Dynamic Programming

- Recursion
  + Memoization (= remembering, keeping in memory)
- Systematic implementation of a brute-force strategy.

Def Fibonacci Numbers

\[
F_0, F_1, F_2, F_3, F_4, F_5, F_6, \ldots
\]

\[
0, 1, 1, 2, 3, 5, 8, \ldots
\]

- \(F_0 = 0, F_1 = 1\), \(F_i = F_{i-1} + F_{i-2} \quad \forall i \geq 2\).

Problem Given \(n\), compute \(F_n\).

Recursive algo

```c
int F(n) {
    if n=0, return 0.
    if n=1, return 1.
    Else,
        a = F(n-1),
        b = F(n-1),
        return a+b.
}
```
Unrolling recursion:

- Only \( m+1 \) distinct recursive calls \( F(0), F(1), \ldots, F(m) \).

- Inefficiency is due to big redundancy, e.g. after computing \( F_3 \), at \( (*) \), there was no need to recompute it at \( (***) \). One could have stored the solution.

Modified Algorithm

- Store solution to every "subproblem" that has been solved.
\[ F_5 \quad F_4^* \quad F_3^* \quad F_2^* \quad F_2 \quad F_1 \quad F_0 \]

This is really the same as iterative algorithm that simply computes

\[ F_0, F_1, F_2, \ldots, F_n \]

sequentially in that order.

Store \( F_1 = 1 \)
Store \( F_0 = 0 \)
store \( F_2 = 1 \)
store \( F_3 = 2 \)
store \( F_4 = 3 \)

stored values of \( F_1, F_2, F_3 \) used at \((\star)\).
Dynamic Programming

- Identify a set of polynomially many subproblems such that the original problem is included as one of these subproblems.
- Identify an order from smallest to the largest subproblem.
- Identify a recurrence formula that allows one to compute a subproblem given solutions to all the smaller subproblems.

Fibonacci Example:

Subproblems: \( F(0), F(1), \ldots, F(n) \).

Order: \( \rightarrow \)

Recurrence: \( F_i = F_{i-1} + F_{i-2} \).

- Dynamic Prog. Algorithm then simply computes solutions to all the subproblems in the given order using the recursive formula.
Subset Sum (with bounded integers)

Problem
Given positive integers \(a_1, a_2, \ldots, a_n\) such that \(\forall i, \ a_i \in \{1, 2, 3, \ldots, W\}\),
\[W = n^2.\]

Given a number \(b\) (s.t. \(1 \leq b \leq n \cdot W\)).

Goal is to decide whether there is a subset \(S \subseteq \{1, 2, \ldots, n^2\}\) s.t.
\[\sum_{i \in S} a_i = b.\]

Note: Algorithms usually solve the decision problem and also find an actual solution (e.g., the desired set \(S\) above if one exists).

Note: Subproblems usually correspond to prefixes, suffixes, or consecutive subsequences of some given sequence.
\[\#\text{prefixes, } \#\text{suffixes} = O(n),\]
\[\#\text{consecutive subsequences} = O(n^2).\]
\[\#\text{subproblems must be at most polynomial in } n.\]
Either there is a subset of \( \{a_1, \ldots, a_k\} \) that sums to \( B \)

or there is a subset of \( \{a_1, \ldots, a_k\} \) that sums to \( B - a_{k+1} \).

In the latter case, if \( T \subseteq \{a_1, \ldots, a_k\} \) sums to \( B - a_{k+1} \), then \( S = T \cup \{k+1\} \) sums to \( B \).

Can be ordered as per increasing value of \( k \) and for given \( k \), as per increasing order of \( B \).
There is a subproblem

\[ \text{SUM}(\{a_1, \ldots, a_k\}, B) \]

for every \(1 \leq k \leq n\) and every \(0 \leq B \leq n \cdot W\).

- The number of subproblems is at most \(n \cdot (nW) = n^4\).
- The original problem is \(\text{SUM}(\{a_1, \ldots, a_n\}, b)\).

