Dynamic Programming

December 15, 2016

Why Dynamic Programming

- Often recursive algorithms solve fairly difficult problems efficiently
  - BUT in other cases they are inefficient because they recalculate certain function values many times.
- For example: Fibonacci numbers.

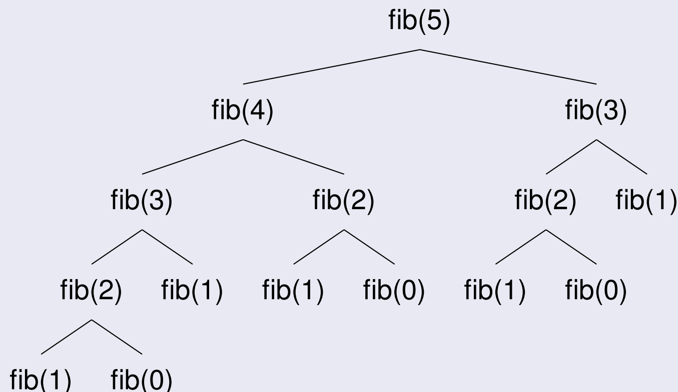
Recursive solution

- The Recursive solution to finding the nth Fibonacci number:

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

- Time Complexity: $T(n) = T(n-1) + T(n-2)$ which is exponential.
- Extra Space:$O(n)$ if we consider the function call stack size, otherwise $O(1)$.

The Problem



- Many calls to `fib(3)`,`fib(2)`, `fib(1)` and `fib(0)` are made.
  - It would be nice if we only made those calls once, then simply used those values as necessary.

Dynamic Programming

- The idea of dynamic programming is to avoid making redundant function calls
  - Instead, one should store the answers to all necessary function calls in memory and simply look these up as necessary.
- Using this idea, we can code up a dynamic programming solution to the Fibonacci number question that is far more efficient than the recursive version:

Dynamic Programming Solution

```
int fibonacci(int n) {

        // Declare an array to store Fibonacci numbers.
        int fibnumbers[n+1];
        // 0th and 1st number of the series are 0 and 1
        fibnumbers[0] = 1;
        fibnumbers[1] = 1;

        for (int i = 2; i < n+1; i++)
          // Add the previous 2 numbers in the series and store it
          fibnumbers[i] = fibnumbers[i - 1] + fibnumbers[i - 2];

        return fibnumbers[n];
}
```

- Time Complexity: $O(n)$
- Extra Space: $O(n)$

Dynamic Programming

- The only requirement this program has that the recursive one doesn't is the space requirement of an entire array of values.
  - In fact, at a particular moment in time while the recursive program is running, it has at least n recursive calls in the middle of execution all at once. The amount of memory necessary to simultaneously keep track of each of these is at least as much as the memory the array we are using above needs.
- Usually however, a dynamic programming algorithm presents a time-space trade off.
  - More space is used to store values,
  - BUT less time is spent because these values can be looked up.
- Can we do even better (with respect to memory) with our Fibonacci method above?
  - What numbers do we really have to keep track of all the time?

Improved Dynamic Programming Solution

- We can optimize the space used the dynamic programming solution by storing the previous two numbers only, because that is all we need to get the next Fibonacci number in series.

```
int fibonacci(int n)
{
  int prevFib1 = 0, prevFib2 = 1, newFib;
  if( n == 0)
    return prevFib1;
  for (int i = 2; i <= n; i++)
  {
     newFib = prevFib1 + prevFib2;
     prevFib1 = prevFib2;
     prevFib2 = newFib;
  }
  return prevFib2;
}
```

- Time Complexity: O(n)
- Extra Space: O(1)

Why the term "Dynamic Programming"

- The origin of the term dynamic programming has very little to do with writing code.
- It was first coined by Richard Bellman in the 1950s, a time when computer programming was practiced by so few people as to not even deserve a name.
- At the time, programming meant *planning*, and dynamic programming was developed to optimally plan multistage processes.

Properties of problems that can be solved using Dynamic Programming

- Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.
- The two main properties of a problem that suggest that the given problem can be solved using Dynamic programming are:
  1. Overlapping Subproblems
  2. Optimal Substructure

Overlapping Subproblems Property

- Dynamic Programming combines solutions to sub-problems.
- Dynamic Programming is mainly used when solutions of the same subproblems are needed again and again.
- Computed solutions to subproblems are stored in a table so that these do not have to recomputed.
- Example: Recursive version of Fibonacci numbers.

