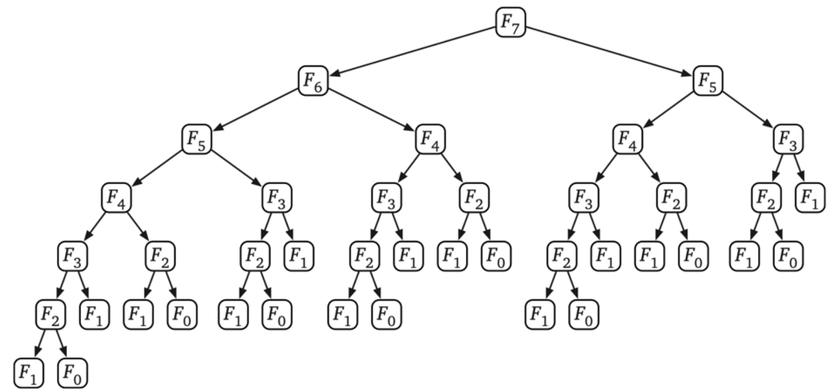
Dynamic Programming

Fibonacci Numbers

- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$
- $F(n) = F(n-1) + F(n-2)$



- Solving the recurrence recursively takes $\Omega(1.62^n)$ time
  - Problem: Recompute the same values $F(i)$ many times
- Two ways to improve the running time
  - Remember values you've already computed ("top down")
  - Iterate over all values $F(i)$ ("bottom up")

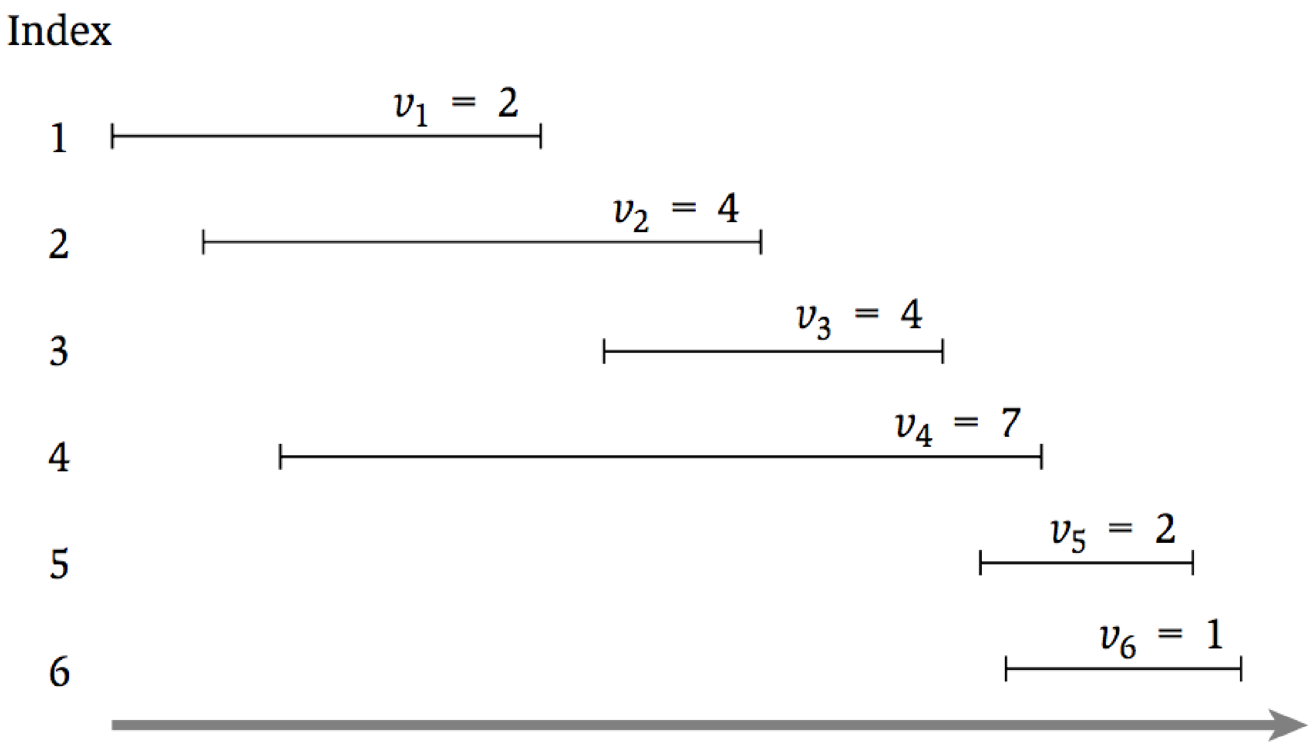
- **Fact:** Fastest algorithms solve in logarithmic time

