Dynamic Programming (DP)

A typical algorithm:

1. Make some decision(s)
2. Problem is broken into k subproblems $(n_1, \ldots, n_k)$
3. Solve k subproblems recursively
4. Combine decision(s) from (1) with solutions from (3), to output solution

Dynamic Programming (DP)

GREEDY:

1. Make greedy choice g
2. Problem is reduced into one subproblem
3. Solve subproblem recursively $\Rightarrow SOL_{sub}$
4. Combine choice from (1) with solution from (3), to output solution SOL

Dynamic Programming (DP)

GREEDY:

1. Make greedy choice g
2. Problem is reduced into one subproblem
3. Solve subproblem recursively $\Rightarrow SOL_{sub}$
4. Combine choice from (1) with solution from (3), to output solution SOL

Correctness:

Theorem

If we have that

1. *greedy choice g is part of an OPT solution*
2. *$SOL = g \cup SOL_{sub}$ is a feasible solution*

then SOL is an optimal solution (i.e., greedy alg is correct).

Dynamic Programming (DP)

GENERAL:

1. Make choice g
2. Problem is reduced into one subproblem
3. Solve subproblem recursively $\Rightarrow SOL_{sub}$
4. Combine choice from (1) with solution from (3), to output solution SOL

Correctness:

Theorem

If we have that

1. *choice g is part of an OPT solution*
2. *$SOL = g \cup SOL_{sub}$ is a feasible solution*

then SOL is an optimal solution (i.e., general alg is correct).

EVEN-MORE-GENERAL:

1. Make choice g
2. Problem is reduced into k subproblems $n_1, n_2, \ldots, n_k$
3. Solve subproblems recursively $\Rightarrow SOL_1, SOL_2, \ldots, SOL_k$
4. Combine choice from (1) with solutions from (3), to output solution SOL

Correctness:

Theorem

If we have that

1. *choice g is part of an OPT solution*
2. *$SOL = g \cup SOL_1 \cup SOL_2 \cup \ldots \cup SOL_k$ is a feasible solution*

then SOL is an optimal solution (i.e., general alg is correct).

Dynamic Programming (DP)

EVEN-MORE-GENERAL:

1. Make choice g
2. Problem is reduced into k subproblems $n_1, n_2, \ldots, n_k$
3. Solve subproblems recursively $\Rightarrow SOL_1, SOL_2, \ldots, SOL_k$
4. Combine choice from (1) with solutions from (3), to output solution SOL

Our only problem is...

EVEN-MORE-GENERAL:

1. Make choice g
2. Problem is reduced into k subproblems $n_1, n_2, \ldots, n_k$
3. Solve subproblems recursively $\Rightarrow SOL_1, SOL_2, \ldots, SOL_k$
4. Combine choice from (1) with solutions from (3), to output solution SOL

Our only problem is...**which choice g to make?**

Dynamic Programming (DP)

EVEN-MORE-GENERAL:

1. Make choice g
2. Problem is reduced into k subproblems $n_1, n_2, \ldots, n_k$
3. Solve subproblems recursively $\Rightarrow SOL_1, SOL_2, \ldots, SOL_k$
4. Combine choice from (1) with solutions from (3), to output solution SOL

Our only problem is...**which choice g to make?**
We need to know (an) OPT... vicious circle!

Dynamic Programming (DP)

EVEN-MORE-GENERAL:

1. Make choice g
2. Problem is reduced into k subproblems $n_1, n_2, \ldots, n_k$
3. Solve subproblems recursively $\Rightarrow SOL_1, SOL_2, \ldots, SOL_k$
4. Combine choice from (1) with solutions from (3), to output solution SOL

Our only problem is...**which choice g to make?**
We need to know (an) OPT... vicious circle!

- Note that we had the answer for both D&C (**no** choice!), and GREEDY (**greedy** choice both computable and part of OPT).

Dynamic Programming (DP)

EVEN-MORE-GENERAL:

1. Make choice g
2. Problem is reduced into k subproblems $n_1, n_2, \ldots, n_k$
3. Solve subproblems recursively $\Rightarrow SOL_1, SOL_2, \ldots, SOL_k$
4. Combine choice from (1) with solutions from (3), to output solution SOL

Our only problem is...**which choice g to make?**
We need to know (an) OPT... vicious circle!

- Note that we had the answer for both D&C (**no** choice!), and GREEDY (**greedy** choice both computable and part of OPT).
- Brute force: Try **all** g!

Dynamic Programming (DP)

EVEN-MORE-GENERAL:

1. Make choice g
2. Problem is reduced into k subproblems $n_1, n_2, \ldots, n_k$
3. Solve subproblems recursively $\Rightarrow$ $SOL_1, SOL_2, \ldots, SOL_k$
4. Combine choice from (1) with solutions from (3), to output solution SOL

Our only problem is...**which choice g to make?**
We need to know (an) OPT... vicious circle!

