Number Theory

Binomial Coefficients

$\binom{n}{k}$ (read "n choose k") This is number of ways to select *k* items from *n* possibilities. The coefficients in a nth power polynomial: $(a+b)^n$ The $a^k b^{n-k}$ term is n choose k..

Pascal's Triangle

```
      1
     1 1
    1 2 1
   1 3 3 1
```

The (n+1)st row gives the values of n choose i for 0 <=i <=n.

Computation: n choose k = n!/(n-k)!k!

Since factorials are too hard to compute, can use recurrence relation:

$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ (remembering that when k=0, n choose k is zero and when k=1, n choose k is n. Dynamic programming solution:

```
#define MAXN 100 // largest n or m
long binomial_coefficient(n,m) // compute n choose m
int n,m;
{
    int i,j;
    long bc[MAXN][MAXN];

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

Catalan Numbers

$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} = \frac{1}{n+1}\binom{2n}{n}$ The first terms of this sequence are 2, 5, 14, 42, 132, 429, 1430 when $C_0 = 1$.This is the number of ways to build a balanced formula from n sets of left and right parentheses. It is also the number of triangulations of a convex polygon, the number of rooted binary tress on n+1 leaves and the number of paths across a lattice which do not rise above the main diagonal.

Eulerian numbers

$\left\langle {n \atop k} \right\rangle = k\left\langle {n-1 \atop k} \right\rangle + (n-k+1)\left\langle {n-1 \atop k-1} \right\rangle$ This is the number of permutations of length *n* with exactly *k* ascending sequences or runs. (Basis: k=0 has value 1)

```
#define MAXN 100 // largest n or k
long eularian(n,k)
int n,m;
{
    int i,j;
    long e[MAXN][MAXN];

    for (i=0; i<=n; i++) e[i][0] = 1;
    for (j=0; j<=n; j++) e[0][j] = 0;
    for (i=1; i<=n; i++)
        for (j=1; j<i; j++)
            e[i][j] = k*e[i-1][j] + (i-j+1)*e[i-1][j-1];
    return e[n][k];
}
```

GCD/GCF

Greatest Common Denominator – Euclid's Algorithm

```
A = max(n1, n2);
B = min(n2, n3);
while (A% B!= 0)
{
    C = A%B;
    A = B;      //Note that to find the GCD for more numbers,
            //take the GCD of each
    B = C;
}   // additional number and the result, B, from the last set.
Return B;
```

LCM

Least Common Denominator

```
    LCM(A, B) = A* B / GCD(A, B)
```

Partitions of Integers

An integer partition of *n* is an unordered set of positive integer which adds up to *n*. The number of integer partitions of *n* with the largest element is at most *k* is: f(n,k) = f(n-k,k)+f(n,k-1) where f(1,1) = 1 and f(n,k) = 0 if k>n or k< =0

```
#define MAXN 100 // largest n or m
long int_coefficient(n,k) // compute f(n,k)
int n,m;
{
    int i,j;
    long f[[MAXN][MAXN];

    f [1][1] = 1;
    for (i=0;i<=n;i++) f[i][0] = 0;
    for (i=1; i<=n; i++)
        for (j=1; j<i; j++)
            if (i-j <= 0)
                f[i][j] = f[i][k-1];
            else
                f[i][j] = f[i-j][k]+f[i][k-1];
    return f[n][k];
}
```

Partitions of Sets

$\left\{ \begin{matrix} n \\ k \end{matrix} \right\} = k \left\{ \begin{matrix} n-1 \\ k \end{matrix} \right\} + \left\{ \begin{matrix} n-1 \\ k \end{matrix} \right\}$ counts the number of ways to partition *n* items into *k* sets. Basis: when k=2, this has a value of 2^{n-1}-1

Primeness Fast

Fermat's Theorem, which states that a^tot(b) % b = 1 where the GCF of b and a is zero, can be used to determine primeness.

tot(b) = count of numbers smaller than b where the GCF between b and the number is 1.
For primes, tot(b) = b – 1.

Therefore, if b is prime, a^(b-1) should equal 1 modulus b if a and b are relatively prime. The powermod algorithm takes advantage of the fact that (a % b)(c % b) = (a * c) % b.

```
bool isPrime(long input)
{
    long cpowmod = 1;
    long cpow;
    long targpwr;
    //Take care of the special freak cases.
    if (input==1)
        return false;
    if (input==2)
        return true;
    // The base I am choosing, two, must be relatively prime to
    // the target.
    if (input%2==0)
        return false;
    //If this is prime, then 2^p-1 will equal 1.  If not, dream on.
    targpwr = input - 1;
    //My chosen base to start with is 2.
    cpow = 2;
    //Now split target power into binary.  a*b mod n = amodn * bmodn
    while(targpwr > 0)
    {
```

```
        //binary 1
        if (targpwr % 2==1)
        {
            targpwr -= 1;
            //Why it's called powermod...
            cpowmod *= cpow;
            cpowmod %= input;
        }
        //Divide targpwr by 2
        targpwr >>=1;
        //Get the next powermod of 2.
        cpow *= cpow;
        cpow %= input;
    }
    //If I'm left with 1, this is prime.
    if (cpowmod==1)
        return true;
    else
        return false;
}
```

Prime Factorization

Unfortunately, no great way to factor numbers into their prime factors exists. This function will return the smallest prime factor of a number.

```
function factor(int n)
{
    int a;
    if (n%2==0)
        return 2;
    for (a=3;a<=sqrt(n);a++++)
    {
        if (n%a==0)
        return a;
    }
    return n;
}
```

By putting it in a while loop, you can find the complete prime factorization.

```
int r;
while (n>1)
{
    r = factor(n);
    printf("%d ", r);
    n /= r;
}
```

Probability

```
P(event) = (Count desired outcomes) / (Count all possible outcomes)
P(not x) = 1 – P(x)
```

If the events x and y do not depend on one another, P(x and y) = P(x) * P(y)

If the events x and y depend on one another, P(x and y) = P(not x) * P(y given not x) + P(x) + P(y given x)

```
P(x exclusive or y) = P(x and not y) + P(not x and y)
P(x inclusive or y) = P(x and not y) + P(not x and y) + P(x and y)
```

Geometry

Angles or Sides of Triangles

If a triangle has three sides (a, b, and c) and three angles (A, B, and C) where A is opposite a, etc. these rules apply:
sine rule: $\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$
cosine rule:
$a^2 = b^2 + c^2 - 2bc\cos A$ and $b^2 = a^2 + c^2 - 2ac\cos B$ and $c^2 = b^2 + a^2 - 2ba\cos C$
So, to get the angles from the sides:

1. Use the cosine rule to find the largest angle $\cos C = \left(a^2 + b^2 - c^2\right)/2ab$ where c is the longest side
2. Use the sine rule to find the second angles
3. Use the fact that the sum of the angles is 180 to find the last angle.

Area of a Polygon with Vertices on Lattice Points: Pick's Theorem

