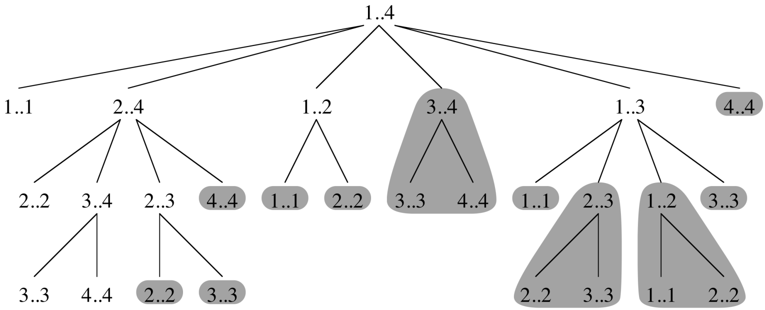
Dynamic Programming (DP)

Have we being doing too much work in our recursions?

Dynamic Programming (DP)

Have we being doing too much work in our recursions?



Dynamic Programming (DP)

Solving optimization (maximization or minimization) problems

Solving optimization (maximization or minimization) problems

1. Characterize the structure of an optimal solution.

Solving optimization (maximization or minimization) problems

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.

Dynamic Programming (DP)

Solving optimization (maximization or minimization) problems

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.

Dynamic Programming (DP)

Solving optimization (maximization or minimization) problems

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

Dynamic Programming (DP)

Solving optimization (maximization or minimization) problems

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

Step 4 is **not needed** if want only the value of the optimal solution.

Characterize the structure of an optimal solution.

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Divide-and-conquer algorithms:

1. No choice to define subproblems (e.g. split in halves).
2. Optimal solution of (the many) subproblems.
3. A theorem that combines (2) $\Rightarrow$ Optimal solution.

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Greedy algorithms:

1. Greedy choice (out of many) defines subproblem.
2. Optimal solution of (the one) subproblem.
3. A theorem that combines $(1) + (2) \Rightarrow$ Optimal solution.

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Dynamic Programming:

1. Best choice (out of many) defines subproblems.
2. Optimal solution of (the many) subproblems.
3. A theorem that combines $(1) + (2) \Rightarrow$ Optimal solution.

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

Divide-and conquer:

$$e.g. \quad OPT(P) = OPT(P/2) \uplus OPT(P/2)$$

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

Greedy:

$$OPT(P) = cost(g) + OPT(SP(g)), \text{ for greedy choice } g.$$

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

DP:

$OPT(P) = cost(b) + OPT(SP_1(b)) + \ldots + OPT(SP_k(b))$, for best b.

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

DP:

$OPT(P) = cost(b) + OPT(SP_1(b)) + \ldots + OPT(SP_k(b)),$ for best b.

Q: Wait a minute...which choice is the best choice b???

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

DP:

$OPT(P) = cost(b) + OPT(SP_1(b)) + \ldots + OPT(SP_k(b))$, for best b.

Q: Wait a minute...which choice is the best choice b???
A: *I don't know!*

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

DP:

$OPT(P) = cost(b) + OPT(SP_1(b)) + \ldots + OPT(SP_k(b))$, for best b.

Q: Wait a minute...which choice is the best choice b???
A: *I don't know!* Try them all and pick the one that gives you the best solution !

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

DP:

$$OPT(P) = \min_{b}\{cost(b) + OPT(SP_1(b)) + \ldots + OPT(SP_k(b))\}$$

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

Compute the value of an optimal solution.

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

Compute the value of an optimal solution.

Solution 1: We have the recursion, implement recursive (or iterative) algorithm.

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

Compute the value of an optimal solution.

Solution 1: We have the recursion, implement recursive (or iterative) algorithm.
Solution 2 (only for DP): Implement recursive algorithm but also use a **table** with optimal values of subproblems we have already solved.

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

Compute the value of an optimal solution.

Construct an optimal solution from computed information.

