Topic 26
Dynamic Programming

"Thus, I thought ***dynamic programming*** was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities"

- Richard E. Bellman

Origins

- A method for solving complex problems by breaking them into smaller, easier, sub problems
- Term *Dynamic Programming* coined by mathematician Richard Bellman in early 1950s
  - employed by Rand Corporation
  - Rand had many, large military contracts
  - Secretary of Defense, Charles Wilson “against research, especially mathematical research”
  - how could any one oppose "dynamic"?

Dynamic Programming

- Break big problem up into smaller problems ...
- Sound familiar?
- Recursion?

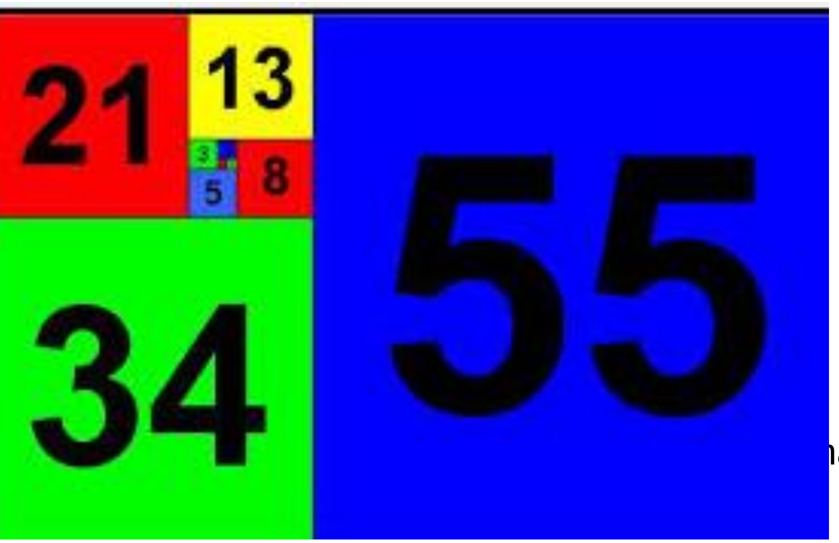
  N! = 1 for N == 0

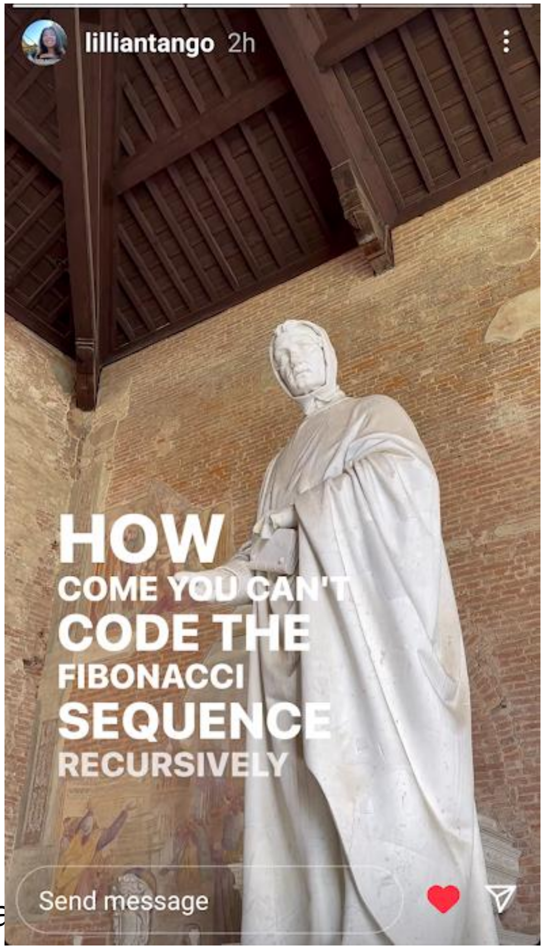
  N! = N * (N - 1)! for N > 0

Fibonacci Numbers

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 114, …
- $F_1 = 1$
- $F_2 = 1$
- $F_N = F_{N-1} + F_{N-2}$

- Recursive Solution?