Storing values for reuse

- There are two different ways to store the values so that these values can be reused:
  - Memoization (Top Down)
  - Tabulation (Bottom Up)

Memoization

- The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions.
- We initialize a lookup array with all initial values as a default value.
- Whenever we need solution to a subproblem, we first look into the lookup table.
- If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.

Memoization solution for the nth Fibonacci Number

```
/* C++ program for Memoized version for nth Fibonacci number */
#include<iostream>
using namespace std;

const int NIL= -1;
const int MAX= 100;

int lookup[MAX];

/* Function to initialize NIL values in lookup table */
void initialize()
{
  for (int i = 0; i < MAX; i++)
    lookup[i] = NIL;
}

/* function for nth Fibonacci number */
int fib(int n)
{
   if (lookup[n] == NIL)
   {
      if (n <= 1)
         lookup[n] = n;
      else
         lookup[n] = fib(n-1) + fib(n-2);
   }

   return lookup[n];
}

int main ()
{
  int n = 40;
  initialize();
  cout << "Fibonacci number is " << fib(n);
  return 0;
}
```

Tabulation

- The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from the table.
- Starting from the first entry, all entries are filled one by one.

Tabulated version for the nth Fibonacci Number

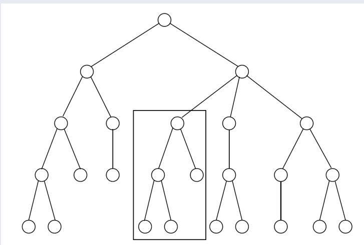
```
int fib(int n)
{
  int f[n+1];

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

  return f[n];
}
```

Optimal Substructure Property

- A given problem has Optimal Substructure Property if the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
- Example: the Shortest Path problem has the following optimal substructure property:
  - If a vertex x lies in the shortest path from a source vertex u to destination vertex v then the shortest path from u to v is the combination of the shortest path from u to x and the shortest path from x to v.

Common subproblems

- Finding the right subproblem takes creativity and experimentation. But there are a few standard choices that seem to arise repeatedly in dynamic programming.
  - The input is $x_1, x_2, \ldots, x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.
    The number of subproblems is therefore linear.
  - The input is $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$.
    $|x_1 \;\; x_2 \;\; x_3 \;\; x_4 \;\; x_5 \;\; x_6| \;\; x_7 \;\; x_8 \;\; x_9 \;\; x_{10}$
    $|y_1 \;\; y_2 \;\; y_3 \;\; y_4 \;\; y_5| \;\; x_6 \;\; x_7 \;\; x_8$
    The number of subproblems is $O(mn)$.
  - The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$.
    $x_1 \;\; x_2 \;\; |x_3 \;\; x_4 \;\; x_5 \;\; x_6| \;\; x_7 \;\; x_8 \;\; x_9 \;\; x_{10}$
    The number of subproblems is $O(n^2)$.
  - The input is a rooted tree. A subproblem is a rooted subtree.



Matrix Chain Multiplication

- Matrix multiplication:
  - The product $C = AB$ of a $p \times q$ matrix A and a $q \times r$ matrix B is a $p \times r$ matrix given by

    $$c[i,j] = \sum_{k=1}^{q} a[i,k]b[k,j]$$

    for $1 \leq i \leq p$ and $1 \leq j \leq r$.
  - If AB is defined, BA may not be defined.
  - Multiplication is recursively defined by

    $$A_1 A_2 A_3 \ldots A_{s-1} A_s = A_1(A_2(A_3 \ldots (A_{s-1} A_s)))$$

  - Matrix multiplication is associative , e.g

    $$A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1(A_2 A_3)),$$

    so parenthesization does not change result.

Matrix Chain Multiplication

Problem Definition: Given
dimensions $p_0, p_1, \ldots, p_n$,
corresponding to matrix sequence $A_1, A_2, \ldots, A_n$,
where A_i has dimension $p_{i-1} \times p_i$,
find the most efficient way to multiply these matrices together.

- The problem is not to actually to perform the multiplications, but merely to decide in which order to perform the multiplications.
- Many options to multiply a chain of matrices because matrix multiplication is associative.
- The order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency.
- For example, suppose A is a 10×30 matrix, B is a 30×5 matrix, and C is a 5×60 matrix. Then,
  (AB)C = $(10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500$ operations
  A(BC) = $(30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000$ operations.

