Solving Problems Recursively

- Recursion is an indispensable tool in a programmer’s toolkit
  - Allows many complex problems to be solved simply
  - Elegance and understanding in code often leads to better programs: easier to modify, extend, verify (and sometimes more efficient!!)
  - Sometimes recursion isn’t appropriate, when it’s bad it can be very bad—every tool requires knowledge and experience in how to use it

- The basic idea is to get help solving a problem from coworkers (clones) who work and act like you do
  - Ask clone to solve a simpler but similar problem
  - Use clone’s result to put together your answer
- Need both concepts: call on the clone and use the result

Print words entered, but backwards

- Can use an ArrayList, store all the words and print in reverse order
  - Probably the best approach, recursion works too
    ```java
    public void printReversed(Scanner s){
      if (s.hasNext()){ // reading succeeded?
        String word = s.next(); // store word
        printReversed(s); // print rest
        System.out.println(word); // print the word
      }
    }
    ```
  - The function printReversed reads a word, prints the word only after the clones finish printing in reverse order
  - Each clone has own version of the code, own word variable
  - Who keeps track of the clones?
  - How many words are created when reading N words?
    - What about when ArrayList<String> used?

Exponentiation

- Computing \( x^n \) means multiplying n numbers (or does it?)
  - What’s the easiest value of n to compute \( x^n \)?
  - If you want to multiply only once, what can you ask a clone?
    ```java
    public static double power(double x, int n){
      if (n == 0){
        return 1.0;
      }
      double semi = power(x, n/2);
      if (n % 2 == 0) {
        return semi * semi;
      }
      return x * semi * semi;
    }
    ```
  - What about an iterative version?

Faster exponentiation

- How many recursive calls are made to compute \( 2^{1024} \)?
  - How many multiplies on each call? Is this better?
    ```java
    public static double power(double x, int n){
      if (n == 0) {
        return 1.0;
      }
      double semi = power(x, n/2);
      if (n % 2 == 0) {
        return semi * semi;
      }
      return x * semi * semi;
    }
    ```
  - What about an iterative version of this function?
Back to Recursion

- Recursive functions have two key attributes
  - There is a base case, sometimes called the exit case, which does not make a recursive call
    - See print reversed, exponentiation
  - All other cases make a recursive call, with some parameter or other measure that decreases or moves towards the base case
    - Ensure that sequence of calls eventually reaches the base case
    - “Measure” can be tricky, but usually it’s straightforward

- Example: sequential search in an array
  - If first element is search key, done and return
  - Otherwise look in the “rest of the array”
  - How can we recurse on “rest of array”?

Thinking recursively

- Problem: find the largest element in an array
  - Iteratively: loop, remember largest seen so far
  - Recursive: find largest in [1..n), then compare to 0th element

```java
public static double max(double[] a){
    double maxSoFar = a[0];
    for(int k=1; k < a.length; k++) {
        maxSoFar = Math.max(maxSoFar,a[k]);
    }
    return maxSoFar;
}
```

- In a recursive version what is base case, what is measure of problem size that decreases (towards base case)?

Why we study recurrences/complexity?

- Tools to analyze algorithms
- Machine-independent measuring methods
- Familiarity with good data structures/algorithms

- What is CS person: programmer, scientist, engineer?
  *scientists build to learn, engineers learn to build*

- Mathematics is a notation that helps in thinking, discussion, programming

Recurrences

- Summing Numbers
  ```java
  int sum(int n)
  {  
    if (0 == n) return 0;
    else return n + sum(n-1);
  }
  ```

- What is complexity? justification?
- T(n) = time to compute sum for n

  ```
  T(n) = T(n-1) + 1  
  T(0) = 1
  ```

- instead of 1, use O(1) for constant time
- independent of n, the measure of problem size
Solving recurrence relations

- **plug, simplify, reduce, guess, verify?**

\[
T(n) = T(n-1) + 1 \\
T(0) = 1 \\
T(n-1) = T(n-1-1) + 1 \\
T(n) = [T(n-2) + 1] + 1 = T(n-2)+2 \\
T(n-2) = T(n-2-1) + 1 \\
T(n) = [(T(n-3) + 1) + 1] + 1 = T(n-3)+3 \\
T(n) = T(n-k) + k \quad \text{find the pattern!}
\]