The intention is that

\[ \text{SUM}(\{a_1, \ldots, a_k\}, B) = \begin{cases} \text{YES} & \text{if } \exists S \subseteq \{1, \ldots, k\} \text{ s.t. } \sum_{i \in S} a_i = B \\ \text{NO} & \text{otherwise} \end{cases} \]

Recursive formula

\[ \text{SUM}(\{a_1, \ldots, a_{k+1}\}, B) = \text{SUM}(\{a_1, \ldots, a_k\}, B) \begin{cases} \text{OR} & \text{SUM}(\{a_1, \ldots, a_k\}, B-a_{k+1}) \end{cases} \]

In words, there is a subset of \(\{a_1, \ldots, a_{k+1}\}\) that sums to \(B\) iff
**Initialization**

\[
\text{SUM}(\{a_i\}, B) = \begin{cases} 
\text{YES} & \text{if } a_i = B \\
\text{NO} & \text{otherwise}
\end{cases}
\]

\[
\text{SUM}(\{a_1, \ldots, a_k\}, 0) = \text{YES} \quad (\because S \subseteq \{1, \ldots, k\} \text{ can be taken as empty})
\]

---

**Matrix Chain Multiplication**

Def Given two matrices \(A, B\) of sizes \(p \times q, q \times r\) respectively

\(C = A \cdot B\) is a \(p \times r\) matrix s.t.

\[
C_{ij} = \sum_{k=1}^{q} A_{ik} \cdot B_{kj} \quad \text{for } 1 \leq i \leq p, 1 \leq j \leq r.
\]

\[
\begin{array}{c|c}
\text{i} & \text{q} \\
\hline
\text{2} & \text{q} \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{i} & \text{j} & \text{q} \\
\hline
1 & 2 & \text{q} \\
\hline
\end{array}
\]

\[
A \cdot B = C.
\]

Time to compute = \(p \cdot q \cdot r\).
Fact: Given matrices $A$, $B$, $C$, where $A \in \mathbb{R}^{p \times q}$, $B \in \mathbb{R}^{q \times r}$, and $C \in \mathbb{R}^{r \times s}$, the associative property holds:

$$(A \cdot B) \cdot C = A \cdot (B \cdot C).$$

However, the time required for these two cases could be very different.

Example:

$$A \in \mathbb{R}^{1 \times q}, \quad B \in \mathbb{R}^{q \times 1}, \quad C \in \mathbb{R}^{1 \times q}.$$  

Time to compute $(A \cdot B) \cdot C = 1 \cdot q \cdot 1 + 1 \cdot 1 \cdot q = 2q$.

Time to compute $A \cdot (B \cdot C) = q \cdot 1 \cdot 1 + 1 \cdot q \cdot q = 2q^2$.

Problem: Given $n$ matrices $A_1, A_2, \ldots, A_n$, find the minimum cost way to compute the multiplication $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

Note: Example: $n = 4$. There are 5 possible ways:

$$((1 \cdot 2 \cdot (3, 4))), \quad ((1, 2), (3, 4)), \quad ((1 \cdot (2, 3)), 4),$$

$$((1 \cdot 2), (3), 4)), \quad ((1 \cdot (2, 3)), 4).$$

The number of ways $C_n$ is called the "Catalan number".

$$C_n = \frac{1}{2n-1} \binom{2n-1}{n-1}, \quad c_n \text{ is exponential in } n.$$
Dynamic Programming based algorithm

Idea: Suppose the "outermost" product is

\[ A_1 \cdot A_2 \cdot \ldots \cdot A_n = (A_1 \cdot A_2 \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_n) \cdot \]

If \( \text{cost}(\cdot) \) denotes optimal cost of computing product then

\[ \text{cost}(A_1 \cdot A_2 \cdot \ldots \cdot A_n) = \text{cost}(A_1 \cdot A_2 \cdot \ldots \cdot A_k) + \]
\[ \text{cost}(A_{k+1} \cdot \ldots \cdot A_n) + \]
\[ P_1 \cdot P_{k+1} \cdot P_n. \]

We can try out all "splitting points" \( k \) and take the best.