Dynamic Programming (DP)

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

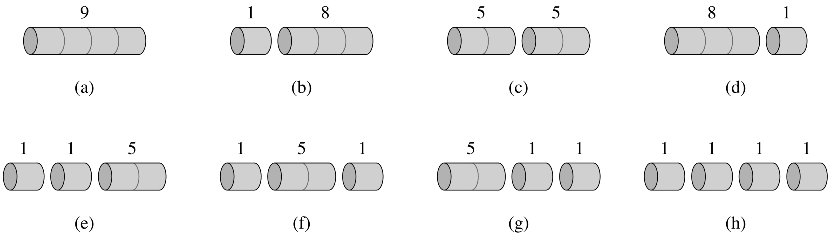
Compute the value of an optimal solution.

Construct an optimal solution from computed information.

Solution: Keep track of the best choice b in the recursion every time you find it!

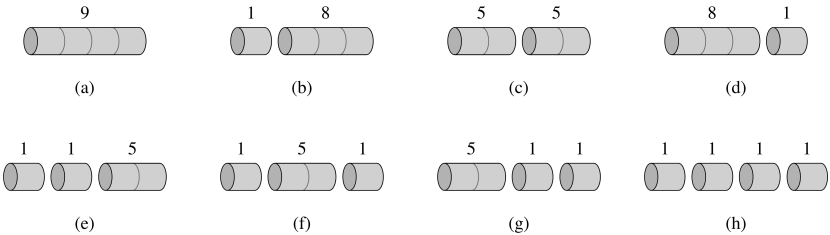
Example: Rod-Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



Example: Rod-Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



Note: There are 2^{n-1} ways to cut a rod of length n.

Dynamic Programming (DP)

Step 1: **Characterize the structure of an optimal solution.**

Optimal break: $i_1 + i_2 + \ldots + i_k = n$

Optimal revenue: $p_{i_1} + p_{i_2} + \ldots + p_{i_k} = r_n$

Dynamic Programming (DP)

Step 1: **Characterize the structure of an optimal solution.**

Optimal break: $i_1 + i_2 + \ldots + i_k = n$

Optimal revenue: $p_{i_1} + p_{i_2} + \ldots + p_{i_k} = r_n$

$\Rightarrow$ Best first cut of length i + optimal cutting of rest $n - i$

Dynamic Programming (DP)

Step 1: **Characterize the structure of an optimal solution.**

Optimal break: $i_1 + i_2 + \ldots + i_k = n$

Optimal revenue: $p_{i_1} + p_{i_2} + \ldots + p_{i_k} = r_n$

$\Rightarrow$ Best first cut of length i + optimal cutting of rest $n - i$

Step 2: **Recursively define the value of an optimal solution.**

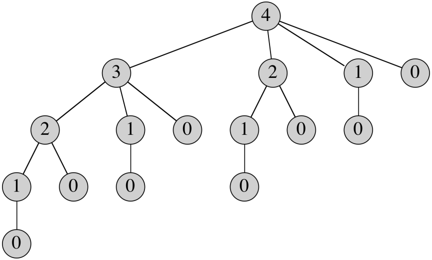
$$r_0 = 0, \quad r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\}$$

Step 3: **Compute the value of an optimal solution.**

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

Step 3: **Compute the value of an optimal solution.**

$\textsc{Cut-Rod}(p, n)$
  if $n == 0$
    return 0
  $q = -\infty$
  for $i = 1$ **to** n
    $q = \max(q,\ p[i] + \textsc{Cut-Rod}(p, n - i))$
  return q



Step 3: **Compute the value of an optimal solution.**

```
CUT-ROD(p, n)
  if n == 0
      return 0
  q = −∞
  for i = 1 to n
      q = max(q, p[i] + CUT-ROD(p, n − i))
  return q
```

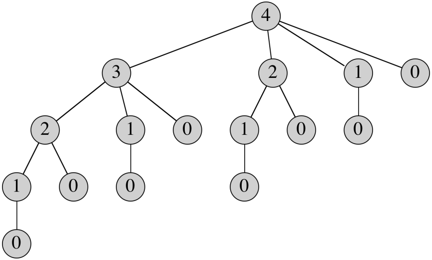
$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

Step 3: **Compute the value of an optimal solution.**

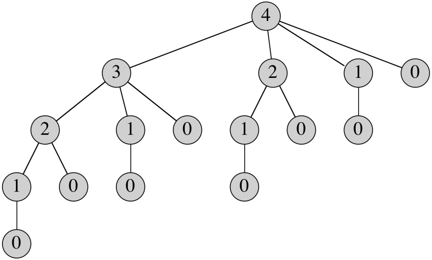
```
CUT-ROD(p, n)
  if n == 0
      return 0
  q = −∞
  for i = 1 to n
      q = max(q, p[i] + CUT-ROD(p, n − i))
  return q
```

