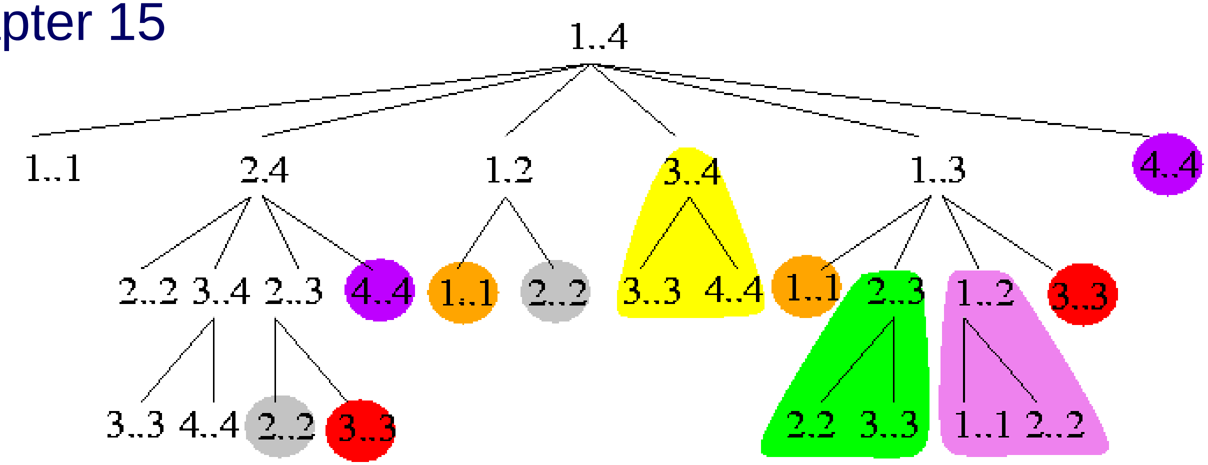
Unit 4: Dynamic Programming

- Course contents:
  - Assembly-line scheduling
  - Matrix-chain multiplication
  - Longest common subsequence
  - Optimal binary search trees
  - Maximum planar subset of chords
- Readings:
  - Chapter 15



Divide-and-Conquer

- **The divide-and-conquer paradigm**
  - **Divide** the problem into a number of subproblems.
  - **Conquer** the subproblems (solve them).
  - **Combine** the subproblem solutions to get the solution to the original problem.

- **Complexity:** determined by solving recurrence relations

Dynamic Programming (DP)

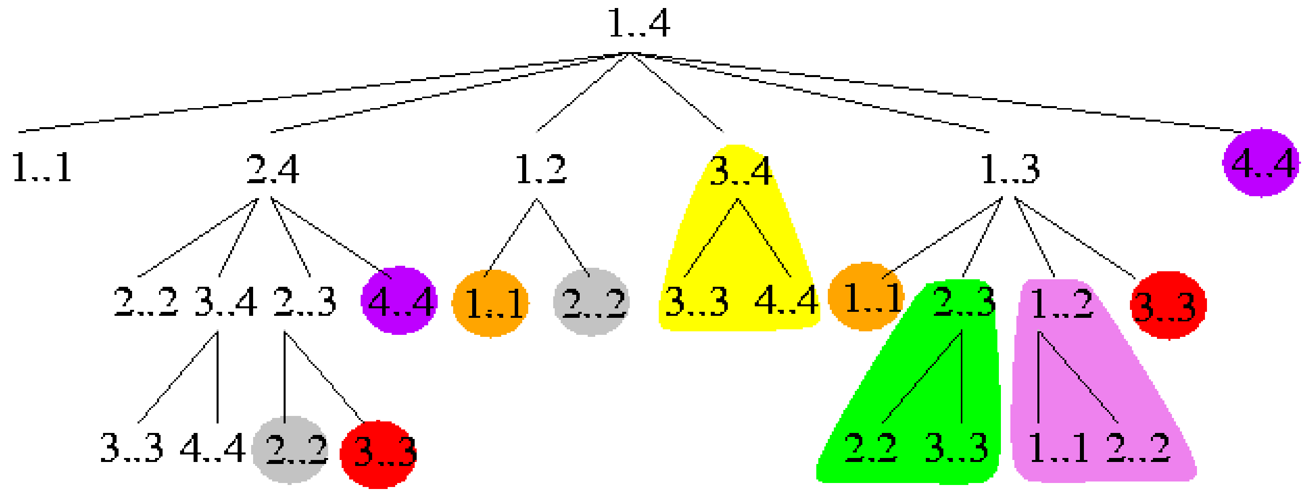
- “Programming” in DP refers to a tabular method, not to writing computer code.

- **Basic idea:** One **implicitly** explores the space of all possible solutions by
  - Carefully decomposing things into a series of subproblems
  - Building up correct solutions to larger and larger subproblems

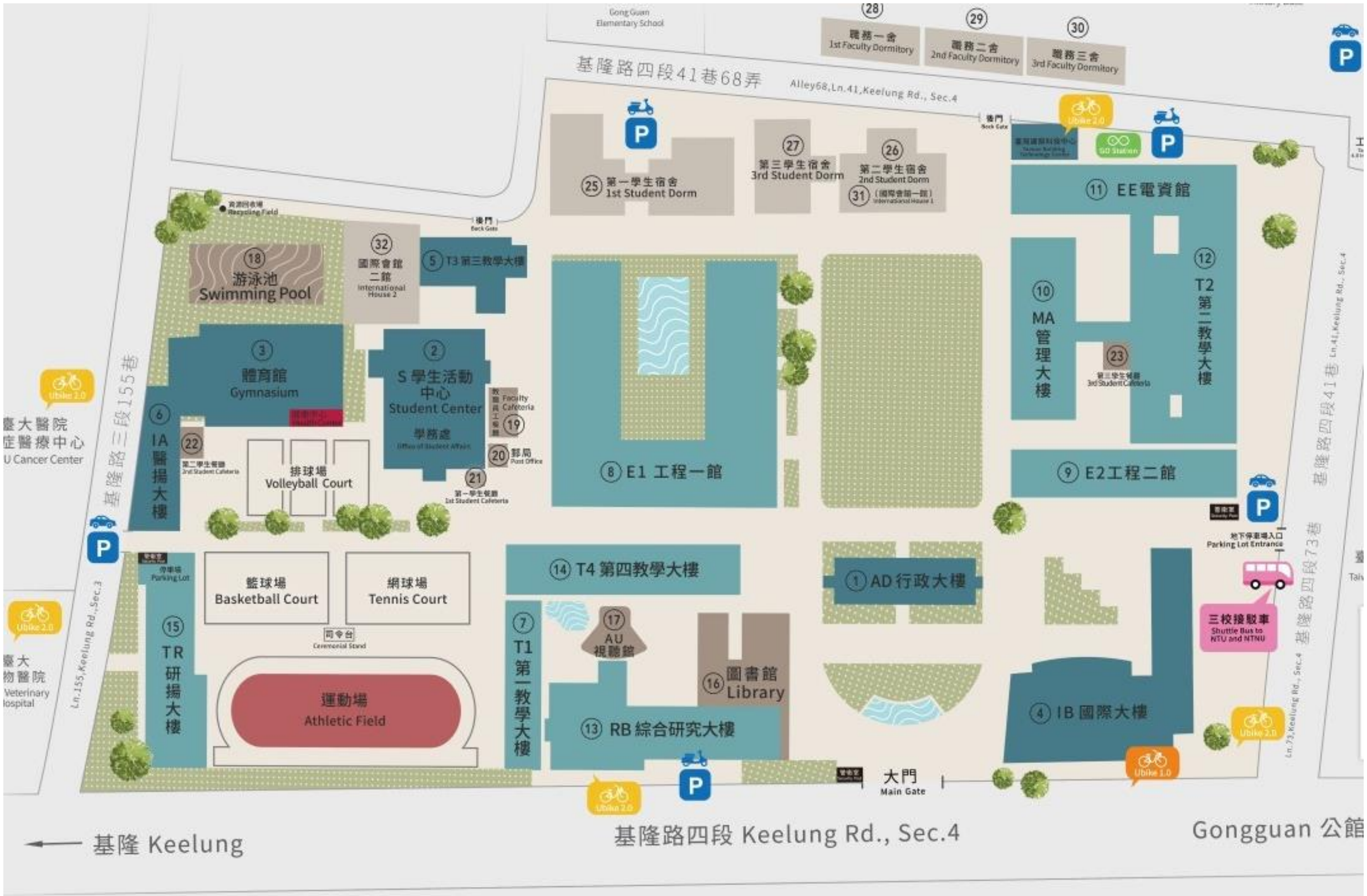
- Can you smell the D&C flavor? However, DP is another story!
  - DP does not exam all possible solutions **explicitly**

Dynamic Programming (DP) vs. Divide-and-Conquer

- Both solve problems by combining the solutions to subproblems.
- Divide-and-conquer algorithms
  - Partition a problem into **independent** subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
  - Inefficient if they solve the same subproblem more than once.
- Dynamic programming (DP)
  - Applicable when the subproblems are **not independent.**
  - DP solves each subproblem just once.



