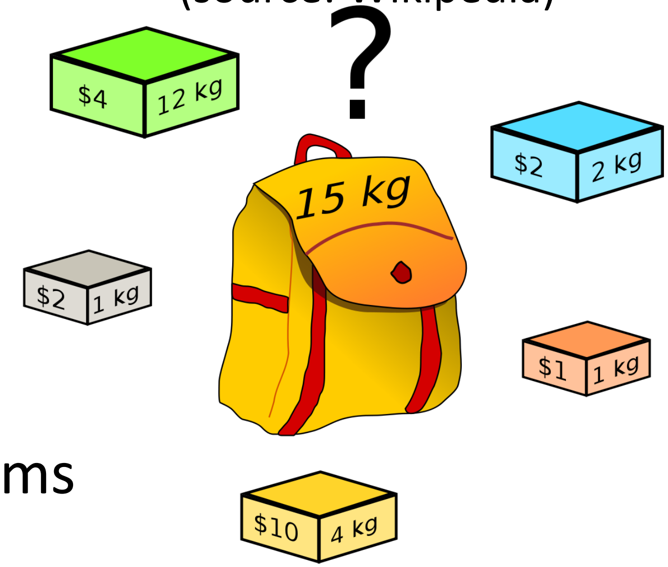
Dynamic Programming

a. Fibonacci Series
b. Weighted Interval Scheduling
c. Knapsack

The Knapsack Problem

(source: Wikipedia)



- **Input:** n items for your knapsack
  - value v_i and a weight $w_i \in \mathbb{N}$ for n items
  - capacity of your knapsack $T \in \mathbb{N}$
- **Output:** the most valuable subset of items that fits in the knapsack
  - Subset $S \subseteq \{1, \ldots, n\}$
  - Value $V_S = \sum_{i \in S} v_i$ as large as possible
  - Weight $W_S = \sum_{i \in S} w_i$ at most T
- **Want:** $\mathbf{argmax}_{S \subseteq \{1,\ldots,n\}} V_S$ s.t. $W_S \leq T$
- **(SubsetSum:** $v_i = w_i$,
- **TugOfWar:** $v_i = w_i, T = \frac{1}{2} \sum_i v_i$)

$n =$ $T =$

$v_1 =$ $w_1 =$

$v_2 =$ $w_2 =$

$v_3 =$ $w_3 =$

$v_4 =$ $w_4 =$

$v_5 =$ $w_5 =$

Do we really need DP?

Items with large $\frac{v_i}{w_i}$ seem like good choices…

Ex. *T = 8,* $(v_1 = 6, w_1 \ = 5), (v_2 = 4, w_2 \ = 4), (v_3 = 4, w_3 \ = 4)$

- Strategy 1: Repeatedly pick items that fit with largest $\frac{v_i}{w_i}$

- Is this optimal?

Knapsack – what to do with n-th item?

Want: $\mathbf{argmax}_{S \subseteq \{1,\ldots,n\}} V_S$ s.t. $W_S \leq T$

Knapsack - subproblems

- Let $O_n \subseteq \{1, \dots, n\}$ be the **optimal** subset of items given the first *n* items
- **Case 1:** $\text{n} \notin O_n$

  O_n =

- **Case 2:** $n \in O_n$

  O_n =

Knapsack - recurrence

- Let $\mathbf{OPT}(\boldsymbol{j}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \dots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$
- **Case 2:** $j \in O_{j,S}$

Knapsack - recurrence

- Let $\mathbf{OPT}(\boldsymbol{j}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \ldots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$
  - $OPT(j, S) = \ OPT(j-1, S)$
- **Case 2:** $j \in O_{j,S}$
  - $OPT(j, S) = v_j + \ OPT(j-1, S - w_j)$

Recurrence:

$\mathrm{OPT}(j, S) =$

Base Cases:

$\mathrm{OPT}(j, 0) =$

$\mathrm{OPT}(0, S) =$

Knapsack - recurrence

- Let $\mathbf{OPT}(\boldsymbol{j}, \boldsymbol{S})$ be the **value** of the optimal subset of items $\{1, \dots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$
  - $OPT(j, S) = OPT(j-1, S)$
- **Case 2:** $j \in O_{j,S}$
  - $OPT(j, S) = v_j + OPT(j-1, S-w_j)$

Recurrence:

$$\mathrm{OPT}(j, S) = \begin{cases} \max\{OPT(j-1, S), v_j + OPT(j-1, S-w_j)\} & S \geq w_j \\ OPT(j-1, S) & S < w_j \end{cases}$$

Base Cases:

$$\mathrm{OPT}(j, 0) = \mathrm{OPT}(0, S) = 0$$

Knapsack ("Bottom-Up")

```
// All inputs are global vars
FindOPT(n,T):
  M[0,S] ← 0, M[j,0] ← 0

  for (j = 1,…,n):
    for (S = 1,…,T):
      if (w_j > S): M[j,S] ← M[j-1,S]
      else: M[j,S] ← max{M[j-1,S],v_j + M[j-1,S-w_j]}

  return M[n,T]
```

Ask the Audience

Space:

- Input: $T = 8, n = 3$
  - $w_1 = 2\ , v_1 = 4$
  - $w_2 = 3\ , v_2 = 5$
  - $w_3 = 5\ , v_3 = 8$

items

3									
2									
1									
0									
-	0	1	2	3	4	5	6	7	8

capacities

$$\mathrm{OPT}(j,S) = \begin{cases} \max\{OPT(j-1,S), v_j + OPT(j-1,S-w_j)\} & \text{If } S \geq w_j \\ OPT(j-1,S) & \text{If } S < w_j \end{cases}$$

