"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
  – Charles E Wilson, Secretary of Defense against research, especially mathematical research
  – how could any one oppose "dynamic"?
Dynamic Programming

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

\[
N! = 1 \text{ for } N == 0 \\
N! = N \times (N - 1)! \text{ for } N > 0
\]
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

```java
// 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);
}

› Order of this method?
A. O(1)   B. O(log N)   C. O(N)    D. O(N^2)    E. O(2^N)
```
Failing Spectacularly

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

36th fibonacci number: 14930352  -  Time: 0.045372057
37th fibonacci number: 24157817  -  Time: 0.071195386
38th fibonacci number: 39088169  -  Time: 0.116922086
39th fibonacci number: 63245986  -  Time: 0.186926245
40th fibonacci number: 102334155  -  Time: 0.308602967
41th fibonacci number: 165580141  -  Time: 0.498588795
42th fibonacci number: 267914296  -  Time: 0.793824734
43th fibonacci number: 433494437  -  Time: 1.323325593
44th fibonacci number: 701408733  -  Time: 2.098209943
45th fibonacci number: 1134903170  -  Time: 3.392917489
46th fibonacci number: 1836311903  -  Time: 5.506675921
47th fibonacci number: -1323752223  -  Time: 8.803592621
48th fibonacci number: 512559680  -  Time: 14.295023778
49th fibonacci number: -811192543  -  Time: 23.030062974
50th fibonacci number: -298632863  -  Time: 37.217244704
51th fibonacci number: -1109825406  -  Time: 60.224418869
How long to calculate the 70\textsuperscript{th} 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

```java
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 46\textsuperscript{th} Fibonacci number
- Even slower due to creation of so many objects

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

```java
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

<table>
<thead>
<tr>
<th>Position</th>
<th>Fibonacci Number</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4.467E-6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4.47E-7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>7.146E-6</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2.68E-6</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>2.68E-6</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>2.679E-6</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>3.573E-6</td>
</tr>
<tr>
<td>8</td>
<td>21</td>
<td>4.02E-6</td>
</tr>
<tr>
<td>9</td>
<td>34</td>
<td>4.466E-6</td>
</tr>
<tr>
<td>10</td>
<td>55</td>
<td>4.467E-6</td>
</tr>
<tr>
<td>11</td>
<td>89</td>
<td>4.913E-6</td>
</tr>
<tr>
<td>12</td>
<td>144</td>
<td>6.253E-6</td>
</tr>
<tr>
<td>13</td>
<td>233</td>
<td>6.253E-6</td>
</tr>
<tr>
<td>14</td>
<td>377</td>
<td>5.806E-6</td>
</tr>
<tr>
<td>15</td>
<td>610</td>
<td>6.7E-6</td>
</tr>
<tr>
<td>16</td>
<td>987</td>
<td>7.146E-6</td>
</tr>
<tr>
<td>17</td>
<td>1597</td>
<td>7.146E-6</td>
</tr>
</tbody>
</table>
# Fast Fibonacci

<table>
<thead>
<tr>
<th>Fibonacci Number</th>
<th>Number</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>45th</td>
<td>1134903170</td>
<td>1.7419E-5</td>
</tr>
<tr>
<td>46th</td>
<td>1836311903</td>
<td>1.6972E-5</td>
</tr>
<tr>
<td>47th</td>
<td>2971215073</td>
<td>1.6973E-5</td>
</tr>
<tr>
<td>48th</td>
<td>4807526976</td>
<td>2.3673E-5</td>
</tr>
<tr>
<td>49th</td>
<td>7778742049</td>
<td>1.9653E-5</td>
</tr>
<tr>
<td>50th</td>
<td>12586269025</td>
<td>2.01E-5</td>
</tr>
<tr>
<td>51th</td>
<td>20365011074</td>
<td>1.9207E-5</td>
</tr>
<tr>
<td>52th</td>
<td>32951280099</td>
<td>2.0546E-5</td>
</tr>
<tr>
<td>67th</td>
<td>44945570212853</td>
<td>2.3673E-5</td>
</tr>
<tr>
<td>68th</td>
<td>72723460248141</td>
<td>2.3673E-5</td>
</tr>
<tr>
<td>69th</td>
<td>117669030460994</td>
<td>2.412E-5</td>
</tr>
<tr>
<td>70th</td>
<td>190392490709135</td>
<td>2.4566E-5</td>
</tr>
<tr>
<td>71th</td>
<td>308061521170129</td>
<td>2.4566E-5</td>
</tr>
<tr>
<td>72th</td>
<td>498454011879264</td>
<td>2.5906E-5</td>
</tr>
<tr>
<td>73th</td>
<td>806515533049393</td>
<td>2.5459E-5</td>
</tr>
<tr>
<td>74th</td>
<td>1304969544928657</td>
<td>2.546E-5</td>
</tr>
<tr>
<td>200th</td>
<td>280571172992510140037611932413038677189525</td>
<td>1.0273E-5</td>
</tr>
</tbody>
</table>
Memoization

- Store (cache) results from functions for later lookup

- Memoization of Fibonacci Numbers

```java
public class FibMemo {

    private static List<BigInteger> lookupTable = new ArrayList<BigInteger>();

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

    static {
        // no fib for n == 0
        lookupTable.add(null);
        lookupTable.add(one);
        lookupTable.add(one);
    }
}
```
public static BigInteger fib(int n) {
    // check lookup table
    if (n < lookupTable.size())
        return lookupTable.get(n);