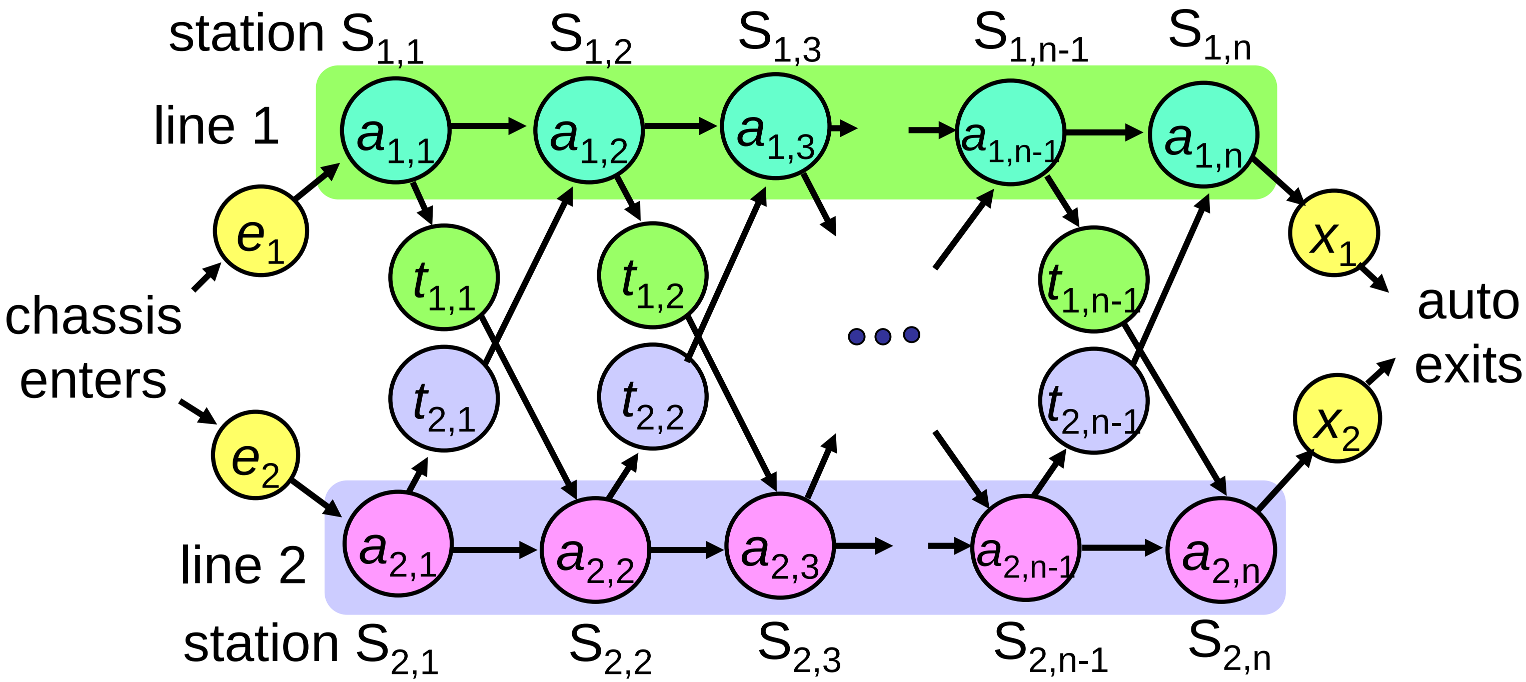
An Example



基隆路四段41巷68弄
Alley68,Ln.41,Keelung Rd., Sec.4
(28) 職務一舍 1st Faculty Dormitory
(29) 職務二舍 2nd Faculty Dormitory
(30) 職務三舍 3rd Faculty Dormitory
(25) 第一學生宿舍 1st Student Dorm
(27) 第三學生宿舍 3rd Student Dorm
(26) 第二學生宿舍 2nd Student Dorm
(31)
(11) EE電資館
(12) T2第二教學大樓
(10) MA管理大樓
(23)
(18) 游泳池 Swimming Pool
(32) 國際會館二館 International House 2
(5) T3第三教學大樓
(3) 體育館 Gymnasium
(2) S學生活動中心 Student Center
學務處
(19)
(20) 郵局 Post Office
(21)
(6) IA醫揚大樓
(22)
排球場 Volleyball Court
(8) E1 工程一館
(9) E2工程二館
臺大醫院
癌醫療中心
U Cancer Center
基隆路三段155巷
Ln.155,Keelung Rd.,Sec.3
籃球場 Basketball Court
網球場 Tennis Court
(14) T4 第四教學大樓
(1) AD 行政大樓
(15) TR研揚大樓
司令台
運動場 Athletic Field
(7) T1第一教學大樓
(17) AU視聽館
(16) 圖書館 Library
(13) RB 綜合研究大樓
(4) IB 國際大樓
三校接駁車
Shuttle Bus to NTU and NTNU
Parking Lot Entrance
大門 Main Gate
臺大
動物醫院
Veterinary
Hospital
基隆路四段41巷
基隆路四段73巷
基隆 Keelung
基隆路四段 Keelung Rd., Sec.4
Gongguan 公館

Assembly-line Scheduling

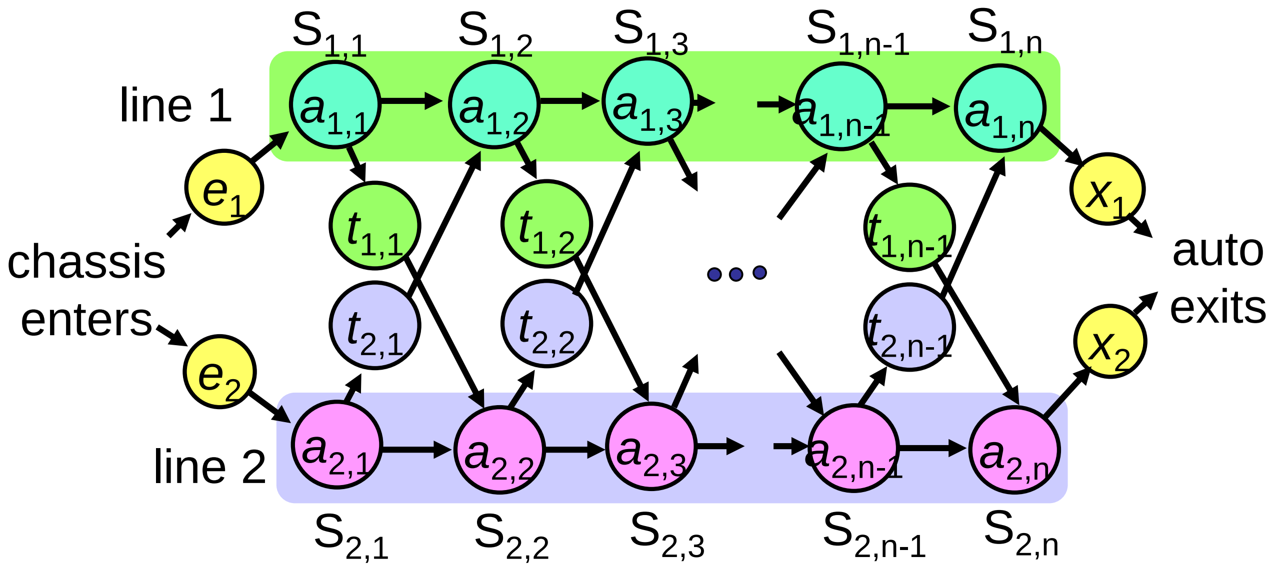
Assembly-line Scheduling



- An auto chassis enters each assembly line, has parts added at stations, and a finished auto exits at the end of the line.
  - $S_{i,j}$: the *j*th station on line *i*
  - $a_{i,j}$: the assembly time required at station $S_{i,j}$
  - $t_{i,j}$: transfer time from station $S_{i,j}$ to the *j+1* station of the other line.
  - e_i (x_i): time to enter (exit) line *i*

Optimal Substructure

- Objective: Determine the stations to choose to minimize the total manufacturing time for one auto.
  - Brute force: $\Omega(2^n)$, why?
  - The problem is linearly ordered and cannot be rearranged => Dynamic programming?
- **Optimal substructure:** If the fastest way through station $S_{i,j}$ is through $S_{1,j-1}$, then the chassis must have taken a fastest way from the starting point through $S_{1,j-1}$.



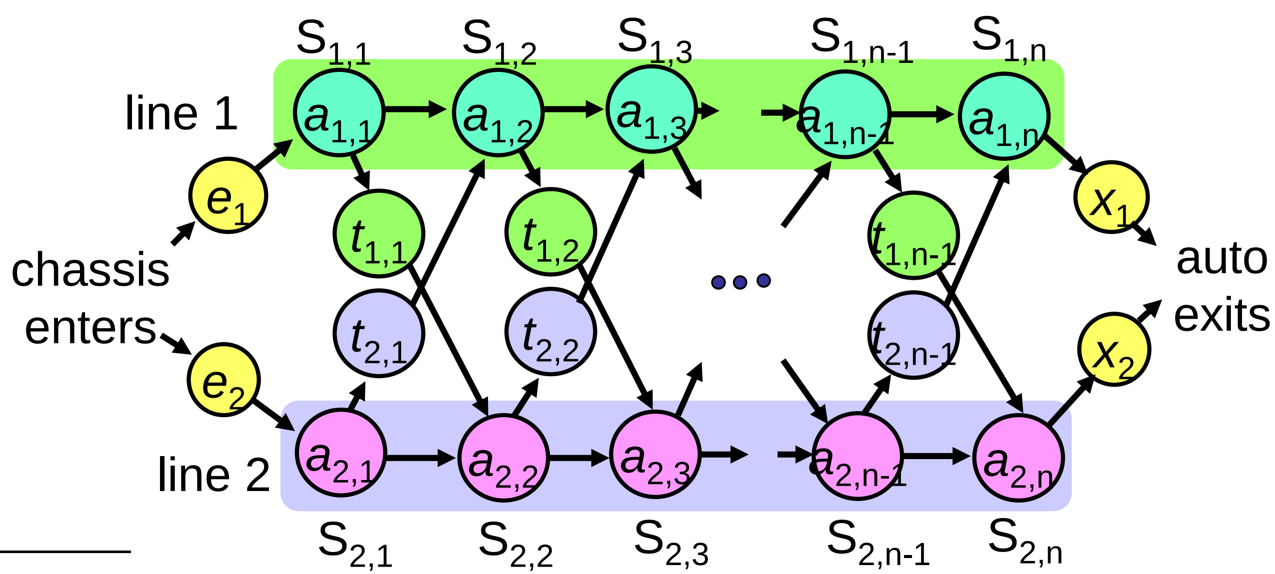
Overlapping Subproblem: Recurrence

- **Overlapping subproblem:** The fastest way through station $S_{1,j}$ is either through $S_{1,j-1}$ and then $S_{1,j}$, or through $S_{2,j-1}$ and then transfer to line 1 and through $S_{1,j}$.
- $f_i[j]$: fastest time from the starting point through $S_{i,j}$

