Consider the change making problem

- What is the minimum number of coins needed to make up a given value k?
- If you were working as a cashier, what would your algorithm be to solve this problem?
This is a *greedy algorithm*

- At each step, the algorithm makes the choice that seems to be best at the moment

- Have we seen greedy algorithms already this term?
  - Yes!
    - Building Huffman trees
    - Nearest neighbor approach to travelling salesman
... But wait ...

- Nearest neighbor doesn’t solve travelling salesman
  - (does not produce an optimal result)

- Does our change making algorithm solve the change making problem?
  - For US currency...
  - But what about a currency composed of pennies (1 cent), thrickels (3 cents), and fourters (4 cents)?
    - What denominations would it pick for k=6?
So what changed about the problem?

- For greedy algorithms to produce optimal results, problems must have two properties:
  - Optimal substructure
    - Optimal solution to a subproblem leads to an optimal solution to the overall problem
  - The greedy choice property
    - Globally optimal solutions can be assembled from locally optimal choices
- Why is optimal substructure not enough?
Finding all subproblems solutions can be inefficient

- Consider computing the Fibonacci sequence:

```c
int fib(n) {
    if (n == 0) { return 0; }
    else if (n == 1) { return 1; }
    else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

- What does the call tree for n = 5 look like?
The diagram illustrates the computation of the Fibonacci sequence for $fib(5)$. Each node in the tree represents a call to the $fib$ function, with the children representing the recursive calls to find the Fibonacci numbers for the adjacent indices. The computation follows the recursive definition of the Fibonacci sequence: $fib(n) = fib(n-1) + fib(n-2)$, with base cases $fib(0) = 0$ and $fib(1) = 1$. The tree structure shows the breakdown of the computation, with $fib(5)$ being computed by recursively calculating $fib(4)$ and $fib(3)$, and so on, until reaching the base cases.
How do we improve?
Memoization

```java
int[] F = new int[n+1];
F[0] = 0;
F[1] = 1;
for(int i = 2; i <= n; i++) { F[i] = -1; }

int dp_fib(x) {
    if (F[x] == -1) {
        F[x] = dp_fib(x-1) + dp_fib(x-2);
    }
    return F[x];
}
```
Note that we can also do this bottom-up

```java
int bottomup_fib(n) {
    if (n == 0)
        return 0;
    int[] F = new int[n+1];
    F[0] = 0;
    F[1] = 1;
    for(int i = 2; i <= n; i++) {
        F[i] = F[i-1] + F[i-2];
    }
    return F[n];
}
```
Can we improve this bottom-up approach?

```c
int improve_bottomup_fib(n) {
    int prev = 0;
    int cur = 1;
    int new;
    for (int i = 0; i < n; i++) {
        new = prev + cur;
        prev = cur;
        cur = new;
    }
    return cur;
}
```
Where can we apply dynamic programming?

- To problems with two properties:
  - Optimal substructure
    - Optimal solution to a subproblem leads to an optimal solution to the overall problem
  - Overlapping subproblems
    - Naively, we would need to recompute the same subproblem multiple times
Given a knapsack that can hold a weight limit $L$, and a set of $n$ types items that each has a weight ($w_i$) and value ($v_i$), what is the maximum value we can fit in the knapsack if we assume we have unbounded copies of each item?
Recursive example

weight: 6 3 4 2
value: 30 14 16 9

How much value in 10 lbs?

4 lbs?
4 lbs?
7 lbs?
6 lbs?
8 lbs?

1? 0? 2?
1? 4? 3? 5? 0?
3? 2? 4?
2? 5? 4? 6?
Recursive example

weight: 6 3 4 2
value: 30 14 16 9

10 lb. capacity

How much value in 10 lbs?

4 lbs?

7 lbs?

6 lbs?

8 lbs?

## Bottom-up example

<table>
<thead>
<tr>
<th>weight</th>
<th>6</th>
<th>3</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>30</td>
<td>14</td>
<td>16</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size</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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max val</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>14</td>
<td>18</td>
<td>23</td>
<td>30</td>
<td>32</td>
<td>39</td>
<td>44</td>
<td>48</td>
</tr>
</tbody>
</table>
Bottom-up solution

\[ K[0] = 0 \]

\[
\text{for } (l = 1; l <= L; l++) \{ \\
\hspace{1em} \text{int max } = 0; \\
\hspace{1em} \text{for } (i = 0; i < n; i++) \{ \\
\hspace{2em} \text{if } (w_i <= l \&\& v_i + K[l - w_i]) > max \} \{
\hspace{3em} \max = v_i + K[l - w_i]; \\
\hspace{2em}\} \\
\} \}

K[l] = \max;
Try adding as many copies of highest value per pound item as possible:

- Water: $30/6 = 5$
- Rope: $14/3 = 4.66$
- Flashlight: $16/4 = 4$
- Moonpie: $9/2 = 4.5$

Highest value per pound item? Water
  - Can fit 1 with 4 space left over

Next highest value per pound item? Rope
  - Can fit 1 with 1 space left over

No room for anything else

Total value in the 10 lb knapsack?
  - 44
    - Bogus!
What if we have a finite set of items that each has a weight and value?

- Two choices for each item:
  - Goes in the knapsack
  - Is left out

The 0/1 knapsack problem
0/1 Recursive example

How much value in 10 lbs?

weight: 6 3 4 2
value: 30 14 16 9

10 lbs?

10 lbs?

6 lbs?

3 lbs?

0 lbs?

7 lbs?

4 lbs?

1 lbs?
Recursive solution

