Dynamic Programming

Dynamic Programming

- Dynamic Programming is an algorithm design technique for soling complex *optimization problems:* often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to sub-problems.
- Unlike divide and conquer, sub-problems are not independent.
  - » Sub-problems may share sub-sub-problems,
  - » A divide and conquer approach would repeatedly solve the common subproblems
  - » Dynamic programming solves every subproblem just once and stores the answer in a table

Dynamic Programming

- The term Dynamic Programming comes from Control Theory, not computer science. Programming refers to the use of tables (arrays) to construct a solution.
- In dynamic programming we usually reduce time by increasing the amount of space.
- We solve the problem by solving sub-problems of increasing size and saving each optimal solution in a table (usually).
- The table is then used for finding the optimal solution to larger problems.
- Time is saved since each sub-problem is solved only once.

Dynamic Programming Steps

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 in a bottom-up fashion
4. **Construct** an optimal solution from computed information (not always necessary)

Memoization

- Dynamic algorithms use Memoization.
- Top-down approach with the efficiency of typical dynamic programming approach
- **Memoization** improves the performance of recursive algorithms.
- It involves rewriting the recursive algorithm so that as answers to problems are found, they are stored in an array.
- Recursive calls can look up results in the array rather than having to recalculate them.

Dynamic Progamming vs. Memoization

- Advantages of dynamic programming vs. memoized algorithms
  - » No overhead for recursion, less overhead for maintaining the table
  - » The regular pattern of table accesses may be used to reduce time or space requirements
- Advantages of memoized algorithms vs. dynamic programming
  - » Some subproblems do not need to be solved

Dynamic Programming

- The best way to get a clear idea on DP is through some examples.
  - » Fibonacci
  - » Factorials
  - » Matrix Chaining optimization
  - » Longest Common Subsequence
  - » 0-1 Knapsack Problem
  - » Transitive Closure of a direct graph
  - » String Matching

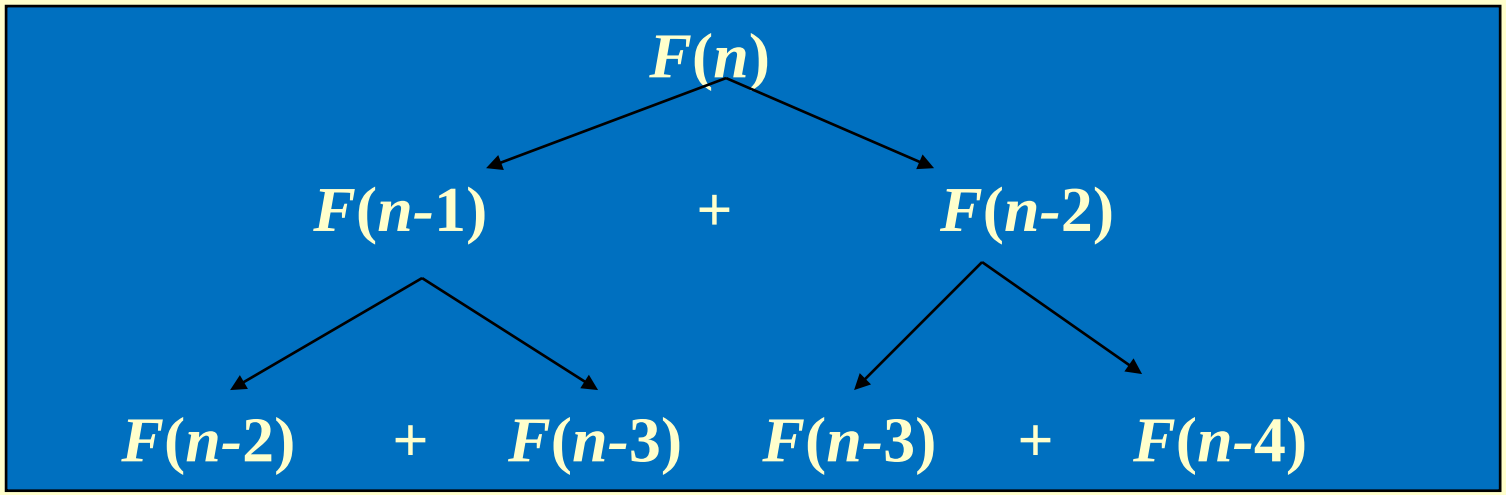
Fibonacci Numbers

Fibonacci Numbers

- Computing the n^{th} Fibonacci number recursively:
  - F(n) = F(n-1) + F(n-2)
  - F(0) = 0
  - F(1) = 1
  - Top-down approach

Top-down recursive solution

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



Fibonacci Numbers

- What is the Recurrence relationship?
  - » $T(n) = T(n-1) + T(n-2) + 1$
- What is the solution to this?
  - » Clearly it is $O(2^n)$, but this is not tight.
  - » A lower bound is $\Omega(2^{n/2})$.
  - » You should notice that T(n) grows very similarly to F(n), so in fact $T(n) = \Theta(F(n))$.
- Obviously not very good, but we know that there is a better way to solve it!

Fibonacci Numbers

- Computing the n^{th} Fibonacci number using a bottom-up approach:
  - F(0) = 0
  - F(1) = 1
  - F(2) = 1+0 = 1
  - …
  - F(n-2) =
  - F(n-1) =
  - F(n) = F(n-1) + F(n-2)

Top-down recursive solution

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

0	**1**	**1**	**. . .**	***F*(*n*-2)**	***F*(*n*-1)**	***F*(*n*)**

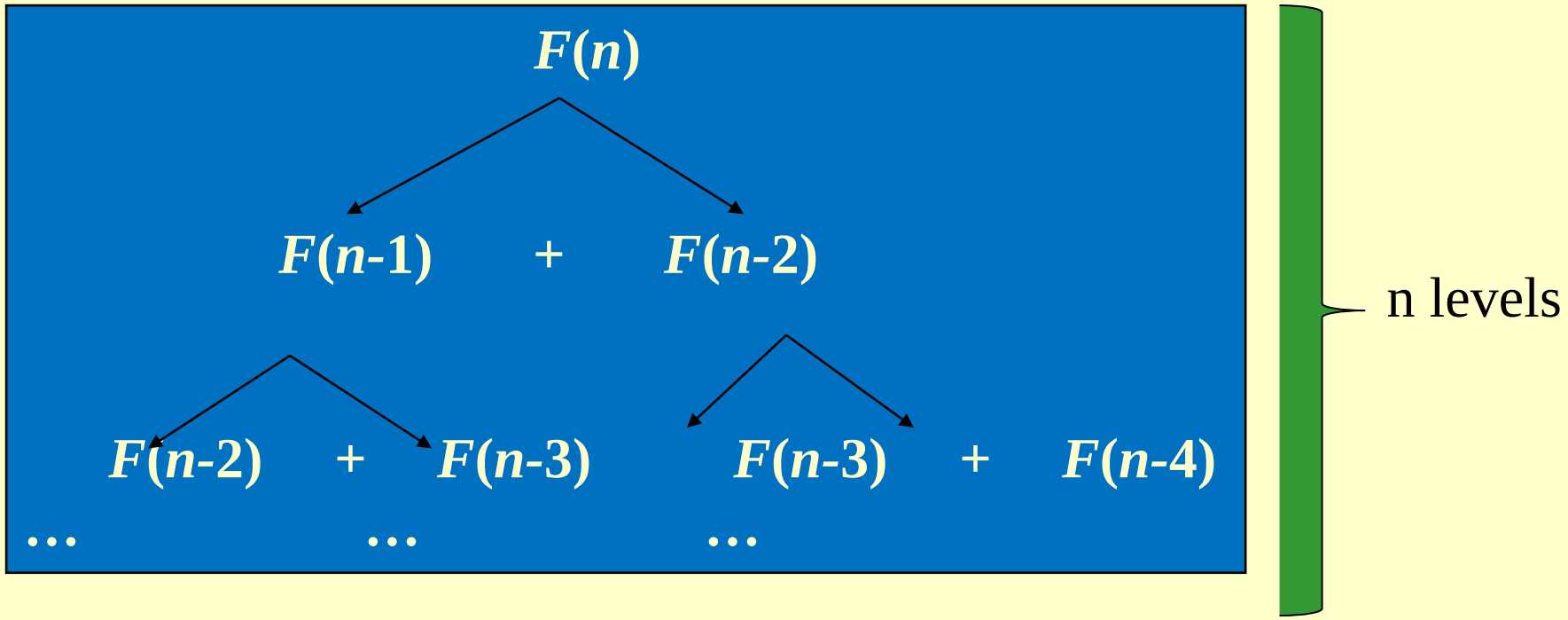
- Efficiency:
  - Time – O(n)
  - Space – O(n)

