Course Content

I. Code Design Patterns
   1. Motivation, Classification, UML
   2. Creational Patterns
   3. Structural Patterns
   4. Behavioral Patterns

II. Algorithm Design Patterns

Slides on Weblearn
Algorithm Classification

- Much like with code design patterns, algorithms that use similar problem-solving approaches can be grouped together.
- These groups, include, but are not limited to, e.g.:
  - Backtracking
  - Divide and conquer
  - Greedy
  - Dynamic programming
- There are others, e.g. branch and bound, randomized, simple heuristics, etc.
Backtracking: Idea

- Backtracking aims to solve problems with a large search space, by (recursively) trying every alternative, and choosing the best one.
- A example is finding a path through a maze.
Backtracking: Idea

- At some point you might have two possible directions to explore.
- One strategy would be to go through path A first. If you get stuck, you’d *backtrack* to the junction.
- You’d then start searching path B.
Backtracking: Idea

• At every single junction you’d have 2 or more choices.
• The backtracking strategy aims to try every single choice, one after the other.
• If all choices fail, there is no solution to the maze.
Backtracking: Idea

Backtracking: Idea

Often implemented using recursion:
• From your start point, you iterate through each possible starting move.
• Next you recursively move forward.
• If you ever get stuck, the recursion takes you back to where you were, and you try the next possible move.

Note if you’re curious: going BFS on the tree would take you to branch and bound.

```cpp
bool findSolution(node n)
{
    if (n is a leaf node)
    {
        if (the leaf is a goal node) return true;
        else return false;
    }
    else
    {
        foreach (child c of n) {
            if (findSolution(c) succeeds) return true;
        }
        return false;
    }
}
```
Backtracking: n Queens Problem

• Find an arrangement of 8 queens on a chess board, such that two queens cannot attack each other.
• Queens can move all the way on any row, column or diagonal.
• Our restriction is that each row and column on the board should have exactly one queen.
Backtracking: n Queens Problem

• Possible backtracking solution:
  – Place a queen on the 1\textsuperscript{st} square in row 1.
  – Move to the next row, and place a queen on the 1\textsuperscript{st} available square that does not conflict with the 1\textsuperscript{st} queen.
  – Repeat until:
    • You solved the problem.
    • You get stuck – remove the queens in reverse order until you find another valid square to try.

http://www.hbmeyer.de/backtrack/achtdamen/autoeight.htm
**Divide and Conquer: Idea (from B16 SP – repeated for completeness)**

- **Divide and conquer** is another *recursive strategy* applicable to the solution of a wide variety of problems.
- The idea is to split each problem instance into two or more smaller parts, solve those, and recombine the results.
- Some of the best known and most famous (and useful) algorithms are of this form, notably quicksort and the Fast Fourier Transform (FFT).

```plaintext
% Divide and conquer pseudocode
solution = solve(problem)
If problem is easy, compute solution
Else
    Subdivide problem into subproblem1, subproblem2, ...
    sol1 = solve(subproblem1), sol2 = solve(subproblem2), ...
    Get solution by combining sol1, sol2, ...
```
Divide and Conquer: Quick Sort

• Assume you have an array of unsorted numbers.
• Idea:
  – chose one element as a pivot and partition the array, such that:
    • left side of the array contains < pivot.
    • right side of the array contains > pivot.
  – repeat for each side.
int partition(int[] A, int start, int end){
    int i = start + 1;
    int pivot = A[start]; //select the first element as pivot element.
    for (int j = start + 1; j <= end; j++)
    {
        // reorder the array such that elements which are less than pivot
        // are on the left, and elements that are more than the pivot are on the right
        if (A[j] < pivot) {
            swap(A, i, j); // swaps the value at location i with the value at location j
            i += 1;
        }
    }

    swap(A, start, i - 1); //put the pivot element in the right place.
    return i - 1; //return the position of the pivot
}

void quickSort(int[] A, int start, int end) {
    if (start < end) {
        int pivot_pos = partition(A, start, end); // the pivot position
        quickSort(A, start, pivot_pos - 1); //sorts the left side of pivot.
        quickSort(A, pivot_pos + 1, end); //sorts the right side of pivot.
    }
}
Divide and Conquer: Quick Sort

[Diagram of Quick Sort algorithm]

---

https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/tutorial/
Greedy: Idea