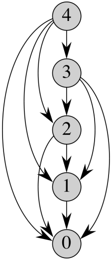
$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \Rightarrow T(n) = 2^n$$

Dynamic Programming (DP)

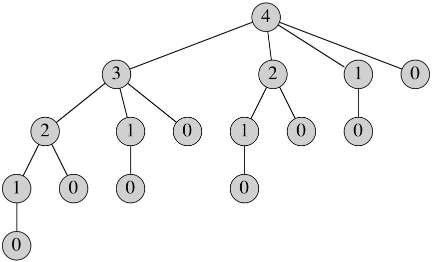


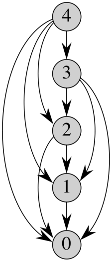
Dynamic Programming (DP)





Dynamic Programming (DP)





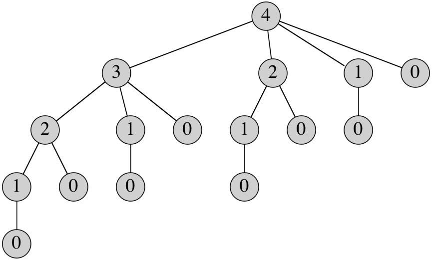
MEMOIZED-CUT-ROD(p, n)

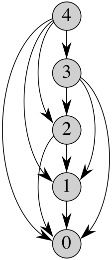
let $r[0..n]$ be a new array
for $i = 0$ **to** n
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)

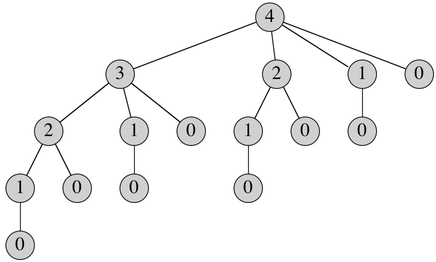
if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ **to** n
        $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$
$r[n] = q$
return q

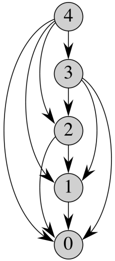
Dynamic Programming (DP)





```
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
```

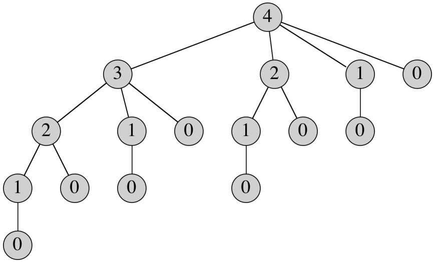


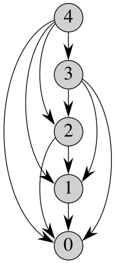


```
MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0
      return r[n]
  if n == 0
      q = 0
  else q = −∞
      for i = 1 to n
          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r
  r[n] = q
  return q
```

```
CUT-ROD(p, n)
  if n == 0
      return 0
  q = −∞
  for i = 1 to n
      q = max(q, p[i] + CUT-ROD(p, n − i))
  return q
```

Dynamic Programming (DP)





MEMOIZED-CUT-ROD-AUX(p, n, r)

if $r[n] \geq 0$
  return $r[n]$
if $n == 0$
  $q = 0$
else $q = -\infty$
  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-CUT-ROD(p, n)

let $r[0\,..\,n]$ be a new array
$r[0] = 0$
for $j = 1$ **to** n
  $q = -\infty$
  for $i = 1$ **to** j
    $q = \max(q, p[i] + r[j - i])$
  $r[j] = q$
return $r[n]$

Step 4: **Construct an optimal solution from computed information.**

Step 4: **Construct** an optimal solution from computed information.

```
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]
```

```
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
```

Dynamic Programming (DP)

Step 4: **Construct an optimal solution from computed information.**

```
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]
```

```
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
```