Failing Spectacularly

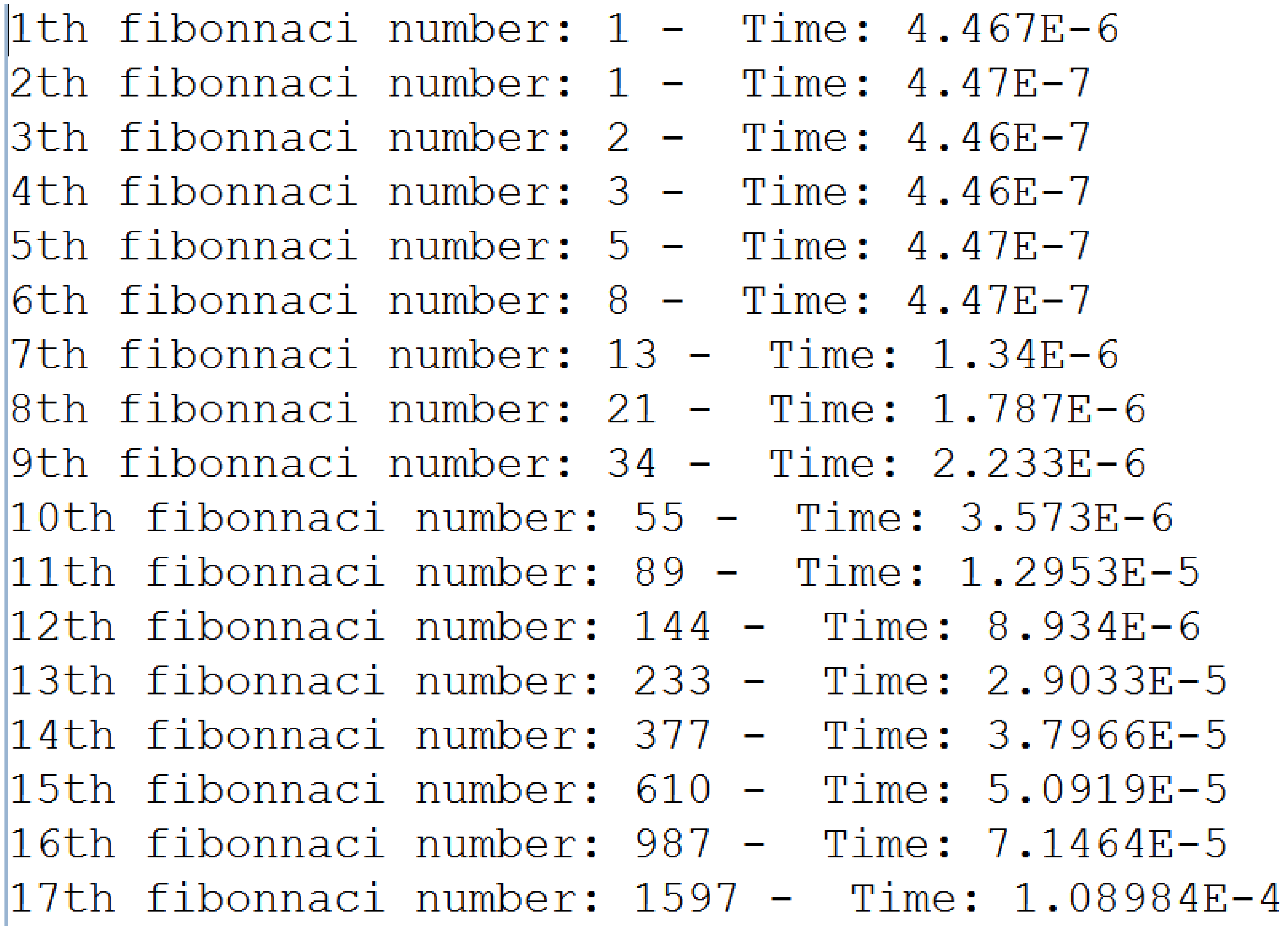
- Naïve recursive method

```
// pre: n > 0
// post: return the nth Fibonacci number
public int fib(int n) {
     if (n <= 2)
          return 1;
     else
          return fib(n – 1) + fib (n – 2);
}
```

- Clicker 1 - Order of this method?

A. O(1)   B. O(log N)   C. O(N)   D. O(N^2)   E. O(2^N)

Failing Spectacularly

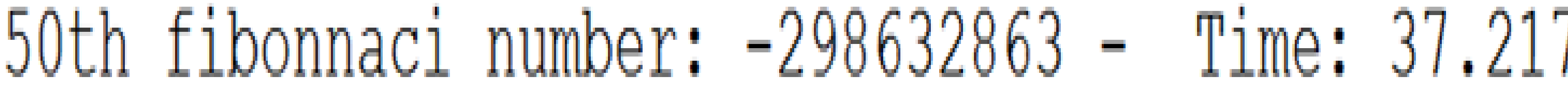


```
1th fibonnaci number: 1 -  Time: 4.467E-6
2th fibonnaci number: 1 -  Time: 4.47E-7
3th fibonnaci number: 2 -  Time: 4.46E-7
4th fibonnaci number: 3 -  Time: 4.46E-7
5th fibonnaci number: 5 -  Time: 4.47E-7
6th fibonnaci number: 8 -  Time: 4.47E-7
7th fibonnaci number: 13 -  Time: 1.34E-6
8th fibonnaci number: 21 -  Time: 1.787E-6
9th fibonnaci number: 34 -  Time: 2.233E-6
10th fibonnaci number: 55 -  Time: 3.573E-6
11th fibonnaci number: 89 -  Time: 1.2953E-5
12th fibonnaci number: 144 -  Time: 8.934E-6
13th fibonnaci number: 233 -  Time: 2.9033E-5
14th fibonnaci number: 377 -  Time: 3.7966E-5
15th fibonnaci number: 610 -  Time: 5.0919E-5
16th fibonnaci number: 987 -  Time: 7.1464E-5
17th fibonnaci number: 1597 -  Time: 1.08984E-4
```