Bottom-up DP-based solution

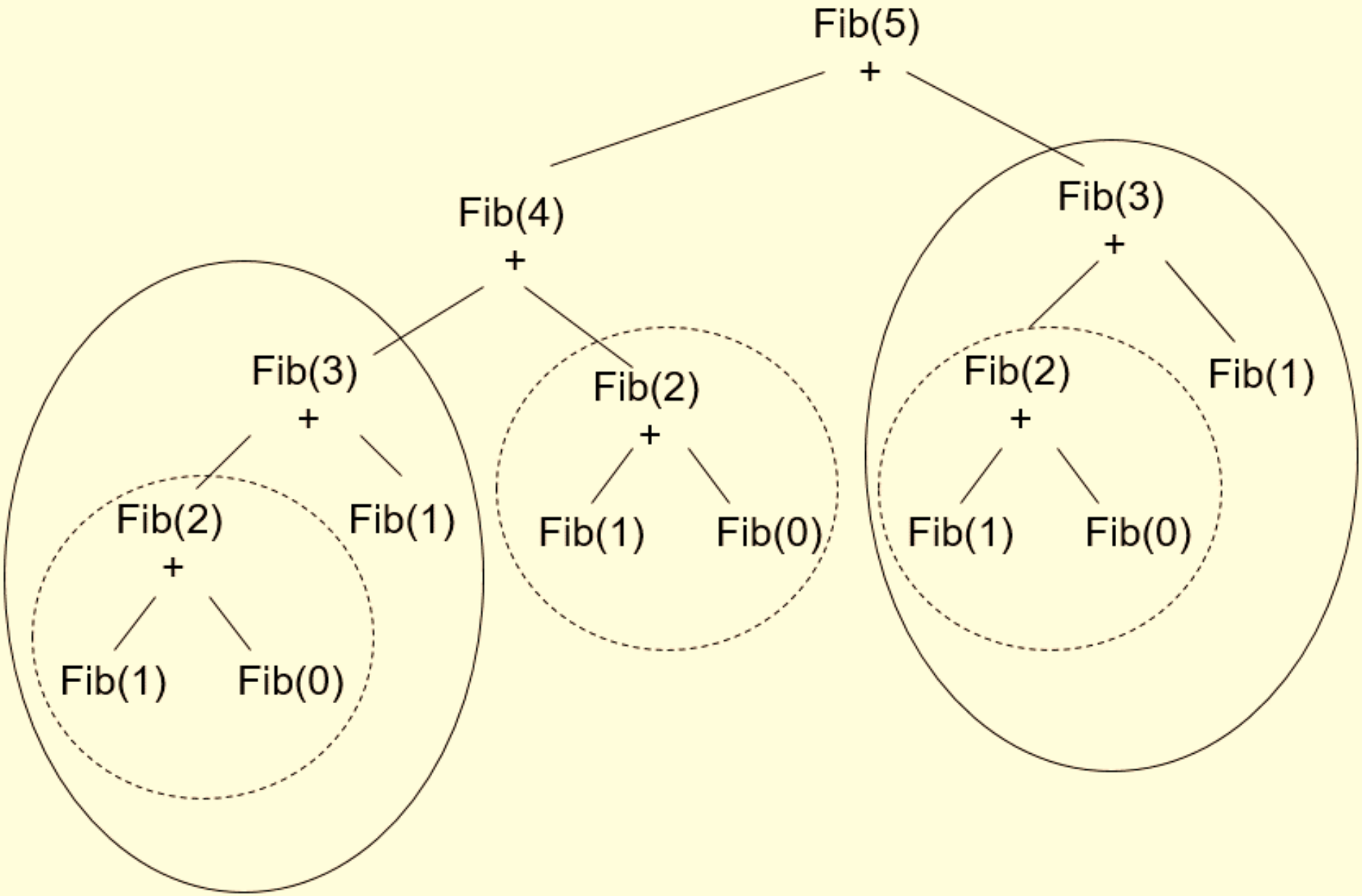
```
int Fib(int n){
   int fibArray [n], i;
   fibArray[0] = 0, fibArray[1] = 1;
   for(i=2; i < n; i++)
      fibArray[i] = fibArray[i-1]+fibArray[i-2];
   return fibArray[n-1];
}
```

Fibonacci Numbers

- The bottom-up approach is only Θ(n).
- Why is the top-down so inefficient?
  - » Recomputes many sub-problems.



Fibonacci Numbers



Memoized Fibonacci

```
term [100] = 0;
 fib(n)
{
        if (n <= 1)
        return n;
     if (term[n] != 0)
        return term[n];
     else
    {
         term[n] = fib(n - 1) + fib(n - 2);
         return term[n];
    }
}
```

- Efficiency:
  - » Time – O(n)
  - » Space – O(n)

Dynamic Programming

- Two main properties of a problem that suggests that the given problem can be solved using Dynamic programming:

  1) Overlapping Subproblems: solutions of same subproblems are needed again and again

  2) Optimal Substructure: optimal solution of the given problem can be obtained by using optimal solutions of its subproblems

- Two different ways to store the values so that these values can be reused:

  a) Memoization (Top Down)

  b) Tabulation (Bottom Up)

Tabulation Vs Memoization

	Tabulation	Memoization
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

Binomial Coefficient

Binomial Coefficient

- A binomial coefficient C(n, k) can be defined as the coefficient of x^k in the expansion of $(1 + x)^n$.
- Given the values of n and k such that $n \geq k \geq 0$. Find the value of Binomial Coefficient C(n, k).

$$C(n, k) = \binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Binomial Coefficient: Optimal Substructure

- The value of C(n, k) can be recursively calculated using the following standard formula for Binomial Coefficients.

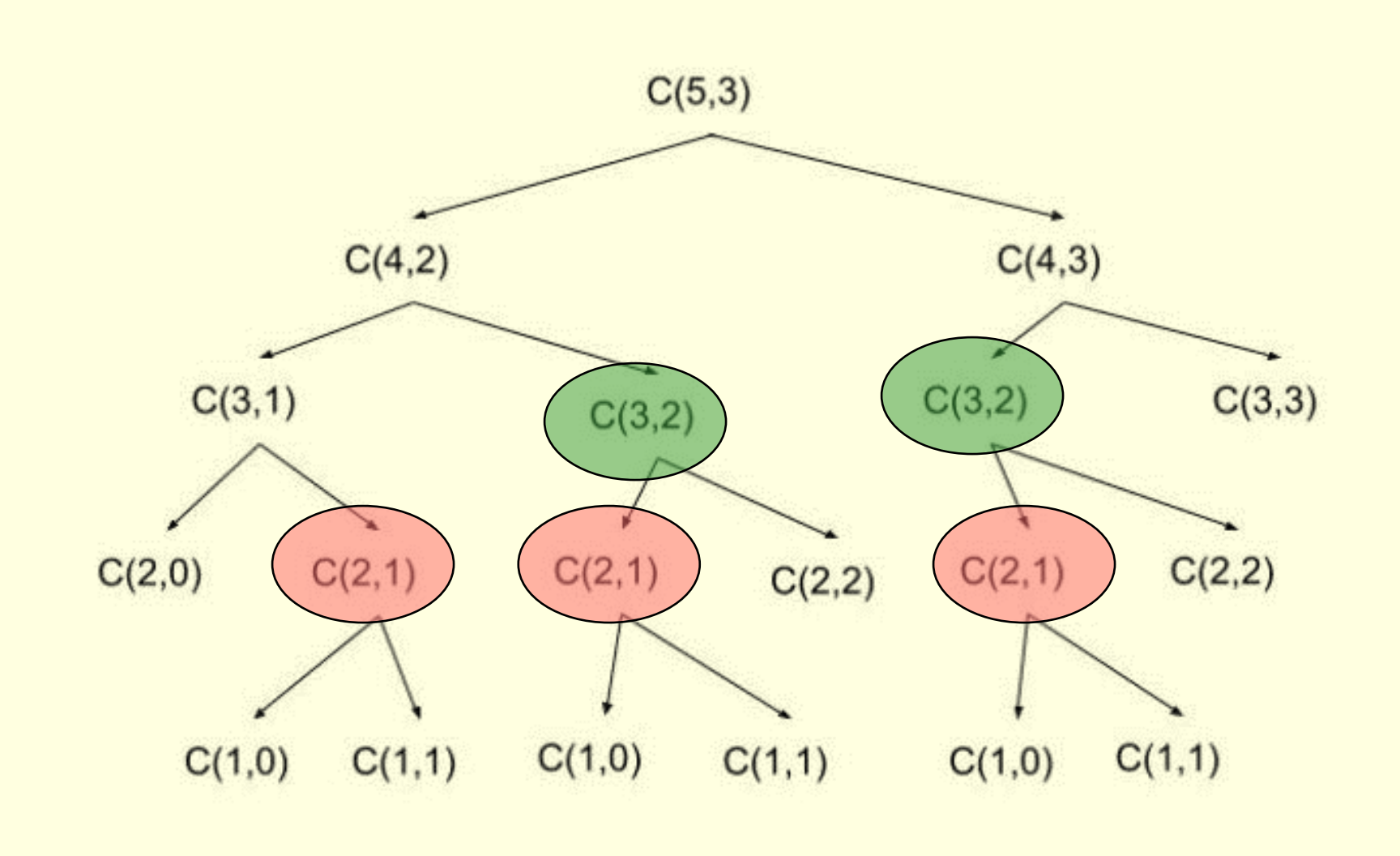
$$C(n, k) = C(n-1, k-1) + C(n-1, k), \text{ for } n > k > 0$$
$$C(n, 0) = C(n, n) = 1$$

Binomial Coefficient: Recursive approach

```
binomialCoeff(n, k)
    {
     if (k > n)
            return 0;
        if (k == 0 || k == n)
            return 1;
 return binomialCoeff(n - 1, k - 1) + binomialCoeff(n - 1, k);
    }
```

Time complexity: $O(2^n)$

Binomial Coefficient: Overlapping Subproblems



Binomial Coefficient: DP approach

Algorithm *Binomial*(n, k)

```
for i ← 0 to n do
    for j = 0 to min(i, k) do
        if j==0 or j==i then C[i, j] ← 1
        else C[i, j] ← C[i-1, j-1] + C[i-1, j]
return C[n, k]
```

Time Complexity: O(n*k)

Space Complexity: O(n*k)

Binomial Coefficient: DP approach

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

Matrix-Chain Multiplication

Matrix-Chain Multiplication

Problem: given a sequence $\langle A_1, A_2, \ldots, A_n \rangle$, compute the product:

$$A_1 \cdot A_2 \cdots A_n$$

- Matrix compatibility for multiplication:

$$C = A \cdot B$$

$col_A = row_B$

$row_C = row_A$

$col_C = col_B$

$$C = A_1 \cdot A_2 \cdots A_i \cdot A_{i+1} \cdots A_n$$

$col_i = row_{i+1}$

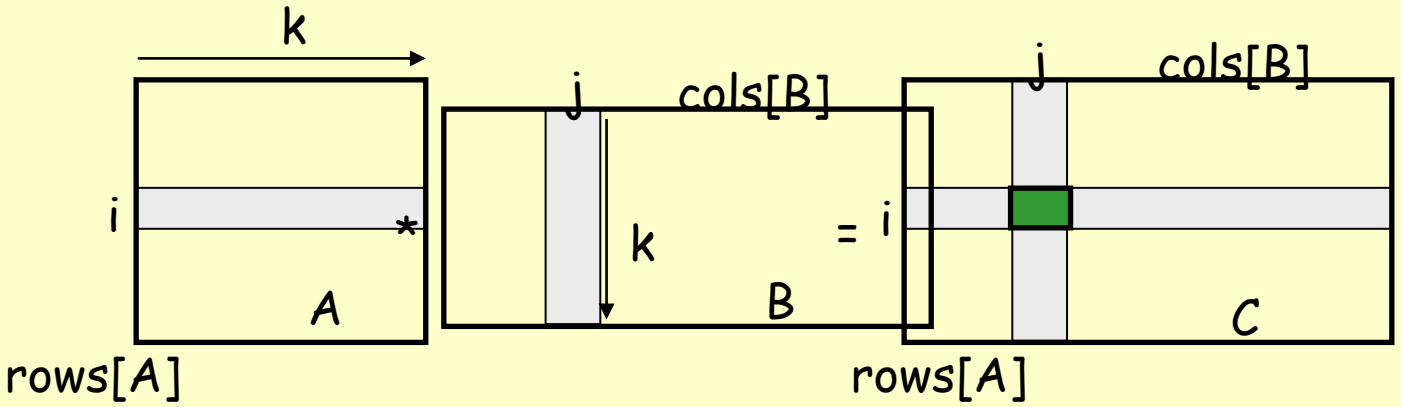
$row_C = row_{A1}$

$col_C = col_{An}$

MATRIX-MULTIPLY(A, B)

```
if columns[A] ≠ rows[B]
    then error "incompatible dimensions"
    else for i ← 1 to rows[A]
              do for j ← 1 to columns[B]
                    do C[i, j] = 0
                       for k ← 1 to columns[A]
                           do C[i, j] ← C[i, j] + A[i, k] B[k, j]
```

rows[A] · cols[A] · cols[B] multiplications



Matrix-Chain Multiplication

- In what order should we multiply the matrices?

