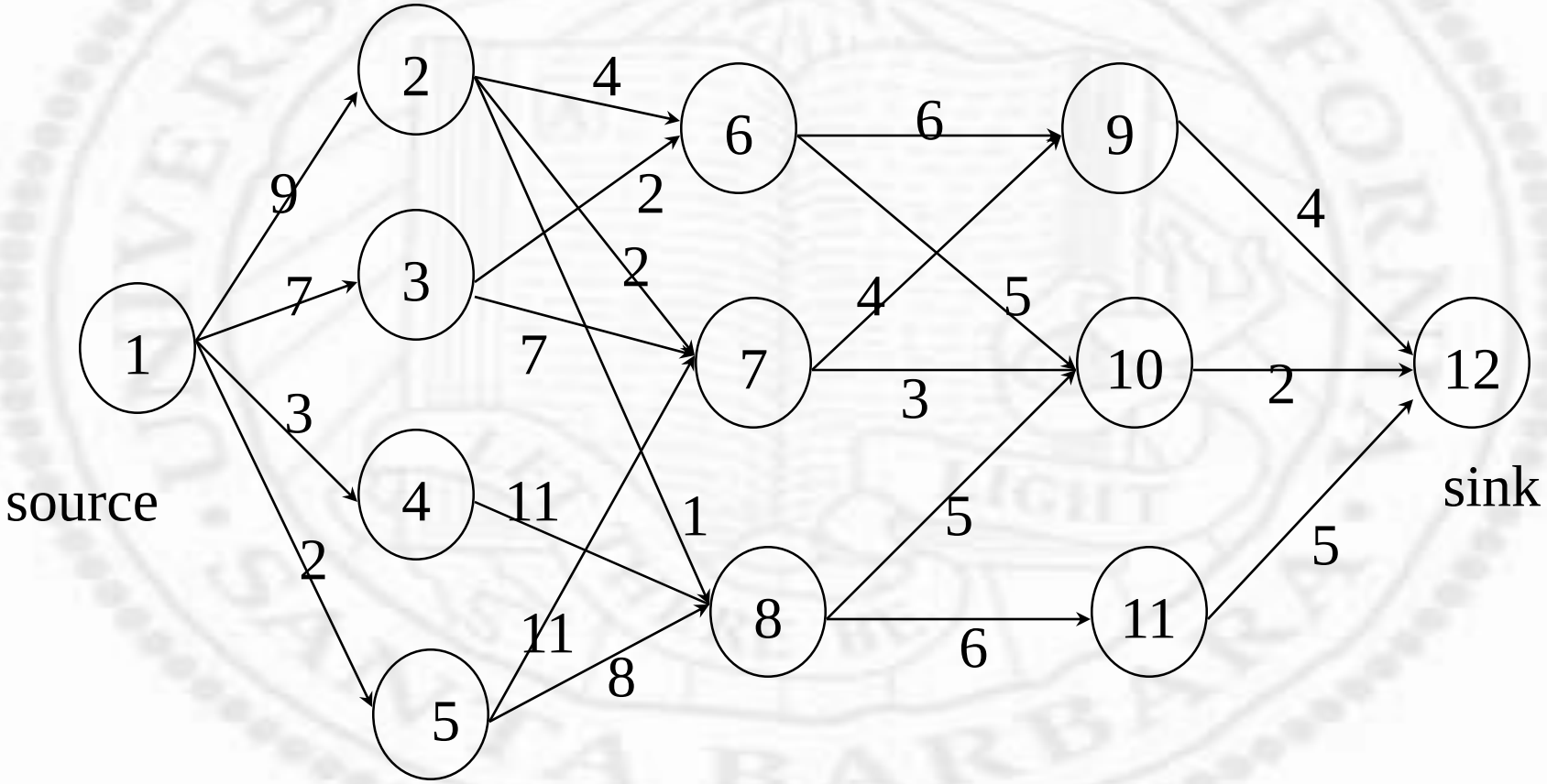
Dynamic Programming

- Example - multi-stage graph



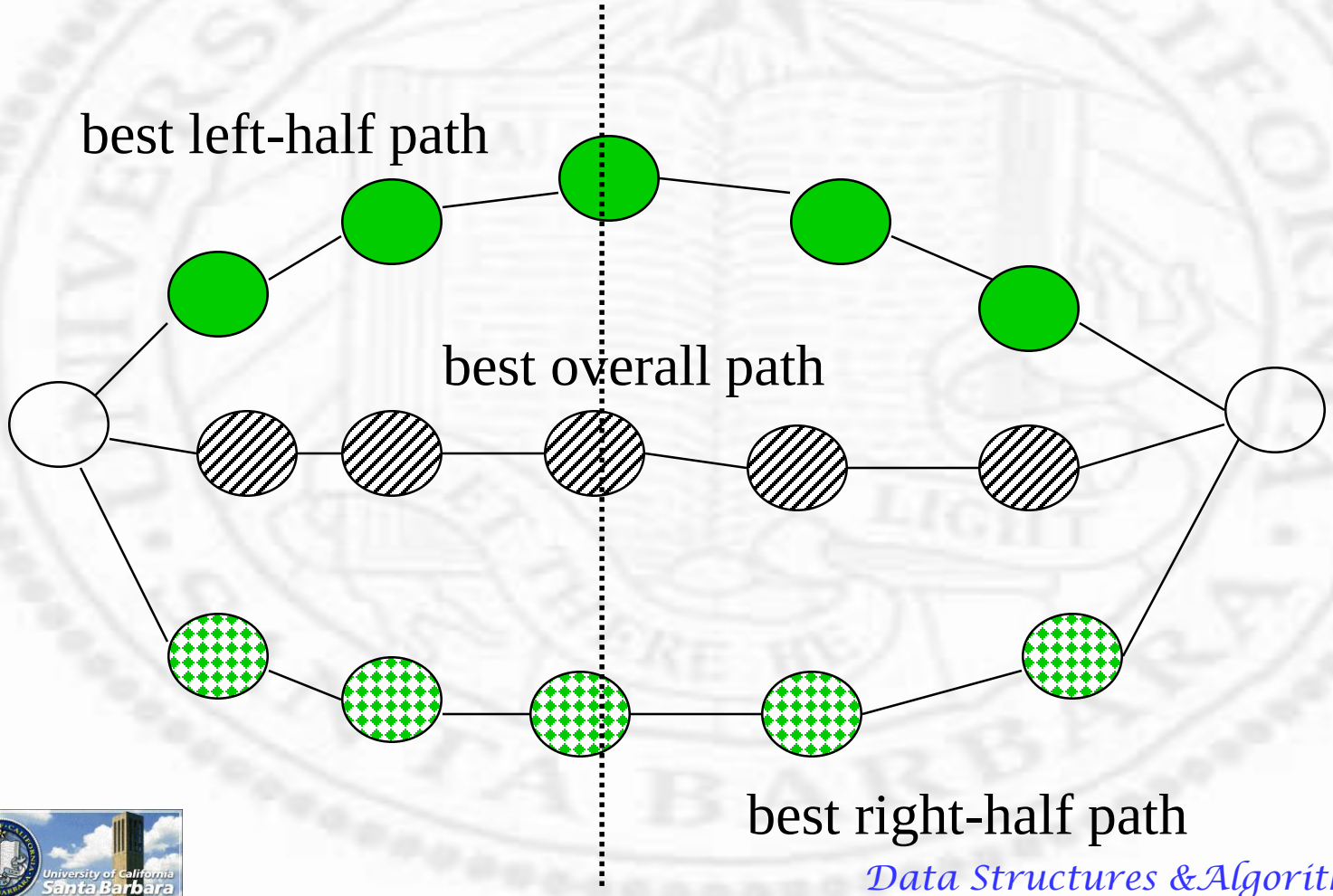
- A labeled, directed graph
- Vertices can be partitioned into k disjoint sets

  $$< u, v > \in E, \rightarrow u \in V_i, v \in V_{i+1}, 1 \leq i < k$$

  $$|V_1| \text{ (source)} = |V_k| \text{ (sink)} = 1$$

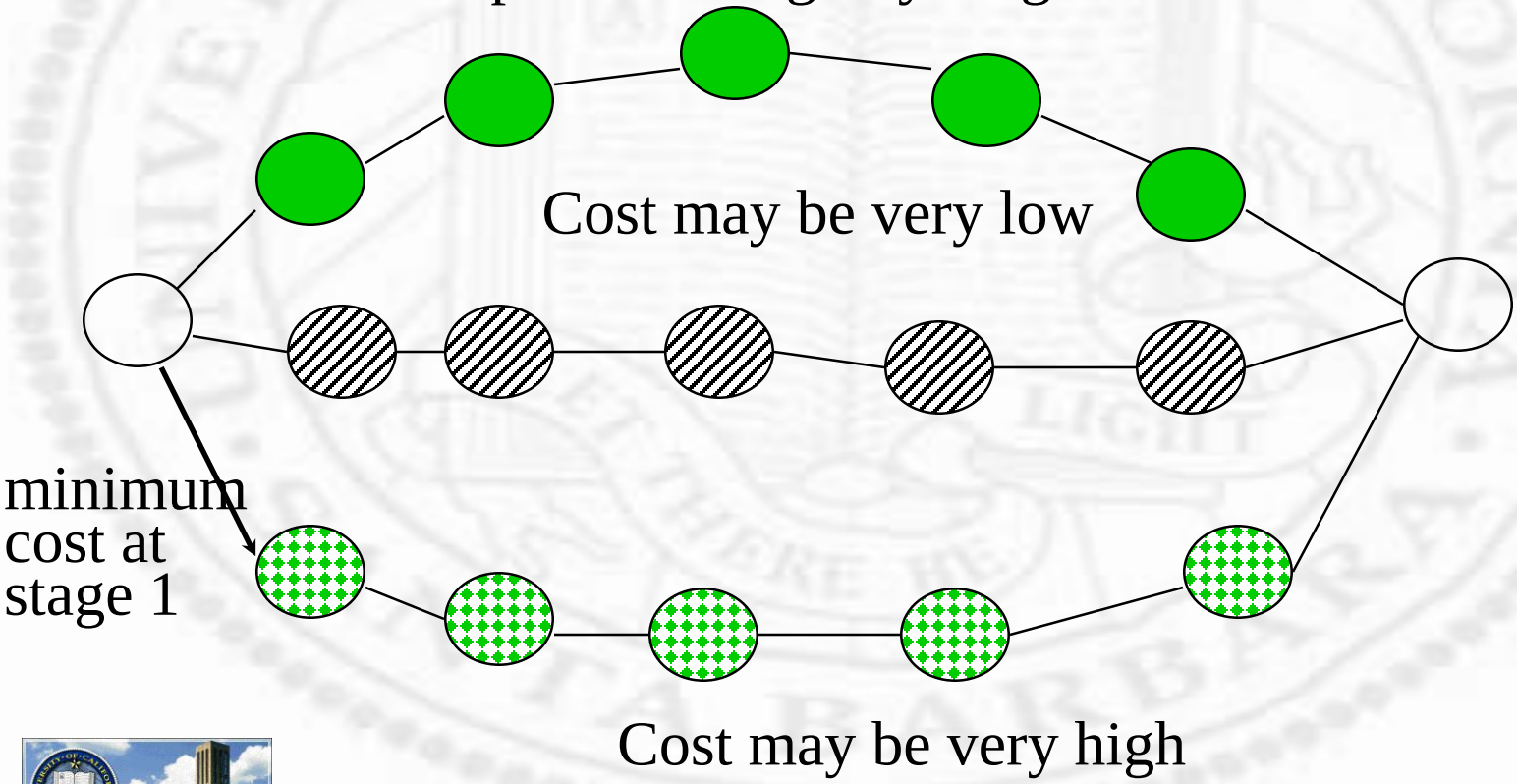
- Find the min cost path from source to sink

- Q: Will divide-and-conquer find the minimum cost path?
- A: Probably not



- Best paths found independently may not form a path
- Best overall paths may be suboptimal at different subproblem stages
- Divide-and-Conquer requires subproblems to be *independent!*

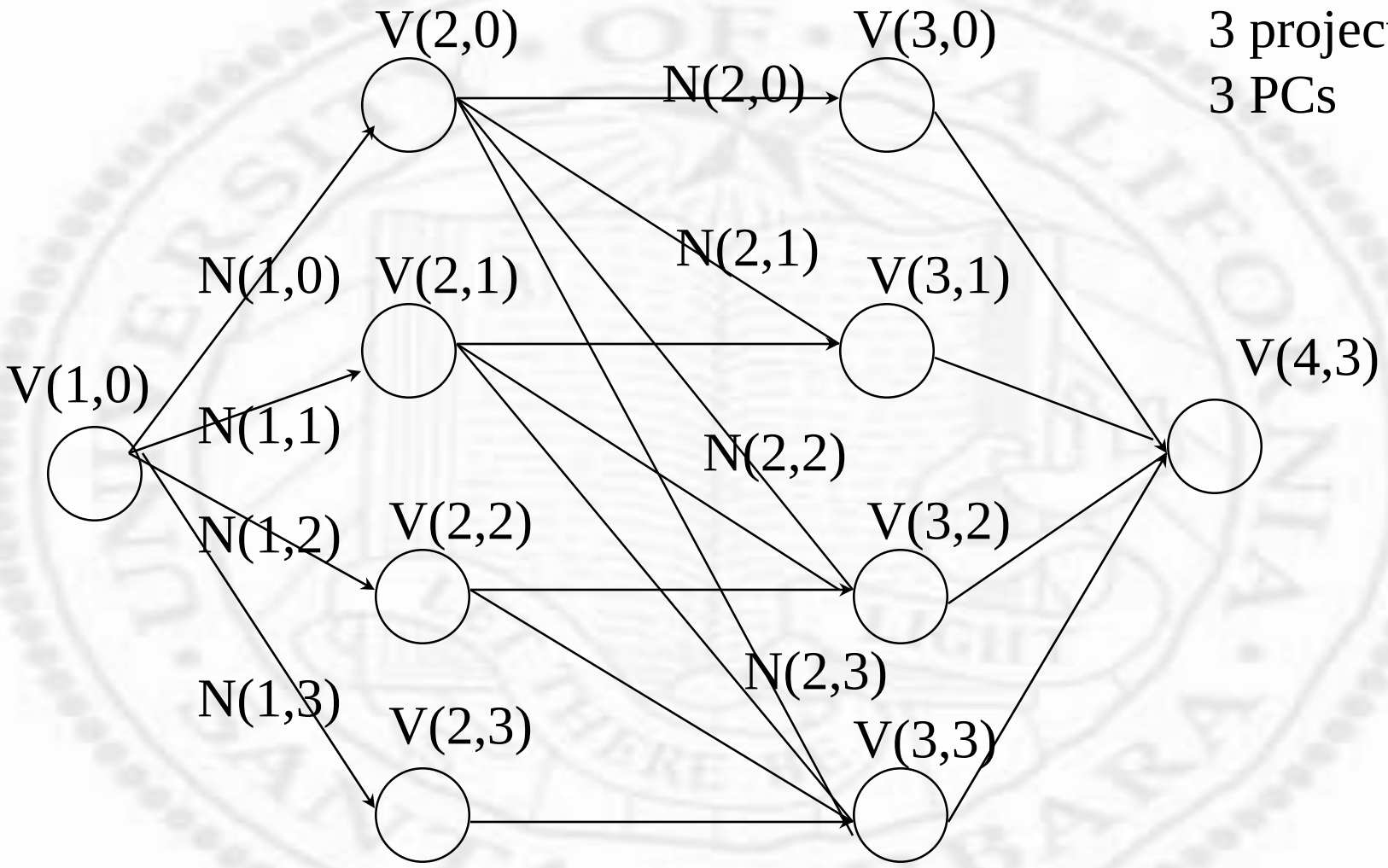
- Q: Will the greedy method finds the minimum cost path?
- A: May not (if you are not Dijkstra)
  - Choose the shortest link first
  - Solve the problem stage-by-stage



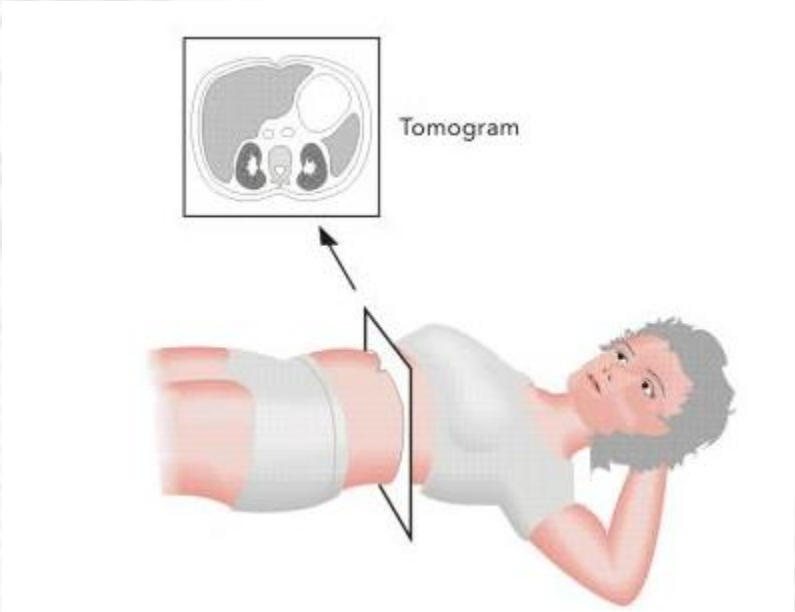
- A low cost edge may be followed by paths of a very high cost
- A high cost edge may be followed by paths of a very low cost
- Based on local information (one stage at a time) it might not be possible to "look-ahead"
- Picking the remaining lowest cost edge may not generate a path

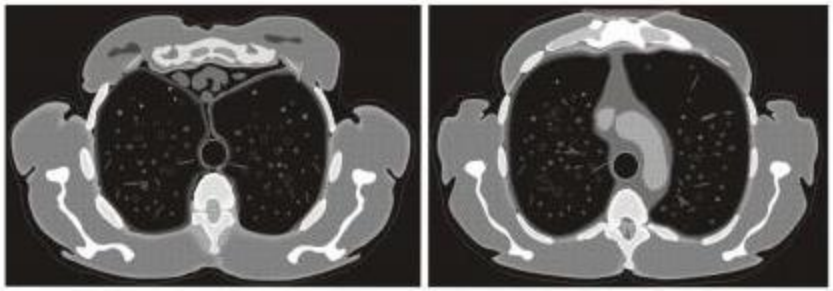
- Q: Is there an application?
- A: Yes, e.g., resource allocation
  - *n* units of resources to be allocated to *r* projects
  - *N(i,j)* profit earned if *j* units of resources are allocated to project *i*
  - goal is to maximize the profit earned

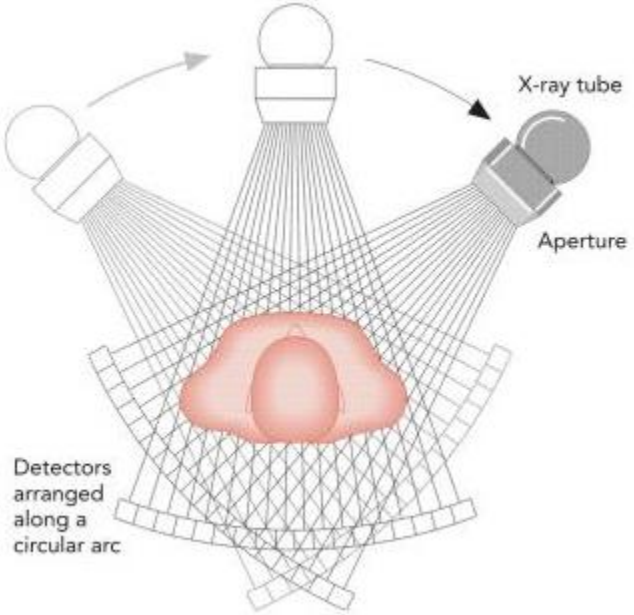
V(project being considered, resources committed)



