Dynamic Programming

Mohammad Javad Dousti

Overview

- Introduction
  - Calculating Fibonacci numbers
- Assembly-line scheduling problem
- Longest common subsequence (LCS) problem
- Knapsack problem
- Matrix-chain multiplication problem
- Corporate party planning problem
- Sample problems

Calculating Fibonacci Numbers

Calculating Fibonacci Numbers

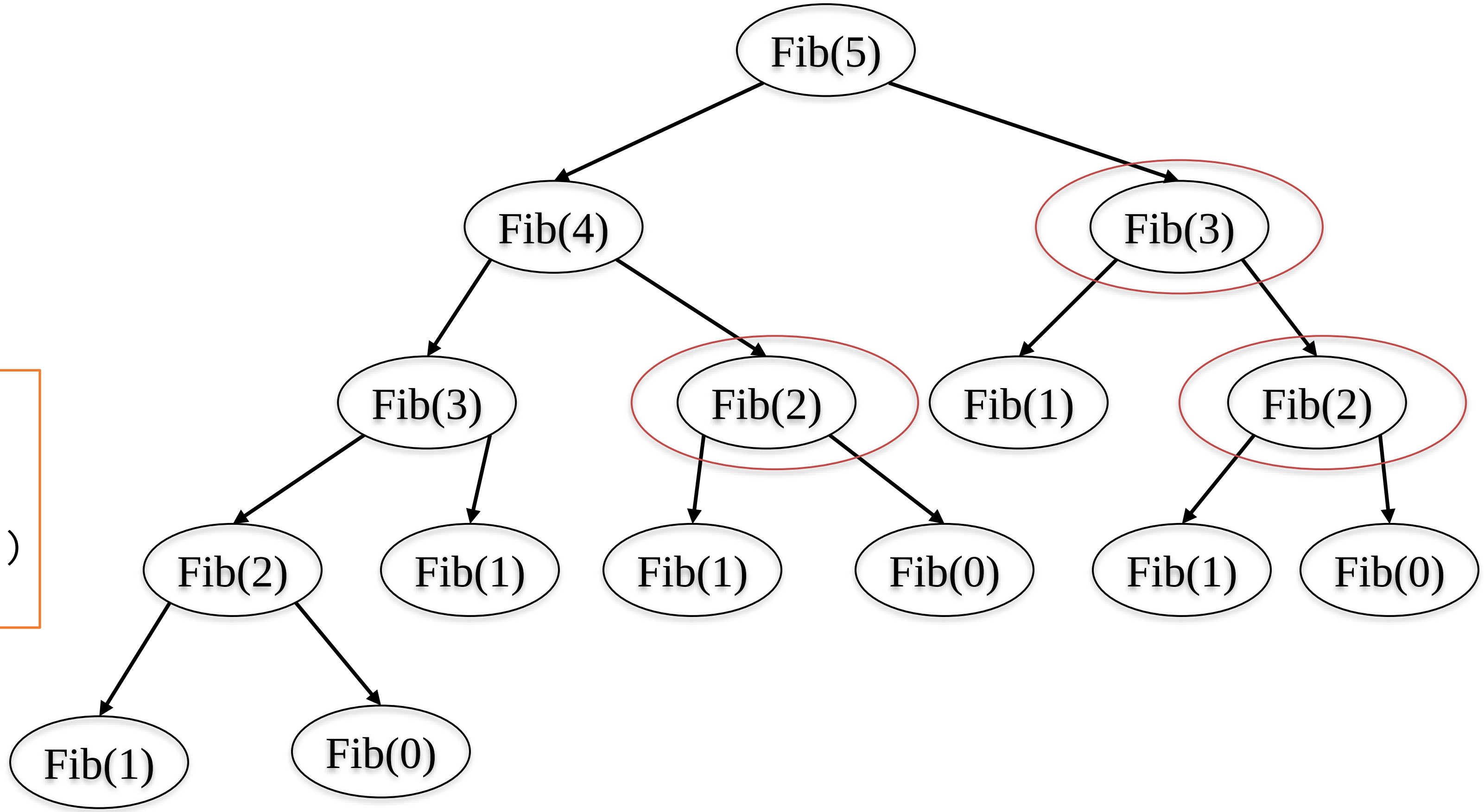
- Let's consider the calculation of Fibonacci numbers:
  - $\mathrm{Fib}(n) = \mathrm{Fib}(n-2) + \mathrm{Fib}(n-1)$
  - $\mathrm{Fib}(0) = 0, \mathrm{Fib}(1) = 1$
- The series looks like this:

$\boldsymbol{n}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**
$\mathbf{Fib}(\boldsymbol{n})$	0	1	1	2	3	5	8

- How to calculate n^{th} value in the series?
- See a nice visualization here:
  - https://www.cs.usfca.edu/~galles/visualization/DPFib.html

Calculating Fibonacci Numbers: Recursive

```
Fib(n){
    if n <= 1
        return n;
    return Fib(n - 1) + Fib(n - 2)
}
```



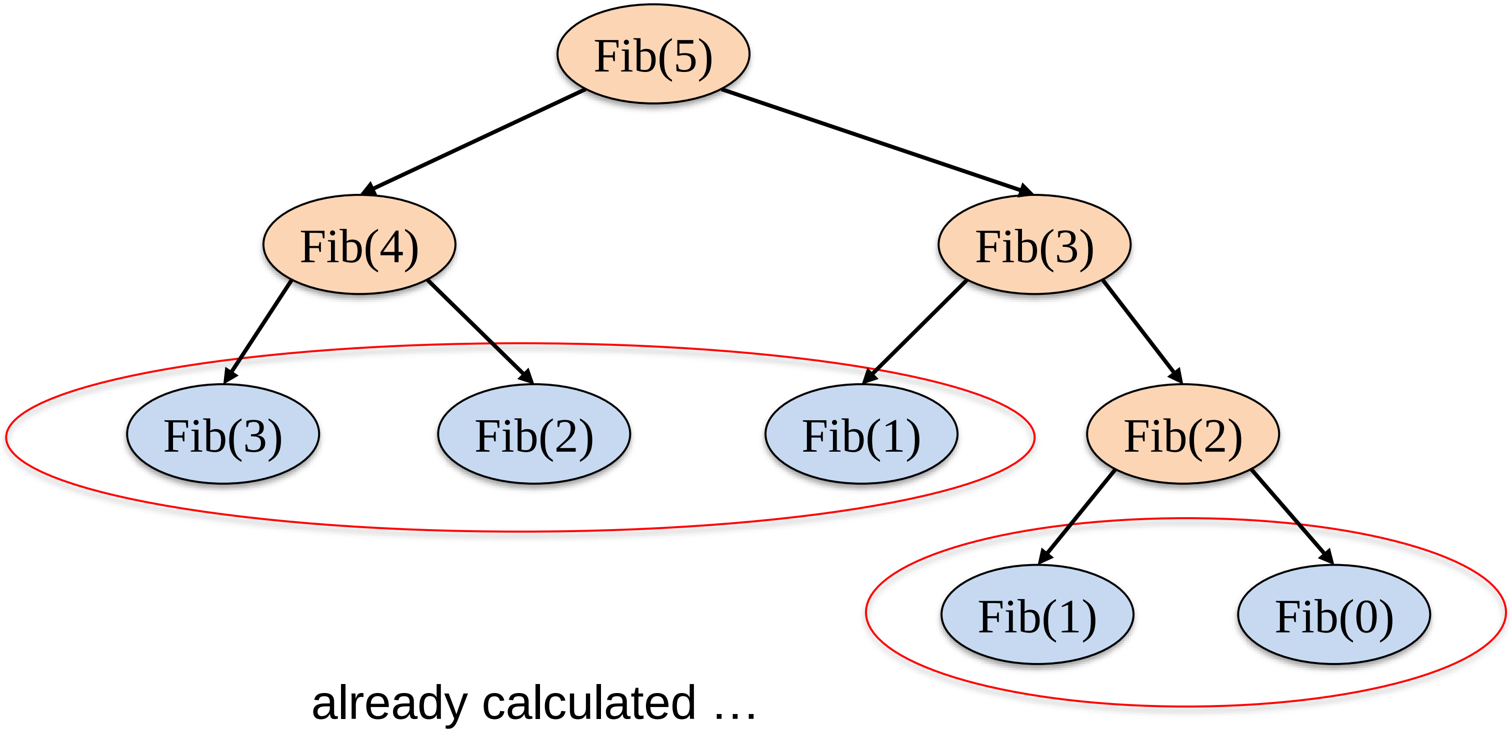
$T(n) = T(n-1) + T(n-2) + O(1)$ → Runtime complexity: $O(2^n)$; why?

