COMP2421—DATA STRUCTURES & ALGORITHMS

Dynamic Programming

Dr. Radi Jarrar
Department of Computer Science
Birzeit University

Slides and material are adapted from George Bebis Analysis of Algorithms at the University of Nevada, Reno

Dynamic Programming

- Dynamic programming is strategy optimize certain classes of algorithms
- Dynamic programming is a technique for efficiently implementing a recursive algorithm by storing partial results
- The idea is to see whether the naive recursive algorithm computes the same subproblems over and over again. If so, storing the answer for each subproblems in a table to look up instead of recompute can lead to an efficient algorithm

Dynamic Programming

- It is based on a caching mechanism that aims to reuse heavy computations
- This caching mechanism is called **memorization**
- Dynamic programming provides good performance benefits when the problem we are trying to solve can be divided into subproblems
- The subproblems partly involve a calculation that is repeated in those subproblems

Dynamic Programming

- The idea is to perform that calculation once (which is the time-consuming step) and then reuse it on the other subproblems
- This is achieved using memorization, which is especially useful in solving recursive problems that may evaluate the same inputs multiple times
- Dynamic programming is a tradeoff of space for time
  - Instead of re-computing a given quantity, it is better to store the results of the initial computation and looking them up instead of recomputing them again

Dynamic Programming

- Dynamic programming is an algorithm design technique (like divide and conquer)
- Divide and conquer
  - Tend to be recursive solutions
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem
- Dynamic programming solutions are non-recursive

Dynamic Programming

- Examples
  - Fibonacci sequence
    - Using dynamic programming will enhance the calculation of the nth number of the Fibonacci sequence
    - Suppose we have a map of objects that maps each call to its value if it was calculated
    - This technique of saving values that have been already calculated is called memorization
  - Binomial Coefficients
  - Job/task scheduling
  - Longest common subsequences
  - Matrix-chain multiplication
- Applicable when subproblems are **not** independent
  - Subproblems share subsubproblems

Fibonacci Numbers by Recursion

- The base cases $F_0 = 0$ and $F_1 = 1$
- Thus, $F_2 = 1$, $F_3 = 2$, and the series continues as 3,5,8,13,21,34,55,89,144
- Recursive solution is

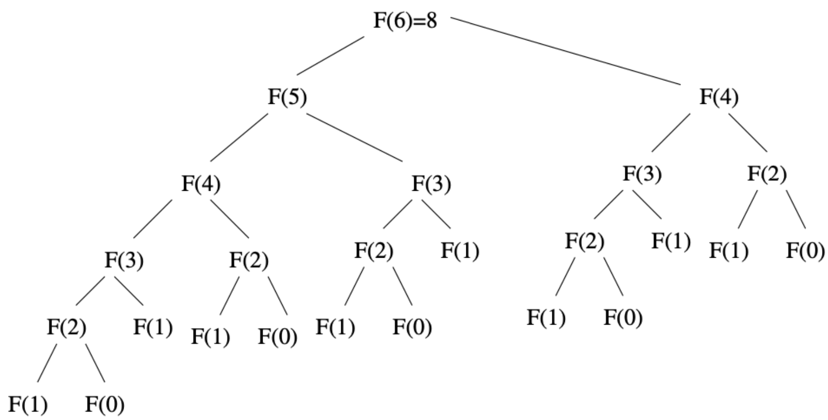
```
int fib_r(int n){
```


Fibonacci Numbers by Recursion

- The base cases $F_0 = 0$ and $F_1 = 1$
- Thus, $F_2 = 1$, $F_3 = 2$, and the series continues as 3,5,8,13,21,34,55,89,144
- Recursive solution is

```
int fib_r(int n){
   if (n == 0)
      return 0;
   if (n == 1)
      return 1;
   return fib_r(n-1) + fib_r(n-2));
}
```


Fibonacci Numbers by Recursion



Fibonacci Numbers by Recursion

- How much time does this algorithm take to compute F(n)?
- The time complexity of Fibonacci series using recursion is $O(2^n)$
- So this program takes exponential time to run

Fibonacci Numbers by Cashing

- Another method to compute the Fibonacci series is by using a cashing technique
- Explicitly store (or cache) the results of each Fibonacci computation F(k) in a table indexed by the parameter k
- The key to avoiding recomputation is to explicitly check for the value before trying to compute it

Fibonacci Numbers by Cashing

```
#define MAXN 45             /* largest interesting n */
#define UNKNOWN -1          /* contents denote an empty cell */
long f[MAXN+1];        /* array for caching computed fib values */
```

```
int fib_c_driver(int n){                int fib_c(int n){
```


Fibonacci Numbers by Cashing

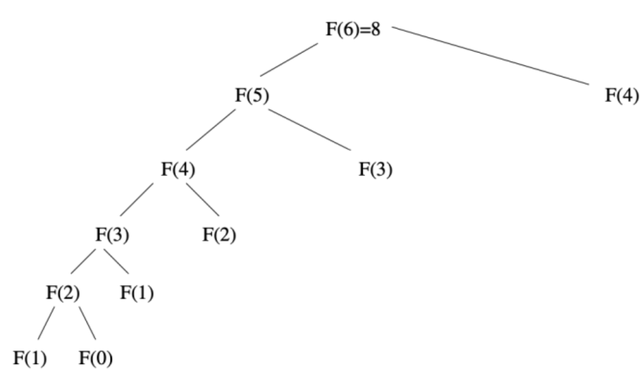
```
#define MAXN 45              /* largest interesting n */
#define UNKNOWN -1           /* contents denote an empty cell */
long f[MAXN+1];        /* array for caching computed fib values */
```

```
int fib_c_driver(int n){
   int i;
   f[0] = 0;
   f[1] = 1;

   for(i=2; i<=n; i++)
         f[i] = UNKNOWN;

   return fib_c(n);
}
```

```
int fib_c(int n){
   if( f[n] == UNKNOWN )
     f[n] = fib_c(n-1) + fib_c(n-2);
   return f[n];
}
```


Fibonacci Numbers by Cashing

- This provides a O(n) solution



Fibonacci Numbers using Dynamic Programming

- In the previous solution we computed the Fibonacci recursively and stored the results in an array
- In DP we need a non-recursive solution

```
int fib_dp(int n) {
```

-

Fibonacci Numbers using Dynamic Programming

- In the previous solution we computed the Fibonacci recursively and stored the results in an array
- In DP we need a non-recursive solution

```
int fib_dp(int n) {
      int i;
      int f[MAXN+1]; /* array to cache computed fib values */

      f[0] = 0;
      f[1] = 1;
      for (i=2; i<=n; i++)
            f[i] = f[i-1]+f[i-2];
      return f[n];
}
```