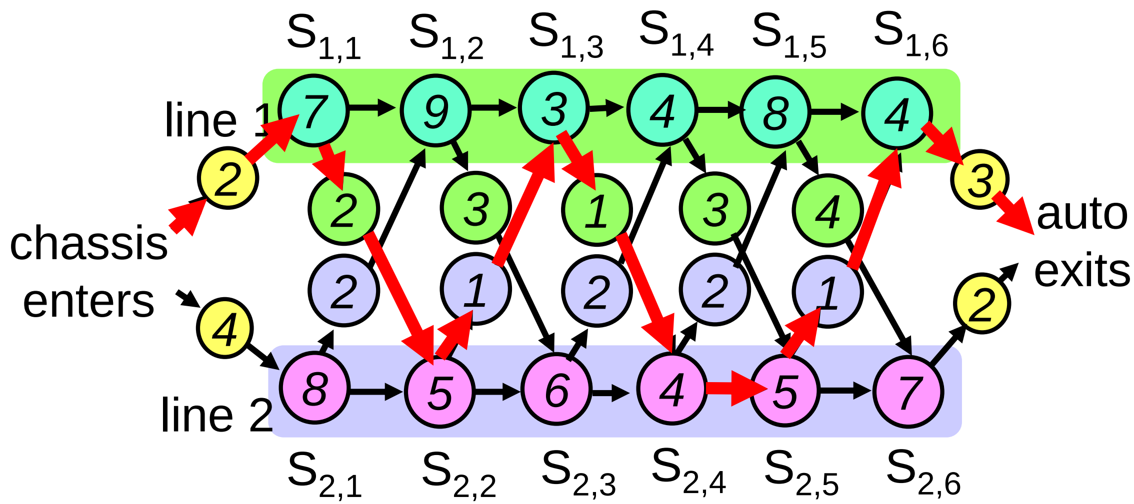
$$f_1[j] = \begin{cases} e_1 + a_{1,1} & if\ j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & if\ j \geq 2 \end{cases}$$

- The fastest time all the way through the factory

$$\boldsymbol{f^* = \min(f_1[n] + x_1, f_2[n] + x_2)}$$



An Example



$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \textit{if } j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \textit{if } j \geq 2 \end{cases}$$

$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37
j	1	2	3	4	5	6

$f^* = 38$

$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

$$f_1[1] = 2 + 7 = 9$$
$$f_2[1] = 4 + 8 = 12$$

$$f_1[2] = \min(9 + 9, 12 + 2 + 9) = 18$$
$$f_2[2] = \min(12 + 5, 9 + 2 + 5) = 16$$

$$f_1[3] = \min(18 + 3, 16 + 1 + 3) = 20$$
$$f_2[3] = \min(16 + 6, 18 + 3 + 6) = 22$$

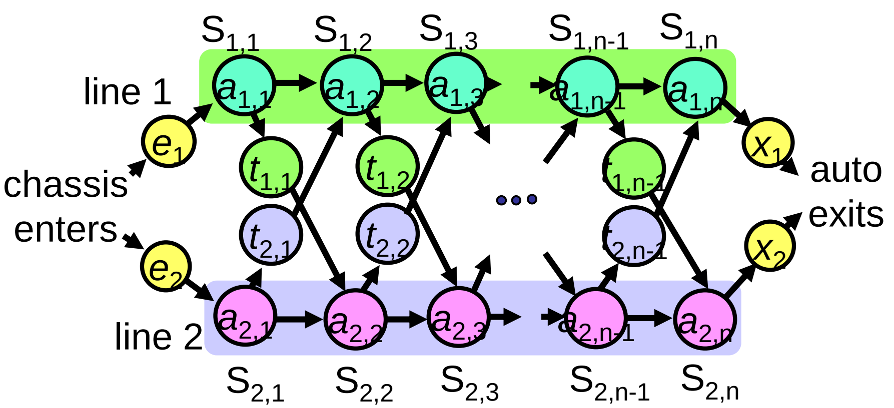
Computing the Fastest Time

Fastest-Way(a, t, e, x, n)

$f_1[1] = e_1 + a_{1,1}$
$f_2[1] = e_2 + a_{2,1}$
for $j = 2$ **to** n
    if $f_1[j\text{-}1] + a_{1,j} \le f_2[j\text{-}1] + t_{2,j-1} + a_{1,j}$
        $f_1[j] = f_1[j\text{-}1] + a_{1,j}$
        $l_1[j] = 1$
    else $f_1[j] = f_2[j\text{-}1] + t_{2,j-1} + a_{1,j}$
        $l_1[j] = 2$
    if $f_2[j\text{-}1] + a_{2,j} \le f_1[j\text{-}1] + t_{1,j-1} + a_{2,j}$
        $f_2[j] = f_2[j\text{-}1] + a_{2,j}$
        $l_2[j] = 2$
    else $f_2[j] = f_1[j\text{-}1] + t_{1,j-1} + a_{2,j}$
        $l_2[j] = 1$
if $f_1[n] + x_1 \le f_2[n] + x_2$
    $f^* = f_1[n] + x_1$
    $l^* = 1$
else $f^* = f_2[n] + x_2$
    $l^* = 2$

Time complexity: $\Theta(n)$

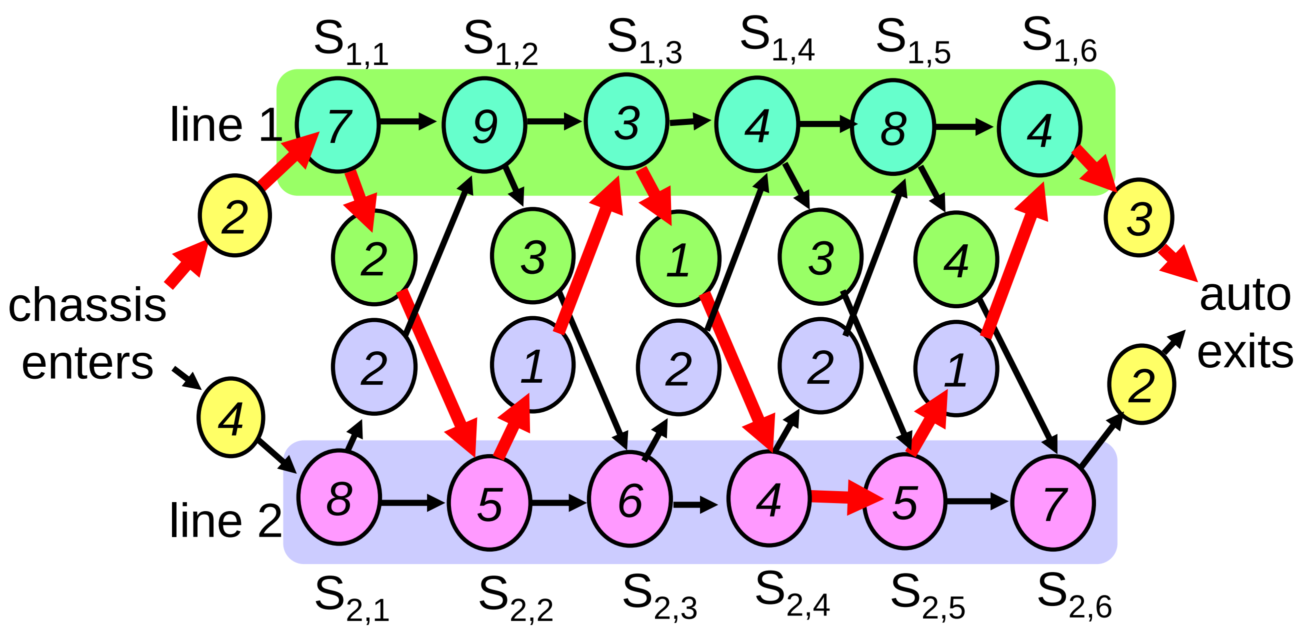
- $S_{i,j}$: the jth station on line i
- $a_{i,j}$: the assembly time required at $S_{i,j}$
- $t_{i,j}$: transfer time from station Si,j to the $j+1$th station of the other line
- $e_i(x_i)$: time to enter (exit) line i
- $l_i[j]$: The line number whose station ***j-1*** is used in a fastest way through $S_{i,j}$



Constructing the Fastest Way

Print-Station(l, n)
1. $i = l^*$
2. Print "line" i ", station " n
3. **for** j = n **downto** 2
4. $i = l_i[j]$
5. *Print "line " i ", station " j-1*

line 1, station 6
line 2, station 5
line 2, station 4
line 1, station 3
line 2, station 2
line 1, station 1



Dynamic Programming (DP)

- Typically applied to **optimization problem**.
- Generic approach
  - Calculate the solutions to all subproblems.
  - Proceed computation from the small subproblems to larger subproblems.
  - Compute a subproblem based on previously computed results for smaller subproblems.
  - Store the solution to a subproblem in a table and never recompute.
- Development of a 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 bottom-up.
  4. Construct an optimal solution from computed information (omitted if only the optimal value is required).

When to Use Dynamic Programming (DP)

- DP computes recurrence efficiently by storing partial results $\Rightarrow$ efficient only when the number of partial results is small.
- **Hopeless configurations:** n! permutations of an n-element set, 2^n subsets of an n-element set, etc.
- **Promising configurations:** $\sum_{i=1}^{n} i = n(n+1)/2$ contiguous substrings of an n-character string, $n(n+1)/2$ possible subtrees of a binary search tree, etc.
- ***DP works best on objects that are linearly ordered and cannot be rearranged!!***
  - Linear assembly lines, matrices in a chain, characters in a string, points around the boundary of a polygon, points on a line/circle, the left-to-right order of leaves in a search tree, etc.
  - Objects are ordered left-to-right $\Rightarrow$ Smell DP?

Keys to Dynamic Programming

- **Smart recursion**: dynamic programming is recursion without repetition.
  - Dynamic programming is **NOT** about filling in tables; it's about smart recursion.
  - Dynamic programming algorithms store the solutions of intermediate subproblems often but not always in some kind of array or table.
  - A common mistake: focusing on the table (because tables are easy and familiar) instead of the much more important (and difficult) task of finding a correct recurrence.
- If the recurrence is wrong, or if we try to build up answers in the wrong order, the algorithm will NOT work!

Summary: Algorithmic Paradigms

- **Brute-force (Exhaustive):** Examine the entire set of possible solutions explicitly
  - A victim to show the efficiencies of the following methods
- **Greedy:** Build up a solution incrementally, myopically optimizing some local criterion.
  - Optimization problems that can be solved correctly by a greedy algorithm are very rare.
- **Divide-and-conquer:** Break up a problem into two subproblems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- **Dynamic programming:** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Matrix-Chain Multiplication

Matrix-Chain Multiplication

- If A is a p **x** q matrix and B a q **x** r matrix, then $C = AB$ is a p **x** r matrix

$$C[i,j] = \sum_{k=1}^{q} A[i,k]B[k,j]$$

Dot Product

time complexity: $O(pqr)$.