$T(0) = T(1) = O(1)$

Calculating Fibonacci Numbers: Memoized

- Store (cache) results from computations for later lookup.

```
// Initialize A[1…n] array with null
Fib(n){
    if n <= 1
        return n;

    if A[n] != null
        return A[n]

    A[n] = Fib(n - 1) + Fib(n - 2)

    return A[n]
}
```



Runtime complexity: O(n)

Calculating Fibonacci Numbers: Dynamic Programming

- Filling out the array from bottom up.
- Using calculated values in the *A* array.
  - It doesn't require recursive calls.
  - Any required value is already calculated.

```
Fib(n){
    A[0] = 0
    A[1] = 1
    for i = 2 to n
        A[i] = A[i-1] + A[i-2]
    return A[n]
}
```

Runtime complexity: O(n)

- An array of size n.
- Calculation of each array cell takes O(1)

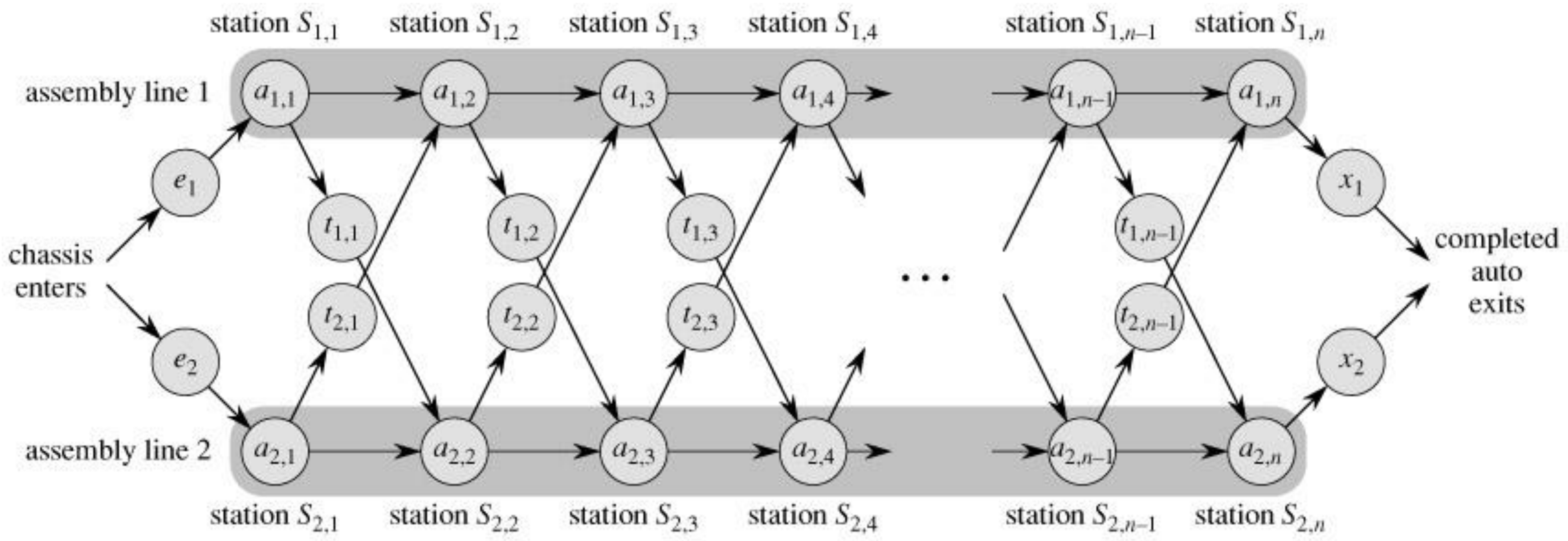
When to use dynamic programming?

- You can solve the problem based on sub-problems.
- The main result of each sub-problem can be stored and retrieved.
- You need to solve a particular sub-problem many times.
- You would like to use bottom-up approach, i.e., from small sub-problems to big ones.

Assembly-Line Scheduling

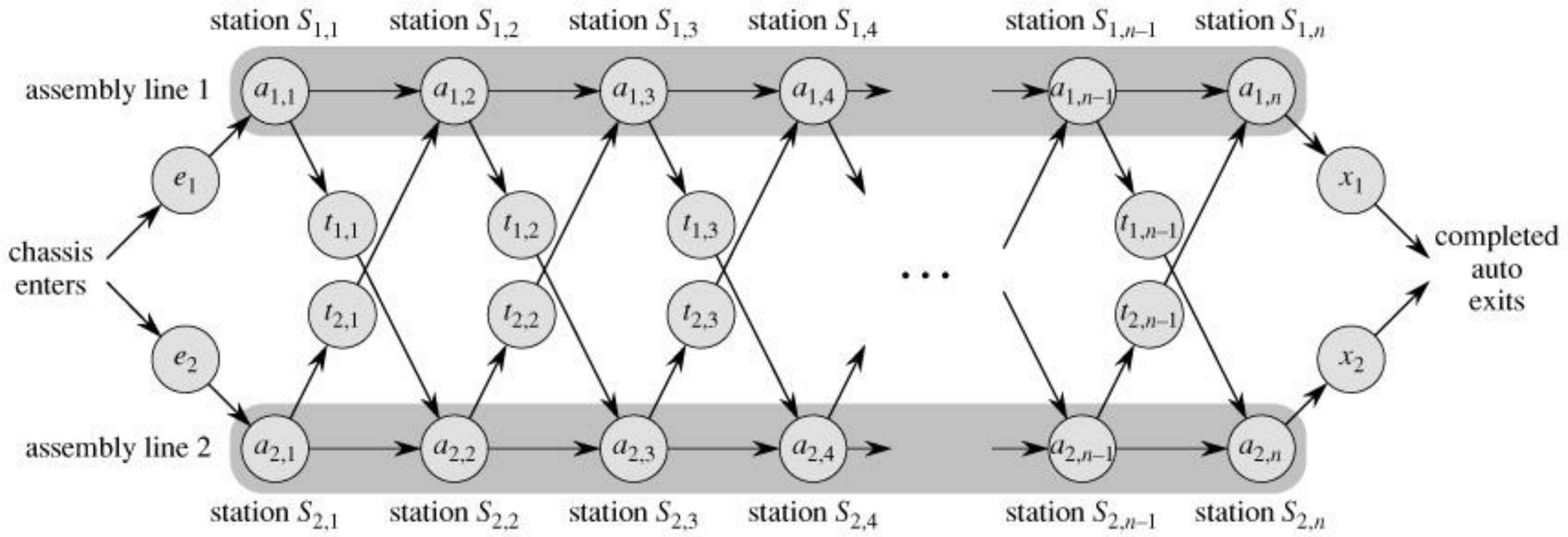
Assembly-Line Scheduling: Problem Statement (1)

- An automotive company produces cars in a factory that has two assembly lines, denoted as $i = 1$ or 2.
- A vehicle chassis enters each assembly line, and has parts added to it at n different stations. The finished vehicle exits at the end of the line.