• A greedy algorithm will always make the best choice at the moment, and ignore future consequences.
• Does often work in practice, especially if there is no principled path towards identifying future consequences.
• Examples:
  – Most of non linear optimization examples from the Optimisation course.
Greedy: Huffman Encoding

• Used a lot for compression of data.
• Assume we want to encode a text with characters a,b,...,g with the following occurrences:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency:</td>
<td>37</td>
<td>18</td>
<td>29</td>
<td>13</td>
<td>30</td>
<td>17</td>
<td>6</td>
</tr>
</tbody>
</table>

• You could encode with a fixed length code:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency:</td>
<td>37</td>
<td>18</td>
<td>29</td>
<td>13</td>
<td>30</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>Fixed length code:</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
</tr>
<tr>
<td>Total cost:</td>
<td>111 bits</td>
<td>54 bits</td>
<td>87 bits</td>
<td>39 bits</td>
<td>90 bits</td>
<td>51 bits</td>
<td>18 bits</td>
</tr>
</tbody>
</table>
Greedy: Huffman Encoding

- We can come up with a variable-length code, where we use fewer bits for characters with a higher frequency.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency:</td>
<td>37</td>
<td>18</td>
<td>29</td>
<td>13</td>
<td>30</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>Fixed length code:</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
</tr>
<tr>
<td>Total cost (fixed):</td>
<td>111 bits</td>
<td>54 bits</td>
<td>87 bits</td>
<td>39 bits</td>
<td>90 bits</td>
<td>51 bits</td>
<td>18 bits</td>
</tr>
<tr>
<td>Variable length code:</td>
<td>10</td>
<td>011</td>
<td>111</td>
<td>1101</td>
<td>00</td>
<td>010</td>
<td>1100</td>
</tr>
<tr>
<td>Total cost (variable):</td>
<td>74 bits</td>
<td>54 bits</td>
<td>87 bits</td>
<td>52 bits</td>
<td>60 bits</td>
<td>51 bits</td>
<td>24 bits</td>
</tr>
</tbody>
</table>
Greedy: Huffman Tree

g, 6  d, 13  f, 17  b, 18  c, 29  e, 30  a, 37
Greedy: Huffman Tree

g,6  d,13  f,17  b,18  c,29  e,30  a,37
Greedy: Huffman Tree

19

g,6  d,13  f,17  b,18  c,29  e,30  a,37
Greedy: Huffman Tree

f,17  b,18  19  c,29  e,30  a,37

g,6  d,13
Greedy: Huffman Tree

```
<table>
<thead>
<tr>
<th>f,17</th>
<th>b,18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>g,6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d,13</td>
</tr>
<tr>
<td>c,29</td>
<td>e,30</td>
<td>a,37</td>
</tr>
</tbody>
</table>
```
Greedy: Huffman Tree

```
35
/   
f,17 b,18
|   /
|  19
| /  
g,6 d,13
c,29 e,30 a,37
```

© Otávio Braga
Greedy: Huffman Tree
Greedy: Huffman Tree
Greedy: Huffman Tree
Greedy: Huffman Tree

```
e, 30
  +---  35 ---+
  |     |     |
  |     |     |
  f, 17 b, 18

a, 37
  +---  48 ---+
  |     |     |
  |     |     |
  g, 6  d, 13

  +---  19 ---+
  |     |     |
  |     |     |
  c, 29
```

© Otávio Braga
Greedy: Huffman Tree
Greedy: Huffman Tree
1. Initialise: start with an unvisited set (all nodes apart from origin) and a visited set (origin). Set all distances to infinity. Make the origin the current node.

2. For current node, inspect all neighbours and compute tentative distance (from origin!).

3. If this distance is shorter than the one previously recorded for this neighbour (via another route) then overwrite previously recorded distance (and optimal route).

4. Mark current node as visited and remove from unvisited set.

5. From amongst set of all unvisited nodes, select node marked with smallest tentative distance and make it new current node.

6. Go to step 2, unless destination node has been marked visited.
Dynamic Programming: Idea

• Breaks up a problem into a collection of overlapping sub-problems, and builds up solutions to larger and larger (optimal) sub-problems.

• It is often applied, in problems such as:
  – The traveling salesman.
  – Fibonacci sequences.
  – Knapsack.
Dynamic Programming: 0-1 Knapsack

• You’re given a set $S$ of $n$ items, each having:
  – A positive value $v_i$
  – A positive weight $w_i$

• You’re asked to chose items with maximum total value, but with a weight of, at most, $W$.

