SLU's Programming Contest Cribsheet (C++ Edition)

©Michael H. Goldwasser, Saint Louis University, Fall 2011

Contents

1 Getting Started

1.1 Sample Program (use as template for others)

```
/**
   This program is one that computes the sum of given numbers for
   multiple trials. Input format is presume to be one where first
   number on a line N, with N <= 25, dictactes how many numbers will
   follow in this data set. Program should end when N=0. For example,
   input file might be:

   3 5 11 2
   1 8
   2 5 5
   0

   And output is to label trials starting with one, formated as

   Total 1: 18
   Total 2: 8
   Total 3: 10

 */

#include <iostream>
#include <fstream>
using namespace std;

// We love globals
ifstream fin("demo.in");              // assuming problem named "demo"
int data[25];                         // maximum size is 25

int main() {
    int trial(1);                     // start with number 1
    while (true) {
        int n;
        fin >> n;
        if (n==0) break;              // all done

        for (int k=0; k<n; k++)       // read next n integers into the global array
            fin >> data[k];           // (no other initialization needed in this example)

        int sum(0);                   // compute result
        for (int k=0; k<n; k++)
            sum += data[k];

        cout << "Total " << trial++ << ": " << sum << endl;  // POST-INCREMENT trial counter
    }
}
```

1.2 Commonly used libraries

```
// always need these
#include <iostream>      // cout, endl, (typically string as well)
#include <fstream>       // file input
#include <string>        // just in case not already included

// standard STL containers/algorithms
#include <algorithm>     // max, min, make_pair, and much more...
#include <vector>        // vector
#include <list>          // list
#include <map>           // map, multimap
#include <set>           // set, multiset
#include <queue>         // for queue data structure
#include <dqueue>        // for double-ended queue

// some other useful libraries
#include <sstream>       // stringstream for converting data to/from strings, or buffering output
#include <iomanip>       // for controlling precision, width, etc.
#include <cmath>         // higher-level mathematics functions
#include <limits>        // to get numeric_limit values

using namespace std;     // avoid fully-qualified std::cout and so on
```

1.3 Globals are your friends

When possible, we strongly recommend the use of global arrays for storing large amounts of data. Typically, a maximum problem size is specified in the problem statement, and so an array can be declared statically based upon the maximum possible size that you will ever need. You can always use a smaller portion of it for some cases (see, for example, the treatment of `data` in Section 1.1).

The advantage is that all functions have access to the data, and the memory management is much easier for the system as you do not need to repeatedly allocate and de-allocate large containers when performing multiple tests. In fact, for really large arrays, there may be a stricter limit on what can be allocated dynamically at run-time versus such a static declaration of a global at compile-time. Other global variables could be used to store additional problem parameters, to avoid the burden of having to pass so many values to subroutines.

Warning: When reusing globals from trial to trial, you must properly re-initialize.

2 Parsing Input

By the conventions of our region, input is always read from a file, with the name of the file depending on the official name of the problem. For example, a problem named "demo" will have its input in the file `demo.cpp`. Since the filename is known and assumed t be valid, the easier way is to declare an input file stream object globally, with the filename hardcoded. For example, our initial program from Section 1.1 declared

```
ifstream fin("demo.in");                // assuming problem named "demo"
```

2.1 Commin input formats

2.1.1 Unknown number of trials, terminated by sentinel

The sample code from Section 1.1 solves a hypothetical problem for our region, defined as follows. Each line of input will start with an integer $n \leq 25$, followed by n other integers. The goal is to output line of form "Total [case#]: sum" for each input line. End of input designated with a line containing 0. Sample input and output might appear as follows:

Sample Input	Sample Output
`3 5 11 2`	`Total 1: 18`
`1 8`	`Total 2: 8`
`2 5 5`	`Total 3: 10`
`0`	

For simplicitly, we prefer a style in which use a **while(true)** structure with an explicit break statement after we read the closing sentinel, as with

```
    while (true) {
        int n;
        fin >> n;
        if (n==0) break;                // all done
```

2.1.2 Known number of trials

Another commonly seen input format in our region is one for which the first line indicates the number of trials. If using that convention, our sample problem would have the following format:

Sample Input	Sample Output
`3`	`Total 1: 18`
`3 5 11 2`	`Total 2: 8`
`1 8`	`Total 3: 10`
`2 5 5`	

This could be parsed as follows:

```
    int numTrials;
    fin >> numTrials;
    for (int t=0; t < numTrials; t++) {    // zero−indexed in this example
        int n;
        fin >> n;

        for (int k=0; k<n; k++)        // read data into the global container
            fin >> data[k];            // (no other initialization needed in this example)
```

2.1.3 Reading a line at a time

Instead of numeric data, there might be times where the input consists of strings, one per line, with a sentinel. For example, we might have a list of words, with the string # as a sentinel to end the data set. That could be parsed quite similarly to the example from Section 2.1.1:

```
    while (true) {
        string word;
        fin >> word;
        if (word == "#") break;   // all done
```

However, this technique assumes that there is a single "token" on each line (i.e., there is no whitespace within the line).

Occassionally, there will be a problem in which there is need to read a complete line, even when that line possibly contains whitespace. In this case, the >> operator does not suffice. Instead, it is easiest to rely on the getline method, using a syntax as:

```
string s;
getline(fin, s);
```

However, **be very carefuly if you mix use of getline and >> within the same program.** A call to getline clears the trailing newline from the stream, but after use of >>, the subsequent whitespace remains on the stream. This means that if you first use >> and then immediately call getline, you will only get the remainder of the line that had the previous token (even if the rest of that line is blank). As an example, assuming that a first line has integer N describing the number of subsequent lines, such as

Sample Input
2
This is a test
This is still a test

That type of problem could be parsed as

```
    int numLines;
    fin >> numLines;
    string example;
    getline(fin, example);          // reads ``empty string'' and clears newline after the ``2''
    for (int k=0; k < lines; k++) {
        getline(fin, example);      // reads the real line and purges the ending newline
```

3 Generating Output

For our region, **all official output is to be printed to the standard output stream** (cout in C++). Programs are judged by doing a verbatim "diff" between the standard output and the expected result. You must produce the correct answer character-for-character. Typical "Presentation Errors" including incorrect spacing, spurious spacing at end of line, or spurious newlines, incorrect capitalization, spelling, or punctuation. While it is easy to get caught in the moment focused on the correct computations, please proofread your formatted output before submitting.

3.1 Outputting space-separated sequence

An absolute rule for our region is that the **formatted output never include any trailing whitespace at the end of a line**. Of course, inadvertent trailing space in your own program is easy to overlook as it is not visible on the console. One common output format involves a sequence of values that have been computed and a requirement that they be printed out on one line, with separating spaces (but never a trailing space). If the sequence is known to be nonempty, one approach is to print the first one by itself, then all subsequent ones with *leading* space as follows:

```
cout << data[0];                          // Print first value (assuming nonempty sequence)
for (int k=1; k < data.size(); k++)       // Note: loop starts at 1
    cout << '␣' << data[k];               // preceding space for all others
cout << endl;                             // final newline
```

Printing sequence after some preface token

This task is easier for problems that expect some leading token on the output line, such as `"Values:"` with a separating space. In this case, you can print that token *without* the space, and use a standard loop to print all values with *leading* space before each.

```
// print first. Then loop through rest printing: ' ' + value
cout << "Values:";                        // print token, without trailing space
for (int k=0; k < data.size(); k++)       // Note: loop starts at 0
    cout << '␣' << data[k];               // preceding space for all others
cout << endl;                             // final newline
```

Note well that this code even works for an empty sequence, as there will be the token but without any spurious spacing.

