Dynamic Programming

Dynamic Programming (DP) is a design technique related to **Divide and Conquer** and **Greedy** algorithms:

The problem at hand is subdivided into partial problems (subproblems) whose results are combined to solve the original problem.

- In Divide and Conquer, you also divide the problem into subproblems, but each subproblem is solved only once (so D&C can be efficient).
- In Greedy algorithms, at each step the problem is reduced to its subproblem by 1) making an optimal local decision, and then 2) solving the resulting subproblem.

DP applies where D&C would cause repeated solutions of some subproblems, and where Greedy would be making wrong (locally optimal, but not globally optimal) choices.

The main principle of DP:

Make a table (array, multi-dimensional array, or a hash-table) and solve each subproblem *once and only once*. Whenever you need to solve it again, just look up the answer in the table.

```
function f(n):

    table[n] = {0, 1, Nil, Nil, Nil, ... Nil}

    function g(n):
        if table[n] != Nil:
            return table[n]
        else:
            ans = g(n-1) + g(n-2)
            table[n] = ans
            return ans

    return g(n)
```

Rod Cutting Problem

How to cut a steel rod of length n into pieces (of integral length) to maximize the revenue?

Input:

- n, the length of the given rod.
- p_i, the table of prices for rods of length $i = 1, 2, 3, \ldots n$.

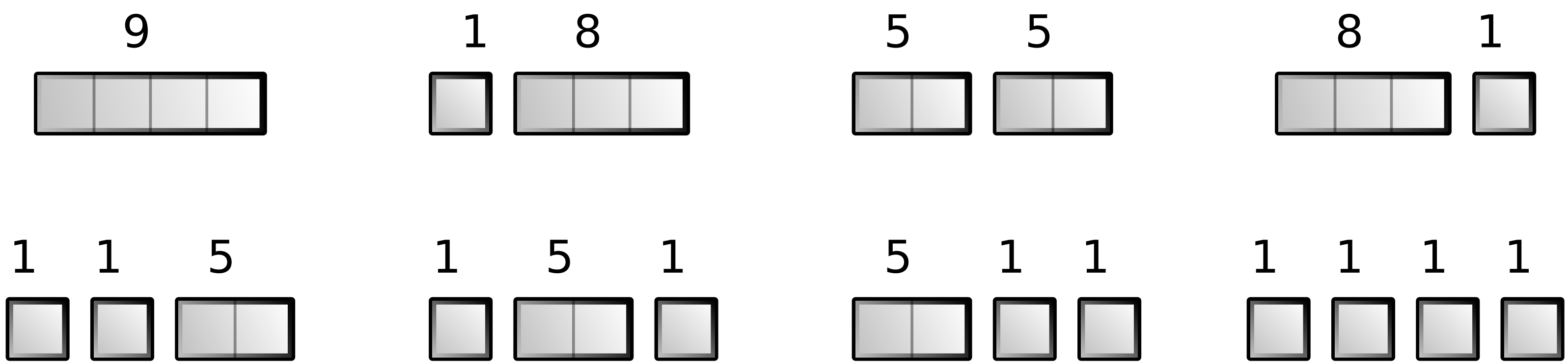
Output: The maximum revenue obtainable for cutting the rod n into pieces (whose length sums up to n), computed as the sum of the prices for the individual rods.

Example:

Given the table p_i:

i	1	2	3	4	5	6	7	8
p_i	1	5	8	9	10	17	17	20

If length $n = 4$, then there are $2^{4-1} = 8$ ways to cut the rod:



What way to cut gives the highest revenue?