• It’s the 0-1 problem (vs the factorial one) because we can’t divide the items.
Dynamic Programming: 0-1 Knapsack

• First attempt would be to think, maybe, greedy works.
• For that, the optimal subset of, e.g., 4 items, would be linked to the set of, e.g., 3 items.
• That is not the case.
Dynamic Programming: 0-1 Knapsack

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_0$</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>$I_1$</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>$I_2$</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$I_3$</td>
<td>8</td>
<td>11</td>
</tr>
</tbody>
</table>

- If we were to select the best set from $S_3 = \{I_0, I_1, I_2\}$ we’d find:
  \[ \{1 \times I_0, 1 \times I_1, 1 \times I_2\} \]
- If we were next to select the best set from $S_4 = \{I_0, I_1, I_2, I_3\}$ we’d find:
  \[ \{1 \times I_0, 1 \times I_2, 1 \times I_3\} \]
- However, $\{1 \times I_0, 1 \times I_1, 1 \times I_2\} \neq \{1 \times I_0, 1 \times I_1, 1 \times I_3\}$.
- Therefore, greedy style approaches would not work with this configuration of subproblem.
Dynamic Programming: 0-1 Knapsack

• Better idea: the best subset of \( S_k \) that has the total weight \( w \), is:
  – The best subset of \( S_{k-1} \) that has a total weight \( w \), or
  – The best subset of \( S_{k-1} \) that has a total weight \( w - w_k \), plus the item \( k \).

• We’ll demonstrate this with the following data:
  – \( n = 4 \) (# of elements)
  – \( W = 5 \) (max weight)
  – Elements (weight, value): (2,3), (3,4), (4,5), (5,6)
We define a value table $B[i, w]$ (see above):

- **Rows**: our items
- **Columns**: the maximum value obtained for a given maximum $w$
Dynamic Programming: 0-1 Knapsack

<table>
<thead>
<tr>
<th></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for i = 1 to n { // adding each item incrementally
    for w = 0 to W { // increasing the weight incrementally
        if (wi <= w) {
            // item i fits, so could be part of the solution
            if (vi + B[i - 1, w - wi] > B[i - 1, w])
                B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
            else
                B[i, w] = B[i - 1, w] // value didn't change with the new item
        }
        else {
            // item i does not fit, so would not be part of the solution
            B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
        }
    }
}
Dynamic Programming: 0-1 Knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

<table>
<thead>
<tr>
<th>I / w</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for i = 1 to n { // adding each item incrementally
    for w = 0 to W { // increasing the weight incrementally
        if (wi <= w) {
            // item i fits, so could be part of the solution
            if (vi + B[i - 1, w - wi] > B[i - 1, w])
                B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
            else
                B[i, w] = B[i - 1, w] // value didn't change with the new item
        }
        else {
            // item i does not fit, so would not be part of the solution
            B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
        }
    }
}
for i = 1 to n { // adding each item incrementally
    for w = 0 to W { // increasing the weight incrementally
        if (wi <= w) {
            // item i fits, so could be part of the solution
            if (vi + B[i - 1, w - wi] > B[i - 1, w])
                B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
            else
                B[i, w] = B[i - 1, w] // value didn't change with the new item
        } else {
            // item i does not fit, so would not be part of the solution
            B[i, w] = B[i - 1, w] // value won’t change, since we didn’t add a new item
        }
    }
}

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)
Dynamic Programming: 0-1 Knapsack

Items:

1: (2,3)  
2: (3,4)  
3: (4,5)  
4: (5,6)

<table>
<thead>
<tr>
<th>I / w</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for i = 1 to n { // adding each item incrementally
   for w = 0 to W { // increasing the weight incrementally
      if (wi <= w) {
         // item i fits, so could be part of the solution
         if (vi + B[i - 1, w - wi] > B[i - 1, w])
            B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
         else
            B[i, w] = B[i - 1, w] // value didn't change with the new item
      }
      else {
         // item i does not fit, so would not be part of the solution
         B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
      }
   }
}
for i = 1 to n { // adding each item incrementally
  for w = 0 to W { // increasing the weight incrementally
    if (wi <= w) {
      // item i fits, so could be part of the solution
      if (vi + B[i - 1, w - wi] > B[i - 1, w])
        B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
      else
        B[i, w] = B[i - 1, w] // value didn't change with the new item
    }
    else {
      // item i does not fit, so would not be part of the solution
      B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
    }
  }
}
Dynamic Programming: 0-1 Knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