```
Area = {Number of Lattice Points on the Inside} +
       {Number of Lattice Points on the Border} / 2 – 1
```

Note: Works for ANY polygon, concave or convex.

Area of a Polygon:

Green's Theorem: Area = ½ * Sum({Sum of Coordinates Cross Product of each pair of adjacent points. Consistently clockwise or counterclockwise}) Note: Sign of result indicates whether clockwise (-) or counterclockwise (+).

Collinearity

If the dot product of two vectors is zero, they are orthogonal. Take the three points and then create two vectors, then check for collinearity with the dot product.

Geometric Classes

```
class Point
{
    public:
    double x;
    double y;
    Point(double nx = 0, double ny = 0)
    {
        x = nx; y = ny;
    }
    bool operator == (Point const &pt) const
    {
        return ( (x == pt.x) && (y == pt.y) );
    }
    void print() const
    {
        cout << "(" << x << "," << y << ")" << " ";
    }
};
class Line
{
    public:
    double a,b,c;
    Line(double na = 0, double nb = 0, double nc = 0)
    {
        a = na; b = nb; c = nc;
    }
    Line(Point p1, Point p2)
    {
        if (p1.x == p2.x)
        {
            a = 1;
            b = 0;
            c = -p1.x;
        }
        else
        {
            b = 1;
            a = -(p1.y-p2.y)/(p1.x-p2.x);
            c = -(a * p1.x) - (b * p1.y);
        }
    }
    bool operator == (Line const &ln)
    {
```

```
        return ( (a == ln.a) && (b == ln.b) && (c == ln.c) );
    }
};
bool pointlessthan(Point left, Point right)
{
    if (left.x < right.x) return true;
    if ((left.x == right.x) && (left.y < right.y)) return true;
    return false;
}
bool ccw(Point p1, Point p2, Point p3)
{
    // true if p3 is to the left of the line made by p1 and p2
    double k = p1.x*p2.y + p3.x*p1.y + p2.x*p3.y - p3.x*p2.y - p2.x*p1.y - p1.x*p3.y;
    return (k < 0.0000000000000001);
}
bool collinear(Point p1, Point p2, Point p3)
{
    return ( Line(p1, p2) == Line(p2, p3) );
}
double sqr(double a)
{
    return a * a;
}
double pt_distance(Point p1, Point p2)
{
    return sqrt( sqr(p1.x - p2.x) + sqr(p1.y - p2.y) );
}
void printpoint(Point &pt)
{
    pt.print();
}
```

Polygon Area

```
double polygon_area(vector<Point> poly)
{
    double total = 0.0, val = 0.0;
    int i, j;
    for (i=0; i < poly.size(); i++)
    {
        j = (i+1) % (poly.size());
        val = (poly[i].x * poly[j].y) –
            (poly[j].x * poly[i].y);
        total += val;
    }
    return total / 2;
}
```

```
int main()
{
    vector<Point> poly;
    // Points must be in CCW order
    poly.resize(3);
    poly[0] = Point(2,2);
    poly[1] = Point(5,2);
    poly[2] = Point(5,6);
    cout << polygon_area(poly) << endl;
}
```

Convex Hull

```
Point first_point;
bool smaller_angle(Point p1, Point p2)
{
    if (collinear(first_point, p1, p2))
    {
        if (pt_distance(first_point, p1) <=
                pt_distance(first_point, p2))
            return true;
        else
            return false;
    }
    if (ccw(first_point, p1, p2))
        return true;
    else
        return false;
}
void GSHull(list<Point> ps)
{
    vector<Point> hull;
    hull.resize(ps.size());

    int n = ps.size();
    if (n <= 3) // all points on hull
    {
        for (list<Point>::iterator cur = ps.begin();
                        cur != ps.end(); cur++)
            hull.push_back(*cur);
```

```
    }
    else
    {
        ps.unique();
        ps.sort(pointlessthan);
        //for_each(ps.begin(), ps.end(), printpoint);
        //cout << endl;
        hull[0] = *ps.begin();
        ps.pop_front();
        first_point = hull[0];
        ps.sort(smaller_angle);
        hull[1] = *ps.begin();
        ps.pop_front();
        ps.push_back(first_point);

        int top = 1;
        for (list<Point>::iterator cur = ps.begin();
                    cur != ps.end();)
        {
            if (!ccw(hull[top-1], hull[top], *cur))
                top--;
            else
            {
                top++;
                hull[top] = *cur;
                cur++;
            }
        }
        hull.resize(top);
    }
    cout << "Hull:" << endl;
    for_each(hull.begin(), hull.end(), printpoint);
}
```

```
int main()
{
    list<Point> ps;
    ps.push_back(Point(8,9));
    ps.push_back(Point(6,8));
    ps.push_back(Point(6,11));
    ps.push_back(Point(11,9));
    ps.push_back(Point(10,5));
    ps.push_back(Point(4,8));
    ps.push_back(Point(6,13));
    ps.push_back(Point(6,4));
    ps.push_back(Point(10,12));
    ps.push_back(Point(10,2));
    ps.push_back(Point(13,11));
    ps.push_back(Point(12,6));
    GSHull(ps);
    return 0;
}
```

Triangulation

```
class Triangulation
{
    public:
    int n;
    int t[MAXPOLY][3];
    Triangulation()
    {
        n = 0;
    }
    void addTriangle(int a, int b, int c, vector<Point> &p)
    {
        n++;
        t[n][0] = a; t[n][1] = b; t[n][2] = c;
        printpoint(p[a]); printpoint(p[b]);
            printpoint(p[c]); cout << endl;
    }
};
class Triangle
{
    public:
    Point a,b,c;
    Triangle(Point na = Point(), Point nb = Point(),
            Point nc = Point())
    {
        a = na; b = nb; c = nc;
    }
    bool pointIn(Point p)
    {
        if (ccw(a,b,p)) return false;
        if (ccw(b,c,p)) return false;
        if (ccw(c,a,p)) return false;
        return true;
    }
};
bool ear_Q(int i, int j, int k, vector<Point> &p)
{
    Triangle t(p[i], p[j], p[k]);
    if (ccw(p[i], p[j], p[k])) return false;
    for (int m = 0; m < p.size(); m++)
        if ((m != i) && (m != j) && (m != k))
```

