Matrix Chain Multiplication (MCM)

Let's start by analyzing the cost of multiplying 2 matrices:

```
Matrix-Multiply(A,B):
 if columns[A] != columns[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]
 return C
```

Time complexity = O(pqr) where |A|=p x q and |B| = q x r

|A| = 2 * 3, |B| = 3 * 1, therefore to multiply these 2 matrices, we need O(2*3*1)=O(6) **scalar multiplication**. The result is matrix C with |C| = 2 * 1

TEST YOUR MATRIX MULTIPLICATION KNOWLEDGE

Solve UVa problems related with Matrix Multiplication:

442 - Matrix Chain Multiplication - Straightforward problem

Matrix Chain Multiplication Problem

Input: Matrices $A_1,A_2,...A_n$, each A_i of size P_{i-1} x P_i
Output: Fully parenthesized product $A_1A_2...A_n$ that minimizes the number of scalar multiplications

A product of matrices is fully parenthesized if it is either
1. a single matrix
2. the product of 2 fully parenthesized matrix products surrounded by parentheses

Example of MCM problem:
We have 3 matrices and the size of each matrix:
A_1 (10 x 100), A_2 (100 x 5), A_3 (5 x 50)

We can fully parenthesized them in two ways:
1. $(A_1 (A_2 A_3)) = 100 \times 5 \times 50 + 10 * 100 * 50 = 75000$
2. $((A_1 A_2) A_3) = 10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ (10 times better)

See how the cost of multiplying these 3 matrices differ significantly. The cost truly depend on the choice of the fully parenthesization of the matrices. However, exhaustively checking all possible parenthesizations take exponential time.

Now let's see how MCM problem can be solved using DP.

Step 1: characterize the optimal sub-structure of this problem.

Let $A_{i..j}$ ($i < j$) denote the result of multiplying $A_iA_{i+1}..A_j$. $A_{i..j}$ can be obtained by splitting it into $A_{i..k}$ and $A_{k+1..j}$ and then multiplying the sub-products. There are j-i possible splits (i.e. k=i,...,j-1)

Within the optimal parenthesization of $A_{i..j}$:
(a) the parenthesization of $A_{i..k}$ must be optimal
(b) the parenthesization of $A_{k+1..j}$ must be optimal

Because if they are not optimal, then there exist other split which is better, and we should choose that split and not this split.

Step 2: Recursive formulation

Need to find $A_{1..n}$
Let m[i,j] = minimum number of scalar multiplications needed to compute $A_{i..j}$

Since $A_{i..j}$ can be obtained by breaking it into $A_{i..k}$ $A_{k+1..j}$, we have

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

let s[i,j] be the value k where the optimal split occurs.

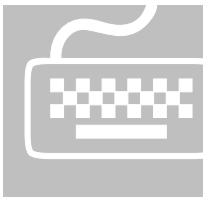
Step 3 Computing the Optimal Costs

```
Matric-Chain-Order(p)
  n = length[p]-1
```

```
  for i = 1 to n do
    m[i,i] = 0
  for l = 2 to n do
    for i = 1 to n-l+1 do
      j = i+l-1
      m[i,j] = infinity
      for k = i to j-1 do
        q = m[i,k] + m[k+1,j] + pi-1*pk*pj
        if q < m[i,j] then
          m[i,j] = q
          s[i,j] = k
  return m and s
```

Step 4: Constructing an Optimal Solution

```
Print-MCM(s,i,j)
  if i=j then
    print Ai
  else
    print "(" + Print-MCM(s,1,s[i,j]) + "*" + Print-MCM(s,s[i,j]+1,j) +
")"
```



Note: As any other DP solution, MCM also can be solved using Top Down recursive algorithm using memoization. Sometimes, if you cannot visualize the Bottom Up, approach, just modify your original Top Down recursive solution by including memoization. You'll save a lot of time by avoiding repetitive calculation of sub-problems.

TEST YOUR MATRIX CHAIN MULTIPLICATION KNOWLEDGE

Solve UVa problems related with Matrix Chain Multiplication:

348 - Optimal Array Multiplication Sequence - Use algorithm above

Longest Common Subsequence (LCS)

Input: Two sequence
Output: A longest common subsequence of those two sequences, see details below.