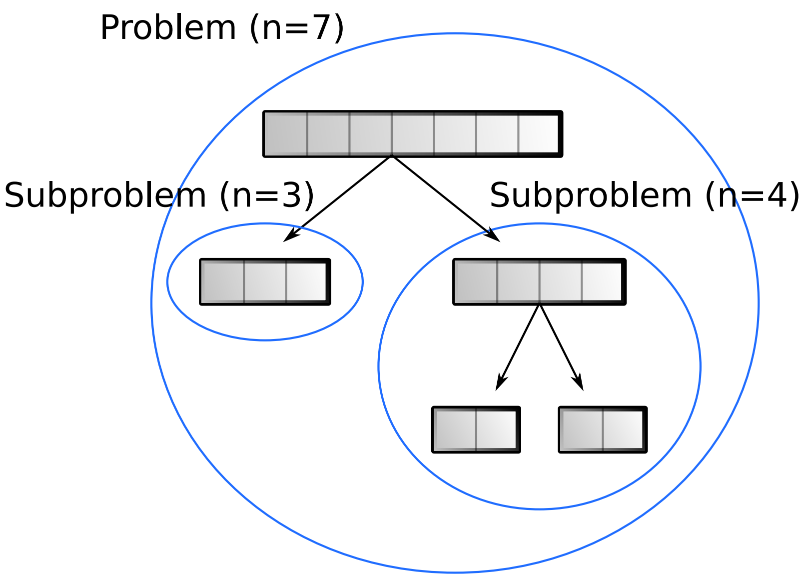
What would be the time complexity of such an exhaustive (brute-force) search?

Optimal substructure

To solve the original problem of size n, solve subproblems on smaller sizes. After making a cut, we have two subproblems.

An optimal solution to the original problem incorporates optimal solutions to the subproblems.

Example: For $n = 7$, one of the optimal solutions makes a cut at 3 inches, giving two subproblems, of lengths 3 and 4.



Since that division is optimal, then the remaining pieces of size 3 and 4 must be also divided optimally.

A simpler way to decompose the problem:

- Every optimal solution has a leftmost cut.
- Need to divide only the remainder, not the first piece.
- Leaves only one subproblem to solve, rather than two subproblems.

Recursive top-down solution:

CUT-ROD(p, n)

  if $n == 0$

      return 0

  $q = -\infty$

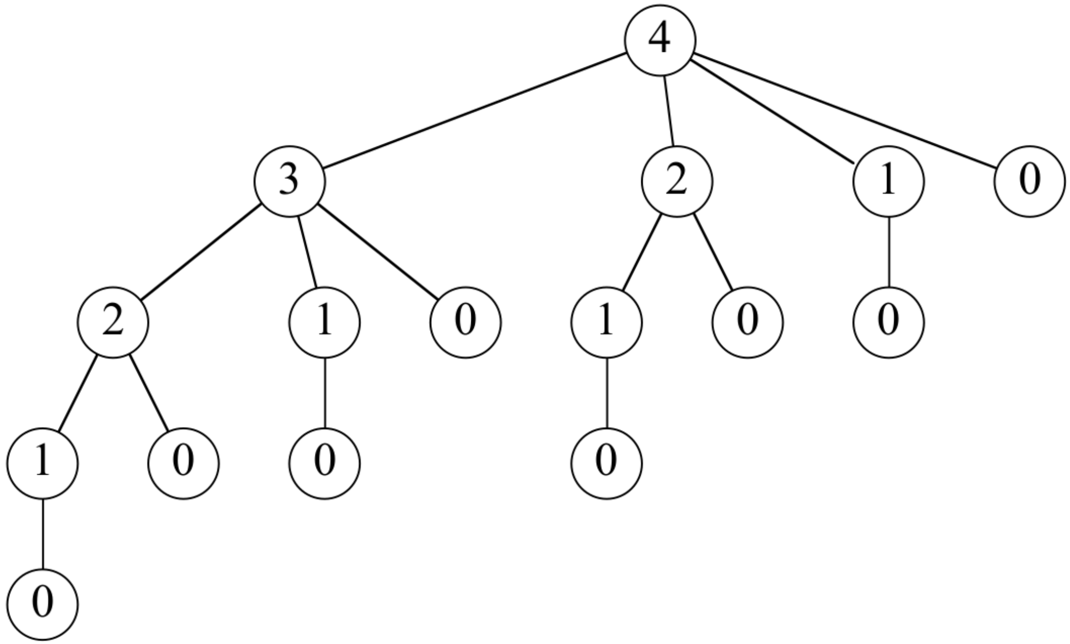
  for $i = 1$ **to** n

      $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$

  return q

However, its time complexity is “not very good”:

The tree of recursive calls for $n = 4$:



The number of function calls for a problem of length n:

$$T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n > 0. \end{cases} \quad \rightarrow \quad T(n) = 2^n. \text{ Exponential.}$$

Dynamic programming solution

Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.

Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.

"*Store, don't recompute*": time-memory trade-off. Can turn an exponential-time solution into a polynomial-time solution.

Two basic approaches:

- top-down with memoization, and
- bottom-up.

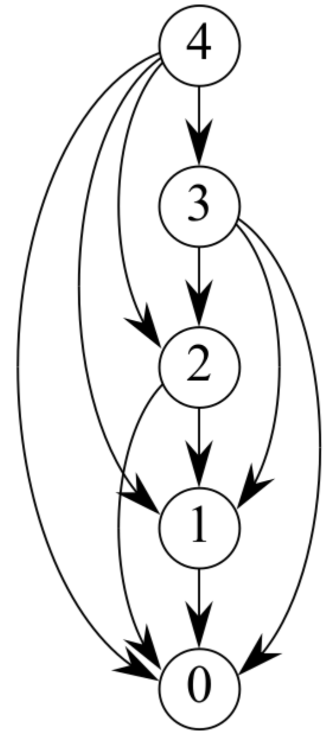
Top-down DP with *memoization*:

```
MEMOIZED-CUT-ROD(p, n)
  let r[0..n] be a new array
  for i = 0 to n
      r[i] = -∞
  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0
      return r[n]
  if n == 0
      q = 0
  else q = -∞
      for i = 1 to n
          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
  r[n] = q
  return q
```

Bottom-up DP:

```
BOTTOM-UP-CUT-ROD(p, n)
  let r[0..n] be a new array
  r[0] = 0
  for j = 1 to n
      q = −∞
      for i = 1 to j
          q = max(q, p[i] + r[j − i])
      r[j] = q
  return r[n]
```



Both the top-down and bottom-up versions run in $\Theta(n^2)$ time:

- Bottom-up: Doubly nested loops. Number of iterations of inner for loop forms an arithmetic series.
- Same complexity (just computed in different order).

Recovering the optimal way to cut. Additionally to the max revenue $r[i]$ for a subproblem i, we remember the first cut $s[i]$:

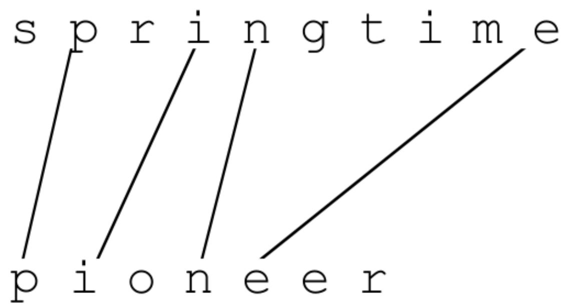
```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  let r[0..n] and s[0..n] be new arrays
  r[0] = 0
  for j = 1 to n
      q = -∞
      for i = 1 to j
          if q < p[i] + r[j - i]
              q = p[i] + r[j - i]
              s[j] = i
      r[j] = q
  return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
      print s[n]
      n = n - s[n]
```

i	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$	0	1	2	3	2	2	6	1	2

Longest Common Subsequence (LCS)

Problem:
Given 2 sequences, $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$.
Find a subsequence common to both whose length is longest.
A subsequence doesn't have to be consecutive, but it has to be in order.



Brute-force algorithm: For every subsequence of X, check whether it's a subsequence of Y. Time: $\Theta(n2^m)$:

- 2^m subsequences of X to check.
- Each subsequence takes n time to check, by scanning through Y.

Optimal substructure

Prefix notation:
$X_i = \langle x_1, \ldots, x_i \rangle$,
$Y_i = \langle y_1, \ldots, y_i \rangle$.

We focus of finding the length of LCS first, then extend solution to find the subsequence itself.

