Dynamic Programming

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

Problem Solving Strategies

- **Divide and Conquer**
  - Divide (reduce) the problem into one or more subproblems;
  - Recursively solve subproblems;
  - Combine partial solutions to obtain complete solution.
  - **Example:** merge-sort, quick-sort, binary-search, …

- **Greedy**

  Greedy choice property. (Could be hard to identify.)

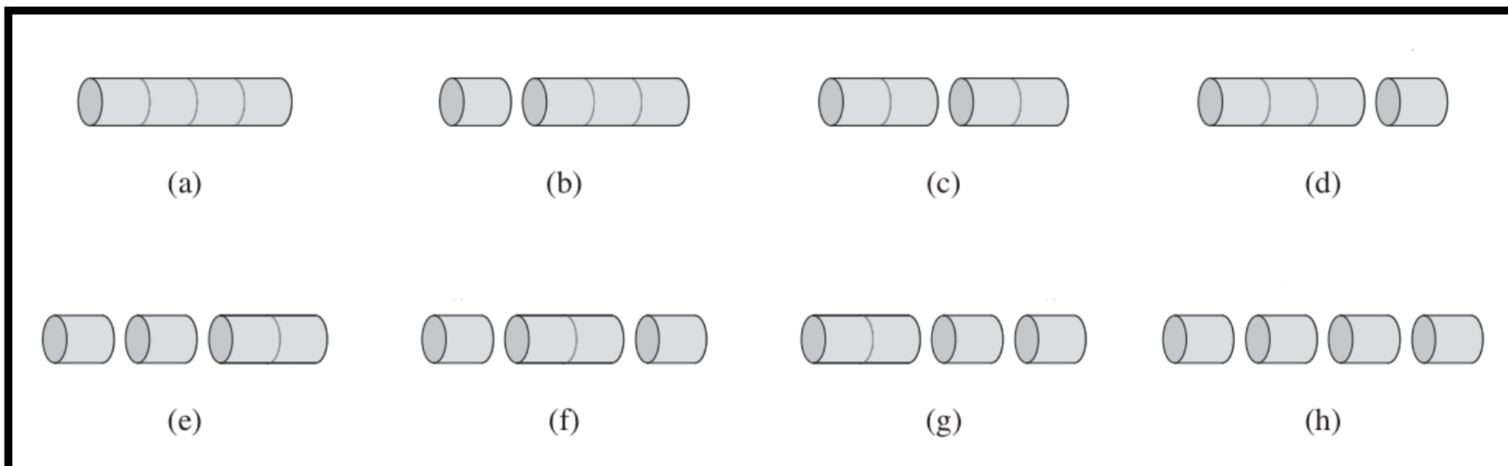
  - Gradually generate a solution for the problem;
  - At each step: make an optimal choice, then compute optimal solution of the subproblem induced by the choice made.
  - **Example:** MST, Dijkstra, Huffman codes, …

rty.

What if a problem does not exhibit greedy choice property?

The Rod-Cutting Problem

- Assume we are given a rod of length n. We sell length i rod for a price of p_i, where $i \in \mathbb{N}^+$ and $1 \leq i \leq n$.
- How to cut the rod to gain maximum revenue?
- Enumerate all possibilities?
- There are 2^{n-1} ways to cut up a length n rod…



The Rod-Cutting Problem

- Assume we are given a rod of length n. We sell length i rod for a price of p_i, where $i \in \mathbb{N}^+$ and $1 \le i \le n$.
- How to cut the rod to gain maximum revenue?
- Greedy algorithm?
- Let r_k denote max profit for a length k rod.
- Optimal substructure property?
- $r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$
- ~~Greedy choice property?~~
- Always cut at the most profitable position? $(\max(p_i/i))$
- Does NOT yield optimal solution! $(n = 3, p_1 = 1, p_2 = 7, p_3 = 9)$

The Rod-Cutting Problem

- Assume we are given a rod of length n. We sell length i rod for a price of p_i, where $i \in \mathbb{N}^+$ and $1 \le i \le n$.
- How to cut the rod to gain maximum revenue?

- Simple recursive algorithm.
- Let r_k denote max profit for a length k rod.

CutRodRec(prices,n):

Performance of this algorithm?

```
if (n==0)
  return 0
r = -INF
for (i=1 to n)
  r = Max(r, prices[i]+CutRodRec(prices,n-i))
return r
```

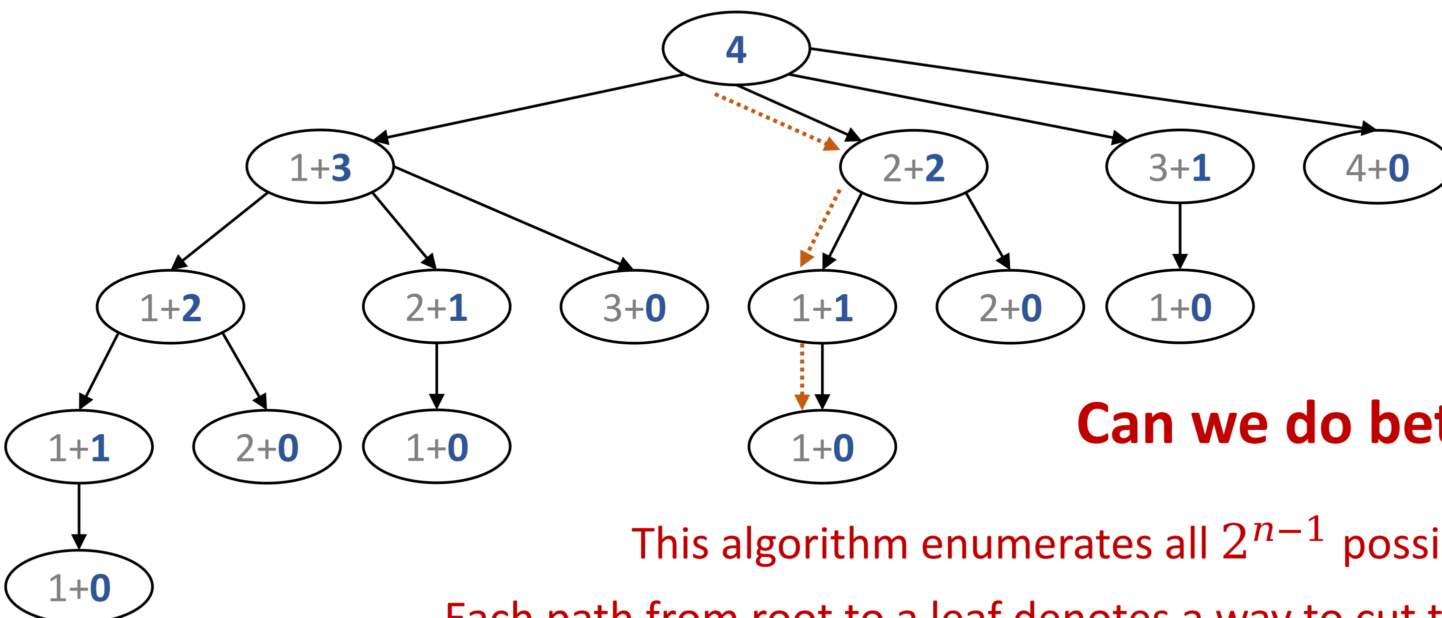
  - For each cut, (recursively) find optimal solution. (Find all r_{n-i})
  - Find optimal solution for original problem. (Find $\max(p_i + r_{n-i})$)

The Rod-Cu

CutRodRec(prices,n):

```
if (n==0)
  return 0
r = -INF
for (i=1 to n)
  r = Max(r, prices[i]+CutRodRec(prices,n-i))
return r
```

- Assume we are given a rod of length n. We sell length i rod for a price of p_i, where $i \in \mathbb{N}^+$ and $1 \leq i \leq n$.
- How to cut the rod to gain maximum revenue?
- Optimal substructure property gives a simple recursive alg.



