Dynamic Programming

a. Fibonacci Series
b. Weighted Interval Scheduling
c. Knapsack
d. Longest Common Subsequence
e. Longest Increasing Subsequence
f. Edit Distance

Edit Distance

- **Input:** two string A[1...n] and B[1...m]
- **Output:** minimum number of letter insertions, letter deletions, and letter substitutions required to transform one string into the other.

$$\underline{\mathrm{F}}\mathrm{OOD} \longrightarrow \mathrm{MO}\underline{\mathrm{O}}\mathrm{D} \longrightarrow \mathrm{MON}_{\wedge}\mathrm{D} \longrightarrow \mathrm{MONE}\underline{\mathrm{D}} \longrightarrow \mathrm{MONEY}$$

Edit Distance

- We can visualize this editing process by aligning the strings one above the other:
  - a gap in the first word for each insertion
  - a gap in the second word for each deletion
  - columns with two *different* characters correspond to substitutions

FOOD → MOOD → MON^D → MONED → MONEY

F	O	O		D
M	O	N	E	Y

A	L	G	O	R		I		T	H	M
A	L		T	R	U	I	S	T	I	C

Writing the Recurrence

- Let $\mathrm{Edit}(i, j)$ be the minimum number of edits to turn A[1...i] into B[1...j].
- Consider the last column of the visualization:

- **Case 1:** the last element in the top row is empty
  - This is an **insertion** to the first string.
  - In this case $\mathrm{Edit}(i, j) = \mathrm{Edit}(i, j-1) + 1$.

Writing the Recurrence

- **Case 2:** the last element in the bottom row is empty
  - This is a **deletion** from the first string.
  - In this case $\text{Edit}(i, j) = \text{Edit}(i - 1, j) + 1$.

Writing the Recurrence

- **Case 3:** both rows have characters in the last column
  - If the last characters are the same, then
$$\mathrm{Edit}(i, j) = \mathrm{Edit}(i-1, j-1)$$
  - If the last characters are different, then we need **substitution:**
$$\mathrm{Edit}(i, j) = \mathrm{Edit}(i-1, j-1) + 1$$

Writing the Recurrence

- **Base Case:**
  - $\text{Edit}(i, 0) = i$
  - $\text{Edit}(0, j) = \text{j}$

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

Compute Edit (i,j) for each subproblem of x=peat and y=leapt

- **Base Case:**
  - $\text{Edit}(i, 0) = i$
  - $\text{Edit}(0, j) = \text{j}$

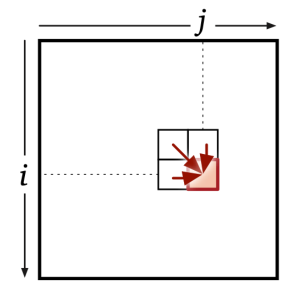
$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

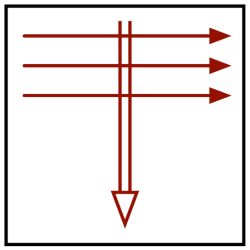
		j = 0	1	2	3	4	5
		-	l	e	a	p	t
i = 0	-						
1	p						
2	e						
3	a						
4	t						

```
EditDistance(A[1..m], B[1..n]):
    for j ← 0 to n
        Edit[0, j] ← j

    for i ← 1 to m
        Edit[i, 0] ← i
        for j ← 1 to n
            ins ← Edit[i, j − 1] + 1
            del ← Edit[i − 1, j] + 1
            if A[i] = B[j]
                rep ← Edit[i − 1, j − 1]
            else
                rep ← Edit[i − 1, j − 1] + 1
            Edit[i, j] ← min {ins, del, rep}

    return Edit[m, n]
```





Dynamic Programming

a. Fibonacci Series
b. Weighted Interval Scheduling
c. Knapsack
d. Longest Common Subsequence
e. Longest Increasing Subsequence
f. Edit Distance
g. Wrap Up

