Dynamic Programming

What is DP?

- Wikipedia definition: "method for solving complex problems by breaking them down into simpler subproblems"

Steps for Solving DP Problems

1. Define subproblems
2. Write down the recurrence that relates subproblems
3. Recognize and solve the base cases

- Each step is very important!

Greedy Algorithms

- Greedy algorithms focus on making the best local choice at each decision point.
- For example, a natural way to compute a shortest path from x to y might be to walk out of x, repeatedly following the cheapest edge until we get to y. WRONG!
- In the absence of a correctness proof greedy algorithms are very likely to fail.

Recall Divide & Conquer

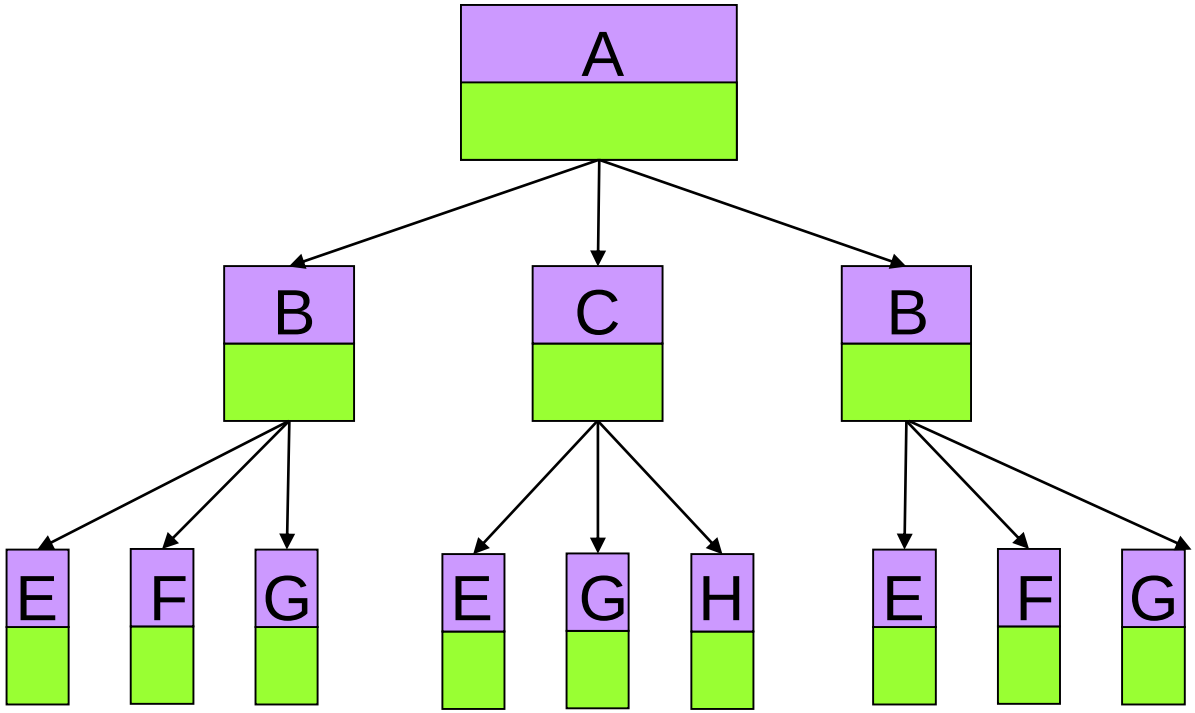
- Five pillars of Divide and Conquer algorithms:
  - How to divide into sub-problems?
  - Conquer through recursion
  - How to recombine sub-results?
  - When to stop dividing?
  - Analysis using Master Theorem or Recurrence Relations

Dynamic Programming: Computing View

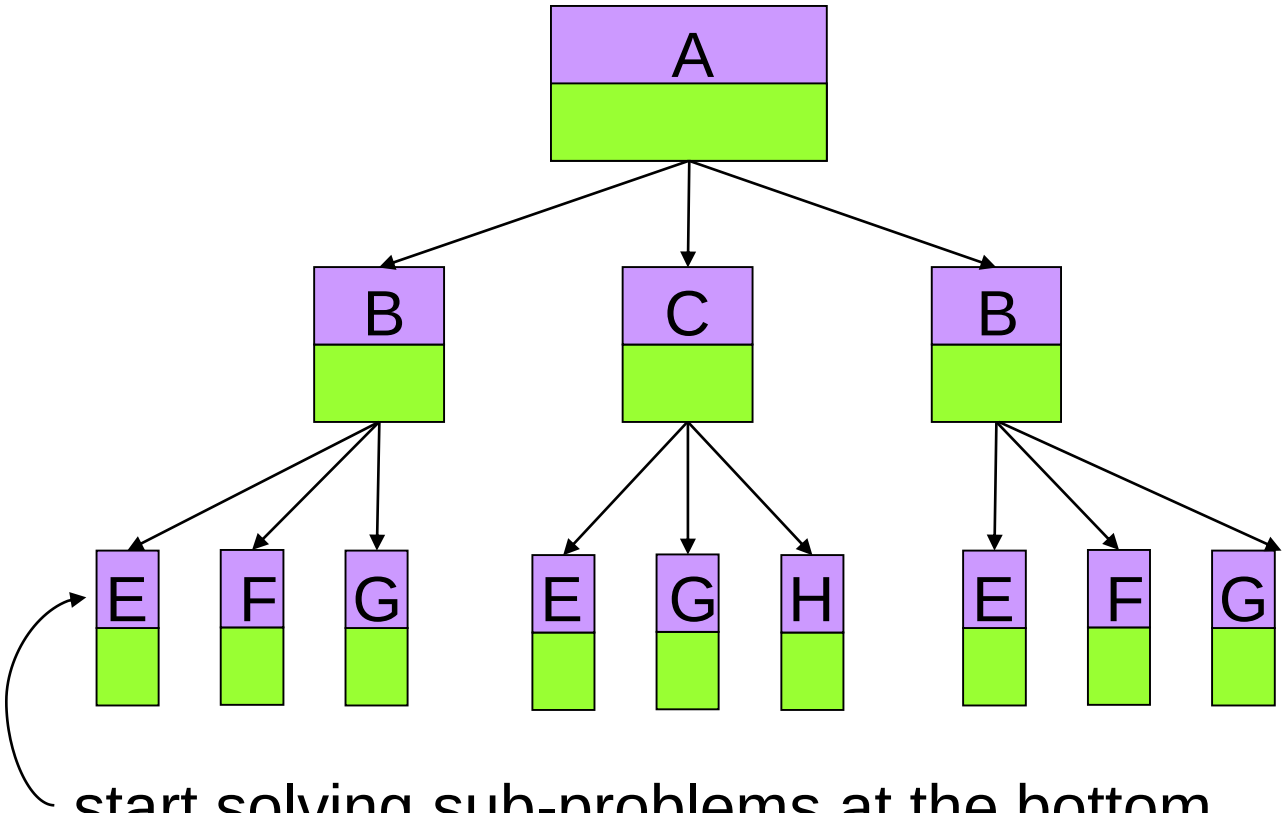
Not every sub-problem is new.

Save time: retain prior results.

Dynamic Programming: Computing View

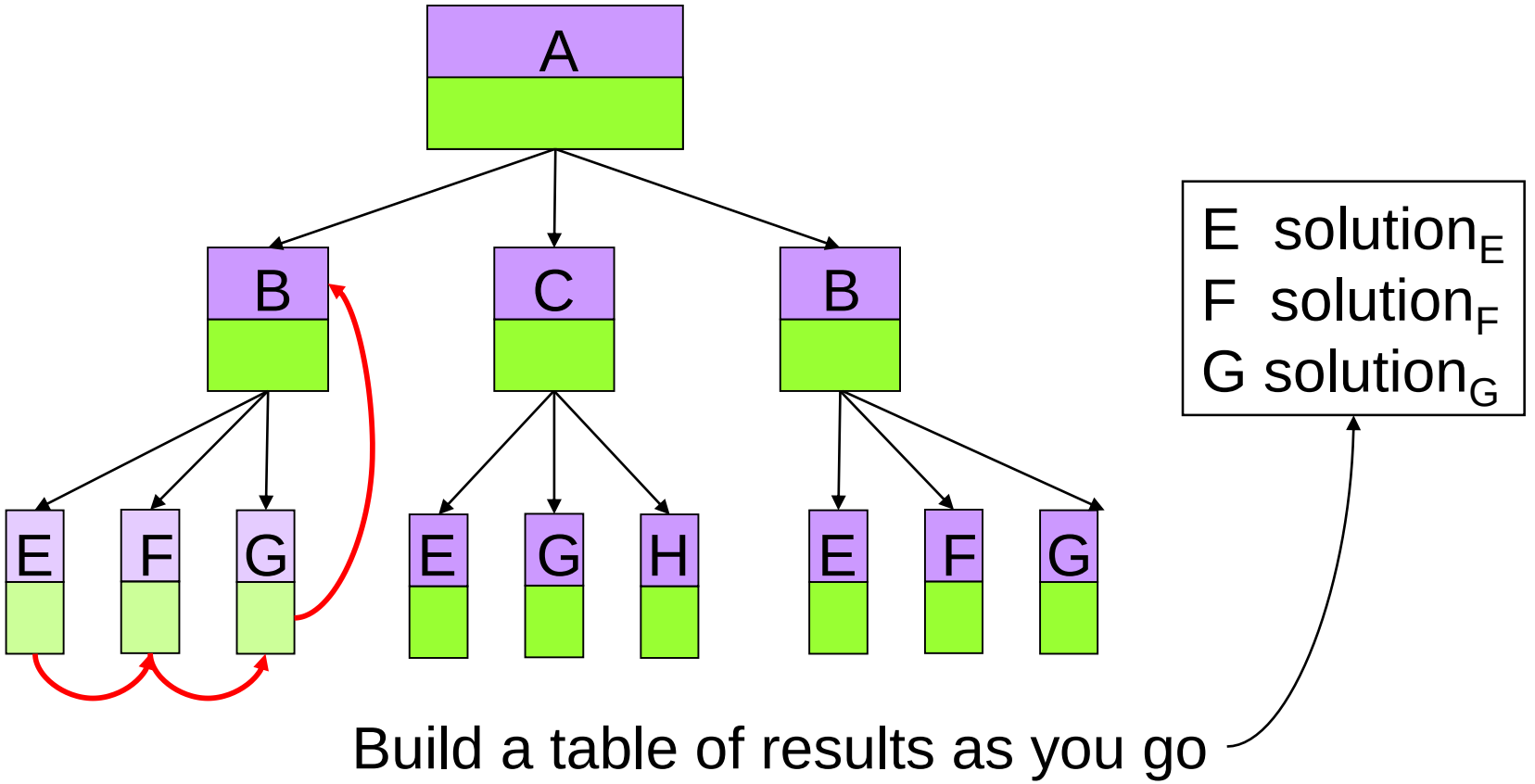


Dynamic Programming: Computing View

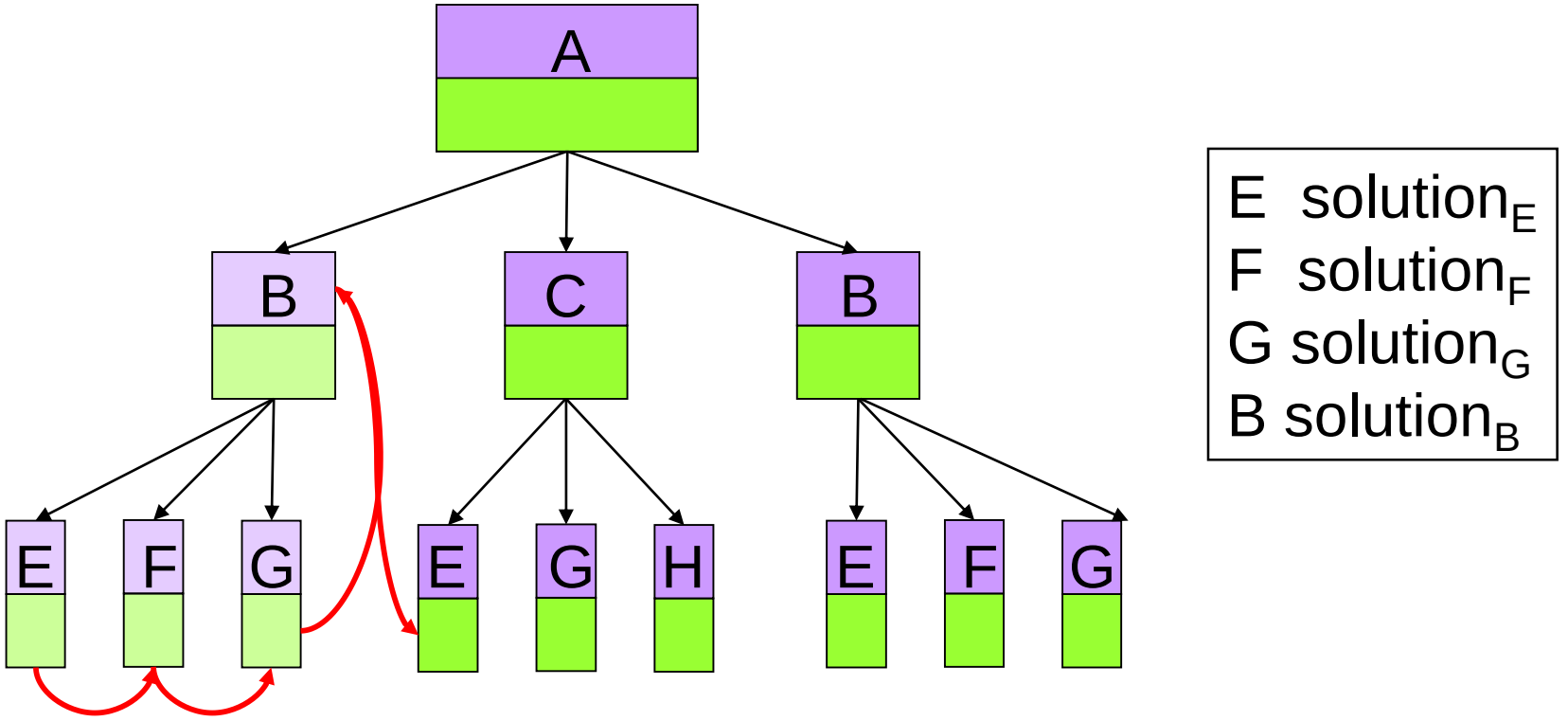


start solving sub-problems at the bottom.