Weighted Interval Scheduling

- How can we optimally schedule a resource?
  - This classroom, a computing cluster, …

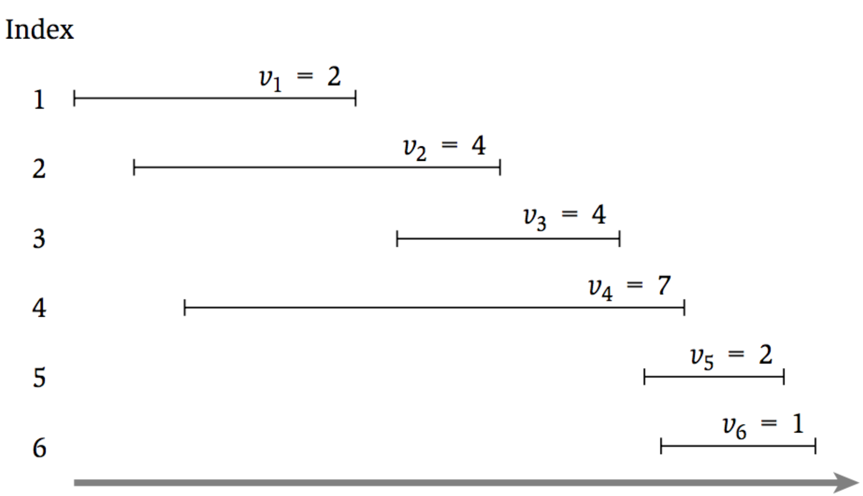
- **Input:** n intervals (s_i, f_i) each with value v_i
  - Assume intervals are sorted so $f_1 < f_2 < \cdots < f_n$
- **Output:** a compatible schedule S **maximizing** the total value of all intervals
  - A **schedule** is a subset of intervals $S \subseteq \{1, \ldots, n\}$
  - A schedule S is **compatible**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

Interval Scheduling



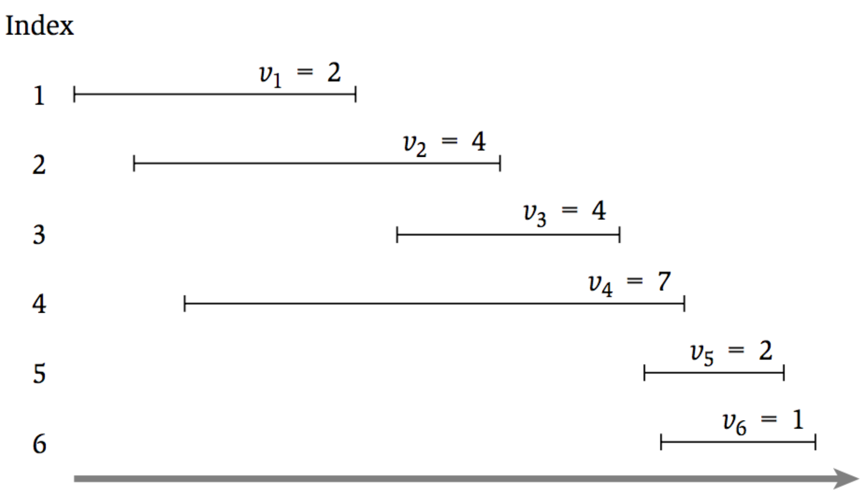
A Recursive Formulation

- Let O be the **optimal** schedule
- **Case 1:** Final interval is not in O (i.e. $6 \notin O$)
  - Then O must be the optimal solution for $\{1, \ldots, 5\}$