Failing Spectacularly

```
36th fibonnaci number: 14930352 -  Time: 0.045372057
37th fibonnaci number: 24157817 -  Time: 0.071195386
38th fibonnaci number: 39088169 -  Time: 0.116922086
39th fibonnaci number: 63245986 -  Time: 0.186926245
40th fibonnaci number: 102334155 -  Time: 0.308602967
41th fibonnaci number: 165580141 -  Time: 0.498588795
42th fibonnaci number: 267914296 -  Time: 0.793824734
43th fibonnaci number: 433494437 -  Time: 1.323325593
44th fibonnaci number: 701408733 -  Time: 2.098209943
45th fibonnaci number: 1134903170 -  Time: 3.392917489
46th fibonnaci number: 1836311903 -  Time: 5.506675921
47th fibonnaci number: -1323752223 -  Time: 8.803592621
48th fibonnaci number: 512559680 -  Time: 14.295023778
49th fibonnaci number: -811192543 -  Time: 23.030062974
50th fibonnaci number: -298632863 -  Time: 37.217244704
51th fibonnaci number: -1109825406 -  Time: 60.224418869
```

Clicker 2 - Failing Spectacularly



```
50th fibonnaci number: -298632863 -  Time: 37.21
```

- How long to calculate the 70th Fibonacci Number with this method?

A. 37 seconds

B. 74 seconds

C. 740 seconds

D. 14,800 seconds

E. None of these

Aside - Overflow

- at 47th Fibonacci number overflows int
- Could use BigInteger class instead

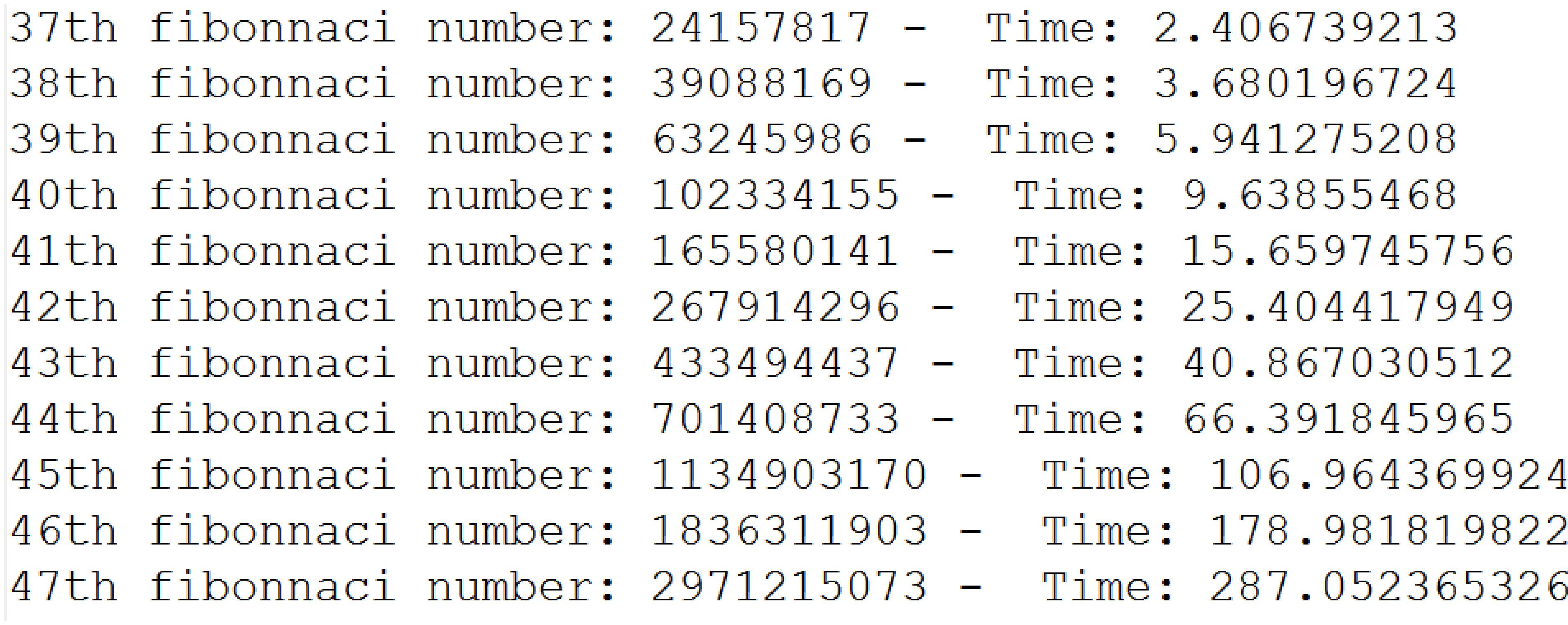
```
private static final BigInteger one
        = new BigInteger("1");

private static final BigInteger two
        = new BigInteger("2");

public static BigInteger fib(BigInteger n) {
        if (n.compareTo(two) <= 0)
                return one;
        else {
                BigInteger firstTerm = fib(n.subtract(two));
                BigInteger secondTerm = fib(n.subtract(one));
                return firstTerm.add(secondTerm);
        }
}
```

