Project 1 Review

- The Xref app: suppose this was a serious app to be implemented by a team of programmers “a crucial tool” for programmers.
- Design phase: Someone says: Hey, we need to pull the ids out of the program text, and that’s a separate job from building the data structure summarizing where each id shows up.
- The team agrees, so two teams are formed, one for tokenization and one for indexing provided ids.

Two teams, two classes: how to communicate between them?

- We now have two teams, one for tokenization in say class Tokenizer and one for indexing given ids, the main code, in class Xref.
- Next design step: Exactly how does Xref call Tokenizer?
- Xref needs to get tokens and their line numbers.
- Could have just one method that returns both, but to return a String and an int, need to create a helper object with both, say class IdLocation with fields String id and int lineNumber.
- Someone says: I know an easier way: have two methods…

Two teams, and an interface between them

- Idea of two methods to communicate ids and line numbers, looks like this:
  - String getNextID()
  - int getLineNumber() (to be called after getNextID)
- The teams agree to this idea, though one person says “They should be tied together in one method to ensure that the ID and line number are in fact related”
- They say “Let’s write an interface to codify this API”

An Interface is a contract

- By agreeing to this API, codified in the Java interface, the two teams have a contract between them:
  - The Tokenizer team implements the API
  - The Xref team implements the code that calls the API
- Note that the interface doesn’t capture everything about the interactions: for example, the constructor syntax is missing, and the treatment of end-of-data.
- But it is an excellent start on the full contract.

The story, continued

- Someone on the Tokenizer team says “I just saw a regex on StackOverflow that does most of our work! Let’s go with it…”
- Another says: “I’ve heard that Java regex processing can crash an app: it builds a huge stack to do its work. Let’s stick with simple parsing…”
- They let the first person build a competing Tokenizer, and then test them against each other using something like TestTokenizer, which takes advantage of the fact that a each Tokenizer ISA JavaTokenizer, so the test code can be written once for either Tokenizer.
The morals of the story

- Interfaces are crucial to software design, especially for larger programs with multiple source files and multiple teams.
- Interfaces let us treat multiple implementations of a certain API as the same Java type, allowing client code to run using either one.
  - Example: LinkedList and ArrayList are both type List, so code using List can switch off List implementations at will: just change the List creation step.

What's next in our Java type coverage?

- Answer: inheritance in general (Using an interface is a special case of inheritance.)
- One class can be a subclass of another, not just "implements another".
  - If B is a subclass of A, then B ISA A (A is a class)
  - If B implements A, then B ISA A (A is an interface)
  - And any class A is a subclass of Object, so A ISA Object
- Either way, we can write client code that treats a family of types as a single type.

Inheritance

- For decades considered the bedrock of object oriented software practice, but lately less championed.
- Where can I read about inheritance?
  - Not in S&W: not even in index! But "inherited methods" are, mostly methods of Object. For example, we can use Object.hashCode() for any object, since its class is a subclass of Object.
  - Weiss, Chap. 4 (also has interfaces)
  - Java tutorial at Oracle, other tutorials on Web

Example: A Student ISA Person

- Recall our Student object, with fields id, name, and gpa.
- GPA is specific to students, but id and name could work for any person, so let's set up a Person class
  ```java
  class Person {
      private int id;
      private String name;
      public Person(String n, int i) { name = n; id = i; }
      // getters and setters for name and id
      public toString() { return name +": " + id; }
  }
  ```
- A Student ISA Person
  ```java
  class Student extends Person {  
      private double gpa;  // additional field
      public Student(String n, int i, double gpa) { super(n, i); this.gpa = gpa; }
      double getGpa { return gpa; }
      String toString() { return getName() +": " + getId() + " + gpa " + gpa; }
  }
  ```

Next time

- We'll examine this example further next time
- Similar to example in Weiss
- Java tutorial has class Bicycle and subclass MountainBike
  - Bicycle has fields cadence, gear, and speed
  - What's cadence?
  - Turns out it's the speed of pedaling (cycles/min, say)
Algorithm Techniques

There are patterns in algorithms worth studying.
We’ll cover:
- Divide and conquer: we already saw examples
- Greedy algorithm: follow what appears to be best at each step, example coming up
- Dynamic programming: save partial results as you go, then reuse them

Problem – making change

- Task – buy a cup of coffee (say it costs 63 cents).
- You are given an unlimited number of coins of all types (neglect 50 cents and 1 dollar).
- Pay exact change.
- What is the combination of coins you’d use?