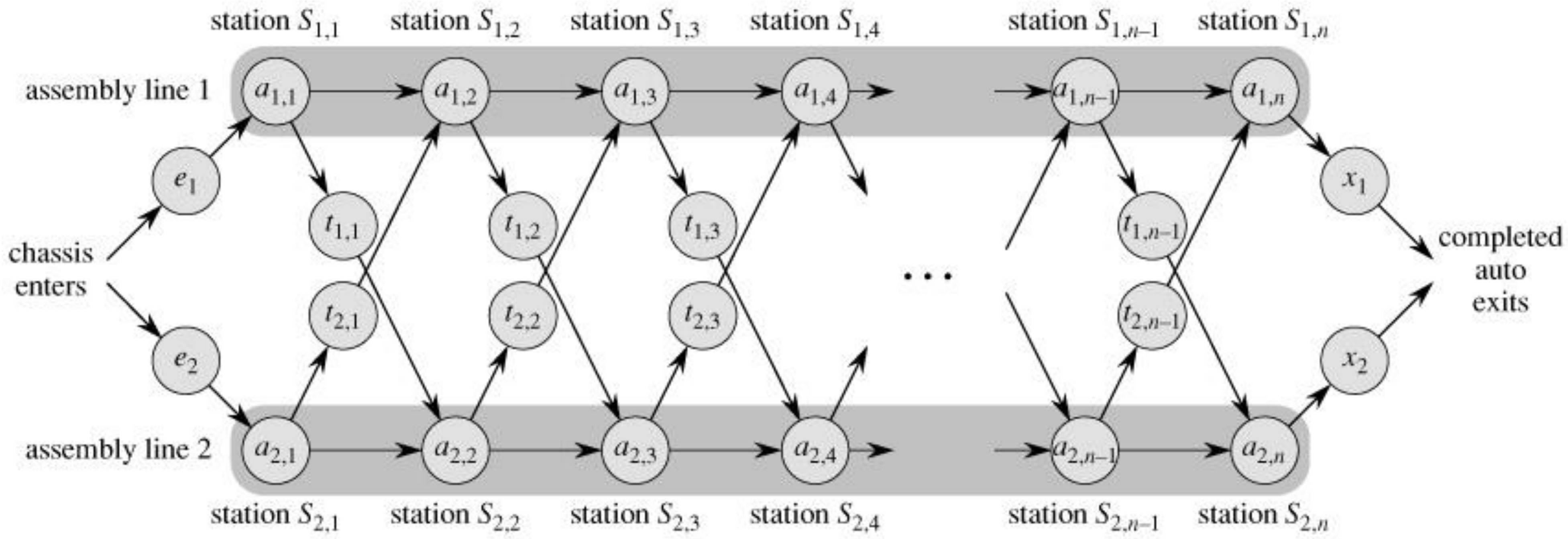


Assembly-Line Scheduling: Problem Statement (2)

- Each assembly line has *n* stations, numbered j = 1, 2, …, n. We denote the j^{th} station on line i as: $S_{i,j}$
- The assembly time taken at station $S_{i,j}$ is $a_{i,j}$
- Each line also has an entry time, e_i, the time taken for the chassis to enter line i, and an exit time, x_i, the time taken for the completed vehicle to leave line i.
- The time to transfer a chassis between line i after leaving station $S_{i,j}$ is $t_{i,j}$, where i = 1, 2 and j = 1, 2, …, n-1 (since we can't transfer after the last station.)



- **Problem:** Find which stations to choose from line 1 and which to choose from line 2 in order to minimize the total time through the factory for one auto.



Brute Force Solution

- A brute force approach yields exponential complexity time.
- At each station, we can make two choices: stay on the line, or transfer.
  - Our set of possible decisions doubles at each station.
  - Since we have *n* stations, there are 2^n possible ways to choose stations.
  - Hence, the runtime complexity is $O(2^n)$.

How to design a faster algorithm?

Fastest Way Through the Factory

- Consider the fastest way for a chassis to move from the start, to station $S_{1,j}$.
  - If j = 1, it has come from the entry point (trivial.)
  - For j = 2, 3, …, n, there are two choices:
    1. The chassis came from station $S_{1,j-1}$. The time of moving between j-1 to j is zero, as they are on the same line.
    2. The chassis came from $S_{2,j-1}$. The transfer time between lines was $t_{2,j-1}$.
- Suppose the fastest way through $S_{1,j}$ is from $S_{1,j-1}$.
  - The chassis must've taken the fastest time to station $S_{1,j-1}$.
  - If there exists a faster way to $S_{1,j-1}$, our chassis would've taken it. We could then substitute this faster sub-route into our route to $S_{1,j}$.
  - But this would then lead to a faster time for $S_{1,j}$, which implies that $S_{1,j}$ was not the fastest way through the plant: a contradiction.
- The symmetric argument applies to the fastest time through $S_{2,j}$

Optimal Substructure

- The optimal solution (fastest time through Station $S_{i,j}$) for assembly line scheduling contains within it, other optimal solutions to subproblems.
- This property is known as ***optimal substructure***.
- This problem property is an essential requirement for a dynamic programming solution.
- Therefore, we can build an optimal solution to the fastest time problem, by building optimal solutions to subproblems.

Developing a Recursive Solution (1)

- Let $f_i[j]$ be the fastest time to get a chassis from the start through to station j on line i (i.e., $S_{i,j}$).
- Let f^* be the fastest time for the chassis to get all the way through the factory, arriving at the exit as a finished vehicle.
- For the station to reach the exit, it must get all the way to station n on either line 1 or 2, and then exit the factory.
- Since $f_1[n]$ or $f_2[n]$ must be the fastest way, we can define our fastest time solution as:

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

where x_1 and x_2 are exit times from lines 1 and 2, respectively.

Developing a Recursive Solution (2)

- The fastest times through station 1 is simply the entry time e_i plus the first station assembly time $a_{i,1}$:

$$f_1[1] = e_1 + a_{1,1}$$
$$f_2[1] = e_2 + a_{2,1}$$

- Let's now define the fastest time for $f_i[j]$ for stations $j = 2, \ldots,$ n
- We already established that the fastest time through $S_{1,j}$ is either:
  - From $S_{1,j-1}$ then into $S_{1,j}$ ***OR***
  - From $S_{2,j-1}$, transfer from line 2 to line 1, then through S1,j.
- Therefore:

$$f_1[j] = \min\left(f_1[j-1], f_2[j-1] + t_{2,j-1}\right) + a_{1,j}$$
$$f_2[j] = \min\left(f_2[j-1], f_1[j-1] + t_{1,j-1}\right) + a_{2,j}$$

Developing a Recursive Solution (3)

- We can now define our final recursive equations:
  - $$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min\left(f_1[j-1], f_2[j-1] + t_{2,j-1}\right) + a_{1,j} & \text{otherwise} \end{cases}$$
  - $$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min\left(f_2[j-1], f_1[j-1] + t_{1,j-1}\right) + a_{2,j} & \text{otherwise} \end{cases}$$

Tracing our way through the factory

- The last thing we need to do is keep track of the stations that we passed through, which are used in constructing our fastest way solution.
- Let $l_i[j]$ be the line number, either 1 or 2, whose station $j - 1$ is used in a fastest way through station $S_{i,j}$.
- Let l^* be the line whose station n is used in a fastest way through the entire factory.

An Iterative Solution

- We can avoid exponential running time by referencing previously calculated values, which we will store in our *f*-table.
  - This saves us wasted cycles recomputing the same values.
- Dynamic programming uses additional memory to save computation time. The cached results always take the form of *a table*.
- Recall for $j \geq 2$, $f_i[j]$ depends only on $f_1[j\text{-}1]$ and $f_2[j\text{-}1]$.
  - By computing $f_i[j]$ as j increases, j moves from left to right, we can compute our fastest way in $O(\text{n})$ time.