Aside - BigInteger

- Answers correct beyond 46th Fibonacci number
- Even slower, math on BigIntegers, object creation, and garbage collection



```
37th fibonnaci number: 24157817 -  Time: 2.406739213
38th fibonnaci number: 39088169 -  Time: 3.680196724
39th fibonnaci number: 63245986 -  Time: 5.941275208
40th fibonnaci number: 102334155 -  Time: 9.63855468
41th fibonnaci number: 165580141 -  Time: 15.659745756
42th fibonnaci number: 267914296 -  Time: 25.404417949
43th fibonnaci number: 433494437 -  Time: 40.867030512
44th fibonnaci number: 701408733 -  Time: 66.391845965
45th fibonnaci number: 1134903170 -  Time: 106.964369924
46th fibonnaci number: 1836311903 -  Time: 178.981819822
47th fibonnaci number: 2971215073 -  Time: 287.052365326
```

Slow Fibonacci

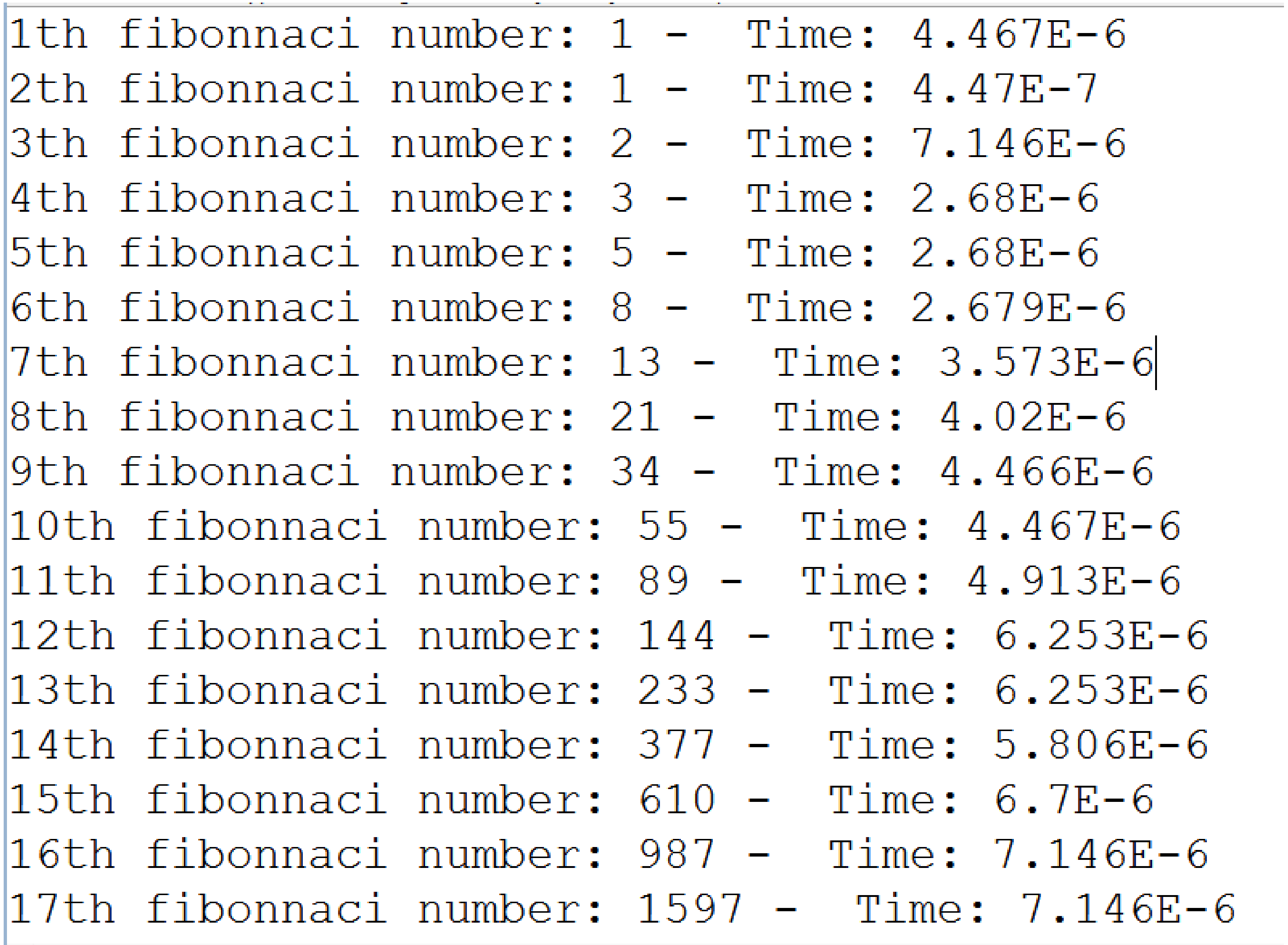
- Why so slow?
- Algorithm keeps calculating the same value over and over
- When calculating the 40th Fibonacci number the algorithm calculates the 4th Fibonacci number **24,157,817** times!!!

