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;
```

```
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;


}

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++)
    {
```

```
return false;
         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;
 }
```

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

```
#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;
```

```
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;
            }
        }
    }
}
```

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

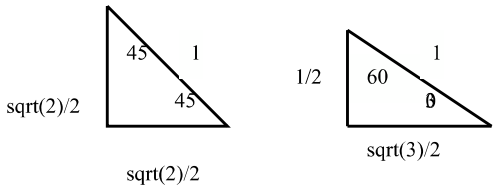
Similar to a Rectilinear grid, but do not allow passage between some cells.

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());
        }
    }
}
```