Dynamic Programming: Computing View

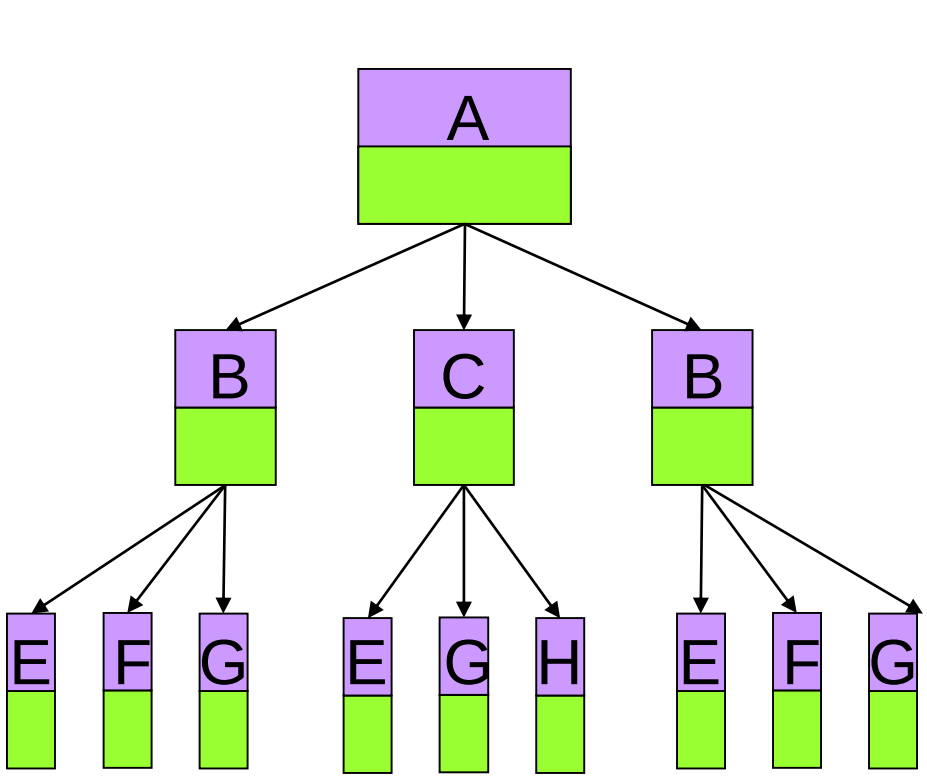


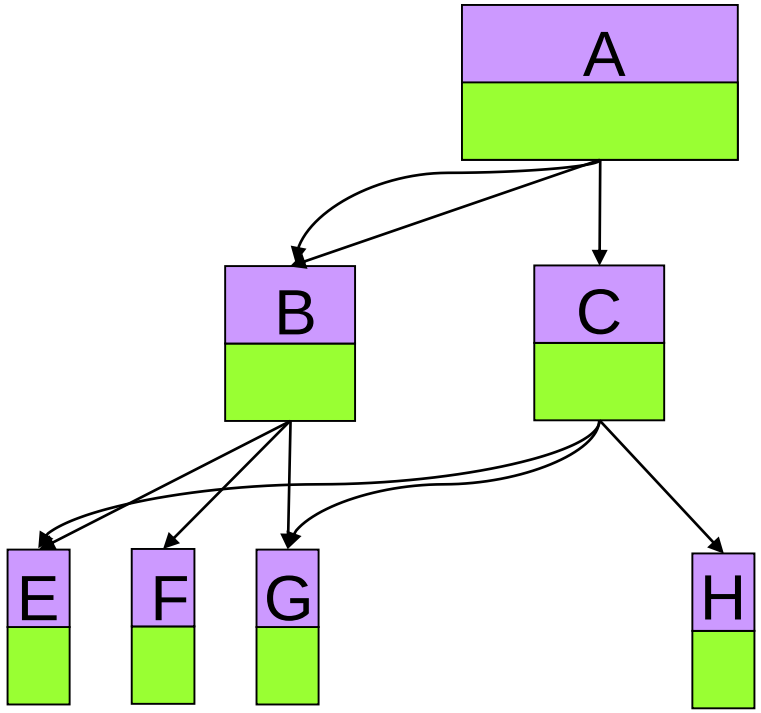
Dynamic Programming



Avoid recomputing intermediate results

Dynamic Programming:





Problem:

Let's consider the calculation of **Fibonacci** numbers:

F(n) = F(n-2) + F(n-1)

with seed values *F(1) = 1, F(2) = 1*

or *F(0) = 0, F(1) = 1*

What would a series look like:

0, 1, 1, 2, 3, 4, 5, 8, 13, 21, 34, 55, 89, 144, …

Recursive Algorithm:

```
Fib(n)
{
   if (n == 0)
      return 0;

   if (n == 1)
      return 1;

  Return Fib(n-1)+Fib(n-2)
}
```

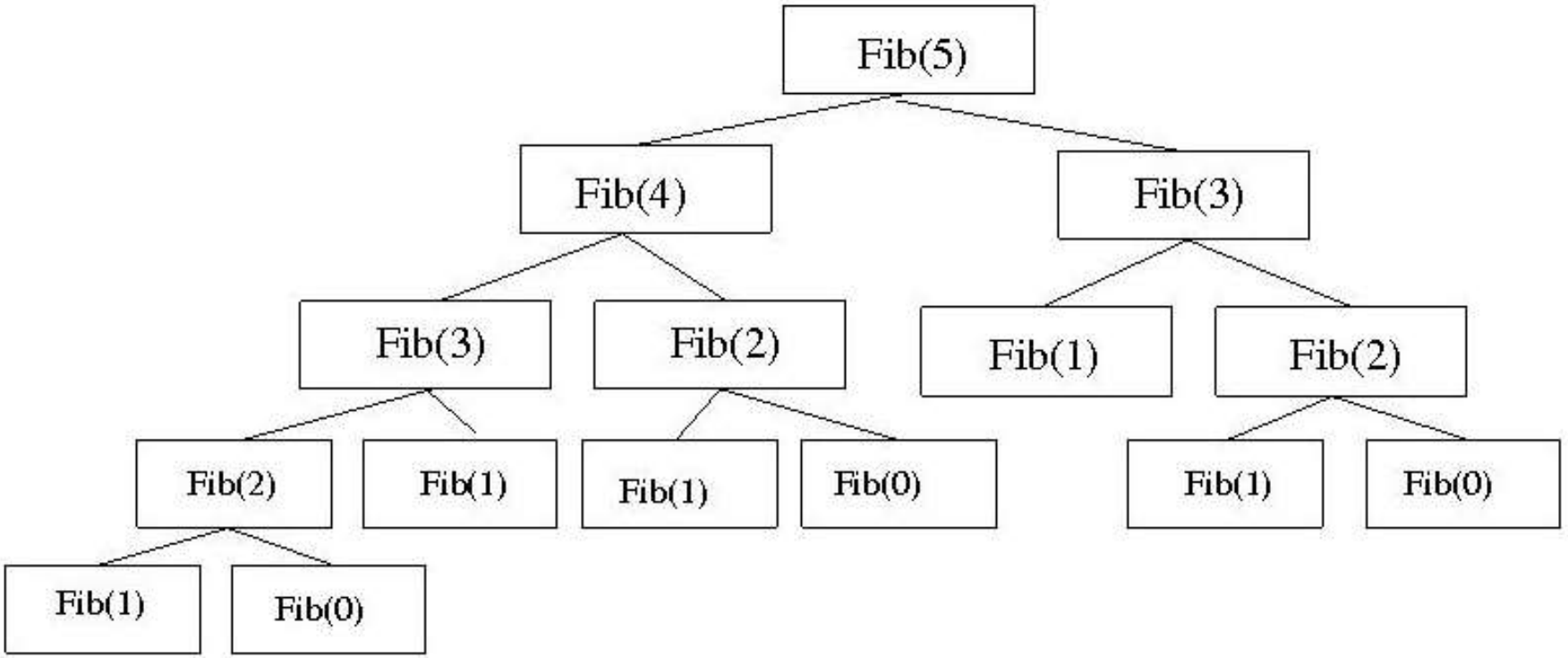
Recursive Algorithm:

```
Fib(n)
{
   if (n == 0)
      return 0;

   if (n == 1)
      return 1;

  Return Fib(n-1)+Fib(n-2)
}
```

It has a serious issue!

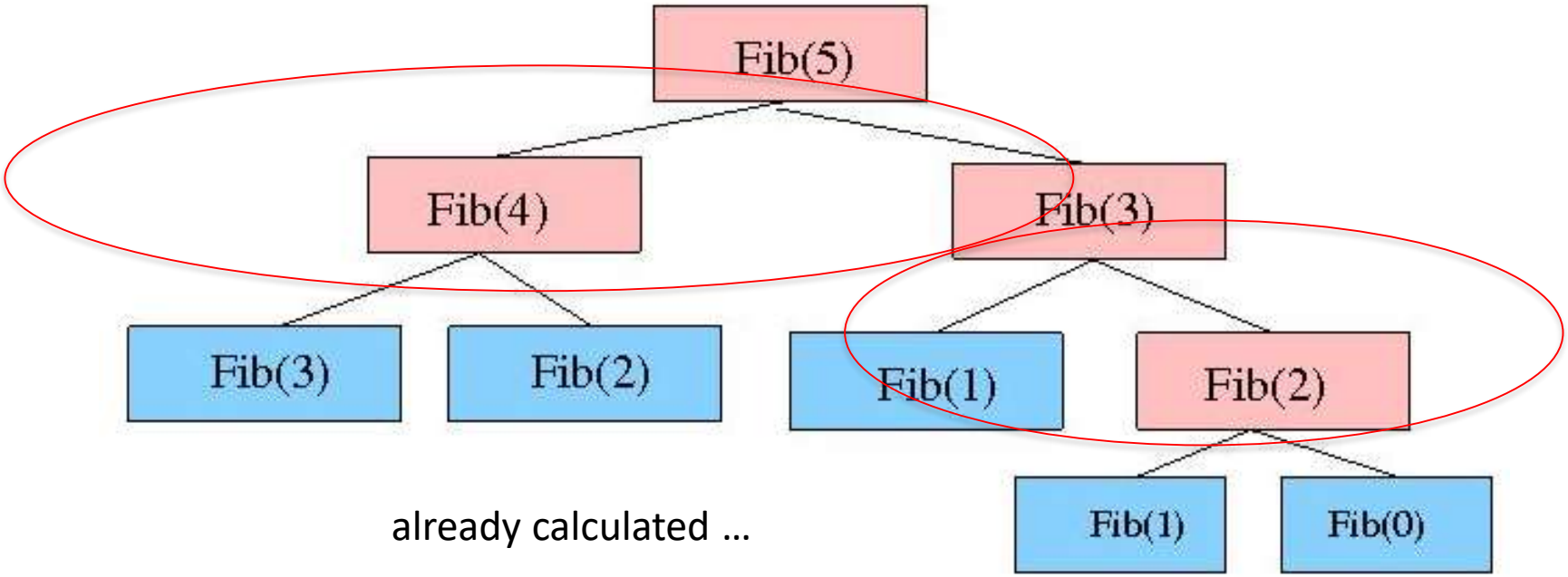
Recursion tree

What's the problem?



Memoization:

```
Fib(n)
{
   if (n == 0)
      return M[0];

   if (n == 1)
      return M[1];

   if (Fib(n-2) is not already calculated)
      call Fib(n-2);

   if(Fib(n-1) is already calculated)
      call Fib(n-1);

   //Store the ${n}^{th}$ Fibonacci no. in memory & use previous results.
   M[n] = M[n-1] + M[n-2]

   Return M[n];
}
```



already calculated …

Dynamic programming

- Main approach: recursive, holds answers to a sub problem in a table, can be used without recomputing.

- Can be formulated both via recursion and saving results in a table (*memoization*). Typically, we first formulate the recursive solution and then turn it into recursion plus dynamic programming via *memoization* or bottom-up.

-"*programming*" as in tabular not programming code

1-dimensional DP Problem

- Problem: given n, find the number of different ways to write n as the sum of 1, 3, 4
- Example: for $n = 5$, the answer is 6

$$
\begin{aligned}
5 &= 1+1+1+1+1 \\
&= 1+1+3 \\
&= 1+3+1 \\
&= 3+1+1 \\
&= 1+4 \\
&= 4+1
\end{aligned}
$$

1-dimensional DP Problem

- Define subproblems
  - Let D_n be the number of ways to write n as the sum of 1, 3, 4
- Find the recurrence
  - Consider one possible solution $n = x_1 + x_2 + \cdots + x_m$
  - If $x_m = 1$, the rest of the terms must sum to $n - 1$
  - Thus, the number of sums that end with $x_m = 1$ is equal to D_{n-1}
  - Take other cases into account $(x_m = 3,\ x_m = 4)$

1-dimensional DP Problem

- Recurrence is then

$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$

- Solve the base cases
  - $D_0 = 1$
  - $D_n = 0$ for all negative n
  - Alternatively, can set: $D_0 = D_1 = D_2 = 1$, and $D_3 = 2$

- We're basically done!

1-dimensional DP Problem

```
D[0] = D[1] = D[2] = 1; D[3] = 2;
for(i = 4; i <= n; i++)
    D[i] = D[i-1] + D[i-3] + D[i-4];
```

- Very short!

What happens when n is extremely large?

Extension: Solving this for huge n,
$n \approx 10^{12}$!

We have $D(n) = D(n-1) + D(n-3) + D(n-4)$, $n \geq 4$

$D(0) = D(1) = D(2) = 1$

$D(3) = 2$.