Properties of Matrix Chain Multiplication (I)

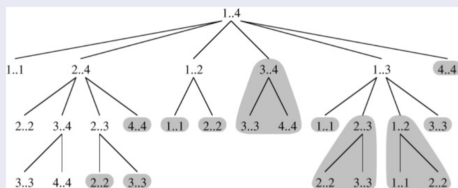
- **Optimal Substructure:** A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value.
- In a chain of matrices of size n, we can place the first set of parenthesis in $n - 1$ ways.
  - For example, if the given chain consists of 4 matrices, assuming the chain to be ABCD, then there are 3 ways to place the first set of parenthesis outer side: (A)(BCD), (AB)(CD) and (ABC)(D).
- So when we place a set of parenthesis, we divide the problem into subproblems of smaller size.
- Therefore, the problem has the optimal substructure property and can be easily solved using recursion.
- Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements (these placements create subproblems of smaller size).

A naive recursive implementation

```
/* A naive recursive implementation based on the optimal substructure property */
#include<iostream>
#include<climits>
using namespace std;

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int i, int j)
{
    if(i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    // place parenthesis at different places between first
    // and last matrix, recursively calculate count of
    // multiplications for each parenthesis placement and
    // return the minimum count
    for (k = i; k <j; k++)
    {
        count = MatrixChainOrder(p, i, k) +
                MatrixChainOrder(p, k+1, j) +
                p[i-1]*p[k]*p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 2, 3, 4, 3};
    int n = sizeof(arr)/sizeof(arr[0]);

    cout <<"Minimum number of multiplications is " <<
                      MatrixChainOrder(arr, 1, n-1);

    return 0;
}
```

Properties of Matrix Chain Multiplication (II)

- **Overlapping Subproblems:** Time complexity of the naive recursive approach is exponential. It should be noted that this function computes the same subproblems again and again.
- Consider the following recursion tree for a matrix chain of size 4. The function MatrixChainOrder(p, 3, 3) is called four times.



- Since the same subproblems are called again, this problem has the Overlapping Subproblems property.
- So Matrix Chain Multiplication problem has both properties of a dynamic programming problem.
- Recomputations of the same subproblems can be avoided by constructing a temporary array in bottom up manner.

Developing a Dynamic Programming Algorithm

- **Step 1:** Determine the structure of an optimal solution (in this case, a parenthesization).
- **Decompose the problem into subproblems:** For each pair $1 \leq i \leq j \leq n$, determine the multiplication sequence for $A_{i..j} = A_i A_{i+1} \dots A_j$ that minimizes the number of multiplications. $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix.

Developing a Dynamic Programming Algorithm

- **Step 1:** Determine the structure of an optimal solution (in this case, a parenthesization).
- **High-Level Parenthesization for $A_{i..j}$**
  For any optimal multiplication sequence, at the last step we multiply two matrices $A_{i..k}$ and $A_{k+1..j}$ for some k . That is,

  $$A_{i..j} = (A_i \ldots A_k)(A_{k+1} \ldots A_j) = A_{i..k}A_{k+1..j}$$

- **Example:**

  $$A_{3..6} = (A_3(A_4A_5))(A_6) = A_{3..5}A_{6..6}$$

  Here $k = 5$.

Developing a Dynamic Programming Algorithm

- **Step 1- Continued:** Thus the problem of determining the optimal sequence of multiplications is broken down into two questions:
  - How do we decide where to split the chain (what is k)?
    - Search all possible values of k
  - How do we parenthesize the subchains $A_{i..k}$ and $A_{k+1..j}$?
    - The problem has optimal substructure property that $A_{i..k}$ and $A_{k+1..j}$ must be optimal so we can apply the same procedure recursively.

Developing a Dynamic Programming Algorithm

- **Step 2:** Recursively define the value of an optimal solution.
  - We will store the solutions to the subproblems in an array.
  - For $1 \leq i \leq j \leq n$, let $m[i,j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$.
    The optimum cost can be described by the following recursive definition:
    $$m[i,j] = \begin{cases} 0 & i = j \\ min_{i \leq k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) & i < j \end{cases}$$

Developing a Dynamic Programming Algorithm

- **Step 2– Continued:** We know that, for some k

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

- We do not know what k should be.
- But, there are only $j - i$ possible values of k so we can check them all and find the one which returns the smallest cost.