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix} \times \begin{bmatrix}7 & 8\\ 9 & 10\\ 11 & 12\end{bmatrix} = \begin{bmatrix}58 & ?\\ ? & ?\end{bmatrix}$$

```
Matrix-Multiply(A, B)
if A.columns ≠ B.rows
    error "incompatible dimensions"
else let C be a new A.rows * B.columns matrix
    for i = 1 to A.rows
        for j = 1  to B.columns
            c_ij = 0
            for k = 1 to A.columns
                c_ij = c_ij + a_ik b_kj
    return C
```

Matrix-Chain Multiplication (cont'd)

- **The matrix-chain multiplication problem**
  - Input: Given a chain <A_1, A_2, …, A_n> of n matrices, matrix $\boldsymbol{A_i}$ has dimension $\boldsymbol{p_{i-1}}$ **x** $\boldsymbol{p_i}$
  - Objective: Parenthesize the product $A_1 A_2 \ldots A_n$ to minimize the number of scalar multiplications

- **Exp:** dimensions: A_1: 4 **x** 2; A_2: 2 **x** 5; A_3: 5 **x** 1

  $(A_1A_2)A_3$: total multiplications = 4 **x** 2 **x** 5 + 4 **x** 5 **x** 1 = 60

  $A_1(A_2A_3)$: total multiplications = 2 **x** 5 **x** 1 + 4 **x** 2 **x** 1 = 18

- So the order of multiplications can make a big difference!

Matrix-Chain Multiplication: Brute Force

- $A = A_1 A_2 \ldots A_n$: How to evaluate A using the minimum number of multiplications?
- Brute force: check all possible orders?
  - $P(n)$: number of ways to multiply n matrices.

$$P(n) = \begin{cases} 1 & if\ n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & if\ n \geq 2 \end{cases}$$

  - $P(n) = \Omega\left(\frac{4^n}{n^{3/2}}\right)$, **exponential** in n.
- Any efficient solution?
  - The matrix chain **is linearly ordered and cannot be rearranged!!**
  - Smell Dynamic programming?

Using DP for Matrix-Chain Multiplication

- Applicability of dynamic programming
  - **Optimal substructure:** an optimal solution contains within its optimal solutions to subproblems.
  - **Overlapping subproblem:** a recursive algorithm revisits the same subproblems over and over again; only $\theta(n^2)$ subproblems.
  - Given a chain $<A_1, A_2, \ldots, A_n>$ of n matrices
    - # of single matrix: n
    - # of two consecutive matrices: n-1
    - # of three consecutive matrices: n-2

    …
    - # of n consecutive matrices: 1

Smart Recursion

- $m[i, j]$: minimum number of multiplications to compute matrix $A_{i..j} = A_i A_{i+1} \ldots A_j$, $1 \le i \le j \le n$.
  - $m[1, n]$: the cheapest cost to compute $A_{1..n}$.
  - $m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$
  - matrix A_i has dimension $p_{i-1} \times p_i$

$k = i$

$(A_i)(A_{i+1} A_{i+2} \ldots A_{j-2} A_{j-1} A_j)$

$\downarrow$

$k = i+1$

$(A_i A_{i+1})(A_{i+2} \ldots A_{j-2} A_{j-1} A_j)$

$\downarrow$

$k = i+2$

$(A_i A_{i+1} A_{i+2})(\ldots A_{j-2} A_{j-1} A_j)$

$k = j-2$

$(A_i A_{i+1} A_{i+2} \ldots A_{j-2})(A_{j-1} A_j)$

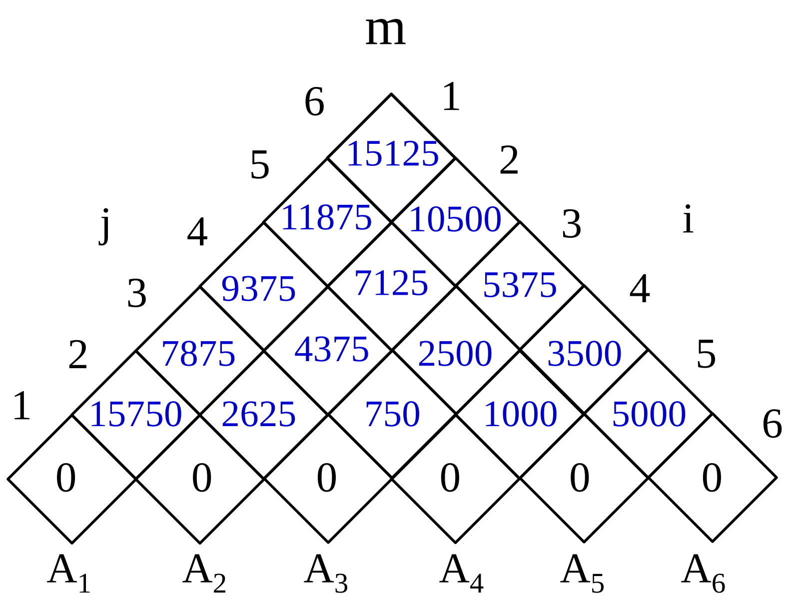
$\downarrow$

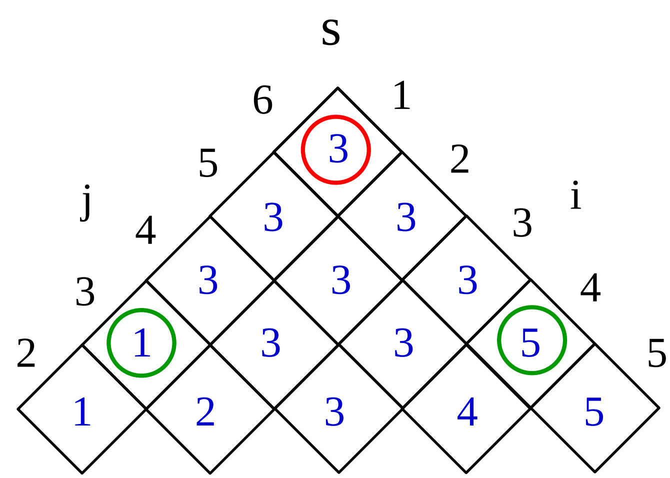
$k = j-1$

$(A_i A_{i+1} A_{i+2} \ldots A_{j-2} A_{j-1})(A_j)$

An Example

matrix	dimension
A_1	30 * 35
A_2	35 * 15
A_3	15 * 5
A_4	5 * 10
A_5	10 * 20
A_6	20 * 25





$((A_1)(A_2\ \ A_3))((A_4\ \ A_5)(A_6))$

$$m[2,4] = \min \begin{cases} m[2,2] + m[3,4] + p1p2p4 = 0 + 750 + 35 \times 15 \times 10 = 6000 \\ m[2,3] + m[4,4] + p1p3p4 = 2625 + 0 + 35 \times 5 \times 10 = \mathbf{4375} \end{cases}$$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p1p2p5 = 0 + 2500 + 35 \times 15 \times 20 = 13000 \\ m[2,3] + m[4,5] + p1p3p5 = 2625 + 1000 + 35 \times 5 \times 20 = \mathbf{7125} \\ m[2,4] + m[5,5] + p1p4p5 = 4375 + 0 + 35 \times 10 \times 20 = 11375 \end{cases}$$

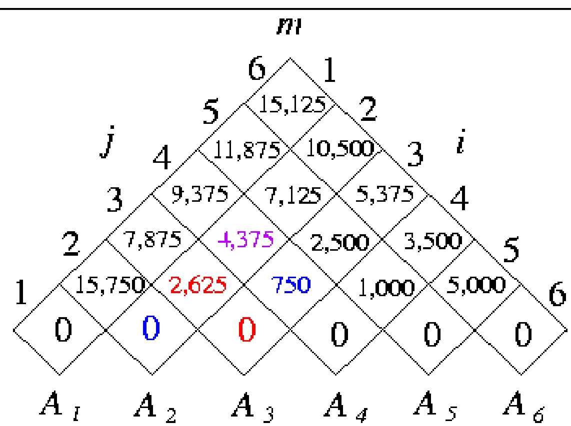
Bottom-Up DP Matrix-Chain Order

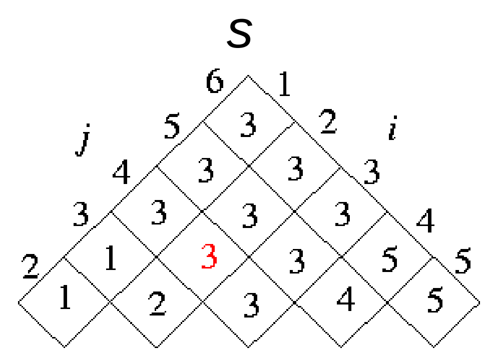
```
Matrix-Chain-Order(p) // p = <p_0, p_1, .., p_n>
n = p.length – 1
Let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
      m[i, i] = 0
for  l = 2 to n          // l is the chain length
      for i = 1 to n – l +1
            j = i + l -1
            m[i, j] = ∞
            for k = i  to j -1
                  q = m[i, k] + m[k+1, j] + p_{i-1}p_k p_j
                  if q < m[i, j]
                        m[i, j] = q
                        s[i, j] = k
return m and s
```

A_i dimension

$p_{i-1} \times p_i$

matrix	dimension
A_1	30 * 35
A_2	35 * 15
A_3	15 * 5
A_4	5 * 10
A_5	10 * 20
A_6	20 * 25