```
            if (t.pointIn(p[m])) return false;
    return true;
}
void triangulate(vector<Point> &p, Triangulation &t)
{
    vector<int> l, r;
    l.resize(MAXPOLY);
    r.resize(MAXPOLY);
    int i;

    for (i = 0; i < p.size(); i++)
    {
        l[i] = ((i-1) + p.size()) % p.size();
        r[i] = ((i+1) + p.size()) % p.size();
    }

    t.n = 0;
    i = p.size() - 1;
    int numtriangles = p.size()-2;
    while (t.n < numtriangles)
    {
        i = r[i];
        if (ear_Q(l[i],i,r[i],p))
        {
            t.addTriangle(l[i], i, r[i], p);
            l[ r[i] ] = l[i];
            r[ l[i] ] = r[i];
        }
        else
            numtriangles--;
    }
}
```

```
int main()
{
    vector<Point> poly;
    Triangulation t;

    poly.resize(6);
    poly[0] = Point(1,1);
    poly[1] = Point(3,1);
    poly[2] = Point(3,4);
    poly[3] = Point(2,4);
    poly[4] = Point(1,3);
    poly[5] = Point(2,2);
    triangulate(poly,t);
    return 0;
}
```

Trigonometry

Sometimes it is necessary to convert a slope in rise/run to radians, which can be done with C's atan2(rise, run) function, or atan2(dy, dx). If you need to convert a radian measure to rise/run, you can use tan(angle), or sin(angle) / cos(angle).

Alpha / beta relations sin(a+B) = (sin a)(cos B)+(cos a)(sin B) cos(a+B) = (cos a)(cos B)-(sin a)(sin B) tan(a+B) = (tan a+tan B)/(1-(tan a)(tan B)) sin(a-B) = (sin a)(cos B) - (cos a)(sin B) cos(a-B) = (cos a)(cos B) + (sin a)(sin B) tan(a-B) = (tan a-tan B)/(1+(tan a)(tan B))	**Pythagorean Relations** $\sin^2 a + \cos^2 a = 1$ $1 + \tan^2 a = \sec^2 a$ $\cot^2 a + 1 = \csc^2 a$
Cofunction Relations sin a = cos(90 - a) cos a = sin(90 - a) tan a = cot(90 - a) cot a = tan(90 - a) sec a = csc(90 - a) csc a = sec(90 - a)	**Reciprocal Relations** sin a = 1 / csc a cos a = 1 / sec a tan a = 1 / cot a cot a = 1 / tan a sec a = 1 / cos a csc a = 1 / sin a
Double angle relations sin 2a = 2(sin a)(cos a) $\cos 2a = \cos^2 a - \sin^2 a$ $\cos 2a = 1 - 2\sin^2 a$ $\cos 2a = 2\cos^2 a - 1$ $\tan 2a = (2\tan a)/(1 - \tan^2 a)$	**Quotient Relations** tan a = (sin a)/(cos a) cot a = (cos a)/(sin a)
Half angle relations $\sin x/2 = \pm((1 - \cos x)/ 2)^{1/2}$ $\cos x/2 = \pm((1 + \cos x)/ 2)^{1/2}$ $\tan x/2 = \pm((1 - \cos x)/(1 + \cos x))^{1/2}$	**Sum and differences to product relations** sin x+sin y = 2(sin(x+y/2)(cos(x - y)/ 2) sin x - sin y = 2(cos(x+y)/2)(sin(x-y)/ 2) cos x + cos y = 2(cos(x + y)/2)(cos(x-y)/2) cos x - cos y = -2(sin(x+y)/2)(sin(x-y)/2) $\sin^2 x = (1 / 2)(1 - \cos 2x)$ $\cos^2 x = (1 / 2)(1 + \cos 2x)$
Law of sines a / sin A = b / sin B = c / sin C	**Law of cosines** $a^2 = b^2 + c^2 - 2bc(\cos A)$ $b^2 = a^2 + c^2 - 2ac(\cos B)$ $c^2 = a^2 + b^2 - 2ab(\cos C)$

String Algorithms

Longest Common Sequence

This is a dynamic programming solution to finding the longest common (not necessarily consecutive) sequence in two strings. For example, if the two strings are abcbdab and bdcaba, the lcs is bcba. The c [i][j] is the length of the longest sequence using the first i chars of the first string and the first j chars of the second string. The b array is used to remember which part of the recurrence we used so that we can rebuild the lcs at the end.

$$c[i,j] = \begin{cases} 0 & or(i=0, j=0) \\ c[i-1, j-1]+1 & x_i = y_j \\ \max(c[i, j-1], c[i-1, j] & x_i \neq y \end{cases}$$

```
#include <stdio.h>
#include <string.h>
#define MAX_CHARS 8
#define DIAG 0
#define UP 1
#define RIGHT 2
int c[MAX_CHARS][MAX_CHARS];  // length of lcs
int b[MAX_CHARS][MAX_CHARS];  // direction we used in the table
void lcs_build_table(char* s1, char *s2)
{
    int m = strlen(s1); int n = strlen(s2);
    for (int i=1; i<=m; i++)
        c[i][0] = 0;
    for (int j=0; j<=n; j++)
        c[0][j] = 0;
    for (int i=1; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            if (s1[i-1] == s2[j-1])
            {
                c[i][j] = c[i-1][j-1]+1;
                b[i][j] = DIAG;
            }
            else if (c[i-1][j] >= c[i][j-1])
            {
                c[i][j] = c[i-1][j];
                b[i][j] = UP;
            }
            else
            {
                c[i][j] = c[i][j-1];
                b[i][j] = RIGHT;
            }
        }
    }
}
int main()
{
    char s1[MAX_CHARS],s2[MAX_CHARS];
    while (gets(s1))
    {
        gets(s2);
        lcs_build_table(s1,s2);
        print_lcs(strlen(s1),strlen(s2),s1);
        printf("\n");
    }
}
```

```
void print_lcs(int i, int j, char *s)
{
    if ((i==0) || (j==0))  return;
    if (b[i][j] == DIAG)
    {
        print_lcs(i-1,j-1,s);
        printf("%c",s[i-1]);
    }
    else if (b[i][j] == UP)
        print_lcs(i-1,j,s);
    else
        print_lcs(i,j-1,s);
}
```

Pattern Matching

This is the Boyer-Moore algorithm for searching for a pattern (string) within another string. It gets its efficiency by looking at the pattern from the end instead of the beginning so that it can shift the pattern by multiply spaces at each mismatch (depending on the character that didn't match)