We can write

$$\begin{bmatrix} D(n) \\ D(n-1) \\ D(n-2) \\ D(n-3) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} D(n-1) \\ D(n-2) \\ D(n-3) \\ D(n-4) \end{bmatrix}$$

Let $A = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

$\therefore$ $\begin{bmatrix} D(n) \\ D(n-1) \\ D(n-2) \\ D(n-3) \end{bmatrix} = A \begin{bmatrix} D(n-1) \\ D(n-2) \\ D(n-3) \\ D(n-4) \end{bmatrix}$ —(1)

We can write again

$$\begin{bmatrix} D(n-1) \\ D(n-2) \\ D(n-3) \\ D(n-4) \end{bmatrix} = A \begin{bmatrix} D(n-2) \\ D(n-3) \\ D(n-4) \\ D(n-5 \end{bmatrix}$$

$\therefore$ (1) becomes

$$\begin{bmatrix} D(n) \\ D(n-2) \\ D(n-3) \\ D(n-4) \end{bmatrix} = A^2 \begin{bmatrix} D(n-2) \\ D(n-3) \\ D(n-4) \\ D(n-5) \end{bmatrix} = \cdots$$

$$= A^{n-3} \begin{bmatrix} D(3) \\ D(2) \\ D(1) \\ D(0) \end{bmatrix}$$

Need to evalnate A^{n-3}

Evalnate (A^k)

Step 1 : if $k=1$ return A^k

Step 2 : Compute $B = A^{\lfloor k/2 \rfloor}$

Step 3 : If k is even return B^2

else return $B^2 . A$

Total cost : $O(\log n)$. ~~[illegible]~~ #

Logarithmic time

Dynamic Programming

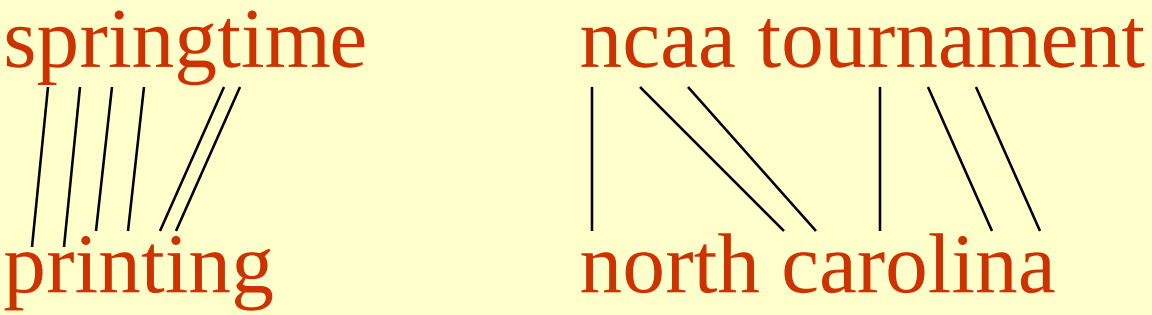
- Dynamic Programming is an algorithm design technique for optimization problems: often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent.
  - Subproblems may share subsubproblems,
  - However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)
- DP reduces computation by
  - Solving subproblems in a bottom-up fashion.
  - Storing solution to a subproblem the first time it is solved.
  - Looking up the solution when subproblem is encountered again.
- Key: determine structure of optimal solutions

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

Longest Common Subsequence

- ***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.

Naïve Algorithm

- For every subsequence of X, check whether it's a subsequence of Y .
- Time: $\Theta(n2^m)$.
  - 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.

Optimal Substructure

Theorem

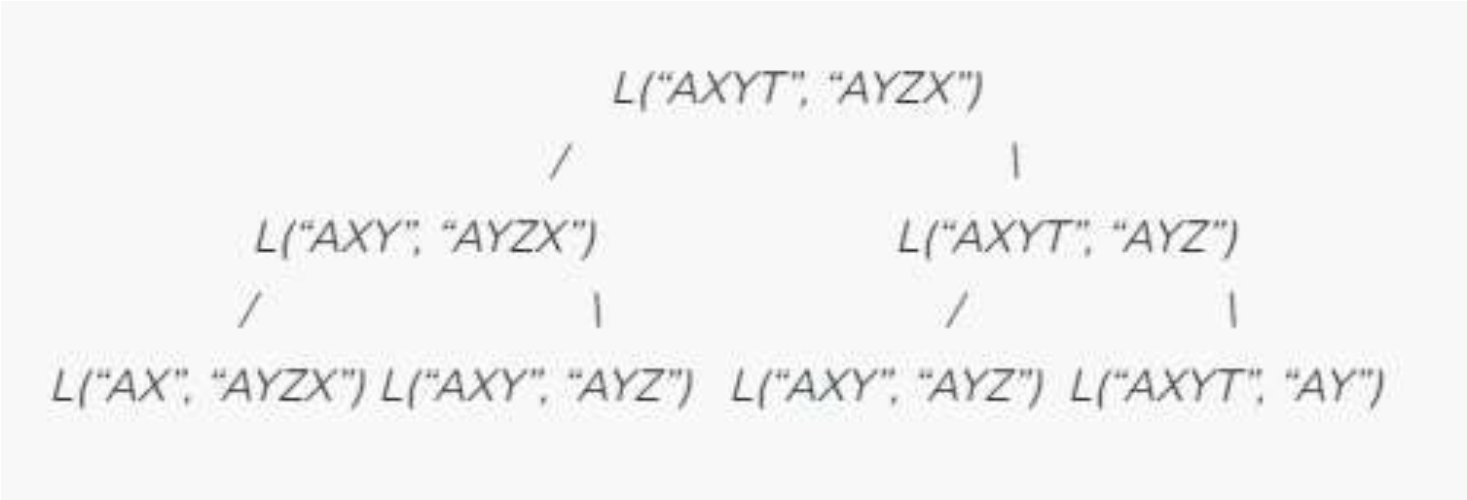
Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y .

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 $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y .
3. or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Notation:

prefix $X_i = \langle x_1, \ldots, x_i \rangle$ is the first i letters of X.

Overlapping Subproblem



Recursive Solution

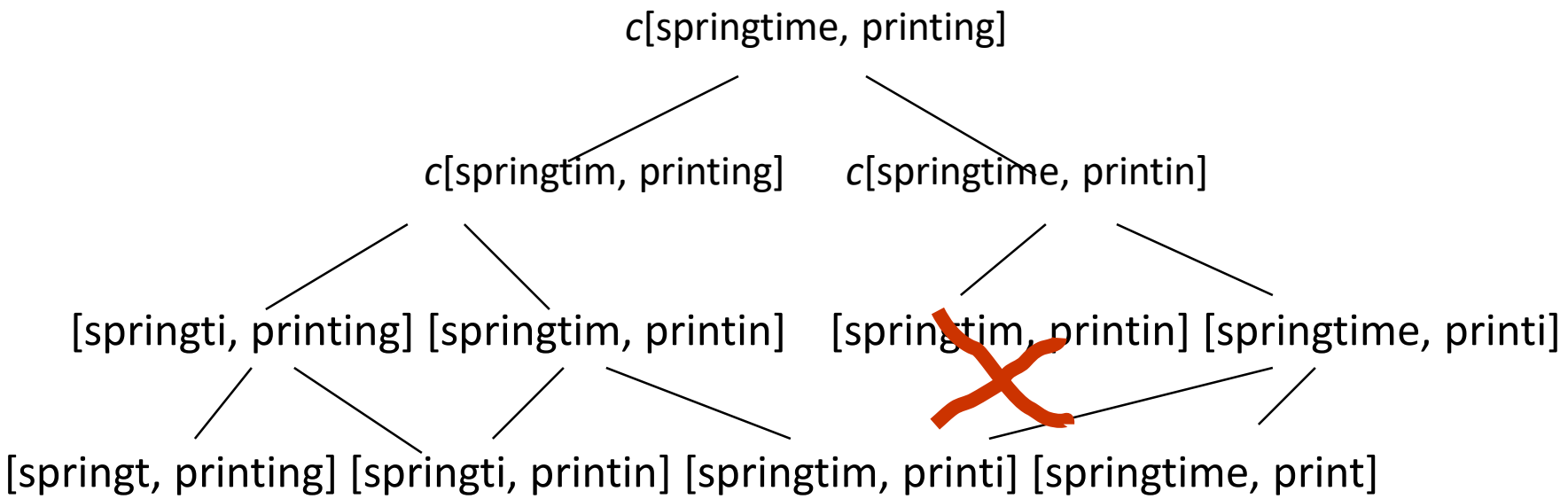
- Define $c[i, j]$ = length of LCS of X_i and Y_j .
- We want $c[m,n]$.

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

This gives a recursive algorithm and solves the problem.
But does it solve it well?

Recursive Solution

$$c[\alpha,\beta] = \begin{cases} 0 & \text{if } \alpha \text{ empty or } \beta \text{ empty}, \\ c[\,prefix\alpha, prefix\beta\,]+1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\ \max(\,c[\,prefix\alpha, \beta\,], c[\alpha, prefix\beta\,]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta). \end{cases}$$



Recursive Solution

$$c[\alpha,\beta] = \begin{cases} 0 & \text{if } \alpha \text{ empty or } \beta \text{ empty}, \\ c[\,prefix\alpha,\, prefix\beta\,]+1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\ \max(\,c[\,prefix\alpha,\,\beta\,], c[\,\alpha,\, prefix\beta\,]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta). \end{cases}$$

- Keep track of $c[\alpha,\beta]$ in a table of nm entries:
  - top/down
  - bottom/up

Computing the length of an LCS

LCS-LENGTH (*X*, *Y*)

```
1.  m ← length[X]
2.  n ← length[Y]
3.  for i ← 1 to m
4.      do c[i, 0] ← 0
5.  for j ← 0 to n
6.      do c[0, j ] ← 0
7.  for i ← 1 to m
8.      do for j ← 1 to n
9.          do if x_i = y_j
10.                then c[i, j ] ← c[i–1, j–1] + 1
11.                     b[i, j ] ← "↖"
12.                else if c[i–1, j ] ≥ c[i, j–1]
13.                     then c[i, j ] ← c[i – 1, j ]
14.                          b[i, j ] ← "↑"
15.                     else c[i, j ] ← c[i, j–1]
16.                          b[i, j ] ← "←"
17. return c and b
```

Time: $O(mn)$

$b[i, j]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j.

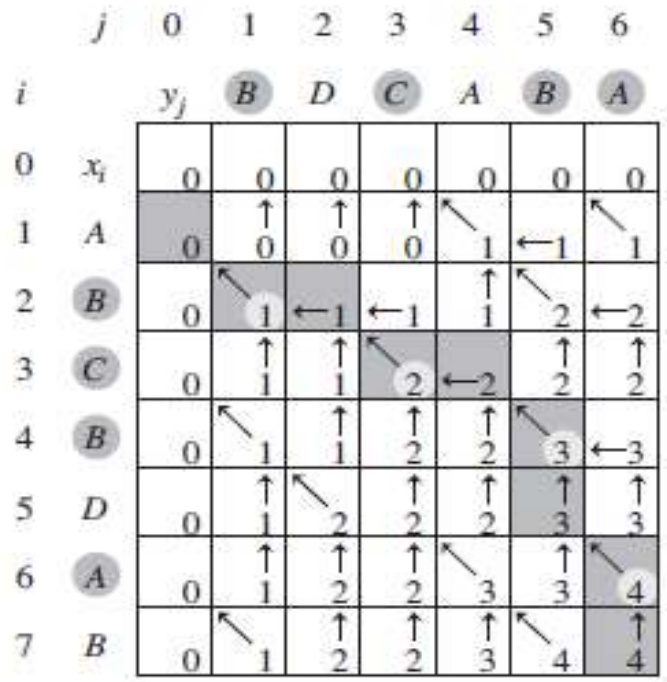
$c[m,n]$ contains the length of an LCS of X and Y.

		0	1	2	3	4	5	6
		y_i	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						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Constructing an LCS

PRINT-LCS (*b, X, i, j*)

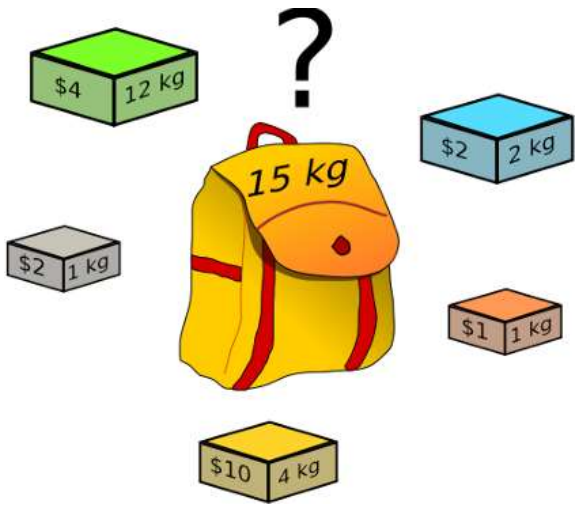
1. **if** $i = 0$ or $j = 0$
2. **then return**
3. **if** $b[i, j] =$ "↖"
4. **then** PRINT-LCS(*b, X, i*–1, *j*–1)
5. print x_i
6. **elseif** $b[i, j] =$ "↑"
7. **then** PRINT-LCS(*b, X, i*–1, *j*)
8. **else** PRINT-LCS(*b, X, i, j*–1)



- Initial call is PRINT-LCS (*b, X,m, n*).
- When $b[i, j] =$ ↖, we have extended LCS by one character. So LCS = entries with ↖ in them.
- Time: $O(m+n)$

Knapsack Problem

- The knapsack problem can be posed as follows.
  - A thief robbing a store finds n items : the i-th item is worth p_i and weight w_i. The thief would take as valuable as possible, but he can carry at most W weights in his knapsack. Which items should the thief take?