Developing a Dynamic Programming Algorithm

- **Step 3:** Compute the value of an optimal solution in a bottom-up fashion.
- **Our Table:** $m[1..n, 1, ..n]$
  - $m[i, j]$ only defined for $i \leq j$
- The important point is that when we use the equation

  $$min_{i \leq k < j}(m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j)$$

  to calculate $m[i, j]$, we must have already evaluated $m[i, k]$ and $m[k + 1, j]$.
- For both cases, the corresponding length of the matrix-chain are both less than $j - i + 1$.
- Hence, the algorithm should fill the table in increasing order of the length of the matrix-chain.

  That is, we calculate in the order

  m[1, 2], m[2, 3],m[3,4],... ,m[n - 3,n - 2],m[n - 2,n- 1],m[n- 1,n]

  m[1, 3],m[2,4],m[3, 5],... ,m[n - 3,n - 1],m[n - 2,n]

  m[1,4],m[2,5],m[3,6],... ,m[n - 3,n]

  .

  .

  .

  m[1, n - 1], m[2, n]

  m[1, n]

Developing a Dynamic Programming Algorithm

- **Step 4:** Construct an optimal solution from computed information – extract the actual sequence.
- **Idea:** Maintain an array $s[1..n, 1..n]$, where $s[i, j]$ denotes k for the optimal splitting in computing $A_{i..j} = A_{i..k}A_{k+1..j}$.
- The array $s[1..n, 1..n]$ can be used recursively to recover the multiplication sequence.
- How to Recover the Multiplication Sequence?
  $s[1, n]$ $(A_1 \dots A_{s[1,n]})(A_{s[1,n]+1} \dots A_n)$
  $s[1, s[1, n]]$ $(A_1 \dots A_{s[1,s[1,n]]})(A_{s[1,s[1,n]]+1} \dots A_{s[1,n]})$
  $s[s[1, n] + 1, n]$ $(A_{s[1,n]+1} \dots A_{s[s[1,n]+1,n]}) \times (A_{s[s[1,n]+1,n]+1} \dots A_n)$
  Do this recursively until the multiplication sequence is determined.

Developing a Dynamic Programming Algorithm

- **Step 4:** Step 4: Construct an optimal solution from computed information - extract the actual sequence.
- **Example of Finding the Multiplication Sequence:**
  - Consider $n = 6$. Assume that the array $s[1..6, 1..6]$ has been computed. The multiplication sequence is recovered as follows.
    $s[1,6] = 3$ $(A_1 A_2 A_3)(A_4 A_5 A_6)$
    $s[1,3] = 1$ $(A_1(A_2 A_3))$
    $s[4,6] = 5$ $((A_4 A_5) A_6)$
  - Hence the final multiplication sequence is

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

Dynamic Programming implementation

```
#include<iostream>
#include<climits>

using namespace std;

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int**s, int n) {

        /* For simplicity of the program, one extra row and one
         extra column are allocated in m[][].  0th row and 0th
         column of m[][] are not used */
        int m[n][n];

        int i, j, k, L, q;

        /* m[i,j] = Minimum number of scalar multiplications needed
         to compute the matrix A[i]A[i+1]...A[j] = A[i..j] where
         dimension of A[i] is p[i-1] x p[i] */

        // cost is zero when multiplying one matrix.
        for (i = 1; i < n; i++)
                m[i][i] = 0;

        // L is chain length.
        for (L = 2; L < n; L++) {
                for (i = 1; i < n - L + 1; i++) {
                        j = i + L - 1;
                        m[i][j] = INT_MAX;
                        for (k = i; k <= j - 1; k++) {
                                // q = cost/scalar multiplications
                                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[1][n - 1];
}
```

Dynamic Programming implementation

```
void printOptimalParens(int**s, int i, int j) {
        if (i == j)
                cout << "A" << i;
        else {
                cout << "(";
                printOptimalParens(s, i, s[i][j]);
                printOptimalParens(s, s[i][j] + 1, j);
                cout << ")";
        }

}

int main() {

        // Dimensions of the matrices
        int p[] = { 10,100,5,50 };

        int size = sizeof(p) / sizeof(p[0]);

        int **s;
        s = new int *[size];
        for (int i = 0; i < size; i++)
                s[i] = new int[size];

        cout << "Minimum number of multiplications is "
                        << MatrixChainOrder(p, s, size) << endl;

        cout << "Optimal parenthesization:" << endl;
        printOptimalParens(s, 1, size - 1);

        return 0;
}
```