Optimal Algorithm

```
FastestWay(a, t, e, x, n){
    f₁[1] = e₁ + a₁,₁
    f₂[1] = e₂ + a₂,₁
    for j = 2 to n {
        if f₁[j - 1] + a₁,ⱼ ≤ f₂[j - 1] + t₂,ⱼ₋₁ + a₁,ⱼ {
            f₁[j] = f₁[j - 1] + a₁,ⱼ
            l₁[j] = 1
        } else {
            f₁[j] = f₂[j - 1] + t₂,ⱼ₋₁ + a₁,ⱼ
            l₁[j] = 2
        }
        if f₂[j - 1] + a₂,ⱼ ≤ f₁[j - 1] + t₁,ⱼ₋₁ + a₂,ⱼ {
            f₂[j] = f₂[j - 1] + a₂,ⱼ
            l₂[j] = 2
        } else {
            f₂[j] = f₁[j - 1] + t₁,ⱼ₋₁ + a₂,ⱼ
            l₂[j] = 1
        }
    }
    if f₁[n] + x₁ ≤ f₂[n] + x₂ {
        f* = f₁[n] + x₁
        l* = 1
    } else {
        f* = f₂[n] + x₂
        l* = 2
    }
}
```

Runtime complexity: $O(n)$

Print output

- Having computed our fastest way solution, we need a final helper method to convert our *l** and *l*-table into a path of stations.
- This procedure outputs stations in reverse order, beginning with our exit station & line.

```
i = l*
print "line " l* ", station " n
for j = n downto 2
    i = li[j]
    print “line “ li[j] “, station ” j - 1
```

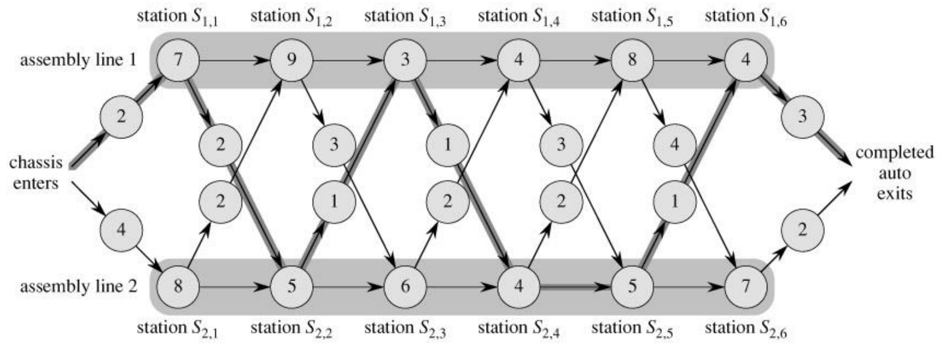
Output:

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

Tables from our example

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$



- It’s relatively easy to rework our algorithm to print stations in increasing order, to do so we would use recursion. Can you do it?

Longest Common Subsequence (LCS)

Longest Common Subsequence (LCS)

- **Problem:** Given 2 sequences, $X = \langle x_1,\ldots,x_m\rangle$ and $Y = \langle y_1,\ldots,y_n\rangle$, find a common subsequence whose length is maximum.
  - Subsequence need not be consecutive, but must be in order.
- Example 1:
  - X = ⟨a, a, b, a, a, c, d⟩
  - Y = ⟨b, b, a, d, a, a, c⟩
  - LCS = ⟨a, a, a, c⟩
- Example 2:
  - X = ⟨b, a, b, a, c, a, c, a⟩
  - Y = ⟨a, d, a, d, b, b, c, c, a⟩
  - LCS = ⟨a, a, c, c, a⟩
  - Note that LCS might not be unique. For instance, you can consider, ⟨b, b, c, c, a⟩ and ⟨a, b, c, c, a⟩ with length 5.

Brute Force

- For every subsequence of X, check whether it's a subsequence of Y .
- Runtime complexity: $\Theta(n2^m)$.
  - 2^m subsequences of X to check.
  - Each subsequence takes $\Theta(n)$ time to check.
    - Scan Y for first letter, for second, and so on.

How to design a faster algorithm?

Optimal Substructure

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any 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 **either** $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. **or** $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

- **Notation:** $X_i = \langle x_1,\ldots,x_i \rangle$, $Y_i = \langle y_1,\ldots,y_i \rangle$, and $Z_i = \langle z_1,\ldots,z_i \rangle$ are the first i letters of X, Y, and Z, respectively.
- This says what any longest common subsequence must look like!

Optimal Substructure: Proof (1)

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any 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 **either** $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .
3. **or** $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

- **Proof:** Case 1: $x_m = y_n$
  - Any sequence Z' that does not end in $x_m = y_n$ can be made longer by adding $x_m = y_n$ to the end.
  - Hence, LCS Z must end in $x_m = y_n$.
  - Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}, and there is no longer common subsequence of X_{m-1} and Y_{n-1}, or Z would not be an LCS.

Optimal Substructure: Proof (2)

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any 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 **either** $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. **or** $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

- **Proof:** Case 2: $x_m \neq y_n$, and $z_k \neq x_m$
  - Since Z does not end in x_m,
  - Z is a common subsequence of X_{m-1} and Y, and
  - there is no longer common subsequence of X_{m-1} and Y, or Z would not be an LCS.
- Case 3 can be proven similarly.

Recursive Solution

- Define $c[i, j]$ = length of LCS of X_i and Y_j .
- We want $c[m,n]$.

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

- This gives a recursive algorithm and solves the problem.
  - But does it solve it well?
  - It becomes exponential in terms of m and n.

Computing the Length of an LCS

- $b[i, j\ ]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j.
- $c[m,n]$ contains the length of an LCS of X and Y.
- Runtime Complexity: $O(mn)$

```
LCS-Length(X, Y){
    m = length[X]
    n = length[Y]
    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 xi = yj {
                c[i, j] = c[i–1, j–1] + 1
                b[i, j] = "↖"
            } else if 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
}
```

An example

- Suppose X= ⟨A, B, C, B, D, A, B⟩ and
  Y = ⟨B, D, C, A, B, A⟩.
- The given algorithm fills tables b and c in ***row-major*** way.
  - Fills in the first row of *b* and *c* tables from left to right, then the second row, and so on
- One can easily use these tables to find the LCS length and reconstruct the solution.

	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

- ❑ Initial call is Print-LCS($\boldsymbol{b}$, $\boldsymbol{X}$, $\boldsymbol{m}$, $\boldsymbol{n}$).
- ❑ When $\boldsymbol{b}[\boldsymbol{i}, \boldsymbol{j}]$ = “↖”, we have extended LCS by one character.
- ❑ Runtime complexity: $O(m+n)$