Knapsack Problem

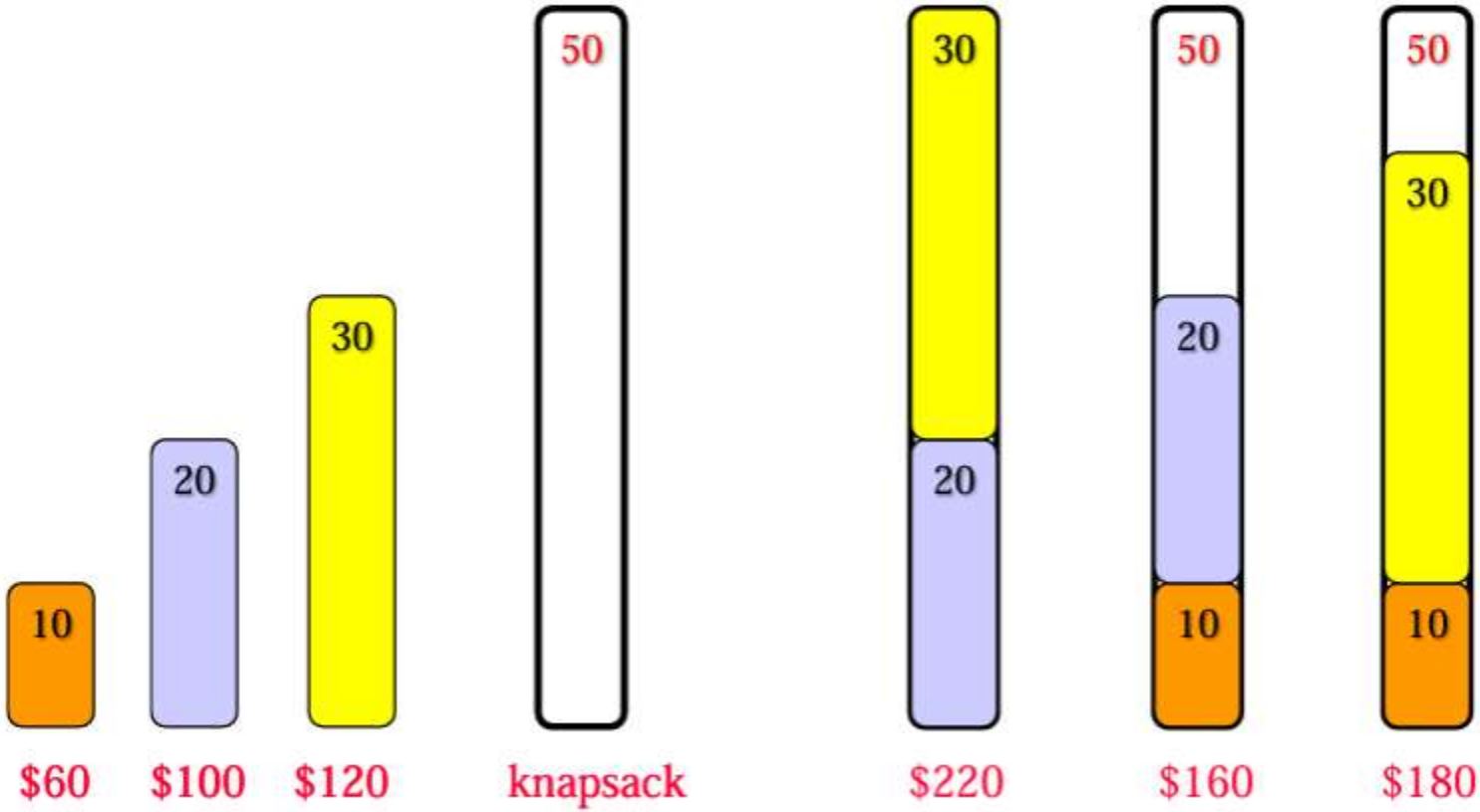
- Given some items, pack the knapsack to get the maximum total value.
- Each item has some weight and some value.
- Total weight that we can carry is no more than some fixed number W.
- So we must consider weights of items as well as their value.

Item #	Weight	Value
1	12	$4
2	1	$2
3	4	$10
4	1	$1
5	2	$2

0/1 Knapsack problem

- It's called "0/1" because each item is either all in, or all out. There is no possibility to put some portion of the item in the knapsack.

The Greedy method does not work for the 0/1 Knapsack Problem!



0/1 Knapsack problem

- Let $KNAP(1, n, W)$ denote the 0/1 Knapsack problem, choosing objects from $[1 \dots n]$ under the capacity constraint of W.
- If $(x_1, x_2, \dots, x_n)$ is an optimal solution for the problem $KNAP(1, n, W)$, then:
  1. If $x_n = 0$ (we do not pick the n-th object), then $(x_1, x_2, \dots, x_{n-1})$ must be an optimal solution for the problem $KNAP(1, n-1, W)$.
  2. If $x_n = 1$ (we pick the n-th object), then $(x_1, x_2, \dots, x_{n-1})$ must be an optimal solution for the problem $KNAP(1, n-1, W - w_n)$.

Solution in terms of subproblems

- Based on the optimal substructure, we can write down the solution for the 0/1 Knapsack problem as follows:
- Let $C[n, W]$ be the value (total profits) of the optimal solution for $KNAP(1, n, W)$.

- $C[n, W] = \max(\, profits\ for\ case\ 1, profits\ for\ case\ 2)$
  $= \max(\, C[n-1, W], C[n-1, W - w_n] + p_n)$.

Solution in terms of subproblems

- Recursive formula for subproblems

$$C[k,w] = \begin{cases} C[k-1,w] & if\ w_k > w \\ \max(\ C[n-1,w], C[n-1,w-\ w_n] +\ p_n)\ , else \end{cases}$$

- For example, if $n\ =\ 4, W\ =\ 9;\ w_4\ =\ 4,$ $p_4\ =\ 2$, then
- $C[4,9]\ =\ \max(\ C[3,9], C[3,9-\ 4]\ +\ 2).$

0-1 Knapsack DP Algorithm

for w = 0 to W
    C[0,w] = 0
for k = 0 to n
    C[k,0] = 0
    for w = 0 to W
        if $w_k \leq w$ // item k can be part of the solution
            if $p_k + C[k-1, w-w_k] > C[k-1, w]$
            $C[k, w] = p_k + C[k-1, w-w_k]$
        else
            $C[k, w] = C[k-1, w]$
        else
            $C[k, w] = C[k-1, w]$ // $wk > w$

Example

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (3)

```
for w = 0 to W
    C[0,w] = 0
for k = 0 to n
    C[k,0] = 0
    for w = 0 to W
        if w_k ≤ w // item k can be part of the solution
            if p_k + C[k − 1, w − w_k] > C[k − 1, w]
                C[k, w] = p_k + C[k − 1, w − w_k]
            else
                C[k, w] = C[k − 1, w]
        else
            C[k, w] = C[k − 1, w] // wk > w
```

i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

Example

i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Matrix-chain Multiplication

- Suppose we have a sequence or chain A_1, A_2, …, A_n of *n* matrices to be multiplied
  - That is, we want to compute the product $A_1A_2…A_n$

- There are many possible ways (parenthesizations) to compute the product

Matrix-chain Multiplication …contd

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:
  1. $(A_1(A_2(A_3A_4)))$
  2. $(A_1((A_2A_3)A_4))$
  3. $((A_1A_2)(A_3A_4))$
  4. $((A_1(A_2A_3))A_4)$
  5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication …contd

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p\times q}$ and $B_{q\times r}$ (with dimensions $p\times q$ and $q\times r$)

Result: Matrix $C_{p\times r}$ resulting from the product $A\cdot B$

MATRIX-MULTIPLY$(A_{p\times q}, B_{q\times r})$

```
1.  for i ← 1 to p
2.      for j ← 1 to r
3.          C[i, j] ← 0
4.          for k ← 1 to q
5.              C[i, j] ← C[i, j] + A[i, k] · B[k, j]
6.  return C
```

Scalar multiplication in line 5 dominates time to compute CNumber of scalar multiplications = *pqr*

Matrix-chain Multiplication …contd

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 = 2{,}500$ scalar multiplications

    } Total: 7,500
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 = 50{,}000$ scalar multiplications

    } Total: 75,000

Matrix-chain Multiplication …contd

- Matrix-chain multiplication problem
  - Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for $i=1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$
  - Parenthesize the product $A_1 A_2 \ldots A_n$ such that the total number of scalar multiplications is minimized
- Brute force method of exhaustive search takes time exponential in n

Dynamic Programming Approach

- The structure of an optimal solution
  - Let us use the notation $A_{i..j}$ for the matrix that results from the product $A_i A_{i+1} \ldots A_j$
  - An optimal parenthesization of the product $A_1A_2\ldots A_n$ splits the product between A_k and A_{k+1} for some integer k where $1 \leq k < n$
  - First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$

Dynamic Programming Approach …contd

- **Key observation**: parenthesizations of the subchains $A_1A_2…A_k$ and $A_{k+1}A_{k+2}…A_n$ must also be optimal if the parenthesization of the chain $A_1A_2…A_n$ is optimal (why?)

- That is, the optimal solution to the problem contains within it the optimal solution to subproblems

Dynamic Programming Approach …contd

- Recursive definition of the value of an optimal solution
  - Let $m[i, j]$ be the minimum number of scalar multiplications necessary to compute $A_{i..j}$
  - Minimum cost to compute $A_{1..n}$ is $m[1, n]$
  - Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1} for some integer k where $i \leq k < j$

Dynamic Programming Approach …contd

- $A_{i..j} = (A_i A_{i+1} \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_j) = A_{i..k} \cdot A_{k+1..j}$
- Cost of computing $A_{i..j}$ = cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying $A_{i..k}$ and $A_{k+1..j}$
- Cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ is $p_{i-1} p_k p_j$

- $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$ for $i \leq k < j$
- $m[i, i] = 0$ for $i=1,2,\ldots,n$

Dynamic Programming Approach …contd

- But… optimal parenthesization occurs at one value of k among all possible $i \le k < j$
- Check all these and select the best one

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

Dynamic Programming Approach …contd

- To keep track of how to construct an optimal solution, we use a table *C*
- *C*[*i*, *j*] = value of *k* at which $A_i A_{i+1} \ldots A_j$ is split for optimal parenthesization
- Algorithm: next slide
  - First computes costs for chains of length l=1
  - Then for chains of length l=2,3, … and so on
  - Computes the optimal cost bottom-up

Algorithm to Compute Optimal Cost

Input: Array $p[0\ldots n]$ containing matrix dimensions and n

Result: Minimum-cost table C and split table s

MATRIX-CHAIN-ORDER($p[\,]$, n)

```
for i ← 1 to n
    C[i, i] ← 0
for l ← 2 to n
    for i ← 1 to n-l+1
        j ← i+l-1
        C[i, j] ← ∞
        for k ← i to j-1
            q ← C[i, k] + C[k+1, j] + p[i-1] p[k] p[j]
            if q < C[i, j]
                C[i, j] ← q
                s[i, j] ← k
return m and s
```

Takes $O(n^3)$ time

Requires $O(n^2)$ space

Matrix chain multiplication

- The algorithm takes $O(n^3)$ time and requires $O(n^2)$ space.
- Example:
- Consider the following six matrix problem.

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimensions	10x20	20x5	5x15	15x50	50x10	10x15

- The problem therefore can be phrased as one of filling in the following table representing the values m.

i\j	1	2	3	4	5	6
1	0					
2		0				
3			0			
4				0		
5					0	
6						0

Matrix chain multiplication

- Chains of length 2 are easy, as there is no minimization required, so
- $m[i, i+1] = p_{i-1}p_ip_{i+1}$
- m[1, 2] = 10x20x5 = 1000
- m[2, 3] = 20x5x15 = 1500
- m[3, 4] = 5x15x50 = 3750
- m[4, 5] = 15x50x10 = 7500
- m[5, 6] = 50x10x15 = 7500

i\j	1	2	3	4	5	6
1	0	1000				
2		0	1500			
3			0	3750		
4				0	7500	
5					0	7500
6						0

Matrix chain multiplication

- Chains of length 3 require some minimization – but only one each.

- $m[1,3]=\min\{(m[1,1]+m[2,3]+p_0p_1p_3),(m[1,2]+m[3,3]+p_0p_2p_3)\}$
  $= \min\{(0+1500+10x20x15), (1000+0+10x5x15)\}$
  $= \min\{ 4500, 1750 \} = 1750$
- $m[2,4]=\min\{(m[2,2]+m[3,4]+p_1p_2p_4),(m[2,3]+m[4,4]+p_1p_3p_4)\}$
  $= \min\{(0+3750+20x5x50), (1500+0+20x15x50)\}$
  $= \min\{ 8750, 16500 \} = 8750$
- $m[3,5]=\min\{(m[3,3]+m[4,5]+p_2p_3p_5),(m[3,4]+m[5,5]+p_2p_4p_5)\}$
  $= \min\{(0+7500+5x15x10), (3750+0+5x50x10)\}$
  $= \min\{ 8250, 6250 \} = 6250$
- $m[4,6]=\min\{(m[4,4]+m[5,6]+p_3p_4p_6),(m[4,5]+m[6,6]+p_3p_5p_6)\}$
  $= \min\{(0+7500+15x50x15), (7500+0+15x10x15)\}$
  $= \min\{ 18750, 9750 \} = 9750$

Matrix chain multiplication

i\j	1	2	3	4	5	6
1	0	1000	1750			
2		0	1500	8750		
3			0	3750	6250	
4				0	7500	9750
5					0	7500
6						0

Matrix chain multiplication

- $m[1,4]=\min\{(m[1,1]+m[2,4]+p_0p_1p_4),(m[1,2]+m[3,4]+p_0p_2p_4),$
  $(m[1,3]+m[4,4]+p_0p_3p_4)\}$
  $= \min\{(0+8750+10x20x50), (1000+3750+10x5x50),$
  $(1750+0+10x15x50)\}$
  $= \min\{ 18750, 7250, 9250 \} = 7250$
- $m[2,5]=\min\{(m[2,2]+m[3,5]+p_1p_2p_5),(m[2,3]+m[4,5]+p_1p_3p_5),$
  $(m[2,4]+m[5,5]+p_1p_4p_5)\}$
  $= \min\{(0+6250+20x5x10), (1500+7500+20x15x10),$
  $(8750+0+20x50x10)\}$
  $= \min\{ 7250, 12000, 18750 \} = 7250$
- $m[3,6]=\min\{(m[3,3]+m[4,6]+p_2p_3p_6),(m[3,4]+m[5,6]+p_2p_4p_6),$
  $(m[3,5]+m[6,6]+p_2p_5p_6)\}$
  $= \min\{(0+9750+5x15x15), (3750+7500+5x50x15),$
  $(6250+0+5x10x15)\}$
  $= \min\{ 10875, 15000, 7000 \} = 7000$

