Return to Recursive algorithms:

Divide-and-Conquer

• Divide-and-Conquer
  – Divide big problem into smaller subproblems
  – Conquer each subproblem separately
  – Merge the solutions of the subproblems into the solution of the big problem

• Example:

Fibonacci(n)

if (n ≤ 1) then return n
else return Fibonacci(n-1) + Fibonacci(n-2)

Very slow algorithm because we recompute Fibonacci(i) many many times...
Dynamic programming

- Solve each small problem once, saving their solution
- Use the solutions of small problems to obtain solutions to larger problems

FibonacciDynProg(n)

```c
int F[0...n];
F[0] = 0 ;
F[1] = 1;
for i = 2 to n do
  F[i] = F[i-2] + F[i-1]
return F[n]
```
The change making problem

- A country has coins worth 1, 3, 5, and 8 cents
- What is the smallest number of coins needed to make
  - 25 cents?
  - 15 cents?
- In general, with coins denominations $C_1, C_2, ..., C_k$, how to find the smallest number of coins needed to make a total of $n$ cents?
Recursive algo. for making change

- Define $\text{Opt}(n)$ as the optimal number of coins needed to make $n$ cents
- We first write a recursive formula for $\text{Opt}(n)$:

  $\text{Opt}(0) = 0$

  $\text{Opt}(n) = 1 + \min\{ \text{Opt}(n - C_1), \text{Opt}(n - C_2), \ldots, \text{Opt}(n - C_k) \}$

  (excluding cases where $C_i > n$)

Example: with coins 1, 3, 5, 8

<table>
<thead>
<tr>
<th>n</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>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt(n)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recursive algo for making change

- Define Opt(n) as the optimal number of coins needed to make n cents
- We first write a recursive formula for Opt(n):
  \[
  \text{Opt}(0) = 0 \\
  \text{Opt}(n) = 1 + \min\{ \text{Opt} (n - C_1), \text{Opt} (n - C_2), \ldots, \text{Opt} (n - C_k) \} \\
  \text{(excluding cases where } C_i > n) 
  \]

Example: with coins 1, 3, 5, 8

<table>
<thead>
<tr>
<th>n</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>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt(n)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

\[
\text{Opt}(15) = 1 + \min\{ \text{Opt}(15 - 1), \text{Opt}(15 - 3), \text{Opt}(15 - 5), \text{Opt}(15 - 8) \} \\
= 1 + \min\{ 3, 3, 2, 3 \} \\
= 1 + 2 = 3
\]
Recursive algo for making change

Opt(0) = 0
Opt(n) = 1 + \min\{ \text{Opt}( n - C_1 ), \text{Opt}( n - C_2 ), \ldots, \text{Opt}( n - C_k ) \} \\
(excluding cases where C_i > n)

Problem: Recursive algorithm is very slow, because it keeps recomputing the same Opt values over and over again
Recursive algo for making change

Opt(0) = 0
Opt(n) = 1 + min{ Opt( n – C_1 ), Opt( n – C_2 ), … Opt( n – C_k ) }
(excluding cases where C_i > n)

Problem: Recursive algorithm is very slow, because it keeps recomputing the same Opt values over and over again.

<table>
<thead>
<tr>
<th>n</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>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt(n)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Recursive algo for making change

Opt(0) = 0
Opt(n) = 1 + min\{ Opt( n − C_1 ), Opt( n − C_2 ), \ldots, Opt( n − C_k ) \}
(excluding cases where C_i > n)

Problem: Recursive algorithm is very slow, because it keeps recomputing the same Opt values over and over again

<table>
<thead>
<tr>
<th>n</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>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt(n)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Recursive algo for making change

Opt(0) = 0
Opt(n) = 1 + min{ Opt( n − C_1 ), Opt( n − C_2 ), … Opt( n − C_k ) } 
(excluding cases where C_i > n)

Problem: Recursive algorithm is very slow, because it keeps recomputing the same Opt values over and over again

<table>
<thead>
<tr>
<th>n</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>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt(n)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Dyn. Prog. Algo. for making change

- Use the same formula...
  
  \[
  \begin{align*}
  \text{Opt}(0) &= 0 \\
  \text{Opt}(n) &= 1 + \min\{ \text{Opt}(n - C_1), \text{Opt}(n - C_2), \ldots, \text{Opt}(n - C_k) \} \\
  \end{align*}
  \]
  (excluding cases where \( C_i > n \))

