DESIGN AND ANALYSIS OF ALGORITHMS
Module 3

SPACE AND TIME TRADE-OFFS: Introduction, Sorting by Counting, Input Enhancement in String Matching, Horspool, Hashing

DYNAMIC PROGRAMMING: Introduction, Warshall's and Floyd's Algorithms, analysis. Matrix-chain multiplication, analysis, Longest common subsequence, Analysis.

SPACE AND TIME TRADE-OFFS:

Introduction

- An algorithm must be time efficient and space efficient
- To achieve both may not be possible for some algorithms
- In some situations space may be an important factor or time.
- Space and time tradeoff is a situation in which either time efficiency is achieved at the cost of extra memory usage or space efficiency can be achieved at the cost of execution speed.
- Methods using which time efficiency is achieved at the cost of extra space
  1. Input Enhancement
  2. Pre structuring
  3. Dynamic Programming

Sorting by Counting

Input Enhancement

Given a problem and various inputs the input is pre -processed to get additional information about the problem. The additional information obtained may be stored in the form of table which may be used by an algorithm to get required results in less time.

Sorting by Counting

Two methods:
1. Sorting by comparison
2. Sorting by Distribution

Sorting by comparison
- **For each element a[i] in the given list find the total number of elements say c[i] that are less than a[i].**
- **The count c[i] obtained in step 1 will be the position of a[i] in the final sorted list.**

Eg:

a	**0**	**1**	**2**	**3**	**4**	**5**
Value	**25**	**45**	**10**	**20**	**50**	**15**

c	**0**	**1**	**2**	**3**	**4**	**5**
Value	**3**	**4**	**0**	**2**	**5**	**1**

B	0	1	2	3	4	5
Value	**10**	**15**	**20**	**25**	**45**	**50**

Algorithm:

```
 for i□0 to n-1
   c[i]□0

for i□ 0 to n-2
  for j□i+1 to n-1 do
   if(a[i]<a[j])

c[j]□c[j]+1
   else
 c[i]□c[i]+1

end if
    end for
end for

for i□0 to n-1
b[c[i]]□a[i]
```

Time Complexity: n^2

Sorting by Distribution

- In this method frequency of each element is calculated and accumulated frequency of each element is calculated.
- Obtain the distribution value.
- The elements in the array must be in the range without missing any number.
- It is applicable if same elements are repeated many times.

Algorithm:

```
lb□min(a,n)
ub□max(a,n)

for i□0 to ub-lb
 d[i]□0
end for

for i□0 to n-1    // computing the frequency
 j□a[i]-lb
 d[j]□ d[j]+1
end for
```

for i□ 1 to ub-lb
d[i]□ d[i]+d[i-1]
end for

for i□n-1 down to 0
j□a[i]-lb
d[j]□d[j]-1
b[d[j]]□a[i]
end for

Efficiency : for i□n-1 down to 0
b[d[j]]□a[i]

$$\sum_{i=0}^{n-1} 1 = n - 1 - 0 + 1 = O(n)$$

Eg:

a	0	1	2	3	4	5	6	7	8
value	12	13	10	12	10	12	11	10	13

Ub=13
Lb=10

0	1	2	3
10	11	12	13

Frequency

0	1	2	3
3	1	3	2

D

0	1	2	3
0	3	4	7

Accumulated Frequency

B	0	1	2	3	4	5	6	7	8
value	10	10	10	11	12	12	12	13	13

Enhancement in String Matching

Horspool Algorithm for string matching

Recall the string matching problem, a pattern $P[0...m-1]$ is searched for in a text $T[0...n-1]$. The brute force algorithm in worst case makes $m(n-m+1)$ comparisons, so the cost is $\Theta(nm)$. But on average only a few comparisons are made before shifting the pattern, so the cost is $\Theta(n)$. We consider two algorithms that also achieve this cost.

Horspool's Algorithm

Horspool's algorithm shifts the pattern by looking up shift value in the character of the text aligned with the last character of the pattern in table made during the initialization of the algorithm. The pattern is check with the text from right to left and progresses left to right through the text.