    // must calculate nth fibonacci
    // don't repeat work
    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.
- Sub problems have a natural ordering from smallest to largest.
  - larger problems depend on previous solutions
- Multiple techniques within DP
DP Algorithms

- Step 1: Define the *meaning* of the subproblems (in English for sure, in Math 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 Example

- Another simple example
- Finding the best solution involves finding the best answer to simpler problems
- Given a set of coins with values \((V_1, V_2, \ldots, V_N)\) and a target sum \(S\), find the fewest coins required to equal \(S\)
- Recall, Greedy algorithm does not always work:
  - \(\{1, 5, 12\}\) and target sum = 15
- 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 \( M(S) = \text{minimum number of coins to sum to } S \)

- 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 + $M(5)$)

- $M(11) = 2$ (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 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:**

- **Weight Limit = 8**

- One greedy solution: Take the highest ratio item that will fit: (1, 6), (2, 11), and (4, 12)

- Total value = 6 + 11 + 12 = 29

- Is this optimal? A. Yes B. No

<table>
<thead>
<tr>
<th>Item Number</th>
<th>Weight of Item</th>
<th>Value of Item</th>
<th>Value per unit Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
<td>6.0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>11</td>
<td>5.5</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>0.25</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>12</td>
<td>3.0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>19</td>
<td>3.167</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>12</td>
<td>1.714</td>
</tr>
</tbody>
</table>
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 - 1 with 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 - 1 with new weight limit
Knapsack Optimal Function

- $\text{OptimalSolution}(\text{items}, \text{weight limit}) =$

  0 if 0 items

  $\text{OptimalSolution}(\text{items} - 1, \text{weight})$ if weight of $i^{th}$ item is greater than allowed weight $w_i > w$ (In others $i^{th}$ item doesn't fit)

  $\max \text{ of } (\text{OptimalSolution}(\text{items} - 1, w), \text{value of } i^{th} \text{ item } + \text{OptimalSolution}(\text{items} - 1, w - w_i))$
Knapsack - Algorithm

- Create a 2d array to store value of best option given subset of items and possible weights
- In our example 0 to 6 items and weight limits of 0 to 8
- Fill in table using OptimalSolution Function

<table>
<thead>
<tr>
<th>Item Number</th>
<th>Weight of Item</th>
<th>Value of Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>12</td>
</tr>
</tbody>
</table>
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]
<table>
<thead>
<tr>
<th>Items / Weight</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1, 2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1, 2, 3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1, 2, 3, 4}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1, 2, 3, 4, 5}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1, 2, 3, 4, 5, 6}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Knapsack - Table

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>items / weight</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>---------------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>{}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{1}</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>[1, 6]</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>{1, 2}</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>[2, 11]</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>{1, 2, 3}</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>[4, 1]</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>{1, 2, 3, 4}</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>[4, 12]</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>{1, 2, 3, 4, 5}</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>[6, 19]</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>{1, 2, 3, 4, 5, 6}</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>[7, 12]</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
## Knapsack - Items to Take

<table>
<thead>
<tr>
<th>items / weight</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{1} [1, 6]</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>{1, 2} [2, 11]</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>{1, 2, 3} [4, 1]</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>{1, 2, 3, 4} [4, 12]</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>18</td>
<td>23</td>
<td>29</td>
</tr>
<tr>
<td>{1, 2, 3, 4, 5} [6, 19]</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>17</td>
<td>17</td>
<td>18</td>
<td>23</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>{1, 2, 3, 4, 5, 6} [7, 12]</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>17</td>
<td>17</td>
<td>18</td>
<td>23</td>
<td>29</td>
<td>30</td>
</tr>
</tbody>
</table>
Dynamic Knapsack

```java
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];

    for (int item = 1; item <= items.size(); item++) {
        for (int capacity = 0; capacity <= maxCapacity; capacity++) {
            Item currentItem = items.get(item - 1);
            int best = partialSolutions[item - 1][capacity];
            if (currentItem.weight <= capacity) {
                int withItem = currentItem.value;
                int capLeft = capacity - currentItem.weight;
                withItem += partialSolutions[item - 1][capLeft];
                if (withItem > best) {
                    best = withItem;
                }
            }
            partialSolutions[item][capacity] = best;
        }
    }
    return partialSolutions[ROWS - 1][COLS - 1];
}
```
Dynamic vs. Recursive Backtracking

Number of items: 34. Capacity: 258
Recursive knapsack. Answer: 433, time: 111.77610595
Dynamic knapsack. Answer: 433, time: 2.6353E-5

Number of items: 35. Capacity: 199
Recursive knapsack. Answer: 318, time: 154.049166387
Dynamic knapsack. Answer: 318, time: 2.3673E-5

Number of items: 36. Capacity: 260
Recursive knapsack. Answer: 436, time: 451.122478468
Dynamic knapsack. Answer: 436, time: 3.0373E-5

Number of items: 37. Capacity: 238
Recursive knapsack. Answer: 411, time: 636.560835011
Dynamic knapsack. Answer: 411, time: 3.5285E-5

Number of items: 38. Capacity: 308