A sequence Z is a **subsequence** of X $<x_1,x_2,...,x_m>$, if there exists a strictly increasing sequence $<i_1,i_2,...,i_k>$ of indices of X such that for all j=1,2,..,k, we have $x_i=z_j$. example: X=<B,C,A,D> and Z=<C,A>.

A sequence Z is called **common subsequence** of sequence X and Y if Z is subsequence of both X and Y.

longest common subsequence (LCS) is just the longest "common subsequence" of two sequences.

A brute force approach of finding LCS such as enumerating all subsequences and finding the longest common one takes too much time. However, Computer Scientist has found a Dynamic Programming solution for LCS problem, we will only write the final code here, written in C, ready to use. Note that this code is slightly modified and we use global variables (yes this is not Object Oriented).

```
#include <stdio.h>
#include <string.h>
#define MAX 100

char X[MAX],Y[MAX];
int i,j,m,n,c[MAX][MAX],b[MAX][MAX];

int LCSlength() {


  m=strlen(X);
  n=strlen(Y);

  for (i=1;i<=m;i++) c[i][0]=0;
  for (j=0;j<=n;j++) c[0][j]=0;

  for (i=1;i<=m;i++)
    for (j=1;j<=n;j++) {
      if (X[i-1]==Y[j-1]) {
        c[i][j]=c[i-1][j-1]+1;
        b[i][j]=1; /* from north west */
      }


      else if (c[i-1][j]>=c[i][j-1]) {
      c[i][j]=c[i-1][j];

       b[i][j]=2; /* from north */
      }


      else {
        c[i][j]=c[i][j-1];
        b[i][j]=3; /* from west */
      }
    }
  return c[m][n];
}
```

```
void printLCS(int i,int j) {
  if (i==0 || j==0) return;

  if (b[i][j]==1) {
    printLCS(i-1,j-1);
   printf("%c",X[i-1]);}
  else if (b[i][j]==2)
    printLCS(i-1,j);
  else
    printLCS(i,j-1);
}

void main() {
  while (1) {
    gets(X);
    if (feof(stdin)) break; /* press ctrl+z to terminate */
    gets(Y);
    printf("LCS length -> %d\n",LCSlength()); /* count length */
    printLCS(m,n); /* reconstruct LCS */
    printf("\n");
  }
}
```

TEST YOUR LONGEST COMMON SUBSEQUENCE KNOWLEDGE

Solve UVa problems related with LCS:

531 - Compromise
10066 - The Twin Towers
10100 - Longest Match
10192 - Vacation
10405 - Longest Common Subsequence

Edit Distance

Input: Given two string, Cost for deletion, insertion, and replace
Output: Give the minimum actions needed to transform first string into the second one.

Edit Distance problem is a bit similar to LCS. DP Solution for this problem is very useful in Computational Biology such as for comparing DNA.

Let d(string1,string2) be the distance between these 2 strings.

A two-dimensional matrix, m[0..|s1|,0..|s2|] is used to hold the edit distance values, such that m[i,j] = d(s1[1..i], s2[1..j]).

```
m[0][0] = 0;
for (i=1; i<length(s1); i++) m[i][0] = i;
for (j=1; j<length(s2); j++) m[0][j] = j;

for (i=0; i<length(s1); i++)
  for (j=0; j<length(s2); j++) {
    val = (s1[i] == s2[j]) ? 0 : 1;
    m[i][j] = min( m[i-1][j-1] + val,
                   min(m[i-1][j]+1 , m[i][j-1]+1));
  }
```

To output the trace, use another array to store our action along the way. Trace back these values later.

TEST YOUR EDIT DISTANCE KNOWLEDGE

164 - String Computer
526 - String Distance and Edit Process

Longest Inc/Dec-reasing Subsequence (LIS/LDS)

Input: Given a sequence
Output: The longest subsequence of the given sequence such that all values in this longest subsequence is strictly increasing/decreasing.

