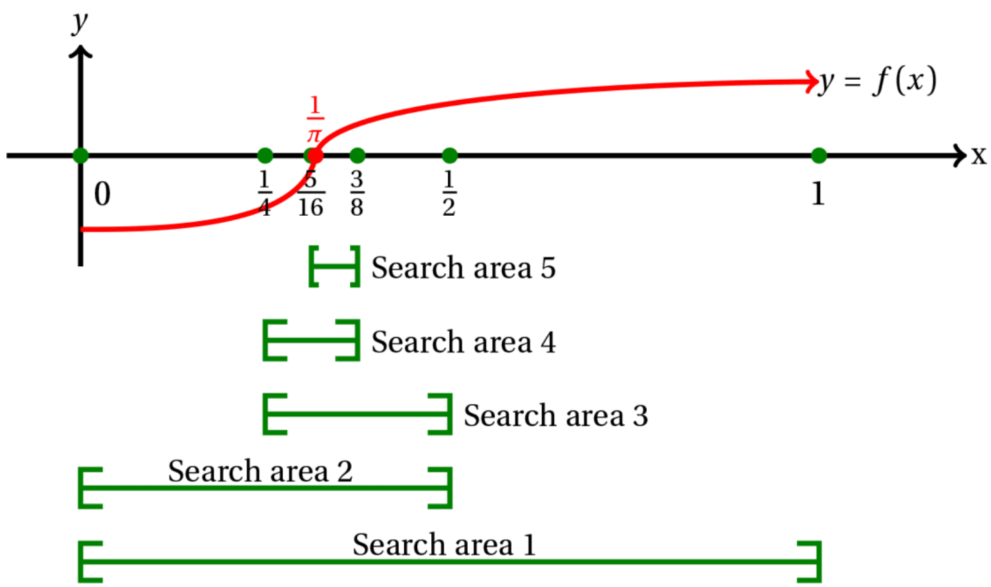
- diskUsge is O(n)
  - There are n invocations of diskUsage, each of which uses O(1) outside of the loop
  - The overall number of operations due to the loop is O(n)
  - diskUsage is $O(n)$: $n \cdot O(1) + O(n)$

Assignment 6

- In this assignment you will implement the following numerical methods that can *find a root of an equation*
  - Bisection
  - Newton's method
- Due date: 4/28/2022

Bisection

- Given an interval [a, b], recursively halve the interval until | a – b | < ε
- Let m = (a + b)/2, then the halved interval is
  - [a, m] if $f(a) \cdot f(m) \leq 0$
  - [m, b] if $f(b) \cdot f(m) \leq 0$
  - Error otherwise



Bisection

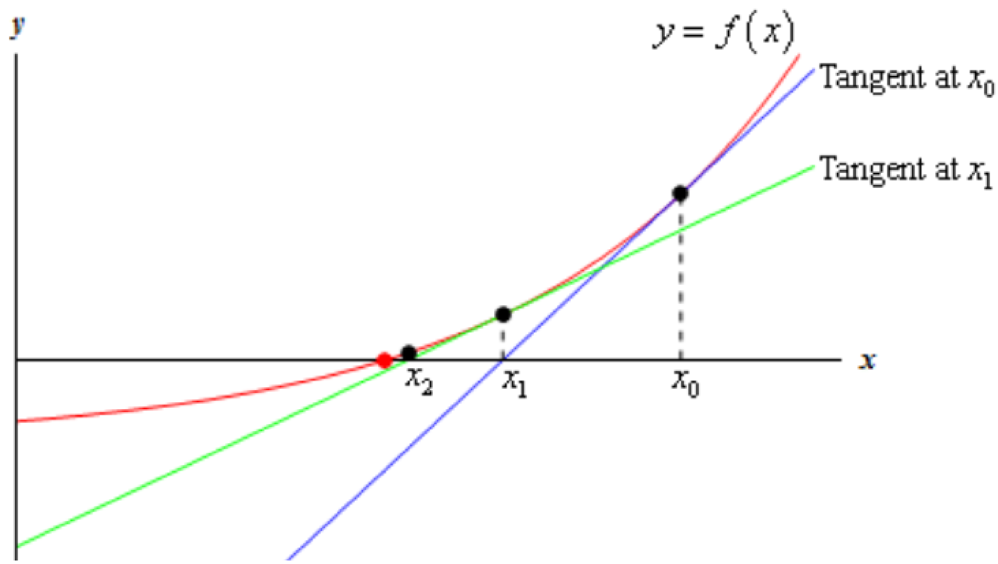
```
public class Lambda {
    //EPS is a small number
    public static final double EPS = 1e-10;

…
    //Bisection method
    public interface Bisection {
        //Single Abstract Method
        public double fun(double x);

        //Solve: finds x such that fun(x) = 0
        public default double solve(double xl, double xr) {
            /*TODO: implement this function recursively
                    repeat until xr - xl < EPS*/
    }
…
```