Single-Source Shortest Paths: the Bellman-Ford Algorithm

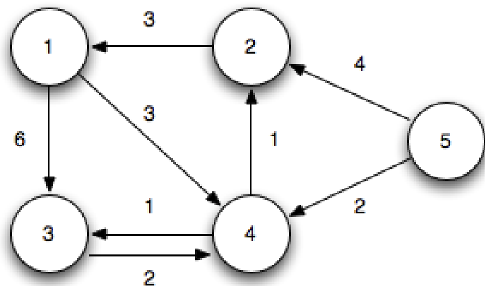
- Given a graph G and a source vertex *src* in G, find the shortest paths from *src* to all vertices in the given graph.
- The graph may contain negative weight edges.
- We have discussed Dijkstra's algorithm for this problem.
- Dijkstra's algorithm is a Greedy algorithm.
- Dijkstra does not work for graphs with negative weight edges, Bellman-Ford works for such graphs.
- Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems.

Belman-Ford Pseudocode

```
// At the beginning , all vertices have a weight of infinity
// And a null predecessor
// Except for the Source, where the Weight is zero
for each vertex v in vertices
       distance[v] <- inf
       predecessor[v] <- null

   distance[source] <- 0

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1
       for each edge (u, v) with weight w in edges
           if distance[u] + w < distance[v]
               distance[v] <- distance[u] + w
               predecessor[v] <- u

   // Step 3: check for negative-weight cycles
   for each edge (u, v) with weight w in edges
       if distance[u] + w < distance[v]:
           error "Graph contains a negative-weight cycle"
```

Belman-Ford Example

Given the following directed graph



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

Belman-Ford Example

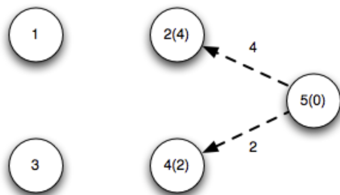
Using vertex 5 as the source (setting its distance to 0), we initialize all the other distances to ∞.

(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	∞	∞	∞	∞	**0**
π	/	/	/	/	/

Belman-Ford Example

Iteration 1: Edges (u_5, u_2) and (u_5, u_4) relax updating the distances to 2 and 4.

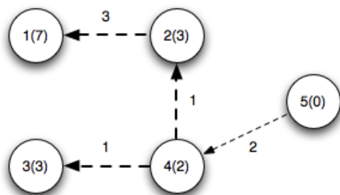


(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	∞	**4**	∞	**2**	0
π	/	**5**	/	**5**	/

Belman-Ford Example

Iteration 2: Edges (u_2, u_1), (u_4, u_2) and (u_4, u_3) relax updating the distances to 1, 2, and 4 respectively. Note edge (u_4, u_2) finds a shorter path to vertex 2 by going through vertex 4

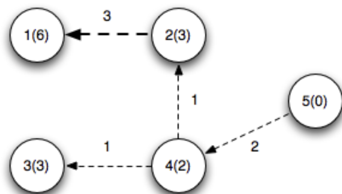


(1,3) = 6
(1,4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	**7**	**3**	**3**	2	0
π	**2**	**4**	**4**	5	/

Belman-Ford Example

Iteration 3: Edge (u_2, u_1) relaxes (since a shorter path to vertex 2 was found in the previous iteration) updating the distance to 1

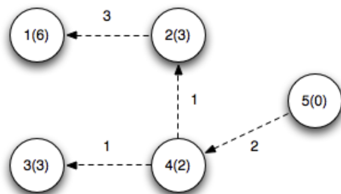


(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	**6**	3	3	2	0
π	**2**	4	4	5	/

Belman-Ford Example

Iteration 4: No edges relax



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

Dynamic Programming implementation

```
// C++ program for Bellman-Ford's single source
// shortest path algorithm.

#include <iostream>
#include <stack>
#include <climits>

using namespace std;

// a structure to represent a weighted edge in graph
struct Edge {
	int src, dest, weight;
};

// a class to represent a connected, directed and
// weighted graph
class Graph {
	// V-> Number of vertices, E-> Number of edges
	int V, E;

	// graph is represented as an array of edges.
	struct Edge* edge;
	// Creates a graph with V vertices and E edges

public:
	Graph(int V, int E) {
		this->V = V;
		this->E = E;
		edge = new struct Edge[E];
	}
	~Graph() {
		delete[] edge;
	}
	void addEdge(int E, int src, int dest, int weight) {
		edge[E].src = src;
		edge[E].dest = dest;
		edge[E].weight = weight;

	}
	struct Edge getEdge(int E) {
		return edge[E];

	}
	int getNumVertex() {
		return V;
	}
	int getNumEdge() {
		return E;
	}

};
```