A Recursive Formulation

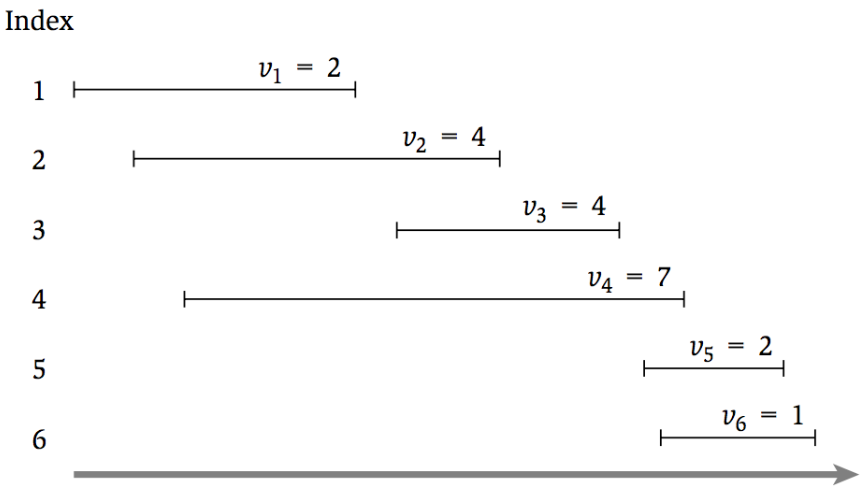
- Let O be the **optimal** schedule
- **Case 2:** Final interval is in O (i.e. $6 \in O$)
  - Then O must be $\{6\}$ + the optimal solution for $\{1, \ldots, 3\}$



A Recursive Formulation

Which is better?

- the optimal solution for $\{1, \ldots, 5\}$
- $\{6\}$ + the optimal solution for $\{1, \ldots, 3\}$



A Recursive Formulation: 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 & Recurrence

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

- **Case 1:** Final interval is not in O_i ($i \notin O_i$)
  - Then O_i must be the optimal solution for $\{1, \dots, i-1\}$
- **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 $\{1, \dots, p(i)\}$

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

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

Interval Scheduling: Straight Recursion

```
// All inputs are global vars
FindOPT(n):
  if (n = 0): return 0
  elseif (n = 1): return v_1
  else:
    return max{FindOPT(n-1), v_n + FindOPT(p(n))}
```

- What is the worst-case running time of **`FindOPT(n)`** (how many recursive calls)?

Interval Scheduling: Top Down

```
// All inputs are global vars
M ← empty array, M[0] ← 0, M[1] ← v1
FindOPT(n):
  if (M[n] is not empty): return M[n]
  else:
    M[n] ← max{FindOPT(n-1), vn + FindOPT(p(n))}
    return M[n]
```

- What is the running time of **`FindOPT(n)`**?

Interval Scheduling: Bottom Up

```
// All inputs are global vars
FindOPT(n):
  M[0] ← 0, M[1] ← v_1
  for (i = 2,…,n):
    M[i] ← max{M[i-1], v_i + M[p(i)]}
  return M[n]
```