- This provides O(n) running time

Fibonacci Numbers using Dynamic Programming

- A better solution that does not store all the intermediate values for the entire period of execution
- This is because the recurrence depends on two arguments, so we need to retain the last two values we have seen

```
int fib_dp_2(int n){
   int i;                      /* counter */
   int back2=0, back1=1;       /* last two values of f[n] */
   int next; /* placeholder for sum */
   if (n == 0)
      return 0;
   for (i=2; i<n; i++) {
      next = back1+back2;
      back2 = back1;
      back1 = next;
   }
   return back1+back2;
}
```


BINOMIAL COEFFICIENTS

Combinations

- Another example that utilizes Dynamic Programming is Binomial Coefficients
- Combinations: the binomial coefficients are the most important class of counting numbers

$\binom{n}{k}$ counts the number of ways to choose k things out of n possibilities

$$\binom{n}{k} = n!/((n-k)!k!)$$

- n choose k can be solved using factorials
- However, intermediate calculations can easily cause arithmetic overflow, even when the final coefficient fits comfortably within an integer

Combinations

- A more stable way to compute binomial coefficients is using the recurrence relation implicit in the construction of Pascal's triangle:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5  10   10  5   1
```

- Each number is the sum of the two numbers directly above it

Combinations

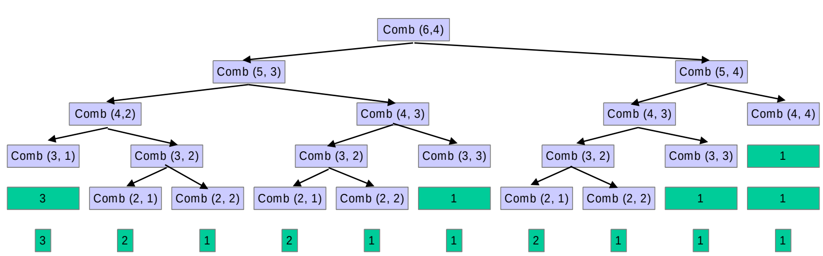
- The recurrence relation implicit in this is that

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

$$\binom{n}{1} = n \qquad \binom{n}{n} = 1$$

- 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

Combinations



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

Combinations

$$\binom{5}{4} = 5$$

```
int binomial_coefficient(int n, int m){
   int i, j; //counters
   int bc[MAXN][MAXN]; /*table of binomial
                         coefficients */
   for (i=0; i<=n; i++)
      bc[i][0] = 1;

   for (j=0; j<=n; j++)
      bc[j][j] = 1;
   for (i=1; i<=n; i++)
      for (j=1; j<i; j++)
         bc[i][j] = bc[i-1][j-1] + bc[i-1][j];