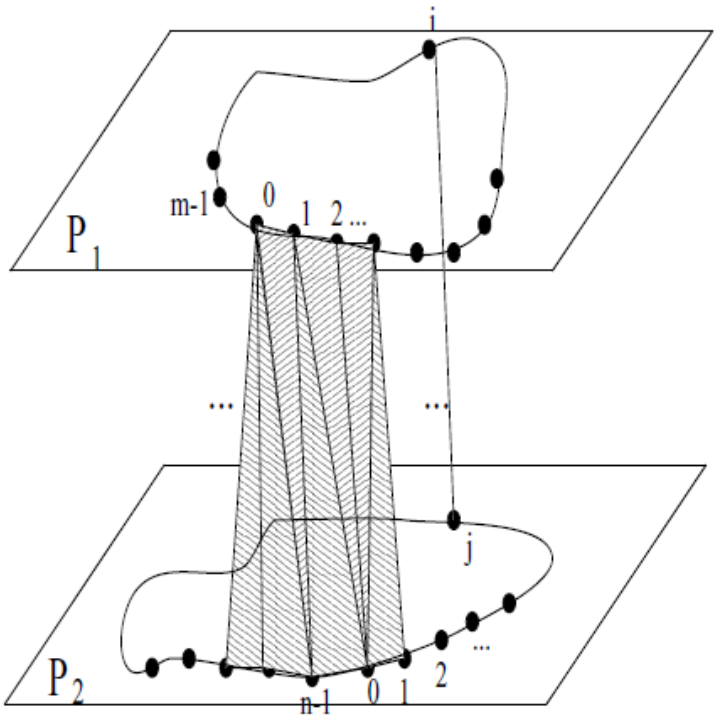
Surface Generation in Tomography

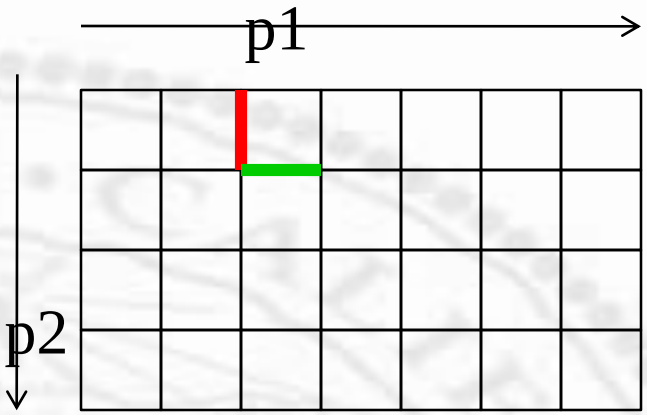






University of California Santa Barbara

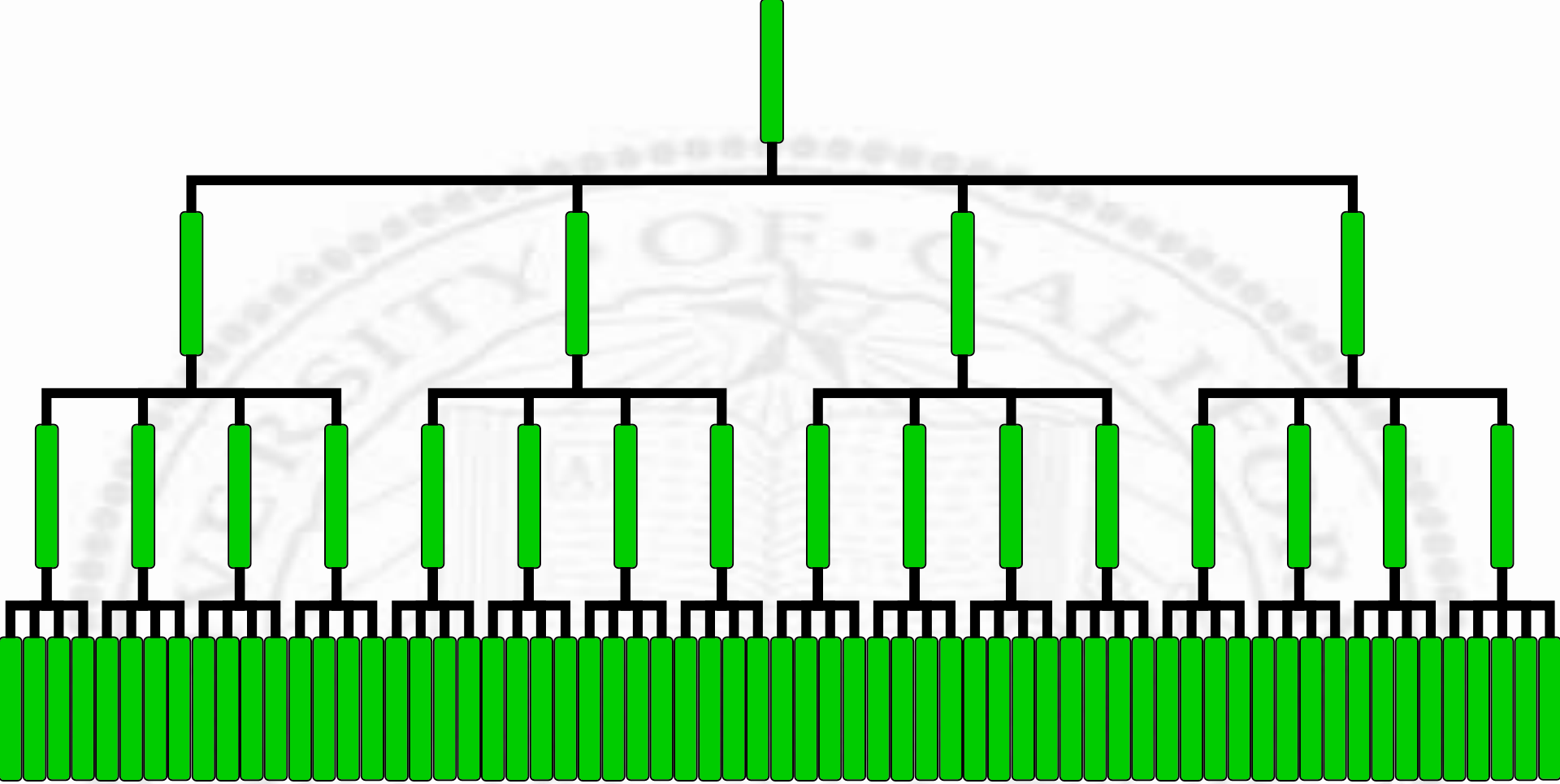


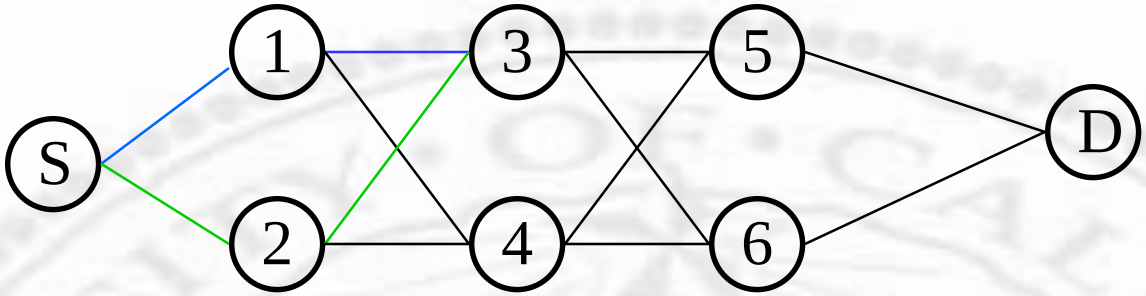


- m by n lattice
  - Vertical edge: an upright triangle
  - Horizontal edge: an inverted triangle
- Closed surfaces correspond to paths of length m+n
- Best path (surface) has the lowest cost

- Q: What should we do?
- A: Enumerate all possibilities
- Q: How much is the cost of enumeration?
- A: High, for
  - complete connection between two adjacent stages
  - n stages
  - m vertices per stages

$$O(m^n)$$





S 1 3 5 D
S 1 3 6 D
S 1 4 5 D
S 1 4 6 D
S 2 3 5 D
S 2 3 6 D
S 2 4 5 D
S 2 4 6 D

A lot of repetitions:

build tables to remember partial solutions (reuse)

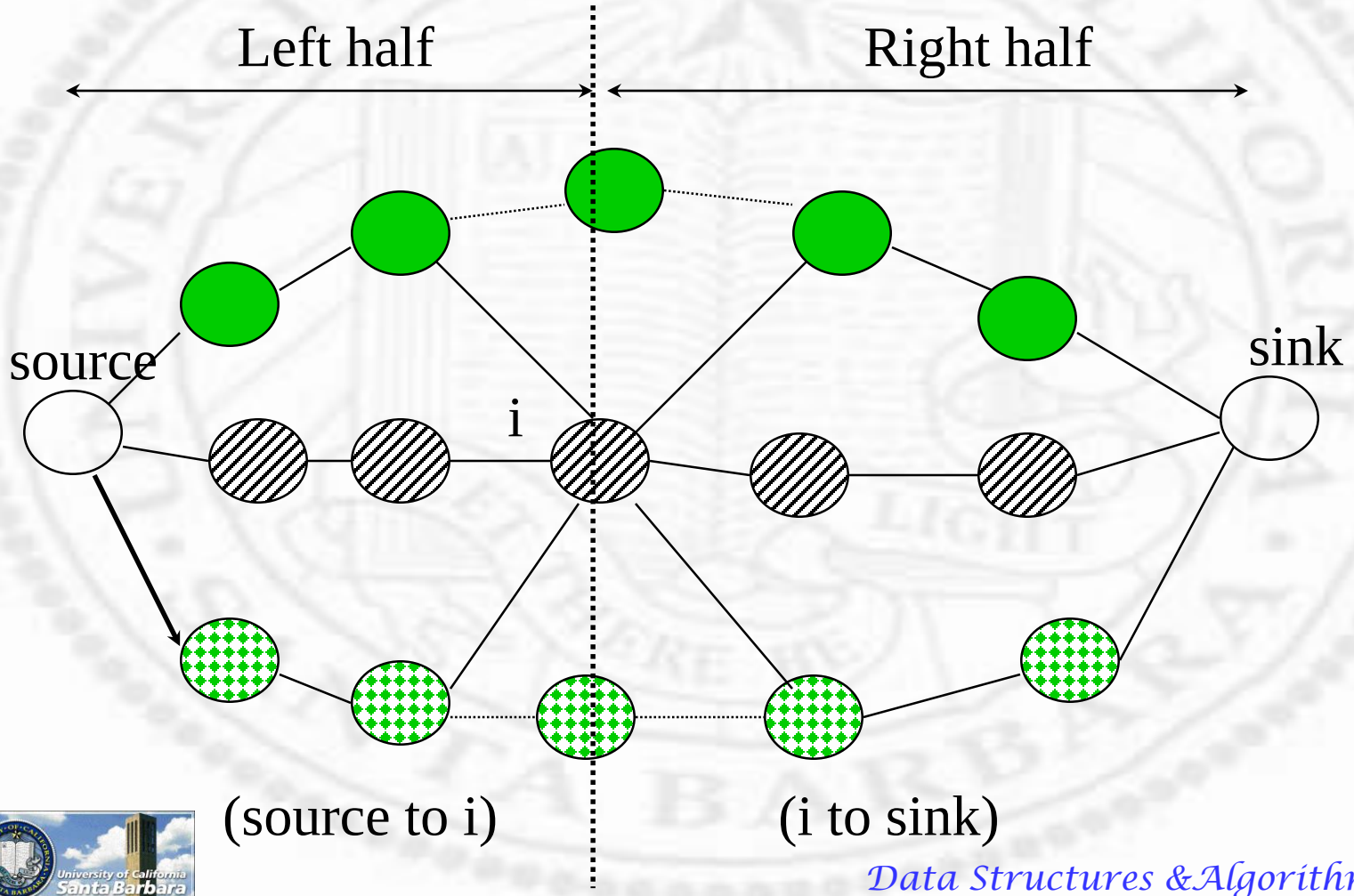
S 1 3 5 D
S 1 3 6 D
S 1 4 5 D
S 1 4 6 D
S 2 3 5 D
S 2 3 6 D
S 2 4 5 D
S 2 4 6 D

A lot of alternatives:

build tables to remember optimal partial solutions (principle of optimality)

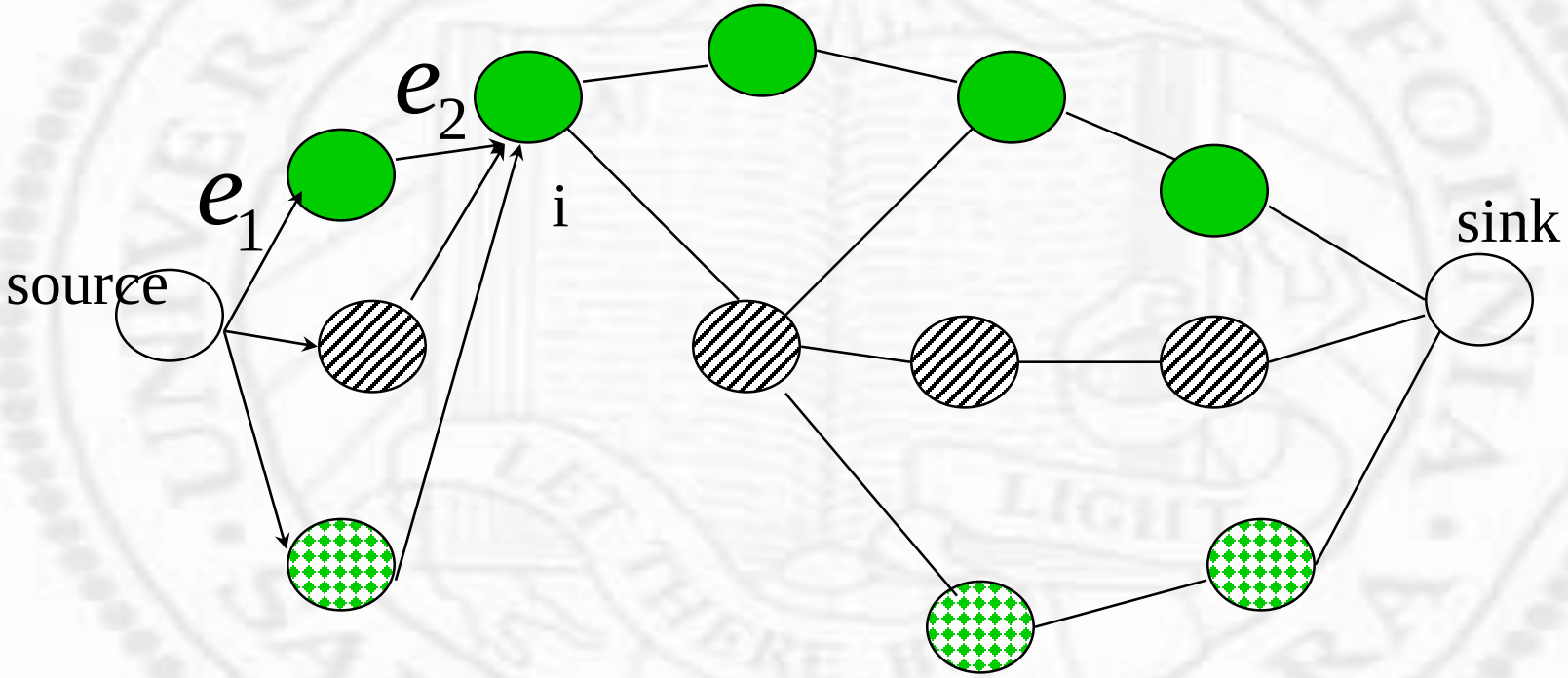
- Q: Is there a more efficient method of enumeration?
- A: Yes, *dynamic programming*
- *Underlying principles:*
  - Principle of optimality
    - *Early elimination of suboptimal subsolutions*
  - Recursion and reuse
    - *Construct solutions by reusing optimal subsolutions*
  - Early termination
    - *By feasibility or optimality*

Principle of Optimality



- The optimal solution (*if it go through node i*) must contains the best left path from source to *i* and best right path from *i* to sink
- Any other left paths from source to *i* and any other right paths from *i* to sink need not be extended any further

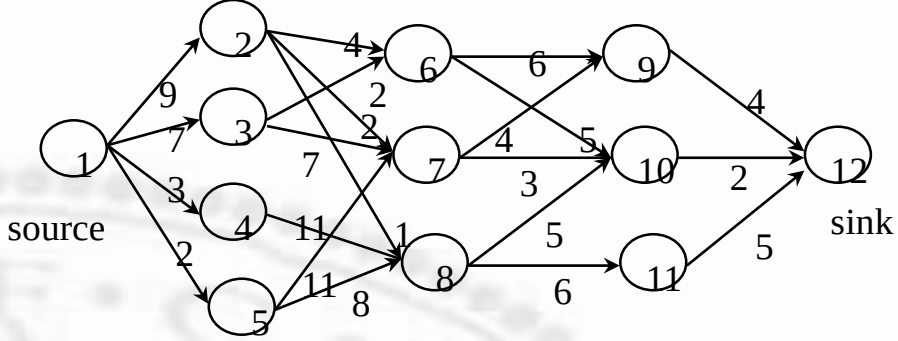
- Recursion and Reuse
  - Identify subproblems
  - Record the optimal solutions of subproblems
  - Build larger and larger solutions



- Any path that includes a portion of (source to *i*), the cost of that particular portion is known

- *Feasibility*
  - Is a *partial* solution still feasible?
  - Based on the current path alone
- *Optimality*
  - Is a *partial* solution going to be optimal?
  - Based on comparison of the current path with others

Backward approach



$(1-2)$	$(1-3)$	$(1-4)$	$(1-5)$
9	7	3	2

$(1-6)$	$(1-7)$	$(1-8)$
$9(\min(4+9,2+7))$	$11(\min(2+9,7+7,11+2))$	$10(\min(1+9,11+3,8+2))$

$(1-9)$	$(1-10)$	$(1-11)$
$15(\min(9+6,4+11))$	$14(\min(5+9,3+11,5+10))$	$16(6+10)$

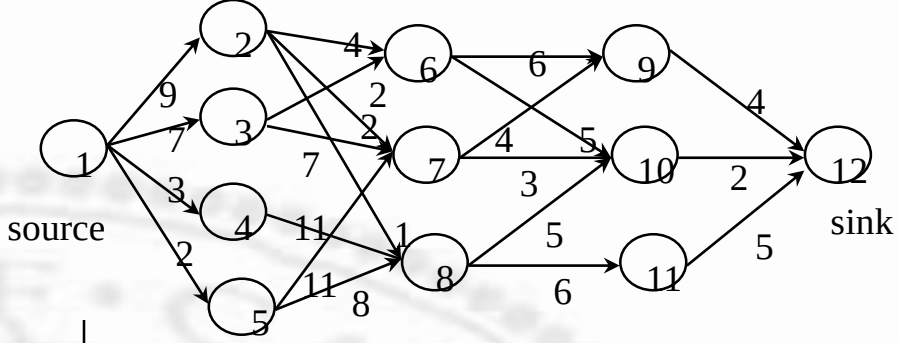
$(1-12)$

$16(\min(4+15,2+14,5+16))$

$BCost(i, j)$: the cost of the optimal path from the source to vertex j at stage i

$$BCost(i, j) = \min_{\substack{l \in V_{i-1} \\ <j,l> \in E}} \{BCost(i-1, l) + c(j, l)\}$$

Forward approach



$(9-12)$	$(10-12)$	$(11-12)$
4	2	5

$(6-12)$	$(7-12)$	$(8-12)$
$7(\min(6+4,5+2))$	$5(\min(4+4,3+2))$	$7(\min(5+2,6+5))$

$(2-12)$	$(3-12)$	$(4-12)$	$(5-12)$
$7(\min(4+7,2+5,1+7))$	$9(\min(2+7,7+5))$	$18(11+7)$	$15(\min(11+5,8+7))$

$(1-12)$

$16((\min(9+7,7+9,3+18,2+15))$

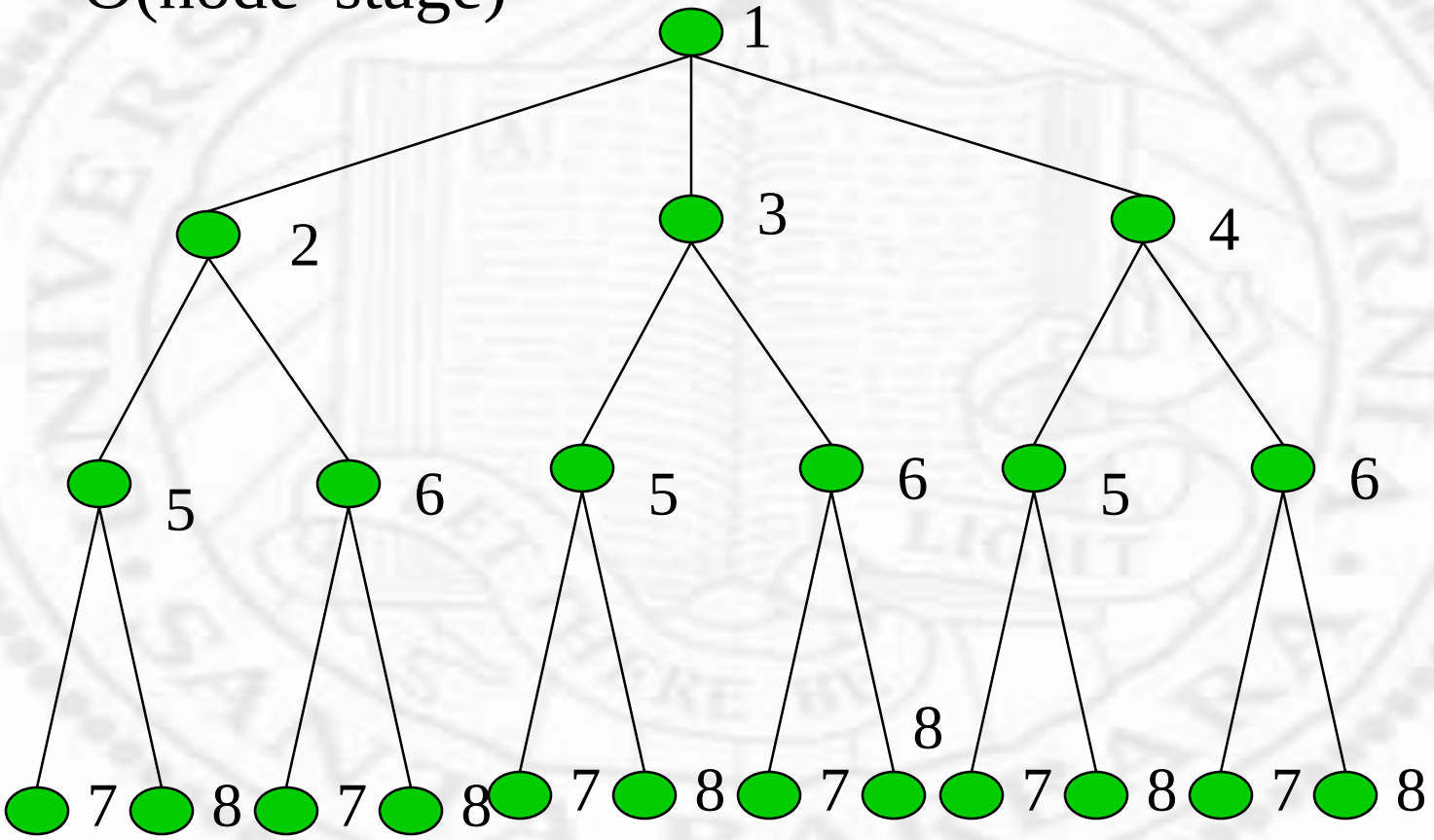
$Cost(i, j)$: the cost of the optimal path from vertex j at stage i to sink

$$Cost(i, j) = \min_{\substack{l \in V_{i+1} \\ <j,l> \in E}} \{c(j,l) + Cost(i+1,l)\}$$