Subproblems

For every \( 1 \leq i \leq j \leq n \), there is a subproblem

\[ \text{cost}(A_i \cdot A_{i+1} \cdot \ldots \cdot A_j) \]

that asks for the minimum cost of computing the product \( A_i \cdot A_{i+1} \cdot \ldots \cdot A_j \).

\# subproblems is \( O(n^2) \).

Original problem is \( A_1 \cdot A_2 \cdot \ldots \cdot A_n \) i.e. \( i=1 \) \( j=n \).
Recurrence. For \( i \leq j \),

\[
\text{cost}(A_i : A_{i+1} \ldots A_j) = \\
\min_{i \leq k \leq j} \left\{ \text{cost}(A_i : A_k) + \text{cost}(A_{k+1} : A_j) + P_i P_{k+1} P_j \right\}
\]

- For \( i = j \), \( \text{cost}(A_i) = 0 \).

Order. According to increasing order of length of the consecutive subsequence \( A_i : A_{i+1} \ldots A_j \).

---

Longest Common Subsequence

Example

\[
X = \text{AGCGTAG} \\
Y = \text{GTCAGA}
\]

Def. \( Z = (z_1 \ldots z_k) \) is a subsequence of \( X = (x_1 x_2 \ldots x_n) \) if there are indices \( 1 \leq i_1 < i_2 \ldots < i_k \leq n \) such that \( Z_1 = x_{i_1}, Z_2 = x_{i_2}, \ldots, Z_k = x_{i_k} \).
Problem: Given two sequences \( X = (x_1, x_2, \ldots, x_n) \) \( Y = (y_1, y_2, \ldots, y_m) \), find L.C.S. i.e. sequence \( Z \) that is a subsequence of both \( X \) and \( Y \) and s.t. \( |Z| \) is maximum.

Idea: Given \( X = x_1, x_2, \ldots, x_n \) \( Y = y_1, y_2, \ldots, y_m \), one can begin by considering cases depending on whether \( x_1 \) is "matched" with \( y_1 \).

Subproblems: All pairs of suffixes
\( (x_i, x_{i+1}, \ldots, x_n, y_j, y_{j+1}, \ldots, y_m) \).
- \( \# \) subproblems = \( O(m \cdot n) \).
- Original problem corresponds to \( i=1, j=1 \).

Let LCS \( (x_i, x_{i+1}, \ldots, x_n, y_j, y_{j+1}, \ldots, y_m) \) denote the length of the largest common subseq.
Recursive formula

\[ \text{LCS} \left( x_i, x_{i+1}, \ldots, x_n, y_j, y_{j+1}, \ldots, y_m \right) \]

\[ = \max \left\{ \begin{array}{l}
\text{LCS} \left( x_{i+1}, \ldots, x_n, y_j, y_{j+1}, \ldots, y_m \right) \\
\text{LCS} \left( x_i, x_{i+1}, \ldots, x_n, y_{j+1}, \ldots, y_m \right) \\
1 + \text{LCS} \left( x_{i+1}, \ldots, x_n, y_{j+1}, \ldots, y_m \right) \quad \text{if } x_i = y_j
\end{array} \right. \]

Order In increasing order of sum of lengths of two sequences i.e., \( |n-i+1| + |m-j+1| \).

Base case

\[ \text{LCS} \left( x_n, \emptyset \right) = 0 \quad \text{LCS} \left( \emptyset, y_m \right) = 0 \]

\[ \text{LCS} \left( \emptyset, \emptyset \right) = 0 \]

Note. There are three choices in \( \max \left\{ \#1, \#2, \#3 \right\} \).

While computation, one can keep track of which of the three choices was the best (maximum) one. This information can be used to find the LCS and not just the length of the LCS.