```
#include <stdio.h>
#include <string.h>
char s[80];
char p[80];

int last[128];

void compute_last()
{
    for (int i=33; i<128; i++)
        last[i] = 0;
    for (int j = 0; j<strlen(p); j++)
        last[p[j]] = j+1;
```

```
}

int main()
{
    while (gets(s))
    {
        gets(p);
        int n = strlen(s);
        int m = strlen(p);
        compute_last();
        int t = 0;
        while (t <= n-m)
        {
            int j = m-1;
            while ((j>=0) && (p[j] == s[t+j]))
                j--;
            if (j == -1)
            {
                printf( "pattern match at %d\n",t);
                t++;
            }
            else
            {
                int mismatch = (int)s[t+j];
                if (j < last[mismatch])
                    t++;
                else
                    t = t+j-last[mismatch]+1;
            }
        }
    }
}
```

Graph Algorithms

Storing in arrays

Rather than using a data structure, it can be much easier to store data in binary trees in a linear array. Place the head node in position 1. Now, the left child of any node can be placed in 2n and the right child can be placed in 2n+1 with no gaps or redundancy.

```
Left child: 2*n+1
Right child: 2*n+2
Parent: (n-1)/2 truncated
Level: log2(n+1) = log(n+1) / log(2) truncated
```

Storing a graph explicitly:

```
#define MAXV 100 // max number of vertices
#define MAXDEGREE 50   // max out degree of a vertex
typedef struct {
    int edges[MAXV+1][MAXDEGREE];  // storing edges
               // in adjacency lists
    int degree[MAXV+1]; // degree of each vertex
    int nvertices;
    int nedges;
} graph;

void insert_edge(graph *g, int x, int y, bool
directed)
{
    if (g->degree[x] > MAXDEGREE)
        printf("Error: degree: %d %",x,y);
    g->edges[x][g->degree[x]] = y;
    g->degree[x]++;
    if (directed == FALSE)
        insert_edge(g,y,x,TRUE);
    else
        g->nedges++;
}

void print_graph(graph *g)
{
    int i,j;
    for(i=1;i<=g->nvertices;i++)
    {
        printf("%d:  ",i);
        for (j=0;j<g->degree[i];j++)
            printf(" %d",g->edges[i][j]);
        printf("\n");
    }
}
```

```
void init_graph(graph *g)
{
    int i;
    g->nvertices = 0;
    g->nedges = 0;
    for (i=1;i<=MAXV; i++) g->degree[i] = 0;
}
```

Breadth First Search

Remember:

- each vertex goes from undiscovered to discovered to processed
- parent[i] is the vertex that we discovered the ith vertex from (we traveled the edge from i to this node. You can use this to recreate the path backwards (see the **find_path** method which takes the starting node, the ending node and the parents array as its parameters).
- this will always give **a** shortest path
- you may need to do something for each vertex or each edge. Use process_vertex or process_edge as appropriate

```
void bfs(graph *g, int start){
    deque<int> q;
    int v;  // current vertex;
    int i;  // counter

    q.push_front(start);
    while (q.empty() == FALSE){
        v = q.front();
        q.pop_front();
        process_vertex(v); // do whatever
                   // you're supposed
                   // to do at each vertex
        processed[v] = TRUE;
        for (i=0;i<g->degree[v];i++) {
            if (discovered[g->edges[v][i]] == FALSE){
                q.push_front(g->edges[v][i]);
                discovered[g->edges[v][i]] = TRUE;
```

```
#include <deque>
using namespace std;
bool processed[MAXV];
bool discovered[MAXV];
bool parent[MAXV];
void init_search(graph *g){
    int i;
    for (i=1; i<=g->nvertices; i++) {
        processed[i] = discovered[i] = FALSE;
        parent[i] = -1;
    }
}
```

```
                parent[g->edges[v][i]] = v;
            }
            if (processed[g->edges[v][i]] = FALSE)
                process_edge(v,g->edges[v][i]);
        }
    }
}
void find_path(int start, int end, int parents[])
{
    if ((start == end) || (end == -1))
        printf("\n%d",start);
    else{
        find_path(start, parent[end], parents);
        printf(" %d",end);
    }
}
```

```
int main()
{
    graph realg;
    graph *g = &realg;
    init_graph(g);
    for (int i=1;i<5;i++)
    {
        insert_edge(g,i, i+1,TRUE);
    }
    insert_edge(g,3,5,FALSE);
    insert_edge(g,1,4,FALSE);
    bfs(g,1);
    find_path(1,4,parent);
}
```

Connected Components

This will print out the vertices in each component of a graph; if you make process_vertex just print out the number of that vertex: printf(" %d",v);

```
connected_components(graph *g)
{
    int c,i;
    init_search(g);
    c = 0;
    for (i=1;i<=g->nvertices;i++){
        if (discovered[i] == FALSE){
            c++;
            printf("Component %d:",c);
            dfs(g,i);
            printf("\n");
        }
    }
}
```

Cycle Detection

To find cycles in a graph, use dfs with this process_edge method (and no need for process_vertex)

```
process_edge(int x, int y)
{
    // if we've been here from a different parent, we found a back
    // edge which means we found a cycle
    if (parent[x] != y) {
        printf("Cycle from %d to %d:",y,x);
        find_path(y,x,parent);
        finished = TRUE;
    }
```

Depth First Search on a Graph

Remember

- Uses the same graph structure and initialization as bfs
- the boolean "finished" can be used to stop the search before traversing the entire graph if necessary. Just set it "true" based on the appropriate condition

```
dfs(graph *g, int v){
    int i;
    int y;
    if (finished) return;
    discovered[v] = TRUE;
    process_vertex(v);//do what you need to do as you see each vertex
    for(i=0; i<g->degree[v]; i++)
    {
        y = g->edges[v][i];
        if (discovered[y] == FALSE){
            parent[y] = v;
            dfs(v);
        } else{
            if (processed[y] == FALSE)
                process_edge(v,y); // do what you need to as
                    // you see each edge
        }
        if (finished) return;
    }
    processed[v] = TRUE;
}
```

Network Flows

```
#include <deque>
#include <iostream>
using namespace std;