Can we do better?!

This algorithm enumerates all 2^{n-1} possibilities!

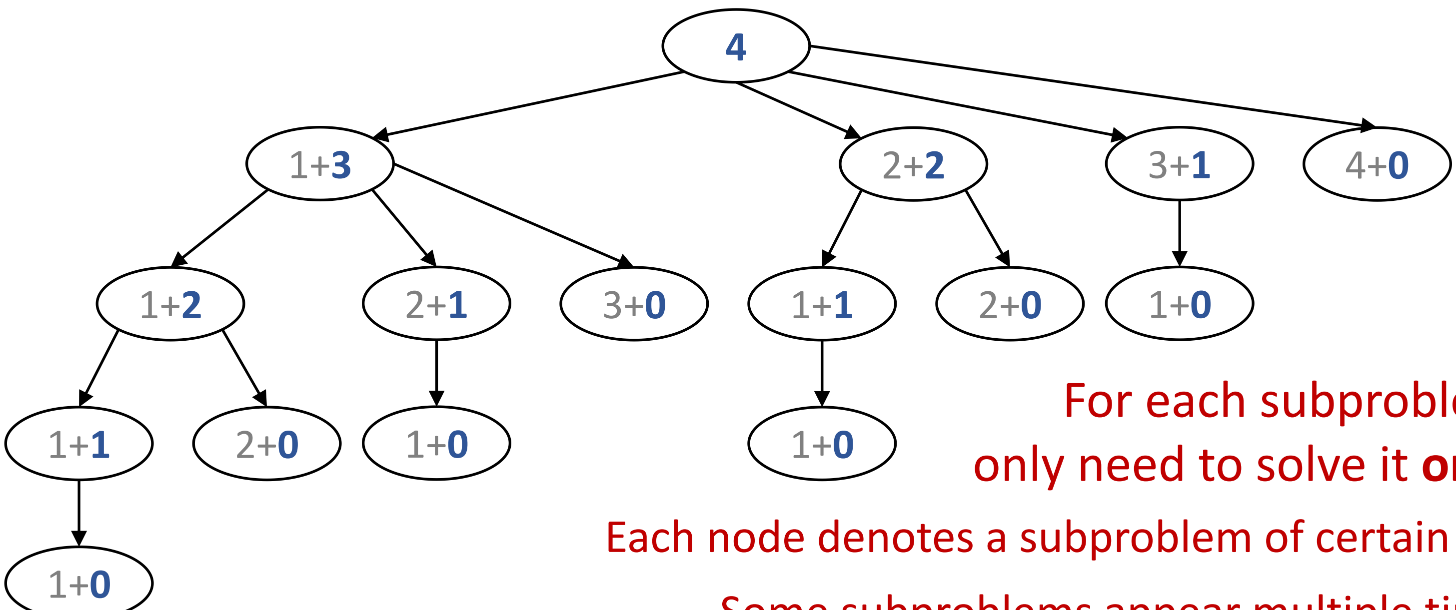
Each path from root to a leaf denotes a way to cut the rod.

The Rod-Cu

CutRodRec(prices,n):

```
if (n==0)
  return 0
r = -INF
for (i=1 to n)
  r = Max(r, prices[i]+CutRodRec(prices,n-i))
return r
```

- Assume we are given a rod of length n. We sell length i rod for a price of p_i, where $i \in \mathbb{N}^+$ and $1 \leq i \leq n$.
- How to cut the rod to gain maximum revenue?
- Optimal substructure property gives a simple recursive alg.



For each subproblem, only need to solve it **once**!

Each node denotes a subproblem of certain size.

Some subproblems appear multiple times.

The Rod-Cutting Problem

- Assume we are given a rod of length n. We sell length i rod for a price of p_i, where $i \in \mathbb{N}^+$ and $1 \leq i \leq n$.
- How to cut the rod to gain maximum revenue?
- Optimal substructure property gives a simple recursive alg.
- Simple recursion solves same subproblem multiple times.
- Solve each subproblem ***once*** and ***remember*** solution!

CutRodRecMemAux(prices,r,n):

```
if (r[n]>0)
  return r[n]
if (n==0)
  q = 0
else
  q = -INF
  for (i=1 to n)
    q = Max(q, prices[i]+CutRodRecMemAux(prices,r,n-i))
r[n] = q
return q
```

CutRodRecMem(prices,n):

```
for (i=0 to n)
  r[i] = -INF
return CutRodRecMemAux(prices,r,n)
```

The Rod-Cutting Problem

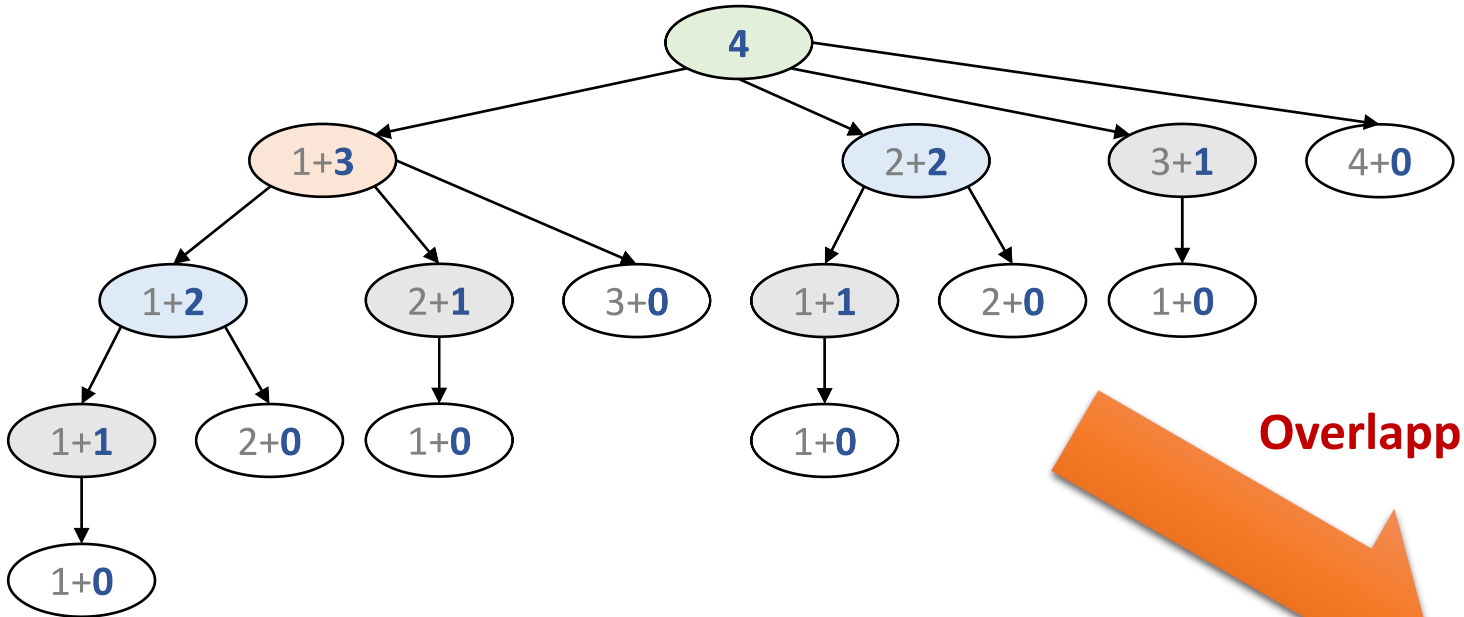
- Runtime of this algorithm?
- Each subproblem (optimal revenue for length i rod) is solved **once**.
- When actually solving the size i problem, optimal solutions of subproblems are known. (Otherwise we would recurse first.)
- Thus solving size i problem needs $\Theta(i)$ time.
- Total runtime is $\Theta(1 + 2 + \cdots + n) = \Theta(n^2)$.