Dynamic Programming implementation

```
// A utility function used to print the solution
void printArr(int dist[], int predecessor[], int src, int n) {

        cout << "Vertex  Distance from Source Shortest Path\n";
        for (int i = 0; i < n; ++i) {
                cout << " " << i << "          ";
                if (dist[i] != INT_MAX)
                        cout << dist[i];
                else
                        cout << "INF";

                cout << "                  ";

                int u = i;
                stack<int> predStack;
                if (dist[u] != INT_MAX && u != src) {

                        do {
                                u = predecessor[u];
                                predStack.push(u);

                        } while (u != src);
                }

                //print in reverse

                while (!predStack.empty()) {
                        cout << " " << predStack.top();
                        predStack.pop();
                }
                cout << endl;

        }
}
```

Dynamic Programming implementation

```
// The function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm.  The function
// also detects negative weight cycle
void BellmanFord(Graph graph, int src) {
        int V = graph.getNumVertex();
        int E = graph.getNumEdge();
        int dist[V], predecessor[V];
        struct Edge theEdge;
        // Step 1: Initialize distances from src to all other vertices
        // as INFINITE
        for (int i = 0; i < V; i++) {
                dist[i] = INT_MAX;
                predecessor[i] = -1;
        }
        dist[src] = 0;

        // Step 2: Relax all edges |V| - 1 times. A simple shortest
        // path from src to any other vertex can have at-most |V| - 1
        // edges
        for (int i = 1; i <= V - 1; i++) {
                for (int j = 0; j < E; j++) {
                        theEdge = graph.getEdge(j);
                        int u = theEdge.src;
                        int v = theEdge.dest;
                        int weight = theEdge.weight;
                        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                                dist[v] = dist[u] + weight;
                                predecessor[v] = u;

                        }
                }
        }

        // Step 3: check for negative-weight cycles.  The above step
        // guarantees shortest distances if graph doesn't contain
        // negative weight cycle.  If we get a shorter path, then there
        // is a cycle.
        for (int i = 0; i < E; i++) {
                theEdge = graph.getEdge(i);
                int u = theEdge.src;
                int v = theEdge.dest;
                int weight = theEdge.weight;
                if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                        cout << "Graph contains negative weight cycle\n";
        }

        printArr(dist, predecessor, src, V);

        return;
}
```

Dynamic Programming implementation

```
int main() {

        int V = 5;  // Number of vertices in graph
        int E = 8;  // Number of edges in graph
        Graph graph(V, E);



        graph.addEdge(0, 0, 2, 6);
        graph.addEdge(1, 2, 3, 2);
        graph.addEdge(2, 3, 2, 1);
        graph.addEdge(3, 0, 3, 3);
        graph.addEdge(4, 3, 1, 1);
        graph.addEdge(5, 4, 3, 2);
        graph.addEdge(6, 1, 0, 3);
        graph.addEdge(7, 4, 1, 4);

        BellmanFord(graph, 4);

        return 0;
}
```

All pairs shortest paths

- All pairs shortest paths problem: compute the length of the shortest path between every pair of vertices.
- If we use Bellman-Ford for all n possible destinations t, this would take time $O(mn^2)$.
- An alternative dynamic programming algorithm for solving this problem is called the Floyd-Warshall algorithm.
- The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with $\Theta(|V|^3)$ comparisons in a graph.
- There may be up to $\Omega(|V|^2)$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.

Idea of the Algorithm

- The Floyd-Warshall algorithm relies on the principle of dynamic programming.
- This means that all possible paths between pairs of nodes are being compared step by step, while only saving the best values found so far.
- The algorithm begins with the following observation: If the shortest path from u to v passes through w, then the partial paths from u to w and w to v must be minimal as well. Correctness of this statement can be shown by induction.
- The algorithm of Floyd-Warshall works in an iterative way.