Breaking sequences across multiple lines

Another format that is sometimes used involves breaking a very long sequence into lines so that there are at most a fixed number per line (e.g., 10). This can be done as follows:

```
cout << data[0];
for (int k=1; k < data.size(); k++) {
    if (k % PERLINE)                      // for example, perhaps PERLINE = 10
        cout << '␣';                      // continue same line with leading space
    else
        cout << endl;                     // start newline before next data value
    cout << data[k];
}
cout << endl;                             // final newline to end last line (partial or full)
```

3.2 Controlling precision for floating-point numbers

When contest problems involve floating-point calculations, they typically ask you to print the result to some fixed precision, typically as number of digits after the floating point. This can be controlled using manipulators from the <iomanip> library. For example, to round to exactly three digits following the decimal point, use the following:

```
cout << fixed << setprecision(3) << floatval << endl;
```

For example, the float 3.1 is printed as `3.100`, and 3.14159 is printed as `3.142`. Note well that it is properly rounded (not truncated). Also note that fixed and setprecision manipulators affect all subsequent values on the output stream, not just the immediate command.

3.3 Converting a number to a string (or vice versa)

Although it is easy to output a number directly to a stream, C++ does not make it easy to create a string representation of a number (for example because you are going to check its width, or use it to compose some other string). The easiest way to do this is to use a string stream. Here is templated code that allows you to turn any data type to a string, so long as that data type supports the output stream operator <<.

```
#include <sstream>                    // library <sstream> includes stringstream class
template <typename T>
string toString(const T& val) {       // works for any type that supports << operator
    stringstream ss;
    ss << val;                        // insert value onto stringstream
    return ss.str();                  // recover internal string buffer.
}
```

It is less common that you have string data that needs to be converted to a number, because if that data was known to be a number in the original input, you could typically read it in directly as a number using the stream operators. But if you find yourself with string data that needs to be converted, here is an example that coverts a string to a double.

```
double fromString(const string& orig) {    // could be defined for some other type
    double result;
    stringstream ss;
    ss << orig;                            // insert string data
    ss >> result;                          // extract (formatted) data
    return result;
}
```

4 Debugging Practices

4.1 Debug to Standard Err

When judging programs, our region specifically captures output sent to the standard output stream, but guarantees that it ignores any incidental output to the standard *error* stream. Therefore, we should get into the habit of sending all debug output to standard error (cerr in C++).

As noted, you are free to leave all the cerr chatter in your program when submitting it. That said, if you would like to turn it off for your own sake, you can cleanly disable all cerr output with the following additional line, placed as the first command within main()

```
int main() {
    cerr.rdbuf( (new stringstream)->rdbuf() );      // divert all output sent to cerr
```

Please be aware that the debugging is still performed, even though not displayed. So if you are concerned that your debugging code might impose on the time limit, you should comment it out by other means for efficiency.

4.2 Debugging with "visible" spaces

An absolute rule for our region is that the **formatted output never include any trailing whitespace at the end of a line**. Of course, inadvertent trailing space in your own program is easy to overlook as it is not visible on the console. If you wish to have a more automated approach, the following code could be added as a preface to each program (at least you'd only have to type it once).

```
 1  class Safe : public stringstream {
 2  public:
 3      streambuf* temp;
 4      Safe() : temp(cout.rdbuf()) { cout.rdbuf(rdbuf()); }
 5
 6      ~Safe() {
 7          cout.rdbuf(temp);           // switches back to restore cout behavior
 8          string result = str();
 9
10          if (result.find("␣\n") != string::npos)
11              cerr << "WARNING:␣trailing␣whitespace␣detected." << endl;
12
13          cerr << "--------␣stdout␣follows␣--------" << endl;
14          cout << result;             // Actual results to standard out
15
16          cerr << "----␣stdout␣echo␣with␣visible␣spaces␣----" << endl;
17          for (int k=0; k < result.size(); k++)
18              if (result[k] == '␣')    result[k] = '_';         // replace space with underscore
19          cerr << result;
20      }
21  } hidden;    // 'hidden' instance activates safeguarding (omit variable 'hidden' to disable entirely)
```

Defining this class and delcaring the `hidden` instance causes `cout` to be effectively hijacked for the entire rest of the program. All data sent to `cout` will be temporarily buffered rather than displayed. When the program ends and the destructor for the safeguard instance is automatically invoked, the buffered text is finally sent to standard out for judging. But this also gives us the opportunity to do an examination and post-processing. Lines 10-11 specifically warn of unintended spaces that immediately precede a newline, line 14 echos the unedited result to standard out, and lines 16–19 echo a copy of all output to `cerr`, but with all spaces replaced by undersscore characters.

4.3 Printing containers

Here is generic code for printing contents of an STL container, invoked using a syntax such as `dump(v.begin(), v.end())` for a vector or list, or `dump(a, a+n)` for an array.

```
template <typename T>
void dump(T begin, T end) {
    cerr << "[";
    for (T walk=begin; walk != end; ++walk) {
        if (walk != begin)
            cerr << ",␣";
        cerr << *walk;
    }
    cerr << "]" << endl;
}
```

5 Standard Algorithms

5.1 Sorting

For built-in types, you may get standard sorting algorithms from the STL libraries. The <algorithm> library has a general method sort that takes begin and end iterators to delimit the range of interest for any container supporting RandomAccessIterators. For an array, it can be invoked as

```
sort(data, data+n);                          // sorts n-element array data[n]
```

For a vector (or even a string), the calling syntax would be

```
sort(data.begin(), data.end());              // sorts vector or string
```

If it is every important to guarantee *stable* sorting, the algorithm library function stable_sort. Niether sort nor stable_sort work with the list class, as that does not support random-access iterators. Instead, that class has a dedicated method named sort, invoked as data.sort().

By default, all of the library sorting routines depend upon elements having a well-defined < operator. If you are sorting objects from a newly defined class or struct, you are responsible for overriding **operator**< in a way that properly defines a *total ordering*. Here is an example of such a Point class, which defines a total ordering based lexicographically on (x,y); that is smallest to largest x-coordinates, and smallest-to-largest y-coordinate as tie-breaker.

```
struct Point {
    int x,y;            // can be initialized using syntax: Point p = {3,5}

    bool operator<(const Point& q) const {          // left-to-right ordering for syntax: p < q
        return (x < q.x || (x == q.x && y < q.y));
    }
};
```

Alternatively, you can control the sorting order by providing an external comparison function (or function-like object) as an optional parameter to the sorting function. This can be particularly helpful for classes that you did not author, or when you want a non-standard definition for sorting. Here is an example of the latter style, providing the same lexicographical order.

```
struct Point {
    int x,y;            // can be initialized using syntax: Point p = {3,5}
};

// independent function defining left-to-right ordering for syntax: p < q
bool lexCompare(const Point& p, const Point& q) {
    return (p.x < q.x || (p.x == q.x && p.y < q.y));
}

int main() {
    vector<Point> myV;                              // presumably will be filled with data
    sort(myV.begin(), myV.end(), lexCompare);       // syntax for vectors
    list<Point> myL;                                // presumably will be filled with data
    myL.sort(lexCompare);                           // syntax for lists
}
```

5.2 Binary Search

The <algorithm> library has very convenient implementations of binary search, assuming you are working with a *sorted* container that has random-access iterators (most notably, arrays or vectors).

Function: binary_search

This function returns **true** if it finds an indicated value in the container, and **false** otherwise. As with sort, there are two signatures, the second using a comparator to override the natural ordering.

```
bool binary_search(Iterator start, Iterator stop, T value);
bool binary_search(Iterator start, Iterator stop, T value, Compare comp);
```

Typical calling syntax might be:

```
bool found = binary_search(data, data+n);                  // for array of length n
bool found = binary_search(data.begin(), data.end());      // for vector of length n
```

Function: lower_bound

Whereas binary_search returns a true/false, lower_bound can be used to find the *location* at which a value appears, or to find a value that is just greater than a target value, if the target cannot be find. It returns the location of **the first element that is ≥ the target value (if any)**. The function will return the "end" iterator of the range to designate that no such element exists. There are two signatures, again depending on whether to use the natural element ordering or a comparator.

```
Iterator lower_bound(Iterator start, Iterator stop, T value);
Iterator lower_bound(Iterator start, Iterator stop, T value, Compare comp);
```

Typical calling syntax might be:

```
foo& answer = lower_bound(data, data+n);                              // for array of length n
vector<foo>::iterator it = lower_bound(data.begin(), data.end());     // for vector of length n
```

Function: upper_bound

The difference between lower_bound and upper_bound is not the direction of the inequality, but whether or not it is a *strict* inequality. The upper bound function uses a strict inequality, identifying the location of **the first element that is > the target value (if any)**. So the only time that the results of this and lower bound differ is when the target value exists: lower_bound will return the location of that element, while upper_bound returns the location of the next bigger element.

```
Iterator upper_bound(Iterator start, Iterator stop, T value);
Iterator upper_bound(Iterator start, Iterator stop, T value, Compare comp);
```

5.3 Priority Queues

If you need a priority queue, there is an adapter class in the <priority_queue> library. It supports methods by the name of empty, size, push, top, pop. By default, the priority queue relies on **operator**< for ordering, and is oriented so that the "top" of the heap is the element with *maximum* value. A different ordering can be defined using a comparison function that is sent as a parameter to the constructor.