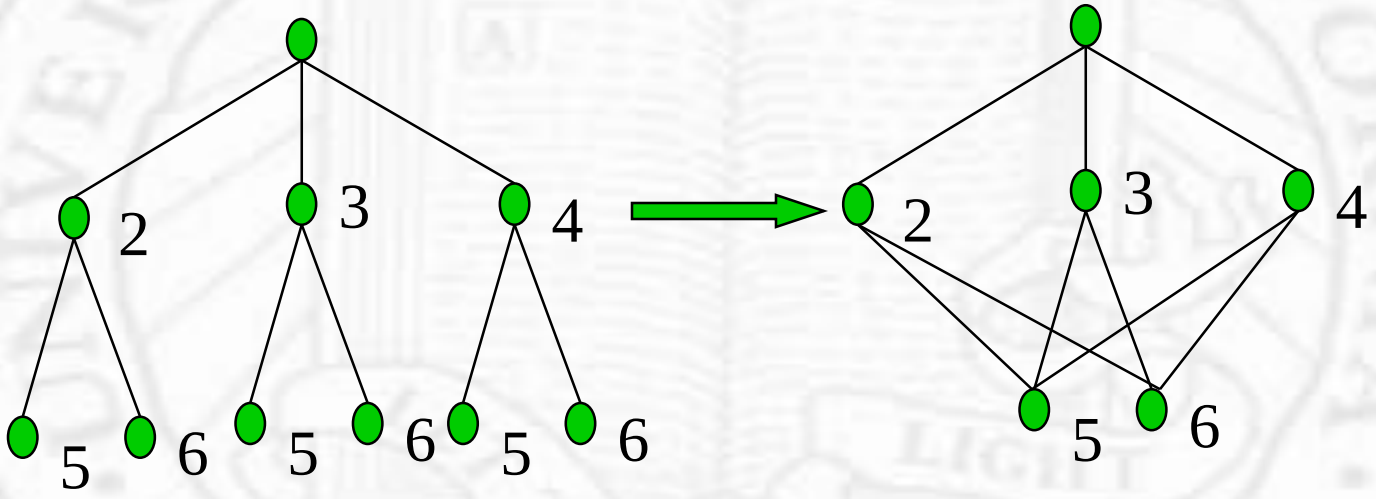
Intuition on DP

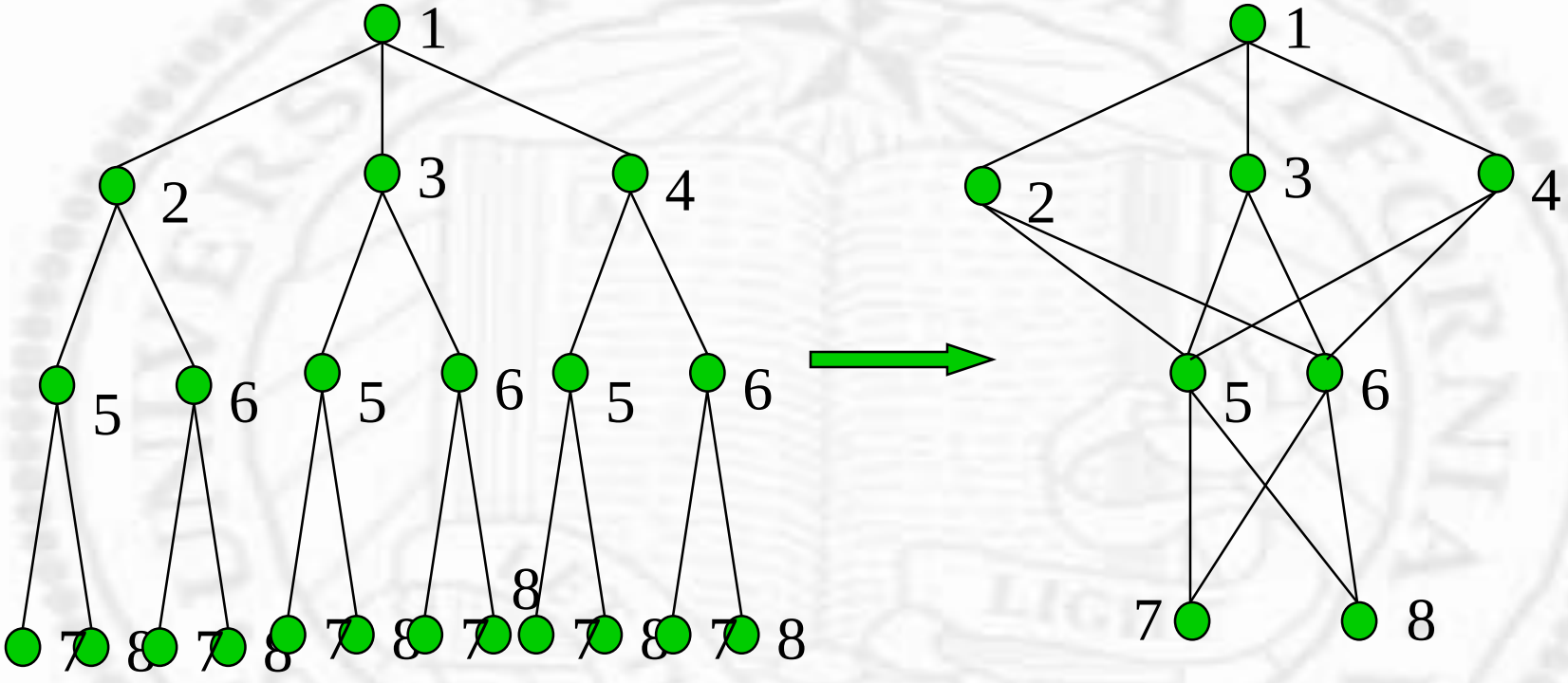
- Dynamic programming sometimes can be confusing because *it is basically recursion but is slightly more than recursion*
- A general solution pattern is
  - identify stages and all possible alternatives in a stage
  - recursion to generate all possible solutions
  - need to combine and eliminate partial solutions using principle of optimality

- For multi-stage graph
  - steps are defined by stages
  - straight recursion generates brushy tree O(node^stage)



- The important thing is to trim the tree by combining and coalescing nodes by principle of optimality

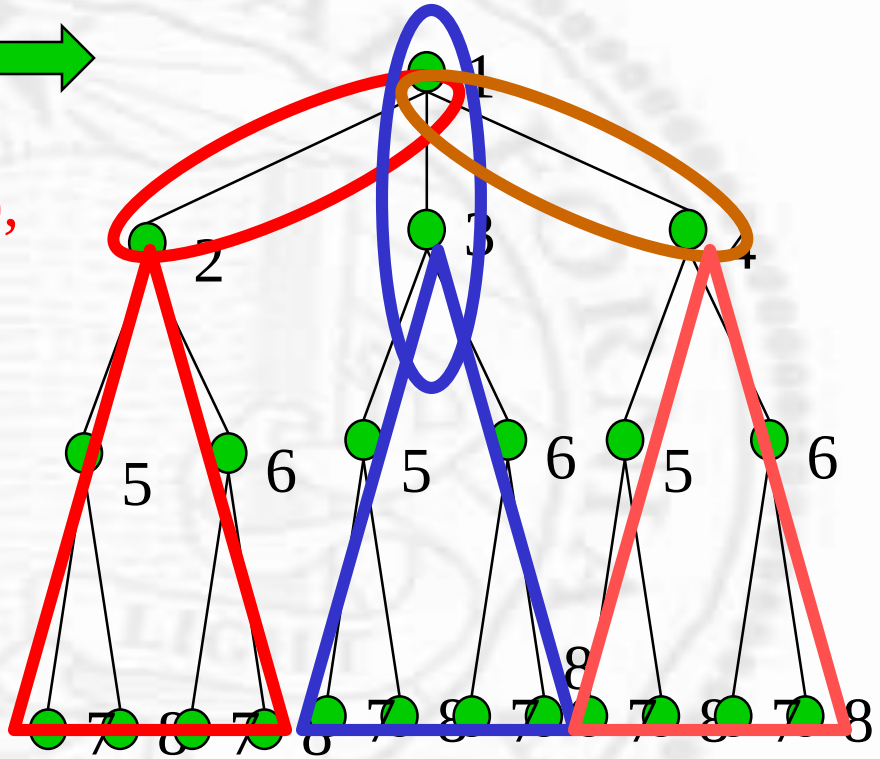



1
2
3
4
5
6
5
6
5
6
8
7
8
7
8
7
8
7
8
7
8
1
2
3
4
5
6
7
8

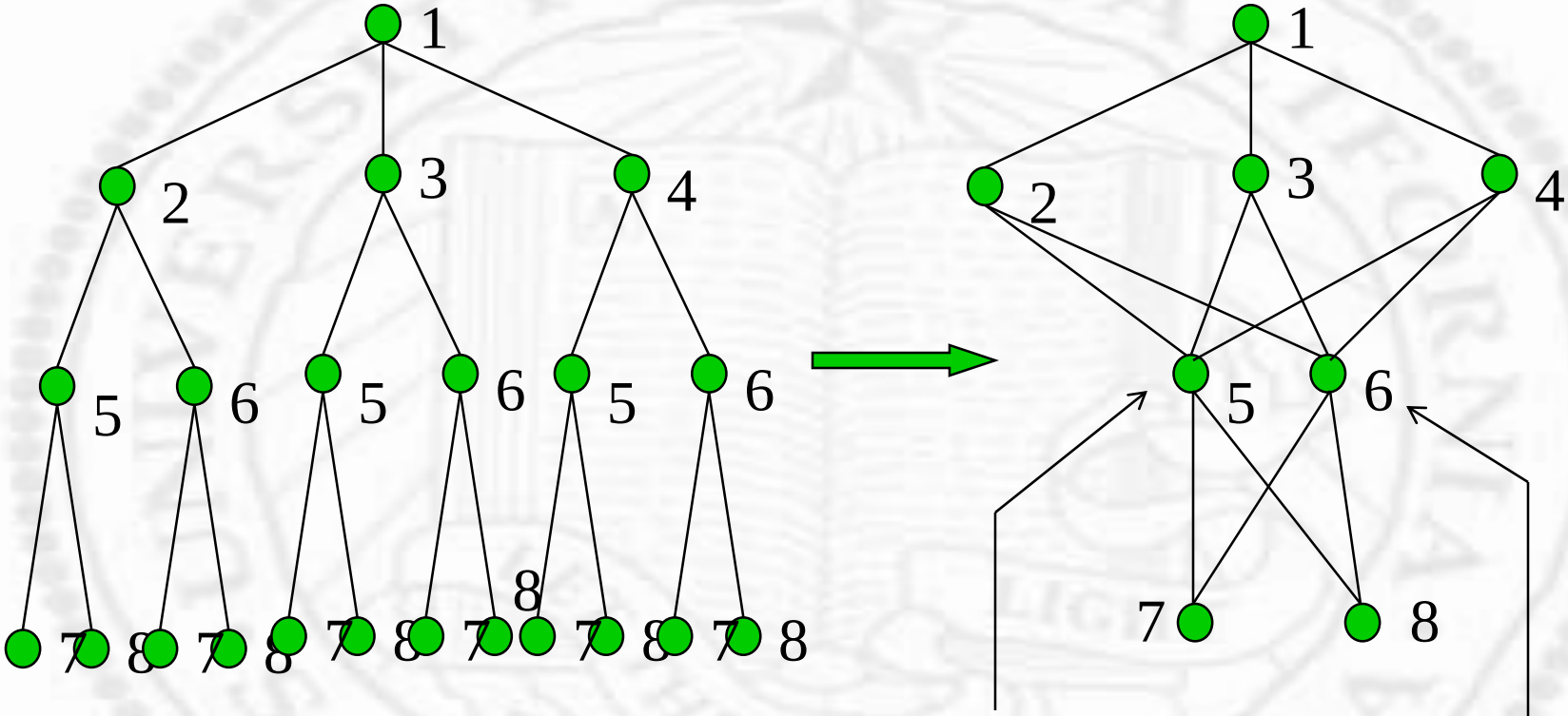
Fancy Recursive Equations

Cost (node 1 at level 1)

Min (c(1,2)+Cost(node 2 at level2),
c(1,3)+Cost(node 3 at level2),
c(1,4)+Cost(node 4 at level2),)



Fancy equation describes the recursion
DP says that all such Cost functions should be reused!!



Remember the costs here

Don’t compute again and again

Important Characteristics of Dynamic Programming

- A sequence of decisions to be made
- Decisions are inter-dependent (Divide-and-Conquer not applicable)
- Local information not sufficient (Greedy not work)

Important Characteristics of Dynamic Programming (cont.)

- It examines all solutions in an *efficient* manner
- It involves building solutions recursively
- Principle of optimality is used to eliminate sub-optimal solutions
- Table of some sort are usually used to store optimal partial solutions
- Some reuse of optimal partial solutions
- Mathematically as recursion

Time Complexity of DP

- DP == building tables of partial solutions
  1. How many tables?
  2. How many entries per table?
  3. How much effort to fill in entries?
  - 1*2*3 gives the complexity

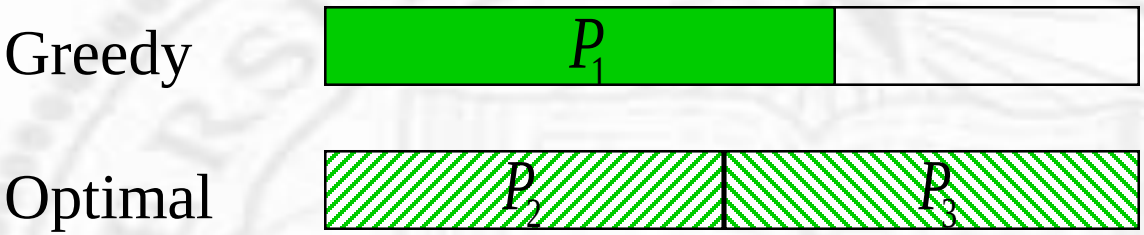
- Time complexity of multi-stage graph
  - One table is built
  - There are |V| entries in the table
  - The cost of generating an entry is proportional to the incident edges
  - *O(|V|+|E|):* a significant saving over exponent runtime

0/1-Knapsack

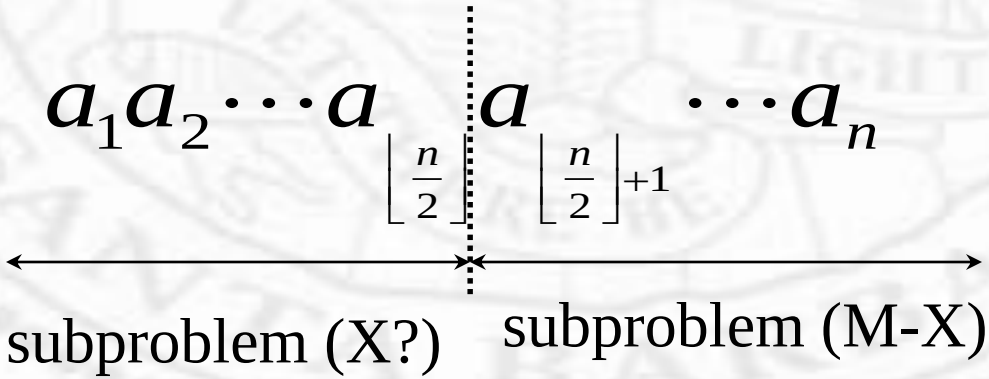
- Input:
  - a set of *n* objects
  - a knapsack of capacity *M*
- Output: fill the knapsack (*no partial inclusion*) to maximize the total profit earned

- Greedy method can fail

$$P = (9,7,7), W = (6,5,5), M = 10$$



- Divide-and-conquer may not apply



- How to build solutions recursively?
  - One object at a time
- How does principle of optimality apply?

$$x_1 x_2 \cdots x_i \cdots x_n$$

$$(1, x \cdots x \cdots x) : (x \cdots x \cdots x) \text{ must be optimal for } M - W_1$$

$$(0, y \cdots y \cdots y) : (y \cdots y \cdots y) \text{ must be optimal for } M$$

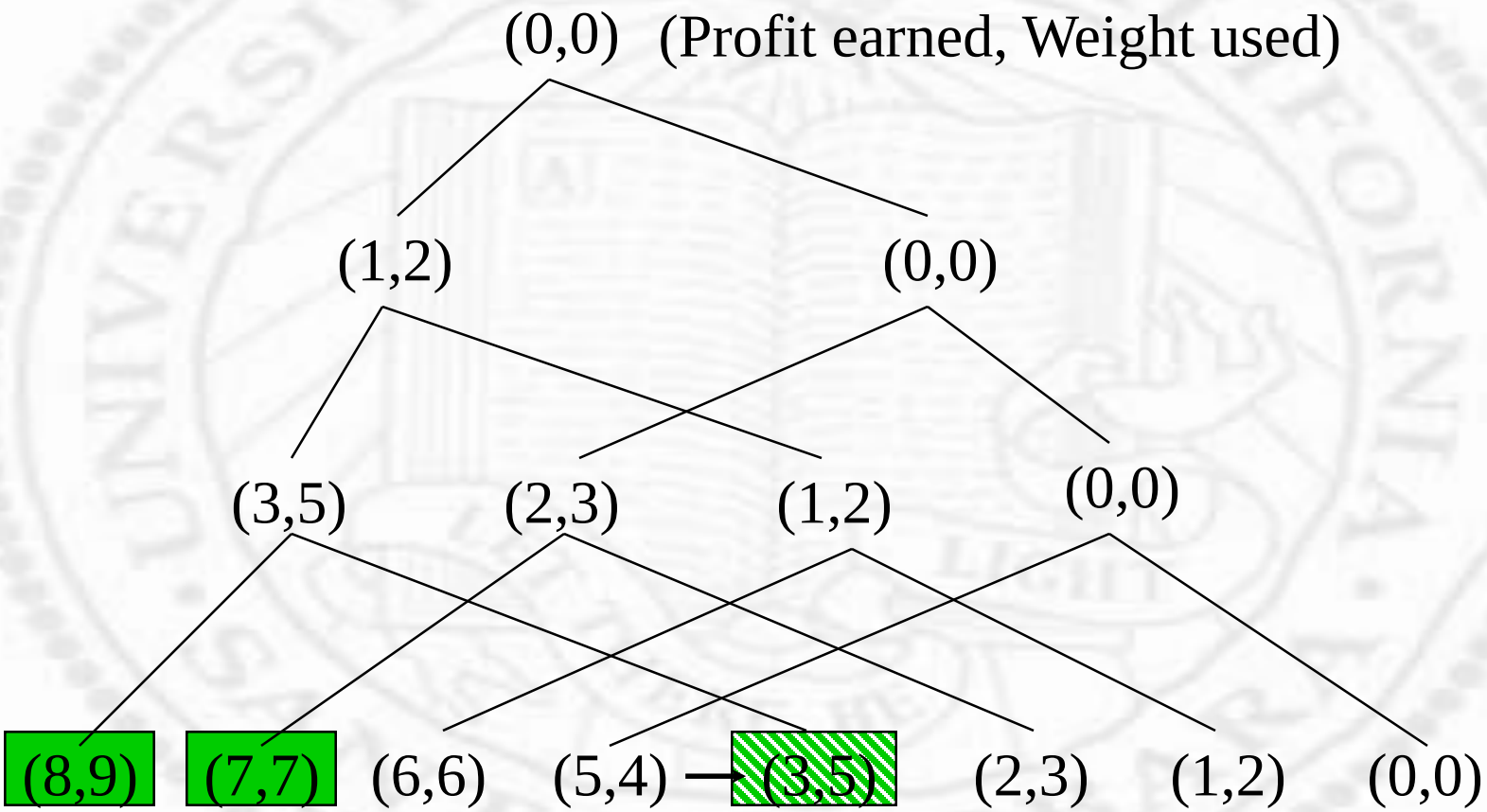
- How to identify sub-optimal solutions?
  - if two solutions: (1,…,0,x,…,x), and (0,…,1,x,…, x) are such that one achieves better profit with less weight, then the other cannot be optimal

How to build table?

Weight \ Profit	$\geq, \geq$	$\geq, \leq$
	$\leq, \geq$	$\leq, \leq$

Bounding possible!

P=(1,2,5), W=(2,3,4),M=6



- How to write the recursive equation?
  - Knap(*i*,*X*): current profit with objects *i* to *n* left to be processed with a remaining capacity of *X*
  - Initially, we have Knap(*1*,*M*)

$$Knap(1, M) = \max\{Knap(2, M), Knap(2, M - W_1) + P_1\}$$

$$Knap(i, X) = \max\{Knap(i+1, X), Knap(i+1, X - W_i) + P_i\}$$

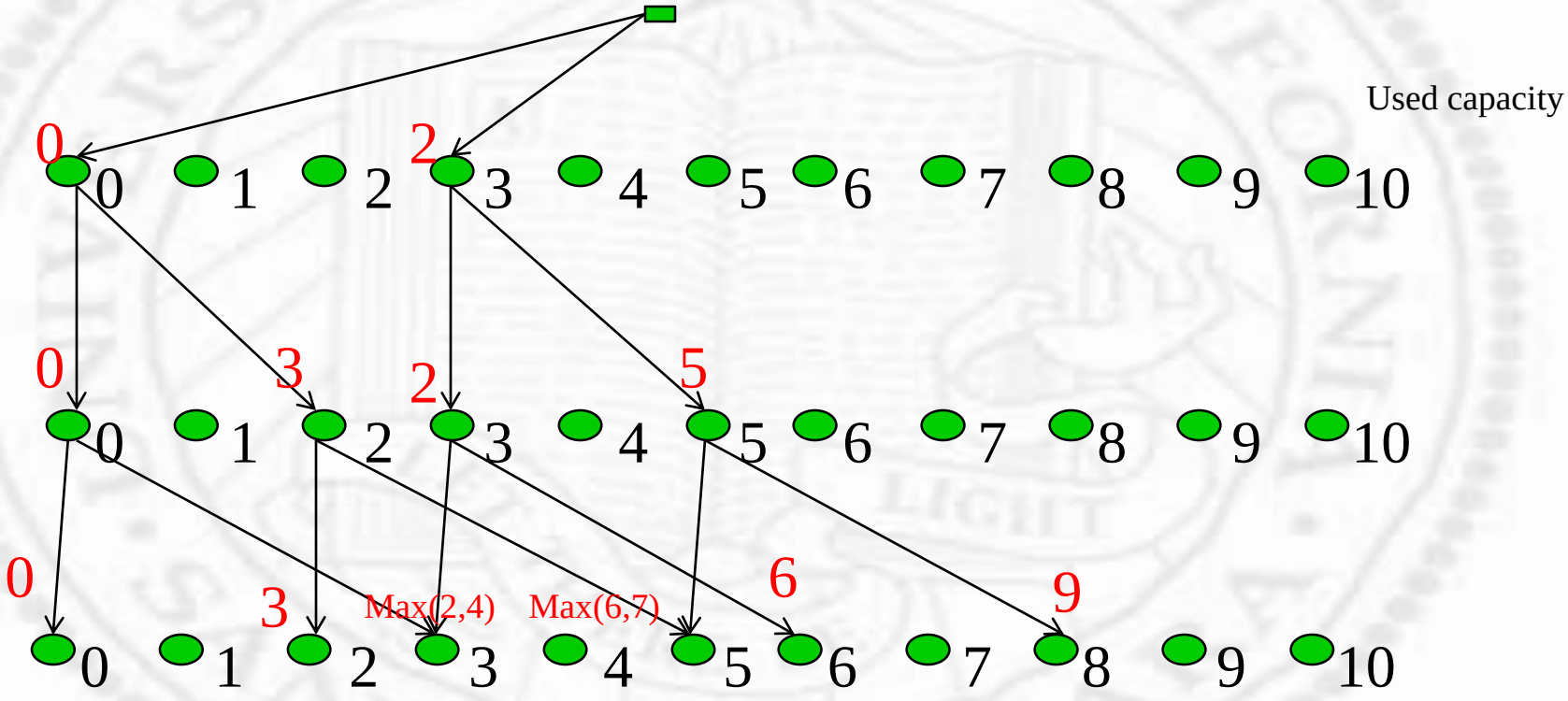
- Time complexity
  - An brushy tree for the table
  - Constant time to generate entries $O(2^n)$
  - DP can help, but the complexity will depend on actual problem instance

Three Useful Tricks

- Feasibility
  - If a branch is over capacity, don't expand it anymore
- Optimality
  - If a branch is worse than another branch (more capacity used with smaller profit), don't expand it anymore
- Feasibility and optimality are problem instance specific, cannot guarantee worst runtime in general
- Reuse
  - Remember optimal partial solutions, don't regenerate over and over again

Reuse Examples

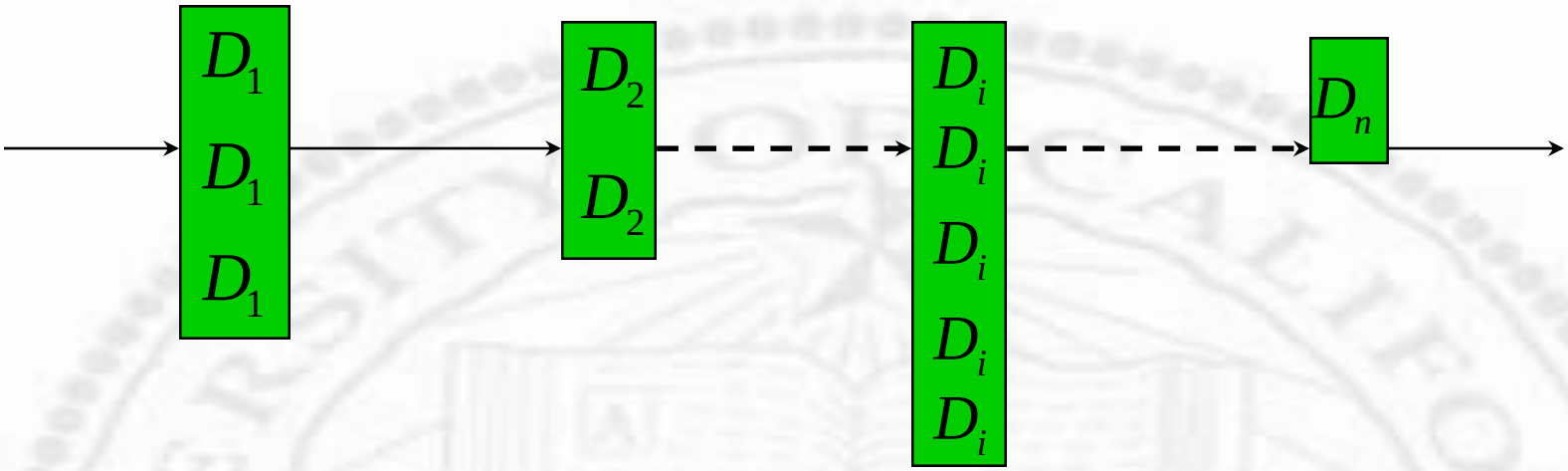
Weight = (3, 2, 3, 1, 4, 5), knapsack capacity = 10
Profit = (2, 3, 4, 1, 5, 1)



The complexity is O(n) with reuse!