Let c be the character in the text that aligns with the last character of the pattern. If the pattern does not match there are 4 cases to consider.

The **mismatch occurs at the last character** of the pattern:

Case 1: ***c* does not exist** in the pattern (Not the mismatch occurred here) then shift pattern right the size of the pattern.

```
T[0] ...    S    ... T[n-1]
          |
      LEADER
                    LEADER
```

Case 2: The mismatch happens at the last character of the pattern and ***c* does exist** in the pattern then the shift should be to the **right most *c*** in the m-1 remaining characters of the pattern.

```
T[0] ...    A    ... T[n-1]
          |
      LEADER
                LEADER
```

The **mismatch happens in the middle** of the pattern:

Case 3: The mismatch happens in the middle (therefore c is in pattern) and there are **no other *c* in the pattern** then the shift should be the pattern length.

```
T[0] ...   MER    ... T[n-1]
         |
      LEADER
                    LEADER
```

Case 4: The mismatch happens in the middle of the pattern but **there is other *c* in pattern** then the shift should be the **right most *c*** in the m-1 remaining characters of the pattern.

```
T[0] ...  EDER     ... T[n-1]
            |
          LEADER
              LEADER
```

The table of shift values, *table*(c), is a table of the entire alphabet of the text and should give

$t(c) = m$ if c is not in the first m-1 characters of the pattern

$t(c)$ = distance of the right most c in the first m-1 characters of the pattern

ALGORITHM *ShiftTable(P* [0..$m-1$]*)*
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: $Table[0..size-1]$ indexed by the alphabet's characters and
// filled with shift sizes computed by formula (7.1)
for $i \leftarrow 0$ **to** $size-1$ **do**
  $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m-2$ **do**
$Table[P[j]] \leftarrow m-1-j$
return *Table*

Shift Table for pattern BARBER

0	1	2	3	4
B	A	R	E	*

ALGORITHM *HorspoolMatching(P* [0..$m-1$], T [0..$n-1$]*)*
//Implements Horspool's algorithm for string matching
//Input: Pattern $P[0..m-1]$ and text $T[0..n-1]$
//Output: The index of the left end of the first matching substring
// or −1 if there are no matches
ShiftTable(P [0..$m-1$]*)* //generate *Table* of shifts
$i \leftarrow m-1$ //position of the pattern's right end
while $i \le n-1$ **do**
$k \leftarrow 0$ //number of matched characters
while $k \le m-1$ **and** $P[m-1-k] = T[i-k]$ **do**
$k \leftarrow k+1$
if $k = m$
return $i-m+1$
else $i \leftarrow i + Table[T[i]]$
return −1

INDEX	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
SRC	J	I	M	-	S	A	W	-	M	E	-	I	N	-	B	A	R	B	E	R	-	S	H	O	P
PATTERN	B	A	R	B	E	R																			
					B	A	R	B	E	R															
						B	A	R	B	E	R														
												B	A	R	B	E	R								
															B	A	R	B	E	R					

HASHING

Hashing is a common method of accessing data records using the hash table. Hashing can be used to build, search, or delete from a table.

Hash Table: A hash table is a data structure that stores records in an array, called a hash table. A Hash table can be used for quick insertion and searching.

Load Factor: The ratio of the *number of items in a table* to *the table's size* is called the *load factor*.

Hash Function:

- It is a method for computing table index from key.
- A good hash function is simple, so it can be computed quickly.
- The major advantage of hash tables is their speed.
- If the hash function is slow, this speed will be degraded.
- The purpose of a hash function is to take a range of key values and transform them into index values in such a way that the key values are distributed randomly across all the indices of the hash table.

There are many hash functions approaches as follows:

Division Method:

- Mapping a key K into one of m slots by taking the remainder of K divided by m.

$$h(K) = K \bmod m$$

- Example: Assume a table has 8 slots (m=8). Using division method, insert the following elements into the hash table. 36, 18, 72, 43, and 6 are inserted in the order.