Matrix chain multiplication

i\j	1	2	3	4	5	6
1	0	1000	1750	7250		
2		0	1500	8750	7250	
3			0	3750	6250	7000
4				0	7500	9750
5					0	7500
6						0

Matrix chain multiplication

- $m[1,5]=\min\{(m[1,1]+m[2,5]+p_0p_1p_5),(m[1,2]+m[3,5]+p_0p_2p_5),$
  $(m[1,3]+m[4,5]+p_0p_3p_5),(m[1,4]+m[5,5]+p_0p_4p_5)\}$
  $= \min\{(0+7250+10x20x10), (1000+6250+10x5x10),$
  $(1750+7500+10x15x10), (7250+0+10x50x10)\}$
  $= \min \{ 9250, 7750, 10750, 12250 \} = 7750$
- $m[2,6]=\min\{(m[2,2]+m[3,6]+p_1p_2p_6),(m[2,3]+m[4,6]+p_1p_3p_6),$
  $(m[2,4]+m[5,6]+p_1p_4p_6),(m[2,5]+m[6,6]+p_1p_5p_6)\}$
  $= \min\{(0+7000+20x5x15), (1500+9750+20x15x15),$
  $(8750+7500+20x50x15), (7250+0+20x10x15)\}$
  $= \min \{ 8500, 15750, 31,250, 10250 \} = 8500$
- $m[1,6]=\min\{(m[1,1]+m[2,6]+p_0p_1p_6),(m[1,2]+m[3,6]+p_0p_2p_6),$
  $(m[1,3]+m[4,6]+p_0p_3p_6),(m[1,4]+m[5,6]+p_0p_4p_6),$
  $(m[1,5]+m[6,6]+p_0p_5p_6)\}$
  $= \min\{(11500, 8750, 13750, 22250, 9250 \} = 8750$

Matrix chain multiplication

i\j	1	2	3	4	5	6
1	0	1000	1750	7250	7750	8750
2		0	1500	8750	7250	8500
3			0	3750	6250	7000
4				0	7500	9750
5					0	7500
6						0

Matrix chain multiplication

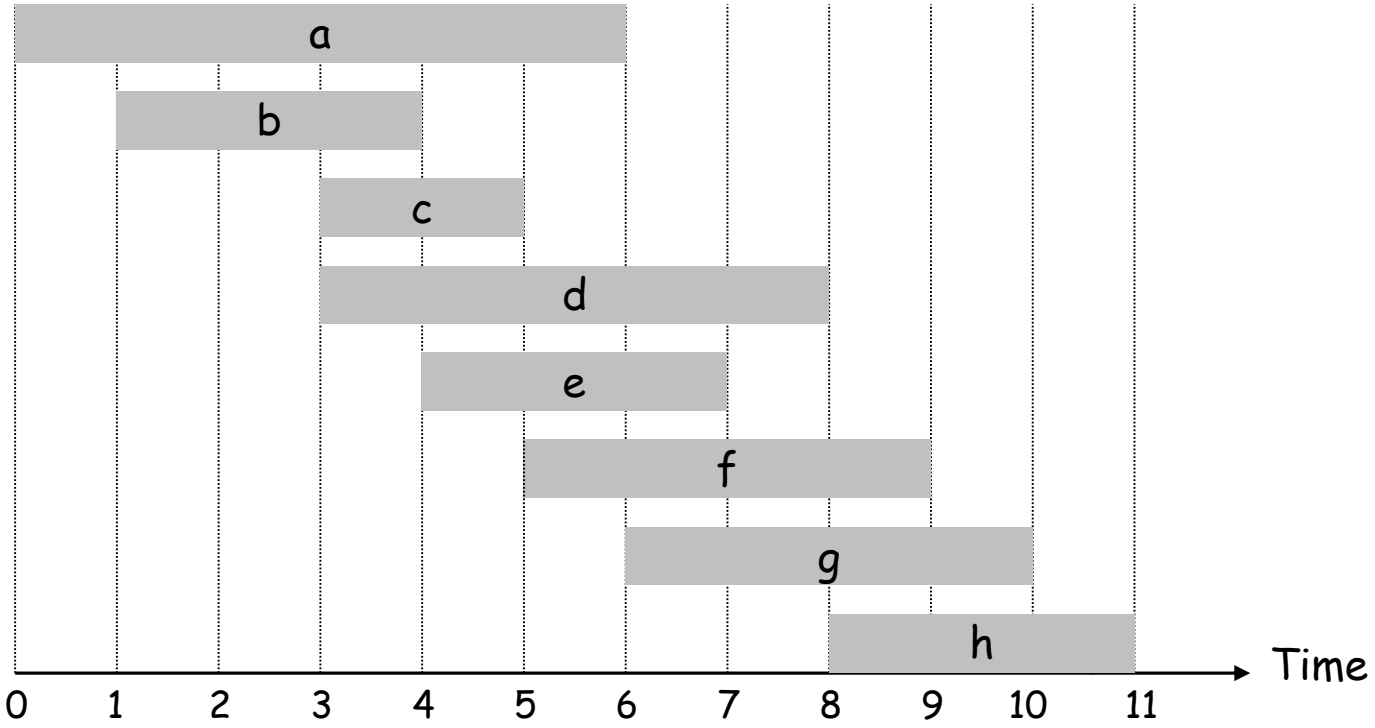
```
Algorithm PrintOptimalPerens(s, i, j)
    {   if (i=j) then
            Print "A"i;
        else
        {   Print "(";
            PrintOptimalPerens(s, i, s[i,j]);
            PrintOptimalPerens(s, s[i,j]+1, j);
            Print ")";
        }
    }
```

i\j	1	2	3	4	5	6
1	1	1	2	2	2	2
2		2	2	2	2	2
3			3	3	4	5
4				4	4	5
5					5	5
6						6

Weighted Interval Scheduling

Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight or value v_j .
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

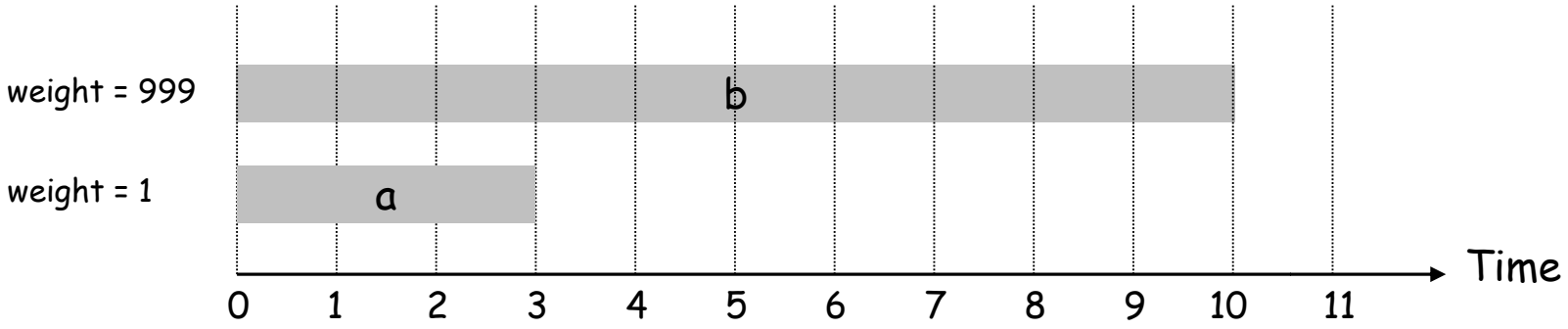


Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.



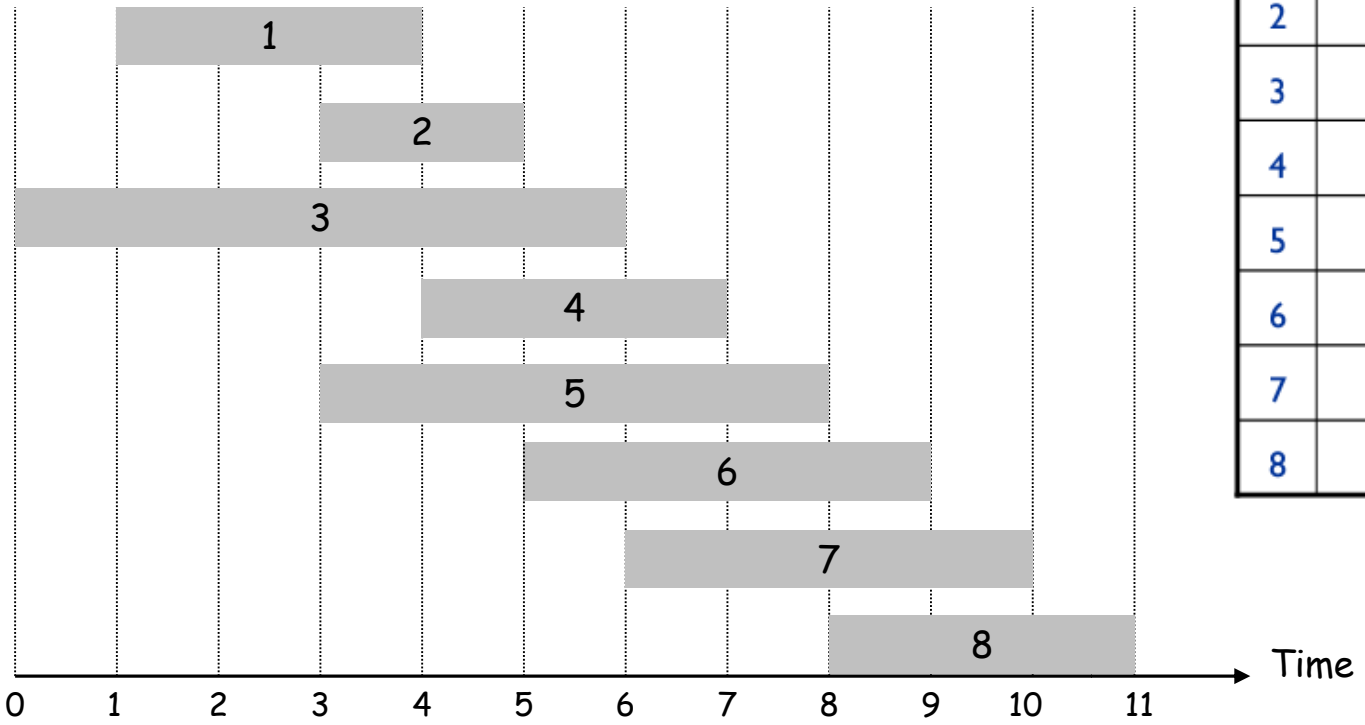
Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. p(j) = largest index i < j such that job i is compatible with job j.

Ex: p(8) = 5, p(7) = 3, p(2) = 0.

j	vj	pj	optj
0	-	-	0
1		0	
2		0	
3		0	
4		1	
5		0	
6		2	
7		3	
8		5	



Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT selects job j.
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., **p(j)**

optimal substructure

- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., **j-1**

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{ v_j + OPT(p(j)),\ OPT(j-1) \} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Bottom-Up

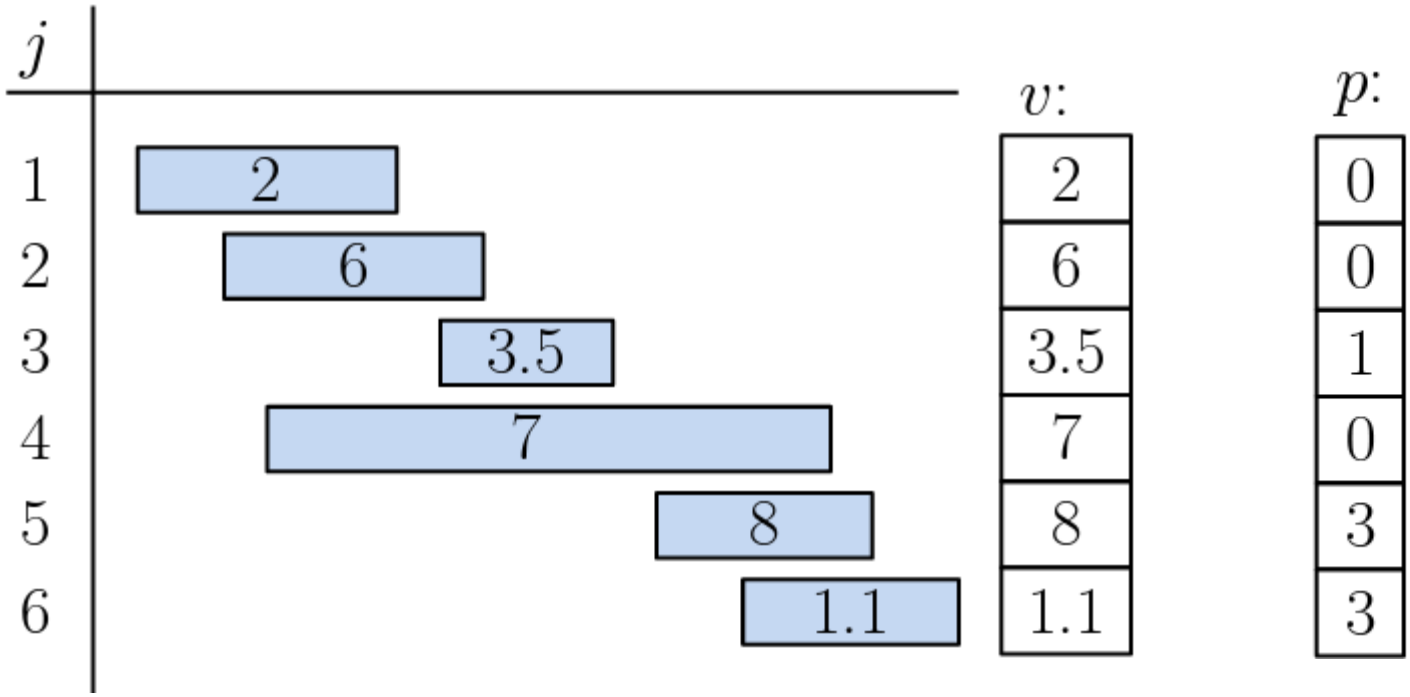
Input: $n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$