Reliable Design

- Input:
  - A system composed of several devices in serial
  - Each device (D) has a fixed reliability rating (r)
  - Multiple copies of the same device can be used in parallel to increase reliability
- Output:
  - A system with highest reliability rating subject to a cost constraint



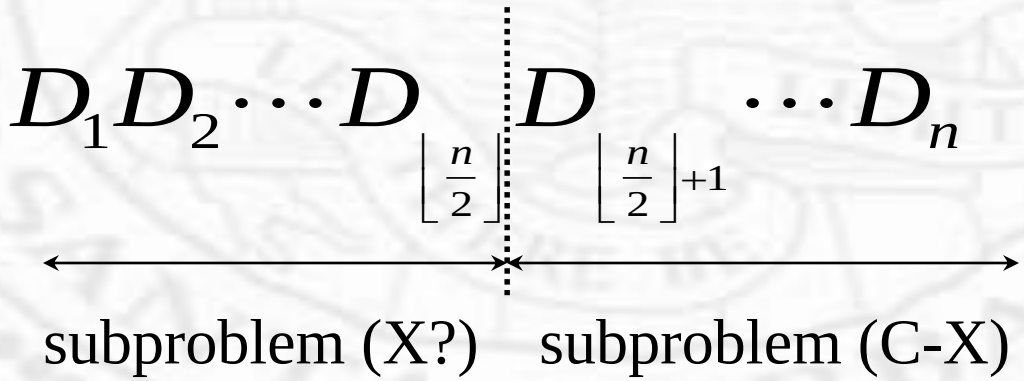
m_i copies of devices D_i at stage i, r_i : *say*, 90%

with a reliability rating of $\Phi_i = 1-(1-r_i)^{m_i}$ ← Connected in parallel
At least one should work

$$\max \prod_{1 \le i \le n} \Phi_i(m_i)$$ ← Connected in series
All of them have to work

subject to $\sum_{1 \le i \le n} c_i m_i \le C$ and $m_i \ge 1, 1 \le i \le n$

- Greedy method may not be applicable
  - Strategy to maximize reliability: Buy more less reliable units (Costs may be high)
  - Strategy to minimize cost: Buy more less expensive units (Reliability may not improve significantly)
- Divide-and-Conquer may fail



Comparison

- A knapsack of capacity C
- Objects of size *ci* and profit *pi*
- Fill up the knapsack with 0 or 1 copy of *i*
- Maximize profit

- Total expenditure of C
- Stages of cost *ci* and reliability *ri*
- Construct a system with 1 or more copies of *i*
- Maximize reliability

$$u_i = 1 + \left\lfloor \frac{C - \sum_{j=1}^{n} c_j}{c_i} \right\rfloor$$

- How to build solutions recursively?
  - One stage and one device at a time
- How does principle of optimality apply?

$$m_1 m_2 \cdots m_i \cdots m_n$$

$$(1, x \cdots x \cdots x) : (x \cdots x \cdots x) \text{ must be optimal for } C - c_1$$

$$(2, y \cdots y \cdots y) : (y \cdots y \cdots y) \text{ must be optimal for } C - 2 \times c_1$$

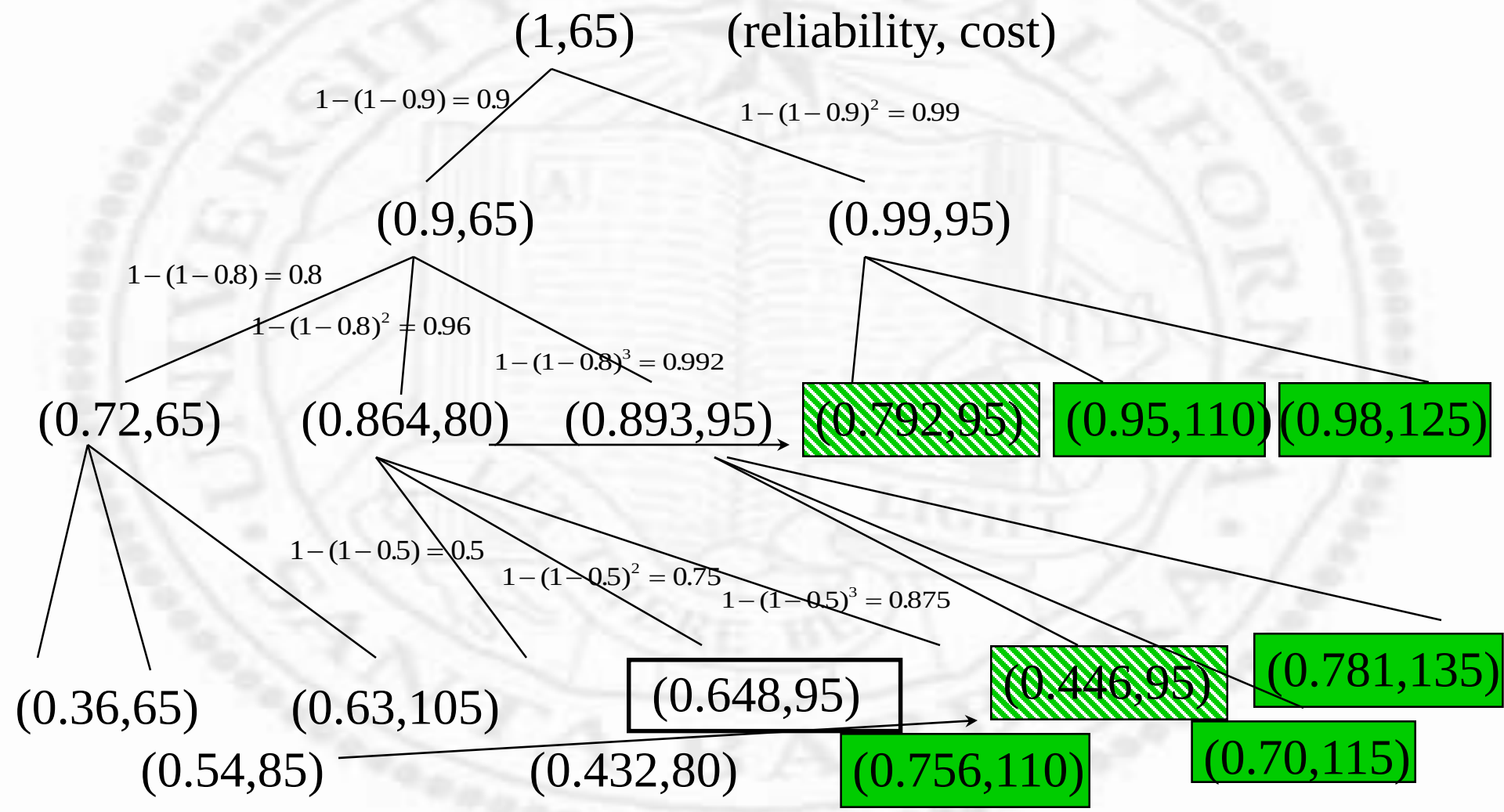
$$\cdots$$

$$(u_1, y \cdots y \cdots y) : (y \cdots y \cdots y) \text{ must be optimal for } C - u_1 \times c_1$$

- How to identify sub-optimal solutions?
  - if two solutions: (*m1*,…,*mi*,x,…,x), and (*n1*,…,*ni*,x,…, x) are such that one achieves higher reliability with a smaller cost, then the other cannot be optimal

- How to build table?

r=(0.9,0.8,0.5), c=(30,15,20),C=105



- How to write the recursive equation?

$$f_i(X): \max \prod_{1 \le j \le i} \Phi_j(m_j)$$

$$subject\ to \sum_{1 \le j \le i} c_j m_j \le X, 1 \le m_j \le \mu_{jn}, 1 \le j \le i$$

$$f_n(C) = \max_{1 \le m_n \le \mu_n} \{\Phi_n(m_n) f_{n-1}(C - c_n m_n)\}$$

$$f_i(X) = \max_{1 \le m_i \le \mu_i} \{\Phi_i(m_i) f_{i-1}(X - c_i m_i)\}$$

- Time complexity
  - An brushy tree for the table
  - Constant time to generate entries

$O(2^n)$

Chain Matrix Multiplication

- Input:
  - A sequence of *n* matrices
- Output
  - Their products

- Even though not so obvious, greedy algorithms (e.g., keep individual multiplications small) do not always produce optimal solutions
- Furthermore, the costs can vary quite a bit depending on the ordering

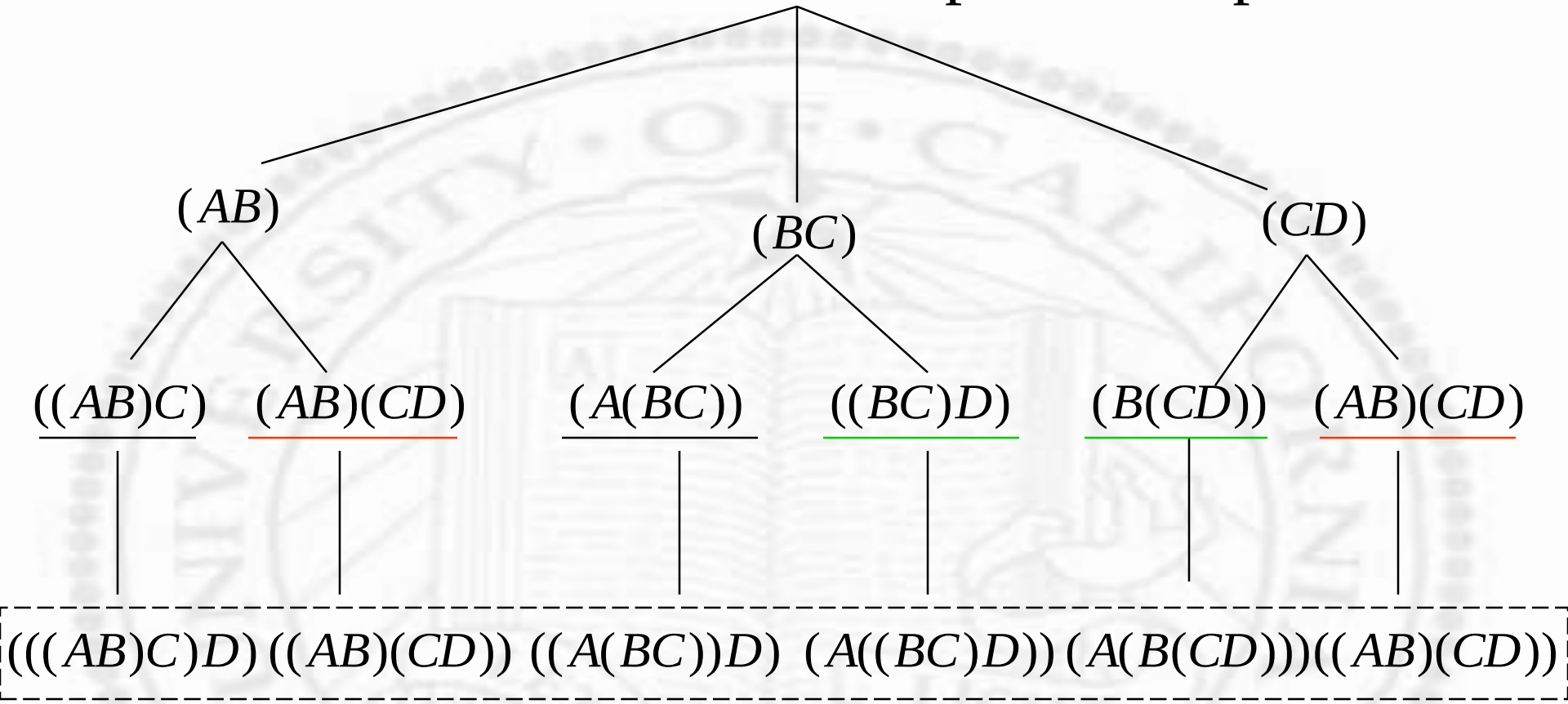
$$A_{13\times5} \times B_{5\times89} \times C_{89\times3} \times D_{3\times34}$$

$((AB)C)D$	10582
$(AB)(CD)$	54201
$(A(BC)D$	2856
$A((BC)D)$	4055
$A(B(CD))$	26418

- Can dynamic programming be used?
  - Does the principle of optimality apply?
  - Are there small problems?
  - Can the subsolutions be reused and how?

- Yes!
- There are many possible ways to apply DP, as long as
  - do things in stages
  - *merge and reuse* nodes based on principle of optimality
- We will show some examples below

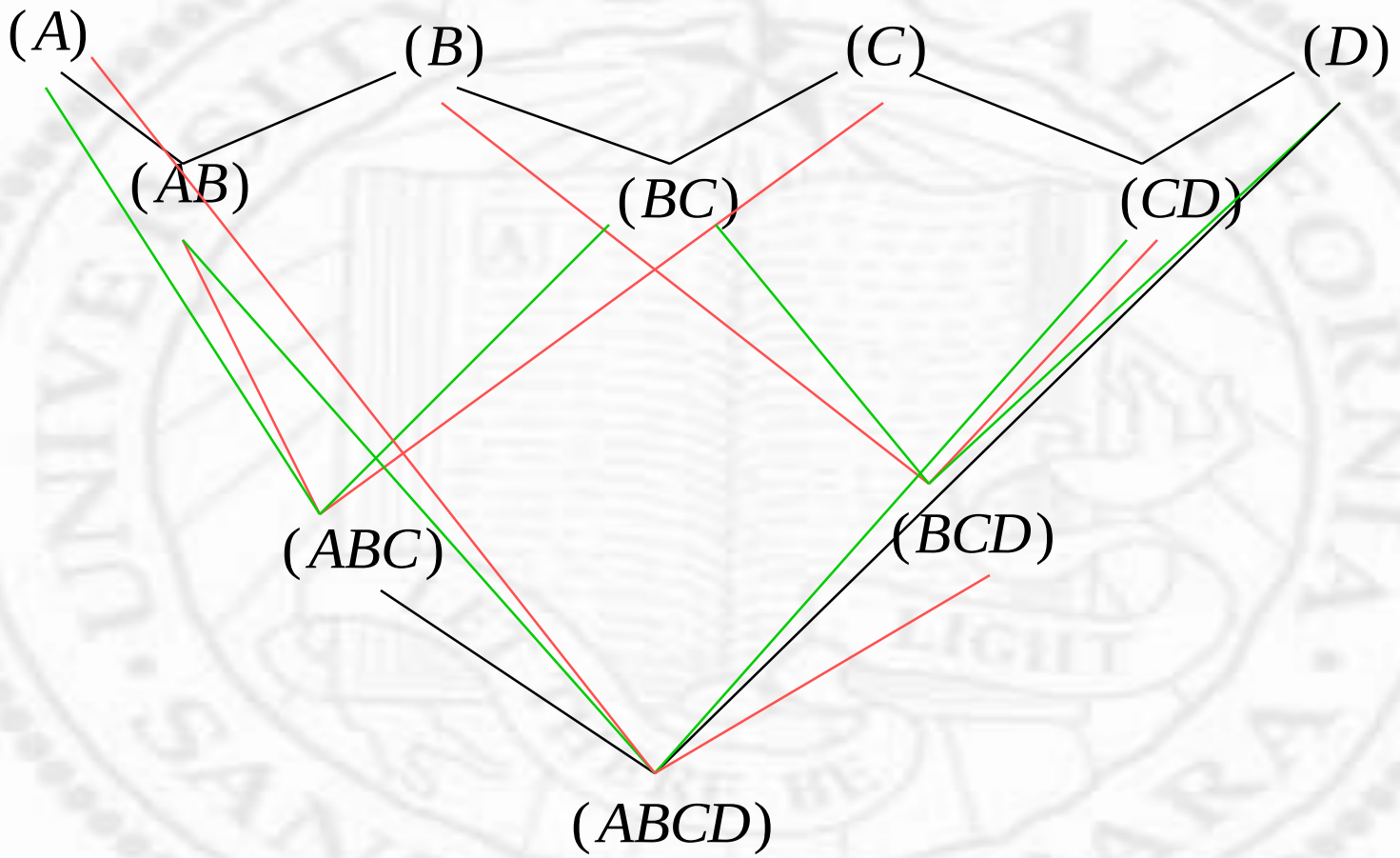
- Based on the number of multiplications performed



- An obvious DP algorithm
  - need to multiply all the matrices
  - individual steps will be multiplying two adjacent matrices and reduce the number of matrices by one
  - at each step, choose any two adjacent matrices to multiply
  - in *n-1* steps, we will be done
  - reuse: (ABC) = ((AB) C), reuse results of (AB)
  - principle of optimality: ((AB) C) and (A (BC)) produce the same results, keep one
- A perfectly legal DP algorithm!

- The problem is that it is not a good DP algorithm
  - with n matrices
  - first multiplication: *n-1* possibilities
  - second multiplication: ???
  - third multiplication: ???
  - even with reuse and principle of optimality the numbers of intermediate stages are large
  - many multiplications are repeated many times AB ((CD)(EF)) and ((AB)C)D(EF), (EF) is done more than once

- Based on the number of matrices multiplied together
  - (the range of indices)



- Does the principle of optimality apply?
  - Yes, whatever the last step in the chain multiplication, the steps leading to those two matrices must be optimal
- Are there small problems?
  - Yes, multiplications of two adjacent matrices

$$m_{i,i+1} = d_{i-1} d_i d_{i+1} \quad i = 1,2,\ldots, n-1$$

- Can the subsolutions be reused and how?
  - Yes, $$m_{i,i+s} = \min_{i \le k \le i+s} (m_{i,k} + m_{k+1,i+s} + d_{i-1} d_k d_{i+s}) \quad i = 1,2,\ldots, n-s$$

$$A_{13\times5} \times B_{5\times89} \times C_{89\times3} \times D_{3\times34}$$

	$j=1$	2	3	4	
$i=1$	0	5785	1530	2856	$s=3$
2		0	1335	1845	$s=2$
3			0	9078	$s=1$
4				0	$s=0$

$$m_{13} = \min(m_{11} + m_{23} + 13\times5\times3, m_{12} + m_{33} + 13\times89\times3) = 1530$$

$$m_{24} = \min(m_{22} + m_{34} + 5\times89\times34, m_{23} + m_{44} + 5\times3\times34) = 1845$$

$$m_{14} = \min(m_{11} + m_{24} + 13\times5\times34,$$
$$m_{12} + m_{34} + 13\times89\times34, m_{13} + m_{44} + 13\times3\times34) = 2856$$

Longest Increasing Subsequence

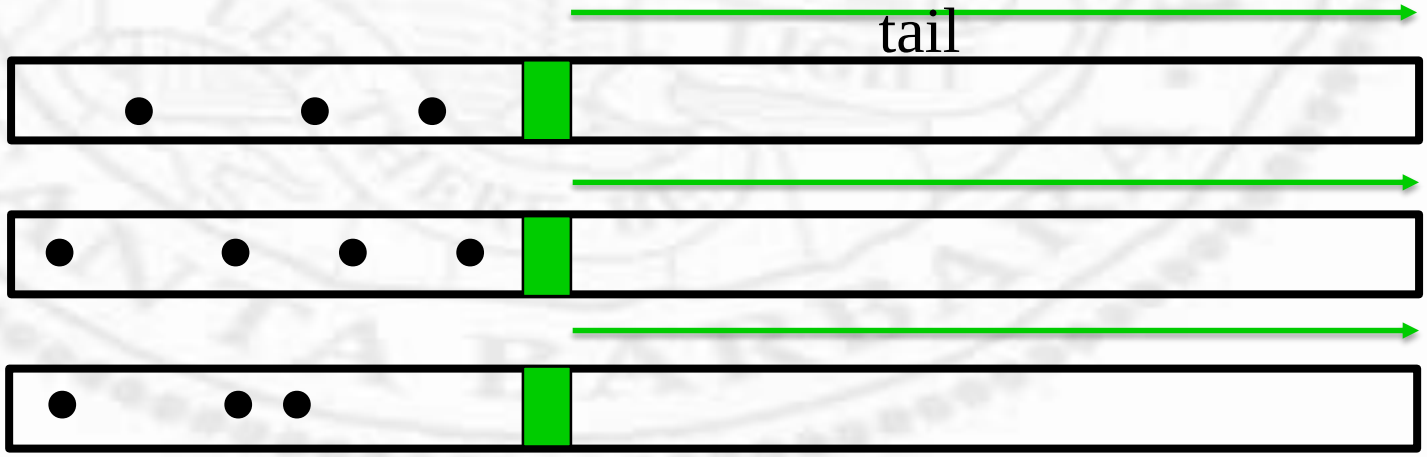
- Given an array of n numbers [0..n-1], find a subset of numbers that are increasing
- [0 8 4 12 2 10 6 14 1 9 5 13 3 11 7 15]
  - [0 8 15]
  - [2 6 11 15]
  - [2 3 7]
  - [0 2 6 9 11 15] <- longest one
  - [0 4 6 9 11 15]
  - [0 4 6 9 13 15]