Fast Fibonacci

- Instead of starting with the big problem and working down to the small problems
- ... start with the small problem and work up to the big problem

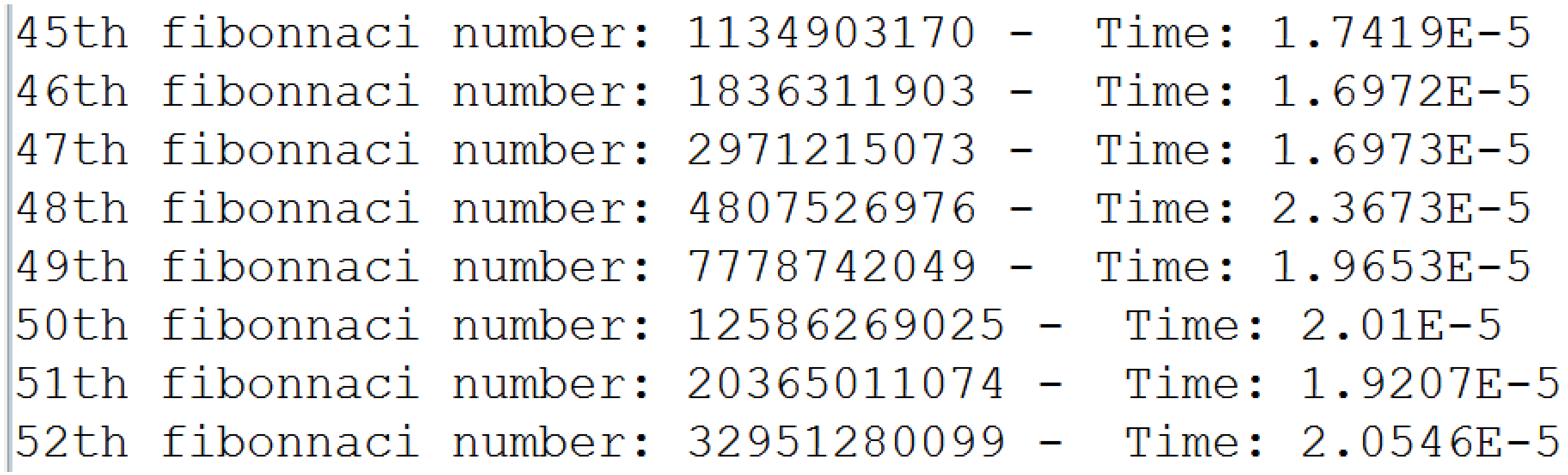
```
public static BigInteger fastFib(int n) {
        BigInteger smallTerm = one;
        BigInteger largeTerm = one;
        for (int i = 3; i <= n; i++) {
                BigInteger temp = largeTerm;
                largeTerm = largeTerm.add(smallTerm);
                smallTerm = temp;
        }
        return largeTerm;
}
```