36 % 8 = 4

18 % 8 = 2

72 % 8 = 0

43 % 8 = 3

6 % 8 = 6

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
72		18	43	36		6	

Mid-Square Method:

Mapping a key K into one of m slots, by getting the some middle digits from value K^2.

$$h(k) = K^2 \text{ and get middle } (\log_{10} m) \text{ digits}$$

Example: 3121 is a key and square of 3121 is 9740641. Middle part is 406 (with a table size of 1000)

Folding Method:

Divide the key K into some sections, besides the last section, have same length. Then, add these sections together.

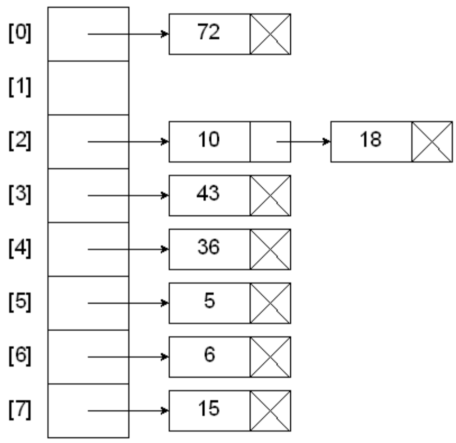
- Shift folding (123456 is folded as 12+34+56)
- Folding at the boundaries (123456 is folded as 12+43+56)

Problems with hashing:

- **Collision:** No matter what the hash function, there is the possibility that two different keys could resolve to the same hash address. This situation is known as a collision.
- **Handling the Collisions:** The following techniques can be used to handle the collisions.
  - Chaining
  - Double hashing (Re-hashing)
  - Open Addressing (Linear probing, Quadratic probing, Random probing), etc.

Chaining:

- A chain is simply a linked list of all the elements with the same hash key.
- A linked list is created at each index in the hash table.
- Hash Function: h (K) = K mod m
- Example: Assume a table has 8 slots (m=8). Using the chaining, insert the following elements into the hash table. 36, 18, 72, 43, 6, 10, 5, and 15 are inserted in the order.



- A data item's key is hashed to the index in simple hashing, and the item is inserted into the linked list at that index.
- Other items that hash to the same index are simply added to the linked list.
- Cost is proportional to length of list.

Advantages of Chaining:

- Unbounded elements can be stored (No bound to the size of table).
- Searching and Deletion is easier

Disadvantage of Chaining:

- Too many linked lists (overhead of linked lists)

Open Addressing:

In open addressing, when a data item can't be placed at the hashed index value, another location in the array is sought. We'll explore three methods of open addressing, which vary in the method used to find the next empty location. These methods are linear probing, quadratic probing, and double hashing.

Linear Probing:

- When using a linear probing method the item will be stored in the **next available slot** in the table, assuming that the table is not already full.
- This is implemented via a linear searching for an empty slot, from the point of collision.
- If the end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there.
- Example: Assume a table has 8 slots (m=8). Using Linear probing, insert the following elements into the hash table. 36, 18, 72, 43, 6, 10, 5, and 15 are inserted in the order.

Hash key = key % table size

36 % 8 = 4
18 % 8 = 2
72 % 8 = 0
43 % 8 = 3
6 % 8 = 6
10 % 8 = 2
5 % 8 = 5
15 % 8 = 7

[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5

Relationship between probe length (P) and load factor (L) for linear probing:

- For a successful search: $P = (1 + 1 / (1–L)^2) / 2$
- For an unsuccessful search: $P = (1 + 1 / (1–L)) / 2$

Analysis of Linear Probing:

- If load factor is too small then too many empty cells.

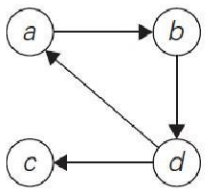
Dynamic Programming

Transitive Closure using Warshall's Algorithm,

Definition: The **transitive closure** of a directed graph with n vertices can be defined as the n

× n boolean matrix T = {tij }, in which the element in the i^{th} row and the j^{th} column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the i^{th} vertex to the j^{th} vertex; otherwise, t^{ij} is 0.