Brute Force Method

- Every number can be either in or not in in LIS
- With n numbers, there are 2^n subsequences
- Generate all, discard those that are not increasing subsequences
- Complexity $O(2^n)$

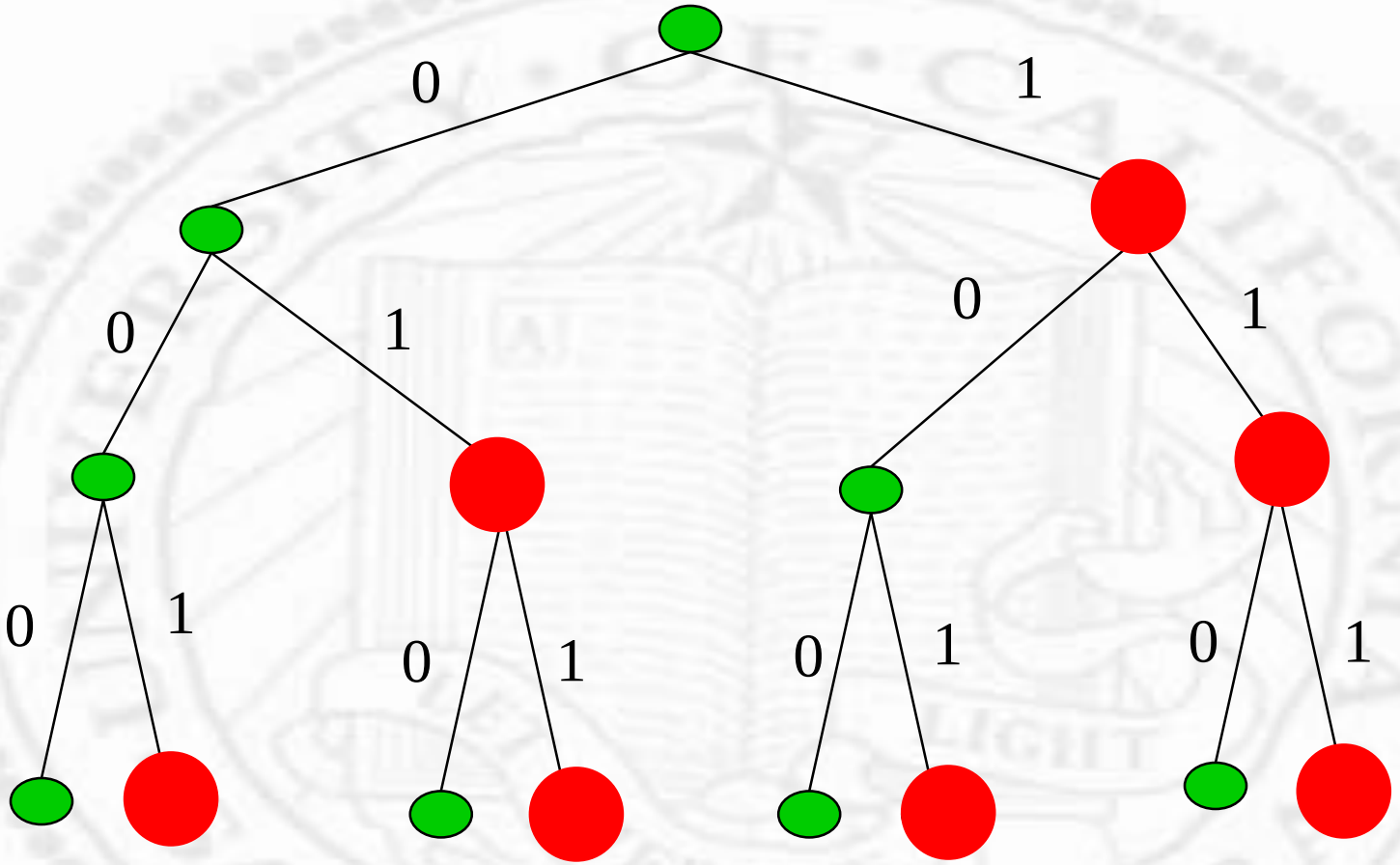
DP – Key Idea

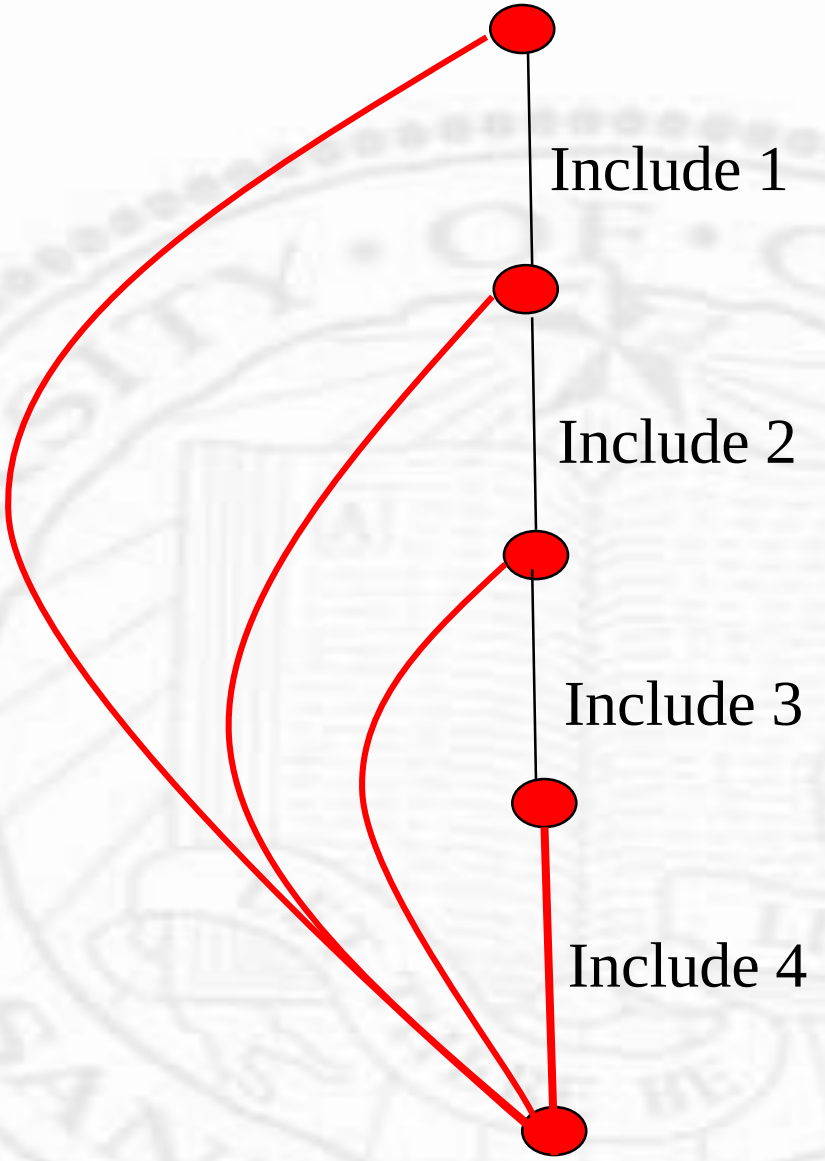
- An partial LIS soln (head) must end at some index
- The same tail portion can be added to all these solns
- Only best soln is kept



0: not in the IS

1: in the IS





$$LIS_k = \text{best}_{0<=i<k} (LIS_i + 1)$$

$LIS_k = best_{0<=i<k} (LIS_i + 1)$

0, 8, 4, 1, 2, 2, 10, 6, 14, 1, 9, 5, 13

0	1 (0)
8	2 (0, 8)
4	2 (0, 4)
1	2 (0, 1)
2	3 (0, 1, 2)
2	3 (0, 1, 2)

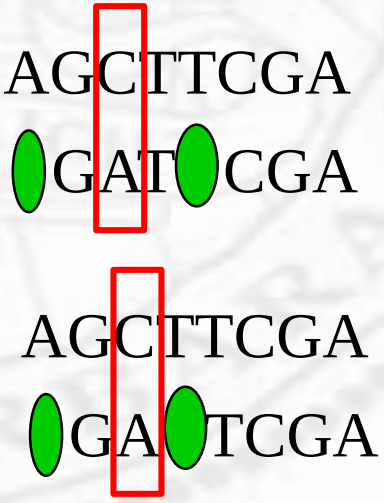
10	4 (0, 1, 2, 10) (0, 1, 2, 10)
6	4 (0, 1, 2, 6) (0, 1, 2, 6)
14	5 (0, 1, 2, 10, 14) (0, 1, 2, 10, 14 (0,1,2,6, 14) (0, 1,2,6,14)
1	2 (0, 1)
9	5 (0, 1, 2, 6, 9) (0, 1, 2, 6, 9)
5	4 (0, 1, 2 , 5) (0, 1, 2, 5)
13	6 (0, 1, 2, 6, 9, 13) (0, 1, 2, 6, 9

DP – Key Idea (Reuse)

- How to build table?
  - LIS_k = best $_{0<=i<k}$ (LIS_i +1)
- Final solution?
  - LIS = best $_{0<=k<n}$ (LIS_k)
- Complexity
  - One table
  - n entries LIS_k
  - Most expensive entry O(n)
  - $O(n^2)$
  - More efficient (O(nlogn)) exists

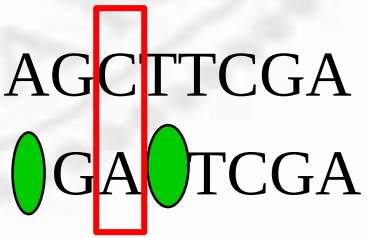
Sequence Alignment

- Given two sequences a, b of length m, n
- Align them to match
- Use in DNA matching:
  - a: AGCTTCGA
  - b: GATCGA
- Deletion (insertion):
  - 1st A, 4th or5th T
- Change:
  - 3rd C->A



Constraints

- Linear ordering
  - If a_i matches with b_j,
    - a_k, $k<i$ must match only with b_l, $l<j$
    - a_k, $k>i$ must match only with b_l, $l>j$
- Deletion, insertion, and change all have associated costs (domain dependent)
- Also called the longest common subsequence (LCS) problem (GTCGA)

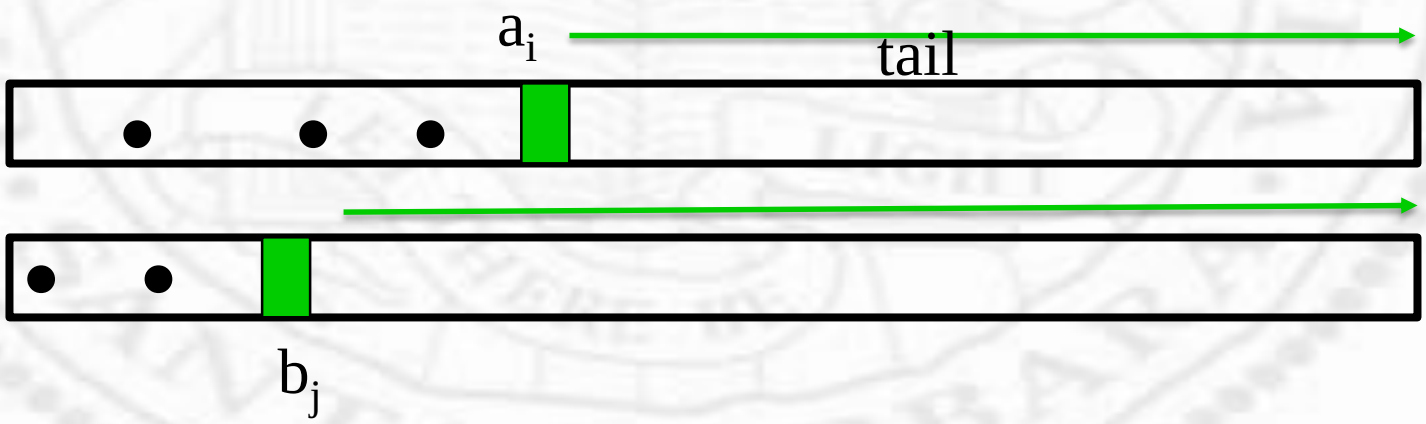


Brute Force Method

- Again, think about tree
- At each tree node, looking at some a_i and some b_j (initially, a_o, b_o)
  - Match a_i and b_j
    - No change necessary
    - Change a_i <-> b_j
  - Skip (delete) a_i , but keep b_j
  - Skip (delete) b_j , but keep a_i
  - Skip (delete) both a_i and b_j
- Max fan out is 4, Max tree depth is m+n, bad

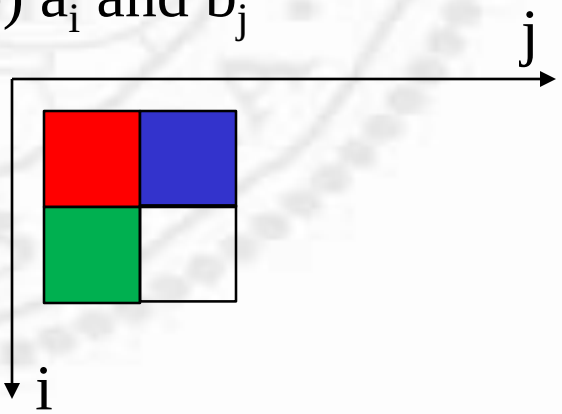
Principle of Optimality

- Similar to LIS
  - Head: some partial results (match, delete, insert, change, etc.) up to a_i and b_j
  - Tail: (match, delete, insert, change) results for a_{i+1} and b_{j+1}



Reuse

- Build a table of size m by n to store the partial results
  - (i,j)th entry is results up to a_i b_j
- How to fill the table?
  - Fill them in diagonally
  - C(i,j) (W) = min among
    - R: C(i-1,j-1) + (match, change, skip) a_i and b_j
    - G: C(i,j-1) + skip b_j
    - B: C(i-1,j) + skip a_i
- Complexity: $O(n^2)$



a: AGCTTCGA
b: GATCGA

Match: 0
Delete (insert): 1
Change: 1

b

a \ b	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1						
2	2						
3	3						
4	4						
5	5						
6	6						
7	7						
8	8						

a

a: AGCTTCGA
b: GATCGA

Match: 0
Delete (insert): 1
Change: 1

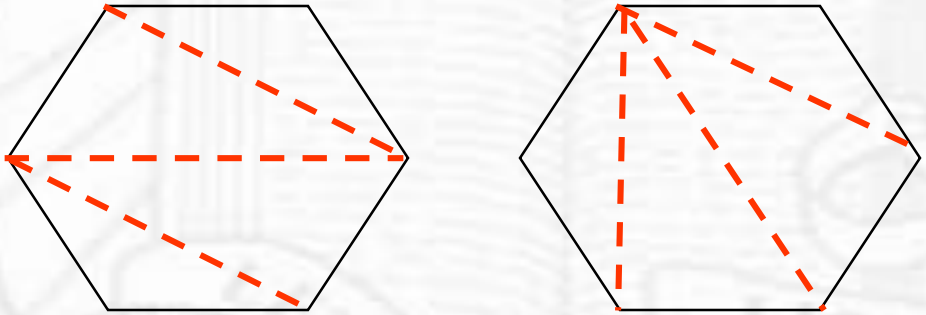
b

a \ b	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	1					
2	2						
3	3						
4	4			?			
5	5						
6	6						
7	7						
8	8						

a

Polygon Triangulation

- Given: A convex polygon with *n* sides, a triangulation is a set of chords of the polygon that divide the polygon into disjoint triangles



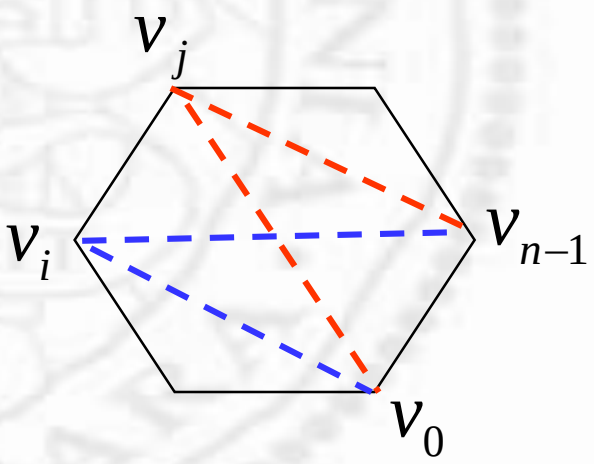
- There are *n-2* triangles with *n-3* chords

- Not all triangulations are equally good
- Need a cost function to evaluate the cost of a triangulation
- The cost of a triangulation is the total costs of its component triangles
- The cost of a particular triangle is the sum of some distance measure (e.g., Euclidean) of all its sides

$$\Delta(v_i, v_j, v_k) = |v_i, v_j| + |v_j, v_k| + |v_k, v_i|$$

- Again, divide-and-conquer might not work
  - say, chose a chord to divide the polygon into two parts and perform triangulation for both parts independently
  - the said chord will be in the final triangulation
  - however, the optimal triangulation may not include that particular chord
- Greedy?
  - the polygon of the smallest cost may not be in the final triangulation
- Need to look at all possible combinations

- Intuitively, when we consider the first step in triangulation, say, using v(0) and v(n-1) as base, the vertex can be v(1), v(2), …, v(n-2)
  - we do not know which one is the best, should consider all possibilities
- Furthermore, if we pick, say v(j), then triangulation of v(0) to v(j) and v(j) to v(n-1) must be optimal w.r.t each sub problems (principle of optimality)

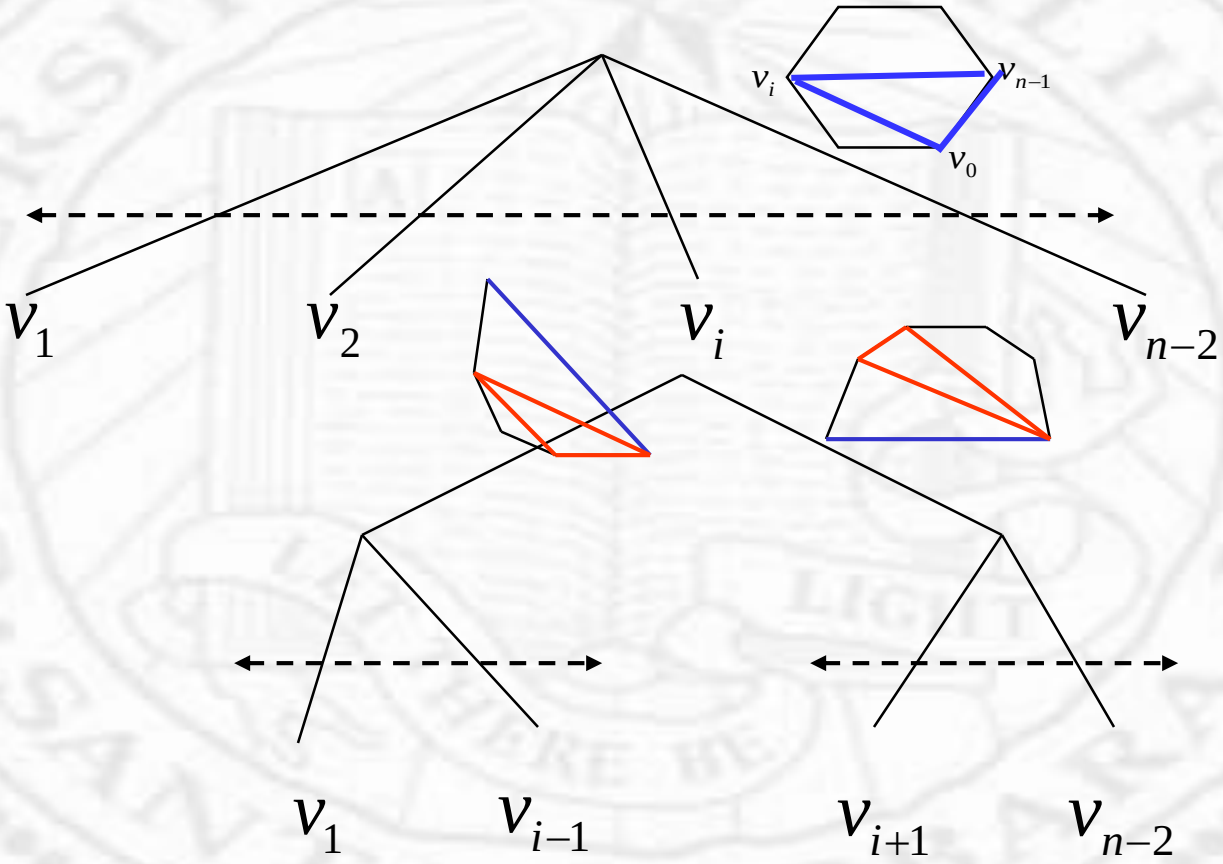


Like Matrix Multiplication

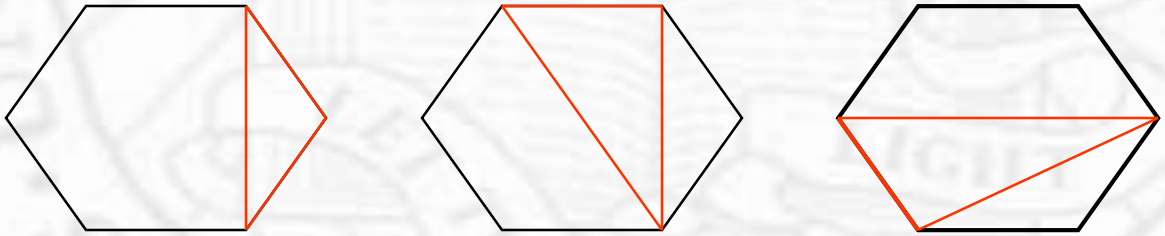
- Multiple two matrices
- Multiply n-1 times
- Randomly pick two matrices to multiply
- Better to do it by gradually enlarge the chain
- Adjacency

- Create one triangle
- Create n-2 triangles
- Randomly pick a triangle to add
- Better to do it by gradually enlarge the triangle area
- Adjacency

- Principle of Optimality



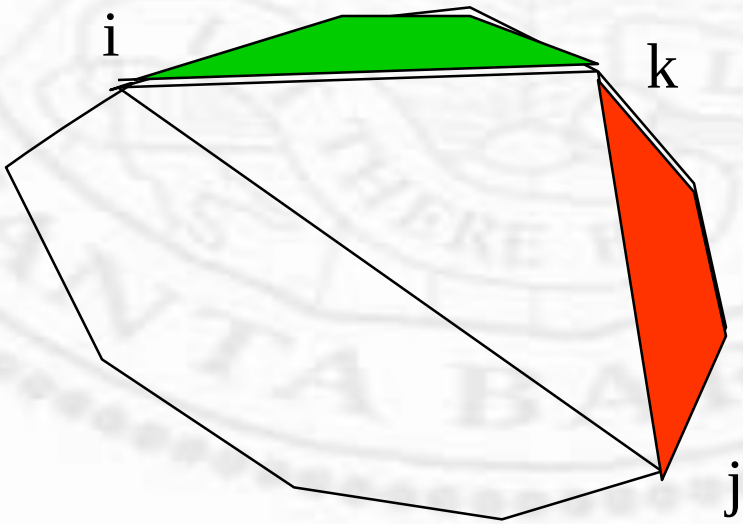
- Reuse
  - if we need to triangulate an area inside the original polygon spanned by, say *k*, vertices
  - if we already know the best way to triangulate an area spanned by *k-1* vertices or less
  - then we can take advantage of that!