CutRodRecMemAux(prices,r,n):

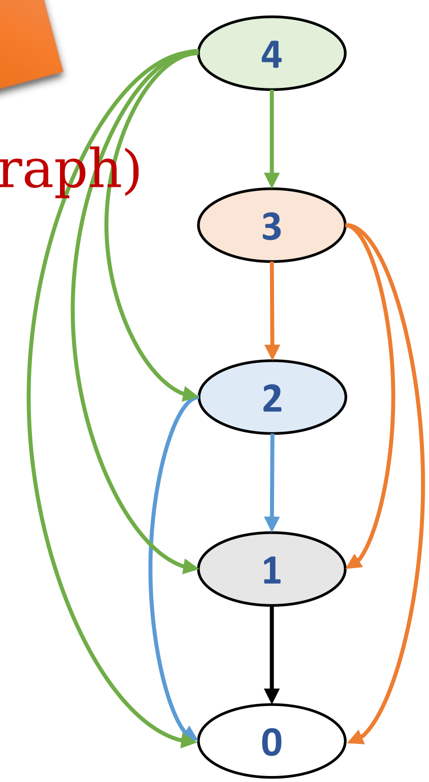
```
if (r[n]>0)
  return r[n]
if (n==0)
  q = 0
else
  q = -INF
  for (i=1 to n)
    q = Max(q, prices[i]+CutRodRecMemAux(prices,r,n-i))
r[n] = q
return q
```

CutRodRecMem(prices,n):

```
for (i=0 to n)
  r[i] = -INF
return CutRodRecMemAux(prices,r,n)
```



Overlapping subproblems



Runtime = Θ(# of lines and nodes in subproblem graph)

CutRodRecMemAux(prices,r,n):

```
if (r[n]>0)
  return r[n]
if (n==0)
  q = 0
else
  q = -INF
  for (i=1 to n)
    q = Max(q, prices[i]+CutRodRecMemAux(prices,r,n-i))
r[n] = q
return q
```

CutRodRecMem(prices,n):

```
for (i=0 to n)
  r[i] = -INF
return CutRodRecMemAux(prices,r,n)
```

The Rod-Cutting Problem

The Top-Down Approach

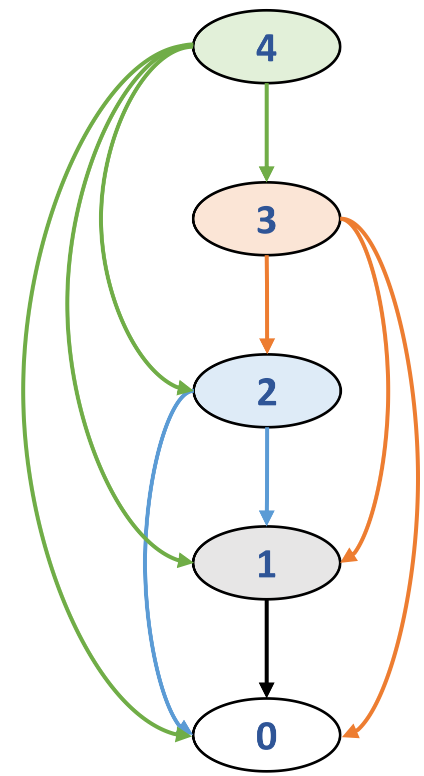
CutRodRecMemAux(prices,r,n):

```
if (r[n]>0)
  return r[n]
if (n==0)
  q = 0
else
  q = -INF
  for (i=1 to n)
    q = Max(q, prices[i]+CutRodRecMemAux(prices,r,n-i))
r[n] = q
return q
```

CutRodRecMem(prices,n):

```
for (i=0 to n)
  r[i] = -INF
return CutRodRecMemAux(prices,r,n)
```

- **The subproblem graph is a DAG!** (WHY?)
- Solving the problem using recursion is like DFS.
- Convert recursion to iteration?



The Rod-Cutting Problem

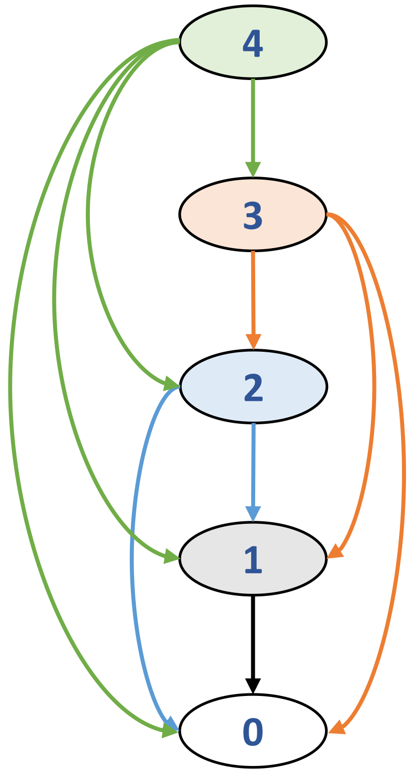
The Bottom-Up Approach

- Convert recursion to iteration?
- The subproblem graph is a DAG.
- A problem cannot be solved until all subproblems it depends upon are solved.
- Consider subproblems in reverse topo order!

CutRodIter(prices,n):

Runtime is $\Theta(n^2)$

```
r[0] = 0
for (i=1 to n)
  q = -INF
  for (j=1 to i)
    q = Max(q, prices[j] + r[i-j])
  r[i] = q
return r[n]
```



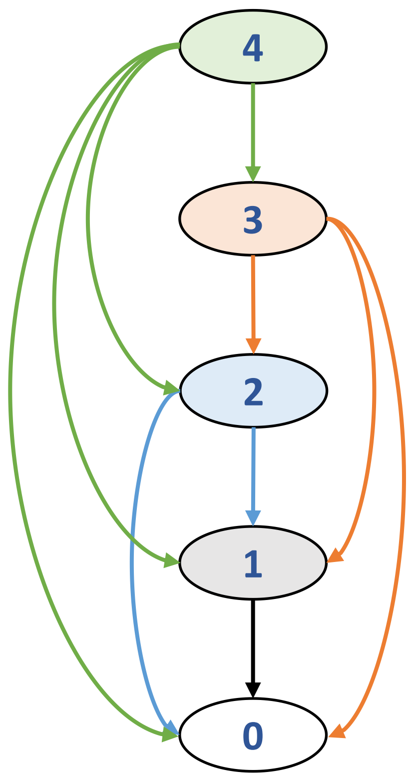
The Rod-Cutting Problem

The Bottom-Up Approach

- Convert recursion to iteration?
- Consider subproblems in reverse topo order!