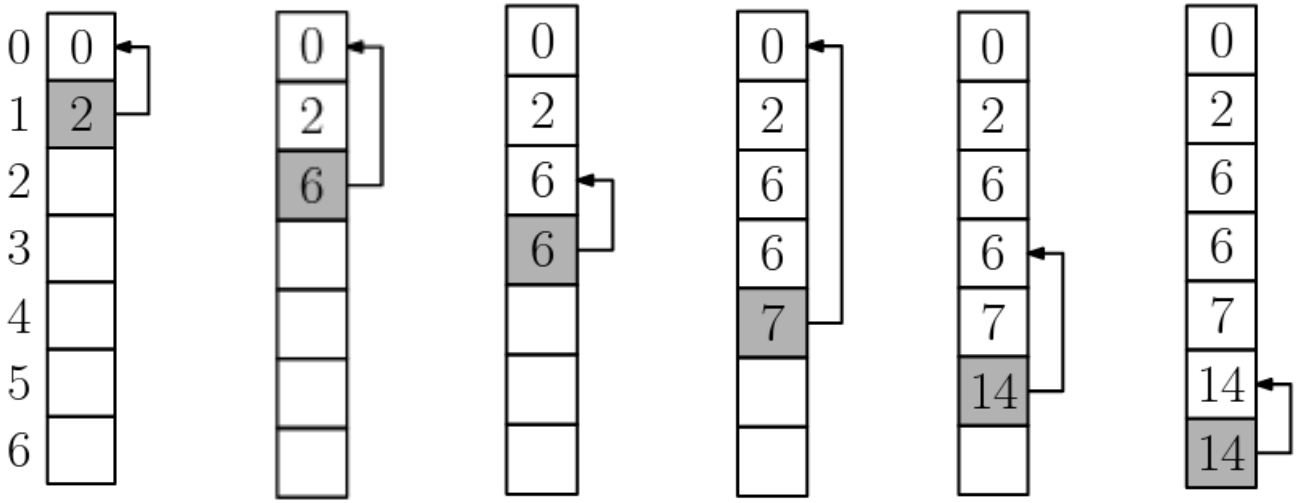
Sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$.

Compute $p(1), p(2), \ldots, p(n)$

```
Iterative-Compute-Opt {
   OPT[0] = 0
   for j = 1 to n
      OPT[j] = max(v_j + OPT[p(j)], OPT[j-1])
}

Output OPT[n]
```

Example


j
v:
p:
1
2
3
4
5
6
2
6
3.5
7
8
1.1
2
6
3.5
7
8
1.1
0
0
1
0
3
3


0
1
2
3
4
5
6
0
2
0
2
6
0
2
6
6
0
2
6
6
7
0
2
6
6
7
14
0
2
6
6
7
14
14

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

A. Do some post-processing – "traceback"

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (v_j + OPT[p(j)] > OPT[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

the condition determining the max when computing OPT[ ]

the relevant sub-problem

All-Pair Shortest Path
Single Source Shortest Path
Optimal Binary Search Tree
Multistage Graph

Floyd’s Algorithm

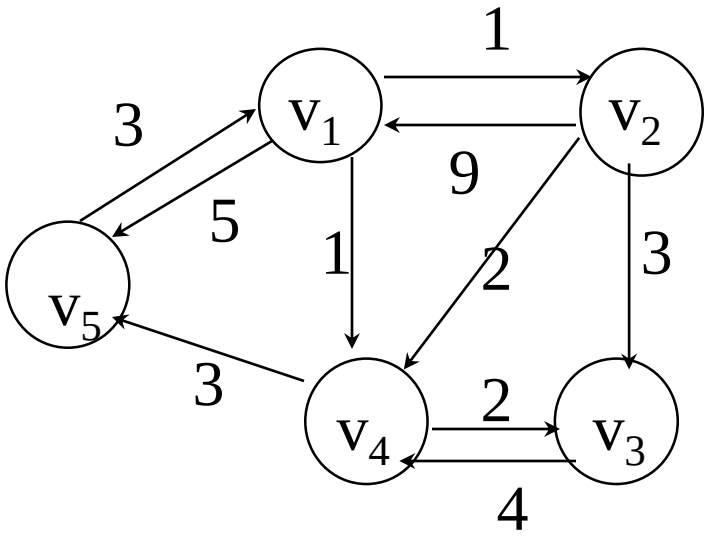
All pairs shortest path

All pairs shortest path

- ***The problem:*** **find the shortest path between every pair of vertices of a graph**
- ***The graph*****: may contain negative edges but no negative cycles**
- ***A representation*****: a weight matrix where**
  W(i,j)=0 if i=j.
  W(i,j)=∞ if there is no edge between i and j.
  W(i,j)="weight of edge"
- **Principle of optimality applies to shortest path problems**

The weight matrix and the graph

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0



The subproblems

- **How can we define the shortest distance $d_{i,j}$ in terms of "smaller" problems?**
- **One way is to restrict the paths to only include vertices from a restricted subset.**
- **Initially, the subset is empty.**
- **Then, it is incrementally increased until it includes all the vertices.**

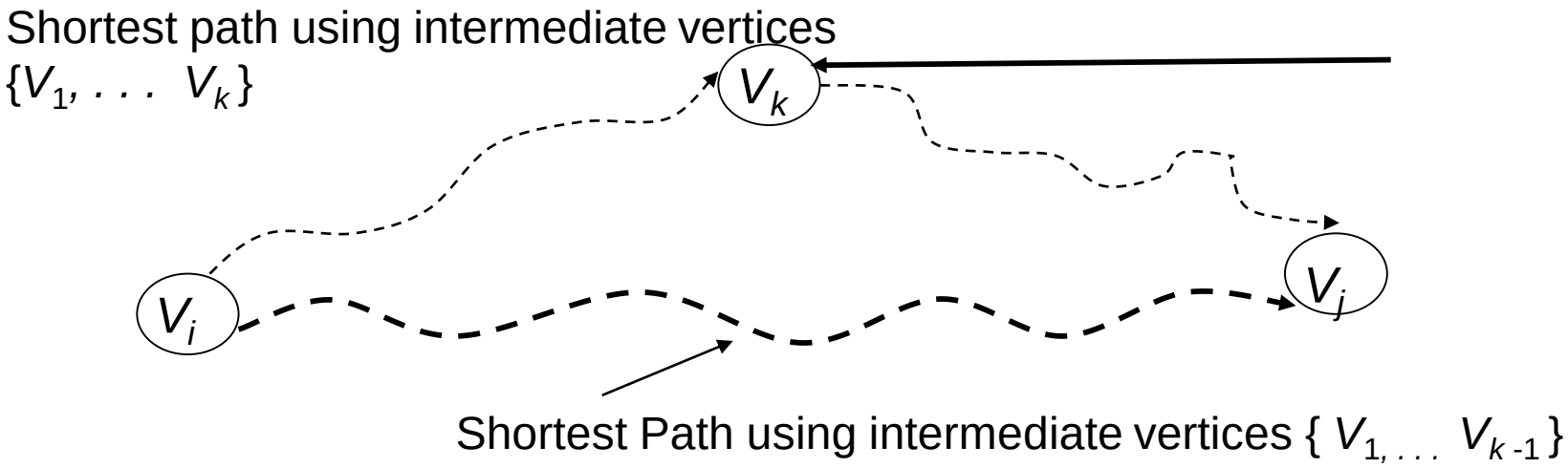
The subproblems

- Let $D^{(k)}[i,j]$=weight of a shortest path from v_i to v_j using only vertices from $\{v_1,v_2,\ldots,v_k\}$ as intermediate vertices in the path
  - $D^{(0)}=W$
  - $D^{(n)}=D$ which is the goal matrix
- How do we compute $D^{(k)}$ from $D^{(k-1)}$?

The Recursive Definition:

Case 1: A shortest path from v_i to v_j restricted to using only vertices from $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices does not use v_k. Then $D^{(k)}[i,j] = D^{(k-1)}[i,j]$.

Case 2: A shortest path from v_i to v_j restricted to using only vertices from $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices does use v_k. Then $D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$.



The recursive definition

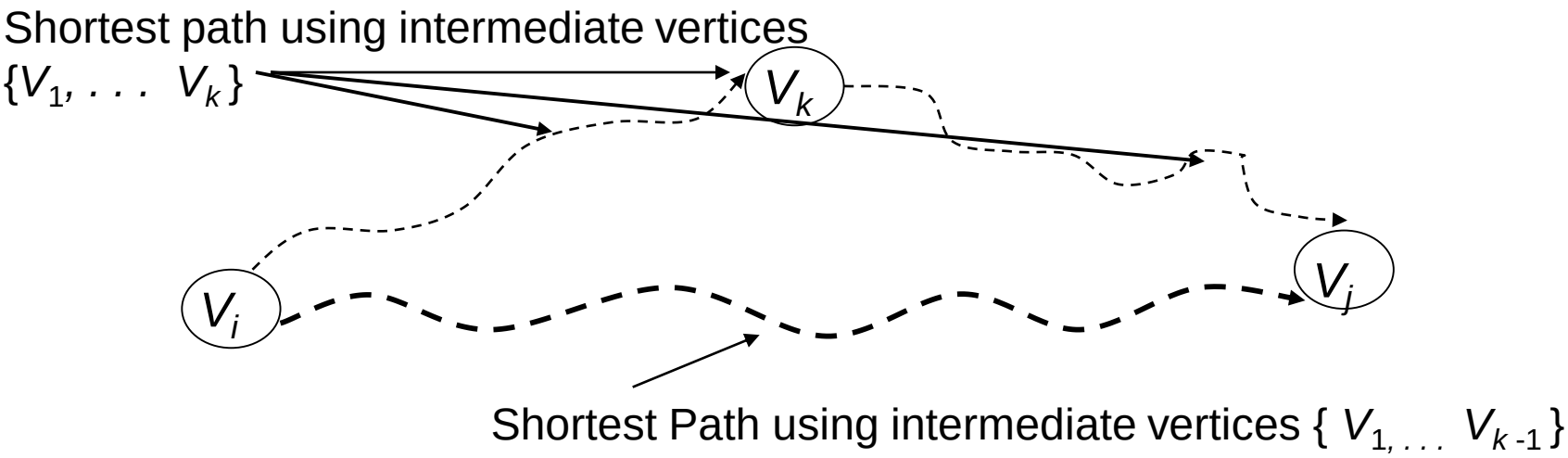
- **Since**

  $D^{(k)}[i,j] = D^{(k-1)}[i,j]$ **or**

  $D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$.

  We conclude:

  $D^{(k)}[i,j] = \min\{ D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \}$.



The pointer array P

- **Used to enable finding a shortest path**
- **Initially the array contains 0**

- **Each time that a shorter path from *i* to *j* is found the *k* that provided the minimum is saved (highest index node on the path from *i* to *j*)**

- **To print the intermediate nodes on the shortest path a recursive procedure that print the shortest paths from *i* and *k*, and from *k* to *j* can be used**

Floyd's Algorithm Using n+1 *D* matrices

Floyd//Computes shortest distance between all pairs of
//nodes, and saves P to enable finding shortest paths

1. $D^0 \leftarrow W$ // initialize *D* array to *W* [ ]

2. $P \leftarrow 0$ // initialize P array to [0]

3. **for** $k \leftarrow 1$ **to** n

4. **do for** $i \leftarrow 1$ **to** n

5. **do for** $j \leftarrow 1$ **to** n

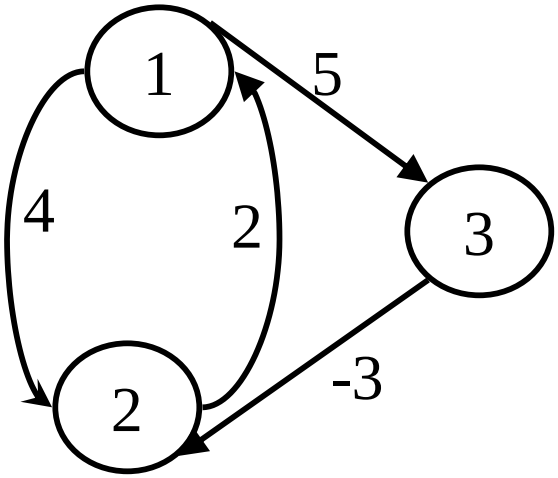
6. **if** $(D^{k-1}[\, i, j\,] > D^{k-1}[\, i, k\,] + D^{k-1}[\, k, j\,])$

7. **then** $D^k[\, i, j\,] \leftarrow D^{k-1}[\, i, k\,] + D^{k-1}[\, k, j\,]$

8. $P[\, i, j\,] \leftarrow k$;

9. **else** $D^k[\, i, j\,] \leftarrow D^{k-1}[\, i, j\,]$

Example

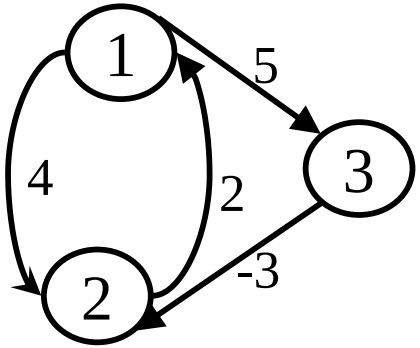


$W = D^0 =$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

$P =$

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0



$D^0 =$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

k = 1
Vertex 1 can be intermediate node

$D^1 =$

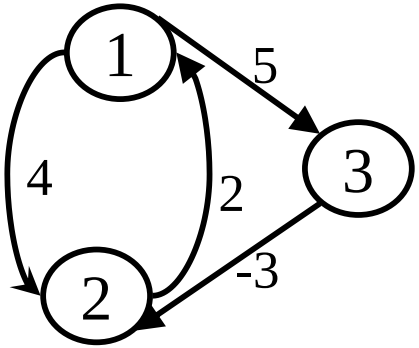
	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