   return bc[n][m] ;
}
```

m / n	0	1	2	3	4	5
0	1					
1	1	1				
2	1	1	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	**5**	1

Dynamic Programming

- Used for **optimization problems**
  - A set of choices must be made to get an optimal solution
  - Find a solution with the optimal value (minimum or maximum)
  - There may be many solutions that lead to an optimal value
  - Our goal: **find an optimal solution**

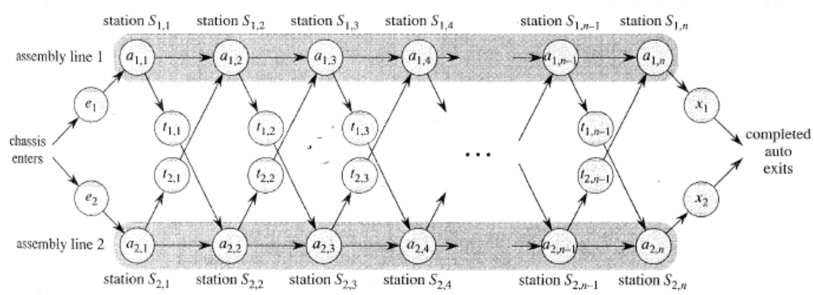
Dynamic Programming Algorithm

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)

ASSEMBLY LINE SCHEDULING

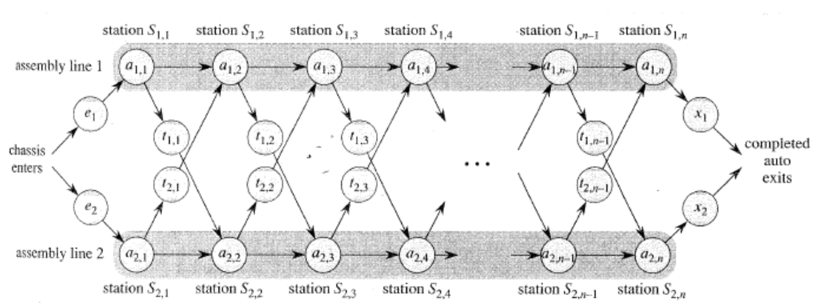
Assembly Line Scheduling

- Automobile factory with two assembly lines
  - Each line has n stations: $S_{1,1}, \ldots, S_{1,n}$ and $S_{2,1}, \ldots, S_{2,n}$
  - Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amounts of time $a_{1,j}$ and $a_{2,j}$
  - Entry times are: e_1 and e_2; exit times are: x_1 and x_2



Assembly Line Scheduling

- After going through a station, can either:
  - stay on same line at no cost, or
  - transfer to other line: cost after $S_{i,j}$ is $t_{i,j}$, $j = 1, \ldots, n - 1$

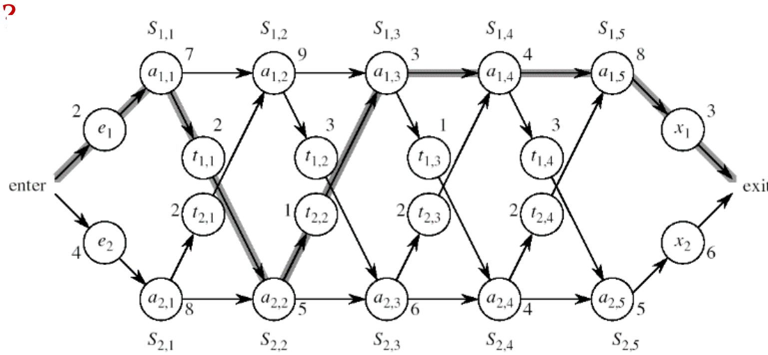


Dr. Radi Jarrar, 2022 29

Assembly Line Scheduling

- Problem:

  What stations should be chosen from line 1 and which from line 2 in order to minimize the total time through the factory for one car?

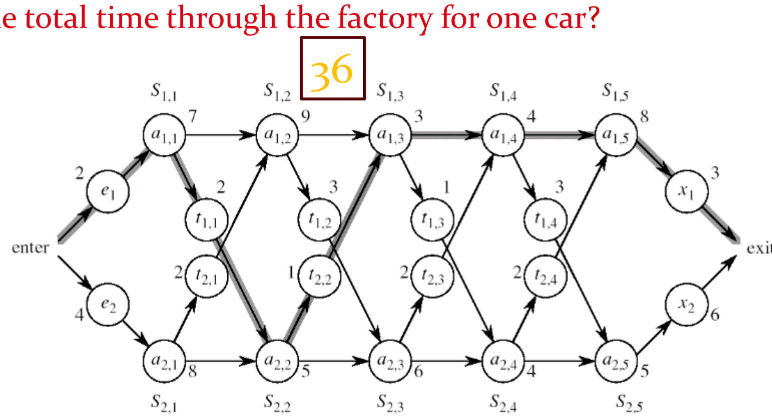


Dr. Radi Jarrar, 2022 30

Assembly Line Scheduling

- Problem:

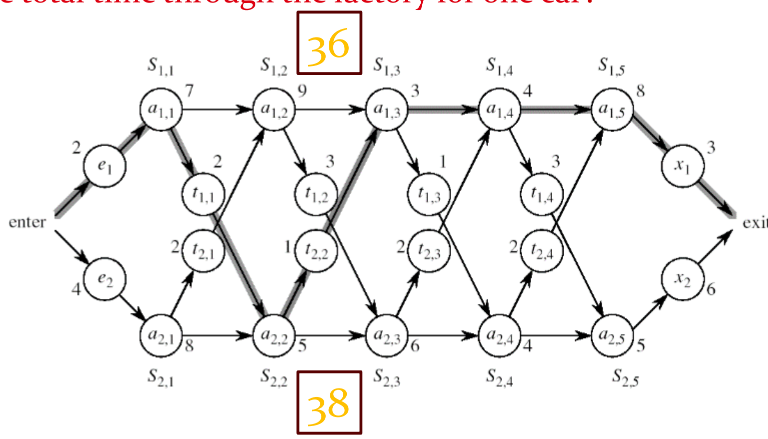
  what stations should be chosen from line 1 and which from line 2 in order to minimize the total time through the factory for one car?



Assembly Line Scheduling

- Problem:

  what stations should be chosen from line 1 and which from line 2 in order to minimize the total time through the factory for one car?

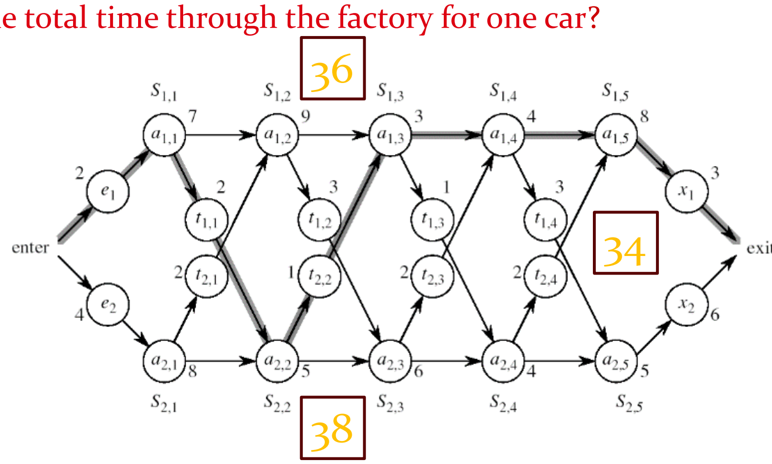


Dr. Radi Jarrar, 2022 32

Assembly Line Scheduling

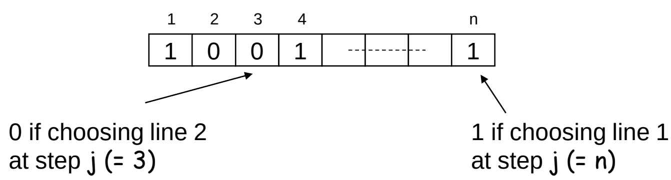
- Problem:

  what stations should be chosen from line 1 and which from line 2 in order to minimize the total time through the factory for one car?



One Solution

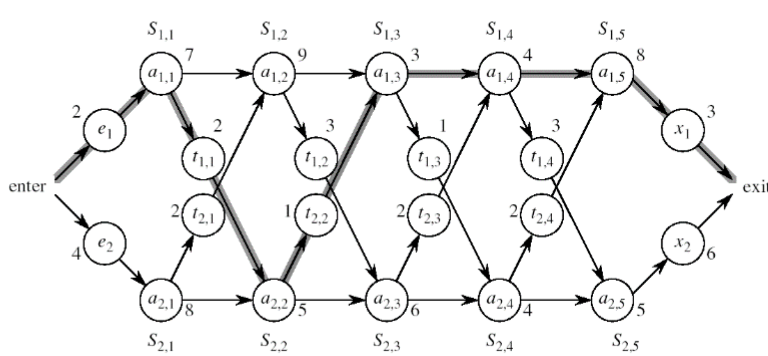
- Brute force
  - Enumerate all possibilities of selecting stations
  - Compute how long it takes in each case and choose the best one
- Solution:



  - There are 2^n possible ways to choose stations
  - Infeasible when n is large!!

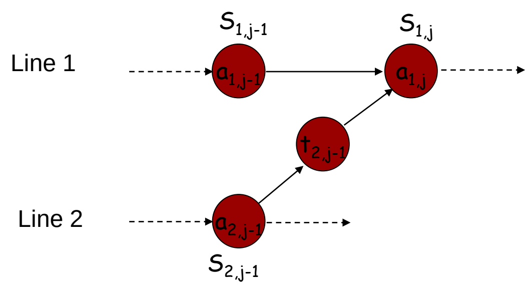
1. Structure of the Optimal Solution

- How do we compute the minimum time of going through a station?



1. Structure of the Optimal Solution

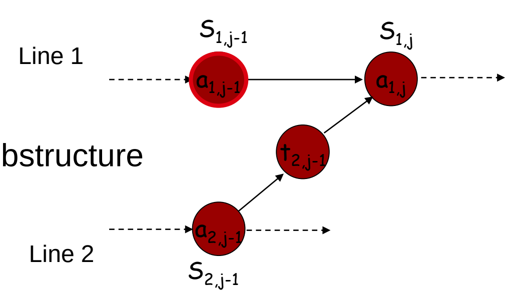
- Let's consider all possible ways to get from the starting point through station $S_{1,j}$
  - We have two choices of how to get to $S_{1,j}$:
    - Through $S_{1,j-1}$, then directly to $S_{1,j}$
    - Through $S_{2,j-1}$, then transfer over to $S_{1,j}$



1. Structure of the Optimal Solution

- Suppose that the fastest way through $S_{1,j}$ is through $S_{1,j-1}$
  - We must have taken a fastest way from entry through $S_{1,j-1}$