```c
int knapSack(int[] wt, int[] val, int L, int n) {
    if (n == 0 || L == 0) { return 0; }
    if (wt[n-1] > L) {
        return knapSack(wt, val, L, n-1)
    }
    else {
        return max( val[n-1] + knapSack(wt, val, L-wt[n-1], n-1),
                    knapSack(wt, val, L, n-1) );
    }
}
```
The 0/1 knapsack dynamic programming example

\[ \text{wt} = [2, 3, 4, 5] \]
\[ \text{val} = [3, 4, 5, 6] \]

<table>
<thead>
<tr>
<th>i |</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

22
### The 0/1 Knapsack Dynamic Programming Example

Weights: \(wt = [2, 3, 4, 5]\)

Values: \(val = [3, 4, 5, 6]\)

<table>
<thead>
<tr>
<th>i|</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The 0/1 knapsack dynamic programming example

\[
\begin{aligned}
\text{wt} &= [2, 3, 4, 5] \\
\text{val} &= [3, 4, 5, 6]
\end{aligned}
\]

<table>
<thead>
<tr>
<th>i\n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<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>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The 0/1 knapsack dynamic programming example

\[ wt = [2, 3, 4, 5] \]
\[ val = [3, 4, 5, 6] \]

<table>
<thead>
<tr>
<th>( i ) ( \backslash )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<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>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The 0/1 knapsack dynamic programming example

wt = [ 2, 3, 4, 5 ]
val = [ 3, 4, 5, 6 ]

<table>
<thead>
<tr>
<th>i\l</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<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>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The 0/1 knapsack dynamic programming solution

```c
int knapSack(int wt[], int val[], int L, int n) {
    int[][][] K = new int[n+1][L+1];
    for (int i = 0; i <= n; i++) {
        for (int l = 0; l <= L; l++) {
            if (i==0 || l==0){ K[i][l] = 0 };  
            else if (wt[i-1] > l){ K[i][l] = K[i-1][l] };  
            else {
                K[i][l] = max(val[i-1] + K[i-1][l-wt[i-1]], K[i-1][l]);
            }
        }
    }
    return K[n][L];
}
```
Questions to ask in finding dynamic programming solutions:

- Does the problem have optimal substructure?
  - Can solve the problem by splitting it into smaller problems?
  - Can you identify subproblems that build up to a solution?

- Does the problem have overlapping subproblems?
  - Where would you find yourself recomputing values?
    - How can you save and reuse these values?
Consider a currency with $n$ different denominations of coins $d_1, d_2, \ldots, d_n$. What is the minimum number of coins needed to make up a given value $k$?