#define MAXV 100 // max number of vertices
#define MAXDEGREE 50    // max out degree of a vertex

typedef struct {
    int v;
    int capacity;
    int flow;
    int residual;
} edge;

typedef struct {
    edge edges[MAXV+1][MAXDEGREE];  // storing edges
            // in adjacency lists
    int degree[MAXV+1]; // degree of each vertex
    int nvertices;
    int nedges;
} graph;

void init_graph(graph *g)
{
    int i;
    g->nvertices = 0;
    g->nedges = 0;
    for (i=1;i<=MAXV; i++) g->degree[i] = 0;
}

void insert_edge(graph *g, int x, int y, bool directed, int capacity)
{
    if (g->degree[x] > MAXDEGREE)
        printf("Error: bigger than max degree: %d %d",x,y);
    g->edges[x][g->degree[x]].v = y;
    g->edges[x][g->degree[x]].capacity = capacity;
    g->edges[x][g->degree[x]].flow = 0;
    g->edges[x][g->degree[x]].residual = capacity;
    g->degree[x]++;

    if (directed == false)
        insert_edge(g,y,x,true,capacity);
    else
        g->nedges++;
}

void print_graph(graph *g)
{
    int i,j;
    for(i=1;i<=g->nvertices;i++)
    {
        printf("%d:  ",i);
        for (j=0;j<g->degree[i];j++)
            printf(" %d (cp: %d, fl: %d, rs: %d)",
                    g->edges[i][j].v,
                    g->edges[i][j].capacity,
                    g->edges[i][j].flow,
                    g->edges[i][j].residual);
        printf("\n");
    }
}
bool processed[MAXV];
bool discovered[MAXV];
int parent[MAXV];

void init_search(graph *g){
    int i;
    for (i=1; i<=g->nvertices; i++) {
        processed[i] = discovered[i] = false;
        parent[i] = -1;
    }
}
```

```
bool valid_edge(edge e)
{
    if (e.residual > 0) return true; else return false;
}

void bfs(graph *g, int start){
    deque<int> q;
    int v;  // current vertex;
    int i;  // counter

    q.push_front(start);
    while (q.empty() == false){
        v = q.front();
        q.pop_front();
        processed[v] = true;
        for (i=0;i<g->degree[v];i++) {
            if ((discovered[g->edges[v][i].v] == false) &&
                    (valid_edge(g->edges[v][i]))){
                q.push_front(g->edges[v][i].v);
                discovered[g->edges[v][i].v] = true;
                parent[g->edges[v][i].v] = v;
            }
         }
    }
}
void find_path(int start, int end, int
parents[])
{
    if ((start == end) || (end = -1))
        printf("\n%d",start);
    else{
        find_path(start, parents[end], parents);
        printf(" %d",end);
    }
}
```

```
edge * find_edge(graph *g, int x, int y)
{
    int i;
    for (i = 0; i < g->degree[x]; i++)
        if (g->edges[x][i].v == y)
            return (&g->edges[x][i]);

    return NULL;
}
```

```
int path_volume(graph *g, int start, int end, int parents[])
{
    edge *e;

    if (parents[end] == -1) return 0;
    e = find_edge(g,parents[end],end);
    if (start == parents[end])
        return (e->residual);
    else
        return(min(path_volume(g,start,parents[end],parents),
                    e->residual));
}
void augment_path(graph *g, int start, int end, int parents[],
        int volume)
{
    edge *e;
    edge *find_edge();

    if (start == end) return;

    e = find_edge(g, parents[end], end);
    e->flow += volume;
    e->residual -= volume;
    e = find_edge(g,end,parents[end]);
    e->residual += volume;
    augment_path(g,start,parents[end],parents,volume);
}

void netflow(graph *g, int source, int sink)
{
    int volume, totalvolume;
    init_search(g);
    bfs(g,source);
    volume = path_volume(g, source, sink, parent);
    totalvolume = volume;
    while (volume > 0)
    {
        augment_path(g,source,sink,parent,volume);
```

```
int main()
{
    graph g;
    init_graph(&g);
    insert_edge(&g,1,2,false,30);
    insert_edge(&g,1,3,false,100);
    insert_edge(&g,1,4,false,25);
    insert_edge(&g,2,6,false,25);
    insert_edge(&g,3,6,false,30);
    insert_edge(&g,3,5,false,15);
    insert_edge(&g,4,5,false,20);
    insert_edge(&g,5,6,false,45);
    insert_edge(&g,6,7,false,50);
    g.nvertices = 7;
    netflow(&g, 1, 7);
    return 0;
}
```

```
        init_search(g);
        bfs(g,source);
        volume = path_volume(g, source, sink, parent);
        totalvolume += volume;
    }

    cout << "Total capacity from " << source << " to " << sink <<
            " is " << totalvolume << endl;
}
```

Strongly Connected Components

In dfs, compute finish time of each vertex by adding an int array f and a counter. After each vertex completes, increment the counter and store it at that vertices position in f.
Reverse all of the edges in the graph
DFS again, but visit the vertices in the order of their finish times. each component will be a strongly connected component

Topological Sorting

Construct an ordering of vertices so all edges go from left to right (edges represent precedence relations)
Remember:

- The strategy here is to find all of the vertices with no incoming edges – they can go at the left of the output. For each of those, delete all of its edges and you'll find more vertices with indegree of zero (who could then be output). If it is a directed, acyclic graph, every vertex will eventually have an indegree of zero and get processed
- this uses a queue to hold the vertices with no preceeding edges (indegree = 0). In a true topological sort, it doesn't matter which of these vertices we process next. In a particular problem, if the ordering matters, replace the queue with an appropriate structure (sorted vector?)

```
#include <deque>
using namespace std;
topsort(graph *g, int sorted[])
{
    int indegree[MAXV];
    deque<int> zeroin;
    int x,y,i,j;
    compute_indegrees(g,indegree);
    for (i=1;i<=g->nvertices;i++)
        if (indegree[i] == 0) zeroin.push_front(i);
    j=0;
    while (zeroin.empty() == FALSE){
        j++;
        x = zeroin.pop_front()
        sorted[j] = x;
        for (i=0;i<g->degree[x];i++){
            y = g->edges[x][i];
            indegree[y]--;
            if (indegree[y] == 0) zeroin.push_front(y);
        }
    }
    if (j != g->nvertices)
        printf ("Not a DAG\n");
}
```

```
compute_indegrees(graph *g, int in[])
{
    int i,j;
    for (i=1;i<=g->nvertices;i++)
        in[i] = 0;
    for (i=1;i<=g->nvertices;i++)
        for (j=0;j<degree[i];j++)
            in[g->edges[i][j]]++;
}
```

Traversal

```
#define NULLCHILD -1
int tree[7] = {0, 1, 2, 3, NULLCHILD, 5,
6};

void dfPreOrder(int n)
{
    if(tree[n] == NULLCHILD) return;

    printf("%d\n", tree[n]);  // Do us
    dfPreOrder(2*n+1);  // Do left child
    dfPreOrder(2*n+2);  // Do right child
}
```

