Lecture 11

Announcements:

1. Today is the last day to submit regrade requests for Exam 1.
2. Homework 4 will be up today and will be due Feb 22nd at 6:59 PM.
3. Quiz 6 will go live today at 7PM and will be open until Friday 10PM.
4. Homework 3 is due today at 6:59PM.

Last class:

Weighted Interval Scheduling problem:

(s_1, f_1) v_1

⋮

(s_n, f_n) v_n

Goal: Find the subset of non-conflicting jobs with most value.

Iterative-Compute-Opt

① Initialize $M[0] = 0$

② Order jobs by finish time

③ Compute $p(j)$ for each j.

④ For $j = 1, \ldots n$

$$M[j] = \max\left(v_j + M[p(j)],\ M[j-1]\right)$$

⑤ Output $M[n]$

} → ?? "Dynamic Programming".

Last class: What I wrote down for computing $p(j)$ doesn't quite work.

Iterative-Compute-Opt

① Initialize $M[0] = 0$

② Order jobs by finish time

③ Compute $p(j)$ for each j.

④ For $j = 1, \ldots n$

$$M[j] = \max\left(v_j + M[p(j)],\ M[j-1]\right)$$

⑤ Output $M[n]$

→ ?? "Dynamic Programming".

If $j \in$ the optimal for $\{1, \ldots, j\}$

then

How to compute the best subset?

(Assuming we already have the OPT costs.)

If for example,

$$v_n + M[p(n)] > M[n-1]$$

Then, n^{th} job is part of optimal subset.

You need to find optimal subset for $\{1, \ldots, p(n)\}$.

Else, $M[n-1] \geq v_n + M[p(n)]$

→ You just need to find the optimal subset for $\{1, \ldots, n-1\}$.

FindSubset(j) : (Computes an optimal value subset from {1, ..., j})

① If $j = 0$

Return $\emptyset$.

② Else If $v_j + M[P(j)] > M[j-1]$

Return $\{j\} + \text{FindSubset}(P(j))$

Else

Return $\text{FindSubset}(j-1)$.

Principles of Dynamic Programming

(1) Find some subproblems.
→ # subproblems should be small (polynomial number, ...)

(2) Can recover the solution to the original problem from solving the subproblems.

(3) There should be an "ordering" of the subproblems, so that we can recover the solution to a subproblem from solutions to "smaller" subproblems.

A recurrence relation.

Applications of Dynamic Programming

1. Bioinformatics
2. Compilers
3. Networks
4. Graphics
5. AI

Famous algorithms:

1. Unix "Diff"
2. CKY parsing algorithm
3. Smith-Waterman sequence alignment algorithm

.
.
.

Knapsack Problem

Knapsack with a weight limit 

Items	1	2	...	n
	v_1	v_2		v_n → values of
	w_1	w_2	...	w_n the items

Goal: Find a subset of items of most value that fits into the knapsack.

Find subset $S \subseteq [n]$ that maximizes $\sum_{i \in S} v_i$, among all subsets whose total weight $< W$.

→ The weights w_i, and the total weight W are all integers.

"No greedy algorithm works!"

Weights:	$W/2+1$	$W/2$	$W/2$
Value:	15	10	10

Let O be an optimal solution.

If $n \notin O$, solve the problem for items $\{1, \ldots, n-1\}$.

If $n \in O$, solve the problem for items $\{1, \ldots, n-1\}$ but the total available weight is W - w_n.

for example,

$OPT(j)$ = best value that we can get from items $\{1, \ldots, j\}$.

As this does not work, it makes sense to pass the weight also as an argument.

$OPT(j, w)$ = best value we can get from jobs $\{1, \dots, j\}$ when the total weight capacity is $\underline{\underline{w}}$.

$$= \max_{S \subseteq \{1, \dots j\}} \sum_{i \in S} v_i$$

where the max is over subsets with total weight at most w.

If $\omega_n > W$, then $OPT(n, W) = OPT(n-1, W)$

Else

$$OPT(n, W) = \max \begin{cases} OPT(n-1, W) \\ v_n + OPT(n-1, W - \omega_n) \end{cases}$$

By the same reasoning,

If $w_i > w$, then $OPT(i,w) = OPT(i-1, w)$

Else $OPT(i,w) = \max \begin{cases} OPT(i-1, w) \\ v_i + OPT(i-1, w-w_i) \end{cases}$

→ Initialization: $OPT(0, w) = 0$ $OPT(i,0) = 0$ $\forall i$
$\forall w = 1, \cdots, W.$

→ Order of evaluating subproblems: Small i to big i
Small w to big w.

