Algorithms so Far

**Divide and Conquer:**

- Integer Multiplication: \(O(n \log^2 3) = O(n^{1.58})\)
- Matrix Multiplication: \(O(n \log^2 7) = O(n^{2.81})\)
- Merge Sort: \(O(n \log n)\)
- FFT: \(O(n \log n)\)

**Simple Graph Algorithms**

- DFS, connected components: \(O(n+m)\) \(n=111, m=1E1\)
- Topological search, SCC

**Single Source Shortest Paths**

- DFS: \(O(n+m)\)
- Dijkstra: \(O((n+m) \log n)\)
- Bellman-Ford: \(O(nm)\)
- DAG-SSSP: \(O(n+m)\)

**Greedy**

- Scheduling: \(O(n)\)
- Huffman Coding: \(O(n \log n)\)
- Kruskal and Prim (DST): \(O((n+m) \log n)\)
- Horn Formulae: \(O(1F1)\)
- Greedy Set Cover (later, only finds approx. min)

- Important basic algorithms, fast
- Not a very general tool

**Dynamic Programming**

- A versatile, powerful algorithm design principle
1) Longest path in a DAG

Input: DAG $G = (V, E)$

Goal: Find length of longest path in $G$

Recursive Algorithm

$L = \text{length of longest path in } G$

$$\max_{v \in V} L(v)$$

$L(v) = \text{length of longest path ending in } v$

$L(v) = \max_{(w, v) \in E} L(w) + 1$

Recursive Relation

$L(v) = \max \left\{ L(u)+1, L(w)+1 \right\}$
\[ L(v) = \max_{(u,v) \in E} \left( L(u) + 1 \right) \]

0 \hspace{1cm} \text{if no incom. edge}

\underline{Algorithm}

\[ L(v) \]

"Returns length of longest path ending in \textit{v}"

\begin{align*}
\text{IF no incom. edge} & \quad \text{L}(v) = 0 \\
\text{ELSE:} & \quad \text{L}(v) = \max_{w \in E} \left( \text{L}(w) + 1 \right)
\end{align*}

\underline{Implementation:}

\begin{align*}
\text{current} & = 0 \\
\text{For all w} & \in E \\
\text{if } & \quad \text{L}(w) + 1 > \text{current} \\
\text{current} & = \text{L}(w) + 1
\end{align*}

\text{Return current}

Does it terminate? Yes. Each iterative call explores edges pointing backwards in the DAG, so eventually will end at the sources.

How long does it take?
Calls: \[ L(5) \]
\[ L(4) \quad L(3) \]
\[ L(3) \quad L(2) \quad L(1) \quad L(1) \]
\[ L(2) \quad L(1) \quad L(1) \quad L(1) \]
\[ L(1) \quad \text{grows exp } \]

\[ T(i) = T(i-1) + T(i-2) \]
-> Fibonacci # s

Solution: Recursion with memorization

(\text{Remember } L(i) \text{ if calculated once})

Non recursive Implementation

Subproblems:
\[ L(1), \ldots, L(19) \]
**Dependence:**

$L(i)$ depends on $L(j)$, $j < i$

**Compute in order:**

$L(1), L(2), \ldots, L(9)$

**General Graph:**

**Subproblem:** $L(v)$ for all $v \in V$

**Dependency:** $L(v)$ depends on all incoming edges $uv \in E$

**Order to compute:**

*topological sorted order* of $G$

**Pseudo Code:**

*Topologically sort $G$

Let $i$ be the $i^{th}$ vertex in topol. sort order

• For all $i$, $L[i] = 0$

• For $i = 1, \ldots, n$,

\[ L[i] = \max_{j \in E} L[j] \]

```
current = 0
for all $wv \in E$
    if $L(w) + 1 > current$
        current = $L(w) + 1$
return current
```

**Run time:**
2) Longest Increasing Subsequence

Reduce to previous problem:

Sequence \( a_1, a_2, \ldots, a_n \)

Make it into DAG

\[ \forall i < j \in E \ \text{and} \ a_i < a_j \]

Running time \( O(n^2) \) (we need to check \( a_i < a_j \) \( \forall \binom{n}{2} \) pairs \( i, j \))

3) Edit Distance

Input: two strings \( x[1, \ldots, n] \) \( y[1, \ldots, m] \)

Task: Find the minimum \# of keystrokes to edit \( x \) into \( y \)

\[ \text{[insert a char, delete a char, substitute a char]} \]

\[ x = \text{SUNNY} \]
\[ y = \text{SOWNY} \]

\[ \text{CAT} \rightarrow \text{HAT} \quad \text{cost} = 1 \]

\[ \text{ABABABA} \quad \text{BABABA} \quad \text{cost} = 2 \] (place first A, insert A at end)
Why is this interesting?

• spell checker
• DNA

How can we represent a sequence of edits?

• Complicated deleting, inserting, shifting things
• Better visualization needed

\[ \text{S U N N Y} \]
\[ \downarrow \]
\[ \text{S N O W Y} \]

where do these letters come from?

\[ \text{S} \quad \text{S} \quad \text{S} \quad \text{O} \]

where do these letters go to?

\[ \text{S} \quad \text{U} \quad \text{N} \quad \text{N} \quad \text{Y} \]

Tiles:
\[ \text{delete } \quad \text{insert} \quad \text{keep} \quad \text{substitute} \]

\[
\begin{align*}
\text{S} & \quad \text{U} \quad \text{N} \quad \text{N} \quad \text{Y} \\
\text{K} & \quad \text{D} \quad \text{K} \quad \text{S} \quad \text{I} \quad \text{K} \\
\text{S} & \quad \text{N} \quad \text{O} \quad \text{W} \quad \text{Y}
\end{align*}
\]

Can do this in arb. order.

**Dynamic Programming strategy**

3 steps:
1) Define subproblem
2) Write down recurrence
3) Determine order of calculations

**Step 1:**

\[ E[i,j] = \text{Edit dist. between } x[1, \cdots, i] \text{ and } y[1, \cdots, j] \]

\[ x = \text{SUNNY} \]
\[ y = \text{SNOWY} \]

\[ \text{e.g. } E[5,5] \quad E[SU, SNOW] \]

\[ \cdots \]

\[ (m+1)(n+1) \text{ subproblems (include empty string)} \]

**Step 2:** Recurrence relation

Write down cases
Look at optimal solution

\[
\begin{bmatrix}
- & S & U & N & N & - \\
S & N & O & W & - & W
\end{bmatrix} + \begin{cases}
\text{keep Case 1} & 0 \\
\text{insert Case 2} & 1 \\
\text{delete Case 3} & 1
\end{cases}
\]

These are the possibilities
Actual answer → minimum

\[
E[\text{SUNNY, SNOWY}] = \min \left\{ E[\text{SUNN, SNOW}] + 1, E[\text{SUNNY, SNOW}] + 1, E[\text{SUNN, SNOWY}] + 1 \right\}
\]

In general
\[ E[i,j] = \min \left\{ \begin{array}{ll} E[i-1,j] + 1 & \text{Insert} \\ E[i,j-1] + 1 & \text{Delete} \\ E[i-1,j-1] + \text{DIFF}(x_i, y_j) & \text{Keep} \\ \end{array} \right. \]

Step 3: Pick an order:

Next lecture!