Fast Fibonacci

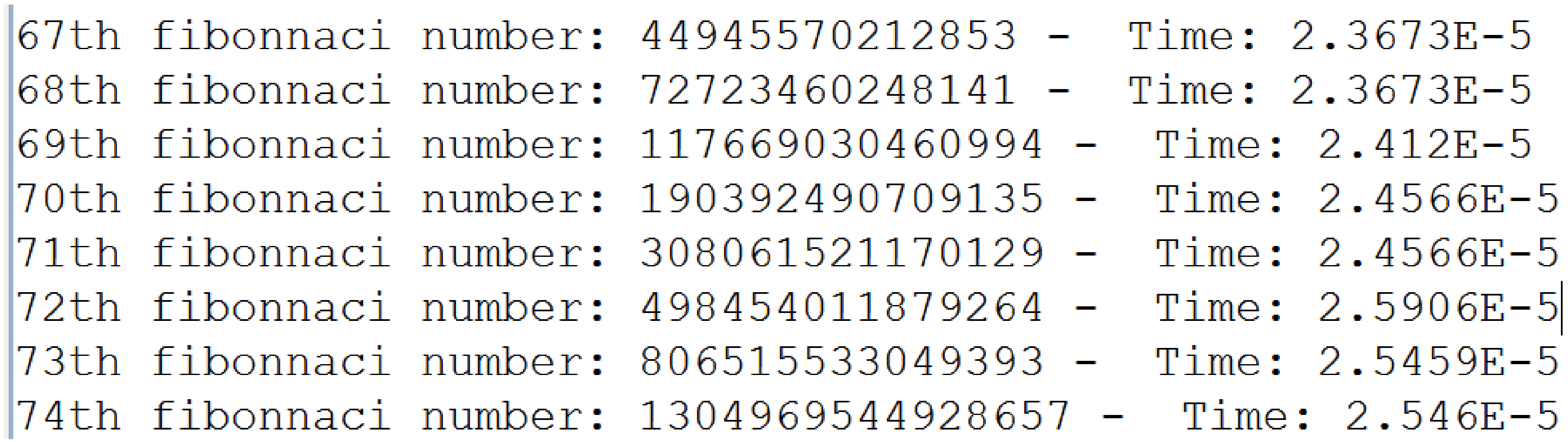


```
1th fibonnaci number: 1 -  Time: 4.467E-6
2th fibonnaci number: 1 -  Time: 4.47E-7
3th fibonnaci number: 2 -  Time: 7.146E-6
4th fibonnaci number: 3 -  Time: 2.68E-6
5th fibonnaci number: 5 -  Time: 2.68E-6
6th fibonnaci number: 8 -  Time: 2.679E-6
7th fibonnaci number: 13 -  Time: 3.573E-6
8th fibonnaci number: 21 -  Time: 4.02E-6
9th fibonnaci number: 34 -  Time: 4.466E-6
10th fibonnaci number: 55 -  Time: 4.467E-6
11th fibonnaci number: 89 -  Time: 4.913E-6
12th fibonnaci number: 144 -  Time: 6.253E-6
13th fibonnaci number: 233 -  Time: 6.253E-6
14th fibonnaci number: 377 -  Time: 5.806E-6
15th fibonnaci number: 610 -  Time: 6.7E-6
16th fibonnaci number: 987 -  Time: 7.146E-6
17th fibonnaci number: 1597 -  Time: 7.146E-6
```

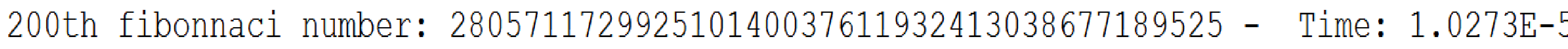
Fast Fibonacci



```
45th fibonnaci number: 1134903170 -  Time: 1.7419E-5
46th fibonnaci number: 1836311903 -  Time: 1.6972E-5
47th fibonnaci number: 2971215073 -  Time: 1.6973E-5
48th fibonnaci number: 4807526976 -  Time: 2.3673E-5
49th fibonnaci number: 7778742049 -  Time: 1.9653E-5
50th fibonnaci number: 12586269025 -  Time: 2.01E-5
51th fibonnaci number: 20365011074 -  Time: 1.9207E-5
52th fibonnaci number: 32951280099 -  Time: 2.0546E-5
```



```
67th fibonnaci number: 44945570212853 -  Time: 2.3673E-5
68th fibonnaci number: 72723460248141 -  Time: 2.3673E-5
69th fibonnaci number: 117669030460994 -  Time: 2.412E-5
70th fibonnaci number: 190392490709135 -  Time: 2.4566E-5
71th fibonnaci number: 308061521170129 -  Time: 2.4566E-5
72th fibonnaci number: 498454011879264 -  Time: 2.5906E-5
73th fibonnaci number: 806515533049393 -  Time: 2.5459E-5
74th fibonnaci number: 1304969544928657 -  Time: 2.546E-5
```



```
200th fibonnaci number: 280571172992510140037611932413038677189525 -  Time: 1.0273E-
```

Memoization

- Store (cache) results from computations for later lookup
- Memoization of Fibonacci Numbers

```
public class FibMemo {

      private static List<BigInteger> lookupTable;

      private static final BigInteger ONE
            = new BigInteger("1");

      static {
            lookupTable = new ArrayList<>();
            lookupTable.add(null);
            lookupTable.add(ONE);
            lookupTable.add(ONE);
      }
```

Fibonacci Memoization

```
public static BigInteger fib(int n) {
    // check lookup table
    if (n < lookupTable.size()) {
        return lookupTable.get(n);
    }

    // Calculate nth Fibonacci.
    // Don't repeat work. Start with the last known.
    BigInteger smallTerm
        = lookupTable.get(lookupTable.size() - 2);
    BigInteger largeTerm
        = lookupTable.get(lookupTable.size() - 1);
    for(int i = lookupTable.size(); i <= n; i++) {
        BigInteger temp = largeTerm;
        largeTerm = largeTerm.add(smallTerm);
        lookupTable.add(largeTerm); // memo
        smallTerm = temp;
    }
    return largeTerm;
}
```

Dynamic Programming

- When to use?
- When a big problem can be broken up into sub problems.
- **Solution to original problem can be calculated from results of smaller problems.**
  - larger problems depend on previous solutions
- **Sub problems must have a natural ordering from smallest to largest (simplest to hardest)**
- Multiple techniques within DP

DP Algorithms

- Step 1: Define the *meaning* of the subproblems (in English for sure, Mathematically as well if you find it helpful).
- Step 2: Show where the solution will be found.
- Step 3: Show how to set the first subproblem.
- Step 4: Define the order in which the subproblems are solved.
- Step 5: Show how to compute the answer to each subproblem using the previously computed subproblems. (This step is typically polynomial, once the other subproblems are solved.)