  - If there were a faster way through $S_{1,j-1}$, we would use it instead
- Similarly for $S_{2,j-1}$

Optimal Substructure



2. A Recursive Solution

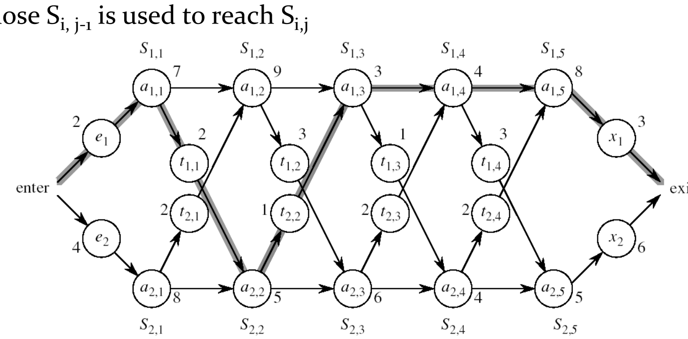
- Generalisation of the problem: an optimal solution to the problem (find the shortest way to $S_{i,j}$) contains optimal solutions to subproblems (find the shortest way to $S_{1,j-1}$ or $S_{2,j-2}$
- This is the optimal substructure property
- This property is used to reconstruct the optimal solution to the problem

2. A Recursive Solution (cont.)

- Define the value of the optimal solution in terms of the optimal solution to subproblems
- Definitions:
  - f* : the fastest time to get through the entire factory
  - $f_i[j]$: the fastest time to get from the starting point through station $S_{i,j}$
  - l*: the line number which is used to exit the factory from the n^{th} station
  - $l_i[j]$: the line number which is (1 or 2) whose $S_{i,\,j-1}$ is used to reach $S_{i,j}$

The objective function is:

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

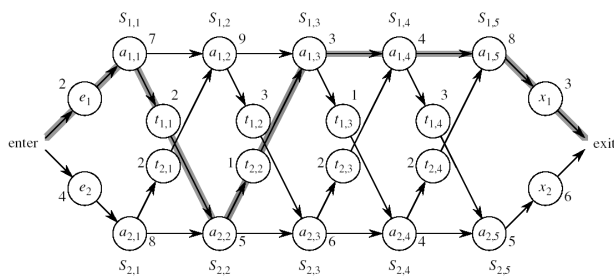


2. A Recursive Solution (cont.)

- Base case: j = 1, i=1,2 (getting through station 1)

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

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



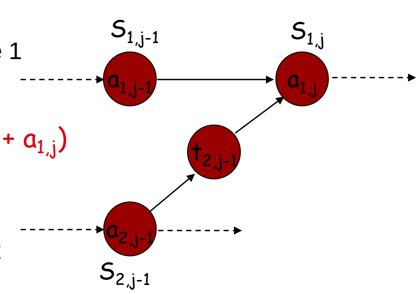
2. A Recursive Solution (cont.)

- General Case: j = 2, 3, …,n, and i = 1, 2
- Fastest way through $S_{1,j}$ is either:
  - the way through $S_{1,j-1}$ then directly through $S_{1,j}$, or

    $f_1[j-1] + a_{1,j}$
  - the way through $S_{2,j-1}$, transfer from line 2 to line 1, then through $S_{1,j}$

    $f_2[j-1] + t_{2,j-1} + a_{1,j}$

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



2. A Recursive Solution (cont.)

Recursively define the value of the optimal solution:

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

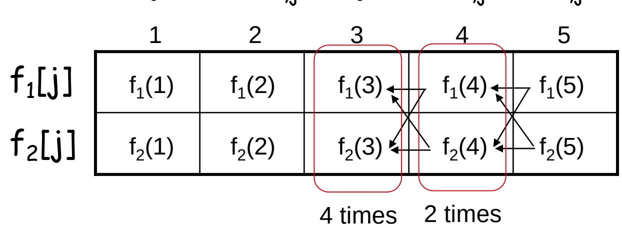
$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

3. Computing the Optimal Solution

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

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

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



- Solving top-down would result in exponential running time

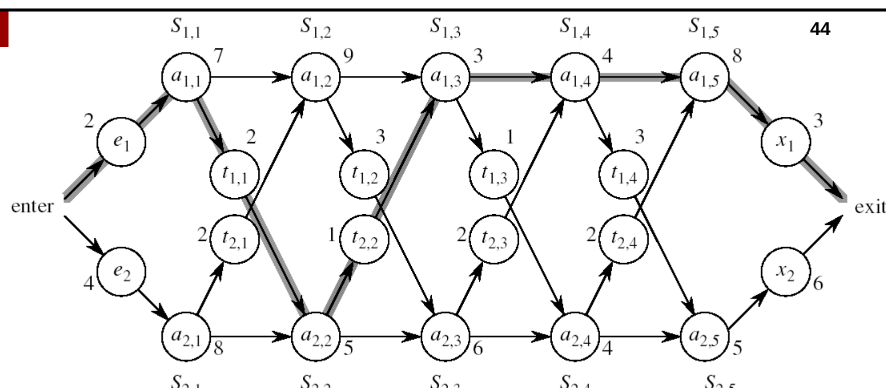
3. Computing the Optimal Solution

- For j ≥ 2, each value $f_i[j]$ depends only on the values of $f_1[j - 1]$ and $f_2[j - 1]$
- Idea: compute the values of $f_i[j]$ as follows:

in increasing order of j →

	1	2	3	4	5
$f_1[j]$					
$f_2[j]$					

- Bottom-up approach
  - First find optimal solutions to subproblems
  - Find an optimal solution to the problem from the subproblems



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

f* = 35
l* = 1

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

$f_1[j]$	1	1	2	1	1
$f_2[j]$	2	1	2	1	2

FASTEST-WAY(a, t, e, x, n)

O(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
      then 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
      then 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
   then f* = f1[n] + x1
        l* = 1
   else f* = f2[n] + x2
        l* = 2
```

