Recursion (Note, Efficiency in previous slide deck)

Chapters 9, 14, 5.22
Recursion

Something defined in terms of itself
Déjà vu?

- We assume you have seen **recursion** before
  - A function may call itself
  - Good match for **divide and conquer** problem-solving

- But we’ll be extending it to OOP and Data Structures
  - **Client** and **implementer** perspectives
  - **Recursive data structures**
  - Recursive **methods** on objects
  - Tracing recursion with a **stack**
  - Efficiency/complexity of recursive solutions
Warmup: Sum numbers from 1 to $N$

/** Return the sum of the integers from 1 to `n` (inclusive). Requires `n` is positive. */

```c
static int sumRange(int n) {
    if (n == 1) { return 1; }
    return sumRange(n - 1) + n;
}
```

- If you could solve smaller versions of the *same problem*, could you compute the answer?
  - $\sum_{k=1}^{n} k = 1 + 2 + \cdots + n$
  - $= 1 + (2 + \cdots + n)$ ?
  - $= (1 + 2 + \cdots + (n - 1)) + n$
  - $= \sum_{k=1}^{n-1} k + n$

- What’s the problem?
Rules of recursive functions

1. **Must have a base case**
   - For at least one special input value, can compute answer without help
   - Without a base case, will recurse infinitely

2. **In other cases, assume you know the answer to any “smaller” version of the same problem**
   - Call yourself to get those answers
     - Adopt client role
     - Only look at spec; don’t think about impl (it’s not even written yet)
   - Compute your answer using that info

3. **Don’t think too hard**
   - **Abstraction barrier** means you don’t have to understand how the smaller problems get solved as long as you obey the spec
Recursive data structures
Recursion

Something defined in terms of itself
Linked list node

```java
class Node<T> {
    T data;
    Node<T> next;
}
```

- A `Node` is defined as having a field of type `Node` - recursive
- How can a Node contain a Node (which contains a Node…)?
  - Reference variables contain a pointer (not the object itself)
  - Reference variables may be null: base case
List suffixes

• A List is a *head value* + a *remainder list*

• Null represents an empty list

• The remainder list is smaller than the full list – recursively delegate work to it!
Example: positionOf

/** Return the position (1-based) of the first occurrence of `target` in the list starting at `head`. Throws NoSuchElementException if `target` is not in the list. */
static int <T> positionOf(Node<T> head, T target);

Assume head.data is not target and head.next is not null. What is the answer?
A. The position in head’s remainder list
B. The position in head’s remainder list + 1
C. The position in head’s remainder list – 1
D. Cannot determine answer by delegating to head’s remainder list
Coding demo
/** Return the position (1-based) of the first occurrence of `target` in the list starting at `head`. Throws NoSuchElementException if `target` is not in the list. */

static <T> int positionOf(Node<T> head, T target) {
    if (head.data.equals(target)) { return 1; }
    if (head.next == null) {
        throw new NoSuchElementException();
    }
    return positionOf(head.next, target) + 1;
}
Recursive methods (OOP)

• A method’s inputs include the object it was invoked on as well as its arguments
• Can answer “smaller” data structure problem by invoking same method on “later” node
• Exercise: make position0f() a method of Node instead of a static function
Helper methods and encapsulation

• Node isn’t public!
• Recursive methods using Nodes expose implementation details!

• That’s okay – don’t make them public methods
• An ADT can implement a public operation in terms of a private recursive method
  • The recursive method becomes a helper to the operation
  • This is a common pattern – the public method can “translate” the operation into a recursive problem (and handle corner cases)
Tracing recursion

The call stack
To trace or not to trace?

• Recursive algorithms are best *developed* by acting as the client to their specification (*not* by tracing)
  • Respect the abstraction barrier when analyzing correctness

• But tracing helps when analyzing efficiency
• And tracing helps explain error messages when things go wrong
The call stack

Look familiar?

Exception in thread "main" java.lang.AssertionError
  at cs2110.LinkedSeq.assertInv(LinkedSeq.java:57)
  at cs2110.LinkedSeq.append(LinkedSeq.java:189)
  at cs2110.CsvJoin.csvToList(CsvJoin.java:26)
  at cs2110.CsvJoin.main(CsvJoin.java:121)

• Every function call in Java is pushed onto a stack
  • The bottom of the stack is main()

• When a function returns (or throws), it is popped off the stack
  • Execution resumes for the function that was under it
What are the elements of this stack?