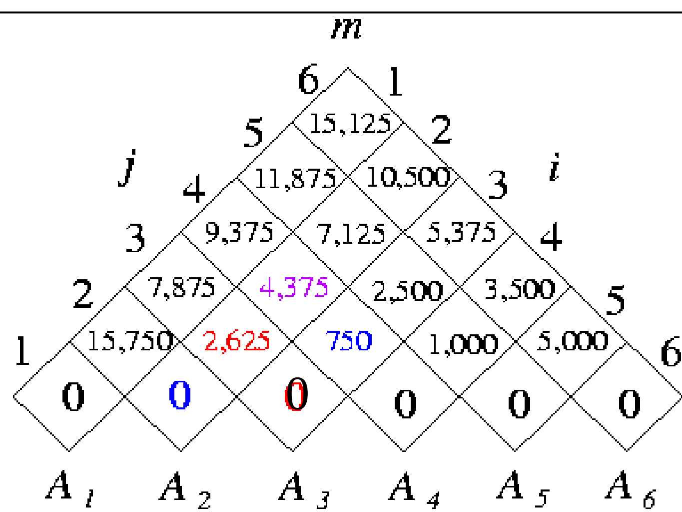
$$m[2,4] = \min \begin{cases} m[2,2] + m[3,4] + p_1 p_2 p_4 = 0 + 750 + 35 \times 15 \times 10 = 6000. \\ m[2,3] + m[4,4] + p_1 p_3 p_4 = 2625 + 0 + 35 \times 5 \times 10 = 4375. \end{cases}$$

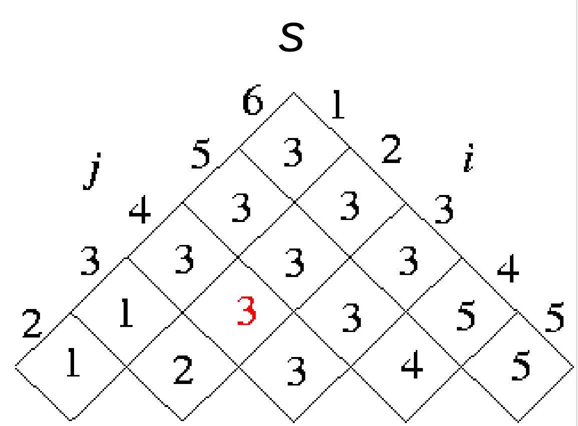
Constructing an Optimal Solution

- $s[i, j]$: value of k such that the optimal parenthesization of $A_i A_{i+1} \ldots A_j$ splits between A_k and A_{k+1}
- Optimal matrix $A_{1..n}$ multiplication: $A_{1..s[1, n]} A_{s[1, n] + 1..n}$
- **Exp:** call Print-Optimal-Parens(s, 1, 6): $((A_1 (A_2 A_3))((A_4 A_5) A_6))$

```
Print-Optimal-Parens(s, i, j)
1. if i == j
2.        print "Ai"
3. else print "("
4.        Print-Optimal-Parens(s, i, s[i, j])
5.        Print-Optimal-Parens(s, s[i, j] + 1, j)
6.        print ")"
```

matrix	dimension
A_1	30 * 35
A_2	35 * 15
A_3	15 * 5
A_4	5 * 10
A_5	10 * 20
A_6	20 * 25

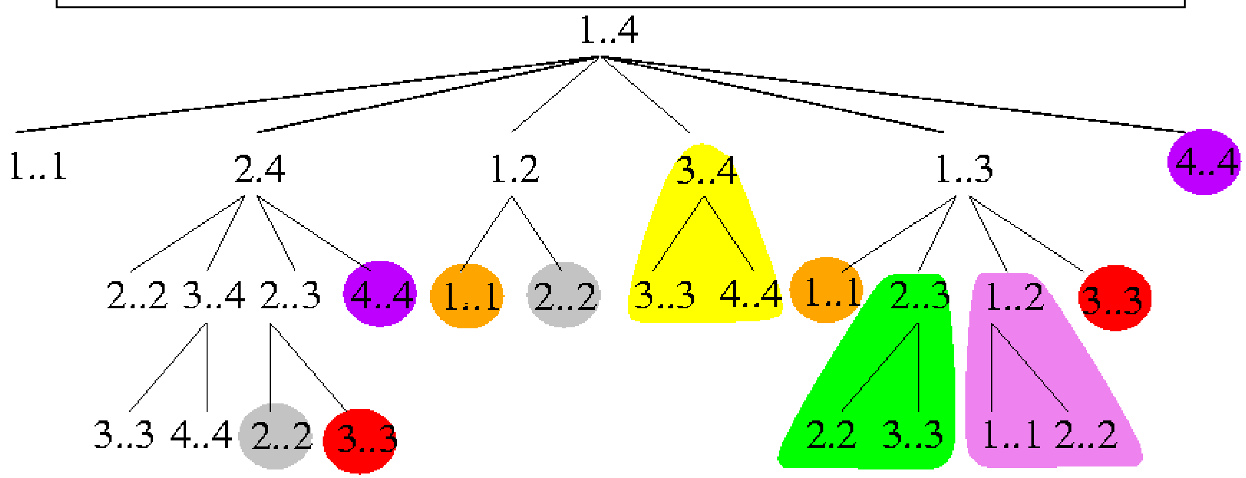




Top-Down, Recursive Matrix-Chain Order

- Time complexity: $\Omega(2^n)$ $(\sum_{k=1}^{n-1}(T(k) + T(n-k) + 1))$.

```
Recursive-Matrix-Chain(p, i, j)
1. if i == j
2.       return 0
3. m[i, j] = ∞
4. for k = i to j -1
5        q = Recursive-Matrix-Chain(p, i, k)
             + Recursive-Matrix-Chain(p, k+1, j) + p_{i-1}p_k p_j
6.       if q < m[i, j]
7.             m[i, j] = q
8. return m[i, j]
```



Top-Down DP Matrix-Chain Order (Memorization)

- Complexity: $O(n^2)$ space for $m[]$ matrix and $O(n^3)$ time to fill in $O(n^2)$ entries (each takes $O(n)$ time)

```
Memoized-Matrix-Chain(p) // p = <p_0, p_1, .., p_n>
1. n = p.length - 1
2. let m[1..n, 1..n] be a new table
3. for i = 1 to n
4.       for j = i to n
5.             m[i, j] = ∞
6. return Lookup-Chain(m, p,1,n)
```

```
Lookup-Chain(m, p, i, j)
1. if m[i, j] < ∞
2.     return m[i, j]
3. if i == j
4.      m[ i, j] = 0
5. else for k = i to j -1
6.             q = Lookup-Chain(m, p, i, k) + Lookup-Chain(m, p, k+1, j) + p_{i-1}p_kp_j
7.             if q < m[i, j]
8.                   m[i, j] = q
9. return m[i, j]
```

Two Approaches to DP

1. Bottom-up iterative approach
   - Start with recursive divide-and-conquer algorithm.
   - Find the dependencies between the subproblems (whose solutions are needed for computing a subproblem).
   - Solve the subproblems in the correct order.
2. Top-down recursive approach (**memorization**)
   - Start with recursive divide-and-conquer algorithm.
   - Keep top-down approach of original algorithms.
   - Save solutions to subproblems in a table (possibly a lot of storage).
   - Recurse only on a subproblem if the solution is not already available in the table.

- If all subproblems must be solved at least once, bottom-up DP is better due to less overhead for recursion and for maintaining tables.
- If many subproblems need not be solved, top-down DP is better since it computes only those required.

Longest Common Subsequence

Longest Common Subsequence

- **Problem:** Given $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, find the **longest common subsequence (LCS)** of X and Y.
- **Exp:** $X = \langle a, b, c, b, d, a, b \rangle$ and $Y = \langle b, d, c, a, b, a \rangle$
  LCS = $\langle b, c, b, a \rangle$ (also, LCS = $\langle b, d, a, b \rangle$).
- **Exp:** DNA sequencing:
  - S1 = ACCGGTCGAGATGCAG;
  - S2 = GTCGTTCGGAATGCAT;
  - LCS S3 = CGTCGGATGCA
- Brute-force method:
  - Enumerate all subsequences of X and check if they appear in Y.
  - Each subsequence of X corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ of the elements of X.
  - There are 2^m subsequences of X. Why?

Optimal Substructure for LCS

- Let $\boldsymbol{X_m} = <x_1, x_2, \ldots, x_m>$ and $\boldsymbol{Y_n} = <y_1, y_2, \ldots, y_n>$ be sequences, and $\boldsymbol{Z_k} = <z_1, z_2, \ldots, z_k>$ be LCS of X_m and Y_n.
  - Case 1: $x_m = y_n$

$$X_m=<\overbrace{a, b, c, d}^{X_{m-1}=<x_1,\, x_{,2\ldots,}\, x_{m-1}>}, a> \qquad Z_k=<\underbrace{\ldots\ldots,}_{Z_{k-1}} a>$$

$$Y_n=<\underbrace{c, b, d,}_{Y_{n-1}} a>$$

  - Case 2: $x_m \neq y_n$
    - z_k may not be x_m
    - z_k may not be y_n

$$X_m=<\overbrace{a, b, c, d}^{X_{m-1}}, a> \qquad\qquad X_m=<\overbrace{a, b, c, d, a}^{X_m}>$$

$$Y_n=<\underbrace{c, b, d, b}_{Y_n}> \qquad\qquad Y_n=<\underbrace{c, b, d,}_{Y_{n-1}} b>$$

Optimal Substructure for LCS (cont'd)

- Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be LCS of X and Y.
  1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
  2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies Z is an LCS of X_{m-1} and Y.
  3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies Z is an LCS of X and Y_{n-1}.

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

- $c[i, j]$: length of the LCS of X_i and Y_j
- $c[m, n]$: length of LCS of X and Y
- Basis: $c[0, j] = 0$ and $c[i, 0] = 0$

Bottom-Up DP for LCS

- Find the right order to solve the subproblems
- To compute $c[i, j]$, we need $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$
- $b[i, j]$: points to the table entry w.r.t. the optimal subproblem solution chosen when computing $c[i, j]$

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

Example of LCS

- LCS time and space complexity: $O(mn)$.
- X = <A, B, C, B, D, A, B> and Y = <B, D, C, A, B, A> $\Rightarrow$ LCS = <B, C, B, A>.