$D^1[2,3] = \min(D^0[2,3], D^0[2,1]+D^0[1,3])$
$= \min (\infty, 7)$
$= 7$

$P =$

	1	2	3
1	0	0	0
2	0	0	1
3	0	0	0

$D^1[3,2] = \min(D^0[3,2], D^0[3,1]+D^0[1,2])$
$= \min (-3,\infty)$
$= -3$



$D^1 =$

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

k = 2
Vertices 1, 2 can be intermediate

$D^2 =$

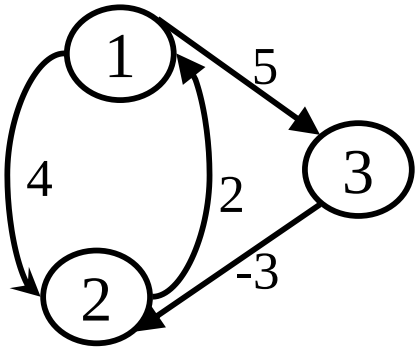
	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

$D^2[1,3] = \min(D^1[1,3], D^1[1,2]+D^1[2,3])$
$= \min (5, 4+7)$
$= 5$

$P =$

	1	2	3
1	0	0	0
2	0	0	1
3	2	0	0

$D^2[3,1] = \min(D^1[3,1], D^1[3,2]+D^1[2,1])$
$= \min (\propto, -3+2)$
$= -1$



$D^2 =$

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

k = 3
Vertices 1, 2, 3 can be intermediate

$D^3 =$

	1	2	3
1	0	2	5
2	2	0	7
3	-1	-3	0

$D^3[1,2] = \min(D^2[1,2], D^2[1,3]+D^2[3,2])$
$= \min(4, 5+(-3))$
$= 2$

$P =$

	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

$D^3[2,1] = \min(D^2[2,1], D^2[2,3]+D^2[3,1])$
$= \min(2, 7+(-1))$
$= 2$

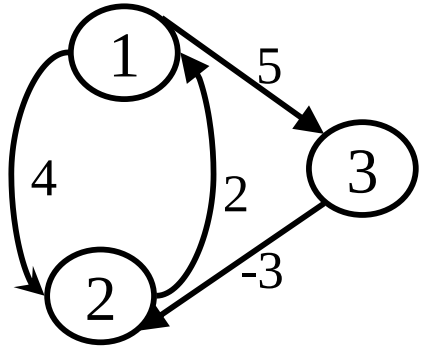
Printing intermediate nodes on shortest path from q to r

```
path(index q, r)
    if (P[ q, r ]!=0)
            path(q, P[q, r])
            println( "v"+ P[q, r])
            path(P[q, r], r)
           return;
    //no intermediate nodes
    else return
```

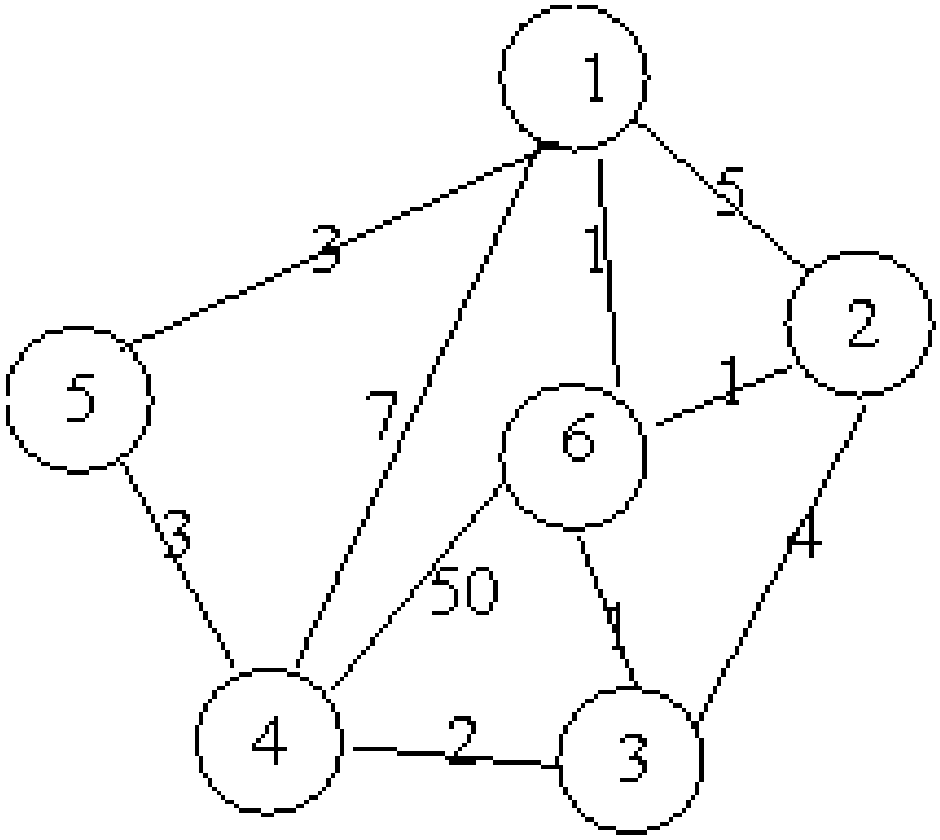
P =

	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

Before calling path check D[q, r] < ∞, and print node q, after the call to path print node r



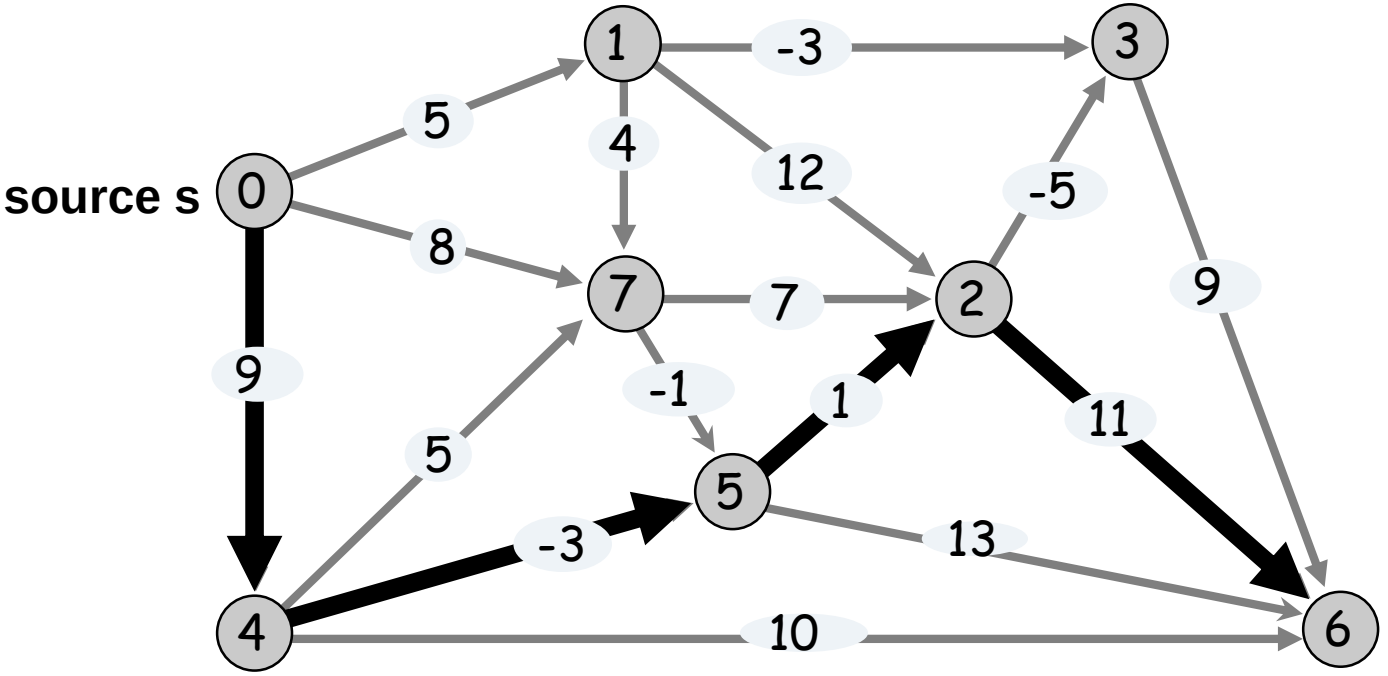
Example



Single Source Shortest paths

Shortest path problem. Given a digraph $G = (V, E)$, with arbitrary edge weights or costs c_{vw}, find cheapest path from node s to node t.

- Allow negative weights

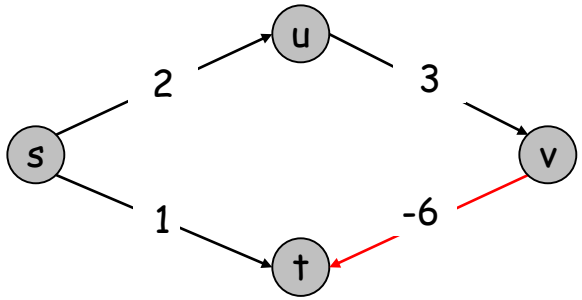


cost of the shown path s-t = 9 - 3 + 1 + 11 = 18 (min. cost = 11)

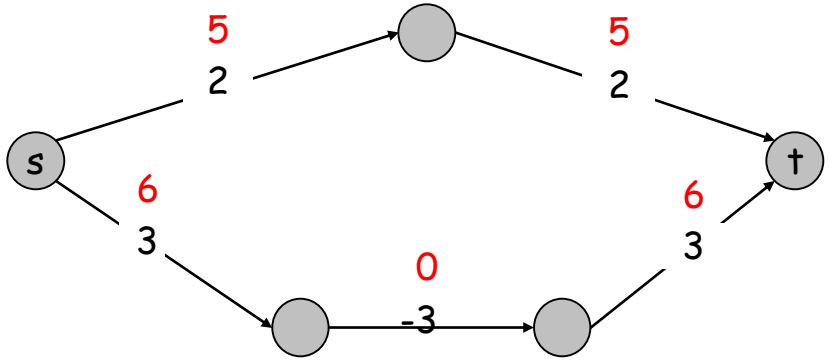
destination t

Shortest Paths: Failed Attempts

Dijkstra. Can fail if negative edge costs/weights are present.

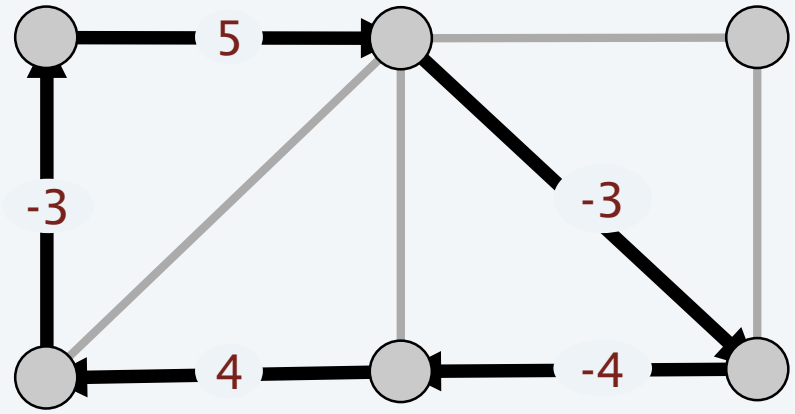


Re-weighting. Adding a constant to every edge weight can fail.



Negative cycles

Def. A negative cycle is a directed cycle such that the sum of its edge weights is negative.

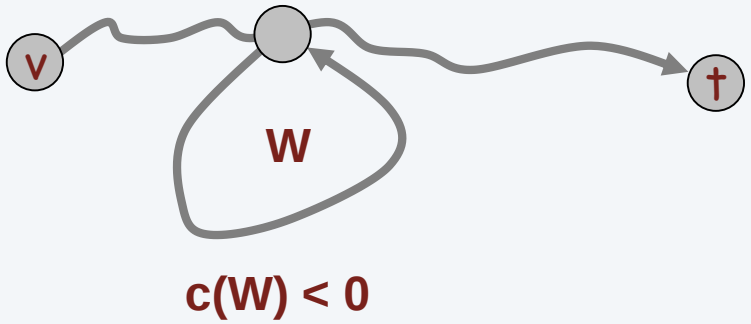


a negative cycle W : $c(W) = \sum_{e \in W} c_e < 0$

Shortest paths and negative cycles

Lemma 1. If some path from v to t contains a negative cycle, then there does not exist a cheapest path from v to t.

Pf. If there exists such a cycle W, then we can build a $v \rightsquigarrow t$ path of arbitrarily negative weight by detouring around the cycle as many times as desired. ▪

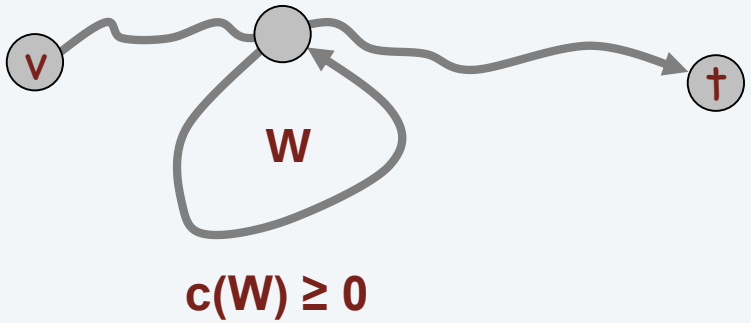


Shortest paths and negative cycles

Lemma 2. If G has no negative cycle, then there exists a cheapest path from v to t that is *simple* (and that has $\leq n - 1$ edges).

Pf.