Dynamic Programming Recipe

- **Recipe:**

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

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

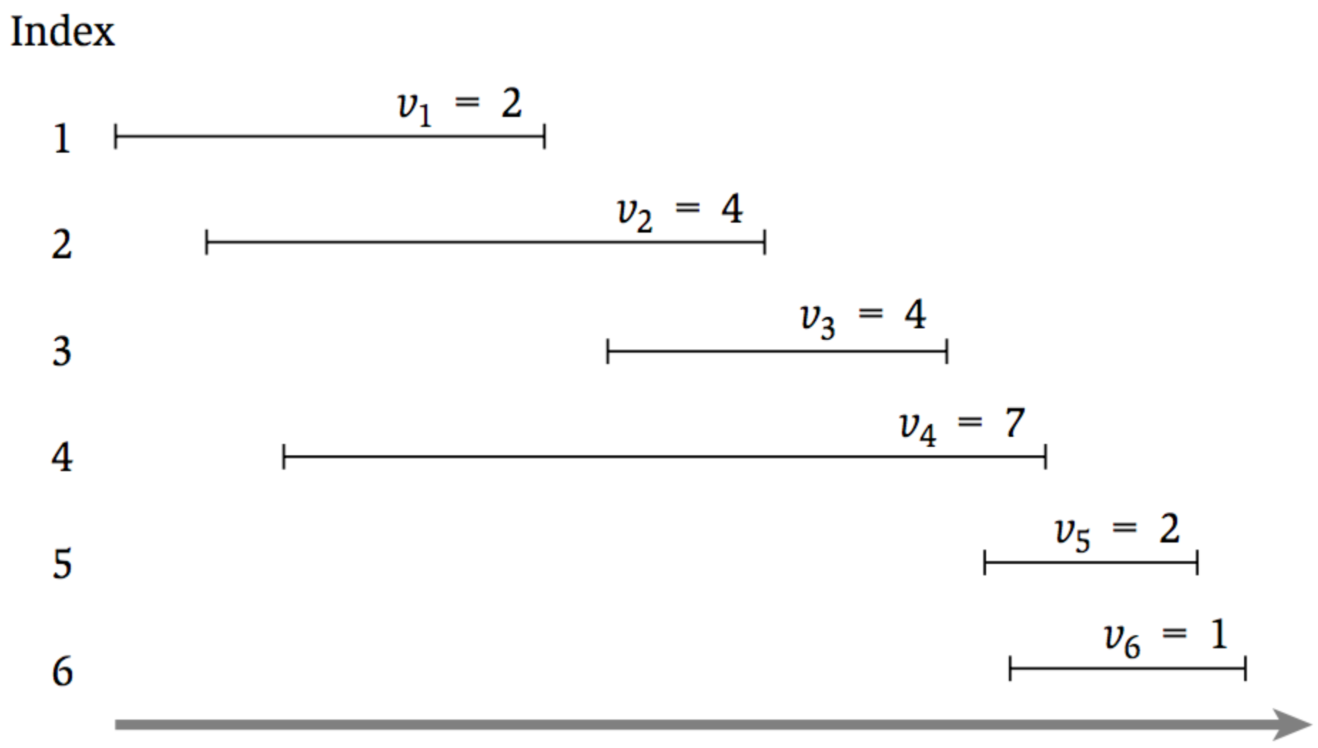
  (3) find an **efficient implementation** of the recurrence (top down or bottom up)

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

Interval Scheduling

- **Input:** n intervals (s_i, f_i) each with value v_i
  - Assume intervals are sorted so $f_1 < f_2 < \cdots < f_n$
- **Output:** a compatible schedule S **maximizing** the total value of all intervals
  - A **schedule** is a subset of intervals $S \subseteq \{1, \ldots, n\}$
  - A schedule S is **compatible** if no $i, j \in S$ overlap
  - The **total value** of S is $\sum_{i \in S} v_i$