Knapsack-Compute-Opt

① Set $OPT(0,w) = 0$ $\forall$ $w = 1, \ldots, W$. $OPT(i,0) = 0$ $\forall i = 1, \ldots, n$.

② For $i = 1, \ldots, n$

For $w = 1, 2, \ldots, W$ → (input)

Compute $OPT(i,w)$ using the recurrence! } takes $O(1)$ time.

③ Output $OPT(n, W)$.

Correctness: Induction using the recurrence.

Run-time: $O(n \cdot W)$.

```
FindSubset (i, w)
    If   w_i > w
         Return  FindSubset(i-1, w).
    Else     If   v_i + OPT(i-1, w-w_i) > OPT(i-1, w)
                  Return  {i} + FindSubset(i-1, w-w_i)
    Else
         Return  FindSubset(i-1, w).
```

Sequence Alignment or Edit distance Computation

How to Compare two strings?

S1 = D Y N A M I C

S2 = D N A M I C

If we Compare letters: # mismatches = 5.

But if we use any auto-spell program, it'll suggest S1 when entering S2.

D Y N A M I C

D — N A M I C

diff-score = 1 - blank !!

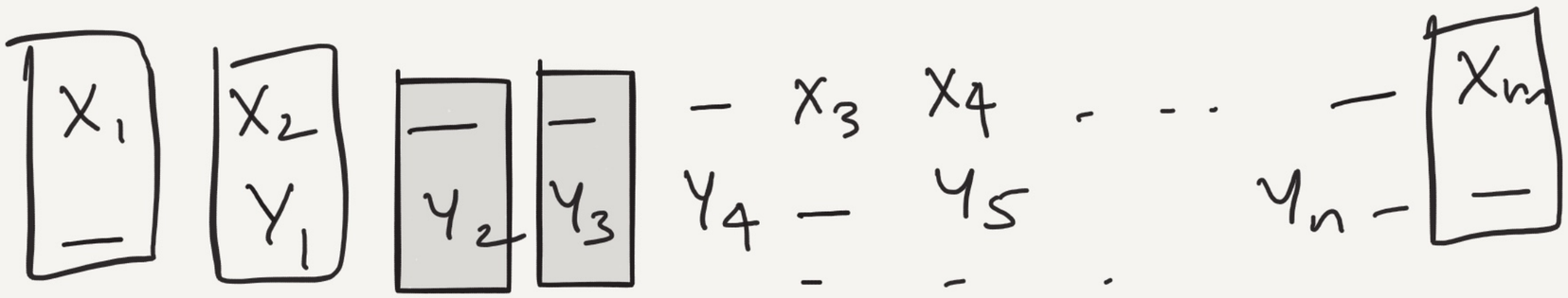
"Edit distance" Needleman & Munsch,
Levenstein

Strings $X = x_1\ x_2\ \ldots\ x_m$
$Y = y_1\ y_2\ -\ y_n$

} over some alphabet

eg: {A, C, G, T}.

Goal is to "align" them



→ to minimize the number of mismatched Columns.

D Y N A M I C
D — N A M I C

Cost = 1

D Y N A M I C
D N A M I C —

↓

Cost = 6

How to find the best alignment?

INPUT: $(x_1 \ldots x_m)$ & $(y_1, \ldots y_n)$

OUTPUT: An alignment of least possible cost.

$$\begin{matrix} x_1 & x_2 & x_3 & \cdots & - & - & \cdots & x_m \\ y_1 & y_2 & y_3 & \cdots & y_{i'} & - & \cdots & y_{n'} \end{matrix}$$

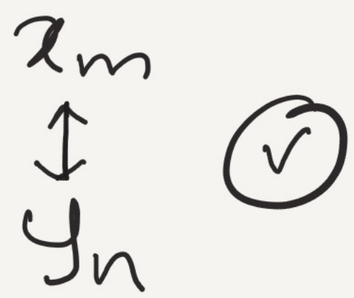
What can the last column look like in any alignment?