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

Constructing an LCS

- Trace back from $b[m, n]$ to $b[1, 1]$, following the arrows: $O(m+n)$ time.

```
Print-LCS(b, X, i,  j)
1. if i == 0 or j == 0
2.       return
3. if b[i, j] == "↖ "
4.       Print-LCS(b, X, i-1, j-1)
5.       print x_i
6. elseif b[i, j] == " ↑ "
7.       Print-LCS(b, X, i -1, j)
8. else Print-LCS(b, X, i, j-1)
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	↑ 0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Top-Down DP for LCS

- $c[i, j]$: length of the LCS of X_i and Y_j, where $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$.
- $c[m, n]$: LCS of X and Y.
- Basis: $c[0, j] = 0$ and $c[i, 0] = 0$.

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

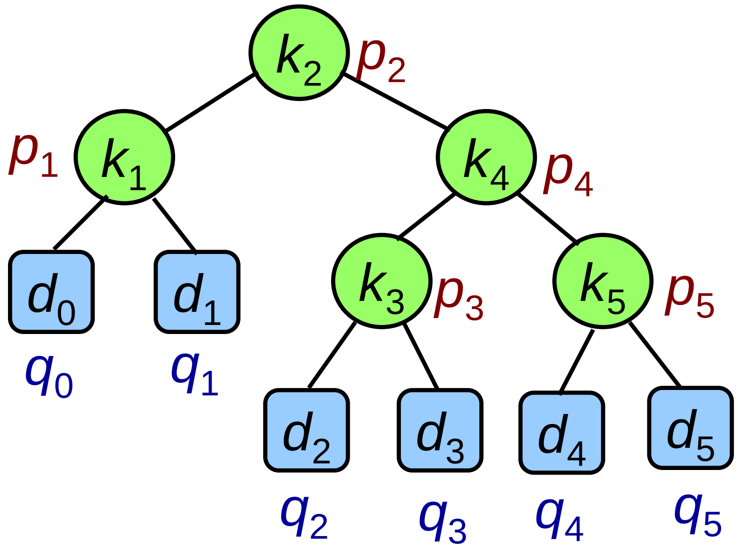
- The top-down dynamic programming: initialize $c[i, 0] = c[0, j] = 0$, $c[i, j]$ = NIL

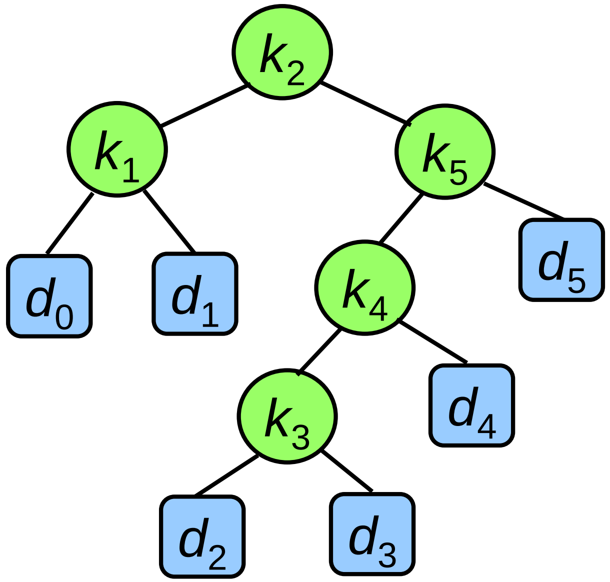
```
TD-LCS(i, j)
1. if c[i,j] == NIL
2.     if x_i == y_j
3.         c[i, j] = TD-LCS(i-1, j-1) + 1
4.     else c[i, j] = max(TD-LCS(i, j-1), TD-LCS(i-1, j))
5. return c[i, j]
```

Optimal Binary Search Trees

Optimal Binary Search Tree

- Given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order ($k_1 < k_2 < \ldots < k_n$) and a set of probabilities $P = \langle p_1, p_2, \ldots, p_n \rangle$ for searching the keys in K and $Q = \langle q_0, q_1, q_2, \ldots, q_n \rangle$ for unsuccessful searches (corresponding to $D = \langle d_0, d_1, d_2, \ldots, d_n \rangle$ of ***n+1*** distinct dummy keys with d_i representing all values between k_i and k_{i+1}), construct a binary search tree whose expected search cost is smallest.

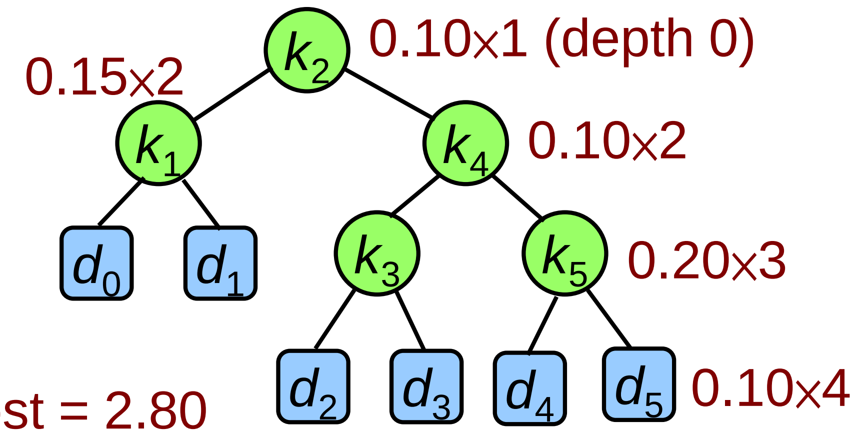




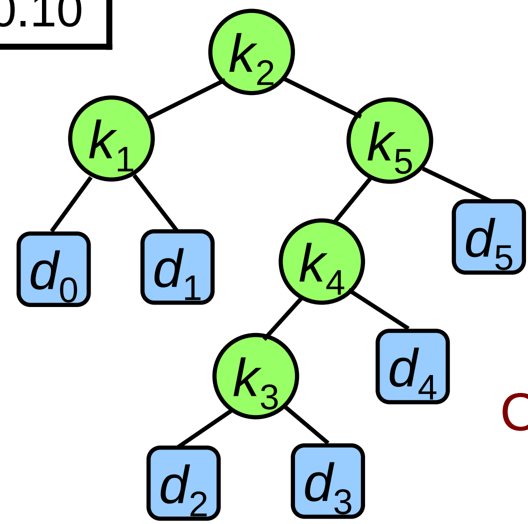
An Example

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$



Cost = 2.80



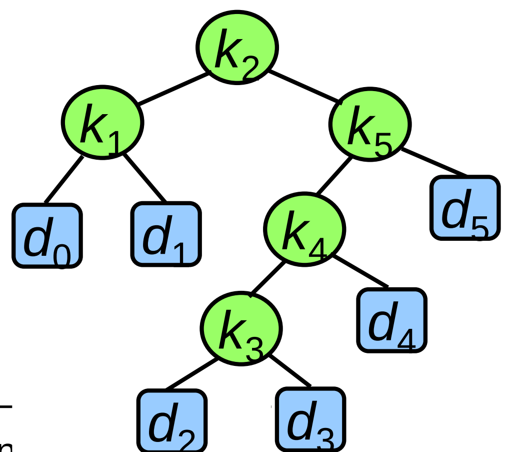
Cost = 2.75
Optimal!!

$$E[search\ cost\ in\ T] = \sum_{i=1}^{n} (depth_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (depth_T(d_i) + 1) \cdot q_i$$

$$= 1 + \sum_{i=1}^{n} depth_T(k_i) \cdot p_i + \sum_{i=0}^{n} depth_T(d_i) \cdot q_i$$

Optimal Substructure

- If an optimal binary search tree *T* has a subtree *T'* containing keys k_i, …, k_j, then this subtree *T'* must be optimal as well for the subproblem with keys k_i, …, k_j and dummy keys d_{i-1}, …, d_j.
  - Given keys k_i, …, k_j with k_r ($i \le r \le j$) as the root, the left subtree contains the keys k_i, …, k_{r-1} (and dummy keys d_{i-1}, …, d_{r-1}) and the right subtree contains the keys k_{r+1}, …, k_j (and dummy keys d_r, …, d_j).
  - For the subtree with keys k_i, …, k_j with root k_i, the left subtree contains keys **k_i, .., k_{i-1} (no key)** with the dummy key d_{i-1}.



Overlapping Subproblem: Recurrence

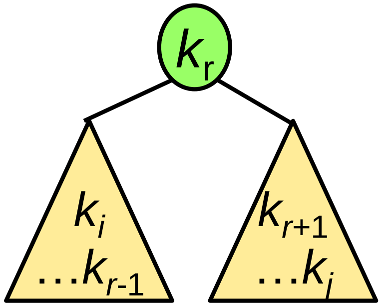
- $e[i, j]$: expected cost of searching an optimal binary search tree containing the keys $k_i, \ldots, k_j$.
  - Want to find $e[1, n]$.
  - $e[i, i-1] = q_{i-1}$ (only the dummy key d_{i-1}).
- If k_r ($i \le r \le j$) is the root of an optimal subtree containing keys $k_i, \ldots, k_j$ and let $w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$, then

$$e[i, j] = \mathbf{p_r} + (e[i, r-1] + \mathbf{w(i, r-1)}) + (e[r+1, j] + \mathbf{w(r+1, j)})$$
$$= e[i, r-1] + e[r+1, j] + \mathbf{w(i, j)}$$

Node depths increase by 1 after merging two subtrees, and so do the costs

- Recurrence:

$$e[i,j] = \begin{cases} q_{i-1} & if\ j = i-1 \\ \min\limits_{i \le r \le j}\{e[i, r-1] + e[r+1, j] + w(i,j)\} & if\ i \le j \end{cases}$$

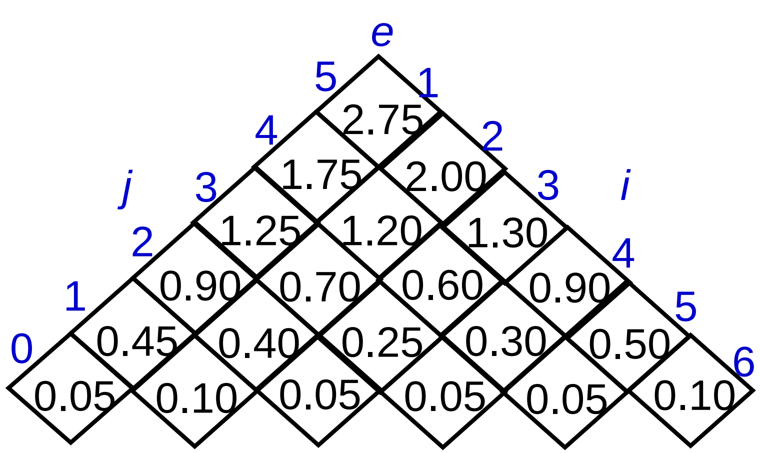


Computing the Optimal Cost

- Need a table *e*[1..*n*+1, 0..*n*] for *e*[*i*, *j*] (why *e*[1, 0] and *e*[*n*+1, *n*]?)
- Apply the recurrence to compute *w*(*i*, *j*) (why?)

$$w[i,j] = \begin{cases} q_{i-1} & if\ j = i-1 \\ w[i,j-1] + p_j + q_j & if\ i \le j \end{cases}$$

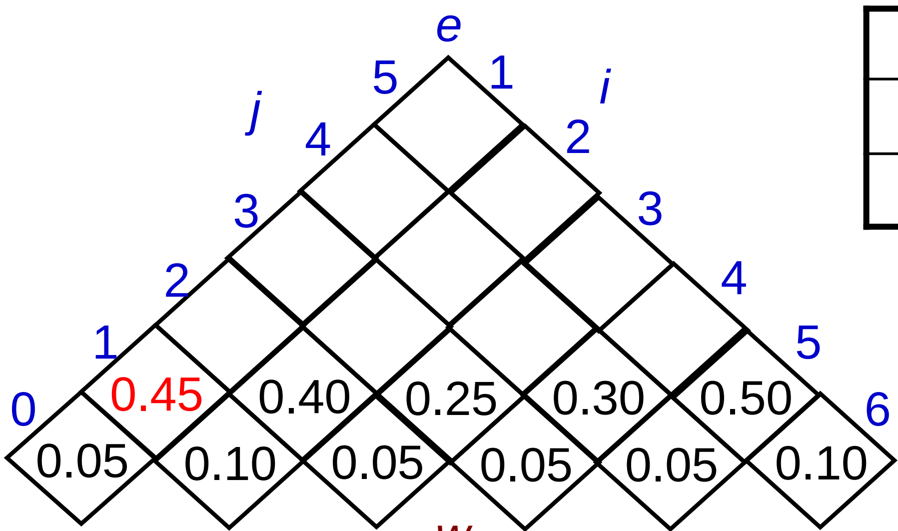
- *root*[*i*, *j*] : index *r* for which k_r is the root of an optimal search tree containing keys k_i, …, k_j.

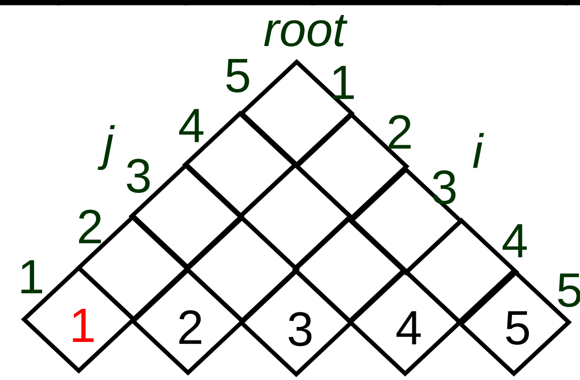


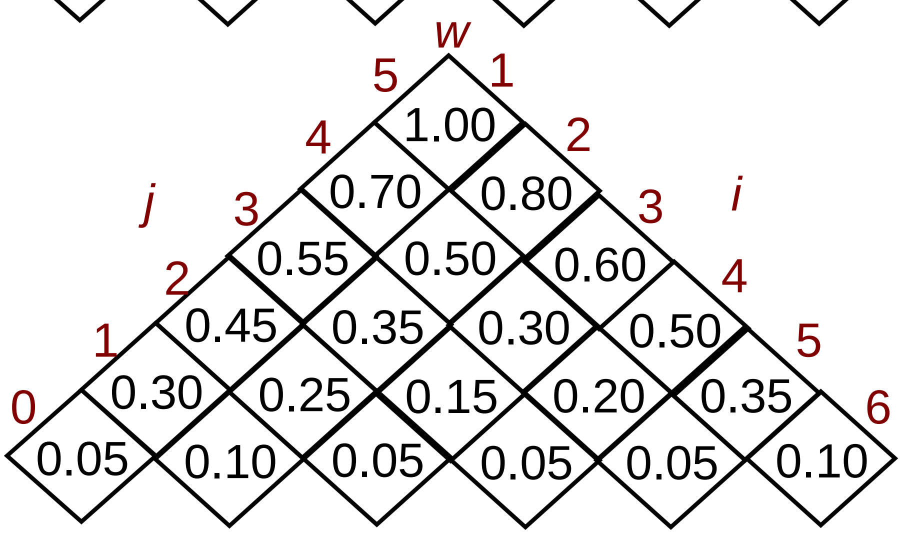

