• Divide and Conquer (recursiveness)

• Dynamic programming
Recursiveness: divide and conquer

• One breaks the input into several parts solve the problem in each part recursively.
• Then, combines the solution to these problems into an overall solution.

\[ F(n) = n! = n(n-1)(n-2) \cdots 1 \]

\[ F(n) = nF(n-1) \]
Divide and Conquer

- Sort a List of elements in ascending order:

```
5 2 4 7 1 3 2 6
```
Divide and Conquer

• Sort a List of elements in ascending order:

\[
\begin{array}{cccccc}
5 & 2 & 4 & 7 & 1 & 3 & 2 & 6 \\
\end{array}
\]

\[
\begin{array}{cccc}
5 & 2 & 4 & 7 \\
1 & 3 & 2 & 6 \\
\end{array}
\]
Divide and Conquer

• Sort a List of elements in ascending order:

\[ \begin{array}{c c c c}
2 & 4 & 5 & 7 \\
1 & 2 & 3 & 6 \\
\end{array} \]
Divide and Conquer

• Sort a List of elements in ascending order:

1 2 2 3 4 5 6 7

2 4 5 7

1 2 3 6
The Change Problem

- Given a set of coins: 
  \[ 1 = V_1, V_2, \ldots, V_n \]
- Find the minimum number of coins to make a given amount \( C \)

\[ S(C) = \min\{S(C-V_i)\} + 1 \]
Why It Takes So Long

- For each value, we repeatedly compute the same result numerous times.

- For 99: 74, 79, 89, 94, 98
- For 98: 73, 78, 88, 93, 97
- For 97: 72, 76, 87, 92, 96
- For 96: 71, 75, 86, 91, 95
- For 95: 70, 74, 85, 82, 94
Save Time by Expending Space

- Instead of calling the function recursively as we need it...
  - Compute each amount from 1 to 99
  - Save each result as we need it
  - Instead of doing a recursive computation, look up the value in the array of results
- This is called dynamic programming
Why it Works?

- By the order we’re computing in, we know that we have every value less than $amt$ covered when we get to $amt$.
- We never need a value larger than $amt$, so all the values we need are in the table.
Dynamic Programming in General

• (1) Decide on subproblems to solve and construct table to hold subproblem's solutions.
• (2) Recurrence: How to solve “large” problems using solutions to “smaller” subproblems.
• (3) Tabular computation: use recurrence relation to fill in table.
• (4) Traceback: Go backwards in table to recover an actual solution.
Edit distance of two sequence

S1 : CGTTAC
S2 : GGTAA

- **Edit distance**: Number of edit operations (mutations) to transform S1 into S2.

- **Edit operations**:
  - M : match two char.
  - I : insert one char.
  - D : delete one char
  - R : Replace one

MUTATIONS
Sequence (String) Alignment

- An alignment of S1 and S2 adds '-' (space) into S1 to get string S1', and S2 to get S2' such that |S1'| = |S2'|

S1' : C G T T A  -  C
S2' :  -  G G T A A  -

Number of mismatch = 4 (same as # of edits)
Edit distance Problem

• Compute the smallest edit distance between two sequences $S_1$ (size n) and $S_2$ (size m).

$$ED(S_1, S_2)$$

$S_1 : CGTTAC$

$S_2 : GGTAA$
STEP 1: Subproblems, compute the edit distance of a prefix of S1 and a prefix of S2.

$S1[1 .. i] \quad 0 \leq i \leq n$

$S2[1 .. j] \quad 0 \leq j \leq m$

$D(i,j) = ED(S1[1 .. i], S2[1 .. j])$
• BASE CASE:

\[ D(0,j) = j \text{ (j inserts) } 0 \leq j \leq m \]
\[ D(i,0) = i \text{ (i deletes) } 0 \leq i \leq n \]

• GENERAL CASE \( i=1..n \), \( j=1..m \)

\[
D(i,j) = \min \begin{cases} 
1 + D(i-1,j) \\
1 + D(i,j-1) \\
1 + D(i-1,j-1) & \text{if } S1[i] \neq S2[j] \\
D(i-1,j) & \text{if } S1[i] = S2[j]
\end{cases}
\]