CutRodIter(prices,n):

```
r[0] = 0
for (i=1 to n)
  q = -INF
  for (j=1 to i)
    q = Max(q, prices[j] + r[i-j])
  r[i] = q
return r[n]
```



- Or, inspect the recurrence more carefully!
- $r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$

The Rod-Cutting Problem

Reconstructing optimal solution

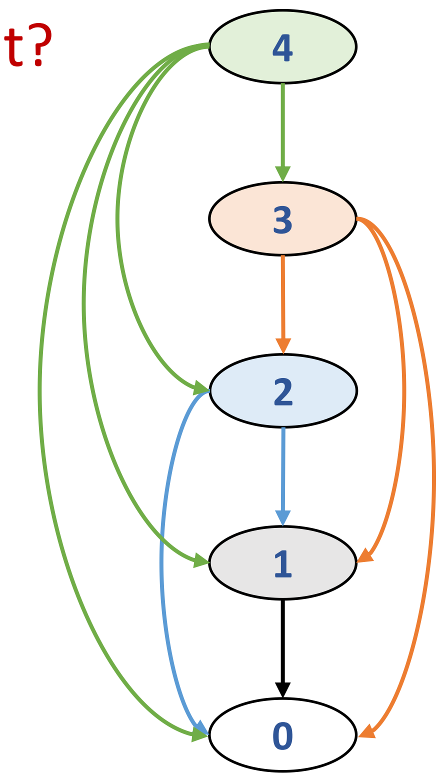
- Assume we are given a rod of length n. We sell length i rod for a price of p_i, where $i \in \mathbb{N}^+$ and $1 \leq i \leq n$.
- How to cut the rod to gain maximum revenue?
- Algorithm gives optimal revenue, but how to cut?

CutRodIter(prices,n):

```
r[0] = 0
for (i=1 to n)
  q = -INF
  for (j=1 to i)
    if (q < prices[j] + r[i-j])
      q = prices[j] + r[i-j]
      cuts[i] = j
  r[i] = q
return r[n]
```

PrintOpt(cuts,n):

```
while (n>0)
  Print cuts[n]
  n = n - cuts[n]
```



Dynamic Programming (DP)

- Consider an (optimization) problem:
  - Build optimal solution step by step.
  - Problem has **optimal substructure** property.
    - We can design a recursive algorithm.
  - Problem has lots of **overlapping subproblems**.
    - Recursion and *memorize* solutions. (Top-Down)
    - Or, consider subproblems in the *right order*. (Bottom-Up)
- We have seen such algorithms previously!

APSP via Dynamic Programming

The Floyd-Warshall Algorithm

- **Strategy:** recuse on the *set of node* the shortest paths use.
- Define $dist(u, v, r)$ be length of shortest path from u to v, s.t. only nodes in $V_r = \{x_1, x_2, \cdots, x_r\}$ can be intermediate nodes in paths.
- $dist(u, v, r) =$

$$\begin{cases} w(u,v) & \text{if } r = 0 \text{ and } (u,v) \in E \\ \infty & \text{if } r = 0 \text{ and } (u,v) \notin E \\ \min \begin{Bmatrix} dist(u, v, r-1) \\ dist(u, x_r, r-1) + dist(x_r, v, r-1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

FloydWarshallAPSP(G):

Bottom-up Approach.

```
for (every pair (u,v) in V*V)
  if ((u,v) in E) then dist[u,v,0]=w(u,v)
  else dist[u,v,0]=INF
for (r=1 to n)
  for (each node u)
    for (each node v)
      dist[u,v,r] = dist[u,v,r-1]
      if (dist[u,v,r] > dist[u,x_r,r-1] + dist[x_r,v,r-1])
        dist[u,v,r] = dist[u,x_r,r-1] + dist[x_r,v,r-1]
```

Developing a DP algorithm

- **Characterize the structure of solution.**
  - E.g. [rod-cutting]: (one cut of length i) + (solution for length $n - i$)
- **Recursively define the value of an optimal solution.**
  - E.g. [rod-cutting]: $r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$
- **Compute the value of an optimal solution.**
  - Top-down or Bottom-up. (Usually use bottom-up.)
- **[$\ast$] Construct an optimal solution.**
  - Remember optimal choices (beside optimal solution values).

Matrix-chain Multiplication

- **Input:** Matrices $A_1,\ A_2, \cdots, A_n$, with A_i of size $p_{i-1} \times p_i$.
- **Output:** $A_1 A_2 \cdots A_n$.
- **Problem:** Compute output with minimum work?

- Matrix multiplication is associative, and order does matter!
- ***Example***: $|A_1| = 10 \times 100, |A_2| = 100 \times 5, |A_3| = 5 \times 50$
- $(A_1 A_2) A_3$ costs $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$
- $A_1 (A_2 A_3)$ costs $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$

- Optimal order for minimum cost?

Matrix-chain Multiplication

Developing a DP algorithm

- **Input:** Matrices $A_1,\ A_2, \cdots, A_n$, with A_i of size $p_{i-1} \times p_i$.
- **Problem:** Compute $A_1 A_2 \cdots A_n$ with minimum work?
- **Characterize the structure of solution.**
  - What's the last step in computing $A_1 A_2 \cdots A_n$?
  - For every order, last step is $(A_1 A_2 \cdots A_k) \cdot (A_{k+1} A_{k+2} \cdots A_n)$.
  - In general, $A_i A_{i+1} \cdots A_j = (A_i A_{i+1} \cdots A_k) \cdot (A_{k+1} A_{k+2} \cdots A_j)$
- **Recursively define the value of an optimal solution.**
  - Let $m[i, j]$ be min cost for computing $A_i A_{i+1} \cdots A_j$
  - $m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$
    - Optimal Substructure Property!

Matrix

Dev... am

MatrixChainDP($A_1, A_2, \ldots, A_n$):

```
for (i=1 to n)
  m[i,i] = 0
for (l=2 to n)
  for (i=1 to n-l+1)
    j = i+l-1
    m[i,j] = INF
    for (k=i to j-1)
      cost = m[i,k] + m[k+1,j] + p_{i-1}*p_k*p_j
      if (cost < m[i,j])
        m[i,j] = cost
return m
```

Runtime is $O(n^3)$.

Space cost is $O(n^2)$.

- **Input:** ... p_i.
- **Proble...**
- **Chara...**
- **Recursively define the value of an optimal solution.**
  - Let $m[i,j]$ be min cost for computing $A_i A_{i+1} \cdots A_j$
  - $m[i,j] = \min_{i \le k < j} \left(m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \right)$
- **Compute the value of an optimal solution.**
  - Top-down (recursion with memorization) is easy, but bottom-up?
  - What does $m[i,j]$ depend upon?
  - $m[i,j]$ depend upon $m[i',j']$ where $j' - i' < j - i$.
  - Compute $m[i,j]$ in *length increasing* order!

Matrix-chain Multiplication

Deve… n

- **Input:** Ma…
- **Problem:** …
- **Charact…**
- **Recursi…** …n.
  - Let m…
  - $m[i,$ … $j)$
- **Comput…**
  - Com…
- **Constru…**
  - For e… …al “split”.