```
Optimal-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n],
   and root[1..n, 1..n] be new tables
for i = 1 to n + 1
    e[i, i-1] = q_{i-1}
    w[i, i-1] = q_{i-1}
for l = 1 to n
    for i = 1 to n – l + 1
        j = i + l -1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + p_j + q_j
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

Example

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

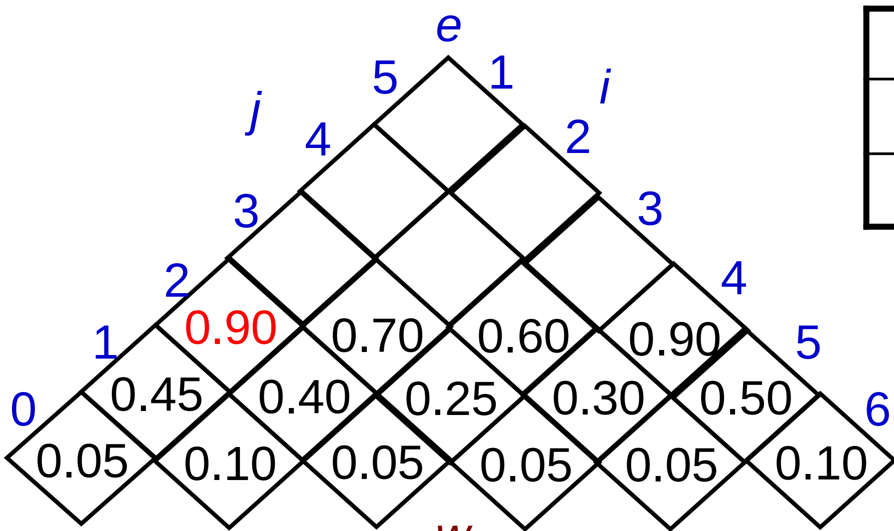




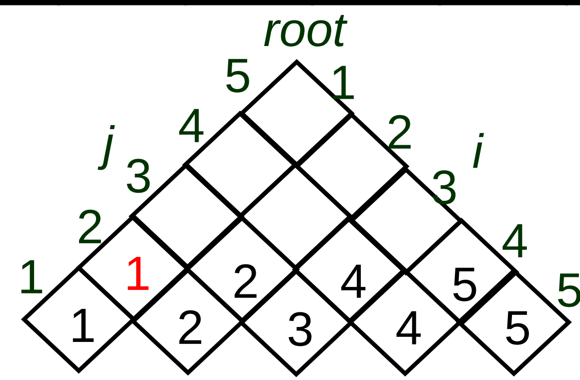


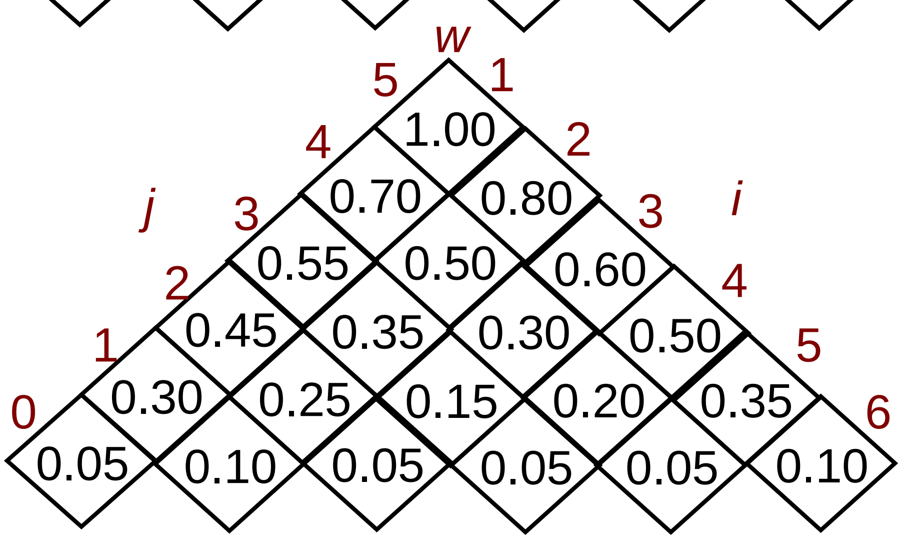
e[1, 1] = e[1, 0] + e[2, 1] + w(1,1)
= 0.05 + 0.10 + 0.3
= 0.45

Example



i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

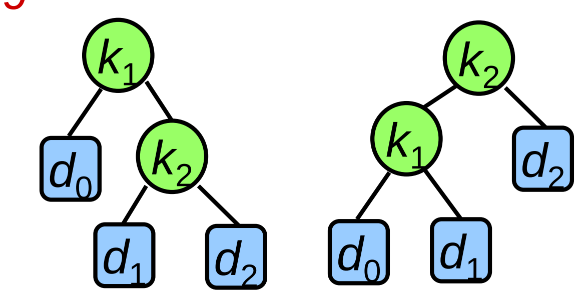




$$e[1,2] = \min\begin{Bmatrix} e[1,0] + e[2,2] + w[1,2] \\ e[1,1] + e[3,2] + w[1,2] \end{Bmatrix}$$

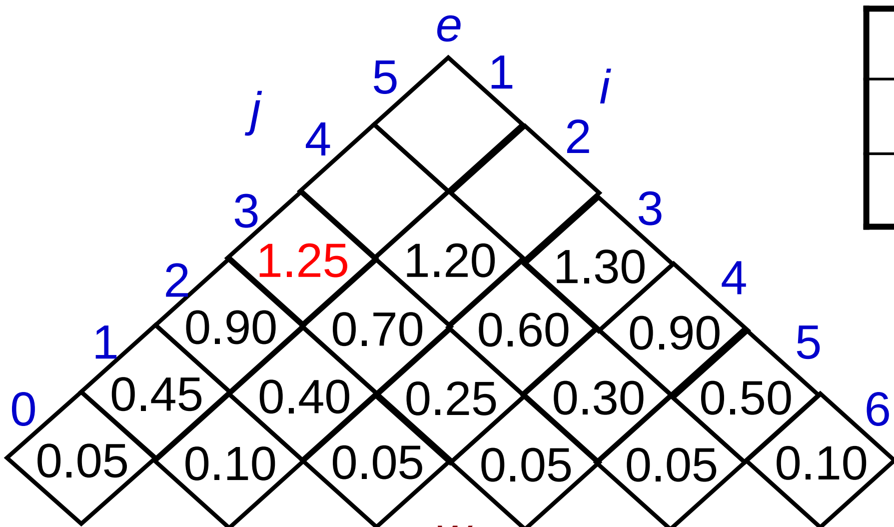
$$= \min\begin{Bmatrix} 0.05 + 0.4 + 0.45 \\ 0.45 + 0.05 + 0.45 \end{Bmatrix}$$