```
void dfInOrder(int n)
{
    if(tree[n] == NULLCHILD) return;

    dfInOrder(2*n+1);  // Do left child
    printf("%d\n", tree[n]);  // Do us
    dfInOrder(2*n+2);  // Do right child
}

void dfPostOrder(int n)
{
    if(tree[n] == NULLCHILD) return;

    dfPostOrder(2*n+1);  // Do left child
    dfPostOrder(2*n+2);  // Do right child
    printf("%d\n", tree[n]);  // Do us
}
```

Breadth-first can be implemented by simply traversing the array linearly. Right to left traversals are implemented as above, just switch left and right child calls.

String-Based Math

Negative numbers are not supported, decimals are not supported, and division by zero is not checked for. The division used is integer division, therefore the decimal point and after are chopped off. The functions will work for arbitrarily large string integers, and will support any base 2 - 36. Leading zeros should be stripped off of all input.

Prototypes:

```
int Map(char);
char ReverseMap(int);
int Max(int, int);
int bigcmp(char*, char*);

char* Add(char*, char*, int);
char* Subtract(char*, char*, int);
char* Multiply(char*, char*, int);
char* Divide(char*, char*, int);
char* Modulus(char*, char*, int);
```

Utility Functions (used by the main ones):

Map a character to its associated value.

```
int Map(char input)
{
if ((input>='0')&&(input<='9'))
    return input - '0';

if ((input>='A')&&(input<='Z'))
    return input - 'A' + 10;
}
```

ReverseMap a value to its associated ASCII value.

```
char ReverseMap(int input)
{
if ((input<10))
    return '0' + input;
else
    return 'A' + input - 10;
}
```

Max finds the larger of two values.

```
int Max(int a, int b)
{
if (a>b)
    return a;
else
    return b;
}
```

BigCmp is like strcmp, except for char* numbers of the type used in these functions.

```
int bigcmp(char* op1, char* op2)
{
if (strlen(op1)>strlen(op2))
    return 1;
if (strlen(op2)>strlen(op1))
    return -1;
return strcmp(op1, op2);
}
```

Addition:

```
char* Add(char* op1, char* op2, int base)
{
unsigned int a;
int startfound = 0;
unsigned int c = 1;
int* sum;
char* response;
unsigned int size = Max(strlen(op1),strlen(op2)) + 2;
sum = new int[size];
response = new char[size];
for (a=0;a<size;a++)
    sum[a] = 0;
while (c<=size)
{
    if (c <= strlen(op1))
        sum[size - c] += Map(op1[strlen(op1) - c]);
    if (c <= strlen(op2))
        sum[size - c] += Map(op2[strlen(op2) - c]);
    sum[size - c - 1] += sum[size - c] / base;
    sum[size - c] %= base;
    c++;
}
c = 0;
for (a=0;a<size;a++)
{
    if (startfound==0)
        if ((sum[a] == 0)&&(a!=size-1))
            continue;
        else
            startfound=1;
    response[c] = ReverseMap(sum[a]);
    c++;
}
response[c] = '\0';
return response;
}
```

Division

```
char* Divide(char* op1, char* op2, int base)
{
unsigned int a;
int b;
int startfound = 0;
unsigned int c = 1;
char* response;
unsigned int size = strlen(op1);
response = new char[size];
response[0] = ReverseMap(base-1);
response[1] = '\0';
c = 1;
while (bigcmp(Multiply(response,op2,base),op1) < 0)
{
    response[c] = ReverseMap(base-1);
    response[c+1] = '\0';
    c++;
}
response[c] = '\0';
for (a=0;a<strlen(response);a++)
    response[a] = '0';
for(a=0;a<strlen(response);a++)
{
    for(b=0;b<base;b++)
    {
        response[a] = ReverseMap(b);
        if (bigcmp(Multiply(response,op2,base),op1) > 0)
            break;
    }
    b--;
    response[a] = ReverseMap(b);
}
return response;
}
```

Modulus (requires Divide, Multiply, Subtract):

```
char* Modulus(char* op1, char* op2, int base)
{
return Subtract(op1, Multiply(Divide(op1, op2, base), op2,base),base);
}
```

Multiplication

```
char* Multiply(char* op1, char* op2, int base)
{
unsigned int a;
int startfound = 0;
unsigned int c = 1;
int* sum;
char* response;
unsigned int size = strlen(op1) + strlen(op2) + 1;
sum = new int[size];
response = new char[size];
for (a=0;a<size;a++)
{
    sum[a] = 0;
}
while (c<=size)
{
    for (a=1;a<=c;a++)
    {
        if ((strlen(op2) >=  c - a + 1)&&(strlen(op1) >= a))
        {
            sum[size - c] += Map(op1[strlen(op1) - a]) *
    Map(op2[strlen(op2) - c + a - 1]);
            while (sum[size - c] >= base)
            {
                sum[size - c -1] += 1;
                sum[size - c] -= base;
            }
        }
    }
    c++;
}
c = 0;
```

```
for (a=0;a<size;a++)
{
    if (startfound==0)
        if ((sum[a] == 0)&&(a!=size-1))
            continue;
        else
            startfound=1;
    response[c] = ReverseMap(sum[a]);
    c++;
}
response[c] = '\0';
return response;
}
```

Subtraction:

```
char* Subtract(char* op1, char* op2, int base)
{
unsigned int a;
int startfound = 0;
unsigned int c = 1;
int* sum;
char* response;
unsigned int size = strlen(op1);
sum = new int[size];
response = new char[size];
for (a=0;a<size;a++)
{
    sum[a] = Map(op1[a]);
}
while (c<=size)
{
    if (c <= strlen(op2))
        sum[size - c] -= Map(op2[strlen(op2) - c]);
    while (sum[size-c]<0)
    {
        sum[size - c - 1] -= 1;
        sum[size - c] += base;
    }
    c++;
}
c = 0;
for (a=0;a<size;a++)
{
    if (startfound==0)
        if ((sum[a] == 0)&&(a!=size-1))
            continue;
        else
            startfound=1;
    response[c] = ReverseMap(sum[a]);
    c++;
}
response[c] = '\0';
return response;
}
```

Dynamic Programming

Edit Distance

```
-Set of rules for finding the difference between two strings.
-Substitution:  One character changes to another.
-Insertion:  Add a character.
-Deletion:  Remove a character.