Example: An example of a digraph, its adjacency matrix, and its transitive closure is given below.



(a) Digraph.

$$A = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b) Its adjacency matrix.

$$T = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

(c) Its transitive closure.

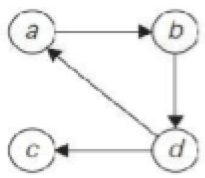
We can generate the transitive closure of a digraph with the help of depth first search or breadth-first search. Performing either traversal starting at the i^{th} vertex gives the information about the vertices reachable from it and hence the columns that contain 1's in the i^{th} row of the transitive closure. Thus, doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.

Since this method traverses the same digraph several times, we can use a better algorithm called **Warshall's algorithm**. Warshall's algorithm constructs the transitive closure through a series of n × n boolean matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}.$$

Each of these matrices provides certain information about directed paths in the digraph. Specifically, the element $r_{ij}^{(k)}$ in the i^{th} row and j^{th} column of matrix $R^{(k)}$ (i, j = 1, 2, . . . , n, k = 0, 1, . . . , n) is equal to 1 if and only if there exists a directed path of a positive length from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any, numbered not higher than k.

Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph. $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate. The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

digraph's transitive closure.

This means that there exists a path from the ith vertex vi to the jth vertex vj with each intermediate vertex numbered not higher than k:

vi, a list of intermediate vertices each numbered not higher than k, vj . --- (*)

Two situations regarding this path are possible.

1. In the first, the list of its intermediate vertices **does not** contain the k^{th} vertex. Then this path from v_i to v_j has intermediate vertices numbered not higher than k−1. i.e. $r_{ij}^{(k-1)} = 1$
2. The second possibility is that path (*) **does contain** the k^{th} vertex vk among the intermediate vertices. Then path (*) can be rewritten as;

vi, vertices numbered ≤ k − 1, vk, vertices numbered ≤ k − 1, vj .

i.e $r_{ik}^{(k-1)} = 1$ and $r_{kj}^{(k-1)} = 1$

Thus, we have the following formula for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \quad \text{or} \quad \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right)$$

The Warshall's algorithm works based on the above formula.

As an example, the application of Warshall's algorithm to the digraph is shown below. New 1's are in bold.

ALGORITHM *Warshall*($A[1..n, 1..n]$)

//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

Analysis

Its time efficiency is Θ(n3). We can make the algorithm to run faster by treating matrix rows as bit strings and employ the bitwise or operatic

most modern computer Languages

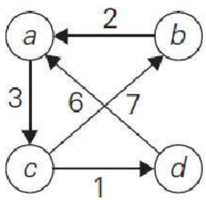
Space efficiency: Although separate matrices for recording intermediate results of the algorithm are used, that can be avoided.

All Pairs Shortest Paths using Floyd's Algorithm,

Problem definition: Given a weighted connected graph (undirected or directed), the all-pairs shortest paths problem asks to find the distances—i.e., the lengths of the shortest paths - from each vertex to all other vertices.

Applications: Solution to this problem finds applications in communications, transportation networks, and operations research. Among recent applications of the all-pairs shortest-path problem is pre-computing distances for motion planning in computer games.

We store the lengths of shortest paths in an n x n matrix D called the distance matrix: the element dij in the i^{th} row and the j^{th} column of this matrix indicates the length of the shortest path from the i^{th} vertex to the j^{th} vertex.



$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array} \qquad D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(a) Digraph. (b) Its weight matrix. (c) Its distance matrix

We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm. It is called **Floyd's algorithm.**

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of n × n matrices:

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}.$$

The element $d_{ij}^{(k)}$ in the i^{th} row and the j^{th} column of matrix $D^{(k)}$ (i, j = 1, 2, . . . , n, k = 0, 1, . . . , n) is equal to the length of the shortest path among all paths from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any, numbered not higher than k.