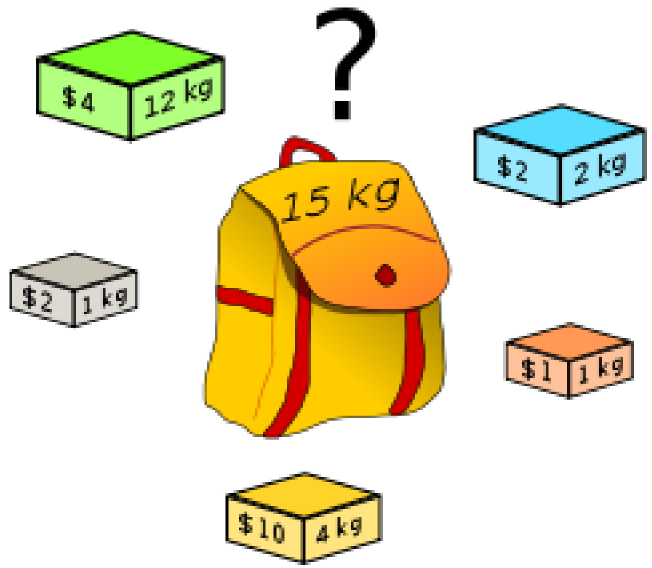
```
Print-LCS(b, X, i, j){
    if i = 0 or j = 0 {
        return

    if b[i, j] = “↖” {
        Print-LCS(b, X, i-1, j-1)
        print xi
    } else if b[i, j] = “↑” {
        Print-LCS(b, X, i-1, j)
    } 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

Knapsack Problem

The Knapsack Problem

- A thief breaks into a house, carrying a knapsack...
  - He can carry up to 15kg of loot.
  - He has to choose which of *n* items to steal.
  - Each item has some weight and some value.
- **Goal:** Maximize the value of looted items, while not exceeding the weight limit (i.e., 15kg.)



Knapsack: Formal Definition

- Given n objects, with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$, find $S \subseteq \{1, 2, \ldots, n\}$ such that:

$$\text{Maximize } \Sigma_{i \in S} v_i$$
$$\text{subject to } \Sigma_{i \in S} w_i \le W$$

- Alternatively, one can find x_i's such that:

$$\text{Maximize } \Sigma_{i=1}^{n} x_i v_i$$
$$\text{subject to } \Sigma_{i=1}^{n} x_i w_i \le W \text{ and } x_i \in \{0, 1\}$$

- The second formulation brought the name "0-1 knapsack" for the problem.

Brute Force

- The straight forward way:
  - The complexity is $O(2^n)$
  - Example:

$n = 3$

$(v_1, v_2, v_3) = (1, 2, 5)$

$(w_1, w_2, w_3) = (2, 3, 4)$

$W = 6$

x_1 x_2 x_3	$\Sigma x_i v_i$	$\Sigma x_i w_i$
0 0 0	0	0
0 0 1	5	4
0 1 0	2	3
0 1 1	–	–
1 0 0	1	2
1 0 1	$\underline{6}$ $\leftarrow$ *solution*	6
1 1 0	3	5
1 1 1	–	–

Optimal Substructure

- Consider the i^{th} object. The optimal solution can either:
  - Contain this object. Hence, the new limit will be W - w_i
  - Not contain this object. Hence, the limit remains W
- In both cases, the remaining problem is smaller, because we have fewer objects to consider.
- Suppose $A[i, j]$ is the highest values of objects we can take if we can choose from the first i objects and the weight limit is j.

$$A[i,j] = \begin{cases} A[i-1,j], & \text{if } j < w_i \\ \max(A[i-1,j], A[i-1,j-w_i] + v_i), & \text{if } j \geq w_i \end{cases}$$

- What's the base case?
  - $A[0, j] = 0$

Knapsack: Dynamic Programming Solution

- Array needs to be filled in row-major order.

$W+1$ (columns)

$n+1$ (rows)

	0	1	…	j-w_i	…	j	…	W
0	0	0	…	0	0	0	0	0
1								
…								
i-1				$A[i\text{-}1, j\text{-}w_i)$		$A[i\text{-}1, j]$		
i				$+v_i$		$A[i, j]$		
…								
n								

```
Knapsack(v, w, n, W) {
    for j=0 to W {
        A[0, j] = 0
    }
    for i = 1 to n {
        for j=0 to W {
            A[i, j] = A[i-1, j]
            if w_i ≤ j and A[i, j]<v_i + A[i-1, j-w_i] {
                A[i, j] = v_i + A[i-1, j-w_i]
            }
        }
    }
    return A[n, W]
}
```

Runtime complexity: $O(nW)$

Space complexity: $O(nW)$

Reducing the Required Space for Knapsack Problem (1)

```
Knapsack(v, w, n, W) {
    for j=0 to W
        A[0, j] = 0

    for i = 1 to n {
        for j=0 to W {
            A[i, j] = A[i-1, j]
            if A[i, j] < vᵢ + A[i-1, j-wᵢ] and wᵢ ≤ j
                A[i, j] = vᵢ + A[i-1, j-wᵢ]
        }
    }
    return A[n, W]
}
```

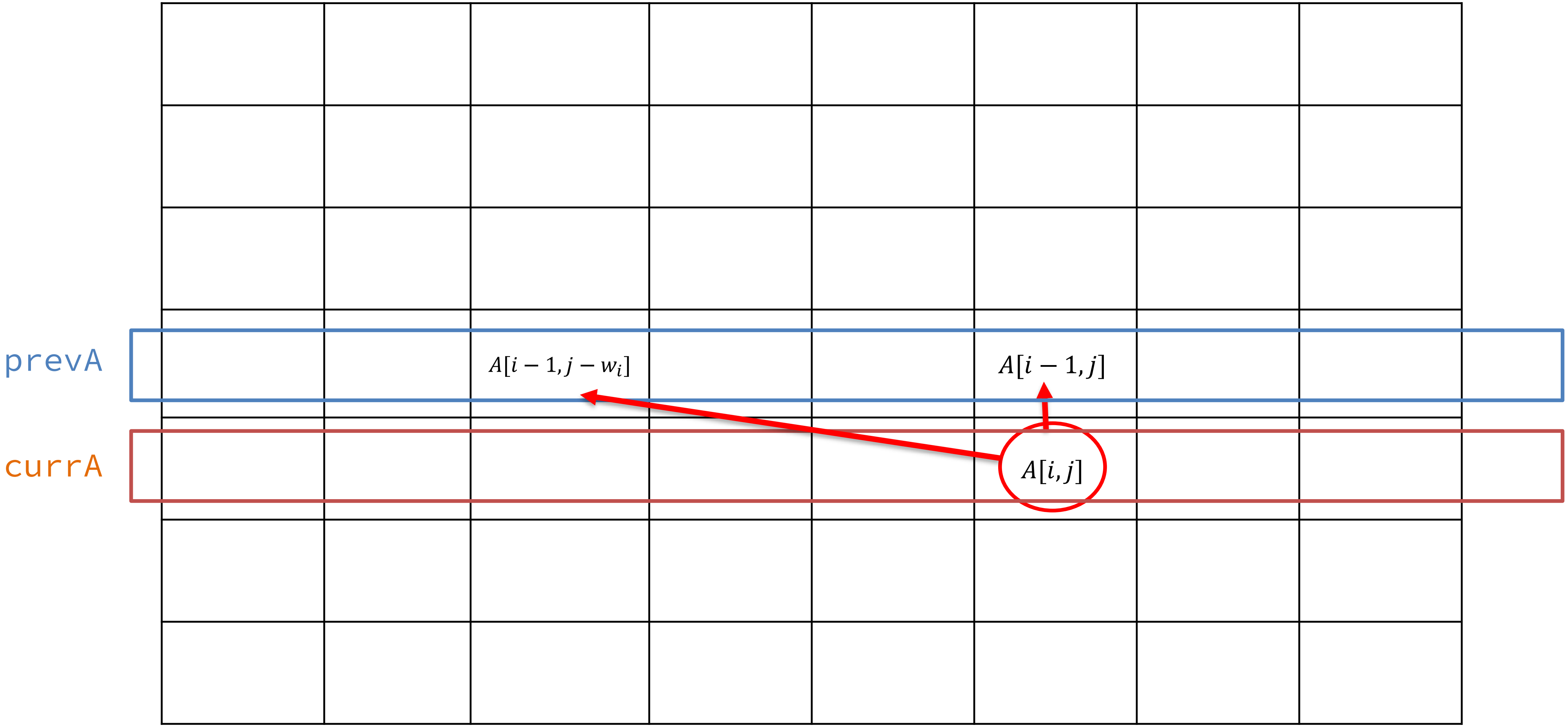
Implementation with $O(nW)$ space