Now, let \( k = n \), then \( T(n) = T(0)+n = 1+n \)

- **get to base case, solve the recurrence: \( O(n) \)**

Recognizing Recurrences

- **Solve once, re-use in new contexts**
  - \( T \) must be explicitly identified
  - \( n \) must be some measure of size of input/parameter
    - \( T(n) \) is the time for quicksort to run on an \( n \)-element vector

\[
T(n) = T(n/2) + O(1) \quad \text{binary search} \quad O(\log n) \\
T(n) = T(n-1) + O(1) \quad \text{sequential search} \quad O(n) \\
T(n) = 2T(n/2) + O(1) \quad \text{tree traversal} \quad O(n) \\
T(n) = 2T(n/2) + O(n) \quad \text{quicksort} \quad O(n \log n) \\
T(n) = T(n-1) + O(n) \quad \text{selection sort} \quad O(n^2)
\]

- **Remember the algorithm, re-derive complexity**

The Power of Recursion: Brute force

- Consider the TypingJob APT problem: What is minimum number of minutes needed to type \( n \) term papers given page counts and three typists typing one page/minute? (assign papers to typists to minimize minutes to completion)
  - Example: \{3, 3, 3, 5, 9, 10, 10\} as page counts

- How can we solve this in general? Suppose we're told that there are no more than 10 papers on a given day.
  - How does this constraint help us?
  - What is complexity of using brute-force?

Recasting the problem

- Instead of writing this function, write another and call it

\[
\begin{align*}
\text{// @return min minutes to type papers in pages} \\
\text{int bestTime(int[] pages)} \\
\{ \\
\text{ \quad return best(pages,0,0,0,0); } \\
\}
\end{align*}
\]

- What cases do we consider in function below?

\[
\begin{align*}
\text{int best(int[] pages, int index,} \\
\text{ \quad int t1, int t2, int t3) } \\
\text{\quad // returns min minutes to type papers in pages} \\
\text{\quad // starting with index-th paper and given} \\
\text{\quad // minutes assigned to typists, t1, t2, t3} \\
\text{\quad \{ } \\
\}
\end{align*}
\]
Backtracking, Search, Heuristics

- Many problems require an approach similar to solving a maze
  - Certain mazes can be solved using the “right-hand” rule
  - Other mazes, e.g., with islands, require another approach
  - If you have “markers”, leave them at intersections, don’t explore the same place twice

- What happens if you try to search the web, using links on pages to explore other links, using those links to ...
  - How many web pages are there?
  - What rules to webcrawlers/webspiders follow?
    - Who enforces the rules?
  - Keep track of where you’ve been don’t go there again
  - Any problems with this approach?

Backtracking with Boggle

- Boggle™ played on 4x4 board
  - Other sizes possible
  - Form words by connecting letters horizontally, vertically, diagonally
  - Cannot re-use letters (normally)

- Two approaches
  - Build words from connections, find partial words in dictionary
  - Look up every word in the dictionary on the board

- Which is better? How is backtracking used?

Classic problem: N queens

- Can queens be placed on a chess board so that no queens attack each other?
  - Easily place two queens
  - What about 8 queens?

- Make the board N x N, this is the N queens problem
  - Place one queen/column
  - # different tries/column?

- Backtracking
  - Use “current” row in a col
  - If ok, try next col
  - If fail, back-up, next row

Backtracking idea with N queens

- Try to place a queen in each column in turn
  - Try first row in column C, if ok, move onto next column
  - If solved, great, otherwise try next row in column C, place queen, move onto the next column
    - Must unplace the placed queen to keep going

- What happens when we start in a column, where to start?
  - If we fail, move back to previous column (which remembers where it is/failed)
  - When starting in a column anew, start at beginning
    - When backing up, try next location, not beginning

- Backtracking in general, record an attempt go forward
  - If going forward fails, undo the record and backup
Basic ideas in backtracking search

- We need to be able to enumerate all possible choices/moves
  - We try these choices in order, committing to a choice
  - If the choice doesn’t pan out we must undo the choice
    - This is the backtracking step, choices must be undoable

- Process is inherently recursive, so we need to know when the search finishes
  - When all columns tried in N queens
  - When all board locations tried in boggle
  - When every possible moved tried in Tic-tac-toe or chess?
    - Is there a difference between these games?

