Dynamic Programming: Knapsack Problem

Data Structures and Algorithms
Emory University
Jinho D. Choi
Knapsack Problem
Knapsack Problem

- Given a **knapsack** that can hold up to a certain **weight**, pack as many **items** as you can that give the maximum **value**.
Knapsack Problem

- Given a **knapsack** that can hold up to a certain **weight**, pack as many **items** as you can that give the maximum **value**.
Knapsack Problem

• Given a knapsack that can hold up to a certain weight, pack as many items as you can that give the maximum value.
Knapsack Problem

- Given a knapsack that can hold up to a certain weight, pack as many items as you can that give the maximum value.

<table>
<thead>
<tr>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1500</td>
<td>1.5kg</td>
</tr>
<tr>
<td>$2400</td>
<td>2kg</td>
</tr>
<tr>
<td>$63</td>
<td>1.3kg</td>
</tr>
<tr>
<td>$150</td>
<td>1.3kg</td>
</tr>
</tbody>
</table>

Knapsack Problem

- Given a knapsack that can hold up to a certain weight, pack as many items as you can that give the maximum value.

<table>
<thead>
<tr>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1500</td>
<td>1.5kg</td>
</tr>
<tr>
<td>$2400</td>
<td>2kg</td>
</tr>
<tr>
<td>$63</td>
<td>1.3kg</td>
</tr>
<tr>
<td>$150</td>
<td>1.3kg</td>
</tr>
</tbody>
</table>
Knapsack Item

```java
public class KnapsackItem implements Comparable<KnapsackItem>
{
    private int i_weight;
    private int i_value;

    public KnapsackItem(int weight, int value)
    {
        set(weight, value);
    }

    ...}
```
public class KnapsackItem implements Comparable<KnapsackItem> {
    private int i_weight;
    private int i_value;

    public KnapsackItem(int weight, int value) {
        set(weight, value);
    }

    ...
Knapsack Item

```java
public class KnapsackItem implements Comparable<KnapsackItem> {
    private int i_weight;
    private int i_value;

    public KnapsackItem(int weight, int value) {
        set(weight, value);
    }

    @Override
    public int compareTo(KnapsackItem o) {
        return i_weight - o.i_weight;
    }
}
```
Abstract Knapsack

```java
public abstract class AbstractKnapsack {
    public abstract List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight);

    protected int getTotalValue(Collection<KnapsackItem> items) {
        int total = 0;

        for (KnapsackItem item : items)
            total += item.getValue();

        return total;
    }
}
```
Abstract Knapsack

```java
public abstract class AbstractKnapsack {
    public abstract List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight);

    protected int getTotalValue(Collection<KnapsackItem> items) {
        int total = 0;
        for (KnapsackItem item : items) {
            total += item.getValue();
        }
        return total;
    }
}
```
Knapsack - Recursive
Knapsack - Recursive

```java
public List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight)
{
    Arrays.sort(items);
    return solve(items, maxWeight, items.length-1);
}
```
public List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight) {
    Arrays.sort(items);
    return solve(items, maxWeight, items.length - 1);
}
public List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight)
{
    Arrays.sort(items);
    return solve(items, maxWeight, items.length-1);
}

private List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight, int index)
{
    if (index < 0 || maxWeight == 0) return new ArrayList<>();
    KnapsackItem item = items[index];
    if (item.getWeight() > maxWeight)
        return solve(items, maxWeight, index-1);
    else
    {
        List<KnapsackItem> with = solve(items, maxWeight-item.getWeight(), index-1);
        List<KnapsackItem> without = solve(items, maxWeight, index-1);
        with.add(item);
        return (getTotalValue(with) > getTotalValue(without)) ? with : without;
    }
}
public List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight)
{
    Arrays.sort(items);
    return solve(items, maxWeight, items.length-1);
}

private List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight, int index)
{
    if (index < 0 || maxWeight == 0) return new ArrayList<>();
    KnapsackItem item = items[index];

    if (item.getWeight() > maxWeight)
        return solve(items, maxWeight, index-1);
    else
    {
        List<KnapsackItem> with = solve(items, maxWeight-item.getWeight(), index-1);
        List<KnapsackItem> without = solve(items, maxWeight, index-1);
        with.add(item);
        return getTotalValue(with) > getTotalValue(without) ? with : without;
    }
}
Knapsack - Recursive

```java
public List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight)
{
    Arrays.sort(items);
    return solve(items, maxWeight, items.length-1);
}

private List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight, int index)
{
    if (index < 0 || maxWeight == 0) return new ArrayList<>();
    KnapsackItem item = items[index];

    if (item.getWeight() > maxWeight)
        return solve(items, maxWeight, index-1);
    else
    {
        List<KnapsackItem> with = solve(items, maxWeight-item.getWeight(), index-1);
        List<KnapsackItem> without = solve(items, maxWeight, index-1);
        with.add(item);
        return (getTotalValue(with) > getTotalValue(without)) ? with : without;
    }
}
```