```
Knapsack(v, w, n, W) {
    for j=0 to W
    prevA[j] = 0

    for i = 1 to n {
        for j=0 to W {
            currA[j] = prevA[j]
            if wᵢ ≤ j and currA[j] < vᵢ + prevA[j-wᵢ]
                currA[j] = vᵢ + prevA[j-wᵢ]
        }
        prevA = currA
    }
    return currA[W]
}
```

Replace `A[i,j]` with `currA[j]` and `A[i-1,j]` with `prevA[j]`

Implementation with $O(W)$ space

Reducing the Required Space for Knapsack Problem (2)



Matrix-Chain Multiplication Problem

Matrix-Chain Multiplication Problem

- If B is a $p \times q$ matrix and C is a $q \times r$ matrix, their multiplication (i.e., $B \times C$) requires $p \times q \times r$ scalar multiplications.
- In order to calculate $A_1 \times A_2 \times \ldots \times A_n$, two matrices can be multiplied together at a time.
- Problem: Find the best parenthesization of $A_1 \times A_2 \times \ldots \times A_n$ such that it requires the fewest scalar multiplications.

Matrix-Chain Multiplication: An Example

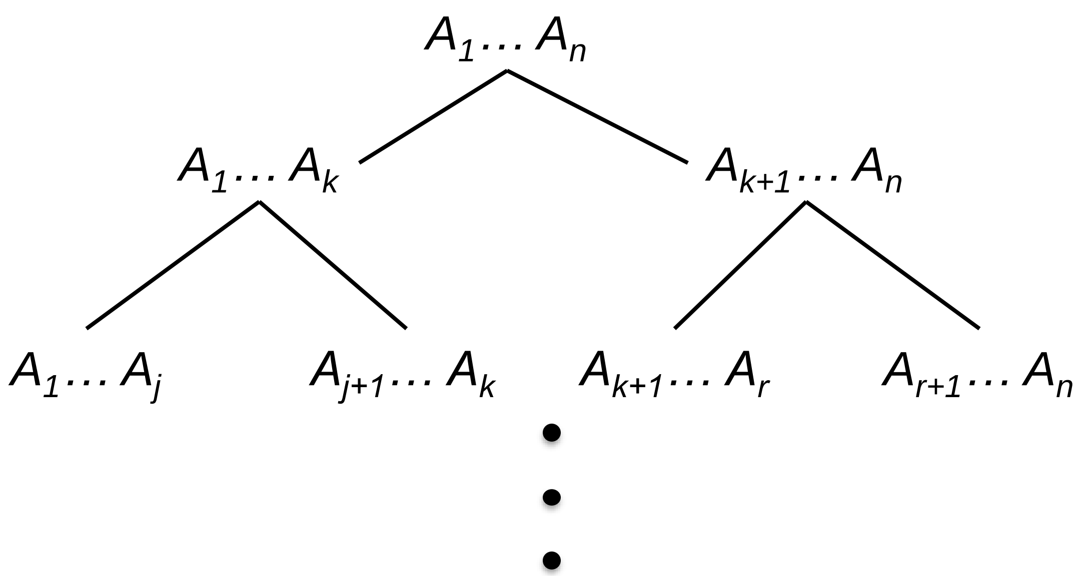
- Suppose you want to multiply A_1, A_2, and A_3 matrices.
  - Matrix dimensions: A_1=10×100, A_2=100 ×5, A_3=5×10
- Method 1: $(A_1 \times A_2) \times A_3$
  - Required scalar multiplications:
    - $B=A_1 \times A_2$ requires 10×100×5=5000 multiplications.
    - $B \times A_3$ requires 10×5×10=500 multiplications.
    - Total multiplications: 5,500
- Method 2: $A_1 \times (A_2 \times A_3)$
  - Required scalar multiplications:
    - $B=A_2 \times A_3$ requires 100×5×10=5000 multiplications.
    - $A_1 \times B$ requires 10×100×10=10,000 multiplications.
    - Total multiplications: 15,000

Optimal Substructure

- Suppose A_i is a $p_{i-1}\times p_i$ matrix.
- Where to place outer parentheses?

$$\underbrace{(A_1\times\ldots\times A_k)}_{\substack{\text{B} \\ \text{Dimension: } p_0\times p_k}}\times\underbrace{(A_{k+1}\times\ldots\times A_n)}_{\substack{\text{C} \\ \text{Dimension: } p_k\times p_n}}$$

- How subproblems would look like?



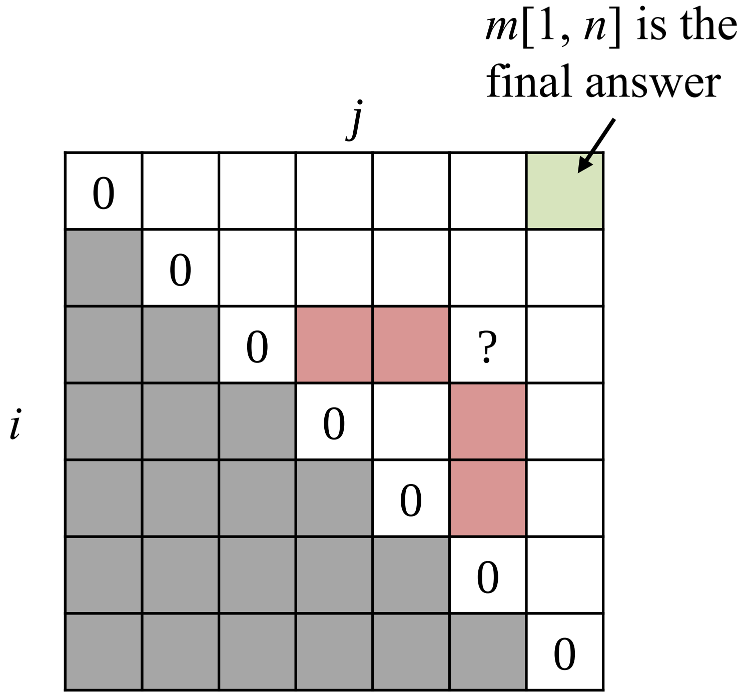
A Recursive Solution

- Suppose $m[i, j]$ is the minimum number of scalar multiplications needed to compute matrix $A_i \times \ldots \times A_j$ (shown as $A_{i\ldots j}$).
- Assuming that the best place to do outer parenthesization is at location k, $m[i, j]$ can be written as

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

Computing the Optimal Cost

- How to fill out the table?



m table

```
MatrixChainOrder(p) {
    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_kp_j
                if q < m[i, j] {
                    m[i, j] = q
                    s[i, j] = k
                }
            }
        }
    }
    return m[1, n], s
}
```

Runtime complexity: $O(n^3)$
Space complexity: $O(n^2)$

Constructing an Optimal Solution

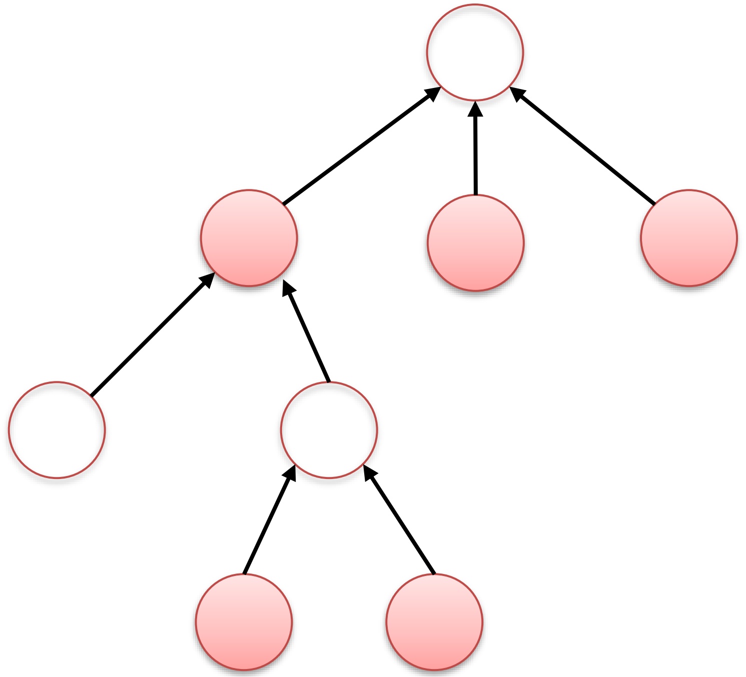
- $s[i, j]$ records k such that an optimal parenthesization of $A_i \ldots A_j$ splits the product between A_k and A_{k+1}.
- In other words, split happens as follows: $(A_i \ldots A_{s[i,j]}) \times (A_{s[i,j]+1} \ldots A_j)$.