5.4 Other convenient algorithms

Some other goodies from <algorithm> library working with containers:

- Iterator reverse(Iterator start, Iterator stop)
  Reverse order of elments in range $[start, stop)$.
- Iterator min_element(Iterator start, Iterator stop)
  Returns iterator to position of minimum value in $[start, stop)$.
- Iterator max_element(Iterator start, Iterator stop)
  Returns iterator to position of maximum value in $[start, stop)$.
- **int** count(Iterator start, Iterator stop, T value)
  Returns number of occurrences of value within range $[start, stop)$.
- Iterator find(Iterator start, Iterator stop, T value)
  Returns iterator to first position of $[start, stop)$ at which value is found (else stop).
- **void** replace(Iterator start, Iterator stop, T oldVal, T newVal)
  Replaces *all* occurrences of oldVal with newVal within $[start, stop)$.
- Iterator unique(Iterator start, Iterator stop)
  Removes all *consecutive* duplicates in the given range, and returns an iterator to the *new* stop of possibly shorter range (rest of original container may have garbage).

Recall that for arrays, the range is described with parameters such as (data, data+n), while most STL containers use syntax (data.begin(), data.end()).

6 String Manipulations

6.1 Built-in Functions

A few quick reminders about some of the commonly used behaviors for the string class, keeping in mind that strings are *mutable*.

- **constructor**
  Default string is empty. Can make a string of repeated characters, such as string(5,'-') to construct "-----".

  For creating strings from other forms of primitive data, please see discussion of stringstreams in Section 3.3.
- **substr method**
  Form is that first parameter is starting index, and second parameter (if given) is *number of characters*. By default, it will go to the end of the string if second parameter is omitted. For example, given string s = "This␣is␣a␣test", the return value of s.substr(5,4) will be "is␣a",and the result of s.substr(10) will be "test". *Note that s is unchanged.*
- **find method, rfind method**
  Form s.find(pattern) returns the index at which the first occurrence of pattern starts in the original string (or returns string::npos if not found).

  Form s.find(pattern, k) begins the search starting at s[k] rather than s[0].

The rfind method has similar signatures, but starts searching from the end of the string. However, note that the return value of rfind is still the *starting* index of the pattern which was identified.

- **find_first_of method, find_last_of method**
  Whereas a call like s.find("the") looks for the pattern "the" occurring as a substring, the call s.find_first_of("the") looks for the first single character that is either 't', 'h', or 'e'. As with find, a non-initial starting index for the search can be given as s.find_first_of("the", 5). The find_last_of does the search from end toward beginning.

- **+= operator**
  Used to append one or more characters to the end of a string. The operand can either be a single character or a string, e.g., s += '.' or s += t

- **insert method**
  Used to insert one copy of a string at an internal position within another string. As an example, assume that s = "test" and t = "asti". Then after call to s.insert(1, t), The string s has value "tastiest". Can also use a character sequence as a literal, as in s.insert(1, "asti").

  For single characters, the calling syntax for insert allows (and requires) that you specified the number of copies of the given character that you with to insert. For example, if we go back to s = "test", we can give command s.insert(2, 5, '-') to change s to value "te-----st".

- **erase method**
  s.erase(k,n) will erase up to n characters, starting with index k. If n is not designated, then it erases everything from index k onward.

- **replace method**
  A single operation that effectively performs an erase followed by an insert at that same position. Read documenation carefully, as there are several signatures that allow you to describe the length and location of the erasure and the replacement.

6.2 Prefix Search

If you have a large collections of strings and have them *sorted* in an array or vector, you can make use of the general lower_bound and upper_bound functions, described in Section 5.2, to efficiently determine all strings that start with a given prefix.

In particular, if you call lower_bound(data.begin(), data.end(), prefix), it will return to you an iterator of the first string that is lexicographically greater than or equal to prefix. If there is any string starting with those characters, it will be the one identified by the return value (conversely, if the returned location does not start with this prefix, no strings do).

If you want to find *all* such strings, you can start at the first one and keep advancing until reaching one that does not have the prefix (or the end of the collection).

If you want to count *how many* strings have that prefix, but do not want to take the time to iterate through all of them, you can likely get by with the following. Call lower_bound to get the location of the first. Then call upper_bound using the target value of prefix + (**char**) 255 (we assume that 255 as a byte is never an actual character of the collection). That will result in the first subsequent position that does *not* use the given prefix. Then, the number of matching strings can be determined by taking the difference between the upper and lower iterators.

6.3 Trie Structure

The techniques from Section 6.2 will most likely be sufficient for most programming contest problems that involve prefixes of strings, because usually, the full set of strings is dictated in advanced and thus can be stored in an array and sorted. But the more general approach is to use a data structure known as a *trie*, which implicitly represents a set of strings as a trie in which the root represents the empty string, and each substring node has children corresponding to the possible continuations of that substring that occur in the data set. An implementation of a basic trie is given in Figure 1 on page 15, using the notion of a public Trie::Position to navigate the internal structure of the trie.

7 Bitwise Manipulations

One of the most common uses of bitwise operators for programming contests is to use the individual bits of an integer type to efficiently represent a sequence of boolean values. The STL `bitset` class has such functionality for any number of bits, but in our contest, this is almost always done for a case when the maximum number of bits is at most 64, therefore allowing us to use a **long long** which is minimally a 64-bit integer on all systems.

If particular, if you need to represent n bits for $n \leq 64$, we will do this by using the least-significant n bits of a long integer, numbering those bits starting with bit 0 as the least significant. The basic operations that we need hinge on use of the expression (1ULL << k) which represents 2^k (which is a 1 followed by k zeros); we use literal 1ULL which is the unsigned long long representation of 1 to ensure that all intermediate calculations are performed properly. From there, we use a combination of masks to set, clear, or test individual bits. We suggest the following combination of functions. Note: you are free to use **int** rather than **unsigned long long** if you only need 31 bits; in that case use 1 rather than 1ULL for the literal).

```
// returns true if k'th bit is set (where k=0 is least significant)
bool testBit(unsigned long long code, int k) {
    return (code & 1ULL << k) > 0;              // equivalently (code & (1ULL << k)) > 0
}
```

```
// set the k'th bit of code (modifies parameter)
void setBit(unsigned long long &code, int k) {       // remember the ampersand!
    code |= 1ULL << k;
}
```

```
// clear the k'th bit of code (modifies parameter)
void clearBit(unsigned long long &code, int k) {     // remember the ampersand!
    code &= ~(1ULL << k);                            // Note well the tilde
}
```