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

DP application: All-Pairs Shortest Paths

Assumption

No negative cycles

Assumption

No negative cycles

Input: Directed $G = (V, E)$, $n \times n$ weights matrix $W = [w_{ij}]$

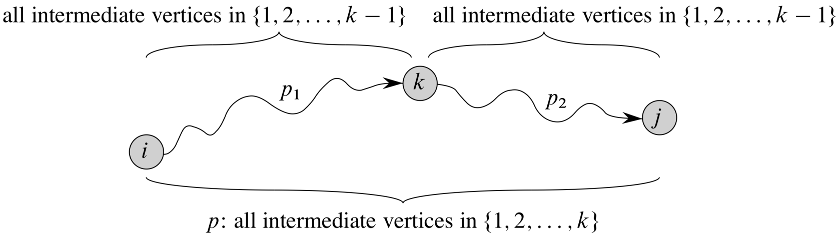
Output: $n \times n$ **distance** matrix $D = [d_{i,j}]$, and **predecessor** matrix $\Pi = [\pi_{ij}]$

DP application: All-Pairs Shortest Paths

Step 1: **Structure of shortest paths.**

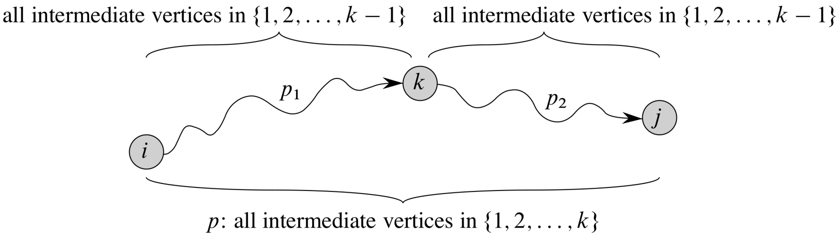
DP application: All-Pairs Shortest Paths

Step 1: **Structure of shortest paths.**



DP application: All-Pairs Shortest Paths

Step 1: **Structure of shortest paths.**



Step 2: **Recursively define the value of an optimal solution.**

$$d_{ij}^{(0)} = w_{ij}$$

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}, \quad k \geq 1$$

DP application: All-Pairs Shortest Paths

Step 3: Compute the value of an optimal solution.

FLOYD-WARSHALL(W, n)

$D^{(0)} = W$
for $k = 1$ **to** n
  let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
  for $i = 1$ **to** n
    for $j = 1$ **to** n
      $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Step 3: **Compute the value of an optimal solution.**

```
FLOYD-WARSHALL(W, n)
 D^(0) = W
 for k = 1 to n
     let D^(k) = (d_ij^(k)) be a new n × n matrix
     for i = 1 to n
         for j = 1 to n
             d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
 return D^(n)
```

Step 4: **Construct an optimal solution from computed information.**

$$\pi_{ij}^{(0)} = \begin{cases} NIL, & (i = j) \vee (w_{ij} = \infty) \\ i, & (i \neq j) \wedge (w_{ij} < \infty) \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)}, & \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} = d_{ij}^{(k-1)} \\ \pi_{kj}^{(k-1)}, & \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

DP application: Transitive closure

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

Output: Directed $G = (V, E^*)$ where

$$E^* = \{(i, j) : \text{there is a path from } i \text{ to } j \text{ in } G\}$$

DP application: Transitive closure

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

Output: Directed $G = (V, E^*)$ where

$$E^* = \{(i, j) : \text{there is a path from } i \text{ to } j \text{ in } G\}$$

Can apply Floyd-Warshall indirectly, or directly:

$$t_{ij}^{(0)} = \begin{cases} 1, & (i = j) \vee ((i, j) \in E) \\ 0, & (i \neq j) \wedge ((i, j) \notin E) \end{cases}$$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}), \quad k \geq 1$$

DP application: Transitive closure

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

Output: Directed $G = (V, E^*)$ where

$$E^* = \{(i, j) : \text{there is a path from } i \text{ to } j \text{ in } G\}$$

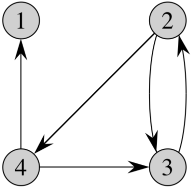
Can apply Floyd-Warshall indirectly, or directly:

$$t_{ij}^{(0)} = \begin{cases} 1, & (i = j) \vee ((i, j) \in E) \\ 0, & (i \neq j) \wedge ((i, j) \notin E) \end{cases}$$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}), \quad k \geq 1$$

(compare to $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$)

DP application: Transitive closure



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$