- That allows us to write the following recurrence relation
  - let *c(i,j)* be the cost of an optimal triangulation of polygon <v(i), v(i+1), … v(j)>, then

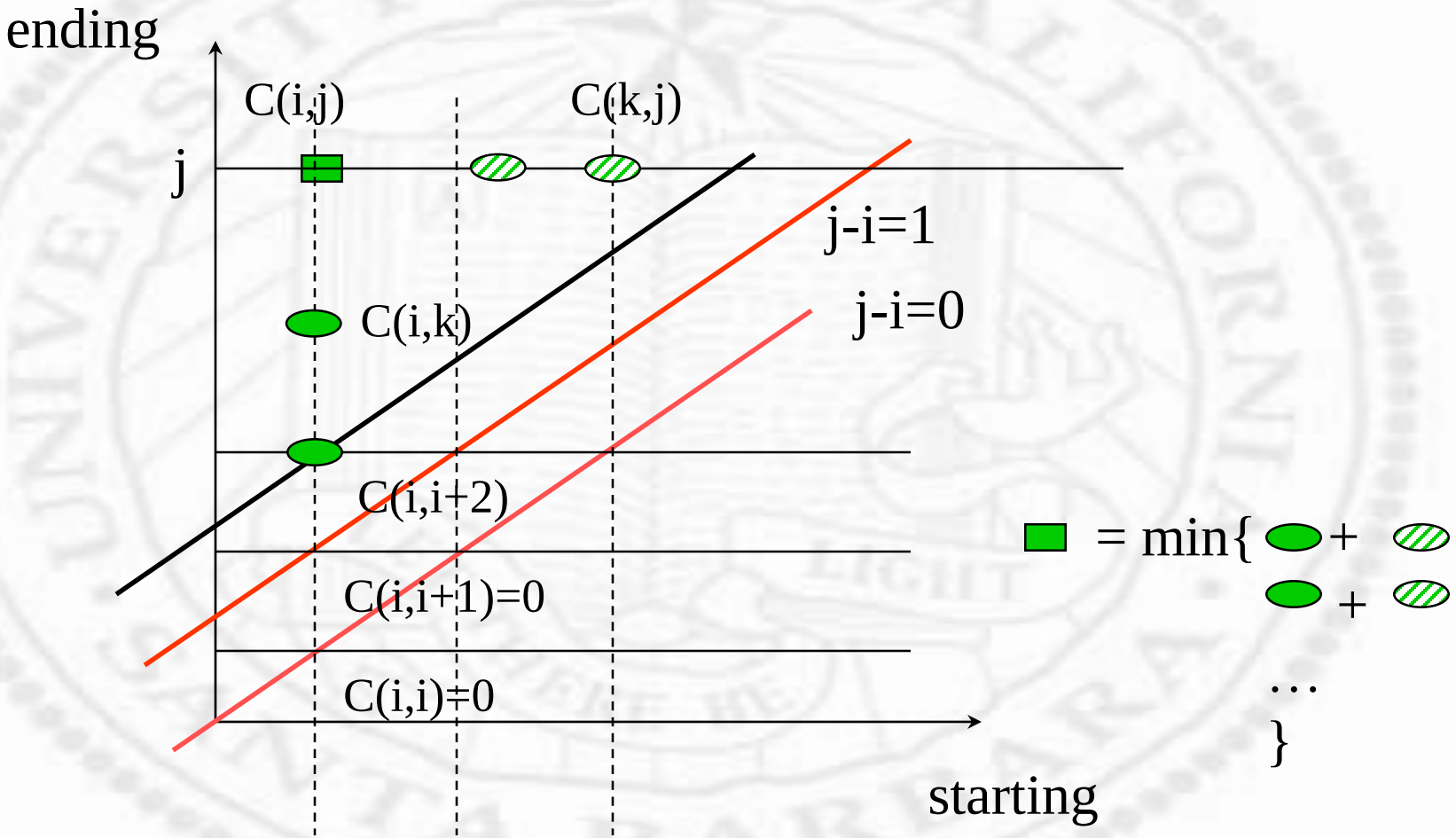
$$c(i, j) = 0 \qquad i = j \text{ or } i = j - 1$$

$$c(i, j) = \min_{i<k<j}(c(i,k) + c(k, j) + \Delta(v_i, v_k, v_j)) \qquad i + 1 < j$$



$$c(i, j) = 0 \qquad i = j \text{ or } i = j - 1$$

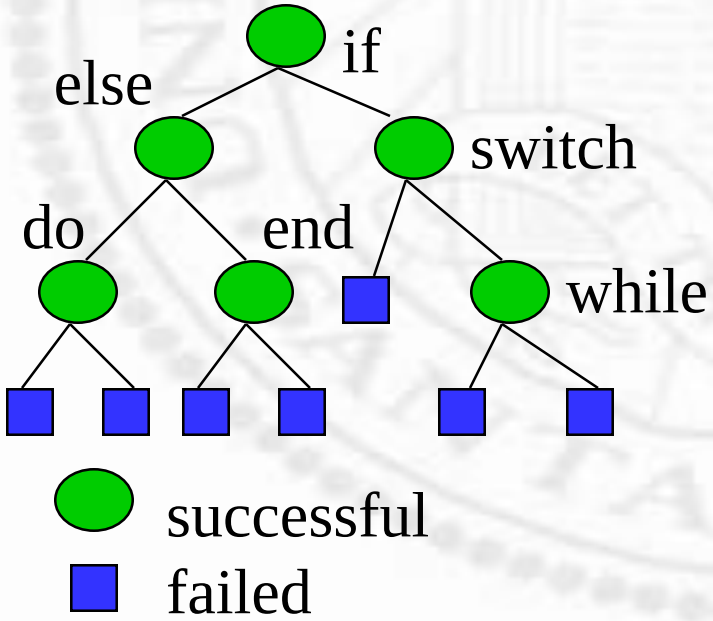
$$c(i, j) = \min_{i<k<j}(c(i,k) + c(k, j) + \Delta(v_i, v_k, v_j) \qquad i + 1 < j$$



Optimal Binary Search Tree

- Input:
  - A set of *n* identifiers
- Output:
  - An optimal binary search tree that minimizes the average search effort

- n identifiers $\{a_1 \le a_2 \le,...,\le a_n\}$
- $P(i)$: probability that a_i is searched
- $Q(i)$: probability that search is for a symbol E_i that $a_i < E_i < a_{i+1}$
- $\sum_{i=1}^{n} P(i) + \sum_{i=0}^{n} Q(i) = 1$

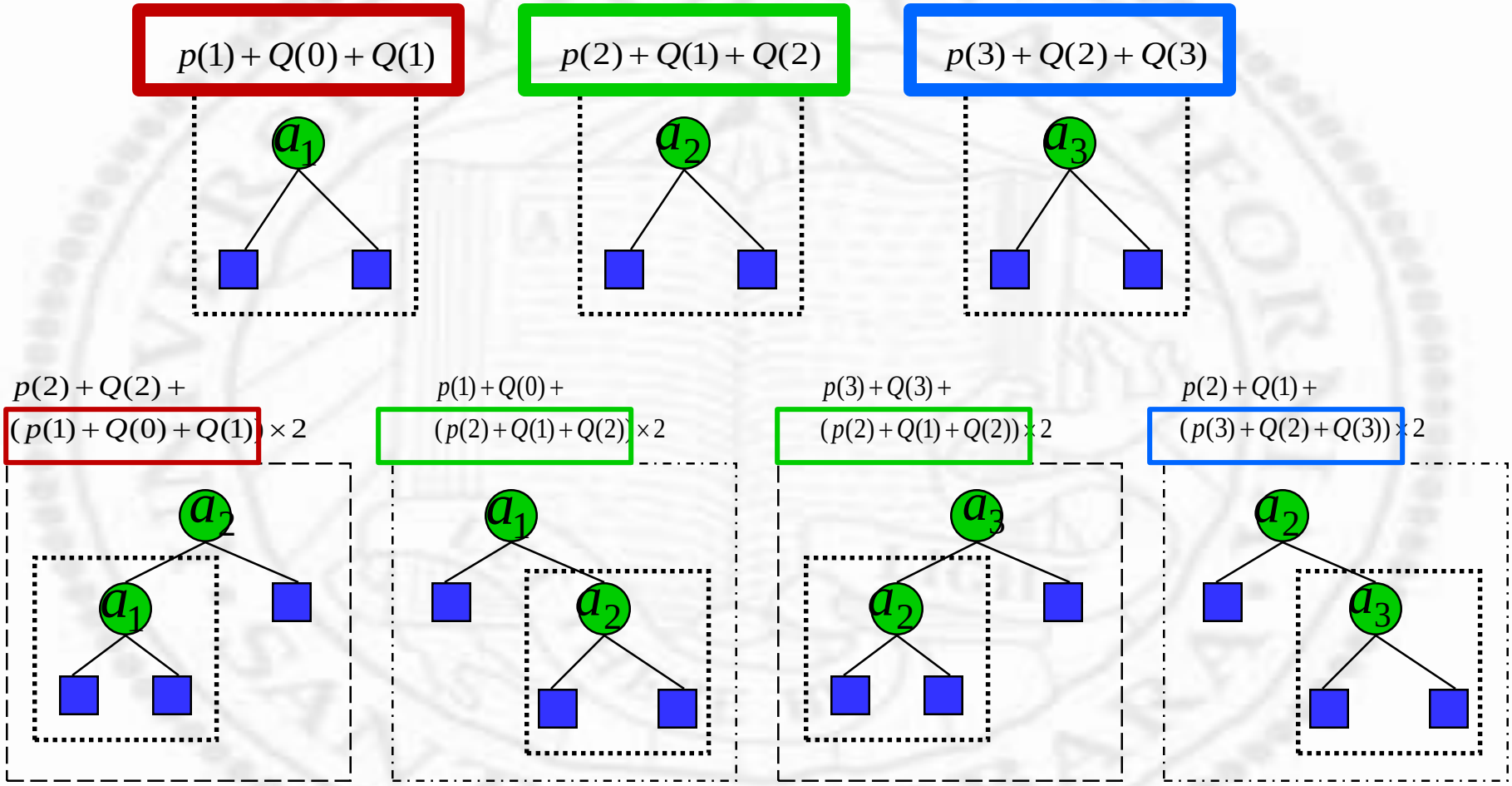


$$successful: \sum_{i=1}^{n} P(i) \times level(a_i)$$

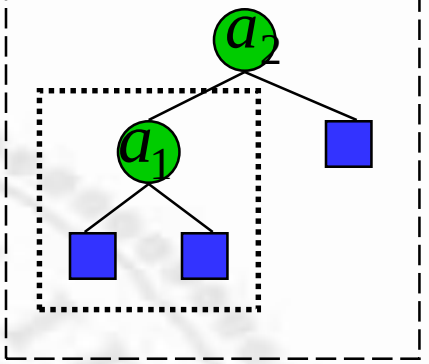
$$failed: \sum_{i=0}^{n} Q(i) \times (level(E_i) - 1)$$

- Convince yourself that divide-and-conquer and greedy methods are not suitable
- Dynamic Programming
  - Identify small problems
  - Progressively build larger problems
  - *Reuse* optimal sub-solutions (table building)

How to build solution recursively (*reuse*)?

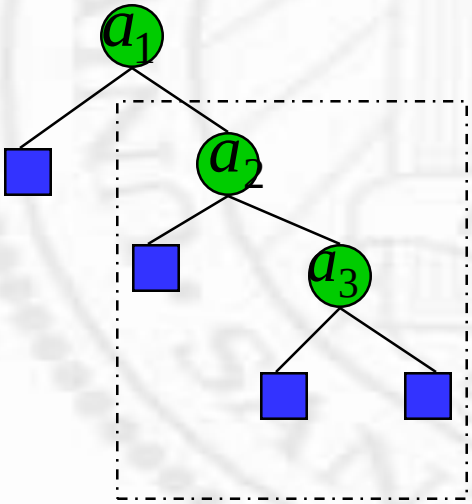


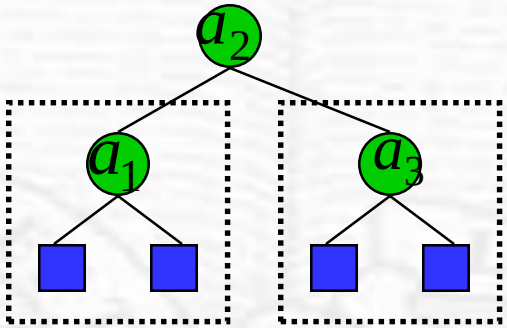
$$p(2) + Q(2) + (p(1) + Q(0) + Q(1)) \times 2$$



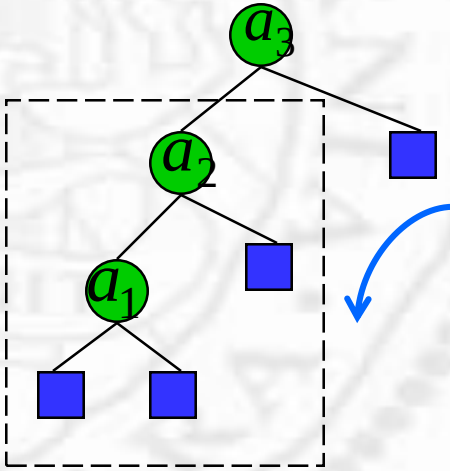
$$p(2) + (p(1) + Q(0) + Q(1)) \times 2 + (p(3) + Q(2) + Q(3)) \times 2$$

$$p(1) + Q(0) + ((p(2) + Q(1)) \times 2 + (p(3) + Q(2) + Q(3)) \times 3)$$





$$p(3) + Q(3) + ((p(2) + Q(2)) \times 2 + (p(1) + Q(0) + Q(1)) \times 3)$$



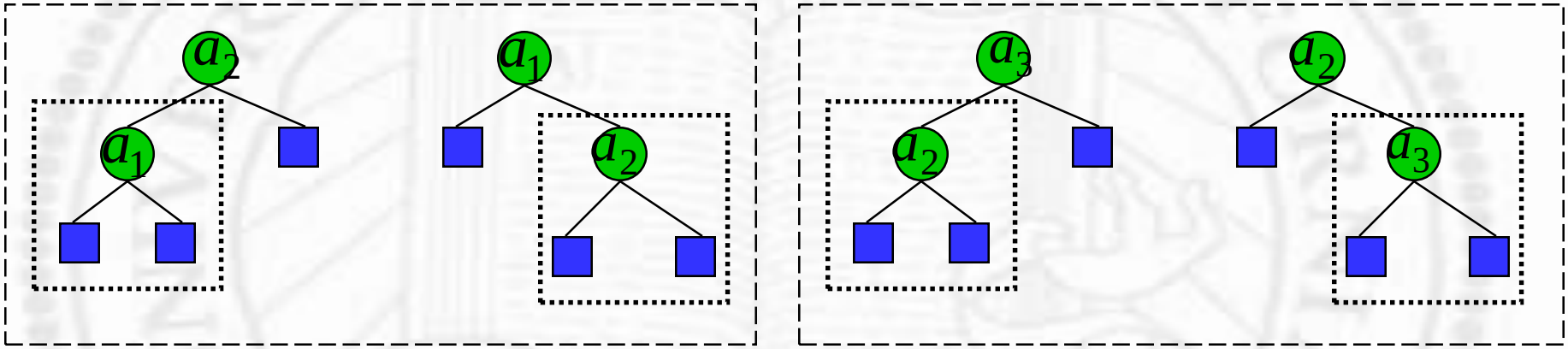
How does principle of optimality apply?

$p(2)+Q(2)+$
$(p(1)+Q(0)+Q(1))\times 2$

$p(1)+Q(0)+$
$(p(2)+Q(1)+Q(2))\times 2$

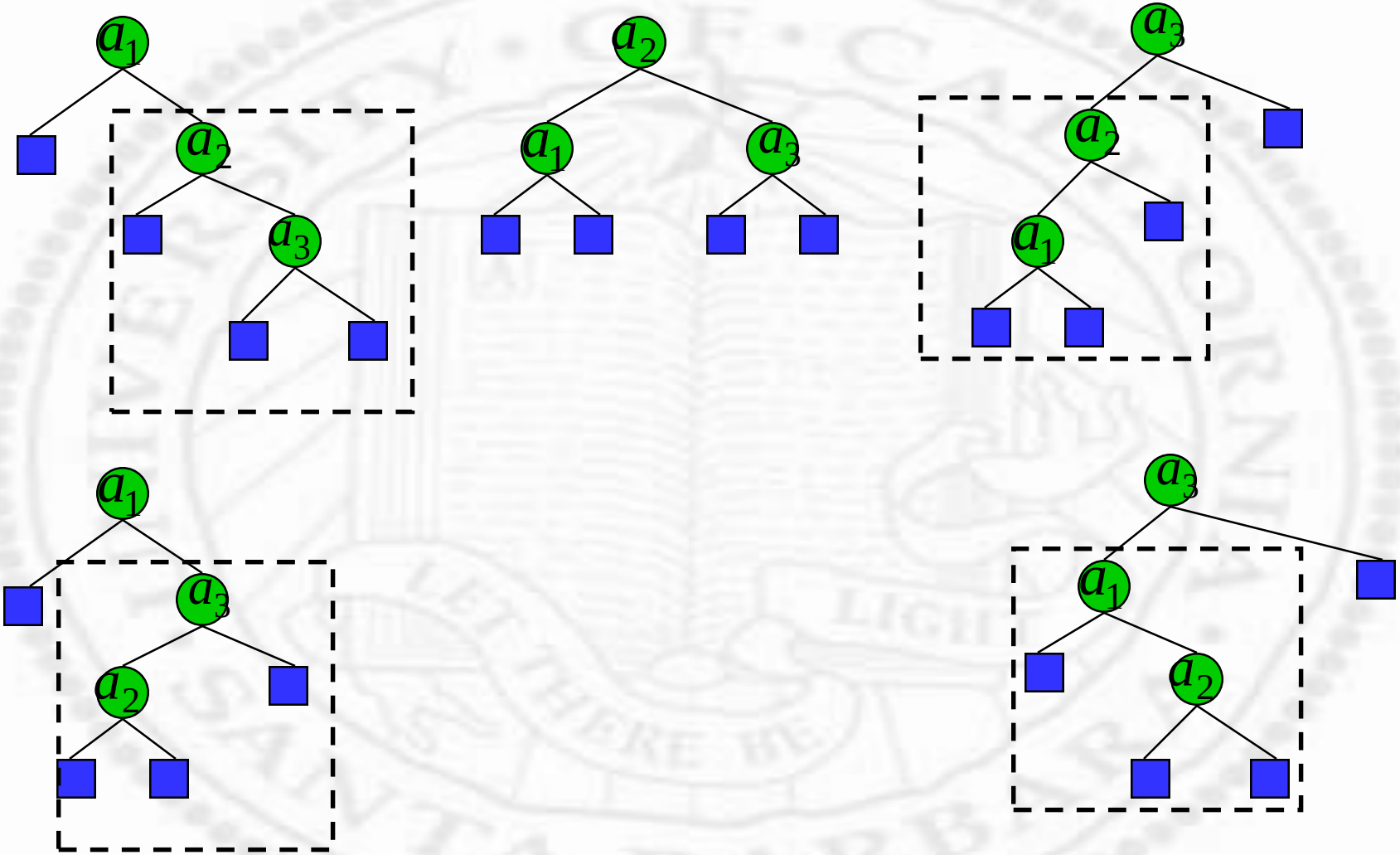
$p(3)+Q(3)+$
$(p(2)+Q(1)+Q(2))\times 2$

$p(2)+Q(1)+$
$(p(3)+Q(2)+Q(3))\times 3$



Only one of the two configurations should be kept!

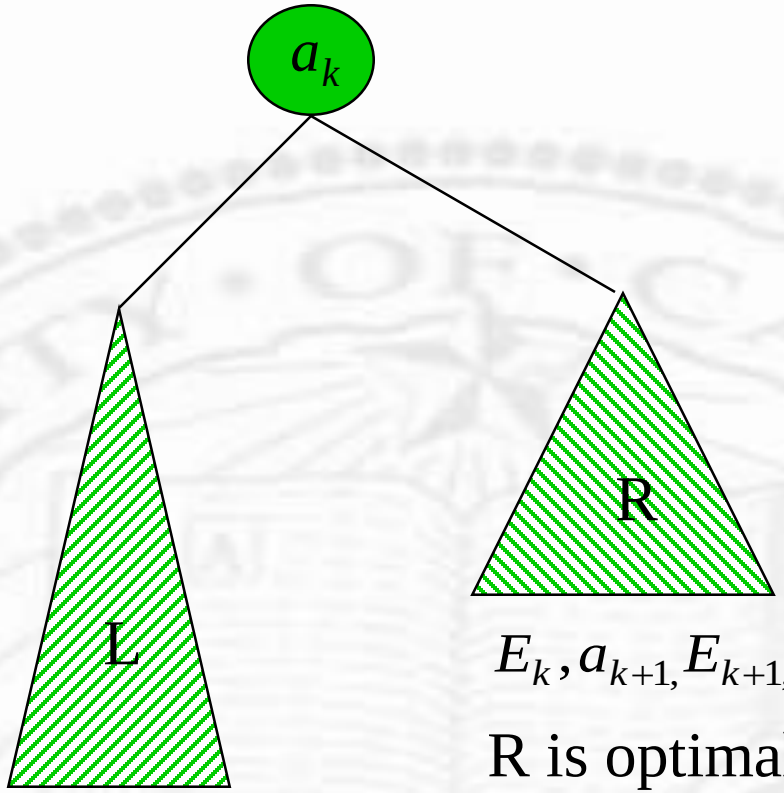
Only one of the five configurations should be kept!