• Activation record
  • aka “Call frame”
  • aka “Stack frame”

• A fixed-size chunk of memory containing:
  • A function’s parameters
  • A function’s local variables
  • The instruction the function is currently executing
Example: `sumRange()`

```c
int sumRange(int n) {
    int ans;
    if (n == 1) {
        ans = 1;
    } else {
        int tmp = sumRange(n-1);
        ans = tmp + n;
    }
    return ans;
}
```

<table>
<thead>
<tr>
<th>Function</th>
<th>n</th>
<th>tmp</th>
<th>ans</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sumRange()</code></td>
<td>1</td>
<td>N/A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><code>sumRange()</code></td>
<td>2</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td><code>sumRange()</code></td>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td><code>main()</code></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```markdown

main():
args: []
PC: 1
```
Example: sumRange()

```c
int sumRange(int n) {
    int ans;
    if (n == 1) {
        ans = 1;
    } else {
        int tmp = sumRange(n-1);
        ans = tmp + n;
    }
    return ans;
}
```

```
main():
  args: []
  PC: 1
sumRange():
  n: 3     tmp: N/A     ans: 1
  PC: 3
sumRange():
  n: 2     tmp: 1      ans: 3
  PC: 4
sumRange():
  n: 1     tmp: N/A     ans: 1
  PC: 2
```
Time and space complexity

• Last lecture: **time complexity**
  • How do # of operations scale with problem size?

• If recursive function body contains no loops, # of operations is proportional to total # of recursive function calls

• New idea: **space complexity**
  • How does memory usage scale with problem size?
    • (don’t count size of input objects)

• Recursive functions require memory proportional to maximum depth of call stack
Poll: Time and space complexity of sumRange

```java
static int sumRange(int n) {
    if (n == 1) { return 1; }
    return sumRange(n - 1) + n;
}
```

A. Time: O(1), Space: O(1)
B. Time: O(1), Space: O(n)
C. Time: O(n), Space: O(1)
D. Time: O(n), Space: O(n)
Binary search
General idea

• Array is sorted
• Divide array in half; discard half that cannot contain target
• Repeat until nothing left

• Looks recursive: delegate to binary search on an array half as long
• BUT spec is not precise enough!
  • What if target is not in array?
  • What if target occurs multiple times in array?
More precisely, ...

• a is a sorted array
• Return k such that:
  • a[i] < target if i < k and
  • a[i] >= target if i >= k

Assuming target is in a, this returns the index of the:

A. First occurrence
B. Last occurrence
C. An arbitrary occurrence
How do we pass a subarray?

• Some languages (Python, MATLAB) support “slicing”
  • Not Java
• Don’t want to copy data
  • Copying requires $O(a\.length)$ time and $O(a\.length)$ space

• Solution: specify an “array view” with a “begin” and “end” index
  • Convention: includes “begin”, does not include “end”
  • Notation: $a[begin..end)$
  • “All of a”: begin=0, end=a.length
  • begin==end represents empty array between begin-1 and begin

• Requirement: algorithm must never look before “begin” or at or after “end”
Array views

```
| 0 | 1 | ... | begin | begin+1 | ... | end-1 | end | ... |
```

\[ a[\text{begin}..\text{end}) \]
The spec

/** Return the index `k` in `[begin..end]` such that
 * `a[i] < target` for `i < k` and `a[i] >= target`
 * for `i > k`.
 * Requires `0 <= begin <= end <= a.length`,
 * `a[begin..end)` is sorted in ascending order. */

static int binarySearch(int[] a, int begin, int end, int target);
The base case

// Array view is empty;
// only one legal index to return
if (begin == end) {
    return begin;
}

Choosing a midpoint

Constraint: midpoint must be in \([\text{begin}..\text{end})\)
- What if size is 1?
- What if size is even?

Constraint: both halves must be smaller than original array
- What if size is 1?
The recursive case

```c
int m = begin + (end - begin)/2;
// Guarantee: begin <= m < end
if (a[m] < target) {
    // if a[m]<target, then m<ans; search right half
    return binarySearch(a, m + 1, end, target);
} else {
    // ans<=m; search left half
    return binarySearch(a, begin, m, target);
}
```
Complexity analysis

• Each call does constant (O(1)) work on its own
  • Assuming array indexing requires O(1) work (not true for linked list)
• Each call (except base case) makes 1 recursive call
  • Array view is at most half as big
  • All calls will accumulate on call stack (O(1) space each)
• How many calls?
  • How many times can you divide an array in half until it is empty?
  • Alternatively, starting with 1 element, how many times can you double an array until it reaches a desired size?
Logarithms

\[ \log_b x = y \quad \leftrightarrow \quad b^y = x \]

\[ b \cdot b \cdot b \cdot b \cdots b = x \]

\[ y \]
Logarithms and Efficiency

Change of base:

\[
\log_b n = \frac{1}{\log_d b} \cdot \log_d n
\]

Example:

\[
\log_2 n = \frac{1}{\log_{10} 2} \cdot \log_{10} n
\]

This is a constant factor: something we ignore!
Recursion vs. iteration

• Once you embrace the client perspective (“leap of faith”), recursion makes it easier to write correct algorithms

• Recursion is more powerful than simple loops, more convenient than loops + your own stack
  • Coming up: trees, sorting

• Recursion uses more space (in principle) than simple loops
  • Tail recursion: if the last instruction in the recursive case is a recursive call, can rewrite as a simple loop

• Tracing recursive calls might be trickier than tracing loops
  • Remember: activation records do not share variables