MatrixChainDP(A_1, A_2,…,A_n):

```
for (i=1 to n)
  m[i,i] = 0
for (l=2 to n)
  for (i=1 to n-l+1)
    j = i+l-1
    m[i,j] = INF
    for (k=i to j-1)
      cost = m[i,k] + m[k+1,j] + p_{i-1}*p_k*p_j
      if (cost < m[i,j])
        m[i,j] = cost
        s[i,j] = k
return <m,s>
```

MatrixChainPrintOpt(s,i,j):

```
if (i==j)
  Print("A_i")
else
  Print("(")
  MatrixChainPrintOpt(s,i,s[i,j])
  MatrixChainPrintOpt(s,s[i,j]+1,j)
  Print(")")
```

Edit Distance

- Given two strings, how *similar* are they?
- **Application:** when a spell checker encounters a possible misspelling, it needs to search dictionary to find *nearby* words.
- Consider following three type of operations for a string:
  - **Insertion:** insert a character at a position.
  - **Deletion:** remove a character at a position.
  - **Substitution:** change a character to another character.
- Edit Distance of A and B: min # of ops to transform A into B.
- *Example*: transform "`SNOWY`" to "`SUNNY`"
  - Insertion: `SNOWY -> SUNOWY`
  - Deletion: `SUNOWY -> SUNOY`
  - Substitution: `SUNOY -> SUNNY`
  - Edit distance is *at most* 3 (and it indeed *is* 3).

Edit Distance

- Edit Distance of A and B: min # of ops to transform A into B.
  - Operations: **Insertion**, **Deletion**, and **Substitution**.
- One way to visualize the editing process:
  - Align string A above string B;
  - A gap in first line indicates an insertion (to A);
  - A gap in second line indicates a deletion (from A);
  - A column with different characters indicates a substitution.

S		N	O	W	Y
S	U	N	N		Y

Edit Distance

- Edit Distance: min # of ins/del/sub to transform A into B.
- **Problem:** Given A and B, what is the edit distance?
- **Step 1: Characterize the structure of solution.**
  - Consider transform $A[1\cdots m]$ to $B[1\cdots n]$
  - Each solution can be visualized in the way described earlier.
  - Last column must be one of three cases: $\begin{array}{c} - \\ B[n] \end{array}$ or $\begin{array}{c} A[m] \\ B[n] \end{array}$ or $\begin{array}{c} A[m] \\ - \end{array}$
  - Each case reduces the problem to a subproblem:
    - $(-, B[n])$: edit distance of $A[1\cdots m]$ and $B[1\cdots (n-1)]$
    - $(A[m], B[n])$: edit distance of $A[1\cdots (m-1)]$ and $B[1\cdots (n-1)]$
    - $(A[m], -)$: edit distance of $A[1\cdots (m-1)]$ and $B[1\cdots n]$

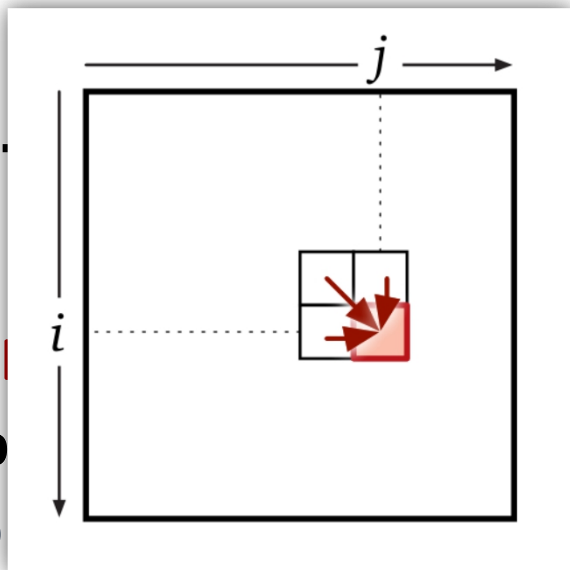
S	–	N	O	W	Y
S	U	N	N	–	Y

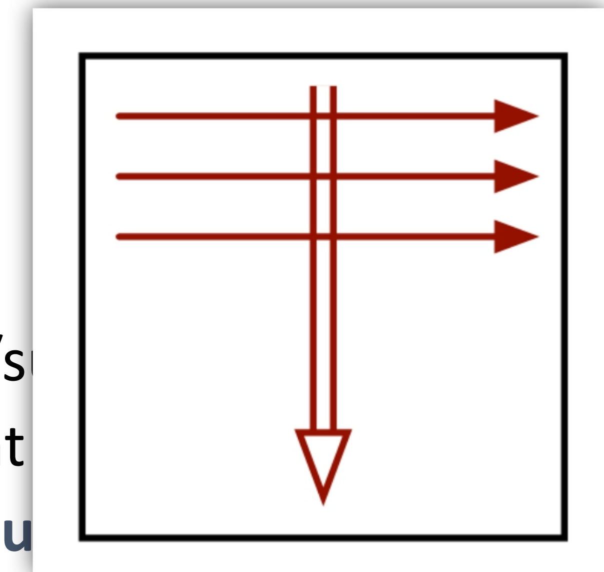
Edit Distance

- Edit Distance: min # of ins/del/sub to transform A into B.
- **Problem:** Given A and B, what is the edit distance?
- **Step 1: Characterize the structure of solution.**
  - Removing last column reduces the problem to a subproblem:
    - $(-, B[n])$: edit distance of $A[1 \cdots m]$ and $B[1 \cdots (n-1)]$
    - $(A[m], B[n])$: edit distance of $A[1 \cdots (m-1)]$ and $B[1 \cdots (n-1)]$
    - $(A[m], -)$: edit distance of $A[1 \cdots (m-1)]$ and $B[1 \cdots n]$
- **Step 2: Recursively define the value of an optimal solution.**
- $dist(i, j) =$

$$\begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} dist(i, j-1) + 1 \\ dist(i-1, j) + 1 \\ dist(i-1, j-1) + I[A[i] = B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

Edi





- Edit /del/s o B.
- **Prob** what
- **Step** ructu
- **Step 2: Recursively define the value of an optimal solution.**
- $dist(i$