Dynamic Programming Requires:

- overlapping sub problems:
  - problem can be broken down into sub problems
  - obvious with Fibonacci
  - Fib(N) = Fib(N - 2) + Fib(N - 1) for N >= 3
- optimal substructure:
  - the optimal solution for a problem can be constructed from optimal solutions of its sub problems
  - In Fibonacci just sub problems, no optimality
  - min coins opt(36) = 1_{12} + opt(24)   [1, 5, 12]

Dynamic Programing Example

- Another simple example
- Finding the best solution involves finding the best answer to simpler problems
- Given a set of coins with values (V_1, V_2, … V_N) and a target sum S, find the fewest coins required to equal S
- What is Greedy Algorithm approach?
- Does it always work?
- {1, 5, 12} and target sum = 15 (12, 1, 1, 1)
- Could use recursive backtracking …

Minimum Number of Coins

- To find minimum number of coins to sum to 15 with values {1, 5, 12} start with sum 0
  - recursive backtracking would likely start with 15
- Let M(S) = minimum number of coins to sum to S
- At each step look at target sum, coins available, and previous sums
  - pick the smallest option

Minimum Number of Coins

- M(0) = 0 coins
- M(1) = 1 coin (1 coin)
- M(2) = 2 coins (1 coin + M(1))
- M(3) = 3 coins (1 coin + M(2))
- M(4) = 4 coins (1 coin + M(3))
- M(5) = interesting, 2 options available:

  1 + others   OR   single 5

if 1 then 1 + M(4) = 5, if 5 then 1 + M(0) = 1

clearly better to pick the coin worth 5

Minimum Number of Coins

- M(0) = 0
- M(1) = 1 (1 coin)
- M(2) = 2 (1 coin + M(1))
- M(3) = 3 (1 coin + M(2))
- M(4) = 4 (1 coin + M(3))
- M(5) = 1 (1 coin + M(0))
- M(6) = 2 (1 coin + M(5))
- M(7) = 3 (1 coin + M(6))
- M(8) = 4 (1 coin + M(7))
- M(9) = 5 (1 coin + M(8))
- M(10) = 2 (1 coin + M(5))
  options: 1, 5
- M(11) = 2 (1 coin + M(10))
  options: 1, 5
- M(12) = 1 (1 coin + M(0))
  options: 1, 5, 12
- M(13) = 2 (1 coin + M(12))
  options: 1, 12
- M(14) = 3 (1 coin + M(13))
  options: 1, 12
- M(15) = 3 (1 coin + M(10))
  options: 1, 5, 12

KNAPSACK PROBLEM - RECURSIVE BACKTRACKING AND DYNAMIC PROGRAMMING

Knapsack Problem

- A variation of a *bin packing* problem
- Similar to fair teams problem from recursion assignment
- You have a set of items
- Each item has a weight and a value
- You have a knapsack with a weight limit
- Goal: Maximize the ***value*** of the items you put in the knapsack without exceeding the weight limit

Knapsack Example

- Items:

Item Number	Weight of Item	Value of Item	Value per unit Weight
1	1	6	6.0
2	2	11	5.5
3	4	1	0.25
4	4	12	3.0
5	6	19	3.167
6	7	12	1.714

- Weight Limit = 8
- A greedy solution: Take the highest ratio item that will fit: (1, 6), (2, 11), and (4, 12)
- Total value = 6 + 11 + 12 = 29
- Clicker 3 - Is this optimal? A. No B. Yes

Knapsack - Recursive Backtracking

```
    private static int knapsack(ArrayList<Item> items,
            int current, int capacity) {

        int result = 0;
        if (current < items.size()) {
            // don't use item
            int withoutItem
                = knapsack(items, current + 1, capacity);
            int withItem = 0;
            // if current item will fit, try it
            Item currentItem = items.get(current);
            if (currentItem.weight <= capacity) {
                withItem += currentItem.value;
                withItem += knapsack(items, current + 1,
                        capacity - currentItem.weight);
            }
            result = Math.max(withoutItem, withItem);
        }
        return result;
    }
```

Knapsack - Dynamic Programming

- Recursive backtracking starts with max capacity and makes choice for items: choices are:
  – take the item if it fits
  – don't take the item
- Dynamic Programming, start with simpler problems
- Reduce number of items available
- … AND Reduce weight limit on knapsack
- Creates a 2d array of possibilities

Knapsack - Optimal Function

- OptimalSolution(items, weight) is best solution given a subset of items and a weight limit
- 2 options:
- OptimalSolution does not select i^th item
  - select best solution for items 1 to i - 1with weight limit of w
- OptimalSolution selects i^th item
  - New weight limit = w - weight of i^th item
  - select best solution for items 1 to i - 1with new weight limit