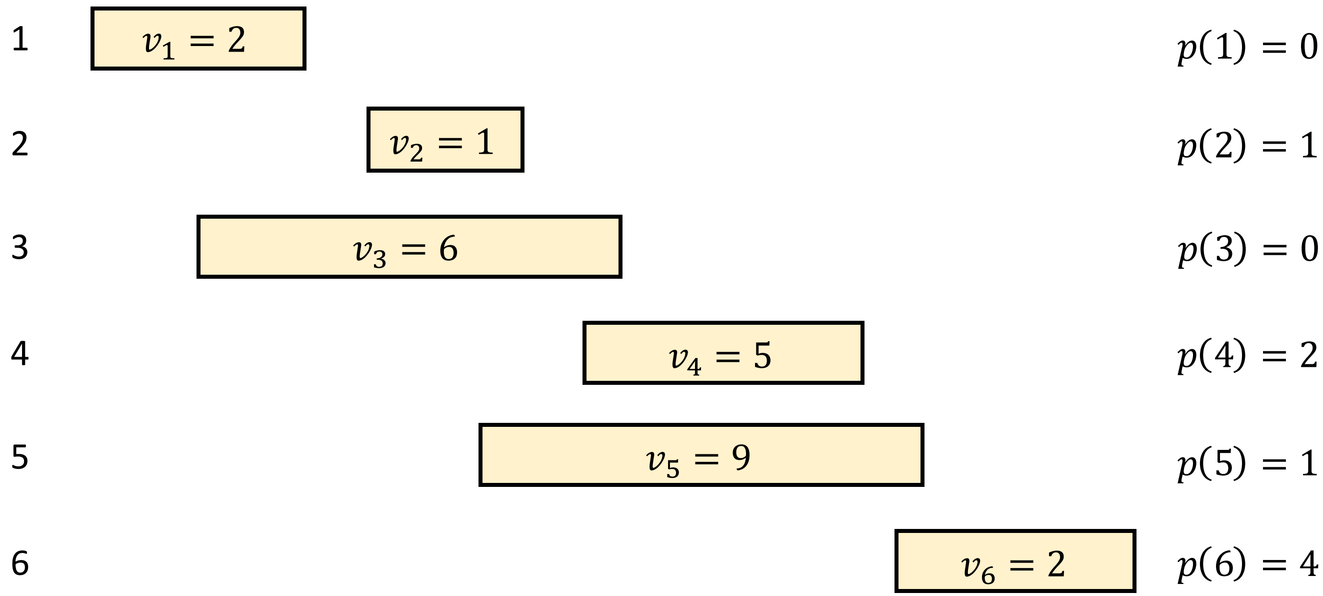
- What is the running time of **`FindOPT(n)`**?

Finding the Optimal Solution

```
// All inputs are global vars
FindSched(M,n):
  if (n = 0): return ∅
  elseif (n = 1): return {1}
  elseif (v_n + M[p(n)] > M[n-1]):
    return {n} + FindSched(M,p(n))
  else:
    return FindSched(M,n-1)
```

- What is the running time of **`FindSched(n)`**?

Now You Try

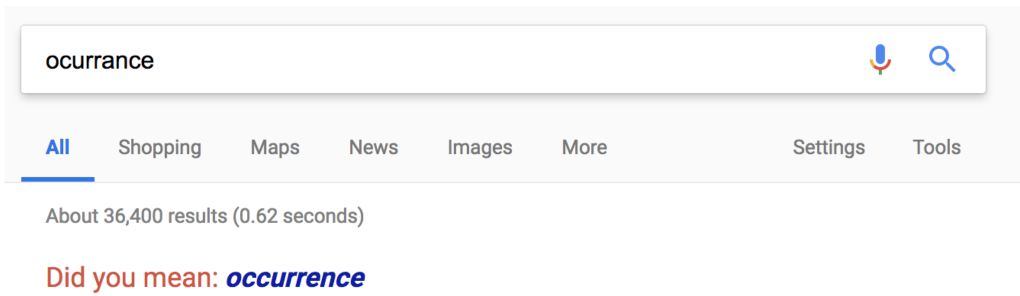


M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]

Edit Distance

Distance Between Strings

- Autocorrect works by finding similar strings



- `ocurrance` and `occurrence` seem similar, but only if we define similarity carefully

```
ocurrance        oc urrance
occurrence       occurrence
```

Edit Distance / Alignments

- Given two strings $x \in \Sigma^n, y \in \Sigma^m$, the **edit distance** is the number of insertions, deletions, and swaps required to turn x into y.