- But compute the values of \( \text{Opt}(i) \), starting with \( i=0 \), then \( i=1 \), … up to \( i=n \). Save them in an array \( X \)

\[
\begin{array}{ccccccccccccccc}
 n & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
 X[n] & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 1 & 2 & 2 & 2 & 3 & 2 & 3 & 3 \\
\end{array}
\]

\[
X[15] = 1 + \min\{ X[15 - 1], X[15 - 3], X[15 - 5], X[15 - 8] \} \\
= 1 + \min\{ 3, 3, 2, 3 \} \\
= 1 + 2 = 3
\]

Important: This is not a recursive algorithm!
Each entry in the array is computed once.
Algorithm makeChange(C[0..k-1], n)
Input: an array C containing the values of the coins
        an integer n
Output: The minimal number of coins needed to
        make a total of n
int X[] = new int[n+1];       // X[0...n]
X[0] = 0
for i =1 to n do  // compute \( \min\{ \text{Opt} (i - C_j) \} \)
    smallest = +\(\infty\)
    for j = 0 to k-1 do
        if ( C[j] \(\leq\) i ) then smallest=min(smallest, X[i-C[j]])
    X[i] = 1 + smallest
Return X[n]
Making change - Greedy algorithm

• You need to give x ¢ in change, using coins of 1, 5, 10, and 25 cents. What is the smallest number of coins needed?

• Greedy approach:
  – Take as many 25 ¢ as possible, then
  – take as many 10 ¢ as possible, then
  – take as many 5 ¢ as possible, then
  – take as many 1 ¢ as needed to complete

• Example: 99 ¢ = 3* 25 ¢ + 2*10 ¢ + 1*5 ¢ + 4*1 ¢

• Is this always optimal?
Greedy-choice property

- A problem has the greedy choice property if:
  - An optimal solution can be reached by a series of locally optimal choices
- Change making: 1, 5, 10, 25 ¢: greedy is optimal
  1, 6, 10 ¢: greedy is not optimal

- For most optimization problems, greedy algorithms are not optimal. However, when they are, they are usually the fastest available.
Longest Increasing Subsequence

Problem: Given an array \( A[0..n-1] \) of integers, find the longest increasing subsequence in \( A \).

Example: \( A = 5 \ 1 \ 4 \ 2 \ 8 \ 4 \ 9 \ 1 \ 8 \ 9 \ 2 \)

Solution:

Slow algorithm: Try all possible subsequences…

\[
\text{for each possible subsequences } s \text{ of } A \text{ do} \\
\quad \text{if } (s \text{ is in increasing order}) \text{ then} \\
\quad \quad \text{if } (s \text{ is best seen so far}) \text{ then save } s \\
\text{return best seen so far}
\]
Dynamic Programming Solution

Let \( \text{LIS}[i] \) = length of the longest increasing subsequence ending at position \( i \) and containing \( A[i] \).

\[
\begin{align*}
A &= 5 \ 1 \ 4 \ 2 \ 8 \ 4 \ 9 \ 1 \ 8 \ 9 \ 2 \\
\text{LIS} &= 1 \ 1 \ 2 \ 2 \ 3 \\
\text{LIS}[0] &= 1 \\
\text{LIS}[i] &= 1 + \max \{ \text{LIS}[j] : j < i \text{ and } A[j] < A[i] \}
\end{align*}
\]
Dynamic Programming Solution

**Algorithm** LongestIncreasingSubsequence(A, n)

**Input:** an array A[0...n-1] of numbers

**Output:** the length of the longest increasing subsequence of A

LIS[0] = 1

for i = 1 to n-1 do
    LIS[i] = -1  // dummy initialization
    for j = 0 to i-1 do

return max(LIS)
Dynamic Programming Framework

- Dynamic Programming Algorithms are mostly used for optimization problems.
- To be able to use Dyn. Prog. Algo., the problem must have certain properties:
  - **Simple subproblems**: There must be a way to break the big problem into smaller subproblems. Subproblems must be identified with just a few indices.
  - **Subproblem optimization**: An optimal solution to the big problem must always be a combination of optimal solutions to the subproblems.
  - **Subproblem overlap**: Optimal solutions to unrelated problems can contain subproblems in common.