Greedy algorithms - change making

- Logically, we’d minimize the number of coins.
  - Change-making with the fewest number of US coins—have 1, 5, 10, 25 unit coins to work with.
  - Clearly we want to mainly use large-value coins to minimize the total number. So for 27 cents, clearly we can’t do better than 25 + 2(1).
  - What about 63? Use as many 25s as fit, 63 = 2*(25) + 13, then as many 10s as fit in the remainder: 63 = 2*(25) + 1*(10) + 3, no 5’s fit, so we have 63 = 2*(25) + 1*(10) + 3*(1), 6 coins.

Greedy algorithms

- Not discussed as such in S&W (has index entry). See pg. 287 of Weiss. Whole chapter (Chap. 4) in Kleinberg and Tardos.
- A greedy person grabs everything they can as soon as possible.
- Similarly a greedy algorithm makes decisions that appear to be the best thing to do at each step.
- Example: Change-making greedy algorithm for “change” amount, given many coins of each size:
  Loop until change == 0:
  Find largest-valued coin less than change, use it.
  change = change – coin-value;

Change making: when greedy doesn’t work…

- The greedy method gives the optimal solution for US coinage.
- With different coinage, the greedy algorithm doesn’t always find the optimal solution.
- Example of a coinage with an additional 21 cent piece. Then 63 = 3*(21), but greedy says use 2 25s, 1 10, and 3 1’s, a total of 6 coins.
- The coin values need to be spread out enough to make greedy work. But even some spread-out cases don’t work. Consider having pennies, dimes and quarters, but no nickels.
- Then 30 by greedy uses 1 quarter and 5 pennies, ignoring the best solution of 3 dimes.

(Very bad) recursive solution

coins = {25, 10, 5, 1, 21}
makeChange(amt)
If amt in coins return 1
minCoins = amt
Loop over j from 1 to amt/2
thisCoins = makeChange(j) + makeChange(amt-j)
if thisCoins < minCoins
  minCoins = thisCoins
Lots and lots of redundant calls!
(Very bad) recursive solution

\[ T(n) = T(n-1) + T(n-2) + T(n-3) + \ldots + T(n/2) + \ldots \]

worse than \( T(n) = T(n-1) + T(n-2) \) the famous Fibonacci sequence discussed previously. Fibonacci is exponential, so this certainly is.

Better idea

- We know we have 1, 5, 10, 21, and 25.
- Therefore, the optimal solution must be the minimum of the following:
  
  \[ 1 \text{ (A 1 cent)} + \text{optimal solution for 62} \]
  
  \[ 1 \text{ (A 5 cent)} + \text{optimal solution for 58} \]
  
  \[ 1 \text{ (A 10 cent)} + \text{optimal solution for 53} \]
  
  \[ 1 \text{ (A 21 cent)} + \text{optimal solution for 42} \]
  
  \[ 1 \text{ (A 25 cent)} + \text{optimal solution for 38} \]
- This reduces the number of recursive calls drastically.
- Naïve implementation still makes lots of redundant calls.

Dynamic programming implementation

- Idea — instead of performing the same calculation over and over again, save pre-calculated results to an array.
- The answer to a large change depends only on results of smaller calculations, so we can calculate the optimal answer for all the smaller change values and save it to an array.
- Then go over the array and minimize on:
  
  \[ \text{change}(K) = \min\{\text{change}(K-n)+1\} \]
  
  For all N types of coins of value n
- Runtime - \( O(N^2K) \).

Dynamic programming for change problem

```java
public static void makeChange( int[] coins, int differentCoins, int maxChange, int[] coinsUsed,
  int[] lastCoin )
{
  coinsUsed[0] = 0; lastCoin[0] = 1;
  for( int cents = 1; cents <= maxChange; cents++ ) {
    int minCoins = cents;
    int newCoin = 1;
    for( int j = 0; j < differentCoins; j++ ) {
      if( coins[j] > cents ) continue;
      // Cannot use coin j
      if( coinsUsed[cents-coins[j]] + 1 < minCoins ) {
        minCoins = coinsUsed[cents-coins[j]] + 1;
        newCoin = coins[j];
      }
    }
    coinsUsed[cents] = minCoins;
    lastCoin[cents] = newCoin;
  }
}
```

Binomial coefficients

Another famous example is the sequence of binomial coefficients. Can be generated by Pascal’s triangle:

\[
\begin{array}{ccccccc}
1 & & & & & & \\
1 & 1 & & & & & \\
1 & 2 & 1 & & & & \\
1 & 3 & 3 & 1 & & & \\
1 & 4 & 6 & 4 & 1 & & \\
\end{array}
\]

Each number is the sum of the two closest above it.