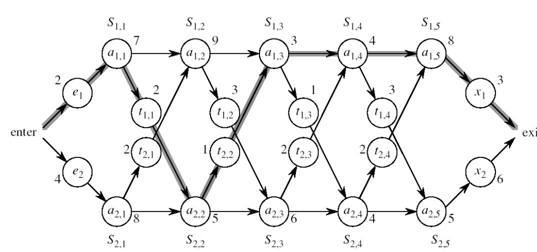
- Lines 1–3: Compute initial values of f_1 and f_2
- Lines 4–8: Compute the values of $f_1[j]$ and $l_1[j]$
- Lines 9–13: Compute the values of $f_2[j]$ and $l_2[j]$
- Lines 14–18: Compute the values of the fastest time through the entire factory

4. Construct an Optimal Solution

The last step is to construct the optimal solution once the optimal solution is calculated

Alg.: PRINT-STATIONS(l, n)

```
i ← l*
print "line " i ", station " n
for j ← n downto 2
    do i ← li[j]
       print "line " i ", station " j - 1
```



From the value of l* we will backtrack into the path

$f_1[j]/l_1[j]$ / $f_2[j]/l_2[j]$	1	2	3	4	5
$f_1[j]/l_1[j]$	9	18[1]	20[2]	24[1]	32[1]
$f_2[j]/l_2[j]$	12	16[1]	22[2]	25[1]	30[2]

l* = 1

MATRIX-CHAIN MULTIPLICATION

Matrix-Chain Multiplication

- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 3)

$$\begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix} \times \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix} = \begin{bmatrix} \ & \ & \ \\ \ & \ & \ \end{bmatrix}$$

2 x 10 + 3 x 40 + 4 x 70 = 420

Matrix-Chain Multiplication

- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 3)

2	3	4
5	6	7

✖

10	20	30
40	50	60
70	80	90

=

420		

2 x 20 + 3 x 50 + 4 x 80 = 510

Matrix-Chain Multiplication

- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 3)

2	3	4
5	6	7

✖

10	20	30
40	50	60
70	80	90

=

420	510	

2 x 30 + 3 x 40 + 4 x 90 = 540

Matrix-Chain Multiplication

- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 3)

2	3	4
5	6	7

✖

10	20	30
40	50	60
70	80	90

=

420	510	540

5 x 10 + 6 x 40 + 7 x 70 = 780

Matrix-Chain Multiplication

- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 3)

2	3	4
5	6	7

✖

10	20	30
40	50	60
70	80	90

=

420	510	540
780		

5 x 20 + 6 x 50 + 7 x 80 = 960

Matrix-Chain Multiplication

- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 3)

2	3	4
5	6	7

×

10	20	30
40	50	60
70	80	90

=

420	510	540
780	960	

5 x 30 + 6 x 60 + 7 x 90 = 1140

Matrix-Chain Multiplication

- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 3)

2	3	4
5	6	7

×

10	20	30
40	50	60
70	80	90

=

420	510	540
780	960	1140

5 x 30 + 6 x 60 + 7 x 90 = 1140

Matrix-Chain Multiplication

- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 3)

2	3	4
5	6	7

✖

10	20	30
40	50	60
70	80	90

=

420	510	540
780	960	1140

- Overall, we have 12 additions and 18 multiplications!
- 3.3^2 = 27 multiplications
- 2.3^2 = 18 additions
- Computing summation in computers is quite faster than multiplication

Matrix-Chain Multiplication

- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 3)

2	3	4
5	6	7

✖

10	20	30
40	50	60
70	80	90

=

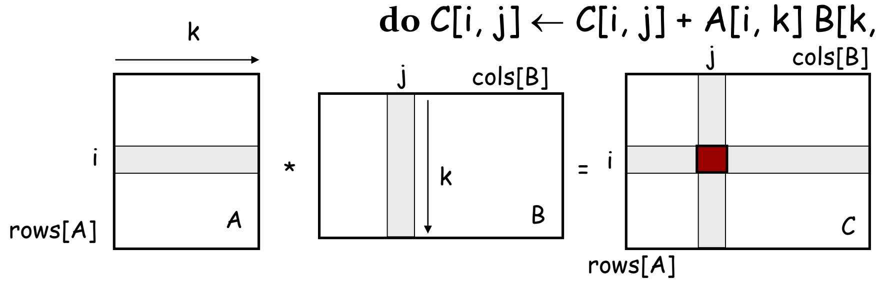
420	510	540
780	960	1140

- So for n x n matrix we will have $n.n^2$ multiplications and $(n-1).n^2$ additions
- So this will result in $O(n^3)$ time complexity

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 (annotation on A[i, k] B[k, j])



Matrix-Chain Multiplication

- Assume we have the following matrices

	A_1	A_2	A_3	A_4	A_5
with sizes	4x10	10x3	3x12	12x20	20x7

- First we check if we can multiply them
  - If the inner dimensions of the adjacent matrices match
- If we want to multiply them from the beginning to end
  - This will take: 4x10x3 + 10x3x12 + 3x12x20 + 12x20x7 = 1784 multiplications
  - Expensive computation!

Matrix-Chain Multiplication

- Assume we have the following matrices

	A_1	A_2	A_3	A_4	A_5
with sizes	4x10	10x3	3x12	12x20	20x7

- First we check if we can multiply them
  - If the inner dimensions of the adjacent matrices match
- If we want to multiply them from the beginning to end
  - This will take: 4x10x3 + 10x3x12 + 3x12x20 + 12x20x7 = 1784 multiplications
  - Expensive computation!

Matrix-Chain Multiplication

- One strategy can be done to reduce the number of multiplications is to parenthesize the product
- 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

Matrix-Chain Multiplication

- Goal: find the optimal way to multiply these matrices to perform the fewest multiplications
- Easy approach: Try them all and pick the most optimal
- Running time would be exponential!

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 x 100 x 5 = 5,000 (10 x 5)

   $((A_1 \cdot A_2) \cdot A_3)$ = 10 x 5 x 50 = 2,500

   Total: 7,500 scalar multiplications

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

   $(A_1 \cdot (A_2 \cdot A_3))$ = 10 x 100 x 50 = 50,000

   Total: 75,000 scalar multiplications

Matrix-Chain Multiplication

- Goal: find the optimal way to multiply these matrices to perform the fewest multiplications
- Easy approach: Try them all and pick the most optimal
- Running time would be exponential!

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.

$$\underset{p_0 \times p_1}{A_1} \cdot \underset{p_1 \times p_2}{A_2} \cdots \underset{p_{i-1} \times p_i}{A_i} \cdot \underset{p_i \times p_{i+1}}{A_{i+1}} \cdots \underset{p_{n-1} \times p_n}{A_n}$$

What is the number of possible parenthesizations?

- Exhaustively checking all possible parenthesizations is not efficient!