- Note that we had the answer for both D&C (**no** choice!), and GREEDY (**greedy** choice both computable and part of OPT).
- Brute force: Try **all** g! **NOOOOOOOOOOOO!!!**

Dynamic Programming (DP)

BRUTE-FORCE:

`forall` choice g `do`

1. Problem is reduced into k subproblems $n_1, n_2, \ldots, n_k$
2. Solve subproblems recursively $\Rightarrow SOL_1, SOL_2, \ldots, SOL_k$
3. Combine choice from (1) with solutions from (3), to output solution $SOL[g]$

`endfor`

Dynamic Programming (DP)

BRUTE-FORCE:

`forall` choice g `do`

1. Problem is reduced into k subproblems $n_1, n_2, \ldots, n_k$
2. Solve subproblems recursively $\Rightarrow SOL_1, SOL_2, \ldots, SOL_k$
3. Combine choice from (1) with solutions from (3), to output solution $SOL[g]$

`endfor`

`Output` $\min / \max_g SOL[g]$

Dynamic Programming (DP)

BRUTE-FORCE:

`forall` choice g `do`

1. Problem is reduced into k subproblems $n_1, n_2, \ldots, n_k$
2. Solve subproblems recursively $\Rightarrow SOL_1, SOL_2, \ldots, SOL_k$
3. Combine choice from (1) with solutions from (3), to output solution $SOL[g]$

`endfor`

`Output` $\min / \max_g SOL[g]$

- Note than now last step is **computable**. All you have to do is solve for **all** g, store **all** solutions $SOL[g]$, and find the min.

BRUTE-FORCE:

`forall` choice g `do`

1. Problem is reduced into k subproblems $n_1, n_2, \ldots, n_k$
2. Solve subproblems recursively $\Rightarrow$ $SOL_1, SOL_2, \ldots, SOL_k$
3. Combine choice from (1) with solutions from (3), to output solution $SOL[g]$

`endfor`

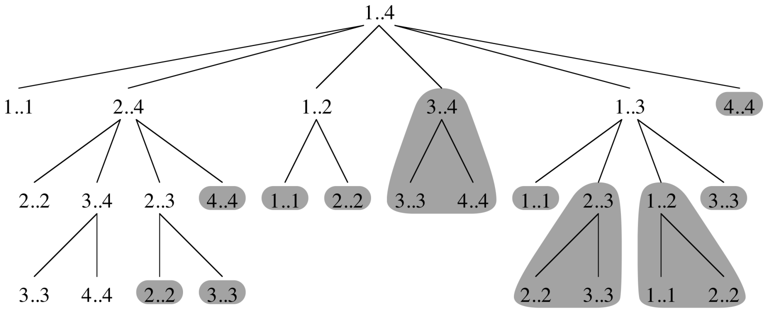
`Output` $\min / \max_g SOL[g]$

- Note than now last step is **computable**. All you have to do is solve for **all** g, store **all** solutions $SOL[g]$, and find the min.
- **NOOOOOOOOOOOO!!!**

Can we avoid doing too much work in our brute-force recursions?

Dynamic Programming (DP)

Can we avoid doing too much work in our brute-force recursions?



Solving optimization (maximization/minimization) problems

Solving optimization (maximization/minimization) problems

1. Characterize the structure of an optimal solution.

Solving optimization (maximization/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/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/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/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.

Dynamic Programming (DP)

Solving optimization (maximization/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.
- To implement Step 4, just keep track of the best g over all iterations of the loop.

Dynamic Programming (DP)

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

Dynamic Programming (DP)

1. Characterize the structure of an optimal solution.

Divide-and-conquer:

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)

1. Characterize the structure of an optimal solution.

Greedy:

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)

1. Characterize the structure of an optimal solution.

DP & Brute-force:

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)

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

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

Dynamic Programming (DP)

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

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

Divide-and conquer:

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

Dynamic Programming (DP)

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

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

Greedy:

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

Dynamic Programming (DP)

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

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

DP & Brute-force:

$$OPT(P) = \min_{g}\{cost(g) + SOL_1(g) + \ldots + SOL_k(g)\}$$

Dynamic Programming (DP)

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

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

Dynamic Programming (DP)

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)

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.**
   - We have the recursion, implement recursive (or iterative) algorithm.

Dynamic Programming (DP)

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.**
   - We have the recursion, implement recursive (or iterative) algorithm.
   - (only for DP) Use a **table** with optimal values of subproblems we have already solved.

Dynamic Programming (DP)

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)

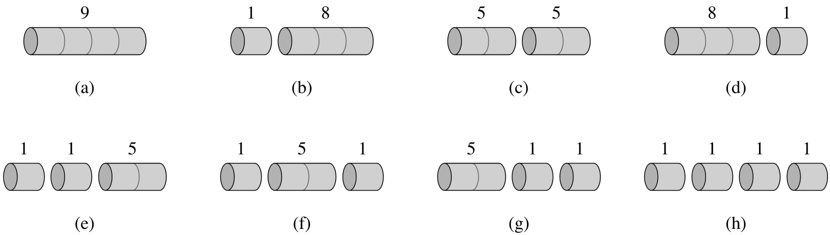
1. **Characterize the structure of an optimal solution.**
2. **Recursively define the value of an optimal solution.**
3. **Compute the value of an optimal solution.**
4. **Construct an optimal solution from computed information.**

Dynamic Programming (DP)

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.**
   - (DP & Brute-force) Keep track of the best choice g over all for-loop iterations.