As in Warshall's algorithm, we can compute all the elements of each matrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$

If $d_{ij}^{(k)}$ =1, then it means that there is a path;

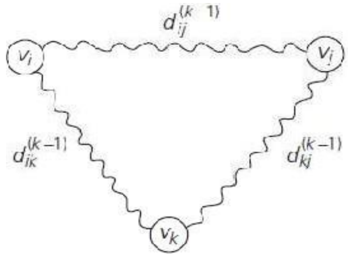
vi, a list of intermediate vertices each numbered not higher than k, vj .

We can partition all such paths into two disjoint subsets: those that do not use the k^{th} vertex vk as intermediate and those that do.

i. Since the paths of the first subset have their intermediate vertices numbered not higher than k − 1, the shortest of them is, by definition of our matrices, of length $d_{ij}^{(k-1)}$

ii. In the second subset the paths are of the form
vi, vertices numbered ≤ k − 1, vk, vertices numbered ≤ k − 1, vj .

The situation is depicted symbolically in Figure, which shows the underlying idea of Floyd's algorithm.



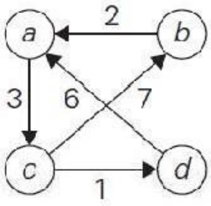
Taking into account the lengths of the shortest paths in both subsets leads to the following recurrence:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1,\ \ d_{ij}^{(0)} = w_{ij}.$$

```
ALGORITHM  Floyd(W[1..n, 1..n])
    //Implements Floyd's algorithm for the all-pairs shortest-paths problem
    //Input: The weight matrix W of a graph with no negative-length cycle
    //Output: The distance matrix of the shortest paths' lengths
    D ← W  //is not necessary if W can be overwritten
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
    return D
```

Analysis: Its time efficiency is $\Theta(n^3)$, similar to the warshall's algorithm.

Application of Floyd's algorithm to the digraph is shown below. Updated elements are shown in bold.



$$D^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$$D^{(2)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note a new shortest path from *c* to *a*).

$$D^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (note four new shortest paths from *a* to *b*, from *a* to *d*, from *b* to *d*, and from *d* to *b*).

$$D^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

Sample Example:

Solve the all-pairs shortest-path problem for the digraph with the following weight matrix:

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

Matrix Chain Multiplication using Dynamic Programming

In our day to day life when we do making coin change, robotics world, aircraft, mathematical problems like Fibonacci sequence, simple matrix multiplication of more than two matrices and its multiplication possibility is many more so in that get the best and optimal solution. NOW we can look about one problem that is ***MATRIX CHAIN MULTIPLICATION PROBLEM.***

Suppose, We are given a sequence (chain) ***(A1, A2……An)*** of n matrices to be multiplied, and we wish to compute the product ***(A1A2…..An)***.We can evaluate the above expression using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. For example, if the chain of matrices is ***(A1, A2, A3, A4)*** then we can fully parenthesize the product (***A1A2A3A4)*** in five distinct ways:

1:-(A1(A2(A3A4)))
2:-(A1((A2A3)A4))
3:- ((A1A2)(A3A4))
4:-((A1(A2A3))A4)
5:-(((A1A2)A3)A4)

We can multiply two matrices A and B only if they are compatible. the number of columns of A must equal the number of rows of B. If A is a p x q matrix and B is a q x r matrix, the resulting matrix C is a p x r matrix. The time to compute C is dominated by the number of scalar multiplications is pqr. we shall express costs in terms of the number of scalar multiplications. For example, if we have three matrices (A1,A2,A3) and its cost is (10x100),(100x5),(5x500) respectively. so we can calculate the cost of scalar multiplication is 10*100*5=5000 if ((A1A2)A3), 10*5*500=25000 if (A1(A2A3)), and so on cost calculation. ***Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.*** that is here is minimum cost is 5000 for above example. So problem is we can perform a many time of cost multiplication and repeatedly the calculation is performing. so this general method is very time consuming and tedious. So we can apply ***dynamic programming*** for solve this kind of problem.

when we used the Dynamic programming technique we shall follow some 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.
4. Construct an optimal solution from computed information.

we have matrices of any of order. our goal is find optimal cost multiplication of matrices. when we solve the this kind of problem using DP step 2 we can get
m[i , j] = min { m[i , k] + m[i+k , j] + pi-1*pk*pj } if i < j…. where p is dimension of matrix , i ≤ k < j …..