$$A_1 \cdot A_2 \cdots A_n$$

- Parenthesize the product to get the order in which matrices are multiplied
- *E.g.:* $A_1 \cdot A_2 \cdot A_3 = ((A_1 \cdot A_2) \cdot A_3)$

$$= (A_1 \cdot (A_2 \cdot A_3))$$

- Which one of these orderings should we choose?
  - » The order in which we multiply the matrices has a significant impact on the cost of evaluating the product

Example

$$A_1 \cdot A_2 \cdot A_3$$

- A_1: 10 x 100
- A_2: 100 x 5
- A_3: 5 x 50

1. $((A_1 \cdot A_2) \cdot A_3)$: $A_1 \cdot A_2 = 10 \times 100 \times 5 = 5{,}000$ (10 x 5)

   $((A_1 \cdot A_2) \cdot A_3) = 10 \times 5 \times 50 = 2{,}500$

   Total: 7,500 scalar multiplications

2. $(A_1 \cdot (A_2 \cdot A_3))$: $A_2 \cdot A_3 = 100 \times 5 \times 50 = 25{,}000$ (100 x 50)

   $(A_1 \cdot (A_2 \cdot A_3)) = 10 \times 100 \times 50 = 50{,}000$

   Total: 75,000 scalar multiplications

one order of magnitude difference!!

Matrix-Chain Multiplication: Problem Statement

- Given a chain of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, where A_i has dimensions $p_{i-1} \times p_i$, fully parenthesize the product $A_1 \cdot A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

$$A_1 \cdot A_2 \cdots A_i \cdot A_{i+1} \cdots A_n$$

$$p_0 \times p_1 \quad p_1 \times p_2 \quad p_{i-1} \times p_i \quad p_i \times p_{i+1} \quad p_{n-1} \times p_n$$

What is the number of possible parenthesizations?

- Exhaustively checking all possible parenthesizations is not efficient!
- It can be shown that the number of parenthesizations grows as $\Omega(4^n/n^{3/2})$
- The solution is exponential – Not desirable!!

(see page *372 Chapter 15* in CLRS book)

1. The Structure of an Optimal Parenthesization

- Notation:

$$A_{i\ldots j} = A_i\, A_{i+1} \cdots A_j,\ i \le j$$

- Suppose that an optimal parenthesization of $A_{i\ldots j}$ splits the product between A_k and A_{k+1}, where $i \le k < j$

$$\begin{aligned} A_{i\ldots j} &= A_i\, A_{i+1} \cdots A_j \\ &= A_i\, A_{i+1} \cdots A_k\, A_{k+1} \cdots A_j \\ &= A_{i\ldots k}\, A_{k+1\ldots j} \end{aligned}$$

Optimal Substructure

$$A_{i\ldots j} = A_{i\ldots k}\, A_{k+1\ldots j}$$

- The parenthesization of the "prefix" $A_{i\ldots k}$ must be an optimal parentesization
- If there were a less costly way to parenthesize $A_{i\ldots k}$, we could substitute that one in the parenthesization of $A_{i\ldots j}$ and produce a parenthesization with a lower cost than the optimum $\Rightarrow$ contradiction!
- An optimal solution to an instance of the matrix-chain multiplication contains within it optimal solutions to subproblems

2. A Recursive Solution

- Subproblem:

  determine the minimum cost of parenthesizing $A_{i \cdots j} = A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$

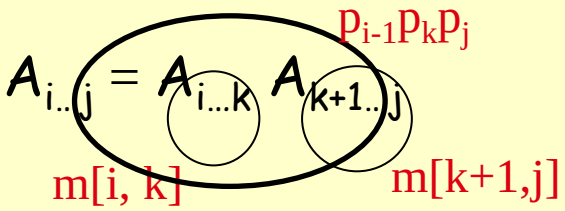
- Let $m[i, j]$ = the minimum number of multiplications needed to compute $A_{i \cdots j}$

  » full problem ($A_{1..n}$): $m[1, n]$

  » $i = j$: $A_{i \cdots i} = A_i \Rightarrow m[i, i] = 0$, for $i = 1, 2, \ldots, n$

2. A Recursive Solution

- Consider the subproblem of parenthesizing $A_{i...j} = A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$



for $i \le k < j$

- Assume that the optimal parenthesization splits the product $A_i A_{i+1} \cdots A_j$ at k ($i \le k < j$)

$m[i, j] = \underbrace{m[i, k]}_{\text{min \# of multiplications to compute } A_{i...k}} + \underbrace{m[k+1, j]}_{\text{min \# of multiplications to compute } A_{k+1...j}} + \underbrace{p_{i-1}p_kp_j}_{\text{\# of multiplications to compute } A_{i...k}A_{k...j}}$

2. A Recursive Solution (cont.)

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$$

- We do not know the value of k
  - » There are j – i possible values for k: k = i, i+1, …, j-1
- Minimizing the cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes:

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

3. Computing the Optimal Costs

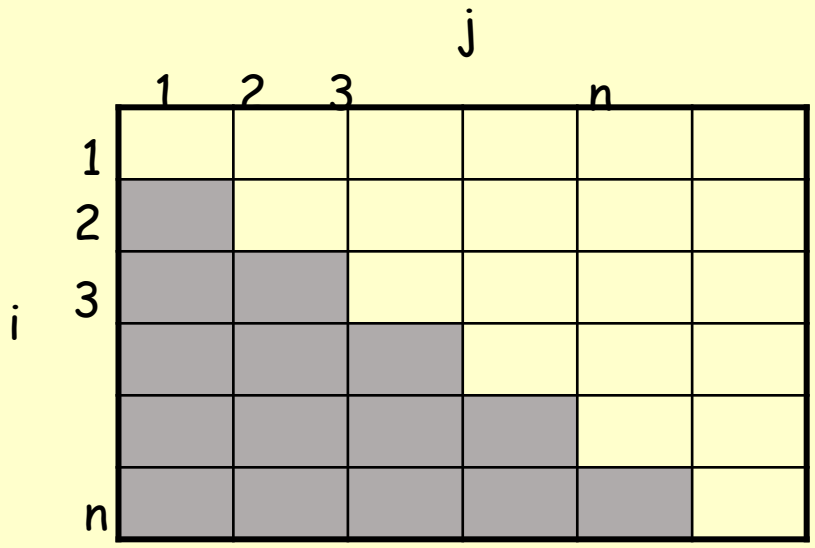
$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$

- Computing the optimal solution recursively takes exponential time!
- How many subproblems?

  $\Rightarrow \Theta(n^2)$

  » Parenthesize $A_{i \ldots j}$ for $1 \le i \le j \le n$

  » One problem for each choice of i and j



3. Computing the Optimal Costs (cont.)

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

- How do we fill in the tables m[1..n, 1..n]?
  - » Determine which entries of the table are used in computing $m[i, j]$

$$A_{i \ldots j} = A_{i \ldots k} \, A_{k+1 \ldots j}$$

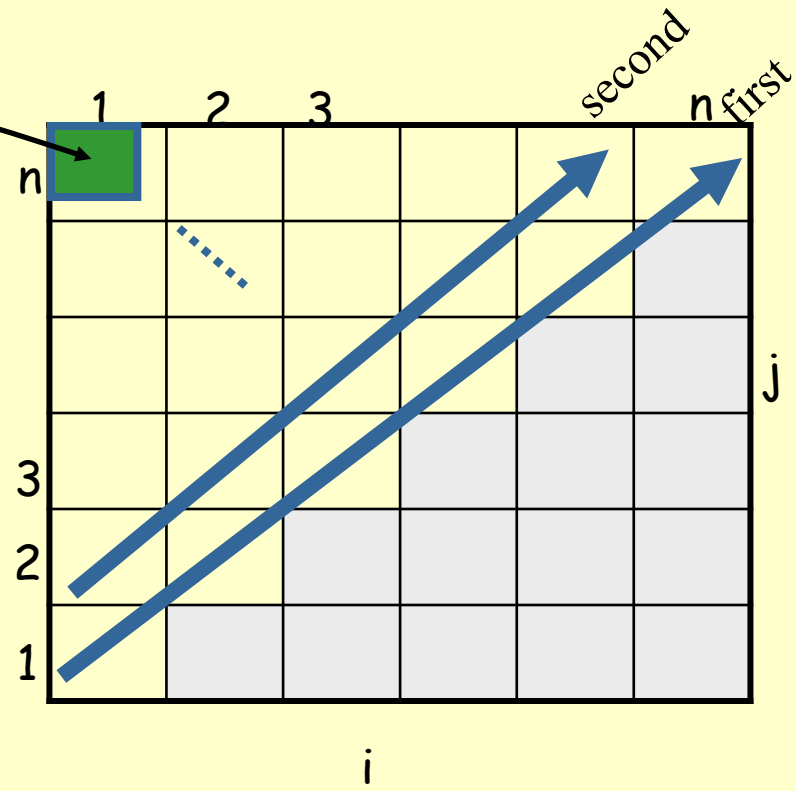
  - » Subproblems' size is atleast one less than the original size
  - » **Idea:** fill in m such that it corresponds to solving problems of increasing length

3. Computing the Optimal Costs (cont.)

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

- Length = 1: $i = j$, $i = 1, 2, \ldots, n$
- Length = 2: $j = i + 1$, $i = 1, 2, \ldots, n-1$