1. The Structure of an Optimal Parenthesization

- Notation:

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

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

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

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
- 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

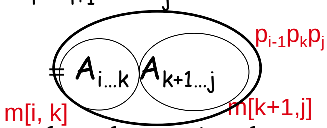
$$A_{i\ldots j} = A_i\, A_{i+1} \cdots A_j \quad \text{for } 1 \le i \le j \le n$$

- Let $m[i, j]$ = the minimum number of multiplications needed to compute $A_{i\ldots j}$
  - full problem ($A_{1..n}$): $m[1, n]$
  - $i = j$: $A_{i\ldots i} = A_i \Rightarrow m[i, i] =$

0, for $i = 1, 2, \ldots, n$

2. A Recursive Solution

- Consider the subproblem of parenthesizing
- $A_{i\ldots j} = A_i A_{i+1} \cdots A_j$ 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 ($i \le k < j$)

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

2. A Recursive Solution (cont.)

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

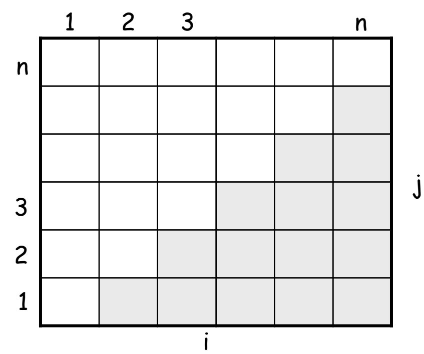
- We do not know the value of k
  - There are j – i possible values for k: k = i, i+1, …, j-1
- Minimizing the cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes:

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

3. Computing the Optimal Costs

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

- Computing the optimal solution recursively takes exponential time!
- How many subproblems?
  $\Rightarrow O(n^2)$
  - Parenthesize $A_{i…j}$ for $1 \le i \le j \le n$
  - One problem for each choice of i and j



3. Computing the Optimal Costs (cont.)

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

- 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]

$$A_{i…j} = A_{i…k} A_{k+1…j}$$

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

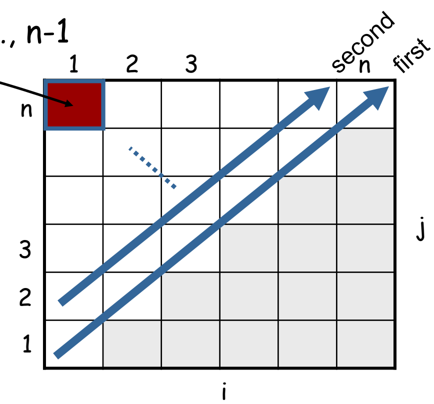
3. Computing the Optimal Costs (cont.)

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

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

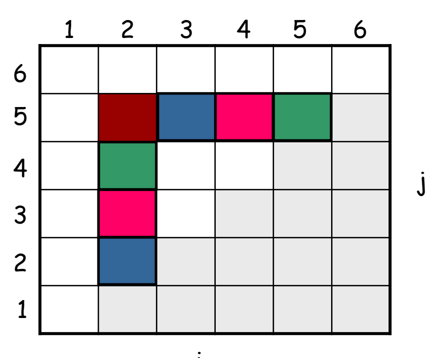
m[1, n] gives the optimal solution to the problem

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

Compute $A_1 \cdot A_2 \cdot A_3$

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

	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)

$= 0 + 0 + 100 * 5 * 50 = 25{,}000$

$m[1, 3] = \min \begin{cases} m[1, 1] + m[2, 3] + p_0p_1p_3 = 75{,}000 & (A_1(A_2A_3)) \\ m[1, 2] + m[3, 3] + p_0p_2p_3 = 7{,}500 & ((A_1A_2)A_3) \end{cases}$

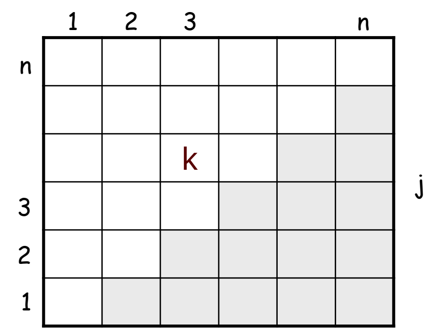
Matrix-Chain-Order(p)