8 Brute-Force Enumerations (and Pruning)

There are many problems that, in one form or another, require that you consider all possible configurations for some data set (e.g., all permutations, all subsets, all subsets of size k), or similar combinatorics for some set of decisions.

```
class Trie {
private:
    struct Node {
        bool isWord;                    // does a word end at this node?
        int freq;                       // how many words pass through this node?
        map<char, Node> children;       // must #include<map> for definition
        Node() : isWord(false), freq(0), children() {}
    };
    Node rt;
public:
    class Position {
    private:
        const Node* nd;
    public:
        Position(const Node& nd) : nd(&nd) {}
        bool isWord() const { return nd->isWord; }
        int freq() const { return nd->freq; }
        bool hasChild(char c) const { return nd->children.find(c) != nd->children.end(); }
        // UNSAFE to call child, unless hasChild(c) known to be true
        Position child(char c) const { return Position(nd->children.find(c)->second); }
        bool operator==(Position p) const { return nd == p.nd; }
        bool operator!=(Position p) const { return nd != p.nd; }
        bool operator<(Position p) const { return &nd < &p.nd; }  // allows Positions as key for set/map
        long long id() const { return (long long) nd; }              // can be used for debugging
        string children() const {
            string result = "";
            const map<char,Node>& c(nd->children);
            for (map<char,Node>::const_iterator it = c.begin(); it != c.end(); ++it)
                result += it->first;
            return result;
        }
        friend class Trie;
    };

    Position root() const {return Position(rt);}

    void add(const string& word) {
        Node* n = &rt;
        for (int j=0; j < word.size(); j++) {
            n->freq++;
            n = &n->children[word[j]];
        }
        n->isWord = true;  // final node of the walk
    }

    bool has(const string& word) {
        Position p = root();
        for (int j=0; j < word.size(); j++) {
            if (!p.hasChild(word[j]))
                return false;
            p = p.child(word[j]);
        }
        return p.isWord();
    }
};
```

Figure 1: Trie implementation

For some of those problems, added efficiency is needed because it is too expensive to truly enuerate through all possibilities. In many cases, it is possible to prune many partial solutions. For example, the classic n-queens problem formally considers all possible choice of n locations for the queens. However, if the second placement of the queen conflicts with the first, there is no reason to continue finding all extentions of that partial solution.

To solve this style of problem, we favor a recursive approach for "filling in" such a configuration piecewise, with one callback function that can be used to check the consistency of a partial configuration before proceeding, and another callback that is used to report complete configurations.

8.1 Generic approach

For convenience, we assume that the "solution" to be build is represented using global variables of the apporiate type, so that they do not need to be passed as parameters. We also assume that there is some canonical "order" for filling in pieces of the solutions, and that it is apparant what choices are allowed for filling in the next piece. Given that, we use the following approach:

```
def build():
    if sufficient:                          // is current state sufficiently "full"?
        process                             // analyze the current state as potential solution
    for each next step s:                   // possibly none
        Enact change to configuration for s
        if consistent:                      // is "new" state consistent?
            build() recursively
        Undo change associated with s
```

8.2 Generating all possible combinations ("n choose k")

Typical goal is to enumerate all possible combinations, say of k elements chosen from a set of n elements (with $k \leq n$). The standard combinatorics expression for the number of such subsets is $\binom{n}{k}$, pronounced "n choose k". We will focus on the special case where the data set are numbers from 0 to n-1. As an example, here is the lexicographical enumeration of all ten ways of selecting 3 out of 5 objects.

```
0 1 2
0 1 3
0 1 4
0 2 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4
```

If working with some other sequence of data values, it is possible to consider mutating that directly, but we will rely on an indirect approach based on the above permutations of the indices from $0, \ldots, n-1$. If the original values are stored in data and we create the sequence choice of the k indices, then the item with index j in the permuted data set would be data[choice[j]].

We give two implementations, depending on whether we want to grow and shrink a vector, or whether we use a pre-sized array or vector (given knowledge that there will eventually be k entries). Here is the one that begins with an empty vector:

```
vector<int> choice;        // global, and initially empty; mutated within recursion

// computes all possible combinations of k disinct values from {0, ..., n−1}
void combinations(int n, int k) {
    if (choice.size() == k)
        process();                     // the choices are complete. Do something.
    else {
        int next = (choice.empty() ? 0 : 1 + choice.back());
        for (int j = next; j < n; j++) {
            choice.push_back(j);       // add j before recursion
            if (validPartial())        // opportunity to prune,
                combinations(n, k);    //     otherwise recurse
            choice.pop_back();         // undo addition of j
        }
    }
}
```

This function should be started with an empty choice vector and an initial calling syntax such as combinations(5,3). In comparing this form to the "general" approach, note that when choosing the next element to append to the sequence of choices, we must pick something that is strictly greater than the previous entry (unless this is the first). Also, in this example, if the "sufficiency" condition is true, namely that the length of the vector is k, then there is no need to consider additional steps, so the remainder of the function is in an **else** clause.

Using a fixed size array or vector will be slightly more efficient. The only complication is that we cannot rely on choices.size() to determine the number of completed steps thus far. So we instead add a thusFar parameter to the signatures, and invoke the original recursion as combinations(n,k,0).

```
int choice[MAX];        // global, so that it can be mutated within recursion

// computes all possible combinations of k disinct values from {0, ..., n−1}
void combinations(int n, int k, int thusFar) {
    if (thusFar == k)
        process(thusFar);                          // the choices are complete. Do something.
    else {
        int next = (thusFar == 0 ? 0 : 1 + choice[thusFar−1]);
        for (int j = next; j < n; j++) {
            choice[thusFar] = j;                   // add j before recursion
            if (validPartial(thusFar))             // opportunity to prune,
                combinations(n, k, 1 + thusFar);   //     otherwise recurse
                                                   // undo not technically necessary
        }
    }
}
```

8.3 Generating all possible base B values

As an example, we might want to compute all possible d-digit values in base B. For example, here are all two-digit base 3 numbers:

```
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

```
vector<int> digit;          // global, and initially empty; mutated within recursion

// computes all possible d−digit values in base B
void baseEnumeration(int d, int B) {
    if (digit.size() == d)
        process();                          // the digits are complete. Do something.
    else {
        for (int j = 0; j < B; j++) {
            digit.push_back(j);             // add j before recursion
            if (validPartial())             // opportunity to prune,
                baseEnumeration(d,B);       //     otherwise recurse
            digit.pop_back();               // undo addition of j
        }
    }
}
#else
```

Although we will not do so here, it is also possible to implement this approach with a fixed size array or vector, in which case a thusFar parameter will need to be used, as shown in the second example of Section 8.2.

Note: if pruning is not needed, an easier way to get a base B counter is given in Section 10.2.1.

8.4 Generating all possible subsets

The easiest way to generate all possible subsets of n items is by considering all possible n-bit patterns while counting from 0 to $2^n - 1$ using bit manipulations as discussed in Section 7. Elements whose index are 1 can be considered part of the subset.

If pruning is possible, we could generate all such n-bit patterns by using the code from Section 8.3 with $B = 2$. For variety, we show a different approach in this section. Rather than building a fixed length set of bits, we can instead generate a variable length vector containing those indices that should be considered in the set. Using this approach, an enumeration of subsets of the set $\{0, 1, 2\}$ might be genereated as follows.

```
{ }
{ 0 }
{ 1 }
{ 0, 1 }
{ 2 }
{ 0, 2 }
{ 1, 2 }
{ 0, 1, 2 }
```

Our code for generating these subsets in lexicographical order is as follows:

```
vector<int> group;        // global, and initially empty; mutated within recursion

// computes all possible subsets of {0, ..., n−1} in lexicographical order
void subsets(int n) {
    process();                    // everything is a subset

    // consider adding additional numbers
    int next = (group.empty() ? 0 : 1 + group.back());
    for (int j = next; j < n; j++) {
        group.push_back(j);
        if (validPartial())
            subsets(n);
        group.pop_back();
    }
}
```

This is quite similar to our code for generating all combinations, in that it keeps trying to add one additional entry to the end of the group that is strictly bigger than the previous. However, there are two key differences. First, we report every (partial) solution as a subset, thus the call to process() is not conditional. Secondly, we do not stop the process based on the size of the group. We continue to enumerate until we find no ways to extend the current subset with a lexicographically larger number from $\{0, \ldots, n-1\}$.

8.5 Generating all possible permutations

Typical goal is to enumerate all possible permutations of a sequence of values. For example, the numbers 0,1,2 could result in the following lexicographical enumeration of 6 permutations.