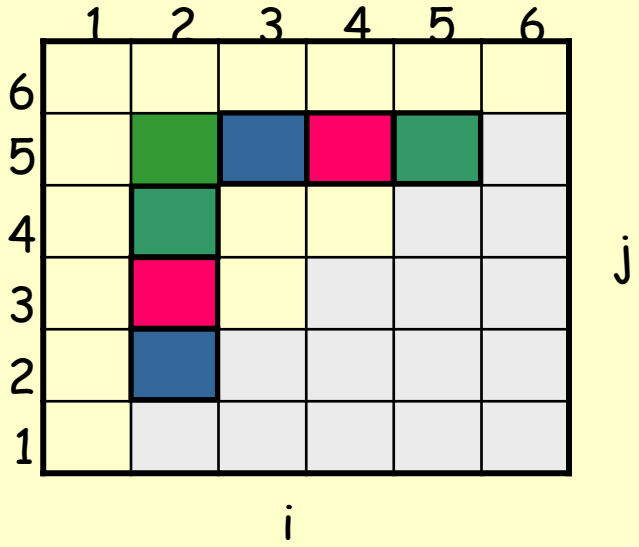
$m[1, n]$ gives the optimal solution to the problem



Compute rows from bottom to top and from left to right

Example: $\min \{m[i, k] + m[k+1, j] + p_{i-1}p_k p_j\}$

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1p_2p_5 & k = 2 \\ m[2, 3] + m[4, 5] + p_1p_3p_5 & k = 3 \\ m[2, 4] + m[5, 5] + p_1p_4p_5 & k = 4 \end{cases}$$



- Values m[i, j] depend only on values that have been previously computed

Example $\min \{m[i, k] + m[k+1, j] + p_{i-1}p_k p_j\}$

Compute $A_1 \cdot A_2 \cdot A_3$

- A_1: 10 x 100 (p_0 x p_1)
- A_2: 100 x 5 (p_1 x p_2)
- A_3: 5 x 50 (p_2 x p_3)

	1	2	3
3	2 7500	2 25000	0
2	1 5000	0	
1	0		

m[i, i] = 0 for i = 1, 2, 3

m[1, 2] = m[1, 1] + m[2, 2] + $p_0p_1p_2$ (A_1A_2)

= 0 + 0 + 10 *100* 5 = 5,000

m[2, 3] = m[2, 2] + m[3, 3] + $p_1p_2p_3$ (A_2A_3)

= 0 + 0 + 100 * 5 * 50 = 25,000

m[1, 3] = min { m[1, 1] + m[2, 3] + $p_0p_1p_3$ = 75,000 ($A_1(A_2A_3)$)

m[1, 2] + m[3, 3] + $p_0p_2p_3$ = 7,500 ($(A_1A_2)A_3$)

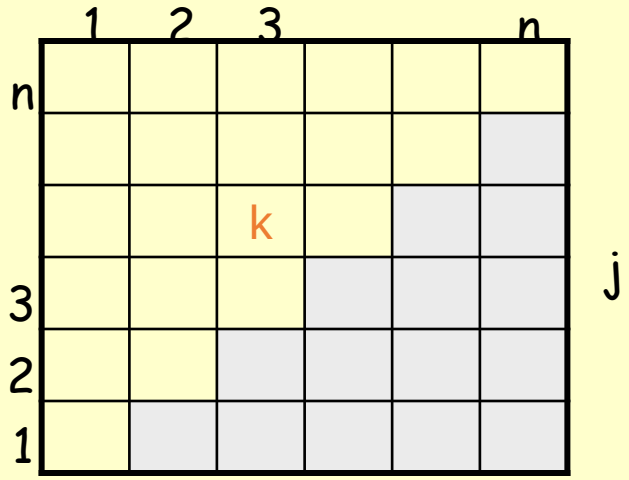
Matrix-Chain-Order(p)

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

$O(n^3)$

4. Construct the Optimal Solution

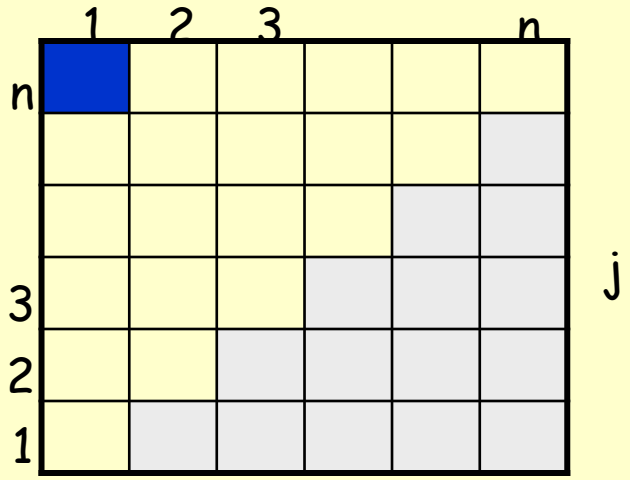
- In a similar matrix s we keep the optimal values of k
- $s[i, j]$ = a value of k such that an optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1}



4. Construct the Optimal Solution

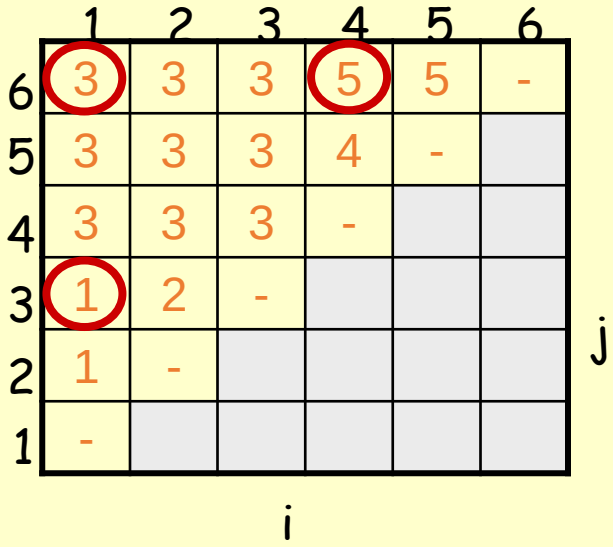
- $s[1, n]$ is associated with the entire product $A_{1..n}$
  - » The final matrix multiplication will be split at k = $s[1, n]$

    $A_{1..n} = A_{1..s[1, n]} \cdot A_{s[1, n]+1..n}$
  - » For each subproduct recursively find the corresponding value of k that results in an optimal parenthesization.



4. Construct the Optimal Solution

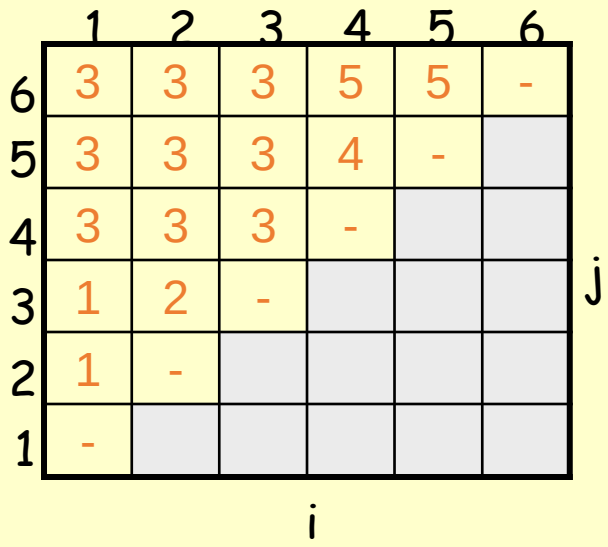
- $s[i, j]$ = value of k such that the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}



- $s[1, n] = 3 \Rightarrow A_{1..6} = A_{1..3} A_{4..6}$
- $s[1, 3] = 1 \Rightarrow A_{1..3} = A_{1..1} A_{2..3}$
- $s[4, 6] = 5 \Rightarrow A_{4..6} = A_{4..5} A_{6..6}$

4. Construct the Optimal Solution (cont.)

```
PRINT-OPT-PARENS(s, i, j)
if i = j
   then print "A"_i
   else print "("
        PRINT-OPT-PARENS(s, i, s[i, j])
        PRINT-OPT-PARENS(s, s[i, j] + 1, j)
        print ")"
```



Example: $A_1 \cdot \cdot \cdot A_6$

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

```
PRINT-OPT-PARENS(s, i, j)
if i = j
   then print "A"i
   else print "("
        PRINT-OPT-PARENS(s, i, s[i, j])
        PRINT-OPT-PARENS(s, s[i, j] + 1,
    j)
        print ")"
```

s[1..6, 1..6]

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

j

i

P-O-P(s, 1, 6) s[1, 6] = 3

i = 1, j = 6 "(" P-O-P (s, 1, 3) s[1, 3] = 1

i = 1, j = 3 "(" P-O-P(s, 1, 1) $\Rightarrow$ "A_1"

P-O-P(s, 2, 3) s[2, 3] = 2

i = 2, j = 3 "(" P-O-P (s, 2, 2) $\Rightarrow$ "A_2"

P-O-P (s, 3, 3) $\Rightarrow$ "A_3"

")"

... ")"

Memoization

- Top-down approach with the efficiency of typical dynamic programming approach
- Maintaining an entry in a table for the solution to each subproblem
  - » **memoize** the inefficient recursive algorithm
- When a subproblem is first encountered its solution is computed and stored in that table
- Subsequent “calls” to the subproblem simply look up that value

Memoized Matrix-Chain

Alg.: MEMOIZED-MATRIX-CHAIN(p)

```
1.  n ← length[p] - 1
2.  for i ← 1 to n
3.      do for j ← i to n
4.          do m[i, j] ← ∞
5.  return LOOKUP-CHAIN(p, 1, n)
```

Initialize the m table with large values that indicate whether the values of m[i, j have been computed

← Top-down approac

Memoized Matrix-Chain

Alg.: LOOKUP-CHAIN(p, i, j)

Running time is $O(n^3)$

```
1.  if m[i, j] < ∞
2.      then return m[i, j]
3.  if i = j
4.     then m[i, j] ← 0
5.     else for k ← i to j – 1
6.                do q ← LOOKUP-CHAIN(p, i, k) +
                          LOOKUP-CHAIN(p, k+1, j) + p_{i-1} p_k p_j
7.                   if q < m[i, j]
8.                      then m[i, j] ← q
9.  return m[i, j]
```