Filling the Knapsack

- Let $\boldsymbol{O_{j,S}}$ be the **optimal subset of items** $\{1, \ldots, j\}$ in a knapsack of size S
- **Case 1:** $j \notin O_{j,S}$
  - Use opt. solution for items 1 to j-1 in a knapsack of size S
- **Case 2:** $j \in O_{j,S}$
  - Use j + opt. solution for items 1 to j-1 in a knapsack of size $S - w_j$

Filling the Knapsack

```
// All inputs are global vars
// M[0:n,0:T] contains solutions to subproblems
FindSol(M,n,T):
  if (n = 0 or T = 0): return Ø
  else:
    if (wn > T): return FindSol(M,n-1,T)
    else:
      if (M[n-1,T] > vn + M[n-1,T-wn]):
        return FindSol(M,n-1,T)
      else:
        return {n} + FindSol(M,n-1,T-wn)
```

Knapsack Wrapup

- Can solve knapsack problems in time/space $O(nT)$
  - **Recipe:**

    (1) identify a set of **subproblems**

    (2) relate the subproblems via a **recurrence**

    (3) find an **efficient implementation** of the recurrence (top down or bottom up)

    (4) **reconstruct the solution** from the DP table

Dynamic Programming

a. Fibonacci Series
b. Weighted Interval Scheduling
c. Knapsack
d. Longest Common Subsequence

Common Subsequences

- Given a string $x \in \Sigma^n$ a **subsequence** is any string obtained by deleting a subset of the symbols

r	e	c	u	r	a	n	c	e

- Given two strings $x \in \Sigma^n, y \in \Sigma^m$, a **common subsequence** is a **subsequence** of both x and y

r	e	c	u	r	a	n	c	e

r	e	c	u	r	r	e	n	c	e

Longest Common Subsequence (LCS)

- **Input:** Two strings $x \in \Sigma^n, y \in \Sigma^m$
- **Output:** The longest common subsequence of x and y

Writing the Recurrence

- Consider the **LCS** of x, y
- **Question:** Are the last symbols of x and y in the subsequence?
- **Observation:** Suppose $x_n = y_m$
  - Then these symbols are always part of some LCS
  - **Ask the Audience:** Why?

Writing the Recurrence

- Consider the **LCS** of x, y
- **Question:** Are the last symbols of x and y in the subsequence?
- **Observation:** Suppose $x_n \neq y_m$
  - **Case 1:** x_n is **not** in the LCS
  - **Case 2:** y_m is **not** in the LCS
  - **Case 3:** Neither is in the LCS

Writing the Recurrence

- $\text{LCS}(i, j)$ = Length of LCS of $x_{1:i}$ and $y_{1:j}$
- **Equal:** If $x_i = y_j$ then
- **Not Equal:**
    - **Case 1:** x_i is **not** in the LCS
    - **Case 2:** y_j is **not** in the LCS

Recurrence:

Base Cases:

Writing the Recurrence

Recurrence:

$$\mathrm{LCS}(i,j) = \begin{cases} 1 + \mathrm{LCS}(i-1, j-1) & \text{if } x_i = y_j \\ \max\{\mathrm{LCS}(i-1, j), \mathrm{LCS}(i, j-1)\} & \text{if } x_i \neq y_j \end{cases}$$

Base Cases:

$\mathrm{LCS}(i,0) = 0$, $\mathrm{LCS}(0,j) = 0$

Solving the Recurrence: Bottom-Up

```
// All inputs are global vars
FindOPT(n,m):
  M[i,0] ← 0,    M[0,j] ← 0

  for (i= 1,…,n):
    for (j = 1,…,m):
      if (x_i = y_j):
        M[i,j] ← 1 + M[i-1,j-1]
      else:
        M[i,j] ← max{M[i-1,j],M[i,j-1]}

  return M[n,m]
```

Ask the Audience

x = peat

y = leapt

Compute LCS(i,j) for each subproblem

		j = 0	1	2	2	4	5
		-	l	e	a	p	t
i = 0	-	**0**	**0**	**0**	**0**	**0**	**0**
1	p	**0**					
2	e	**0**					
3	a	**0**					
4	t	**0**					

Ask the Audience

x = peat

y = leapt

Compute LCS(i,j) for each subproblem

		j = 0	1	2	2	4	5
		-	l	e	a	p	t
i = 0	-	**0**	**0**	**0**	**0**	**0**	**0**
1	p	**0**	**0**	**0**	**0**	**1**	**1**
2	e	**0**	**0**	**1**	**1**	**1**	**1**
3	a	**0**	**0**	**1**	**2**	**2**	**2**
4	t	**0**	**0**	**1**	**2**	**2**	**3**

Finding the Solution

```
// All inputs are global vars
FindLCS(i,j):
  if (i = 0 or j = 0)
    return ""
  if (x_i = y_j):
    return FindLCS(i-1,j-1)+ x_i
  else:
    if (M[i-1,j] > M[i,j-1])
      return FindLCS(i-1,j)
    else:
      return FindLCS(i,j-1)
  return M[n,m]
```

Summary

- Compute the **longest common subsequence** between two strings of length n and m in time $O(nm)$

- Dynamic Programming:
  - Question: Which of the final letters are part of the LCS?

- **Ask the Audience:** How do we recover the LCS itself from the values $\mathrm{LCS}(\mathrm{i},\mathrm{j})$