<table>
<thead>
<tr>
<th></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></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for i = 1 to n { // adding each item incrementally
  for w = 0 to W { // increasing the weight incrementally
    if (wi <= w) {
      // item i fits, so could be part of the solution
      if (vi + B[i - 1, w - wi] > B[i - 1, w])
        B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
      else
        B[i, w] = B[i - 1, w] // value didn't change with the new item
    }
    else {
      // item i does not fit, so would not be part of the solution
      B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
    }
  }
}
Dynamic Programming: 0-1 Knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

<table>
<thead>
<tr>
<th>I/w</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for i = 1 to n { // adding each item incrementally
    for w = 0 to W { // increasing the weight incrementally
        if (wi <= w) {
            // item i fits, so could be part of the solution
            if (vi + B[i - 1, w - wi] > B[i - 1, w])
                B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
            else
                B[i, w] = B[i - 1, w] // value didn't change with the new item
        }
        else {
            // item i does not fit, so would not be part of the solution
            B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
        }
    }
}

© Sarah Buchanan
Dynamic Programming: 0-1 Knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

<table>
<thead>
<tr>
<th>I \ w</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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```java
def 01Knapsack(values, weights, n, W):
    # Initialize the DP table
    B = [[0 for w in range(W+1)] for i in range(n+1)]

    for i = 1 to n:  // adding each item incrementally
        for w = 0 to W:  // increasing the weight incrementally
            if (weights[i] <= w):
                # item i fits, so could be part of the solution
                if (values[i] + B[i-1, w-weights[i]] > B[i-1, w]):
                    B[i, w] = values[i] + B[i-1, w-weights[i]]  // value changed with the new item
                else:
                    B[i, w] = B[i-1, w]  // value didn't change with the new item
            else:
                # item i does not fit, so would not be part of the solution
                B[i, w] = B[i-1, w]  // value won't change, since we didn't add a new item
```

© Sarah Buchanan
Dynamic Programming: 0-1 Knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

<table>
<thead>
<tr>
<th>I</th>
<th>w</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>
<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>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for i = 1 to n { // adding each item incrementally
    for w = 0 to W { // increasing the weight incrementally
        if (wi <= w) {
            // item i fits, so could be part of the solution
            if (vi + B[i - 1, w - wi] > B[i - 1, w])
                B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
            else
                B[i, w] = B[i - 1, w] // value didn't change with the new item
        }
        else {
            // item i does not fit, so would not be part of the solution
            B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
        }
    }
}

© Sarah Buchanan
Dynamic Programming: 0-1 Knapsack

Items:
1: (2,3)  
2: (3,4)  
3: (4,5)  
4: (5,6)

<table>
<thead>
<tr>
<th>I / w</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></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for i = 1 to n { // adding each item incrementally
  for w = 0 to W { // increasing the weight incrementally
    if (wi <= w) {
      //item i fits, so could be part of the solution
      if (vi + B[i - 1, w - wi] > B[i - 1, w])
        B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
      else
        B[i, w] = B[i - 1, w] // value didn't change with the new item
    }
    else {
      // item i does not fit, so would not be part of the solution
      B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
    }
  }
}
Dynamic Programming: 0-1 Knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

<table>
<thead>
<tr>
<th>I / w</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>4</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for i = 1 to n { // adding each item incrementally
    for w = 0 to W { // increasing the weight incrementally
        if (wi <= w) {
            // item i fits, so could be part of the solution
            if (vi + B[i - 1, w - wi] > B[i - 1, w])
                B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
            else
                B[i, w] = B[i - 1, w] // value didn't change with the new item
        }
        else {
            // item i does not fit, so would not be part of the solution
            B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
        }
    }
}
Dynamic Programming: 0-1 Knapsack

Items:
1: (2,3)
**2: (3,4)**
3: (4,5)
4: (5,6)

<table>
<thead>
<tr>
<th></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for i = 1 to n { // adding each item incrementally
    for w = 0 to W { // increasing the weight incrementally
        if (wi <= w) {
            // item i fits, so could be part of the solution
            if (vi + B[i - 1, w - wi] > B[i - 1, w])
                B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
            else
                B[i, w] = B[i - 1, w] // value didn't change with the new item
        } else {
            // item i does not fit, so would not be part of the solution
            B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
        }
    }
}
Dynamic Programming: 0-1 Knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

<table>
<thead>
<tr>
<th>I / w</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>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

for i = 1 to n { // adding each item incrementally
    for w = 0 to W { // increasing the weight incrementally
        if (wi <= w) {
            // item i fits, so could be part of the solution
            if (vi + B[i - 1, w - wi] > B[i - 1, w])
                B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
            else
                B[i, w] = B[i - 1, w] // value didn't change with the new item
        }
        else {
            // item i does not fit, so would not be part of the solution
            B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
        }
    }
}
Dynamic Programming: 0-1 Knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

<table>
<thead>
<tr>
<th>I / w</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></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for i = 1 to n { // adding each item incrementally
    for w = 0 to W { // increasing the weight incrementally
        if (wi <= w) {
            // item i fits, so could be part of the solution
            if (vi + B[i - 1, w - wi] > B[i - 1, w])
                B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
            else
                B[i, w] = B[i - 1, w] // value didn't change with the new item
        }
        else {
            // item i does not fit, so would not be part of the solution
            B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
        }
    }
}
for i = 1 to n { // adding each item incrementally
    for w = 0 to W { // increasing the weight incrementally
        if (wi <= w) {
            // item i fits, so could be part of the solution
            if (vi + B[i - 1, w - wi] > B[i - 1, w])
                B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
            else
                B[i, w] = B[i - 1, w] // value didn't change with the new item
        }
        else {
            // item i does not fit, so would not be part of the solution
            B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
        }
    }
}
Dynamic Programming: 0-1 Knapsack

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

<table>
<thead>
<tr>
<th>I/w</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>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

for i = 1 to n {  // adding each item incrementally
    for w = 0 to W {  // increasing the weight incrementally
        if (wi <= w) {
            // item i fits, so could be part of the solution
            if (vi + B[i - 1, w - wi] > B[i - 1, w])
                B[i, w] = vi + B[i - 1, w - wi]  // value changed with the new item
            else
                B[i, w] = B[i - 1, w]  // value didn't change with the new item
        }
        else {
            // item i does not fit, so would not be part of the solution
            B[i, w] = B[i - 1, w]  // value won't change, since we didn't add a new item
        }
    }
}
Dynamic Programming: 0-1 Knapsack

### Items:
1: (2,3)  
2: (3,4)  
3: (4,5)  
4: (5,6)

<table>
<thead>
<tr>
<th>$I/w$</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>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