$$\min\{$$ se

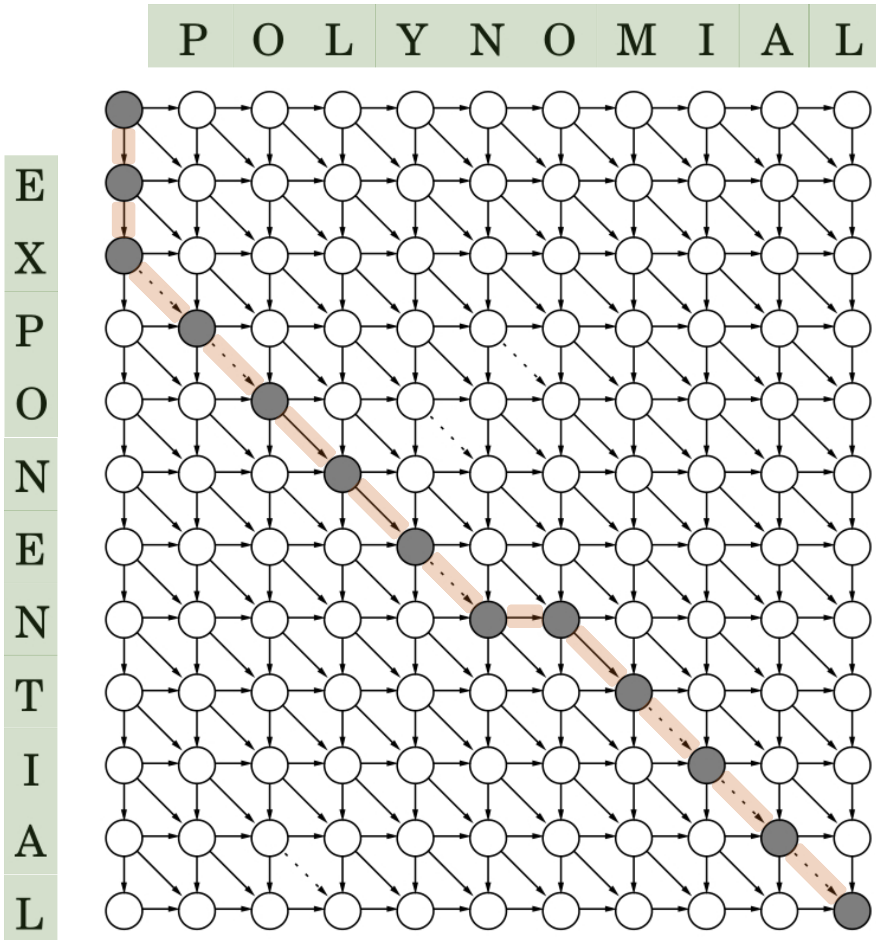
EditDistDP(A[1…m],B[1…n]):

```
for (i=0 to m)

fo
  dist[0,j] = j
for (i=1 to m)
  for (j=1 to n)
    delDist = dist[i-1,j] + 1
    insDist = dist[i,j-1] + 1
    subDist = dist[i-1,j-1] + Diff(A[i],B[j])
    dist[i,j] = Min(delDist,insDist,subDist)
return dist
```

Step 4: Construct an optimal solution.

- **Step 3** ottom-Up).
  - Wh
  - Outer-loop: increasing i; Inner-loop: increasing j.



Subproblem graph (DAG)

Transform “EXPONENTIAL” to “POLYNOMIAL”

→ Insertion (costs 1)

↓ Deletion (costs 1)

↘ Substitution [diff] (costs 1)

⇢ Substitution (costs 0)

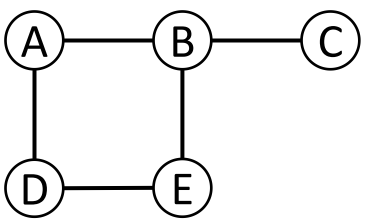
Edit distance:

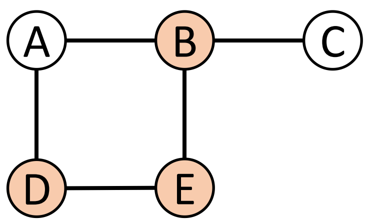
Shortest path from top-left to right-bottom.

E	X	P	O	N	E	N	–	T	I	A	L
–	–	P	O	L	Y	N	O	M	I	A	L
1	1			1	1		1	1			

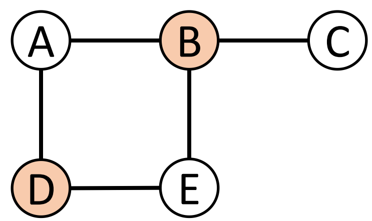
Maximum Independent Set

- Given an undirected graph $G = (V, E)$, an **independent set** I is a subset of V, such that no vertices in I are adjacent.
  (Put another way, for all $(u, v) \in I \times I$, we have $(u, v) \notin E$.)
- A **maximum independent set** (**MaxIS**) is an independent set of maximum size.



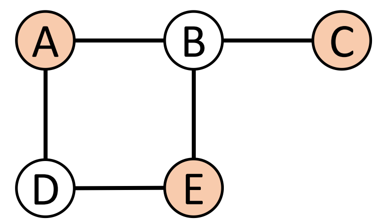


$\{B, D, E\}$ is NOT IS



$\{B, D\}$ is IS

It is NOT MaxIS



$\{A, C, E\}$ is IS

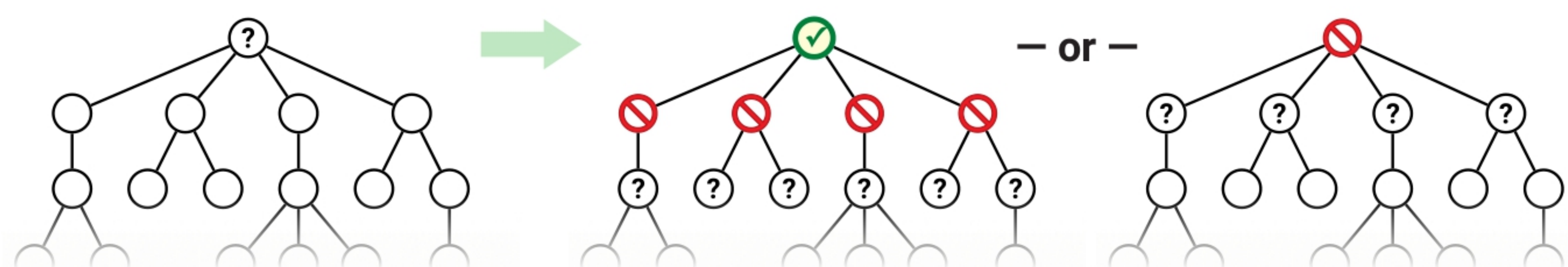
It is also MaxIS

Maximum Independent Set

- Given an undirected graph $G = (V, E)$, an **independent set** I is a subset of V, such that no vertices in I are adjacent.
  (Put another way, for all $(u, v) \in I \times I$, we have $(u, v) \notin E$.)
- A **maximum independent set** (**MaxIS**) is an independent set of maximum size.
- Computing MaxIS in an arbitrary graph is (likely) very hard. Even getting an approximate MaxIS is (likely) very hard!
- But if we only consider **trees**, MaxIS is very easy!

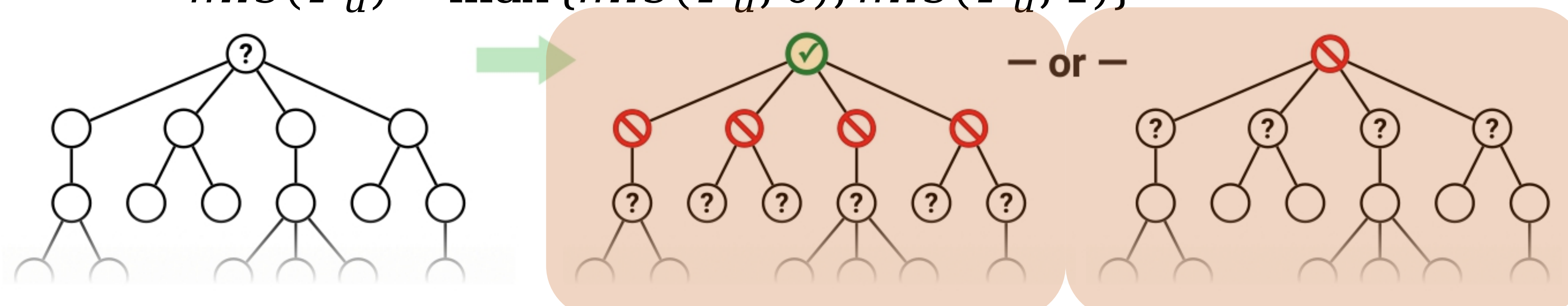
MaxIS of Trees

- **Problem:** Given a tree T with root r, compute a MaxIS of it.
- **Step 1: Characterize the structure of solution.**
  - Given an IS I of T, for each child u of r, set $I \cap V(T_u)$ is an IS of T_u.
- **Step 2: Recursively define the value of an optimal solution.**
  - ~~Let $mis(T_u)$ be size of MaxIS of (sub)tree~~ rooted at node u.
  - $mis(T_u) = 1 + \sum_{v \text{ is a child of } u} mis(T_v)$
  - **NO!** The recurrence depends on whether u in the MaxIS of T_u.