The basic algorithm of matrix chain multiplication is:

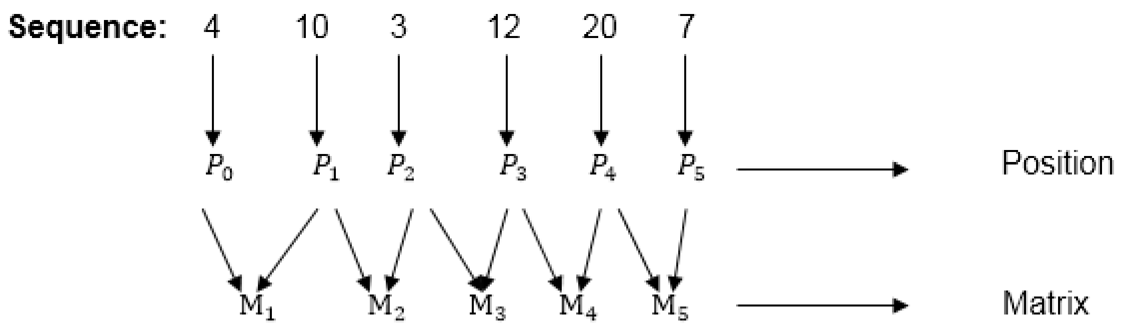
```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k]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

Example of Matrix Chain Multiplication

Example: We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], 0 ≤ i, j≤ 5. We know M [i, i] = 0 for all i.

1	2	3	4	5	
0					1
	0				2
		0			3
			0		4
				0	5

Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.



In Dynamic Programming, initialization of every method done by ‘0’.So we initialize it by ‘0’.It will sort out diagonally.

We have to sort out all the combination but the minimum output combination is taken into consideration.

Calculation of Product of 2 matrices:

1. m (1,2) = m1 x m2
 = 4 x 10 x 10 x 3
 = 4 x 10 x 3 = 120

2. m (2, 3) = m2 x m3
 = 10 x 3 x 3 x 12
 = 10 x 3 x 12 = 360

3. m (3, 4) = m3 x m4
 = 3 x 12 x 12 x 20
 = 3 x 12 x 20 = 720

4. m (4,5) = m4 x m5
 = 12 x 20 x 20 x 7
 = 12 x 20 x 7 = 1680

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

- We initialize the diagonal element with equal i,j value with ‘0’.
- After that second diagonal is sorted out and we get all the values corresponded to it

Now the third diagonal will be solved out in the same way.

Now product of 3 matrices:

M [1, 3] = M1 M2 M3

1. There are two cases by which we can solve this multiplication: (M1 x M2) + M3, M1+ (M2x M3)

2. After solving both cases we choose the case in which minimum output is there.

$$M\,[1,3] = \min \begin{Bmatrix} M\,[1,2] + M\,[3,3] + p_0\, p_2 p_3 = 120 + 0 + 4.3.12 \;=\; 264 \\ M\,[1,1] + M\,[2,3] + p_0\, p_1 p_3 = 0 + 360 + 4.10.12 \;=\; 840 \end{Bmatrix}$$

M [1, 3] =264

As Comparing both output **264** is minimum in both cases so we insert **264** in table and (M1 x M2) + M3 this combination is chosen for the output making.
M [2, 4] = M2 M3 M4

1. There are two cases by which we can solve this multiplication: (M2x M3)+M4, M2+(M3 x M4)

2. After solving both cases we choose the case in which minimum output is there.

$$M\ [2,\ 4] = \min \begin{Bmatrix} M[2,3] + M[4,4] + \ p_1p_3p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + \ M[3,4] + \ p_1p_2p_4 = 0 + 720 + 10.3.20 = \ 1320 \end{Bmatrix}$$

M [2, 4] = 1320

As Comparing both output **1320** is minimum in both cases so we insert **1320** in table and M2+(M3 x M4) this combination is chosen for the output making.
M [3, 5] = M3 M4 M5

1. There are two cases by which we can solve this multiplication: (M3 x M4) + M5, M3+ (M4xM5)

2. After solving both cases we choose the case in which minimum output is there.

$$M\ [3,\ 5] = \min \begin{Bmatrix} M[3,4] + M[5,5] + p_2p_4p_5 = 720 + 0 + 3\ .20.7\ = \quad 1140 \\ M[3,3] + \ M\ [4,5] + \ p_2p_3p_5 = 0 + 1680 + 3.12.7 = 1932 \end{Bmatrix}$$

