11 Recursion

11-2 More Recursion

The Recursive Method power

```
public class XToThePowerN {
   public static void main(String[] args) {
      for (int n = 0; n < 4; n++) {
         System.out.println("3 to the power " + n + " is " + power(3,n));
      }
   }
   public static int power(int x, int n) {
      if (n < 0) {
         System.out.println("Illegal argument to power");
         System.exit(0);
      }
      if (n > 0) // x^n = x^(n-1) * x (recursive call)
         return power(x, n-1) * x;
      else // n == 0, (stopping case)
         return 1;
   }
}
```

```
3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27
```

Evaluating the Recursive Method Call power(2,3)

1) power(2,3) = power(2,2) * 2
2) power(2,2) = power(2,1) * 2
3) power(2,1) = power(2,0) * 2
4) power(2,0) = 1,
5) power(2,1) = power(2,0) * 2 = 1 * 2 = 2,   4)를 power(2,0)에 대입
6) power(2,2) = power(2,1) * 2 = 2 * 2 = 4,   5)을 power(2,1)에 대입
7) power(2,3) = power(2,2) * 2 = 4 * 2 = 8,   6)을 power(2,2)에 대입

Recursive Design Techniques (Checking Steps)

1. Confirm there is no infinite recursion
2. Confirm each stopping case performs the correct action for that case
3. Confirm if all recursive calls perform their actions correctly, then the entire case performs correctly

Binary Search

- Searching an array to find a given value
- Condition: the array should be a sorted array:

  a[0] ≤ a[1] ≤ a[2] ≤ . . . ≤ a[finalIndex]
- If the value is found, its index is returned
- If the value is not found, -1 is returned
- Implemented using recursion
- **Note**: Each execution of the recursive method reduces the search space by about a half
  - ”Divide and Conquer” technique

Execution of the Method search

search(first=0, last=9, key=63)
mid=(0+9)/2=4
a[4]=57 < 63

a[0] = 15
a[1] = 20
a[2] = 35
a[3] = 41
a[4] = 57
a[5] = 63
a[6] = 75
a[7] = 80
a[8] = 85
a[9] = 90

search(first=mid+1=5, last=9, key=63)
mid=(5+9)/2=7
a[7]=80 > 63

a[0] = 15
a[1] = 20
a[2] = 35
a[3] = 41
a[4] = 57
a[5] = 63
a[6] = 75
a[7] = 80
a[8] = 85
a[9] = 90

search(first=5, last=mid-1=6, key=63)
mid=(5+6)/2=5
a[5]==63

a[0] = 15
a[1] = 20
a[2] = 35
a[3] = 41
a[4] = 57
a[5] = 63
a[6] = 75
a[7] = 80
a[8] = 85
a[9] = 90

No Existence Case

search(0, 9, 37)	search(0, 3, 37)	search(3, 3, 37)	search(3, 3, 37)	search(3, 2, 37)
mid=(0+9)/2=4	mid=(0+3)/2=1	mid=(3+3)/2=3	mid=(3+3)/2=3	first(3) > last(2)
a[4]=57 > 37	a[1]=20 < 37	a[3]=41 > 37	a[3]=41 > 37	not exist, quit
a[0] = 15	a[0] = 15	a[0] = 15	a[0] = 15	a[0] = 15
a[1] = 20	a[1] = 20	a[1] = 20	a[1] = 20	a[1] = 20
a[2] = 35	a[2] = 35	a[2] = 35	a[2] = 35	a[2] = 35
a[3] = 41	a[3] = 41	a[3] = 41	a[3] = 41	a[3] = 41
a[4] = 57	a[4] = 57	a[4] = 57	a[4] = 57	a[4] = 57
a[5] = 63	a[5] = 63	a[5] = 63	a[5] = 63	a[5] = 63
a[6] = 75	a[6] = 75	a[6] = 75	a[6] = 75	a[6] = 75
a[7] = 80	a[7] = 80	a[7] = 80	a[7] = 80	a[7] = 80
a[8] = 85	a[8] = 85	a[8] = 85	a[8] = 85	a[8] = 85
a[9] = 90	a[9] = 90	a[9] = 90	a[9] = 90	a[9] = 90

Recursive Method for Binary Search

```
public static int search(int[] a, int first, int last, int key) {
    int result = 0;
    if (first > last) result = -1;  // stopping case
    else { // recursive call
        int mid = (first + last)/2;
        if (key == a[mid]) result = mid; // stopping case
        else if (key < a[mid])
            result = search(a, first, mid-1, key);
        else if (key > a[mid])
            result = search(a, mid+1, last, key);
    }
    return result;
}
```

Checking the `search` Method (1/3)

1. There is no infinite recursion
   - On each recursive call
     - The value of **`first`** is increased
     - The value of **`last`** is decreased
   - So, eventually the method will be called with **`first`** larger than **`last`**

```
public static int search(int[] a, int first, int last, int key) {
   int result = 0;
   if (first > last) result = -1;  // stopping case
   else {
      int mid = (first + last)/2;
      if (key == a[mid]) result = mid; // stopping case
      else if (key < a[mid])
         result = search(a, first, mid-1, key); // recursive call
      else if (key > a[mid])
         result = search(a, mid+1, last, key);  // recursive call

   }
   return result;
}
```

Checking the `search` Method (2/3)

2. Each stopping case performs the correct action for that case
   - If `first > last,` there are no array elements between `a[first]` and `a[last]`, so `key` is not in this segment of the array, and `result` is correctly set to `-1`
   - If `key == a[mid]`, `result` is correctly set to `mid`

```
public static int search(int[] a, int first, int last, int key) {
    int result = 0;
    if (first > last) result = -1;  // stopping case
    else {
        int mid = (first + last)/2;
        if (key == a[mid]) result = mid; // stopping case
        else if (key < a[mid])
            result = search(a, first, mid-1, key);
        else if (key > a[mid])
            result = search(a, mid+1, last, key);
    }
    return result;
}
```

Checking the `search` Method (3/3)

3. Check all recursive calls perform their actions correctly
   - If key < a[mid], then key must be one of the elements a[first] through a[mid-1], or it is not in the array, so we should search a from first to mid – 1.
   - If key > a[mid], then key must be one of the elements a[mid-1] through a[last], or it is not in the array, so we should search a from mid + 1 to last.

```
public static int search(int[] a, int first, int last, int key) {
   int result = 0;
   if (first > last) result = -1;  // stopping case
   else {
      int mid = (first + last)/2;
      if (key == a[mid]) result = mid; // stopping case
      else if (key < a[mid])
         result = search(a, first, mid-1, key);
      else if (key > a[mid])
         result = search(a, mid+1, last, key);
   }
   return result;
}
```

Efficiency of Binary Search

- Array size = n
- Serial search algorithm
  - time complexity: $O(n)$ ... we should see all n elements in the worst case
- Binary search algorithm
  - time complexity: $O(\log n)$ ... we should see $\log(n)$ elements in the worst case