#define MATCH 0
#define INSERT 1
#define DELETE 2

struct cell {
    int cost;      //Cost of reaching
    int parent;  //Parent Cell
};

cell table[maxlength + 1][maxlength + 1];

//Change S into T
//Pad S and T with a character and start with 1 to make initialization easier.

int compare(String s, String t)
{int i, j, k, opt[3];

for (i=0;i<maxlength;i++)
{   //Initialize the first row and column here.  Typically, across
    // the row, INSERT and down the column DELETE, and give an
    // appropriate COST to each}

for (i=1;i<s.size();i++)
    for (j=1;j<t.size();j++)  {
     opt[MATCH] = table[i-1][j-1].cost + match(s[i], t[j]);
     opt[INSERT] = table[i][j-1].cost + {cost of deleting t[j]};
     opt[DELETE] = table[i-1][j].cost + {cost of deleting s[i]};

     table[i][j].cost = opt[MATCH];
     table[i][j].parent = MATCH;

     for(k=INSERT;k<=DELETE;k++)
        if (opt[k] < table[i][j].cost) {
            table[i][j].cost = opt[k];
            table[i][j].parent = k;}
    //Identify the appropriate goal cell.  This is typical.
    return tables[s.size()-1][t.size()-1].cost;
}
```

Memoization

Take any algorithm that might repeat a certain calculation multiple times.
Create an STL map that takes as input the critical parameter and saves the value of the result, and check to see if the memoized value exists in the map before evaluating the function. If the parameter is an int, you can use an array.

Greedy Activity Selection

In this problem, you are given a list of jobs that are specified by starting and finishing times. You have to select the largest set of jobs whose times don't overlap (one job can start exactly when another is finishing). The strategy is to sort the jobs by finish time and then be greedy by picking the job with the earliest finish time that is compatible with the set you've already chosen

```
#include <iostream>
#include <string.h>
#include <vector>
using namespace std;

typedef pair<int,int> job;
vector<job> joblist;
vector<job> picked;

bool compare_finish(const job &p1, const job &p2)
{
    printf("comparing %d, %d\n",p1.second, p2.second);
    return(p1.second < p2.second);
}
void printjob(const job &p1)
{
```

```
    printf("(%d %d)\n",p1.first, p1.second);
}
int main()
{
    char s[80];
    while (gets(s))
    {
        char *p = strtok(s," ");
        int s = atoi(p);
        p = strtok(NULL,"\n");
        int f = atoi(p);
        joblist.push_back(make_pair(s,f));
    }
    sort(joblist.begin(), joblist.end(), compare_finish);
    picked.push_back(joblist[0]);
    for (int i=1;i<joblist.size();i++)
    {
        if (joblist[i].first >= picked.back().second)
        {
            picked.push_back(joblist[i]);
        }
    }
    printf("Selected\n");
    for_each(picked.begin(),picked.end(),printjob);
}
```

Maze Traversal

Simple

```
#include <iostream>
#include <bitset>
#include <vector>
using namespace std;
#define MAX_SIZE 8
typedef struct row_struct
{
    bitset<MAX_SIZE+2> d;
} row;
vector<row> maze;
int log(int x)
{
    int l = 0;
    while (x >= 1)
    {
        x = x/10;
        l++;
    }
    return l;
}

int main()
{
    int temp,n;
    maze.reserve(MAX_SIZE+2);
    cin >> n;
    // rows and columns are numbered from 1
    for (int i=1;i<=n;i++)
    {
        maze[i].d.set(0);
        maze[i].d.set(n+1);
        maze[0].d.set(i);
        maze[n+1].d.set(i);
        for (int j=1;j<=n;j++)
        {
            cin >> temp;
            maze[i].d[j] = temp;
        }
    }
    if (!search(1,1,n,n))
        cout << "No path found" << endl;
}
```

```
bool search(int sx, int sy, int fx, int fy)
{
    cout << "(" << sx << "," << sy << ")";
    if ((sx  == fx) && (sy == fy))
        return true;
    maze[sx].d.set(sy);
    if ((!maze[sx+1].d[sy]) &&
        search (sx+1, sy, fx, fy))
        return true;
    if ((!maze[sx-1].d[sy]) &&
        search (sx-1, sy, fx, fy))
        return true;
    if ((!maze[sx].d[sy+1]) &&
        search (sx, sy+1, fx, fy))
        return true;
    if ((!maze[sx].d[sy-1]) &&
        search (sx, sy-1, fx, fy))
        return true;

    for (int i=0;i<3+log(sx)+log(sy);i++)
        cout << (char)8;

    return false;

}
```

Min Spanning Trees and Shortest Path

The strategy for Prim's MST and Dijkstra's shortest path algorithms is exactly the same. The comments highlight the differences.

- Dijkstra's gives a shortest path tree, so giving shortest paths from starting vertex to ALL other vertices
- You can find a maximum spanning tree by negating the weights and then finding the min spanning tree
- If we want a min spanning tree with the smallest product of weights, we can use log(a*b) = log(a)+log(b) and replace each with with its logarithm and use the normal MST algorithm

```
void mst_sp(graph *g, int start){
    int i,j,v,w,weight,dist;
    bool intree[MAXV];
    int distance[MAXV];
    int parent[MAXV];
    for (i=1; i<=g->nvertices;i++){
        intree[i] = FALSE;
        distance[i] = MAXINT;
        parent[i] = -1;
    }
    distance[start] = 0;
    v = start;
    while (intree[v] == FALSE) {
        intree[v] = true;
        for (i=0;i<g->degree[v];i++){
            w = g->edges[v][i].v;
            weight = g->edges[v][i].weight;
            // use the following for MST
            if ((distance[w] > weight) && (intree[w] = FALSE)){
                distance[w] = weight;
                parent[w] = v;
            }
            // use the following for shortest path
            if (distance[w] >distance[v]+weight){
                distance[w] = distance[v] + weight;
                parent[w] = v;
            }
        }
        v = 1; // make v will be closest vertex that not in tree
        dist = MAXINT;
        for (i=2; i<=g->nvertices; i++){
            if ((intree[i] == FALSE) && (dist > distance[i])){
                dist = distance[i];
                v = i;
            }
        }
    }
}
```

```
#define MAXV 50
#define MAXDEGREE 40
#define FALSE 0
typedef struct{
    int v;  // neighboring vertex
    int weight;// edge weight
} edge;
typedef struct{
    edge edges[MAXV+1][MAXDEGREE];// adjacency list
    int degree[MAXV+1];       // outdegree
    int nvertices;
    int nedges;
} graph;
```

All Pairs Shortest Path

If you care about the path, the most efficient algorithm is dijkstra's from each vertex. This will give the distance, but won't let you recreate the path.