```
PrintOptimalParens(s, i, j) {
    if i == j {
        print “Ai”
    } else {
        print “(“
        PrintOptimalParens(s, i, s[i, j])
        PrintOptimalParens(s, s[i, j] + 1, j)
        print “)“
    }
}
```

Corporate Party Planning Problem

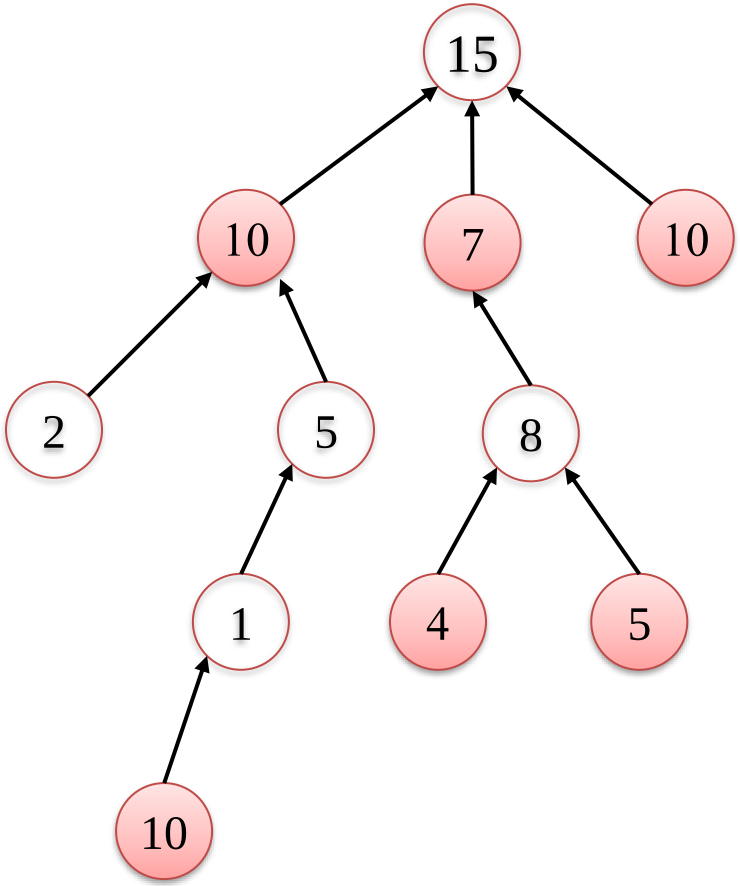
Corporate Party Planning Problem

- Suppose you want to invite employees of a corporation to a party.
- Each employee i has a friendliness score of v_i.
- You need to make sure no employee is invited with their direct manager.
- How do you choose employees such that total friendliness of invitees is maximized?



Corporate Party Planning: An Example

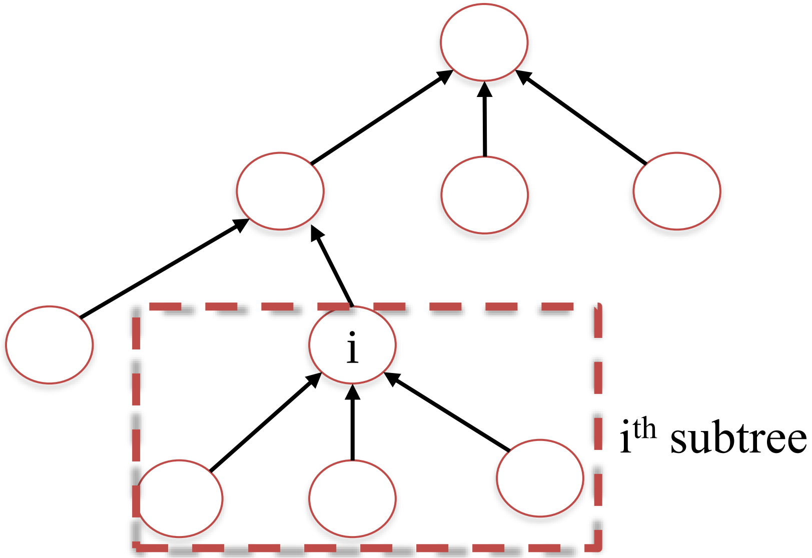
- Total friendliness of invitees: 10+7+10+4+5+10 = 46



Optimal Substructure

- Define $A[i]$ as the best party if i was the CEO.
- $A[i]$ can be defined as follows:

$$A[i] = \begin{cases} v_i + \Sigma_{j:\,\mathrm{grandchild}(i)}\, A[\mathrm{j}] & \text{if } i \text{ is invited} \\ \Sigma_{j:child(i)} A[j] & \text{if } i \text{ isn't invited} \end{cases}$$



A Recursive Solution

- $A[i]$: The best party if i was the CEO.
- $B[i]$: The best party if i was the CEO and they aren't invited.
- Using these definitions, one can write:

$$B[i] = \Sigma_{j \in child(i)} A[j]$$

$$A[i] = \max\{v_i + \Sigma_{j \in child(i)} B[j], B[i]\}$$

- $A[i]$ and $B[i]$ can be calculated for node i, if they are calculated for all i's children.

Compute the Solution

- Reminder
  - $B[i] = \Sigma_{j \in child(i)} A[j]$
  - $A[i] = \max\{v_i + \Sigma_{j \in child(i)} B[j], B[i]\}$

```
PartyPlanning(v, i) {
    // A and B are defined outside the function
    A[i] = v[i]
    B[i] = 0
    for j ∈ i’s children {
        PartyPlanning(v, j)  // DFS
        // A[j] and B[j] are already calculated
        A[i] += B[j]
        B[i] += A[j]
    }
    A[i] = max(A[i], B[i])
}
```

Runtime complexity: $O(n)$
Space complexity: $O(n)$

Solving a Problem using Dynamic Programming

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, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Where did the name "*dynamic programming*" come from?

R. Bellman Coined the Term “Dynamic Programming” (1)

- The word *dynamic* was chosen by Richard Bellman to capture the time-varying aspect of the problems, and because it sounded impressive.
- The word programming referred to the use of the method to find an optimal program, in the sense of a military schedule for training or logistics.
  - This usage is the same as that in the phrases *linear programming* and *mathematical programming*, a synonym for *mathematical optimization*.

R. Bellman Coined the Term “Dynamic Programming” (2)

- Richard Bellman, “Eye of the Hurricane: an autobiography,” 1984:
  - *“…The 1950s were not good years for mathematical research. [the] Secretary of Defense …had a pathological fear and hatred of the word, research…*

    I decided therefore to use the word, “programming”.

    I wanted to get across the idea that this was dynamic, this was multistage… I thought, let’s … take a word that has an absolutely precise meaning, namely dynamic… it’s impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It’s impossible.

    Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to.”

Edit Distance: A Practical Example*

* Some slides are courtesy of Prof. Dan Jurafsky of Stanford (CS 124 course.)

How similar are two strings?

- Spell correction
  - The user typed "graffe"
  - Which is closest?
    - graf
    - grab
    - grail
    - giraffe
- Computational Biology
  - Align two sequences of nucleotides:

  AGGCTATCACCTGACCTCCAGGCCGATGCCC

  TAGCTATCACGACCGCGGTCGATTTGCCCGAC
- Resulting alignment:

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC
```

Edit Distance

- The minimum edit distance between two strings.
- Is the minimum number of editing operations
  - Insertion
  - Deletion
  - Substitution (A.K.A. Replacement)
- Needed to transform one into the other.
- Example: Intention → Execution

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

Usage in Natural Language Processing (NLP)

- Evaluating Machine Translation and speech recognition models.
  - Spokesman confirms senior government adviser was shot!
  - Spokesman said the senior adviser was shot dead!

S I D I

How to Find Minimum Edit Distance?

- **Problem:** Suppose two strings X (w/ length *m*) and Y (w/ length *n*) are given and we want to transform X to Y using minimum number of insertions, deletions, and substitutions.
- Brute force (search all possible ways) is very slow.
- Is there any better way?