- Summary: enumerate choices, try a choice, undo a choice, this is brute force search: try everything

---

Computer v. Human in Games

- Computers can explore a large search space of moves quickly
  - How many moves possible in chess, for example?

- Computers cannot explore every move (why) so must use heuristics
  - Rules of thumb about position, strategy, board evaluation
  - Try a move, undo it and try another, track the best move

- What do humans do well in these games? What about computers?
  - What about at Duke?

---

Backtracking, minimax, game search

- We’ll use tic-tac-toe to illustrate the idea, but it’s a silly game to show the power of the method
  - What games might be better? Problems?

- Minimax idea: two players, one maximizes score, the other minimizes score, search complete/partial game tree for best possible move
  - In tic-tac-toe we can search until the end-of-the game, but this isn’t possible in general, why not?
  - Use static board evaluation functions instead of searching all the way until the game ends

- Minimax leads to alpha-beta search, then to other rules and heuristics

---

Minimax for tic-tac-toe

- Players alternate, one might be computer, one human (or two computer players)

- Simple rules: win scores +10, loss scores -10, tie is zero
  - X maximizes, O minimizes

- Assume opponent plays smart
  - What happens otherwise?

- As game tree is explored is there redundant search?
  - What can we do about this?
What’s the Difference Here?

- How does find-a-track work? Fast forward?

Contrast LinkedList and ArrayList

- See ISimpleList, SimpleLinkedList, SimpleArrayList
  - Meant to illustrate concepts, not industrial-strength
  - Very similar to industrial-strength, however
- ArrayList --- why is access O(1) or constant time?
  - Storage in memory is contiguous, all elements same size
  - Where is the 1<sup>st</sup> element? 40<sup>th</sup>? 360<sup>th</sup>?
  - Doesn’t matter what’s in the ArrayList, everything is a pointer or a reference (what about null?)

What about LinkedList?

- Why is access of N<sup>th</sup> element linear time?
- Why is adding to front constant-time O(1)?

ArrayLists and linked lists as ADTs

- As an ADT (abstract data type) ArrayLists support
  - Constant-time or O(1) access to the k-th element
  - Amortized linear or O(n) storage/time with add
    - Total storage used in n-element vector is approx. 2n, spread over all accesses/additions (why?)
    - Adding a new value in the middle of an ArrayList is expensive, linear or O(n) because shifting required
- Linked lists as ADT
  - Constant-time or O(1) insertion/deletion anywhere, but…
  - Linear or O(n) time to find where, sequential search

- Good for sparse structures: when data are scarce, allocate exactly as many list elements as needed, no wasted space/copying (e.g., what happens when vector grows?)
Linked list applications

- Remove element from middle of a collection, maintain order, no shifting. Add an element in the middle, no shifting
  - What's the problem with a vector (array)?
  - Emacs visits several files, internally keeps a linked-list of buffers
  - Naively keep characters in a linked list, but in practice too much storage, need more esoteric data structures

- What's \((3x^3 + 2x^3 + x + 5) + (2x^4 + 5x^3 + x^2 + 4x)\)?
  - As a vector \((3, 0, 2, 0, 1, 5)\) and \((0, 2, 5, 1, 4, 0)\)
  - As a list \(((3,5), (2,3), (1,1), (5,0))\) and ________?
  - Most polynomial operations sequentially visit terms, don't need random access, do need “splicing”

- What about \((3x^{100} + 5)\)?