O(N^2) DP solution for LIS problem (this code check for increasing values):

```
for i = 1 to total-1
  for j = i+1 to total
    if height[j] > height[i] then
      if length[i] + 1 > length[j] then
        length[j] = length[i] + 1
        predecessor[j] = i
```

Example of LIS

height sequence: 1,6,2,3,5
length initially: [1,1,1,1,1] - because max length is at least 1 rite...
predecessor initially: [nil,nil,nil,nil,nil] - assume no predecessor so far

```
After first loop of j:
  length: [1,2,2,2,2], because 6,2,3,5 are all > 1
  predecessor: [nil,1,1,1,1]
After second loop of j: (No change)
  length: [1,2,2,2,2], because 2,3,5 are all < 6
  predecessor: [nil,1,1,1,1]
After third loop:
  length: [1,2,2,3,3], because 3,5 are all > 2
  predecessor: [nil,1,1,3,3]

  After fourth loop:
  length: [1,2,2,3,4], because 5 > 3
  predecessor: [nil,1,1,3,4]
```

We can reconstruct the solution using recursion and predecessor array.

Is O(n^2) is the best algorithm to solve LIS/LDS ?

Fortunately, the answer is "No".

There exist an O(n log k) algorithm to compute LIS (for LDS, this is just a reversed-LIS), where k is the size of the actual LIS.

This algorithm use some invariant, where for each longest subsequence with length l, it will terminate with value A[l]. (Notice that by maintaining this invariant, array A will be naturally sorted.) Subsequent insertion (you will only do n insertions, one number at one time) will use binary search to find the appropriate position in this sorted array A

```
0  1  2  3  4  5  6  7  8
a    -7,10, 9, 2, 3, 8, 8, 1

A -i  i, i, i, i, i, i, i, i (iteration number, i = infinity)
A -i -7, i, i, i, i, i, i, i (1)
A -i -7,10, i, i, i, i, i, i (2)
A -i -7, 9, i, i, i, i, i, i (3)
A -i -7, 2, i, i, i, i, i, i (4)
A -i -7, 2, 3, i, i, i, i, i (5)
A -i -7, 2, 3, 8, i, i, i, i (6)
A -i -7, 2, 3, 8, i, i, i, i (7)
A -i -7, 1, 3, 8, i, i, i, i (8)
```

You can see that the length of LIS is 4, which is correct. To reconstruct the LIS, at each step, store the predecessor array as in standard LIS + this time remember the actual values, since array A only store the last element in the subsequence, not the actual values.

TEST YOUR LONGEST INC/DEC-REASING SUBSEQUENCE KNOWLEDGE

111 - History Grading
231 - Testing the CATCHER
481 - What Goes Up - need O(n log k) LIS
497 - Strategic Defense Initiative
10051 - Tower of Cubes
10131 - Is Bigger Smarter

Zero-One Knapsack

Input: N items, each with various Vi (Value) and Wi (Weight) and max Knapsack size MW.
Output: Maximum value of items that one can carry, if he can either take or not-take a particular item.

Let C[i][w] be the maximum value if the available items are $\{X_1,X_2,...,X_i\}$ and the knapsack size is w.

- if i == 0 or w == 0 (if no item or knapsack full), we can't take anything C[i][w] = 0
- if Wi > w (this item too heavy for our knapsack),skip this item C[i][w] = C[i-1][w];
- if Wi <= w, take the maximum of "not-take" or "take" C[i][w] = max(C[i-1][w] , C[i-1][w-Wi]+Vi);
- The solution can be found in C[N][W];

```
for (i=0;i<=N ;i++) C[i][0] = 0;
for (w=0;w<=MW;w++) C[0][w] = 0;

for (i=1;i<=N;i++)
  for (w=1;w<=MW;w++) {
    if (Wi[i] > w)
      C[i][w] = C[i-1][w];
    else
      C[i][w] = max(C[i-1][w] , C[i-1][w-Wi[i]]+Vi[i]);
  }
output(C[N][MW]);
```

TEST YOUR 0-1 KNAPSACK KNOWLEDGE

Solve UVa problems related with 0-1 Knapsack:

10130 - SuperSale

Counting Change

Input: A list of denominations and a value N to be changed with these denominations
Output: Number of ways to change N

Suppose you have coins of 1 cent, 5 cents and 10 cents. You are asked to pay 16 cents, therefore you have to give 1 one cent, 1 five cents, and 1 ten cents. Counting Change algorithm can be used to determine how many ways you can use to pay an amount of money.