Idea of the Algorithm

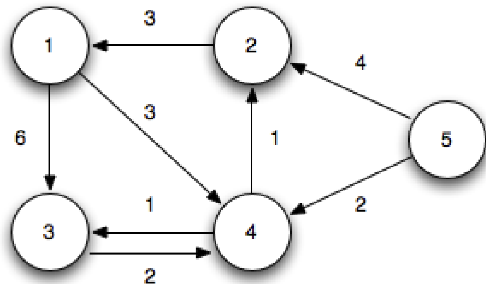
- Let G be a graph with numbered vertices 1 to N. In the kth step, let *shortestPath(i,j,k)* be a function that yields the shortest path from i to j that only uses nodes from the set $\{1, 2, ..., k\}$.
- In the next step, the algorithm will then have to find the shortest paths between all pairs i, j using only the vertices from $\{1, 2, ..., k, k + 1\}$.
- For all pairs of vertices it holds that the shortest path must either only contain vertices in the set $\{1, ..., k\}$, or otherwise must be a path that goes from i to j via k + 1.
- This implies that in the (k+1)th step, the shortest path from i to j either remains *shortestPath(i,j,k)* or is being improved to *shortestPath(i,k+1,k) + shortestPath(k+1, j, k)*, depending on which of these paths is shorter.
- Therefore, each shortest path remains the same, or contains the node k + 1 whenever it is improved.
- This is the idea of dynamic programming. In each iteration, all pairs of nodes are assigned the cost for the shortest path found so far: *shortestPath(i, j, k) = min(shortestPath(i, j, k), shortestPath(i, k + 1, k) + shortestPath(k + 1, j, k))*

Floyd-Warshall Pseudocode

```
let dist be a |V| X |V| array of minimum distances initialized
     to infinity
for each vertex v
   dist[v][v] <- 0
for each edge (u,v)
   dist[u][v] <- w(u,v)  // the weight of the edge (u,v)
for k from 1 to |V|
   for i from 1 to |V|
      for j from 1 to |V|
         if dist[i][j] > dist[i][k] + dist[k][j]
            dist[i][j] <- dist[i][k] + dist[k][j]
```

Floyd-Warshall Example

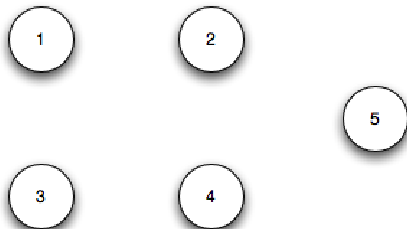
Given the following directed graph



(1,3) = 6
(1,4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

Floyd-Warshall Example

Initialization: (k = 0)

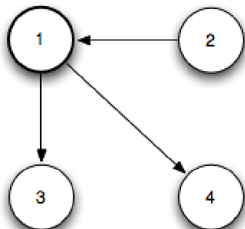


D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	∞	∞	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Floyd-Warshall Example

Iteration 1: (k = 1) Shorter paths from 2 $\curvearrowright$ 3 and 2 $\curvearrowright$ 4 are found through vertex 1

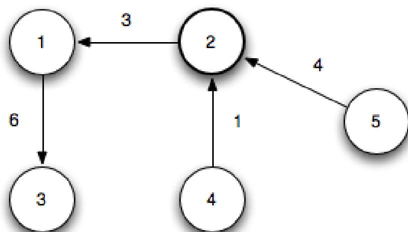


D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Floyd-Warshall Example

Iteration 2: (k = 2) Shorter paths from $4 \curvearrowright 1$, $5 \curvearrowright 1$, and $5 \curvearrowright 3$ are found through vertex 2

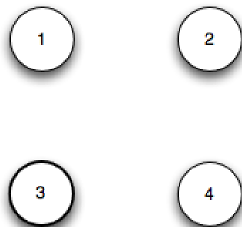


D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	**4**	1	1	0	∞
5	**7**	4	**13**	2	0

Floyd-Warshall Example

Iteration 3: (k = 3) No shorter paths are found through vertex 3



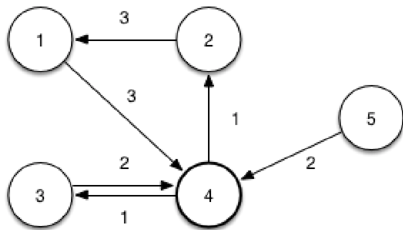
5

D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	4	1	1	0	∞
5	7	4	13	2	0

Floyd-Warshall Example

Iteration 4: (k = 4) Shorter paths from 1 $\curvearrowright$ 2, 1 $\curvearrowright$ 3, 2 $\curvearrowright$ 3, 3 $\curvearrowright$ 1, 3 $\curvearrowright$ 2, 5 $\curvearrowright$ 1, 5 $\curvearrowright$ 2, 5 $\curvearrowright$ 3, and 5 $\curvearrowright$ 4 are found through vertex 4