Matrix-Chain Multiplication - Summary

- Both the dynamic programming approach and the memoized algorithm can solve the matrix-chain multiplication problem in $O(n^3)$
- Both methods take advantage of the overlapping subproblems property
- There are only $\Theta(n^2)$ different subproblems
  - » Solutions to these problems are computed only once
- Without memoization the natural recursive algorithm runs in exponential time

Elements of Dynamic Programming

- Optimal Substructure
  - » An optimal solution to a problem contains within it an optimal solution to subproblems
  - » Optimal solution to the entire problem is build in a bottom-up manner from optimal solutions to subproblems
- Overlapping Subproblems
  - » If a recursive algorithm revisits the same subproblems over and over $\Rightarrow$ the problem has overlapping subproblems

Parameters of Optimal Substructure

- How many subproblems are used in an optimal solution for the original problem
  - » Matrix multiplication: Two subproblems (subproducts $A_{i..k}$, $A_{k+1..j}$)
- How many choices we have in determining which subproblems to use in an optimal solution
  - » Matrix multiplication: $j - i$ choices for k (splitting the product)

Parameters of Optimal Substructure

- Intuitively, the running time of a dynamic programming algorithm depends on two factors:
  - » Number of subproblems overall
  - » How many choices we look at for each subproblem

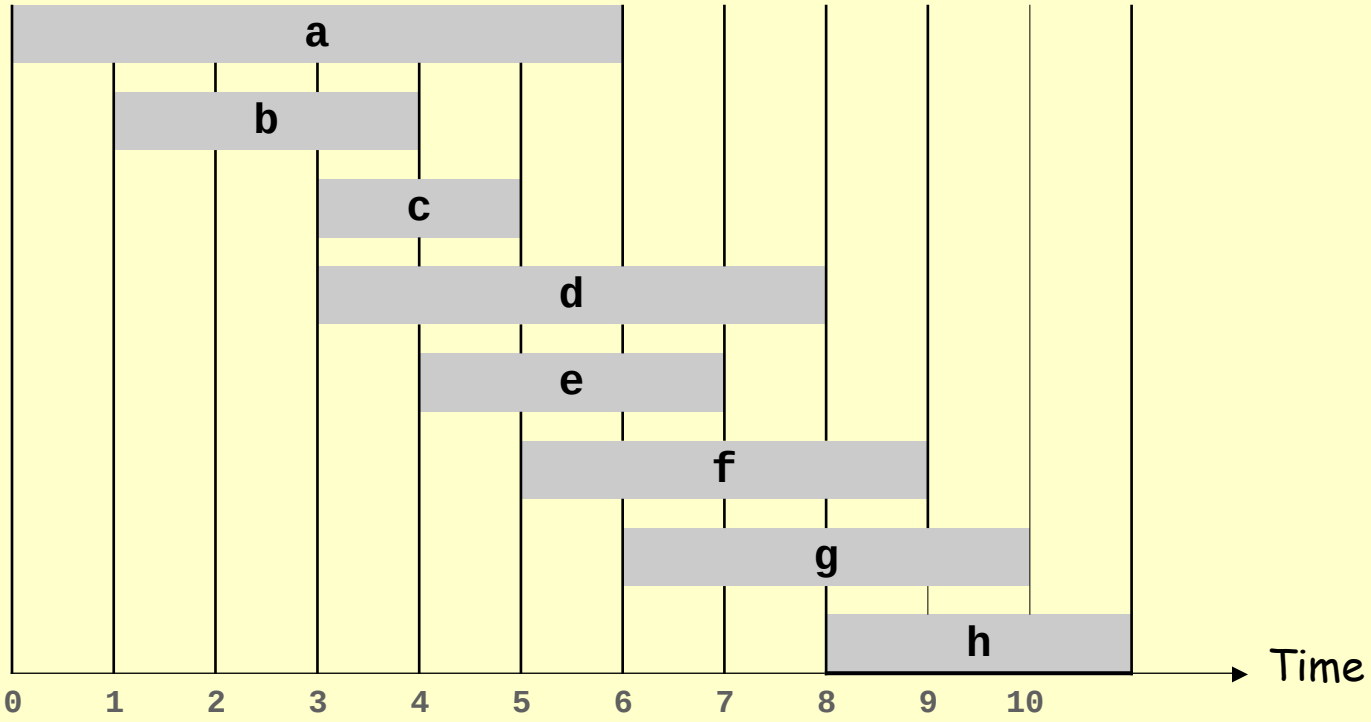
- Matrix multiplication:
  - » $\Theta(n^2)$ subproblems ($1 \le i \le j \le n$) $\Theta(n^3)$ overall
  - » At most $n-1$ choices

Weighted Interval Scheduling

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.

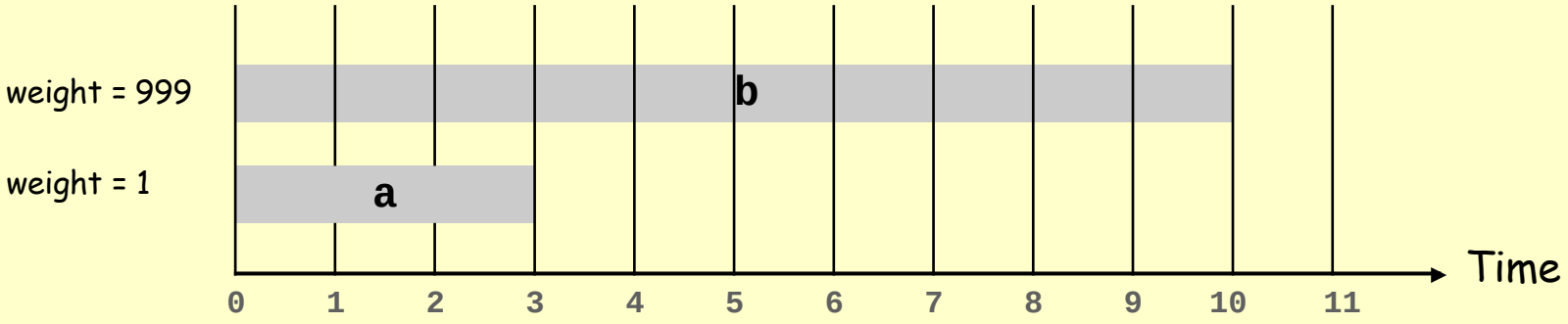


Unweighted Interval Scheduling Review

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.

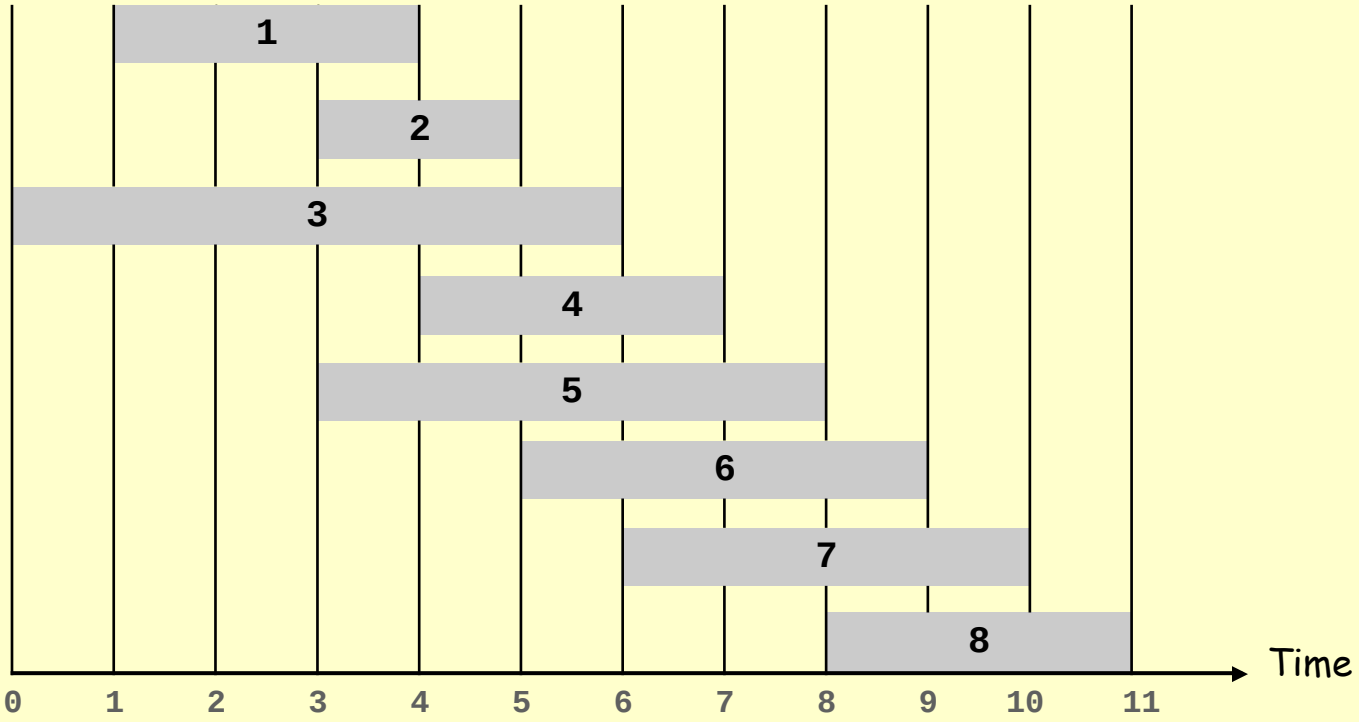


Weighted Interval Scheduling

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 j.

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



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.
  - collect profit v_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: Brute Force

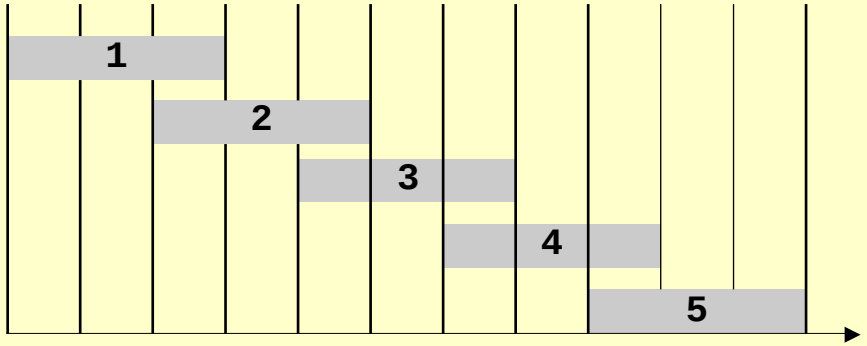
Brute force algorithm.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

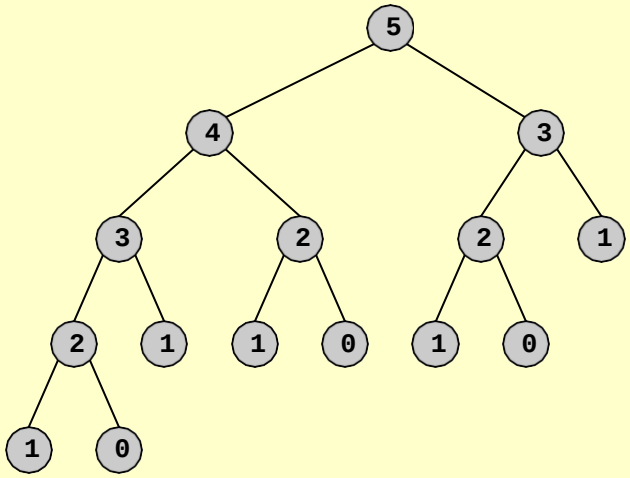
Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2



Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

global array

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing p(j) : $O(n \log n)$ via sorting by start time.