```
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

We can produce lexicographically ordered permutations in much the same way that we produced all combinations, albeit with linear auxillary space at each level of the recursion, thus quadratic overall. This is because we create a boolean vector to track which numbers have already been used, picking our next value, in order, from those that remain.

```
vector<int> perm;       // global, and initially empty; mutated within recursion

// computes all possible permutations of {0, ..., n−1}
void permutations(int n) {
    if (perm.size() == n)
        process();                          // the perms are complete. Do something.
    else {
        vector<bool> used(n, false);
        for (int k=0; k < perm.size(); k++)
            used[perm[k]] = true;
        for (int j=0; j < n; j++)
            if (!used[j]) {
                perm.push_back(j);          // add j before recursion
                if (validPartial())         // opportunity to prune,
                    permutations(n);        //     otherwise recurse
                perm.pop_back();            // undo addition of j
            }
    }
}
```

There are easier approaches for enumerating all permutations that do not require the quadratic overhead, if we do not need them to be lexicographical, or if we do not need our same framework for pruning. Of particular note, the <algorithm> library has an in-place and stateless function that transforms any permutation of objects into the next lexicographical permutation. The routine simply requires bidirectional iterators, and thus it can be applied to an array, vector, string, or list. It has a return value that returns false if there is none (because the elements are in reverse order) and true otherwise. To get all permutations, you must start with the elements in sorted order. Here is an example to get all permutations of the numbers $\{0, \ldots, n-1\}$ using an array as the underlying data storage:

```
int perm[n];                                // hard−wired for n
for (int j=0; j<n; j++)   perm[j] = j;      // insert values 0, ..., n−1
sort(perm, perm+n);                         // useful when not known to be sorted

do {
    // process current permutation (but do not mutate it)
} while ( next_permutation(perm, perm+n) );
```

Here is similar code using the syntax of vectors:

```
vector<int> perm;
for (int j=0; j<n; j++)   perm.push_back(j);    // insert values 0,...,n−1
sort(perm.begin(), perm.end());                 // useful when not known to be sorted

do {
    // process current permutation (but do not mutate it)
} while ( next_permutation(perm.begin(), perm.end()) );
```

Cyclical Permutations

In some cases you are considering permutations of items in a cycle, and so without loss of generality you can assume that value 0 is at index 0. Cycling through all such canonical permutations can be accomplished as:

```
vector<int> perm;
for (int j=0; j<n; j++)    perm.push_back(j);     // insert values 0,...,n−1
sort(perm.begin(), perm.end());                   // useful when not known to be sorted

do {
    // process current permutation (but do not mutate it)
    next_permutation(perm.begin(), perm.end());
} while (perm[0] == 0);                           // Stop if the first element ceases being 0
```

8.6 Classic n-Queens example

A classic example of enumeration with pruning involves computing configurations of an $n \times n$ chessboard wih n queens, so that no two queens attack each other. Given that each row must have exactly one queen, and each column must as well, we can model each potential configuration as a permutation. We let perm[k] represent the row number for the unique queen that lies in column k. If considering a prune after each additional queen is places, we check the position of that newest queen against all previously placed queens. A concrete implementation counting the number of unique solutions for an $n \times n$ board is given in Figure 2 on page 22.

9 Puzzle Solving

Start configuration, goal configuration, question about reachability, or shortest number of *steps* (rather than a weighted cost).

9.1 Implicit Breadth-First Search

10 Mathematical Fundamentals

10.1 Geometry

For simplicity, we assume the following basic classes, although in some cases you may want to use integer coordinates for computations.

```
struct Point {        // could presumably be defined with integers
    double x,y;
    Point(double x=0, double y=0) : x(x), y(y) {}
};

struct Line {        // Represents standard Ax + By = C line in normalized form
  double A,B,C;
  Line(double a=0, double b=0, double c=0) : A(a), B(b), C(c) {
    double temp = sqrt(a*a + b*b);
    if (temp > 0) {
```

```
vector<int> perm;       // perm[k] represents the row number for the queen in column k
int total(0);

// Given that one number is added at a time, we check whether the most
// recent entry conflicts with any preceding ones
bool validPartial() {
    int k = perm.size() − 1;
    for (int j=0; j < k; j++) {
        int diff = abs(perm[k] − perm[j]);
        if (diff == (k−j))
            return false;
    }
    return true;
}

// Called once for each complete answer
void process() {
    total++;
}

// computes all feasible arrangments of queens
void queens(int n) {
    if (perm.size() == n)
        process();                          // the perms are complete. Do something.
    else {
        vector<bool> used(n, false);
        for (int k=0; k < perm.size(); k++)
            used[perm[k]] = true;
        for (int j=0; j < n; j++)
            if (!used[j]) {
                perm.push_back(j);          // add j before recursion
                if (validPartial())         // opportunity to prune,
                    queens(n);              //     otherwise recurse
                perm.pop_back();            // undo addition of j
            }
    }
}
```

Figure 2: Solution to counting N-Queens solutions

```
      a /= temp;
      b /= temp;
      c /= temp;
    }
  }
};

struct Circle {      // Represents circle of radius r centered at (x,y)
  double r;
  Point ctr;
  Circle(double r=0, double x=0, double y=0) : r(r), ctr(x,y) {}
};
```

10.1.1 Distance between two points

Here is the classic formula for computing the distance between two points:

```
#include <cmath>                         // for sqrt
double dist(Point a, Point b) {
    double dx = (a.x - b.x);
    double dy = (a.y - b.y);
    return ( sqrt(dx*dx + dy*dy) );
}
```

Note that this relies on floating-point precisions. In many cases, it is possible to compute relative distances while working with integral coordinates by using the distance squared rather than the true distance, as:

```
double distSquared(Point a, Point b) {
    double dx = (a.x - b.x);
    double dy = (a.y - b.y);
    return ( dx*dx + dy*dy );
}
```

10.1.2 Interpolating along a line

There have been many problems that in one way or another involve calculating a point somewhere on the line defined by points A and B, for example if simulating a walk, starting at A, moving d units toward (and possible beyond) B. I have seen many students tempted to start computing angles or slopes, or to write code with many cases depending upon whether A is left or right of B, and above or below B. *Do not use such complicated approaches.* The computations can be done directly by simply computing the vector representing the directed distance between the two points and then scaling that vector.

```
// produce a point that is the given length away from 'src' in the
// direction toward 'dest' (with negative distance meaning away from)
Point interpolate(Point src, Point dest, double length) {
    double dx = (dest.x - src.x);
    double dy = (dest.y - src.y);
    double scale = length / sqrt(dx*dx + dy*dy);
    Point result;
    result.x = src.x + scale*dx;
    result.y = src.y + scale*dy;
    return result;
}
```

10.1.3 Area of triangles and polygons

Computing signed area of a triangle:

```
// Returns the "signed" area of triangle.     If a, b, c are given in
// counterclockwise order, this will be positive; if in clockwise
// order, this will be negative.     If colinear, then presumably 0.
double signedArea(Point a, Point b, Point c) {
    return ((b.x − a.x) * (c.y − a.y) − (c.x − a.x) * (b.y − a.y)) / 2.0;
}
```

Computing signed area of any simple polygon:

```
// Returns the "signed" area of any simple polygon. Assumes at least 3 points.
// If the points are given in counterclockwise order, area is positive;
// if in clockwise order, area is negative.
template <typename FI>   // foward iterator for any Point sequence (e.g. array, vector, list)
double signedArea(FI polyBegin, FI polyEnd) {
  double total(0);
  Point ref=*polyBegin;
  FI two(polyBegin);
  FI three(polyBegin);
  ++two;            // second point
  ++(++three);      // third point
  while (three != polyEnd) {
    total += signedArea(ref, *two, *three);
    two = three;
    ++three;
  }
  return total;
}
```

10.1.4 Line segment intersections

The signed area test can also be used for determining the relative orientation of triples of points. In particular, $\angle abc$ has counterclockwise orientation precisely when the signed area of triangle $\triangle abc$ is positive. This also becomes the basis for a test to determine whether two line segments intersect. Consider segment $\overline{ab}$ and segment $\overline{cd}$. Those intersect based on the following rule:

```
// checks if segment ab intersects segment cd (ill−defined if any colinearity)
bool segmentsIntersect(Point a, Point b, Point c, Point d) {
    return (  (signedArea(a,b,c) > 0   !=  signedArea(a,b,d) > 0) &&
            (signedArea(c,d,a) > 0   !=  signedArea(c,d,b) > 0) );
}
```