Newton's Method

- Newton's method is a numerical method that can find a root of an equation as below
  - $x_{n+1} = x_n - f(x_n) / f'(x_n)$



Newton's Method

- Fixed point of a function f: $\mathbb{R}\rightarrow\mathbb{R}$ is x such that f(x) = x
  - fixedPoint: $(\mathbb{R}\rightarrow\mathbb{R})\rightarrow\mathbb{R}$ is a function that returns the fixed point of f
  - Apply x_n to f until $|x_{n+1} - x_n| < \varepsilon$, where $x_{n+1} = f(x_n)$
- Given a function f: $\mathbb{R}\rightarrow\mathbb{R}$, next: $(\mathbb{R}\rightarrow\mathbb{R}) \rightarrow (\mathbb{R}\rightarrow\mathbb{R})$ is a function such that
  - Let g = next(f), then g(x) = x - f(x) / f'(x)
- Given a function f: $\mathbb{R}\rightarrow\mathbb{R}$, Newton's method finds a fixed point of next(f)
  - fixedPoint(next(f))

```java
public class Lambda {
    //EPS is a small number
    public static final double EPS = 1e-10;

    //Interfaces

    //function
    public interface Fun<T, R> {
        //Single Abstract Method
        public R apply(T a);
    }

    //recursive function
    public interface Rec<T, R> {
        //Single Abstract Method
        public R fun(Rec<T, R> self, T a);
        public default R apply(T a) {
            return fun(this, a);
        }
    }
```

```java
//recursive function with 2 parameters
public interface Rec2<T1, T2, R> {
    //Single Abstract Method
    public R fun(Rec2<T1, T2, R> self, T1 a, T2 b);
    public default R apply(T1 a, T2 b) {
        return fun(this, a, b);
    }
}

//Factorial
public static Rec<Integer, Integer> fact =
    (self, a) -> a <= 1 ? a
               : a * self.apply(a - 1)
               ;

//GCD
public static Rec2<Integer, Integer, Integer> gcd =
    (self, a, b) -> a > b ? self.apply(a - b, b)
                  : b > a ? self.apply(b - a, a)
                  : a
                  ;
```

…

```java
//Newton's method
public interface Newton {
    //Single Abstract Method
    public double fun(double x);

    //Solve: finds x such that fun(x) = 0
    public default double solve(double x0) {
        //f: the function to solve
        //g: next guess of Newton's method
        //   e.g. g(x0) -> x1, g(x1) -> x2, ...
        Fun<Double, Double> fx = x -> fun(x);
        Fun<Double, Double> gx = next.apply(fx);
        return fixedPoint.apply(gx, x0);
    }
}

//Derivative
public static Func<Func<Double, Double>, Func<Double, Double>>
derivative = f -> x -> (f.apply(x + EPS) - f.apply(x)) / EPS;
```

```
//next: it returns the next guess of Netwon's method
//next(f) = g, g(x) = x - f(x) / f'(x)
//i.e., let g = next(f), g(x0) = x1, g(x1) = x2, g(x2) = x3,...
public static Fun<Fun<Double, Double>, Fun<Double, Double>>
next = f -> /*TODO: implement this function*/


//fixedPoint: returns the fixed point of f
//fixed point of f is x such that |f(x) - x| < EPS
public static Rec2<Fun<Double, Double>, Double, Double>
fixedPoint = (self, f, x) -> /*TODO: implement this function*/


//fixedPoint2: curried version of fixedPoint
public static Rec<Fun<Double, Double>, Fun<Double, Double>>
fixedPoint2 = (self, f) -> x -> /*TODO: implement this function*/


//Square root function
public static Func<Double, Double>
sqrt = x -> ((Newton) y -> y * y - x).solve(1.0);
```

```java
    public static void main(String[] args) {
        //Recursion
        System.out.println("fact(5) :      " + fact.apply(5));
        System.out.println("gcd(12, 30) : " + gcd.apply(12, 30));

        //Bisection
        Bisection b = x -> x*x + 4*x - 8;
        System.out.println("bisection: " + b.solve(-10,  10));

        //Newton's method
        Newton n = x -> x*x + 4*x - 8;
        System.out.println("netwon:     " + n.solve(-10));
        System.out.println("sqrt(2):    " + sqrt.apply(2.0));

        //Fixed point
        Fun<Double, Double> cx = x -> Math.cos(x);
        System.out.println("fixedPoint:  " + fixedPoint.apply(cx, 1.0));
        System.out.println("fixedPoint2: " +
                              fixedPoint2.apply(cx).apply(1.0));
    }
}
```

```
Expected output:

fact(5) :     120
gcd(12, 30) : 6
bisection: -5.464101615107211
netwon:    -5.464101615137754
sqrt(2):   1.4142135623730951
fixedPoint:  0.7390851332451103
fixedPoint2: 0.7390851332451103
```