Knapsack - Recursive

```java
public List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight) {
    Arrays.sort(items);
    return solve(items, maxWeight, items.length - 1);
}

private List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight, int index) {
    if (index < 0 || maxWeight == 0) return new ArrayList<>();
    KnapsackItem item = items[index];

    if (item.getWeight() > maxWeight)
        return solve(items, maxWeight, index - 1);
    else {
        List<KnapsackItem> with = solve(items, maxWeight - item.getWeight(), index - 1);
        List<KnapsackItem> without = solve(items, maxWeight, index - 1);
        with.add(item);
        return getTotalValue(with) > getTotalValue(without) ? with : without;
    }
}
```
Knapsack - Dynamic Programming

private List<KnapsackItem> solveAux(KnapsackItem[] items, int maxWeight,
                                      int index, Map<String,List<KnapsackItem>> map)
{
    String key = maxWeight+" "+index;
    List<KnapsackItem> list = map.get(key);

    if (list == null)
    {
        list = solve(items, maxWeight, index, map);
        map.put(key, list);
    }

    return new ArrayList<>(list);
}
Knapsack - Dynamic Programming

private List<KnapsackItem> solveAux(KnapsackItem[] items, int maxWeight, int index, Map<String, List<KnapsackItem>> map) {
    String key = maxWeight + " " + index;
    List<KnapsackItem> list = map.get(key);

    if (list == null) {
        list = solve(items, maxWeight, index, map);
        map.put(key, list);
    }

    return new ArrayList<>(list);
}
Knapsack - Dynamic Programming

```java
private List<KnapsackItem> solveAux(KnapsackItem[] items, int maxWeight, int index, Map<String,List<KnapsackItem>> map) {
    String key = maxWeight + index;
    List<KnapsackItem> list = map.get(key);

    if (list == null) {
        list = solve(items, maxWeight, index, map);
        map.put(key, list);
    }

    return new ArrayList<>(list);
}
```
Knapsack - Dynamic Programming

private List<KnapsackItem> solveAux(KnapsackItem[] items, int maxWeight, int index, Map<String, List<KnapsackItem>> map) {
    String key = maxWeight + " " + index;
    List<KnapsackItem> list = map.get(key);

    if (list == null) {
        list = solve(items, maxWeight, index, map);
        map.put(key, list);
    }

    return new ArrayList<>(list);
}
private List<KnapsackItem> solveAux(KnapsackItem[] items, int maxWeight, int index, Map<String, List<KnapsackItem>> map) {
    String key = maxWeight + " " + index;
    List<KnapsackItem> list = map.get(key);

    if (list == null) {
        list = solve(items, maxWeight, index, map);
        map.put(key, list);
    }

    return new ArrayList<>(list);
}
private List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight, int index
, Map<String,List<KnapsackItem>> map)
{
    if (index < 0 || maxWeight == 0) return new ArrayList<>();
    KnapsackItem item = items[index--];

    if (item.getWeight() > maxWeight)
        return solveAux(items, maxWeight, index, map);
    else
    {
        List<KnapsackItem> with = solveAux(items, maxWeight-item.getWeight(), index, map);
        List<KnapsackItem> without = solveAux(items, maxWeight, index, map);
        with.add(item);
        return (getTotalValue(with) > getTotalValue(without)) ? with : without;
    }
}
Knapsack - Dynamic Programming

```java
private List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight, int index)
{
    if (index < 0 || maxWeight == 0) return new ArrayList<>();
    KnapsackItem item = items[index--];

    if (item.getWeight() > maxWeight)
        return solveAux(items, maxWeight, index--);
    else
    {
        List<KnapsackItem> with = solveAux(items, maxWeight-item.getWeight(), index);
        List<KnapsackItem> without = solveAux(items, maxWeight, index);
        with.add(item);
        return (getTotalValue(with) > getTotalValue(without)) ? with : without;
    }
}
```
Knapsack - Dynamic Programming

```java
private List<KnapsackItem> solve(KnapsackItem[] items, int maxWeight, int index)
{
    if (index < 0 || maxWeight == 0) return new ArrayList<>();
    KnapsackItem item = items[index--];

    if (item.getWeight() > maxWeight)
        return solve(items, maxWeight, index);
    else
    {
        List<KnapsackItem> with = solve(items, maxWeight - item.getWeight(), index);
        List<KnapsackItem> without = solve(items, maxWeight, index);
        with.add(item);
        return getTotalValue(with) > getTotalValue(without)) ? with : without;
    }
}
```
Recursion vs. Dynamic Programming
Recursion vs. Dynamic Programming

- When should I use dynamic programming?
Recursion vs. Dynamic Programming

- When should I use dynamic programming?
  - When there are many redundant processes in recursive calls.
Recursion vs. Dynamic Programming

• When should I use dynamic programming?
  - When there are many redundant processes in recursive calls.

• How can I turn recursion into dynamic programming?
Recursion vs. Dynamic Programming

- When should I use **dynamic programming**?
  - When there are many redundant processes in **recursive calls**.

- How can I turn **recursion** into **dynamic programming**?
  - Create a (or more) **table** that stores necessary **states**.
Recursion vs. Dynamic Programming

• When should I use dynamic programming?
  - When there are many redundant processes in recursive calls.

• How can I turn recursion into dynamic programming?
  - Create a (or more) table that stores necessary states.

• How is the dynamic table populated?
Recursion vs. Dynamic Programming

- When should I use **dynamic programming**?
  - When there are many redundant processes in **recursive calls**.

- How can I turn **recursion** into **dynamic programming**?
  - Create a (or more) **table** that stores necessary **states**.

- How is the **dynamic table** populated?
  - By saving previously found **states** during **recursive calls**.