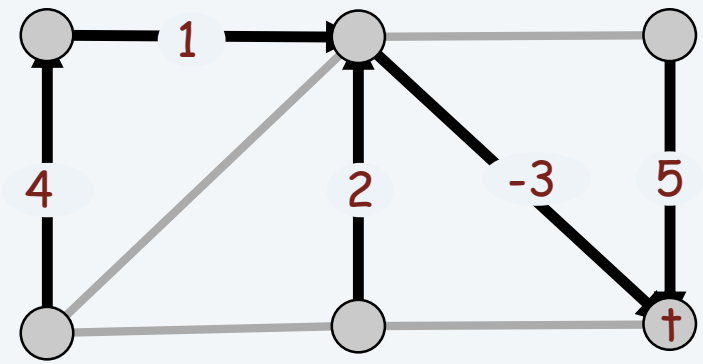
- Consider a cheapest $v \rightsquigarrow t$ path P that uses the fewest number of edges.
- If P contains a cycle W, we can remove portion of P corresponding to W, *obtaining a path with fewer edges,* without increasing the cost. ▪



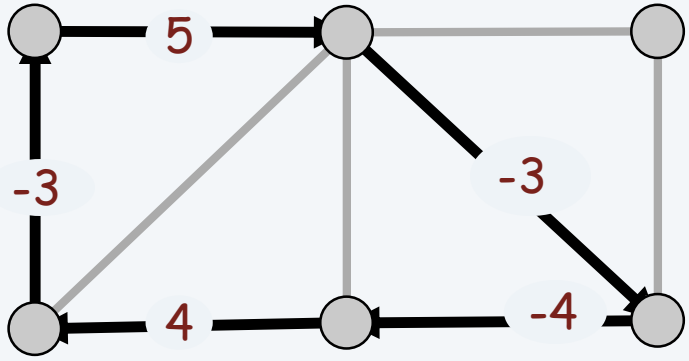
Shortest path and negative cycle problems

Shortest path problem. Given a digraph $G = (V, E)$ with edge weights c_{vw} and no negative cycles, find cheapest $v \rightsquigarrow t$ path for each node v.

Negative cycle problem. Given a digraph $G = (V, E)$ with edge weights c_{vw}, find a negative cycle (if one exists).



shortest-paths tree



negative cycle

Shortest paths: dynamic programming

- Let $dist^{l}[u]$ be the shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most l edges. Then
  - $dist^{1}[u] = \text{cost}[\text{v}, \text{u}], 1 \leq u \leq n$

- When there is no cycles of negative length, we can limit our search for shortest paths to paths with at most $n-1$ edges.

- Goal: Find out $dist^{n-1}[u], 1 \leq u \leq n$

Shortest paths: dynamic programming

Observations

- If the shortest path from v to u with at most $k, k > 1$, edges has no more than $k-1$ edges,then $dist^k[u] = dist^{k-1}[u]$.

- If the shortest path from v to u with at most $k, k > 1$,edges has exactly k edges, then it is made up of a shortest path from v to some vertex j followed by the $edge\ (j,u)$.
  - The path from v to j has k-1 edges, and its length is $dist^{k-1}[j]$.
  - All vertices i such that the edge (i,u) is in the graph are candidates for j.
  - Since we are interested in a shortest path, the i that minimizes $dist^{k-1}[i] + cost[i,u]$, is the correct value for j.

Shortest paths: dynamic programming

The observations result in the following recurrence for $dist$

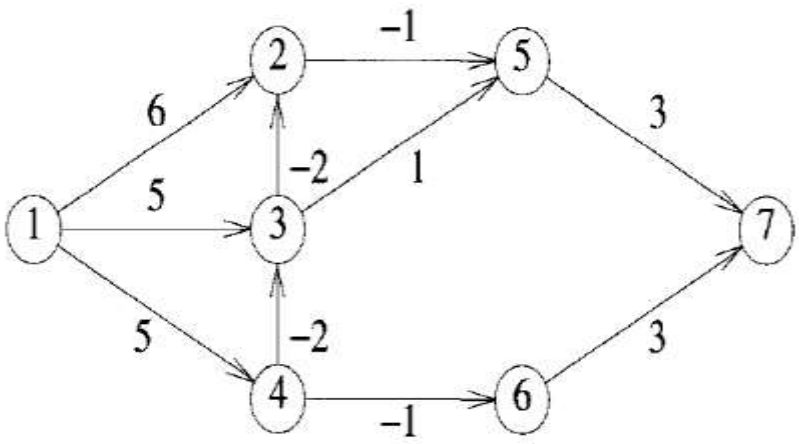
$$dist^k[u] = \min\left\{dist^{k-1}[u], min_i\{dist^{k-1}[i] + cost[i,u]\}\right\}$$

- This recurrence can be used to compute $dist^k[u]$ from $dist^{k-1}[u]$, for $k = 2, 3, \ldots, n-1$

Bellman Ford Algorithm

```
Algorithm BellmanFord(v, cost, dist, n)
// Single-source/all-destinations shortest
// paths with negative edge costs
{
     for i := 1 to n do // Initialize dist.
          dist[i] := cost[v, i];
     for k := 2 to n − 1 do
          for each u such that u ≠ v and u has
                    at least one incoming edge do
               for each ⟨i, u⟩ in the graph do
                    if dist[u] > dist[i] + cost[i, u] then
                         dist[u] := dist[i] + cost[i, u];
}
```

Example of Bellman-Ford



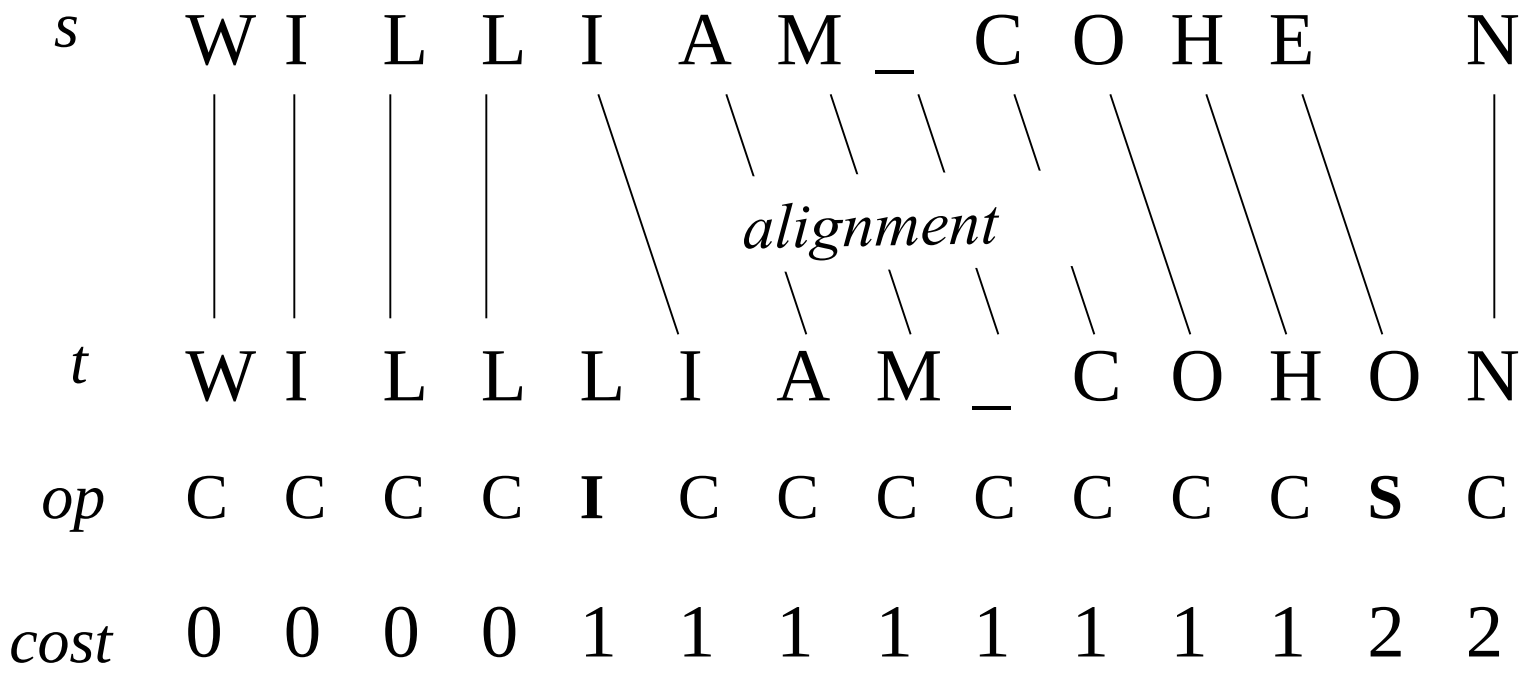
	$dist^k[1 \ldots 7]$						
k	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

String distance metrics: Levenshtein

- Edit-distance metrics
  - Distance is **shortest sequence of edit commands** that transform *s* to *t*.
  - Simplest set of operations:
    - Copy character from *s* over to *t*
    - Delete a character in *s* (cost 1)
    - Insert a character in *t* (cost 1)
    - Substitute one character for another (cost 1)
  - This is "Levenshtein distance"

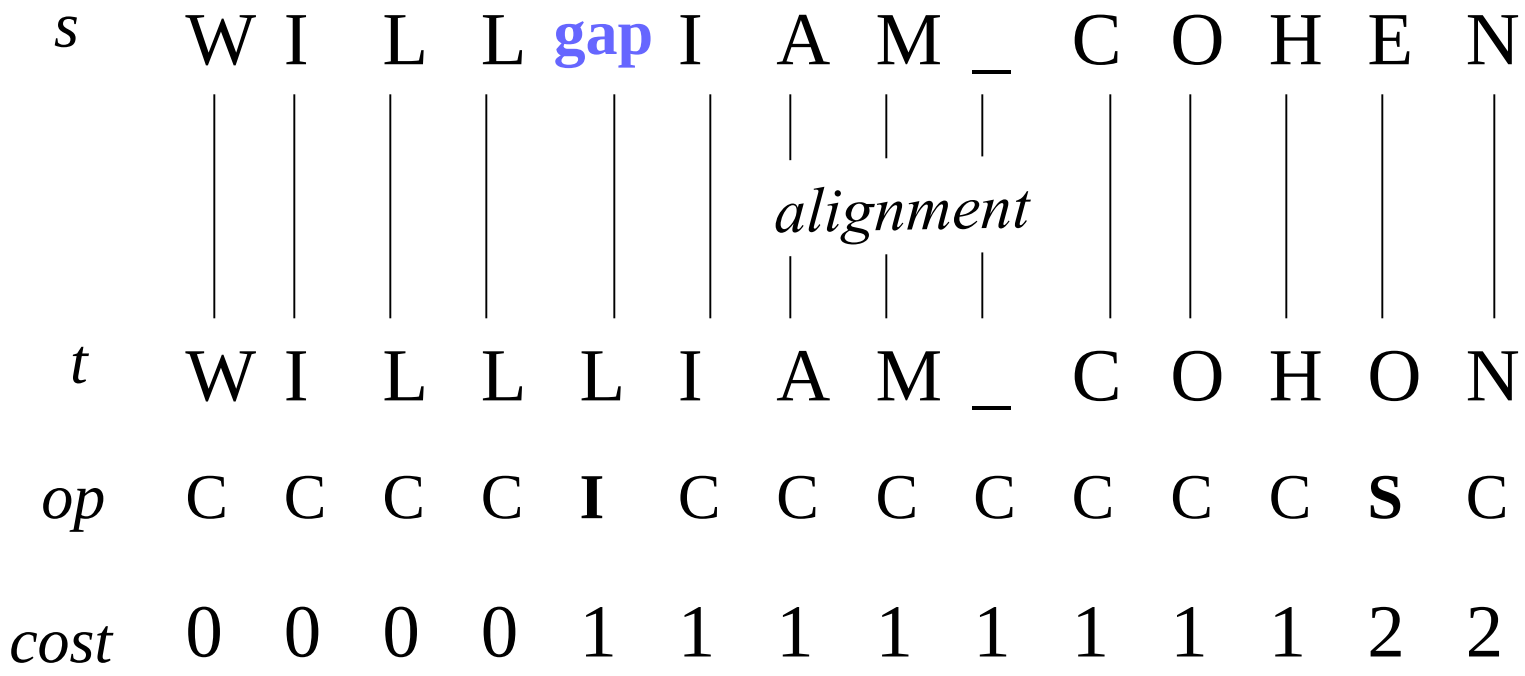
Levenshtein distance - example

- distance("William Cohen", "Willliam Cohon")



Levenshtein distance - example

- distance("William Cohen", "Willliam Cohon")



Computing Levenshtein distance - 1

D(i,j) = score of **best** alignment from $s_1 \ldots s_i$ to $t_1 \ldots t_j$

$$= \min \begin{cases} D(i-1, j-1), & if\ s_i = t_j \quad //copy \\ D(i-1, j-1) + 1, & if\ s_i != t_j \quad //substitute \\ D(i-1, j) + 1 & //insert \\ D(i, j-1) + 1 & //delete \end{cases}$$

Computing Levenshtein distance - 2

D(i,j) = score of **best** alignment from *s1..si* to *t1..tj*

$$= \min \begin{cases} D(i-1, j-1) + d(s_i, t_j) & \textit{//subst/copy} \\ D(i-1, j) + 1 & \textit{//insert} \\ D(i, j-1) + 1 & \textit{//delete} \end{cases}$$

(simplify by letting d(c,d)=0 if c=d, 1 else)

also let D(i,0)=i *(for i inserts)* and D(0,j)=j

Computing Levenshtein distance - 3

$$\text{D(i,j)= min} \begin{cases} D(i-1,j-1) + d(s_i,t_j) & \textit{//subst/copy} \\ D(i-1,j)+1 & \textit{//insert} \\ D(i,j-1)+1 & \textit{//delete} \end{cases}$$

		0	1	2	3	4	5
		t_j	C	O	H	E	N
0	s_i	0	1	2	3	4	5
1	M	1	1	2	3	4	5
2	C	2	1	2	3	4	5
3	C	3	2	2	3	4	5
4	O	4	3	2	3	4	5
5	H	5	4	3	2	3	4
6	N	6	5	4	3	3	3

$= D(s,t)$

Computing Levenshtein distance – 4

$$D(i,j) = \min \begin{cases} D(i-1,j-1) + d(si,tj) & \textit{//subst/copy} \\ D(i-1,j)+1 & \textit{//insert} \\ D(i,j-1)+1 & \textit{//delete} \end{cases}$$

A *trace* indicates where the min value came from, and can be used to find edit operations and/or a best *alignment* (may be more than 1)

	C	O	H	E	N
M	**1**	2	3	4	5
C	**1**	2	3	4	5
C	**2**	3	3	4	5
O	3	**2**	3	4	5
H	4	3	**2**	**3**	4
N	5	4	3	3	**3**