RECURSION

Prof. Chris Jermaine
cmj4@cs.rice.edu

What Is Recursion?

- Using a call to a method (directly or indirectly) to implement itself

```
T findLargest (ArrayList <T> inMe) {
  if (inMe.size () <= 0)
    return null;
  else if (inMe.size () == 1)
    return inMe.get (0);
  else {
    T temp = inMe.remove (inMe.size () - 1);
    T largestInRest = findLargest (inMe);
    inMe.add (temp);
    if (temp.compareTo (largestInRest) > 0)
      return temp;
    else return
      largestInRest;
  }}
```

What Does Every Recursive Routine Need?

- A base case
  - — A problem simple enough that you can solve it directly
  - — Without a recursive call

```
T findLargest (ArrayList <T> inMe) {
  if (inMe.size () <= 0)
    return null;
  else if (inMe.size () == 1)
    return inMe.get (0);
  else {
    T temp = inMe.remove (inMe.size () - 1);
    T largestInRest = findLargest (inMe);
    inMe.add (temp);
    if (temp.compareTo (largestInRest) > 0)
      return temp;
    else return
      largestInRest;
  }}
```

What Does Every Recursive Routine Need?

- The recurrence
  - — A way to make the problem smaller
  - — Such that you can use your *own method* to solve the smaller problem

```
T findLargest (ArrayList <T> inMe) {
  if (inMe.size () <= 0)
    return null;
  else if (inMe.size () == 1)
    return inMe.get (0);
  else {
    T temp = inMe.remove (inMe.size () - 1);
    T largestInRest = findLargest (inMe);
    inMe.add (temp);
    if (temp.compareTo (largestInRest) > 0)
      return temp;
    else return
      largestInRest;
  }}
```

The Most Important Take-Home Point

- Most lecturers, upon introducing recursion, draw a call stack
  - — I will do this in a second...
  - — Based on experience, it is better to get it out of the way!
- But that is **absolutely the wrong way to think about recursion**
  - — Instead, always think abstractly
  - — Just assume that someone else has coded up a correct version of your method
  - — But it only works on smaller sized problems
  - — Then use that version to solve your problem (the recurrence)
  - — Will always work (mathematical induction!)
  - — Bad idea to ever reason through what happens on the call stack!

Last Example Was Silly

- No one would ever write that methods recursively
- But many methods are much easier to solve using recursion!
- Example: subset sum
  - — Can I find a subset of the numbers in a set that add to a target *t*?
  - — Is NP-hard
  - — But $O(2^n)$ algorithm is easy to code... if you use recursion!

Subset Sum

```
boolean subsetSum (ArrayList <Integer> nums, int target) {
```

Base case?

What is an instance for which we know the answer right away?

Subset Sum

```
boolean subsetSum (ArrayList <Integer> nums, int target) {

  if (nums.size () == 0 && target == 0)
    return true;
  else if (nums.size () == 0 && target != 0)
    return false;
```

Subset Sum

```
boolean subsetSum (ArrayList <Integer> nums, int target) {

  if (nums.size () == 0 && target == 0)
    return true;
  else if (nums.size () == 0 && target != 0)
    return false;
```

Recurrence?

How can we use a correct subsetSum to solve smaller problem?

Subset Sum

```
boolean subsetSum (ArrayList <Integer> nums, int target) {

  if (nums.size () == 0 && target == 0)
    return true;
  else if (nums.size () == 0 && target != 0)
    return false;
  else {
    int temp = nums.remove (nums.size () - 1);
    boolean res = subsetSum (nums, target) ||
                  subsetSum (nums, target - temp);
    nums.add (temp);
    return res;
}
```

Find *k*th Largest

- Want to find *k*th largest item in a list, in O(*n*) time, any *k*
- Easy to do using recursion!
- Basic idea in recurrence
  - — Choose a random item from the list; call this item *l*
  - — Partition all items around *l*
  - — Then call yourself on one of the two sublists

Find *k*th Largest

```
int findKthLargest (ArrayList <Integer> nums, int k) {
```

Base case?

What is an instance for which we know the answer right away?

Find *k*th Largest

```
int findKthLargest (ArrayList <Integer> nums, int k) {
  if (k == 1 && nums.size () == 1)
    return nums.get (0);
  else if (k == 1 && nums.size () == 0)
    throw new RuntimeException (“you fool”);
```

Find *k*th Largest

```
int findKthLargest (ArrayList <Integer> nums, int k) {
  if (k == 1 && nums.size () == 1)
    return nums.get (0);
  else if (k == 1 && nums.size () == 0)
    throw new RuntimeException (“you fool”);
```

Recurrence?

How can we use a correct findKth to solve smaller problem?

Find *k*th Largest

```
int findKthLargest (ArrayList <Integer> nums, int k) {
  if (k == 1 && nums.size () == 1)
    return nums.get (0);
  else if (k == 1 && nums.size () == 0)
    throw new RuntimeException (“you fool”);
  else {
    int l = nums.get (new Random (12).nextInt (nums.size () - 1));
    ArrayList <Integer> up = new ArrayList <Integer> ();
    ArrayList <Integer> down = new ArrayList <Integer> ();
    for (Integer i : nums)
      if (i > l)
        up.add (i);
      else
        down.add (i);
```

Now, can we solve the problem in four more lines?

Find *k*th Largest

```
int findKthLargest (ArrayList <Integer> nums, int k) {
  if (k == 1 && nums.size () == 1)
    return nums.get (0);
  else if (k == 1 && nums.size () == 0)
    throw new RuntimeException (“you fool”);
  else {
    int l = nums.get (new Random (12).nextInt (nums.size () - 1));
    ArrayList <Integer> up = new ArrayList <Integer> ();
    ArrayList <Integer> down = new ArrayList <Integer> ();
    for (Integer i : nums)
      if (i > l)
        up.add (i);
      else
        down.add (i);
   if (up.size () >= k)
     return findKthLargest (up, k);
   else
     return findKthLargest (down, k - up.size ())};
```

Finally, a Note on Modifying As You Recurse

- Note that SubsetSum modified the list as it went
  - — But it repaired the list after
- Many would have a serious problem with this
  - — Me, not so much
  - — Though I accept it can cause problems in multi-threaded programs
- If this bothers you, two options
  - — Make a copy of the structure before you begin, call second method
  - — Use auxiliary indices, call second method
  - — See, for example, the next slide

Subset Sum

```
boolean subsetSum (ArrayList <Integer> nums,
  int lastConsidered, int target) {

  if (lastConsidered < 0 && target == 0)
    return true;
  else if (lastConsidered < 0 && target != 0)
    return false;
  else {
    int temp = nums.get (lastConsidered);
    boolean res = subsetSum (nums, lastConsidered - 1,
                             target) ||
                  subsetSum (nums, lastConsidered - 1,
                             target - temp);
    return res;
  }
}
```

Questions?