10.1.5 Distance from a point to a line

10.1.6 Circle through three points

10.2 Number Theory

10.2.1 Values in Base B

Here, we discuss ways to view a standard integer as a base B value, either by determining the individual digits of that value, or by creating the integer having a known set of base B digits.

```
// Retrieve digit "d" of value in base B, where least signigicant digit is d=0
int getDigit(long long val, int d, int B) {
    for (int i=0; i < d; i++)
        val /= B;
    return val % B;
}
```

```
// Returns the value represented by sequence of integer digits in base B
// (presume to be from MOST significant to LEAST significant)
template <typename FI>
long long fromBase(FI start, FI end, int B) {
    long long result = 0;
    while (start != end) {
        result = B * result + (*start);
        ++start;
    }
    return result;
}
```

```
// Returns the value represented by sequence of integer digits in base B
// (presume to be from LEAST significant to MOST significant)
template <typename FI>
long long fromBaseReverse(FI start, FI end, int B) {
    long long result = 0;
    long long unit = 1;
    while (start != end) {
        result += unit * (*start);
        unit *= B;
        ++start;
    }
    return result;
}
```

Base B counter

If there is need for a base B counter with d digits, this can be done by computing B^d and then allowing a standard loop to progress as follows, and then using getDigit in the body.

```
for (long long k=0; k < LIMIT; k++)  // assuming LIMIT is value B^d
```

However, if this is for an application where there is potential for pruning many configurations, see Section 8.3 for an alternative approach.

10.2.2 Greatest Common Divisor (gcd)

The greatest common divisor of two non-negative integers can be computed using Euclid's algorithm as follows:

```
long long gcd(long long a, long long b) {
    if (b)
        return  gcd(b, a % b);
    else
        return a;
}
```

Note that if you are trying to compute the gcd of more than two values, you can do this by repeated application of pairwise. For example, gcd(a,b,c) = gcd(gcd(a,b), c).

10.2.3 Least Common Multiple (lcm)

The least common multiple of two numbers a and b can be computed as a ∗ b / gcd(a,b). The least common multiple of three or more numbers can be computed similarly, for example as a ∗ b ∗ c / gcd(a,b,c).

10.2.4 Fraction class

Figure 3 on page 27 contains a definition for a basic Fraction class that ensures that all values are stroed in reduced form (requires gcd function from Section 10.2.2). By convention, we ensure that the denominator is non-negative, however numerators may be negative. Other operators could be implemented in natural ways.

10.3 Linear Algebra

10.3.1 Gaussian Elimination

11 Dynamic Programming

11.1 Selecting maximum cardinality subset with property

For example, maximum length increasing sequence.

11.2 Optimal Binary Tree

Given a fixed sequence of items, find the binary tree that has those items in order while minimizing an objective defined as a linear combination of costs for each branching point.

Equivalent to optimum parenthisization, optimal triangulation of convex polygon, etc.

```
class Fraction {
public:
    long long n,d;    // numerator and denominator left public for convenience

    Fraction(long long num=0, long long denom=1) : n(num), d(denom) {
        long long temp = gcd(n,d);              // need gcd definition from elsewhere
        if ((temp < 0) != (denom < 0))
            temp = −temp;                       // reverse sign
        n /= temp;
        d /= temp;
    }

    // You can decide which of the following behaviors you need

    bool operator<(const Fraction& other) const {
        return (n*other.d < other.n*d);
    }

    bool operator==(const Fraction& other) const {
        return (n == other.n && d == other.d);        // since already reduced
    }

    bool operator!=(const Fraction& other) const {
        return (n != other.n || d != other.d);
    }

    Fraction operator+(const Fraction& other) const {
        return Fraction(n*other.d + d*other.n, d*other.d);
    }

    Fraction operator−(const Fraction& other) const {
        return Fraction(n*other.d − d*other.n, d*other.d);
    }

    Fraction operator*(const Fraction& other) const {
        return Fraction(n*other.n, d*other.d);
    }

    Fraction operator/(const Fraction& other) const {
        return Fraction(n*other.d, d*other.n);
    }
};

// output a fraction in standard form
ostream& operator<<(ostream& out, const Fraction& f) {
    out << f.n << "/" << f.d;
    return out;
}
```

Figure 3: Fraction class

11.3 Partitioning an ordered sequence

Some optimization problems involve taking an ordered sequence of n items, and partitioning it into at most $k \leq n$ subsequences for known k, where the overall cost is a linear combination of some metric evaluated on each of the k subsequences. For this problem, a general framework for dynamic programming is to consider a table parameterized as opt[j][c] for $0 \leq j < n$ and $0 \leq c < k$ representing the optimal cost for partitioning elements $0, \ldots, j$ using at most c "cuts" (i.e., partioning into $c + 1$ pieces). The precise metric for individual subsequences will depend on the problem definition. We presume that there is another precomputed array, in which cost[i][j] represents the cost for using a subsequence $i, \ldots, j$ for all pairs $0 \leq i \leq j < n$. An almost complete implementation of this framework is given in Figure 4.

11.4 Sequence alignment

e.g. edit distance between two strings

12 Graph Algorithms

We begin with a discussion of all functionality. The actual code is contained in figures at the end of this section.

12.1 Basic graph library

We propose the following general purpose graph library that allows for a variety of types to serve as the ID for a node (e.g., int, string, pair) and any typical numeric type as an edge weight (e.g., int, double, Fraction). The minimal requirements are that the node type implements **operator**< (since tree maps are used for storage), and that edge weights support a variety of arithmetic operators (e.g., <, >, +, !=, ==) depending upon the precise graph algorithm being used. Our overall approach is to use an adjacency list representation, albeit using a map of maps rather than an array of lists. Our code relies on maps in the following ways.

We define a typedef VM that represents a "Value Map," which is a map going from nodes to weights. We use this internally to keep track of of a node's neighbors, and the edge weights to reach those neighbors. We use this externally as a return type, for example when computing single-source shortest path, this map is used to store each node's distance from the source.

The other important map we use as a return value is a "Parent Map" (defined as typedef PM). This is formally a map from Node to Node, and we use is mainly to represent various trees by mapping from each node to its parent in the tree (or itself, if it is a root of the tree).

To use any of our graph capabilities, being with the class definition given in Figures 5–6. Then add additional functionality, outlined in the coming subsections, by inserting the desired code *just before the end of the formal class definition.*

12.2 Debugging utilities

If everything goes well, there is no need for debugging. But otherwise, you might want to be able to print out a graph. Code for producing such output is given in Figure 7.

```
#define MAX_N  1000    // Based on problem statement
#define MAX_K  1000    // Also based on problem statement (presumably with K <= N)

int k,n;       // actual values of k and n for current trial

// cost[i][j] for 0 <= i <= j < n represents cost of using {i, ..., j} as single piece
int cost[MAX_N][MAX_N];

// opt[j][c] is optimal cost for covering {0, ..., j} with c "cuts" (i.e., c+1 pieces)
long opt[MAX_N][MAX_K];

int main() {
    // parse input in way that is appropriate for specific problem (minimally including n,k)

    precomputeCosts();    // this function should compute appropriate cost[i][j] values for 0 <= i <= j < n

    // do dynamic programming
    for (int c=0; c < k; c++) {          // consider c cuts
        for (int j=0; j < n; j++) {       // consider prefix {0, ..., j}
            opt[j][c] = cost[0][j];              // cost if we use 0 cuts
            if (c > 0) {
                // guess index 'g' of last item before rightmost cut
                for (int g=0; g < j; g++) {
                    long temp = opt[g][c−1] + cost[g+1][j];
                    if (temp < opt[j][c]) {
                        opt[j][c] = temp;
                        // cerr << "Improved opt[" << j << "][" << c << "] using g=" << g << endl;
                        // cerr << "New value is " << temp << end;
                    }
                }
            }
        }
    }

    /*
    cerr << "Final table computed as:" << endl;
    for (int c=0; c < k; c++) {
        for (int m=0; m < n; m++) {
            cerr << " " << setw(10) << opt[m][c];     // setw from <iomanip>
        }
        cerr << endl;
    }
    */

    cout << opt[n−1][k−1] << endl;  // final result {0, ..., n−1} with k−1 cuts
}
```

Figure 4: Dynamic Programming for Sequence Partitioning

12.3 Breadth First Search and Reachability

Breadth first search can be performed using the code of Figure 8. This returns a Parent Map to represent the resulting tree of all nodes that are reachable from the source. Nodes that were not reachable will not be included in that map. This provides a convenient way to check if the entire graph is (strongly) connected, namely by checking of the size of the parent map is equal to the total number of nodes. You can also check reachability of a particular pair of nodes as shown in the `reachable` function withing Figure 8, or you could trace the path backwards between the destination and the source by following parent pointers.