```
typedef struct {
    int weight[MAXV+1][MAXV+1];
    int nvertices;
} adjacency_matrix;
init_adj_matrix(adjency_matrix *g){
    int i,j;
    g->nvertices=0;
    for(i=1;i<MAXV;i++)
        for(j=1; j<=MAXV; j++)
            g->weight[i][j] = MAXINT;
}
```

```
floyd(adjacency_matrix *g)
{
    int i,j,k;
    int through_k;
    for (k=1;k<=g->nvertices;k++)
        for (i=1;i<=g->nvertices;i++)
            for (j=1;j<=g->nvertices;j++) {
                through_k = g->weight[i][k]+
                            g->weight[k][j]l
                if (through_k < g->weight[i][j])
                    g->weight[i][j];
            }
}
```

Grids

Rectilinear

Much like the Cartesian plane, they are relatively simple to traverse in array form. Note that a Hexagonal grid is practically identical to two rectilinear grids, slightly offset.

Triangular Lattice

This lattice is practically identical to a Hexagonal grid. From any point, there are 6 paths one might take.

Triangular Cell-Wise

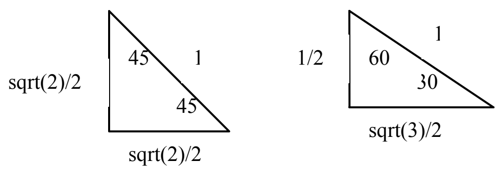
Similar to a Rectilinear grid, but do not allow passage between some cells.

Hexagonal

Choose one direction to be "x", and another to be "y". Moving in the other direction (the one that is a linear combination of the two) adds 1 to x and subtracts one from y or vice-versa.

Circle Packing

The densest possible placement of circles is with a placement analogous to the centers of hexagons in a hexagonal grid. To find the height and width of an array of packed circles, make use of the hexagonal grid and the properties of a 30-60-90 triangle. Both special triangles are below for reference.



Coding Tricks

STL algorithms

The examples are relative to vector<int> data; However, they can be applied to any of the containers.

adjacent_find: finedfirst element that has a value equal to its neighbor. Second form allows you to make the criteria of "equal" so you can make it anything. For instance, you could make a method "doubled" that return true if the first param has half of the second and find elements whose neighbors were doubles.

```
adjacent_find(data.begin(), data.end());
adjacent_find(data.begin(), data.end(), compareOp);
```

count: count the number of elements with a particular value

```
count(data.begin(), data.end(), value)
```

count_if: (operation is a boolean method with a single parameter that each element will be passed into)

```
countif(data.begin(), data.end(), operation)
```

equal: write your own compare operation to make this very powerful. For example, compareOp could check if the elements in the second list are doubles of the elements in the first list (in order).

```
equal(data.begin(), data.end(), data2.begin())
equal(data.begin(), data.end(), data2.begin(), compareOp)
```

find:

```
find(data.begin(), data.end(), value)
```

find_if: (operation is a boolean method with a single parameter that each element will be passed into)

```
find_if(data.begin(), data.end(), operation)
```

find_first_of: return position of first element of first range that is also in second range

```
find_first_of(data.begin(), data.end(),
           data2.begin(), data2.end());
find_first_of(data.begin(), data.end(),
           data2.begin(), data2.end(), compareOp);
```

for_each: (operation is a method to be applied to every member of the vector. can be applied to any InputInterators)

```
for_each(data.begin(), data.end(), operation)
```

min_element:

```
min_element(data.begin(), data.end())
min_element(data.begin(), data.end(), compareOperation)
```

max_element:

```
max_element(data.begin(), data.end())
max_element(data.begin(), data.end(), compareOperation)
```

next_permutation: permutes the elements to give the next permutation. returns FALSE if elements have lexicographic order (see also: prev_permutation)

```
next_permutation(data.begin(), data.end();
```

prev_permutation: permutes the elements to give the previous permutation. returns FALSE if elements have lexicographic order (see also: next_permutation)

```
prev_permutation(data.begin(), data.end();
```

remove:

```
remove(data.begin(), data.end(), remVal);
```

remove_if:

```
remove_if(data.begin(), data.end(), booleanOp);
```

replace:

```
replace(data.begin(), data.end(), old, new);
```

replace_copy:

```
replace_copy(data.begin(), data.end(), newdata.begin(),
old, new);
```

replace_copy_if:

```
replace(data.begin(), data.end(), newdata.begin(), booleanOp,
           old, new);
```

replace_if:

```
replace(data.begin(), data.end(), booleanOp, newValue);
```

search: return an iterator pointing at the first occurrence of one container in another.

```
vector<int>:: iterator pos;
pos = search(data.begin(), data.end(),
           data2.begin(), data2.end())
pos = search(data.begin(), data.end(),
           data2.begin(), data2.end(),compareOp)
```

search_n: returns position of the first of *size* consecutive elements in the range whose value match. Note that the criteria operation in the second for MUST be a binary operation.

```
search_n(data.begin(), data.end(), size, value)
search_n(data.begin(), data.end(), size, greater<int>())
```

unique: remove duplicate values (with "equal" defined by the boolean binary operator *compareOp*)

```
    unique(data.begin(), data.end());
unique(data.begin(), data.end(), compareOp);
```

C/C++ Tricks

ItoA: Need something like itoa(int x, string targ)? Use sprintf(targ,"%d",x);

STL – Reading Input

```
ifstream dataFile(“ints.dat”);
istream_iterator<int> dataBegin(datafile);
istream_iterator<int> dataEnd;
list<int> data(dataBegin,dataEnd);
```

Map Example

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
void main()
{
    map<string,int> freq;
    string word;

    while (cin >> word)
    {
        freq[word]++;
    }

    for (map<string,int>::iterator iter = freq.begin(); iter != freq.end(); iter++)
    {
        cout << iter->second << " " << iter->first << endl;
    }
}
```

Java Framework

- Be careful to name to class (and file) in the way the problem specifies (This one is Main)

```
import java.io.*;
import java.util.*;
public class Main
{
    static String readLn (int maxLg)
    {
        byte lin[] = new byte [maxLg];
        int lg = 0, car = -1;
        String line = "";
        try
        {
            while (lg < maxLg)
            {
                car = System.in.read();
                if ((car < 0) || (car == '\n')) break;
                lin [lg++] += car;
            }
        }
        catch (IOException e)
        {
            return (null);
        }
        if ((car < 0) && (lg == 0)) return (null);  // eof
        return (new String (lin, 0, lg));
    }
    public static void main (String args[])  // entry point from OS
    {
        String input;
        StringTokenizer idata;
        int n,curr;

        while ((input = readLn (255)) != null)
        {
            idata = new StringTokenizer (input);
            n = idata.countTokens();
            curr = Integer.parseInt (idata.nextToken());
        }
    }
}
```