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
    do m[i, i] ← 0
for l ← 2 to n              ▷ l is the chain length.
    do for i ← 1 to n − l + 1
           do j ← i + l − 1
              m[i, j] ← ∞
              for k ← i to j − 1
                  do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                     if q < m[i, j]
                        then m[i, j] ← q
                             s[i, j] ← k
return m and s
```

$O(N^3)$

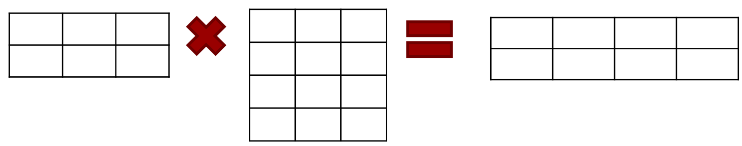
4. Construct the Optimal Solution

- 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}

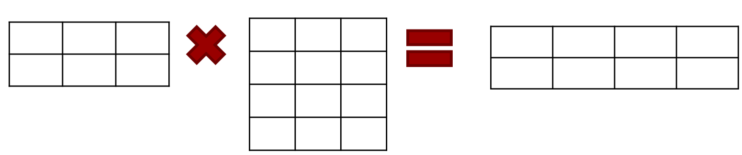


- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 4)

- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 4)

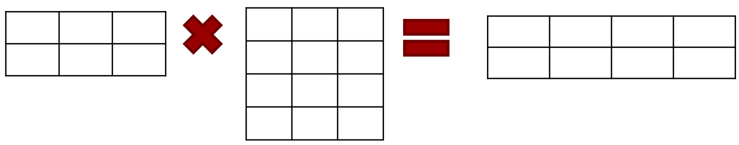


- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 4)



- # of multiplications to get one element in resultant matrix =3

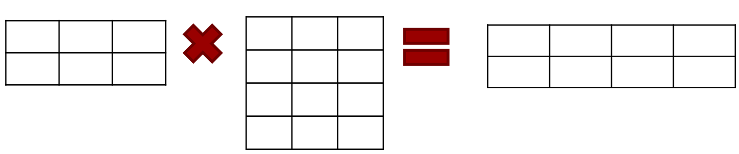
- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 4)



- # of multiplications to get one element in resultant matrix =3
- # of multiplications to get one row in resultant matrix = 3 * 4 = 12

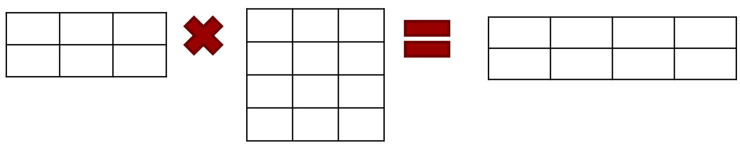
Dr. Radi Jarrar, 2022

- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 4)



- # of multiplications to get one element in resultant matrix =3
- # of multiplications to get one row in resultant matrix = 3 * 4 = 12
- # of multiplications to get all elements in resultant matrix = cost(A1, A2) = 3 * 4 * 2 = 24

- Consider 2 matrices A1 and A2 of sizes (2, 3) and (3, 4)



- # of multiplications to get one element in resultant matrix =3
- # of multiplications to get one row in resultant matrix = 3 * 4 = 12
- # of multiplications to get all elements in resultant matrix = cost(A1, A2) = 3 * 4 * 2 = 24 → cost of multiplying A1 and A2

Dr. Radi Jarrar, 2022

Problem

- In matrix chain multiplication problem, we need to find the minimum cost of when multiplying more than one matrix

Problem

- In matrix chain multiplication problem, we need to find the minimum cost of when multiplying more than one matrix
- Consider the following 4 matrices A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)

Dr. Radi Jarrar, 2022

Problem

- In matrix chain multiplication problem, we need to find the minimum cost of when multiplying more than one matrix
- Consider the following 4 matrices A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- The number of possible combinations to perform A1*A2*A3*A4 is
- (A1*A2)*(A3*A4) or (A1)*(A2*A3*A4) or (((A1*A2)*A3)*A4) …

Problem

- In matrix chain multiplication problem, we need to find the minimum cost of when multiplying more than one matrix
- Consider the following 4 matrices A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- The number of possible combinations to perform A1*A2*A3*A4 is
- (A1*A2)*(A3*A4) or (A1)*(A2*A3*A4) or (((A1*A2)*A3)*A4) …
- Time complexity of this method is 2nCn/(n+1) = 2(4)C(3)/5 = 336

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- Consider the following table M of size n x n (n is the number of matrices)

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- Consider the following table M of size n x n (n is the number of matrices)
- n = 4
- Initialise M[i, j] = 0, where i == j
- M[i, j] = MIN{ M[i, k] + M[k+1, j] + d(i-1)*d(k)*d(j)}
- where d(k) is the dimension of matrix k

	1	2	3	4
1				
2				
3				
4				

Dr. Radi Jarrar, 2022

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- Consider the following table M of size n x n (n is the number of matrices)
- n = 4
- Initialise M[i, j] = 0, where i == j
- M[i, j] = MIN{ M[i, k] + M[k+1, j] + d(i-1)*d(k)*d(j)}
- where d(k) is the dimension of matrix k and i≤k<j

	1	2	3	4
1	0			
2		0		
3			0	
4				0

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- M[1, 2] = A1 x A2 = 5 x 4 x 6 = 120

	1	2	3	4
1	0			
2		0		
3			0	
4				0

Dr. Radi Jarrar, 2022

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- M[1, 2] = A1 x A2 = 5 x 4 x 6 = 120

	1	2	3	4
1	0	120		
2		0		
3			0	
4				0

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- M[1, 2] = A1 x A2 = 5 x 4 x 6 = 120
- M[2, 3] = A2 x A3 = 4 x 6 x 2 = 48

	1	2	3	4
1	0	120		
2		0		
3			0	
4				0

Dr. Radi Jarrar, 2022

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- M[1, 2] = A1 x A2 = 5 x 4 x 6 = 120
- M[2, 3] = A2 x A3 = 4 x 6 x 2 = 48

	1	2	3	4
1	0	120		
2		0	48	
3			0	
4				0

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- M[1, 2] = A1 x A2 = 5 x 4 x 6 = 120
- M[2, 3] = A2 x A3 = 4 x 6 x 2 = 48
- M[3, 4] = A3 x A4 = 6 x 2 x 7 = 84

	1	2	3	4
1	0	120		
2		0	48	
3			0	
4				0

Dr. Radi Jarrar, 2022

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- M[1, 2] = A1 x A2 = 5 x 4 x 6 = 120
- M[2, 3] = A2 x A3 = 4 x 6 x 2 = 48
- M[3, 4] = A3 x A4 = 6 x 2 x 7 = 84

	1	2	3	4
1	0	120		
2		0	48	
3			0	84
4				0

Dr. Radi Jarrar, 2022

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- M[1, 3] = M[1, 1] + M[2, 3] + 5 X 4 X 2
  = 0 + 48 + 40 = 88
- M[1, 3] = M[1, 2] + M[3, 3] + 5 X 6 X 2
  = 120 + 0 + 60 = 180
- M[1, 3] = MIN{88, 180} = 88

	1	2	3	4
1	0	120		
2		0	48	
3			0	84
4				0

Dr. Radi Jarrar, 2022

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- M[1, 3] = M[1, 1] + M[2, 3] + 5 X 4 X 2
  = 0 + 48 + 40 = 88
- M[1, 3] = M[1, 2] + M[3, 3] + 5 X 6 X 2
  = 120 + 0 + 60 = 180
- M[1, 3] = MIN{88, 180} = 88

	1	2	3	4
1	0	120	88	
2		0	48	
3			0	84
4				0

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- M[2, 4] = M[2, 2] + M[3, 4] + 4 X 6 X 7
  = 0 + 84 + 168 = 252
- M[2, 4] = M[2, 3] + M[4, 4] + 4 X 2 X 7
  = 48 + 0 + 56 = 104
- M[2, 4] = MIN{252, 104} = 104

	1	2	3	4
1	0	120	88	
2		0	48	
3			0	84
4				0

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- M[2, 4] = M[2, 2] + M[3, 4] + 4 X 6 X 7
  = 0 + 84 + 168 = 252
- M[2, 4] = M[2, 3] + M[4, 4] + 4 X 2 X 7
  = 48 + 0 + 56 = 104
- M[2, 4] = MIN{252, 104} = 104

	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- M[1, 4] = M[1, 1] + M[2, 4] + 5 X 4 X 7
  = 0 + 104 + 140 = 244
- M[1, 4] = M[1, 2] + M[3, 4] + 5 X 6 X 7
  = 120 + 84 + 210 = 414
- M[1, 4] = M[1, 3] + M[4, 4] + 5 X 2 X 7
  = 88 + 0 + 70 = 158
- M[1, 4] = MIN{244, 414, 158} = 158

	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

Dr. Radi Jarrar, 2022

Problem – Using Dynamic Programming

- A1 = (5 x 4), A2 = (4 x 6), A3 = (6 x 2), and A4 = (2 x 7)
- M[1, 4] = M[1, 1] + M[2, 4] + 5 X 4 X 7
  = 0 + 104 + 140 = 244
- M[1, 4] = M[1, 2] + M[3, 4] + 5 X 6 X 7
  = 120 + 84 + 210 = 414
- M[1, 4] = M[1, 3] + M[4, 4] + 5 X 2 X 7
  = 88 + 0 + 70 = 158
- M[1, 4] = MIN{244, 414, 158} = 158

	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

Problem – Using Dynamic Programming

- So the minimum number of multiplications required for these matrices is 158

	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

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 approach

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) + $p_{i-1}p_kp_j$
7. **if** q < m[i, j]
8. **then** m[i, j] ← q
9. **return** m[i, j]

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

LONGEST INCREASING SUBSEQUENCE

Longest Increasing Subsequence

- The Longest Increasing Subsequence (LIS) is the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order
- For example given S = {2,4,3,5,1,7,6,9,8}
  - What is the Longest Increasing Subsequence?

Longest Increasing Subsequence

- The Longest Increasing Subsequence (LIS) is the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order
- For example given S = {2,4,3,5,1,7,6,9,8}
  - What is the Longest Increasing Subsequence?
- The length of 5 with {2, 4, 5, 6, 8}

Longest Increasing Subsequence

- The Longest Increasing Subsequence (LIS) is the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order
- For example given S = {2,4,3,5,1,7,6,9,8}
  - What is the Longest Increasing Subsequence?
- The length of 5 with {2, 4, 5, 6, 8}
- There are 8 more with this length!

Longest Increasing Subsequence

- Given the following list {10, 22, 9, 33, 21, 50, 41, 60, 80}, what is the length of the Longest Increasing Subsequence?

Longest Increasing Subsequence

- Given the following list {10, 22, 9, 33, 21, 50, 41, 60, 80}, what is the length of the Longest Increasing Subsequence?
- The length is 6 and LIS is {10, 22, 33, 50, 60, 80}

Longest Increasing Subsequence

- Finding the longest increasing run in a numerical sequence is straightforward
- Indeed, you should be able to devise a linear-time algorithm easily
- To apply dynamic programming, we need to construct a recurrence that computes the length of the longest sequence
- To find the right recurrence, ask what information about the first n − 1 elements of S would help you to find the answer for the entire sequence?

Longest Increasing Subsequence

- *Recursion:*

1. Find the possible subsequences for the current number
2. If the current item is greater than the previous element in the subsequence, include the current item in the subsequence and recur for the remaining items
3. Exclude the current item from the sequence and recur for the remaining items
4. Return the maximum value reached by including or excluding the current item.

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

- Increasing subsequences:

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}
- Increasing subsequences:

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

Exclude (less than 3)

- Increasing subsequences:
- {3}

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

Include (6 is greater than 3)

- Increasing subsequences:
- {3, 6}

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

Exclude

- Increasing subsequences:
- {3, 6}

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

Exclude

- Increasing subsequences:
- {3, 6}

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

Exclude

- Increasing subsequences:
- {3, 6}

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

Start

- Increasing subsequences:
- {3, 6}
- {2}

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

Include

- Increasing subsequences:
- {3, 6}
- {2, 6}

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

Exclude

- Increasing subsequences:
- {3, 6}
- {2, 6}

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

Exclude

- Increasing subsequences:
- {3, 6}
- {2, 6}

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

Exclude

- Increasing subsequences:
- {3, 6}
- {2, 6}

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

Exclude

- Increasing subsequences:
- {3, 6}
- {2, 6}

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

- Increasing subsequences:
- {3, 6}
- {2, 6}
- {2, 4, 5}
- {5}
- {1}
- ...etc

Longest Increasing Subsequence

- Example:
- S={3, 2, 6, 4, 5, 1}

- Increasing subsequences:
- {3, 6}
- {2, 6}
- {2, 4, 5} longest with length 3
- {5}
- {1}
- ...etc

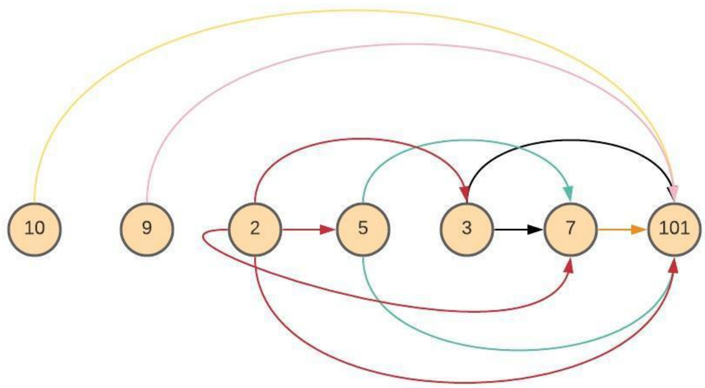
Longest Increasing Subsequence