MaxIS of Trees

- **Problem:** Given a tree T with root r, compute a MaxIS of it.
- **Step 2: Recursively define the value of an optimal solution.**
  - Let $mis(T_u)$ be size of MaxIS of (sub)tree rooted at node u.
  - The recurrence depends on whether u in the MaxIS of T_u.
  - Let $mis(T_u, 1)$ be size of MaxIS of T_u, s.t. u in the MaxIS.
  - Let $mis(T_u, 0)$ be size of MaxIS of T_u, s.t. u NOT in the MaxIS.
  - $mis(T_u, 1) = 1 + \sum_{v \text{ is a child of } u} mis(T_v, 0)$
  - $mis(T_u, 0) = \sum_{v \text{ is a child of } u} mis(T_v)$
  - $mis(T_u) = \max\{mis(T_u, 0), mis(T_u, 1)\}$



MaxIS of Trees

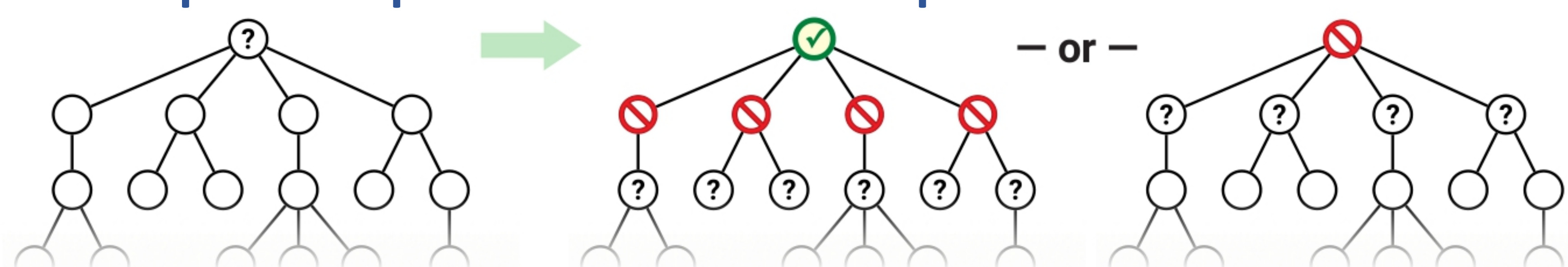
- **Problem:** Given a tree T with root r, compute a MaxIS of it.
- **Step 2: R**... **al solution.**
  - Let mi ... ode u.
  - Let mi ... MaxIS.
  - Let mi ... the MaxIS.
  - $mis(T$
  - $mis(T$
  - $mis(T_u$ ...

MaxIsDP(u):

Runtime is $O(V+E)=O(V)$

```
mis1 = 1
mis0 = 0
for (each child v of u)
  mis1 = mis1 + MaxISDP(v).mis0
  mis0 = mis0 + MaxISDP(v).mis
mis = Max(mis0,mis1)
return <mis,mis0,mis1>
```

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



Dynamic Programming (DP)

- Consider an (optimization) problem:
  - Build optimal solution step by step.
  - Problem has **optimal substructure** property.
    - We can design a recursive algorithm.
  - Problem has lots of **overlapping subproblems**.
    - Recursion and *memorize* solutions. (Top-Down)
    - Or, consider subproblems in the *right order*. (Bottom-Up)

Optimal substructure not always true

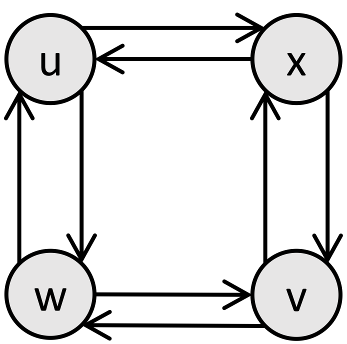
- Shortest path in unit-weight graph:
  - Assume $w \in OPT(u \to v)$
  - $OPT(u \to v) = u \xrightarrow{P_1} w \xrightarrow{P_2} v$
  - $P_1 = OPT(u \to w)$ and $P_2 = OPT(w \to v)$

Optimal substructure property!
Subproblems are *independent*!

- Longest *simple* path in unit-weight graph:
  - Assume $w \in OPT(u \to v)$
  - $OPT(u \to v) = u \xrightarrow{P_1} w \xrightarrow{P_2} v$
  - $P_1 = OPT(u \to w)$ and $P_2 = OPT(w \to v)$

    Subproblems are NOT *independent*!
  - Clearly $OPT(u \to v) = u \to w \to v$.
  - ~~Therefore $OPT(u \to w) = u \to w$~~?
  - But $OPT(u \to w) = u \to x \to v \to w$.
  - Similarly, $OPT(w \to v) = w \to u \to x \to v \neq w \to v$.

NO optimal substructure property!



Dynamic Programming (DP)

- Consider an (optimization) problem:
  - Build optimal solution step by step.
  - Problem has **optimal substructure** property.
    - We can design a recursive algorithm.
  - Problem has lots of **overlapping subproblems**.
    - Recursion and *memorize* solutions. (Top-Down)
    - Or, consider subproblems in the *right order*. (Bottom-Up)

Top-Down vs Bottom-Up

- Dynamic programming *trades space for time*.
  - Save solutions for subproblems to avoid repeat computation.
- [**Top-Down**] Recursion with memorization.
  - Very straightforward, easy to write down the code.
  - Use array or hash-table to memorize solutions.
  - Array may cost more space, but hash-table may cost more time.
- [**Bottom-Up**] Solve subproblems in the right order.
  - Finding the right order might be non-trivial. (Subproblem graph?)
  - Usually use array to memorize solutions.
  - Might be able to reduce the size of array to save even more space.
- **Top-down vs Bottom-up**
  - Top-down often costs more time in practice. (Recursion is costly!)
  - But not always! (Top-down only considers *necessary* subproblems.)