Like Matrix Multiplication

- Multiple two matrices
- Multiply n-1 times
- Randomly pick two matrices to multiply
- Better to do it by gradually enlarge the chain
- Adjacency

- Pull one node up to root
- Create n-2 subtrees
- Randomly pick a node up to be root
- Better to do it by gradually enlarge the subtree size
- Adjacency



$E_0, a_1, E_1, a_2, \ldots, a_{k-1}, E_{k-1}$

L is optimal w.r.t.
all binary search trees
with the above elements

$E_k, a_{k+1}, E_{k+1}, a_{k+2}, \ldots, a_n, E_n$

R is optimal w.r.t.
all binary search trees
with the above elements

$$C(L) = \sum_{1 \le i < k} p(i) \times level(a_i) + \sum_{0 \le i < k} Q(i) \times (level(E_i) - 1)$$

$$C(R) = \sum_{k < i \le n} p(i) \times level(a_i) + \sum_{k \le i \le n} Q(i) \times (level(E_i) - 1)$$

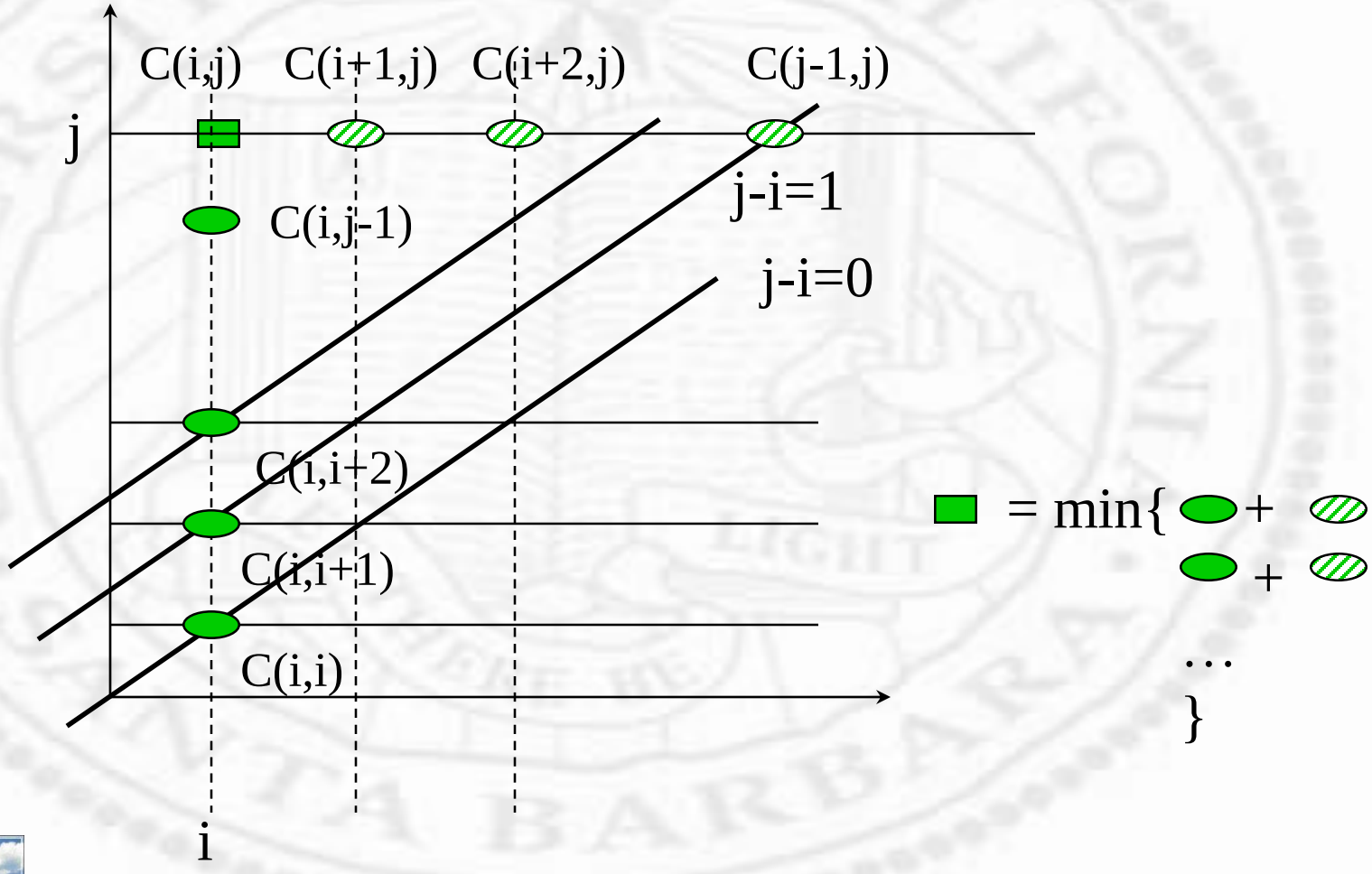
$$C(combined) = p(k) + C(L) + C(R)$$
$$+ \sum_{1 \le i < k} p(i) + \sum_{0 \le i < k} Q(i)$$
$$+ \sum_{k < i \le n} p(i) + \sum_{k \le i \le n} Q(i)$$
$$= C(L) + C(R) + W(combined)$$
$$W(combined) = \sum_{1 \le i < k} p(i) + \sum_{0 \le i < k} Q(i) + p(k) + \sum_{k < i \le n} p(i) + \sum_{k \le i \le n} Q(i)$$
$$= \sum_{1 \le i \le n} p(i) + \sum_{0 \le i \le n} Q(i)$$

Red: left got one deeper
Blue: right got one deeper
Green: root

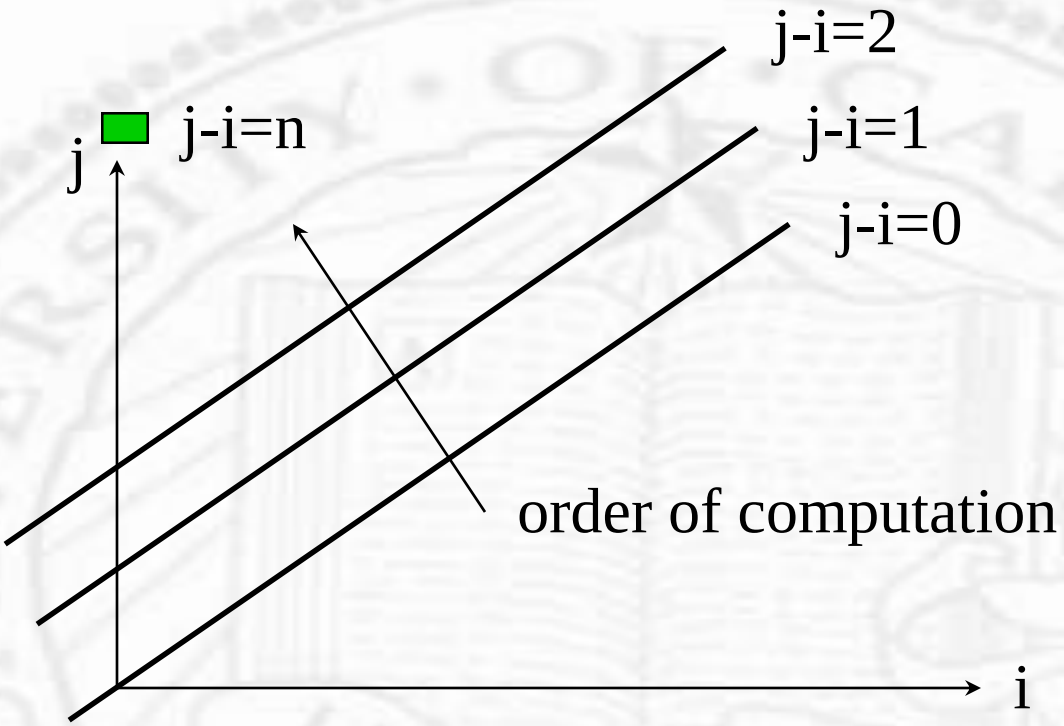
- Recurrence relation

$$C(0,n) = \min_{0 < k \le n} \{C(0,k-1) + C(k,n) + W(0,n)\}$$
$$C(i,j) = \min_{i < k \le j} \{C(i,k-1) + C(k,j) + W(i,j)\}$$

$$C(i,j) = \min_{i<k\le j}\{C(i,k-1) + C(k,j) + W(i,j)\}$$



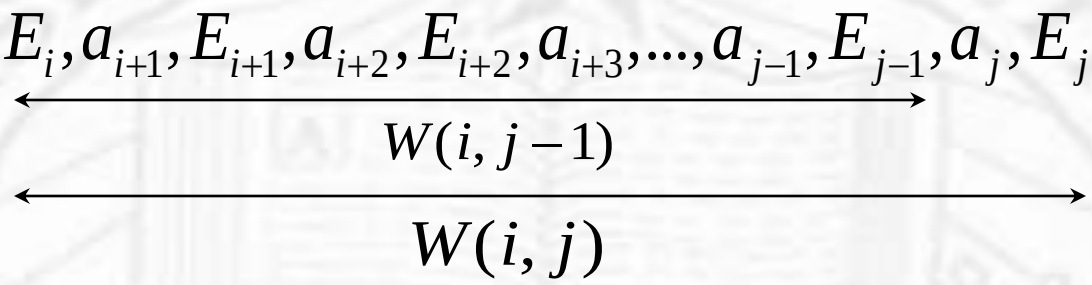
- Table building



$$C(i,i) = 0$$

$$W(i,i) = \sum_{i+1 \le k \le i} P(k) + \sum_{i \le k \le i} Q(k) = Q(i)$$

- How to compute *W(i,j)*?
  - Why not recursively?



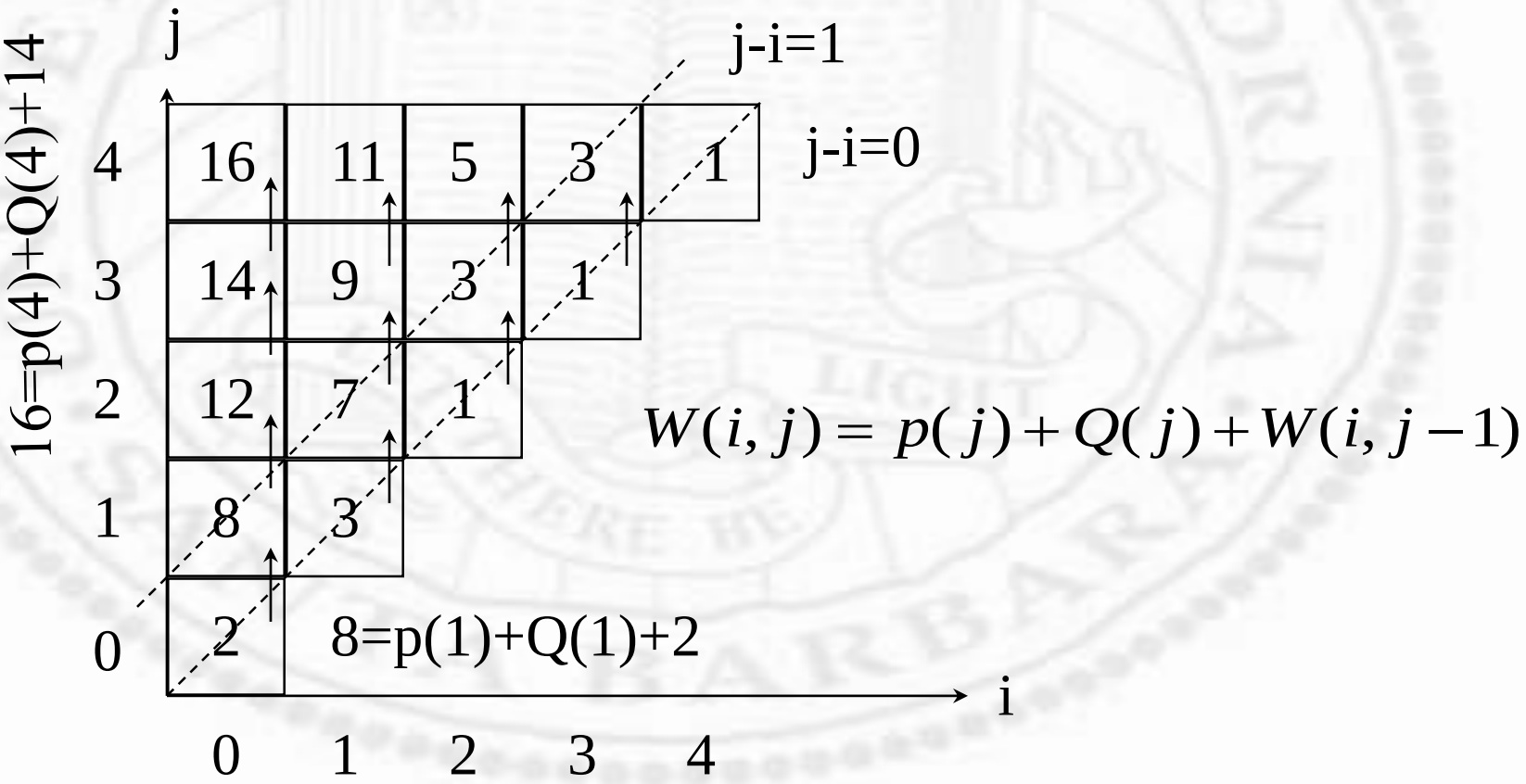
$$W(i, j) = p(j) + Q(j) + W(i, j-1)$$

$n = 4$

$(a_1, a_2, a_3, a_4) = (do, if, read, while)$

$(p_1, p_2, p_3, p_4) = (3,3,1,1)$

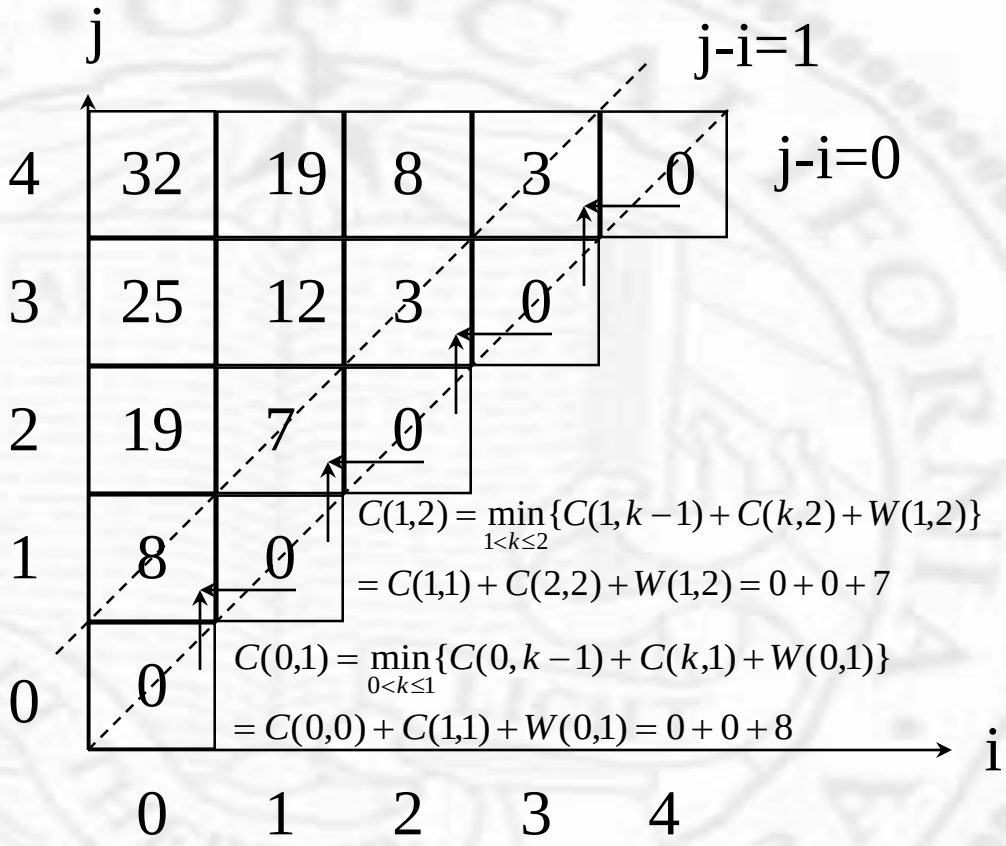
$(Q_0, Q_1, Q_2, Q_3, Q_4) = (2,3,1,1,1)$



$$W(i, j) = p(j) + Q(j) + W(i, j-1)$$

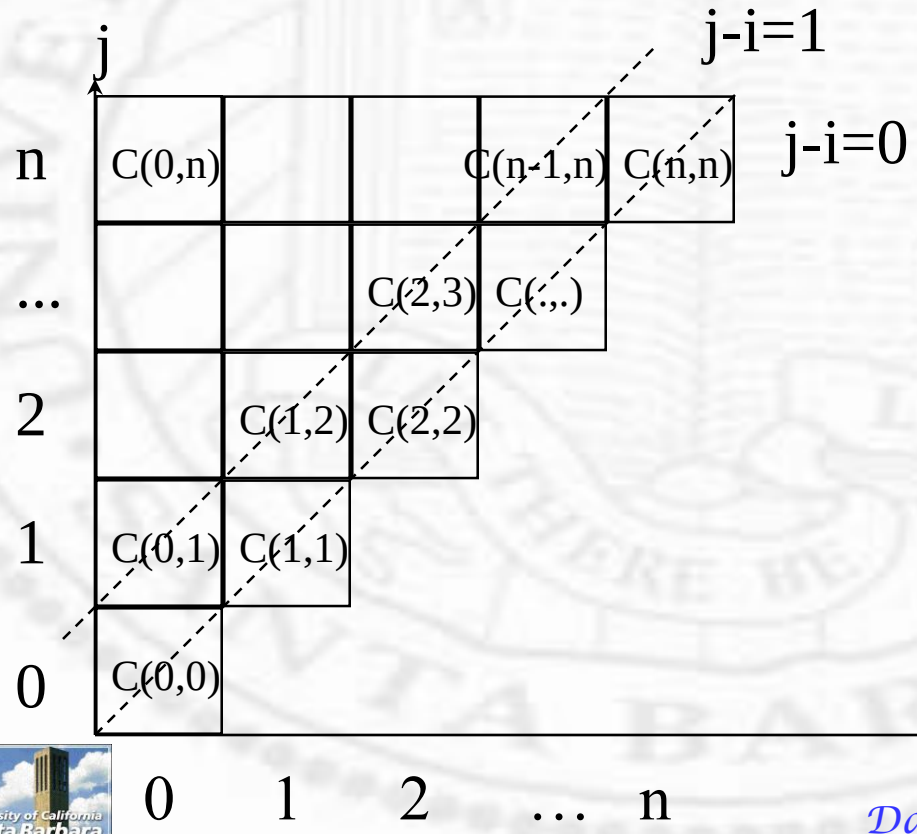
$$C(i,j) = \min_{i<k\le j}\{C(i,k-1) + C(k,j) + W(i,j)\}$$

$$C(0,4) = \min_{0<k\le 4}\{C(0,k-1) + C(k,4) + W(0,4)\}$$
$$= \min\{$$
$$C(0,0) + C(1,4), (0+19)$$
$$C(0,1) + C(2,4), (8+8)$$
$$C(0,2) + C(3,4), (19+3)$$
$$C(0,3) + C(4,4), (25+0)\}$$
$$+ W(0,4), (16) = 8+8+16$$



- Time Complexity - *C(i,j)*
  - *j-i=m*, there are *n-m+1* of them
  - Each one takes minimum of *m* quantities

$$\sum_{0 \le m \le n} (n-m+1)m = O(n^3)$$



DP-based Graph Algorithms

- Graph = (vertices, edges)
- Edges
  - Build a long path with many edges with short paths with fewer edges
  - Stop when the path is longer than min(e,n)
- Vertices
  - Build a subgraph with with many vertices with smaller subgraphs with fewer vertices
  - Stop when all vertices are considered

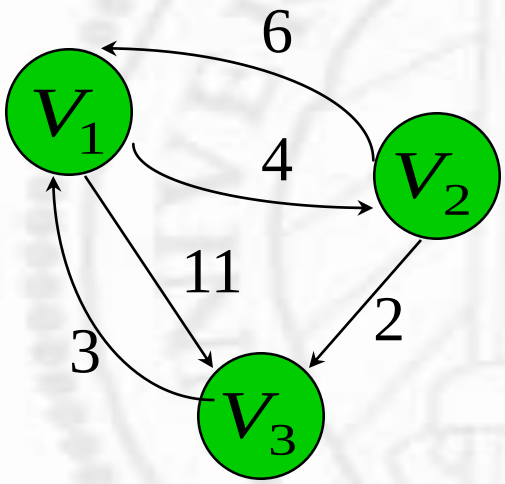
All Pairs of Shortest Paths

- Input: a labeled graph $G=(V,E)$
- Output: the shortest path from very vertex to very other vertices

- Solution 1: Iteration on the number of edges
  - a direct path (length=1)
  - paths of length=2
  - …
  - paths of length=min(*e*,*n*)
- Solution 2: Iteration on the number of vertices
  - a direct path (no intervening vertices)
  - paths with one intervening vertex
  - …
  - paths with (n-2) intervening vertices

- Both solutions have this recurrence relation:

$$C^t(i,j) = \min\{C^{t-1}(i,j), \min_{k \neq i,j}\{C^1(i,k) + C^{t-1}(k,j)\}\}$$



Going through no other vertex

A1	1	2	3
1	0	4	11
2	6	0	2
3	3	inf	0

Going through one vertex

A1'	1	2	3
1	0+0 4+6(0) 11+3	0+4 4+0(4) 11+inf	0+11 4+2(6) 11+0
2	6+0 0+6(5) 2+3	6+4 0+0(0) 2+inf	6+11 0+3(2) 2+0
3	3+0 inf+6(3) 0+3	3+4 inf+0(7) 0+inf	3+11 inf+2(0) 0+0

min

A2	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

- Time Complexity
  - How many tables? *O(min(e,n))*
  - How many entries per table? $O(n^2)$
  - *How much effort to generate each entry*? *O(n)*

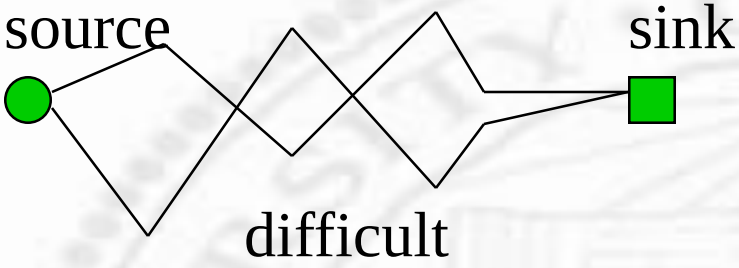
$$O(n^3 \times O(\min(e,n)) \approx O(n^4)$$