M [3, 5] = 1140

As Comparing both output **1140** is minimum in both cases so we insert **1140** in table and (M3 x M4) + M5this combination is chosen for the output making.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now Product of 4 matrices:
M [1, 4] = M1 M2 M3 M4

There are three cases by which we can solve this multiplication:

1. (M1 x M2 x M3) M4
2. M1 x(M2 x M3 x M4)
3. (M1 xM2) x (M3 x M4)

After solving these cases we choose the case in which minimum output is there

$$M\,[1, 4] = \min \begin{cases} M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4.3.20 = 1080 \\ M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases}$$

M [1, 4] =1080

As comparing the output of different cases then '**1080**' is minimum output, so we insert 1080 in the table and (M1 xM2) x (M3 x M4) combination is taken out in output making,
M [2, 5] = M2 M3 M4 M5

There are three cases by which we can solve this multiplication:

1. (M2 x M3 x M4)x M5
2. M2 x(M3 x M4 x M5)
3. (M2 x M3)x (M4 x M5)

After solving these cases we choose the case in which minimum output is there

$$M\,[2, 5] = \min \begin{cases} M[2,4] + M[5,5] + p_1p_4p_5 = 1320 + 0 + 10.20.7 = 2720 \\ M[2,3] + M[4,5] + p_1p_3p_5 = 360 + 1680 + 10.12.7 = 2880 \\ M[2,2] + M[3,5] + p_1p_2p_5 = 0 + 1140 + 10.3.7 = 1350 \end{cases}$$

M [2, 5] = 1350

As comparing the output of different cases then '**1350**' is minimum output, so we insert 1350 in the table and M2 x(M3 x M4xM5)combination is taken out in output making.

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Now Product of 5 matrices:

M [1, 5] = M1 M2 M3 M4 M5

There are five cases by which we can solve this multiplication:

1. (M1 x M2 xM3 x M4)x M5
2. M1 x(M2 xM3 x M4 xM5)
3. (M1 x M2 xM3)x M4 xM5
4. M1 x M2x(M3 x M4 xM5)

After solving these cases we choose the case in which minimum output is there

$$M[1,5] = \min \begin{Bmatrix} M[1,4] + M[5,5] + p_0p_4p_5 = 1080 + 0 + 4.20.7 = 1544 \\ M[1,3] + M[4,5] + p_0p_3p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0p_2p_5 = 120 + 1140 + 4.3.7 = 1344 \\ M[1,1] + M[2,5] + p_0p_1p_5 = 0 + 1350 + 4.10.7 = 1630 \end{Bmatrix}$$

M [1, 5] = 1344

As comparing the output of different cases then ‘**1344**’ is minimum output, so we insert 1344 in the table and M1 x M2 x(M3 x M4 x M5)combination is taken out in output making.

Final Output is:

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

$\longrightarrow$

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

So we can get the optimal solution of matrices multiplication….

Also calculate the table S which stores the values of parenthesization. Use this table to get the best possible combination.

Longest common subsequence (LCS)

Attached separately