APSP via Dynamic Programming

The Floyd-Warshall Algorithm

- S
- D

  s … in paths.
- d … E … E

  $\left(dist(u, x_r, r-1) + dist(x_r, v, r-1)\right)$

FloydWarshallAPSP(G):

```
for (every pair (u,v) in V*V)
  if ((u,v) in E) then dist[u,v]=w(u,v)
  else dist[u,v]=INF
for (r=1 to n)
  for (each node u)
    for (each node v)
      if (dist[u,v] > dist[u,x_r] + dist[x_r,v])
        dist[u,v] = dist[u,x_r] + dist[x_r,v]
```

Space cost $O(n^2)$

FloydWarshallAPSP(G):

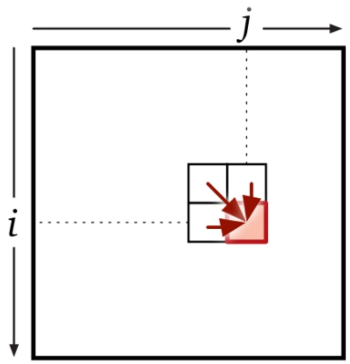
```
for (every pair (u,v) in V*V)
  if ((u,v) in E) then dist[u,v,0]=w(u,v)
  else dist[u,v,0]=INF
for (r=1 to n)
  for (each node u)
    for (each node v)
      dist[u,v,r] = dist[u,v,r-1]
      if (dist[u,v,r] > dist[u,x_r,r-1] + dist[x_r,v,r-1])
        dist[u,v,r] = dist[u,x_r,r-1] + dist[x_r,v,r-1]
```

Space cost $O(n^3)$

Edit Dist

- **Edit Distance:**

- $dist(i, j) =$ { mi



EditDistDP(A[1…m],B[1…n]):

Space cost $O(n)$

```
for (j=0 to n)
  distLast[j] = j
for (i=1 to m)       //distLast[j] = dist[i-1,j]
  distCur[0] = i     //distCur[j] = dist[i,j]
  for (j=1 to n)
    delDist = distLast[j] + 1
    insDist = distCur[j-1] + 1
    subDist = distLast[j-1] + Diff(A[i],B[j])
    distCur[j] = Min(delDist,insDist,subDist)
  distLast = distCur
return distCur[n]
```

EditDistDP(A[1…m],B[1…n]):

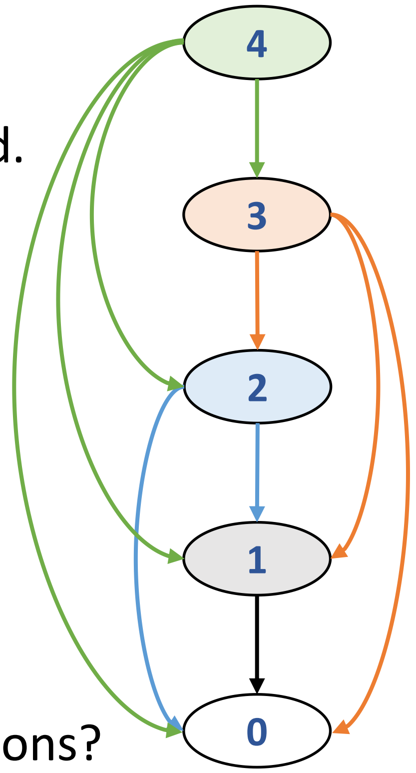
Space cost $O(n^2)$

```
for (i=0 to m)
  dist[i,0] = i
for (j=0 to n)
  dist[0,j] = j
for (i=1 to m)
  for (j=1 to n)
    delDist = dist[i-1,j] + 1
    insDist = dist[i,j-1] + 1
    subDist = dist[i-1,j-1] + Diff(A[i],B[j])
    dist[i,j] = Min(delDist,insDist,subDist)
return dist
```

Analysis of DP Algorithms

- **Correctness:**
  - Optimal substructure property.
  - **Bottom-up approach:** subproblems are already solved.

- **Complexity:**
  - **Space complexity:** obvious with array. (Hash table?)
  - **Time complexity [bottom-up]:** usually obvious.
  - **Time complexity [top-down]:**
    - How many subproblems in total?
      (# of nodes in the subproblem DAG.)
    - Time to solve a problem, given subproblem solutions?
      (# of edges in the subproblem DAG.)



Subset Sum

- **Problem:** Given an array $X[1 \cdots n]$ of n positive integers, can we find a subset in X that sums to given integer T?
- **Simple solution:** recursively enumerates all 2^n subsets, leading to an algorithm costing $O(2^n)$ time.
- Can we do better with dynamic programming?
  (Notice this is **not** an optimization problem.)

Subset Sum

- **Problem:** Given an array $X[1 \cdots n]$ of n positive integers, can we find a subset in X that sums to given integer T?
- **Step 1: Characterize the structure of solution.**
  - If there is a solution S, either $X[1]$ is in it or not.
  - If $X[1] \in S$, then there is a solution to instance "$X[2 \cdots n], T - X[1]$"; If $X[1] \notin S$, then there is a solution to instance "$X[2 \cdots n], T$".
- **Step 2: Recursively define the value of an optimal solution.**
  - Let $ss(i,t) = true$ iff instance "$X[i \cdots n], t$" has a solution.
  - $ss(i,t) = \begin{cases} true & \text{if } t = 0 \\ ss(i+1,t) & \text{if } t < X[i] \\ false & \text{if } i > n \\ ss(i+1,t) \vee ss(i+1, t - X[i]) & \text{otherwise} \end{cases}$

Su

SubsetSumDP(X,T):

```
ss[n,0] = True
for (t=1 to T)
  ss[n,t] = (X[n]==t)?True:False
for (i=n-1 downto 1)
  ss[i,0] = True
  for (t=1 to X[i]-1)
    ss[i,t] = ss[i+1,t]
  for (t=X[i] to T)
    ss[i,t] = Or( ss[i+1,t], ss[i+1,t-X[i]] )
return ss[1,T]
```

Runtime is $O(nT)$

- **Pro** ... rs, can ... ?
- **Ste**
- **Ste** ... **tion.**
  - Let $ss(i,t) = true$ if instance $\langle X[i \cdots n], t \rangle$ has a solution.
  - $$ss(i,t) = \begin{cases} true & \text{if } t = 0 \\ ss(i+1,t) & \text{if } t < X[i] \\ false & \text{if } i > n \\ ss(i+1,t) \vee ss(i+1,t-X[i]) & \text{otherwise} \end{cases}$$
- **Step 3: Compute the value of an optimal solution (Bottom-Up).**
  - Build an 2D array $ss[1 \cdots n, 0 \cdots T]$
  - Evaluation order: bottom row to top row; left to right within each row.

Subset Sum

- **Problem:** Given an array $X[1 \cdots n]$ of n positive integers, can we find a subset in X that sums to given integer T?
- **Simple solution:** recursively enumerates all 2^n subsets, leading to an algorithm costing $O(2^n)$ time.
- **Dynamic programming:** costing $O(nT)$ time.
- **Dynamic programming isn't *always* an improvement!**

Dynamic Programming vs Greedy

- Common strategies for solving optimization problems.
- Gradually generates a solution for the problem.
- **Dynamic Programming**
  - *At each step*: multiple potential choices, each reducing the problem to a subproblem, compute optimal solutions of all subproblems and then find optimal solution of original problem.
  - **Optimal substructure + Overlapping subproblems.**
- **Greedy**
  - *At each step*: make an optimal choice, then compute optimal solution of the subproblem induced by the choice made.
  - **Optimal substructure + Greedy choice.**
- Try DP first, then check if greedy works! (If does, prove it!)
  (Come up with a working algorithm first, then develop a faster one.)

Reading

- [CLRS] Ch.15
- Optional reading: [DPV] Ch.6; [Erickson v1] Ch.3