- `M-Compute-Opt(j)`: each invocation takes $O(1)$ time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls

- Progress measure Φ = # nonempty entries of `M[]`.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases Φ by 1 $\Rightarrow$ at most 2n recursive calls.

- Overall running time of `M-Compute-Opt(n)` is $O(n)$. ▪

Remark. $O(n)$ if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

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

$O(n)$

Weighted Interval Scheduling: Finding a Solution

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

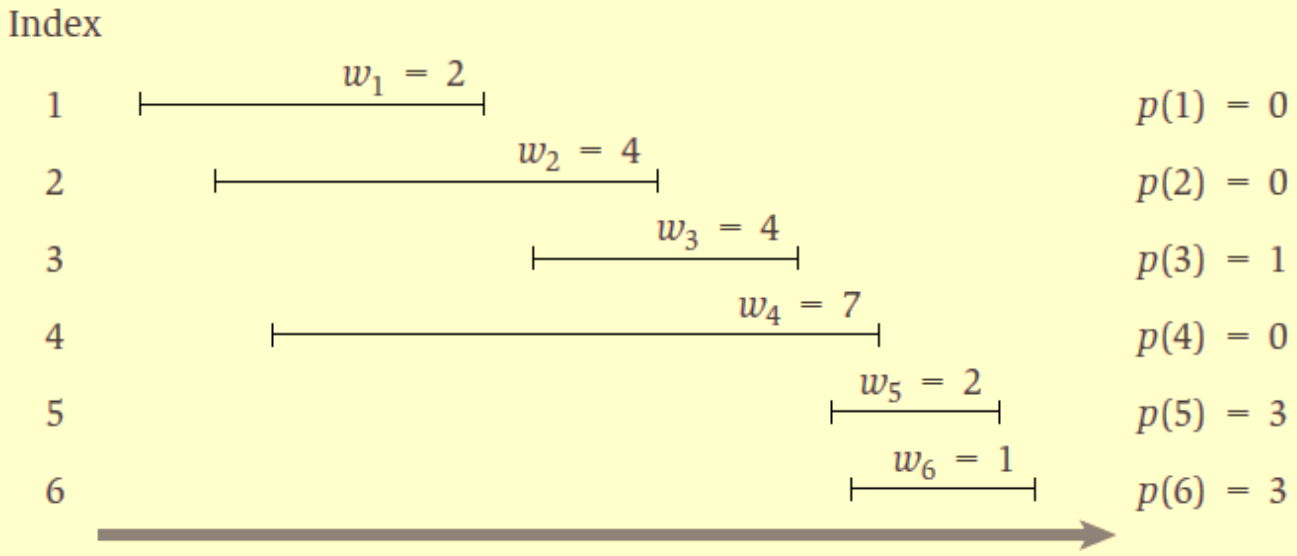
A. Do some post-processing.

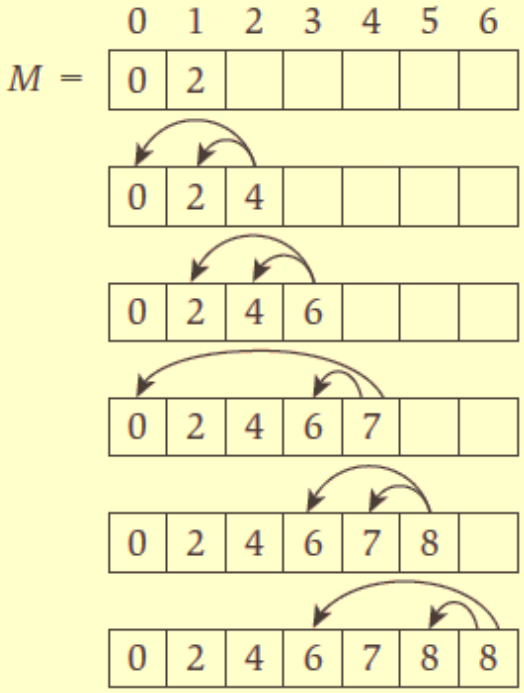
```
Run M-Compute-Opt(n)
Run Find-Solution(n)

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

- # of recursive calls $\leq n \Rightarrow O(n)$.

Weighted Interval Scheduling: EXAMPLE

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



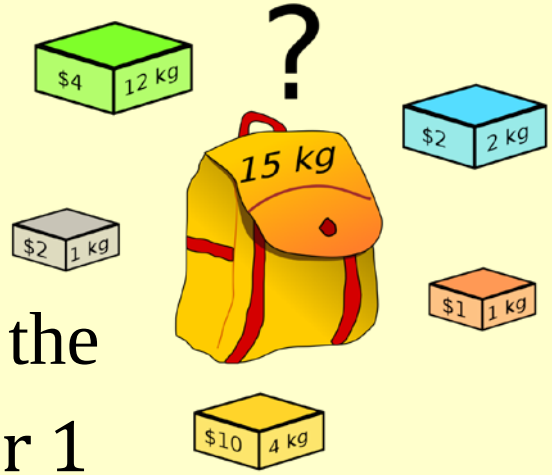


Optimal Solution: 1,3,5

0/1 Knapsack Problem

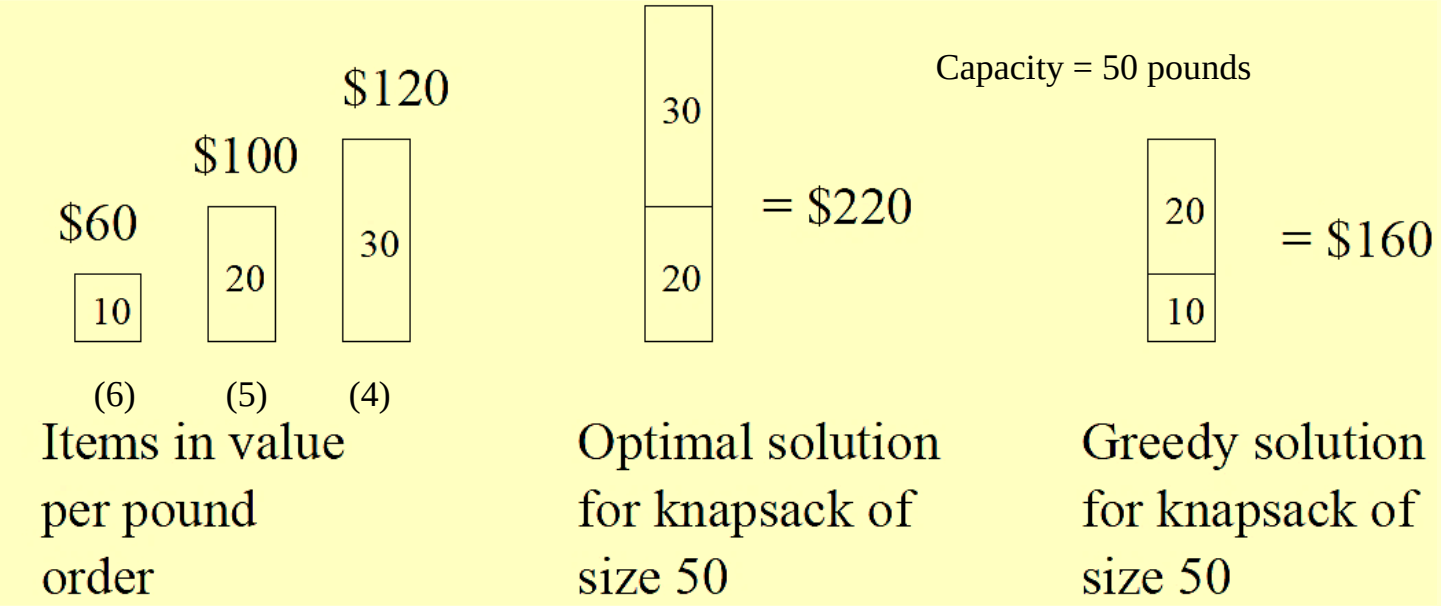
Knapsack Problem

- Given *n* items, with item *i* being valued v_i and having weight w_i units, Fill a knapsack of capacity W units with maximal value.
- If T denote the set of items, the Knapsack problem is defined as follows.
- Objective: maximize $\sum_{i \in T} x_i \cdot v_i$
- Constraint: subject to $\sum_{i \in T} x_i \cdot w_i \leq W$
- In 0/1 Knapsack problem, we either take the whole item or don't take it. i.e ., $x_i = 0$ or 1
- In Fractional Knapsack problem, we can break items for maximizing the total value of knapsack.



Knapsack Problem: Solution

- Perhaps a greedy strategy of picking the item with the biggest value per pound might work?
- Here is a counter-example showing that this does not work:



0/1 Knapsack problem: Brute-force Approach

- We can take all possible combination of items.
- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the maximum value and with total weight less or equal to W.
- Running time will be $O(2^n)$.

0/1 Knapsack problem: Dynamic Programming

- We can improve with an algorithm based on dynamic programming.
- Two key ingredients of optimization problems that lead to a dynamic programming solution:
  - – Optimal substructure: an optimal solution to the problem contains within it optimal solutions to subproblems.
  - – Overlapping subproblems: same subproblem will be visited again and again (i.e., subproblems share subsubproblems).

0/1 Knapsack problem: Optimal Substructure

- Let KNAP(1, *n*, *W*) denote the 0/1 Knapsack problem, choosing objects from [1..*n*] under the capacity constraint of *W*.
- If (x_1, x_2, ..., x_n) is an optimal solution for the problem KNAP(1, *n*, *W*), then:

  Case 1: If x_n = 0 (we do not pick the *n*-th object), then (x_1, x_2, ..., x_{n-1}) must be an optimal solution for the problem KNAP(1, *n*-1, *W*).

  Case 2: If x_n = 1 (we pick the *n*-th object), then (x_1, x_2, ..., x_{n-1}) must be an optimal solution for the problem KNAP(1, *n*-1, *W* - w_n).

0/1 Knapsack problem: Recursive Definition

- 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 ($C[n\text{-}1, W]$, $C[n\text{-}1, W - w_n] + p_n$).
- Similarly,

$C[n\text{-}1, W]$ = max ($C[n\text{-}2, W]$, $C[n\text{-}2, W - w_{n-1}] + p_{n-1}$).

$C[n\text{-}1, W - w_n]$ = max ($C[n\text{-}2, W - w_n]$, $C[n\text{-}2, W - w_n - w_{n-1}] + p_{n-1}$).

0/1 Knapsack problem: Recursive Definition

- For example, if $n = 4$, $W = 9$; $w_4 = 4$, $p_4 = 2$, then $C[4, 9] = \max(C[3, 9], C[3, 9 - 4] + 2)$.
- If we want to compute $C[4, 9]$, $C[3, 9]$ and $C[3, 9 - 4]$ have to be ready.
- Look at the value $C[n, W] = \max (C[n - 1, W], C[n\text{-}1, W - w_n] + p_n)$, to compute $C[n, W]$, we only need the values in the row $C[n - 1, \cdot]$.
- So the table $C[\cdot, \cdot]$ can be built in a bottom up fashion:

  1) compute the first row $C[0, 0]$, $C[0, 1]$, $C[0, 2]$ ... etc;

  2) row by row, fill the table.

0/1 Knapsack problem: Recursive Definition

	1	2	3	4	5	6	7	8	9
1									
2									
3					$C[3, 5]$				$C[3, 9]$
4									$C[4, 9]$

0/1 Knapsack problem: Compute the Optimal Solution

- Let $C[i, w]$ be a cell in the table $C[\cdot, \cdot]$; it represents the value (total profits) of the optimal solution for the problem KNAP(1, i, *w*), which is the subproblem of selecting items in [1..i] subject to the capacity constraint of *w*.

- Then $C[i, w] = \max(C[i-1, w], C[i-1, w - w_i] + p_i)$.

0/1 Knapsack problem: Compute the optimal solution

Boundary Conditions:

- When i = 0; no object to choose, So $C[i, w] = 0$
- When $w = 0$; no capacity available, $C[i, w] = 0$
- When $w_i > w$ the current object i exceeds the capacity, definitely we can not pick it. So, $C[i, w] = C[i - 1, w]$ for this case.

0/1 Knapsack problem: Compute the optimal solution

- Thus overall the recursive solution is:

- $$C[i, w] = \begin{cases} 0, & \text{If } i = 0 \text{ or } w = 0 \\ C[i-1, w], & \text{If } w_i > w \\ \max(C[i-1, w], C[i-1, w-w_i] + p_i), & \text{Otherwise} \end{cases}$$

- The solution (optimal total profits) for the original 0/1 Knapsack problem KNAP(1, n, W) is in $C[n, W]$.

0/1 Knapsack problem: Algorithm

DP-01KNAPSACK(p[], t[], n, W) // n: number of items; W: capacity; p[]: profit array, t[]: weights array.

1. **for** w := 0 to W **do** C[0, w] := 0;
2. **for** i := 0 to n **do** C[i, 0] := 0;
3. **for** i := 1 to n
4. **do for** w := 1 to W
5. **do if** (t[i] > w) // cannot pick item i
6. **then** C[i, w] := C[i - 1, w];
7. **else**
8. **if** ((p[i] + C[i-1,w - t[i]]) > C[i-1, w])
9. **then** C[i, w] := p[i] + C[i - 1, w - t[i]];
10. **else**
11. C[i, w] := C[i - 1, w];

12.return C[n, W];

0/1 Knapsack problem: Complexity

DP-01KNAPSACK(*p*[], *t*[], *n*, *W*) // *n*: number of items; *W*: capacity; *p*[]: profit array, *t*[]: weights array.