12.4 Depth First Search

We suggest using BFS (rather than DFS) for any problems about basic reachability, because we have given a simpler coding of BFS. The code base we give for DFS is intentionally complex because it has added functionality that is used for subsequent support of algorithms for computed strongly connected components and topological orders. The relevant code is given in Figure 9. The top-level `dfs` function is similar in interface to `bfs`, in that it returns a parent map, and only containing those nodes that were reachable from the source.

12.5 Strongly Connected Components

Code for computing (strongly) connected components is given in Figure 10. *This code base relies upon the DFS suite.* The `computeComponents` code returns a "parent map," but each node is mapped directly to a canonical *captain* of its component, allowing for efficient checks of whether two nodes are in the same component.

12.6 Topological Orderings for DAGs

Code for computing a topological order of a directed graph is given in Figure 11. *This code base relies upon the DFS suite.* The function `computeTopological` returns a vector of nodes, sorted according to the topological order. If the graph is not actually a DAG, this still returns an ordering but obviously not one that is a proper topological order.

12.7 Shortest Paths

Shortest path code is given in Figure 12. There are two key functions. The first, `compute_SP_distances` is used to compute the distance from a chosen source node to all other nodes. It returns a Value Map with the resulting distances. An unreachable will have `graph<Node,Weight>::INF` as its distance.

In cases where you need the actual tree structure, such as to be able to identify the path that was used to reach a node, there is a second routine `compute_SP_parents` that takes the result of the first call as a parameter. A typical calling syntax is `g.compute_SP_parents( g.compute_SP_distances(src))`.

12.8 Minimum Spanning Trees

Minimum Spanning Tree code is given in Figure 13. There are two key functions. The first, `compute_MST` is used to compute the underlying tree. It returns a Parent Map. The second function computes the weight of the minimum spanning tree (or really, of any tree), given the result of the first function. A typical calling syntax is `g.compute_MST_Weight( g.compute_MST())`.

12.9 Network Flow

Code for network flow is included in Figure 14. This code requires inclusion of the bfs code from Figure 8. We note that the current implementation is correct, but not streamlined for efficiency. It should work for moderate sized graphs, but may cause trouble for larger graphs, or graphs with larger resulting flows.

```
#include <iostream>
#include <map>
#include <set>
#include <queue>
#include <limits>
#include <algorithm>
using namespace std;

template <typename Node, typename Weight = int>
class graph {
public:
    // MUST BE INITIALIZED JUST AFTER CLASS DEFINITION ENDS.
    static const Weight INF;

    typedef map<Node, Node> PM;                  // PM:   Parent Map (represents a tree structure)
    typedef typename PM::iterator PMI;           // PMI:  Parent Map Iterator
    typedef typename PM::const_iterator CPMI;    // CPMI: const Parent Map Iterator

    typedef map<Node, Weight> VM;                // VM:   Value Map (maps nodes to values)
    typedef typename VM::iterator VMI;           // VMI:  Value Map Iterator
    typedef typename VM::const_iterator CVMI;    // CVMI: const Value Map Iterator

private:
    typedef map<Node, VM> AM;                    // AM:   Adjacency Map
    typedef typename AM::const_iterator CAMI;    // CAMI: const Adajacency Map Iter

    AM in, out;          // Adjacency maps
    bool dir;

public:
    graph(bool dir) : dir(dir) { }               // true=directed, false=undirected
    bool isDirected() const { return dir; }

    /* Note: adding an edge that already exists overwrites weight */
    void addEdge(Node from, Node to, Weight wt = 1) {
        out[from][to] = wt;
        if (dir) {
            in[to][from] = wt;   // reverse edge
            out[to];             // creates empty neighborhood if 'to' is new node
            in[from];            // creates empty neighborhood if 'from' is new node
        } else {
            out[to][from] = wt;  // reverse edge
        }
    }

    bool hasEdge(Node from, Node to) const {
        CAMI it = out.find(from);
        return (it != out.end() && it->second.find(to) != it->second.end());
    }

    Weight getEdgeWeight(Node from, Node to) const {  // UNSAFE if no such edge
        const VM& dm = out.find(from)->second;
        return dm.find(to)->second;
    }
```

Figure 5: Graph code base (part 1 of 2)

```
    VM empty;   // sentinel
    const VM& getNeighbors(Node node, bool outward=true) const {
        const AM &adj( (dir && !outward) ? in : out);
        CAMI iter = adj.find(node);
        if (iter != adj.end())
            return iter−>second;
        else
            return empty;
    }

    int getDegree(Node node, bool outbound=true) {
        return getNeighbors(node,outbound).size();
    }

    void reverse() {
        if (dir) in.swap(out);
    }

    set<Node> getNodes() const {
        set<Node> nodes;
        for (CAMI it = out.begin(); it != out.end(); ++it)
            nodes.insert(it−>first);
        if (dir)
            for (CAMI it = in.begin(); it != in.end(); ++it)
                nodes.insert(it−>first);
        return nodes;
    }

    // OPTIONAL METHOD.  ONLY NECESSARY WHEN MODEL NEEDS EDGE DELETIONS
    void deleteEdge(Node from, Node to) {          // UNSAFE if no such edge
        out[from].erase(to);
        if (dir)
            in[to].erase(from);
        else
            out[to].erase(from);

        if (out[from].size() + in[from].size() == 0) {
            out.erase(from);
            in.erase(from);
        }

        if (out[to].size() + in[to].size() == 0) {
            out.erase(to);
            in.erase(to);
        }
    }

    //      =====>  INSERT ADDITIONAL SUPPORT FOR ALGORITHMS HERE  <=====

};  // end of graph class

// must initialize INF constant
template <typename Node, typename Weight>
const Weight graph<Node,Weight>::INF = numeric_limits<Weight>::max();
```

Figure 6: Graph code base (part 2 of 2)

```
void printNeighbors(Node node) const {
    cout << node << "'s ";
    if (isDirected()) cout << "outgoing ";
    cout << "neighbors are:" << endl;
    const VM &hNeigh(getNeighbors(node));
    for (CVMI it = hNeigh.begin(); it != hNeigh.end(); ++it)
        cout << "  " << it->first << " with edge weight " << it->second << endl;

    if (isDirected()) {
        cout << node << "'s incoming neighbors are:" << endl;
        const VM &hNeigh(getNeighbors(node, false));
        for (CVMI it = hNeigh.begin(); it != hNeigh.end(); ++it)
            cout << "  " << it->first << " with edge weight " << it->second << endl;
    }
}

void debugDump() const {
    cout << endl << "DEBUG DUMP" << endl;
    set<Node> nodes = getNodes();
    typename set<Node>::const_iterator it;
    for (it=nodes.begin(); it != nodes.end(); ++it) {
        cout << "Node " << *it << endl;
        printNeighbors(*it);
    }
}
```

Figure 7: Debugging code

```
PM bfs(Node start) const { // Note:  unreachable nodes will not be included in PM **/
    queue<Node> itemsToVisit;
    PM visitedNodes; //first is the node visited; second is it's parent
    itemsToVisit.push(start);
    visitedNodes[start] = start;
    while (!itemsToVisit.empty()) {
        Node item = itemsToVisit.front();
        itemsToVisit.pop();
        VM neighbors = getNeighbors(item);
        for (VMI iter = neighbors.begin(); iter != neighbors.end(); ++iter)
            if (visitedNodes.find(iter->first) == visitedNodes.end()) {
                visitedNodes[iter->first] = item;
                itemsToVisit.push(iter->first);
            }
    }
    return visitedNodes;
}

bool reachable(Node source, Node to) const {   // Is destination reachable from source?
    PM tree = bfs(source);
    return (tree.find(to) != tree.end());
}
```