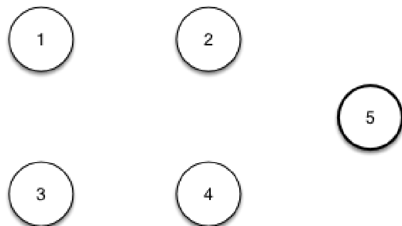


D

	1	2	3	4	5
1	0	**4**	**4**	3	∞
2	3	0	**7**	6	∞
3	**6**	**3**	0	2	∞
4	4	1	1	0	∞
5	**6**	**3**	**3**	2	0

Floyd-Warshall Example

Iteration 5: (k = 5) No shorter paths are found through vertex 5



D

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

Floyd-Warshall Implementation

```
// C++ Program for Floyd Warshall Algorithm
#include<iostream>
using namespace std;

// Number of vertices in the graph
const int V= 4;

/* Define Infinite as a large enough value. This value will be used  for vertices not connected to each other */

const int INF= 99999;

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd-Warshall algorithm

void floydWarshall (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the shortest distances between every pair of vertices */
    int dist[V][V];

    /* Initialize the solution matrix same as input graph matrix. Or we can say the initial values of shortest distances are based  on shortest paths considering no intermediate vertex. */
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate vertices.
            ---> Before start of a iteration, we have shortest distances between all pairs of vertices such that the shortest distances consider only the  vertices in set {0, 1, 2, .. k-1} as
                intermediate vertices.
            ----> After the end of a iteration, vertex no. k is added to the set of intermediate vertices and the set becomes {0, 1, 2, .. k} */
    for (int k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (int i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the above picked source
            for (int j = 0; j < V; j++)
            {
                // If vertex k is on the shortest path from i to j, then update the value of dist[i][j]
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}
```

Floyd-Warshall Implementation (cont.)

```
/* A utility function to print solution */
void printSolution(int dist[][V])
{
    cout <<"Following matrix shows the shortest distances"
           " between every pair of vertices \n";
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                cout<< "INF ";
            else
                cout<< dist[i][j]<<" ";
        }
        cout <<endl;
    }
}

// driver program to test above function
int main()
{
    int graph[V][V] = { {0,   INF,  6, 3, INF},
                        {3, 0,   INF, INF, INF},
                        {INF, INF, 0,   2, INF},
                        {INF, 1, 1, 0, INF},
                        {INF, 4, INF, 2, 0},
                      };

    // Print the solution
    floydWarshall(graph);
    return 0;
```

Problem

- Let us define a binary operation $\otimes$ on three symbols a, b, c according to the following table; thus $a \otimes b = b$, $b \otimes a = c$, and so on. Notice that the operation defined by the table is neither associative nor commutative.

$\otimes$	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

Describe an efficient algorithm that examines a string of these symbols, say *bbbbac* , and decides whether or not it is possible to parenthesize the string in such a way that the value of the resulting expression is $p = a$. For example, on input *bbbbac* your algorithm should return *yes* because $((b \otimes (b \otimes b)) \otimes (b \otimes a)) \otimes c = a$.

Solution

- For $1 \leq i \leq j \leq n$, we let $T[i, j] \subseteq \{a, b, c\}$ denote the set of symbols e for which there is a parenthesization of $x_i..x_j$ yielding e . We let $e \otimes b$ denote the product of e and b using the table. $p \in \{a, b, c\}$ is the symbol we are considering.
- Pseudocode

```
for i ← 1 to n do
    T[i, i] ← x_i
for s ← 1 to n-1 do
    for i ← 1 to n-s do
        T[i,i + s] ← ∅
        for k ← i to i+s-1 do
            for each e ∈ T[i,k] do
                for each b ∈ T[k + 1,i + s] do
                    T[i,i+s] ← T[i,i+s] ∪ e ⊗ b
if p ∈ T[1,n] then
    return yes
    else
        return no
```