```
1.  for w := 0 to W  do C[0, w] := 0;        ← O(W)
2.  for i := 0 to n do C[i, 0] := 0;         ← O(n)
3.  for i := 1 to n                          ← O(n)
4.      do for w := 1 to W                   ← O(W*n)
5.           do if (t[i] > w) // cannot pick item i
6.                then C[i, w] := C[i - 1, w];
7.                else
8.                     if (( p[i] + C[i-1,w - t[i]]) > C[i-1, w])
9.                        then C[i, w] := p[i] + C[i - 1, w - t[i]];
10.                    else
11.                       C[i, w] := C[i - 1, w];
12. return C[n, W];
```

0/1 Knapsack problem: Example

Items (a_i)	a_1	a_2	a_3	a_4
Weights(w_i)	3	2	5	4
Price(p_i)	10	8	12	9

Total Capacity $W = 8$

$$C[i, w] = \begin{cases} 0, & \text{If } i = 0 \text{ or } w = 0 \\ C[i-1, w], & \text{If } w_i > w \\ \max(C[i-1, w], C[i-1, w-w_i] + p_i) & \text{Otherwise} \end{cases}$$

0/1 Knapsack problem: Example

Items (a_i)	a_1	a_2	a_3	a_4
Weights(w_i)	3	2	5	4
Price(p_i)	10	8	12	9

$$C[i, w] = \begin{cases} 0, & \text{If } i = 0 \text{ or } w = 0 \\ C[i-1, w], & \text{If } w_i > w \\ \max(C[i-1, w], C[i-1, w-w_i] + p_i) & \text{Otherwise} \end{cases}$$

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
$p_1 = 10$ 1	0	0	0	10	10	10	10	10	10
$p_2 = 8$ 2	0	0	8	10	10	18	18	18	18
$p_3 = 12$ 3	0	0	8	10	10	18	18	20	22
$p_4 = 9$ 4	0	0	8	10	10	18	18	20	22

0/1 Knapsack problem: Construct the optimal solution

- All of the information we need is in the table.
- $C[n, W]$ is the maximal value/profit of items that can be placed in the Knapsack.

1. Let $i = n$ and $k = W$
2. **While** $i \neq 0$
3. **do if** $C[i, k] \neq C[i - 1, k]$
4. **then** mark the i-th item to be in the knapsack
5. $i = i - 1$, $k = k - w_i$.
6. **else**
7. $i = i - 1$

0/1 Knapsack problem: Construct the optimal solution

Items (a_i)	a_1	a_2	a_3	a_4
Weights(w_i)	3	2	5	4
Price(p_i)	10	8	12	9

$$C[i, w] = \begin{cases} 0, & \text{If } i = 0 \text{ or } w = 0 \\ C[i-1, w], & \text{If } w_i > w \\ \max(C[i-1, w], C[i-1, w-w_i] + p_i) & \text{Otherwise} \end{cases}$$

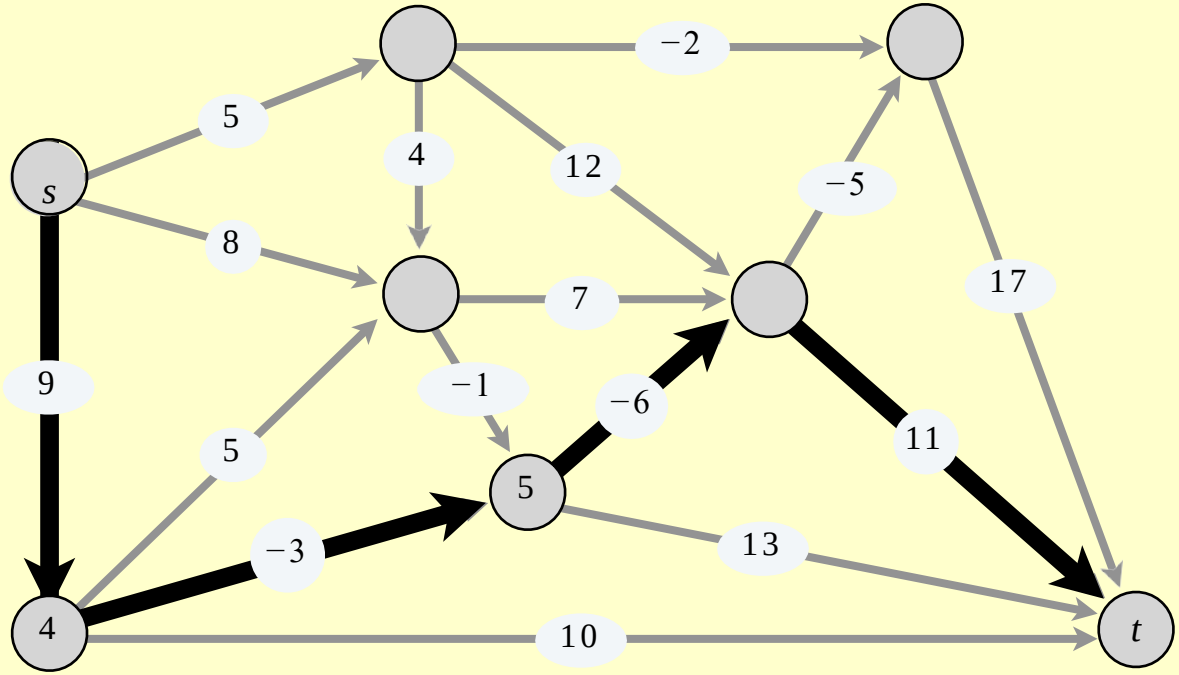
		0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
$p_1 = 10$	1	0	0	0	10	10	10	10	10	10
$p_2 = 8$	2	0	0	8	10	10	18	18	18	18
$p_3 = 12$	3	0	0	8	10	10	18	18	20	22
$p_4 = 9$	4	0	0	8	10	10	18	18	20	22

1, 3

Shortest Paths in a Graph

Shortest paths with negative weights

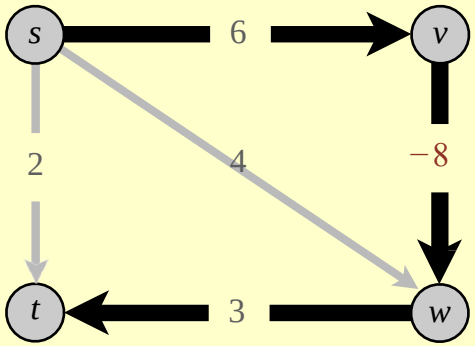
Shortest-path problem. Given a digraph $G = (V, E)$, with arbitrary edge lengths $C_{i,j}$, find shortest path from source node s to destination node t. (assume there exists a path from every node to t)



length of shortest $s \rightsquigarrow t$ path = 9 − 3 − 6 + 11 = 11

Shortest paths with negative weights: failed attempts

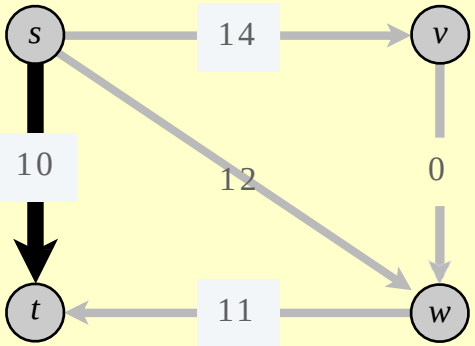
Dijkstra. May not produce shortest paths when edge lengths are negative.



Dijkstra selects the vertices in the order s, t, w, v

But shortest path from s to t is $s \to v \to w \to t$.

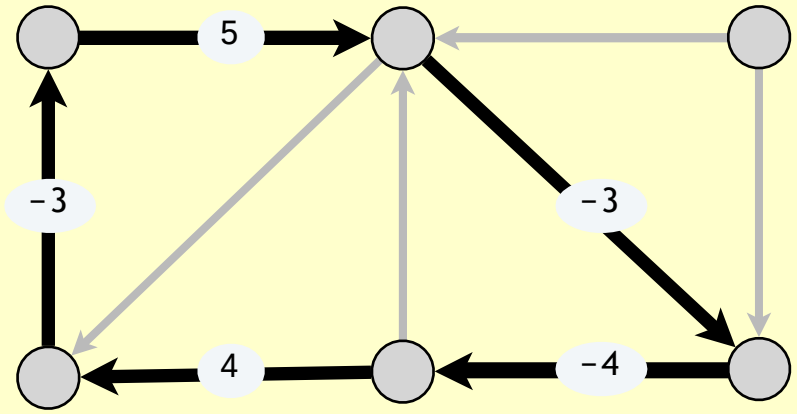
Reweighting. Adding a constant to every edge length does not necessarily make Dijkstra's algorithm produce shortest paths.



Adding 8 to each edge weight changes the shortest path from $s \to v \to w \to t$ to $s \to t$ which is not the shortest path in the actual graph.

Negative cycles

Def. A negative cycle is a directed cycle for which the sum of its edge lengths is negative.

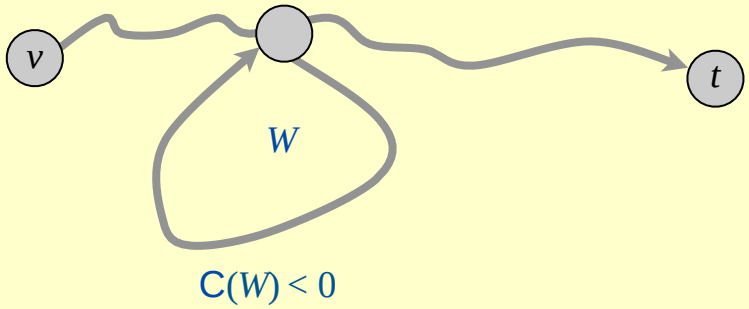


In the given graph, the cycle marked in bold is a negative cycle and the sum of its edge lengths is -1.

Shortest paths and negative cycles

Lemma 1. If some path $v \rightsquigarrow t$ contains a negative cycle, then there does not exist a shortest path $v \rightsquigarrow t$.