```plaintext
for i = 1 to n { // adding each item incrementally
    for w = 0 to W { // increasing weight incrementally
        if (wi <= w) {
            // item i fits, so could be part of the solution
            if (vi + B[i - 1, w - wi] > B[i - 1, w])
                B[i, w] = vi + B[i - 1, w - wi] // value changed with the new item
            else
                B[i, w] = B[i - 1, w] // value didn't change with the new item
        }
        else {
            // item i does not fit, so would not be part of the solution
            B[i, w] = B[i - 1, w] // value won't change, since we didn't add a new item
        }
    }
}
```
Dynamic Programming: 0-1 Knapsack

• This algorithm will find the maximum value possible in the knapsack.

• You’ll have to track back through the table to figure out the actual items – homework if you’re curious.
• **Backtracking**: solves problems with a large search space, by (recursively) trying every alternative, and choosing the best one.

• **Divide and conquer**: split each problem instance into two or more smaller parts, solve those, and recombine the results.

• **Greedy**: always make the best choice at the moment, and ignore future consequences.

• **Dynamic programming**: break up a problem into a collection of overlapping sub-problems, and build up solutions to larger and larger (optimal) sub-problems.
Summary

- **Code Design Patterns:**
  - **Creational**
    - They abstract the instantiation process.
    - Make systems independent on how objects are compared, created and represented.
  - **Structural**
    - Focus on how classes and objects are composed to form (relatively) large structures.
    - Generally use inheritance.
  - **Behavioral**
    - Describe how different objects work together.
    - Focus on
      - The algorithms and assignment of responsibilities among objects.
      - The communication and interconnection between objects.

- **Algorithm Design Patterns:** *backtracking, divide and conquer, greedy, dynamic programming*, others, not discussed.