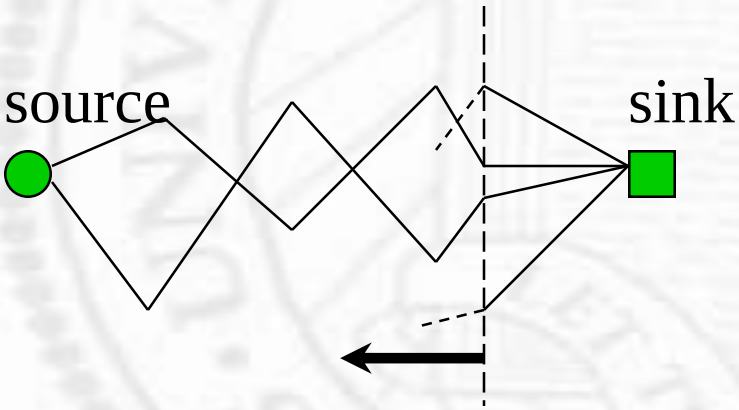
- OK solutions, but not great
- Try Floyd algorithm "iterating on vertex's cardinal number"

Traveling Salesperson

- Input: a directed labeled graph $G=(V,E)$
- Output: a tour of the minimum cost
  - a tour visits all vertices
  - a tour visit any vertex exactly once

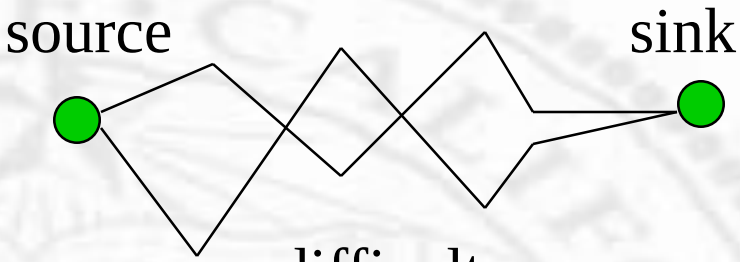
- Multi-stage graph

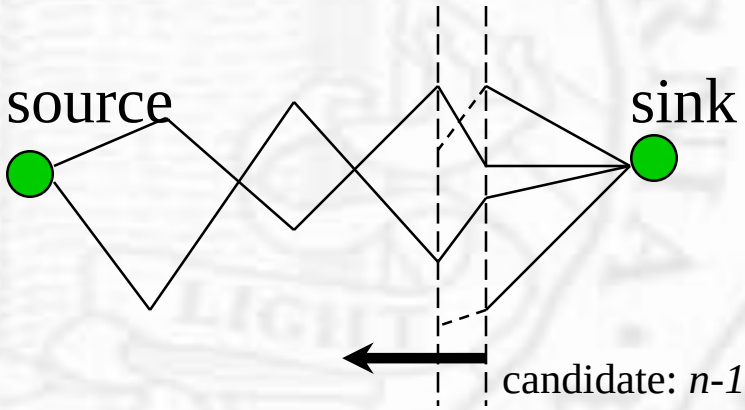




$$Cost(i,j) = \min_{\substack{l \in V_{i+1} \\ <j,l> \in E}} \{c(j,l) + Cost(i+1,l)\}$$

- Traveling salesman



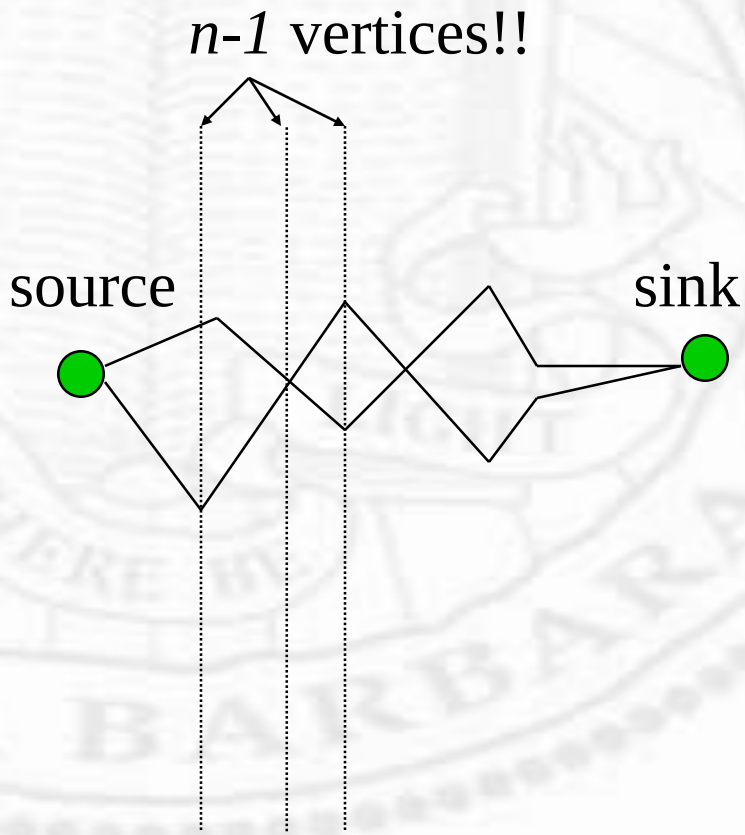


candidate: *n-1* vertices

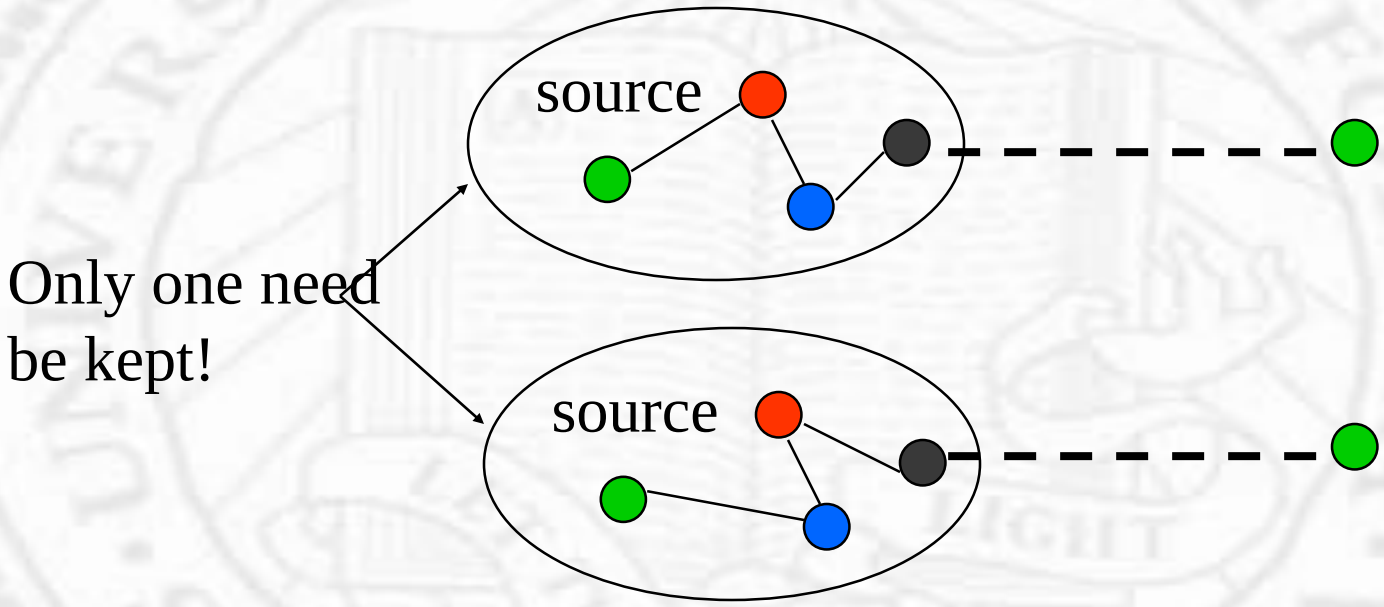
candidate: *n-2* vertices

$$g(i,S) = \min_{j \in S} \{c(i,j) + g(j, S - \{j\})\}$$

- Difference
  - For travelling salesman *source* = *sink*
  - Every vertex can possibly be at every stage
  - *O((n-1)!)* complexity



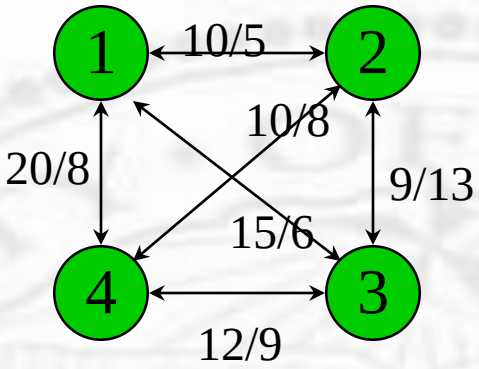
- Does the principle of optimality apply?
- Small problems with reuse?



$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c(1,k) + g(k, V - \{1,k\})\}$$

$$g(i, S) = \min_{j \in S} \{c(i, j) + g(j, S - \{j\})\}$$

- *g(i,S):* the length of the shortest path starting at vertex *i*, going through all vertices in *S*, then back to the source



$g(2,\phi)$	$g(3,\phi)$	$g(4,\phi)$
5	6	8

$g(2,\{3\})$	$g(2,\{4\})$	$g(3,\{2\})$	$g(3,\{4\})$	$g(4,\{2\})$	$g(4,\{3\})$
$15(9+6)$	$18(10+8)$	$18(13+5)$	$20(12+8)$	$13(8+5)$	$15(9+6)$

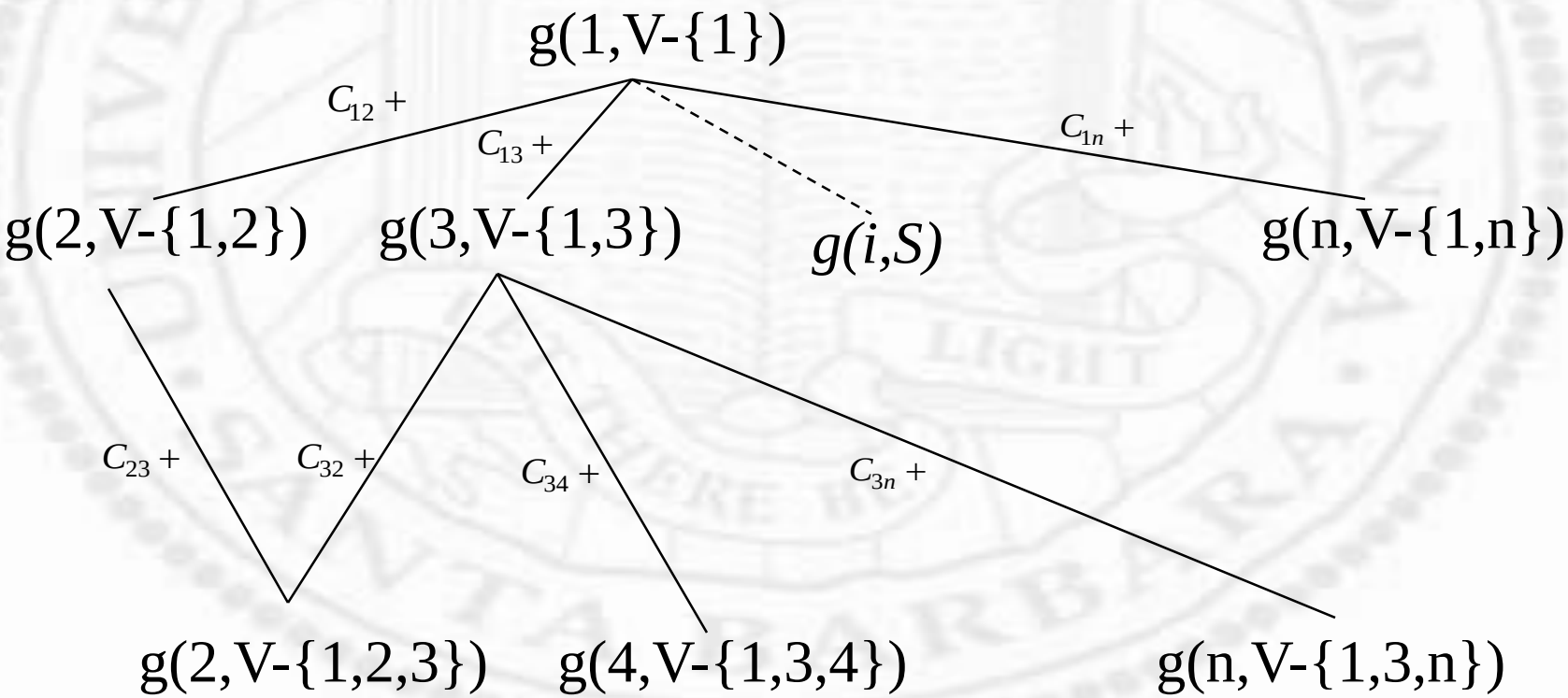
$g(2,\{3,4\})$	$g(3,\{2,4\})$	$g(4,\{2,3\})$
$\min(9+20,10+15)$	$\min(13+18,12+13)$	$\min(8+15,9+18)$
$=25$	$=25$	$=23$

$g(1,\{2,3,4\})$

$= \min(10+25,15+25,20+23) = 35$

- Time Complexity
  - *i*: there are *n-1* vertices to visit at each level
  - *S*: there are $\binom{n-2}{k}$ choices

$$\sum_{k=0}^{n-2}(n-1)\binom{n-2}{k} = O(n^2 2^n)$$



World Series Odds

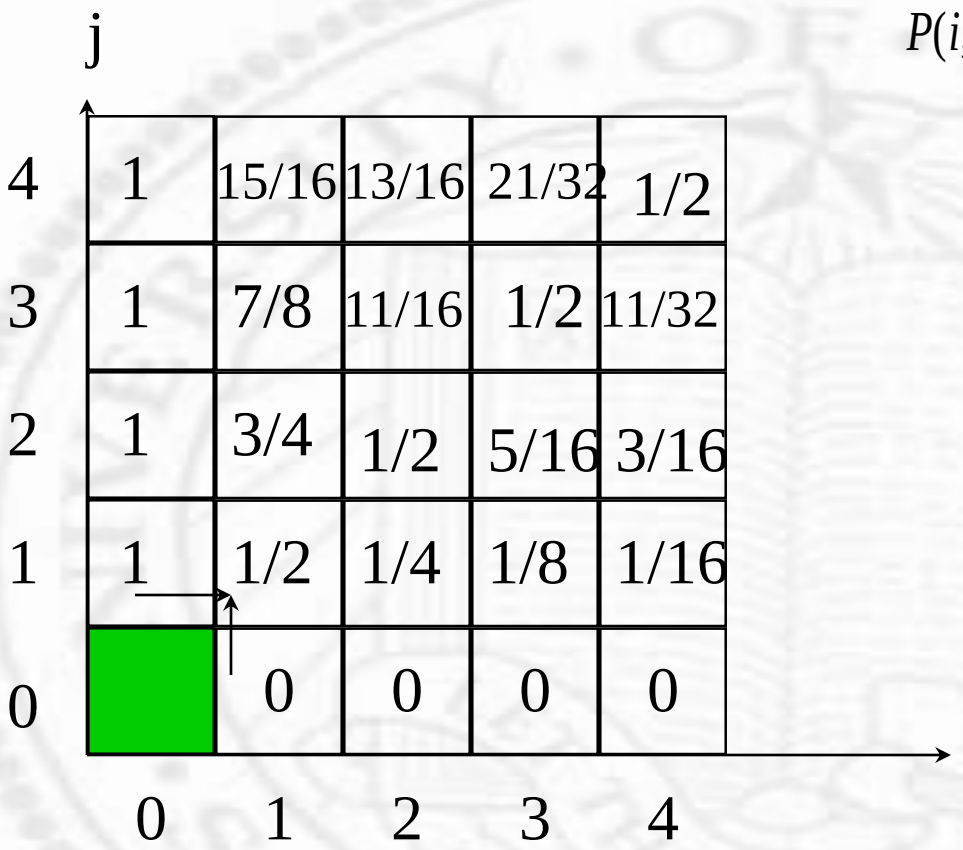
- DP may be used to solve problems where principle of optimality is not applicable
- Input:
  - two teams A and B
  - play a maximum of 7 games
  - whichever team first wins 4 wins the series
- Output:
  - *P(i,j):* conditional probability(A wins the series| A needs *i* more games and B need *j* more games)

- Even though principle of optimality does not apply here, but
  - the problem does possess recursive nature
  - solutions can be constructed by reuse

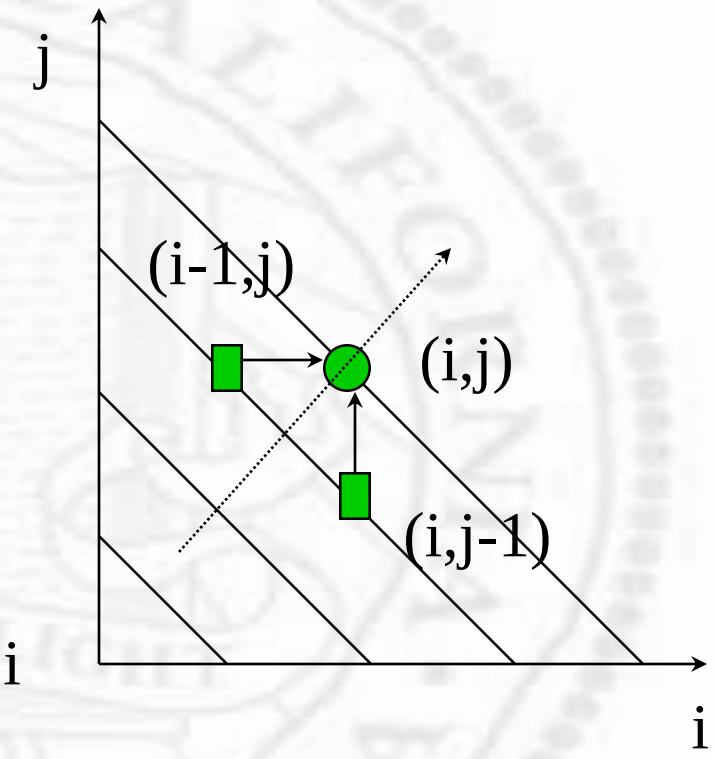
$$P(i,j) = \begin{cases} 1 & i = 0,\ j > 0 \\ 0 & i > 0,\ j = 0 \end{cases}$$

$$P(i,j) = (p(i-1,j) + p(i,j-1))/2, i > 0,\ j > 0$$

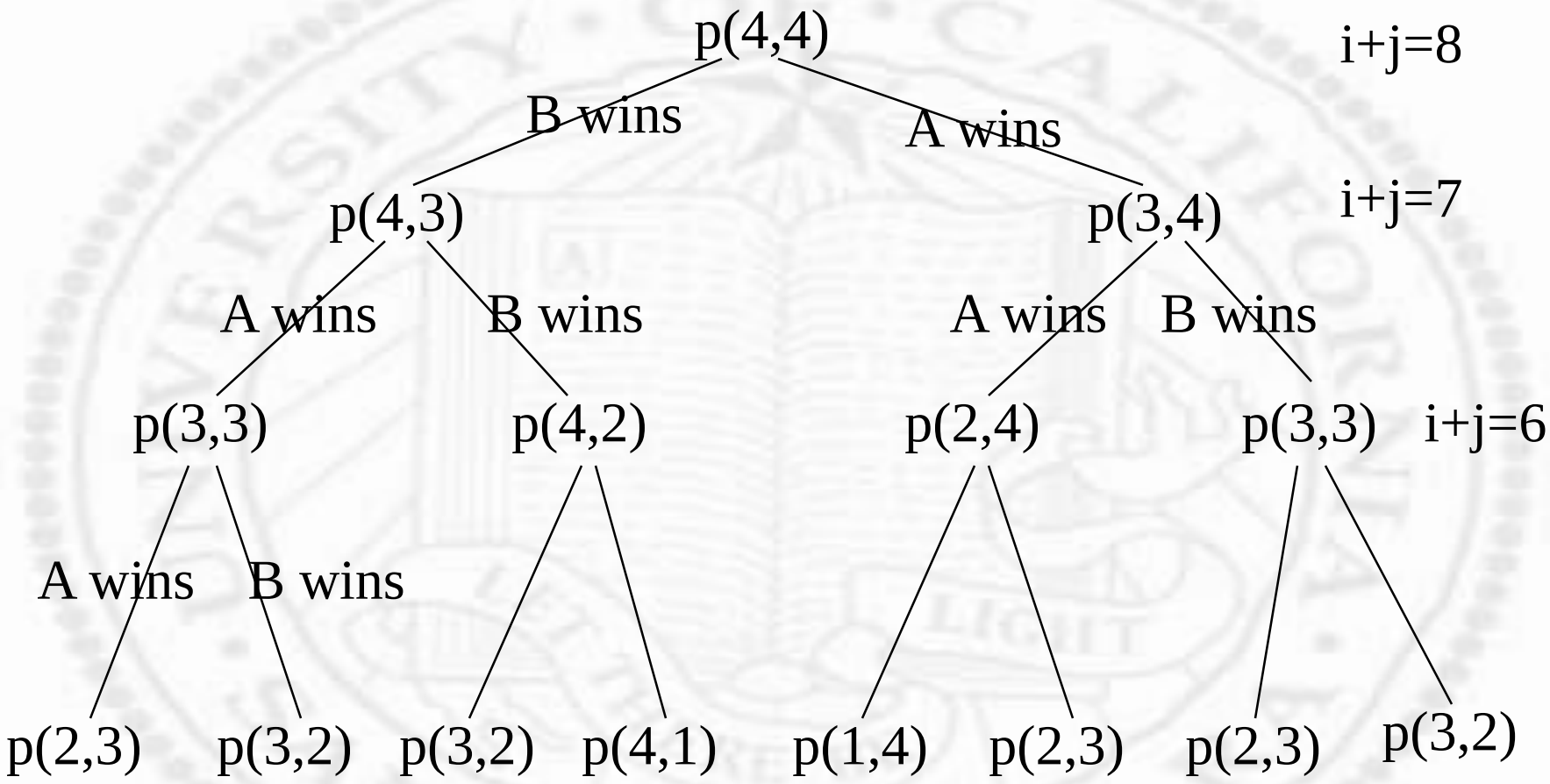
P(i,j): conditional probability(A wins the series| A needs *i* more games and B need *j* more games)



$$P(i,j) = (p(i-1,j) + p(i,j-1))/2, i > 0, j > 0$$



Brute force method



$O(2^n)$ for brute force vs. $O(n^2)$ for DP

Lessons Learned

- Basic principles (Multi-stage graphs, 0/1-knapsack, Reliable design)
  - Brute force
  - Reuse, Feasibility, Optimality
  - Table building (recursion)
- Being Smart (Matrix multiplication, polygon triangulation)
  - There are different tables and different recursions
- Being flexible (World series odds)
  - Reuse regardless of optimality constraint (more later)
- Nothing really matter much (Traveling Salesperson)
  - There are hard problems in the universe