- Given an alignment, the cost is the number of positions where the two strings don't agree

o	c		u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

Edit Distance / Alignments

- **Input:** Two strings $x \in \Sigma^n, y \in \Sigma^m$
- **Output:** The minimum cost alignment of x and y
  - **Edit Distance** = cost of the minimum cost alignment

Dynamic Programming

- Consider the **optimal** alignment of x, y
- Three choices for the final column
  - **Case I:** only use x ($x_n, -$)
  - **Case II:** only use y ($-, y_m$)
  - **Case III:** use one symbol from each (x_n, y_m)

Dynamic Programming

- Consider the **optimal** alignment of x, y
- **Case I:** only use x ($x_n, -$)
  - deletion + optimal alignment of $x_{1:n-1}, y_{1:m}$
- **Case II:** only use y ($-, y_m$)
  - insertion + optimal alignment of $x_{1:n}, y_{1:m-1}$
- **Case III:** use one symbol from each (x_n, y_m)
  - If $x_n = y_m$: optimal alignment of $x_{1:n-1}, y_{1:m-1}$
  - If $x_n \neq y_m$: mismatch + opt. alignment of $x_{1:n-1}, y_{1:m-1}$

Dynamic Programming

- $\mathbf{OPT}(\boldsymbol{i}, \boldsymbol{j})$ = cost of opt. alignment of $x_{1:i}$ and $y_{1:j}$
- **Case I:** only use x ($x_i, -$)
- **Case II:** only use y ($-, y_j$)
- **Case III:** use one symbol from each (x_i, y_j)

Dynamic Programming

- $\mathbf{OPT}(\boldsymbol{i},\boldsymbol{j})$ = cost of opt. alignment of $x_{1:i}$ and $y_{1:j}$
- **Case I:** only use x ($x_i, -$)
- **Case II:** only use y ($-, y_j$)
- **Case III:** use one symbol from each (x_i, y_j)

Recurrence:

$$\mathrm{OPT}(i,j) = \begin{cases} 1 + \min\{\mathrm{OPT}(i-1,j), \mathrm{OPT}(i,j-1), \mathrm{OPT}(i-1,j-1)\ \} \\ \min\{1 + OPT(i-1,j), 1 + \mathrm{OPT}(i,j-1), \mathrm{OPT}(i-1,j-1)\} \end{cases}$$

Base Cases:

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

Edit Distance ("Bottom-Up")

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

  for (i= 1,…,n):
    for (j = 1,…,m):
      if (xi = yj):
        M[i,j] = min{1+M[i-1,j],1+M[i,j-1],M[i-1,j-1]
      elseif (xi != yj):
        M[i,j] = 1+min{M[i-1,j],M[i,j-1],M[i-1,j-1]}