Define $c[i, j] = |\mathrm{LCS}(X_i, Y_j)|$**, the length of an LCS of** X_i **and** Y_j**.**
(Then we are looking for $c[m, n]$, the length of $\mathrm{LCS}(X, Y)$.)

Theorem:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j, \\ \max(c[i-1, j], c[i, j-1]) & \text{otherwise.} \end{cases}$$

Proof: <u>Case</u> $i = 0$ or $j = 0$ is trivial.

<u>Case</u> $x_i = y_j$ (the two last symbols match):
Let us denote an optimal solution as
$Z = \langle z_1, \ldots, z_k \rangle = LCS(X_i, Y_j)$, where $k = c[i, j]$.
z_k must be equal to the matched last symbol $x_i = y_j$. (Assume it's not, then we can make an even better optimal solution by adding x_i at the end: $Z' = \langle z_1, \ldots, z_k, x_i \rangle$.)

To prove optimal substructure, we have to show that
$Z_{k-1} = LCS(X_{i-1}, Y_{j-1})$.

By contradiction (*"cut-and-paste"* argument): Assume Z_{k-1} is not LCS of X_{i-1} and Y_{j-1} and there is a longer common subsequence W with length $|W| > k - 1$. Then join W with x_i, constructing a common subsequence of X_i and Y_j of length $|W| + 1 > k$, which contradicts with that $c[i, j] = |LCS(X_i, Y_j)| = k$. Therefore $Z_{k-1} = LCS(X_{i-1}, Y_{j-1})$, and so $c[i, j] = c[i - 1, j - 1] + 1$.

Case $x_i \neq y_j$ (the two last symbols do not match):

Again, we consider an optimal solution
$Z = \langle z_1, \ldots, z_k \rangle = LCS(X_i, Y_j)$, where $k = c[i, j]$.

There are two possibilities:
1) If $z_k \neq x_i$ then we can "discard" x_i and the whole Z is a common subsequence of X_{i-1} and Y_j.

To prove optimal substructure, we have to show that
$Z = LCS(X_{i-1}, Y_j)$, and so $c[i, j] = c[i-1, j]$.

Again, by contradiction: Assuming there exists a common subsequence W of X_i and Y, such that $|W| > |Z| = k$. Then W would be also a common subsequence of X_i and Y_j, violating the assumption that $c[i, j] = k$.

2) If $z_k \neq y_j$, similarly, we can "discard" y_j, and the optimal substructure is: $Z = LCS(X_i, Y_{j-1})$, and so $c[i, j] = c[i, j-1]$.

In the optimal solution either x_i or y_j are "discarded", which concisely can be written as: $c[i, j] = max(c[i-1, j], c[i, j-1])$.

Bottom-up DP:

```
LCS-LENGTH(X, Y, m, n)
  let b[1..m, 1..n] and c[0..m, o..n] be new tables
  for i = 1 to m
      c[i, 0] = 0
  for j = 0 to n
      c[0, j] = 0
  for i = 1 to m
      for j = 1 to n
          if x_i == y_j
              c[i, j] = c[i − 1, j − 1] + 1
              b[i, j] = "↖"
          else if c[i − 1, j] ≥ c[i, j − 1]
                  c[i, j] = c[i − 1, j]
                  b[i, j] = "↑"
              else c[i, j] = c[i, j − 1]
                  b[i, j] = "←"
  return c and b
```

Reconstructing the solution:

```
PRINT-LCS(b, X, i, j)
  if i == 0 or j = 0
      return
  if b[i, j] == "↖"
      PRINT-LCS(b, X, i − 1, j − 1)
      print x_i
  elseif b[i, j] == "↑"
      PRINT-LCS(b, X, i − 1, j)
  else PRINT-LCS(b, X, i, j − 1)
```

Also see
`https://nghiatran.me/longest-common-subsequence-diff-part-1/`
with examples and illustrations.

		A	B	C	D	A
	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
B	0	1	2	2	2	2
D	0	1	2	2	3	3
E	0	1	2	2	3	3
A	0	1	2	2	3	4

LCS - "ACDA"