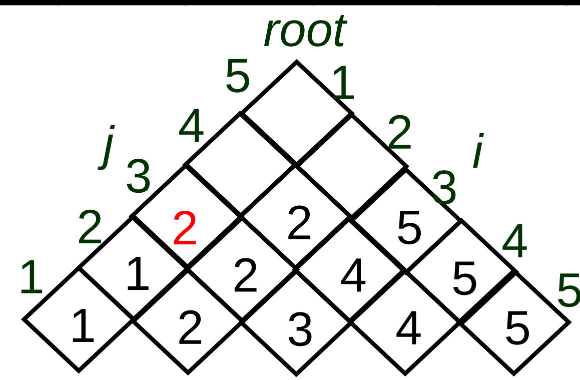
$$= 0.9$$

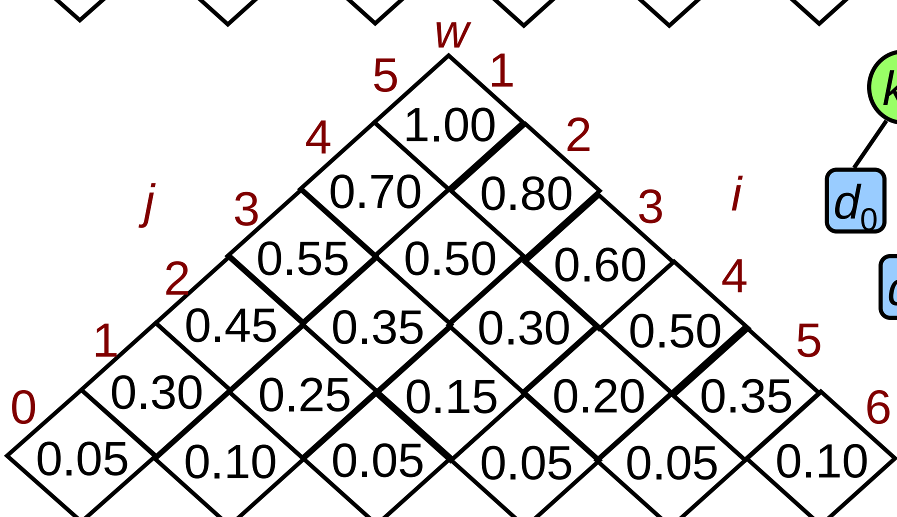


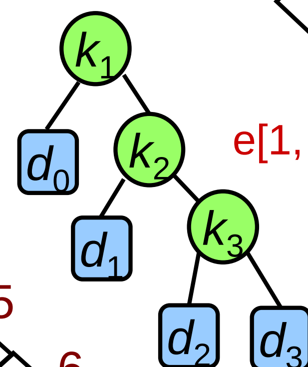
Example

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10





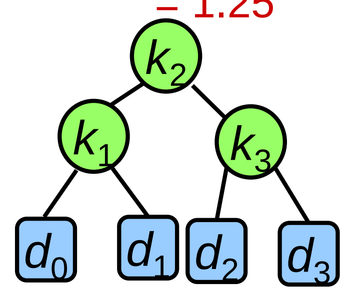


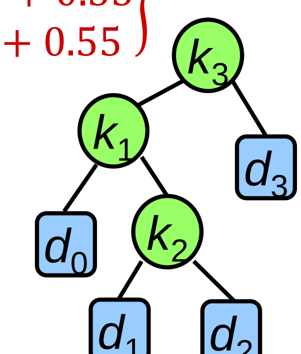


$$e[1,3] = \min\begin{cases} e[1,0] + e[2,3] + w[1,3] \\ e[1,1] + e[3,3] + w[1,3] \\ e[1,2] + e[4,3] + w[1,3] \end{cases}$$

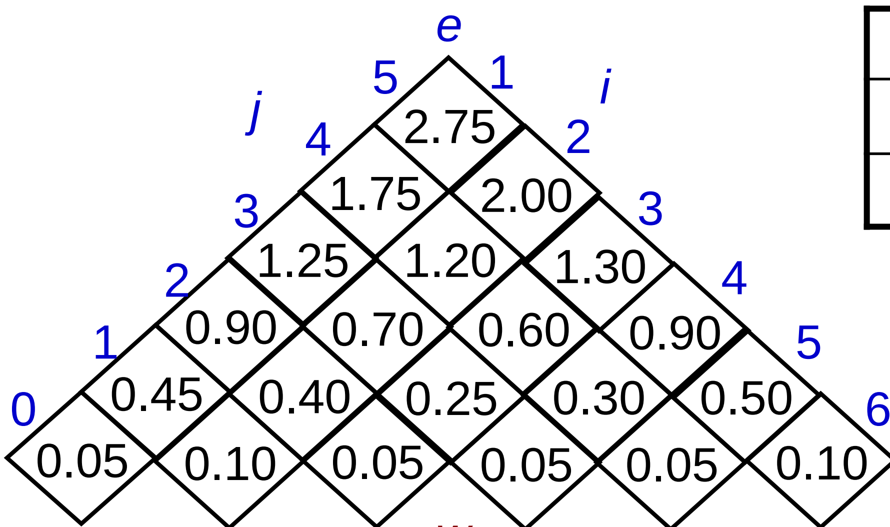
$$= \min\begin{cases} 0.05 + 0.7 + 0.55 \\ 0.45 + 0.25 + 0.55 \\ 0.9 + 0.05 + 0.55 \end{cases}$$

$= 1.25$

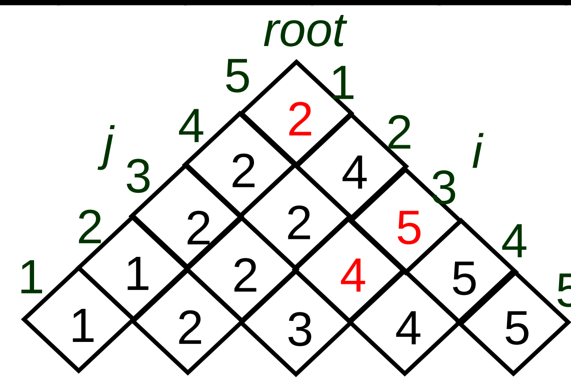


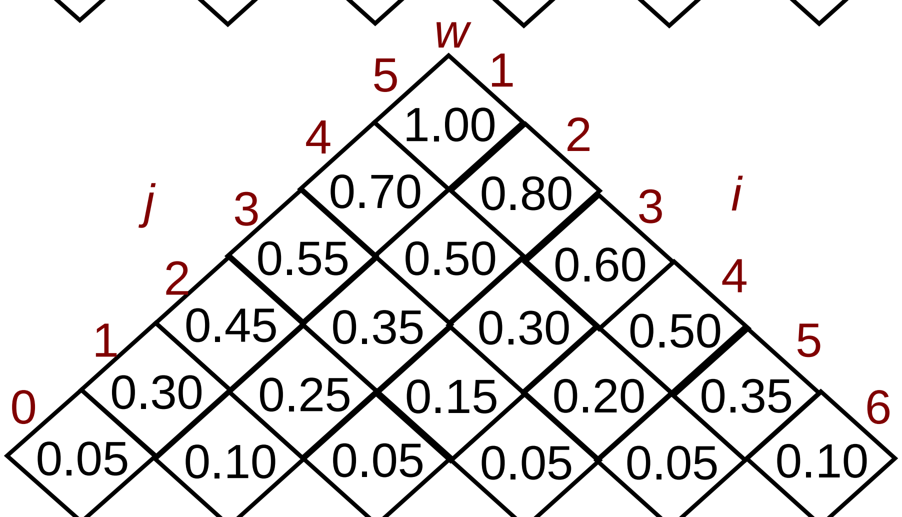


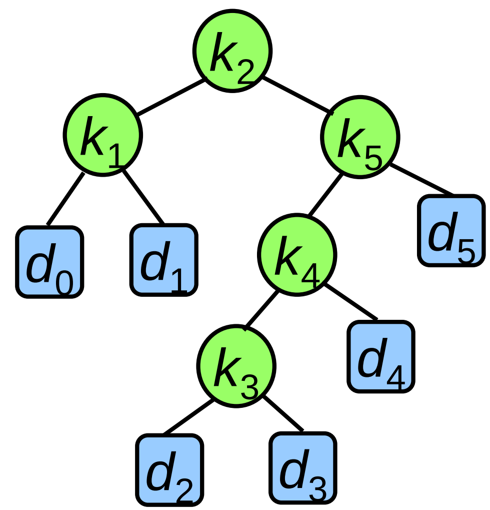
Example



i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10







Keys for Dynamic Programming

- DP typically is applied to optimization problems.
- **DP works best on objects that are linearly ordered and cannot be rearranged**
- Elements of DP
  - **Optimal substructure:** an optimal solution contains within its optimal solutions to subproblems.
  - **Overlapping subproblem:** a recursive algorithm revisits the same problem over and over again; typically, the total number of distinct subproblems is a polynomial in the input size.

Keys for Dynamic Programming

- Dynamic programming can be used if the problem satisfies the following properties:
  - There are only a polynomial number of subproblems
  - The solution to the original problem can be easily computed from the solutions to the subproblems
  - There is a natural ordering on subproblems from “smallest” to “largest,” together with an easy-to-compute recurrence
- Standard operation procedure for DP:
  1. Formulate the answer as a recurrence relation or recursive algorithm (start with divide-and-conquer).
  2. Show that the number of different instances of your recurrence is bounded by a polynomial.
  3. Specify an order of evaluation for the recurrence so you always have what you need.