```
int LIS(int arr[], int i, int n, int prev)
{
	// Base case - empty list
	if (i == n) return 0;

	//case 1-exclude the current element and process the remaining
elements
	int exclude = LIS(arr, i + 1, n, prev);

	// case 2-include the current element if it is greater than previous
	element in LIS
	int include = 0;
	if (arr[i] > prev)
		include = 1 + LIS(arr, i + 1, n, arr[i]);

	// return maximum of above two choices
		return max(include, exclude);

}
```

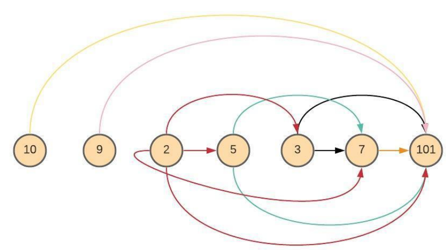

Longest Increasing Subsequence

- One approach to find the LIS is to create a Directed Acyclic Graph (DAG) with each element acting as a node
- An edge between every pair of ordered nodes. Being able to visualize the DAG will make solving problems easier
- The problems will generally ask to form a sequence or chain of elements. The solution to the problem is found by choosing the path that has the most number of edges in the DAG.

Longest Increasing Subsequence



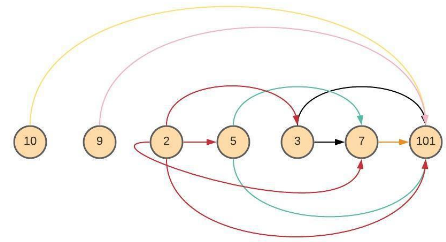
Longest Increasing Subsequence



- As for the DAG above, note the following:

1. We should order elements as (p, q) if q > p. In the above diagram for the node 2, the ordered pairs are (2, 5), (2, 3), (2, 7), (2, 101). Each of these ordered ordered pair has an edge.
2. We are asked to form the longest chain/subsequence of increasing elements. The solution to the problem is the path that has the most number of edges (2, 5) -> (5, 7) -> (7, 101). This path gives the longest increasing subsequence as [2, 5, 7, 101]

Longest Increasing Subsequence



- As for the DAG above, note the following:

1. We should order elements as (p, q) if q > p. In the above diagram for the node 2, the ordered pairs are (2, 5), (2, 3), (2, 7), (2, 101). Each of these ordered ordered pair has an edge.
2. We are asked to form the longest chain/subsequence of increasing elements. The solution to the problem is the path that has the most number of edges (2, 5) -> (5, 7) -> (7, 101). This path gives the longest increasing subsequence as [2, 5, 7, 101]