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

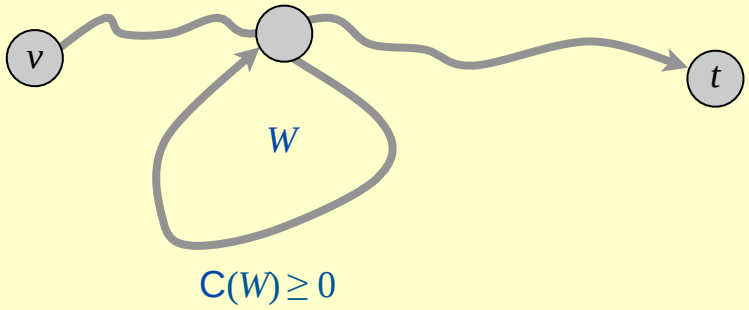


Shortest paths and negative cycles

Lemma 2. If G has no negative cycles, then there exists a shortest path $v \rightsquigarrow t$ that is simple (no repetition of nodes) and has $\leq n - 1$ edges.

Pf.

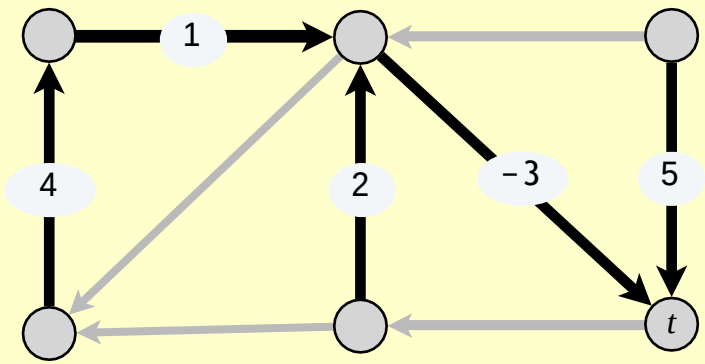
- Among all shortest paths $v \rightsquigarrow t$, consider one that uses the fewest edges.
- If that path P contains a directed cycle W, can remove the portion of P corresponding to W without increasing its length. ▪



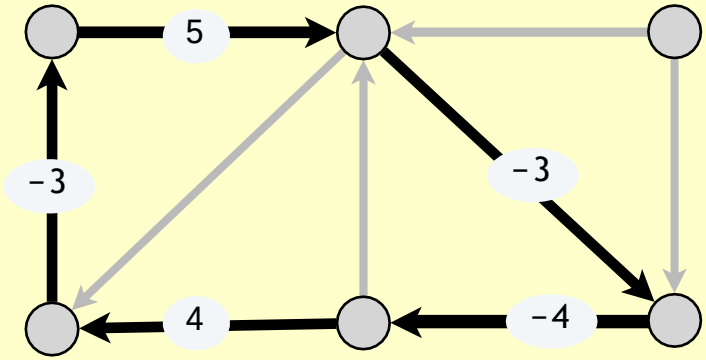
Shortest-paths and negative-cycle problems

Single-destination shortest-paths problem. Given a digraph $G = (V, E)$ with edge lengths $C_{i,j}$ (but no negative cycles) and a distinguished node t, find a shortest path $v \rightsquigarrow t$ for every node v.

Negative-cycle problem. Given a digraph $G = (V, E)$ with edge lengths $C_{i,j}$, find a negative cycle (if one exists).



shortest-paths tree



negative cycle

Shortest paths with negative weights: Dynamic programming

Def. $OPT(i, v)$ = Length of shortest path $v \rightsquigarrow t$ (for any $v \in V$) that uses $\leq i$ edges.

Goal. $OPT(n-1, v)$ for each v.

by Lemma 2, if no negative cycles, there exists a shortest $v \rightsquigarrow t$ path that is simple

Case 1. Shortest path $v \rightsquigarrow t$ uses $\leq i-1$ edges.

- $OPT(i, v) = OPT(i-1, v)$.

optimal substructure property

Case 2. Shortest path $v \rightsquigarrow t$ uses exactly i edges.

- if (v, w) is first edge in shortest such path $v \rightsquigarrow t$, incur a cost of C_{vw}.
- Then, select the best path $w \rightsquigarrow t$ using $\leq i-1$ edges.

Bellman equation.

$$OPT(i,v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = t \\ \infty & \text{if } i = 0 \text{ and } v \neq t \\ \min\left\{ OPT(i-1,v),\ \min_{(v,w)\in E} \{OPT(i-1,w) + C_{vw}\} \right\} & \text{if } i > 0 \end{cases}$$

Shortest paths with negative weights: Bellman-Ford Algorithm

SHORTEST-PATHS(V, E, C, t)

FOREACH node $v \in V$:

  $M[0, v] \leftarrow \infty$.

$M[0, t] \leftarrow 0$.

FOR i = 1 TO $n - 1$

  FOREACH node $v \in V$:

    $M[i, v] \leftarrow M[i-1, v]$.

    FOREACH edge $(v, w) \in E$:

      $M[i, v] \leftarrow \min \{ M[i, v],\ M[i-1, w] + C_{vw} \}$.

$O(V^3)$

$$OPT(i,v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = t \\ \infty & \text{if } i = 0 \text{ and } v \neq t \\ \min \left\{ OPT(i-1,v),\ \min_{(v,w) \in E} \{ OPT(i-1,w) + \right. & \left. \right\} \quad \text{if } i > 0 \end{cases}$$

Shortest paths with negative weights: Improvemnents

Claim. Given a digraph $G = (V, E)$ with no negative cycles, the DP algorithm computes the length of a shortest path $v \rightsquigarrow t$ for every node v in $\Theta(mn)$ time and $\Theta(n^2)$ space.

Finding the shortest paths.

Space optimization. Maintain two 1D arrays (instead of 2D array).

- d[v] = length of a shortest path v$\rightsquigarrow$t that we have found so far.
- *successor*[v] = next node on a v$\rightsquigarrow$t path.

Performance optimization. If $d[w]$ was not updated in iteration $i - 1$, then no reason to consider edges entering w in iteration i.

Bellman–Ford–Moore: Efficient Implementation

BELLMAN–FORD–MOORE(V, E, C, t)

FOREACH node $v \in V$:

  $d[v] \leftarrow \infty$.

  $successor[v] \leftarrow null$.

$d[t] \leftarrow 0$.

FOR $i = 1$ TO $n - 1$

  FOREACH node $w \in V$:

    IF ($d[w]$ was updated in previous pass)

      FOREACH edge $(v, w) \in E$:

        IF ($d[v] > d[w] + C_{vw}$)

          $d[v] \leftarrow d[w] + C_{vw}$.

          $successor[v] \leftarrow w$.

  IF (no $d[\cdot]$ value changed in pass i) STOP.

pass i
$O(m)$ time

Shortest Path: Example