Lemma: The last column in any alignment is one of

Proof:

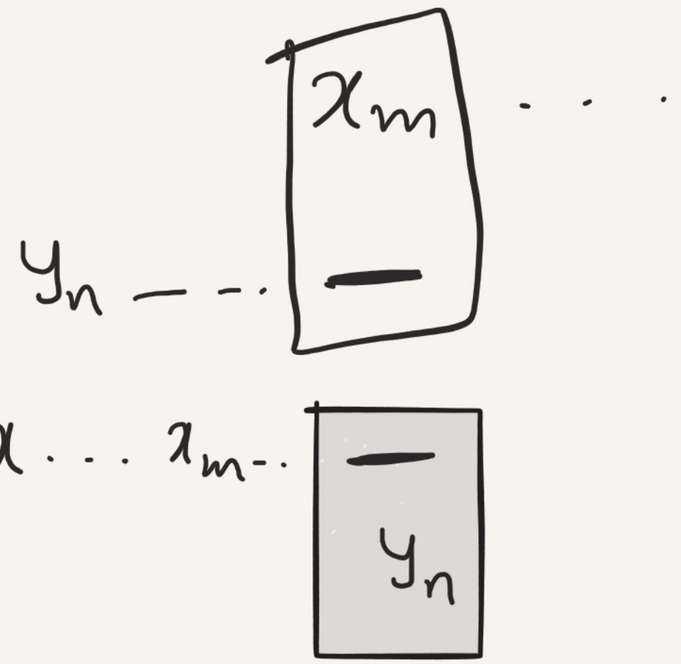
$x_1 \ldots$

$y_1 \ldots$



✓

if



We are trying to align

$x_1 \ldots x_m$

$y_1 \ldots y_n$

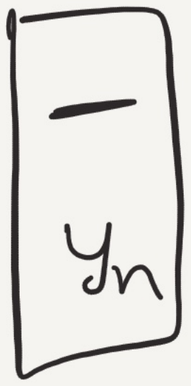
Ex: DYNAMICS
DYNAMIC-

Suppose we had an optimal alignment.
and the last column was

Case 1: $\begin{bmatrix} x_m \\ y_n \end{bmatrix}$ $\Rightarrow$ Need to find optimal alignment for $(x_1 \ldots x_{m-1})$ and $(y_1 \ldots y_{n-1})$.

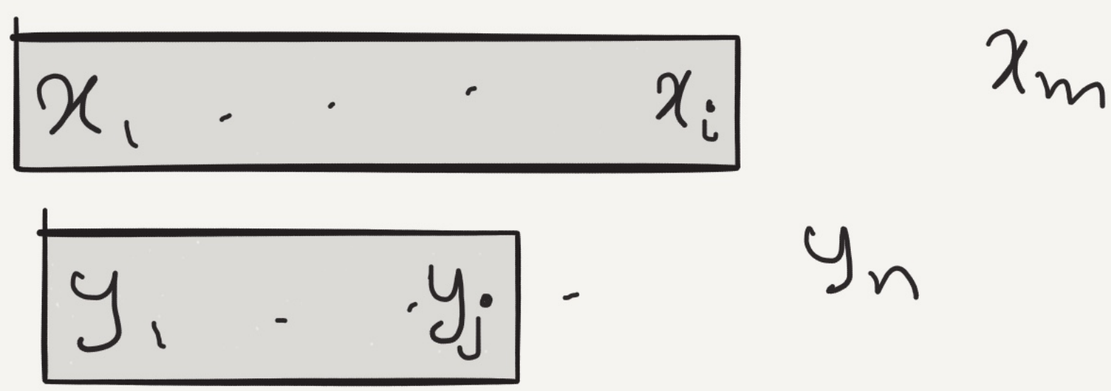
Case 2: $\begin{bmatrix} x_m \\ - \end{bmatrix}$ $\Rightarrow$ Need to find optimal alignmat for $(x_1 \ldots x_{m-1})$ and $(y_1 \ldots y_n)$

Case 3: If last column is



$\Rightarrow$ Need to find best alignment for $(x_1 \ldots x_m)$ and $(y_1 \ldots, y_{n-1})$.

What should the subproblems be ??



Let $OPT(i,j)$ = least cost of aligning $(x_1 \ldots x_i)$ and $(y_1, \ldots, y_j)$.

Recurrence Relation:

$\mathbb{1}(x_m \neq y_n)$ is 1 if $x_m \neq y_n$
0 else.

$$OPT(m,n) = \min \begin{cases} \mathbb{1}(x_m \neq y_n) + OPT(m-1, n-1) & \text{Case 1} \\ 1 + OPT(m-1, n). & \\ 1 + OPT(m, n-1). & \end{cases}$$

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities" (p. 159).