Recursive Solution

- $D(i, j)$ is defined as the edit distance between X[:i] and Y[:j].
  - The edit distance between X and Y is defined as $D(m, n)$.
- Recursive equation for $D(i, j)$:

$$\begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ D(i-1, j-1) & \text{if } X[i] = Y[j] \\ \min(\underbrace{D(i-1, j)}_{\text{deletion}} + 1, \underbrace{D(i, j-1)}_{\text{insertion}} + 1, \underbrace{D(i-1, j-1) + 1}_{\text{substitution}}) & \text{if } X[i] \neq Y[j] \end{cases}$$

Examples

- Tables for transforming “sitting” to “kitten” and “Saturday” to “Sunday”.

		k	**i**	**t**	**t**	**e**	**n**
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

		S	**a**	**t**	**u**	**r**	**d**	**a**	**y**
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Optimal Solution: Wagner–Fischer Algorithm

```
EditDistance(X, Y) {
    m, n = X.length, Y.length
    for i = 1 to m {
        d[i, 0] = i
    }
    for j = 1 to n {
        d[0, j] = j
    }
    for i = 1 to m {
        for j = 1 to n {
            if X[i-1] = Y[j-1] {
                d[i,j] = d[i-1, j-1]
                ptr[i, j] = 2    // Assume that array indices are zero based.
            } else {
                d[i,j] = min(d[i-1,j], d[i,j-1], d[i-1, j-1]) + 1
                ptr[i, j] = argmin(d[i-1, j], d[i, j-1], d[i-1, j-1])
            }
        }
    }
    return d[m,n], ptr
}
```

Runtime complexity?
Space complexity?
Constructing the solution complexity?

Sample Problems

True or False?

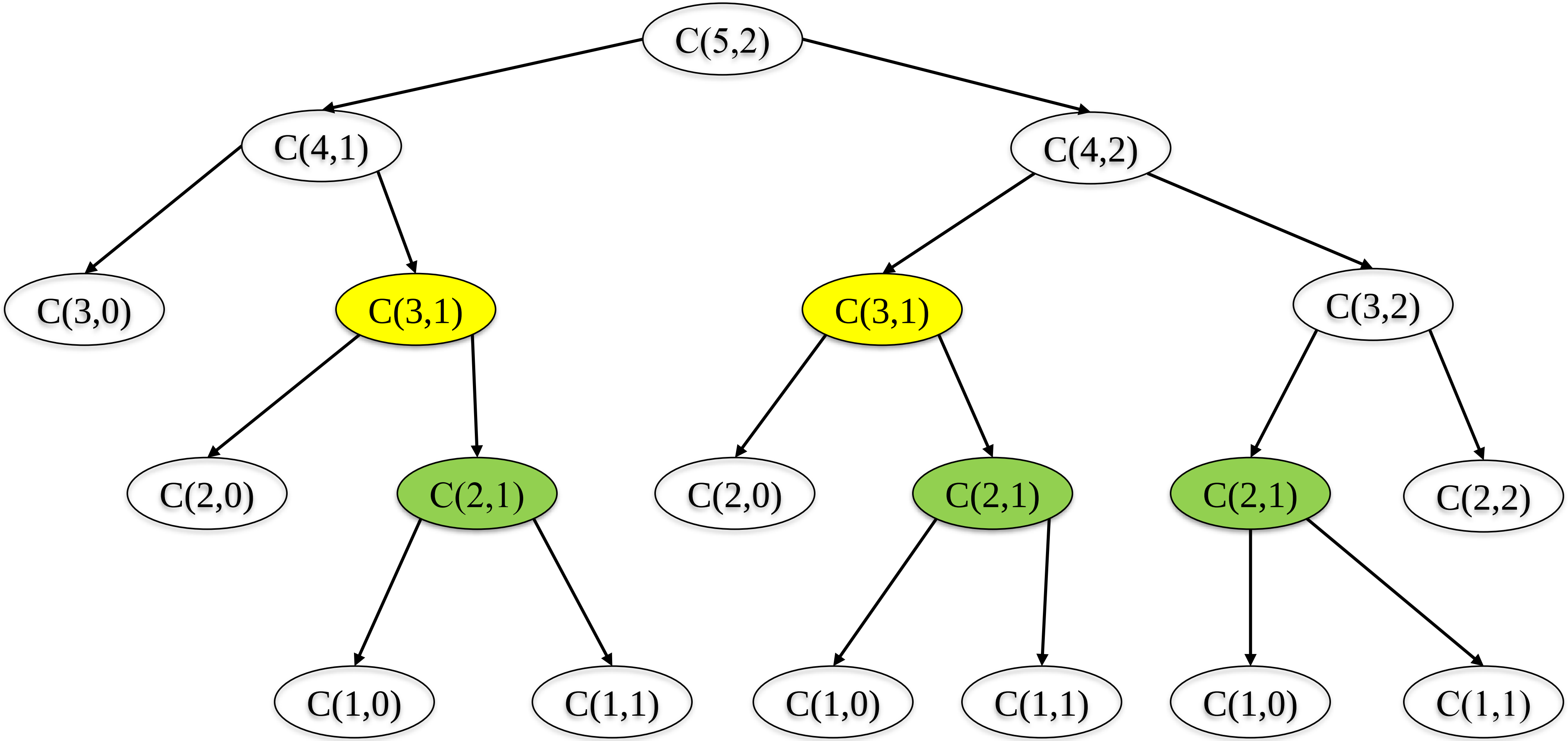
- If a dynamic programming solution is set up correctly, i.e., the recurrence equation is correct and the subproblems are always smaller than the original problem, then the resulting algorithm will always find the optimal solution in polynomial time.

- 0-1 knapsack problem can be solved using dynamic programming in polynomial time.

Binomial Coefficient

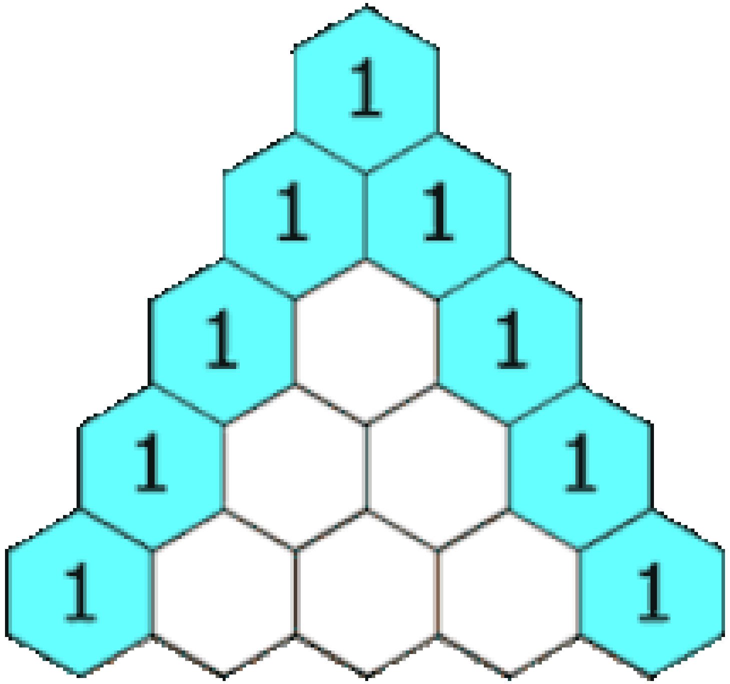
- Consider using two different methods, recursive computation and dynamic programming, to compute "*n* choose *k*" (or the more familiar form of $\binom{n}{k}$), also known as *Binomial coefficient*.
- Analyze and compare the complexity in each case to each other. Note that your complexity in the dynamic programming approach must be as good as polynomial; specifically $O(n^2)$.

Recursive Tree



Do you know what we just calculated?

- What we just calculated was Khayyam-Pascal triangle!
  - $(a+b)^n \;=\; C(n,0)a^n \;+\; \ldots + \;C(n,k)a^{n-k}b^k \;+\; \ldots + \;C(n,n)b^n$



n	$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	$\binom{n}{4}$	$\binom{n}{5}$	$\binom{n}{6}$	$\binom{n}{7}$
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

Rod Cutting

- How to cut steel rods into pieces in order to maximize the revenue you can get?
- Each cut is free. Rod lengths are always an integral number of centimeters.
- **Input:** A length N and table of prices p_i, for $i = 1, 2, \ldots, N$.
- **Output:** The maximum revenue obtainable for rods whose lengths sum to N, computed as the sum of the prices for the individual rods.

Rope Cutting

- A rope has length of N units, where N is an integer.
- You are asked to cut the rope (**at least once**) into different smaller pieces of integer lengths so that the product of the lengths of those new smaller ropes is maximized.