The number of ways to change amount A using N kinds of coins equals to:

1. The number of ways to change amount A using all but the first kind of coins, +
2. The number of ways to change amount A-D using all N kinds of coins, where D is the denomination of the first kind of coin.

The tree recursive process will gradually reduce the value of A, then using this rule, we can determine how many ways to change coins.

1. If A is exactly 0, we should count that as 1 way to make change.
2. If A is less than 0, we should count that as 0 ways to make change.
3. If N kinds of coins is 0, we should count that as 0 ways to make change.

```
#include <stdio.h>
#define MAXTOTAL 10000
long long nway[MAXTOTAL+1];

int coin[5] = { 50,25,10,5,1 };

void main()

{
  int i,j,n,v,c;
  scanf("%d",&n);
  v = 5;
  nway[0] = 1;
  for (i=0; i<v; i++) {
    c = coin[i];
    for (j=c; j<=n; j++)
      nway[j] += nway[j-c];
  }
  printf("%lld\n",nway[n]);
}
```

TEST YOUR COUNTING CHANGE KNOWLEDGE

147 - Dollars
357 - Let Me Count The Ways - Must use Big Integer 674 - Coin Change

Maximum Interval Sum

Input: A sequence of integers
Output: A sum of an interval starting from index i to index j (consecutive), this sum must be maximum among all possible sums.

```
Numbers : -1 6
Sum     : -1 6
           ^
         max sum

Numbers : 4 -5  4 -3  4  4 -4  4 -5
Sum     : 4 -1  4  1  5  9  5  9  4
           ^              ^
         stop           max sum

Numbers : -2 -3 -4
Sum     : -2 -3 -4
          ^
       max sum, but negative... (this is the maximum anyway)
```

So, just do a linear sweep from left to right, accumulate the sum one element by one element, start new interval whenever you encounter partial sum < 0 (and record current best maximum interval encountered so far)...

At the end, output the value of the maximum intervals.

TEST YOUR MAXIMUM INTERVAL SUM KNOWLEDGE

Solve UVa problems related with Maximum Interval Sum:

507 - Jill Rides Again

Other Dynamic Programming Algorithms

Problems that can be solved using Floyd Warshall and it's variant, which belong to the category all-pairs shortest path algorithm, can be categorized as Dynamic Programming solution. Explanation regarding Floyd Warshall can be found in Graph section.

Other than that, there are a lot of ad hoc problems that can utilize DP, just remember that when the problem that you encountered exploits optimal sub-structure and repeating sub-problems, apply DP techniques, it may be helpful.

TEST YOUR DP KNOWLEDGE

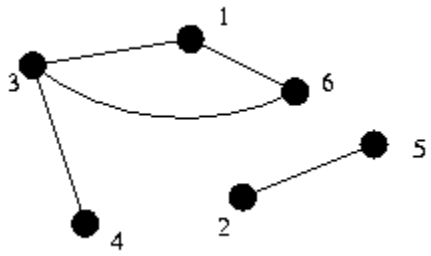
Solve UVa problems related with Ad-Hoc DP:

108 - Maximum Sum
836 - Largest Submatrix
10003 - Cutting Sticks
10465 - Homer Simpson

CHAPTER 12 GRAPHS

A *graph* is a collection of *vertices* V and a collection of *edges* E consisting of pairs of vertices. Think of vertices as ``locations''. The set of vertices is the set of all the possible locations. In this analogy, edges represent paths between pairs of those locations. The set E contains all the paths between the locations.[2]

Vertices and Edges



The graph is normally represented using that analogy. Vertices are points or circles, edges are lines between them.

In this example graph:
V = {1, 2, 3, 4, 5, 6}
E = {(1,3), (1,6), (2,5), (3,4), (3,6)}.

Each *vertex* is a member of the set V. A vertex is sometimes called a *node*.

Each *edge* is a member of the set E. Note that some vertices might not be the end point of any edge. Such vertices are termed ``*isolated*''.