Linked list applications continued

- If programming in C, there are no “growable-arrays”, so typically linked lists used when # elements in a collection varies, isn’t known, can’t be fixed at compile time
  - Could grow array, potentially expensive/wasteful especially if # elements is small.
  - Also need # elements in array, requires extra parameter
  - With linked list, one pointer used to access all the elements in a collection

- Simulation/modeling of DNA gene-splicing
  - Given list of millions of CGTA... for DNA strand, find locations where new DNA/gene can be spliced in
    - Remove target sequence, insert new sequence

Linked lists, CDT and ADT

- As an ADT
  - A list is empty, or contains an element and a list
  - \((\ )\) or \((x, (y, (\ )\))\)

- As a picture

- As a CDT (concrete data type)

```java
public class Node {
    String value;
    Node next;
    Node(String s, Node link){
        value = s;
        next = link;
    }
}
```

// … declarations here

```java
Node list = null;
while (scanner.hasNext()) {
    list = new Node(scanner.next(), list);
}
```

Building linked lists

- Add words to the front of a list (draw a picture)
  - Create new node with next pointing to list, reset start of list

```java
public class Node {
    String value;
    Node next;
    Node(String s, Node link){
        value = s;
        next = link;
    }
}
```

// … declarations here

```java
Node list = null;
while (scanner.hasNext()) {
    list = new Node(scanner.next(), list);
}
```
Dissection of add-to-front

- List initially empty
- First node has first word

```java
list = new Node(word, list);
Node(String s, Node link) {
    info = s;
    next = link;
}
```

- Each new word causes new node to be created
  - New node added to front
- Rhs of operator = completely evaluated before assignment

Standard list processing (iterative)

- Visit all nodes once, e.g., count them or process them

```java
public int size(Node list) {
    int count = 0;
    while (list != null) {
        count++;
        list = list.next;
    }
    return count;
}
```

- What changes in code above if we change what “process” means?
  - Print nodes?
  - Append “s” to all strings in list?

Building linked lists continued

- What about adding a node to the end of the list?
  - Can we search and find the end?
  - If we do this every time, what’s complexity of building an N-node list? Why?

- Alternatively, keep pointers to first and last nodes of list
  - If we add node to end, which pointer changes?
  - What about initially empty list: values of pointers?
    - Will lead to consideration of header node to avoid special cases in writing code

- What about keeping list in order, adding nodes by splicing into list? Issues in writing code? When do we stop searching?

Standard list processing (recursive)

- Visit all nodes once, e.g., count them

```java
public int recsize(Node list) {
    if (list == null) return 0;
    return 1 + recsize(list.next);
}
```

- Base case is almost always empty list: null pointer
  - Must return correct value, perform correct action
  - Recursive calls use this value/state to anchor recursion
  - Sometimes one node list also used, two “base” cases
- Recursive calls make progress towards base case
  - Almost always using list.next as argument
Recursion with pictures

- Counting recursively

```java
int recsize(Node list){
if (list == null)
return 0;
return 1 +
recsize(list.next);
}
```

```
recsize(Node list)
return 1 +
recsize(list.next)
```

```
recsize(Node list)
return 1 +
recsize(list.next)
```

```
recsize(Node list)
return 1 +
recsize(list.next)
```

```java
ptr
System.out.println(recsize(ptr));
```

Recursion and linked lists

- Print nodes in reverse order
  - Print all but first node and...
    - Print first node before or after other printing?

```java
public void print(Node list) {
if (list == null) {
    System.out.println(list.info);
    System.out.println(list.info);
    print(list.next);
}
```

Complexity Practice

- What is complexity of Build? (what does it do?)

```java
public Node build(int n) {
if (null == n) return null;
Node first = new Node(n, build(n-1));
for(int k = 0; k < n-1; k++) {
    first = new Node(n,first);
}
return first;
}
```

- Write an expression for T(n) and for T(0), solve.
  - Let T(n) be time for build to execute with n-node list
  - T(n) = T(n-1) + O(n)

Changing a linked list recursively

- Pass list to method, return altered list, assign to list
  - Idiom for changing value parameters

```java
list = change(list, “apple”);
public Node change(Node list, String key) {
if (list != null) {
    list.next = change(list.next, key);
    if (list.info.equals(key))
        return list.next;
    else
        return list;
}
return null;
```

- What does this code do? How can we reason about it?
  - Empty list, one-node list, two-node list, n-node list
  - Similar to proof by induction