Knapsack Optimal Function

- OptimalSolution(items, weight limit) =

  0 if 0 items

  OptimalSolution(items - 1, weight) if weight of ith item is greater than allowed weight
  $w_i > w$ (In others ith item doesn't fit)

  max of (OptimalSolution(items - 1, w),
  value of ith item +
  OptimalSolution(items - 1, $w - w_i$)

Knapsack - Algorithm

- Create a 2d array to store value of best option given subset of items and possible weights

Item Number	Weight of Item	Value of Item
1	1	6
2	2	11
3	4	1
4	4	12
5	6	19
6	7	12

- In our example 0 to 6 items and weight limits of of 0 to 8

- Fill in table using OptimalSolution Function

Knapsack Algorithm

```
Given N items and WeightLimit

Create Matrix M with N + 1 rows and WeightLimit + 1 columns

For weight = 0 to WeightLimit
    M[0, w] = 0

For item = 1 to N
    for weight = 1 to WeightLimit
        if(weight of ith item > weight)
            M[item, weight] = M[item - 1, weight]
        else
            M[item, weight] = max of
            M[item - 1, weight] AND
            value of item + M[item - 1, weight - weight of item]
```

Knapsack - Table

Item	Weight	Value
1	1	6
2	2	11
3	4	1
4	4	12
5	6	19
6	7	12

items / capacity	0	1	2	3	4	5	6	7	8
{}	0	0	0	0	0	0	0	0	0
{<u>1</u>}									
{1,<u>2</u>}									
{1, 2, <u>3</u>}									
{1, 2, 3, <u>4</u>}									
{1, 2, 3, 4, <u>5</u>}									
{1, 2, 3, 4, 5, <u>6</u>}									

Knapsack - Completed Table

items / weight	0	1	2	3	4	5	6	7	8
{}	0	0	0	0	0	0	0	0	0
{1} [1, 6]	0	6	6	6	6	6	6	6	6
{1,2} [2, 11]	0	6	11	17	17	17	17	17	17
{1, 2, 3} [4, 1]	0	6	11	17	17	17	17	18	18
{1, 2, 3, 4} [4, 12]	0	6	11	17	17	18	23	29	29
{1, 2, 3, 4, 5} [6, 19]	0	6	11	17	17	18	23	29	30
{1, 2, 3, 4, 5, 6} [7, 12]	0	6	11	17	17	18	23	29	30

Knapsack - Items to Take

items / weight	0	1	2	3	4	5	6	7	8
{}	0	0	0	0	0	0	0	0	0
{1} [1, 6]	0	6	6	6	6	6	6	6	6
{1,2} [2, 11]	0	6	11	17	17	17	17	17	17
{1, 2, 3} [4, 1]	0	6	11	17	17	17	17	17	17
{1, 2, 3, 4} [4, 12]	0	6	11	17	17	18	23	29	29
{1, 2, 3, 4, 5} [6, 19]	0	6	11	17	17	18	23	29	30
{1, 2, 3, 4, 5, 6} [7, 12]	0	6	11	17	17	18	23	29	30

Dynamic Knapsack

```
// dynamic programming approach
public static int knapsack(ArrayList<Item> items, int maxCapacity) {
    final int ROWS = items.size() + 1;
    final int COLS = maxCapacity + 1;
    int[][] partialSolutions = new int[ROWS][COLS];
    // first row and first column all zeros

    for(int item = 1; item <= items.size(); item++) {
        for(int capacity = 1; capacity <= maxCapacity; capacity++) {
            Item currentItem = items.get(item - 1);
            int bestSoFar = partialSolutions[item - 1][capacity];
            if( currentItem.weight <= capacity) {
                int withItem = currentItem.value;
                int capLeft = capacity - currentItem.weight;
                withItem += partialSolutions[item - 1][capLeft];
                if (withItem > bestSoFar) {
                    bestSoFar = withItem;
                }
            }
            partialSolutions[item][capacity] = bestSoFar;
        }
    }
    return partialSolutions[ROWS - 1][COLS - 1];
}
```

Dynamic vs. Recursive Backtracking Timing Data

```
Number of items: 32. Capacity: 123
Recursive knapsack. Answer: 740, time: 10.0268025
Dynamic knapsack.   Answer: 740, time: 3.43999E-4

Number of items: 33. Capacity: 210
Recursive knapsack. Answer: 893, time: 23.0677814
Dynamic knapsack.   Answer: 893, time: 6.76899E-4

Number of items: 34. Capacity: 173
Recursive knapsack. Answer: 941, time: 89.8400178
Dynamic knapsack.   Answer: 941, time: 0.0015702

Number of items: 35. Capacity: 93
Recursive knapsack. Answer: 638, time: 81.0132219
Dynamic knapsack.   Answer: 638, time: 2.95601E-4
```

Clicker 4

- Which approach to the knapsack problem uses more memory?

A. the recursive backtracking approach

B. the dynamic programming approach

C. they use about the same amount of memory