Sometimes, numerical values are associated with edges, specifying lengths or costs; such graphs are called *edge-weighted* graphs (or *weighted* graphs). The value associated with an edge is called the *weight* of the edge. A similar definition holds for node-weighted graphs.

Telecommunication

Given a set of computers and a set of wires running between pairs of computers, what is the minimum number of machines whose crash causes two given machines to be unable to communicate? (The two given machines will not crash.)

Graph: The vertices of the graph are the computers. The edges are the wires between the computers. Graph problem: minimum dominating sub-graph.

Riding The Fences

Farmer John owns a large number of fences, which he must periodically check for integrity. He keeps track of his fences by maintaining a list of points at which fences intersect. He records the name of the point and the one or two fence names that touch that point. Every fence has two end points, each at some intersection point, although the intersection point may be the end point of only one fence.

Given a fence layout, calculate if there is a way for Farmer John to ride his horse to all of his fences without riding along a fence more than once. Farmer John can start and finish anywhere, but cannot cut across his fields (i.e., the only way he can travel between intersection points is along a fence). If there is a way, find one way.

Graph: Farmer John starts at intersection points and travels between the points along fences. Thus, the vertices of the underlying graph are the intersection points, and the fences represent edges. Graph problem: Traveling Salesman Problem.

Knight moves

Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.

Graph: The graph here is harder to see. Each location on the chessboard represents a vertex. There is an edge between two positions if it is a legal knight move. Graph Problem: Single Source Shortest Path.

Overfencing

Farmer John created a huge maze of fences in a field. He omitted two fence segments on the edges, thus creating two ``exits'' for the maze. The maze is a ``perfect'' maze; you can find a way out of the maze from any point inside it.

Given the layout of the maze, calculate the number of steps required to exit the maze from the ``worst'' point in the maze (the point that is ``farther'' from either exit when walking optimally to the closest exit).

Here's what one particular W=5, H=3 maze looks like:

```
+-+-+-+-+-+
|       |
+-+ +-+ + +
|   | | |
+ +-+-+ + +
| |   |
+-+ +-+-+-+
```

Graph: The vertices of the graph are positions in the grid. There is an edge between two vertices if they represent adjacent positions that are not separated by a wall.

Terminology

An edge is a *self-loop* if it is of the form (u,u). The sample graph contains no self-loops.

A graph is *simple* if it neither contains self-loops nor contains an edge that is repeated in E. A graph is called a *multigraph* if it contains a given edge more than once or contain self-loops. For our discussions, graphs are assumed to be simple. The example graph is a simple graph.

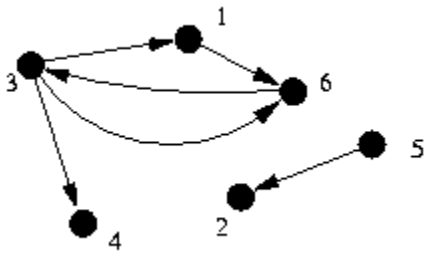
An edge (u,v) is *incident* to both vertex u and vertex v. For example, the edge (1,3) is incident to vertex 3.

The *degree* of a vertex is the number of edges which are incident to it. For example, vertex 3 has degree 3, while vertex 4 has degree 1.

Vertex u is *adjacent* to vertex v if there is some edge to which both are incident (that is, there is an edge between them). For example, vertex 2 is adjacent to vertex 5.

A graph is said to be *sparse* if the total number of edges is small compared to the total number possible ((N x (N-1))/2) and *dense* otherwise. For a given graph, whether it is dense or sparse is not well-defined.

Directed Graph



Graphs described thus far are called *undirected*, as the edges go `both ways'. So far, the graphs have connoted that if one can travel from vertex 1 to vertex 3, one can also travel from vertex 1 to vertex 3. In other words, (1,3) being in the edge set implies (3,1) is in the edge set.

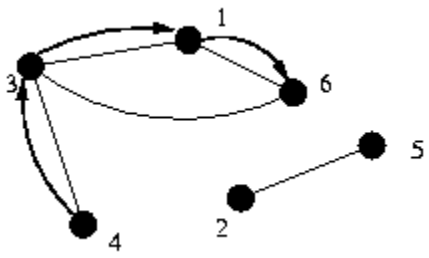
Sometimes, however, a graph is *directed*, in which case the edges have a direction. In this case, the edges are called *arcs*.