Figure 8: Breadth First Search and Reachability

```
void dfsUtil(Node node, Node from, PM& parent, vector<Node>& finished) const {
    if (parent.find(node) == parent.end()) {
      parent[node] = from;    // mark node as visited
      VM neighbors = getNeighbors(node);
      for (VMI iter = neighbors.begin(); iter != neighbors.end(); ++iter)
          if (parent.find(iter->first) == parent.end())
              dfsUtil(iter->first, node, parent, finished);
      finished.push_back(node);
    }
}

// Returns DFS-tree and a vector of finish times, using 'order' for restarts
template <typename Container>
pair<PM, vector<Node> > dfsHint(Node s, const Container& order) const {
    PM parent;
    vector<Node> finished;
    typename Container::const_iterator walk(order.begin());
    dfsUtil(s, s, parent, finished);
    while (walk != order.end()) {
        if (parent.find(*walk) == parent.end())
            dfsUtil(*walk, *walk, parent, finished);
        ++walk;
    }
    return make_pair(parent, finished);
}

PM dfs(Node start) const {        /** Note:  unreachable nodes will not be included in PM **/
    return dfsHint(start, getNodes()).first;
}
```

Figure 9: Depth First Search

```
void traceRoot(Node n, PM& roots) const {  // pointer hop to root of tree
    if (roots[n] != n) {
        traceRoot(roots[n], roots);
        roots[n] = roots[roots[n]];
    }
}

/* Result actually maps each node to the "captain" of its component */
PM computeComponents() const {
    PM result;
    if (!dir) {
        result = dfsHint(out.begin()->first, getNodes()).first;
    } else {      // only needed for strongly connected components of directed graph
        const_cast<graph*>(this)->reverse();
        vector<Node> finish = dfsHint(out.begin()->first, getNodes()).second;
        std::reverse(finish.begin(), finish.end());
        const_cast<graph*>(this)->reverse();
        result = dfsHint(finish.front(), finish).first;
    }
    for (CPMI it = result.begin(); it != result.end(); ++it)
        traceRoot(it->first, result);
    return result;
}
```

Figure 10: (Strongly) Connected Components

```
vector<Node> computeTopological() const {
  vector<Node> finish = dfsHint(out.begin()->first, getNodes()).second;
  std::reverse(finish.begin(), finish.end());
  return finish;
}
```

Figure 11: Topological Ordering

```
/**
 * result[node] equals the distance from source to node.
 * graph::INF is used if node was unreachable.
 */
VM compute_SP_distances(Node source) const {
    set<Node> nodes = getNodes();
    typename set<Node>::iterator it;
    VM estimate;
    for (it = nodes.begin(); it != nodes.end(); ++it)
        estimate[*it] = ( (*it == source) ? 0 : INF);

    int V = nodes.size();
    bool changed = true;
    int passes = 0;
    while (changed && passes < V-1) {
        changed = false;
        for (it = nodes.begin(); it != nodes.end(); ++it) {
            if (estimate[*it] != INF) {
                const VM& neighbors = getNeighbors(*it);
                for (CVMI d = neighbors.begin(); d != neighbors.end(); ++d) {
                    if (estimate[*it] + d->second < estimate[d->first]) {
                        changed = true;
                        estimate[d->first] = estimate[*it] + d->second;
                    }
                }
            }
        }
        passes++;
    }
    return estimate;
}

/**
 * result[node] is the parent of node in shortest-path tree.
 * Note:  result[node] = node for root and unreachable nodes.
 */
PM compute_SP_parents(const VM& sp) const {
    PM parent;
    for (CVMI it = sp.begin(); it != sp.end(); ++it) {
        parent[it->first] = it->first;      // by default, be own parent
        const VM& neighbors = getNeighbors(it->first);
        for (CVMI d = neighbors.begin(); d != neighbors.end(); ++d) {
            if (sp.find(d->first)->second + d->second == sp.find(it->first)->second)
                parent[it->first] = d->first;
        }
    }
    return parent;
}
```

Figure 12: Shortest Path Code

```
/* result[node] = parent of that node in MST, or self if root */
PM compute_MST() const {
    typedef multimap<Weight, Node> PQ;
    typedef typename PQ::iterator pqIter;
    typedef map<Node, pqIter>  Trace;
    PQ fringe;
    Trace locators;
    PM parent;

    // initially, all nodes have priority infinity except a chosen root
    set<Node> nodes = getNodes();
    typename set<Node>::iterator it;
    Node root = *nodes.begin();
    locators[root] = fringe.insert(make_pair(0, root));
    parent[root] = root;
    for (it = ++nodes.begin(); it != nodes.end(); ++it) {
        locators[*it] = fringe.insert(make_pair(INF, *it));
        parent[*it] = *it;
    }

    while (!fringe.empty()) {
        pqIter small = fringe.begin();
        Node removed = small->second;
        fringe.erase(small);
        locators.erase(locators.find(removed));

        // check neighbors of removed
        const VM& neighbors = getNeighbors(removed);
        for (CVMI d = neighbors.begin(); d != neighbors.end(); ++d) {
            Node dest = d->first;
            Weight edgecost = d->second;
            typename Trace::iterator loc = locators.find(dest);
            if (loc != locators.end() && edgecost < loc->second->first) {
                // found better edge to dest;     remove/reinsert PQ entry
                fringe.erase(loc->second);
                locators[dest] = fringe.insert(make_pair(edgecost, dest));
                parent[dest] = removed;
            }
        }
    }
    return parent;
}

/* computes the total cost of the MST returned by compute_MST */
Weight compute_MST_Weight(const PM& mst) const {
    Weight wt = 0;
    for (CPMI it = mst.begin(); it != mst.end(); ++it)
        if (it->first != it->second)
            wt += getEdgeWeight(it->second, it->first);
    return wt;
}
```

Figure 13: Minimum Spanning Tree Code

```
graph<Node,Weight> computeMaxFlow(Node from, Node to) const {  // Edmonds−Karp variant
    graph<Node,Weight> flow(true);
    const set<Node>& nodes = getNodes();
    typename set<Node>::const_iterator a;
    for (a=nodes.begin(); a != nodes.end(); ++a) {    // initial flow with all zero weights
        const VM& neigh(getNeighbors(*a));
        for (CVMI b = neigh.begin(); b != neigh.end(); ++b)
            flow.addEdge(*a, b−>first, 0);
    }

    bool improving;
    while (true) {
        graph<Node,Weight> residual(true);
        for (a=nodes.begin(); a != nodes.end(); ++a) {
            const VM& neigh(getNeighbors(*a));
            for (CVMI b = neigh.begin(); b != neigh.end(); ++b) {
              Weight capacity(getEdgeWeight(*a, b−>first));
              Weight fwd(flow.getEdgeWeight(*a, b−>first));
              if (capacity > fwd)
                  residual.addEdge(*a, b−>first, capacity−fwd);    // remaining capacity
              if (fwd > 0)
                  residual.addEdge(b−>first, *a, fwd);             // reverse edge
            }
        }
        PM aug = residual.bfs(from);
        if (aug.find(to) == aug.end()) break;                  // no augmenting path
        Weight bottle = INF;
        Node walk(to);
        while (walk != from) {
            Node parent = aug[walk];
            bottle = min(bottle, residual.getEdgeWeight(parent,walk));
            walk = parent;
        }
        walk = to;
        while (walk != from) {
            Node parent = aug[walk];
            if (flow.hasEdge(walk, parent) && flow.getEdgeWeight(walk,parent)>0)
                flow.addEdge(walk, parent, flow.getEdgeWeight(walk,parent)−bottle);
            else
                flow.addEdge(parent, walk, bottle + flow.getEdgeWeight(parent,walk));
            walk = parent;
        }
    }
    return flow;
}

// NOTE: invoked on the FLOW, perhaps as g.computeMaxFlow(from,to).flowVal()
Weight flowVal(Node src) const {
    Weight total = 0;
    VM neigh(getNeighbors(src));
    for (CVMI it = neigh.begin(); it != neigh.end(); ++it)
        total += it−>second;
    return total;
}
```

Figure 14: Network Flow code