  return M[n,m]
```

Summary

- Can compute EDIT in time $O(nm)$
  - If both strings have length $\leq n$ this is $O(n^2)$ time
  - Same algorithm works for any set of costs you choose for swaps, insertions, and deletions

- Dynamic Programming:
  - Question: Which of the two final symbols are in the optimal alignment?
  - Subproblems: EDIT between each pair of prefixes

Big Open Problem: Can we solve EDIT in $n^{2-\Omega(1)}$ time?

CS3000: Algorithms & Data

Unit 3: Dynamic Programming

a. Fibonacci Numbers
b. First Problem: Weighted Interval Scheduling
c. Knapsacks
d. Edit Distance
e. Longest Increasing Subsequence

Longest Increasing Subsequence (LIS)

- **Input:** a sequence of numbers $x_1, \ldots, x_n$

sequence

4	0	8	2	9	3	1	2	3	7	4	6	3

Longest Increasing Subsequence (LIS)

- **Input:** a sequence of numbers $x_1, \dots, x_n$

sequence

4	0	8	2	9	3	1	2	3	7	4	6	3

increasing subsequence

- Increasing Subsequence:
  indices $1 \leq i_1 \leq i_2 \leq \cdots \leq i_k \leq n$
  such that $x_{i_1} < x_{i_2} < \cdots < x_{i_k}$

Longest Increasing Subsequence (LIS)

- **Input:** a sequence of numbers $x_1, \ldots, x_n$

sequence

4	0	8	2	9	3	1	2	3	7	4	6	3

increasing subsequence

- **Output:** a longest increasing subsequence

sequence

4	0	8	2	9	3	1	2	3	7	4	6	3

longest increasing subsequence

Ask the Audience

- Find a longest increasing subsequence of

14	7	5	6	2	12

Writing the Recurrence

- Let $\mathrm{LIS}(j)$ be the length of the longest increasing subsequence of $x_1, \ldots, x_j$

- **Case i:** the last element of the sequence is x_i

6	10	14	5	12	8

Writing the Recurrence: Take II

- Let $\text{LIS}(j)$ be the length of the longest increasing subsequence **that ends with** x_j

- **Case i:** the last two numbers are x_i and x_j

6	10	14	5	12	8

Writing the Recurrence

- Let $\mathrm{LIS}(j)$ be the length of the longest increasing subsequence **that ends with** x_j
  - Note $\mathrm{LIS}(n)$ is **not necessarily** the length of the LIS
  - Need to know $\mathrm{LIS} = \max_{j=1\ldots n} \mathrm{LIS}(j)$
- **Case i:** the last two numbers are x_i and x_j

Writing the Recurrence

- Let $\text{LIS}(j)$ be the length of the longest increasing subsequence **that ends with** x_j
  - Note $\text{LIS}(n)$ is **not necessarily** the length of the LIS
  - Need to know $\text{LIS} = \max_{j=1\ldots n} \text{LIS}(j)$
- **Case i:** the last two numbers are x_i and x_j

Recurrence:

$$\text{LIS}(j) = 1 + \max_{1 \le i < j \text{ and } x_i < x_j} \text{LIS}(i)$$

Base Case:

$$\text{LIS}(1) = 1$$

Practice

- Fill out the values $\mathrm{LIS}(j)$ for $j = 1, \dots, 6$

6	10	5	14	8	7

j	1	2	3	4	5	6
LIS(j)	1					

Practice

- Fill out the values $\mathrm{LIS}(j)$ for $j = 1, \ldots, 6$

6	10	5	14	8	7

j	1	2	3	4	5	6
LIS(j)	1	2	1	3	2	2

Solving the Recurrence: Bottom-Up

```
// All inputs are global vars
FindOPT(n):
  M[1] ← 1

  for (j = 2,…,n):
```

$$M[j] = 1 + \max_{1 \le i < j \text{ and } x_i < x_j} M[i]$$

```
  return
```

$$\max_{1 \le j \le n} M[j]$$

Recovering the LIS (Final Symbol)

- Let $\text{LIS}(j)$ be the length of the longest increasing subsequence **that ends with** x_j

Recurrence:

$$\text{LIS}(j) = 1 + \max_{1 \le i < j \text{ and } x_i < x_j} \text{LIS}(i)$$

Base Case:

$$\text{LIS}(1) = 1$$

Length of LIS

$$\text{LIS} = \max_{1 \le j \le n} \text{LIS}(j)$$

Recovering the LIS (Other Symbols)

- Let $\mathrm{LIS}(j)$ be the length of the longest increasing subsequence **that ends with** x_j

Recurrence:

$$\mathrm{LIS}(j) = 1 + \max_{1 \le i < j \text{ and } x_i < x_j} \mathrm{LIS}(i)$$

Base Case:

$$\mathrm{LIS}(1) = 1$$

Length of LIS

$$\mathrm{LIS} = \max_{1 \le j \le n} \mathrm{LIS}(j)$$

Recovering the LIS

- Fill out the values $\mathrm{LIS}(j)$ for $j = 1, \ldots, 6$

6	10	5	14	8	7

j	1	2	3	4	5	6
LIS(j)	1	2	1	3	2	2

Summary

- Can compute a LIS in time $O(n^2)$
  - Same algorithm works for longest non-decreasing, longest decreasing, longest non-increasing, and more

- Dynamic Programming:
  - Question: What is the final symbol in the LIS?
  - Subproblems represent LIS **with a specific final symbol**
  - The actual optimal value is not always in LIS(n)