Directed graphs are drawn with arrows to show direction.

The *out-degree* of a vertex is the number of arcs which begin at that vertex. The *in-degree* of a vertex is the number of arcs which end at that vertex. For example, vertex 6 has in-degree 2 and out-degree 1.

A graph is assumed to be undirected unless specifically called a directed graph.

Paths



A *path* from vertex u to vertex x is a sequence of vertices (v 0, v 1, ..., v k) such that v 0 = u and v k = x and (v 0, v 1) is an edge in the graph, as is (v 1, v 2), (v 2, v 3), etc. The length of such a path is k.

For example, in the undirected graph above, (4, 3, 1, 6) is a path.

This path is said to *contain* the vertices v 0, v 1, etc., as well as the edges (v 0, v 1), (v 1, v 2), etc.

Vertex x is said to be *reachable* from vertex u if a path exists from u to x.

A path is *simple* if it contains no vertex more than once.

A path is a *cycle* if it is a path from some vertex to that same vertex. A cycle is *simple* if it contains no vertex more than once, except the start (and end) vertex, which only appears as the first and last vertex in the path.

These definitions extend similarly to directed graphs (e.g., (v 0, v 1), (v 1, v 2), etc. must be arcs).

Graph Representation

The choice of representation of a graph is important, as different representations have very different time and space costs.

The vertices are generally tracked by numbering them, so that one can index them just by their number. Thus, the representations focus on how to store the edges.

Edge List

The most obvious way to keep track of the edges is to keep a list of the pairs of vertices representing the edges in the graph.

This representation is easy to code, fairly easy to debug, and fairly space efficient. However, determining the edges incident to a given vertex is expensive, as is determining if two vertices are adjacent. Adding an edge is quick, but deleting one is difficult if its location in the list is not known.

For weighted graphs, this representation also keeps one more number for each edge, the edge weight. Extending this data structure to handle directed graphs is straightforward. Representing multigraphs is also trivial.

Example

	V1	V2
e1	4	3
e2	1	3
e3	2	5
e4	6	1
e5	3	6

Adjacency Matrix

A second way to represent a graph utilized an *adjacency matrix*. This is a N by N array (N is the number of vertices). The i,j entry contains a 1 if the edge (i,j) is in the graph; otherwise it contains a 0. For an undirected graph, this matrix is symmetric.

This representation is easy to code. It's much less space efficient, especially for large, sparse graphs. Debugging is harder, as the matrix is large. Finding all the edges incident to a given vertex is fairly expensive (linear in the number of vertices), but checking if two vertices are adjacent is very quick. Adding and removing edges are also very inexpensive operations.

For weighted graphs, the value of the (i,j) entry is used to store the weight of the edge. For an unweighted multigraph, the (i,j) entry can maintain the number of edges between the vertices. For a weighted multigraph, it's harder to extend this.

Example

The sample undirected graph would be represented by the following adjacency matrix:

	V1	V2	V3	V4	V5	V6
V1	0	0	1	0	0	1
V2	0	0	0	0	1	0
V3	1	0	0	1	0	1
V4	0	0	1	0	0	0
V5	0	1	0	0	0	0
V6	1	0	1	0	0	0

It is sometimes helpful to use the fact that the (i,j) entry of the adjacency matrix raised to the k-th power gives the number of paths from vertex i to vertex j consisting of exactly k edges.

Adjacency List

The third representation of a matrix is to keep track of all the edges incident to a given vertex. This can be done by using an array of length N, where N is the number of vertices. The i-th entry in this array is a list of the edges incident to i-th vertex (edges are represented by the index of the other vertex incident to that edge).

This representation is much more difficult to code, especially if the number of edges incident to each vertex is not bounded, so the lists must be linked lists (or dynamically allocated). Debugging this is difficult, as following linked lists is more difficult. However, this representation uses about as much memory as the edge list. Finding the vertices adjacent to each node is very cheap in this structure, but checking if two vertices are adjacent requires checking all the edges adjacent to one of the vertices. Adding an edge is easy, but deleting an edge is difficult, if the locations of the edge in the appropriate lists are not known.