Example: Rod-Cutting

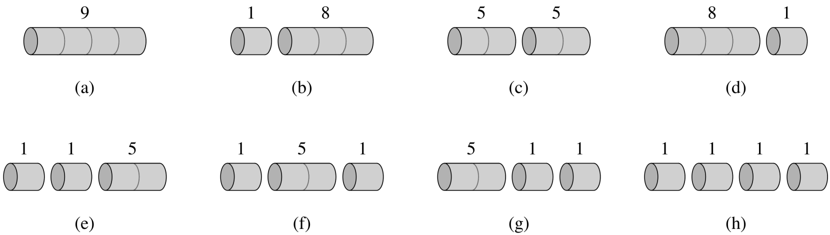
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



Dynamic Programming (DP)

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.

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 $i \cup$ 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 $\cup$ optimal cutting of rest $n - i$

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

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

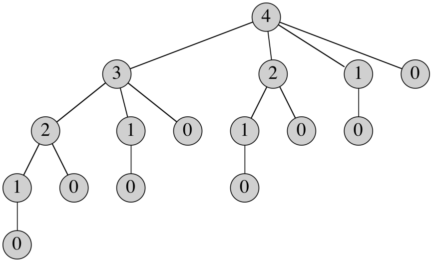
Step 3: **Compute the value of an optimal solution.**

```
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
```

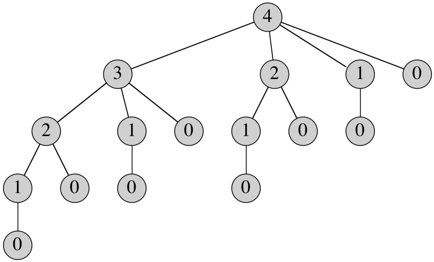
$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

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

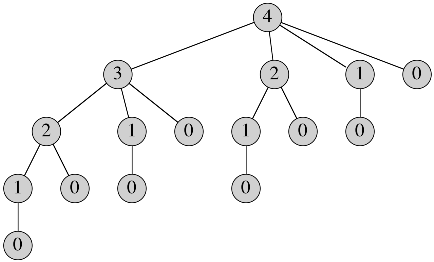
```
CUT-ROD(p, n)
  if n == 0
      return 0
  q = −∞
  for i = 1 to n
      q = max(q, p[i] + CUT-ROD(p, n − i))
  return q
```

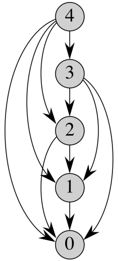
$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \Rightarrow T(n) = 2^n$$

Dynamic Programming (DP)

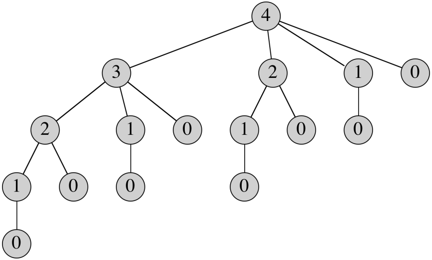


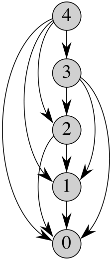
Dynamic Programming (DP)





Dynamic Programming (DP)

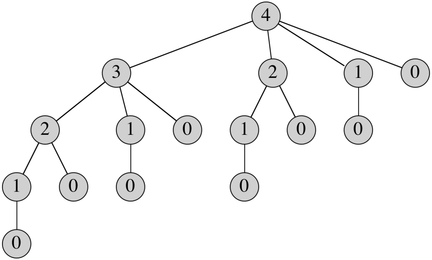


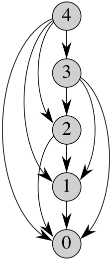



```
MEMOIZED-CUT-ROD(p, n)
  let r[0..n] be a new array
  for i = 0 to n
      r[i] = -∞
  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

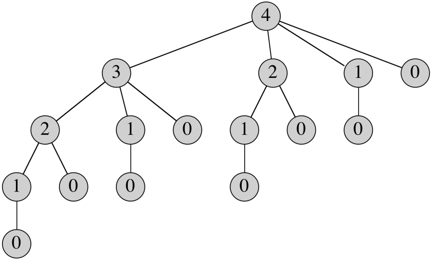
```
MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0
      return r[n]
  if n == 0
      q = 0
  else q = -∞
      for i = 1 to n
          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
  r[n] = q
  return q
```

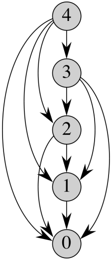
Dynamic Programming (DP)





```
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
```

```
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]
```

Dynamic Programming (DP)

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