Interval Scheduling



Subproblems

- ***Subproblems:*** Let O_i be the **optimal schedule** using only the intervals $\{1, \ldots, i\}$

Relating the Subproblems

- ***Subproblems:*** Let O_i be the **optimal schedule** using only the intervals $\{1, \ldots, i\}$
- **Case 1:** Final interval is not in O_i ($i \notin O_i$)
  - Then O_i must be the optimal solution for $\{1, \ldots, i-1\}$
    - $O_i = O_{i-1}$
- **Case 2:** Final interval is in O_i ($i \in O_i$)
  - Assume intervals are sorted so that $f_1 < f_2 < \cdots < f_n$
  - Let $p(i)$ be the largest j such that $f_j < s_i$
  - Then O_i must be i + the optimal solution for $\{1, \ldots, p(i)\}$
    - $O_i = \{i\} + O_{p(i)}$

A Recursive Formulation

- **Subproblems:** Let $OPT(i)$ be the ***value*** **of the optimal schedule** using only the intervals $\{1, \dots, i\}$

- $OPT(i) = \max\{OPT(i-1), v_i + OPT(p(i))\}$

- $OPT(0) = 0, OPT(1) = v_1$

Top-down Recipe

```
FindOpt(subproblem s):

  if (s is a base case):
    1-Find the solution directly with no recursion
    2-Return the solution.

  if you already have the solution memorized:
    1-Return the solution.

  else:
    1-Identify the subproblems needed for solving s.
    2-Recursively call FindOpt on these subprobelms.
    3-Solve s using these results.
    4-Store the solution for s in an array.(memorize)
    5-Return the solution.
```

Buttom-up Recipe

```
FindOpt():
  Let M be an array for storing the values of the
  optimal solutions for all the subproblems.

  Initialize M with the value for the base cases.

  Iterate over subproblems starting from the smallest:
    1- Find the value for the subproblems using the
       recursive formula and the value of the smaller
       subproblems stored in M.
    2-Store the value in array M.

  Return the solution based on M.
```

Dynamic Programming

a. Fibonacci Series
b. Weighted Interval Scheduling
c. Knapsack
d. Longest Common Subsequence
e. Longest Increasing Subsequence
f. Edit Distance
g. Wrap Up
h. Quiz 1 Review

Question 1

2 pts

Consider the following algorithm.

Let M be a global array of size n+1

```
FindOPT(n):
   If (n <= 1):
      Return 0
   Else
      M[n] = max{1+FindOPT(n-1), FindOPT(n-2)}
      Return M[n]
```

Which of the following best describes the algorithm being used here?

- ◯ This is a bottom up dynamic program running in O(n) time
- ◯ This is not a dynamic program and takes exponential in n time.
- ◯ This is a top down dynamic program running in O(n) time

Question 2

3 pts

This problem will test your understanding of dynamic programming by having you run through the algorithm for weighted interval scheduling that we saw in class. Consider the following input for the interval scheduling problem:

Interval 1: (0, 2), so $s1 = 0$, $f1 = 2$; value $v1 = 3$

Interval 2: (1, 4), $v2 = 4$

Interval 3: (3, 6), $v3 = 3$

Interval 4: (5, 10), $v4 = 6$

Interval 5: (9, 12), $v5 = 3$

What is the value of the optimal schedule where Interval 5 is **not** included? 10

What is the value of the optimal schedule where Interval 5 is included? 9

Question 3

2 pts

What is the solution to the following recurrence?

$$T(n) = T(n-1) + 4,\ T(1) = 1$$

- $\Theta(n \log n)$
- $\Theta(n)$
- $\Theta(n^2)$
- $\Theta(n^4)$

Question 4

3 pts

Which function grows fastest than the others?

$$f_1(n) = \sqrt{n}, \quad f_2(n) = (n!)^{0.01}, \quad f_3(n) = 99^n, \; f_4(n) = 2^{\log^3 n}$$

- f4
- f3
- f1
- f2