Extend this representation to handle weighted graphs by maintaining both the weight and the other incident vertex for each edge instead of just the other incident vertex. Multigraphs are already representable. Directed graphs are also easily handled by this representation, in one of several ways: store only the edges in one direction, keep a separate list of incoming and outgoing arcs, or denote the direction of each arc in the list.

Example

The adjacency list representation of the example undirected graph is as follows:

Vertex	Adjacent Vertices
1	3, 6
2	5
3	6, 4, 1
4	3
5	2
6	3, 1

Implicit Representation

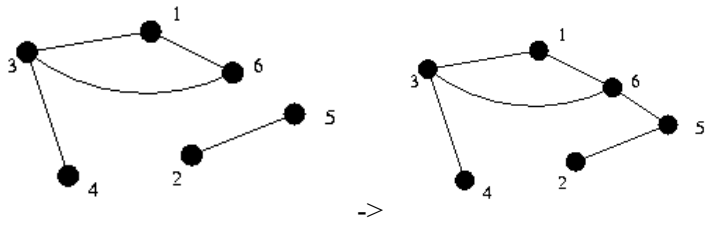
For some graphs, the graph itself does not have to be stored at all. For example, for the Knight moves and Overfencing problems, it is easy to calculate the neighbors of a vertex, check adjacency, and determine all the edges without actually storing that information, thus, there is no reason to actually store that information; the graph is implicit in the data itself.

If it is possible to store the graph in this format, it is generally the correct thing to do, as it saves a lot on storage and reduces the complexity of your code, making it easy to both write and debug.

If N is the number of vertices, M the number of edges, and d max the maximum degree of a node, the following table summarizes the differences between the representations:

Efficiency	Edge List	Adj Matrix	Adj List
Space	2*M	N^2	2xM
Adjacency Check	M	1	d max
List of Adjacent Vertices	M	N	d max
Add Edge	1	1	1
Delete Edge	M	2	2*d max

Connectedness



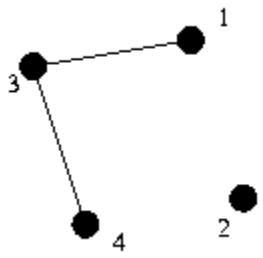
An undirected graph is said to be *connected* if there is a path from every vertex to every other vertex. The example graph is not connected, as there is no path from vertex 2 to vertex 4. However, if you add an edge between vertex 5 and vertex 6, then the graph becomes connected.

A *component* of a graph is a maximal subset of the vertices such that every vertex is reachable from each other vertex in the component. The original example graph has two components: {1, 3, 4, 6} and {2, 5}. Note that {1, 3, 4} is not a component, as it is not maximal.

A directed graph is said to be *strongly connected* if there is a path from every vertex to every other vertex.

A *strongly connected component* of a directed graph is a vertex u and the collection of all vertices v such that there is a path from u to v and a path from v to u.

Subgraphs

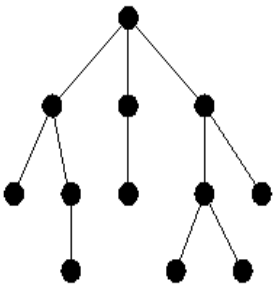


Graph G' = (V', E') is a subgraph of G = (V, E) if V' is a subset of V and E' is a subset of E.

The subgraph of G *induced* by V' is the graph (V', E'), where E' consists of all the edges of E that are between members of V'.

For example, for V' = {1, 3, 4, 2}, the subgraph is like the one shown ->

Tree

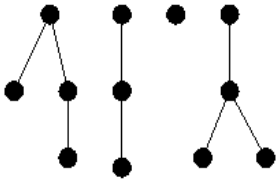


An undirected graph is said to be a *tree* if it contains no cycles and is connected.

A rooted tree

Many trees are what is called *rooted*, where there is a notion of the "top" node, which is called the root. Thus, each node has one *parent*, which is the adjacent node which is closer to the root, and may have any number of *children*, which are the rest of the nodes adjacent to it. The tree above was drawn as a rooted tree.

Forest



An undirected graph which contains no cycles is called a *forest*. A directed acyclic graph is often referred to as a *dag*.