Binomial coefficients

- Start a new row with 1’s on the edges.
- The row number is \( N \), and the entries are \( k=0, k=1, \ldots, k=N \) across a row, so for example

\[ C(4,0) = 1, C(4,1) = 4, C(4,2) = 6, C(4,3) = 4, C(4,4) = 1. \]
- These are the coefficients of the binomial expansion

\[ (x + y)^4 = x^4 + 4 x^3 y + 6 x^2 y^2 + 4 x y^3 + y^4 \]
Binomial coefficient and \( n \) choose \( k \)

- Also, \( C(N, k) \) = number of ways to choose a set of \( k \) objects from \( N \)
- Ex. \( C(4, 2) = 6 \) The 2-sets of 4 numbers are the 6 sets:
  \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}
- Recursion: \( C(N, k) = C(N-1, k) + C(N-1, k-1) \) This is just the sum rule of Pascal’s triangle.

\( n \) choose \( k \) - relationships

- Base cases: \( C(N, 0) = 1 \), \( C(N, N) = 1 \)
- To choose \( k \) objects from \( N \), set one object \( x \) aside and find all the ways of choosing \( k \) objects from the remaining \( N-1 \).
  - These are all the sets we want that don’t include \( x \), \( C(N-1, k) \) in number.
  - The sets that do include \( x \) also need \( k-1 \) other objects from the other \( N-1 \), \( C(N-1, k-1) \) in number.
  - So \( C(N, k) = C(N-1, k) + C(N-1, k-1) \)

Binomial coefficient - recursion

If we write a recursive function:

```
combo(N, k):
    if (k == 0) return 1
    if (k == N) return 1
    return combo(N-1, k) + combo(N-1, k-1)
```

Note the double recursion, without halving the “\( N \)” value, so dangerous recursion.

Efficient calculation of binomial coefficients

- If we save and reuse values, it’s much faster. In other words, use Pascal’s triangle to generate all the coefficients.
- One way: set up a table and use it for each \( N \) in turn.

```
C[1][0] = 1
C[1][1] = 1
for n up to N
    for k up to n
        C[n][k] = C[n-1][k] + C[n-1][k-1]
```

- \( O(1) \) to fill each spot in \( N \times N \) array, so \( O(N^2) \)

Map approach to dynamic programming

- Another approach: set up Map from \( (N, k) \) to value.
- Case of classic dynamic programming, saving partial results along the way.
- If \( N \) and \( k \) both ints, long key = \( N + (\text{long})k>>32 \) (Pack two ints in a long)
- Or OO way: create a Pair class, with \( N, k \) fields, getters and setters.
- Either way, have key(\( N, k \)) holding the pair.
Map approach to dynamic programming

```java
combo(N, k):
    val = M.get(key(N,k))
    if (val != null) return val
    if (k == 0 || k == N)
        val = 1
    else val = combo(N-1, k) + combo(N-1, k-1)
    M.put(key(N, k), val)
    return val
```

once this recursion reaches a cell, fills it in, so work bounded by number of cells below (N, k), which is < N^2.

Maximum Contiguous Subsequence Sum

Problem: Given a sequence of integers (A_1, A_2, ..., A_N), possibly negative.
Identify the subsequence (A_i, ..., A_j) that corresponds to the maximum value of
\[ \sum_{k=i}^{j} A_k \]

Reference: Weiss Sec. 7.5.1,

- Naïve approach is cubic (examine all O(N^2) sequences and sum each one).
- Use a divide-and-conquer algorithm.

Divide-and-Conquer Algorithm

Sample input is {4, -3, 5, 2, 1, 2, 6, -2}
3 possible cases:
1. in the first half
2. in the second half
3. begins in the first half and ends in the second half

For case 3: sum = sum 1st + sum 2nd

<table>
<thead>
<tr>
<th>First Half</th>
<th>Second Half</th>
<th>Values</th>
<th>Running sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>-3</td>
<td>5 -2</td>
<td>-1 1 7 5</td>
</tr>
<tr>
<td>4 -3 5 -2</td>
<td>-1 2 6 -2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Running sum from the center (*denotes maximum for each half*).

Case 3 is solved in linear time.
Apply case 3's strategy to solve case 1 and 2
Summary:
Recursively compute the max subsequence sum in the first half
Recursively compute the max subsequence sum in the second half
Compute, via 2 consecutive loops, the max subsequence sum that begins in the first half but ends in the second half
Choose the largest of the 3 sums
See Weiss if interested in performance analysis
Result: O(N log N)